

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

Ed. BHAX, DEBRECEN,
2019. február 19, v. 0.0.4

Copyright © 2019 Dr. Bátfai Norbert, Kincs Ákos Levente

Copyright (C) 2019, Norbert Bátfai Ph.D., batfai.norbert@inf.unideb.hu, nbatfai@gmail.com,

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

<https://www.gnu.org/licenses/fdl.html>

Engedélyt adunk Önnek a jelen dokumentum sokszorosítására, terjesztésére és/vagy módosítására a Free Software Foundation által kiadott GNU FDL 1.3-as, vagy bármely azt követő verziójának feltételei alapján. Nincs Nem Változtatható szakasz, nincs Címlapszöveg, nincs Hátlapszöveg.

<http://gnu.hu/fdl.html>

COLLABORATORS

	<i>TITLE :</i>		
	Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert ÁCs Kincs, Ákos Levente	2019. december 12.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai
0.1.0	2019-03-04	Első csokor befejezve.	kincsa
0.2.0	2019-03-12	Második csokor befejezve.	kincsa
0.3.0	2019-03-19	Harmadik csokor befejezve.	kincsa
0.4.0	2019-03-19	Negyedik csokor befejezve.	kincsa

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.5.0	2019-03-26	Ötödik csokor befejezve.	kincsa
0.6.0	2019-04-02	Hatodik csokor befejezve.	kincsa
0.6.1	2019-04-05	Az eddigi fejezetek ellenőrzése, javítása.	kincsa
0.6.2	2019-04-06	Az eddigi fejezetek ellenőrzése, javítása.	kincsa
0.7.0	2019-04-09	Hetedik csokor befejezve.	kincsa
0.7.1	2019-04-13	Az eddigi fejezetek javítása.	kincsa
0.7.2	2019-04-19	Az eddigi fejezetek javítása.	kincsa
0.8.0	2019-04-23	Nyolcadik csokor tervezett feladatának elkészítése, előző fejezetek javításai.	kincsa
0.9.0	2019-04-29	Kilencedik csokor befejezve.	kincsa
0.9.1	2019-05-01	Az eddigi fejezetek javítása.	kincsa
0.9.2	2019-05-03	Az eddigi fejezetek javítása.	kincsa
0.9.3	2019-05-04	Az eddigi fejezetek javítása.	kincsa
0.9.4	2019-05-05	Az eddigi fejezetek javítása.	kincsa
0.9.5	2019-05-07	Az eddigi fejezetek javítása.	kincsa

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.9.6	2019-05-08	Az eddigi fejezetek javítása.	kincsa
0.9.7	2019-05-10	Az eddigi fejezetek javítása.	kincsa

Ajánlás

„To me, you understand something only if you can program it. (You, not someone else!) Otherwise you don't really understand it, you only think you understand it.”

—Gregory Chaitin, *META MATH! The Quest for Omega*, [[METAMATH](#)]

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	8
2.4. Labdapattogás	9
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	13
2.6. 100 éves a Brun tétele	17
2.7. Helló, Google!	19
2.8. A Monty Hall probléma	23
3. Helló, Chomsky!	26
3.1. Decimálisból unárisba átváltó Turing gép	26
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	27
3.3. Hivatalos nyelv	29
3.4. Saját lexikális elemző	31
3.5. l33t.l	32
3.6. A források olvasása	34
3.7. Logikus	41
3.8. Deklaráció	42

4. Helló, Caesar!	45
4.1. double ** háromszögmátrix	45
4.2. C EXOR titkosító	48
4.3. Java EXOR titkosító	49
4.4. C EXOR törő	51
4.5. Neurális OR, AND és EXOR kapu	55
4.6. Hiba-visszaterjesztéses perceptron	60
5. Helló, Mandelbrot!	63
5.1. A Mandelbrot halmaz	63
5.2. A Mandelbrot halmaz a std::complex osztállyal	65
5.3. Biomorfok	69
5.4. A Mandelbrot halmaz CUDA megvalósítása	72
5.5. Mandelbrot nagyító és utazó C++ nyelven	76
5.6. Mandelbrot nagyító és utazó Java nyelven	85
6. Helló, Welch!	91
6.1. Első osztályom	91
6.2. LZW	94
6.3. Fabejárás	100
6.4. Tag a gyökér	102
6.5. Mutató a gyökér	110
6.6. Mozgató szemantika	112
7. Helló, Conway!	115
7.1. Hangyszimulációk	115
7.2. Java életjáték	133
7.3. Qt C++ életjáték	141
7.4. BrainB Benchmark	148
8. Helló, Schwarzenegger!	150
8.1. Szoftmax Py MNIST	150
8.2. Mély MNIST	153
8.3. Minecraft-MALMÖ	153

9. Helló, Chaitin!	154
9.1. Iteratív és rekurzív faktoriális Lisp-ben	154
9.2. Gimp Scheme Script-fu: króm effekt	155
9.3. Gimp Scheme Script-fu: név mandala	162
10. Helló, Gutenberg!	168
10.1. Juhász István: Magas szintű programozási nyelvek - olvasónapló	168
10.2. KR: A C programozási nyelv - olvasónapló	175
10.3. Benedek Zoltán, Levendovszky Tihamér: Szoftverfejlesztés C++ nyelven - olvasónapló	179
10.4. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba	184
10.5. Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II.	185
III. Második felvonás	188
11. Helló, Arroway!	190
11.1. OO szemlélet	190
11.2. „Gagyi”	194
11.3. Yoda	196
11.4. Kódolás from scratch	197
12. Helló, Liskov!	201
12.1. Liskov helyettesítés sértése	201
12.2. Szülő-gyerek	203
12.3. Anti OO	205
12.4. Ciklomatikus komplexitás	206
13. Helló, Mandelbrot!	208
13.1. Reverse engineering UML osztálydiagram	208
13.2. Forward engineering UML osztálydiagram	209
13.3. BPMN	217
14. Helló, Chomsky!	218
14.1. Encoding	218
14.2. l334d1c4 - deprecated	222
14.3. Full screen - deprecated	224
14.4. Perceptron osztály	229

15. Helló, Stroustrup!	232
15.1. JDK osztályok	232
15.2. Hibásan implementált RSA törése	235
15.3. Változó argumentumszámú ctor és Összefoglaló	238
16. Helló, Gödel!	242
16.1. Gengszterek	242
16.2. STL map érték szerinti rendezése	243
16.3. Alternatív Tabella rendezése	245
16.4. GIMP Scheme hack	248
17. Helló, !	255
17.1. OOCWC Boost ASIO hálózatkezelése	255
17.2. SamuCam	256
17.3. BrainB	259
18. Helló, Lauda!	264
18.1. Port scan	264
18.2. Android Játék	266
18.3. Junit teszt	274
19. Helló, Calvin!	278
19.1. MNIST	278
19.2. CIFAR-10 -deprecated	281
19.3. Android telefonra a TF objektum detektálója	285
20. Helló, Berners-Lee!	294
20.1. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba	294
20.2. Java és C++ nyelv összehasonlítása Nyékyné Dr. Gaizler Judit Java 2 útikalauz programozóknak 5.0 I-II. segítségével	295
20.3. Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven	300
IV. Irodalomjegyzék	302
20.4. Általános	303
20.5. C	303
20.6. C++	303
20.7. Lisp	303

Előszó

Amikor programozónak terveztem állni, ellenezték a környezetemben, mondván, hogy kell szövegszerkesztő meg táblázatkezelő, de az már van... nem lesz programozói munka.

Tévedtek. Hogy egy generáció múlva kell-e még tömegesen hús-vér programozó vagy olcsóbb lesz alkalmi igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Minden esetre a programozás a teoretikus kultúra csúcsa. A GNU mozgalomban látom annak garanciáját, hogy ebben a szellemi kalandban a gyerekeim is részt vehessenek majd. Ezért programozunk.

Hogyan forgasd

A könyv célja egy stabil programozási szemlélet kialakítása az olvasóban. Módszere, hogy hetekre bontva ad egy tematikus feladatcsokrot. minden feladathoz megadja a megoldás forráskódját és forrásokat feldolgozó videókat. Az olvasó feladata, hogy ezek tanulmányozása után maga adja meg a feladat megoldásának lényegi magyarázatát, avagy írja meg a könyvet.

Miért univerzális? Mert az olvasótól (kvázi az írótól) függ, hogy kinek szól a könyv. Alapértelmezésben gyereknek, mert velük készítem az iniciális változatot. Ám tervezem felhasználását az egyetemi programozás oktatásban is. Ahogy szélesedni tudna a felhasználók köre, akkor lehetne kiadása különböző korosztályú gyereknek, családoknak, szakköröknek, programozás kurzusoknak, felnőtt és továbbképzési műhelyeknek és sorolhatnánk...

Milyen nyelven nyomjuk?

C (mutatók), C++ (másoló és mozgató szemantika) és Java (lebutított C++) nyelvekből kell egy jó alap, ezt kell kiegészíteni pár R (vektoros szemlélet), Python (gépi tanulás bevezető), Lisp és Prolog (hogy lássuk más is) példával.

Hogyan nyomjuk?

Rántsd le a <https://gitlab.com/nbatfai/bhax> git repót, vagy méginkább forkolj belőle magadnak egy sajátot a GitLabon, ha már saját könyvön dolgozol!

Ha megvannak a könyv DocBook XML forrásai, akkor az alább látható **make** parancs ellenőrzi, hogy „jól formázottak” és „érvényesek-e” ezek az XML források, majd elkészíti a dblatex programmal a könyved pdf változatát, íme:

```
batfai@entropy:~$ cd glrepos/bhax/thematic_tutorials/bhax_textbook/
batfai@entropy:~/glrepos/bhax/thematic_tutorials/bhax_textbook$ make
rm -f bhax-textbook-fdl.pdf
xmllint --xinclude bhax-textbook-fdl.xml --output output.xml
xmllint --relaxng http://docbook.org/xml/5.0/rng/docbookxi.rng output.xml ←
    --noout
output.xml validates
rm -f output.xml
dblatex bhax-textbook-fdl.xml -p bhax-textbook.xls
Build the book set list...
Build the listings...
XSLT stylesheets DocBook - LaTeX 2e (0.3.10)
=====
Stripping NS from DocBook 5/NG document.
Processing stripped document.
Image 'dblatex' not found
Build bhax-textbook-fdl.pdf
'bhax-textbook-fdl.pdf' successfully built
```

Ha minden igaz, akkor most éppen ezt a legenerált **bhax-textbook-fdl.pdf** fájlt olvasod.



A DocBook XML 5.1 új neked?

Ez esetben forgasd a <https://tdg.docbook.org/tdg/5.1/> könyvet, a végén találod az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

- Olvasgasd a kézikönyv lapjait, kezd a **man man** parancs kiadásával. A C programozásban a 3-as szintű lapokat fogod nézegetni, például az első feladat kapcsán ezt a **man 3 sleep** lapot
- [KERNIGHANRITCHIE]
- [BMECPP]
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.

1.3. Milyen filmeket nézzek meg?

- 21 - Las Vegas ostroma, <https://www.imdb.com/title/tt0478087/>, benne a **Monty Hall probléma** bemutatása.

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

Írj olyan C végtelen ciklusokat, amelyek 0 illetve 100 százalékban dolgoztatnak egy magot és egy olyat, amely 100 százalékban minden magot!

Megoldás videó:

Megoldás forrása: https://gitlab.com/kincsa/bhax/tree/master/codes/turing_codes/vegtelen

100 százalékban dolgoztat meg egy magot:

```
#include <stdio.h>

int main()
{
    for( ; ; )
    {

    }

    return 0;
}
```

A kódcipet működése nagyon egyszerű, nem szeretném túlbonyolítani. Lényegében annyi történik, hogy egy olyan for ciklust hoztunk létre, amelynek a fejrésze üres. Ez azt jelenti, hogy a futási feltétel rész is, ez azt eredményezi, hogy a ciklus addig fut, amíg valaki kívülről meg nem szakítja, egy processzormagot végig dolgoztatva. Természetesen a feladat while ciklussal is kivitelezhető lett volna.

0 százalékban dolgoztat meg minden magot:

```
#include <unistd.h>

int main()
{
    for( ; ; )
    {
        sleep(1);
    }
}
```

```
}

return 0;
}
```

Ez a program két sortól eltekintve megegyezik az első programmal. Az egyik különbség az első sorban található: az `unistd.h` header fájlt include-oljuk a jól megszokott `stdio.h` helyett. Ennek oka egyszerű, ez a fájl tartalmazza a `sleep()` functiont, amelynek működése szintén nem "rakétatudomány"....:) A `sleep()` function a zárójelében megadott időmennyiséggel szüneteltezteti a program végrehajtását. Mivel a `for` ciklus a végtelenséggel megy (vagy legalábbis amíg mi meg nem szakítjuk azt..), így a `sleep()` számtalansor végrehajtódik, szüneteltetve a végrehajtást, nem leterhelve erőforrásainkat.

100 százalékban dolgoztat meg minden magot:

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

int main()
{
    #pragma omp parallel
    for(;;)
    {

    }
    return 0;
}
```

Az utolsó, harmadik programban szintén feltűnhet, hogy egy új header-rel találkozunk, méghozzá a `omp.h`-val. Az előbb említett header-rel az OpenMP-t include-oljuk, amely számunkra elengedhetetlen lesz a feladat megoldásához. Az OpenMP egy olyan könyvtár, ami lehetőséget ad a párhuzamos kódolásra. A mi esetünkben a fenti, egyben harmadik kód esetében megvalósul a magok párhuzamos munkavégzése, mindenkorrel a maxon pörög majd a

```
#pragma omp parallel for
```

sor miatt.

Minden program futásának eredményét a `htop` parancssal lett ellenőrizve. További tudnivalókért lásd a manuált: `man htop`.

2.2. Lefagyott, nem fagyott, akkor most mi van?

Mutasd meg, hogy nem lehet olyan programot írni, amely bármely más programról eldönti, hogy le fog-e fagyni vagy sem!

Megoldás videó:

Megoldás forrása: tegyük fel, hogy akkora haxorok vagyunk, hogy meg tudjuk írni a `Lefagy` függvényt, amely tetszőleges programról el tudja dönteni, hogy van-e benne végtelen ciklus:

```
Program T100
{
    boolean Lefagy(Program P)
    {
        if (P-ben van végtelen ciklus)
            return true;
        else
            return false;
    }

    main(Input Q)
    {
        Lefagy(Q)
    }
}
```

A program futtatása, például akár az előző v.c ilyen pszeudókódjára:

```
T100(t.c.pseudo)
true
```

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

A T100-as programot felhasználva készítsük most el az alábbi T1000-set, amelyben a Lefagy-ra épőlő Lefagy2 már nem tartalmaz feltételezett, csak csak konkrét kódot:

```
Program T1000
{
    boolean Lefagy(Program P)
    {
        if (P-ben van végtelen ciklus)
            return true;
        else
            return false;
    }

    boolean Lefagy2(Program P)
    {
        if (Lefagy(P))
            return true;
        else
            for(;;);
    }
}
```

```
main(Input Q)
{
    Lefagy2(Q)
}

}
```

Mit fog kiírni erre a T1000 (T1000) futtatásra?

- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true
- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

akkor most hogy fog működni? Sehogy, mert ilyen Lefagy függvényt, azaz a T100 program nem is létezik.

Tanulságok, tapasztalatok, magyarázat...: A T100-as program nem csinál más, mint a neki átadott programról eldönti, hogy van-e benne végtelen ciklus vagy sem, ennek megfelelően tér vissza a true-val vagy false-szal. A T1000-es programban azonban ha a paraméterként átadott programban végtelen ciklus van, igazat kapunk eredményül, ellenkező esetben, tehát amikor nincs végtelen ciklus az átadott programban, akkor végtelen ciklusba kerül. Ezek egymásnak ellentmondó feltételek, nem írható ilyen program.

2.3. Változók értékének felcserélése

Írj olyan C programot, amely felcseréli két változó értékét, bármiféle logikai utasítás vagy kifejezés násználata nélkül!

Megoldás videó:

Megoldás forrása: https://bhaxor.blog.hu/2018/08/28/10_begin_goto_20_avagy_elindulunk

https://gitlab.com/kincsa/bhax/tree/master/codes/turing_codes/csere

```
{
    int elseo= 5;
    int masodik= 10;

    printf("elseo erteke:%d, masodik erteke:%d\n",elseo, masodik);

    elseo = elseo-masodik; //elseo = -5, masodik = 10
    masodik = elseo+masodik; // masodik = 15, elseo = 5
    elseo = masodik-elseo; // elseo = 10, masodik = 5

    printf("Ertekeik a csere után: ");

    printf("elseo:%d, masodik:%d\n",elseo, masodik);

}
```

Tanulságok, tapasztalatok, magyarázat...

A feladat megoldása során csupán egyszerű számítani alapműveleteket alkalmaztam, csupa összeadást és kivonást, ami a feladat követelményeinek teljes mértékben megfelel, ugyanis a program nem használ logikai utasítást vagy kifejezést. A cserét, vagyis a változók "megforgatását" lényegében 3 sorban meg lehet valósítani, a műveletek részeredményei minden sor végén kommentek formájában megtalálhatóak. Az algoritmus végére a két változó értéke felcsérélődik. Az eredményt a `printf` függvénnyel kiíratjuk a standard outputra ami alapesetben, így most is, vagyis a képernyőn jelenik meg programunk futásának eredménye.

```
int elso = 5;
int masodik = 10;

printf("%d, %d \n",elso,masodik);
printf("%s\n", "A csere után:");

elso = elso*masodik; //elso = 5*10=50;
masodik=elso/masodik;//masodik = 50/10=5;
elso=elso/masodik;//elso = 50/5=10;

printf("%d, %d \n",elso,masodik);
```

Egy újabb verzió a változók megcserélésére, ahol ismételten számítani eszközökhöz nyúltam a feladat megoldásához: szorzást és osztást használtam. A részeredmények ismét megtalálhatóak a sorok végén megjegyzés formájában.

```
int elso = 5;
int masodik = 10;

printf("%d, %d \n",elso,masodik);
printf("%s\n", "A csere után:");

elso = elso^masodik; //0000 1111 = 15 - elso
masodik=elso^masodik;//0000 0101 = 5 - masodik
elso=elso^masodik;//0000 1010 = 10 - elso

printf("%d, %d \n",elso,masodik);
```

Ez a megoldás már nem az eddigi, matematikai oldalról közelíti meg a problémát, inkább egyfajta logikai módon. Az eszköz, amit használunk, a kizárt vagy (XOR), amelynek lényege, hogy ha két bitsorozatot (két szám binárisan felírva) összehasonlítunk, a művelet ott ad vissza 1-est eredményül, ahol a két bitsorozat közül az egyikben az aktuális bit 1-es, de NEM MINDKETTŐBEN, ellenkező esetben ugyanis a művelet 0-s eredményt ad vissza. A megoldáshoz először megjegyzésben felírtam mindkét számot kettes számrendszerbeli alakban, hogy szemléletesebb legyenek a műveletek. Az eddigiekhez hasonlóan itt is kommentek formájában lehet követni a számok "átalakulását".

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés nasználata nélkül írj egy olyan programot, ami egy labdát pattogtat a karakteres konzolon! (Hogy mit értek pattogtatás alatt, alább láthatod a videónkon.)

Megoldás videó: <https://bhaxor.blog.hu/2018/08/28/labdapattogas>

Megoldás forrása:

- https://gitlab.com/kincsa/bhax/blob/master/codes/turing_codes/pattog.c
- https://gitlab.com/kincsa/bhax/blob/master/codes/turing_codes/pattog2.c

if-es megoldás:

A programkód elemzése:

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

int
main ( void )
{

WINDOW *ablak;
ablak = initscr ();

int x = 0;
int y = 0;

int xnov = 1;
int ynov = 1;

int mx;
int my;
```

Az stdio header fájl szükséges ahhoz, hogy az outputon adatot (még ha jelen esetben nem is a klasszikus adatról beszélünk) jelenítsünk meg. A curses könyvtár szükséges többek között az initscr függvény hívásához. Az unistd.h header a később megtalálható usleep függvényhez elengedhetetlen. Ebben a szekcióban történik meg az ablak nevű mutató deklarálása és inicializálása az initscr function segítségével, ugyanis a mutató a függvény által visszaadott érték címére fog mutatni. Az initscr() function meghatározza végrehajtja az első frissítési (refresh) parancsot, ennek eredménye, hogy a program futása elején az ablakunk "tiszta" lesz. Természetesen deklarálásra kerülnek még a labda induló -és maximum helyét (ablak mérete) tároló változók, valamint a labda pozícióváltását megvalósító xnov és ynov változók is.

```
for ( ; ; )
{
    getmaxyx ( ablak, my , mx );

    mvprintw ( y, x, "O" );

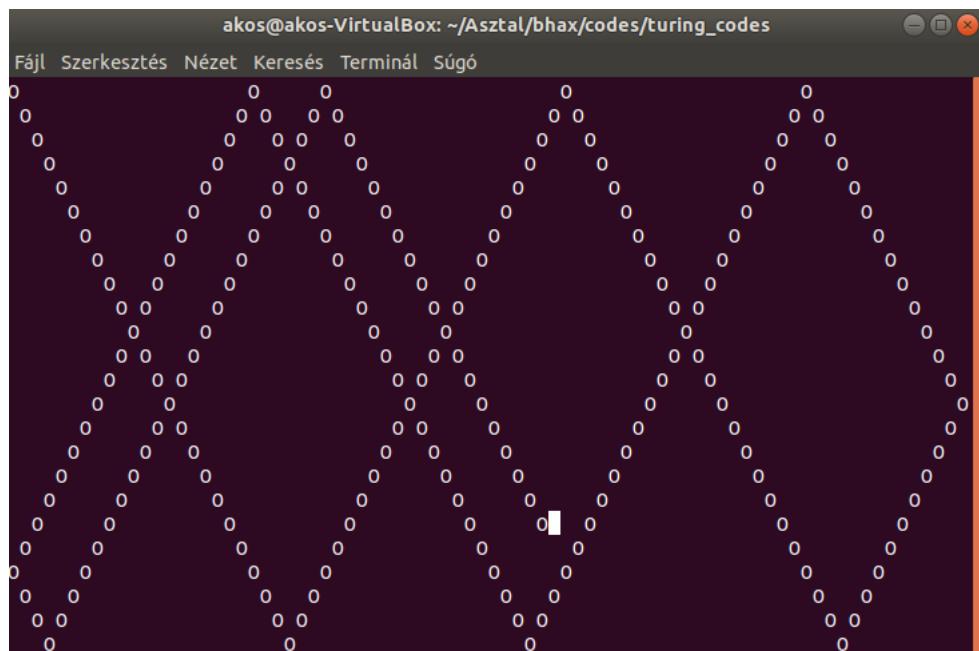
    refresh ();
    usleep ( 100000 ); //mikroszekundum
```

A program második felét egy végtelen ciklus foglalja magába. Még ha a ciklus feje nem is tartogat óriási meglepetéseket, a törzsében találhatóak újdonságok. A `getmaxyx` makró tárolja a labdánk kiinduló koordinátáit valamint az ablak méretét. Ezekből az adatokból megállapítható az ablak dimenziója. A `mvprintw` hasonlít a `printf`-hez, az y-odik és x-edik koordinátára írja ki állandó jelleggel a harmadik paraméterként megadott értéket, jelen esetben "O"-t. A `refresh`-t is alkalmazzuk, amivel értelemszerűen frissítjük a képernyő képét annak érdekében, hogy minden aktuális képet lássunk. A `usleep(x)` a labdánk gyorsaságáért felel, a zárójelében lévő x (tetszőleges) mikroszekundumig szünetelte a program további futását. Minél nagyobb a zárójelben található érték, annál lassabb lesz a labda pattogása.

```
x = x + xnov;
y = y + ynov;

if ( x>=mx-1 ) { // elerte-e a jobb oldalt?
    xnov = xnov * -1;
}
if ( x<=0 ) { // elerte-e a bal oldalt?
    xnov = xnov * -1;
}
if ( y<=0 ) { // elerte-e a tetejet?
    ynov = ynov * -1;
}
if ( y>=my-1 ) { // elerte-e a aljat?
    ynov = ynov * -1;
}
```

Ami itt történik: x és y értékét növeljük vagyis léptetjük a labdát, így valósul meg a labdánk mozgása. Ezek után már "csak" az maradt hátra, hogy megoldjuk azt, hogy amikor a labdánk falhoz ér, visszapattanjon róla ahelyett, hogy "áthaladna" rajta és eltűne az éterben. Ezen jelenség megakadályozását szolgálja a fent látható 4 db feltételvizsgálat, melyek mellett kommentben megtalálhatók, hogy a képernyő éppen melyik pontjára vizsgáljuk meg a feltételt. Ha változtatni kell, a labda pozícióját módosító változókat a -1-szeresére változtatjuk, mely azt eredményezi, hogy a labda mozgása "megfordul", vagyis sikerült kivitelezni, hogy visszapattanjon a falról. Ez az egyik legfontosabb pontja programunknak, enélkül ugyanis nem lenne élethű a programunk. Hiszen (alapesetben) a való életben sem pattan át egy labda a falon... Fordításnál csatolnunk kell az `-lncurses` kapcsolót, hogy `curses` könyvtárunkat futásnál használhassuk!



if nélküli megoldás:

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

int main (void)
{
    int xj = 0, xk = 0, yj = 0, yk = 0;
    int mx = 160, my = 48;

    WINDOW *ablak;
    ablak = initscr();
```

Mi is történt itt? Ismét szükséges include-olni az előző headerfájlokat, illetve az `stdlib.h`-t is. Ez tartalmazza ugyanis az `abs()` függvényt, amelyre nekünk a későbbiekben még szükség lesz. A main-ben történik meg a kezdeti (`xk`, `yk`), jelenlegi (`xj`, `yj`) illetve maximum koordináták (`mx`,`my` melyek az ablak méretét határozzák meg), az ablak nevű pointer deklarálása és értékkadása az `initscr` function használatával, lásd az előző, if-es megoldásnál.

```
for (;;)
{
    xj = (xj - 1) % mx;
    xk = (xk + 1) % mx;
    yj = (yj - 1) % my;
    yk = (yk + 1) % my;
    clear ();

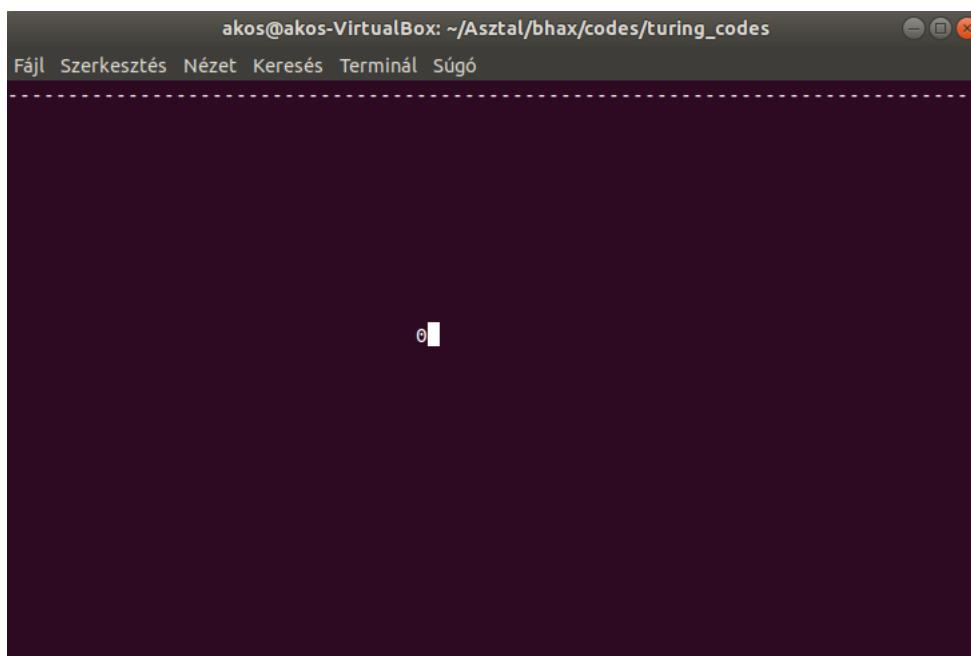
    mvprintw (0, 0,
              " ←
```

```
        " );
mvprintw (24, 0,
        " ←
-----
        " );
mvprintw (abs ((yj + (my - yk)) / 2),
        abs ((xj + (mx - xk)) / 2), "0");

refresh ();
usleep (100000);

}
```

Ez előző végtelen (for) ciklus végzi el a program oroszlánrészét. A ciklus első 5 sorában definiálásra kerülnek a labda helyzetét meghatározó változók. Majd a `clear()` function biztosítja számunkra, hogy "tiszta lappal" indítsuk a programot, vagyis letisztítja nekünk a képernyőt. A következő 3 sorban `mvprintw`-vel kirajzoltatjuk a pálya tetejét és alját jelképező szaggatott vonalakat, a következő sorban pedig már magát a labdát is, a fentebb látható abszolútértékes képlettel meghatározott koordinátkon. A képernyőn frissítésre kerül a `refresh()` function segítségével. Az utolsó érdemi sorral is találkoztunk már az előző, if-es példában is: a `usleep` függvény egy tizedmásodpercet lett beállítva, vagyis ilyen időközönként változik a labda pozíciója.



2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

Írj egy programot, ami megnézi, hogy hány bites a szó a gépeden, azaz mekkora az int mérete. Használd ugyanazt a while ciklus fejet, amit Linus Torvalds a BogoMIPS rutinjában!

Megoldás videó:

Megoldás forrása: https://gitlab.com/kincsa/bhax/blob/master/codes/turing_codes/szohossz2.c

https://gitlab.com/kincsa/bhax/blob/master/codes/turing_codes/bogomips.c

A BogoMIPS kód forrása: UDPORG Facebook csoport - <https://www.facebook.com/groups/udprog/permalink/1051972948323927/>

Tanulságok, tapasztalatok, magyarázat...

A programkód elemzése:

```
#include <stdio.h>

int main()
{
    int i=1;
    int hossz=0;
```

Deklaráljuk és inicializáljuk az integer típusú `i` és `szohossz` változókat.

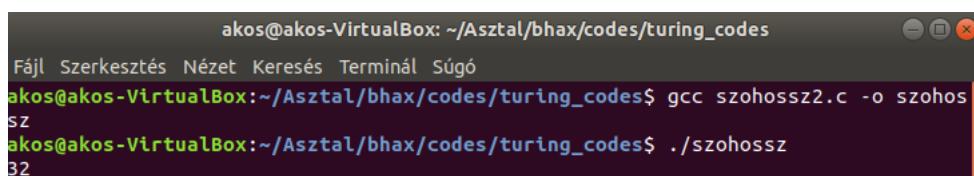
```
do
{
    ++hossz;

}
while (i<<=1);
```

Elindítunk egy `do-while` ciklust, amely addig fut, míg `i` értéke a bitshiftelés során 1 nem lesz. Kis magyarázat ehhez kapcsolódóan: a bitshiftelésből is megkülönböztetünk többfélét, amit mi ebben a feladatban használtunk az nem más, mint a left shifting. Az `i` változó értéke minden egyes ciklusfutáskor egyel bára tolódik. Mindezt kettes számrendszerben kell elköpelní. A számláló értékét minden egyes "tologatás" vagyis shiftelés esetén növeljük.

```
printf("%d\n", hossz);
}
```

Kiiratásra kerül a `hossz` változó értéke és ahogy az várható volt. A program fordítása és futtatása után láthatjuk az outputon a végeredményt: 32, vagyis 32 bit az int mérete.



```
akos@akos-VirtualBox: ~/Asztal/bhax/codes/turing_codes
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos-VirtualBox:~/Asztal/bhax/codes/turing_codes$ gcc szohossz2.c -o szohoss
Sz
akos@akos-VirtualBox:~/Asztal/bhax/codes/turing_codes$ ./szohoss
32
```

A BogoMips

Mi is az a BogoMips? A BogoMips (MIPS = Millions of Instructions Per Seconds) lényegében egy mérőszám, ami a processzor sebességét hivatott prezentálni. Nem mellesleg atyja Linus Torvalds.

Elemezzük ki a kódot!

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

void delay (unsigned long long int loops)
{
```

```
unsigned long long int i;
for (i = 0; i < loops; i++);
}
```

Az `stdio` könyvtár include-olása megtörténik, illetve a `delay` függvény létrehozása, ami egyfajta "időjelzőként" fog működni, $i=0$ -től elszámol `loops`-ig.

```
unsigned long long int loops_per_sec = 1;
unsigned long long int ticks;

printf ("Calibrating delay loop..");
fflush (stdout);

while ((loops_per_sec <= 1))
{
    ticks = clock ();
    delay (loops_per_sec);
    ticks = clock () - ticks;

    printf ("%llu %llu\n", ticks, loops_per_sec);
```

A `loops_per_sec` és a `ticks` változók deklarálásra kerülnek, készült egy kiiratás `printf`-fel, majd használjuk a `fflush` függvényt. A `fflush` biztosítja számunkra, hogy az egyel fentebb lévő sorban lévő kiiratás fixen megtörténjen. Majd jön a `while` ciklusunk, melynek fejében a `loops_per_sec` értékét minden "egyel balra toljuk" left shifting segítségével, addig fut, míg az egész számsor 0 nem lesz. Ha jobban megnézzük, ez a `while` ciklus nagyon hasonlít arra, amelyet mi is alkalmaztunk az első programunkban. A `ticks` változó megkapja a `clock ()` függvény eredményét, vagyis a processzoridőt. Meghívásra kerül a `delay` függvény is, amely paraméterként a `loops_per_sec`-et kapja meg. A `ticks` értéke változik, a `clock` függvény visszatérési értékéből kivonva a `ticks` értékét, megkapjuk `ticks` új értékét. Kiiratásra kerülnek a `ticks` és a `loops_per_sec` változók.

```
if (ticks >= CLOCKS_PER_SEC)
{
    loops_per_sec = (loops_per_sec / ticks) * CLOCKS_PER_SEC;

    printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec / 500000,
           (loops_per_sec / 5000) % 100);

    return 0;
}

printf ("failed\n");
return -1;
}
```

Feltételvizsgálat következik. Ha `ticks` értéke nagyobb vagy egyenlő mint `CLOCKS_PER_SEC`-é, a

loops_per_sec értékét a képen látható módon módosítjuk, majd egy arányszámként kiírjuk az eredményt két sorral lejjebb. Ellenkező esetben hibaüzenetet iratunk ki és a programunk -1-gel tér vissza.

A futtatás eredménye:

```
akos@akos-VirtualBox:~/Asztal/book/codes$ gcc bogomips.c -o bogomips
akos@akos-VirtualBox:~/Asztal/book/codes$ ./bogomips
Calibrating delay loop..2
1 4
1 8
1 16
1 32
1 64
1 128
2 256
2 512
3 1024
7 2048
14 4096
26 8192
51 16384
101 32768
225 65536
465 131072
1071 262144
1853 524288
3339 1048576
6276 2097152
13375 4194304
24204 8388608
50030 16777216
102359 33554432
208603 67108864
421543 134217728
846110 268435456
1725053 536870912
ok - 622.00 BogoMIPS
akos@akos-VirtualBox:~/Asztal/book/codes$ 24     printf ("%llu %llu\n", ticks, loops_per_sec);
25
26 if (ticks >= CLOCKS_PER_SEC)
27 {
28     loops_per_sec = (loops_per_sec / ticks) * CLOCKS_PER_SEC;
29
30     printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec / 500000,
31             (loops_per_sec / 5000) % 100);
32
33     return 0;
34 }
```

Miért volt fontos elemezni ezt a programot is? Azért, mert megbizonyosodhatunk arról, hogy a mi, saját megoldásunkban használt ciklusfej szinte megegyezik a Torvalds által használta.

2.6. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

Megoldás videó: <https://youtu.be/xbYhp9G6VqQ>

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/Primek_R

https://gitlab.com/kincsa/bhax/blob/master/codes/turing_codes;brunetel.r

Tutoriált: Szegedi Csaba -<https://gitlab.com/dev.csaba.szegedi>

Először tisztázzuk, mit is mond ki a Brun téTEL! A Brun téTEL azt mondja ki, hogy az ikerprímek reciprokainak összege nem divergens, sokkal inkább konvergens egy véges értékhez mely értékre Brun-konstansként is szoktak hivatkozni. A programunk is ezt hivatott bizonyítani! A félreértések elkerülése érdekében a prím- és ikerprímszámok fogalmát tisztázni szükséges. Prímek azok az 1-nél nagyobb természetes számok, melyeknek pontosan 2 db. osztójuk van. (1 és önmaga) Az ikerprímek olyan számpárok, melyek tagjai prímek, és különbségük 2. Példának okáért az 5 és a 7 ikerprímek.

A bevezető elméleti háttérének forrásául szolgált: [link](#)

Most, hogy tisztázva lettek az alapfogalmak, kezdjük a forráskód elemzését.

```
library (matlab)
```

A program elején segítségül hívjuk a matlab könyvtárat.

```
stp <- function (x) {
```

Maga az ikerprímek reciprokösszegének kiszámolása ennek a függvénynek a törzsében fog megvalósulni. A függvény egy számot vár paraméterként és a kapott számmal végzi tovább a számításokat.

```
primes = primes (x)
```

A primes vektorban tároljuk el a primes függvény eredményét, vagyis a zárójelen belüli számig megkapjuk az előforduló prímszámokat. Ez az első érdemi lépés, ugyanis ahhoz, hogy dolgozni tudjunk a téTEL-en, először a prímeket kell tudnunk. Ha megvannak, csak utána lehet eldönteni róluk, hogy ikerprímek-e vagy sem. Jelen példában 17-ig keressük a prímszámokat, de ez a szám nyilván tetszőleges lehet.

```
> primes = primes (17)
> primes
[1]  2  3  5  7 11 13 17
```

```
diff = primes [2:length (primes)] - primes [1:length (primes)-1]
```

Ebben a sorban először érdemes megfigyelni, hogy 2 ciklust indítunk. Az egyiket 2-től primes hosszáig, a másodikat 1-től primes hossza-1-ig. Vesszük a két kifejezés különbségét és eltároljuk a diff vektorban, így megvannak a prímszámjaink különbségei egytől-egyig.

```
> diff = primes [2:length (primes)] - primes [1:length (primes)-1]
> diff
[1] 1 2 2 4 2 4
```

```
idx = which (diff==2)
```

Az `idx` vektorba kerülnek tárolásra azok az indexet, ahol a két prímszám különbsége 2 volt, ezt a `which` függvényel hajtjuk végre. Így most már tudjuk, melyik indexű elemek az ikerprímek. Mert hiába csak az ikerprímpár első tagja ismert, ha tudjuk, hogy a párja 2-vel több nála, ezért a pájrát is ismerjük természetesen.

```
> idx = which(diff==2)
> idx
[1] 2 3 5
```

```
t1primes = primes[idx]
t2primes = primes[idx]+2
```

Tárolásra kerülnek az ikerprímek, az ikerpárok első és második tagjai külön-külön vektorban: `t1primes` és `t2primes`-ben.

```
t1primes = primes[idx]
> t2primes = primes[idx]+2
> t1primes
[1] 3 5 11
> t2primes
[1] 5 7 13
```

```
rt1plust2 = 1/t1primes+1/t2primes
```

Etároljuk ikerprímeink reciprokjait.

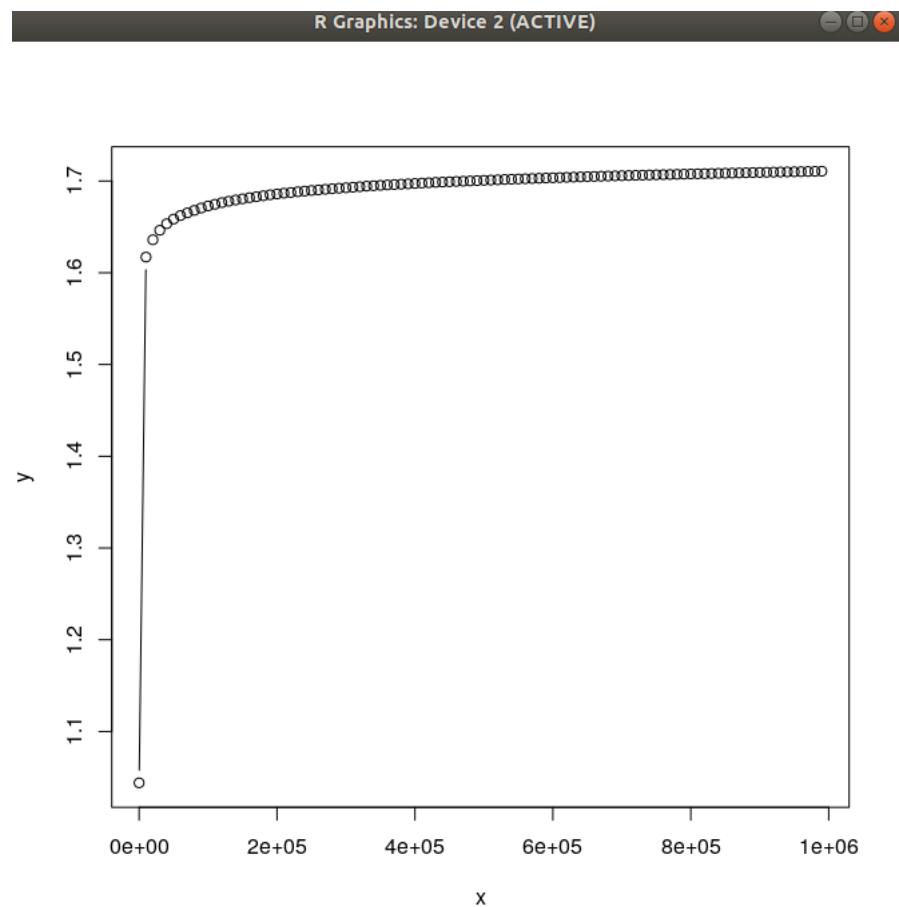
```
rt1plust2 = 1/t1primes+1/t2primes
> rt1plust2
[1] 0.5333333 0.3428571 0.1678322
```

```
return(sum(rt1plust2))
}
```

Megkapjuk a tételek hiányzó láncszemét, vagyis az ikerprímjeink reciprokainak összegét!

```
x=seq(17, 1000000, by=10000)
y=sapply(x, FUN = stp)
plot(x,y,type="b")
```

A függvényünket ha már elkészítettük, rajzoltassuk is ki az eredményt! Az `x` tengelyen 17-től kezdve, 10 000-es lépésekkel, maximum 1 000 000-ig ábrázoljuk! Definiáljuk az `y` változót is, ami majd a `y` tengelyünket adja majd. A `plot`-os matlab parancsot használva, ábrázoljuk a képernyőn!



2.7. Hello, Google!

Írj olyan C programot, amely egy 4 honlapból álló hálózatra kiszámolja a négy lap Page-Rank értékét!

Megoldás videó:

Megoldás forrása: https://sourceforge.net/p/udprog/code/ci/master/tree/source/kezdo/elsoc/pagerank/pagerank_13.c

https://gitlab.com/kincsa/bhax/blob/master/codes/turing_codes/pagerank.c

Tutoriált: Bacsik Mátyás -<https://gitlab.com/bacsikmatyas>

Tanulságok, tapasztalatok, magyarázat...:

A PageRank a Google által, a keresésekkel alkalmazott algoritmus. Minél nagyobb a PageRank értéke egy adott honlapnak, az annál fontosabbnak minősül. Az algoritmus megalkotói Sergey Brin és Larry Page voltak. További információkat a PageRank-ről [itt](#) lehet olvasni.

Egy adott weblap PageRank értékét meghatározó összefüggés:

$$\text{PageRank}(i) = (1 - d) + d \sum_{j \in M(i)} \frac{\text{PageRank}(j)}{L(j)}$$

- **M(i)**: azoknak az oldalaknak a halmaza, amik tartalmaznak linket az i. oldalra
- **L(j)**: a j. oldalról kimenő linkek száma
- **d**(csillapító tényezőt)
 - az oldalak a szavazatukból csak d részt osztanak tovább, (1-d)-t pedig megtartanak

A kép forrása: https://moodle.sapidoc.ms.sapientia.ro/pluginfile.php/7585/mod_resource/content/1/PageRank.pdf

Kezdődjön a kód elemzése!

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

A műveletek eredményeinek megjelnítéséhez illetve a matematikai műveletek elvégzéséhez include-olunk kell az `stdio` és a `math` könyvtárakat.

```
void kiir(double tomb[], int db)
{
    int i;

    for(i=0; i<db; ++i)
    {
        printf("%d%f \n", i, tomb[i]);
    }
}
```

Deklaráljuk a `kiir` eljárást, paraméterei egy double típusú tömb, illetve egy Integer típusú, a tömb darabszámát tárolni hivatott `db` változó. Deklaráljuk és inicializáljuk az `int` típusú `i` változót. A `for` ciklus `i=0`-től `db`-ig megy, és rendre kiírja a tömb `i`-edik elemeit. Eljárásról beszélünk, hiszen `void` típusú, így nincs visszatérési értéke. "Csupán" végrehajt, de azt helyesen teszi.

```
double tavolsag( double PR[], double PRv[], int n)
{
    int i;
    double osszeg=0.0;

    for(i =0; i<n; ++i)
        osszeg+= (PRv[i] - PR[i]) * (PRv[i]-PR[i]);
    return sqrt (osszeg);
}
```

Deklarálásra kerül a `tavolsag` függvény, aminek paraméterei: két double típusú tömb, amikre a `PR` és a `PRv` jelölésekkel hivatkozunk, illetve egy integer típusú, `n` nevű változó. Deklaráljuk és inicializáljuk az `i` és az `osszeg` változót. Előzőt `int`, míg utóbbit `double` típussal. A `for` ciklus `i=0`-től `n`-ig megy, és

a `for` ciklus `i=0`-tól `db`-ig megy, és az `oszeg` változó értéke a `Prv` és a `PR` tömbök `i`-edik elemeinek különbségének szorzata lesz. A `PR` mátrix az aktuális PageRank értékeit, míg a `PRv` az előző körös PageRank értékeit fogja tartalmazni. A függvény az `sqrt` functionnel az `oszeg` változó négyzetgyökét adja vissza.

```
void pagerank(double T[4][4]) {
    double PR[4] = { 0.0, 0.0, 0.0, 0.0 };

    double PRv[4] = { 1.0/4.0, 1.0/4.0, 1.0/4.0, 1.0/4.0};

    int i, j;

    for(;;) {

        for (i=0; i<4; i++) {
            PR[i]=0.0;
            for (j=0; j<4; j++) {
                PR[i] = PR[i] + T[i][j]*PRv[j];
            }
        }

        if (tavolsag(PR,PRv,4) < 0.0000000001)
            break;

        for (i=0;i<4; i++) {
            PRv[i]=PR[i];
        }
    }

    kiir (PR, 4);
}
```

Magá a `pagerank` függvényünk, ami a matematikai számítást végzi. Esetünkben ez a függvény egy 4 soros, 4 oszlopos táblázatot vár paraméterül. A `PR` tömb minden a 4 eleme a függvény meghívása előtt csupa 0, az algoritmus végén ez a tömb fogja tartalmazni az eredményeket. A `PRv` tömb elemeivel fogjuk a szorzást elvégezni. Deklarálásra kerülnek az `i` és `j` változók majd egy végtelen cikluson belül az egyes oldalak `PR` (értsd PageRank) értékét meghatározó algoritmus található meg ami két `for` ciklussal is operál, egyik a sorokat míg a másik az oszlopokat járja be, a `PR` mátrix tartalma frissítésre kerül. A ciklus leáll, ha a `tavolsag` függvény rendkívül kicsi értéket ad vissza mely érték egyébként az ábrán d-vel jelölt úgynvezett csillapító tényező. Az utolsó előtti említésre méltó dolog, hogy ezután egy egyszerű `for` ciklussal végigmegyünk a `PRv` tömbön és átmásoljuk az elemeit a `PR` tömbbe, végül az utóbbi tömböt kiiratjuk.

```
int main (void)
{
double L[4][4] =
{
{0.0, 0.0, 1.0 / 3.0, 0.0},
{1.0, 1.0 / 2.0, 1.0/ 3.0, 1.0},
{0.0, 1.0 / 2.0, 0.0, 0.0},
{0.0, 0.0, 1.0 / 3.0, 0.0}
```

```
};

double L1[4][4] = {
    {0.0, 0.0, 1.0/3.0, 0.0},
    {1.0, 1.0/2.0, 1.0/3.0, 0.0},
    {0.0, 1.0/2.0, 0.0, 0.0},
    {0.0, 0.0, 1.0/3.0, 0.0}
};

double L2[4][4] = {
    {0.0, 0.0, 1.0/3.0, 0.0},
    {1.0, 1.0/2.0, 1.0/3.0, 0.0},
    {0.0, 1.0/2.0, 0.0, 0.0},
    {0.0, 0.0, 1.0/3.0, 1.0}
};
```

Kezdődik a main függvényünk. Létrehozásra kerülnek a számolás alapját adó kétdimenziós, 4 soros, 4 oszlopos tömbök (amik a 4 honlapból álló hálózatunkat szimbolizálják), négyzetes mátrixként is hivatkozhatunk rájuk. Mindegyik tömb más és más esetet szimbolizál. Az első az "optimális" eset, ilyenkor úgymond klasszikus PageRank értékeket kapunk, ez a futtatásnál látszik is majd.

```
printf("\nAz eredeti mátrix értékeivel történő futás:\n");
pagerank(L);

printf("\nAmikor az egyik oldal semmire sem mutat:\n");
pagerank(L1);

printf("\nAmikor az egyik oldal csak magára mutat:\n");
pagerank(L2);

printf("\n");

return 0;
}
```

A pagerank függvény meghívásra kerül mindenhol alkalmmal más és más mátrixokkal a lehetséges eseteket reprezentálva.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/turing_codes$ gcc pagerank.c -o pagerank -lm
akos@akos-VirtualBox:~/Asztal/bhax/codes/turing_codes$ ./pagerank
```

```
Az eredeti mátrix értékeivel történő futás:
0.090909
0.545455
0.272727
0.090909
```

Lássuk, milyen eredményt kapunk abban az esetben, mikor az egyik oldal semmire sem mutat:

```
Amikor az egyik oldal semmire sem mutat:
```

```
0.000000  
0.000000  
0.000000  
0.000000
```

És végül de nem utolsósorban azt az esetet vizsgálva, mikor az egyik oldal csak önmagára mutat:

Amikor az egyik oldal csak magára mutat:

```
0.000000  
0.000000  
0.000000  
1.000000
```

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Megoldás videó: https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcsan

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/MontyHall_R

https://gitlab.com/kincsa/bhax/blob/master/codes/turing_codes/monty.r

Tanulságok, tapasztalatok, magyarázat...:

Kezdjük először is azzal, mit is takar a probléma. Monty Hall amerikai televíziós műsorvezető volt, az egyik televíziós vetélkedőjének utolsó feladatára utal a probléma. A műsor végén a játékosnak 3 csukott ajtó közül kell választania. 2 mögött egy-egy kevésbé értékes nyeremény található, a harmadik mögött azonban egy igazán értékes, magyarul a főnyeremény. A játékos értelemszerűen azt nyeri, ami a kiválasztott ajtaja mögött "rejtőzik", azonban van az egészben egy csavar.

A játékos kiválaszt egy ajtót, ám mielőtt az kinyílna neki, a műsorvezető kinyit a másik két választható ajtó közül azt, ami mögött a nem értékes nyeremény található. Ezt követően ami az ajtókat illeti, a játékosnak lehetősége van váltni vagy akár maradhat is az eredeti megérzésénél.

Tehát a kérdés a következő: érdemes-e változtatni az eredeti elképzelen?

A bevezető elméleti hátterének forrásául szolgált: [link](#)

A programkód elemzése:

```
kiserletek_szama=100  
kiserlet = sample(1:3, kiserletek_szama, replace=TRUE)  
jatekos = sample(1:3, kiserletek_szama, replace=TRUE)  
musorvezeto=vector(length = kiserletek_szama)
```

Mi is történik ebben a kódicspetben? A `kiserletek_szama` változót deklaráljuk és inicializáljuk. A `kiserlet` és `jatekos` vektorokat szintén deklaráljuk a `sample` function-nel, amely minden esetben `kiserletek_szama` darabszámú (vagyis 100), 1-től 3-ig terjedő intervallumból sorsolt véletlen számmal tölti fel a vektort. A `kiserlet` vektor azt fogja tárolni, hogy éppen melyik ajtót mögött van a nyeremény, míg a `jatekos` azt, hogy melyik ajtót választotta maga a játékos. A `musorvezeto` egy vektor lesz, aminek mérete a `kiserletek_szama`.

```
for (i in 1:kiserletek_szama) {  
  
  if(kiserlet[i]==jatekos[i]) {  
  
    mibol=setdiff(c(1,2,3), kiserlet[i])  
  
  } else {  
  
    mibol=setdiff(c(1,2,3), c(kiserlet[i], jatekos[i]))  
  
  }  
  
  musorvezeto[i] = mibol[sample(1:length(mibol),1)]  
  
}
```

A program ezen része egy for ciklussal indul, amelyben az i 1-től indul egészen a `kiserletek_szama` által tárolt értéig. Ezután jön az if-feltétel: ha a `kiserlet` i-edik eleme megegyezik a `jatekos` i-edik elemével, akkor a `mibol` vektor értéke egyenlő lesz az 1,2,3 számok és a `kiserlet` i-edik elemének "különbségével". Azért idézőjelessen beszélünk erről, ugyanis nem matematikai értelemben vett különbségről van szó. Lényegében egy olyan sorszám lesz ez esetben a `mibol` értéke, amely mögött biztosan nincs már a nyeremény, ugyanis a játékos elsőre beletrafált, melyik mögött van a nyeremény, igaz, ő még nem tud róla. Ha ez nem teljesül, jön az else ág, amely lényegében ugyanez "pepitában". A `mibol` értéke ugyanúgy az az érték lesz, ami az 1,2,3 számsorban megtalálható, azonban nem i-edik eleme sem a `kiserlet`nek, sem a `jatekos` i-edik eleme, hiszen a `musorvezető` csak olyan ajtót nyithat ki, amelyet a játékos még nem választott, és nyeremény sincs mögötte.

A `musorvezető` vektor elemei pedig az előző sorban tárgyalt `mibol` vektor elemeit tartalmazza.

```
nemvaltoztatesnyer= which(kiserlet==jatekos)  
valtoztat=vector(length = kiserletek_szama)  
  
for (i in 1:kiserletek_szama) {  
  
  holvalt = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i]))  
  valtoztat[i] = holvalt[sample(1:length(holvart),1)]  
  
}  
  
valtoztatesnyer = which(kiserlet==valtoztat)
```

A következő pár sorról néhány gondolat. Az első sorban a `which` függvénytel megkapjuk azt az indexet, ahol a `kiserlet` és a `jatekos` értékek megegyeznek, ez lesz a `nemvaltoztatesnyer` értéke. Ez az az eset, amikor a játékos elsőre jól tippel és kitart a tippje mellett. A `valtoztat` egy vektor lesz, aminek mérete a `kiserletek_szamaval` egyenlő. A következő cikluson belül történnek még érdekességek: a `holvart` értéke az lesz, ami sem a `musorvezető`, sem pedig a `jatekos` vektornak nem eleme, vagyis se nem a játékos előzőleg, se nem a `musorvezető` nem választotta még ki.

A változtat vektor feltöltésre kerül az előző holvolt vektor elemeiből. Így a valtoztatesnyer vektor értékeit is megkaptuk, nyilván ez az az eset mikor a játékos által kiválasztott új ajtó és a nyereményt rejtvő ajtó megegyezik.

```
sprintf("Kiserletek szama: %i", kiserletek_szama)
length(nemvaltoztatesnyer)
length(valtoztatesnyer)
length(nemvaltoztatesnyer)/length(valtoztatesnyer)
length(nemvaltoztatesnyer)+length(valtoztatesnyer)
```

A program utolsó pár sorában kiirásra kerülnek az eredmények, amelyről leolvasható, hogy ténylegesen jobbak az esélyeink akkor, hogyha változtatunk eredeti választásunkon:

```
[1] "Kiserletek szama: 100"
> length(nemvaltoztatesnyer)
[1] 35
> length(valtoztatesnyer)
[1] 65
> length(nemvaltoztatesnyer)/length(valtoztatesnyer)
[1] 0.5384615
> length(nemvaltoztatesnyer)+length(valtoztatesnyer)
[1] 100
```

A válasz a feladat elején feltett kérdésre: IGEN, érdemes változtatni. A számok legalábbis ezt bizonyítják. Természetesen nagyobb kísérletszámnál szemléletesebb lenne az eredmény de sajnos a számítógépem erőforrásait figyelembe vége úgy gondoltam, jobb lenne csak 100 kísérlettel megnézni a problémát. Azonban ezt a 100 esetet vizsgálva is látszik úgy gondolom, hogy sikerült megfejtenünk a paradoxont.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása: https://gitlab.com/kincsa/bhax/blob/master/codes/chomsky_codes/decitounar.c

Tanulságok, tapasztalatok, magyarázat...

A félreértesek és a feladat meg nem értésének elkerülése érdekében először a fogalmakat szükséges tisztázunk. Mik a decimális számok? A tízes számrendszerbeli számok, amikkel egész kicsi korunk óta számolunk, ergő a számunkra természetes számábrázolási mód. Na és mi az unáris számrendszer? A természetes számok ábrázolására használt számrendszer, ami rendkívül egyszerű. X számot egy rendszeresített szimbólum X-szeri ismétlésével tudunk ábrázolni. A célra használt leggyakoribb szimbólumok: 1 és |.

A bevezető elméleti háttérének forrásául szolgált:

[link](#)

A programunk a következőképpen néz ki:

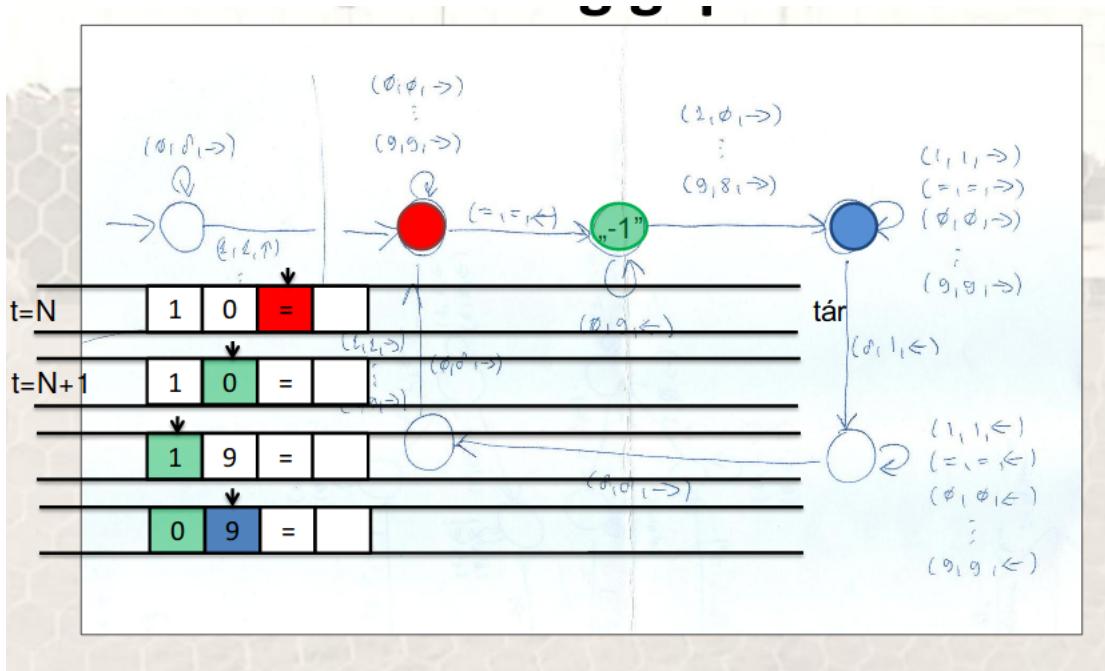
```
int number;
printf("Add meg az átalakítandó szamot! \n");
scanf("%d", &number);

for(int i=number; i>0; --i)
{
    printf("1");
}
printf("\n");
```

És hogy mit csinál? Rövidre fogva: kér a felhasználótól egy tetszőleges számot, amelyet aztán átalakít unárisba. A scanf függvénnyel olvassuk az inputot, majd jön egy for ciklus, ami különlegesebb talán mint az előzőek. Ugyanis i értékét nem növeljük, hanem csökkentjük a ciklusfeltétel teljesülése esetén. Ahányszor a feltétel teljesült, annyi darab 1-es kerül kiiratásra. Ez azt eredményezi, hogy annyi 1-est látunk kiiratva a képernyőre, amennyit mi ténylegesen megadtunk.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/chomsky_codes$ gcc decitounar.c -o deci
maltou
akos@akos-VirtualBox:~/Asztal/bhax/codes/chomsky_codes$ ./decimaltou
Add meg az atalakitando szamot!
9
111111111
akos@akos-VirtualBox:~/Asztal/bhax/codes/chomsky_codes$
```

Lássuk a feladat által kért állapotátmenet gráfját!



Jogosan merül fel a kérdés..Mit csinál ez a Turing gép? A szalagon érkező számnak veszi a megelőzőjét addig, amíg az azt megelőző szám 0 nem lesz. A folyamat során kivont egyeseket eltárolja, méghozzá annyi darabot, ahány darabot szükséges volt kivonni ahhoz, hogy a 0-s értéket kapjuk.

A fent látható ábra Bátfai Norbert tanár úr Magas szintű programozási nyelvek 1 c. téma második előadásának fóliájának 27-es diáján található, melyhez a link: https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_1.pdf?fbclid=IwAR0IRxgOptchfTCwjGC_tGhSiw6GvwzwBOuOgvnID4TeKvgEHL_ihWsK

3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen

Mutass be legalább két környezetfüggő generatív grammatikát, amely ezt a nyelvet generálja!

Megoldás videó:

Megoldás forrása: https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_1.pdf?fbclid=IwAR0IRxgOptchfTCwjGC_tGhSiw6GvwzwBOuOgvnID4TeKvgEHL_ihWsK
30-31. diája

Tanulságok, tapasztalatok, magyarázat..

Akinek a nevéhez leginkább köthető a generatív nyelvtan, az nem más, mint Noam Chomsky, Ő fogalmazta meg ezzel kapcsolatos nézőpontját. Lényege, hogy ha ismerjük az alapszabályokat, az nyelvtan elemeit felhasználva végtelen sok szót és mondatot létre tudunk hozni. Chomsky anno 4 fő csoportra osztotta a generatív nyelvtanokat: reguláris-, környezetfüggő-, környezetfüggetlen-, rekurzív nyelvtanok. Ezen generatív nyelvtanok mind-mind 4 fő részből állnak: nem-terminális jelek, terminális jelek, produkciós

szabályok és kezdőszimbólum. Két új fogalmat is a köztudatba emelt: kompetencia és performanca. A kompetencia a nyelv szabályainak ismeretét jelenti, a performanca pedig annak használatát.

A bevezető elméleti háttérének forrásául szolgált:

[link](#)

[link](#)

Nézzük meg a feladat által kért környezetfüggő grammatikát!

Az első grammatika:

```
S, X, Y „változók” / nem terminális jelek  
a, b, c „konstansok” / terminális jelek  
 $S \rightarrow abc$ ,  $S \rightarrow aXbc$ ,  $Xb \rightarrow bX$ ,  $Xc \rightarrow Ybcc$ ,  
 $bY \rightarrow Yb$ ,  $aY \rightarrow aaX$ ,  $aY \rightarrow aa$ 
```

```
//S-ből indulunk (kezdőszimbólum)
```

```
S (S → aXbc)  
aXbc (Xb → bX)  
abXc (Xc → Ybcc)  
abYbcc (bY → Yb)  
aYbbcc (aY → aa)  
aabbbcc
```

```
S (S → aXbc)  
aXbc (Xb → bX)  
abXc (Xc → Ybcc)  
abYbcc (bY → Yb)  
aYbbcc (aY → aaX)  
aaXbbcc (Xb → bX)  
aabXbcc (Xb → bX)  
aabbXcc (Xc → Ybcc)  
aabbYbcc (bY → Yb)  
aabYbbccc (bY → Yb)  
aaYbbbccc (aY → aa) aaabbccc
```

A második grammatika:

```
A, B, C „változók”  
a, b, c „konstansok”  
 $A \rightarrow aAB$ ,  $A \rightarrow aC$ ,  $CB \rightarrow bCc$ ,  $cB \rightarrow Bc$ ,  $C \rightarrow bc$ 
```

```
//S-ben kezdünk  
A (A → aAB)  
aAB ( A → aC)  
aacb (CB → bCc)  
aabCc (C → bc)  
aabbbcc
```

```
aabbccA (A → aAB)  
aAB ( A → aAB)
```

```
aaABB ( A → aAB)
aaaABBB ( A → aC)
aaaaCBBB (CB → bCc)
aaaabCcBB (cB → Bc)
aaaabCBcB (cB → Bc)
aaaabCBBc (CB → bCc)
aaaabbCcBc (cB → Bc)
aaaabbCBcc (CB → bCc)
aaaabbbCccc (C → bc)
aaaabbbbcccc
```

A feladat elején másodikként megadott forrásban található az egyes nyelvek részletezése. A környezetfüggetlen nyelvtanok produkciós szabályára jellemző, hogy bal oldalán csak nem terminális jelek lehetnek. Nálunk ez a feltétel nem teljesül, ezért környezetfüggő a fenti két nyelvtanunk. Látható, hogyan juthatunk el S-ből a kívánt alakig produkciós szabályokkal.

3.3. Hivatkozási nyelv

A [KERNIGHANRITCHIE] könyv C referencia-kézikönyv/Utasítások melléklete alapján definiál BNF-ben a C utasítás fogalmát! Majd mutass be olyan kódcsipeteket, amelyek adott szabvánnyal nem fordulnak (például C89), mással (például C99) igen.

Megoldás videó:

Megoldás forrása: https://gitlab.com/kincsa/bhax/blob/master/codes/chomsky_codes/hivnyelv.c

Talán először érdemes tisztázni a BNF mozaikszót! A BNF a Backus-Naur-forma rövidítése. Mire jó? Környezetfüggetlen nyelvtanok leírására alkalmas. Létezik a BNF-nek egy fejlettebb, EBNF nevű változata is, használata szintén népszerű.

Kezdődjön utasítást BNF-ben való definiálása a Kernighan-Ritchie könyv alapján, a 230. oldaltól kezdődően!

```
<kifejezés_utasitas> ::= <kifejezés>

<összetett_utasitas> ::= { {<deklarációs_lista>} {<utasítás_lista>} }

<feltételes_utasitas> ::= if (<kifejezés>) <utasítás> | else <utasítás>

<while_utasitas> ::= while (<kifejezés>) <utasítás>

<do_utasitas> ::= do <utasítás> while (<kifejezés>)

<for_utasitas> ::= for(<kifejezés>, <kifejezés>, <kifejezés>) <utasítás>

<switch_utasitas> ::= switch (<kifejezés>) <utasítás> | case <↔ állandókifejezés> : <utasítás> | default: <utasítás>

<break_utasitas> ::= break
```

```
<continue_utasitas> ::= continue  
  
<return_utasitas> ::= return <kifejezés> | return  
  
<goto_utasitas> ::= goto <azonosító>  
  
<címkezett_utasitas> ::= <azonosító>  
  
<>nulla_utasitas> ::= ;
```

Mely jelölésrendszerben a <> jelek közé a nemterminális jelek kerülnek, a definíálási szabály pedig a ::= jel után található meg. A | jel a logikai vagyot jelöli.

A BNF megértéséhez és a feladat megoldásához nagy segítséget nyújtott a következő oldal: <https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm>

Most, hogy ezzel megvagyunk, nézzünk meg egy olyan kódöt, amit nem mindegy, milyen szabvánnyal fordítunk!

```
#include <stdio.h>  
  
int main()  
{  
    for(int i=0; i<11; ++i)  
    {  
        printf("%d ", i);  
    }  
  
    return 0;  
}
```

Tanulságok, tapasztalatok, magyarázat: A kódot szerintem nem feltétlenül lenne indokolt megmagyarázni, viszont mégis megteszem, ugyanis ismétlés a tudás anyja! A program main-jében csak egy for ciklus áll, ami 1-től 10 írja ki a számokat.

Amikor fordítjuk, majd futtatjuk, a következő eredményt kapjuk:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc hivnyelv.c -o hivnyelv  
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./hivnyelv  
1 2 3 4 5 6 7 8 9 10 akos@akos-VirtualBox:~/Asztal/bhax/codes$
```

Tehát itt semmi probléma, minden úgy működik, ahogy mi azt elvártuk. Azonban nézzük meg, mi történik ha fordításnál a C89-es szabványt használjuk!

```
gcc hivnyelv.c -o hivnyelv -std=c89  
hivnyelv.c: In function 'main':  
hivnyelv.c:6:3: error: 'for' loop initial declarations are only allowed in ←  
C99 or C11 mode  
    for(int i=1; i<11; ++i)
```

```
^~~~  
hivnyelv.c:6:3: note: use option -std=c99, -std=gnu99, -std=c11 or -std= ←  
gnull to compile your code
```

Láthatjuk, hogy nem sikerül a gcc-nek lefordítania a programot a C89-es szabvánnyal, ugyanis a változó ciklusfejben való deklarálása nem volt engedélyezett C89-ben. A fordító egyébként a hibaüzenetben javaslatot is tesz arra, miként érdemes próbálkoznunk annak érdekében, hogy sikerre jussunk. Ez a for ciklus az egyik, ha nem a legszemléletesebb példa a különbségek bemutatására.

3.4. Saját lexikális elemző

Írj olyan programot, ami számolja a bemenetben megjelenő valós számokat! Nem elfogadható olyan megoldás, amely maga olvassa betűnként a bemenetet, a feladat lényege, hogy lexert használunk, azaz óriások vállán álljunk és ne kispályázzunk!

Megoldás videó:

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/thematic_tutorials/bhax_textbook_IgyNeveldaProgramChomsky/realnumber.l

https://gitlab.com/kincsa/bhax/blob/master/codes/chomsky_codes/valosszamok.l

Tanulságok, tapasztalatok, magyarázat...

Kezdjük azzal, hogy mit érdemes a Lexről tudni előljáróban. A Lex nem más, mint egy program, amely az általunk definiált szabályokat felhasználva szabványos programkódot állít elő számunkra melyet fordítás és futtatás után már használhatunk is. A mi feladatunk ezeket a szabályokat definiálni ahhoz, hogy a kódunk sikeresen előálljon.

```
%{  
#include <stdio.h>  
int realnumbers = 0;  
%}
```

Az egyes részegységeket százalékjelek választják el egymástól. Az első rész klasszik C kód, itt definiáljuk és deklaráljuk a szükséges változókat, jelen esetben egy számlálót. Ezen kívül include-olásra kerül az stdio függvénykönyvtár is

```
digit [0-9]  
%%  
{digit}*(\.{digit}+)? {++realnumbers;  
    printf("[realnum=%s %f]", yytext, atof(yytext));}  
%%
```

Itt kezdődik a szabályok definiálása részünk. De még mielőtt a konkrét szabályokba kezdenénk, számjegyeinknek a digit elnevezést adtuk, így fogunk hivatkozunk rájuk a későbbiekben.

```
%%  
{digit}*(\.{digit}+)? {++realnumbers;
```

```
    printf("[realnum=%s %f]", yytext, atof(yytext));  
%
```

A dupla százalékjelek között történik a konkrét szabálydefiniálás. Azon számok definiálása, amit az inputon keresni fog a program. Ha találtunk a szabályunknak megfelelő számot, a számlálónk értékét növeljük és egy egyszerű printf függvényel kiiratjuk magát a számot, ami a yytext változóban tárolódik. Ez azonban egy string, szeretnénk még átalakítani double típusúvá, ezért használjuk az atof függvényt.

A keresendő számok egyébként formailag így néznek ki: Pont előtti rész: 0 vagy több darab számjegy. Ezt követi természetesen egy pont. A pont utáni rész: 1 vagy több darab számjegyet tartalmaz. (mely rész egyébként opcionális, vagyis 0 vagy 1 darab ilyen rész követheti a pont után az első számjegyet)

A különböző jelölésekéről az alábbi dokumentációban olvashatunk: <https://www.epaperpress.com/lexandyacc-download/lexyacc.pdf>

```
int  
main ()  
{  
    yylex ();  
    printf("The number of real numbers is %d\n", realnumbers);  
    return 0;  
}
```

Lényegben a programunk 3. fő része szintén klasszikus C kód, a mainben meghívásra kerül a yylex függvény, ami a dupla százalékjel közötti, 2. fő része a programunknak.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ lex -o valosszamok.c valosszamok. ←  
1  
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc valosszamok.c -o valos -lfl  
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./valos  
32.2  
[realnum=32.2 32.200000]  
daskuhhhrlj65hzfuzgw32.5  
daskuhhhrlj[realnum=65 65.000000]hzfuzgw[realnum=32.5 32.500000]  
The number of real numbers is 3
```

Mit láthatunk itt? Fordítottuk az -lfl kapcsoló segítségével, majd futtattuk a programot. Amint azt látni lehet, hibátlanul működik.

3.5. I33t.I

Lexelj össze egy I33t cipher!

Megoldás videó:

Megoldás forrása: https://gitlab.com/kincsa/bhax/blob/master/codes/chomsky_codes/I33t.I

Tutor: Szegedi Csaba -<https://gitlab.com/dev.csaba.szegedi>

Tanulságok, tapasztalatok, magyarázat...

A "leet speak" kommunikáció során a betűket, számokat azok eredeti formájukra hasonlító karakterrel/karakterekkel helyettesítik, így ellehetetlenítve (vagy legalábbis nagyon megnehezítve) a kódolást nem ismerő személy számára az olvasást. A '80-as években alakult ki.

A bevezető elméleti háttérének forrásául szolgált:

[link](#)

A programkód elemzése:

```
%{  
#include <string.h>  
%}
```

Ahogy arra az előző feladatból emlékezhetünk, az első százaléklelek közötti szekció nem más, mint a definíciós rész. Igaz, jelen esetben egyetlen változót sem szükséges definiálnunk a feladat megoldásához, azonban a `string` függvénykönyvtárat igen, ugyanis a feladat során stringekkel fogunk tevékenykedni.

```
%%  
[a-zA-Z]  
%
```

A dupla százalékkellel elkezdődik a szabályok definiálása szekció. Szögletes zárójelek között megadjuk a Lexnek, hogy mikkel kell majd dolgoznia, vagyis mik lesznek az átalakítandó karakterek. Esetünkben ezek az ábécé kis- és nagybetűi lesznek.

Egy szintén klasszikus C kódcsipettel találjuk szemben magunkat. Egy `switch` szerkezetben megvizsgáljuk az `input` string karaktereit. A `yytext` egy egyelemű karaktertömböt jelöl, aminek első (és egyetlen) tagjára (0-s indexű elemére vagyunk) kíváncsiak, ezért néz ki így a kifejezésünk. A következőkben minden case-ben egy-egy Leet-beli karaktert rendelünk a betűkhöz, egy egyszerű `printf`-fel kiíratva azt, így az eredeti karakter helyett természetesen már az utóbbiak kerülnek kiiratásra. Ezt eljátsszuk az összes előforduló karakterre, de mivel mégig ugyanígy járunk el, ezt nem részletezném. A forrásban természetesen megtalálható az összes átalakítás. Ezzel le is zárul a szabályok definiálása.

```
int  
main()  
{  
yylex();  
return 0;  
}
```

A programunk utolsó harmadában létrehozzuk a `main`-ünket és meghívjuk a `yylex` függvényt, ami nem más, mint az előbb kivesézett második fő része a programunknak.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ lex -o leet.c 133t.l  
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc leet.c -o leet -lfl  
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./leet  
alma  
41m4  
hello
```

```
h3110  
auto  
4ut0
```

Kiadjuk a Lexnek a parancsot, hogy készítsen egy leet.c outputot a l33t.l kódunkból. Miután ez megtörténik, fordítás során linkeljük neki az lfl könyvtárat, majd futtatjuk legenerált programunkat. Amint az látható, helyesen is működik.

3.6. A források olvasása

Hogyan olvasod, hogyan értelmezed természetes nyelven az alábbi kódcsipeteket? Például

```
if(signal(SIGINT, jelkezelő)==SIG_IGN)  
    signal(SIGINT, SIG_IGN);
```

Ha a SIGINT jel kezelése figyelmen kívül volt hagyva, akkor ezen túl is legyen figyelmen kívül hagyva, ha nem volt figyelmen kívül hagyva, akkor a jelkezelő függvény kezelje. (Miután a **man 7 signal** lapon megismertem a SIGINT jelet, a **man 2 signal** lapon pedig a használt rendszerhívást.)

Bugok

Vigyázz, sok csipet kerülendő, mert bugokat visz a kódba! Melyek ezek és miért? Ha nem megy ránézésre, elkapja valamelyiket esetleg a **splint** vagy a **frama**?

i.

```
if(signal(SIGINT, SIG_IGN)!=SIG_IGN)  
    signal(SIGINT, jelkezelő);
```

Értelmezés: Ha a SIGINT jel (SIGINT = megszakítási jel a billentyűzetről) kezelése nem volt figyelmen kívül volt hagyva, akkor akkor a SIGINT jelet a jelkezelő kezelje, ellenkező esetben legyen figyelmen kívül hagyva.

```
#include <stdio.h>  
#include <signal.h>  
  
void jelkezelő()  
{  
    printf("hello");  
}  
  
int main()  
{  
    for(;;)  
    {  
        if(signal(SIGINT, SIG_IGN)!=SIG_IGN)  
            signal(SIGINT, jelkezelő);  
    }  
}
```

```
    return 0;  
}
```

Lássuk mi történik fordítás és futtatás után!

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc forrasok_olvasasa.c -o ←  
forras  
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./forras  
^C  
^X  
^C
```

Amint az látszik, nem reagál a kis programunk a billentyűzetről érkező jelekre, így a program helyesen működik.

ii.

```
for(i=0; i<5; ++i)
```

Értelmezés: A ciklusunk $i=0$ -tól indul, i értékét egyel növelem (postfix inkrementáló operátorral) addig, amíg az kisebb, mint 5, ergő 4-ig.

Tesztelés:

```
#include <stdio.h>  
  
int main()  
{  
    int i;  
  
    for(i=0; i<5; ++i)  
    {  
        printf("%d ", i);  
    }  
  
    printf("\n");  
  
    return 0;  
}
```

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc forrasok_olvasasa2.c -o ←  
forrasok  
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./forrasok  
0 1 2 3 4
```

Fordítás és futtatás után a fenti (hibamentes) eredményt kapjuk.

iii.

```
for(i=0; i<5; i++)
```

Értelmezés: A ciklusunk $i=0$ -tól indul, i értékét egyel növelem (postfix inkrementáló operátorral) addig, amíg az kisebb, mint 5, ergő 4-ig.

Tesztelés:

```
#include <stdio.h>

int main()
{
    int i;

    for(i=0; i<5; i++)
    {
        printf("%d ", i);
    }

    printf("\n");

    return 0;
}
```

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc forrasok_olvasasa3.c -o ←
forrasok
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./forrasok
0 1 2 3 4
```

Fordítás és futtatás után a fenti, ismételten hibamentes eredményt kapjuk!

iv.

```
for(i=0; i<5; tomb[i] = i++)
```

Értelmezés: A `tomb` i-edik eleme tárolja el rendre az `i` változó aktuális értékét, mely `i` érték természetesen -ameddig a feltétel igaz-, növekszik.

Tesztelés:

```
#include <stdio.h>

int main()
{
    int i;

    int tomb[]={1,1,1,1,1};

    for(i=0; i<5; tomb[i] = i++)
    {
        printf("%d ", tomb[i]);
    }

    printf("\n");

    return 0;
}
```

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc forrasok_olvasasa4.c -o ←
forrasok
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./forrasok
1 0 1 2 3
```

Fordítás és futtatás után látható, hogy nem éppen azt kaptuk, amit előzőleg vártunk a programtól... Nézzük, a splint hibásnak találja-e!

```
forrasok_olvasasa4.c:9:27: Expression has undefined behavior (left ←
operand uses
                           i, modified by right operand): tomb[i] = ←
                           i++
Code has unspecified behavior. Order of evaluation of function ←
parameters or
subexpressions is not defined, so if a value is used and modified in
different places not separated by a sequence point constraining ←
evaluation
order, then the result of the expression is unspecified. (Use - ←
evalorder to
inhibit warning)

Finished checking --- 1 code warning
```

A splint szerint is hibás a kód, így ha lehet, kerüljük a használatát!

v. `for(i=0; i<n && (*d++ = *s++); ++i)`

Értelmezés: A ciklus addig fut, amíg i értéke nem éri el n értékét és a logikai ÉSsel összekapcsolt második feltétel is teljesül, vagyis a d és s pointerek növelt értéke "egyenlő". (azért tettem idézőjelek közé, mert látni fogjuk, hogy nem teljesen erről van szó..) Mivel növeljük a pointerek értékét, egy több elemet tartalmazó adatszerkezetre kell mutassanak, hogy az adott adatszerkezet (pl. tömb) minden elemén végigmenjen, így nyer értelmet a kifejezés.

```
#include <stdio.h>

int main()
{
    int a[]={5,10,15};
    int b[]={5,1,15};

    int i;

    int n=3;

    int* d;
    d= a;
    int* s;
    s = b;
```

```
for(i=0; i<n && (*d++ = *s++); ++i)
    printf("%d", i);
    printf("\n");

return 0;
}
```

Az elkészült programot azokat az indexeket hivatott visszaadni, ahol a két tömb értékei megegyeznek.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc forrasok_olvasasa9.c -o ←
forrasok
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./forrasok
0 1 2
```

Nos, nem ezt vártuk eredményül..

Mit mond a splint?

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ splint forrasok_olvasasa9.c
Splint 3.1.2 --- 20 Feb 2018

forrasok_olvasasa9.c: (in function main)
forrasok_olvasasa9.c:17:19: Right operand of && is non-boolean (int):
                  i < n && (*d++ = *s++)
The operand of a boolean operator is not a boolean. Use +ptrnegate ←
to allow !
to be used on pointers. (Use -boolops to inhibit warning)

Finished checking --- 1 code warning
```

Azt mondja, hogy a műveletünk nem boolean művelet, így az ÉSsel kapcsolt második feltételt figyelembe sem veszi a program, így zavartalanul fut addig, amíg i el nem éri n-et. Tehát hibás a kódrészlet.

vi.

```
printf("%d %d", f(a, ++a), f(++a, a));
```

Értelmezés: A kiiratásban kétszer hívjuk meg a 2 paraméterrel rendelkező f függvényt. Először a második paramétert növeljük paraméterátadáskor, majd az első paramétert, szintén paraméterátadáskor.

Tesztelés:

```
#include <stdio.h>

int f(int a, int b)
{
    return a*b;
}
```

```
int main()
{
    int a=4;
    int b=5;

    printf("%d, %d", f(a, ++b), f(++a, b));

    printf("\n");

    return 0;
}
```

Esetünkben az `f` függvény a paraméterként neki átadott két, Integer típusú változó szorzatát adja vissza.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc forrasok_olvasasa5.c -c
o forrasok
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./forrasok
30, 25
```

Fordítjuk és futtatjuk a kódot és..nem azt látjuk, amire számítottunk előzőleg. A várt eredmények 24 és 25 lettek volna, ehelyett 30-at és 25-öt kaptunk. Ellenőrizzük le `splint`-tel.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ splint forrasok_olvasasa5. c
Splint 3.1.2 --- 20 Feb 2018

forrasok_olvasasa5.c: (in function main)
forrasok_olvasasa5.c:17:19: Argument 2 modifies b, used by argument 3 (order of
evaluation of actual parameters is undefined):
printf("%d, %d", f(a, ++b), f(++a, b))
Code has unspecified behavior. Order of evaluation of function parameters or
subexpressions is not defined,
```

Ez a `splint`-nek sem tetszik. Hibás a kód!

vii.

```
printf("%d %d", f(a), a);
```

Értelmezés: Kiiratásra kerül az `f` függvény a paraméterrel és maga az a változó.

Tesztelés:

```
#include <stdio.h>

int f(int a)
{
    return a*2;
```

```
}

int main()
{
    int a=5;

    printf("%d %d", f(a), a);

    printf("\n");

    return 0;
}
```

Most az `f` függvény az átadott szám kétszeresét adja vissza.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc forrasok_olvasasa6.c -c
o forrasok
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./forrasok
10 5
```

Fordítjuk és futtatjuk a kódot és azt látjuk, hogy nem hibás a kód, teljesen jól működik!

viii.

```
printf("%d %d", f(&a), a);
```

Értelmezés: Kiiratásra kerül az `f` függvény melynek paramétere `a` memóriabeli címe lesz és maga az a változó.

Tesztelés:

```
#include <stdio.h>

int f(int* a)
{
    return *a;
}

int main()
{
    int a=5;

    printf("%d %d", f(&a), a);

    return 0;
}
```

Most az `f` függvény az átadott számot adja vissza a `*` operátor segítségével, ami nem más, mint egy mutató (pointer). A kódbeli kiiratásban az `f` függvényt úgy hívjuk meg, hogy paramétere az `a` változó címe lesz. Az `f` függvény az `a` paraméter értékét fejtő- majd adja vissza, vagyis hogy milyen érték található meg az adott memóriacímen. Helyes a megoldásunk, ha a kimeneten ugyanazt a két számot fogjuk látni.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc forrasok_olvasasa7.c - ←
      o forrasok
akos@akos-VirtualBox:~/Asztal/bhax/codes$ ./forrasok
5 5
```

Fordítás és futtatás után látható, hogy mivel ez megtörtént, így a kód helyes.

Megoldás forrása:

https://gitlab.com/kincsa/bhax/tree/master/codes/chomsky_codes/forrasok_olvasasa

Tanulságok, tapasztalatok, magyarázat...

Minden egyes kódrészletre rövid C programot írva ellenőriztem le működésüköt.

Megoldás videó:

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

```
$ (\forall x \exists y ((x < y) \wedge (y \text{ prim})) ) $  
$ (\forall x \exists y ((x < y) \wedge (y \text{ prim}) \wedge (S y \text{ prim})) \leftarrow ) $  
$ (\exists y \forall x (x \text{ prim}) \supset (x < y)) $  
$ (\exists y \forall x (y < x) \supset \neg (x \text{ prim})) $
```

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/MatLog_LaTeX

Megoldás videó: <https://youtu.be/ZexiPy3ZxsA>, https://youtu.be/AJSXOQFF_wk

Tanulságok, tapasztalatok, magyarázat.. Az Ar nyelvben az N interpretáció mellett a felírt formulák a következők:

A nyelv a linkre kattintva található feladatgyűjteményben található meg, a 37. oldaltól kezdődően: https://arato.inf.kovacs.zita/Logika/logika_lengyel.pdf

Az előbbi feladatgyűjteményből szemezgetve a következőképp fejtettem meg a formulákat:

```
$ (\forall x \exists y ((x < y) \wedge (y \text{ prim})) ) $
```

vagyis

```
(\forall x \exists y ((x < y) \wedge (y \text{ prim})) )
```

A prímek száma végtelen.

```
$ (\forall x \exists y ((x < y) \wedge (y \text{ prim}) \wedge (S y \text{ prim})) \leftarrow ) $
```

vagyis

$$\forall x \exists y ((x < y) \wedge (y \text{ prim}) \wedge (\text{SSy} \text{ prim}))$$

Az ikerprímek száma végtelen.

$$\$ (\exists y \forall x (x \text{ prim}) \supset (x < y)) \$$$

vagyis

$$\exists y \forall x (x \text{ prim}) \supset (x < y)$$

A prímek száma véges.

$$\$ (\exists y \forall x (y < x) \supset \neg(x \text{ prim})) \$$$

vagyis

$$(\exists y \forall x (y < x) \supset \neg(x \text{ prim}))$$

A prímek száma véges.

A formulák átírásában segítségemre volt a következő oldal:<https://www.codecogs.com/latex/eqneditor.php>

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciaja
- egések tömbje
- egések tömbjének referenciaja (nem az első elemé)
- egészre mutató mutatók tömbje
- egészre mutató mutatót visszaadó függvény
- egészre mutató mutatót visszaadó függvényre mutató mutató
- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

Mit vezetnek be a programba a következő nevek?

- `int a;`

egész típusú, a azonosítójú változót

- ```
int *b = &a;
```

a egészre mutató b mutatót, mely a memóriabeli címét tárolja

- ```
int &r = a;
```

egész referenciáját (C++ feature)

- ```
int c[5];
```

5 elemű, egészek tömbjét

- ```
int (&tr)[5] = c;
```

5 elemű, egészek tömbjének referenciáját (az egész tömbnek)

- ```
int *d[5];
```

egészre mutató mutatók 5 elemű tömbjét

- ```
int *h();
```

egészre mutató mutatót visszaadó függvényt

- ```
int *(*l)();
```

egészre mutató mutatót visszaadó függvényre mutató mutatót

- ```
int (*v(int c))(int a, int b)
```

egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényt

- ```
int (*(*z)(int))(int, int);
```

függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényt

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/codes/chomsky\\_codes/deklaracio.cpp](https://gitlab.com/kincsa/bhax/blob/master/codes/chomsky_codes/deklaracio.cpp)

```
#include <iostream>

//egészre mutató mutatót visszaadó függvény

int* returnpointer(int* a)
{
 return a;
```

```
}

/*egészet visszaadó
 és két egészet kapó függvényre mutató mutatót visszaadó,
egészet kapó függvény*/

int (*pointer (int c)) (int a, int b)
{
 printf("Siker!\n");
}

int main()
{
 int x = 5; //egész

 int *y = &x; //egészre mutató mutató

 int &ref = x; //egész referenciaja

 int t1 [10] = {1,2,3,4,5,6,7,8,9,10}; //egések tömbje

 int (&tr) [10] = t1; //egések tömbjének referenciaja

 int *d[10]; //egészre mutató mutatók tömbje

 int *ertek = returnpointer(t1); //egészre mutató mutatót visszaadó ←
 függvény

 //egészre mutató mutatót visszaadó függvényremutató mutató

 int* (*pointer) (int*) = returnpointer;

 /*függvénymutató egy egészet visszaadó és
 két egészet kapó függvényre mutató mutatót visszaadó,
 egészet kapó függvényre*/
}

int (*(*z) (int)) (int , int h);

return 0;
}
```

Tanulságok, tapasztalatok, magyarázat...:

A fenti C++ programban bevezetésre került minden, amit a feladat kért. Az eddigiek től eltérően C helyett azért használtunk C++-t, hogy használni tudjuk a referencia operátort (`&`) a szükséges példáknál referenciaját. minden sor vagy éppen sorok után (néhány helyen előtte) megjegyzésben megtalálhatók, hogy éppen mi került bevezetésbe programunkba.

## 4. fejezet

# Helló, Caesar!

### 4.1. double \*\* háromszögmátrix

Írj egy olyan `malloc` és `free` párost használó C programot, amely helyet foglal egy alsó háromszögmátrixnak a szabad tárban!

Megoldás videó: <https://www.youtube.com/watch?v=1MRTuKwRsB0>

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/blob/master/thematic\\_tutorials/bhax\\_textbook\\_IgyNeveldaProgramCaesar/tm.c](https://gitlab.com/nbatfai/bhax/blob/master/thematic_tutorials/bhax_textbook_IgyNeveldaProgramCaesar/tm.c)

[https://gitlab.com/kincsa/bhax/blob/master/codes/caesar\\_codes/haromszogmatrix.c](https://gitlab.com/kincsa/bhax/blob/master/codes/caesar_codes/haromszogmatrix.c)

Tanulságok, tapasztalatok, magyarázat...

A háromszögmátrix csak négyzetes mátrix lehet. Négyzetes mátrix az a mátrix, amelyben a sorok száma megegyezik az oszlopok számával. A háromszögmátrix tehát olyan négyzetes mátrix, amelynek főátlójá felett/alatt CSAKIS 0 található. Attól függően, hogy a 0-k a főátló felett/alatt helyezkednek el beszélhetünk felső/alsó háromszögmátrixról. A mi példánkban egy alsó háromszögmátrix fog elkészülni a következő program által:

A kód elemzése:

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

int main ()
{
 int nr = 5;
 double **tm;
 printf("%p\n", &tm);
```

Include-oljuk a szükséges függvénykönyvtárat, a program a main-nel kezdődik. Itt deklaráljuk az `nr` változót, ez lesz a mátrix sorainak száma. Mivel négyzetes mátrixról van szó, oszlopainak száma is ennyi lesz. Ezen kívül létrehozásra kerül a `tm`, double mutatóra mutató mutató, ami lefoglal a memóriából 8 bájtot. A `printf` függvényteljesen kiiratja az előbb tárgyalta `tm` pointer memóriabeli címét.

```
if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
```

```

{
 return -1;
}
printf("%p\n", tm);

```

Egy feltételvizsgálat jön velünk szembe, mely tartalmazza a lespoilerezett `malloc` függvényt. A `malloc` a zárójelben szereplő értéknek megfelelő bájtmennyiséget foglal le a memóriából, visszatérési értéke egy `void` típusú pointer. A lefoglalt bájtmennyiség ebben az esetben `nr`-szer `double *` lesz, vagyis  $5*8$ , 40 bájt. Típuskényszerítjük a `malloc`-ot, hiszen nekünk nem `void` típusú pointerre, sokkal inkább egy `double**`-ra lenne szükségünk. Ha a `tm` `NULL` értékre mutat, vagyis ha nem sikerült a `malloc` memória-foglalása, -1-gyel tér vissza a program jelezve azt, hogy hiba történt a végrehajtás során. A `malloc`-kal lefoglalt memóriaterület címét kapja meg a `tm`, ez kerül kiiratásra.

```

for (int i = 0; i < nr; ++i)
{
 if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL)
 {
 return -1;
 }
}
printf("%p\n", tm[0]);

```

Indítunk egy `for` ciklust mely törzsében ugyanazt vizsgáljuk mint az előző kódrészletben. A `malloc` most is típuskényszerítésre kerül, `double **`-gal. Az előzőtől különböző módon itt a mutatót úgy használjuk, mintha tömb lenne. (A tömbök és mutatók felcserélésére egyébként gyakran láthatunk példát más programokban is). Kiiratásra kerül a `tm` által tárolt memóriacím. A `malloc` most egyébként  $i+1$ -szer `double`-nyi memóriát foglal le, ez biztosítja azt hogy sorról-sorra több elemmel legyen feltöltve a háromszögmátrix, kialakítva ezzel jellegzetes alakját.

```

for (int i = 0; i < nr; ++i)
 for (int j = 0; j < i + 1; ++j)
 tm[i][j] = i * (i + 1) / 2 + j;

for (int i = 0; i < nr; ++i)
{
 for (int j = 0; j < i + 1; ++j)
 printf ("%f, ", tm[i][j]);
 printf ("\n");
}

```

Két egymásba ágyazott `for` ciklussal a `tm` (látszat)tömb elemeit feltölthjük fent olvasható, `tm[i][j] = i * (i + 1) / 2 + j` képlet segítségével. A második `for` ciklus az elemek kiiratásáért felel.

```

tm[3][0] = 42.0;
(* (tm + 3)) [1] = 43.0;
*(tm[3] + 2) = 44.0;
* (* (tm + 3) + 3) = 45.0;

```

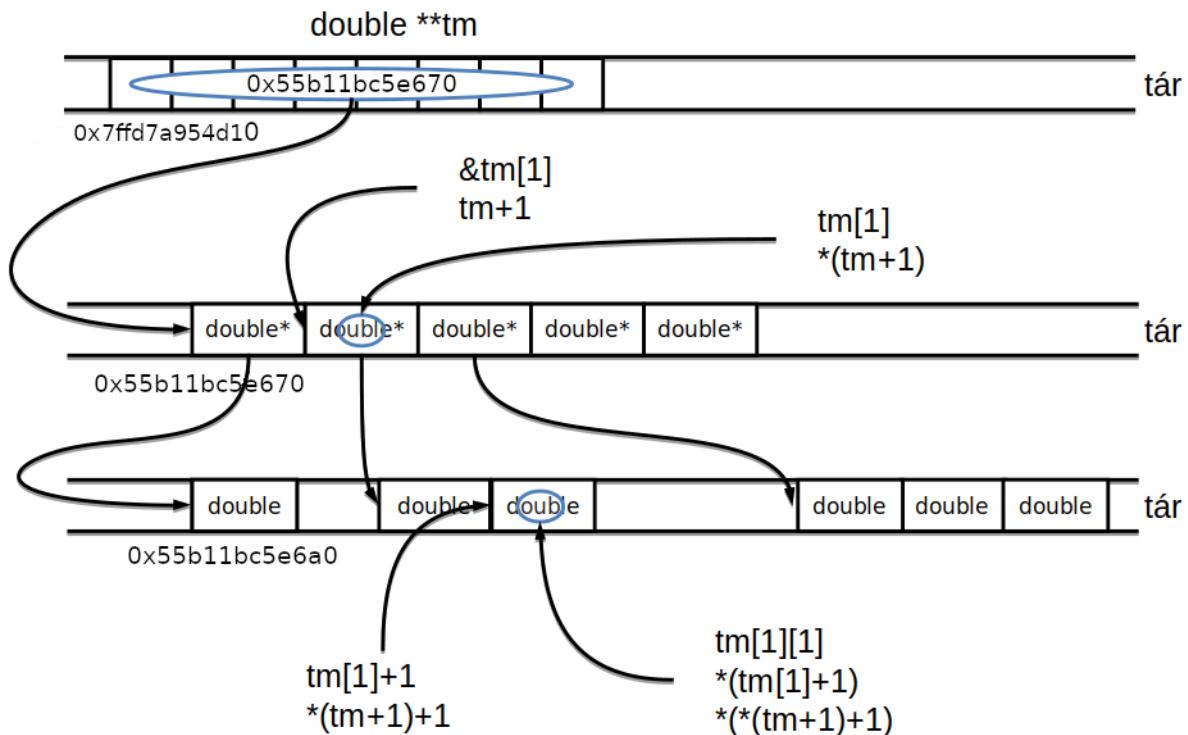
```

for (int i = 0; i < nr; ++i)
{
 for (int j = 0; j < i + 1; ++j)
 printf ("%f, ", tm[i][j]);
 printf ("\n");
}

```

Itt a mutatókkal ügyeskedtünk egy kicsit. Pointerek segítségével töltöttük fel a háromszögmátrixunk 4. sorát a 42.0 értéktől kezdődően. Látható, hogy az értékadás minden esetben különbözőféleképpen megy végbe. Ez a mutatók varázsa. Egy for cikluson belül most sem marad el a kiiratás, az értékeket természetesen megjelenítjük az outputon. Ez volt a háromszögmátrix 4. sora.

A program ezen részének megértéséhez azonban elengedhetetlen a mutatók/pointerek alaposabb szemre-vételezése. Erre tökéletes a Bátfai Norbert Tanár Úr által elkészített szemléletes ábra melyet a példámban megjelenő memóriacímekkel módosítottam.



A kép forrásának linkje: [eredeti](#)

A kép forrásának linkje: [saját](#)

```

for (int i = 0; i < nr; ++i)
 free (tm[i]);

free (tm);

```

A `free` függvénytellyel az előzőleg, `malloc` függvény által lefoglalt memóriamennyiséget szabadítjuk fel, hiszen nem használjuk tovább.

A programunk ezzel véget is ért, fordítás és futtatás után kész a háromszögmátrixunk!

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc haromszogmatrix.c -o haromszog
akos@akos-VirtualBox:~/Asztal/bhax/codes$./haromszog
0x7ffd7a954d10
0x55b11bc5e670
0x55b11bc5e6a0
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
6.000000, 7.000000, 8.000000, 9.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
42.000000, 43.000000, 44.000000, 45.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
```

## 4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása:

<https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/exor/e.c>

[https://gitlab.com/kincsa/bhax/blob/master/codes/caesar\\_codes/e.c](https://gitlab.com/kincsa/bhax/blob/master/codes/caesar_codes/e.c)

Tutor: Szegedi Csaba -<https://gitlab.com/dev.csaba.szegedi>

Tanulságok, tapasztalatok, magyarázat...

Ahogy arra a feladat megnevezéséből is következtetni lehetett, a programunk célja nem más, mint hogy egy számára átadott szöveget titkosítson, vagyis hogy ellehetetlenítse a szöveg olvasását. Erre nem mást, mint az XOR (EXOR) vagyis kizáró vagy logikai műveletet alkalmazza, amivel már találkoztunk a Helló, Turing! fejezet 2.3-as sorszámu feladatában.

A programkód elemzése:

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

#define MAX_KULCS 100
#define BUFFER_MERET 256
```

Include-olásra kerülnek függvénykönyvtáraink és bevezetjük nevesített konstainsainkat: MAX\_KULCS, melynek értéke 100 (a töréshez használt kulcs), illetve BUFFER\_MERET, mely értéke 256 (amiben az eredmény lesz eltárolva) lett. Későbbiekben a programunkban az előző konstansok (ahogy a nevük is sugallja) végig ezekkel az értékkel fognak rendelkezni.

```
int main (int argc, char **argv)
{
 char kulcs[MAX_KULCS];
 char buffer[BUFFER_MERET];

 int kulcs_index = 0;
 int olvasott_bajtok = 0;
```

```
int kulcs_meret = strlen (argv[1]);
strncpy (kulcs, argv[1], MAX_KULCS);
```

Kezdődik a main függvényünk. Ebben a pár sorban kerül létrehozásra a két, char típusú tömbünk melyek 100 és 256 méretűek lesznek. (Természetesen az előbb tárgyalt nevesített konstansok miatt.) Az egyikben a kulcs, míg másikban a már beolvasott karakterek lesznek eltárolva. Továbbá a kulcs\_index értelemszerűen a kulcs indexét, míg az olvasott\_bajtokban tárolódnak a már beolvasott bajtok. A kulcs\_meret változó értékét a futtatáskor megadott parancssori argumentum strlen-nel visszaadott hossza adja. A kulcs tömbbe másolódik az előbb parancssori argumentumként megadott, titkosítandó szöveg minden egyes karaktere strncpy használatával, ugyanis MAX\_KULCS-ig megy a másolás. Ebből következik, hogy minden karakter másolásra kerül.

```
while ((olvasott_bajtok = read (0, (void *) buffer, BUFFER_MERET)))
{
 for (int i = 0; i < olvasott_bajtok; ++i)

 buffer[i] = buffer[i] ^ kulcs[kulcs_index];
 kulcs_index = (kulcs_index + 1) % kulcs_meret;

}
write (1, buffer, olvasott_bajtok);
}
```

Ez a pár sor programunk egyik legfontosabb építőköve. Egy while ciklus az, ami közrefogja ezt a szakaszt mely ciklus addig fut, amíg van beolvasandó bajt. Ezt követően egy belső for ciklussal találkozunk, amely törzsében: A buffer tömb i-edik elemének értéke az i-edik elem és a kulcs tömb kulcs\_index-edik elemének vett EXOR műveleteredménye lesz. A kulcs\_index értékét a fent látható módon növeljük. Végül kiiratásra kerül a buffer összes eleme, vagyis maga a titkosított szövegünk:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ gcc e.c -o e
akos@akos-VirtualBox:~/Asztal/bhax/codes$ cat titkositando.txt
Ez a titkosítando szövegünk! Vajon sikerül majd dekodolni?
akos@akos-VirtualBox:~/Asztal/bhax/codes$./e 98365254 <titkositando.txt >titkositott.txt
akos@akos-VirtualBox:~/Asztal/bhax/codes$ cat titkositott.txt
|B|\@|@|\@RW@_AS[PV|\@LZDPSLVX|\@T^VV|\@|YPFLT|\@TXQ|\@]XYZ9akos@akos-VirtualBox:~/
/Asztal/bhax/codes$
```

Érdekesség, hogy ha újra futtatjuk a programot immáron a titkosított szöveget megadva inputként, az EXOR-ozás természetesen újra végbemegy és megkapjuk eredeti, titkosítatlan szövegünket:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$./e 98365254 <titkositott.txt > nemtit
kositott.txt
akos@akos-VirtualBox:~/Asztal/bhax/codes$ cat nemtitkositott.txt
Ez a titkosítando szövegünk! Vajon sikerül majd dekodolni?
akos@akos-VirtualBox:~/Asztal/bhax/codes$
```

## 4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html> 1.12. példája - Titkosítás kizáró vaggal és

[https://gitlab.com/kincsa/bhax/blob/master/codes/caesar\\_codes/ExorTitkos%C3%ADt%C3%B3.java](https://gitlab.com/kincsa/bhax/blob/master/codes/caesar_codes/ExorTitkos%C3%ADt%C3%B3.java)

Tanulságok, tapasztalatok, magyarázat...

Az elv ugyanaz, csupán a nyelv más. Kezdjünk bele a Java titkosítónk elemzésébe!

```
public class ExorTitkosító {
```

Egy nyilvános(an hozzáférhető) osztályban történik az algoritmusunk megírása. Ez az első sor, a C-től eltérően itt nincs szükség függvénykönyvtárak include-olására.

```
public ExorTitkosító(String kulcsSzöveg,
 java.io.InputStream bejövőCsatorna,
 java.io.OutputStream kimenőCsatorna)
```

Létrehozásra kerül az `ExorTitkosító` objektumunk, ami 3 paramétert kap majd. Magát a kulcsként használt szöveget, a be-illetve kimenő csatornát vagyis az inputot és outputot.

```
throws java.io.IOException {

 byte[] kulcs = kulcsSzöveg.getBytes();
 byte[] buffer = new byte[256];
 int kulcsIndex = 0;
 int olvasottBájtok = 0;
```

Az első sor kivételkezelés szempontjából igen fontos. Ha valamelyen input vagy output hibát talál a program, jelzi majd felénk. Definiáljuk a `kulcs` és definiáljuk `buffer`, byte típusú tömbjeinket. Az elsőbe természetesen a kulcsszöveg kerül majd, míg a másodiknak egyelőre csak a méretét adtuk meg. Két integer változó is létrehozásra került a kulcs indexének és a már beolvasott bájtok számának számontartására.

```
while ((olvasottBájtok =
 bejövőCsatorna.read(buffer)) != -1) {

 for (int i=0; i<olvasottBájtok; ++i) {

 buffer[i] = (byte) (buffer[i] ^ kulcs[kulcsIndex]);
 kulcsIndex = (kulcsIndex+1) % kulcs.length;
 }

 kimenőCsatorna.write(buffer, 0, olvasottBájtok);
}
```

Ha jobban megnézzük, lényegében szinte ugyanaz történik, mint a C-s megoldásunkban! A while ciklus addig fut, amíg a `buffer`ból van mit kiolvasni a `read` függvényvel. Tehát amíg van beolvasandó bájtunk. A `buffer` tömb i-edik elemének értéke az i-edik elem és a `kulcs` tömb `kulcs_index`-edik elemének vett EXOR műveleteredménye lesz, ez a művelet eredményezi majd a számunkra olvashatatlan szöveget. Itt típuskényszerítést alkalmaztunk, amikor az EXOR művelet előttük (`byte`) `kulcsszót`. Ezzel a művelet eredménye byte típusú lesz. A `kulcs_index` értékét a fent látható módon növeljük. Végül kiiratásra kerülnek a `buffer` elemei 0-tól `olvasottBájtok`-ig, a `write` függvényel az `outputra`. (`kimenőCsatorna`)

```
public static void main(String[] args) {

 try {

 new ExorTitkosító(args[0], System.in, System.out);

 } catch(java.io.IOException e) {

 e.printStackTrace();
 }
}
```

Eljutottunk a main-be, ahol egy try-catch szerkezet fogad minket. A try-on belül megtörténik a példányosítás. Létrehozunk egy új ExorTitkosítót a new kulcsszó segítségével amely a parancssori argumentumként kapott szöveget fogja átalakítani. A catch-ben meghívásra kerül a printStackTrace function, ami egy igen erős és hasznos eszköz kivételkezelés szempontjából, ugyanis megmondja, mi a probléma forrása és hogy az hol található.

Nem maradt más dolgunk, minthogy fordítsuk és futtassuk Java programunkat!

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ javac ExorTitkosító.java
akos@akos-VirtualBox:~/Asztal/bhax/codes$ java ExorTitkosító valami > ←
titkos.txt
Lássuk, hogyan működik a titkosító!
```

A titkos.txt fájlba szeretnénk a "Lássuk, hogyan működik a titkosító!" szövegünket titkosítani. Lássuk, sikerült-e!

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ java ExorTitkosító valami > titkos.txt
Lássuk, hogyan működik a titkosító!
akos@akos-VirtualBox:~/Asztal/bhax/codes$ more titkos.txt
:♦♦ IML []
[]♦♦ @g
```

A more parancsal kiiratjuk a fájl tartalmát és tényleg látható, hogy egy számunkra olvashatatlan szöveget kaptunk. Tehát az algoritmus helyes! Próbáljuk meg visszaalakítani!

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ java ExorTitkosító valami < ←
titkos.txt
Lássuk, hogyan működik a titkosító!
```

Tehát ha újra titkosítjuk a már eleve titkosított szövegünket, megkapjuk az eredeti szöveget! Így működik a Java titkosító és egyben törő.

## 4.4. C EXOR törő

Írj egy olyan C programot, amely megtöri az első feladatban előállított titkos szövegeket!

Megoldás videó:

Megoldás forrása: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/exor/t.c>

[https://gitlab.com/kincsa/bhax/blob/master/codes/caesar\\_codes/t.c](https://gitlab.com/kincsa/bhax/blob/master/codes/caesar_codes/t.c)

Tanulságok, tapasztalatok, magyarázat...

Ha már az előzőekben megírtuk C-ben a titkosítót, illene egy törőt is, nem igaz? Mert igaz, hogy a titkosítóval képesek vagyunk törésre is, azonban csak abban az esetben ha ismerjük a kulcsot. A következőkben nézett programban csupán a kulcs méretet szükséges tudnunk a kulcs előállításához. A következőkben a törőt fogjuk megnézni.

```
#define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 8
#define _GNU_SOURCE

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

A titkosító programhoz hasonló résszel találkozunk. Az include-olt függvénykönyvtárak ugyanazok, mint előző alkalommal, ebben nincs változás. Ugyanúgy nevesített konstansokat használunk majd ebben a programban is, az első 3 sorban vezetjük be őket. A program során az itt megadott névvel hivatkozunk rájuk, értékük végig adott lesz. Továbbá definiálásra kerül a \_GNU\_SOURCE is, ami segítségével számos új function-höz férhetünk majd hozzá, ilyen lesz például az strcasestr is.

```
double atlagos_szohossz (const char *titkos, int titkos_meret)
{
 int sz = 0;
 for (int i = 0; i < titkos_meret; ++i)
 if (titkos[i] == ' ')
 ++sz;

 return (double) titkos_meret / sz;
}
```

Itt kezdődik a függvények definiálása. Ez a függvény a szavak átlagos hosszát hivatott visszaadni egy double érték formájában és két paramétert kap. A szóközöket számolja, ha talál, az sz változóz növeli. Visszatérési értéke a méretet és a szószámot tartalmazó változóból megkapott átlag, melyet típuskényszerítünk double-re.

```
int tiszta_lehet (const char *titkos, int titkos_meret)
{
 double szohossz = atlagos_szohossz (titkos, titkos_meret);

 return szohossz > 6.0 && szohossz < 9.0
 && strcasestr (titkos, "hogy") && strcasestr (titkos, "nem")
 && strcasestr (titkos, "az") && strcasestr (titkos, "ha");
}
```

A tiszta\_lehet függvény célja az, hogy kiszűrjük a már eleve tiszta szöveget, szövegrészleteket. A függvényben létrehozásra kerül egy új, double típusú változó, aminek értéke az előbb létrehozott atlagos\_szohossz függvényből származik majd. Vagyis tárolja az input fájl átlag szóhosszát. Megvizsgálja, hogy a fájl

tartalmazza-e a leggyakoribb magyar szavakat, mert ha igen, valószínűleg már tiszta szövegről van szó. Ezt a strcasestr-rel tessük meg. Illetve az átlag szóhosszt is vizsgálja, ami a magyar nyelvben hozzávetőlegesen 6 és 9 közötti. Ezt is felhasználja ahhoz, hogy el tudja dönteni, tiszta vagy épp még titkos szövegről van szó. A visszatérési érték egész lesz.

```
void exor (const char kulcs[], int kulcs_meret, char titkos[], int ←
 titkos_meret)
{
 int kulcs_index = 0;
 for (int i = 0; i < titkos_meret; ++i)
 {
 titkos[i] = titkos[i] ^ kulcs[kulcs_index];
 kulcs_index = (kulcs_index + 1) % kulcs_meret;
 }
}
```

A függvény a paraméterként megkapott tömbök (kulcs, illetve titkos) bitjein rendre elvégzi az EXOR műveletet, ahogyan azt a neve is sugallja számunkra. Itt is a szokásokhoz híven a `^` operátor segítségével történik a művelet. Mivel a függvény void típusú, így nincs visszatérési értéke, csak végrehajt.

```
int
exor_tores (const char kulcs[], int kulcs_meret, char titkos[],
 int titkos_meret)
{
 exor (kulcs, kulcs_meret, titkos, titkos_meret);

 return tiszta_lehet (titkos, titkos_meret);
}
```

Ez a függvény lényegében egyesíti az eddig külön-külön már az előbb tárgyalt függvényeinket. Megtörténik az `exor` fv. meghívása, valamint a `tiszta_lehet`-é is, amely által produkált eredmény a függvény visszatérési értéke lesz. (0 vagy 1)

```
int main (void)
{
 char kulcs[KULCS_MERET];
 char titkos[MAX_TITKOS];
 char *p = titkos;
 int olvasott_bajtok;
```

Itt kezdődik a main-ünk. Inicializálásra kerülnek a `kulcs` és `titkos` tömbök a megfelelő értékekkel. Létrehozásra kerül egy `char` típusú `p` pointer, ami `titkosra` mutat. A beolvasott bajtok számontartására is definiálásra kerül egy változó, `olvasott_bajtok` néven.

```
// a titkos fajl berantasa
while ((olvasott_bajtok =
 read (0, (void *) p,
 (p - titkos + OLVASAS_BUFFER <
```

```

 MAX_TITKOS) ? OLVASAS_BUFFER : titkos + MAX_TITKOS - p));
 p += olvasott_bajtok;

 // maradek hely nullazasa a titkos bufferben
 for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
 titkos[p - titkos + i] = '\0';
}

```

Mi történik itt? Ahogy azt megjegyzésben olvasni lehet, maga a titkos fájl berántása. Egy while ciklusban történik a művelet aminek ciklusfejében lévő feltétel teljesülése esetén p értékét növelem az olvasott\_bajtokéval. Addig olvasom a bufferbe a bajtokat, míg van mit. Továbbá a bufferben fennmaradó helyet nullázzuk, azokkal nem kívánunk a feladat során foglalkozni.

```

//az osszes kulcs eloallitasa
for (int ii = '0'; ii <= '9'; ++ii)
 for (int ji = '0'; ji <= '9'; ++ji)
 for (int ki = '0'; ki <= '9'; ++ki)
 for (int li = '0'; li <= '9'; ++li)
 for (int mi = '0'; mi <= '9'; ++mi)
 for (int ni = '0'; ni <= '9'; ++ni)
 for (int oi = '0'; oi <= '9'; ++oi)
 for (int pi = '0'; pi <= '9'; ++pi)
 {
 kulcs[0] = ii;
 kulcs[1] = ji;
 kulcs[2] = ki;
 kulcs[3] = li;
 kulcs[4] = mi;
 kulcs[5] = ni;
 kulcs[6] = oi;
 kulcs[7] = pi;

 if (exor_tores (kulcs, KULCS_MERET, titkos, p - titkos))
 printf
("Kulcs: [%c%c%c%c%c%c%c]\nTiszta szoveg: [%s ←
] \n",
ii, ji, ki, li, mi, ni, oi, pi, titkos);

 // ujra EXOR-ozunk, igy nem kell egy masodik ←
 // buffer
 exor (kulcs, KULCS_MERET, titkos, p - titkos);
 }

 return 0;
}

```

A program utolsó szekciójához értünk. 8 egymásba ágyazott for ciklussal előállítjuk az összes lehetséges kulcsot, amelyet aztán a kulcs tömbbe tárolunk el. Jelen feladatban 8 bitesként kezeljük a kulcsosszt, ezért indokolt a 8 for ciklus, feltételezve azt, hogy 8 tagú a kulcsunk. A feladat során most épp annyi. Ha megvan a kulcs, kiiratásra kerül a tiszta, már titkosítatlan szöveg.

## 4.5. Neurális OR, AND és EXOR kapu

R

Megoldás videó: <https://youtu.be/Koyw6IH5ScQ>

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/NN\\_R](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/NN_R)

[https://gitlab.com/kincsa/bhax/blob/master/codes/caesar\\_codes/neuralis.r](https://gitlab.com/kincsa/bhax/blob/master/codes/caesar_codes/neuralis.r)

Tutoriált: Szegedi Csaba -<https://gitlab.com/dev.csaba.szegedi>

Tanulságok, tapasztalatok, magyarázat...

Elöljáróban: mik azok a neurális hálózatok, miért is jók nekünk? Manapság a fogalmat kétféleképpen értelmezhetjük. Léteznek biológiai- és mesterséges neurális hálózatok. Minket utóbbi fog érdekelni. A mesterséges neurális hálózatok működése a biológiai hálózatok pár tulajdonságán alapszik. Az egyik legfontosabb jellemzője, hogy képes egy átadott minta alapján tanulni vagyis megvalósul a gépi tanulás. A következő feladatban egy R nyelvben írt programot fogunk kielemezni, hogyan is képes egyes logikai műveletek megtanulására.

```
library(neuralnet)
```

A neuralnet könyvtárra van szükségünk a feladat megoldásához.

```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
OR <- c(0,1,1,1)

or.data <- data.frame(a1, a2, OR)
```

Létrehozzuk az a1, a2 változóinkat a képen látható bájtokat tartalmazva és az OR változót, amely az azonos indexű biteken végzett VAGY művelet eredményét adja vissza. A data.frame függvénytel egy adatkeretet létrehozunk ezekből a változókból majd ez bekerül a program data (adat) részébe, ezekből az adatokból fogja megtanulni a háló a művelet megoldását.

```
nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE, ←
 stepmax = 1e+07, threshold = 0.000001)

plot(nn.or)

compute(nn.or, or.data[,1:2])
```

Az nn.or kerül a neuralnet függvény által visszaadott érték, mely függvény számos paraméterrel rendelkezik.

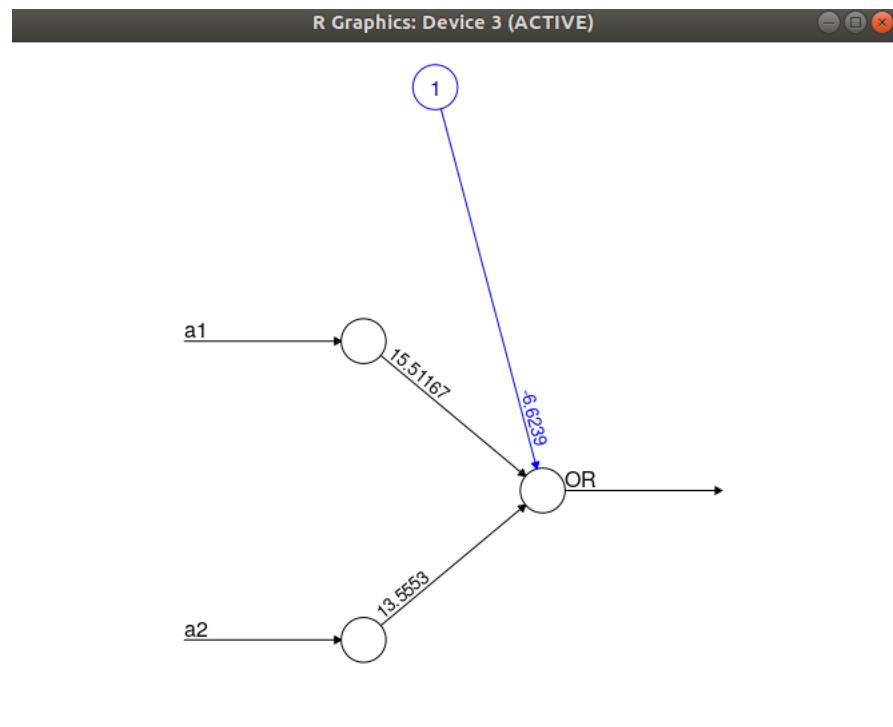
Ezt követően a plot függvénytel kirajzoltatjuk az eredményt.

A compute függvénytel pedig kiszámoltatjuk az eredményt.

```
$neurons[[1]]
 a1 a2
[1,] 1 0
[2,] 1 1
[3,] 1 0
```

```
[4,] 1 1 1
```

```
$net.result
[,1]
[1,] 0.001433089
[2,] 0.999362223
[3,] 0.999118762
[4,] 0.999999999
```



Ebből is látszik, hogy helyes eredményt kaptunk!

```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
OR <- c(0,1,1,1)
AND <- c(0,0,0,1)

operand.data <- data.frame(a1, a2, OR, AND)

nn.operand <- neuralnet(OR+AND~a1+a2, operand.data, hidden=0, linear. ←
output=FALSE, stepmax = 1e+07, threshold = 0.000001)

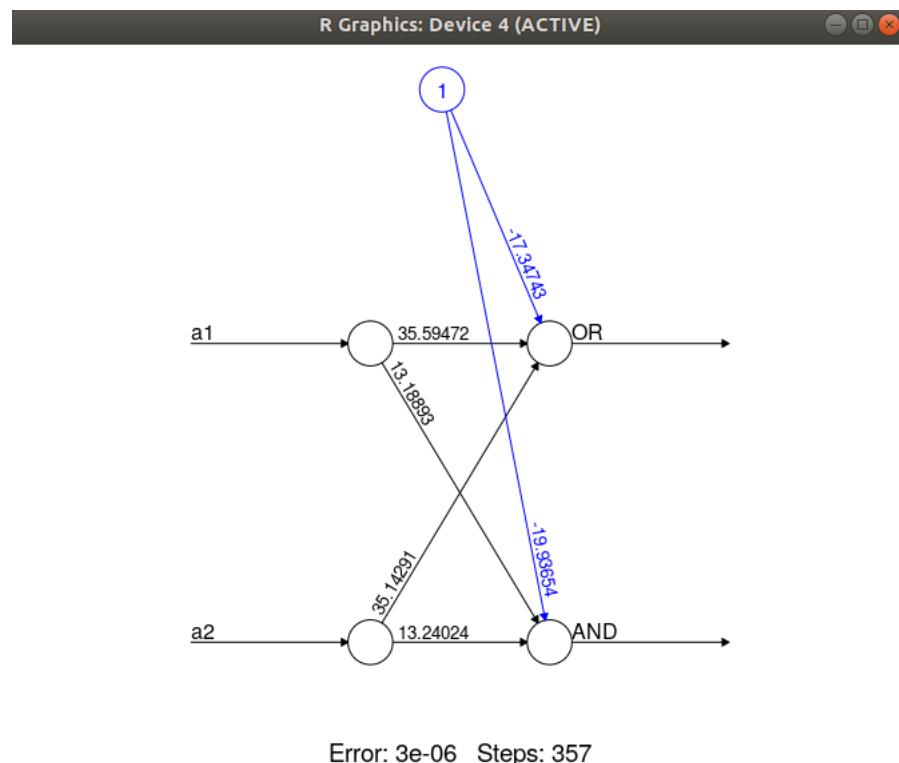
plot(nn.operand)
```

```
compute(nn.operand, operand.data[,1:2])
```

Ugyanezt a folyamatot játszik le akkor, amikor a logikai ÉS műveletet tanítjuk meg a hálózatnak. Lent láthatjuk, hogy az eredmény itt is jó:

```
$neurons[[1]]
 a1 a2
[1,] 1 0
[2,] 1 1
[3,] 1 0
[4,] 1 1

$net.result
 [,1] [,2]
[1,] 5.697714e-05 2.627334e-09
[2,] 9.999619e-01 1.316939e-03
[3,] 9.999816e-01 1.253087e-03
[4,] 1.000000e+00 9.984145e-01
```



```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
EXOR <- c(0,1,1,0)
```

```
exor.data <- data.frame(a1, a2, EXOR)

nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=0, linear.output ←
=FALSE, stepmax = 1e+07, threshold = 0.000001)

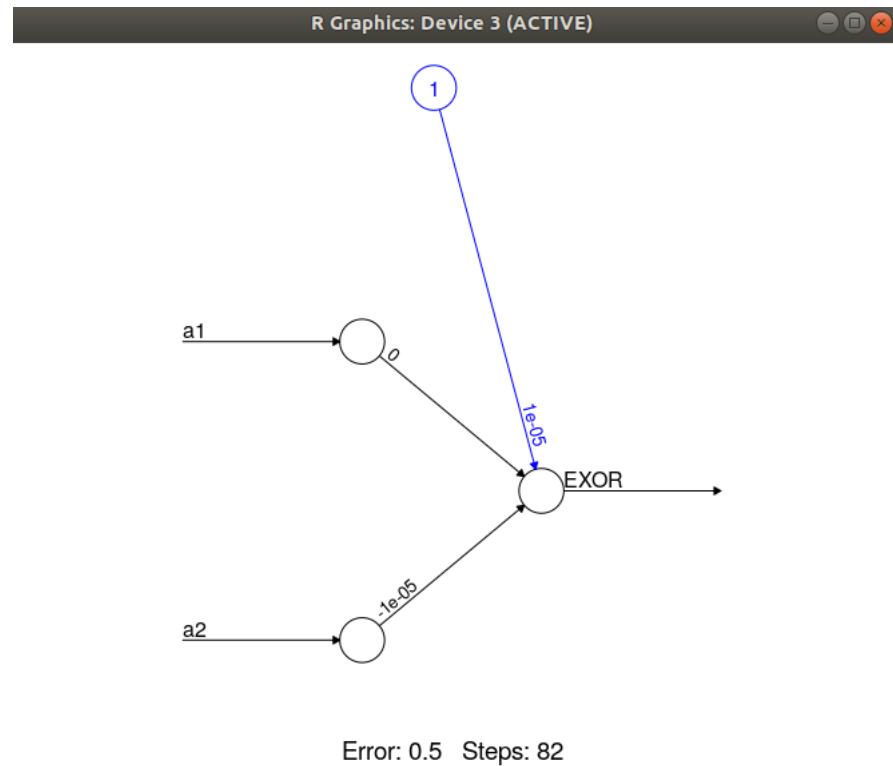
plot(nn.exor)

compute(nn.exor, exor.data[,1:2])
```

Ha már 2 logikai műveletet sikeresen megtanítottunk neki, miért is ne tudná megtanulni a 3.-at is, nem igaz? Az EXOR (XOR/kizáró vagy) műveletét tanítjuk meg itt neki.

```
$neurons[[1]]
 a1 a2
[1,] 1 0
[2,] 1 1
[3,] 1 0
[4,] 1 1

$net.result
 [,1]
[1,] 0.5000021
[2,] 0.5000013
[3,] 0.5000003
[4,] 0.4999995
```



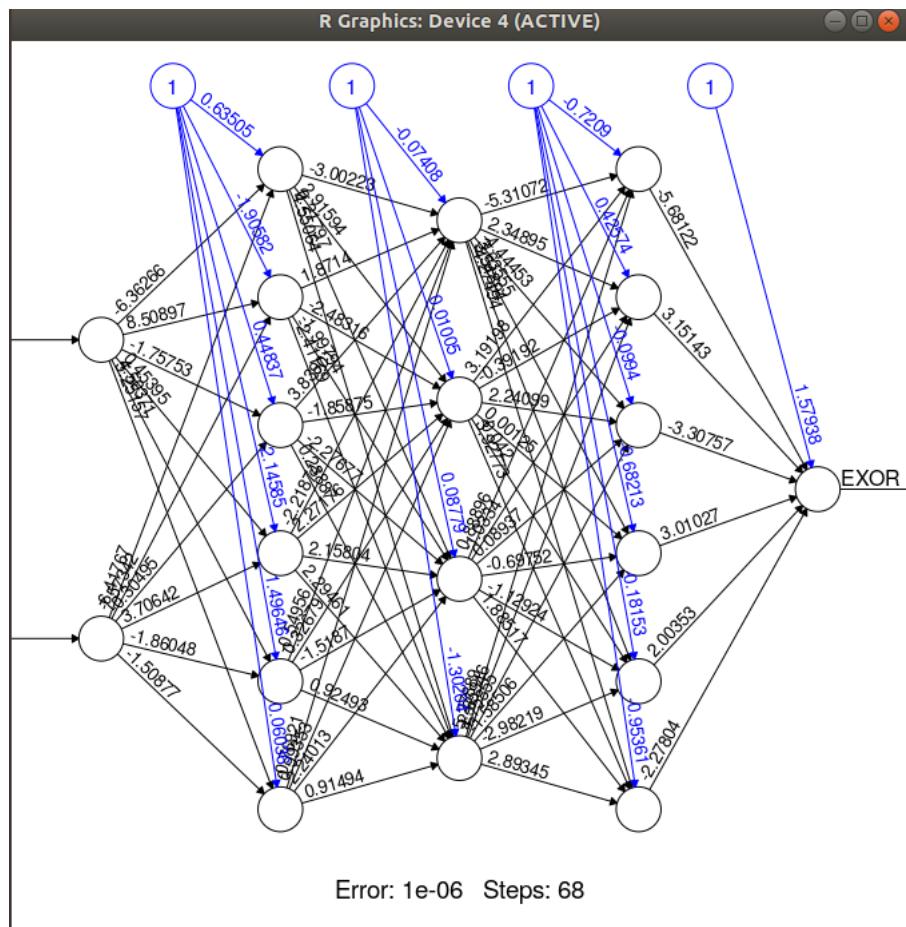
Látható, hogy mindenhol köztes értékeket kaptunk, tehát hiba csúszott a gépezetbe. Itt 50 %-os hibaárányról beszélünk.

Na de mi lehet a megoldás?

```
nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=c(6, 4, 6), ←
linear.output=FALSE, stepmax = 1e+07, threshold = 0.000001)
```

Ez, a fenti egy sor változik. Ha bevezetünk rejtett neuronokat, helyes végeredményt kapunk!

```
$net.result
[,1]
[1,] 0.0003154876
[2,] 0.9991795416
[3,] 0.9995412678
[4,] 0.0009422256
```



Ezzel kijelenthetjük, hogy sikeres volt a gépi tanulás, képesek voltunk a tanításra.

## 4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó: <https://www.youtube.com/watch?v=XpBnR31BRJY>

Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/codes/caesar\\_codes/main.cpp](https://gitlab.com/kincsa/bhax/blob/master/codes/caesar_codes/main.cpp)

[https://gitlab.com/kincsa/bhax/blob/master/codes/caesar\\_codes/mlp.hpp](https://gitlab.com/kincsa/bhax/blob/master/codes/caesar_codes/mlp.hpp)

Tutoriált: Heinrich László -<https://gitlab.com/heinrichlaszlo>

Tanulságok, tapasztalatok, magyarázat...

Érdemes lenne először a fogalmat megmagyarázni. A Neurális OR, AND és EXOR kapu feladatnál már találkozhattunk a neuron és a gépi tanulás fogalmával. A perceptron nem más mint ezen neuron mesterséges intelligenciában használt változata. Szintén tanulásra képes, a bemenő 0-k és 1-esek sorozatából mintákat tanul meg és súlyozott összegzést végez.

A következő feladat során egy ilyen perceptron fogunk elkészíteni, aminek alapvetően a feladata az, hogy a mandelbrot .cpp programunk által létrehozott Mandelbrot-halmazt ábrázoló PNG kép egyik színkódját vegye és az a színkód legyen a többrétegű neurális háló inputja. Lássuk a programot!

```
#include <iostream>
```

```
#include "mlp.hpp"
#include <png++/png.hpp>
```

Include-oljuk az iostream, az mlp és a png++/png könyvtárakat. Utóbbi kettőt azért, mert többrétegű perceptronról akarunk majd létrehozni (Multi Layer Perceptron), így muszáj ezt a könyvtárat include-olunk a programunkba. Utolsó könyvtárunk ahogyan a neve is sugallja, a PNG képállományokkal való munkát teszi lehetővé.

```
using namespace std;

int main(int argc, char **argv)
{
 png::image<png::rgb_pixel> png_image(argv[1]);

 int size = png_image.get_width() * png_image.get_height();

 Perceptron *p = new Perceptron(3, size, 256, 1);
```

Kezdődik a main függvényünk. Első sorában megmondjuk, hogy az 1-es parancssori argumentum alapján kerül beolvasásra a képállomány. Ettől kezdve dolgozni tudunk. A kép méretét a get\_width és a get\_height szorzatából kapjuk, ezt el is tároljuk egy segédváltozóban. A következő sorban létrehozásra kerül a perceptronunk a new operátor segítségével, amely paraméterei balról jobbra haladva: a rétegek száma, 1. réteg neuronjai az inputrétegen, 2. réteg neuronjai az outputrétegen, az eredmény (jelen esetben 1 szám)

```
double* image = new double[size];

for(int i=0; i<png_image.get_width(); ++i)
 for(int j=0; j<png_image.get_height(); ++j)
 image[i*png_image.get_width()+j] = png_image[i][j].red;

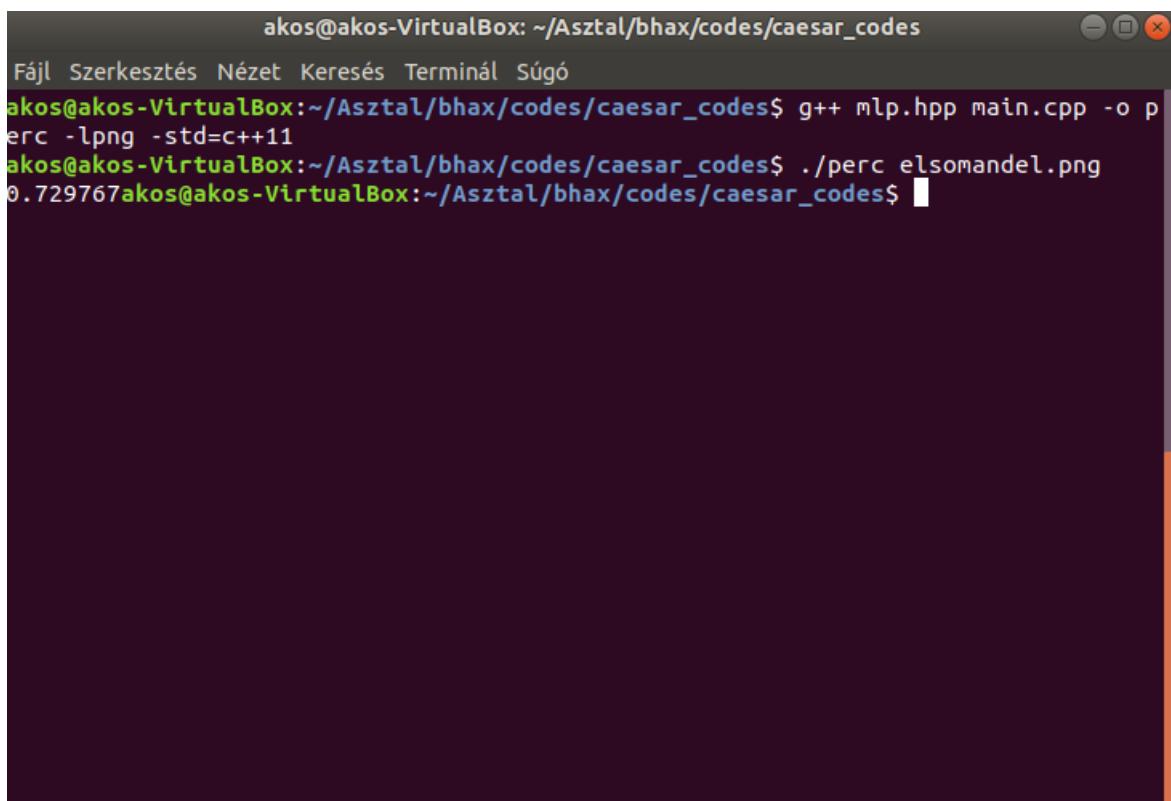
double value = (*p)(image);

cout<<value;

delete p;
delete [] image;
```

Létrehozásra kerül egy double típusú. Az egyik for ciklus végigmegy a kép szélességét alkotó pontokon, a másik pedig a magasságán. Miután végigmentünk a képpontokon, az image tárolni fogja a képállomány vörös színkomponensét. A value értéke a Perceptron image-re történő meghívása adja majd. Így a perceptronban tárolásra kerül a vörös színkomponens. A value változó egy double típusú számot tárol. Ez kiiratásra kerül és töröljük tovább nem használatos elemeket, hiszen a számukra eddig fenntartott memória-területet érdemes felszabadítanunk, így a lefoglalt memóriamennyiséget újra használható. Programunk ezzel véget is ért.

Fordítjuk és futtatjuk a képen látható módon:



A screenshot of a terminal window titled "akos@akos-VirtualBox: ~/Asztal/bhax/codes/caesar\_codes". The window contains the following text:

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos-VirtualBox:~/Asztal/bhax/codes/caesar_codes$ g++ mlp.hpp main.cpp -o perc -lpng -std=c++11
akos@akos-VirtualBox:~/Asztal/bhax/codes/caesar_codes$./perc elsomandel.png
0.729767akos@akos-VirtualBox:~/Asztal/bhax/codes/caesar_codes$ █
```

## 5. fejezet

# Helló, Mandelbrot!

### 5.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása: [https://progpater.blog.hu/2011/03/26/kepes\\_egypercesek](https://progpater.blog.hu/2011/03/26/kepes_egypercesek)

[https://gitlab.com/kincsa/bhax/blob/master/codes/mandelbrot\\_codes/mandelpng.c++](https://gitlab.com/kincsa/bhax/blob/master/codes/mandelbrot_codes/mandelpng.c++)

Tutoriált: Heinrich László -<https://gitlab.com/heinrichlaszlo>

Mandelbrot lengyel származású matematikus, a 20. században alkotott, az Ő nevéhez fűződik a Mandelbrot-halmaz fogalom is, amit a komplex számsíkon ábrázolunk, a '70-es évek végén fedezte fel a halmazt. A Mandelbrot-halmazok ábrázolásának alapjául szolgáló számok tehát a **komplex számok**. Ezen komplex számok közös jellemzője, hogy akkor a Mandelbrot halmaz elemei, ha őket az  $x_{n+1} = x_n^2 + c$  sorozatba behelyettesítve egy konvergens sorozatot kapunk. A halmaz ábrázolása egy **fraktál** alakzatot eredményez majd.

A bevezető elméleti háttérének forrásául szolgált: [link](#)

Ebben a feladatban egy olyan programot fogunk megnézni, ami egy ilyen Mandelbrot-halmazt állít elő. A program a következőképpen indul:

```
#include <iostream>
#include "png++/png.hpp"
```

A programunk legelején inkludáljuk a `png++/png` könyvtárat, ami lehetővé teszi számunkra a PNG kép-állományokkal való munkát. Ez elengedhetetlenül szükséges nekünk, ugyanis egy PNG képet szeretnénk megrajzolatni programunkkal.

Természetesen ha ez a könyvtár még nincs telepítve, a feladat megoldásához telepítenünk szükséges.

```
int main (int argc, char *argv[])
{
 if (argc != 2) {
 std::cout << "Hasznalat: ./mandelpng fajlnev";
 return -1;
 }
}
```

Mivel parancssori argumentumként adjuk meg, hogy milyen fájlba legyen kimentve a képünk, ezért ha ez elmarad, jelezük a felhasználó felé a helyes futtatási módot.

```
double a = -2.0, b = .7, c = -1.35, d = 1.35;
int szelesseg = 600, magassag = 600, iteraciosHatar = 1000;

png::image <png::rgb_pixel> kep (szelesseg, magassag);

double dx = (b-a)/szelesseg;
double dy = (d-c)/magassag;
double reC, imC, rez, imZ, ujrez, ujimZ;

int iteracio = 0;
std::cout << "Szamitas";
```

Ahogy az a megjegyzésben is látható, létrehozzuk a számolásunk alapjait szolgáló változókat. A double típusú változókkal a számsík határait adjuk meg (mind az X, mind pedig az Y-tengelyen - ), ezen kívül létrehozásra kerül két, a kép szélességének és magasságának tárolására alkalmas változó, illetve egy, az iterációk darabszámának felső korlátját jelentő érték is.

Létrehozzuk a képünket, aminek mérete: szelesseg x magassag, amely kép jelen pillanatban még üres. A dx és dy változókat a számsíkon való léptetéshez létrehozzuk. C és Z változókhöz létrehozzuk az im és re előtagú változókat, amik az adott komplex szám (C vagy Z) valós és képzetes részét jelölik.

Az iterációk számának nyilvántartására is létrehozunk egy egész típusú változót 0 értékkel.

```
// Végigzongorázzuk a szélesség x magasság rácson:
for (int j=0; j<magassag; ++j) {
 for (int k=0; k<szelesseg; ++k) {

 reC = a+k*dx;
 imC = d-j*dy;
 // z_0 = 0 = (rez, imZ)
 rez = 0;
 imZ = 0;
 iteracio = 0;
 }
}
```

Mint ahogyan a megjegyzés is írja, két for ciklussal végigmegyünk a szelesseg x magassag rácson, majd inicializáljuk C valós és képzetes részeit tartalmazó változókat, majd ugyanezt tesszük Z esetében is. Az iterációk számát tartalmazó változó is létrehozásra kerül ez fontos lesz a program következő részében.

```
while (rez*rez + imZ*imZ < 4 && iteracio < iteraciosHatar) {

 ujrez = rez*rez - imZ*imZ + reC;
 ujimZ = 2*rez*imZ + imC;
 rez = ujrez;
 imZ = ujimZ;

 ++iteracio;
```

Egy while ciklusban megvizsgáljuk, hogy  $z_n$  négyzete kisebb-e mint 4, illetve hogy elértek-e az iterációs határt. Ha a ciklusfejben található feltétel teljesül, akkor módosítjuk a  $Z$  változó értékeit, úgy, hogy változó értéke így módosuljon:  $z_n^2 + c$ .  $Z$  változó új értékeit  $Z$  eredeti változóiba másoljuk. Mivel ez egy iteráció volt, iteracio számlálót növeljük. Ha az iteracio számláló eléri a határt, azt mondhatjuk, hogy az iterációnak van határértéke, vagyis a C szám eleme a Mandelbrot-halmaznak, így a hozzá tartozó képpontot színezzük.

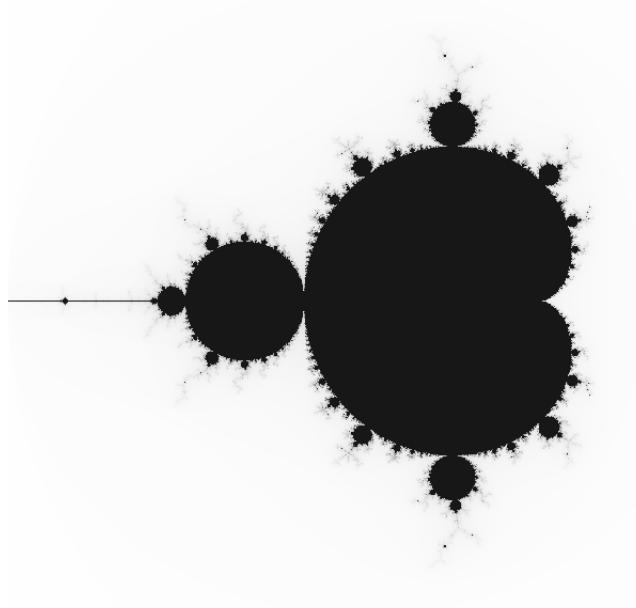
```
kep.set_pixel(k, j, png::rgb_pixel(255-iteracio%256,
 255-iteracio%256, 255- ←
 iteracio%256));
kep.write(argv[1]);
std::cout << argv[1] << " mentve" << std::endl;
```

A set\_pixel függvényel a k,j koordinátákon található képpontokhoz férünk hozzá, majd ezen képpontok színét az rgb\_pixel függvényel módosítjuk.

A kep fájlba írjuk kirajzolt Mandelbrot-halmazunkat. A program legvégén üzenet jelez vissza a felhasználónak a png fájl létrejöttéről.

Fordítjuk és futtattuk a programot a következő parancsokkal:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ g++ mandelpng.cpp `libpng-config ←
--ldflags` -o mandelpng
akos@akos-VirtualBox:~/Asztal/bhax/codes$./mandelpng elsomandel.png
```



## 5.2. A Mandelbrot halmaz a std::complex osztálytal

Megoldás videó:

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/blob/master/attention\\_raising/Mandelbrot/3.1.2.cpp](https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/Mandelbrot/3.1.2.cpp)

[https://gitlab.com/kincsa/bhax/blob/master/codes/mandelbrot\\_codes/3.1.2.cpp](https://gitlab.com/kincsa/bhax/blob/master/codes/mandelbrot_codes/3.1.2.cpp)

A feladat ugyanaz, mint az előbb! Ugyanúgy egy olyan program elkészítése a cél, ami egy Mandelbrot-halmazt generál, azonban most az `std::complex` osztály használatával. A megoldásnak a lényege annyi, hogy nem szükséges a feladat megoldásához szükséges számok valós- és képzetes részét külön-külön változóba tárolni, ugyanis megoldható egyetlen darabban is. A program így kezdődik:

```
#include <iostream>
#include "png++/png.hpp"
#include <complex>
```

Az első két könyvtár már ismerős, ezeket nem részletezném. Továbbá ahogy a feladat is kérte, include-oljuk a komplex számokkal való könnyebb számolást eredményező, `complex` függvénykönyvtárat is.

```
int main (int argc, char *argv[])
{
 int szelesseg = 1920;
 int magassag = 1080;
 int iteraciosHatar = 255;
 double a = -1.9;
 double b = 0.7;
 double c = -1.3;
 double d = 1.3;
```

Kezdődik a `main` függvényünk. Itt megadjuk a készülő képünk méreteit és egy iterációs határt is, ez lesz a maximálisan megengedett iterációk száma. Ezen kívül a komplex számsík határértékei is inicializálásra kerülnek.

```
if (argc == 9)
{
 szelesseg = atoi (argv[2]);
 magassag = atoi (argv[3]);
 iteraciosHatar = atoi (argv[4]);
 a = atof (argv[5]);
 b = atof (argv[6]);
 c = atof (argv[7]);
 d = atof (argv[8]);
}
else
{
 std::cout << "Hasznalat: ./mandelbrot_komplex fajlnev szelesseg ←
 magassag n a b c d" << std::endl;

 return -1;
}
```

Ebben a pár sorban ellenőrizzük a felhasználót. Kezdődik az if ág. Ha az általa, futtatáskor megadott parancssori argumentumok száma 9, akkor az előző programrészben definiált változók értékének beállítjuk

a parancssori argumentumokat. Mindez az `atoi` -amely stringet alakít át integerré -, illetve az `atof` - ami pedig stringet double-lé alakít át - függvényekkel történik.

Érkezik az else ág. Hogyha a felhasználó által megadott parancssori argumentumok száma nem 9, akkor egyszerűen kiiratjuk, mi az argumentumok helyes sorrendje vagyis jóformán egy futtatási "sablont" adunk a felhasználónak. Ezen kívül -1-gyel ergő hibával tér vissza a program.

```
png::image < png::rgb_pixel > kep (szelesseg, magassag);

double dx = (b - a) / szelesseg;
double dy = (d - c) / magassag;
double reC, imC, rez, imZ;
int iteracio = 0;
```

A `png` függvénykönyvtár `image` függvényével létrehozzuk `szelesseg*magassag` felbontású képünket. Jelen pillanatban természetesen ez egy üres `png` állomány. A `dx`, `dy` változókat definiáljuk, ők kellenek a képernyőn való léptetéshez, a `reC`, `imC`, `rez`, `imZ` változókat pedig deklaráljuk. Ezen kívül új, double típusú változókat definiálunk és az `iteracio` változó értékét 0-ra állítjuk be. A "re" előtag minden a komplex szám valós részét, míg az "im" előtag ez esetben is a komplex szám képzetes részét jelöli.

```
std::cout << "Szamitas\n";

for (int j = 0; j < magassag; ++j)
{
 for (int k = 0; k < szelesseg; ++k)
 {
```

Két for ciklust indítunk, az egyiket `magassagig`, a másikat pedig `szelessegig`. Igazából egy "rácsot" járunk be a komplex számsíkon.

```
reC = a + k * dx;
imC = d - j * dy;
std::complex<double> c (reC, imC);

std::complex<double> z_n (0, 0);
iteracio = 0;
```

Az `reC` és a `imC` változó értékét a képen látható módon beállítom. A program elején inkludált `complex` osztály segítségével létrehozzuk a `c` változót, ami megkapja a komplex szám valós- és képzetes részét is. Hasonló módon megkapjuk `z_n` értékét is. Ugyanúgy, a zárójelben lévő 1. szám a valós rész, míg a 2. szám a képzetes része lesz a számnak. A `c` változó lesz a vizsgált rácspont programunk során.

```
while (std::abs (z_n) < 4 && iteracio < iteraciosHatar)
{
 z_n = z_n * z_n + c;

 ++iteracio;
}
```

```
 kep.set_pixel (k, j,
 png::rgb_pixel (iteracio%255, (iteracio*iteracio ←
)%255, 0));
}
```

Egy while ciklusban megvizsgáljuk, hogy  $z_n$  abszolútértéke kisebb-e mint 4, illetve hogy elérőük-e az iterációs határt. Ha a ciklusfejben található feltétel teljesül-e. Ha igen:

$z_n$  értékét  $z_n^2 + c$ -re változtatom, ugyanis ez a halmazt felépítő sorozat képlete. Az `iteracio` szám-lálót egyel növelem, ugyanis csak addig tart a ciklus, míg el nem értük az iterációs határt.

Ha a határt elérjük, az azt jelenti, hogy az adott pont Mandelbrot-halmaz elem, ezért azt be kell jelölnünk a számsíkon.

Itt jön a lényeg. A kepre meghívjuk a `set_pixel` függvényt, ami a  $k$ ,  $j$ -edik képpontot (pixelt) az `rgb_pixel` függvénynek átadott RGB paramétereiből előállított színnel fog kiszínezni.

```
int szazalek = (double) j / (double) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}
```

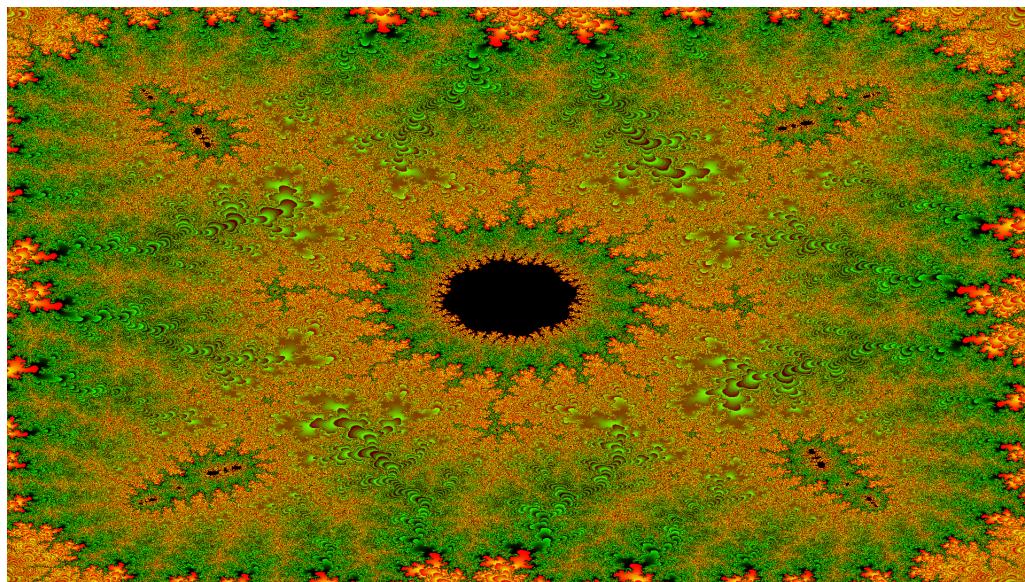
Létrehozásra kerül az integer típusú `szazalek` változónk, ami futtatás után azt jelöli majd, milyen stádiumban áll a képünk kirajzoltatása. Természetesen ez a változó kiiratásra is kerül a következő sorban, a felhasználó tájékoztatása céljából. Meghívjuk a `flush` függvényt, ezzel biztosítva azt, hogy a bufferben lévő összes bájt kiiratásra kerüljön.

```
kep.write (argv[1]);
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}
```

Ez a programot záró két sor. Az elkészült képünket az 1-es indexű parancssori argumentumként megadott fájlba írjuk a `write` függvényvel. A legutolsó sorban tájékoztatásképp kiiratunk egy üzenetet, miszerint a képünk el lett mentve.

Fordítjuk és futtatjuk a programot:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ g++ 3.1.2.cpp -lpng -O3 -o ←
3.1.2
akos@akos-VirtualBox:~/Asztal/bhax/codes$./3.1.2 masodikmandel.png ←
1920 1080 2040 -0.01947381057309366392260585598705802112818 ←
-0.0194738105725413418456426484226540196687 ←
0.7985057569338268601555341774655971676111 ←
0.798505756934379196110285192844457924366
```



### 5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbqRzY76E>

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/Biomorf](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/Biomorf)

[https://gitlab.com/kincsa/bhax/blob/master/codes/mandelbrot\\_codes/3.1.3.cpp](https://gitlab.com/kincsa/bhax/blob/master/codes/mandelbrot_codes/3.1.3.cpp)

Tutor: Heinrich László -<https://gitlab.com/heinrichlaszlo>

Tanulságok, tapasztalatok, magyarázat...

Mostanra már megszokottan kezdjük a fogalmak tisztázásával a feladatot! A biomorfok leginkább olyan formák, alakzatok amelyek valamilyen biológiai organizmusra hasonlítanak. Pickover amerikai kutató egy Julia-halmazhoz kapcsolódó programban található hiba következtében fedezte fel ezeket az alakzatokat.

A bevezető elméleti háttérének forrásául szolgált: [link](#)

[link](#)

Aktuális feladatunk során egy ilyen biomorfot fogunk megrajzolatni felhasználva az előző, Mandelbrot-halmazos feladatot. A kirajzolt alakzat az előzőhez hasonlóan szintén fraktál lesz.

A program alapjául szolgáló cikk itt olvasható: [link](#)

Nézzük a program forráskódját.

```
#include <iostream>
#include "png++/png.hpp"
#include <complex>
```

Az inkludált könyvtárak ugyanazok mint legutóbb, ebben nincs változás.

```
int main (int argc, char *argv[])
{
 int szelesseg = 1920;
```

```
int magassag = 1080;
int iteraciosHatar = 255;
double xmin = -1.9;
double xmax = 0.7;
double ymin = -1.3;
double ymax = 1.3;
double reC = .285, imC = 0;
double R = 10.0;
```

Néhány apró különbség hiján ez a szakasz is megegyezik az előbb látott program azonos szakaszával. A különbség annyi, hogy a C vizsgált rácspontot definiáló `reC` (a C komplex szám valós része) és `imC` (a C komplex szám képzetes része) változókat a program elején definiáljuk.

```
if (argc == 12)
{
 szelesseg = atoi (argv[2]);
 magassag = atoi (argv[3]);
 iteraciosHatar = atoi (argv[4]);
 xmin = atof (argv[5]);
 xmax = atof (argv[6]);
 ymin = atof (argv[7]);
 ymax = atof (argv[8]);
 reC = atof (argv[9]);
 imC = atof (argv[10]);
 R = atof (argv[11]);

}
else
{
 std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag ←
 n a b c d reC imC R" << std::endl;
 return -1;
}
```

A felhasználó ellenőrzése, hogy helyesen futtata-e a programot. Ha nem, kiiratjuk neki az elvárt (és helyes) módot. Ha igen, a megfelelő indexű parancssori argumentumokat az `atof` és `atoi` függvényekkel átalakítjuk megfelelő típusú (integer vagy double) értékeké és ezeket az értékeket hozzárendeljük a változóinkhoz.

```
png::image<png::rgb_pixel> kep (szelesseg, magassag);

double dx = (xmax - xmin) / szelesseg;
double dy = (ymax - ymin) / magassag;

std::complex<double> cc (reC, imC);

std::cout << "Szamitas\n";
```

Létrehozásra kerül a képünk, melynek mérete: `szelesseg*magassag`. Definiáljuk a `cc` komplex számot is, aminek paraméterei az előbb parancssori argumentumként kapott valós és képzetes részek lesznek.

```
for (int y = 0; y < magassag; ++y)
{
 for (int x = 0; x < szelessseg; ++x)
{
```

A már előbb is látott módon két for ciklussal végigmegyünk a rácsokon.

```
double reZ = xmin + x * dx;
double imZ = ymax - y * dy;
std::complex<double> z_n (reZ, imZ);

int iteracio = 0;
for (int i=0; i < iteraciosHatar; ++i)
{
 z_n = std::pow(z_n, 3) + cc;
 //z_n = std::pow(z_n, 2) + std::sin(z_n) + cc;
 if (std::real (z_n) > R || std::imag (z_n) > R)
 {
 iteracio = i;
 break;
 }
}
kep.set_pixel (x, y,
 png::rgb_pixel ((iteracio*20)%255, (iteracio ←
 *40)%255, (iteracio*60)%255));
}
```

A valós és képzetes részek, illetve a complex osztály segítségével létrehozzuk a  $z_n$  változónkat. Egy harmadik for cikluson belül történik a lényeg, ami addig fut, míg el nem éri az iterációs határt. A  $z_n$  értékét a cikkben lévő összefüggés/képlet szerint  $z_n = z_n^3 + cc$ -re változtatjuk meg, az érintett koordinátákon megtalálható képpontokat színezzük.

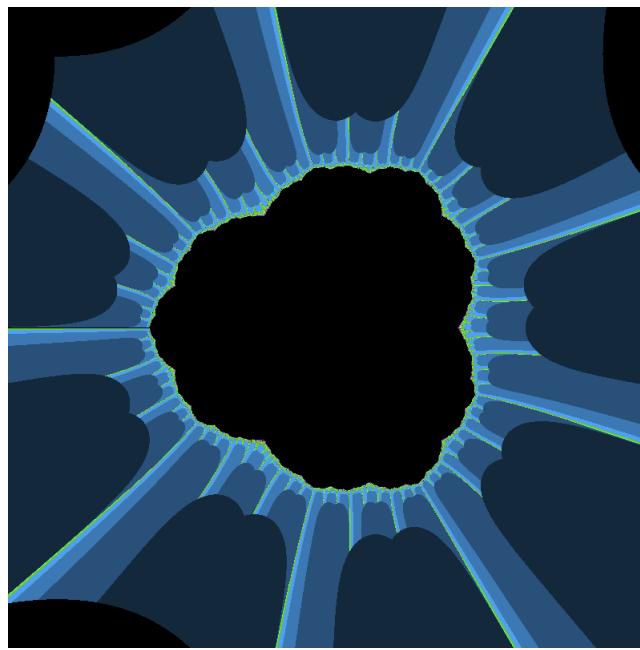
```
int szazalek = (double) y / (double) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;

kep.write (argv[1]);
std::cout << "\r" << argv[1] << " mentve." << std::endl;
```

Az előzővel megegyező módon létrehozásra kerül a százalék nyilvántartására szolgáló változó, aminek értékét annak függvényében változtatjuk, hogyan is áll a rács bejárása. Ezt tájékoztatás céljából a képernyőre is kiírjuk, végül fájlba írjuk a képünket.

Fordítás és futtatás után a következő eredményt kapjuk:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes$ g++ 3.1.3.cpp -lpng -O3 - ←
o 3.1.3
akos@akos-VirtualBox:~/Asztal/bhax/codes$./3.1.3 bmorf.png 800 800 ←
10 -2 2 -2 2 .285 0 10
```



## 5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás video:

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/blob/master/attention\\_raising/CUDA/mandelpngc\\_60x60\\_100.cu](https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/CUDA/mandelpngc_60x60_100.cu)

[https://gitlab.com/kincsa/bhax/blob/master/codes/mandelbrot\\_codes/mandelpngc\\_60x60\\_100.cu](https://gitlab.com/kincsa/bhax/blob/master/codes/mandelbrot_codes/mandelpngc_60x60_100.cu)

Tapasztalat, tanulságok, magyarázat...

Tisztázzuk először mi is a CUDA maga. A CUDA az Nvidia által kifejlesztett, jellemzően C, C++ nyelvekkel remek összhangban működő technológia ami lehetővé teszi a programot író számára a videokártya erőforrásait felhasználva a párhuzamos számítást.

A programot sajnos futtatni nem tudom, ugyanis nem áll rendelkezésemre CUDA kártya, így számomra marad az "elméleti áttekentése". A programkód .cu kiterjesztésű de javarészt C++ szintaktika lelhető fel benne. Kezdem az elemzést:

```
#include <png++/image.hpp>
#include <png++/rgb_pixel.hpp>

#include <sys/times.h>
#include <iostream>

#define MERET 600
#define ITER_HAT 32000
```

Programunkba inkludáljuk a megfelelő könyvtárakat. Az eddigiek től különböző a sys/times.h header fájl ami lehetővé teszi majd számunkra a processzoridővel való munkát. Ahogy azt már az előző Mandelbrotos programok során látható volt, most is nevesített konstansokat használunk a méret és az iterációs határ méretére vonatkozóan a programtörzsben való jobb átláthatóság és kényelem érdekében.

```
__device__ int mandel (int k, int j)
{
 // Végigzongorázza a CUDA a szélesség x magasság rácsot:
 // most eppen a j. sor k. oszlopban vagyunk

 // számítás adatai

 float a = -2.0, b = .7, c = -1.35, d = 1.35;
 int szelesseg = MERET, magassag = MERET, iteraciosHatar = ←
 ITER_HAT;

 // a számítás

 float dx = (b - a) / szelesseg;
 float dy = (d - c) / magassag;
 float reC, imC, rez, imZ, ujrez, ujimZ;

 // Hány iterációt csináltunk?

 int iteracio = 0;

 // c = (reC, imC) a rács csomópontjainak
 // megfelelő komplex szám

 reC = a + k * dx;
 imC = d - j * dy;
 // z_0 = 0 = (rez, imZ)
 rez = 0.0;
 imZ = 0.0;
 iteracio = 0;

 // z_{n+1} = z_n * z_n + c iterációk
 // számítása, amíg |z_n| < 2 vagy még
 // nem értük el a 255 iterációt, ha
 // viszont elértek, akkor úgy vesszük,
 // hogy a kiinduláci c komplex számra
 // az iteráció konvergens, azaz a c a
 // Mandelbrot halmaz eleme

 while (rez * rez + imZ * imZ < 4 && iteracio < iteraciosHatar ←
)
 {
 // z_{n+1} = z_n * z_n + c
 ujrez = rez * rez - imZ * imZ + reC;
 ujimZ = 2 * rez * imZ + imC;
```

```
 reZ = ujreZ;
 imZ = ujimZ;

 ++iteracio;

}

return iteracio;
}
```

A `__device__` jelzés a függvény neve előtt egy azt hivatott megadni, hogy a függvényt az eszköz vagyis a kártya hajtsa végre. Ahogy az a megjegyzésekben is leolvasható, létrehozásra kerülnek a számítás adatait tartalmazó változók mint a szélességet, magasságot, iterációs határt, lépésközt és iterációk számát tartalmazó változók. Hogy mi történik a függvényen belül? Ugyanaz, mint az előző feladatok esetében. A feladatcsokor második feladatához képest most egy kicsit időigényesebb a dolgunk ugyanis ismét nem áll rendelkezésre az `std::complex` osztályunk, ezért külön változóban kell tárolnunk `z` és `c` komplex számok valós és képzetes részeit.

Végigmegyünk a szélesség `x` magasság rácson. Egy while ciklusban megvizsgáljuk, hogy `z_n` abszolútér téke kisebb-e mint 4, illetve hogy elérteük-e az iterációs határt. Ha a ciklusfejben található feltétel teljesül, akkor módosítjuk a `Z` változó értékeit, úgy, hogy változó értéke így módosuljon:  $z_{n+1} = z_n^2 + c$ . `Z` változó új értékeit `Z` eredeti változóiiba másoljuk. Az `iteracio` számlálót növeljük. A függvény az iterációk számát adja vissza.

```
/*
__global__ void mandelkernel (int *kepadat)
{

 int j = blockIdx.x;
 int k = blockIdx.y;

 kepadat[j + k * MERET] = mandel (j, k);

}

*/
__global__ void mandelkernel (int *kepadat)
{

 int tj = threadIdx.x;
 int tk = threadIdx.y;

 int j = blockIdx.x * 10 + tj;
 int k = blockIdx.y * 10 + tk;

 kepadat[j + k * MERET] = mandel (j, k);

}
```

A `__global__` szócska a függvényünk előtt azt jelzi, hogy ezt a functiont is a videokártya fogja végrehajtani. Az előző esettől eltérően azonban itt a CPU hívja meg a függvényt, de mint ahogyan említettem,

a GPU hajta végre. Mivel ebben a feladatban 60x60 db blokkal dolgozunk, blokkonként pedig 100 darab szállal, így a `threadIdx.x` és `threadIdx.y` azt jelölik, melyik szálon fut az értékek kiszámítása, ezeket `tj` és `tk` változókban tároljuk. A következő pár sorban az előbb megkapott értékeket újra felhasználjuk mikor `j` és `k` változókat számoljuk ki. Felhasználjuk a `blockIdx.x` és `blockIdx.y` változókat is amik a blokkokat azonosítják. (`j` és `k` változó értéke: `blokk*10+szál`) A megkapott értékeket a `mandel` függvénynek paraméterként adjuk át.

```
void cudamandel (int kepadat [MERET] [MERET])
{
 int *device_kepadat;
 cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (←
 int));

 dim3 grid (MERET / 10, MERET / 10);
 dim3 tgrid (10, 10);
 mandelkernel <<< grid, tgrid >>> (device_kepadat);

 cudaMemcpy (kepadat, device_kepadat,
 MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost);
 cudaFree (device_kepadat);
}
```

A programunk utolsó függvénye. A függvénynek átadásra kerül egy tömb. Létrehozunk egy integer típusú mutatót majd a `cudaMalloc` függvénnyel memóriát foglalunk számára. Létrehozunk két 3 dimenziós vektort amiben a képen látható számértékeket tároljuk. A függvény végén a `cudaMemcpy` függvénnyel a `device_kepadat` értékét a `kepadat`ba másoljuk. A függvény 3. paramétere egyébként az átmásolandó bájtokat, míg az utolsó a másolás típusát adja meg. (esetünkben Device to Host) A `cudaFree` függvénnyel felszabadítjuk a lefoglalt memóriát.

```
int main (int argc, char *argv[])
{
 // Mérünk időt (PP 64)
 clock_t delta = clock ();
 // Mérünk időt (PP 66)
 struct tms tmsbuf1, tmsbuf2;
 times (&tmsbuf1);

 if (argc != 2)
 {
 std::cout << "Használat: ./mandelpngc fajlnev";
 return -1;
 }

 int kepadat [MERET] [MERET];

 cudamandel (kepadat);
```

```
png::image < png::rgb_pixel > kep (MERET, MERET);
```

A main függvényünk eleje. Ezen program futtatása során szeretnénk látni, hogy milyen gyorsan készül el a kívánt képünk, ezért egy változóban eltároljuk a `clock` függvény értékét, ami a program által elhasznált processzoridőt adja vissza. Most is ha helytelenül futtatná az illető a programot, figyelmeztetjük őt. Meghívásra kerül a `cudamandel` függvény és létrejön az üres kép állományunk.

```
for (int j = 0; j < MERET; ++j)
{
 //sor = j;
 for (int k = 0; k < MERET; ++k)
 {
 kep.set_pixel (k, j,
 png::rgb_pixel (255 -
 (255 * kepadat[j][k]) / ITER_HAT,
 255 -
 (255 * kepadat[j][k]) / ITER_HAT,
 255 -
 (255 * kepadat[j][k]) / ITER_HAT));
 }
}
kep.write (argv[1]);

std::cout << argv[1] << " mentve" << std::endl;

times (&tmsbuf2);
std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
+ tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;

delta = clock () - delta;
std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << std::endl;
```

Végigmegyünk a sorokon és oszlopokon és ha az adott képpontról megállapításra kerül, hogy a Mandelbrot-halmaz eleme, színezésre kerül. Végül tájékoztatjuk a felhasználót a kép mentésének sikereségéről és kiírjuk, mennyi időt is vett igénybe a kép elkészülése.

## 5.5. Mandelbrot nagító és utazó C++ nyelven

Építs GUI-t a Mandelbrot algoritmusra, lehessen egérrel nagytárolni egy területet, illetve egy pontot egérrel kiválasztva vizualizálja onnan a komplex iteráció bejárta  $z_n$  komplex számokat!

Megoldás videó:

Megoldás forrása: [https://sourceforge.net/p/udprog/code/ci/master/tree/source/kozepes/Qt/mandel\\_nagyito/](https://sourceforge.net/p/udprog/code/ci/master/tree/source/kozepes/Qt/mandel_nagyito/)

[https://gitlab.com/kincsa/bhax/tree/master/codes/mandelbrot\\_codes/mandel\\_nagyito](https://gitlab.com/kincsa/bhax/tree/master/codes/mandelbrot_codes/mandel_nagyito)

Tanulságok, tapasztalatok, magyarázat...

A feladat megoldásához a Qt-t fogjuk használni, ha nincs letöltve, le kell töltenünk. A Qt egy eszköztár, amivel könnyedén tudunk grafikus felhasználói felületű programokat létrehozni.(ezen feladat során természetesen mindezt C++ nyelven)

A program 5 fő egységből áll:

Vegyük sorra, mik az egyes egységek feladata:

### main.cpp

```
#include <QApplication>
#include "frakablak.h"

int main(int argc, char *argv[])
{
 QApplication a(argc, argv);

 FrakAblak w1;
 w1.show();
 return a.exec();
}
```

Inkludáljuk a Qt-vel való munkához szükséges könyvtárat, a QApplication-t illetve egy másik .h kiterjesztésű (ún. header fájlt), amelyet mi hoztunk létre Szerepe fontos lesz a későbbiekben. A QApplication segítségével példányosítunk, egy objektumot hozunk létre, meghívásra kerül a konstruktur.

### frakablak.h

```
#ifndef FRAKABLAK_H
#define FRAKABLAK_H

#include <QMainWindow>
#include <QImage>
#include <QPainter>
#include <QMouseEvent>
#include <QKeyEvent>
#include "frakszal.h"

class FrakSzal;

class FrakAblak : public QMainWindow
{
 Q_OBJECT

public:
 FrakAblak(double a = -2.0, double b = .7, double c = -1.35,
 double d = 1.35, int szelesseg = 600,
 int iteraciosHatar = 255, QWidget *parent = 0);
 ~FrakAblak();
 void vissza(int magassag, int * sor, int meret);
 void vissza(void);
 // A komplex sík vizsgált tartománya [a,b]x[c,d].
 double a, b, c, d;
```

```
// A komplex sík vizsgált tartományára feszített
// háló szélessége és magassága.
int szelesseg, magassag;
// Max. hány lépésig vizsgáljuk a z_{n+1} = z_n * z_n + c ←
// iterációt?
// (tk. most a nagyítási pontosság)
int iteraciosHatar;

protected:
void paintEvent(QPaintEvent*);
void mousePressEvent(QMouseEvent*);
void mouseMoveEvent(QMouseEvent*);
void mouseReleaseEvent(QMouseEvent*);
void keyPressEvent(QKeyEvent*);

private:
QImage* fraktal;
FrakSzal* mandelbrot;
bool szamitasFut;
// A nagyítandó kijelölt területet bal felső sarka.
int x, y;
// A nagyítandó kijelölt terület szélessége és magassága.
int mx, my;
};

#endif // FRAKABLAK_H
```

Az előbb már emlegetett header fájlunk, amely egyaránt tartalmaz public (nyilvános), protected (védett) és private (privát) hozzáférésű tagokat is. Azonban ami minket leginkább érdekel azok a protected functionök. Azok a függvények dolgozzák fel a billentyűzetlenyomásokat és egérmozgatásokat- és kattintásokat, vagyis az interaktivitás alapját adják. Ezek a függvények itt még csupán deklarálva vannak, a definiálásra később és máshol kerül sor.

### frakablak.cpp (részlet)

```
void FrakAblak::paintEvent(QPaintEvent*) {
QPainter qpainter(this);
qpainter.drawImage(0, 0, *fraktal);
if(!szamitasFut) {
 qpainter.setPen(QPen(Qt::white, 1));
 qpainter.drawRect(x, y, mx, my);

}
qpainter.end();
}

void FrakAblak::mousePressEvent(QMouseEvent* event) {

// A nagyítandó kijelölt területet bal felső sarka:
x = event->x();
y = event->y();
```

```
mx = 0;
my = 0;

update();
}

void FrakAblak::mouseMoveEvent (QMouseEvent* event) {

 // A nagyítandó kijelölt terület szélessége és magassága:
 mx = event->x() - x;
 my = mx; // négyzet alakú

 update();
}

void FrakAblak::mouseReleaseEvent (QMouseEvent* event) {

 if (szamitasFut)
 return;

 szamitasFut = true;

 double dx = (b-a)/szelesseg;
 double dy = (d-c)/magassag;

 double a = this->a+x*dx;
 double b = this->a+x*dx+mx*dx;
 double c = this->d-y*dy-my*dy;
 double d = this->d-y*dy;

 this->a = a;
 this->b = b;
 this->c = c;
 this->d = d;

 delete mandelbrot;
 mandelbrot = new FrakSzal(a, b, c, d, szelesseg, magassag, ←
 iteraciosHatar, this);
 mandelbrot->start();

 update();
}

void FrakAblak::keyPressEvent (QKeyEvent *event)
{

 if (szamitasFut)
 return;

 if (event->key() == Qt::Key_N)
```

```
 iteraciosHatar *= 2;
 szamitasFut = true;

 delete mandelbrot;
 mandelbrot = new FrakSzal(a, b, c, d, szelesseg, magassag, ←
 iteraciosHatar, this);
 mandelbrot->start();

}
```

Többek között az előző header fájlban deklarált függvények definiálása történik ebben a fájlból. Az egérmozgatást és kattintást, valamint a billentyűzetlenyomást figyelő függvények figyelhetők meg.

### frakszal.h

```
#ifndef FRAKSZAL_H
#define FRAKSZAL_H

#include <QThread>
#include <math.h>
#include "frakablak.h"

class FrakAblak;

class FrakSzal : public QThread
{
 Q_OBJECT

public:
 FrakSzal(double a, double b, double c, double d,
 int szelesseg, int magassag, int iteraciosHatar, FrakAblak *frakAblak);
 ~FrakSzal();
 void run();

protected:
 // A komplex sík vizsgált tartománya [a,b]x[c,d].
 double a, b, c, d;
 // A komplex sík vizsgált tartományára feszített
 // háló szélessége és magassága.
 int szelesseg, magassag;
 // Max. hány lépésig vizsgáljuk a z_{n+1} = z_n * z_n + c iterációt ←
 //
 // (tk. most a nagyítási pontosság)
 int iteraciosHatar;
 // Kinek számolok?
 FrakAblak* frakAblak;
 // Soronként küldöm is neki vissza a kiszámoltakat.
 int* egySor;
```

```
};

#endif // FRAKSZAL_H
```

Az ebben a header fájlból található osztály feladata nem más, minthogy deklarálja azokat a változókat, amiket a számolás és a rajzolás során alkalmazni fogunk.

### frakszal.cpp

```
#include "frakszal.h"

FrakSzal::FrakSzal(double a, double b, double c, double d,
 int szelesseg, int magassag, int ←
 iteraciosHatar, FrakAblak *frakAblak)
{
 this->a = a;
 this->b = b;
 this->c = c;
 this->d = d;
 this->szelesseg = szelesseg;
 this->iteraciosHatar = iteraciosHatar;
 this->frakAblak = frakAblak;
 this->magassag = magassag;

 egySor = new int[szelesseg];
}

FrakSzal::~FrakSzal()
{
 delete[] egySor;
}

void FrakSzal::run()
{
 // A [a,b]x[c,d] tartományon milyen sűrű a
 // megadott szélesség, magasság háló:
 double dx = (b-a)/szelesseg;
 double dy = (d-c)/magassag;
 double reC, imC, reZ, imZ, ujreZ, ujimZ;
 // Hány iterációt csináltunk?
 int iteracio = 0;
 // Végigzongorázzuk a szélesség x magasság hálót:
 for(int j=0; j<magassag; ++j) {
 //sor = j;
 for(int k=0; k<szelesseg; ++k) {
 // c = (reC, imC) a háló rácspontjainak
 // megfelelő komplex szám
 reC = a+k*dx;
 imC = d-j*dy;
 // z_0 = 0 = (reZ, imZ)
 reZ = 0;
```

```
imZ = 0;
iteracio = 0;
// z_{n+1} = z_n * z_n + c iterációk
// számítása, amíg |z_n| < 2 vagy még
// nem értük el a 255 iterációt, ha
// viszont elértek, akkor úgy vesszük,
// hogy a kiinduláci c komplex számra
// az iteráció konvergens, azaz a c a
// Mandelbrot halmaz eleme
while(reZ*reZ + imZ*imZ < 4 && iteracio < ↫
 iteraciosHatar) {
 // z_{n+1} = z_n * z_n + c

 ujreZ = reZ*reZ - imZ*imZ + reC;
 ujimZ = 2*reZ*imZ + imC;

 reZ = ujreZ;
 imZ = ujimZ;

 ++iteracio;

}
// ha a < 4 feltétel nem teljesült és a
// iteráció < iterációsHatár sérülésével lépett ki, ↫
// azaz
// feltessük a c-ről, hogy itt a z_{n+1} = z_n * z_n ↫
// + c
// sorozat konvergens, azaz iteráció = iterációsHatár
// ekkor az iteráció %= 256 egyenlő 255, mert az ↫
// esetleges
// nagyítások során az iteráció = valahány * 256 + ↫
255

iteracio %= 256;

// a színezést viszont már majd a FrakAblak osztályban ↫
lesz
egySor[k] = iteracio;
}
// Ábrázolásra átadjuk a kiszámolt sort a FrakAblak-nak.
frakAblak->vissza(j, egySor, szelesség);
}
frakAblak->vissza();

}
```

A Mandelbrot halmaz rajzolását végző osztály, szokásosan végighaladva a szélesség X magasság rácson minden pontra megnézve, eleme-e a Mandelbrot halmaznak.

Hogyan bírjuk működésre a Qt-t?

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/mandel_nagyito$ qmake - ↵
project
```

A `qmake` functiont használva létrehozásra kerül egy `.pro` kiterjesztésű fájl.

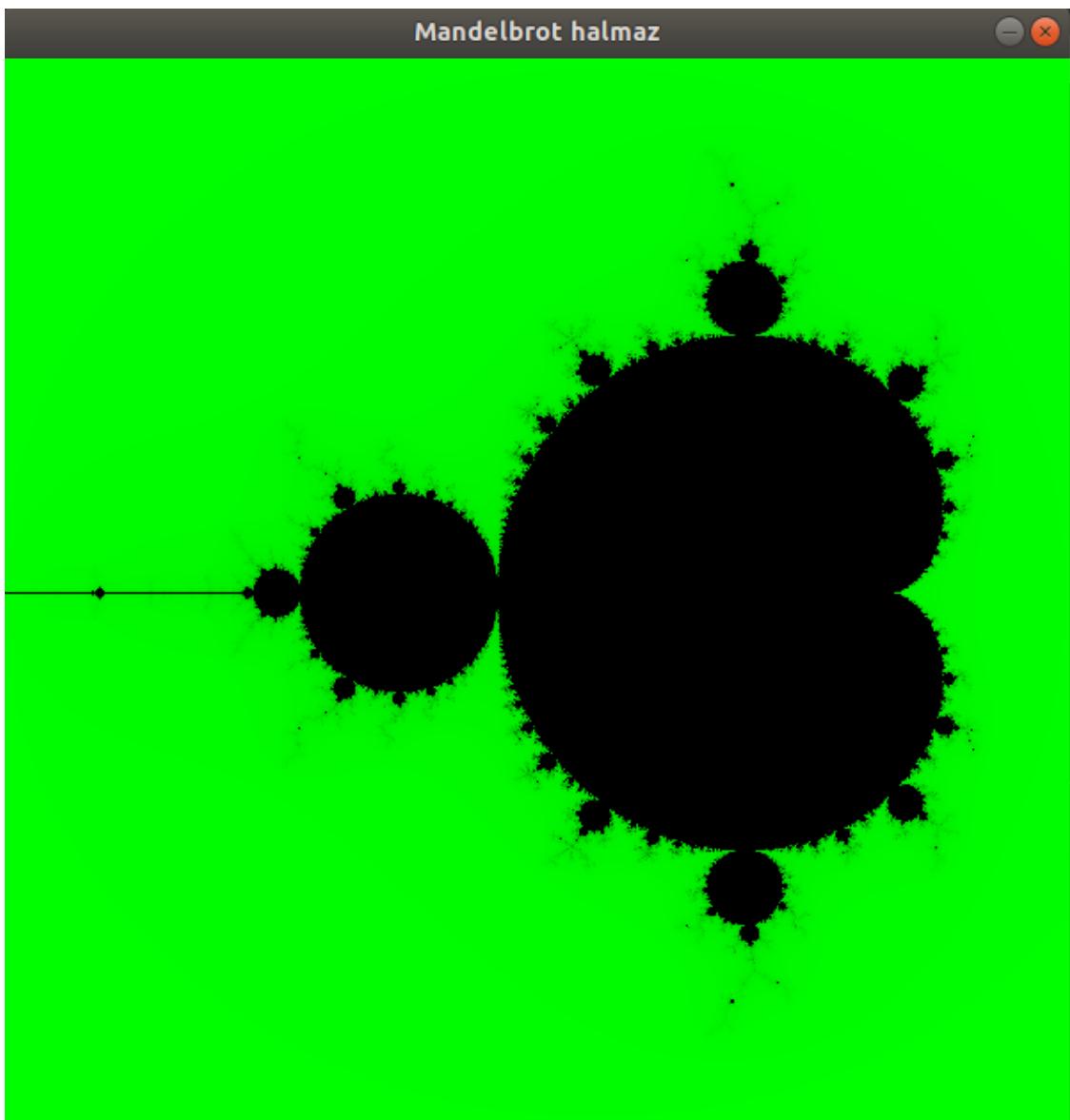
```
akos@akos-VirtualBox:~/Asztal/bhax/codes/mandel_nagyito$ make *.pro
```

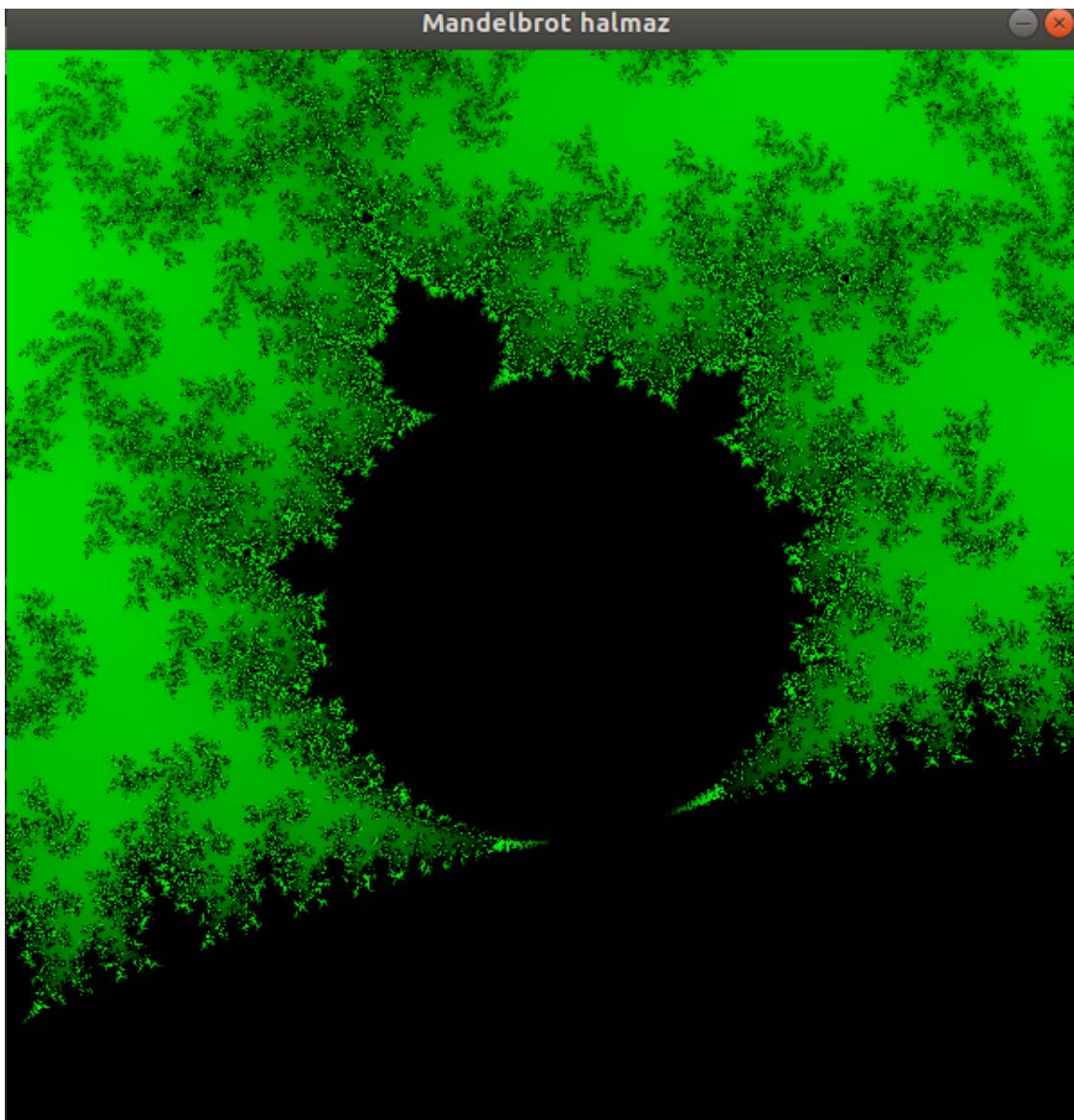
Az előző parancssal a `.pro` kiterjesztésű fájlok ból létrehozunk egy `Makefile`-t.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/mandel_nagyito$ make
```

A `make` parancs beütése után megkapjuk a futtatható állományt, futtassuk is:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/mandel_nagyito$./ ↵
mandelbrot_nagyito
```





## 5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás videó:

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html> - Mandelbrot halmaz nagyító programja

[https://gitlab.com/kincsa/bhax/tree/master/codes/mandelbrot\\_codes/mandel\\_nagyito\\_java](https://gitlab.com/kincsa/bhax/tree/master/codes/mandelbrot_codes/mandel_nagyito_java)

Feladatunk nem más, mint a már előzőleg C++ nyelven megírt Mandelbrot-halmaz kirajzoló programunk továbbfejlesztett változatát, a Mandelbrot nagyító és utazót megírni Java nyelven.

```
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {
 /** A nagyítandó kijelölt területet bal felső sarka. */

 private int x, y;
 /** A nagyítandó kijelölt terület szélessége és magassága. */
```

```
private int mx, my;
```

Létrehozzuk a MandelbrothalmazNagyító osztályunkat amely megnevezése után megtalálható az extends (kiterjeszt) kulcsszó. Ez azt jelenti, hogy a program a Mandelbrothalmaz.java program változóit és függvényeit tudja használni ami nagyon fontos lesz számunkra a későbbiekben. Ezen kívül a nagyítandó területet leíró adatokat tároló, privát hozzáférésű változókat hozunk létre. (ez a megjegyzések-ből is látható)

```
public MandelbrothalmazNagyító(double a, double b, double c, double ←
d,
int szélesség, int iterációsHatár) {

super(a, b, c, d, szélesség, iterációsHatár);
setTitle("A Mandelbrot halmaz nagyításai");

// Egér kattintó események feldolgozása:
addMouseListener(new java.awt.event.MouseAdapter() {
 // Egér kattintással jelöljük ki a nagyítandó területet
 // bal felső sarkát:
 public void mousePressed(java.awt.event.MouseEvent m) {
 // A nagyítandó kijelölt területet bal felső sarka:
 x = m.getX();
 y = m.getY();
 mx = 0;
 my = 0;
 repaint();
 }

 public void mouseReleased(java.awt.event.MouseEvent m) {
 double dx = (MandelbrothalmazNagyító.this.b
 - MandelbrothalmazNagyító.this.a)
 /MandelbrothalmazNagyító.this.szélesség;
 double dy = (MandelbrothalmazNagyító.this.d
 - MandelbrothalmazNagyító.this.c)
 /MandelbrothalmazNagyító.this.magasság;

 // Az új Mandelbrot nagyító objektum elkészítése:
 new MandelbrothalmazNagyító(MandelbrothalmazNagyító.this.a+←
x*dx,
 MandelbrothalmazNagyító.this.a+x*dx+mx*dx,
 MandelbrothalmazNagyító.this.d-y*dy-my*dy,
 MandelbrothalmazNagyító.this.d-y*dy,
 600,
 MandelbrothalmazNagyító.this.iterációsHatár);
 }
});

// Egér mozgás események feldolgozása:
addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
```

```
// Vonszolással jelöljük ki a négyzetet:
public void mouseDragged(java.awt.event.MouseEvent m) {
 // A nagyítandó kijelölt terület szélessége és magassága:
 mx = m.getX() - x;
 my = m.getY() - y;
 repaint();
}
});
}
```

Ebben a függvényben történik az eredeti osztályban található konstruktur hívása is. A program ablakának címét adunk. A különböző egérmóveleteket a megfelelő functionök kezelik, ilyen a kattintás és az egérkomb felengedése mely utóbbi esetében egy új MandelbrotHalmazNagyító objektum is létrehozásra kerül ugyanis ilyenkor tudni lehet, mekkora területet is akarunk nagyítani. Ezeken kívül természetesen nyomon kell követni az egér mozgását is, erre alkalmas az utolsó függvény. Az itt található repaint biztosítja a halmazunk újra kirajzoltatását a nagyítások után.

```
public void pillanatfelvétel() {
 // Az elmentendő kép elkészítése:

 java.awt.image.BufferedImage mentKép =
 new java.awt.image.BufferedImage(szélesség, magasság,
 java.awt.image.BufferedImage.TYPE_INT_RGB);
 java.awt.Graphics g = mentKép.getGraphics();
 g.drawImage(kép, 0, 0, this);
 g.setColor(java.awt.Color.BLUE);
 g.drawString("a=" + a, 10, 15);
 g.drawString("b=" + b, 10, 30);
 g.drawString("c=" + c, 10, 45);
 g.drawString("d=" + d, 10, 60);
 g.drawString("n=" + iterációsHatár, 10, 75);
 if(számításFut) {
 g.setColor(java.awt.Color.RED);
 g.drawLine(0, sor, getWidth(), sor);
 }
 g.setColor(java.awt.Color.GREEN);
 g.drawRect(x, y, mx, my);
 g.dispose();
 // A pillanatfelvétel képfájl nevének képzése:
 StringBuffer sb = new StringBuffer();
 sb = sb.delete(0, sb.length());
 sb.append("MandelbrotHalmazNagyitas_");
 sb.append(++pillanatfelvételszámláló);
 sb.append("_");
 // A fájl nevébe bele vesszük, hogy melyik tartományban
 // találtuk a halmazt:

 sb.append(a);
 sb.append("_");
```

```
sb.append(b);
sb.append("_");
sb.append(c);
sb.append("_");
sb.append(d);
sb.append(".png");
// png formátumú képet mentünk

try {
 javax.imageio.ImageIO.write(mentKép, "png",
 new java.io.File(sb.toString()));
} catch(java.io.IOException e) {
 e.printStackTrace();
}
}
```

Itt, a pillanatfelvétel eljárás az, amiről érdemes pár szót ejteni. Itt jön létre az üres képfájlunk és a megfelelő képpontok színezése is itt valósul meg.

```
public void paint(java.awt.Graphics g) {
 // A Mandelbrot halmaz kirajzolása
 g.drawImage(kép, 0, 0, this);
 // Ha éppen fut a számítás, akkor egy vörös
 // vonallal jelöljük, hogy melyik sorban tart:
 if(számításFut) {
 g.setColor(java.awt.Color.RED);
 g.drawLine(0, sor, getWidth(), sor);
 }
 // A jelző négyzet kirajzolása:
 g.setColor(java.awt.Color.GREEN);
 g.drawRect(x, y, mx, my);
}
```

Itt látható magát a Mandelbrot halmazt kirajzoló függvény. Szükséges egy függvény a nagyításkor használatos keret kirajzoltatására is, ez is látható fent.

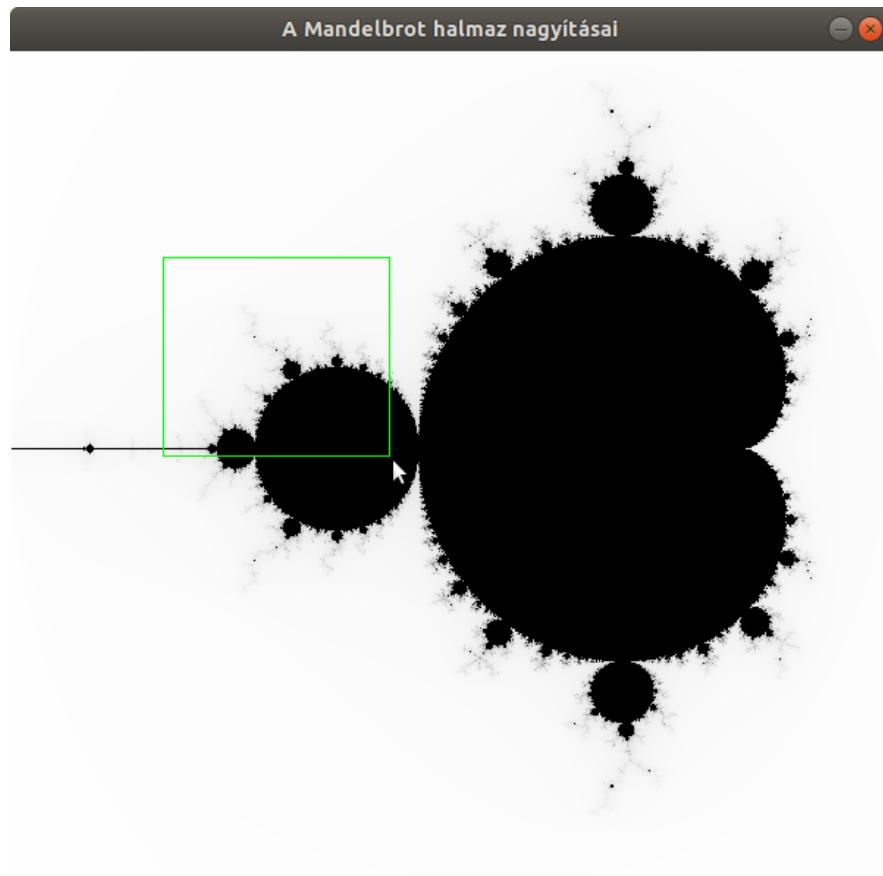
```
public static void main(String[] args) {
 // A kiinduló halmazt a komplex sík [-2.0, .7]x[-1.35, 1.35]
 // tartományában keressük egy 600x600-as hálóval és az
 // aktuális nagyítási pontossággal:
 new MandelbrotHalmazNagyító(-2.0, .7, -1.35, 1.35, 600, 255);
}
```

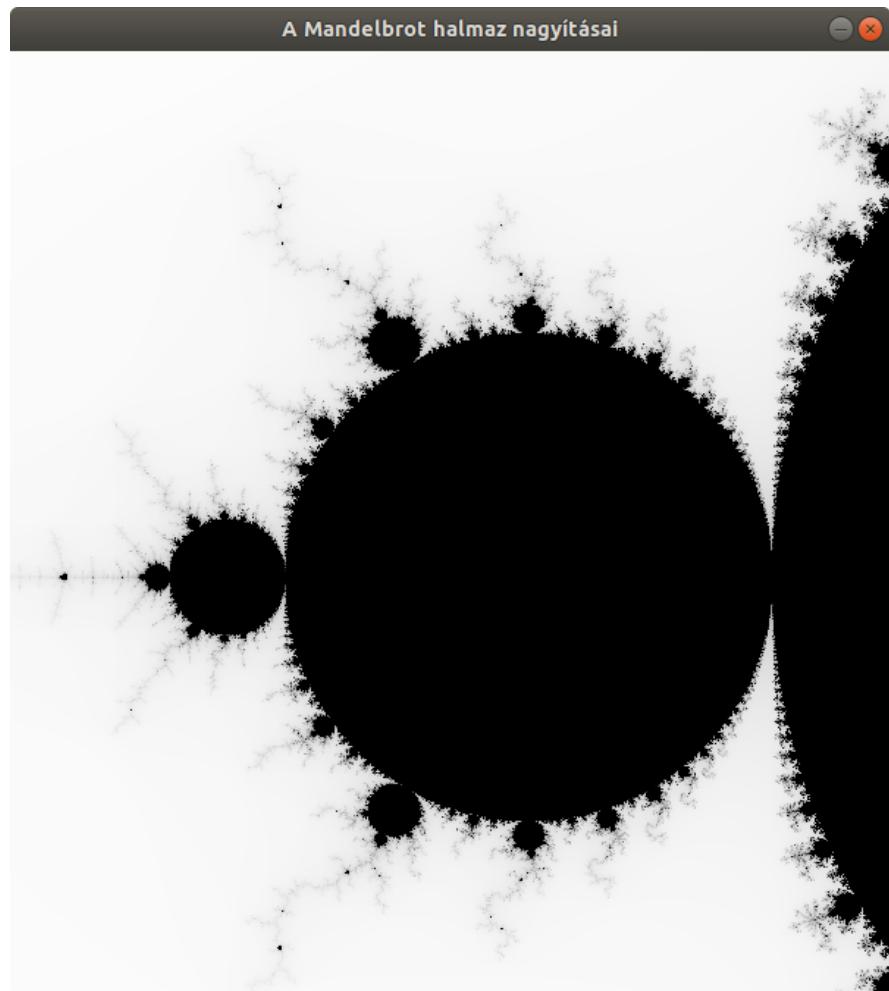
Programunk utolsó, említésre méltó részegységéhez értünk. A main-ben vagyunk ahol már csak a példányosításra kerül sor a megfelelő paraméterekkel.

Fordítjuk és futtatjuk:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/mandelbrot_codes$ javac ←
MandelbrotHalmazNagyító.java
```

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/mandelbrot_codes$ java ←
MandelbrotHalmazNagyító
```





## 6. fejezet

# Helló, Welch!

### 6.1. Első osztályom

Valósítsd meg C++-ban és Java-ban az módosított polártranszformációs algoritmust! A matek háttér teljesen irreleváns, csak annyiban érdekes, hogy az algoritmus egy számítása során két normálist számol ki, az egyiket elspájzolod és egy további logikai taggal az osztályban jelzed, hogy van vagy nincs eltéve kiszámolt szám.

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat... térd ki arra is, hogy a JDK forrásaiban a Sun programozói pont úgy csinálták meg ahogyan te is, azaz az OO nemhogy nem nehéz, hanem éppen természetes neked!

#### A megoldás Java-ban

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html>

[https://gitlab.com/kincsa/bhax/blob/master/codes/welch\\_codes/PolarGenerator.java](https://gitlab.com/kincsa/bhax/blob/master/codes/welch_codes/PolarGenerator.java)

```
public class PolárGenerátor {

 boolean nincsTárolt = true;
 double tárolt;

 public PolárGenerátor() {
 nincsTárolt = true;
 }
```

Programunk a fent látható módon indul. A PolárGenerátor publikus osztályban zajlik maga az algoritmus elkészítése. Itt kerül deklarálásra a boolean típusú nincsTárolt változó is, ami a feladat által kért logikai tag, mely azt jelzi majd nekünk, hogy "van vagy nincs eltéve kiszámolt szám és természetesen a példányosításhoz szükséges konstruktor is itt kap helyet, aminek neve megegyezik (meg kell egyeznie) az osztályéval."

```
public double következő() {

 if(nincsTárolt) {
```

```
 double u1, u2, v1, v2, w;
 do {
 u1 = Math.random();
 u2 = Math.random();

 v1 = 2*u1 - 1;
 v2 = 2*u2 - 1;

 w = v1*v1 + v2*v2;

 } while(w > 1);

 double r = Math.sqrt((-2*Math.log(w))/w);

 tárolt = r*v2;
 nincsTárolt = !nincsTárolt;

 return r*v1;

 }
 else {
 nincsTárolt = !nincsTárolt;
 return tárolt;
 }
}
}
```

Még mindig az eredeti osztályon belül járunk, itt deklarálásra kerül a double típust visszaadó, következő névre hallgató függvény. Ha a nincsTárolt változó igaz, egy do-while ciklus keretein belül, amíg w nagyobb, mint 1: u1 és u2 értékül kap egy véletlen számot a Math.random függvény álltal, v1 és v2 pedig az előző változóból kapja meg az értékét, ugyanez igaz lesz w-re is. Kiszámolásra kerül a r változó a kódcsipetben látható összefüggés alapján (a Math osztály függvényei segítségével) és a tárolt változó is értéket kap, ezzel együtt természetesen a nincsTárolt változó hamis lesz, ugyanis már lesz egy tárolt elemünk.

Ellenkező esetben, ha a függvény elején lévő feltételvizsgálat hamis eredményt ad vissza: lefut az else-ág, amelyben a már eltárolt elemet adja vissza a függvény.

```
public static void main(String[] args) {

 PolárGenerátor g = new PolárGenerátor();

 for(int i=0; i<10; ++i)
 System.out.println(g.következő());
}
```

A main-ben már csak a példányosítás és a függvény meghívása történik.

Fordítás és futtatás:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ javac PolarGenerator.java
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ java PolarGenerator
-0.6473071302488039
-0.9946094587381347
-0.6935017469437147
0.6494595217086153
-0.24033818349852076
1.4733359943036726
1.5794423521328425
-0.17844845343597263
1.7182182490085423
-0.8307400206864562
```

## A megoldás C++-ban

A megoldás forrása: [https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1\\_5.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_5.pdf)

[https://gitlab.com/kincsa/bhax/blob/master/codes/welch\\_codes/polar.cpp](https://gitlab.com/kincsa/bhax/blob/master/codes/welch_codes/polar.cpp)

```
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <iostream>

using namespace std;

class PolarGenerator
{
public:
 PolarGenerator ()
 {
 nincsTarolt = true;
 srand(time(NULL));
 }

 ~PolarGenerator()
 {

 }

 double kovetkezo ()
 {
 if (nincsTarolt)
 {
 double u1, u2, v1, v2, w;
 do
 {
 u1 = rand () / (RAND_MAX + 1.0);
 u2 = rand () / (RAND_MAX + 1.0);
 v1 = 2*u1 - 1;
 v2 = 2*u2 - 1;
 w = v1*v1 + v2*v2;
 }
 while (w > 1);

 double r = sqrt ((-2 * log (w)) / w);
 if (r <= 1)
 nincsTarolt = false;
 }
 return r;
 }
};
```

```
 tarolt = r * v2;
 nincsTarolt = !nincsTarolt;

 return r * v1;
 }
else
{
 nincsTarolt = !nincsTarolt;
 return tarolt;
}
}

bool nincsTarolt=true;
double tarolt;

};

int main (int argc, char **argv)
{
 PolarGenerator polargen;

 for (int i = 0; i < 10; ++i)
 std::cout << polargen.kovetkezo () << "\n";

 return 0;
}
```

A PolarGenerator osztályon belül létrehozásra kerül a konstruktor és a destruktur egyaránt. Előző osztályból való objektum létrehozásához szükséges még utóbbi annak a törléséhez. Itt foglal helyet a kovetkezo függvény is, illetve a nincsTarolt és tarolt változóink is. Igazi különbség talán csak a véletlen számok sorsolásánál figyelhető meg a Java verzióhoz képest. A RAND\_MAX konstans a rand függvény által sorsolható véletlen számok felső határát rögzíti.

Az algoritmus itt is ugyanaz, mint az előbb. Amint azt az ábra mutatja, a C++-os megoldás nagyon könnyedén megírható a Java kódunkból. Természetesen ez igaz fordítva is, azonban mivel mi először a Java verzióval rendelkeztünk, így egyértelmű, hogy az kerül átírásra C++-ra.

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ g++ polar.cpp -o polar
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$./polar
1.7122
-0.12115
0.370601
-1.018
0.162116
-0.661629
1.20271
-0.123688
-0.20081
-1.57585
```

## 6.2. LZW

Valósítsd meg C-ben az LZW algoritmus fa-építését!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/codes/welch\\_codes/binfa\\_z.c](https://gitlab.com/kincsa/bhax/blob/master/codes/welch_codes/binfa_z.c)

Tutor: Heinrich László -<https://gitlab.com/heinrichlaszlo>

A feladat során egy bináris fát építünk fel az előbb említett **LZW** eljárással. Az LZW egy tömörítési eljárás melyet az informatika számos szegmensében előszeretettel alkalmaznak. A feladatunk egy binfát felépíteni ezzel az algoritmussal amely adatszerkezettel már találkoztunk tanulmányaink során is az Adatszerkezetek és algoritmusok tárgy keretein belül. A bináris fa (binfa) egy olyan speciális fa, melyben minden csomó-pontnak legfeljebb két rátörlője van.

Hogyan működik maga a program? A bemenetként kapott bináris állomány bitjein egytől-egyig végigmegy és az algoritmus alapján felépít egy bináris fát (binfát) valamint információkat szolgáltat vele kapcsolatban, mint a mélysége, átlaga, szórása. De mi az algoritmus ami alapján működését végzi?

Két lehetőség állhat elő. Ha az olvasott bit 0 és ha a bit 1.

Ezen esetekkel a feladat taglálása során találkozni fogunk.

Lássuk, hogyan épül fel magát az algoritmust megvalósító program!

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

typedef struct binfa
{
 int ertek;
 struct binfa *bal nulla;
 struct binfa *jobb egy;
} BINFA, *BINFA_PTR;
```

Inkludáljuk a szükséges függvénykönyvtárakat és létrehozzuk a `binfa` struktúrát, aminek tagjai: maga az érték, a `bal nulla` és `jobb egy` mutatók, amik a jobb és bal oldali gyermekekre fognak mutatni. Erre a struktúrára mostantól `BINFA` néven hivatkozhatunk.

```
BINFA_PTR uj_elem ()
{
 BINFA_PTR p;

 if ((p = (BINFA_PTR) malloc (sizeof (BINFA))) == NULL)
 {
 perror ("memoria");
 exit (EXIT_FAILURE);
 }
 return p;
}

extern void kiir (BINFA_PTR elem);
extern void ratlag (BINFA_PTR elem);
extern void rszoras (BINFA_PTR elem);
extern void szabadit (BINFA_PTR elem);
```

Létrehozásra kerül az `uj_elem` függvény, ami hibával tér vissza, hogyha nem sikerül egy BINFA számára elegendő memóriát lefoglalni. Ebben a pár sorban történik még a későbbiekben használt függvények deklarálása is mondhatjuk azt is, hogy ezek a prototípusok. Ezzel csak a fordítónak jelezzük, hogy lesznek ilyen függvényeink valahol a programban.

```
int main (int argc, char **argv)
{
 char b;

 BINFA_PTR gyoker = uj_elem ();
 gyoker->ertek = '/';
 gyoker->bal nulla = gyoker->jobb_egy = NULL;
 BINFA_PTR fa = gyoker;

 while (read (0, (void *) &b, 1))
 {
 if (b == '0')
 {
 if (fa->bal nulla == NULL)
 {
 fa->bal nulla = uj_elem ();
 fa->bal nulla->ertek = 0;
 fa->bal nulla->bal nulla = fa->bal nulla->jobb_egy = NULL;
 fa = gyoker;
 }
 else
 {
 fa = fa->bal nulla;
 }
 }
 else
 {
 if (fa->jobb_egy == NULL)
 {
 fa->jobb_egy = uj_elem ();
 fa->jobb_egy->ertek = 1;
 fa->jobb_egy->bal nulla = fa->jobb_egy->jobb_egy = NULL;
 fa = gyoker;
 }
 else
 {
 fa = fa->jobb_egy;
 }
 }
 }

 printf ("\n");
 kiir (gyoker);
```

Megérkeztünk a main-be. Rendre a b azonosítójú, char típusú változóba kerülnek a beolvasott bájtok. Létrehozásra kerül a gyoker mint új elem és értéket adunk neki, a / jel lesz az, a bal és jobb oldali gyermekekének NULL értéket adunk.

Elérkeztünk a fa felépítésének alapjául szolgáló feltételvizsgálatokhoz. Ha 0-t olvasunk és a csomópontnak nincsen még 0-s gyermeke, akkor az uj\_elelem függvényt meghívva létrehozunk egyet és értékül 0-t kap. Az így újonnan létrejött csomópont bal- és jobb oldali gyermekeinek NULL mutató értéket adunk és visszaugrunk a gyökérre. Ha a beolvasáskor már volt 0-s gyermeke a csomópontnak, a mutatót arra a 0-s gyermekekre állítjuk.

Az 1-es érték vizsgálata is ugyanígy működik. Ha 1-et olvasunk és a csomópontnak nincsen még 1-es gyermeke, akkor az uj\_elelem függvényt meghívva létrehozunk egyet és értékül 1-t kap. Az így újonnan létrejött csomópont bal- és jobb oldali gyermekeinek NULL mutató értéket adunk és visszaugrunk a gyökérre. Ha a beolvasáskor már volt 1-s gyermeke a csomópontnak, a mutatót arra a 1-s gyermekekre állítjuk.

```
extern int max_melyseg, atlagosszeg, melyseg, atlagdb;
extern double szorasosszeg, atlag;

printf ("melyseg=%d\n", max_melyseg-1);

/* Átlagos ághossz kiszámítása */
atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
ratlag (gyoker);
// atlag = atlagosszeg / atlagdb;

atlag = ((double)atlagosszeg) / atlagdb;

atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
szorasosszeg = 0.0;

rszoras (gyoker);

double szoras = 0.0;

if (atlagdb - 1 > 0)
 szoras = sqrt(szorasosszeg / (atlagdb - 1));
else
 szoras = sqrt (szorasosszeg);

printf ("atlag=%f\nszoras=%f\n", atlag, szoras);

szabadit (gyoker);
}
```

A fa szórása és átlaga kerül kiszámolásra, amely aztán kiiratásra is kerül. Ebben a blokkban igaz, eddig még ismeretlen függvényeket hívtunk meg, ezek definiálására a main-en kívül kerül sor, a program maradék pár sornyi kódjában. A main függvényünk itt ér véget.

```
int atlagosszeg = 0, melyseg = 0, atlagdb = 0;

void ratlag (BINFA_PTR fa)
{
 if (fa != NULL)
 {
 ++melyseg;
 ratlag (fa->jobb_egy);
 ratlag (fa->bal nulla);
 --melyseg;

 if (fa->jobb_egy == NULL && fa->bal nulla == NULL)
 {

 ++atlagdb;
 atlagosszeg += melyseg;
 }
 }
}

double szorasosszeg = 0.0, atlag = 0.0;

void rszoras (BINFA_PTR fa)
{
 if (fa != NULL)
 {
 ++melyseg;
 rszoras (fa->jobb_egy);
 rszoras (fa->bal nulla);
 --melyseg;

 if (fa->jobb_egy == NULL && fa->bal nulla == NULL)
 {
 ++atlagdb;
 szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));
 }
 }
}
```

A program első pár sorában deklarált függvények itt kerülnek definiálásra melyekkel komplexebb képet kapunk az előállított bináris fánkról. Az `ratlag` és az `rszoras` függvények működése igen hasonló. Kezdjük az `ratlag` függvénnnyel. Ha a paraméterként kapott mutatója nem `NULL`, a mélységet növeli egyel és meghívásra kerül a függvény a jobb -és bal oldali gyermekekre, a mélységet csökkenti. Ha a bal és jobb oldali gyermek is `NULL` értékre mutat akkor az `atlagdb` változót növeljük és az `atlagosszeg` értékét megváltoztatjuk.

Ugyanez figyelhető meg az `rszoras` függvényben is csak értelemszerűen más változónevekkel. Az algoritmus lényegében ugyanaz.

```
int max_melyseg = 0;

void kiir (BINFA_PTR elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 if (melyseg > max_melyseg)
 max_melyseg = melyseg;
 kiir (elem->jobb_egy);

 for (int i = 0; i < melyseg; ++i)
 printf ("---");
 printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : ←
 elem->ertek,
 melyseg-1);
 kiir (elem->bal nulla);
 --melyseg;
 }
}

void szabadit (BINFA_PTR elem)
{
 if (elem != NULL)
 {
 szabadit (elem->jobb_egy);
 szabadit (elem->bal nulla);
 free (elem);
 }
}
```

A maximum mélység változó deklarálásra kerül. A `kiir` függvény a következőképp működik: ha az adott `elem` nem `NULL`, a `melyseg` változó növelésre kerül. Ha ez a változó nagyobb, mint a fa eddigi maximum mélysége, beállítjuk őt maximum mélységnek. Kiiratásra kerül az elem jobb oldali gyermeke, gyökérelem és a bal oldali gyermeke.

Az utolsó, `szabadit` névre hallgató függvény szerepe, hogy a lefoglalt memóriaterület felszabaduljon. Ha a vizsgált `elem` nem `NULL`, rekurzív módon meghívódik a függvény először a bal- majd a jobb oldali gyermekére, a `free` pedig a `malloc` által lefoglalt memóriát szabadítja fel. Így működik az LZW binfa-építő program.

Fordítás és futtatás után:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ gcc binfa_z.c -o binfa -lm
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$./binfa <stuff.txt > outputtext.txt
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ cat outputtext.txt
-----1(2)
-----0(3)
----1(1)
-----1(3)
-----0(2)
-----0(3)
---/(0)
----1(2)
----0(1)
----0(2)
melyseg=3
altag=2.600000
szoras=0.547723
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$
```

## 6.3. Fabejárás

Járd be az előző (inorder bejárású) fát pre- és posztorder is!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/codes/welch\\_codes/binfa\\_z\\_pre.c](https://gitlab.com/kincsa/bhax/blob/master/codes/welch_codes/binfa_z_pre.c) és [https://gitlab.com/kincsa/bhax/blob/master/codes/welch\\_codes/binfa\\_z\\_post.c](https://gitlab.com/kincsa/bhax/blob/master/codes/welch_codes/binfa_z_post.c)

A különböző fabejárási módokról is volt szó a már előzőleg is említett Adatszerkezetek és algoritmusok tárgyunk pár héttel ezelőtti előadásán. Ezek a módok a feldolgozandó Inorder bejárás esetén a sorrend: bal oldali részfa (inorder módon) --> gyökérelem --> jobb oldali részfa (inorder módon).

Preorder bejárás esetén a sorrend: gyökérelem --> bal oldali részfa (preorder módon) --> jobb oldali részfa (preorder módon).

Postorder bejárás esetén a sorrend: bal oldali részfa (postorder módon) --> jobb oldali részfa (postorder módon) --> gyökérelem.

A feladat az utolsó 2 bejárási mód használatát kéri. Nézzük meg, hogyan is néznek ki a különböző bejárási módok!

**Preorder:**

```
void kiir (BINFA_PTR elem) //preorder bejarasi mod
{
 if (elem != NULL)
 {
 ++melyseg;

 if (melyseg > max_melyseg)
 max_melyseg = melyseg;

 for (int i = 0; i < melyseg; ++i)
 printf ("---");
 printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek, melyseg);
```

```
 kiir (elem->bal_nulla);

 kiir (elem->jobb_egy);
 --melyseg;
}
}
```

## Postorder:

```
void kiir (BINFA_PTR elem) //postorder bejárasi mod
{
 if (elem != NULL)
 {
 ++melyseg;

 if (melyseg > max_melyseg)
 max_melyseg = melyseg;

 kiir (elem->bal_nulla);

 kiir (elem->jobb_egy);

 for (int i = 0; i < melyseg; ++i)
 printf ("---");
 printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek, melyseg);

 --melyseg;
 }
}
```

Ahogy látható, a bejárás egy feltételvizsgálattal indul, ahol azt vizsgáljuk, van-e mit egyáltalán bejárni, csak ezután növeljük a mélységet. Továbbá az is észrevehető, ahhoz, hogy a bejárást megváltoztassuk, csak a `kiir` eljáráshoz kellett hozzányúlnunk.

Fordítjuk és futtatjuk mind a két programot:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ gcc binfa_z_pre.c -o pre -lm
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$./pre <be.txt >pre_out.txt
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ cat pre_out.txt

---/(1)
-----0(2)
-----0(3)
-----1(3)
-----1(2)
-----1(3)
-----0(4)
melyseg=3
altag=2.333333
szoras=0.577350
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ █
```

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ gcc binfa_z_post.c -o post
-lm
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$./post <be.txt >post_out.txt
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ cat post_out.txt
-----0(3)
-----1(3)
----0(2)
-----0(4)
-----1(3)
----1(2)
---/(1)
melyseg=3
altag=2.333333
szoras=0.577350
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ █
```

## 6.4. Tag a gyökér

Az LZW algoritmust ültessd át egy C++ osztályba, legyen egy Tree és egy beágyazott Node osztálya. A gyökér csomópont legyen kompozícióban a fával!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/codes/welch\\_codes/z3a7.cpp](https://gitlab.com/kincsa/bhax/blob/master/codes/welch_codes/z3a7.cpp)

Az előzőek során vizsgált programkódunk (`binfa_z.c`) egyik továbbfejlesztett, C++ nyelvre átírt változat pontosan meg is felel a feladat által kért előírásoknak, nézzük is a kódot:

```
#include <iostream>
#include <cmath>
#include <fstream>

class LZWBinFa
{
public:
 LZWBinFa () :fa (&gyoker)
 {
 }
 ~LZWBinFa ()
 {
 szabadit (gyoker.egyesGyermek ());
 szabadit (gyoker.nullasGyermek ());
 }
 void operator<< (char b)
 {
 if (b == '0')
 {
 if (!fa->nullasGyermek ())
 {
 Csomopont *uj = new Csomopont ('0');
 fa->ujNullasGyermek (uj);
 fa = &gyoker;
 }
 }
 }
}
```

```
 else
 {
 fa = fa->nullasGyermek ();
 }
 }
else
{
 if (!fa->egyesGyermek ())
 {
 Csomopont *uj = new Csomopont ('1');
 fa->ujEgyesGyermek (uj);
 fa = &gyoker;
 }
 else
 {
 fa = fa->egyesGyermek ();
 }
}

void kiir (void)
{
 melyseg = 0;
 kiir (&gyoker, std::cout);
}

int getMelyseg (void);
double getAtlag (void);
double getSzoras (void);

friend std::ostream & operator<< (std::ostream & os, LZWBinFa & bf)
{
 bf.kiir (os);
 return os;
}
void kiir (std::ostream & os)
{
 melyseg = 0;
 kiir (&gyoker, os);
}
```

Include-oljuk a szükséges könyvtárakat. Az `iostream`-et azért, mert a standard inputról olvasunk és a standard outputra írunk majd, a `cmath`-ot azért, mert gyököt kellesz vonunk és az `fstream` könyvtárat azért, mert fájlok ból olvasunk és fájlokba írunk majd. Így indul a programunk az `LZWBinFa` osztályval. Ezen belül található a konstruktor, a destruktur (amiben a szabadit függvényt hívjuk meg több ízben) definiálása és itt történik az `<<` operátor túlterhelése is, mely során a már előzőleg használt binfa-építő algoritmus kerül megírásra, ami természetesen annak függvényében cselekszik, hogy mi a bemenő karakter (0/1), illetve hogy az adott csomópontnak van-e már 0-s/1-es gyermek. A pontos algoritmusrészletek az **LZW** feladatban megtalálhatók. A pár sorral fentebb túlterhelt `<<` operátor ismét túlterhelésre kerül de most abból

a célból, hogy egyszerűbben lehessen kimenetre iratni a binfa adatait.

Az első `kiir` függvény törzsében meghívásra kerül a később definiált, már összetettebb `kiir` függvény. Az előzőeken kívül a binfával kapcsolatos információkat szolgáltató függvények deklarálása történik még itt.

```
private:
class Csomopont
{
public:
 Csomopont (char b = '/') : betu (b), balNulla (0), jobbEgy (0)
 {
 };
 ~Csonopont ()
 {
 };

 Csonopont * nullasGyermek () const
 {
 return balNulla;
 }

 Csonopont * egyesGyermek () const
 {
 return jobbEgy;
 }

 void ujNullasGyermek (Csonopont * gy)
 {
 balNulla = gy;
 }

 void ujEgyesGyermek (Csonopont * gy)
 {
 jobbEgy = gy;
 }

 char getBetu () const
 {
 return betu;
 }

private:

 char betu;
 Csonopont * balNulla;
 Csonopont * jobbEgy;
 Csonopont (const Csonopont &);
 Csonopont & operator= (const Csonopont &);
};
```

```
Csomopont *fa;
int melyseg, atlagosszeg, atlagdb;
double szorasosszeg;
LZWBInFa (const LZWBInFa &);
LZWBInFa & operator= (const LZWBInFa &);

void kiir (Csomopont * elem, std::ostream & os)
{
 if (elem != NULL)
 {
 ++melyseg;
 kiir (elem->egyesGyermek (), os);
 for (int i = 0; i < melyseg; ++i)
 os << "----";
 os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
 kiir (elem->>nullasGyermek (), os);
 --melyseg;
 }
}

void szabadit (Csonopont * elem)
{
 if (elem != NULL)
 {
 szabadit (elem->egyesGyermek ());
 szabadit (elem->>nullasGyermek ());
 delete elem;
 }
}
protected:
Csonopont gyoker;
int maxMelyseg;
double atlag, szoras;

void rmelyseg (Csonopont * elem);
void ratlag (Csonopont * elem);
void rszoras (Csonopont * elem);
};
```

Megérkeztünk az LZWBInFa osztály privát részéhez, melyben a Csonopont osztály is található. Ezen osztály public részben megtalálható: konstruktor mellyel beállítjuk a gyökeret, destruktur, a binfa pilla-natnyi állapotait kezelő, illetve a bemenő karaktereket olvasó függvények. A privát részben foglalt helyet többek között a balNulla és jobbEgy mutatók. A "};" jelöléssel be is fejeződött a beágyazott csomó-pont osztály.

Ebben a kódcsipetben az LZWBInFa osztály privát tagjaival találkozunk még. Itt található meg a fa mutatóink. Szintén itt kerül deklarálásra a másoló konstruktorunk is amit nem szándékozunk használni a program jelenlegi verziójában, így az a következő sorban letiltásra kerül. A következő sorban a fa számadatait táro-

ló változók illetve a `kiir` és a `szabadit` függvények kerülnek deklarálásra melyekkel az LZW feladat során már találkoztunk, így nem részletezném működésüköt. A kódcsipet utolsó részletében a Fa osztály `protected` elemeit láthatjuk.

```
int
LZWBinFa::getMelyseg (void)
{
 melyseg = maxMelyseg = 0;
 rmelyseg (&gyoker);
 return maxMelyseg - 1;
}

double
LZWBinFa::getAtlag (void)
{
 melyseg = atlagosszeg = atlagdb = 0;
 ratlag (&gyoker);
 atlag = ((double) atlagosszeg) / atlagdb;
 return atlag;
}

double
LZWBinFa::getSzoras (void)
{
 atlag = getAtlag ();
 szorasosszeg = 0.0;
 melyseg = atlagdb = 0;

 rszoras (&gyoker);

 if (atlagdb - 1 > 0)
 szoras = std::sqrt (szorasosszeg / (atlagdb - 1));
 else
 szoras = std::sqrt (szorasosszeg);

 return szoras;
}

void
LZWBinFa::rmelyseg (Csomopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 if (melyseg > maxMelyseg)
 maxMelyseg = melyseg;
 rmelyseg (elem->egyesGyermek ());
 rmelyseg (elem->>nullasGyermek ());
 --melyseg;
 }
}
```

```
}

void
LZWBinFa::ratlag (Csomopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 ratlag (elem->egyesGyermek ());
 ratlag (elem->>nullasGyermek ());
 --melyseg;
 if (elem->egyesGyermek () == NULL && elem->>nullasGyermek () == ↔
 NULL)
 {
 ++atlagdb;
 atlagosszeg += melyseg;
 }
 }
}

void
LZWBinFa::rszoras (Csomopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 rszoras (elem->egyesGyermek ());
 rszoras (elem->>nullasGyermek ());
 --melyseg;
 if (elem->egyesGyermek () == NULL && elem->>nullasGyermek () == ↔
 NULL)
 {
 ++atlagdb;
 szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));
 }
 }
}

void
usage (void)
{
 std::cout << "Usage: lzwtree in_file -o out_file" << std::endl;
}
```

Most már az osztály(ok)on kívül vagyunk. Itt kerülnek definiálásra az eddig még csak deklarált függvényeink mely függvények az LZWBinFa osztályon belül kerültek deklarálásra. Említésre méltó még a usage function is, ami helytelen futtatási próbálkozás esetén tájékoztatja a felhasználót a megfelelő futtatási mód ról.

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

```
{
 if (argc != 4)
 {
 usage ();
 return -1;
 }

 char *inFile = *++argv;

 if (*((*++argv) + 1) != 'o')
 {
 usage ();
 return -2;
 }

 std::fstream beFile (inFile, std::ios_base::in);

 if (!beFile)
 {
 std::cout << inFile << " nem létezik..." << std::endl;
 usage ();
 return -3;
 }

 std::fstream kiFile (*++argv, std::ios_base::out);

 unsigned char b;
 LZWBinFa binFa;

 while (beFile.read ((char *) &b, sizeof (unsigned char)))
 if (b == 0xa)
 break;

 bool kommentben = false;

 while (beFile.read ((char *) &b, sizeof (unsigned char)))
 {

 if (b == 0x3e)
 {
 kommentben = true;
 continue;
 }

 if (b == 0xa)
 {
 kommentben = false;
 continue;
 }
 }
```

```
if (kommentben)
 continue;

if (b == 0x4e)
 continue;

for (int i = 0; i < 8; ++i)
{
 if (b & 0x80)
 binFa << '1';
 else
 binFa << '0';
 b <<= 1;
}

kiFile << binFa;

kiFile << "depth = " << binFa.getMelyseg () << std::endl;
kiFile << "mean = " << binFa.getAtlag () << std::endl;
kiFile << "var = " << binFa.getSzoras () << std::endl;

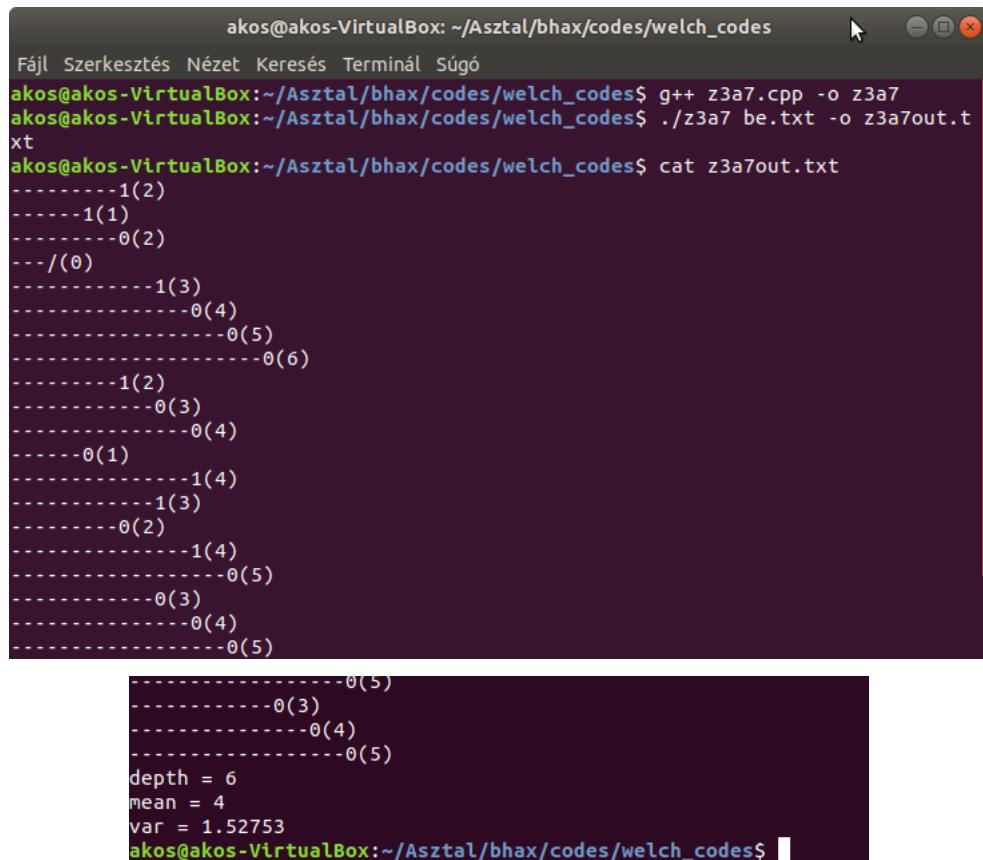
kiFile.close ();
beFile.close ();

return 0;
}
```

A main függvényben vagyunk. Hibaellenőrzés történik több ízben, ha futtatáskor valamelyik argumentum hiányzott, hibával tért vissza a program és az előbb már tárgyalt usage függvény hívódott meg. Ezt követi a bemeneti fájl beolvasása, feldolgozása, majd az elkészült fa kimeneti fájlba nyomása a fát meghatározó számadatokkal egyetemben.

Ahogy azt a kód is mutatja, megtalálható mind a Tree (esetünkben LZWBinFa) osztályunk, mind pedig a Node (nálunk Csomopont) osztály is beágyazottként. A feladat azt is leírja, hogy a gyökér legyen kompozícióban (függésben) a fával. Ez is látható a programban hiszen a gyoker csomópont (ahogy az egy komment formájában is olvasható) tagként van jelen az LZWBinfa osztályon belül, ezzel ez az igény is teljesül.

Fordítjuk és futtatók a usage függvény által leírtak szerint :



```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ g++ z3a7.cpp -o z3a7
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$./z3a7 be.txt -o z3a7out.txt
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ cat z3a7out.txt
-----1(2)
----1(1)
---0(2)
--/(0)
---1(3)
----0(4)
-----0(5)
-----0(6)
---1(2)
----0(3)
-----0(4)
---0(1)
-----1(4)
----1(3)
----0(2)
-----1(4)
-----0(5)
---0(3)
-----0(4)
-----0(5)

-----0(5)
-----0(3)
-----0(4)
-----0(5)
depth = 6
mean = 4
var = 1.52753
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$
```

## 6.5. Mutató a gyökér

Írd át az előző forrást, hogy a gyökér csomópont ne kompozícióban, csak aggregációban legyen a fával!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/codes/welch\\_codes/z3a7\\_new.cpp](https://gitlab.com/kincsa/bhax/blob/master/codes/welch_codes/z3a7_new.cpp)

A feladat címe le is lövi számunkra a poént, hogyan is oldjuk meg ezt az elsőre bonyolultnak hangzó feladatot. Tehát a dolgunk annyi, hogy a gyökér csomópontot mutatóra kell átírnunk a következő módon és helyen:

```
protected:
 Csomopont *gyoker;
 int maxMelyseg;
 double atlag, szoras;
```

Fordításkor azonban jelez a fordító, miszerint valami nincs rendben. Annak érdekében, hogy futó kódot kapjunk, bele kell nyúlnunk a kódunkba másolva is, például a konstruktörben:

```
class LZWBinFa
{
public:
 LZWBinFa ()
```

```
{
 gyoker = new Csomopont();
 fa = gyoker;
}
```

És a destrukturban is:

```
~LZWBInFa ()
{
 szabadit (gyoker->egyesGyermek ());
 szabadit (gyoker->>nullasGyermek ());
 delete gyoker;
}
```

Az előző kód (szabadit (gyoker.egyesGyermek ()) azért nem működött, mivel esetünkben amiért gyoker egy pointer, így neki nincsen nullasGyermek vagy egyesGyermek, viszont az általa mutatott objektumnak van, emiatt használjuk a -> (nyíl/arrow) operátort, ami a mutató által mutatott objektum megfelelő

Ahogyan az a képen is látható, a fordító még mindig elégedetlen:

The terminal window shows several error messages from the compiler (z3a7.cpp) regarding pointer operations and member function calls.

```
akos@akos-VirtualBox: ~/Asztal/bhax/codes/welch_codes
Fájl Szerkesztés Nézet Keresés Terminál Lapok Súgó
akos@akos-VirtualBox: ~/Asztal/bhax/cod... x akos@akos-VirtualBox: ~/Asztal/bhax/the... x
z3a7.cpp: In destructor 'LZWBInFa::~LZWBInFa()':
z3a7.cpp:96:26: error: request for member 'egyesGyermek' in '((LZWBInFa*)this)->LZWBInFa::gyoker', which is of pointer type 'LZWBInFa::Csonopont*' (maybe you meant to use '>-'?)
 szabadit (gyoker.egyesGyermek ());
 ^~~~~~
z3a7.cpp:97:26: error: request for member 'nullasGyermek' in '((LZWBInFa*)this)->LZWBInFa::gyoker', which is of pointer type 'LZWBInFa::Csonopont*' (maybe you meant to use '>-'?)
 szabadit (gyoker.nullasGyermek ());
 ^~~~~~
z3a7.cpp: In member function 'void LZWBInFa::operator<<(char)':
z3a7.cpp:130:23: error: cannot convert 'LZWBInFa::Csonopont**' to 'LZWBInFa::Csonopont*' in assignment
 fa = &gyoker;
 ^~~~
z3a7.cpp:145:23: error: cannot convert 'LZWBInFa::Csonopont**' to 'LZWBInFa::Csonopont*' in assignment
 fa = &gyoker;
 ^~~~
z3a7.cpp: In member function 'void LZWBInFa::kiir()':
z3a7.cpp:168:33: error: no matching function for call to 'LZWBInFa::kiir(LZWBInFa::Csonopont**, std::ostream&)'
 kiir (&gyoker, std::cout);
```

Szükséges még máshol is módosítanunk a programkódot, hogy a fordítás sikeres legyen!

Lényegében minden olyan helyen, ahol eddig az & (address of) operátort használtuk -a gyoker esetében - ott most már hagyogoljuk azt, hiszen a mutató/pointer már alapból címet tartalmaz, emiatt válik az & operátor használata okafogyottá.

Most már sikeres a fordítás és a futtatás.:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ g++ z3a7_new.cpp -o z3a7new
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$./z3a7new be.txt -o z3a7newout.txt
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ cat z3a7newout.txt
-----1(2)
----1(1)
---0(2)
--/(0)
---1(3)
---0(4)
---0(5)
---0(6)
----1(2)
----0(3)
----0(4)
---0(1)
----1(4)
----1(3)
---0(2)
----1(4)
----0(5)
----0(3)
----0(4)
-----0(3)
-----0(4)
-----0(5)
depth = 6
mean = 4
var = 1.52753
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$
```

A programkódban végzett összes változtatás megtekinthető a feladat elején található forrás linken.

## 6.6. Mozgató szemantika

Írj az előző programhoz mozgató konstruktort és értékkadást, a mozgató konstruktor legyen a mozgató értékkadásra alapozva!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/codes/welch\\_codes/z3a7\\_mozgato.cpp](https://gitlab.com/kincsa/bhax/blob/master/codes/welch_codes/z3a7_mozgato.cpp)

Tutor: Bacsik Mátyás -<https://gitlab.com/bacsikmatyas>

Ahogy a feladat leírása is mondja, az előző feladathoz írt programhoz kell elkészítenünk a mozgató konstruktort és értékkadást. Ebből adódik, hogy a programkód kisebb változtatáskótól eltekintve ugyanaz, mint az előbb. Itt is a két osztály, az LZWBInFa és a Csomopont osztályok adják a számolás alapját.

Nézzük meg, hogyan kell módosítanunk az előző kódunkat ahhoz, hogy a kért mozgató konstruktor és értékkadás megvalósuljon!

```
LZWBInFa (LZWBInFa&& masik) {
 gyoker=nullptr;
 *this= std::move(masik);
 std::cout<<"LZWBInFa mozgato ctor\n";

}

LZWBInFa& operator= (LZWBInFa&& masik) {
 std::swap(gyoker,masik.gyoker);
 //std::cout<<"LZWBInFa mozgato ertekekadas";
```

```
 return *this;
}
```

Elsőként a mozgató konstruktornal találkozunk. Ebben nullpointerre állítjuk annak a binfának a gyökerét amibe mozgatni szeretnénk a kiindulási binfánkat és igénybe vesszük a 3 sorral lentebb található mozgató értékadást azzal, hogy az egyenlőségjel operátort használjuk. (hiszen pont ennek az operátornak a túlterhéése történik lentebb) Meghívásra kerül a move függvény ami egy neki átadott lvalue-t alakít át rvalue-é, magyarul balból jobbértéket készít.

Amint említettem, itt található a mozgató értékadás is melyre a mozgató konstruktur megvalósítása épül. Túlterhelésre kerül az egyenlőségjel operátor amit a konstruktornal fel is tudunk használni. Az std függvénykönyvtár swap függvényét felhasználva felcserélésre kerül a kiindulási és az új binfa gyökere. A `*this` egy úgynevezett visszahivatkozási mutató, a sima `this`-szel ellentétben nem egy mutatót, hanem egy "klónt" ad vissza az objektumról.

```
kiFile << binFa;

kiFile << "depth = " << binFa.getMelyseg () << std::endl;
kiFile << "mean = " << binFa.getAtlag () << std::endl;
kiFile << "var = " << binFa.getSzoras () << std::endl;

LZWBinFa binFa3 =std::move(binFa);

kiFile << "depth = " << binFa3.getMelyseg () << std::endl;
kiFile << "mean = " << binFa3.getAtlag () << std::endl;
kiFile << "var = " << binFa3.getSzoras () << std::endl;

kiFile.close ();
beFile.close ();
```

Már a main-ben vagyunk. Megjelenik az std függvénykönyvtár move függvénye is. Fájlba írásra kerül az eredeti, majd az új Binfa példányunk minden adatával együtt. A main végén lezárásra kerül a be- és kimeneti fájl egyaránt.

Fordítjuk és futtatjuk:

```
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ g++ z3a7_mozgato.cpp -o z3a7_mozg
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$./z3a7_mozg befile.txt -o mozgatooutput.txt
LZWBinFa mozgato ctor
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$ cat mozgatooutput.txt
-----1(1)
-----0(2)
-----0(3)
---/(0)
-----1(3)
-----0(4)
-----0(5)
-----1(2)
-----0(3)
-----0(1)
-----1(4)
-----1(3)
-----0(2)
-----1(4)
-----0(5)
-----0(3)
-----1(6)
-----1(5)
-----1(3)
-----0(2)
-----1(4)
-----0(5)
-----0(3)
-----1(6)
-----1(5)
-----0(4)
depth = 6
mean = 4.333333
var = 1.21106
depth = 6
mean = 4.333333
var = 1.21106
akos@akos-VirtualBox:~/Asztal/bhax/codes/welch_codes$
```

Fordítás és futtatás után a fent látható eredményt kapjuk. Tehát a kódunk szintaktikailag és (futtatás után meggyőződhettünk róla) szemantikailag is helyes.

## 7. fejezet

# Helló, Conway!

### 7.1. Hangyszimulációk

Írj Qt C++-ban egy hangyszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

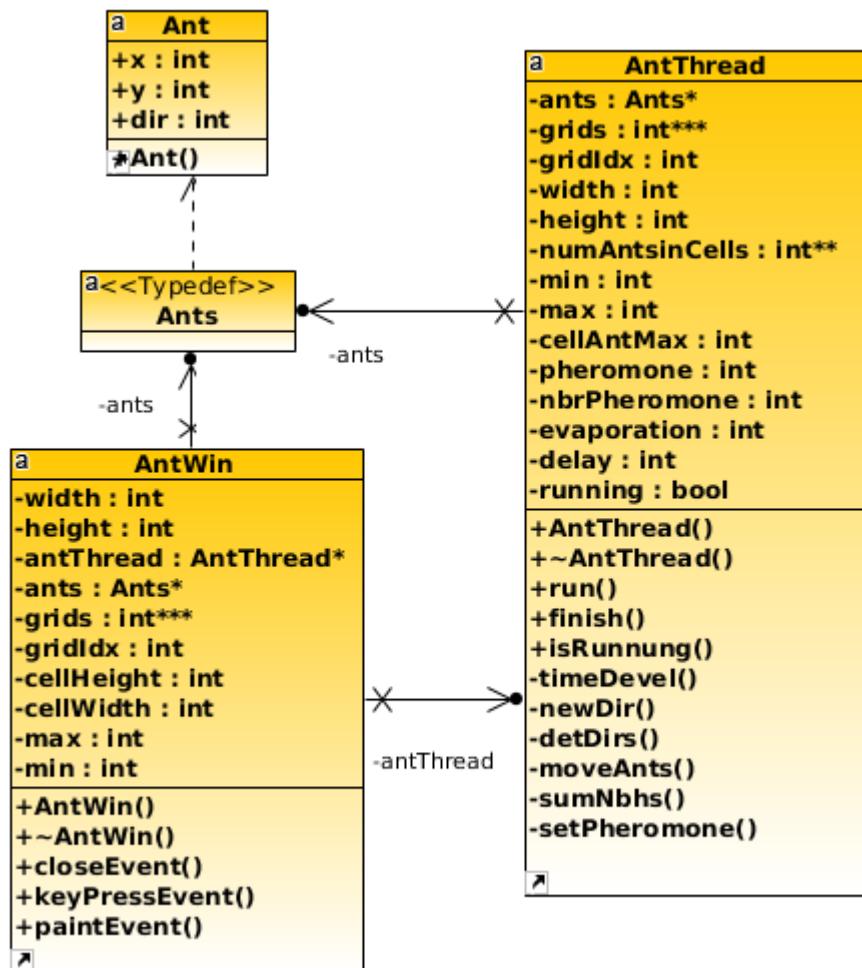
Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/conway\\_codes/myrmecologist/](https://gitlab.com/kincsa/bhax/blob/master/conway_codes/myrmecologist/)

Tanulságok, tapasztalatok, magyarázat...

Elsőre a hangyák és szimulálása egyáltalán nem hangzik izgalmasnak de ha egy kicsit jobban utánajárunk a dolgoknak már egész más lesz a helyzet. A hangyák (sok más állattal sőt az emberekkel is egyetemben) feromonokat bocsátanak ki. A feromon egy olyan kémiai anyag, amely hatására az egyes egyed az anyagot kibocsátó ellentétes nemű egyed iránt erős vonzalmat kezd érezni. A feromon tehát leginkább a párválasztásban játszik fontos szerepet de egyes fajok területmegjelölésre is használják.

A mi programunk is azt hivatott szimulálni, hogy a hangyák feromonpárolgása hogyan befolyásolja a többi hangya mozgását, elhelyezkedését. A cél az, hogy a dinamikus szimuláció észrevegyük, a hangyák azon szomszédjukhoz kerülnek közel, akinek a feromonszintje a legmagasabb.

A feladat leírásának kicsit ellentmondva nem a feladat végén, hanem már az elején nézzük az UML osztálydiagramot!



Összességében nem egy bonyolult ábra, azonban úgy gondolom, egy kis magyaráztra szorul. Az UML (Unified Modeling Language -magyarul: Egységes Modellező Nyelv) a programozásban igen elterjedt módja egy program osztályainak és az osztályok közötti kapcsolat ábrázolására.

Amit tudni kell az UML osztálydiagramról. A neve is elárulja, osztályok reprezentálása történik benne. A képen minden egyes sárga "mező" egy-egy osztály. minden osztálynak van(nak) tulajdonsága(i) és viselkedése(i). A tulajdonságok a változók míg a viselkedések a függvények. A diagramon a kettő közötti elválasztó vonal is be van jelölve minden osztály esetében, a vonal felett a változók míg alatta a függvények kapnak helyet.

Még egy fontos jelölés, ami mellett nem lehet elmenni: mind a változók, mind pedig a függvények előtt megjelenő + vagy - jel. A + jel azt jelenti, hogy az adott változó vagy függvény hozzáférhető más osztályokból is, publikus hozzáférésű vagyis public. A - jel esetében privát hozzáférésről beszélünk, ezekre az elemekre csak osztályon belül hivatkozhatunk. Ezek a private alkotóelemek.

Most, hogy az UML-lel kapcsolatos tudnivalók tisztázva lettek, vegyük sorra a programunkat felépítő fájlok! (osztályokat)

### A main.cpp

```

#include <QApplication>
#include <QDesktopWidget>
#include <QDebug>

```

```
#include <QDateTime>
#include <QCommandLineOption>
#include <QCommandLineParser>

#include "antwin.h"

/*
 *
 * ./myrmecologist -w 250 -m 150 -n 400 -t 10 -p 5 -f 80 -d 0 -a ←
 * 255 -i 3 -s 3 -c 22
 *
 */

int main (int argc, char *argv[])
{

 QApplication a (argc, argv);

 QCommandLineOption szeles_opt ({ "w", "szelesseg" }, "Oszlopok (←
 cellakban) szama.", "szelesseg", "200");
 QCommandLineOption magas_opt ({ "m", "magassag" }, "Sorok (←
 cellakban) szama.", "magassag", "150");
 QCommandLineOption hangyaszam_opt ({ "n", "hangyaszam" }, " ←
 Hangyak szama.", "hangyaszam", "100");
 QCommandLineOption sebesseg_opt ({ "t", "sebesseg" }, "2 lepes ←
 kozotti ido (millisec-ben).", "sebesseg", "100");
 QCommandLineOption parolgas_opt ({ "p", "parolgas" }, "A parolgas ←
 erteke.", "parolgas", "8");
 QCommandLineOption feromon_opt ({ "f", "feromon" }, "A hagyott ←
 nyom erteke.", "feromon", "11");
 QCommandLineOption szomszed_opt ({ "s", "szomszed" }, "A hagyott ←
 nyom erteke a szomszedokban.", "szomszed", "3");
 QCommandLineOption alapertek_opt ({ "d", "alapertek" }, "Indulo ←
 ertekek a cellakban.", "alapertek", "1");
 QCommandLineOption maxcella_opt ({ "a", "maxcella" }, "Cella max ←
 erteke.", "maxcella", "50");
 QCommandLineOption mincella_opt ({ "i", "mincella" }, "Cella min ←
 erteke.", "mincella", "2");
 QCommandLineOption cellamerete_opt ({ "c", "cellameret" }, "Hany ←
 hangya fer egy cellaba.", "cellameret", "4");
 QCommandLineParser parser;

 parser.addHelpOption();
 parser.addVersionOption();
 parser.addOption (szeles_opt);
 parser.addOption (magas_opt);
 parser.addOption (hangyaszam_opt);
 parser.addOption (sebesseg_opt);
 parser.addOption (parolgas_opt);
 parser.addOption (feromon_opt);
```

```
parser.addOption (szomszed_opt);
parser.addOption (alapertek_opt);
parser.addOption (maxcella_opt);
parser.addOption (mincella_opt);
parser.addOption (cellamerete_opt);

parser.process (a);

QString szeles = parser.value (szeles_opt);
QString magas = parser.value (magas_opt);
QString n = parser.value (hangyaszam_opt);
QString t = parser.value (sebesseg_opt);
QString parolgas = parser.value (parolgas_opt);
QString feromon = parser.value (feromon_opt);
QString szomszed = parser.value (szomszed_opt);
QString alapertek = parser.value (alapertek_opt);
QString maxcella = parser.value (maxcella_opt);
QString mincella = parser.value (mincella_opt);
QString cellameret = parser.value (cellamerete_opt);

qsrand (QDateTime::currentMSecsSinceEpoch());

AntWin w (szeles.toInt(), magas.toInt(), t.toInt(), n.toInt(), ←
 feromon.toInt(), szomszed.toInt(), parolgas.toInt(),
 alapertek.toInt(), mincella.toInt(), maxcella. ←
 toInt(),
 cellameret.toInt());

w.show();

return a.exec();
}
```

A Qt-s könyvtárakat és az `antwin.h`-t is include-oljuk. A main-ben létrehozásra kerül egy `QApplication` objektum vagyis példányosítottunk. A futtatáskor esetlegesen csatolható kapcsolókat természetesen parancssori argumentumként adhatjuk meg, ezeket ebben az osztályban dolgozzuk fel.

## A `ant.h`

```
#ifndef ANT_H
#define ANT_H

class Ant
{

public:
 int x;
 int y;
 int dir;

 Ant (int x, int y) : x(x), y(y) {
```

```
 dir = qrand() % 8;

}

};

typedef std::vector<Ant> Ants;

#endif
```

Az első, UML diagramunkon is látható osztályhoz értünk. A hangya irányát véletlenszerűre állítjuk a qrand függvényel. Ahogy az látható, a hangya mozgását leíró változók minden publikusak, a gyakorlatban ez azt jelenti, hogy ha egy másik fájlba include-oljuk a header fájlt, akkor ezek a változók ott is láthatók, sőt használhatók lesznek. Typedefet használva azt mondjuk, hogy az Ant-eket tartalmazó vectorra mostantól Ants néven hivatkozunk, ez az egyszerűséget

Az UML osztálydiagram következő elemezendő osztálya az AntThread. Azonban állományaink között két ilyen nevű is van, az egyik .h a másik pedig .cpp kiterjesztéssel. Utóbbit nem szükséges taglalni, szabványos C++ forráskód. A .h egy ún. header fájl, amit inkludálni tudunk programunkba olyan módon, ahogy azt a függvénykönyvtárakkal tettük eddig. A feladatok más feladataiban is megfigyelhető ez a megoldás.

### Az antthread.h

```
#ifndef ANTTHREAD_H
#define ANTTHREAD_H

#include <QThread>
#include "ant.h"

class AntThread : public QThread
{
 Q_OBJECT

public:
 AntThread(Ants * ants, int ***grids, int width, int height,
 int delay, int numAnts, int pheromone, int ←
 nbrPheromone,
 int evaporation, int min, int max, int cellAntMax);

 ~AntThread();

 void run();
 void finish()
 {
 running = false;
 }

 void pause()
 {
```

```
 paused = !paused;
 }

 bool isRunning()
 {
 return running;
 }

private:
 bool running {true};
 bool paused {false};
 Ants* ants;
 int** numAntsinCells;
 int min, max;
 int cellAntMax;
 int pheromone;
 int evaporation;
 int nbrPheromone;
 int ***grids;
 int width;
 int height;
 int gridIdx;
 int delay;

 void timeDevel();

 int newDir(int sor, int oszlop, int vsor, int voszlop);
 void detDirs(int irany, int& ifrom, int& ito, int& jfrom, int& ←
 jto);
 int moveAnts(int **grid, int row, int col, int& retrow, int& ←
 retcol, int);
 double sumNbhs(int **grid, int row, int col, int);
 void setPheromone(int **grid, int row, int col);

signals:
 void step (const int &);

};

#endif
```

Deklarálásra kerül az objektum példányosításakor meghívódó konstruktor és törlésekor meghívódó destruktur. Szintén itt jönnek deklarálásra az ablak képet megállító és a szimuláció befejezését eredményező függvények is. A privát tagok a grid adatait tárolják. (szélesség, magasság, fut-e, megvan-e állítva, stb...)

### Az antthread.cpp

```
#include "antthread.h"
#include <QDebug>
#include <cmath>
#include <QDateTime>
```

```
AntThread::AntThread (Ants* ants, int*** grids,
 int width, int height,
 int delay, int numAnts,
 int pheromone, int nbrPheromone,
 int evaporation,
 int min, int max, int cellAntMax)
{
 this->ants = ants;
 this->grids = grids;
 this->width = width;
 this->height = height;
 this->delay = delay;
 this->pheromone = pheromone;
 this->evaporation = evaporation;
 this->min = min;
 this->max = max;
 this->cellAntMax = cellAntMax;
 this->nbrPheromone = nbrPheromone;

 numAntsinCells = new int*[height];
 for (int i=0; i<height; ++i) {
 numAntsinCells[i] = new int [width];
 }

 for (int i=0; i<height; ++i)
 for (int j=0; j<width; ++j) {
 numAntsinCells[i][j] = 0;
 }

 qsrand (QDateTime::currentMSecsSinceEpoch());

 Ant h {0, 0};
 for (int i {0}; i<numAnts; ++i) {

 h.y = height/2 + qrand() % 40-20;
 h.x = width/2 + qrand() % 40-20;

 ++numAntsinCells[h.y][h.x];

 ants->push_back (h);
 }

 gridIdx = 0;
}

double AntThread::sumNbhs (int **grid, int row, int col, int ←
 dir)
{
```

```
double sum = 0.0;

int ifrom, ito;
int jfrom, jto;

detDirs (dir, ifrom, ito, jfrom, jto);

for (int i=ifrom; i<ito; ++i)
 for (int j=jfrom; j<jto; ++j)

 if (! ((i==0) && (j==0))) {
 int o = col + j;
 if (o < 0) {
 o = width-1;
 } else if (o >= width) {
 o = 0;
 }

 int s = row + i;
 if (s < 0) {
 s = height-1;
 } else if (s >= height) {
 s = 0;
 }

 sum += (grid[s][o]+1)*(grid[s][o]+1)*(grid[s][o ←
]+1);
 }

 }

return sum;
}

int AntThread::newDir (int sor, int oszlop, int vsor, int ←
 voszlop)
{
 if (vsor == 0 && sor == height -1) {
 if (voszlop < oszlop) {
 return 5;
 } else if (voszlop > oszlop) {
 return 3;
 } else {
 return 4;
 }
 } else if (vsor == height - 1 && sor == 0) {
 if (voszlop < oszlop) {
 return 7;
 } else if (voszlop > oszlop) {
 return 1;
 }
 }
}
```

```
 } else {
 return 0;
 }
 } else if (voszlop == 0 && oszlop == width - 1) {
 if (vsor < sor) {
 return 1;
 } else if (vsor > sor) {
 return 3;
 } else {
 return 2;
 }
 } else if (voszlop == width && oszlop == 0) {
 if (vsor < sor) {
 return 7;
 } else if (vsor > sor) {
 return 5;
 } else {
 return 6;
 }
 } else if (vsor < sor && voszlop < oszlop) {
 return 7;
 } else if (vsor < sor && voszlop == oszlop) {
 return 0;
 } else if (vsor < sor && voszlop > oszlop) {
 return 1;
 }

 else if (vsor > sor && voszlop < oszlop) {
 return 5;
 } else if (vsor > sor && voszlop == oszlop) {
 return 4;
 } else if (vsor > sor && voszlop > oszlop) {
 return 3;
 }

 else if (vsor == sor && voszlop < oszlop) {
 return 6;
 } else if (vsor == sor && voszlop > oszlop) {
 return 2;
 }

 else { // (vsor == sor && voszlop == oszlop)
 qDebug() << "ZAVAR AZ EROBEN az iranynal";
 return -1;
 }
}

void AntThread::detDirs (int dir, int& ifrom, int& ito, int& ←
```

```
jfrom, int& jto)
{

 switch (dir) {
 case 0:
 ifrom = -1;
 ito = 0;
 jfrom = -1;
 jto = 2;
 break;
 case 1:
 ifrom = -1;
 ito = 1;
 jfrom = 0;
 jto = 2;
 break;
 case 2:
 ifrom = -1;
 ito = 2;
 jfrom = 1;
 jto = 2;
 break;
 case 3:
 ifrom =
 0;
 ito = 2;
 jfrom = 0;
 jto = 2;
 break;
 case 4:
 ifrom = 1;
 ito = 2;
 jfrom = -1;
 jto = 2;
 break;
 case 5:
 ifrom = 0;
 ito = 2;
 jfrom = -1;
 jto = 1;
 break;
 case 6:
 ifrom = -1;
 ito = 2;
 jfrom = -1;
 jto = 0;
 break;
 case 7:
 ifrom = -1;
 ito = 1;
```

```
jfrom = -1;
jto = 1;
break;

}

}

int AntThread::moveAnts (int **racs,
 int sor, int oszlop,
 int& vsor, int& voszlop, int dir)
{

 int y = sor;
 int x = oszlop;

 int ifrom, ito;
 int jfrom, jto;

 detDirs (dir, ifrom, ito, jfrom, jto);

 double osszes = sumNbhs (racs, sor, oszlop, dir);
 double random = (double) (qrand() %1000000) / (double ←
) 1000000.0;
 double gvalseg = 0.0;

 for (int i=ifrom; i<ito; ++i)
 for (int j=jfrom; j<jto; ++j)
 if (! ((i==0) && (j==0)))
 {
 int o = oszlop + j;
 if (o < 0) {
 o = width-1;
 } else if (o >= width) {
 o = 0;
 }

 int s = sor + i;
 if (s < 0) {
 s = height-1;
 } else if (s >= height) {
 s = 0;
 }

 //double kedvezo = std::sqrt ((double) (racs[s][o ←
 //] +2)); // (racs[s][o]+2)*(racs[s][o]+2);
 //double kedvezo = (racs[s][o]+b)*(racs[s][o]+b ←
 //);
 //double kedvezo = (racs[s][o]+1);
 }
}
```

```

 double kedvezo = (racs[s][o]+1)*(racs[s][o]+1) ←
 *(racs[s][o]+1);

 double valseg = kedvezo/osszes;
 gvalseg += valseg;

 if (gvalseg >= random) {

 vsor = s;
 voszlop = o;

 return newDir (sor, oszlop, vsor, voszlop ←
);
 }

 }

qDebug() << "ZAVAR AZ EROBEN a lepesnel";
vsor = y;
voszlop = x;

return dir;
}

void AntThread::timeDevel()
{

int **racsElotte = grids[gridIdx];
int **racsUtana = grids[(gridIdx+1) %2];

for (int i=0; i<height; ++i)
 for (int j=0; j<width; ++j)
 {
 racsUtana[i][j] = racsElotte[i][j];

 if (racsUtana[i][j] - evaporation >= 0) {
 racsUtana[i][j] -= evaporation;
 } else {
 racsUtana[i][j] = 0;
 }
 }

for (Ant &h: *ants)
{
 int sor {-1}, oszlop {-1};
 int ujirany = moveAnts(racsElotte, h.y, h.x, sor, ←
 oszlop, h.dir);
}

```

```
 setPheromone (racsUtana, h.y, h.x);

 if (numAntsinCells[sor][oszlop] < cellAntMax) {

 --numAntsinCells[h.y][h.x];
 ++numAntsinCells[sor][oszlop];

 h.x = oszlop;
 h.y = sor;
 h.dir = ujirany;

 }

 }

 gridIdx = (gridIdx+1) %2;
}

}

void AntThread::setPheromone (int **racs,
 int sor, int oszlop)
{

 for (int i=-1; i<2; ++i)
 for (int j=-1; j<2; ++j)
 if (! ((i==0) && (j==0)))
 {
 int o = oszlop + j;
 {
 if (o < 0) {
 o = width-1;
 } else if (o >= width) {
 o = 0;
 }
 }
 int s = sor + i;
 {
 if (s < 0) {
 s = height-1;
 } else if (s >= height) {
 s = 0;
 }
 }

 if (racs[s][o] + nbrPheromone <= max) {
 racs[s][o] += nbrPheromone;
 } else {
 racs[s][o] = max;
 }
 }
 }

}
```

```
 }

 if (racs[sor][oszlop] + pheromone <= max) {
 racs[sor][oszlop] += pheromone;
 } else {
 racs[sor][oszlop] = max;
 }

 }

void AntThread::run()
{
 running = true;
 while (running) {

 QThread::msleep (delay);

 if (!paused) {
 timeDevel();
 }

 emit step (gridIdx);

 }
}

AntThread::~AntThread()
{
 for (int i=0; i<height; ++i) {
 delete [] numAntsinCells[i];
 }

 delete [] numAntsinCells;
}
```

Az előző header fájlban deklarált függvények definiálása történik meg. A teljesség igénye nélkül ilyen például a moveAnts ami magát a hangyamozgást előállítja vagy a timeDevel ami az "idő műlásért" felel.

### Az antwin.h

```
##ifndef ANTWIN_H
#define ANTWIN_H

#include <QMainWindow>
#include <QPainter>
#include <QString>
```

```
#include <QCloseEvent>
#include "antthread.h"
#include "ant.h"

class AntWin : public QMainWindow
{
 Q_OBJECT

public:
 AntWin(int width = 100, int height = 75,
 int delay = 120, int numAnts = 100,
 int pheromone = 10, int nbhPheromon = 3,
 int evaporation = 2, int cellDef = 1,
 int min = 2, int max = 50,
 int cellAntMax = 4, QWidget *parent = 0);

 AntThread* antThread;

 void closeEvent (QCloseEvent *event) {

 antThread->finish();
 antThread->wait();
 event->accept();
 }

 void keyPressEvent (QKeyEvent *event)
 {

 if (event->key() == Qt::Key_P) {
 antThread->pause();
 } else if (event->key() == Qt::Key_Q
 || event->key() == Qt::Key_Escape) {
 close();
 }
 }

 virtual ~AntWin();
 void paintEvent(QPaintEvent*);

private:

 int ***grids;
 int **grid;
 int gridIdx;
 int cellWidth;
 int cellHeight;
 int width;
 int height;
 int max;
```

```
int min;
Ants* ants;

public slots :
void step (const int &);

};

#endif
```

Ebben a header fájlban már a másik két header fájlt is inkludáljuk természetesen az alap Qt header-ökön kívül. Itt kerül deklarálásra a függvény, ami a billentyűlenyomásokat kezelik (keyPressEvent) vagy az a függvény, ami kezeli az esetet, mikor bezárásra kerül az ablak. A privát változók magát az ablakot leíró tulajdonságok.

### Az antwin.cpp

```
#include "antwin.h"
#include <QDebug>

AntWin::AntWin (int width, int height, int delay, int numAnts,
 int pheromone, int nbhPheromon, int evaporation, ←
 int cellDef,
 int min, int max, int cellAntMax, QWidget *parent ←
) : QMainWindow (parent)
{
 setWindowTitle ("Ant Simulation");

 this->width = width;
 this->height = height;
 this->max = max;
 this->min = min;

 cellWidth = 6;
 cellHeight = 6;

 setFixedSize (QSize (width*cellWidth, height*cellHeight));

 grids = new int**[2];
 grids[0] = new int*[height];
 for (int i=0; i<height; ++i) {
 grids[0][i] = new int [width];
 }
 grids[1] = new int*[height];
 for (int i=0; i<height; ++i) {
 grids[1][i] = new int [width];
 }

 gridIdx = 0;
 grid = grids[gridIdx];
```

```
for (int i=0; i<height; ++i)
 for (int j=0; j<width; ++j) {
 grid[i][j] = cellDef;
 }

ants = new Ants();

antThread = new AntThread (ants, grids, width, height, delay, ←
 numAnts, pheromone,
 nbhPheromon, evaporation, min, max, ←
 cellAntMax);

connect (antThread, SIGNAL (step (int)),
 this, SLOT (step (int)));

antThread->start();

}

void AntWin::paintEvent (QPaintEvent*)
{
 QPainter qpainter (this);

grid = grids[gridIdx];

for (int i=0; i<height; ++i) {
 for (int j=0; j<width; ++j) {

 double rel = 255.0/max;

 qpainter.fillRect (j*cellWidth, i*cellHeight,
 cellWidth, cellHeight,
 QColor (255 - grid[i][j]*rel,
 255,
 255 - grid[i][j]*rel));

 if (grid[i][j] != min)
 {
 qpainter.setPen (
 QPen (
 QColor (255 - grid[i][j]*rel,
 255 - grid[i][j]*rel, 255),
 1)
);

 qpainter.drawRect (j*cellWidth, i*cellHeight,
 cellWidth, cellHeight);
 }
 }
}
```

```
 qpainter.setPen (
 QPen (
 QColor (0, 0, 0),
 1)
);

 qpainter.drawRect (j*cellWidth, i*cellHeight,
 cellWidth, cellHeight);

}

}

for (auto h: *ants) {
 qpainter.setPen (QPen (Qt::black, 1));

 qpainter.drawRect (h.x*cellWidth+1, h.y*cellHeight+1,
 cellWidth-2, cellHeight-2);

}

qpainter.end();
}

AntWin::~AntWin()
{
 delete antThread;

 for (int i=0; i<height; ++i) {
 delete[] grids[0][i];
 delete[] grids[1][i];
 }

 delete[] grids[0];
 delete[] grids[1];
 delete[] grids;

 delete ants;
}

void AntWin::step (const int &gridIdx)
{
 this->gridIdx = gridIdx;
 update();
}
```

A konstruktor és destruktör definiálása, cellaméret (szélesség, magasság) beállítása, a hangyák kirajzolása történik a főprogramban, ha részletesek akarunk lenni. Szűkszavúan ismét annyi, hogy a headerben

"beharangozott", ergó deklarált elemeket most definináltuk.

Fordítás és futtatás az 5.5-ös feladatban leírtak szerint.

## 7.2. Java életjáték

Írd meg Java-ban a John Horton Conway-féle életjátékot, valósítsa meg a sikló-kilövőt!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/conway\\_codes/sejtauto\\_java/](https://gitlab.com/kincsa/bhax/blob/master/conway_codes/sejtauto_java/)

Tanulságok, tapasztalatok, magyarázat...

Mi is az életjáték? Egy olyan játék ahol a játékos feladata csupán annyi, hogy megad egy induló alakzatot majd ezután várja az eredményt. A játék alapját egy négyzetrácsos ablak adja amelyben a mezőket celláknak, a mezőkben megtalálható korongokat pedig sejteknek nevezzük. A sejt környezetének a hozzá legközelebb eső 8 mezőt tekintjük, a sejt szomszédján a környezetében található sejteket értjük.

A játék elején a játékosnak lehetősége van cellákba sejteket helyeznie, azonban ahogy már említettem, ezt leszámítva nincs beleszólása a játékosnak a játék kimenetelébe.

A játék körökre van osztva. Egy sejttel minden egyes körben különböző dolgok történhetnek, melyek a következők:

A sejt túlél a kört abban az esetben ha pontosan 2 vagy pontosan 3, még élő szomszédja van.

A sejt elpusztul akkor, ha 2-nél kevesebb vagy 3-nál több, még élő szomszédja van.

Új sejt is születhet, ez akkor valósul meg, ha az adott cella környezetében pontosan 3 sejt található.

Több, egymás közvetlen közelében található sejt alakzatot alkot. Az egyik legjelentősebb ilyen alakzat a sikló, aminek érdekessége, hogy átlósan mozog mindig egy-egy kockányit miközben változtatja formáját, mozgása végén azonban visszatér a kiindulási formájába.

A bevezető elméleti hátterének forrásául szolgált: [link](#)

A következőkben szokás szerint darabokra szedem a programot, így kerül elemzésre:

```
public class Sejtautomata extends java.awt.Frame implements Runnable {
 /** Egy sejt lehet élő */
 public static final boolean ÉLŐ = true;
 /** vagy halott */
 public static final boolean HALOTT = false;
 /** Két rácst használunk majd, az egyik a sejttér állapotát
 * a t_n, a másik a t_n+1 időpillanatban jellemzi. */
 protected boolean[][][] rácok = new boolean[2][][];
 /** Valamelyik rácsra mutat, technikai jellegű, hogy ne kelljen
 * [2][][]-ból az első dimenziót használni, mert vagy az
 * egyikre
 * állítjuk, vagy a másikra. */
 protected boolean[][] rács;
 /** Megmutatja melyik rács az aktuális: [rácsIndex][][] */
```

```
protected int rácsIndex = 0;
/** Pixelben egy cella adatai. */
protected int cellaSzélesség = 20;
protected int cellaMagasság = 20;
/** A sejttér nagysága, azaz hányszor hány cella van? */
protected int szélesség = 20;
protected int magasság = 10;
/** A sejttér két egymást követő t_n és t_{n+1} diszkrét id ←
 őpillanata
 közötti valós idő. */
protected int várakozás = 1000;
// Pillanatfelvétel készítéséhez
private java.awt.Robot robot;
/** Készítsünk pillanatfelvételt? */
private boolean pillanatfelvétel = false;
/** A pillanatfelvételek számozásához. */
private static int pillanatfelvételSzámláló = 0;
```

A Sejtautomata osztályban járunk amely osztály a `java.awt.Frame` osztályt használja majd a keret létrehozásához vagyis a grafikus megjelenítéshez. Itt kerülnek létrehozásra a sejtek állapotát jelző változók, a cella adatokat tároló változók, egy logikai változó ami azt hivatott tárolni, készítettük-e már pillanatképet vagy sem és maga a rács is amin a szimuláció fog zajlani.

Az `implements Runnable` kifejezés is fontos esetünkben, hiszen ez teszi majd lehetővé a futtathatását a `thread`-nek (szálnak).

```
public Sejtautomata(int szélesség, int magasság) {
 this.szélesség = szélesség;
 this.magasság = magasság;
 // A két rács elkészítése
 rácsok[0] = new boolean[magasság][szélesség];
 rácsok[1] = new boolean[magasság][szélesség];
 rácsIndex = 0;
 rács = rácsok[rácsIndex];
 // A kiinduló rács minden cellája HALOTT
 for(int i=0; i<rács.length; ++i)
 for(int j=0; j<rács[0].length; ++j)
 rács[i][j] = HALOTT;
 // A kiinduló rácsra "élőlényeket" helyezünk
 //sikló(rács, 2, 2);
 siklóKilövő(rács, 5, 60);
 // Az ablak bezárásakor kilépünk a programból.
 addWindowListener(new java.awt.event.WindowAdapter() {
 public void windowClosing(java.awt.event.WindowEvent e) {
 setVisible(false);
 System.exit(0);
 }
 });
}
```

Létrehoz egy `Sejtautomata` objektumot vagyis ez a konstruktorunk. A cellák állapotát megfelelően beállítjuk, ahogy az a megjegyzésben is olvasható, kezdéskor minden cella halott, ehhez persze előbb végig

kell menni a sorokon és oszlopokon, ez a for ciklussal valósul meg. A siklókilövőket elhelyezzük a rácson illetve egy figyelő function is bele kerül a függvénybe, ami figyeli, mikor zárjuk be az ablakot. Ha ez megtörténik, kilép a programból, hiszen nem lenne előnyös ha utána is futna...

```
addKeyListener(new java.awt.event.KeyAdapter() {
 // Az 'k', 'n', 'l', 'g' és 's' gombok lenyomását figyeljük
 public void keyPressed(java.awt.event.KeyEvent e) {
 if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K) {
 // Felezük a cella méreteit:
 cellaSzélesség /= 2;
 cellaMagasság /= 2;
 setSize(Sejtautomata.this.szélesség*cellaSzélesség,
 Sejtautomata.this.magasság*cellaMagasság);
 validate();
 } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {
 // Duplázzuk a cella méreteit:
 cellaSzélesség *= 2;
 cellaMagasság *= 2;
 setSize(Sejtautomata.this.szélesség*cellaSzélesség,
 Sejtautomata.this.magasság*cellaMagasság);
 validate();
 } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S) {
 pillanatfelvétel = !pillanatfelvétel;
 } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G) {
 várakozás /= 2;
 } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L) {
 várakozás *= 2;
 }
 repaint();
 }
});

// Egér kattintó események feldolgozása:
addMouseListener(new java.awt.event.MouseAdapter() {
 // Egér kattintással jelöljük ki a nagyítandó területet
 // bal felső sarkát vagy ugyancsak egér kattintással
 // vizsgáljuk egy adott pont iterációit:
 public void mousePressed(MouseEvent m) {
 // Az egérmutató pozíciója
 int x = m.getX()/cellaSzélesség;
 int y = m.getY()/cellaMagasság;
 rácsok[rácsIndex][y][x] = !rácsok[rácsIndex][y][x];
 repaint();
 }
});

// Egér mozgás események feldolgozása:
addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
 // Vonzolással jelöljük ki a négyzetet:
 public void mouseDragged(MouseEvent m) {
 int x = m.getX()/cellaSzélesség;
```

```
 int y = m.getY() / cellaMagasság;
 rácsok[rácsIndex][y][x] = ÉLŐ;
 repaint();
 }
});
```

Szükséges figyelni a billentyűzetről érkező inputokat is, ugyanis egyes gombok lenyomására más és más-féleképpen kell reagálnia programunknak. Ugyanez igaz az egérről érkező inputra is, szintén nyomon kell követni az egészséges működéshez.

```
// Cellaméretek kezdetben
cellaSzélesség = 10;
cellaMagasság = 10;
// Pillanatfelvétel készítéséhez:
try {
 robot = new java.awt.Robot(
 java.awt.GraphicsEnvironment.
 getLocalGraphicsEnvironment().
 getDefaultScreenDevice());
} catch(java.awt.AWTException e) {
 e.printStackTrace();
}

// A program ablakának adatai:
setTitle("Sejtautomata");
setResizable(false);
setSize(szélesség*cellaSzélesség,
 magasság*cellaMagasság);
setVisible(true);
// A sejttér életrekeltése:
new Thread(this).start();
}
/** A sejttér kirajzolása. */
public void paint(java.awt.Graphics g) {
 // Az aktuális
 boolean [][] rács = rácsok[rácsIndex];
 // rácsot rajzoljuk ki:
 for(int i=0; i<rács.length; ++i) { // végig lépked a sorokon
 for(int j=0; j<rács[0].length; ++j) { // s az oszlopok
 // Sejt cella kirajzolása
 if(rács[i][j] == ÉLŐ)
 g.setColor(java.awt.Color.BLACK);
 else
 g.setColor(java.awt.Color.WHITE);
 g.fillRect(j*cellaSzélesség, i*cellaMagasság,
 cellaSzélesség, cellaMagasság);
 // Rács kirajzolása
 g.setColor(java.awt.Color.LIGHT_GRAY);
 g.drawRect(j*cellaSzélesség, i*cellaMagasság,
```

```
 cellaSzélesség, cellaMagasság);
 }

}

// Készítünk pillanatfelvételt?
if(pillanatfelvétel) {
 // a biztonság kedvéért egy kép készítése után
 // kikapcsoljuk a pillanatfelvételt, hogy a
 // programmal ismerkedő Olvasó ne írja tele a
 // fájlrendszerét a pillanatfelvitételekkel
 pillanatfelvétel = false;
 pillanatfelvétel(robot.createScreenCapture
 (new java.awt.Rectangle
 (getLocation().x, getLocation().y,
 szélesség*cellaSzélesség,
 magasság*cellaMagasság)));
}
}
```

Beállításra kerülnek a cellaméretek és a programablak jellemzői is (mérete, címe, átméretezhető lehet-e stb..) illetve a new kulcsszóval létrehozunk egy új Thread-et, amit aztán a start function-nel el is indítunk. Végül a paint függvénnyel a sejttér kerül megrajzolásra, melyhez segítségül hívjuk a Java awt.Graphics osztályát. Megtörténik a rajzolás a fekete, fehér és szürke színek kombinálásával. Megvizsgáljuk, készült-e a rács jelenlegi állapotáról pillanatfelvétel.

```
public int szomszédokSzáma(boolean [][] rács,
 int sor, int oszlop, boolean állapot) {
 int állapotúSzomszéd = 0;
 // A nyolcszomszédok végigzongorázása:
 for(int i=-1; i<2; ++i)
 for(int j=-1; j<2; ++j)
 // A vizsgált sejtet magát kihagyva:
 if(!((i==0) && (j==0))) {
 // A sejttérből szélénék szomszédai
 // a szembe oldalakon ("periódikus határfeltétel")
 int o = oszlop + j;
 if(o < 0)
 o = szélesség-1;
 else if(o >= szélesség)
 o = 0;

 int s = sor + i;
 if(s < 0)
 s = magasság-1;
 else if(s >= magasság)
 s = 0;

 if(rács[s][o] == állapot)
 ++állapotúSzomszéd;
```

```
 }

 return állapotúSzomszéd;
}
```

Az adott sejt mind a 8 szomszédjának vizsgálata annak érdekében, hogy el tudjuk dönteneni, hány szomszédja is lesz a sejtünknek ami befolyásolni fogja, hogy életben marad/megszületik-e vagy sem. A függvény a szomszédok számát adjva vissza

```
public void időFejlődés() {

 boolean [][] rácsElőtte = rácsok[rácsIndex];
 boolean [][] rácsUtána = rácsok[(rácsIndex+1)%2];

 for(int i=0; i<rácsElőtte.length; ++i) { // sorok
 for(int j=0; j<rácsElőtte[0].length; ++j) { // oszlopok

 int élők = szomszédokSzáma(rácsElőtte, i, j, ÉLŐ);

 if(rácsElőtte[i][j] == ÉLŐ) {
 /* Élő élő marad, ha kettő vagy három élő
 szomszedja van, különben halott lesz. */
 if(élők==2 || élők==3)
 rácsUtána[i][j] = ÉLŐ;
 else
 rácsUtána[i][j] = HALOTT;
 } else {
 /* Halott halott marad, ha három élő
 szomszedja van, különben élő lesz. */
 if(élők==3)
 rácsUtána[i][j] = ÉLŐ;
 else
 rácsUtána[i][j] = HALOTT;
 }
 }
 }
 rácsIndex = (rácsIndex+1)%2;
}

/** A sejttér időbeli fejlődése. */
public void run() {

 while(true) {
 try {
 Thread.sleep(várakozás);
 } catch (InterruptedException e) {}

 időFejlődés();
 repaint();
 }
}
```

A feladat elején már tárgyalt Conway-féle életjáték szabályainak megfelelően nézzük, hogy élő vagy halott az adott sejt, esetleg megszületik.

Megérkeztünk a run függvényhez, ami miatt elengedhetetlen volt a program elejére rakkunk a Runnable kulcsszót. Egy végtelen ciklus foglal benne helyet ami azt eredményezi, hogy addig fut amíg mi kívülről meg nem szakítjuk, addig rajzolja ki újra és újra a képernyónkre a képet.

```
public void sikló(boolean [][] rács, int x, int y) {

 rács[y+ 0][x+ 2] = ÉLŐ;
 rács[y+ 1][x+ 1] = ÉLŐ;
 rács[y+ 2][x+ 1] = ÉLŐ;
 rács[y+ 2][x+ 2] = ÉLŐ;
 rács[y+ 2][x+ 3] = ÉLŐ;

}

public void siklóKilövő(boolean [][] rács, int x, int y) {

 rács[y+ 6][x+ 0] = ÉLŐ;
 rács[y+ 6][x+ 1] = ÉLŐ;
 rács[y+ 7][x+ 0] = ÉLŐ;
 rács[y+ 7][x+ 1] = ÉLŐ;

 rács[y+ 3][x+ 13] = ÉLŐ;

 rács[y+ 4][x+ 12] = ÉLŐ;
 rács[y+ 4][x+ 14] = ÉLŐ;

 rács[y+ 5][x+ 11] = ÉLŐ;
 rács[y+ 5][x+ 15] = ÉLŐ;
 rács[y+ 5][x+ 16] = ÉLŐ;
 rács[y+ 5][x+ 25] = ÉLŐ;

 rács[y+ 6][x+ 11] = ÉLŐ;
 rács[y+ 6][x+ 15] = ÉLŐ;
 rács[y+ 6][x+ 16] = ÉLŐ;
 rács[y+ 6][x+ 22] = ÉLŐ;
 rács[y+ 6][x+ 23] = ÉLŐ;
 rács[y+ 6][x+ 24] = ÉLŐ;
 rács[y+ 6][x+ 25] = ÉLŐ;

 rács[y+ 7][x+ 11] = ÉLŐ;
 rács[y+ 7][x+ 15] = ÉLŐ;
 rács[y+ 7][x+ 16] = ÉLŐ;
 rács[y+ 7][x+ 21] = ÉLŐ;
 rács[y+ 7][x+ 22] = ÉLŐ;
 rács[y+ 7][x+ 23] = ÉLŐ;
 rács[y+ 7][x+ 24] = ÉLŐ;
```

```
rács[y+ 8][x+ 12] = ÉLŐ;
rács[y+ 8][x+ 14] = ÉLŐ;
rács[y+ 8][x+ 21] = ÉLŐ;
rács[y+ 8][x+ 24] = ÉLŐ;
rács[y+ 8][x+ 34] = ÉLŐ;
rács[y+ 8][x+ 35] = ÉLŐ;

rács[y+ 9][x+ 13] = ÉLŐ;
rács[y+ 9][x+ 21] = ÉLŐ;
rács[y+ 9][x+ 22] = ÉLŐ;
rács[y+ 9][x+ 23] = ÉLŐ;
rács[y+ 9][x+ 24] = ÉLŐ;
rács[y+ 9][x+ 34] = ÉLŐ;
rács[y+ 9][x+ 35] = ÉLŐ;

rács[y+ 10][x+ 22] = ÉLŐ;
rács[y+ 10][x+ 23] = ÉLŐ;
rács[y+ 10][x+ 24] = ÉLŐ;
rács[y+ 10][x+ 25] = ÉLŐ;

rács[y+ 11][x+ 25] = ÉLŐ;
```

```
}
```

Két function, ami a sejttérbe siklókat és siklókilövőket teszi amikről tudni kell, hogy minden élők.

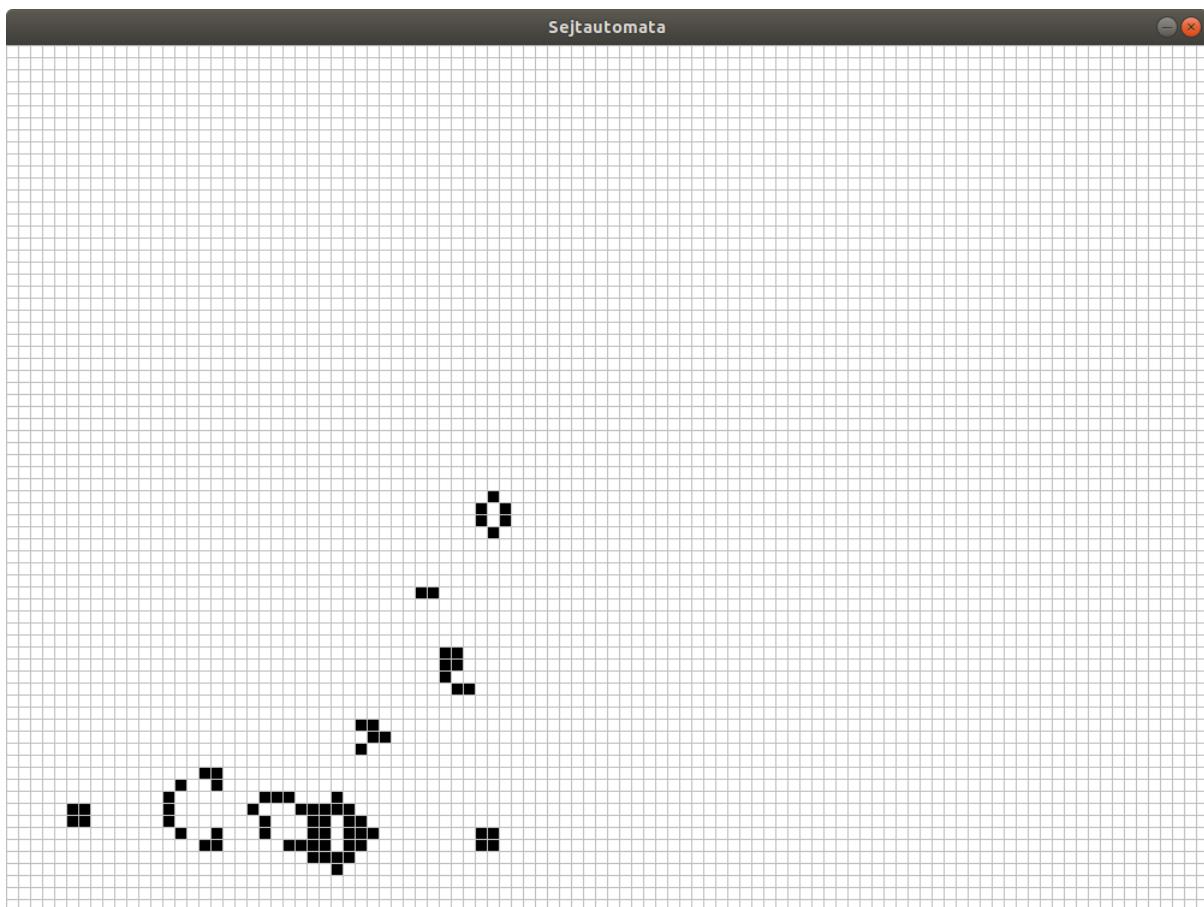
```
/** Pillanatfelvételek készítése. */
public void pillanatfelvétel(java.awt.image.BufferedImage felvetel) {
 // A pillanatfelvétel kép fájlneve
 StringBuffer sb = new StringBuffer();
 sb = sb.delete(0, sb.length());
 sb.append("sejtautomata");
 sb.append(++pillanatfelvételszámláló);
 sb.append(".png");
 // png formátumú képet mentünk
 try {
 javax.imageio.ImageIO.write(felvetel, "png",
 new java.io.File(sb.toString()));
 } catch(java.io.IOException e) {
 e.printStackTrace();
 }
}
// Ne villogjon a felület (mert a "gyári" update()
// lemeszelné a vászon felületét).
public void update(java.awt.Graphics g) {
 paint(g);
}
/**
 * Példányosít egy Conway-féle életjáték szabályos
 * sejttér obektumot.
```

```
/*
public static void main(String[] args) {
 // 100 oszlop, 75 sor mérettel:
 new Sejtautomata(100, 75);
}
```

A pillanatfelvétel függvény az S billentyű lenyomására hívódik meg. Hívásakor egy png képallo-mányt hoz létre, ezen elkészült képállományok számát is eltároljuk egy változóban.

A main-ben vagyunk, ahol már csak a példányosítás történik meg, létrejön egy objektum 100 és 75 paraméterekkel amik az ablak szélességet és magasságot jelölik. Ezzel ha megvagyunk, fordítás és futtatás után már tudjuk elvezni a Conway-féle életjátékot és megfigyelni a siklóagyúk működését.

Fordítás és futtatás után:



### 7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/conway\\_codes/sejtauto-qt\\_eletjatek/](https://gitlab.com/kincsa/bhax/tree/master/codes/conway_codes/sejtauto-qt_eletjatek/)

Tanulságok, tapasztalatok, magyarázat...

A feladat változatlan ami változik, az megvalósítás. Ugyanis Java helyett most C++-ba kell implementálnunk előző, Java forráskódunkat.

#### A `main.cpp`-ben:

```
#include <QApplication>
#include "sejtablak.h"
#include <QDesktopWidget>

int main(int argc, char *argv[])
{
 QApplication a(argc, argv);
 SejtAblak w(100, 75);
 w.show();

 return a.exec();
}
```

Inkludáljuk a Qt-vel való munkához szükséges osztályokat. A `QApplication` segítségével példányosítunk, egy objektumot hozunk létre, meghívásra kerül a konstruktur. Itt történik az ablak méretének megadása is.

#### A `sejtablak.h`-ban:

```
#ifndef SEJTABLAK_H
#define SEJTABLAK_H

#include < QMainWindow>
#include < QPaintEvent >
#include "sejtszal.h"

class SejtSzal;

class SejtAblak : public QMainWindow
{
 Q_OBJECT

public:
 SejtAblak(int szelesseg = 100, int magassag = 75, QWidget *parent ←
 = 0);

 ~SejtAblak();
 // Egy sejt lehet élő
 static const bool ELO = true;
 // vagy halott
 static const bool HALOTT = false;
 void vissza(int racsIndex);

protected:
 // Két rácsot használunk majd, az egyik a sejttér állapotát
 // a t_n, a másik a t_n+1 időpillanatban jellemzi.
```

```
bool ***racsok;
// Valamelyik rácsra mutat, technikai jellegű, hogy ne kelljen a
// [2][][]-ból az első dimenziót használni, mert vagy az egyikre
// állítjuk, vagy a másikra.
bool **racs;
// Megmutatja melyik rács az aktuális: [rácsIndex] []
int racsIndex;
// Pixelben egy cella adatai.
int cellaSzelesseg;
int cellaMagassag;
// A sejttér nagysága, azaz hányszor hány cella van?
int szelesseg;
int magassag;
void paintEvent(QPaintEvent*);
void siklo(bool **racs, int x, int y);
void sikloKilovo(bool **racs, int x, int y);

private:
SejtSzal* eletjatek;

};

#endif // SEJTABLAK_H
```

Itt kerül létrehozásra a SejtAblak kontruktora és destruktora. Ugyanitt kerül deklarálásra számos function, mint például a sejtek kirajzolásáért felelős paintEvent vagy épp a siklók kilövéséért felelős function, illetve a cella szélességi és magassági adatait leíró és tároló egész típusú változók. Ahogy említettem, ebben a header fájlban csak a function-ök és változók deklarálása történik, a definiálásukra később, egy másik fájlban kerül sor. (az már C++ forráskód lesz) Érdekességeképp megfigyelhetjük, hogy programunk legelején nem csupán a Qt-val való munkavégzéshez szükséges könyvtárakat inkludáljuk, hanem a sejtszal.h header fájlt is, tehát lehetőség van az utóbb említett fájlból található globális változókat és függvényeket használni.

### A **sejtablak.cpp**-ben:

```
void SejtAblak::paintEvent(QPaintEvent*) {
QPainter qpainter(this);
// Az aktuális
bool **racs = racsok[racsIndex];
// rácsot rajzoljuk ki:
for(int i=0; i<magassag; ++i) { // végig lépked a sorokon
 for(int j=0; j<szelesseg; ++j) { // s az oszlopok
 // Sejt cella kirajzolása
 if(racs[i][j] == ELO)
 qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
 cellaSzelesseg, cellaMagassag, Qt::black);
 else
 qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
```

```
 cellaSzelesseg, cellaMagassag, Qt::white);
 qpainter.setPen(QPen(Qt::gray, 1));

 qpainter.drawRect(j*cellaSzelesseg, i*cellaMagassag,
 cellaSzelesseg, cellaMagassag);
}
}

qpainter.end();
}

SejtAblak::~SejtAblak()
{
 delete eletjatek;

 for(int i=0; i<magassag; ++i) {
 delete[] racsok[0][i];
 delete[] racsok[1][i];
 }

 delete[] racsok[0];
 delete[] racsok[1];
 delete[] racsok;

 ...
 ...

 void SejtAblak::siklo(bool **racs, int x, int y) {

 racs[y+ 0][x+ 2] = ELO;
 racs[y+ 1][x+ 1] = ELO;
 racs[y+ 2][x+ 1] = ELO;
 racs[y+ 2][x+ 2] = ELO;
 racs[y+ 2][x+ 3] = ELO;

 }

 void SejtAblak::sikloKilovo(bool **racs, int x, int y) {

 racs[y+ 6][x+ 0] = ELO;
 racs[y+ 6][x+ 1] = ELO;
 racs[y+ 7][x+ 0] = ELO;
 racs[y+ 7][x+ 1] = ELO;
 }
}
```

Meg is érkeztünk az előbb emlegetett fájlunkhoz. Itt, a sejtablak.cpp-ben történnek az előbb még csak beharangozott definiálások. Többek között itt kap helyet a paintEvent function - amit a számunkra tökéletesen megfelelőnek tudunk beállítani a számos, Qt által biztosított lehetőségekkel, mint például a

rajzolást végrehajtó "toll" színe vagy vonalvastagsága - illetve a destruktur definiálása is.

Itt kap helyet a siklók megjelenítéséért és kilövéséért felelős function-ök is, utóbbi - ahogy azt a neve is sugallja - folyamatos jelleggel lövi ki magából a feladat elején már tárgyalt speciális élőlényeket.

#### A **sejtszal.h**-ban:

```
#ifndef SEJTSZAL_H
#define SEJTSZAL_H

#include <QThread>
#include "sejtablak.h"

class SejtAblak;

class SejtSzal : public QThread
{
 Q_OBJECT

public:
 SejtSzal(bool ***racsok, int szelesseg, int magassag,
 int varakozas, SejtAblak *sejtAblak);
 ~SejtSzal();
 void run();

protected:
 bool ***racsok;
 int szelesseg, magassag;
 // Megmutatja melyik rács az aktuális: [rácsIndex][][][]
 int racsIndex;
 // A sejttér két egymást követő t_n és t_n+1 diszkrét ←
 // időpillanata
 // közötti valós idő.
 int varakozas;
 void idoFejlodes();
 int szomszedokSzama(bool **racs,
 int sor, int oszlop, bool allapot);
 SejtAblak* sejtAblak;

};

#endif // SEJTSZAL_H
```

Egy újabb header fájl, amiben ugyanúgy inkludáljuk a másik, már kitárgyalt header fájlunkat a fejezet elején említett céllal. Tehát ez a megoldás vica-versa működik. Ebben a fájlban ismét helyet kapnak public és és protected változók, function-ök, a konstruktor és a destruktur deklarálása is. Ez a fájl az, ami a program dinamikusságát, lényegében az interaktivitásáért felelős tagok deklarálását tartalmazza.

#### A **sejtszal.cpp**-ben:

```
#include "sejtszal.h"

SejtSzal::SejtSzal(bool ***racsok, int szelesseg, int magassag, int ←
 varakozas, SejtAblak *sejtAblak)
{
 this->racsok = racsok;
 this->szelesseg = szelesseg;
 this->magassag = magassag;
 this->varakozas = varakozas;
 this->sejtAblak = sejtAblak;

 racsIndex = 0;
}

int SejtSzal::szomszedokSzama(bool **racs,
 int sor, int oszlop, bool allapot) {
 int allapotuSzomszed = 0;
 // A nyolcszomszédök végigzongorázása:
 for(int i=-1; i<2; ++i)
 for(int j=-1; j<2; ++j)
 // A vizsgált sejtet magát kihagyva:
 if(!((i==0) && (j==0))) {
 // A sejttérből szélénk szomszédai
 // a szembe oldalakon ("periódikus határfeltétel")
 int o = oszlop + j;
 if(o < 0)
 o = szelesseg-1;
 else if(o >= szelesseg)
 o = 0;

 int s = sor + i;
 if(s < 0)
 s = magassag-1;
 else if(s >= magassag)
 s = 0;

 if(racs[s][o] == allapot)
 ++allapotuSzomszed;
 }
 }

 return allapotuSzomszed;
}

void SejtSzal::idoFejlodes() {

 bool ***racsElotte = racsok[racsIndex];
 bool ***racsUtana = racsok[(racsIndex+1)%2];

 for(int i=0; i<magassag; ++i) { // sorok
 for(int j=0; j<szelesseg; ++j) { // oszlopok
```

```
int elok = szomszedokSzama(racsElotte, i, j, SejtAblak ←
 ::ELO);

if(racsElotte[i][j] == SejtAblak::ELO) {
 /* Élő élő marad, ha kettő vagy három élő
 szomszedja van, különben halott lesz. */
 if(elok==2 || elok==3)
 racsUtana[i][j] = SejtAblak::ELO;
 else
 racsUtana[i][j] = SejtAblak::HALOTT;
} else {
 /* Halott halott marad, ha három élő
 szomszedja van, különben élő lesz. */
 if(elok==3)
 racsUtana[i][j] = SejtAblak::ELO;
 else
 racsUtana[i][j] = SejtAblak::HALOTT;
}
racsIndex = (racsIndex+1)%2;
}

/** A sejttér időbeli fejlődése. */
void SejtSzal::run()
{
 while(true) {
 QThread::msleep(varakozas);
 idoFejlodes();
 sejtAblak->vissza(racsIndex);
 }
}

SejtSzal::~SejtSzal()
{
```

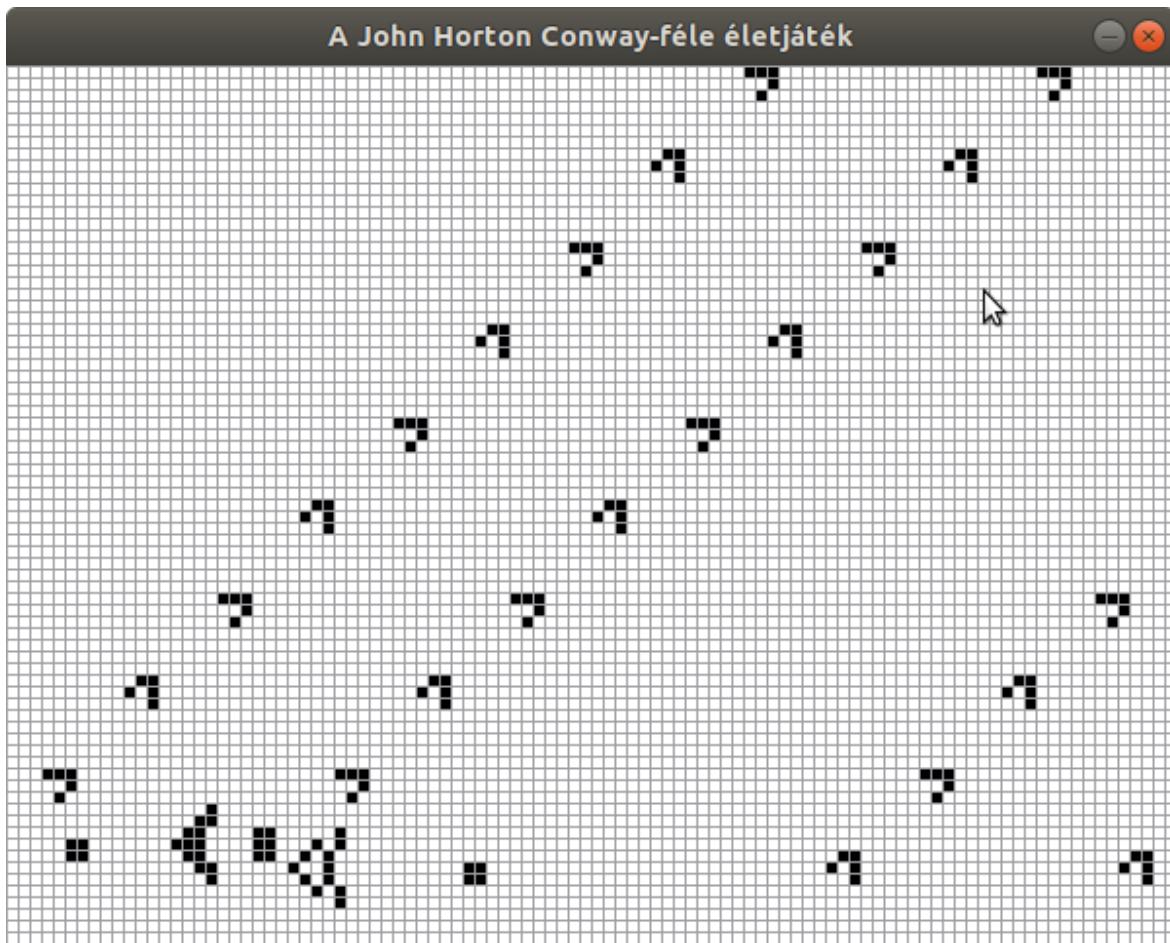
Ebben a fájlban minden fontos, nem lehet csupán pár dolgot kiemelni belőle, muszáj egy nagy egységeként tekintenünk rá. Amit tartalmaz (fentről lefelé haladva, sorban):

A `SejtSzal` konstruktorát, a sejt szomszédjainak számát meghatározó functiont. Az `idoFejlodes` functiont foglalja keretbe többek között ami azt a vizsgálatot végzi, amely során az adott sejtről eldöntjük, hogy halottá vagy élővé válik (vagy éppen marad), esetleg megszületik-e.

A `run` function-nel veszi kezdetét a tényleges időbeli "előrehaladás", benne egy végtelen while ciklus garantálja a megszakításig történő futást.

Ha a létrehozott `SejtSzal` példányra destrukturálásával zárul a programunk.

Az 5.5-ös feladatban leírtak szerinti fordítás és futtatás után:



## 7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/conway\\_codes/brainb/](https://gitlab.com/kincsa/bhax/tree/master/codes/conway_codes/brainb/)

Tanulságok, tapasztalatok, magyarázat...

A BrainB projekt célja a jövő kiemelkedő esportolóinak fellelése. A program az agy kognitív képességét méri.

A játékos feladata a Samu Entropy nevű karakteren tartani az egérmutatót ami az idő teltével természetesen egyre nehezebb hiszen újabb és újabb Entropy karakterek jelennek meg a képernyőn. A program érzékelni, ha Samu Entropy elvesztettük, ekkor csökkenti a többi Entropy számát. Elemezzük ki a kódot!

```
#include <QApplication>
#include <QTextStream>
#include <QtWidgets>
#include "BrainBWin.h"

int main (int argc, char **argv)
{
 QApplication app (argc, argv);
```

```
QTextStream qout (stdout);
qout.setCodec ("UTF-8");

qout << "\n" << BrainBWin::appName << QString::fromUtf8 (" ↵
Copyright (C) 2017, 2018 Norbert Bátfa") << endl;

qout << "This program is free software: you can redistribute it and ↵
/or modify it under" << endl;

.....
.....
.....

QRect rect = QApplication::desktop()->availableGeometry();
BrainBWin brainBWin (rect.width(), rect.height());
brainBWin.setWindowState (brainBWin.windowState() ^ Qt:: ↵
 WindowFullScreen);
brainBWin.show();
return app.exec();
}
```

Mivel a program a Qt grafikus felületét használja a program, az előző programokban is megfigyelhető osztályokat szükséges inkludálnunk, illetve a BrainBWin osztályt is. Deklarálásra kerül egy app nevű, QApplication típusú objektumot. Ebben a fájlban sokat használjuk a qout functiont, ez Qt-ben a cout lényegében, a standard outputra lehet vele írni. A function kódolását is be lehet állítani, esetünkben ez szabványos UTF-8 lesz. Itt történnek még az entropyk deklarálása is. Sőt, létrehozzuk a BrainBWin objektumot is.

Ebben a fejezetben tárgyalt többi programunkhoz hasonlóan a függvények és változók deklarálása itt is a .h kiterjesztésű header fájlokban vannak amiket inkludálunk osztályainkban.

Lássuk, miket tartalmaznak fájljaink!

### **BrainBWin.h**

Konstruktor és destruktur valamint az input "figyelő" függvények (vagyis amelyek az egér és billentyűzetről érkező inputukot értelmezik) illetve a kirajzolást/megjelenítést végző függvény deklarálása. A mentést végző függvény definiálása.

### **BrainBWin.cpp**

A előző header fájlban deklaráltak definiálása.

### **BrainBThread.h**

A 'Hero'-kat tartalmazó vectorra typedefet alkalmazunk, innentől kezdve Hero-sként hivatkozhatunk rá, stb...

### **BrainBThread.cpp**

A 'Hero'-k létrehozásra kerülnek. A destruktur definiálása. A run, pause és set\_paused függvények deklarálása.

## 8. fejezet

# Helló, Schwarzenegger!

### 8.1. Szoftmax Py MNIST

Python

Megoldás videó: <https://youtu.be/j7f9SkJR3oc>

Megoldás forrása: <https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0> (/tensorflow-0.9.0/tensorflow/exa...  
[https://progpater.blog.hu/2016/11/13/hello\\_samu\\_a\\_tensorflow-bol](https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol)

Tanulságok, tapasztalatok, magyarázat...

Tutoriált: Szegedi Csaba -<https://gitlab.com/dev.csaba.szegedi>

Kezdjük az alapokkal! Mi is az a **TensorFlow**? Egy könyvtár amit elsősorban gépi tanulásra használnak. Igen friss algoritmusról van szó, hogy csupán néhány éve jelent meg és minden össze néhány hónapja elérhető belőle a stabil kiadás.

Mi az az **MNIST (Modified National Institute of Standards and Technology) adatbázis**? Egy olyan óriási elemszámú adatbázis, mely kézzel írott számjegyeket tartalmaz, amit mi is fel fogunk használni a feladat-megoldásunk során.

A programkód maga egy nagy múlttal rendelkező nyelvben, **Pythonban** íródott. A programnyelv érdekes-sége, hogy nincsenek benne nyitó- és záró kapcsos zárójelek, ezek hiányát különböző mértékű tabulálással oldja meg.

Ahhoz, hogy a feladatot megoldjuk, telepíteni szükséges a Python-t és a TensorFlow-t is. Ezekhez segédlet [itt](#) és [itt](#) található.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse

Import data
from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf
```

```
import matplotlib.pyplot
```

Mivel Pythonról van szó, most nem include-oljuk, hanem importáljuk a megfelelő könyvtárakat. Lássuk a kódot!

```
def readimg():
 file = tf.read_file("sajat8a.png")
 img = tf.image.decode_png(file)
 return img

def main(_):
 mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
 mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)
 # Create the model
 x = tf.placeholder(tf.float32, [None, 784])
 W = tf.Variable(tf.zeros([784, 10]))
 b = tf.Variable(tf.zeros([10]))
 y = tf.matmul(x, W) + b

 # Define loss and optimizer
 y_ = tf.placeholder(tf.float32, [None, 10])

 # The raw formulation of cross-entropy,
 #
 # tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y))),
 # reduction_indices=[1]))
 #
 # can be numerically unstable.
 #
 # So here we use tf.nn.softmax_cross_entropy_with_logits on the raw
 # outputs of 'y', and then average across the batch.
 cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
 train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

 sess = tf.InteractiveSession()
 # Train
 tf.initialize_all_variables().run()
 print("-- A halozat tanítása")
 for i in range(1000):
 batch_xs, batch_ys = mnist.train.next_batch(100)
 sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
 if i % 100 == 0:
 print(i/10, "%")
```

Két függvénytel találjuk szemben magunkat. Az első azért felel, hogy a képünk beolvasásra kerüljön míg a második a main függvényünk, ahol az érdemi munka történik ugyanis itt kap helyet az az algoritmus, ami

ténylegesen a számokat felismeri. Ugyanitt kap helyet a hálózat tanítása is ami mintákat kap és ezekből a mintákból próbál tanulni.

```
Test trained model
print("-- A halozat tesztelese")
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print("Pontosság: ", sess.run(accuracy, feed_dict={x: mnist.test.images,
 y_: mnist.test.labels}))
print("-----")

print("-- A MNIST 42. tesztképenek felismerése, mutatom a számot, a -->
 továbbolteshez csukd be az ablakat")

img = mnist.test.images[42]
image = img

matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm.binary)
matplotlib.pyplot.savefig("4.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")

print("-- A saját kezi 8-asom felismerése, mutatom a számot, a -->
 továbbolteshez csukd be az ablakat")

img = readimg()
image = img.eval()
image = image.reshape(28*28)

matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm.binary)
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")

if __name__ == '__main__':
 parser = argparse.ArgumentParser()
 parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/mnist/input_data',
 help='Directory for storing input data')
```

```
FLAGS = parser.parse_args()
tf.app.run()
```

Megtörténik a hálózat tesztelése és a várva-várt pillanat, amikor is szembesülünk azzal, miként is ismeri fel a hálónk az egyes számjegyeket.

## 8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása: [https://github.com/tensorflow/tensorflow/blob/r1.4/tensorflow/examples/tutorials/mnist/mnist\\_deep.py?fbclid=IwAR3NgY5V7ybCM6ug\\_KKZUc9MnTjSshpfoCF0MRyCejK4Xmvi3AriS\\_MVZV8](https://github.com/tensorflow/tensorflow/blob/r1.4/tensorflow/examples/tutorials/mnist/mnist_deep.py?fbclid=IwAR3NgY5V7ybCM6ug_KKZUc9MnTjSshpfoCF0MRyCejK4Xmvi3AriS_MVZV8)

Tanulságok, tapasztalatok, magyarázat...

Feladat passzolva SMNIST által.

## 8.3. Minecraft-MALMÖ

Megoldás videó: <https://youtu.be/bAPSu3Rndi8>

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Feladat passzolva SMNIST által.

## 9. fejezet

# Helló, Chaitin!

### 9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó: <https://youtu.be/z6NJE2a1zIA>

Megoldás forrása:

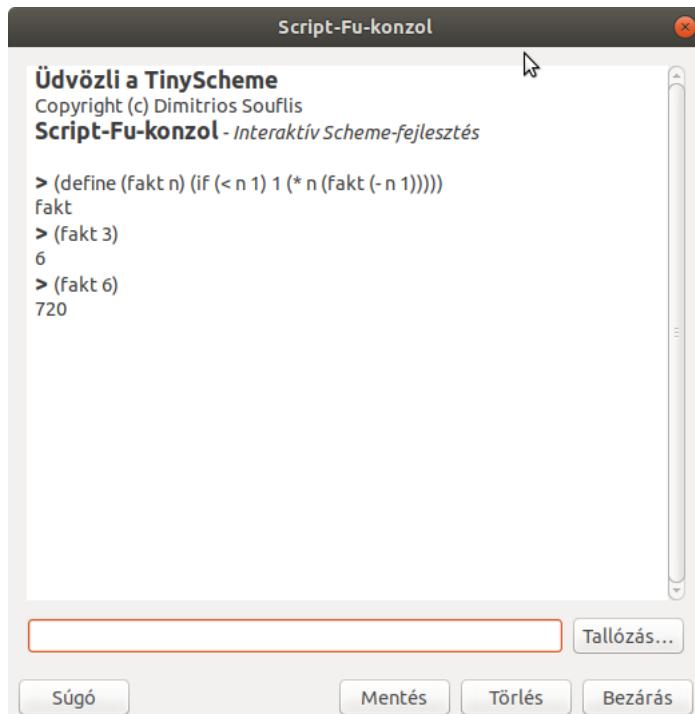
Tanulságok, tapasztalatok, magyarázat...

A **Lisp** nem egy újkeletű nyelv(család) a programnyelvek területén. Eredetileg 1958-ban jelent meg és hamar közkeletű vált a mesterséges intelligencia területén és manapság is több területen van használatban. Nevét (List Processing) az általa főként használt adatszerkezetről, a láncolt listáról kapta. A nyelvcsalád számos nyelvváltozattal rendelkezik, ezek közül példaként néhány: AutoLISP, Common Lisp, Scheme. A Lisp nyelveken írt programok tipikus képét a zárójelek adják.

Most hogy már van egy elképzelésünk a nyelvről, nézzük is meg, hogyan sikerült megvalósítani a feladatot!

A futtatható Lisp kódot a **GIMP** (GNU Image Manipulation Program) nevű képszerkesztő program által biztosított, scriptek írására alkalmas, a programon belül az alábbi módon elérhető konzolba írtam:

Most lássuk magát a kódot!



Ami egyből szembetűnő lehet az a nem kevés zárójel. De ahogy ezt már kitárgyaltuk, ez az egyik ismertetőjele a Lisp programoknak.

Balról jobbra haladva elemezzük a program egyetlen sorát! A `define` kulcsszóval vezetjük be a függvényünket, majd nevet is adunk neki - ez nálunk a `fakt` névre hallgat most, `n` pedig a paraméter, vagyis a szám aminek a faktoriálisára kíváncsiak vagyunk. Feltételvizsgálat következik amit a `if` kulcsszó vezet be. A feltételvizsgálatban megfigyelhető, hogy az eddigi programnyelvekkel ellentétben a műveleti jel nem a két tag között, hanem előttük áll. Ha a zárójelben lévő feltétel teljesül, a feltételvizsgálat a zárójel utáni első értéket adja vissza (jelen esetben 1), ha nem, akkor az azután következő, szóközzel elválasztott értéket, ami nem más, mint `n` paraméter szorzása a `fakt` függvényvel, ami most `n-1`-re kerül meghívásra. Így a kódunk rekurzív és iteratív egyszerre. Rekurzív azért, mert a függvény önmagát hívja meg, iteratív pedig azért, mert vérehajtása ismétlődő.

## 9.2. Gimp Scheme Script-fu: króm effekt

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely megvalósítja a króm effektet egy bemenő szövegre!

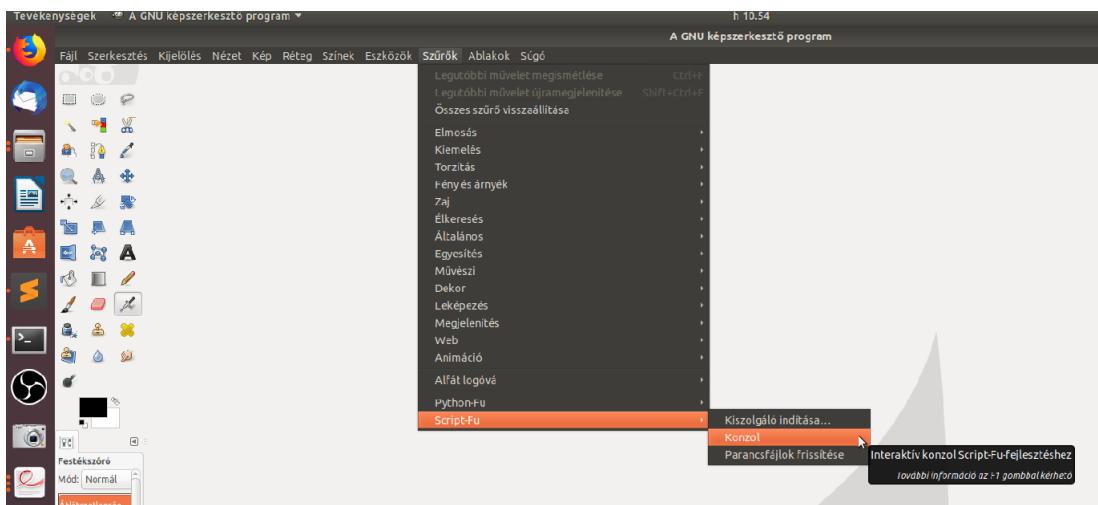
Megoldás videó: [https://youtu.be/OKdAkI\\_c7Sc](https://youtu.be/OKdAkI_c7Sc)

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/GIMP\\_Lisp/Chrome](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Chrome)  
[https://gitlab.com/kincsa/bhax/tree/master/codes/chaitin\\_codes](https://gitlab.com/kincsa/bhax/tree/master/codes/chaitin_codes)

Tanulságok, tapasztalatok, magyarázat...

Az előző feladat során már említést tettem arról, hogy a Lisp nyelvcsaládnak számos képviselője van. A most következő feladat során is egy ilyen képviselővel fogunk dolgozni, a Scheme nyelvvel. Ahogy említettem a Scheme programnyelv is a Lisp programnyelvek családjába tartozik. Az 1970-es évek közepén jelent meg de egyszerűsége miatt mind a mai találkozhatunk Scheme-kódokkal.

A most következő feladat során a Scheme egyik dialektusát, a Script-fut fogjuk használni arra, hogy az előző feladat során már megismert GIMP képszerkesztő programhoz egy olyan scriptet írunk, ami lehetővé teszi a bemenetként megadott szöveg "króm effektezését". Ezen script megírásához használhatjuk a már előző feladat során látott Script-Fu-konzolt is.



Lássuk, hogy néz ez ki (.scm kiterjesztésű) kód formájában!

```
(define (color-curve)
 (let* (
 (tomb (cons-array 8 'byte)))
)
 (aset tomb 0 0)
 (aset tomb 1 0)
 (aset tomb 2 50)
 (aset tomb 3 190)
 (aset tomb 4 110)
 (aset tomb 5 20)
 (aset tomb 6 200)
 (aset tomb 7 190)
 tomb)
)
```

Definiáljuk a `color-curve` függvényünket amivel egy tömböt töltünk fel 8 különböző értékkel. Ez a függvény még visszaköszön majd a feladat során a 9. lépésnél.

```
(define (elem x lista)
 (if (= x 1) (car lista) (elem (- x 1) (cdr lista)))
)

(define (text-wh text font fontsize)
 (let*
 (
 (text-width 1)
 (text-height 1)
)
)
```

```
(set! text-width (car (gimp-text-get-extents-fontname text fontsize ←
 PIXELS font)))
(set! text-height (elem 2 (gimp-text-get-extents-fontname text ←
 fontsize PIXELS font)))

(list text-width text-height)
)
)
```

Ebben a részben két függvény is definiálásra kerül. Az első, `elem` függvény egy paraméterként megadott listából szintén paraméterként megadott sorszámu elem értékét adja vissza. Ez a függvény felhasználásra is kerül a következő `text-wh` függvény során is, amely függvénynek a visszatérési értéke egy lista melyben a font szélessége és magassága található meg.

```
(define (script-fu-bhax-chrome text font fontsize width height color ←
 gradient)
 (let*
 (
 (image (car (gimp-image-new width height 0)))
 (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" ←
 100 LAYER-MODE-NORMAL-LEGACY)))
 (textfs)
 (text-width (car (text-wh text font fontsize)))
 (text-height (elem 2 (text-wh text font fontsize)))
 (layer2)
)
)
```

Az első sorban definiálásra kerül a scriptünk, ahol az látható, hogy a script nevét szóközzel elválasztva követik a paraméterek.

Ezt követi a `let*` függvény ami két részből áll, ezekből az elsőt tárgyaljuk ki ebben a bekezdésben. Ez az ún. "varlist" rész, itt azok a változók kerülnek deklarálásra, amelyeket a későbbiekben sokat fogunk használni. A több helyen is használt `car` függvény a GIMP eljárások és függvények által visszaadott listájának első elemét adja vissza. Természetesen bármilyen megkapott lista első elemét is visszaadja, nem csupán a GIMP-eseket...

```
; step 1
(gimp-image-insert-layer image layer 0 0)
(gimp-context-set-foreground '(0 0 0))
(gimp-drawable-fill layer FILL-FOREGROUND)
(gimp-context-set-foreground '(255 255 255))
```

Első lépésként az effektelni kívánt szöveget fehér betűszínnel fekete háttérre kell írnunk.

A megoldáshoz a GIMP eljárásbongészőjét hívjuk segítségül, ahol a keresőbe bepötyögve a megfelelő függvény/eljárás nevét, minden információt megkapunk róla.

A `gimp-image-insert-layer` egy új réteget ad hozzá képünkhoz, első paramétere a kép maga, második a réteg, harmadik a szülő-réteg, ami jelen esetünkben nincs (ezért 0), az utolsó paraméter a pozíciója a rétegnek, ami jelen esetben 0, hiszen a legfelső rétegnek szeretnénk.

A `gimp-context-set-foreground` eljárással az előtérszínt RGB(0 0 0)-ra, vagyis feketére állítjuk. A `gimp-drawable-fill` layer eljárásralrétegünket kitöljtük az előtérszínnel. Újra a `...set-foreground` eljárást használjuk, az előtérszínt most fehérre állítjuk.

```
(set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ↵
))
(gimp-image-insert-layer image textfs 0 0)
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (- (/ ↵
 height 2) (/ text-height 2)))

(set! layer (car (gimp-image-merge-down image textfs CLIP-TO-BOTTOM- ↵
 LAYER)))
```

A `set!`-tel a `gimp-text-layer-new` függvény által a megfelelő paraméterekkel elkészített, majd visszaadott réteget beállítjuk `textfs` névre. Ezt a réteget a következő sorban az `image-insert-layer` eljárással beszúrjuk.

A `gimp-layer-set-offsets` eljárással a rétegünket középre pozicionáljuk. Az utolsó sorban a `gimp-image-merge-down` függvénnyel lefelé haladva összefésüljük a rétegeinket és ezt a `layer` változóban tároljuk.

Az első lépés után ez a képünk:



**A második lépés:** Gauss-elmosás alkalmazása

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 15 TRUE TRUE)
```

Az eljárás paraméterei a következők: futási mód (esetünkben ez `RUN-INTERACTIVE`), a kép maga, "rajzolható input"(nekünk a réteg), az elmosódás pixelben mért sugara, elmosódás vízszintesen, elmosódás függőlegesen.

A lépés után ez a képünk:

**A harmadik lépés:** játék a színszintekkel

```
(gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE)
```

Az első paraméter, hogy min végezze el a műveletet a függvény, nekünk természetesen a rétegünkön kell majd, hogy elvégezve legyen. minden további paraméter az egyes színértékekkel kapcsolatos, az Eljárás-böngészőből kikeresgélhetőek.

A lépéssel a képünk:

**A negyedik lépés:** Gauss-elmosás, újra

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE)
```

A paraméterek megegyeznek a nemrég nézett Gauss-elmosást létrehozó eljárásával, egyedül az elmosás sugara kisebb most, mint az előbb volt, így ez most egy enyhébb elmosást eredményez, szinte nem is látható a különbség az előző képhez képest:



**Az ötödik lépés:** A fekete rész szín szerinti kijelölése majd a kijelölés invertálása

```
(gimp-image-select-color image CHANNEL-OP-REPLACE layer '(0 0 0))
(gimp-selection-invert image)
```

A `gimp-image-select-color` eljárással történik a kijelölés. Paraméterei: melyik képen kerül majd alkalmazásra, a kijelöléshez használatos művelet, "rajzolható input" és a kijelölendő szín.

A `gimp-selection-invert` pedig az egyetlen paraméterként kapott képen megfordítja a kijelölést.

**A hatodik lépés:** "Lebegő átlátszó kijelölés létrehozása"

```
(set! layer2 (car (gimp-layer-new image width height RGB-IMAGE "2" 100
 ↪ LAYER-MODE-NORMAL-LEGACY)))
(gimp-image-insert-layer image layer2 0 0)
```

Az első sorban a `set!`-tel `layer2`-t egy új rétegként állítjuk be, mely réteget a `gimp-layer-new` függvény állított elő a képen látható paraméterekkel. Ezek közül a 6. a legérdekesebb ami nem más, mint az opacitás (áttetszőség) amit 100-ra állítottunk be. A következő réteget beillesztjük a már előzőek során is látott `gimp-image-insert-layer` eljárással.



**A hetedik lépés:** "Az áttetszőség kitöltése átmenettel

```
(gimp-context-set-gradient gradient)
(gimp-edit-blend layer2 BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY GRADIENT-
 ↪ LINEAR 100 0 REPEAT-NONE
 FALSE TRUE 5 .1 TRUE width (/ height 3) width (- height (/ height
 ↪ 3)))
```

Az első eljárás beállítja az elsődleges átmenetet, a gimp-edit-blend eljárás pedig egy kezdő- és végpont között egy "keverést" hoz létre a paraméterek segítségével, melyekről bővebben az Eljárásböngészőben lehet olvasni.

Így állunk jelenleg:



**A nyolcadik lépés:** Buckaleképezés alkalmazása

```
(plug-in-bump-map RUN-NONINTERACTIVE image layer2 layer 120 25 7 5 5 0 ←
0 TRUE FALSE 2)
```

A plug-in-bump-map-pel buckaleképezést hozunk létre a layer2-n, bemenetként felhasználva másik rétegünket.



**A kilencedik lépés:** Színgörbékkel való játek

```
(gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))
```

A fémesebb hatás érdekében gimp-curves-spline-nal ügyeskedünk még és voilá, kész is vagyunk:



```
(script-fu-register "script-fu-bhax-chrome"
 "Chrome3"
 "Creates a chrome effect on a given text."
 "Norbert Bátfai"
 "Copyright 2019, Norbert Bátfai"
 "January 19, 2019"
 ""
 SF-STRING "Text" "Bátf41 Haxor"
 SF-FONT "Font" "Sans"
 SF-ADJUSTMENT "Font size" '(100 1 1000 1 10 0 1)
 SF-VALUE "Width" "1000"
 SF-VALUE "Height" "1000"
 SF-COLOR "Color" '(255 0 0)
 SF-GRADIENT "Gradient" "Crown molding"
)
(script-fu-menu-register "script-fu-bhax-chrome"
 "<Image>/File/Create/BHAX"
)
```

A scriptet ha nem a script-konzolon keresztül szeretnénk beadni, hanem inkludálni szeretnénk, hogy a GIMP-en belül könnyebben használható legyen, függvényt kell használnunk hozzá. Ezt teszi a `script-fu-register` függvény, amely definiálása során alapértelmezett értékeket is megadunk.

A kódunk zárásaként megírjuk az ún. "Menübe regisztráló függvényt". Ahogy az elnevezés is sugallja, ezen függvény hatására tudjuk majd kiválasztani grafikusan is, a menüből a scriptünket.

### 9.3. Gimp Scheme Script-fu: név mandala

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely név-mandalát készít a bemenő szövegből!

Megoldás videó: [https://bhaxor.blog.hu/2019/01/10/a\\_gimp\\_lisp\\_hackelete\\_a\\_scheme\\_programozasi\\_nyelv](https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelete_a_scheme_programozasi_nyelv)

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/GIMP\\_Lisp/Mandala](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Mandala)

[https://gitlab.com/kincsa/bhax/tree/master/codes/chaitin\\_codes](https://gitlab.com/kincsa/bhax/tree/master/codes/chaitin_codes)

Tanulságok, tapasztalatok, magyarázat...

A feladatunk az feladatunkhoz hasonló, ismét a GIMPet szeretnénk megbabrálni. Egy olyan script elkészítése a cél ami egy látványos **mandalát** állít elő a bemenetként megadott szövegből. Úgy gondolom, hogy bár az előző feladat is érdekes volt, ez méginkább az lesz hiszen egy jobban kézzelfoghatóbb, látványosabb eredményt fogunk kapni.

Lássuk a scriptünk kódját!

```
(define (elem x lista)
 (if (= x 1) (car lista) (elem (- x 1) (cdr lista)))
)
```

A program elején definiálásra kerülnek a későbbiek során használt függvényeink. Az első ilyen függvény a `elem` függvény ami megmondja, hogy a paraméterként megadott sorszámu helyen egy listában milyen elem szerepel. Ez egy rekurzív függvény.

```
(define (text-width text font fontsize)
(let*
 (
 (text-width 1)
)
 (set! text-width (car (gimp-text-get-extents-fontname text ←
 fontsize PIXELS font)))
 text-width
)
)
```

A `text-width` függvény az általunk alkalmazott font méretét adja vissza.

```
(define (text-wh text font fontsize)
(let*
 (
 (text-width 1)
 (text-height 1)
)
 ;;
 (set! text-width (car (gimp-text-get-extents-fontname text ←
 fontsize PIXELS font)))
 ;; ved ki a lista 2. elemét
 (set! text-height (elem 2 (gimp-text-get-extents-fontname text ←
 fontsize PIXELS font)))
 ;;
 (list text-width text-height)
)
)
```

A `wh` függvény a font méretét vagyis szélességét és magasságát adja vissza egy listában, mindezt az előbb említett sorrendben. A magasság megállapításához felhasználja a kódunk legelején definiált `elem` függvényt.

```
(define (script-fu-bhax-mandala text text2 font fontsize width ←
 height color gradient)
(let*
 (
 (image (car (gimp-image-new width height 0)))
 (layer (car (gimp-layer-new image width height RGB-IMAGE "←
 bg" 100 LAYER-MODE-NORMAL-LEGACY)))
 (textfs)
 (text-layer)
 (text-width (text-width text font fontsize))
 ;;;
 (text2-width (car (text-wh text2 font fontsize)))
 (text2-height (elem 2 (text-wh text2 font fontsize)))
 ;;;
 (textfs-width)
 (textfs-height)
 (gradient-layer)
)
)
```

Most, hogy megtettük a szükséges előkészületeket, vagyis definiáltuk a következőkben használatos függvényeinket, kezdődik a fő függvényünk, melyben először is a lokális változókat deklaráljuk és definiáljuk. Látható, hogy itt is felhasználásra kerül az `elem` függvény.

```
(gimp-image-insert-layer image layer 0 0)

(gimp-context-set-foreground '(0 255 0))
(gimp-drawable-fill layer FILL-FOREGROUND)
(gimp-image-undo-disable image)

(gimp-context-set-foreground color)
```

Létrehozásra kerül a réteg amire készül majd a mandalánk, itt kerül beállításra az előtérszín és szintén itt a réteg is kitöltésre előtérszínnel.

```
(set! textfs (car (gimp-text-layer-new image text font fontsize ←
 PIXELS)))
(gimp-image-insert-layer image textfs 0 -1)
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (/ ←
 height 2))
(gimp-layer-resize-to-image-size textfs)
```

A `textfs` változónak értéket adunk, ez lesz a rétegünk. Beállítjuk a réteg eltolását és a `gimp-layer-resize`-eljárással a réteg méretét a kép méretére állítjuk be.

```
(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate-simple text-layer ROTATE-180 TRUE 0 0)
(set! textfs (car (gimp-image-merge-down image text-layer CLIP-TO-←
 BOTTOM-LAYER)))
```

```
(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 2) TRUE 0 0)
(set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO- ↵
 BOTTOM-LAYER)))

(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 4) TRUE 0 0)
(set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO- ↵
 BOTTOM-LAYER)))

(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate text-layer (/ *pi* 6) TRUE 0 0)
(set! textfs (car(gimp-image-merge-down image text-layer CLIP-TO- ↵
 BOTTOM-LAYER)))
```

Több szövegréteg kerül létrehozásra melyeket aztán egy rétegbe fésülünk össze. Ezeket a rétegeket minden különböző mértékbená rotate eljárásal forgatni fogjuk a mandala hatás elérése érdekében.

```
(plug-in-autocrop-layer RUN-NONINTERACTIVE image textfs)
(set! textfs-width (+ (car(gimp-drawable-width textfs)) 100))
(set! textfs-height (+ (car(gimp-drawable-height textfs)) 100))

(gimp-layer-resize-to-image-size textfs)

(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width ↵
 2) (/ textfs-width 2)) 18)
 (- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) ↵
 (+ textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

(gimp-context-set-brush-size 22)
(gimp-edit-stroke textfs)

(set! textfs-width (- textfs-width 70))
(set! textfs-height (- textfs-height 70))

(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width ↵
 2) (/ textfs-width 2)) 18)
 (- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) ↵
 (+ textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

(gimp-context-set-brush-size 8)
(gimp-edit-stroke textfs)

(set! gradient-layer (car (gimp-layer-new image width height RGB- ↵
 IMAGE "gradient" 100 LAYER-MODE-NORMAL-LEGACY)))
```

```
(gimp-image-insert-layer image gradient-layer 0 -1)
(gimp-image-select-item image CHANNEL-OP-REPLACE textfs)
(gimp-context-set-gradient gradient)
(gimp-edit-blend gradient-layer BLEND-CUSTOM LAYER-MODE-NORMAL- ←
 LEGACY GRADIENT-RADIAL 100 0
REPEAT-TRIANGULAR FALSE TRUE 5 .1 TRUE (/ width 2) (/ height 2) (+ ←
 (+ (/ width 2) (/ textfs-width 2)) 8) (/ height 2))

(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

(set! textfs (car (gimp-text-layer-new image text2 font fontsize ←
 PIXELS)))
(gimp-image-insert-layer image textfs 0 -1)
(gimp-message (number->string text2-height))
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text2-width 2)) (- ←
 (/ height 2) (/ text2-height 2)))

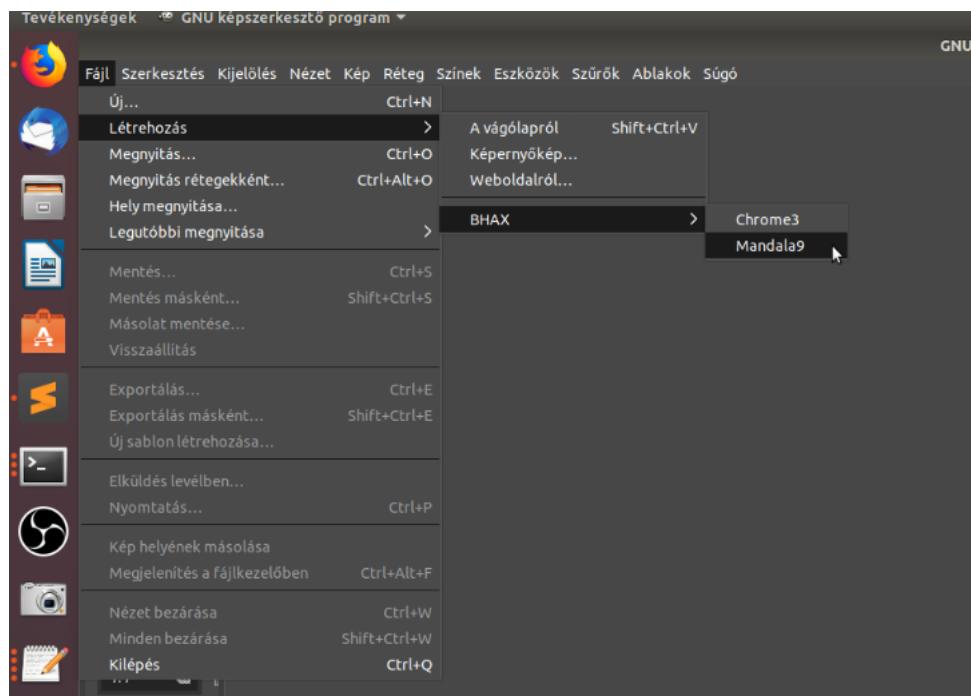
(gimp-display-new image)
(gimp-image-clean-all image)
)
)
```

Mi történik itt? A `textfs-width` és a `textfs-height` változók értéke több ízben változtatásra kerül. Ezen kívül beállítása kerül az `ecsetvastagság` is a `brush size` függvény segítségével, stb.. Ezek minden tőkéletes kinézetet szolgálják. Végül `textfs` változó értéket kap és a réteg is kicsit eltolásra kerül. Létrehozásra kerül egy új képállomány.

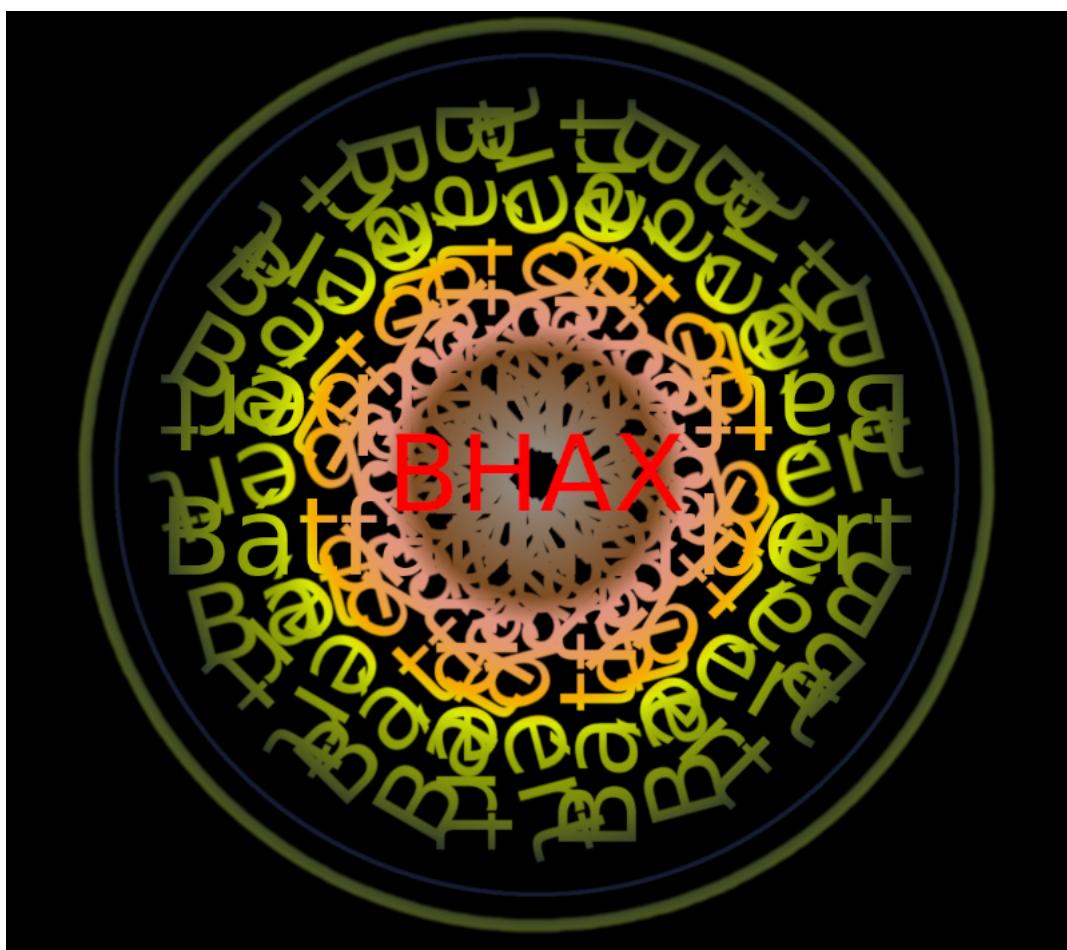
```
(script-fu-register "script-fu-bhax-mandala"
"MANDALA9"
"Creates a mandala from a text box."
"Norbert BátfaI"
"Copyright 2019, Norbert BátfaI"
"January 9, 2019"
""

SF-STRING "Text" "BátfaI Haxor"
SF-STRING "Text2" "BHAX"
SF-FONT "Font" "Sans"
SF-ADJUSTMENT "Font size" '(100 1 1000 1 10 0 1)
SF-VALUE "Width" "1000"
SF-VALUE "Height" "1000"
SF-COLOR "Color" '(255 0 0)
SF-GRADIENT "Gradient" "Deep Sea"
)
(script-fu-menu-register "script-fu-bhax-mandala"
"<Image>/File/Create/BHAX"
)
```

Menü- és script regisztráló függvények, előbbi alapértelmezett értékekkel ellátva, amikkel akkor találkozhatunk, ha a menüben a Fájl-Létrehozás-BHAX-Mandala9 módon hozzuk létre a képünket.



A végeredmény pedig:



## 10. fejezet

# Helló, Gutenberg!

### 10.1. Juhász István: Magas szintű programozási nyelvek - olvasónapló

#### 1.2 Alapfogalmak

Megismertük milyen 3 szintje van a programozási nyelveknek: a gépi nyelv, az assembly szintű nyelv illetve a magas szintű nyelv. A könyv az utóbbit tárgyalja részletesen. A magas szintű programnyelven írt programok a forrásprogramok, forrásszövegek. Megjelenik a szintaktika fogalma: a forrásszöveg írására vonatkozó nyelvtani szabályok. A szemantika pedig a tartalmi szabályok összessége. E két tényező határozza meg a magas szintű programozási nyelvet. Ezeket a szabályokat együttesen hivatkozási nyelvnek hívunk. A forrásszövegből szükséges a processzor által értelmezhető gépi kódá konvertálnunk a kódot, amire a rendelkezésre álló technikák: fordítóprogramos és interpreteres.

A fordítóprogram képes egy magas szintű programnyelven megírt kódból gépi kódú ún. tárgyprogramot előállítani, melynek lépései: lexikális-, szintaktikai-, szemantikai elemzés, kódgenerálás. Ha a lépések végrehajtódtak, egy, a processzor által értelmezhető gépi kódot kapunk amiből a kapcsolatszerkesztő állít elő futtatható kódot.

Az interpreteres megoldás is tartalmazza az első 3 lépést de tárgyprogramot nem készít. Az előző két technika akár együttesen is használható. Érdemes említeni a manapság előszeretettel használt IDE-kről (integrált fejlesztői környezet) amik komplex feladatkörrel rendelkeznek.

#### 1.3 A programnyelvek osztályozása

A programozási nyelvet alapvetően két fő csoportra lehet osztani. Imperatív- és dekleratív nyelvek.

Az imperatív nyelvek jellemzői: algoritmikus nyelvezek, tehát a leprogramozott algoritmus működteti a processzort. A program utasítások sorozata. A legfőbb eszközük a változók. Szorosan kötődnek a Neumann-architektúrához, alcsoportjai: eljárás- és objektumorientált nyelvek (A C és C++ is ilyen).

A dekleratív nyelvek jellemzői: nem algoritmikus nyelvezek, nem kötődnek szorosan a Neumann-architektúrához sem. Memóriaműveletekre sincs lehetőség. Alcsoportjai: funkcionális- és logikai nyelvek

#### Alapelemek

##### 2.1 - Karakterkészlet

A program legkisebb alkotórészei a karakterek. Ezekből épülnek fel a bonyolultabb nyelvi elemek, amelyek a következők: lexikális-, szintaktikai egységek, utasítások, programegységek, fordítási egységek, program. A programnyelvek a karaktereket általában így kategorizálják: betűk, számjegyek, egyéb karakterek.

A lexikális egységeket a fordító a lexikális elemzés során felismeri és tokenizálja. Számos fajtája van: többkarakteres szimbólum, szimbolikus név, címke, megjegyzés, literál.

A többkarakteres szimbólumoknak jellemzően a nyelv tulajdonít jelentést. Gyakran operátorok, mint pl.: ++, --, stb...

A szimbolikus nevek is ebbe a kategóriába tartoznak. Két része van: azonosító és kulcsszó. Az azonosító része arra való, hogy a programozó a saját eszközeit megnevezze vele majd ezzel a megnevezéssel a program során később hivatkozni tudjon. A kulcsszónak az adott nyelv által ismert karaktersorozat, általunk nem megváltoztatható.

A címke az utasítások jelölésére szolgál, hogy a program egy másik pontjáról hivatkozni lehessen rá.

A megjegyzések szerepe azért fontos, mert segítségükkel olyan karaktersorozatot jeleníthetünk meg, amit a fordító figyelmen hagy, ugyanakkor segítheti az olvasást és/vagy az értelmezést.

A literálok segítségével adott, meg nem változtatható értéket építhetünk be programkódunkba. Szintén két része van: típus és érték. C-ben vannak rövid egész-, hosszú egész-, valós-, karakter- és string literálok.

## 2.4 - Adattípusok

Az adatabsztrakció első megjelenési formája, egy absztrakt programozási eszköz. Léteznek típusos (melyek ismerik az eszközt) és nem típusos nyelvek (melyek nem ismerek az előbbi eszközt) is. Az adattípus neve egy azonosító, minden adattípus mögött van egy belső ábrázolási mód is. Egy adattípust 3 dolog határoz meg:

- tartomány (azok az elemek, amelyeket az adott típusú programozási eszköz felvehet értékként)
- műveletek,
- reprezentáció.

Minden típusos nyelv rendelkezik beépített típusokkal, egyes nyelvek engedélyezik a programozó által definiált saját típusokat is, ilyenkor megadjuk a típus tartományát, műveleteit és ábrázolási módját (reprezentációját). Az adattípusoknak két nagy csoportja létezik:

- egyszerű,
- összetett típusok.

Az egyszerű típusok csoportjába tartoznak: egész és valós típusok (numerikus típusúak), karakteres típus, egyes nyelvek esetében a logikai típus. Speciális egyszerű típus a felsorolásos típus és a sorszámozott típus is, amelynek elemei listát alkotnak. A C nyelv esetében a következő egyszerű típusok vannak jelen: egész fixpontos, valós lebegőpontos, karakteres, felsorolásos, mutató típus.

Az eljárásorientált nyelvek esetében a két legjelentősebb összetett típus a tömb és a rekord. A tömb egy statikus és homogén típus. Adatszerkezetek és algoritmusok nevű tárgyunk során is találkoztunk vele, tehát egy absztrakt adatszerkezet megjelenése típus szintjén. A C nyelv esetén a legjelentősebb összetett adattípusok: tömb (homogén) és struktúra, sőt akár kombinálhatjuk is őket: struktúra tömb

### 2.4.3. Mutató típus

Lényegét tekintve egyszerű típus, ami tárcímeket tárol. Segítségükkel valósítható meg az indirekt címzés. Egyik legfontosabb művelete a megcímzett tárterületen található érték elérése. Ha nem mutatnak sehol, akkor azt mondjuk, hogy a NULL (beépített konstans) értékre mutatnak.

### 2.5 A nevesített konstans

3 komponenssel rendelkezik: névvel, típussal és értékkel. Mindig deklarálni kell őket. Szerepük, hogy a programunkban gyakran előforduló értékeknek adhatunk nekik "beszédess" neveket. Könnyítést nyújt abban az esetben is, hogyha az adott értéket meg akarjuk változtatni, hiszen ez esetben elég egy helyen megváltoztatni. A `define` `nev` `literal` begépelésével tudjuk elérni a programban, hogy a `nev` minden előfordulása esetén `literal` értékkel "cserélje ki".

### 2.6 A változók

Olyan programozási eszköz, aminek 4 komponense van: név, attribútumok, cím és érték. A név egy azonosító. Az attribútumok azok a jellemzők, amik a változó futás közbeni működését határozzák meg. A változóknak értéket deklarációval segítségével rendelhetünk, ennek két fajtája van: explicit-, automatikus- és implicit deklaráció. A változo címe/címkomponense a tárnak az a címe, ahol a változó értéke elhelyezkedik. Az értékkomponens a címen elhelyezkedő bitkombinációként jelenik meg. C-ben a változó értékkomponensének meghatározására rendelkezésre áll az értékadó utasítás: `valtozo = kifejezes` ahol a `valtozo` az érték-, míg a `kifejezes` a címkomponenst jelöli.

### 2.7 Alapelemek az egyes nyelvekben

Minden nyelv más és más típusrendszerrel rendelkezik, nincs egységes megvalósítás de mi a C-t megnézzük részletesebben

Két fő típus található meg benne: aritmetikai és származtatott típusok. Előbbiek az egyszerűek míg utóbbiak az összetett típusok. Ezen fő típusok minden számos képviselővel rendelkeznek, mindenről részletesebben olvasni a könyv 42. oldalán lehet. Érdekes, hogy a C nyelvben nincs logikai típus, hamisnak az `int 0`, igaznak pedig az `int 1` értéke felel meg. Az aritmetikai típusok közé tartoznak az egész és karakteres típusok is, előjük írható az `unsigned` kulcsszó, ami a nem előjeles ábrázolást jelöl. Szintén az aritmetikai típusok közé tartoznak a felsorolásos típusok is, erre láthatunk példát a 43. oldalon. Az összetett típusok deklarálása a 44. oldalon található.

## 3. Kifejezések

Olyan szintaktikai eszközök, amiket arra használunk, hogy a program egy adott pontján a már eleve ismert értékekből új értéket határozunk meg. Két részük van: érték és típus. A kifejezések összetevői: operandusok, operátorok, kerek zárójelek. Az operandusok az értéket képviseli, az operátorok a műveleti jelek míg a kerek zárójelek a műveleti sorrend befolyásálásáért felelősek. Az operandusok és operátorok sorrendjétől függően beszélhetünk 3 féle esetről: prefix (az operátor az operandusok előtt áll), infix (az operátor a két operandus között) és postfix eset (az operátor a két operandus után áll). Ezek közül az infix nem feltétlenül egyértelmű minden esetben, a félreértések elkerülése érdekében szokás használni a precedenciáblázatot, mely a könyvben is helyet kapott az 51. oldal alján, hozzá jelmagyarázat pedig az 52. oldalon található. A kifejezés kiértékelése alatt típus- és értékmeghatározást értünk. Logikai kifejezésekben lehetséges opción rövidzár kiértékelés, vagyis hogy csak addig kell kiértékelni egy kifejezést amíg az eredmény egyértelmű nem lesz.

## 4. Utasítások

Két alapvető típusa van, deklarációs és végrehajtható utasítások. A deklarációs utasítások teljes mértékben a fordítóprogramnak szólnak, pl. szolgáltatást kérnek, üzemmódot állítanak be, stb..

A végrehajtható utasításoknak rengeteg kategóriája van, melyek a következők: értékeadó utasítás, üres utasítás, ugró utasítás, elágaztató utasítások, ciklusszervező utasítások, hívó utasítás, vezérlésátadó utasítások, Input/Output utasítások, egyéb utasítások.

Az értékeadó utasítás feladata a változó értékkomponensének beállítása.

Az üres utasításokat nem minden nyelv tartalmazza de fontos szerepük van: átláthatóbb szerkezetű programot kapunk.

Az ugró utasítással pontról egy adott címkével jelölt utasításra lehet ugrani.

Elágaztató utasítások:

Azok az utasítások melyekben egy feltételvizsgálat eredménye dönti el, milyen tevékenység kerüljön végrehajtásra. A vizsgált feltétel természetesen logikai kifejezés. Ezek az ún. if-utasítások. Létezik belőlük hosszú és rövid is. Egymásba is ágyazhatók, ilyenkor jelenhet meg a "csellengő else", amire a könyv megoldást is ad az 59. oldalon.

Az előbb az egyirányú elágaztató utasításokról volt szó de léteznek többirányúak is, C-ben jellemzően switch-case szerkezetben kerül ez megvalósításra.

Ciklusszervező utasítások:

Egy ciklusnak 3 része van: fej, mag és vég. A mag tartalmazza az ismételni kívánt utasításokat. A cikluson nem rendeltetésszerű működésének két iskolapéldája az üres ciklus, ami egyszer sem fut le és a végtelen ciklus amely futása soha nem áll le de természetesen mindenkor megvan a programozás területén számunkra előnyös alkalmazási területe.

Feltételes ciklusok

Az egyik fajtája a kezdőfeltételes ciklus amely esetében a fejben lévő feltétel kiértékelődése után hajtódik végre a ciklusmagban található kód addig amíg hamis nem lesz a feltétel. A végfeltételes ciklus esetében a feltétel a ciklus végében van, esetében először a mag hajtódik végre és csak ezután értékelődik ki a feltétel.

Előírt lépésszámú ciklusok

Amíg a ciklusfejben található ciklusváltozó megfelel a szintén ciklusfejben található feltételnek, fut a ciklusunk. Beszélhetünk előtesztelő vagy hárultesztelő ciklusokról ez esetben. Szükséges egy ún. "lépésköz" megadása is. Ezen kívül léteznek még felsorolásos és összetett ciklusok is, ezek jellemzése a 66. és 67. oldalon történik.

Ciklusszervező utasítások az egyes nyelvekben

Egy pár példasorban bemutatja a könyv az előbb tárgyalt ciklusok formai kinézetét és jellemzi működésüket, végrehajtási sorrendjüket.

Vezérlő utasítások

CONTINUE, BREAK és RETURN.

### A programok szerkezete

Programegységekről az eljárásorientált nyelvek esetében beszélhetünk, ezek a program önálló részei, melyek a következők lehetnek:

alprogram, blokk, csomag és még a taszk is ide sorolható.

Az alprogramok az újrafelhasználás eszközei, akkor használhatóak nagyon effektíven, ha többször szükséges elvégezni egy adott műveletet. Az alprogramot egyszer kell megírni, utána hivatkoznunk kell csak rá

ott, ahol az eredeti programrész szerepelt volna. Formálisan a részei: fej, törzs, vég; De mint programozási eszköu, 4 összetevője van: név, formális paraméterlista, törzs és környezet.

A fejben található a név, ami azonosítja az alprogramot. Szintén itt található a formális paraméterlista is mely kerek zárójelek között áll, fontos szerepe lesz majd a paraméterkiértékelés során. Akár üres is lehet.

A törzsben kapnak a helyet a különböző utasítások, pontosítva a deklarációs és végrehajtható utasítások. Lokális név fogalma: az alprogram lokális eszközeinek neve, vagyis mely eszközök az alprogramban kerültek deklarálásra. Értelemszerűen vannak globális nevek is.

Az alprogram környezete a globális változóinak együttese.

Két fajtája van: eljárás és függvény. Az eljárás tevékenységet hajt végre, azonban visszatérési értéke nincs. A függvény egy eredményt ad vissza, tetszőleges típusú értéket határoz meg. Itt jelenik meg a mellékhatás fogalma is, ez az a helyzet mikor egy függvény megváltoztatja paramétereit vagy környezetét. Ugyanez a fogalom az operátoroknál is visszaköszön majd.

## 5.2 Hívási lánc, rekurzió

Amikor egy programegység meghív egy másikat, az pedig egy következőt, stb-stb., akkor kialakul egy hívási lánc, aminek első tagja mindig a főprogram és az utoljára meghívott tag fejezi be először a működést.

Amikor egy aktív alprogramot hívunk meg, rekurzióról beszélünk, ami kétféle lehet. Lehet közvetlen, amikor egy alprogram önmagát hívja meg és lehet közvetett amikor a hívási láncban már eleve szereplő alprogramot hívunk meg.

## 5.4 Paraméterkiértékelés

A paraméterkiértékelés egy függvény vagy eljárás hívásnál megfigyelhető mechanizmus, amely során egy alprogram formális- és aktuális paraméterei egymáshoz történő megfeleltetése megtörténik. A formális paraméterek a formális paraméterlistában kapnak helyet, amiből csupán egyetlen darab van jelen a programban, míg aktuálisból nyilván annyi, amennyiszer meghívásra kerül az adott függvény vagy eljárás. Itt fontos tehát megjegyezni, hogy minden esetben a formálishoz kerül hozzárendelésre az aktuális lista. Azt, hogy melyik formális paraméterhez melyik aktuális fog hozzárendelődni, általában a sorrendi kötés határozza meg, ilyenkor amilyen sorrendben kerülnek felsorolásra a paraméterek, úgy kerülnek hozzárendelésre. Az első aktuális az első formálishoz, stb.. Létezik nem sorrend, hanem név szerinti kötés is de ez utóbbi kevésbé elterjedt, sőt létezik a 2 kombinálásából származó módszer is.

Az aktuális paraméterek számát illetően tudnillik, hogy meg kell egyeznie a(z általában) fix számú formális paraméterek számával VAGY kevesebb is lehet, ez esetben érték szerinti paraméterátadás lehetséges és a hiányzó aktuális paraméterek alapértelmezett értéket kapnak majd.

Ami az aktuális paraméterek típusát illeti, sok nyelv (köztük a C és a C++ is) azt vallja, hogy típusukban meg kell egyezniük a formális paraméterekkel. Vannak olyan nyelvek is amik engedik az aktuális paraméterek konvertálását megfelelő típusra.

## 5.5 Paraméterátadás

Esetében minden van egy ún. "hívó" és egy "hívott" ami minden az alprogram. Különböző paraméterátadási módok vannak: érték-, cím-, eredmény-, érték-eredmény-, név- és szöveg szerinti. Érték szerinti paraméterátadáskor a formális paraméterek rendelkeznek címkomponenssel az alprogram területén. Esetében az információ áramlása egyirányú, működése során értékmásolás hajtódik végre. Az aktuális paraméter kifejezés.

Cím szerinti paraméterádatásnál a formális paraméterek nem rendelkeznek címkomponenssel az előbb említett helyen. Az aktuális paramétereknek azonban rendelkeznek a címkomponenssel a hívó területen. Kétirányú az információ áramlása és gyors a működése hiszen nincs másolás. Az akt. paraméter változó.

Az eredmény szerinti paraméterátadás esetén elmondhatjuk róla, hogy a formális paraméter rendelkezik címkomponenssel a hívott területen és azt is, hogy az aktuális paraméter rendelkezik érték- és címkomponenssel is. A kommunikáció egyirányú és egyszer hajtódkik végre másolás. Az akt. paraméter változó.

Az érték-eredmény szerinti paraméterátadás: a formális paraméter rendelkezik címkomponenssel a hívott erületén és az aktuális paraméter rendelkezik érték- és címkomponenssel is. A kommunikáció kétirányú és másolás is kétszer hajtódkik végre. Az akt. paraméter változó.

## 5.6 A blokk

Olyan programegység ami egy másik programegységen helyezkedhet el, szerkezetileg van kezdete, törzse (utasításokat tartalmaz amik lehetnek végrehajthatók és deklarációsak) illetve vége. A blokkban megtalálható elemek egyértelműen meghatározhatók. Paraméterrel nem rendelkeznek. A blokk az egyes változó-/függvénynevek láthatóságában játszik fontos szerepet.

## 5.7 Hatáskör

Egy név hatásköre a programnak az a része, ahol az adott név ugyanazt a programozási eszközt (általában változót, függvényt) hivatkozza. Emlékezhetünk, hogy nem egyszer találkoztunk már programozás tanulmányaink során azzal a jelenséggel, hogy ha egy változót deklaráltunk blokkon belül, akkor azt már nem tuduk elérni, számunkra láthatatlan volt a blokkon kívül. Az ilyen nevet lokális névnek hívjuk, a blokkon kívülieket amikre blokkon belül hivatkozunk, szabad névnek hívjuk. Létezik statikus- és dinamikus hatás-körkezelés is, ezek jellemzőit is leírja a könyv. Megjelenik a globális név fogalma, amely név alatt az adott programegységen meg nem található de látható nevet értjük. A blokkok ahogy láthattuk a könyv példáinál már, természetesen egymásba ágyazhatók.

A blokk általános alakja:

```
{
 deklaraciok
 vegrehajthato utasitasok
}
```

## 5.9 Az egyes nyelvek eszközei - C

A fejezet példával szemlélteti a blokk és függvény alakját. Ha a függvény neve előtt nem szerepel típus, C-ben az alapértelmezés int, ha void a típus, függvény helyett eljárásról beszélünk. A függvény befejeződhet többféle módon: RETURN kifejezés vagy RETURN ha void típusú a függvény, egyébként a visszaadott érték határozatlan. A formális paraméterlistában a paramétereket adunk meg, miszerint milyen típusú adatokkal fog dolgozni a függvény, ez a paraméterátadás azonban nem cím-, hanem érték szerinti. Fordítási egységek a program elején include-olt forrásállományokat hívjuk, melyekben a legtöbb esetben deklarációkat találunk. A külső forrásállományokat a #include <forrasalomany> utasítással hivatkozunk. Tárolási osztály attribútumok megjelenése, melyek a következők: extern, auto, register, static. A hatáskör és élettartam szabályozására szolgálnak, bővebben a 97. oldalon olvashatunk róluk. Ugyanezen az oldalon egy rekurzív, faktoriálist kiszámító függvény is helyet kapott, ami egyszerre reprezentálja az előbb tárgyalt blokkokat és függvényeket is.

## 6. Absztrakt adattípus

Az ilyen fajta adattípusoknál nem ismerjük sem a reprezentációt sem pedig a műveletek implementációját ugyanis az adattípus nem mutatja a külvilág számára. Hozzáférésük interfészeken keresztül történik. Fontos fogalom, nagy a jelentőségük a biztonságos programozásban. szempontjából.

## 10. Generikus programozás

Az újrafelhasználhatóság eszköze, egy olyan eszközrendszer ami a legtöbb nyelvbe beépíthető. Lényege, hogy egy paraméterezhető forrásszöveg-mintát adunk meg. A mintaszövegből előállítható egy konkrét szöveg ami fordítható. A mintaszöveg típussal paraméterezhető. A generikus formális paraméterekkel rendelkezik amik száma adott, nem dinamikusan változó, a meghívása során történik meg a paraméterkiértékelés és átadás.

### 13. Input/output

Az Input/Output egy eszközrendszer ami a perifériákkal való kommunikáció megvalósításáért és annak fenntartásáért felelős, a memoriából a perifériákhoz vagy fordítva küld adatokat. Megjelenik az állomány fogalma, hiszen az I/O az állományok köré összpontosul. Az állomány egy programban lehet logikai illetve fizikai is. Funkció szerint is megkülönbezhethetjük az állományokat, lehetnek: input (bemeneti) állományok - csak olvasni lehet belőle; output (kimeneti) - csak írni lehet bele; input-output(be- és kimeneti állomány) - olvasni és írni is lehet. Az I/O során kétféle adatátviteli móddal találkozhatunk amelyekkel az adatok a memória és a periféria közötti mozgásukat végzik: folyamatos vagy bináris. Előbbinél van konverzió a tár és periféria közötti adatmozgatás során, míg utóbbinál nincs.

Ha programunkban állományokkal szeretnénk dolgozni, a következő lépések szükségesek hozzá:

- deklaráció
- összerendelés
- állomány megnyitása
- feldolgozás
- lezárás

Megjelenik az implicit állomány fogalma, amivel való munka során nem kell az előző lépéseket végigvinni, a rendszer oldja meg ezeket.

### Az egyes nyelvek I/O eszközei

Az egyes nyelvek Input/Output szempontjából igen különbözőek, vannak olyan nyelvek amik alapból nem is rendelkeznek olyan eszköztárral, ami képes lenne megvalósítani az I/O-t. Tehát elmondható, hogy az I/O-val való tevékenykedés erősen függ a platformtól, az operációs rendszertől és az adott implementációtól. A könyv ezen fejezete 6 programnyelv -többek között C- nyelv I/O lehetőségeit jellemzi pár szóban.

### 9. kivételkezelés

A fogalmat tisztázzuk: kivételről beszélünk olyan névvel és kódossal rendelkező események esetében, melyek megszakítást okoznak. A kivételkezelés lehetőséget ad a programozó számára, hogy az eredetileg operációs rendszer által betöltött feladatot, a megszakításkezelést felhozza a program szintjére, így a programozó bele tud nyúlni ebbe. A kivételkezelési folyamatot egy, kivételkezelőnek hívott programrész fogja végezni, ami természetesen kivétel esetén lép működésbe. Lehetőségünk van kivételek figyelések tiltására és engedélyezésére is, előbbire láthattunk példát a 3. fejezet 6. számú, **A források olvasása** című feladatában is.

### 9. Kivételkezelés a Javaban

A kivételkezelés Java nyelv esetében másképpen működik, ugyanis itt a nyelv egyik alap eszközéről beszélünk. Ha a program működése közben valamilyen speciális eset következik be, kivétel-objektum jön létre a

kivétel-osztályból, majd a kivétel a JVM-hez (Java Virtual Machine) kerül, aminek a feladata egy megfelelő típusú kivételkezelő (ami lényegében egy blokk, az egymásba ágyazásuk is megoldható) megtalálása. A kivételeknek 2 csoportja létezik Java-ban: ellenőrzöttek és nem ellenőrzöttek és szintén 2 alosztálya: Error és Exception. Ezután a könyv egy jó gyakorlati példán keresztül mutatja be a saját kivétel írását. A kivételkezelőről érdemes tudni, hogy is néz ki a szerkezete:

```
TRY
{utasitasok}

CATCH (tipus valtozonev)
{utasitasok}
...
FINALLY
{utasitasok}
```

Egy kis magyarázat ehhez a kódrészlethez: Ha a try blokkban kivételek találhatók, a JVM által a catch utasításokhoz kerül a vezérlés. Ha talál megfelelő típusú ágat, lefutnak az ott található utasítások, majd végrehajtódnak a FINALLY utáni utasítások és a program a FINALLY-t követően folytatódik. A CATCH ágat akár le is hagyhatjuk, opcionális.

## 10.2. KR: A C programozási nyelv - olvasónapló

### 2. fejezet: Típusok

A C nyelv esetén csupán néhány alap adattípusról beszélhetünk melyek a következők:

- char: mérete 1 byte, a karakterkészlet egy elemét tartalmazza
- int: egész szám
- float: lebegőpontos szám (egyszeres pontosságú)
- double: lebegőpontos szám (kétszeres pontosságú)

Aritmetikai operátorok esetén a következőkről beszélhetünk: +, -, \*, / és %. Ezek sorban: összeadás, kivonás, szorzás, egész osztás és maradékos osztás (modulo) operátorok. A modulo lebegőpontos számokra nem használható.

Természetesen léteznek relációs és logikai operátorok is. A relációsak a következők: >, >=, <, <=, =.

A logikai operátorok esetén a következőkről beszélünk: || (logikai vagy) illetve && (logikai és).

Inkrementáló és dekrementáló operátorokkal is találkoztunk C-s tanulmányaink során, ezek a következők: ++ és --. Előbbi operátor 1-et ad hozzá az operandushoz még utóbbi 1-et von ki belőle. Ezen operátorok állhatnak prefix- vagy postfix operátorként is, de vannak esetek mikor az egyik és vannak mikor a másik használata előnyösebb.

Bitenkénti logikai operátorok is vannak a C nyelvben, melyek a következők: & (bitenkénti és), | (bitenkénti megengedő vagy), ^ (bitenkénti kizáró vagy (XOR)), << (bitshiftelés balra), >> (bitshiftelés jobbra), ~ (egyes komplement)

Értékkadó operátorok is találhatók a nyelvben, ezekkel a könyv 58. oldalától kezdődően találkozhatunk.

A könyv 62. oldalán kezdődő precedenciabírálat sem egy elhanyagolható pontja a könyvnek, sőt, igencsak fontos dologról van szó. Az összes operátor precedencia- és kötési szabályait tartalmazza.

### 3. fejezet: Vezérlési szerkezetek

#### 3.1 Utasítások és blokkok

A C nyelvben a pontosvessző az utasításokat lezáró jel, vagyis ha egy kifejezés után pontosvesszőt teszünk, utasítássá válik. A kapcsos zárójelek segítségével a deklarációkat és utasításokat egy "csokorba", egy nagy blokkba lehet foglani, ami szintaktikailag egyetlen utasítással ekvivalens.

#### 3.2 Az if-else utasítás

Döntést, választást írunk le vele. A könyv 65. oldala alapján szintaxis a következő:

```
if (kifejezes)
 1. utasitas
else
 2. utasitas
```

Az else rész opcionális. Ha a zárójelben lévő kifejezés igaz, az első utasítás kerül végrehajtásra, más esetben (ha létezik) a második.

#### 3.3 Az else-if utasítás

A könyv példája alapján formai alakja:

```
if (kifejezes)
 utasitas
else if (kifejezes)
 utasítás
else if (kifejezes)
 utasítás
else
 utasítás
```

Többszörös elágazás. Ha valamelyik kifejezés igaz, a hozzá tartozó utasítás kerül végrehajtásra és lezárul a szerkezet. Az utolsó else az alapértelmezett esetet kezeli, ez csakúgy mint az előző if-else szintaxis esetén is elhagyható..

#### 3.4 A switch utasítás

Nagyon hasonló az if..else if..else szerkezethez, viszont olvashatóbb, szebb megjelenést ad programunknak, lásd a könyv szerinti 69. oldalon megjelenő mintakód alapján írt kis vázlatot:

```
switch (kifejezes)
case :
case :
case :
case :
case :
default :
```

A switch kiértékeli a zárójelek közti kifejezést és megvizsgálja az összes eset értékkel. Ha a kifejezés értéke megegyezik valamelyik case értékével, a case-hez kapcsolódó utasítás hajtódik végre, egyéb esetben az opcionális default eset. A szerkezetből break-kel lehet kilépni

### 3.5 A while és a for utasítás

A könyv szerinti példa:

```
while (kifejezes)
 utasítás
```

A szerkezetben a gép kiértékeli a kifejezést. Ha értéke nem nulla, akkor végrehajtja az utasítást és ismét kiértékeli a kifejezést. Ez addig megy így tovább, amíg a kifejezés 0 nem lesz, amikor is az utasítás után a végrehajtás véget ér. A

```
for (kifejezes1; kifejezes2; kifejezes3)
 utastas
```

alakú for utasítás egyenértékű a

A for bármely 3 kifejezése elhagyható. A for és a while ciklus között szabadon választhatunk de a for nyilvánvalóan előnyösebb olyankor, amikor egyszerű inicializálás és újrainicializálás fordul elő.

### 3.6 A do-while utasítás

```
do
 utasitas
 while (kifejezes);
```

A harmadik C-beli ciklusfajta, a do-while a kigrási feltételvizsgálatot a ciklus végén, a ciklustörzs végrehajtása után végzi el; ebből következik, hogy a törzs legalább egyszer mindenkorban végrehajtódik. A gép előbb végrehajtja az utasítást, majd kiértékeli a kifejezést. Addig hajtja végre az utasítást amíg a kifejezés értéke hamissá nem válik, ekkor a ciklus véget ér.

### 3.7 A break utasítás

A break utasítással már a ciklusbeli feltételvizsgálat előtt is ki lehet ugrani a ciklusokból és a switch-case szerkezetből is. A break utasítás eredményeképp a vezérlés azonnal kilép a legbelőző zárt ciklusból vagy éppen switchből..

### 3.8 A continue utasítás

A könyv 76. oldala alapján minta a használatra:

```
for (i = 0; i < N; i++) {
 if (a [i] != 0) /*Páros elemek átugrása*/
 continue;
 }
 . . .
/*Páratlan elemek feldolgozása*/
```

A continue utasítás a ciklus következő iterációjának megkezdését idézi elő.

### 3.9 A goto utasítás; címkék

A C-ben is lehet címkére ugrani a goto utasítás segítségével.

A 77. oldalon található példa erre:

```
for (. . .)
for (. . .) {
 .
 .
 if (zavar)
 goto hiba;
 .
}
hiba: számold fel a zavart
```

A címkék alakja ugyanaz, mint a változóneveké, azzal a különbséggel, hogy kettőspont követi nevüket.

## Utasítások

A legtöbb utasítással már találkoztunk a harmadik, Vezérlési szerkezetek c. fejezet, Utasítások és blokkok c. részfejezetében, ezért ezeket csak felsorolásszerűen említtem, azokat pedig amik nem jelentek meg eddig az olvasónaplóban, részletesebben jellemzem.

Utasítások fajtái:

- feltételes utasítás
- while utasítás
- do utasítás
- for utasítás
- switch utasítás
- break utasítás
- continue utasítás
- goto utasítás
- címkézett utasítás

A return utasítás

Több lehetséges alakja van, melyeket a könyv a 235. oldalon ezzel a példával szemléltet:

```
return;
return kifejezés;
```

Mely esetet nézve az először visszaadott érték határozatlan még a másodszor visszaadott a kifejezés értéke.

A nulla utasítás

Lényegében csak egy pontosvessző ami szolgálhat többek között üres ciklustörzset is képezhet.

A kifejezés utasítás

A legtöbb esetben ezek az utasítások vagy függvényhívások vagy pedig értékkedások. Formai alak: kifejezés;

Az összetett utasítás vagy blokk

A blokkok (kapcsos zárójelek által közbezárt részek a programban) lehetőséget adnak arra, hogy ott, ahol eredetileg egy utasítás volt elhelyezhető, ott most több utasítás legyen elhelyezhető de funkcionálitásukban működjenek úgy, mintha még mindig csak egy utasítás lennének.

## 10.3. Benedek Zoltán, Levendovszky Tihamér: Szoftverfejlesztés C++ nyelven - olvasónapló

### A C++ nem objektumorientált újdonságai

#### 2.1 A C és a C++ nyelv

A C nyelvtől eltérően ha egy függvényt üres paraméterlistával definiálunk nem tetszőleges számú paraméterrel hívható, hanem akkor az olyan, mintha egy void paraméterrel definiáltuk volna, erre látunk példát a 3. oldalon. C++-ban tetszőleges számú paramérrrel rendelkező függvény megvalósítása (4. oldal példája alapján):

```
void x(. . .)
{
}
```

#### 2.1 A main függvény

Két formája létezik C++-ban, mindenkorre példát látunk a 4. oldalon.

#### 2.1.3 A bool típus

A C++ nyelvben bezetésre került a bool típus is, ami logikai értéket hivatott tárolni, így búcsút lehet inteni az eddigi, intben tárolt logikai igaznak/hamisnak.

A C++-ban alapból, beépített típusként kaptak helyet a C-ben megismert, több-bájtos sztringek reprezentálását megvalósító wchar\_t típus. Szó esik még a változók deklarálásáról, azok hatóköréről is.

#### 2.2 Függvények túlterhelése

A C nyelvben egy függvényt egyértelműen a neve azonosítja. C++-ban ez más képp van, a név és az argumentumlista együttesen azonosít egy függvényt, ellenben a visszatérési értékkel, amivel egyértelműen nem azonosítható egy függvény. Jellemzésre kerül a linker névelferdítési technikája melyet azonos nevű függvények esetén alkalmaz. Szó esik az extern kulcsszó használatáról is, amit akkor kell használnunk ha egy C++ függvényt akarunk C-ben meghívni.

#### 2.3 Alapértelmezett függvényargumentumok

A C++ nyelv lehetőséget ad ehhez is, függvényeinknek meg lehet adni alapértelmezett értékeket is, így a hívás sokkal egyszerűbb is lehet. Példákon szemlélteti a könyv a 9. oldalon.

#### 2.4 Paraméterátadás referenciatípussal

C nyelvben csakis érték szerinti paraméterátadás lehetséges, ez a C++ nyelv esetén nem így van, megvalósítható a referenciával való paraméterátadás is. Ehhez csupán annyit kell tennünk, hogy a paraméterlistában mutatóra írjuk át az eddigi változót. Ez a cím szerinti paraméterátadás. Így működik C esetében. De a

C++ ilyen téren (is) megkönnyíti a dolgunkat hiszen bevezette a referenciatípust, melyet a változó elé tett & jellel tudunk létrehozni. Ezt részletezi a könyv a 12.-16. oldalakon, kódokkal szemléltetve.

## Objektumok és osztályok

### 3.1 Az objektumorientáltság alapelvei

A programok összetettségének növekedése miatt sokszor már lehetetlen volt a programkódokat kézben tartani. Megnövekedett az igény az átláthatóság iránt, emiatt terjedt el a '90-es években az objektumorientált programozás.

Fontos fogalom az egységbe zárás (enkapszuláció), az egységbe záró adatstruktúra neve osztály. Az osztálynak lehetnek "egyedei", ezek az objektumok. Az objektumok "védekező mechanizmusa" az adatrejtés, ekkor az objektum a program többi része számára nem teszi elérhetővé tartalmát. Megjelenik továbbá az öröklődés és az egységbe zárás fogalma is. Ezek az objektumorientált programozás alapelvei.

Fontos látni, hogy az éppen aktuális feladat dönti el minden, érdemes-e az OOP-t (Objektumorientált Programozás) használni.

### 3.2 Egységbe zárás a C++-ban

Átveszi a tagváltozók és tagfüggvények szerepét, használatát. A tagfüggvények esetében a függvény rendelkezik egy láthatatlan, első paraméterrel, ami alapján tudni fogja, melyik struktúrát kell módosítania. A `this` kulcsszó szerepe.

### 3.3 Adatrejtés

A `private` kulcsszó és szerepe: az utána álló változók és függvények csak osztályon belül láthatóak. Ellentéte a `public`, ami azt jelenti, hogy az adott tag struktúrán kívül is látható

A változó létrehozást osztályból példányosításnak hívjuk, a példány más néven objektum.

### 3.4 Konstruktorok és destruktork

A konstruktor speciális tagfüggvény, példányosításkor hívódik meg. A destruktur az objektumok által esetlegesen lefoglalt erőforrások felszabadításáért felel, automatikusan meghívódik mikor az objektum megsűnik.

### 3.5 Dinamikus adattagot tartalmazó osztályok

A dinamikus memória kezeléséért felelős operátor a `new`, a lefoglalt helyet a `delete` operátorral szabadíthatjuk fel. Tömbök esetén a `new []` és a `delete []` használhatók.

#### 3.5.3 Másoló konstruktor

Ismérve, hogy egy referenciát vár, aminek típusa megegyezik az osztály típusával. Esetében az újonnan létrehozott objektumot egy már létező objektum alapján inicializáljuk. Megjelenik a sekély- (shallow) és a mély (deep) másolás fogalma, ezek közti különbségek tisztázása.

### 3.6 Friend függvények és osztályok

#### 3.6.1 Friend függvények

A `friend` kulcsszó feljogosíthatnak globális függvényeket, illetve más osztályok függvényeit arra, hogy hozzáférhessenek az osztály védett tagjaihoz.

#### 3.6.1 Friend osztályok

Koncepciójukban már-már megegyeznek a `friend` függvényekkel. Az osztályunk egy másik osztályt jogosít fel a védett tagjaihoz való hozzáférésre. A friend osztály tagfüggvényei olyan hozzáférési jogokkal

rendelkeznek mintha alapból az adott osztály tagfüggvényei lennének. A friend tulajdonságról azonban tudni kell, hogy: nem öröklődik és nem tranzitív.

### 3.7 Tagváltozók inicializálása

Az inicializálás és az értékadás közti különböző részletezése. Az inicializálás konstruktorhívás történik, míg értékadás esetén az = operátor hívódik meg. Van lehetőség a tagváltozók inicializálására is, ezt az inicializálási listában tehetjük meg.

### 3.8 Statikus tagok

Olyan tagváltozók amik nem az objektumhoz hanem az osztályhoz tartoznak. A statikus tagfüggvények objektum nélkül, az osztály nevén keresztül is használhatók. Statikus függvények esetében nem használható a this mutató.

### 3.9 Beágyazott definíciók

A C++ nyelv lehetőséget ad az osztály-, struktúra-, típusdefiníciók osztályon belüli megadására. Ezt nevezzük beágyazott definícióknak (nested definition)

## Operátorok és túlterhelésük

### 6.1. Az operátorokról általában

Az operátorok az általunk számunkra átadott argumentumokon végezik el a műveletet majd visszatérési értékként kapjuk meg az eredményt. Itt jelenik meg egy eddig ismeretlen fogalom, a mellékhatás. Mellékhatásnak nevezzük azt a jelenséget, amikor az operátor az argumentum értékét is megváltoztatja, példának okáért ilyen a ++ operátor. A zárójel használata fontos bonyolultabb kifejezések esetén.

### 6.2 Függvényszintaxis és túlterhelés

A C nyelvben nem tudunk olyan függvényt létrehozni, ami képes lenne az előbb említett mellékhatásra, csak ha a változóra mutatót adunk át. C++-ban ez lehetséges, a referencia szerinti paraméterátadással. Ehhez a kódot a 94. oldalon találhatjuk, én is az alapján írtam a következő csipetet:

```
int noveles(int& ertek)
{
 int eredmény = ertek;
 ertek = ertek+1; //mellékhatás
 return eredmény;
}
```

C++-ban az operátorok függvényszintaxissal való meghívására is van lehetőség, ez esetben ez így néz ki:

```
z = x+y;
z = operator+ (x, y);
```

A fenti 2 sor ekvivalens. Ez a módszer azonban csak a felhasználó által definiált típusokra, beépítettekre nem használható. Lehetőségünk van akár az operátorok túlterhelésére is, erre láthatunk példát a könyv 95. oldalán

## C++ sablonok

Léteznek osztály- és függvény sablonok egyaránt, amik esetében néhány elemüket definiáláskor paraméternek veszünk. Gyakori az alkalmazásuk olyan tárolóosztályok létrehozásánál, amik tetszőleges típusú elemek tárolására használhatóak fel.

## 11. 1 C++ függvény sablonok

A következő függvényt sablonná alakítjuk át, amelyben a típus amivel dolgozik, nem rögzített, a sablon felhasználásakor adjuk meg. A könyvben található példakód alapján írt kód:

```
template <class N> inline N max (N lhs, N rhs)
{
 return lhs > rhs ? lhs: rhs;
}
```

Felhasználása:

```
int y = max(3, 5);
double z = max(3.1, 5.5);
```

### 11.1.2 Példák függvény sablonokra

Érdemes tudni, hogy sablonparaméter nem csak típus, hanem típusos konstans is lehet, pl.:

```
template <int N> int Square()
{
 return N*N;
}
int main()
{
 const int x= 10;
 cout <<x<<"négyzete:" <<Square<x>() << endl;
}
```

### 11.1.3 A hívott függvény kiválasztása

Ha az adott függvénynek több implementációja van jelen, annak kiválasztásakor szabályok döntik el, melyik függvény kerül meghívásra.

## 11.2.1 Osztálysablonok írása

1. lépés: Osztálydefinícióból sablondefiníció

2. lépés: Sablonparaméterek bevezetése

3. lépés: Nem implicit inline definiált tagfüggvények szintaktikája

4. lépés: Osztálynév helyett osztálynév ÉS sablonparaméterek szerepeltetése a stípusként való felhasználás során

## A C++ I/O alapjai

### 5.1 A szabványos adatfolyamok

A C nyelvben 3 fájlleíró áll rendelkezésünkre: az `stdin` (szabványos bemenet), a `stdout` (szabványos kimenet) és az `stderr` (szabványos hibakimenet), ezek mindegyike `FILE*` típusú, magas szintű állományleírók.

Ezzel ellentétben a C++ nyelv adatfolyamokban (bájtok sorozata) gondolkodik. Az `istream` típusú objektumok alatt csak olvasható adatfolyamokat értünk, `ostream` típusú alatt pedig csak írhatókat. Ezek az adatfolyamok az `<<` és `>>` operátorokat használják a beolvasásra és kiírásra. Érdekesség, hogy ezek az operátorok mindenkor olyan irányba mutatnak, ami megfelel az adatáramlás irányának.

Ahhoz, hogy használni tudjuk az I/O-t C++-ban, az `iostream` állományt inkludálnunk. Gyakorlati példákon keresztül mutatja be a könyv többek között a 3 paraméterrel rendelkező `ignore` függvényt és a bufferek ürítéséhez használatos `flush` függvényt. Mivel az adatfolyam egy objektum, rendelkezik állapotát (`iostate` típusú tagváltozó) beállító konstansok: `eofbit`, `failbit`, `badbit`, `goodbit`. Az első 3 minden különböző hibát jelez vissza, az utolsó a helyes működésről tájékoztat. A C és C++-beli, beolvasást megvalósító függvények összehasonlítása is megtörténik a 79. oldalon. Ugyanezen az oldalon a `getline` függvény is említésre kerül, amit stringek beolvasásakor használunk arra, hogy a beolvasás ne szakadon meg az első szóköznél. Az előzőhöz hasonló táblázattal találkozunk a kiiratás kapcsán is a 80. oldalon.

## 5.2 Manipulátorok és formázás

A manipulátor egy speciális objektum, amelyet az adatfolyamokra alkamazunk a bemeneti- és kiviteli operátorok argumentumaként. Az előre definiált manipulátorok az `iomanip` fájlban találhatók amiket ha használni akarunk, programunkba inkludálnunk kell. Néhány példa az egyes manipulátorokra: `endl`, `flush`, `noskipws`, `setw`. Jelzőből és maszk fogalmak megjelenése. A leggyakoribb manipulátorok és tagfüggvényeik összefoglaló táblázata a 84-85. oldalon található meg.

## 5.3 Állománykezelés

A C++ programnyelv az állománykezeléshez is adatfolyamokat használ, amiket az `ifstream` (input file stream - bemeneti állomány adatfolyam), az `ofstream` (output file stream - kimeneti állomány adatfolyam) és az `fstream` (kétirányú adatfolyam) osztályok reprezentálnak. Az állományok megnyitását kontrollátorok, még bezárását a destruktörök végezik, hiszen az állományok objektumként vannak kezelve. Ez biztonságosabbá teszi a fájlkezelést. Egy-egy fájl megnyitására többféle mód is létezik, ezekre is példát ad a 87. oldalon található összefoglaló táblázat. Jelzőbit fogalom újból megjelenése, példákat a 89. oldalon találhatunk rájuk.

## Kivitelkezelés

A hagyományos hibakezelésre mutat a könyv példát egy szemléletes kóddal a 188-189. oldalon. A kódban a hibákódok számára egy külön osztály került létrehozásra melyben a kódok definiálása történt meg. Ez több szempontból sem előnyös megoldás, a könyv szerzői meg is indokolják miért. Az előnyös megoldás a kivitelkezelés, amely mechanizmusa lehetővé teszi számunkra azt, hogy kivétel esetén a vezérlés a kivitelkezelőhöz kerüljön. A 190. oldalon található egy remek, szemléletes példa ahhoz, hogy tisztuljon a kép a kivitelkezeléssel kapcsolatosan. A main-ben található try-catch szerkezet try részébe a helyes működés-míg a catch részébe a kivitelkezelő kódja kerül, további és részletesebb magyarázat a 191. oldalon. További jó kódokkal találkozhatunk a következő, 192. oldalon ahol szintén egy try-catch szerkezzel találkozhatunk, azonban az előző példától eltérően most több catch-ággal. Ezen feladattal kapcsolatos példákon keresztül kerül szemléltetésre a következő oldalakon az, hogy a catch blokk csak akkor képes elkapni a throw-t, hogyha paramétereik megegyeznek.

Lehetőségünk van arra is, hogy a try-catch szerkezetek egymásba ágyazásával a kivételeket alacsonyabb szinten kezeljük. Szó esik a catch által elkapott jel újradobásáról is, amit a `throw` parancsal tudunk kivitelezni. Fontos fogalom még a hívási verem visszacsépélése is. Azt a folyamatot hívjuk így, amikor egy kivétel dobásakor annak elkapásáig a függvények hívási láncában felfelé haladva az egyes függvényes lokális változói felszabadulnak, erre láthatunk példát a 197. oldalon lévő kódban az `f1` és `f2` függvények esetében, `f2` kivéttel dob, ebben a függvényben a változó felszabadul, majd `f1` függvényben is felszabadul a változó (jelen esetben objektum) hiszen ez a függvény is a hívási lánc tagja, majd végül a main-ben található catch paraméterei egyeztethetők meg az `f2`-ben lévő throw-éval, tehát itt fut le a catch blokk.

## 10.4. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba

A Python programozási nyelv történeti hátterével kezdi a könyv. A '90-es években került megalkotásra a mai napig, elsősorban prototípusok elkészítésére népszerű, objektumorientált és platformfüggetlen nyelv. Viszonylag könnyen tanulható nyelvként hivatkozik rá az író. A nyelv érdekessége, hogy az eddig tanult C, C++ vagy akár Java nyelvekkel ellentétben nincs szükség fordításra, az interpreter elboldogul csupán a forrással.

A könyv a nyelv bemutatását az alapvető szintaxissal kezdi. Ebből a pár sorból megtudhatjuk, hogy behúzásalapú a szintaxisa vagyis itt elfelejthető az eddigi tanulmányaink során használt kapcsos-zárójelezés, sőt mi több, az utasítások a sor végéig tartanak, így az utasításokat eddig lezáró pontosvesszőt sem használja a nyelv. Példával kerül szemléltetésre, hogyan is néz ki mindez a gyakorlatban. Megkapjuk a kulcsszavak listáját is és megtudjuk, hogy a megjegyzések a kettőskereszt karakterrel jelölhetjük.

A nyelvben minden adatot objektum reprezentál. Érdekesség, hogy nincs szükség egy változó típusának konkrét megadására, van olyan okos és fejlett a rendszer, hogy kitalálja, "mire gondolt a költő". Az adattípusok egyébként a következők lehetnek: számok, sztringek, ennek, listák, szótárak, melyek közül az utolsó 2-ről még nem is hallottam. Mindegyikre mutat gyakorlati példát is a könyv egy pár soros kódcsipetben. A változók a Pythonban az egyes objektumokra mutató referenciai, értékük a már megszokott egyenlőségjellel történik. A "del" kulcsszó szerepéit is taglalja az olvasmány, ezzel változó hozzárendelést törölhetünk. A már előbb említett adattípusok, pontosabban a listákkal és szótárakkal végezhető műveleteket is megtalálhatjuk több táblázatban.

Ezután a nyelv eszközei rész következik a könyvben, ami megleíti a `print` metódust, mellyel a standard kimenetre írhatunk tetszőleges változókat, szöveget. Természetesen elágazásokkal is operál a nyelv, csak úgy, mint ciklusokkal. Természetesen a `for` és a `while` függvények itt is megtalálhatók. Említésre kerül a `range` és az `xrange` függvények is, amelyek két, általunk megadott alsó- és felső határ közötti értékeket sorol fel alapvetően 0-tól kezdődően, megadott lépésközzel.

Létrehozhatunk címeket is melyekre a `goto` parancsral ugorhatunk. A `comefrom` az előző parancs ellentéte. A függvények definiálása a `def` kulcsszóval történik.

A Python lehetőséget ad az objektumorientált fejlesztési eljárások használatára is, ezek is példákon keresztül kerülnek szemléltetésre.

Mivel a Python a mobiltelefonra való fejlesztői munka (egyik) éollovása, modulok kerültek létrehozásra annak érdekében, hogy ez a munka könnyebb legyen. Ilyen modulok például: `camera` amely segítségével a készülék kamerájához férhetünk hozzá, `audio`, ami felelős a hangfelvételekért, `messaging`, ami többek között az SMS-üzenetekért felel vagy épp a `sysinfo`, ami a telefonnal kapcsolatos infók lekérdezését teszi lehetővé.

A nemkívánatos esetek előfordulásakor a kivételkezelés lép életbe. A Java-val és C/C++ -szal ellentétben itt nem `try-catch`, hanem `try-except` szerkezetről beszélhetünk.

A fejezet az előbb bemutatott dolgokat mutatja be részletesebben, egy-egy szemléletes példán keresztül.

A könyvet (részletét) alapvetően korrektnek tartottam, érthetően magyarázza a dolgokat. Összességében tetszett, és örülök, hogy egy újabb (szimpatikus) programozási nyelvbe nyerhettem ezzel egy kis betekintést.

## 10.5. Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II.

### 1. fejezet - Nyolc lap alatt a nyelv körül

Kezdjük a legelején. A Java a C++ nyelvhez hasonlóan egy objektumorientált nyelv. Lényegében ez nem jelent más, minthogy egy Java nyelvben írt program objektumokból és a "tervrajzok" szerepét betöltő osztályokból áll. Az osztályok pedig változókból és metódusokból állnak. A C++ esetében is ugyanez a helyzet.

A könyv szerzői egy egyszerű Hello, World!-ön keresztül mutatják be a Java fordítójának működését. Ebből a szemponból eltér a C++ és a Java, ugyanis utóbbi esetében az ún. Java Virtuális Gép (interpreter) fogja értelmezni a fordítóprogram által előzőleg kreált ún. bajtkodót.

A Java nyelv sikerét alapvetően a világhálón fellelhető oldalakon betöltött funkcióinak köszönheti. Ez alapvetően számomra újdonság, a C++ során nem találkoztam hasonlóval. Egy HTML oldalba beépített Java programot hoz példaként a könyv, mely programokat gyűjtőnél appleteknek nevezzük. Az applet forráskódjában megjelenik az `import` kulcsszó (ezzel eddig tanulmányaink során is találkoztunk már), amivel a feladat végrehajtásához szükséges könyvtárakat teszünk elérhetővé számunkra.

A következő 3 alfejezet rendre a változókkal, konstansokkal és a megjegyzésekkel foglalkozik. (ebben a sorrendben) A változók típusa lehet: boolean (logikai true vagy false értékkel), char (16 bites Unicode karakter), byte (8 bites előjeles egész szám), short (16 bites előjeles egész szám), int (32 bites előjeles egész szám), long (64 bites előjeles egész szám), float (32 bites lebegőpontos racionális szám), double (64 bites lebegőpontos racionális szám). A konstansok szerepe ugyanaz, mint C++ esetén, megadásuk a `final` kulcsszóval történik. A megjegyzések Java esetén is lehetnek egy- vagy akár többsorosak, sőt dokumentációsak is, utóbbi. Mindegyikre hoz példát a könyv.

Java nyelvben a `class` kulcsszóval hozható létre osztály. Tetszőleges sorrendben felsorolhatók az adattagai illetve metódusai, amik az eddigi tanulmányaink során tanult változóknak és függvényeknek nevezhetünk lényegében. Az adattagok és metódusok láthatóságát egyenként lehet beállítani. A könyv által hozott példában megjelenik egy új típus, melynek neve `String`. Igen, a Java-ban a karakterláncok kezelésére létre lett hozva egy új osztály, ez merőben eltér az eddig használt és megszokott (akár) C++-os megoldástól, ahol a `String` típus lényegében karakterek tömbje volt. Ha van osztályunk, természetesen objektumot is létre hozhatunk belőle, mindez a `new` operátor segítségével. Ez az operátor helyet foglal le az új objektum számára és erre a területre vonatkozó referenciaival tér vissza. Megjelenik a `static` kulcsszó szerepe is, ha ezzel deklarálunk elemeket, akkor azok az elemek magához az osztályhoz fognak tartozni. Ez utóbbi is újdonság az eddigi tanulmányainkhoz képest.

Nyilvánvalóan a Java nyelvben is kifejezetten fontos, hogy egyes, nem elvárt helyzetben hogyan reagáljon a program. Ez jellemzően `try-catch` szerkezetben valósul meg, amelyet most nem ragozok, már találkoztunk vele tanulmányaink során, ugyanígy működik C++ alatt is. Ez hivatalosabban fogalmazva nem más, mint a kivételkezelés.

Érdemes még pár szót ejteni a nyelv biztonsági lehetőségeiről, amely rugalmasan az adott igényekhez igazítható. Segítségével meghatározható, hogy egy adott kód részlet mihéz is férhez hozzá.

Az AWT (Abstract Window Toolkit) az adott program felhasználói grafikai felület megvalósításában van nagy segítségünkre. Ha már grafika, érdemes megemlíteni a Swing könyvtárat is amely az AWT-hez hasonló, azonban sokkal gazdagabb lehetőségeket kínál a programozó számára.

### 2. fejezet - Az alapok

A legtöbb számítógépen két karakterkészlet terjedt el az idők során: az ASCII illetve az EBCDIC. Mindkettő 8 bites mérettel rendelkeznek, ezért többek között a magyar nyelv különleges karakterei (pl.ő, ú) nem jeleníthetők meg. Erre nyújt megoldást az Unicode karakterkészlet, ami több, mint 8 biten tárolja karaktereit, ezzel lebontva az előző karakterkészletek korlátait. A pozitívum az, hogy a Java forráskódjában bármilyen tetszőleges Unicode karakterek szerepelhetnek.

Ami az azonosítókat illeti: a Java nyelvben az azonosítóknak betűvel kell kezdődniük és betűvel vagy számmal kell, hogy folytatódjanak. Érdekesség, hogy a betűk nem csak az angol-, hanem akár a hindu karakterkészletből is érkezhetnek. Az azonosítók hossza tetszőleges, azonban vannak olyan szavak, amelyek ilyen célra fel nem használhatók, ugyanis a nyelv kulcsszavai.

A nyelv 8 db. primitív típussal rendelkezik, melyeket előzőleg már ismertettem. A primitív változók magukat az értékeket tárolják, míg a változók másik típusa (referenciatípus) objektumhivatkozásokat tartalmaznak, amely így azt írja le, hogy az adott érték hol lelhető fel a memóriában.

A literálokról annyit érdemes megjegyezni, hogy egyszerű típusok és objektumok inicializálásához használatosak. Többek között használjuk őket speciális karakterek (escape karakterek) megadásához is.

Egy változó deklarációjához szükségünk van legalább egy típusra és egy változónévre. A változókhöz az egyenlőségjel opeátorral rendelhető kezdeti érték ami akár már deklarációkor is megtehető. Nyilván a csak deklárált de értékkel még nem rendelkező változók alapértelmezett értékkel rendelkeznek.

A nyelvben a [] jelöléssel lehet egy tömb típust megadni, ami érdekesség, hogy ez tényleges típus, nem úgy mint C++-ban, ahol ugyanis csak a mutató típus egy megjelenési formája volt. Indexelése 0-val kezdődik.

Beszélhetünk felsorolás típusokról is, melyeket az enum kulcsszóval vezethetünk be, kapcsos zárójelek között a megfelelő értékeket feltüntetve. Az indexelés itt is 0-val kezdődik.

Az operátorok természetesen csakúgy, mint a legtöbb programozási nyelvben, így Java-ban is nagy jelentőséggel bírnak a kiértékelés szempontjából, ugyanis először a legbelő zárójelen belüli részkifejezés kerül kiértékelésre, ha nincs zárójel, akkor először a nagyobb prioritású operátor lesz végrehajtva; ha több operátor prioritása egyenlő, akkor balról jobbra, illetve néhány esetben jobbról balra lesznek azok kiértékelve. Beszélhetünk többek között postfix- vagy akár prefix operátorokról is.

A Java mivel egy erősen típusos nyelv, ezért a kifejezésekben szinte minden esetben megvizsgálja, hogy az automatikus konverziót végrehajtva azonos típusra alakíthatók-e az elemek. Ha az ellenőrzési fordítási időben elvégezhető, akkor hiba esetén a fordítás megszakad. Ha egy konverziós hiba csak futási időben deríthető ki, akkor hibaüzenetet a futtató rendszer generál. Létezik ezen kívül viszont explicit konverzió is, ez azonban általában nem biztonságos, adatvesztéssel járhat. Szövegkonverzióról is beszélhetünk.

Egy struktúra részelemének eléréséhez a . (pont)-tal minősített név használható.

### 3. fejezet - A vezérlés

Ha az utasítás szóval kerülünk szembe, alapvetően két fajtáról beszélhetünk. Az egyik ilyen utasításfajta a kifejezés-utasítás míg a másik a deklarációs-utasítás. Közös jellemzőjük, hogy minden pontossággal zárja le. Kifejezés-utasítás a következő kifejezésfajtból képezhető, például: értékkedás, postfix illetve prefix operátorokkal képzett kifejezések, metódushívások vagy példányosítás. A deklaráció-utasítás egy lokális változó létrehozását jelenti. Utasítások sorozatát {} jelek között blokknak nevezzük.

Természetesen az elágazások fellelhetők Java-ban is. Megkülönböztetünk egyszerű- és összetett elágazásokat. Előző alatt az if-szerkezetet, utóbbi alatt pedig a switch-szerkezetet értjük.

Négyféle ciklust ismer a Java nyelv: előtesztelőt, hátutesztelőt, léptetőt és bejáró ciklust. Ezek szintén mindegyikével találkoztunk már C/C++-os tanulmányaink során, talán a bejáró ciklust érdemes viszon megemlíteni.

A bejáró ciklus a Java 5-ös verziójában jelent meg és egy adatszerkezet bejárását végezhetjük vele, mely adatszerkezet lehet tömb, sorozat vagy akár halmaz is.

Egy egészket tartalmazó tömb bejárása így néz ki a használatával:

```
int [] tomb = {1, 5, 7, 9, 31, 87};
int osszeg = 0;
for(int n: tomb)
 osszeg += n;
```

A Java címkéiről csak ennyit: bármely utasítás elő írható címke, mely az utasítás egyértelmű azonosítását teszi lehetővé feltétlen vezérlésátadás érdekében. Formai kinézete: címke: utasítás.

A most következő pár sor sem lesz ismeretlen számunkra. A bereak utasítás egy blokkból való kilépésre szolgál. Formája: break [címke].

A continue utasítással egy ciklus magjának hátralevő részét lehet átugrani. Kinézete: continue [címke];

Egy metódussal a return utasítással lehet visszatérni. A metódus visszatérési értéke így a return után írt kifejezés értéke lesz.

És most essen pár szó a goto-ról! Többek között a C++-szal ellentétben a Java-ban már nincsen goto utasítás. Helyette alternatív megoldásokat kínál a Java.

#### 4. fejezet - Osztályok

A Java-ban írt programok legkisebb önálló egységei az osztályok. Egy osztály olyan, mint egy tervrajz, azonos típusú "dolgok" modelljét írja le. A program a működése során példányosítja az osztályokat vagyis konkrét példányokat hoz létre a modellek sémaja szerint. Az osztályokat két részből írják le, ez az osztálydefiníció. Az egyik rész deklarálja a változókat, míg a másik az objektum viselkedését meghatározó metódusokat tartalmazza.

Az osztály változóit változódéklarációk adják meg. Egy dékláció egy vagy több azonos típusú változót vezet be. A dékláció a változó típusával kezdődik majd mögötte vesszővel elválasztva változó nevek állnak. Kezdőérték megadása is lehetséges.

Egy osztály metódusait metódusdefiníciók írják le. Egy definíció fejből és törzsből áll. A fej a metódus visszatérési típusát, azonosítóját és formális paramétereit tartalmazza. Első 2 alkotja a metódus szignatúráját. A metódus törzsében a működést definiáló utasításblokk található. A törzsben használható a this ún. pszeudováltozó, amivel az aktuális példányra hivatkozhatunk.

A metódushívásban meg kell adni a metódus definíciójában előírt számú és típusú paramétert.

## **III. rész**

### **Második felvonás**

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

# 11. fejezet

## Helló, Arroway!

### 11.1. OO szemlélet

A módosított polártranszformációs normális generátor beprogramozása Java nyelven. Mutassunk rá, hogy a mi természetes saját megoldásunk (az algoritmus egyszerre két normálist állít elő, kell egy példánytag, amely a nem visszaadottat tárolja és egy logikai tag, hogy van-e tárolt vagy futtatni kell az algot.) és az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua.!

[https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1\\_5.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_5.pdf) (16-22 fólia)

Ugynéz írjuk meg C++ nyelven is! (lásd még UDPROG repó: source/labor/polargen)

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/arroway\\_codes/OO](https://gitlab.com/kincsa/bhax/tree/master/codes/arroway_codes/OO)

Tanulságok, tapasztalatok, magyarázat...

A megoldás Java-ban:

```
public class PolárGenerátor {

 boolean nincsTárolt = true;
 double tárolt;

 public PolárGenerátor() {
 nincsTárolt = true;
 }
}
```

Programunk a fent látható módon indul. A PolárGenerátor publikus osztályban zajlik maga az algoritmus elkészítése. Itt kerül deklarálásra a boolean típusú nincsTárolt változó is, ami a feladat által kért logikai tag, mely azt jelzi majd nekünk, hogy "van vagy nincs eltéve kiszámolt szám.

```
public double következő() {

 if(nincsTárolt) {

 double u1, u2, v1, v2, w;
 do {
 }
```

```
 u1 = Math.random();
 u2 = Math.random();

 v1 = 2*u1 - 1;
 v2 = 2*u2 - 1;

 w = v1*v1 + v2*v2;

 } while(w > 1);

 double r = Math.sqrt((-2*Math.log(w))/w);

 tárolt = r*v2;
 nincsTárolt = !nincsTárolt;

 return r*v1;

}
else {
 nincsTárolt = !nincsTárolt;
 return tárolt;
}
}
```

Még mindig az eredeti osztályon belül járunk, itt deklarálásra kerül a double típust visszaadó, következő névre hallgató függvény. Ha a nincsTárolt változó igaz, egy do-while ciklus keretein belül, amíg w nagyobb, mint 1: u1 és u2 értékül kap egy véletlen számot a Math.random függvény álltal, v1 és v2 pedig az előző változókból kapja meg az értékét, ugyanez igaz lesz w-re is. Kiszámolásra kerül a r változó a kódcsipetben látható összefüggés alapján (a Math osztály függvényei segítségével) és a tárolt változó is értéket kap, ezzel együtt természetesen a nincsTárolt változó hamis lesz, ugyanis már lesz egy tárolt elemünk.

Ellenkező esetben, ha a függvény elején lévő feltételvizsgálat hamis eredményt ad vissza: lefut az else-ág, amelyben a már eltárolt elemet adja vissza a függvény.

A megoldás C++-ban:

```
public static void main(String[] args) {

 PolárGenerátor g = new PolárGenerátor();

 for(int i=0; i<10; ++i)
 System.out.println(g.következő());
}
```

A main-ben már csak a példányosítás és a függvény meghívása történik.

```
#include <cstdlib>
#include <cmath>
#include <ctime>
```

```
#include <iostream>

using namespace std;

class PolarGenerator
{
public:
 PolarGenerator ()
 {
 nincsTarolt = true;
 srand(time(NULL));
 }

 ~PolarGenerator ()
 {

 }

 double kovetkezo ()
 {
 if (nincsTarolt)
 {
 double u1, u2, v1, v2, w;
 do
 {
 u1 = rand () / (RAND_MAX + 1.0);
 u2 = rand () / (RAND_MAX + 1.0);
 v1 = 2*u1 - 1;
 v2 = 2*u2 - 1;
 w = v1*v1 + v2*v2;
 }
 while (w > 1);

 double r = sqrt ((-2 * log (w)) / w);

 tarolt = r * v2;
 nincsTarolt = !nincsTarolt;

 return r * v1;
 }
 else
 {
 nincsTarolt = !nincsTarolt;
 return tarolt;
 }
 }

 bool nincsTarolt=true;
 double tarolt;
```

```
};

int main (int argc, char **argv)
{
 PolarGenerator polargen;

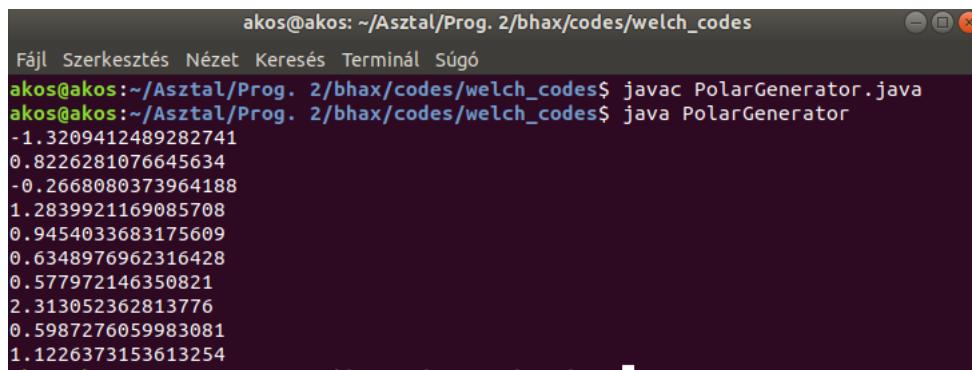
 for (int i = 0; i < 10; ++i)
 std::cout << polargen.kovetkezo () << "\n";

 return 0;
}
```

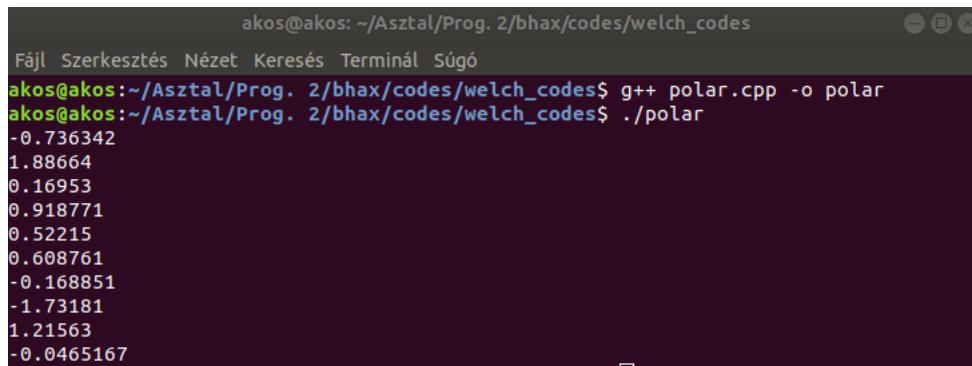
A PolarGenerator osztalyon belül létrehozásra kerül a konstruktor és a destruktur egyaránt. Előző osztályból való objektum létrehozásához szükséges még utóbbi annak a törléséhez. Itt foglal helyet a kovetkezo függvény is, illetve a nincsTarolt és tarolt változóink is. Igazi különbség talán csak a véletlen számok sorsolásánál figyelhető meg a Java verzióhoz képest. A RAND\_MAX konstans a rand függvény által sorsolható véletlen számok felső határát rögzíti.

Az algoritmus itt is ugyanaz, mint az előbb. Amint azt az ábra mutatja, a C++-os megoldás nagyon könnyedén megírható a Java kódunkból. Természetesen ez igaz fordítva is, azonban mivel mi először a Java verzióval rendelkeztünk, így egyértelmű, hogy az kerül átirásra C++-ra.

A kódok fordítása és futtatása után a következő eredményt kapjuk:



```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/welch_codes
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/welch_codes$ javac PolarGenerator.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/welch_codes$ java PolarGenerator
-1.3209412489282741
0.8226281076645634
-0.2668080373964188
1.2839921169085708
0.9454033683175609
0.6348976962316428
0.577972146350821
2.313052362813776
0.5987276059983081
1.1226373153613254
```



```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/welch_codes
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/welch_codes$ g++ polar.cpp -o polar
akos@akos:~/Asztal/Prog. 2/bhax/codes/welch_codes$./polar
-0.736342
1.88664
0.16953
0.918771
0.52215
0.608761
-0.168851
-1.73181
1.21563
-0.0465167
```

Miért fontos a fenti kódokat olvasni? Mert az objektumorientáltság egyik alapelve, az [egysébe zárásra](#), ezen felül pedig még példányosításra is láthatunk benne példát.

És most vessünk egy pillantást az OpenJDK Random.java állományra. Lássuk, más programozók milyen elven gondolkodtak! Kis keresés után a következőket találhatjuk benne:

```
private double nextNextGaussian;
private boolean haveNextNextGaussian = false;

public synchronized double nextGaussian() {
 // See Knuth, ACP, Section 3.4.1 Algorithm C.
 if (haveNextNextGaussian) {
 haveNextNextGaussian = false;
 return nextNextGaussian;
 } else {
 double v1, v2, s;
 do {
 v1 = 2 * nextDouble() - 1; // between -1 and 1
 v2 = 2 * nextDouble() - 1; // between -1 and 1
 s = v1 * v1 + v2 * v2;
 } while (s >= 1 || s == 0);
 double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
 nextNextGaussian = v2 * multiplier;
 haveNextNextGaussian = true;
 return v1 * multiplier;
 }
}
```

És hogy mit igazol ez a pár sor? Azt, hogy a JDK programozói lényegében ugyanúgy gondolkodtak, mint mi, hiszen az Ő megoldásuk és a miénk nem egy helyen mutat azonosságot.

Mit figyelhetünk még meg? A lényeges különbséget az adja, hogy a JDK forrásban találkozunk a `synchronized` kulcsszóval a metódus visszatérési típusa előtt, aminek eredményeképp a metódus szinkronizált lesz.

Mi ennek a szerepe? Ennek akkor van jelentősége, ha több szál akarja meghívni ugyanazt a `synchronized` kulcsszóval ellátott metódust, ilyenkor ugyanis addig, míg az első szál be nem fejezte azzal az adott objektumra való metódushívást, a többi szálnak várakoznia kell.

## 11.2. „Gagyí”

Az ismert formális 2

```
„while (x <= t && x >= t && t != x);”
```

tesztkérdéstípusra adj a szokásosnál (miszerint x, t az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciája) „mélyebb” választ, írj Java példaprogramot mely egyszer végtelen ciklus, más x, t értékekkel meg nem! A példát építsd a JDK Integer.java forrására, hogy a 128-nál inkluzív objektum példányokat poolozza!

Megoldás videó:

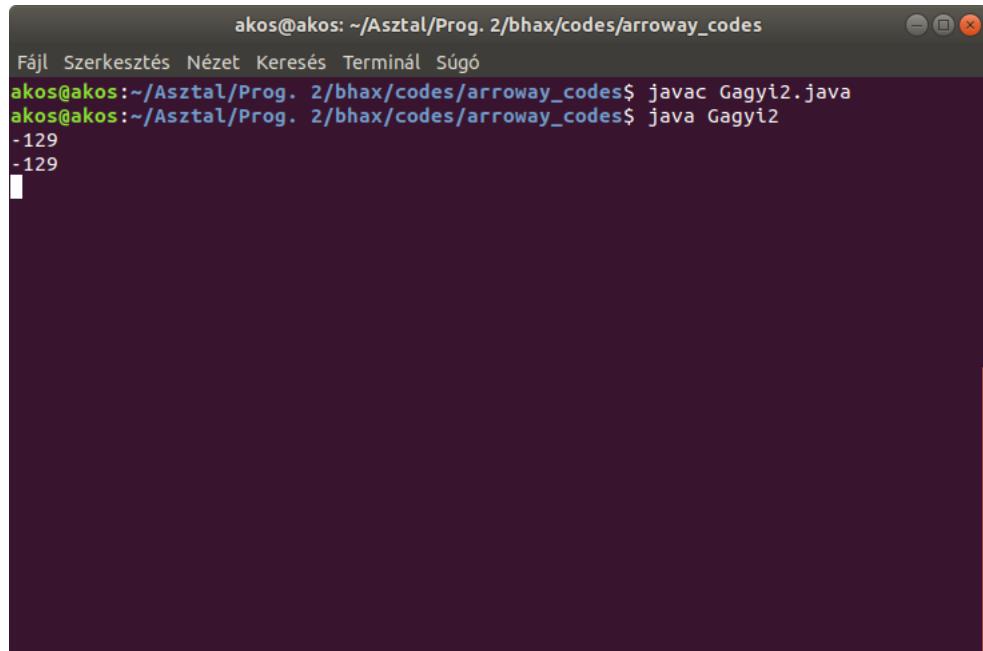
Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/arroway\\_codes/Gagyí](https://gitlab.com/kincsa/bhax/tree/master/codes/arroway_codes/Gagyí)

Tanulságok, tapasztalatok, magyarázat...

A feladat megoldásához kiindulásképp igénybe vettetem a lábjegyzetben található facebook-posztot, amely elolvasása után már sokkal jobban át tudtam látni a feladat lényegét. Az ott talált két példaprogram, majd futtatásuk eredménye:

```
public class Gagyi2
{
 public static void main (String[]args)
 {
 Integer x = -129;
 Integer t = -129;
 System.out.println (x);
 System.out.println (t);

 while (x <= t && x >= t && t != x);
 }
}
```



A screenshot of a terminal window titled "akos@akos: ~/Asztal/Prog. 2/bhax/codes/arroway\_codes". The terminal shows the following command-line session:

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$ javac Gagyi2.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$ java Gagyi2
-129
-129
```

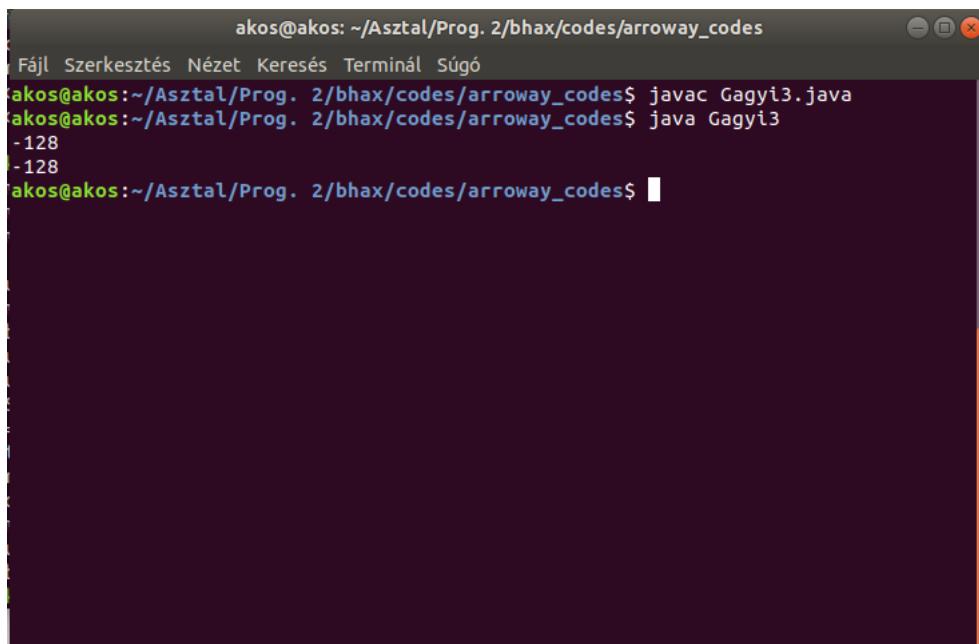
Tehát ez esetben végtelen ciklusba kerülünk.

A második kód esetében:

```
public class Gagyi3
{
 public static void main (String[]args)
 {
 Integer x = -128;
 Integer t = -128;
 System.out.println (x);
 System.out.println (t);

 while (x <= t && x >= t && t != x);
 }
}
```

Immáron nincs végtelen ciklus.



A terminal window titled 'akos@akos: ~/Asztal/Prog. 2/bhax/codes/arroway\_codes'. The window shows the following command-line interaction:

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$ javac Gagyi3.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$ java Gagyi3
-128
-128
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$
```

A feladat megoldásához ahogy az fentebb meg lett említve, meg kell vizsgálnunk a JDK Integer.java forrást! Vessünk egy pillantást a minket érintő sorokra! Többek között a következőt látjuk:

```
public static Integer valueOf(int i) {
 if (i >= IntegerCache.low && i <= IntegerCache.high)
 return IntegerCache.cache[i + (-IntegerCache.low)];
 return new Integer(i);
}
```

Itt olvasható a lényeg. Kezdjük először a sorrendet kicsit felborítva, a -128 példájával. A -128 még az előző forrásban tárgyalt valueOf metódus definíciójában található intervallumon belül található, ezért egy úgynevezett poolból kapjuk meg az Integer objektumokat, amelyek egy és ugyanazon memóriacímre hivatkoznak. Emiatt nem teljesül a feltétel és nem kerülünk végtelen ciklusba.

A -129-es esetnél már más a helyzet, hiszen ez az érték már kívül esik az előbb említett intervallumon. Ekkor a

```
return new Integer(i)
```

sor fog lefutni. Mit eredményez ez? Két különböző Integer objektum jön létre különböző címekkel, így a feltétel mindenkor teljesülni fog, ezért végtelen ciklusba fut bele a programunk.

### 11.3. Yoda

Írunk olyan Java programot, ami java.lang.NullPointerException-t leáll, ha nem követjük a Yoda conditions-t!

[https://en.wikipedia.org/wiki/Yoda\\_conditions](https://en.wikipedia.org/wiki/Yoda_conditions)

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/arroway\\_codes/Yoda](https://gitlab.com/kincsa/bhax/tree/master/codes/arroway_codes/Yoda)

Tanulságok, tapasztalatok, magyarázat...

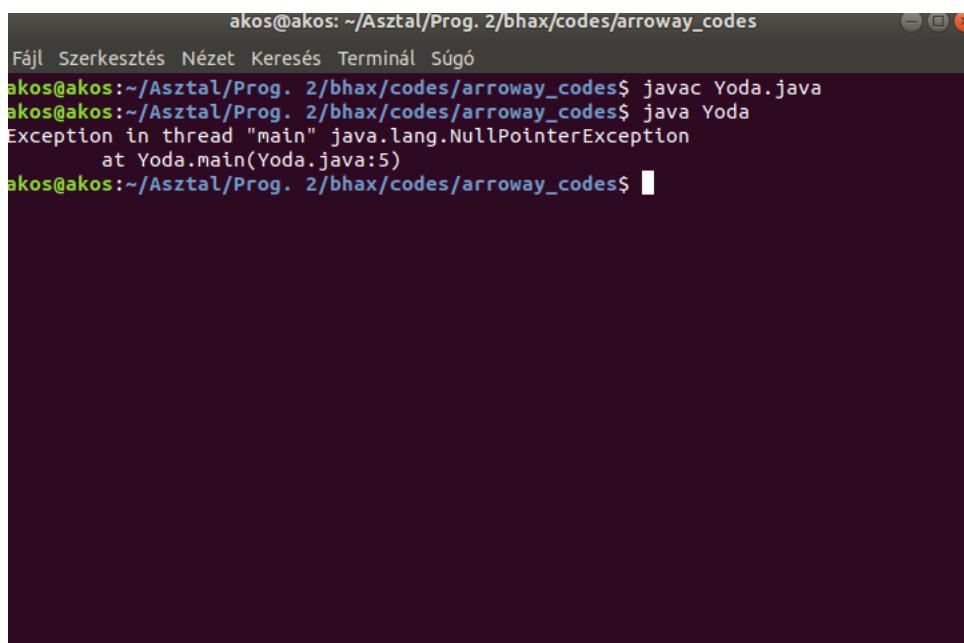
Elöljáróban pár szó a Yoda-conditon-ökről (nincs kifejezett magyar neve). Akkor beszélhetünk róluk, amikor egy adott feltételvizsgálatban a konstans érték az általunk megszokottól eltérően, az egyenlőségjel bal oldalán található. Sokan nem kedvelik nehezen olvashatósága miatt. Nevét természetesen arról a Yodáról kapta, akire elsőre gondolnánk. A Star Wars - univerzum zöld lénye a névadó, aki nem mindig a szokvános szósorrendet használva közölte mondandóját.

```
public class Yoda
{
 public static void main(String[] args)
 {
 String stringem = null;

 if (stringem.equals("star_wars"))

 {
 System.out.println("Nice!");
 }
 }
}
```

Fordítás és futtatás után:



A screenshot of a terminal window titled "akos@akos: ~/Asztal/Prog. 2/bhax/codes/arroway\_codes". The window shows the following command-line session:

```
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$ javac Yoda.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$ java Yoda
Exception in thread "main" java.lang.NullPointerException
 at Yoda.main(Yoda.java:5)
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$
```

Mi okozza az exceptiont? A hibaüzenetről le is olvasható. Egy nullpointerhez akarnánk hasonlítani a string literálunkat, ami természetesen nem megoldható.

## 11.4. Kódolás from scratch

Induljunk ki ebből a tudományos közleményből: <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/bbp-alg.pdf> és csak ezt tanulmányozva írjuk meg Java nyelven a BBP algoritmus megvalósítását!

Ha megakadsz, de csak végső esetben: [https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat-apbs02.html#pi\\_jegyei](https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat-apbs02.html#pi_jegyei) (mert ha csak lemásolod, akkor pont az a fejlesztői élmény marad ki, melyet szeret-

ném, ha átélnél).

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/arroway\\_codes/BBP](https://gitlab.com/kincsa/bhax/tree/master/codes/arroway_codes/BBP)

Tanulságok, tapasztalatok, magyarázat...

A Bailey-Borwein-Plouf (BBP) algoritmus az ún. BBP formulán alapszik, melyet 1995-ben fedeztek fel és a rá következő évben publikáltak. Az algoritmus lényegében a Pi hexa jegyeink meghatározására szolgál, amely a következőképp fest:

A kép forrása:[https://en.wikipedia.org/wiki/Bailey–Borwein–Plouffe\\_formula](https://en.wikipedia.org/wiki/Bailey–Borwein–Plouffe_formula)

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left( \frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right).$$

Lássuk akkor a megvalósítását!

```
public PiBBP(int d) {

 double d16Pi = 0.0d;

 double d16S1t = d16Sj(d, 1);
 double d16S4t = d16Sj(d, 4);
 double d16S5t = d16Sj(d, 5);
 double d16S6t = d16Sj(d, 6);

 d16Pi = 4.0d*d16S1t - 2.0d*d16S4t - d16S5t - d16S6t;

 d16Pi = d16Pi - StrictMath.floor(d16Pi);

 StringBuffer sb = new StringBuffer();

 Character hexaJegyek[] = {'A', 'B', 'C', 'D', 'E', 'F'};

 while (d16Pi != 0.0d) {

 int jegy = (int)StrictMath.floor(16.0d*d16Pi);

 if(jegy<10)
 sb.append(jegy);
 else
 sb.append(hexaJegyek[jegy-10]);

 d16Pi = (16.0d*d16Pi) - StrictMath.floor(16.0d*d16Pi);
 }

 d16PiHexaJegyek = sb.toString();
}
```

Létrehozásra kerül egy, az algoritmust használó objektumot mely egy darab paraméterrel rendelkezik. Az algoritmus zárójelében található értékek, illetve a hexadecimális jegyek megadása is megtörtént.

```
public double d16Sj(int d, int j) {

 double d16Sj = 0.0d;

 for(int k=0; k<=d; ++k)
 d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);

 return d16Sj - StrictMath.floor(d16Sj);
```

A "képlet" további részeinek kiszámítása, a fent látható metódussal, két paraméterrel. A kiszámolt rész a  $16^d S_j$  lett.

```
public long n16modk(int n, int k) {

 int t = 1;
 while(t <= n)
 t *= 2;

 long r = 1;

 while(true) {

 if(n >= t) {
 r = (16*r) % k;
 n = n - t;
 }

 t = t/2;

 if(t < 1)
 break;

 r = (r*r) % k;
 }

 return r;
}
```

A két paraméteres függvényünk a  $16^n$  modulo k értéket számítja ki.

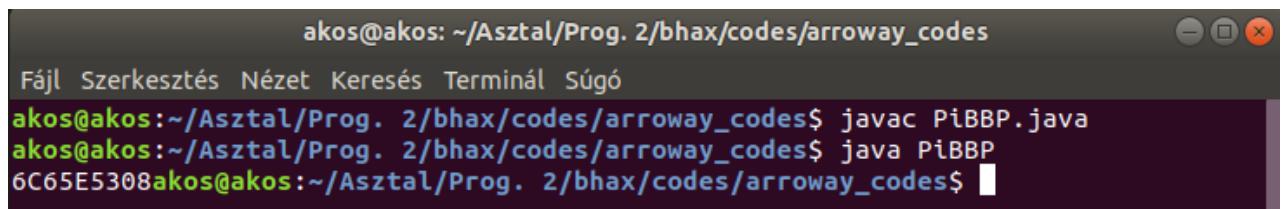
```
public String toString() {

 return d16PiHexaJegyek;
}
public static void main(String args[]) {
 System.out.print(new PiBBP(1000000));
}
```

```
}
```

A kiszámolt hexa-számjegy kiiratásával és egy objektum-példányosítással zárul a program.

Fordítás és futtatás után megkapjuk eredményünket:



The screenshot shows a terminal window with a dark background and light-colored text. The title bar reads "akos@akos: ~/Asztal/Prog. 2/bhax/codes/arroway\_codes". The window menu bar includes "Fájl", "Szerkesztés", "Nézet", "Keresés", "Terminál", and "Súgó". The terminal content is as follows:

```
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$ javac PiBBP.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$ java PiBBP
6C65E5308akos@akos:~/Asztal/Prog. 2/bhax/codes/arroway_codes$
```

## 12. fejezet

# Helló, Liskov!

### 12.1. Liskov helyettesítés sértése

Írunk olyan OO, leforduló Java és C++ kódcsipetet, amely megséríti a Liskov elvet! Mutassunk rá a megoldásra: jobb OO tervezés.

[https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2\\_1.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf) (93-99 fólia) (számos példa szerepel az elv megsértésére az UDPROG repóban, lásd pl. source/binom/Batfai-Barki/madarak/)

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/liskov\\_codes/Liskov](https://gitlab.com/kincsa/bhax/tree/master/codes/liskov_codes/Liskov)

Tanulságok, tapasztalatok, magyarázat...

A Liskov-féle behelyettesítési elv, rövid nevén LSP a következőt mondja ki: Ha S osztály a T osztály leszármazottja, akkor S szabadon behelyettesíthető a programunkban minden olyan helyre -lehet ez paraméter, változó, stb. . . -, ahol eredetileg T szülő-típust várunk.

A feladatunkhoz szemléletes példakód a következő, melyben bizonyítjuk, hogy megsérthető a Liskov-elv:

```
#include <iostream>

using namespace std;

class Madar {
public:
 virtual void repul() {
 cout<<"Tudok repulni!"<<endl;
 }
};

class Program {
public:
 void fgv (Madar &madar) {
 madar.repul();
 }
};
```

```
class Pingvin : public Madar
{
};

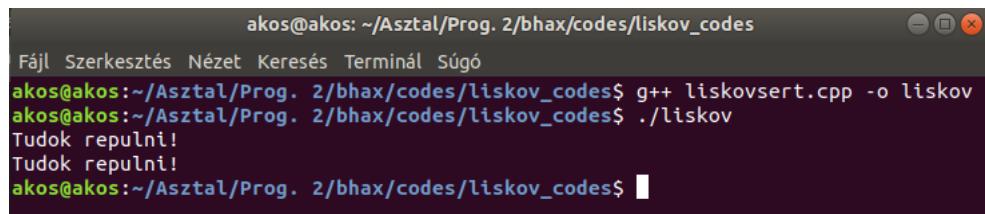
int main (int argc, char **argv)
{
 Program program;
 Madar madar;
 program.fgv (madar);

 Pingvin pingvin;
 program.fgv (pingvin);
}
```

Ha az előző jelölésekkel szeretnénk továbbra is operálni, akkor: a Madar osztály esetünkben a T, míg a Pingvin lesz az S osztály. Magyarul minden olyan helyre, ahol a Madar típusat adnánk meg, megadható akár Pingvin típus is.

Vegyük sorra, mi történik a kódban: létrehozásra kerül a Madar osztály egy repul tagfüggvénytel, majd a Program osztályban helyet kap egy fgv névre hallgató függvény, melynek feladata egy Madar típusú egyedre meghívni az előző repul függvényt. Ezt követően létrehozásra kerül a Pingvin osztály, melynek szülőosztályaként a Madar osztályt adjuk. Ezt követően már csak a példányosítás és a függvények meghívása maradt hátra.

Fordítás és futtatás után:



A terminal window titled "akos@akos: ~/Asztal/Prog. 2/bhax/codes/liskov\_codes". The window shows the following command-line session:

```
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov_codes$ g++ liskovsert.cpp -o liskov
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov_codes$./liskov
Tudok repulni!
Tudok repulni!
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov_codes$
```

És hogy miért probléma ez? Köztudott, hogy annak ellenére, hogy a pingvin madár, repülésre képtelen. A program azonban ezt nem tudja, reptetné a pingvinünket méghozzá azért, mert madár.

A Java átirat a következőképp néz ki:

```
class Madar
{
 public void repul()
 {
 System.out.print("Tudok repulni!\n");
 }
}

class Pingvin extends Madar
{

}

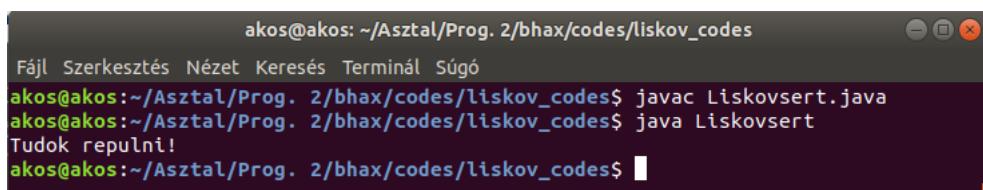
public class Liskovsert
```

```
{
 public static void fgv(Madar madar)
 {
 madar.repul();
 }

 public static void main(String[] args)
 {
 Madar pingvin = new Pingvin();

 fgv(pingvin);
 }
}
```

Fordítás és futtatás után:



A terminal window titled "akos@akos: ~/Asztal/Prog. 2/bhax/codes/liskov\_codes". The window shows the following text:  
Fájl Szerkesztés Nézet Keresés Terminál Súgó  
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov\_codes\$ javac Liskovsert.java  
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov\_codes\$ java Liskovsert  
Tudok repulni!  
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov\_codes\$ █

Mivel csak átiratról van szó, a helyzet ugyanaz. Probléma nélkül lefordul programunk ami aztán repteti pingvinünket annak ellenére, hogy az még mindig egy röpképtelen madár.

## 12.2. Szülő-gyerek

Írunk Szülő-gyerek Java és C++ osztálydefiníciót, amelyben demonstrálni tudjuk, hogy az ősön keresztül csak az ős üzenetei küldhetőek!

[https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2\\_1.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf) (98. fólia)

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/liskov\\_codes/Szulo-gyerek](https://gitlab.com/kincsa/bhax/tree/master/codes/liskov_codes/Szulo-gyerek)

Tanulságok, tapasztalatok, magyarázat...

A feladat megoldása során írt programban egy szülő-gyermekek kapcsolatot veszünk szemügyre mely során belátható, hogy míg a gyermekosztály értelemszerűen használhatja az ősosztály változóit, metódusait, addig a gyermekosztályban definiált változók és metódusok nem értelmezhetők a szülőosztályban. Utóbbit fogalmazza meg a feladat leírása is: "az ősön keresztül csak az ős üzenetei küldhetőek".

A következő szemléletes kódok jól alá tudják támasztani az előbb megfogalmazottakat.

A Java kód:

```
class Lajhár
{
 protected String nev;

 public void setNeve(String neve)
```

```
{
 nev = neve;
}

}

class kisLajhar extends Lajhar
{
 public String getNev()
 {
 return nev;
 }
}

class szuloGyerek
{
 public static void main (String args[])
 {
 Lajhar l = new kisLajhar();
 l.setNev("Sid");
 kisLajhar k = new kisLajhar();
 k.setNev("SidJr.");
 System.out.println(k.getNev() + " " + l.getNev());
 }
}
```

Mi történik itt? Létrehozásra kerül a `Lajhar` osztály melyben egy változó és egy függvény található, mely segítségével példányosításkor megadhatjuk lajhárunk nevét. Ezen osztály leszármazottja a `kisLajhar` nevet kapja. Az `extends` kulcsszó után megadjuk, hogy osztályunk melyik ősosztályból lett származtatva. Itt, a `kisLajhar` osztályban csupán egy `getNev` névre hallgató metódus került létrehozásra mely segítségével le tudjuk kérdezni lajhárunk nevét. A következő pár sort már a `main`-ben találjuk. Megvalósul a példányosítás, létrejönnek objektumaink. Majd végül egy kiiratás keretein belül megpróbáljuk a szülőre meghívni a gyermek osztály metódusát.

Az eredmény a várható volt:

```
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov_codes$ javac szuloGyerek.java
szuloGyerek.java:29: error: cannot find symbol
 System.out.println(k.getNev() + " " + l.getNev());
 ^
 symbol: method getNev()
 location: variable l of type Lajhar
1 error
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov_codes$
```

Ugyanezt igazoljuk C++-ban is, az előző kód átíratával:

```
#include <iostream>

using namespace std;

class Szulo
{
```

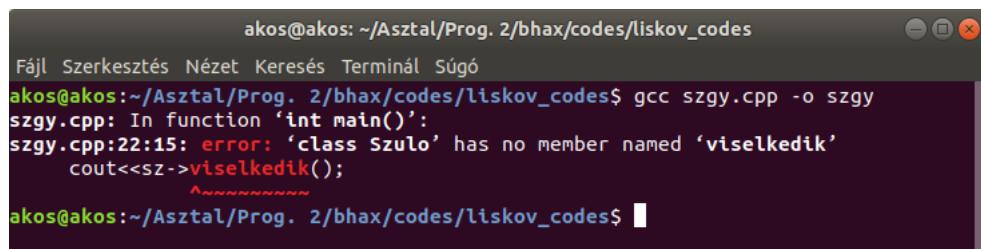
```
};

class Gyerek: public Szulo
{
 void viselkedik()
 {
 cout<<"Viselkedek!";
 }
};

int main ()
{
 Szulo* sz= new Gyerek;

 cout<<sz->viselkedik();
}
```

Fordítás után:



```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/liskov_codes
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov_codes$ gcc szgy.cpp -o szgy
szgy.cpp: In function 'int main()':
szgy.cpp:22:15: error: 'class Szulo' has no member named 'viselkedik'
 cout<<sz->viselkedik();
 ^
akos@akos:~/Asztal/Prog. 2/bhax/codes/liskov_codes$
```

Látható, hogy a fordító megint mi miatt jelez hibát..A szülőosztály nem tudja használni a gyermekosztályban definiált metódusokat.

## 12.3. Anti OO

A BBP algoritmussal a Pi hexadecimális kifejtésének a 0. pozíciótól számított 10 6, 107, 108 darab jegyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket! <https://www.tankonyvtar.hu-hu/tartalom/tkt/javat-tanitok-javat/apas03.html#id561066>

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/blob/master/codes/liskov\\_codes/pi\\_bbp\\_bench.c](https://gitlab.com/kincsa/bhax/blob/master/codes/liskov_codes/pi_bbp_bench.c)

[https://gitlab.com/kincsa/bhax/tree/master/codes/liskov\\_codes/Anti%20OO](https://gitlab.com/kincsa/bhax/tree/master/codes/liskov_codes/Anti%20OO)

Tanulságok, tapasztalatok, magyarázat...

A kód elemzése már megtörtént az előző csokorban, így azt nem ismételném meg ugyanitt. Ami a mostani kódok érdekessége, hogy meg lett szüntetve objektumorientáltságuk a könnyebb összehasonlítás érdekében.

Annak függvényében, hogy Pi-nek hányadik számjegyét akarjuk megkapni, kell a programrészletben található for ciklusban a d ciklusváltozót módosítani.

```
for (int d=100000000; d<100000001; ++d)
```

A fenti részleten éppen a Pi 0-tól számított  $10^8$ . jegyét számoltuk.

Az eredmények rendre a következő sorrendben láthatók majd: C, Java, C# és C++.

|        | C      | Java   | C#     | C++    |
|--------|--------|--------|--------|--------|
| $10^6$ | 2.103  | 1.867  | 3.734  | 2.122  |
| $10^7$ | 24.442 | 22.893 | 45.521 | 24.714 |
| $10^8$ | 280.56 | 250.55 | 566.31 | 286.7  |

Mi szűrhető le az eredményekből? minden esetben a Java kód futott le a leggyorsabban, míg utolsó helyen rendre a C# kód volt található.

A C# kódok futtatására a következő online felületet vettet igénybe: [https://www.onlinedb.com/online\\_csharp\\_compiler](https://www.onlinedb.com/online_csharp_compiler)

## 12.4. Ciklomatikus komplexitás

Számoljuk ki valamelyik programunk függvényeinek ciklomatikus komplexitását! Lásd a fogalom tekintetében a [https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2\\_2.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_2.pdf) (77-79 fóliát)!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Bevezető forrása: [https://hu.wikipedia.org/wiki/Ciklomatikus\\_komplexitás](https://hu.wikipedia.org/wiki/Ciklomatikus_komplexitás)

Ejtsünk pár szót alapvetően arról, mi is a ciklomatikus komplexitás. A ciklomatikus- vagy McCabe komplexitás lényegében egy szoftvermérési szempont, egy számérték, amely számérték a programunk bonyolultságát hivatott jelölni azzal, hogy minél nagyobb ez az érték, annál bonyolultabb programunk is. 1976-ban Thomas J. McCabe publikálta. A számolás alapjait a gráfelmélet szolgáltatja, ugyanis lényegében nem más, mint a vezérlési gráfban található független utak maximális száma.

Ez szoftvereink esetén a következőt jelenti: a gráf csúcsai az utasítás-sorozatoknak felelnek meg, élei pedig a program által, a szekvenciák között létrehozott lehetséges közvetlen átmeneteknek.

Az érték kiszámítása:  $M = E - N + 2P$ , ahol E a szóban forgó gráf éleinek száma, N a gráfban lévő csúcsok száma, P pedig az összefüggő komponensek száma.

A feladat megoldásához a Lizard Code Complexity Analizert használtam. Létezik letölthető verziója is, azonban könnyebnek tartottam az online felületen kielemeztetni a kódomat. Az elemzések kód a Java-s polárgenerátor volt. Az eredmény:

The screenshot shows the Lizard code analysis tool interface. On the left, there is a code editor window titled "Try Lizard in Your Browser" with a ".java" file type selected. The code in the editor is:

```
public static void main(String[] args) {
 PolarGenerator g = new PolarGenerator();

 for(int i=0; i<10; ++i)
 System.out.println(g.következő());
}
```

Below the code editor is a blue "Analyse" button. To the right of the code editor is a summary section with the message "Code analyzed successfully." and some statistics:

File Type .java    Token Count 221    NLOC 31

| Function Name                  | NLOC | Complexity | Token # | Parameters |
|--------------------------------|------|------------|---------|------------|
| PolarGenerator::PolarGenerator | 3    | 1          | 11      |            |
| PolarGenerator::ő              | 19   | 3          | 137     |            |
| PolarGenerator::main           | 5    | 2          | 47      |            |

Látható, hogy a program osztályokra lebontva írja ki a ciklomatikus komplexitást.

## 13. fejezet

# Helló, Mandelbrot!

### 13.1. Reverse engineering UML osztálydiagram

UML osztálydiagram rajzolása az első védési C++ programhoz. Az osztálydiagramot a forrásokból generáljuk (pl. Argo UML, Umbrello, Eclipse UML) Mutassunk rá a kompozíció és aggregáció kapcsolatára a forráskódban és a diagramon, lásd még: [https://youtu.be/Td\\_nlERIEOs](https://youtu.be/Td_nlERIEOs).

[https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1\\_6.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_6.pdf) (28-32 fólia)

Megoldás videó:

Megoldás forrása: [http://www.stud.u-szeged.hu/Deak.Kristof/prog1/oo\\_uml.pdf](http://www.stud.u-szeged.hu/Deak.Kristof/prog1/oo_uml.pdf)

[https://www.tankonyvtar.hu/hu/tartalom/tamop425/0046\\_szoftverfejlesztes/ch09.html](https://www.tankonyvtar.hu/hu/tartalom/tamop425/0046_szoftverfejlesztes/ch09.html)

Tanulságok, tapasztalatok, magyarázat...

A Unified Modeling Language (rövidítve UML) egy szabványosított, modellezésre használható nyelv. Mai formájában a '90-es évek óta működik. Számos diagramtípus megtalálható a nyelvben, mi a feladataink során az osztálydiagrammal fogunk foglalkozni.

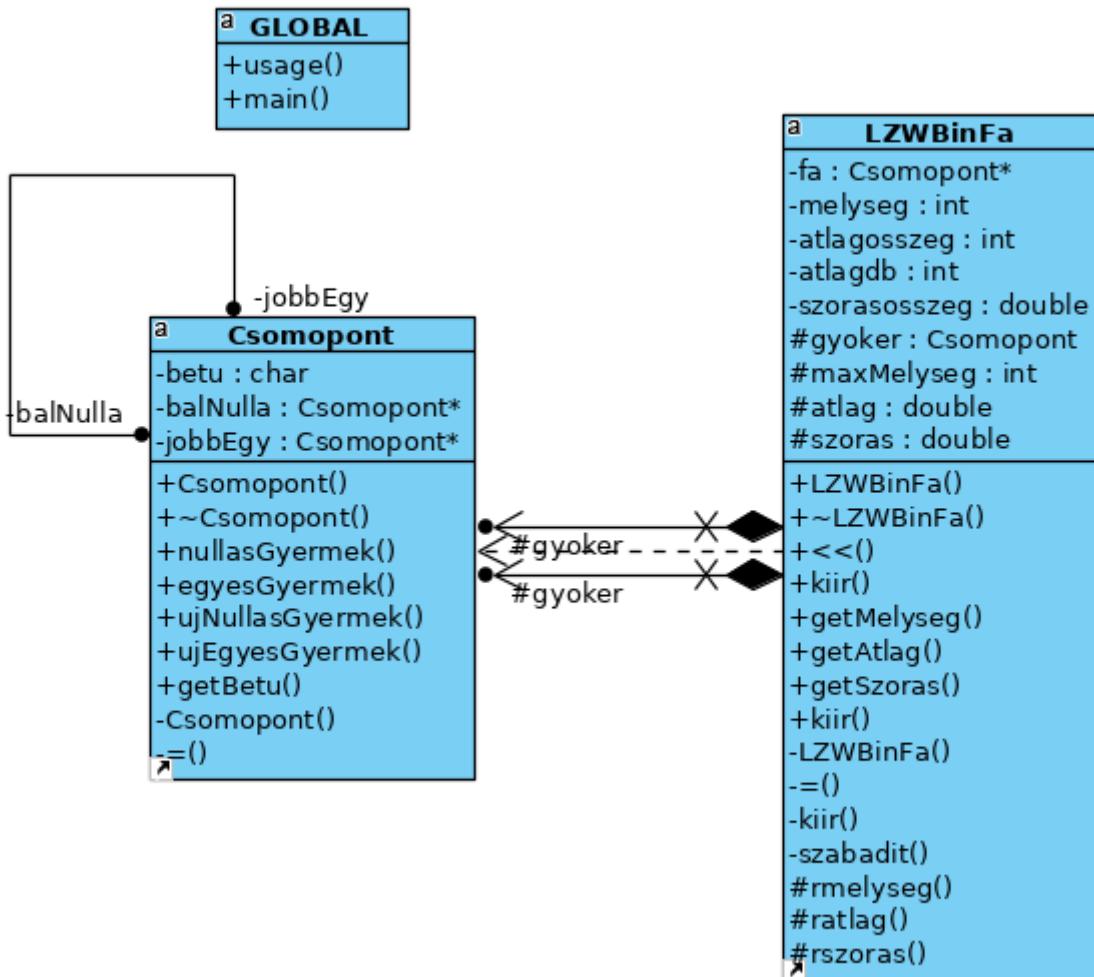
Az osztálydiagram alapvetően egy adott program osztályai közötti kapcsolatrendszert hivatott reprezentálni. minden egyes "téglalap" egy osztályt jelöl, a diagram ábrázolja az osztályok attribútumait és metódusait. Előbbieket a vízszintes vonal felett találjuk meg, utóbbiakat értelemszerűen azalatt. Mindkettő esetében megfigyelhető, hogy előttük +, - vagy éppen # jelek állnak. Ezek az elemek láthatóságát befolyásolják, rendre: public, private és protected.

A feladat külön kéri, hogy térjünk ki a kompozíció és aggregáció kapcsolatára.

Aggregációról akkor beszélhetünk, amikor az egyik objektum részben (vagy akár egészben) is tartalmazza a másikat. Két típusú van: erős- és gyenge aggregáció. Utóbbit nevezzük kompozíciónak, mely szerint a tartalmazó és tartalmazott objektum nagy mértékben függnek egymástól, egyik a másik nélkül nem funkcionál.

Az aggregáció jelölése nem tömött rombusszal, míg a kompozíció tömött rombusszal történik.

A kért osztálydiagramot először Umbrello-val próbáltam legenerálni de az valahogy nem akarta az igazságot, ezért a [Visual Paradigm](#)-ra, azon belül is a Community Edition-re esett a választásom. Az eredmény a következő lett:



## 13.2. Forward engineering UML osztálydiagram

UML-ben tervezünk osztályokat és generálunk belőle forrást!

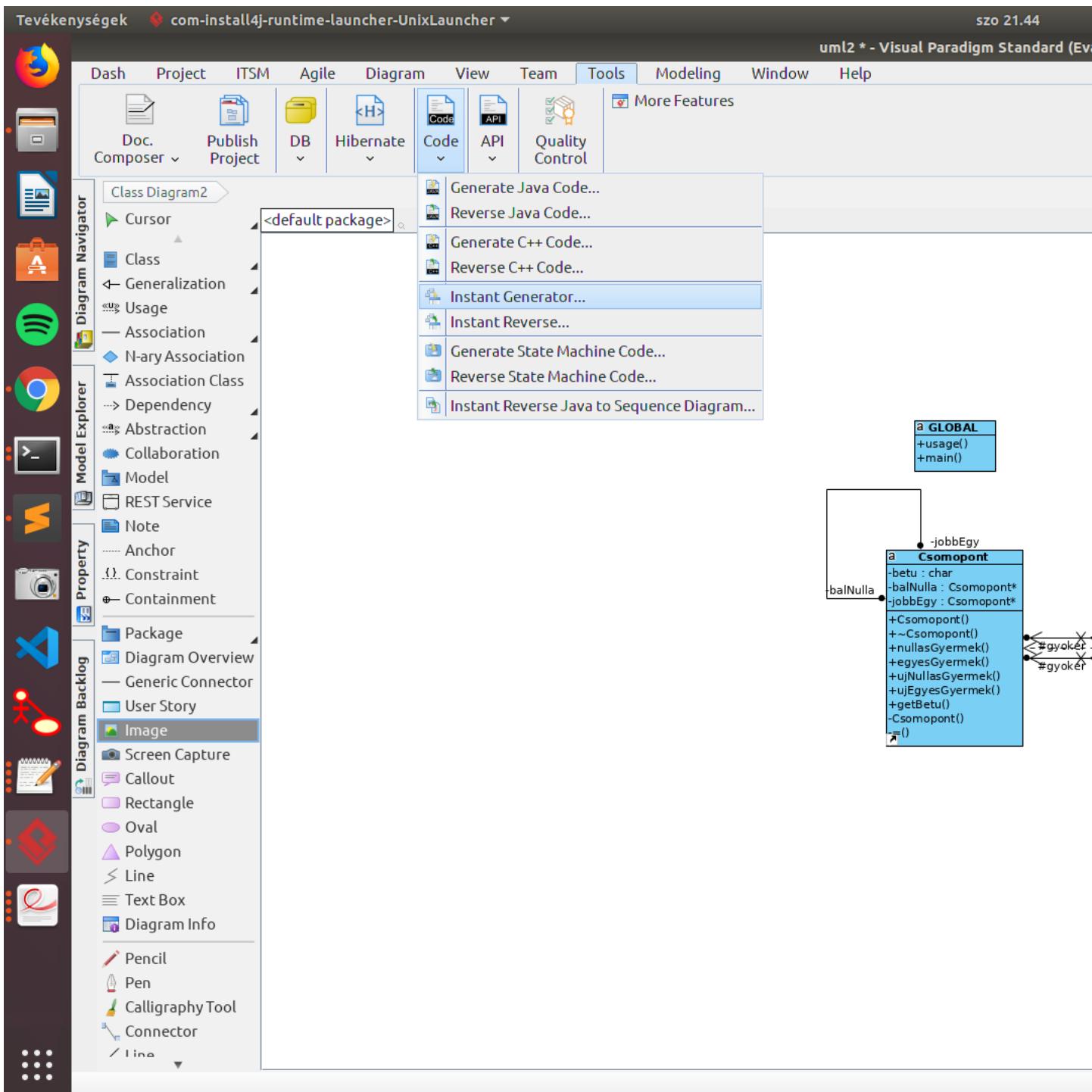
Megoldás videó:

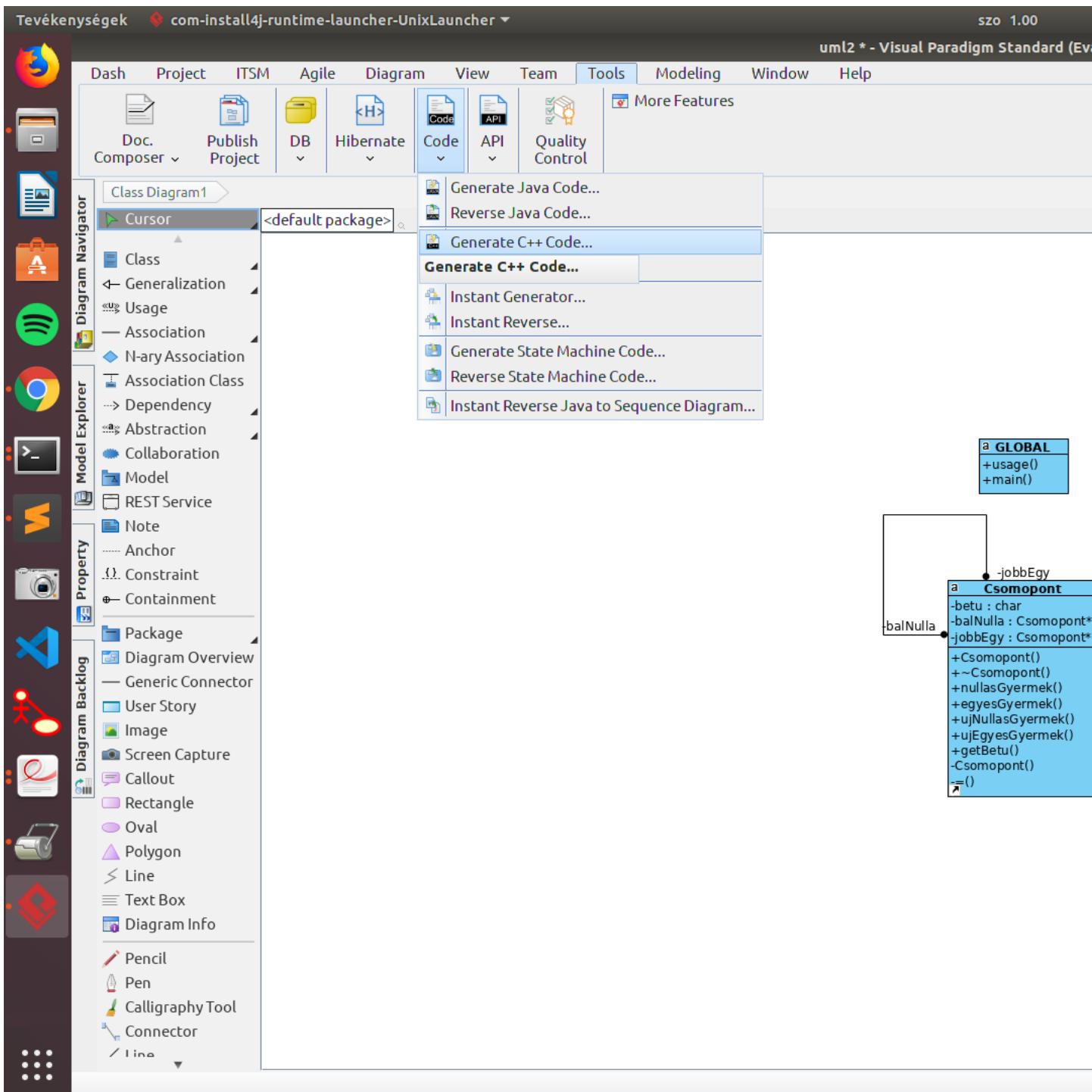
Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Ez a feladat olyan szempontból nagyon érdekes, hogy lényegében az előző fordítottja.

Így neki is álltam és próbáltam tervezni egy nem túl bonyolult osztálydiagramot. Maga a diagram a következőképp néz ki és készült el:





Tehát ugyanott, a Tools menüpontban van lehetőségünk az osztálydiagramunkat "visszafejteni", majd egy pár kattintás után előállnak kódjaink.

Az eredmény:

```
Megnyitás ▾  LZWBinFa.h
~/Asztal/Prog. 2/bhax/codes/mandelbrot2_codes/yeaye/LZWBinFa.h
#include <exception>
using namespace std;

#ifndef __LZWBinFa_h__
#define __LZWBinFa_h__

#include "Csomopont.h"

class Csomopont;
class LZWBinFa;

class LZWBinFa
{
 private: Csomopont* _fa;
 private: int _melyseg;
 private: int _atlagosszeg;
 private: int _atlagdb;
 private: double _szorasosszeg;
 protected: Csomopont _gyoker;
 protected: int _maxMelyseg;
 protected: double _atlag;
 protected: double _szoras;

public: LZWBinFa();
public: void _LZWBinFa();
public: void _<(char aB);
public: void kiir();
public: int getMelyseg();
public: double getAtlag();
public: double getSzoras();
public: void kiir(std::ostream& aOs);
private: LZWBinFa(const LZWBinFa& aUnnamed_1);
private: LZWBinFa& _(const LZWBinFa& aUnnamed_1);
private: void kiir(Csomopont* aElem, std::ostream& aOs);
private: void szabadit(Csomopont* aElem);
protected: void rmelyseg(Csomopont* aElem);
protected: void ratlag(Csomopont* aElem);
protected: void rszoras(Csomopont* aElem);
};

#endif
```

Tevékenységek Szövegszerkesztő ▾

Megnyitás ▾

#include <exception>  
using namespace std;

#include "LZWBinFa.h"  
#include "Csomopont.h"

LZWBinFa::LZWBinFa() {  
}

void LZWBinFa::\_LZWBinFa() {  
 throw "Not yet implemented";  
}

void LZWBinFa::\_<(char aB) {  
 throw "Not yet implemented";  
}

void LZWBinFa::kiir() {  
 throw "Not yet implemented";  
}

int LZWBinFa::getMelyseg() {  
 return this->\_melyseg;  
}

double LZWBinFa::getAtlag() {  
 return this->\_atlag;  
}

double LZWBinFa::getszoras() {  
 return this->\_szoras;  
}

void LZWBinFa::kiir(std::ostream& aOs) {  
 throw "Not yet implemented";  
}

LZWBinFa::LZWBinFa(const LZWBinFa& aUnnamed\_1) {  
}

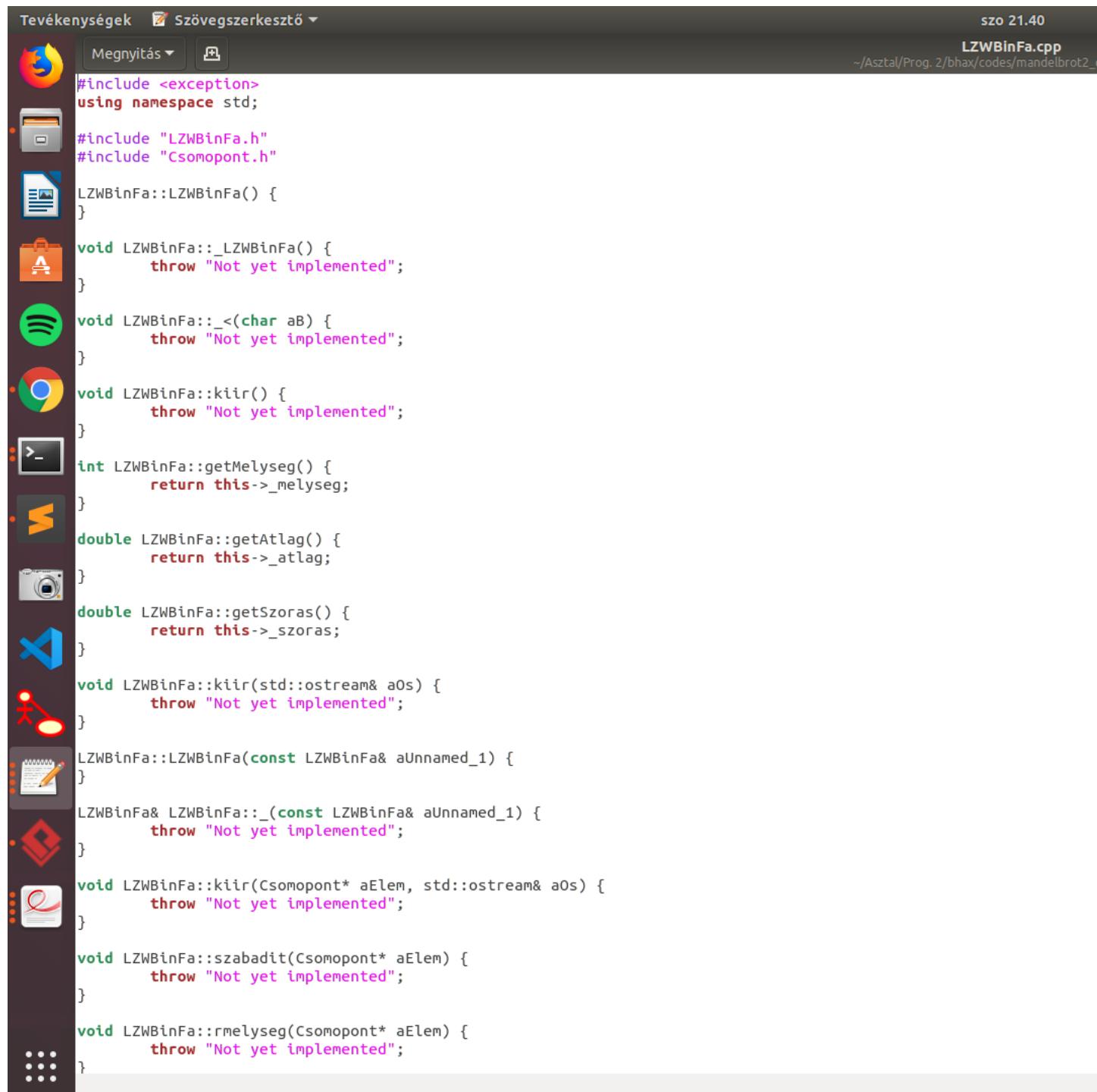
LZWBinFa& LZWBinFa::\_(const LZWBinFa& aUnnamed\_1) {  
 throw "Not yet implemented";  
}

void LZWBinFa::kiir(Csomopont\* aElem, std::ostream& aOs) {  
 throw "Not yet implemented";  
}

void LZWBinFa::szabadit(Csomopont\* aElem) {  
 throw "Not yet implemented";  
}

void LZWBinFa::rmelyseg(Csomopont\* aElem) {  
 throw "Not yet implemented";  
}

szo 21.40  
LZWBinFa.cpp  
~/Aszta/Prog. 2/bhx/codes/mandelbrot2...



Tevékenységek Szövegszerkesztő ▾

Megnyitás ▾

#include <exception>  
using namespace std;

#ifndef \_\_Csomopont\_h\_\_  
#define \_\_Csomopont\_h\_\_

class Csomopont;

class Csomopont  
{

private: char \_betu;  
private: Csomopont\* \_balNulla;  
private: Csomopont\* \_jobbEgy;

public: Csomopont(char aB = '/');

public: void \_Csomopont();

public: Csomopont\* nullasGyermekek();

public: Csomopont\* egyesGyermekek();

public: void ujNullasGyermekek(Csomopont\* aGy);

public: void ujEgyesGyermekek(Csomopont\* aGy);

public: char getBetu();

private: Csomopont(const Csomopont& aUnnamed\_1);

private: Csomopont& \_(const Csomopont& aUnnamed\_1);

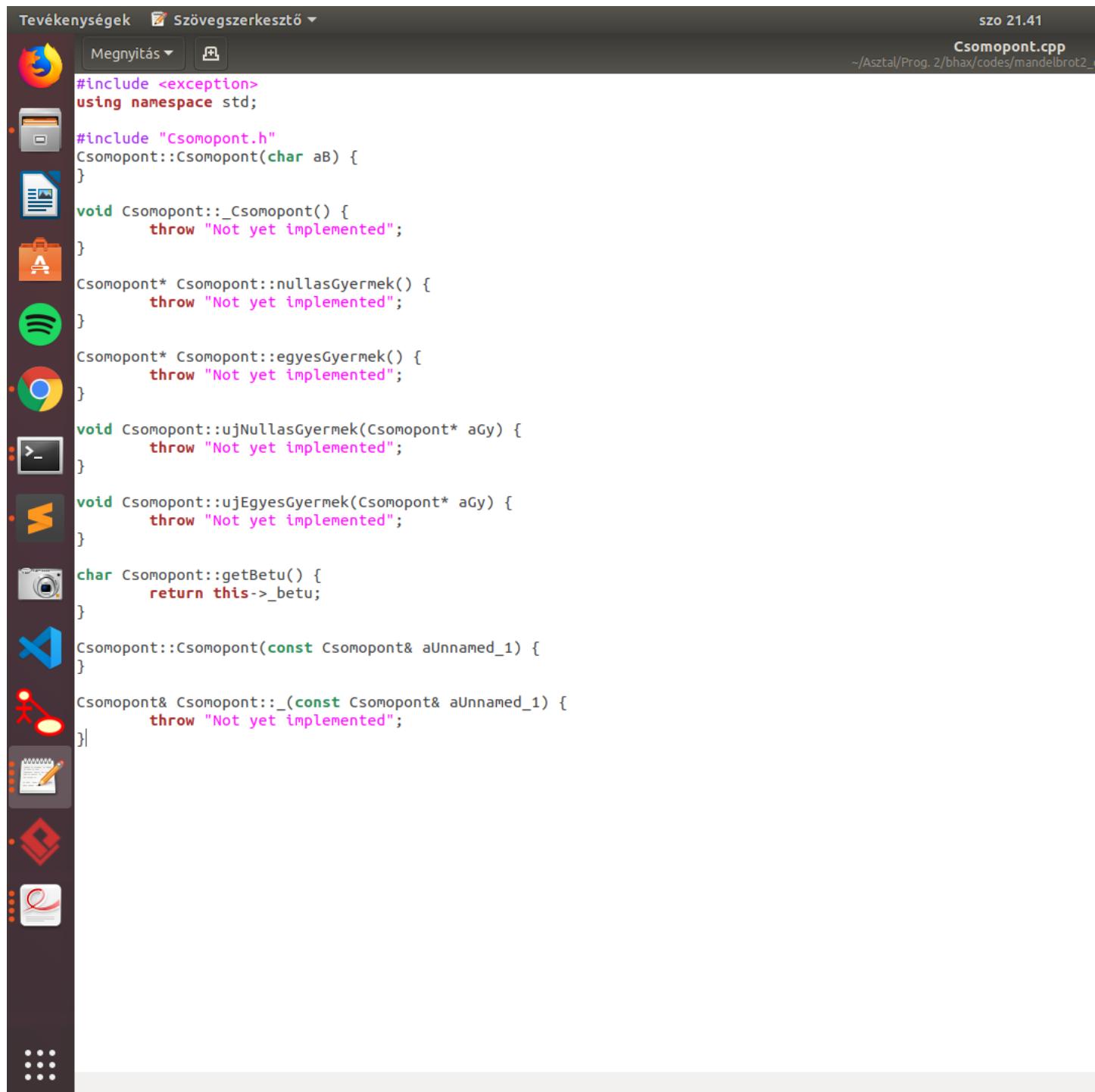
};

#endif

szo 21.40

Csomopont.h  
~/Asztal/Prog. 2/bhax/codes/mandelbrot2...





The screenshot shows the Umbrello application interface. At the top, there's a menu bar with "Tevékenységek" and "Szövegszerkesztő". The main area is a code editor with the following content:

```
#include <exception>
using namespace std;

#include "Csonopont.h"
Csonopont::Csonopont(char aB) {
}

void Csonopont::_Csonopont() {
 throw "Not yet implemented";
}

Csonopont* Csonopont::nullasGyermekek() {
 throw "Not yet implemented";
}

Csonopont* Csonopont::egyesGyermekek() {
 throw "Not yet implemented";
}

void Csonopont::ujNullasGyermek(Csonopont* aGy) {
 throw "Not yet implemented";
}

void Csonopont::ujEgyesGyermek(Csonopont* aGy) {
 throw "Not yet implemented";
}

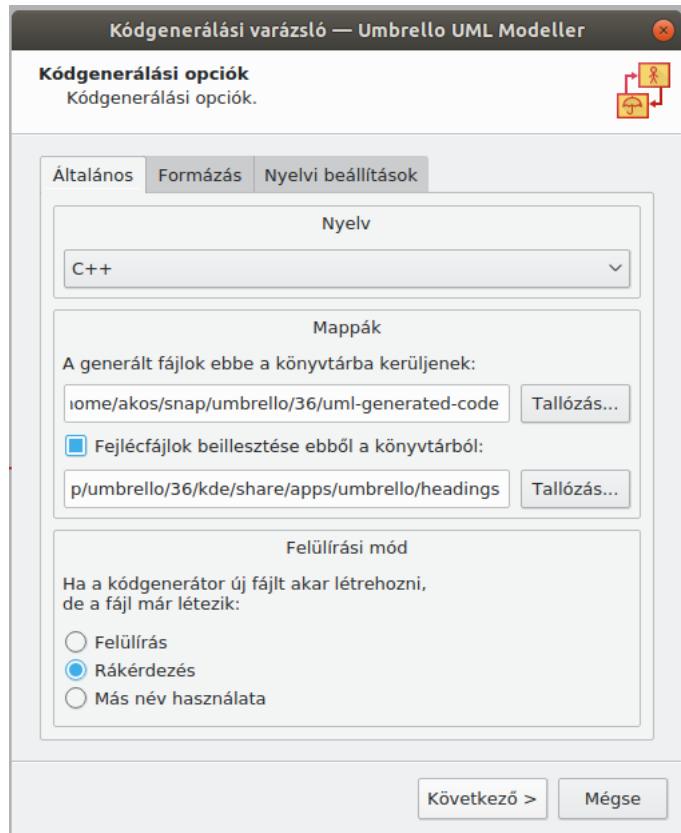
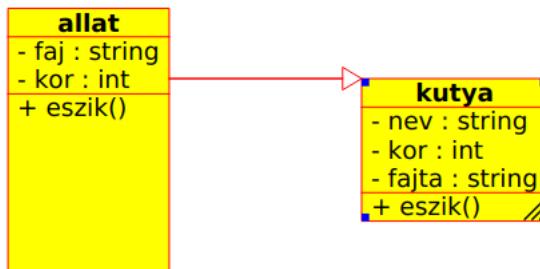
char Csonopont::getBetu() {
 return this->_betu;
}

Csonopont::Csonopont(const Csonopont& aUnnamed_1) {
}

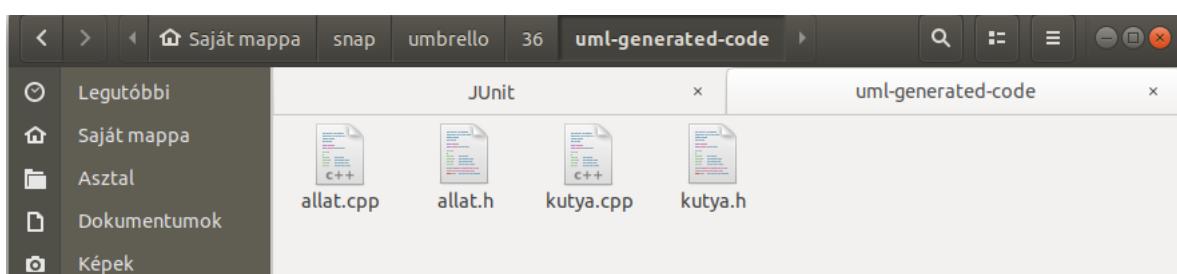
Csonopont& Csonopont::_(const Csonopont& aUnnamed_1) {
 throw "Not yet implemented";
}
```

Látható, hogy legenrálódik a kód. Érdekesség, hogy a függvények természetesen törzs nélkül jönnek létre, így ha ilyformán szeretnénk programkódot generáltatni, ezzel a tényezővel számolnunk kell.

Ki akartam próbálni a feladat megoldásához egy másik programot is, a szintén népszerű Umbrello-t. A legprimitívebb osztálydiagram elkészítése után a következőképp generáltattam kódot:



Egy barátságos párbeszédablakban be tudjuk állítani a generálandó kódunk minden tulajdonságát egytől-egyig, minden egyértelmű.



És létre is jöttek fájljaink, mind a várt tartalommal.

### 13.3. BPMN

Rajzoljunk le egy tevékenységet BPMN-ben! <https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog34-47.pdf>

Megoldás video:

Megoldás forrása: [https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2\\_7.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_7.pdf)

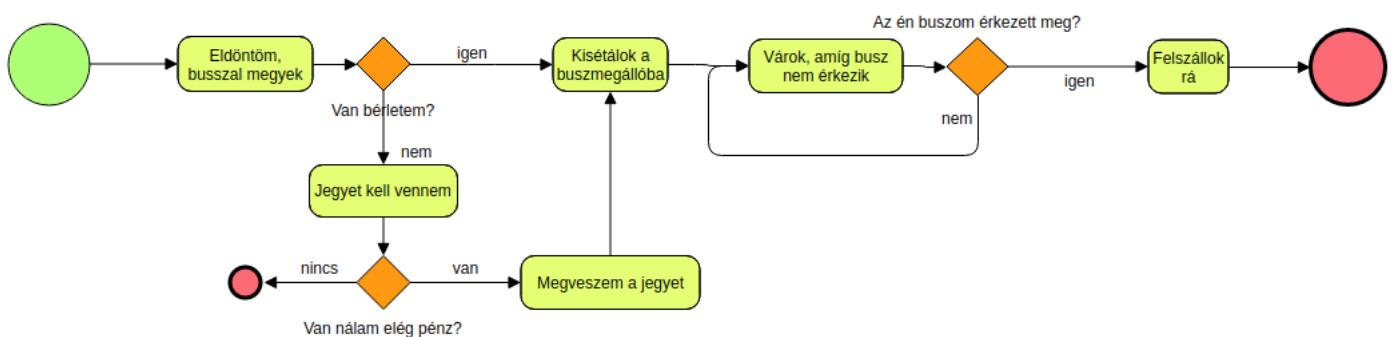
<https://www.omg.org/spec/BPMN/2.0/PDF>

Tanulságok, tapasztalatok, magyarázat...

A BPMN (Business Process Model and Notation) egy folyamatmodellező Ahogy azt a zárójelben olvasható kibontott mozaikszó is sugallja, elsősorban az üzleti, vállalati világban kerül alkalmazásra. A legújabb kiadás 2014-es, ez a 2.0.2-es verziószámú.

A feladat megoldása során a [Visual Paradigm](#) online kiadását használtam fel.

A lemodellezendő hétköznapi tevékenységnek a busszal (természetesen bármely másik tömegközlekedési eszközöt írhattam volna...) történő közlekedést választottam.



Alapvetően a BPMN-ben használt jelölésrendszer nem bonyolult, itt megtalálható minden ezzel kapcsolatos tudnivaló: <https://www.omg.org/spec/BPMN/2.0/PDF/>

Lényegében egy nem túlságosan bonyolult folyamatról van szó, a három elágazástól (narancssárga rombusz) eltekintve, ahol minden alkalommal kétfelé ágazhatott el az út, alapvetően egy lineáris folyamatról beszélhetünk. A jelölésrendszer a következő: a zöld kör a kezdő állapot(esemény), míg a piros a végállapot. A narancssárga rombuszokról már esett szó, a lekerekített sarkú, sárgás színű téglalapok pedig az ún. "tevékenységek". A köztük lévő nyílak az adott folyamat részegységei közötti sorrendet hivatottak jelölni, természetesen a nyíl hegeses része felőli tag időrendi sorrendben később következik, míg a nyíl másik oldalán lévő természetesen hamarabb.

Nyilván lehetett volna sokkal bonyolultabb folyamatot is választani de úgy gondolom, a lényeg ezen a példán keresztül is jól szemléltethető.

## 14. fejezet

# Helló, Chomsky!

### 14.1. Encoding

Fordítsuk le és futtassuk a Javat tanítok könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezes betűket!

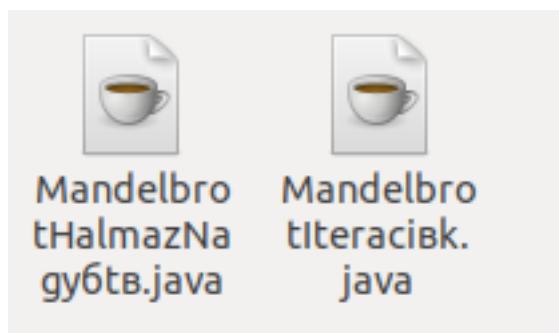
<https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/adatok.html>

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/chomsky2\\_codes/encoding](https://gitlab.com/kincsa/bhax/tree/master/codes/chomsky2_codes/encoding)

Tanulságok, tapasztalatok, magyarázat...

Miután a forrásokat a megadott linkről letöltöttük, egy igencsak szembetűnő dologgal találhatjuk szemben magunkat:



Amint az látható, a letöltött állományok nevei olvashatatlanok. Ezt kijavítva futtatni próbáltam a programokat. A következő fogadott:

```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/encoding
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/encoding$ javac MandelbrotIterációk.java
MandelbrotIterációk.java:2: error: unmappable character (0xE1) for encoding UTF-8
 * MandelbrotIteraciók.java
 ^
MandelbrotIterációk.java:2: error: unmappable character (0xF3) for encoding UTF-8
 * MandelbrotIteraciók.java
 ^
MandelbrotIterációk.java:4: error: unmappable character (0xED) for encoding UTF-8
 * DIGIT 2005, Javat tanítok
 ^
MandelbrotIterációk.java:5: error: unmappable character (0xE1) for encoding UTF-8
 * Batfai Norbert, nbatfai@inf.unideb.hu
 ^
MandelbrotIterációk.java:9: error: unmappable character (0xED) for encoding UTF-8
 * A nagyított Mandelbrot halmazok adott pontjában kópes
 ^
MandelbrotIterációk.java:9: error: unmappable character (0xE1) for encoding UTF-8
```

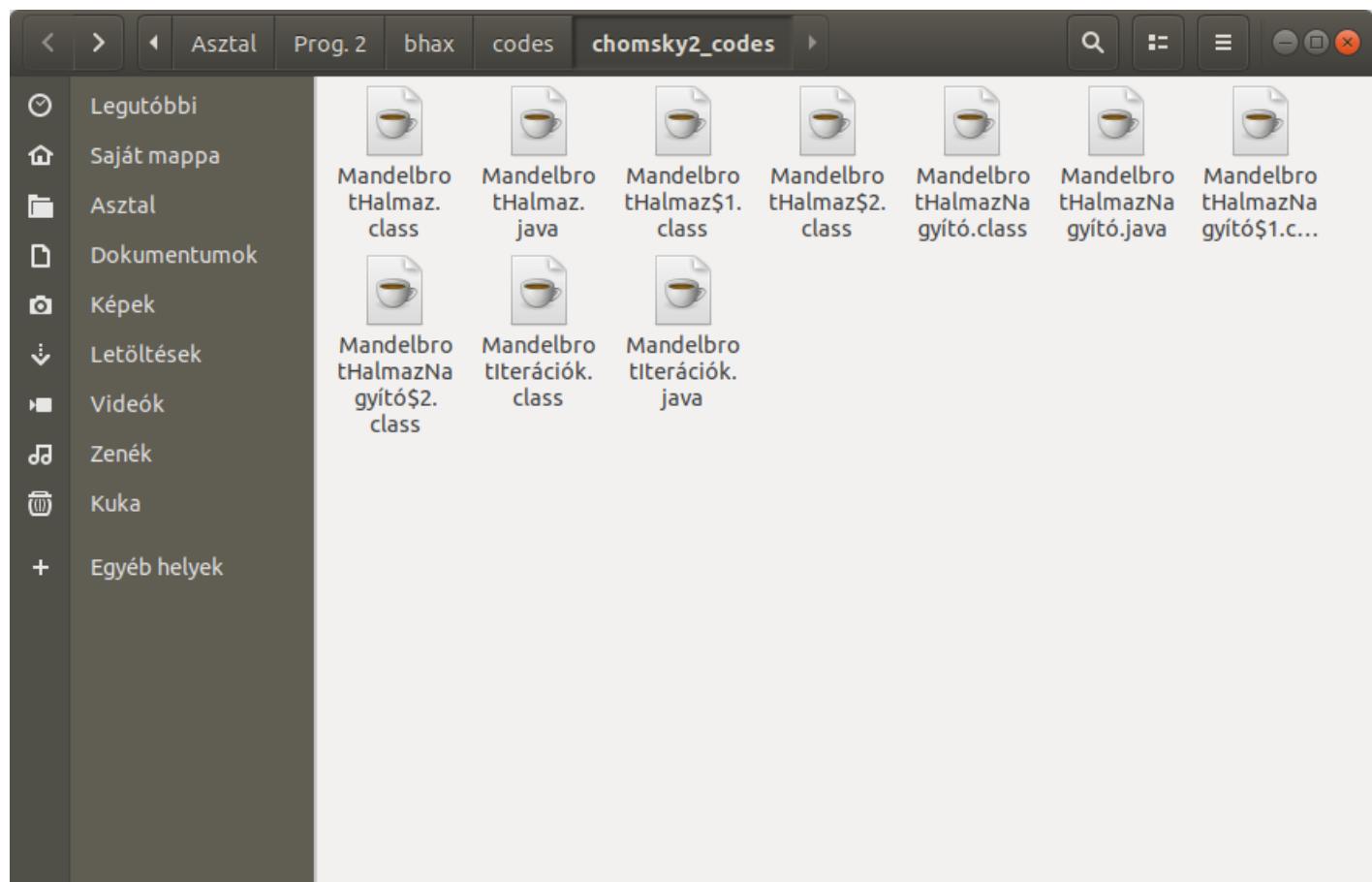
Ha elolvassuk a hibaüzenetet, egyből olvashatjuk, mi a hiba. Az UTF-8-as kódolásban meg nem található karaktereket talál a kódban, amit nem tud értelmezni a fordító. Szerencsére van megoldás, olyan kódolást kell keresni, ami tartalmazza az ékezetes betűket beleértve az ű-t és aző-t is, így nem lesz problémája a fordítónak.

A választás az ISO8859-2-es (Latin 2) kódolásra esett, ez megfelel az elvárásainknak.

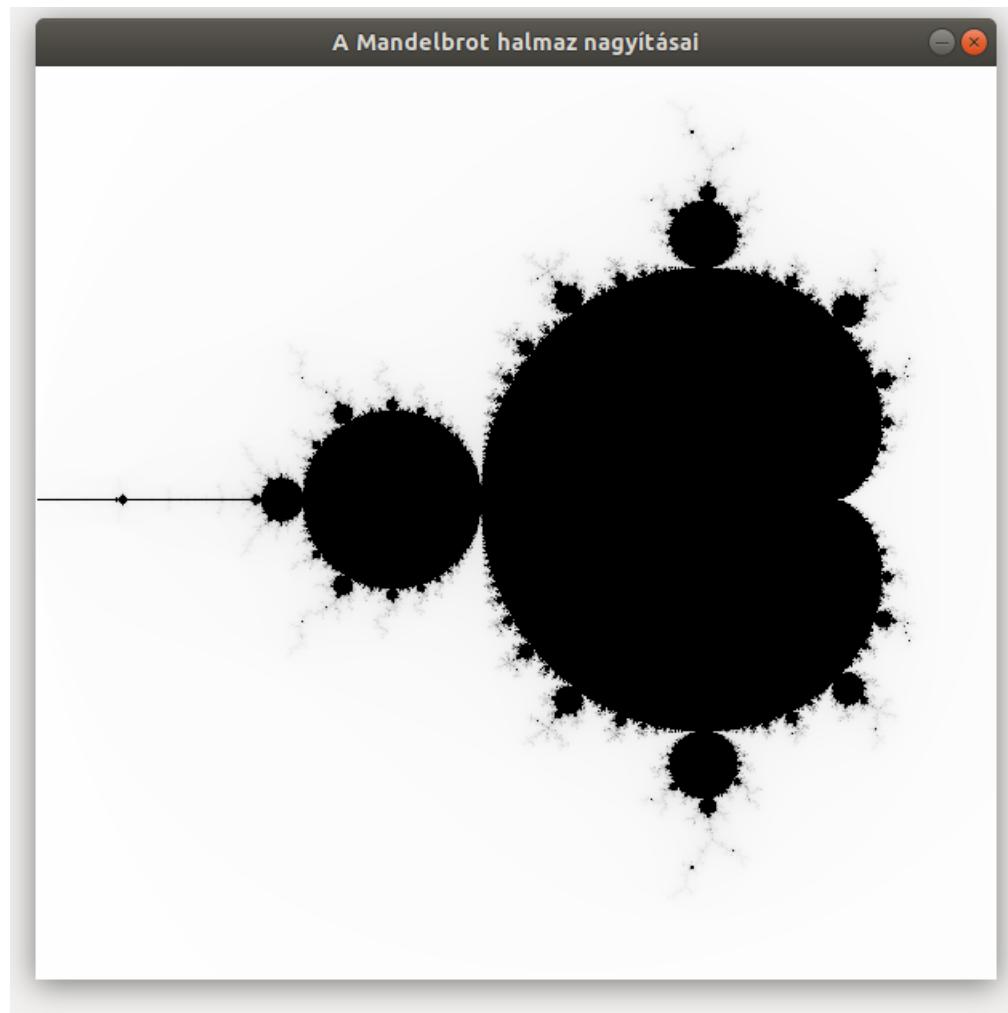
Az encoding kapcsolóval fordítjuk forrásainkat és:

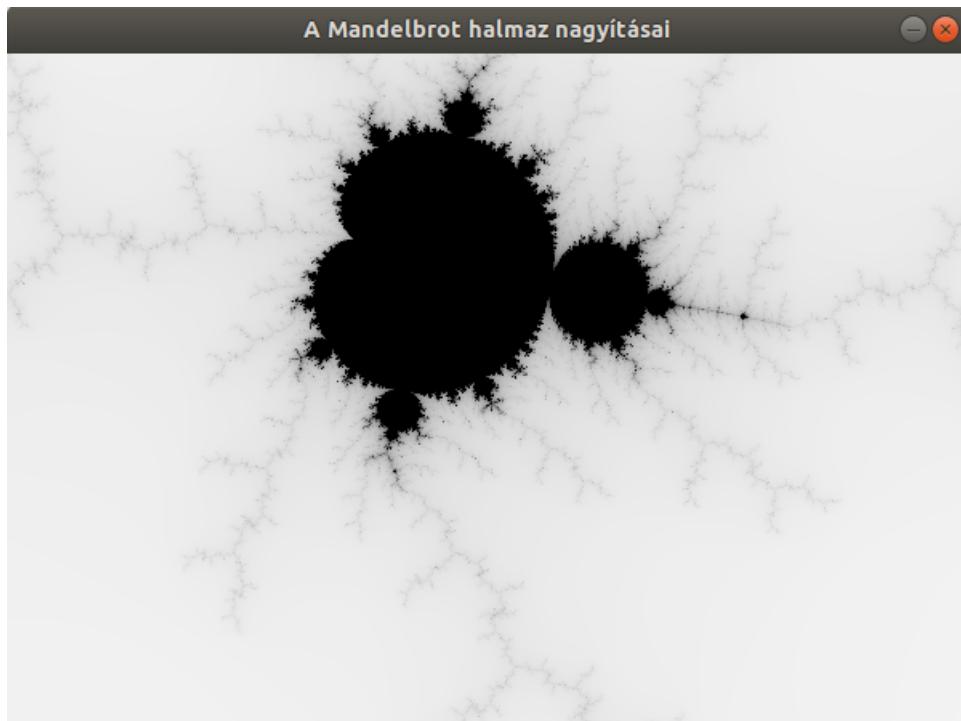
```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/encoding
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/encoding$ javac -encoding ISO8859_2 MandelbrotIterációk.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/encoding$ javac -encoding ISO8859_2 MandelbrotHalmazNagyító.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/encoding$ java MandelbrotHalmazNagyító
```

A kódok minden további probléma nélkül lefordultak.



Láthatjuk, hogy amint lefutott a program, meggyarapodtak a könyvtárban található fájljaink. Ez jó jel, nézzük, működik-e a nagyítás!





Amint az az előző képen látható, úgy működik, ahogy mi azt elvártuk tőle. Ezzel elkészült a feladat.

## 14.2. I334d1c4 - deprecated

Írj olyan OO Java vagy C++ osztályt, amely leet cipherként működik, azaz megvalósítja ezt a betű helyettesítést: <https://simple.wikipedia.org/wiki/Leet> (Ha ez első részben nem tettet meg, akkor írasd ki és magyarázd meg a használt struktúratömb memória foglalását!)

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/chomsky2\\_codes/leet](https://gitlab.com/kincsa/bhax/tree/master/codes/chomsky2_codes/leet)

Tanulságok, tapasztalatok, magyarázat...

Tutor: Harmati Norbert

A "leet speak" kommunikáció során a betűket, számokat azok eredeti formájukra hasonlító karakterrel/karakterekkel helyettesítik, így ellehetetlenítve (vagy legalábbis nagyon megnehezítve) a kódolást nem ismerő személy számára az olvasást. A '80-as években alakult ki, ettől függetlenül még mind a mai napig előszörrel használják az online világban.

Maga a leet nem ismeretlen számunkra, hiszen előző féléves, Prog. 1-es tanulmányaink során már találkoztunk vele. A különbség az akkori és a mostani feladat között az, hogy míg legutóbb a Lex készítette el számunkra a kódot az általunk megadott szabályok alapján, most nekünk kell a kódot is megírnunk. A programunk Java-ban készült. Lássuk a programkódot, hogyan is sikerült megvalósítanunk!

```
import java.util.*;

class LeetCypher
{
 private static String atalakitando = new String();
```

```
private static String leet = new String();

Map<String, String> character = new HashMap<String, String>();

public void print(String atalakitando)
{
 System.out.println(atalakitando);
}
```

Importálunk minden Java könyvtárat, így biztosan rendelkezésünkre fog állni majd minden a feladat sikeres megoldásához. A LeetCypher osztályban létrehozunk két Stringet, az egyikben az átalakítandó, általunk megadott, míg a másikban a már átalakított szöveg lesz eltárolva. Egy Map az az adatszerkezet, amelyben az egyes karakterek és l33t-ábécé-beli megfelelői lesznek tárolva. A print metódussal fogjuk eredményünket kiiratni.

```
public void atalakit(String szo)
{
 character.put("A", "4");
 character.put("a", "4");
 character.put("B", "8");
 character.put("b", "8");
 character.put("C", "<");
 character.put("c", "<");
 character.put("D", "|)");
 character.put("d", "o|");
 character.put("E", "3");
 character.put("e", "3");
 character.put("F", "|=");
 character.put("G", "(");
 character.put("g", "9");

 ...
 ...
 ...
 for (int i = 0; i < szo.length(); i++)
 if (character.get(szo.substring(i, i + 1)) != null)
 atalakitando += character.get(szo.substring(i, i + 1)) + " ";
}
;
```

Érkezik a atalakit függvényünk. Feltöljük a character névre hallgató Map-ünket a megfelelő párokkal. Majd egy for ciklussal végigmegyünk a megadott szó betűin és minden egyes betűhöz tartozó l33t-beli megfelelőt eltároljuk egy változóban, szóközzel elválasztva.

```
public LeetCypher()
{

 atalakit(atalakitott);
 print(atalakitando); //átalakítva
```

```
 }

 public static void main(String args[])
 {
 StringBuilder builder = new StringBuilder(atalakitott.length());
 for (String str : args)
 {
 builder.append(str + ' ');
 }

 atalakitott = builder.toString();

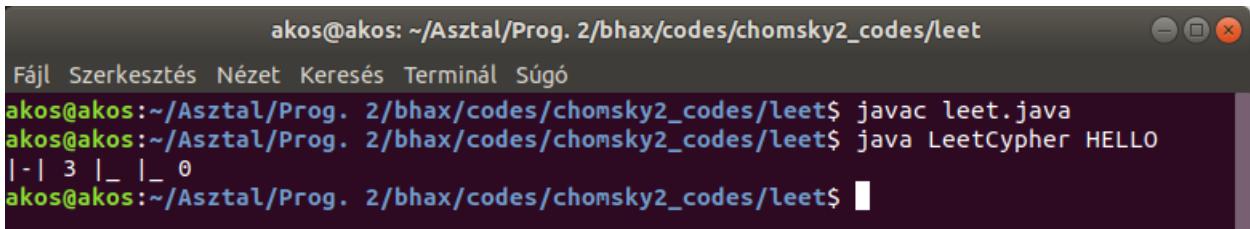
 new LeetCypher();
 }
}
```

Létrejön a konstruktorkiművek is, melyben meghívásra kerül a l33t-átalakításhoz szükséges metódus, majd az eredmény kiíratása is.

És már a main-ben járunk. Létrehozunk egy Stringbuildert, amely segítségével lényegében felépítjük a már átalakított stringünket karakterenként, hiszen egy for ciklus segítségével megyünk végig a parancssori argumentumként megadott szavunkon. Majd eltároljuk ezt, az átalakított szöveget egy String változóban.

Végül létrehozásra kerül egy új LeetCypher objektum is.

Most lássuk akkor futás közben is!



```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/leet
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/leet$ javac leet.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/leet$ java LeetCypher HELLO
| - | 3 | _ | _ 0
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/leet$
```

Látható, hogy fordítás és futtatás után az általam megadott haxor szót sikeresen átalakította a programocska.

### 14.3. Full screen - deprecated

Készítsünk egy teljes képernyős Java programot! Tipp: [https://www.tankonyvtar.hu/en/tartalom/tkt/javatanitok-javat/ch03.html#labirintus\\_](https://www.tankonyvtar.hu/en/tartalom/tkt/javatanitok-javat/ch03.html#labirintus_)

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/chomsky2\\_codes/fullscreen](https://gitlab.com/kincsa/bhax/tree/master/codes/chomsky2_codes/fullscreen)

Tanulságok, tapasztalatok, magyarázat...

A feladat megoldásához alapjául szolgáló eredeti kód: [itt](#) található.

Lássuk, hogyan is épül fel a kis primitív programunk!

```
import java.awt.*;
import javax.swing.JFrame;
import javax.swing.JPanel;
```

Importáljuk a grafikus ablakban való megjelenítéshez és annak beállításához szükséges osztályokat.

```
public class fullscreen extends JPanel{

 String uzenet = "Teljes kepernyos program";

 public void paint(Graphics g)
 {
 g.setFont(new Font("TimesRoman", Font.BOLD, 56));
 g.setColor(Color.red);
 g.drawString(uzenet, 540, 540);

 }
}
```

Létrehozásra kerül a `fullscreen` osztályunk ami az előbb importált `JPanel` osztály leszármazottja lesz. Egy stringben megadjuk a kiiratásra szánt szöveget majd a `paint` metódusban attribútumként megadott objektumpéldány megjelenítését módosítjuk a függvénytörzsben.

A `setFont` a szöveg karaktereinek megjelenését hivatott beállítani. A paraméterek a következők: betűtípus, típus(esetünkben félkövér) és betűméret. A `setColor` értelemszerűen a kiiratandó szöveg színét változtatja meg. A `drawString` függvénykel pedig egy adott stringet adott koordinákon jeleníthetünk meg.

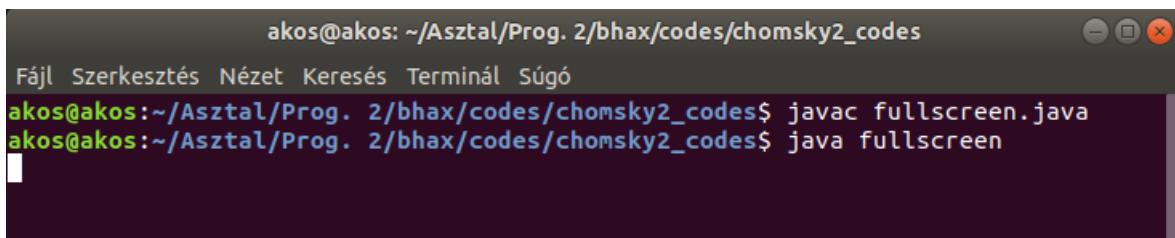
```
public static void main(String[] args)
{
 JFrame frame = new JFrame("Full screen program");

 frame.getContentPane().add(new fullscreen());

 frame.setSize(1920, 1080);
 frame.setVisible(true);
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frame.setResizable(false);
}
```

Már a `main` függvényben vagyunk. Példányosítunk egy `JFrame` példányt, aminek paramétere az a string lesz, mely a programunk ablakának "neve" lesz. Hozzáadjuk a frame-ünkhöz a `fullscreen` osztályunk egy példányát majd beállításra kerül a frame mérete pixelben a `setSize` függvényel, a láthatósága a `setVisible` függvényel, a viselkedés amit tanúsít, amikor bezárjuk az adott ablakot a `setDefaultCloseOperation`-nál - ami esetünkben nem jelent mást, mint kilépést a programból -, illetve az ablak átmérethetőségét a `setResizable` függvényel.

Fordítsuk és futtassuk!



A screenshot of a terminal window titled "akos@akos: ~/Asztal/Prog. 2/bhax/codes/chomsky2\_codes". The window shows the command "javac fullscreen.java" being run, followed by "java fullscreen". The terminal is in full-screen mode, indicated by a "Full screen program" status bar at the bottom.

## Teljes kepernyős program

Azt teszi a program, amit tennie kell!

Másik megoldás - Tutor: Heinrich László

Egy másik feladatmegoldás melynek összehozásában Heinrich László tutorként segédkezett.

```
import java.awt.*;
import java.awt.image.BufferStrategy;

public class BouncingBall {
 int x = 450;
 int y = 450;
 int radius = 50;
 int tempX, tempY;
 int maxX, maxY;

 private static DisplayMode[] BEST_DISPLAY_MODES = new DisplayMode []
 [
 new DisplayMode(1920, 1080, 32, 0),
 new DisplayMode(1920, 1080, 16, 0),
 new DisplayMode(1920, 1080, 8, 0)
];
}
```

```
};
```

A program elején importálunk minden szükséges könyvtárat a grafikus megjelenítéshez. Majd elérkezünk a BouncingBall osztályunkba, ahol a mozgást és annak "határait" definiáló változók kerülnek létrehozásra. Létrehozásra kerül egy `DisplayMode` típusú tömb is, amiben a képernyőtípusok kerülnek eltárolásra, számszerint 3.

```
Frame mainFrame;

public void move()
{
 tempY = (int)(tempY + 1) % (int)(2 * (maxY - radius));
 y = y + (int)Math.pow(-1, Math.floor(tempY / (maxY - radius)));
 tempX = (int)(tempX + 1) % (int)(2 * (maxX - radius));
 x = x + (int)Math.pow(-1, Math.floor(tempX / (maxX - radius)));
}
```

Létrehozunk egy `Frame` objektumot és a `move` metódust is, amellyel a labda mozgatását végezzük a képernyőn. Ehhez a már előzőleg tárgyalta változóinkat is felhasználjuk.

```
public BouncingBall (int numBuffers, GraphicsDevice device)
{
 try
 {
 GraphicsConfiguration gc = device.getDefaultConfiguration() ←
 ;
 mainFrame = new Frame(gc);
 mainFrame.setUndecorated(true);
 mainFrame.setIgnoreRepaint(true);
 device.setFullScreenWindow(mainFrame);
 if (device.isDisplayChangeSupported())
 {
 chooseBestDisplayMode(device);
 }
 Rectangle bounds = mainFrame.getBounds();
 bounds.setSize(device.getDisplayMode().getWidth(), ←
 device.getDisplayMode().
 getHeight());
 maxX = device.getDisplayMode().getWidth();
 maxY = device.getDisplayMode().getHeight();
 tempX = x;
 tempY = y;
 mainFrame.createBufferStrategy(numBuffers);
 BufferStrategy bufferStrategy = mainFrame. ←
 getBufferStrategy();
 while(true)
 {
 Graphics g = bufferStrategy.getDrawGraphics();
 if (!bufferStrategy.contentsLost()) {
 move();
 g.setColor(Color.white);
```

```
 g.fillRect(0, 0, bounds.width, bounds.height);

 g.setColor(Color.blue);
 g.fillOval(x, y, radius, radius);
 bufferStrategy.show();
 g.dispose();
 }
 try
 {
 Thread.sleep(5);
 }
 catch (InterruptedException e) {
 }
}
catch (Exception e)
{
 e.printStackTrace();
} finally
{
 device.setFullScreenWindow(null);
}
}
```

Létrehozunk egy BouncingBall objektumot két paraméterrel. Egy try-catch szerkezet az, ami ebben a blokkban megtalálható. Létrehozzuk a Frame-üket, melynek paramétere a gc, grafikai konfigurációs fájl lesz. A képernyőt mindenekelőtt "tisztítjuk", hogy induláskor a kis labdánkon kívül ne legyen rajta semmi. A teljesség igénye nélkül: a színek, a kép váltásának gyakorisága is beállításra kerül.

```
private static DisplayMode getBestDisplayMode(GraphicsDevice device -->
)
{
 for (int x = 0; x < BEST_DISPLAY_MODES.length; x++)
 {
 DisplayMode[] modes = device.getDisplayModes();
 for (int i = 0; i < modes.length; i++)
 {
 if (modes[i].getWidth() == BEST_DISPLAY_MODES[x].getWidth() && modes[i].getHeight() == BEST_DISPLAY_MODES[x].getHeight() && modes[i].getBitDepth() == BEST_DISPLAY_MODES[x].getBitDepth())
 {
 return BEST_DISPLAY_MODES[x];
 }
 }
 }
 return null;
}
```

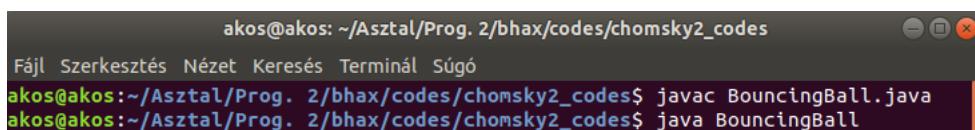
A `getBestDisplayMode`, egy paraméteres függvény melyben végigmegyünk a program elején létrehozott tömbön végigmegyünk. Lényege, hogy az eszközünkhez leginkább megfelelő képmódot adja vissza.

```
public static void chooseBestDisplayMode(GraphicsDevice device)
{
 DisplayMode best = getBestDisplayMode(device);
 if (best != null)
 {
 device.setDisplayMode(best);
 }
}

public static void main(String[] args)
{
 try
 {
 int numBuffers = 2;
 GraphicsEnvironment env = GraphicsEnvironment.
 getLocalGraphicsEnvironment();
 GraphicsDevice device = env.getDefaultScreenDevice();
 BouncingBall ball = new BouncingBall(numBuffers, device);
 }
 catch (Exception e)
 {
 e.printStackTrace();
 }
 System.exit(0);
}
```

A `chooseBestDisplayMode` függvény az, amellyel beállítjuk eszközünknek a képmódját.

És már a mainban járunk. Szintén egy try-catch szerkezetben vagyunk hiszen most is gondolunk a hibakezelésre. A try részben létrehozzuk objektumainkat vagyis példányosítunk.



The screenshot shows a terminal window with the following text:

```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/chomsky2_codes
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes$ javac BouncingBall.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes$ java BouncingBall
```

A program futásáról sajnos nem tudtam ss-t csatolni (fehér képernyőt kapok).

## 14.4. Perceptron osztály

Dolgozzuk be egy külön projektbe a projekt Perceptron osztályát! Lásd <https://youtu.be/XpBnR31BRJY>

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/chomsky2\\_codes/perceptron](https://gitlab.com/kincsa/bhax/tree/master/codes/chomsky2_codes/perceptron)

Tanulságok, tapasztalatok, magyarázat...

Érdemes lenne először a fogalmat megmagyarázni. A perceptron nem más mint ezen neuron mesterséges intelligenciában használt változata. Tanulásra képes, a bemenő 0-k és 1-esek sorozatából mintákat tanul meg és súlyozott összegzést végez.

A következő feladat során egy ilyen perceptronról fogunk elkészíteni, aminek esetünkben a feladata az lesz, hogy a mandelbrot.cpp programunk által létrehozott Mandelbrot-halmazt ábrázoló PNG kép egyik színkódját vegye és az a színkód legyen a többrétegű neurális háló inputja. Lássuk a programot!

```
#include <iostream>
#include "mlp.hpp"
#include <png++/png.hpp>
```

Include-oljuk az iostream, az mlp és a png++/png könyvtárakat. Utóbbi kettőt azért, mert többrétegű perceptronról akarunk majd létrehozni (Multi Layer Perceptron), így muszáj ezt a könyvtárat include-olunk a programunkba. Utolsó könyvtárunk ahogyan a neve is sugallja, a PNG képállományokkal való munkát teszi lehetővé. Előzőleg telepíteni szükséges, ha nem megtalálható a gépünkön.

```
using namespace std;

int main(int argc, char **argv)
{
 png::image<png::rgb_pixel> png_image(argv[1]);

 int size = png_image.get_width() * png_image.get_height();

 Perceptron *p = new Perceptron(3, size, 256, 1);
```

Kezdődik a main függvényünk. Első sorában megmondjuk, hogy az 1-es parancssori argumentum alapján kerül beolvasásra a képállomány. Ettől kezdve dolgozni tudunk. A kép méretét a get\_width és a get\_height szorzatából kapjuk, ezt el is tároljuk egy változóban. A következő sorban létrehozásra példányosítunk egy perceptronról a new operátor segítségével, amely paramétereit balról jobbra haladva: a rétegek száma, 1. réteg neuronjai az inputrétegen, 2. réteg neuronjai az inputrétegen, az eredmény (jelen esetben 1 szám)

```
double* image = new double[size];

for(int i=0; i<png_image.get_width(); ++i)
 for(int j=0; j<png_image.get_height(); ++j)
 image[i*png_image.get_width() + j] = png_image[i][j].red;

double value = (*p)(image);

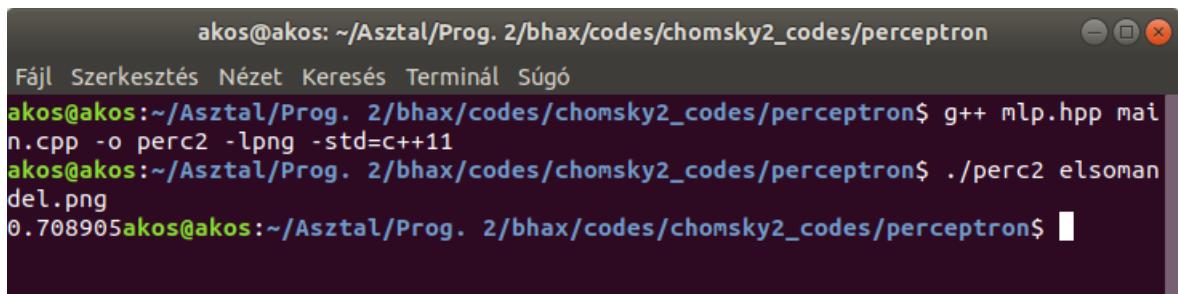
cout << value;

delete p;
delete [] image;
```

Létrehozásra kerül egy double típusú mutató. Jönnek a for ciklusok. Az egyik for ciklus végigmegy a kép szélességét alkotó pontokon, a másik pedig a magasságán. Miután végigmentünk a képpontokon, az image tárolni fogja a képállomány vörös színkomponensét. Tehát a beolvasásra került képállomány

piros vörös komponensét a lefoglalt tárba másoljuk bele. A `value` értéke a Perceptron `image`-re történő meghívása adja majd. Így a perceptronban tárolásra kerül a vörös színkomponens. A `value` változó egy double típusú értéket tárol. Ez kiiratásra kerül és töröljük tovább nem használatos elemeket, hiszen a számukra eddig fenntartott memóriaterületet érdemes felszabadítanunk, így a lefoglalt memóriamennyiséget újra használható. Programunk ezzel véget is ért.

Fordítjuk és futtatók a képen látható módon:



```
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/perceptron
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/perceptron$ g++ mlp.hpp main.cpp -o perc2 -lpng -std=c++11
akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/perceptron$./perc2 elsoman del.png
0.708905akos@akos:~/Asztal/Prog. 2/bhax/codes/chomsky2_codes/perceptron$
```

# 15. fejezet

## Helló, Stroustrup!

### 15.1. JDK osztályok

Írunk olyan Boost C++ programot (indulj ki például a fénykardból) amely kilistázza a JDK összes osztályát (miután kicsomagoltuk az src.zip állományt, arra ráengedve)!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/stroustrup\\_codes/jdk](https://gitlab.com/kincsa/bhax/tree/master/codes/stroustrup_codes/jdk)

Tutor: [GitHub](#)

Tanulságok, tapasztalatok, magyarázat...

A segítségünkre a C++ Boost könyvtárai lesznek segítségünkre. [Boost C++ libraries](#)

A feladat megkezdését természetesen a leírás által is tanácsolt fénykard program megkereséssel kezdtem. Bátfai Tanár Úr FUTURE projektjének programkódjai között sikeresült is rátalálnom: <https://github.com/nbatfai/future/blob/master/cs/F7/fenykard.cpp>

Ami nekünk ebből a kódból érdekes lesz, az a következő, `read_acts` függvény:

```
void read_acts(boost::filesystem::path path, std::map<std::string, int> &acts)
{
 if (is_regular_file(path)) {
 std::string ext(".props");
 if (!ext.compare(boost::filesystem::extension(path))) {
 std::string actpropspath = path.string();
 std::size_t end = actpropspath.find_last_of("/");
 std::string act = actpropspath.substr(0, end);

 acts[act] = get_points(path);

 std::cout << std::setw(4) << acts[act] << " " << act <<
 std::endl;
 }
 }
}
```

```
 }

} else if (is_directory(path))
 for (boost::filesystem::directory_entry & entry : boost::filesystem::directory_iterator(path))
 read_acts(entry.path(), acts);

}
```

Mit csinál ez a függvény? Végigmegy egy állományszerkezeten és egy adott kiterjesztésű (esetben .props) fájlokat keres, majd a bennük található információkat kiolvassa és eltárolja.

Ha jobban belegondolunk, lényegében most is valami hasonlót kell csinálnunk. Hiszen egy állományszerkezetet kell bejárunk és szintén egy adott kiterjesztésű (.java) fájlokat keresünk.

Lássuk, a fénykardos példából kiindulva hogyan is sikerülne ezt megvalósítani!

```
#include <iostream>
#include <string>
#include <fstream>
#include <iomanip>
#include <vector>

#include <boost/filesystem.hpp>

using namespace std;

int counter=0;
```

Inkludáljuk a megfelelő osztályokat és deklarálunk egy változót az osztályok számának tárolására.

```
void read_classes (boost::filesystem::path path, vector<string> & acts)
{
 if(is_regular_file(path))
 {
 string ext(".java");
 if(!ext.compare(boost::filesystem::extension (path)))
 {
 cout<<path.string()<<'\n';
 string actjavaspath=path.string();
 size_t end = actjavaspath.find_last_of("/");
 string act = actjavaspath.substr(0,end);
 acts.push_back(act);
 counter++;
 }
 }
 else if(is_directory(path))
 for(boost::filesystem::directory_entry & entry :
 boost::filesystem::directory_iterator (path))
 read_classes(entry.path(),acts);
```

```
}
```

Az eljárás (aminek egyébként két paramétere van: az elérési hely - vagyis ahol keresnie kell - illetve a keresett elemeket tároló vector) a következőképpen működik:

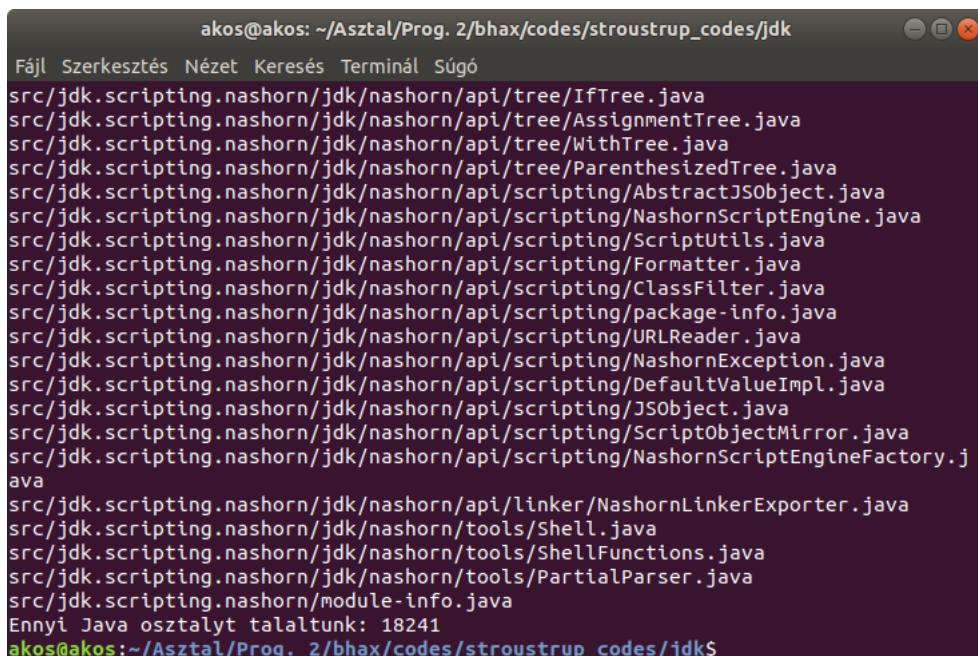
Alapvetően egy feltételvizsgálat következik, aminek két ága van. Az if-fel bevezetett feltételvizsgálatban, azon belül pedig az `is_regular_file()` függvénytel megvizsgáljuk, hogy a fájlhierarchyben jelenleg "hol állunk" .. Tehát hogy ha ún. "regular fájlok" vagyis hagyományos fájlok, nem könyvtárak állnak rendelkezésre vizsgálatra, akkor megvizsgáljuk, mely fájlok kiterjesztése .java. Ezeket az állományokat a vectorba gyűjtiük és növeljük a számlálót. Eltároljuk a keresett fájlok elérési útvonalát is két segédváltozó (end és act javaspath) segítségével.

Ha nem fájlokkal vagyunk körülvéve ott, ahol jelenleg állunk, meghívásra kerül a `is_directory` függvény, tehát ez az a helyzet, mikor még a könyvtárak szintjén állunk a "kutakodásban" .. Ekkor egy for ciklus indul és rekursívan meghívjuk a `read_classes` függvényt. Így előbb-utóbb eljutunk a fájlok szintjére. (Már ha nem vagyunk ott jelenleg..)

```
int main(int argc, char *argv[])
{
 vector<string> acts;
 read_classes("src", acts);
 cout << "Ennyi Java osztályt találtunk: " << counter << "\n";
}
```

A `main` függvényben járunk már. Létrehozásra kerül az osztályokat tároló vectorunk, meghívásra kerül a `read_classes` függvény és kiiratjuk, hány osztály is volt megtalálható az `src` mappában.

Fordítunk és futtatunk:



```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/stroustrup_codes/jdk
Fájl Szerkesztés Nézet Keresés Terminál Súgó
src/jdk.scripting.nashorn/jdk/nashorn/api/tree/IfTree.java
src/jdk.scripting.nashorn/jdk/nashorn/api/tree/AssignmentTree.java
src/jdk.scripting.nashorn/jdk/nashorn/api/tree/WithTree.java
src/jdk.scripting.nashorn/jdk/nashorn/api/tree/ParenthesizedTree.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/AbstractJSObject.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/NashornScriptEngine.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/ScriptUtils.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/Formatter.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/ClassFilter.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/package-info.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/URLReader.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/NashornException.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/DefaultValueImpl.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/JSObject.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/ScriptableObjectMirror.java
src/jdk.scripting.nashorn/jdk/nashorn/api/scripting/NashornScriptEngineFactory.java
src/jdk.scripting.nashorn/jdk/nashorn/api/linker/NashornLinkerExporter.java
src/jdk.scripting.nashorn/jdk/nashorn/tools/Shell.java
src/jdk.scripting.nashorn/jdk/nashorn/tools/ShellFunctions.java
src/jdk.scripting.nashorn/jdk/nashorn/tools/PartialParser.java
src/jdk.scripting.nashorn/module-info.java
Ennyi Java osztályt találtunk: 18241
akos@akos:~/Asztal/Prog. 2/bhax/codes/stroustrup_codes/jdk$
```

## 15.2. Hibásan implementált RSA törése

Készítünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: [https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2\\_3.pdf](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_3.pdf) (71-73 fólia) által készített titkos szövegen.

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/stroustrup\\_codes/RSA](https://gitlab.com/kincsa/bhax/tree/master/codes/stroustrup_codes/RSA)

Tanulságok, tapasztalatok, magyarázat...

Tutor: [Szilágyi Csaba](#)

Bevezető forrásai: <https://hu.wikipedia.org/wiki/RSA-elj%C3%A1r%C3%A1s>, [https://www.tankonyvtar.hu/hu/tartalom/tamop425\\_0046\\_informatikai\\_biztonsag\\_es\\_kriptografia/ch08s07.html](https://www.tankonyvtar.hu/hu/tartalom/tamop425_0046_informatikai_biztonsag_es_kriptografia/ch08s07.html)

Először pár szóban az RSA-ról: napjaink egyik legszélesebb körben használt titkosító eljárása. Az eljárás neve lényegében egy mozaikszó, az algoritmus alkotói: Ron Rivest, Adi Shamir és Len Adleman kezdőbetűiből áll össze, 1978-ban publikálták. Működése matematikai alapokon, pontosabban a Fermat-tételen nyugszik. Érdekes jellemzője az algoritmusnak, hogy két kulccsal, egy nyílttal vagy nyilvánossal és egy titkos kulcssal dolgozik. Mindkét kulcs egy-egy számpár. Ez azért érdekes, mert ezzel teljesen ketté lesz választva a titkosítás és a törés folyamata. Mivel a kódolás és a dekódolás paramétereinek nem lesznek ugyanazok, így az egyik meghatározása a másikból sem lesz megoldható.

Az RSA titkosításról minden további tudnivaló [itt](#)

Fontos említést tennünk a BigInteger osztályról is, amit a feladat során természetesen használni is fogunk. A BigInteger osztály már egész rége óta a JDK részét képezi, jelenleg a java.math csomagban találhatjuk meg. Gyakran használják kriptográfiai kódolók-törők elkészítéséhez, sokak számára használatuk összeolvadt ezzel a fajta felhasználással. A típus egy egész értékből és 32-bites egészből úgynevezett skálázó faktor alkotja.

Lássuk a programunkat!

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
```

Először is importálunk minden olyan könyvtárat, melyekre a feladat megoldása során szükségünk lesz. Itt található a bevezetőben már említett BigInteger osztály is.

```
public class rsa_cipher {
 public static void main(String[] args) {
 int bitlength = 2100;

 SecureRandom random = new SecureRandom();

 BigInteger p = BigInteger.probablePrime(bitlength/2, random);
 BigInteger q = BigInteger.probablePrime(bitlength/2, random);
```

```
BigInteger publicKey = new BigInteger("65537");
BigInteger modulus = p.multiply(q);

String str = "this is a perfect string".toUpperCase();
System.out.println("Eredeti: " + str);
```

Haladjunk logikai sorrendben. A mainünket tartalmazó, rsa\_cipher osztályban járunk. Itt történik a bithossz beállítása, itt kerülnek létrehozásra a BigInteger számaink. Ilyen érték lesz többek között a nyilvános kulcsunk és a modulus is. Megadjuk eredeti, átalakításra váró szövegünköt, amit a str nevű, String típusú változóban el is tárolunk.

```
byte[] out = new byte[str.length()];
for (int i = 0; i < str.length(); i++) {
 char c = str.charAt(i);
 if (c == ' ')
 out[i] = (byte)c;
 else
 out[i] = new BigInteger(new byte[] { (byte)c }).modPow(publicKey, ←
 modulus).byteValue();
}
String encoded = new String(out);
System.out.println("Kodolt: " + encoded);

Decode de = new Decode(encoded);
System.out.println("Visszafejtett: " + de.getDecoded());
}
```

A mainből kiemelt pár sor végzi a titkosítást, még hozzá a stringünk minden egyes karakterén végighaladva, egyenként teszi meg azt. Egy byte elemeket tartalmazó tömbben kerülnek tárolásra a titkosított karakterek, melyekből stringet készítünk, neve encoded lesz melyet ki is iratunk. A következő sorban létrehozunk egy új Decode objektumot, ami a már gyakoriság alapon visszafejtett stringünket tartalmazza. Kérdés az, hogyan is működik ez a fajta visszafejtés. Vessünk egy pillantást a következő függvény(ek)re!

```
private void loadFreqList() {
 BufferedReader reader;
 try {
 reader = new BufferedReader(new FileReader("gyakorisag.txt"));
 String line;
 while((line = reader.readLine()) != null) {
 String[] args = line.split("\t");
 char c = args[0].charAt(0);
 int num = Integer.parseInt(args[1]);
 this.charRank.put(c, num);
 }
 } catch (Exception e) {
 System.out.println("Error when loading list -> " + e.getMessage());
 }
}
```

A függvényen belül egy try-catch szerkezetben történik a lényeg. Ugyanis beolvasásra kerül egy gyakoriság lista. Karakterenként végigmegy a lista elemein és abban az esetben, ha az aktuális betű már megtalálható benne, a gyakorisága növelésre kerül. Ha új betűvel találkozik, gyakorisága 1-re állítódik be. Mindez a try-on belül volt megtalálható. Catch-csel hibaüzenetet dobunk ha nem sikerült a listát betöltenünk.

```
private char nextFreq() {
 char c = 0;
 int nowFreq = 0;
 for(Entry<Character, Integer> e : this.charRank.entrySet()) {
 if (e.getValue() > nowFreq) {
 nowFreq = e.getValue();
 c = e.getKey();
 }
 }
 if (this.charRank.containsKey(c))
 this.charRank.remove(c);
 return c;
}
```

Lássuk, mit tesz a nextFreq függvényünk. Lényegében itt nem történik más, mint a gyakorisági listában lévő betűk helyettesítése történik szüvegünkben. Láthatjuk, hogy ahogy az várható volt, alapvetően a gyakorisági listánk az, ami befolyásolja majd a kapott eredményünket, ugyanis a nagyobb gyakorisági értékkel rendelkező karaktereket helyezi előtérbe majd az algoritmus.

```
public Decode(String str) {
 this.charRank = new HashMap<Character, Integer>();
 this.decoded = str;

 this.loadFreqList();

 HashMap<Character, Integer> frequency = new HashMap<Character, Integer <-> ();
 for (int i = 0; i < str.length(); i++) {
 char c = str.charAt(i);
 if (c != ' ')
 if(frequency.containsKey(c))
 frequency.put(c, frequency.get(c) + 1);
 else
 frequency.put(c, 1);
 }

 while (frequency.size() > 0) {
 int mi = 0;
 char c = 0;
 for (Entry<Character, Integer> e : frequency.entrySet()) {
 if (mi < e.getValue()) {
 mi = e.getValue();
 c = e.getKey();
 }
 }
 str = str.replaceFirst(" " + c, "");
 frequency.remove(c);
 }
}
```

```
 }
 }
 thisdecoded = thisdecoded.replace(c, this.nextFreq());
 frequency.remove(c);
}
}
```

Már csak a mainben meghívott Decode objektum tárgyalása maradt hátra. Nem túlbonyolítva, ez az az objektum, amely a már előzőleg tárgyalt függvényeket és gyakorisági listát felhasználva végzi el a dekódolást. Egy ilyen objektum kerül deklarálásra a main-ünkben is, melyet a feladat elején már láthattunk.

Lássuk működés közben!

```
akos@akos:~/Asztal/Prog. 2/bhax/codes/stroustrup_codes/RSA$ javac rsa_cipher.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/stroustrup_codes/RSA$ java rsa_cipher
Eredeti: THIS IS A PERFECT STRING
Kodolt:♦♦♦@ ♦@ [♦] [♦ @♦♦♦]
Visszafejtett: IIIS IS A ETRTII SIIICI
```

### 15.3. Változó argumentumszámú ctor és Összefoglaló

Készítsünk olyan példát, amely egy képet tesz az alábbi projekt Perceptron osztályának bemenetére és a Perceptron ne egy értéket, hanem egy ugyanakkora méretű „képet” adjon vissza. (Lásd még a 4 hét/Perceptron osztály feladatot is.)

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/stroustrup\\_codes/perceptron](https://gitlab.com/kincsa/bhax/tree/master/codes/stroustrup_codes/perceptron)

Tutor:<https://drive.google.com/file/d/12ahRzjFAYkBWgYpLjjAIWxhHzq6XKFle/view?fbclid=IwAR2UaaLT5D>

Tanulságok, tapasztalatok, magyarázat...

Források:

[http://www.cs.ubbcluj.ro/~csatol/mestint/pdfs/neur\\_halo\\_alap.pdf](http://www.cs.ubbcluj.ro/~csatol/mestint/pdfs/neur_halo_alap.pdf)

[https://www.tankonyvtar.hu/hu/tartalom/tamop425/0026\\_neuralis\\_4\\_4/ch04.html](https://www.tankonyvtar.hu/hu/tartalom/tamop425/0026_neuralis_4_4/ch04.html)

<https://gyires.inf.unideb.hu/GyBITT/19/ch03s02.html>

Érdemes lenne először a fogalmakat megmagyarázni.

Először is, hogyan lehet egy konstruktor változó argumentumszámú és hogyan jelenik meg ez a mi példánkban?

```
class Perceptron
{
public:
 Perceptron (int nof, ...)
{
```

Változó argumentumszámú konstruktőről akkor beszélhetünk, mikor nincsen egyértelműen definiálva, egy konstruktornak hány paraméterrel kell rendelkeznie. A mi programunk mlp.hpp header-fájlijában is ezzel találkozhatunk. Ebből annyit olvashatunk le, hogy egy argumentumot fixen tartalmaznia kell (ez esetünkben az Integer típusú nof lesz), a változó argumentumszámot pedig a "..." karakter sorozat jelzi számunkra.

A perceptron nem más mint ezen neuron mesterséges intelligenciában használt változata. Szintén tanulásra képes, a bemenő 0-k és 1-esek sorozatából mintákat tanul meg és súlyozott összegzést végez. Ennek több változata is létezik, mi ezen feladat során a többrétegűekkel fogunk foglalkozni, ez a Multi Layer Perceptron (MLP), amely az egyik, ha nem a leggyakrabban használt hálózat-architektúra. Ez a fajta neurális hálózat 3 rétegből épül fel, melyek a következők: bemeneti réteg: azok a neuronok találhatók itt, amelyek a bemeneti jel továbbítását végzik a hálózat felé. A legtöbb esetben nem jelöljük őket külön; rejtett réteg: a tulajdonképpeni feldolgozást végző neuronok tartoznak ide. Egy hálózaton belül több rejtett réteg is lehet; kimeneti réteg: az itt található neuronok a körülbelül felé továbbítják az információt. A feladatuk ugyanaz, mint a rejtett rétegbeli neuronoké.

Amint már említettem, a többrétegű perceptron minden egyes rétege neuronokból áll. Biztosítani kell azonban, hogy a neurális hálózatunk kimenete a súlyok folytonos, differenciálható függvénye legyen.

Fontos az is, hogyan történik a neurális hálónk tanítása. A tanítási folyamat egy ún. ellenőrzött tanítás, ahol a hálózat kimenetén értelmezett hiba felhasználásával határozzuk meg a kritériumfüggvény vagy kockázat paraméterfüggését. A tanítás leggyakrabban a hiba-visszaterjesztés módszerrel valósul meg. A tanítás fő lépései: a kezdeti súlyok megadása; a bemeneti jelet (azaz a tanító pontot) végigáramoltatjuk a hálózaton, de a súlyokat nem változtatjuk meg; Az így kapott kimeneti jelet összevetjük a tényleges kimeneti jellel; A hibát visszaáramoltatjuk a hálózaton, súlyokat pedig megváltoztatjuk a hiba csökkentése érdekében.

A következő feladat során egy ilyen perceptron fogunk elkészíteni, aminek alapvetően a feladata az, hogy a mandelbrot.cpp programunk által létrehozott Mandelbrot-halmazt ábrázoló (elsomandel.png) PNG képet felhasználva generáltassunk egy vele megegyező méretű képet. Lássuk a programot!

Kezdjük a feladat értelmezésével. A feladat szinte ugyanazt kéri, mint az előző Perceptronus feladatainkál egy kis csavarral. Ugyanis az eddigi példáink során a Perceptronunk egy képállományra egy értéket adott vissza.

A kód eleje megegyezik az előző fejezetben már tárgyaltéval ezért az elemzés egy része onnan került át-emelésre, innen a hasonlóság.

```
#include <iostream>
#include "mlp.hpp"
#include <png++/png.hpp>
```

Include-oljuk az iostream, az mlp és a png++/png könyvtárakat. Utóbbi kettőt azért, mert többrétegű perceptronról akarunk majd létrehozni (Multi Layer Perceptron), így muszáj ezt a könyvtárat include-olnunk a programunkba. Utolsó könyvtárunk ahogy a neve is sugallja, a PNG képállományokkal való munkát teszi lehetővé. Előzőleg telepíteni szükséges, ha nem megtalálható a gépünkön.

```
using namespace std;

int main(int argc, char **argv)
{
 png::image<png::rgb_pixel> png_image(argv[1]);
```

```
int size = png_image.get_width() * png_image.get_height();

Perceptron *p = new Perceptron(3, size, 256, size);
```

Kezdődik a main függvényünk. Első sorában megmondjuk, hogy az 1-es parancssori argumentum alapján kerül beolvasásra a képállomány. Ettől kezdve dolgozni tudunk A kép méretét a get\_width és a get\_height szorzatából kapjuk, ezt el is tároljuk egy változóban. A következő sorban létrehozásra példányosítunk egy perceptronról a new operátor segítségével, amely paramétereit balról jobbra haladva: a rétegek száma, 1. réteg neuronjai az inputrétegben, 2. réteg neuronjai az inputrétegben, az eredmény (jelen esetben egy, a már megszokott szám helyett egy képállomány)

```
double* image = new double[size];

for(int i=0; i<png_image.get_width(); ++i)
 for(int j=0; j<png_image.get_height(); ++j)
 image[i*png_image.get_width()+j] = png_image[i][j].red;
```

Létrehozásra kerül egy double típusú mutató. Jönnek a for ciklusok. Az egyik for ciklus végigmegy a kép szélességét alkotó pontokon, a másik pedig a magasságán. Miután végigmentünk a képpontokon, az image tárolni fogja a képállomány vörös színkomponensét. Tehát a beolvasásra került képállomány piros vörös komponensét a lefoglalt tárba másoljuk bele.

```
double* newPicture = (*p)(image); // eddig: double value = (*p)(
 image);

for (int i = 0; i<png_image.get_width(); ++i)
 for (int j = 0; j<png_image.get_height(); ++j)
 png_image[i][j].red = newPicture[i*png_image.get_width()+j];

png_image.write("kimeneti.png");

delete p;
delete [] image;
```

Itt válik érdekkessé a dolog, hiszen már nem "csak" egy értéket akarunk kiiratni, hanem egy képet generálni, ahhoz azonban az egy double csillag típusra van már szükségünk. Ezt követően két for ciklussal végigmegyünk az eredeti kép szélességén és magasságán és az új képünk megkapja a színadatokat. Ezt követően a write függvénnyel létrehozásra kerül az új képállományunk kimeneti néven, png formátumban.

A végén törlésre kerülnek az eddig, de tovább már nem használatos elemeket, hiszen a számukra eddig fenntartott memóriaterületet érdemes felszabadítanunk, így a lefoglalt memóriamennyiséget újra használható. Programunk ezzel véget is ért.

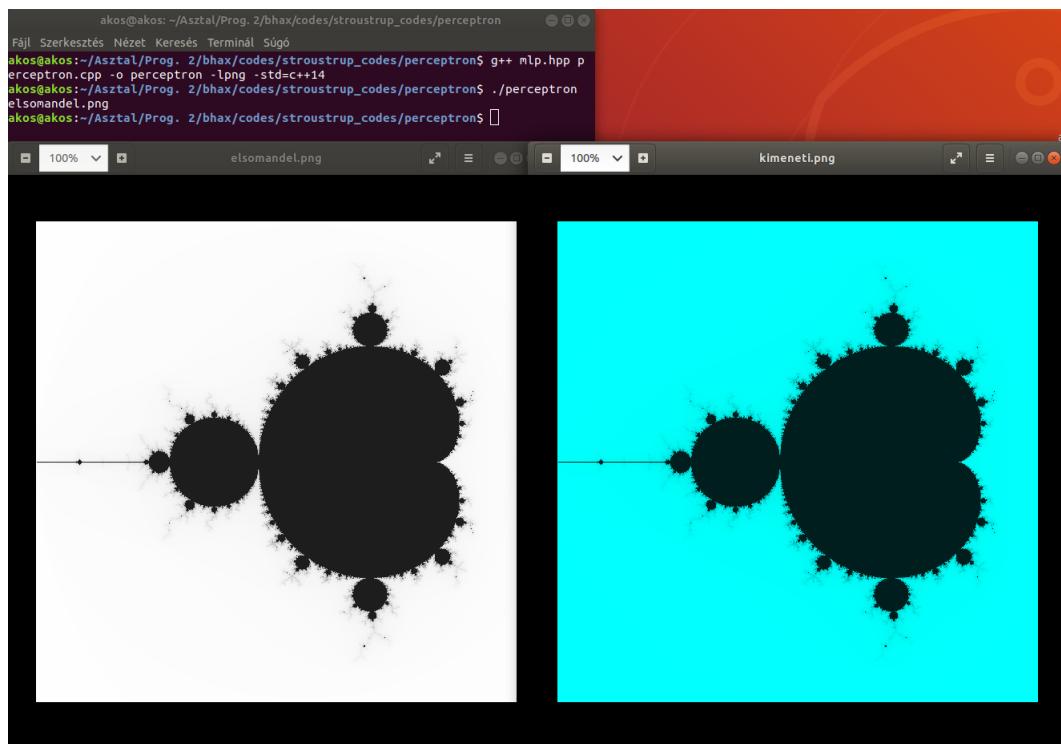
Természetesen ahhoz, hogy megfelelően működjön a programunk, többek között bele kellett nyúlni egy kicsit az mlp.hpp fájlba is.

```
double* operator()(double image [])
{
```

Többek között az () operátor működésébe is, ami mostantól nem egy double értéket, hanem egy double pointert ad vissza.

Fordítsuk és futtassuk a programunkat!

És az eredmény:



# 16. fejezet

## Helló, Gödel!

### 16.1. Gengszterek

Gengszterek rendezése lambdával a Robotautó Világ bajnokságban <https://youtu.be/DL6iQwPx1Yw> (8:05-től)

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/godel\\_codes/Gengszterek](https://gitlab.com/kincsa/bhax/tree/master/codes/godel_codes/Gengszterek)

<https://github.com/nbatfai/robocar-emulator>

Tutor: [GitLab](#), tutoriált: [GitLab](#)

A Robocar World Championship (OOCWC = rObOCar World Championship) egy olyan projekt/platform mely lehetőséget ad/adott a robotautó-kutatásra. Az egész projekt középpontjában a Robocar City Emulator áll.

A lambda-kifejezések a C++ 11-es verziójában jelentek meg. Ezek a kifejezések lehetővé teszik számunkra az úgynevezett in-line function-ök írását, vagyis egy- vagy kevés soros függvényekét. Fontos, hogy ezek olyan függvények, melyekre tipikusan egyszer van szükségünk, többször nem akarjuk felhasználni őket. Lényegében "egyszer használatosak", ezért még nevet SEM adunk nekik.

Egy egyszerű példa lambda-kifejezésre:

```
[] (int x, int y) -> { return x + y; }
```

A szöglletes zárójel jelzi számunkra, hogy lambda-kifejezés kezdődik. Ezután a zárójelekben a paraméterek találhatók, majd a nyilacsát követően a kapcsos zárójelek között a függvény maga található az elvégzendő művelettel. Itt dől el az is, mi lesz a visszatérési érték típusa.

Forrás: <https://en.cppreference.com/w/cpp/language/lambda>

Lássunk a mi példánkat!

```
std::sort (gangsters.begin(), gangsters.end(), [this, cop] (Gangster x, ←
 Gangster y)
{
 return dst (cop, x.to) < dst (cop, y.to);
});
```

```
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare ←
comp);
```

A `gangsters` vektor kerül rendezésre, ezt a `sort` függvény első és második paramétere teszi számunkra világossá (`gangsters.begin()` és `gangsters.end()`). A harmadik paraméter nem lesz más, mint maga a lambda-kifejezésünk. Ezen lambda-kifejezés paramétere két Gangster lesz. A kifejezés egy boolean értéket, pontosabban `true`-t ad vissza, ha az `x` gengszter és a rendőr távolsága kisebb, mint az `y` gengszter és a rendőr távolsága.

Tehát ha értelmezzük az előző pár sort, elmondhatjuk: ez esetben a vektorunk a gengszterek rendőrököz való távolsága alapján lesz rendezve.

A feladat megoldása során egyébent az utolsó sorban található `sort` függvényt használtuk, melynek harmadik paramétere lehetővé teszi, hogy kézzel adjuk meg az összehasonlítási szemponto(ka)t. Ez esetünkben a lambda-kifejezés.

## 16.2. STL map érték szerinti rendezése

Például: <https://github.com/nbatfai/future/blob/master/cs/F9F2/fenykard.cpp#L180>

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/godel\\_codes/STL](https://gitlab.com/kincsa/bhax/tree/master/codes/godel_codes/STL)

Tanulságok, tapasztalatok, magyarázat...

Az STL (Standard Template Library) olyan C++ sablon- vagy mintaosztályok összessége, amelyek implementálnak -ezzel lehetőséget adva használatukra- számos, széles körben előszeretettel használt algoritmus és adatszerkezet használatát. Hárrom fő dolgot tartalmaz: tárolókat, algoritmusokat és iterátorokat.

Esetünkben a tárolók lesznek érdekesek, sőt, abból is egy különleges fajta, a **map**. Mit érdemes tudni róla? Asszociatív tárolók, vagyis a kulcsuk által azonosíthatók az elemek. Ezzel le is lőttem a poént, kulcs-érték párokat tartalmaz. A benne található elemek minden esetben rendezettek, azok kulcsuk alapján.

Bevezető forrása: [link](#), [link](#)

Ez elő is vetíti számunkra a feladatot... Hiszen most nem kulcsuk, hanem értékük szerint kellene rendeznünk a map-et. Nézzük a megoldást!

```
#include <map>
#include <iostream>
#include <algorithm>
#include <vector>

std::vector<std::pair<std::string, int>> sort_map (std::map <std::string, int> &rank)
{
 std::vector<std::pair<std::string, int>> ordered;
 for (auto & i : rank) {
```

```
 if (i.second) {
 std::pair<std::string, int> p {i.first, i.second};
 ordered.push_back (p);
 }
 }

 std::sort (
 std::begin (ordered), std::end (ordered),
 [=] (auto && p1, auto && p2) {
 return p1.second > p2.second;
 }
);

return ordered;
}
```

A megoldás a fénykard programból lett átemelve. A `sort_map` függvényünk fogja megvalósítani az érték szerinti rendezést. A függvény visszatérési értéke vektorpárok lesznek. Mi történik a függvénytörlésben belül? Először is létrehozzuk a vektort, ami a visszatérési értékünk lesz `ordered` névvel. Majd egy for végigmegyünk a `rank_map`-en. Azt nézi, hogy vannak-e a vektorban érték párosok. Ha vannak, akkor egy pairbe kerülnek bele. Ezeket a pairesket pedig az `ordered` vektorba nyomjuk.

Az értékpárokkal feltöltött vektor rendezéshez segítségül a már ismert az `std::sort` függvényt hívjuk meg ami harmadik paramétereként egy, szöglletes zárójellel bevezetett lambda kifejezést alkalmazva rendez úgy, hogy végigmegy a vektorunkon és igazzal tér vissza abban az esetben, ha az első visgált paraméter nagyobb, mint a második. A függvényünk visszatérési értéke a `ordered`, már rendezett vektor lesz.

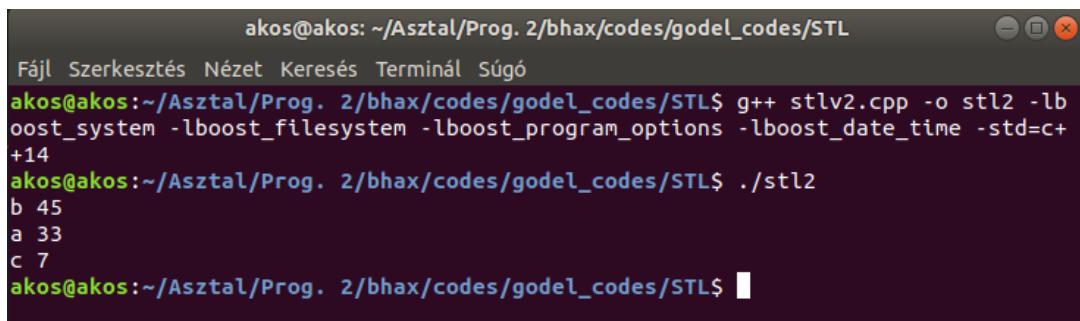
```
int main()
{
 std::map<std::string, int> map;
 map["a"] = 33;
 map["b"] = 45;
 map["c"] = 7;

 std::vector<std::pair<std::string, int>> sorted = sort_map (map) ↪
;

 for (auto & i : sorted)
 {
 std::cout << i.first << " " << i.second << std::endl;
 }
}
```

A `main`ben létrehozunk egy `map`-et, azt feltöljük elemekkel, majd az előbb tárgyalta `sort_map` függvénytel rendezzük azt. Ezt, a rendezés utáni állapotot egy új, `sorted` vektorban tároljuk el. Majd végül egy for ciklussal végigmegyünk új vektorunkon és kiiratjuk a rendezés utáni állapotot.

Fordítás és futtatás után a kis mintaprogramunk a következő eredménnyel tér vissza:



```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/godel_codes/STL
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/godel_codes/STL$ g++ stlv2.cpp -o stl2 -lboost_system -lboost_filesystem -lboost_program_options -lboost_date_time -std=c++14
akos@akos:~/Asztal/Prog. 2/bhax/codes/godel_codes/STL$./stl2
b 45
a 33
c 7
akos@akos:~/Asztal/Prog. 2/bhax/codes/godel_codes/STL$
```

## 16.3. Alternatív Tabella rendezése

Mutassuk be a [https://progpater.blog.hu/2011/03/11/alternativ\\_tabella](https://progpater.blog.hu/2011/03/11/alternativ_tabella) a programban a java.lang Interface Comparable<T> szerepét!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/godel\\_codes/AlternativTabella](https://gitlab.com/kincsa/bhax/tree/master/codes/godel_codes/AlternativTabella)

Tanulságok, tapasztalatok, magyarázat...

Tutor: [link](#), [link](#)

Tutoriált: <https://github.com/Hurip>

Talán érdemes lenne azzal kezdeni, mi is az az alternatív tabella? Az alternatív tabella nem más, mint az egyes futballbajnokságokhoz készített olyan sorrend, ami nem a megszokott módon (amikor 3 a győzelm, 1 a döntetlen, 0 a vereség) számolja a csapatok sorrendjét, hanem megpróbálja figyelembe venni azt, hogy egy adott csapat éppen melyik másik csapattal szemben érte el az éppen adott eredményt. Hiszen ha belegondolunk, a valóságban is "értékesebbnek" tartjuk a csapatunk által szerzett azt a 3 pontot, amelyet valamelyik ellen sikerült szerzett, mintsem azt, amelyet az éppen aktuális kiesőjelöltek ellen sikerült begyűjteni.

Az alternatív tabellák elkészítésének egyik módja, hogy a Google PageRank algoritmusát használjuk fel ehhez. Tehát ez esetben az alternatív sorrend úgy alakul ki, hogy az a csapat kerül rajta előrébb, amely előkelőbb helyen lévő csapatoktól szerez pontot.

A bevezető forrása: [https://hu.wikipedia.org/wiki/Alternativ\\_tabella](https://hu.wikipedia.org/wiki/Alternativ_tabella)

| Hagyományos          | pont | Alternatív           | rang   |
|----------------------|------|----------------------|--------|
| Videoton             | 40   | Videoton             | 0,0841 |
| Ferencváros          | 34   | Debreceni VSC        | 0,0828 |
| Paks                 | 31   | Paksi FC             | 0,0725 |
| Debreceni VSCC       | 31   | BFC Siófok           | 0,0698 |
| Zalaegerszegi TE     | 30   | Budapest Honvéd      | 0,0698 |
| Kaposvári Rákóczi    | 29   | Ferencváros          | 0,0689 |
| Lombard Pápa         | 27   | Győri ETO            | 0,0650 |
| Kecskeméti TE        | 24   | Újpest               | 0,0636 |
| Újpest               | 23   | Zalaegerszegi TE     | 0,0636 |
| Győri ETO            | 23   | MTK Budapest         | 0,0596 |
| Budapest Honvéd      | 22   | Kaposvári Rákóczi    | 0,0570 |
| MTK Budapest         | 22   | Lombard Pápa         | 0,0561 |
| Vasas                | 21   | Szombathelyi Haladás | 0,0559 |
| Szombathelyi Haladás | 20   | Vasas                | 0,0543 |
| BFC Siófok           | 18   | Kecskeméti TE        | 0,0439 |
| Szolnoki MÁV         | 9    | Szolnoki MÁV FC      | 0,0330 |

Az eredeti- és az alternatív tabella összehasonlítása. A kép forrása: [link](#)

Most nézzük meg a feladat által kért interface-t!

```
class Csapat implements Comparable<Csapat>
{
 protected String nev;
 protected double ertek;

 public Csapat(String nev, double ertek) {
 this.nev = nev;
 this.ertek = ertek;
 }

 public int compareTo(Csapat csapat) {
 if (this.ertek < csapat.ertek) {
 return -1;
 } else if (this.ertek > csapat.ertek) {
 return 1;
 } else {
 return 0;
 }
 }
}
```

Íme a java.lang package Comparable interface melyet a Csapat osztály implementál. Két objektum kerül összehasonlításra a compareTo függvénytel, az aktuális, a hívottal. Ha a hívó objektum értéke kisebb, mint a paraméterként átadott objektumé, akkor -1-t ad vissza, fordított esetben 1-et, egyenlőség esetén 0-t.

Miért is jó nekünk ez? A programunk következő sorai miatt:

```
java.util.List<Csapat> rendezettCsapatok = java.util.Arrays.asList(←
 csapatok);
java.util.Collections.sort(rendezettCsapatok);
```

Egy kis magyarázat ehhez a kódcsipethez.. Létrejön a Csapat típusú, rendezettCsapatok névre hallgató listánk, amelyet a következő sorban a sort metódussal rendezünk. Kis utánajárást követően megtaláltam, miért cselekedtünk az előzőekben úgy, ahogy. A sort metódus nem mindenfajta listán működik, hanem csak olyanokon melynek tagjai implementálják a tárgyalt Comparable interface-t.

Ennek "bizonyítéka", mely megtalálható a jdk-ban:

```
"Lists (and arrays) of objects that implement this interface can be sorted
automatically by {@link Collections#sort(List) Collections.sort} (and
{@link Arrays#sort(Object[]) Arrays.sort}). Objects that implement this
interface can be used as keys in a {@link plain SortedMap sorted map} or as
elements in a {@link plain SortedSet sorted set}, without the need to
specify a {@link plain Comparator comparator}."
```

Úgy gondolom, így már mindenki érthetőbb a megoldásunk.

Azt érdemes megemlíteni, hogy a szükségünk lesz a Wiki2Matrix.java állományra melyet ha lefuttatunk, egy linkmátrixszal leszünk gazdagabbak. A linkmátrix tartalmát az AlternativTabella.java kódunkba szükséges illesztenünk, az L kétdimenziós tömbbe. Csak ezután fordíthatjuk és futtathatjuk utóbb említett fájlunkat.

Fordítás és futtatás után:

```
akos@akos: ~/Asztal/Prog. 2/bhax/codes/godel_codes/AlternativTabella
Fájl Szerkesztés Nézet Keresés Terminál Súgó
akos@akos:~/Asztal/Prog. 2/bhax/codes/godel_codes/AlternativTabella$ javac AlternativTabella.java
akos@akos:~/Asztal/Prog. 2/bhax/codes/godel_codes/AlternativTabella$ java AlternativTabella
iteracio...
norma = 0.05918759643574294
osszeg = 1.0
iteracio...
norma = 0.014871507967965858
osszeg = 0.999999999999999
iteracio...
norma = 0.006686056768932613
osszeg = 1.0
iteracio...
norma = 0.007776749198562133
osszeg = 1.0
iteracio...
norma = 0.007661483555477314

Csapatok rendezve:
|-
| Videoton
| 40
| Videoton
0.0841
Ferencvaros
34
Debreceni VSC
0.0828
-
Paks
31
Paksi FC
0.0725
-
Debreceni VSC
31
BFC Siófok
```

```
| Szolnoki MAV
| 9
| Szolnoki MAV FC
| 0.0329
|
akos@akos:~/Asztal/Prog. 2/bhax/codes/godel_codes/AlternativTabella$
```

## 16.4. GIMP Scheme hack

Ha az előző félévben nem dolgoztad fel a témát (például a mandalás vagy a króm szöveges dobozosat) akkor itt az alkalom!

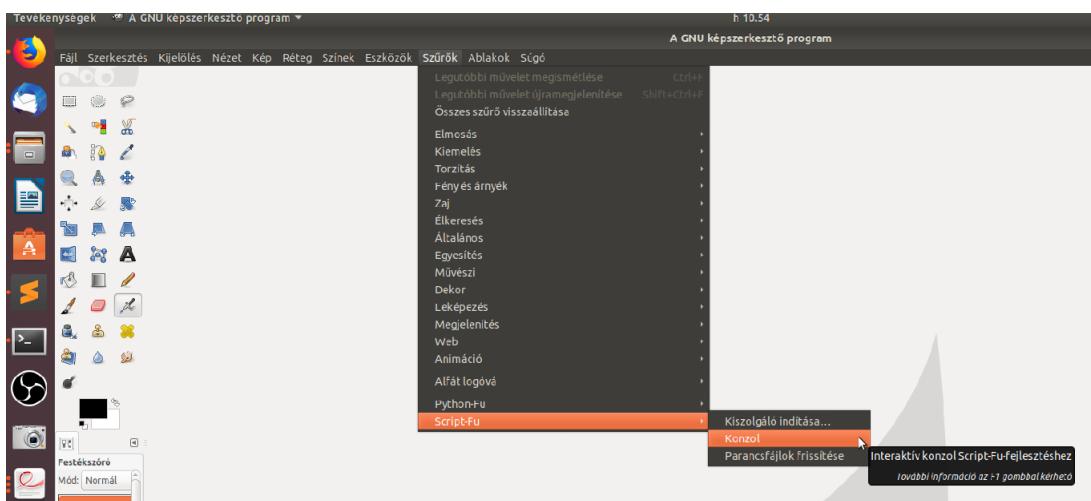
Megoldás videó: [https://youtu.be/OKdAkI\\_c7Sc](https://youtu.be/OKdAkI_c7Sc)

[https://gitlab.com/kincsa/bhax/tree/master/codes/godel\\_codes/Chrome](https://gitlab.com/kincsa/bhax/tree/master/codes/godel_codes/Chrome)

Tanulságok, tapasztalatok, magyarázat...

A most következő feladat során a Lisp programnyelvek egyik képviselőjével fogunk dolgozni, a Scheme nyelvvel. A Scheme programnyelv is a Lisp programnyelvek családjába tartozik. Az 1970-es évek közepén jelent meg de egyszerűsége miatt mind a mai találkozhatunk Scheme-kódokkal.

A most következő feladat során a Scheme egyik dialektusát, a Script-Fu-t fogjuk használni arra, hogy az előző feladat során már megismert GIMP képszerkesztő programhoz egy olyan scriptet írunk, ami lehetővé teszi a bemenetként megadott szöveg "króm effektekézését". Ezen script megírásához használhatjuk a már előző feladat során látott Script-Fu-konzolt is.



Lássuk, hogy néz ez ki (.scm kiterjesztésű) kód formájában!

```
(define (color-curve)
 (let* (
 (tomb (cons-array 8 'byte))
)
 (aset tomb 0 0)
 (aset tomb 1 0)
 (aset tomb 2 50)
 (aset tomb 3 190)
 (aset tomb 4 110)
 (aset tomb 5 20)
 (aset tomb 6 200)
```

```
(aset tomb 7 190)
tomb)
)
```

Definiáljuk a `color-curve` függvényünket amivel egy tömböt töltünk fel 8 különböző értékkel. Ez a függvény még visszaköszön majd a feladat során a 9. lépésnél.

```
(define (elem x lista)

(if (= x 1) (car lista) (elem (- x 1) (cdr lista)))

)

(define (text-wh text font fontsize)
(let*
(
 (text-width 1)
 (text-height 1)
)

(set! text-width (car (gimp-text-get-extents-fontname text fontsize ←
 PIXELS font)))
(set! text-height (elem 2 (gimp-text-get-extents-fontname text ←
 fontsize PIXELS font)))

(list text-width text-height)
)
)
```

Ebben a részben két függvény is definiálásra kerül. Az első, `elem` függvény egy paraméterként megadott listából szintén paraméterként megadott sorszámú elem értékét adja vissza. Ez a függvény felhasználásra is kerül a következő `text-wh` függvény során is, amely függvénynek a visszatérési értéke egy lista melyben a font szélessége és magassága található meg.

```
(define (script-fu-bhax-chrome text font fontsize width height color ←
gradient)
(let*
(
 (image (car (gimp-image-new width height 0)))
 (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" ←
 100 LAYER-MODE-NORMAL-LEGACY)))
 (textfs)
 (text-width (car (text-wh text font fontsize)))
 (text-height (elem 2 (text-wh text font fontsize)))
 (layer2)
)
```

Az első sorban definiálásra kerül a scriptünk, ahol az látható, hogy a `script` nevét szóközzel elválasztva követik a paraméterek.

Ezt követi a `let*` függvény ami két részből áll, ezekből az elsőt tárgyaljuk ki ebben a bekezdésben. Ez az ún. "varlist" rész, itt azok a változók kerülnek deklarálásra, amelyeket a későbbiekben sokat fogunk

használni. A több helyen is használt `car` függvény a GIMP eljárások és függvények által visszaadott listájának első elemét adja vissza. Természetesen bármilyen megkapott lista első elemét is visszaadja, nem csupán a GIMP-eseket...

```
;step 1
(gimp-image-insert-layer image layer 0 0)
(gimp-context-set-foreground '(0 0 0))
(gimp-drawable-fill layer FILL-FOREGROUND)
(gimp-context-set-foreground '(255 255 255))
```

Első lépésként az effektelni kívánt szöveget fehér betűszínnel fekete háttérre kell írnunk.

A megoldáshoz a GIMP eljárásböngészőjét hívjuk segítségül, ahol a keresőbe bepötyögve a megfelelő függvény/eljárás nevét, minden információt megkapunk róla.

A `gimp-image-insert-layer` egy új réteget ad hozzá képünkhez, első paramétere a kép maga, második a réteg, harmadik a szülő-réteg, ami jelen esetünkben nincs (ezért 0), az utolsó paraméter a pozíciója a rétegnek, ami jelen esetben 0, hiszen a legfelső rétegnek szeretnénk.

A `gimp-context-set-foreground` eljárással az előtérszínt RGB(0 0 0)-ra, vagyis feketére állítjuk. A `gimp-drawable-fill` layer eljárással rétegünket kitöljtük az előtérszínnel. Újra a `...set-foreground` eljárást használjuk, az előtérszínt most fehérre állítjuk.

```
(set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ←
))
(gimp-image-insert-layer image textfs 0 0)
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (- (/ ←
 height 2) (/ text-height 2)))

(set! layer (car (gimp-image-merge-down image textfs CLIP-TO-BOTTOM- ←
 LAYER)))
```

A `set!`-tel a `gimp-text-layer-new` függvény által a megfelelő paraméterekkel elkészített, majd visszaadott réteget beállítjuk `textfs` névre. Ezt a réteget a következő sorban az `..image-insert-layer` eljárással beszúrjuk.

A `gimp-layer-set-offsets` eljárással a rétegünket középre pozicionáljuk. Az utolsó sorban a `gimp-image-merge-down` függvénnnyel lefelé haladva összefűsüljük a rétegeinket és ezt a `layer` változóban tároljuk.

Az első lépés után ez a képünk:

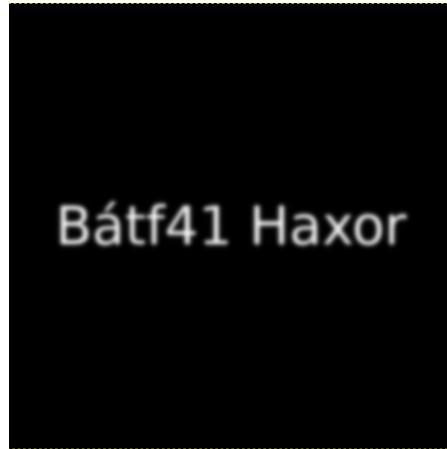


**A második lépés:** Gauss-elmosás alkalmazása

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 15 TRUE TRUE)
```

Az eljárás paraméterei a következők: futási mód (esetünkben ez RUN-INTERACTIVE), a kép maga, "rajzolható input"(nekünk a réteg), az elmosódás pixelben mért sugara, elmosódás vízszintesen, elmosódás függőlegesen.

A lépés után ez a képünk:

**A harmadik lépés:** játék a színszintekkel

```
(gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE)
```

Az első paraméter, hogy min végezze el a műveletet a függvény, nekünk természetesen a rétegünkön kell majd, hogy elvégezve legyen. minden további paraméter az egyes színértékekkel kapcsolatos, az Eljárás-böngészőből kikeresgélhetőek.

A lépéssel a képünk:

**A negyedik lépés:** Gauss-elmosás, újra

```
(plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE)
```

A paraméterek megegyeznek a nemrég nézett Gauss-elmosást létrehozó eljárásval, egyedül az elmosás sugara kisebb most, mint az előbb volt, így ez most egy enyhébb elmosást eredményez, szinte nem is látható a különbség az előző képhez képest:



**Az ötödik lépés:** A fekete rész szín szerinti kijelölése majd a kijelölés invertálása

```
(gimp-image-select-color image CHANNEL-OP-REPLACE layer '(0 0 0))
(gimp-selection-invert image)
```

A `gimp-image-select-color` eljárással történik a kijelölés. Paraméterei: melyik képen kerül majd alkalmazásra, a kijelöléshez használatos művelet, "rajzolható input" és a kijelölendő szín.

A `gimp-selection-invert` pedig az egyetlen paraméterként kapott képen megfordítja a kijelölést.

**A hatodik lépés:** "Lebegő átlátszó kijelölés létrehozása"

```
(set! layer2 (car (gimp-layer-new image width height RGB-IMAGE "2" 100
 LAYER-MODE-NORMAL-LEGACY)))
(gimp-image-insert-layer image layer2 0 0)
```

Az első sorban a `set!`-tel `layer2`-t egy új rétegként állítjuk be, mely réteget a `gimp-layer-new` függvény állított elő a képen látható paraméterekkel. Ezek közül a 6. a legérdekesebb ami nem más, mint az opacitás (áttetszőség) amit 100-ra állítottunk be. A következő réteget beillesztjük a már előzőek során is látott `gimp-image-insert-layer` eljárással.



**A hetedik lépés:** "Az áttetszőség kitöltése átmenettel

```
(gimp-context-set-gradient gradient)
(gimp-edit-blend layer2 BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY GRADIENT-
LINEAR 100 0 REPEAT-NONE
 FALSE TRUE 5 .1 TRUE width (/ height 3) width (- height (/ height
 ←
 3)))
```

Az első eljárás beállítja az elsődleges átmenetet, a gimp-edit-blend eljárás pedig egy kezdő- és végpont között egy "keverést" hoz létre a paraméterek segítségével, melyekről bővebben az Eljárásböngészőben lehet olvasni.

Így állunk jelenleg:



**A nyolcadik lépés:** Buckaleképezés alkalmazása

```
(plug-in-bump-map RUN-NONINTERACTIVE image layer2 layer 120 25 7 5 5 0 ←
 0 TRUE FALSE 2)
```

A plug-in-bump-map-pel buckaleképezést hozunk létre a layer2-n, bemenetként felhasználva másik rétegünket.



**A kilencedik lépés:** Színgörbékkel való játek

```
(gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))
```

A fémesebb hatás érdekében gimp-curves-spline-nal ügyeskedünk még és voilá, kész is vagyunk:



```
(script-fu-register "script-fu-bhax-chrome"
 "Chrome3"
 "Creates a chrome effect on a given text."
 "Norbert Bátfai"
 "Copyright 2019, Norbert Bátfai"
 "January 19, 2019"
 ""
 SF-STRING "Text" "Bátf41 Haxor"
 SF-FONT "Font" "Sans"
 SF-ADJUSTMENT "Font size" '(100 1 1000 1 10 0 1)
 SF-VALUE "Width" "1000"
 SF-VALUE "Height" "1000"
 SF-COLOR "Color" '(255 0 0)
 SF-GRADIENT "Gradient" "Crown molding"
)
(script-fu-menu-register "script-fu-bhax-chrome"
 "<Image>/File/Create/BHAX"
)
```

A scriptet ha nem a script-konzolon keresztül szeretnénk beadni, hanem inkludálni szeretnénk, hogy a GIMP-en belül könnyebben használható legyen, függvényt kell használnunk hozzá. Ezt teszi a `script-fu-register` függvény, amely definiálása során alapértelmezett értékeket is megadunk.

A kódunk zárásaként megírjuk az ún. "Menübe regisztráló függvényt". Ahogy az elnevezés is sugallja, ezen függvény hatására tudjuk majd kiválasztani grafikusan is, a menüből a scriptünket.

# 17. fejezet

## Helló, !

### 17.1. OOCWC Boost ASIO hálózatkezelése

Mutassunk rá a scanf szerepére és használatára! <https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/carlexer.ll>

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/hello\\_codes/OOCWC](https://gitlab.com/kincsa/bhax/tree/master/codes/hello_codes/OOCWC)

<https://github.com/nbatfai/robocar-emulator> h

Tanulságok, tapasztalatok, magyarázat...

Tutor: [link](#)

A Robocar World Championship (OOCWC = rObOCar World Championship) egy olyan projekt/platform mely lehetőséget ad/adott a robotautó-kutatásra. Az egész projekt középpontjában a Robocar City Emulator áll. A következőkben a carlexer.ll-ben található scanf függvény jelentőségével és használatával fogunk foglalkozni.

```
while (std::sscanf (data+nn, "<OK %d %u %u %u>%n", &idd, &f, &t, &s, &n) == 4)
{
 nn += n;
 gangsters.push_back (Gangster {idd, f, t, s});
}
```

Forrás: <http://www.cplusplus.com/reference/cstdio/sscanf/>

A sscanf a sima scanf függvényel ellentétben formázott stringből olvas be adatokat. Alapvetően két részből állnak: egy úgynevezett bufferból és egy formatból. A buffer egy pointer lesz egy karakterstringre, amiből kiolvasásra kerül majd az adat. A format határozza meg, az adatok hogyan kerüljenek olvasásra. Úgynevezett format-specifikátorokból áll, ezek rendre százalékkel kezdődnek.

Lássuk a mi esetünkben ez hogy néz ki! Azt azonban fontos megemlíteni, hogy forrásunkban a sscanf függvénynek több előfordulása van, nem csak az, amit a következőkban meg fogunk tekinteni, azonban lényegében minden egy kaptafára épülnek, ezért úgy gondolom, elég egyet bemutatni közülük.

Példánkban ilyennel találkozhatunk: `<%d %u %u %u>%n`. A d az integerekre illeszkedik, míg a u az unsigned integerre, vagyis az előjel nélküli integerekre. Az utolsó, n pedig arra fog szolgálni, hogy számon tartsa a már beolvasott karakterek számát. Az előbb mintáknak megfelelő adatok rendre a `<&` jellet bevezetett változókban kerülnek tárolásra.

Az beolvasandó adatok addig kerülnek feldolgozásra, míg nincs meg mind a 4 várt argumentum. (ezt láthatjuk a while ciklusfej végén- `== 4` ).

Abban az esetben ha minden a 4 várt argumentumot sikeresen be tudtuk olvasni, az az számunkra azt jelenti, a gengsztereket jellemző minden a 4 tulajdonság meg lett adva, így ez esetben egy új Gangster kerül létrehozásra a megfelelő argumentumokkal és a gangsters vektorba tároljuk az "elkészült" gengszterünket.. Az nn változóban pedig tárolásra kerül az összesen beolvasott karakterek száma.

Látható, hogy az nn változó a `sscanf` függvény buffer részében is megtalálható. Amint mondtam, az nem más, mint egy pointer. A data értékét azért növeljük minden egyes alkalommal nn értékével, hogy onnan olvassunk ki további adatokat, ahonnan még nem tettük azt, ezzel a még kiolvasatlan rész elejére ugrunk minden.

## 17.2. SamuCam

Mutassunk rá a webcam (pl. Androidos mobilod) kezelésére ebben a projektben: <https://github.com/nbatfai/SamuCam>

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/hello\\_codes/SamuCam](https://gitlab.com/kincsa/bhax/tree/master/codes/hello_codes/SamuCam)

Tanulságok, tapasztalatok, magyarázat...

Mivel a feladat kifejezetten a webcam kezelését kéri tőlünk, egyértelmű, hogy a `SamuCam.cpp` állományunk lesz az, amivel foglalkoznunk kell. Kezdjük is el, elemezzük ki a kódot!

```
#include "SamuCam.h"

SamuCam::SamuCam (std::string videoStream, int width = 176, int ←
 height = 144)
 : videoStream (videoStream), width (width), height (height)
{
 openVideoStream();
}

SamuCam::~SamuCam ()
{
}

void SamuCam::openVideoStream()
{
 videoCapture.open (0);

 videoCapture.set (CV_CAP_PROP_FRAME_WIDTH, width);
 videoCapture.set (CV_CAP_PROP_FRAME_HEIGHT, height);
 videoCapture.set (CV_CAP_PROP_FPS, 10);
```

```
}
```

Mindenekelőtt inkludáljuk a header-fájlunkat, enélkül nehézen tudna kódunk normálisan működni. Megjelenik a konstruktorunk mely 3 paraméterrel rendelkezik. Ezek a következők: videoStream, width és height. A destruktorttal is itt találkozhatunk. Az openVideoStream függvényen belül már kezdődnek az érdekkességek. Fontos beszélni a videoCapture.open függvényről és paraméteréről is. Kérdés, hogy miért 0-t találunk ott.. Az eszköz / device indexet találhatjuk meg ott. Ezek az indexek 0-tól kezdődnek, a 0 az alapértelmezett kamera-eszközünk indexe. De akár IP cím is állhatna ott paraméterként.

Ugyanitt történik az előbb említett objektum finomhangolása, tehát a kamera képének szélessége és magassága valamint az FPS szám megadása.

```
void SamuCam::run()
{
 cv::CascadeClassifier faceClassifier;

 std::string faceXML = "lbpcascade_frontalface.xml"; // https://github.com/Itseez/opencv/tree/master/data/lbpcascades

 if (!faceClassifier.load (faceXML))
 {
 qDebug() << "error: cannot found" << faceXML.c_str();
 return;
 }

 cv::Mat frame;
```

Elérkeztünk a run függvényhez. Szükségünk van egy CascadeClassifier-re, amit alapesetben egy adott tárgy "detektálására" szoktak használni az adott videostreamben, OpenCV-s programokban. Esetünkben ez a faceClassifier lesz, aminek a feladata az arc felismerése lesz. Ahhoz viszont, hogy ez működjön, a megjegyzésben és a github repóban is található linkre kattintva le kell töltenünk egy xml fájlt, aminek tartalmát a faceXML stringben tárolunk majd el. Ez tartalmazza majd ténylegesen az arc felismeréséhez használatos információkat. A load függvénytel működésre bírjuk az xml-t, természetesen a fájl hiányában hibaüzenettel térünk vissza.

```
while (videoCapture.isOpened())
{
 QThread::msleep (50);
 while (videoCapture.read (frame))
 {

 if (!frame.empty())
 {

 cv::resize (frame, frame, cv::Size (176, 144), 0, 0, cv::INTER_CUBIC);

 std::vector<cv::Rect> faces;
```

```
cv::Mat grayFrame;

cv::cvtColor (frame, grayFrame, cv::COLOR_BGR2GRAY);
cv::equalizeHist (grayFrame, grayFrame);

faceClassifier.detectMultiScale (grayFrame, faces, 1.1, ←
 3, 0, cv::Size (60, 60));

if (faces.size() > 0)
{
 cv::Mat onlyFace = frame (faces[0]).clone();

 QImage* face = new QImage (onlyFace.data,
 onlyFace.cols,
 onlyFace.rows,
 onlyFace.step,
 QImage::Format_RGB888);

 cv::Point x (faces[0].x-1, faces[0].y-1);
 cv::Point y (faces[0].x + faces[0].width+2, faces ←
 [0].y + faces[0].height+2);
 cv::rectangle (frame, x, y, cv::Scalar (240, 230, ←
 200));

 emit faceChanged (face);
}

QImage* webcam = new QImage (frame.data,
 frame.cols,
 frame.rows,
 frame.step,
 QImage::Format_RGB888);

emit webcamChanged (webcam);

}

QThread::msleep (80);
}
```

Egy while függvényen belül történik a "csoda" ..50 ms-onként ellenőrzi, megvan-e még nyitva kameránk. Ha meg, abban az esetben, ha a vannak képkockáink, azok a frame-ról beolvasásra majd tárolásra is kerülnek.

A resize függvény segítségével átméretezésre kerül a kép. Létrehozunk egy faces vektort, majd a képet a cvtColor segítségével szürkévé alakítjuk. Ezen képpontok számára is létrehozunk egy tárolót grayFrame néven.

Ezt követően a `detectMultiScale` függvényt felhasználva arcokat keresünk, a megtalált arcok pedig egy `rectangle` (téglalap) kerülnek tárolásra. Az arcból `QImage` objektum készül, majd egy `webcam` névre hallgató `QImage` objektum is készül, mindenkorrel.

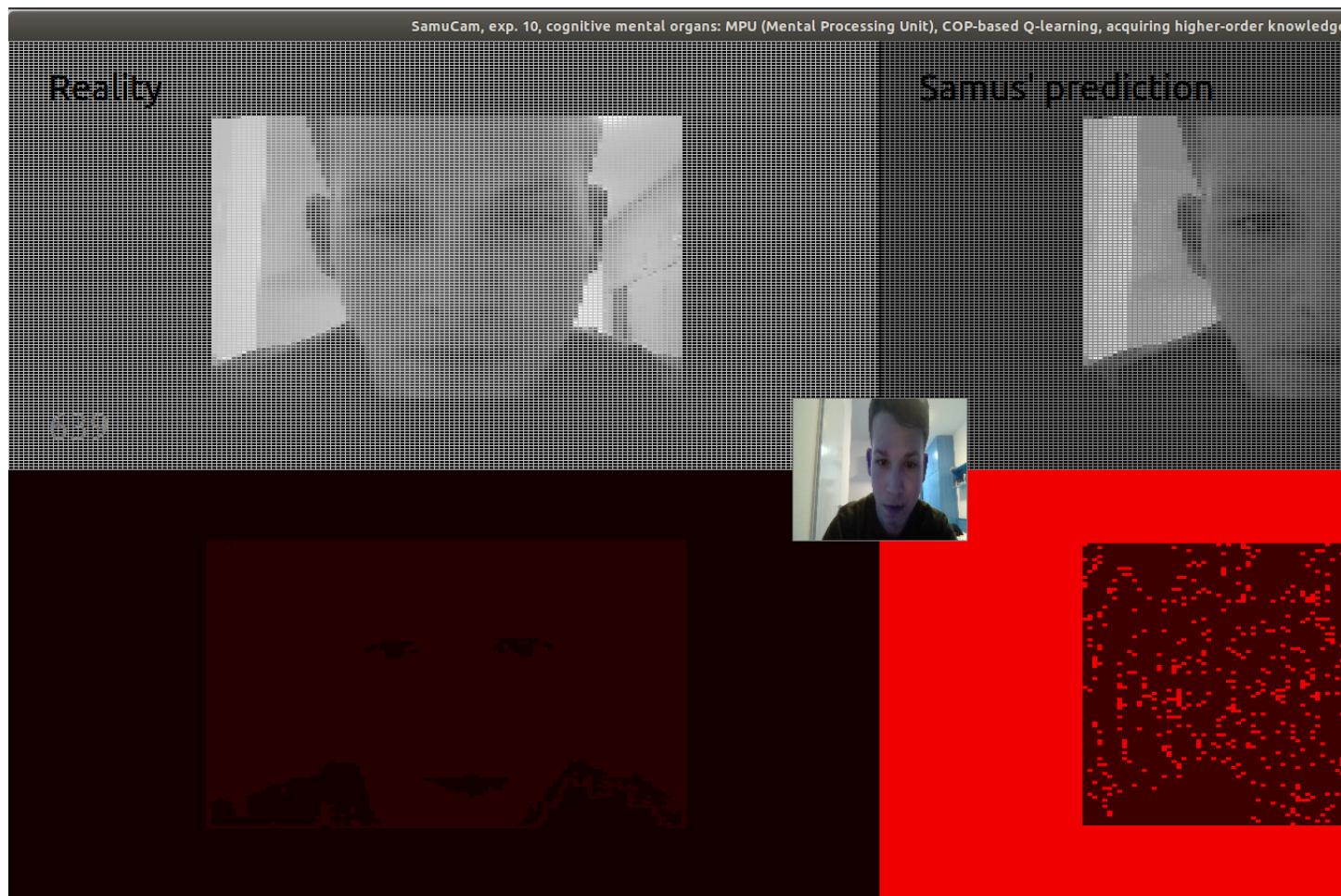
Mindkét esetben találkozhatunk az `emit` függvénygel, ami nem más, mint egy makró. A következő feladatban foglalkozunk majd részletesebben a Qt slot-signal mechanizmusával, egyelőre annyit elég tudni az `emit`ről, hogy slotok és signalok összeegyeztetésénél van szerepe.

Az előbb olvasható blokkban található tartalom pedig rendre 80 ms-enként ismétlődik majd.

```
if (! videoCapture.isOpened())
{
 openVideoStream();
}
```

Megvizsgáljuk azt is, megvan-e nyitva kameránk. Ellenkező esetben megteszí azt.

Fordítjuk és futtatjuk a kódot, és voilá! Látható, hogy sikeres volt az arcfelismerés!



### 17.3. BrainB

Mutassuk be a Qt slot-signal mechanizmust ebben a projektben: <https://github.com/nbatfai/esport-talent-search>

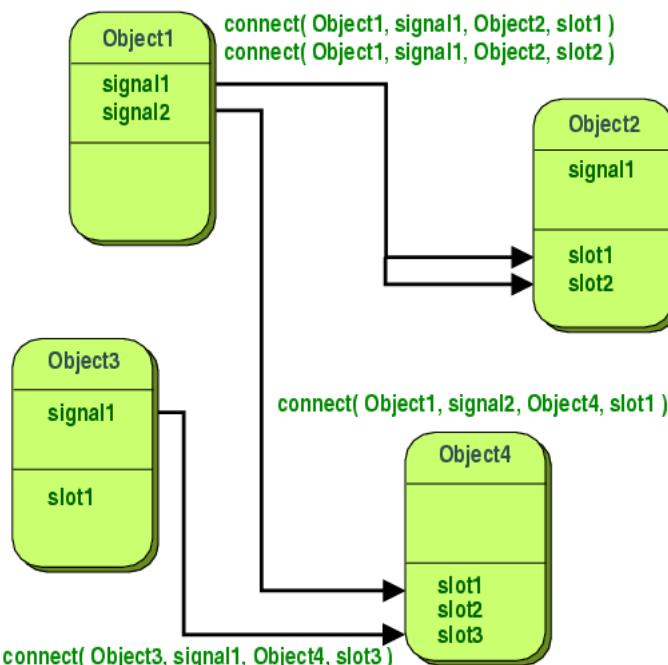
Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/hello\\_codes/BrainB](https://gitlab.com/kincsa/bhax/tree/master/codes/hello_codes/BrainB)

Tanulságok, tapasztalatok, magyarázat...

A BrainB projekttel már előző félévben, Magas szintű programozási nyelvek 1 tárgyunk során volt szerencsém találkozni. A program célja nem más, mint hogy segítse a jövő kiemelkedő esportolónak megtalálását. A program ebben úgy segít, hogy jövő kiemelkedő esportolónak fellelése. A program az agy kognitív képességét méri.

A játékos feladata a Samu Entropy nevű karakteren tartani az egérmutatót ami az idő teltével természetesen egyre nehezebb hiszen újabb és újabb Entropy karakterek jelennek meg a képernyőn. A program érzékeli, ha Samu Entropyt elvesztettük, ekkor csökkenti a többi Entropy számát. A kódok elemzése előző féléves Prog. 1-es tanulmányaink során már megtörtént ezért most arra külön nem térnék ki. Viszont akkor még nem tértünk ki a Qt slot-signal mechanizmusára, most pótoljuk be ezt és nézzük, mi is az!



A kép forrása: <https://doc.qt.io/qt-5/signalsandslots.html>

Mit látunk az ábrán? Signal-okat és slot-okat. A feladat megértéséhez tisztáznunk kell a fogalmakat. A signal-ok az objektumok által kerülnek "kisugározva", lényegében jelként kiküldve többek között akkor ha az adott objektum belső állapota valami oknál fogva megváltozott.

Ezekhez a signal-okhoz vannak kapcsolva a slot-ok, amik akkor kerülnek meghívásra ha a hozzájuk tartozó signal kiküldésre kerül. Ezek a slot-ok egyébként egyszerű C++ függvények, hívásuk viszont nem úgy történik, mint a hétköznapi függvényeknek. Az emit kulcsszóval bocsátják ki. Erre láthatunk példát akár a BrainBThread.cpp forrásunkban is:

```
emit endAndStats(endTime);
```

A slot-okat a signal-akkal minden esetben egy connect függvénytel kapcsolhatjuk egymáshoz. Erre programunkban két ízben láthatunk példát a BrainBWin.cpp fájlban.

```

connect(brainBThread, SIGNAL(heroesChanged(QImage, int, int)),
 this, SLOT(updateHeroes(QImage, int, int)));

connect(brainBThread, SIGNAL(endAndStats(int)),
 this, SLOT(endAndStats(int)));

```

Hogyan is épül fel ez a connect függvény, mik az egyes paraméterek? Formailag a következőképp néz ki: connect (első\_objektum, signal, második\_objektum, slot). Az első objektum küldi magát a signalt, míg a második feladata kezelni azt.

Jogosan merül fel a kérdés, megtörténhet-e az, hogy egy signal-hoz több slot is tartozzon? Természetesen ez is lehetséges, ekkor a slot-ok egymás után sorban kerülnek végrehajtásra. Sőt, akár fordítva is történhet a dolog, több signal is kapcsolható akár egyetlen slothoz. Ha nagyon el akarunk rugaszkodni a tipikus felhasználási módtól, akár két signal egymáshoz kapcsolása is megoldható.

Fontos megemlíteni, hogy a mechanizmus típus biztos, vagyis a signal és a slot szignatúrájának meg kell egyeznie.

Vessünk egy pillantást a mi példánakra ismét!

```
connect (brainBThread, SIGNAL (heroesChanged (QImage, int, int)),
 this, SLOT (updateHeroes (QImage, int, int)));
connect (brainBThread, SIGNAL (endAndStats (int)),
 this, SLOT (endAndStats (int)));
```

Láthatjuk, hogy a slot-ok és a signal-ok paramétereinek száma és típusa vagyis szignatúrája megegyezik.

Az első connect értelmezése: ha a brainBThread objektumban a heroesChanged signál emittálódik, akkor az updateHeroes slotnak kell meghívódnia.

A hívott függvény:

```
void BrainBWin::updateHeroes (const QImage &image, const int &x, ←
 const int &y)
{
 if (start && !brainBThread->get_paused()) {
 int dist = (this->mouse_x - x) * (this->mouse_x - x) + ←
 (this->mouse_y - y) * (this->mouse_y - y);
 if (dist > 121) {
 ++nofLost;
 nofFound = 0;
 if (nofLost > 12) {
 if (state == found && firstLost) {
 found2lost.push_back (brainBThread ←
 ->get_bps());
 }
 firstLost = true;
 state = lost;
 nofLost = 0;
 //qDebug () << "LOST";
 //double mean = brainBThread->meanLost ();
 //qDebug () << mean;
 }
 brainBThread->decComp();
 }
 }
}
```

```
 }
 } else {
 ++noFFound;
 nofLost = 0;
 if (noFFound > 12) {

 if (state == lost && firstLost) {
 lost2found.push_back (brainBThread ->get_bps());
 }

 state = found;
 noFFound = 0;
 //qDebug() << "FOUND";
 //double mean = brainBThread->meanFound();
 //qDebug() << mean;

 brainBThread->incComp();
 }

 }

}

pixmap = QPixmap::fromImage (image);
update();
}
```

Ez azt eredményezi, hogy a hőseink helye megváltozik a képernyőn.

A második connect értelmezése: ha a brainBThread objektumban az endAndStats signál emittálódik, akkor az ugyanilyen nevű endAndStats slotnak kell meghívódnia.

Meghívásra került:

```
void BrainBWin::endAndStats (const int &t)
{

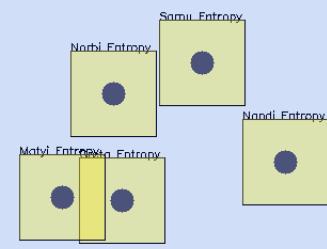
 qDebug() << "\n\n\n";
 qDebug() << "Thank you for using " + appName;
 qDebug() << "The result can be found in the directory " + statDir;
 qDebug() << "\n\n\n";

 save (t);
 close();
}
```

Az endAndStats függvény lényegében "búcsúüzenetet" dob a felhasználónak, elmenti eredményeinket egy fájlba és kilép.

Fordítás és futtatás után már használható is:

Press and hold the mouse button on the center of Samu Entropy  
0:4/10:0 6660 bps



# 18. fejezet

## Helló, Lauda!

### 18.1. Port scan

Mutassunk rá ebben a port szkennelő forrásban a kivételkezelés szerepére!

<https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html#id527287>

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/lauda\\_codes/Port%20scan](https://gitlab.com/kincsa/bhax/tree/master/codes/lauda_codes/Port%20scan)

Tanulságok, tapasztalatok, magyarázat...

Íme a forrásban található kód:

```
public class KapuSzkenner {

 public static void main(String[] args) {

 for(int i=0; i<1024; ++i)

 try {

 java.net.Socket socket = new java.net.Socket(args ←
[0], i);

 System.out.println(i + " figyeli");

 socket.close();

 } catch (Exception e) {

 System.out.println(i + " nem figyeli");

 }
 }
}
```

És hogy mégis mit csinál ez a kis program? Igen egyszerű a működése.

Egy `for` ciklus adja a programunk vázát. A programunk nem csinál mászt, minthogy az 1-estől az 1023-as porttal bezárólag próbál ezeken a portokon TCP kapcsolatot létesíteni, amelyért a `try` blokk következő sora felel:

```
java.net.Socket socket = new java.net.Socket(args[0], i);
```

Ha az aktuális portot egy folyamat figyeli, akkor a port kiiratásra kerül a következő sorban. Majd lezárjuk a socket-et.

Most jöjjön a kivételkezelés része. Kezdjük talán azzal, mi is a kivétel: a program végrehajtásakor keletkező esemény, ami megszakítja az utasítások végrehajtásának normális folyamatát. Tehát: minden amivel eddig foglalkoztunk a `try` blokkban volt megtalálható. Az a kód található meg ebben a blokkban, ami hibát dobhat. Most akkor térjünk ki a `catch` blokkra. A `catch` a zárójelében megadott paraméter típusának megfelelő kivételeket kezel. Ez esetünkben `Exception`, amely egy olyan kivételosztály ami minden más kivételosztályt is magába foglal.

Abban az esetben ha az adott portot nem figyeli egyik folyamat sem, kivétel kerül dobásra. Nézzük meg a JDK forrásban, milyen vagy milyenek lehetnek!

```
public Socket(String host, int port)
 throws UnknownHostException, IOException
{
 this(host != null ? new InetSocketAddress(host, port) :
 new InetSocketAddress(InetAddress.getByName(null), port),
 (SocketAddress) null, true);
}
```

Látható, hogy esetünkben a fenti kettő fordulhat elő. Ha ilyen kivétel dobódik, a végrehajtás a `catch` ágra ugrik, ahol egy tájékoztató kiiratást hajtunk végre.

Fordítjuk a programot. Majd a futtatást a képen látható módon, paraméterként egy IP-cím megadásával végezzük el.

```
akos@akos:~/Asztal/Prog. 2/bhax/codes/lauda_codes/Port scan$ clear
akos@akos:~/Asztal/Prog. 2/bhax/codes/lauda_codes/Port scan$ java KapuSzkenner 127.0.0.1
0 nem figyeli
1 nem figyeli
2 nem figyeli
3 nem figyeli
4 nem figyeli
5 nem figyeli
6 nem figyeli
7 nem figyeli
8 nem figyeli
9 nem figyeli
10 nem figyeli
11 nem figyeli

629 nem figyeli
630 nem figyeli
631 figyeli
632 nem figyeli
633 nem figyeli

1018 nem figyeli
1019 nem figyeli
1020 nem figyeli
1021 nem figyeli
1022 nem figyeli
1023 nem figyeli
akos@akos:~/Asztal/Prog. 2/bhax/codes/lauda_codes/Port scan$
```

## 18.2. Android Játék

Írunk egy egyszerű Androidos „játékot”! Építkezzünk például a 2. hét „Helló, Android!” feladatára!

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/lauda\\_codes/guessingGame](https://gitlab.com/kincsa/bhax/tree/master/codes/lauda_codes/guessingGame)

Tanulságok, tapasztalatok, magyarázat...

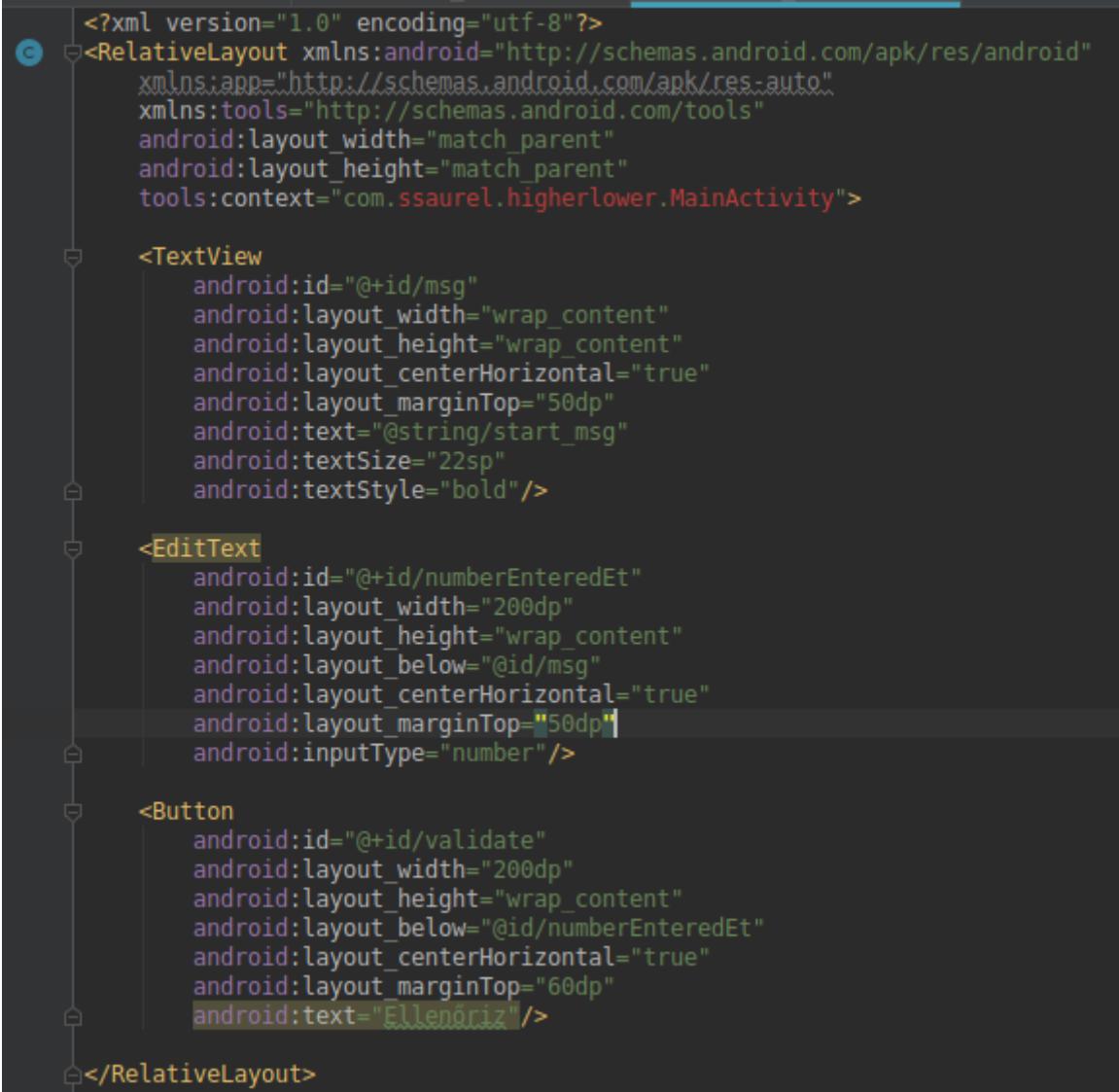
A játék a következő videó alapján készült.

A játék logikailag igen egyszerű. A gép egy random számot sorsol 1 és 100 között, a felhasználó feladata pedig az, hogy minél kevesebb találgatásból rájöjjön, mi is a gondolt szám. Ehhez a program természetesen segítséget ad, a felhasználóval minden egyes tipp után közli, hogy kisebb-, vagy épp nagyobb számot kéne keresnie-e vagy hogy esetleg megtalálta a sorsolt számot.

Nézzük, hogy is épül fel a játékunk!

```
<resources>
 <string name="app_name">guessingGame</string>
 <string name="start_msg">Találd ki a számot! (1 és 100 között)</string>
 <string name = "too_high">Kisebb számmal próbálkozz!</string>
 <string name = "too_low">Nagyobb számmal próbálkozz!</string>
</resources>
```

Mivel a játék során a felhasználó a játék által küldött információkra tud csak támaszkodni, előbb ezeket az üzeneteket fogalmazzuk meg és tároljuk el! A `strings.xml` fájlban létrehozásra kerültek a játék során a felhasználónak címzett üzenetek stringek formájában. Természetesen több .xml fájl is rendelkezésünkre áll szerkesztésre annak érdekében, hogy játékunk kicsit "pofásabb" legyen. Ilyenek a `colors`, `dimens` és a `styles.xml` is, azonban utóbbi hármat most békén hagytuk, a színek és a megjelenés - az üzeneteket leszámítva - alapértelmezetten maradt.



```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 xmlns:tools="http://schemas.android.com/tools"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 tools:context="com.ssaurel.higherlower.MainActivity">

 <TextView
 android:id="@+id/msg"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_centerHorizontal="true"
 android:layout_marginTop="50dp"
 android:text="@string/start_msg"
 android:textSize="22sp"
 android:textStyle="bold"/>

 <EditText
 android:id="@+id/numberEnteredEt"
 android:layout_width="200dp"
 android:layout_height="wrap_content"
 android:layout_below="@+id/msg"
 android:layout_centerHorizontal="true"
 android:layout_marginTop="50dp"
 android:inputType="number"/>

 <Button
 android:id="@+id/validate"
 android:layout_width="200dp"
 android:layout_height="wrap_content"
 android:layout_below="@+id/numberEnteredEt"
 android:layout_centerHorizontal="true"
 android:layout_marginTop="60dp"
 android:text="Ellenőriz" />

</RelativeLayout>
```

Az `activity_main.xml` fájlban az UI (User Interface) vagyis a felhasználói interfész kerül létrehozásra. Ebben a fájlban úgynevezett nézeteket hozunk létre, mindenkorát különböző céllal. A `TextView` a felhasználónak címzett üzenet megjelenítését szolgálja, az `EditText` a felhasználó számára a tippelt szám beírását, míg a `Button` pedig az "ELLENŐRIZ" gomb meglétét és működését.

```
package com.example.guessinggame;
import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;

import java.util.Random;

public class MainActivity extends AppCompatActivity implements View.OnClickListener {
```

```
public static final int MAX_NUMBER = 100;
public static final Random RANDOM = new Random();
private TextView msgTv;
private EditText numberEnteredEt;
private Button validate;
private int numberToFind, numberTries;

@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 msgTv = (TextView) findViewById(R.id.msg);
 numberEnteredEt = (EditText) findViewById(R.id. ←
 numberEnteredEt);
 validate = (Button) findViewById(R.id.validate);
 validate.setOnClickListener(this);

 newGame();
}

@Override
public void onClick(View view) {
 if (view == validate) {
 validate();
 }
}

private void validate() {
 int n = Integer.parseInt(numberEnteredEt.getText().toString ←
());
 numberTries++;

 if (n == numberToFind) {
 Toast.makeText(this, "Gratulálok ! Megvan a szám! " + ←
 numberToFind +
 " Összesen " + numberTries + " próbálkozásra ←
 volt szükséged", Toast.LENGTH_SHORT).show();
 newGame();
 } else if (n > numberToFind) {
 msgTv.setText(R.string.too_high);
 } else if (n < numberToFind) {
 msgTv.setText(R.string.too_low);
 }
}

private void newGame() {
 numberToFind = RANDOM.nextInt(MAX_NUMBER) + 1;
 msgTv.setText(R.string.start_msg);
 numberEnteredEt.setText("");
 numberTries = 0;
```

```
 }
}
```

Elérkeztünk a lényeghez, a `MainActivity.java` fájlunkhoz. mindenekelőtt már jól megszokottan importáljuk a számunkra szükséges könyvtárakat.

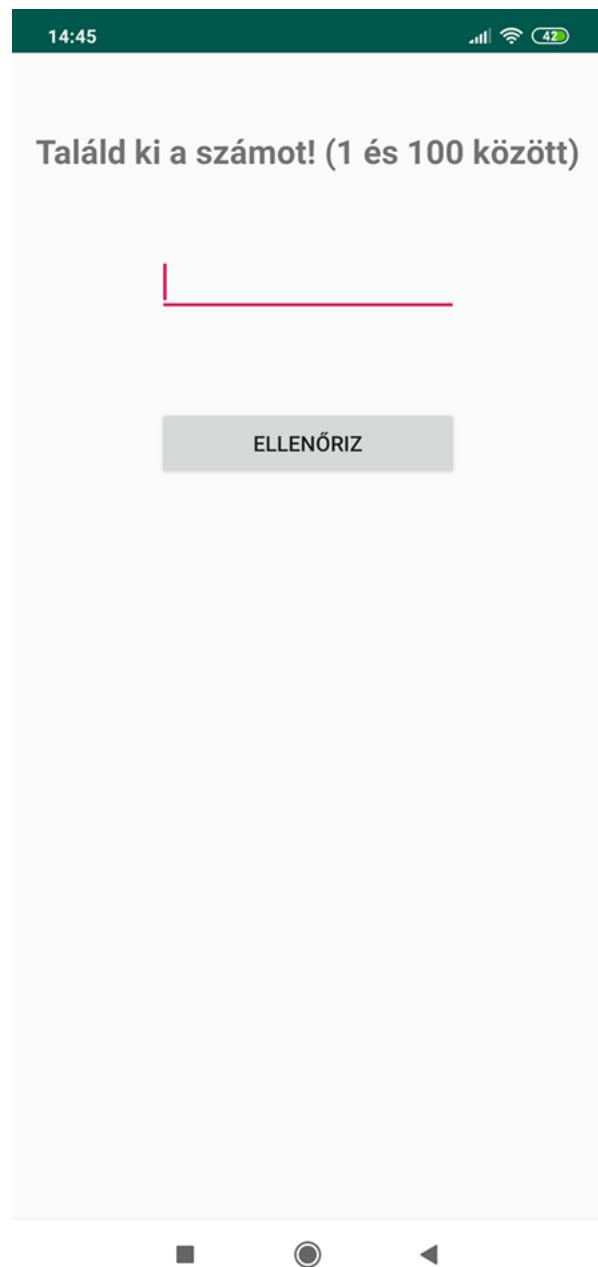
Létrehozzuk változóinkat és konstansainkat is, majd deklarálásra kerül a `onCreate` metódus.

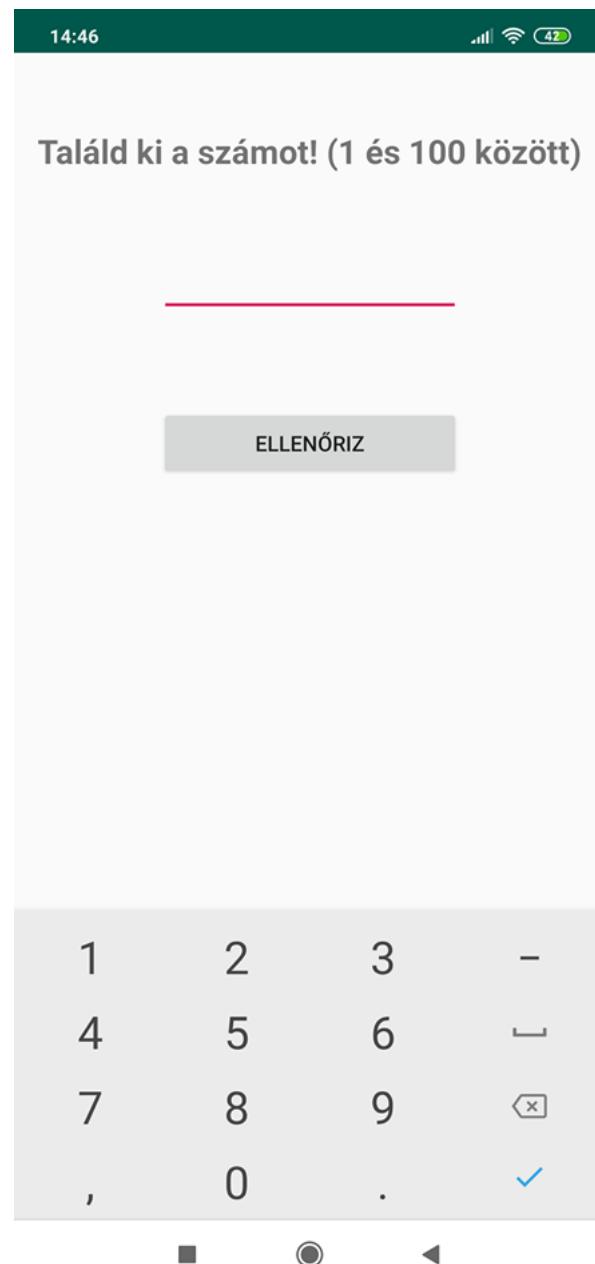
Az `onClick` metódus lehetővé teszi számunkra az "ELLENŐRZÉS" gomb működését, ha oda töltöögünk, ahol a gomb található, meghívásra kerül a `validate` metódus.

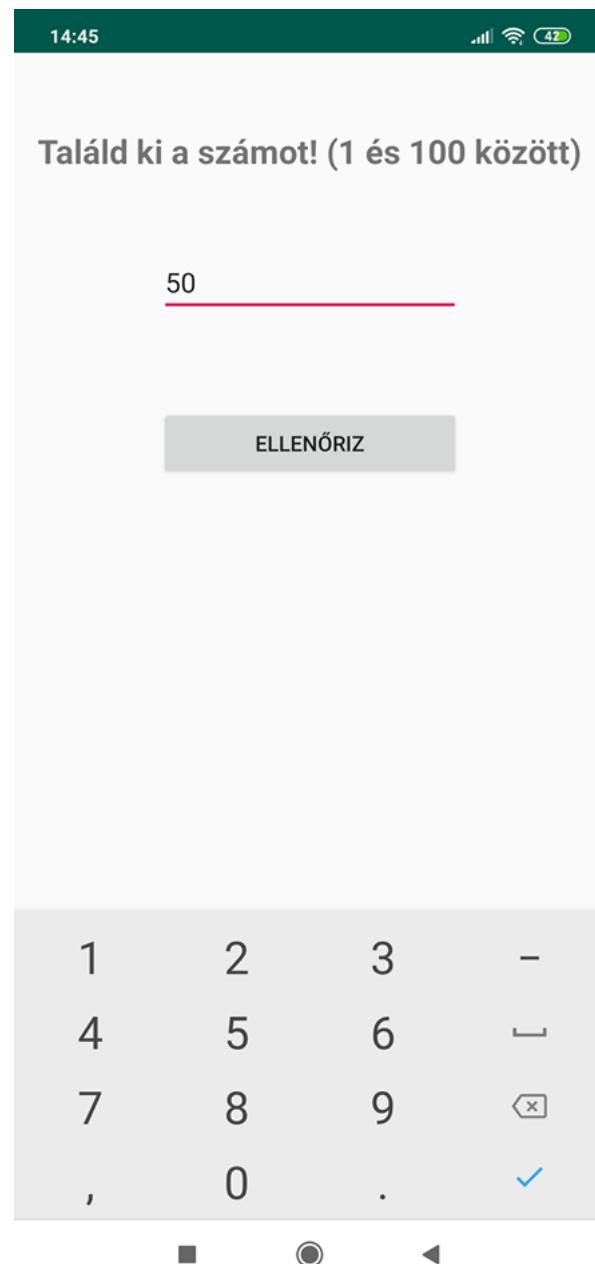
A `validate` metódus rendkívül sok mindenért felel. Eltárolásra kerül a felhasználó által bevitt szám az n változóban. Egy számláló értékét növeljük addig, amíg nem sikerült eltalálni a "gondolt" számot. Abban az esetben ha az általunk beírt és a generált szám megegyezik, gratulálunk a felhasználónak és közöljük vele, hanyadik próbálkozásra sikeresült eltalálnia a számot, valamint meghívjuk a `newGame` metódust, ezzel természetesen új játéket kezdve. Azokban az esetekben mikor nem ilyen szerencsés a felhasználó, annak függvényében, hogy lejjebb vagy feljebb kéne a számok közül tippelnie, az feladat elején már látható üzeneteket iratjuk ki számukra.

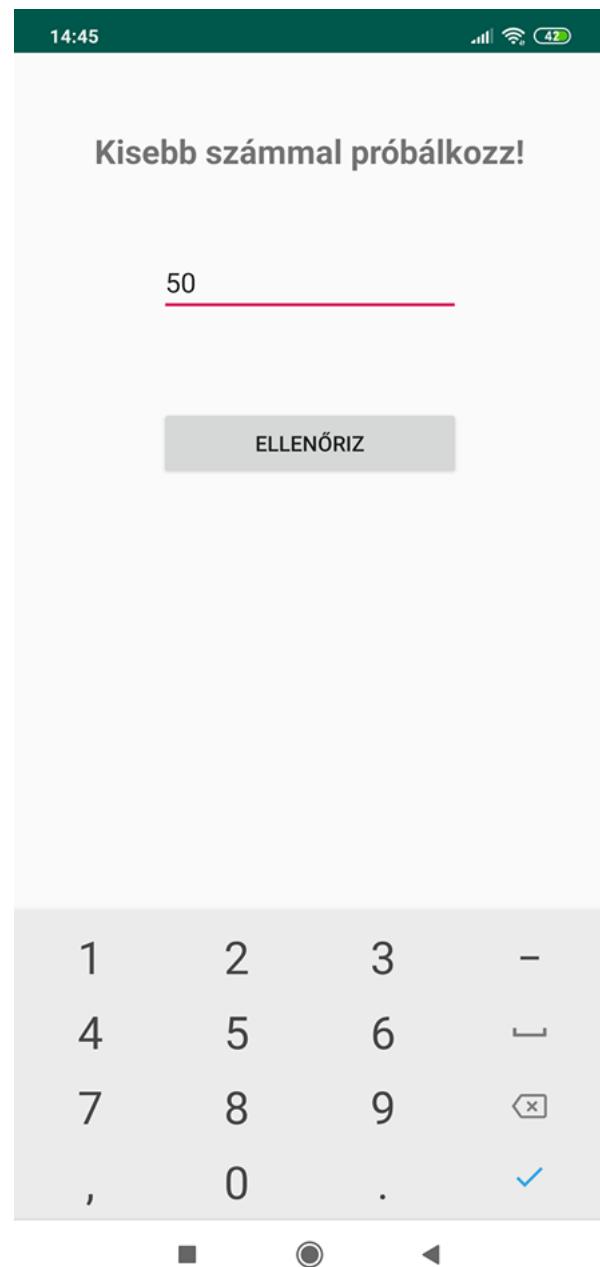
A `newGame` metódusban is történik egy más. Ekkor történik meg egy random szám sorsolás keretein belül az "eltalálandó" szám létrejötte és tárolása. Üzenetet közvetítünk a felhasználó számára és nullázzuk a számlálót.

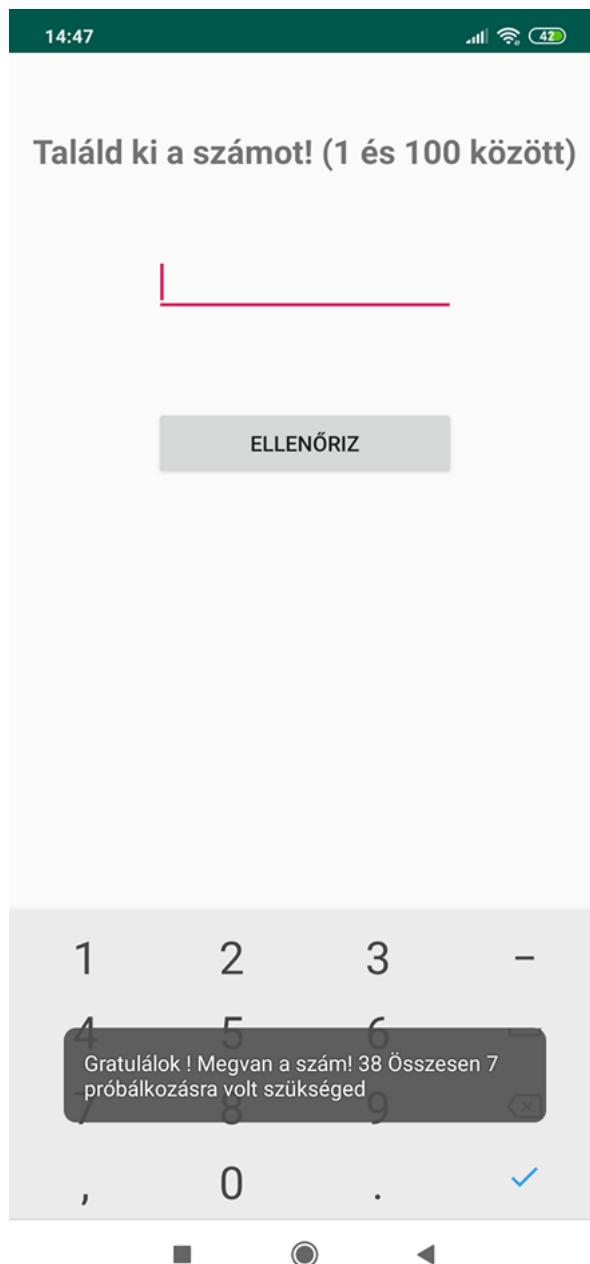
A játékot saját telefonon próbáltam ki, néhány képernyőkép működés közben:











### 18.3. Junit teszt

A [https://progpater.blog.hu/2011/03/05/labormeres\\_otthon\\_avagy\\_hogyan\\_dolgozok\\_fel\\_egy\\_pedat](https://progpater.blog.hu/2011/03/05/labormeres_otthon_avagy_hogyan_dolgozok_fel_egy_pedat) poszt kézzel számított mélységét és szórását dolgozd be egy Junit tesztbe (sztenderd védési feladat volt korábban).

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/lauda\\_codes/JUnit](https://gitlab.com/kincsa/bhax/tree/master/codes/lauda_codes/JUnit)

Tanulságok, tapasztalatok, magyarázat...

Forrás: <https://gyires.inf.unideb.hu/GyBITT/21/ch03s02.html>

Érdemes lenne azzal kezdeni, mi is a JUnit? Miért is jó nekünk a használata? A JUnit nem más, mint egy keretrendszer amit egységesítéshez szokás használni Java nyelv mellé. Utóbbi fogalom kis magyarázatra

szorul: akkor beszélünk egységesítésről, amikor egy adott kóddal együtt az adott kódot tesztelő osztállyal együtt kerül fejlesztésre.

Lássunk kódot! (forrása: [UDPROG repó](#))

```
public class BinfaTest {
LZWBinFa binfa = new LZWBinFa();

@org.junit.Test

public void tesBitFeldolg() {
 for (char c : "01111001001001000111".toCharArray())
 {
 binfa.egyBitFeldolg(c);
 }
}
```

AZ első két sorban semmiféle újdonságot nem találunk. Azonban a harmadik sorban már láthatunk számunkra eddig ismeretlen dolgokat, ezért a kód elemzését kezdjük azzal! A @-cal kezdődő sor jelöli annak a metódusnak a kezdetét, amit a JUnit tesztfuttatójának futtatnia kell. Viszont fontos, ha több ilyen metódus is van, azok végrehajtásának sorrendje nem lesz egyértelmű, ezért úgy érdemes kialakítani ezeket a teszteket, hogy függetlenek legyenek.

Egy teszteset @org.junit.Test annotációval ellátott metódussal kezdődik, törzsében pedig a tesztelendő metódus kerül meghívásra. Az itt kapott eredményt szükséges összhasonlítanunk az általunk várttal. Léteznek egyébként másfajta annotációk is, ezekről is [itt](#) olvashatunk.

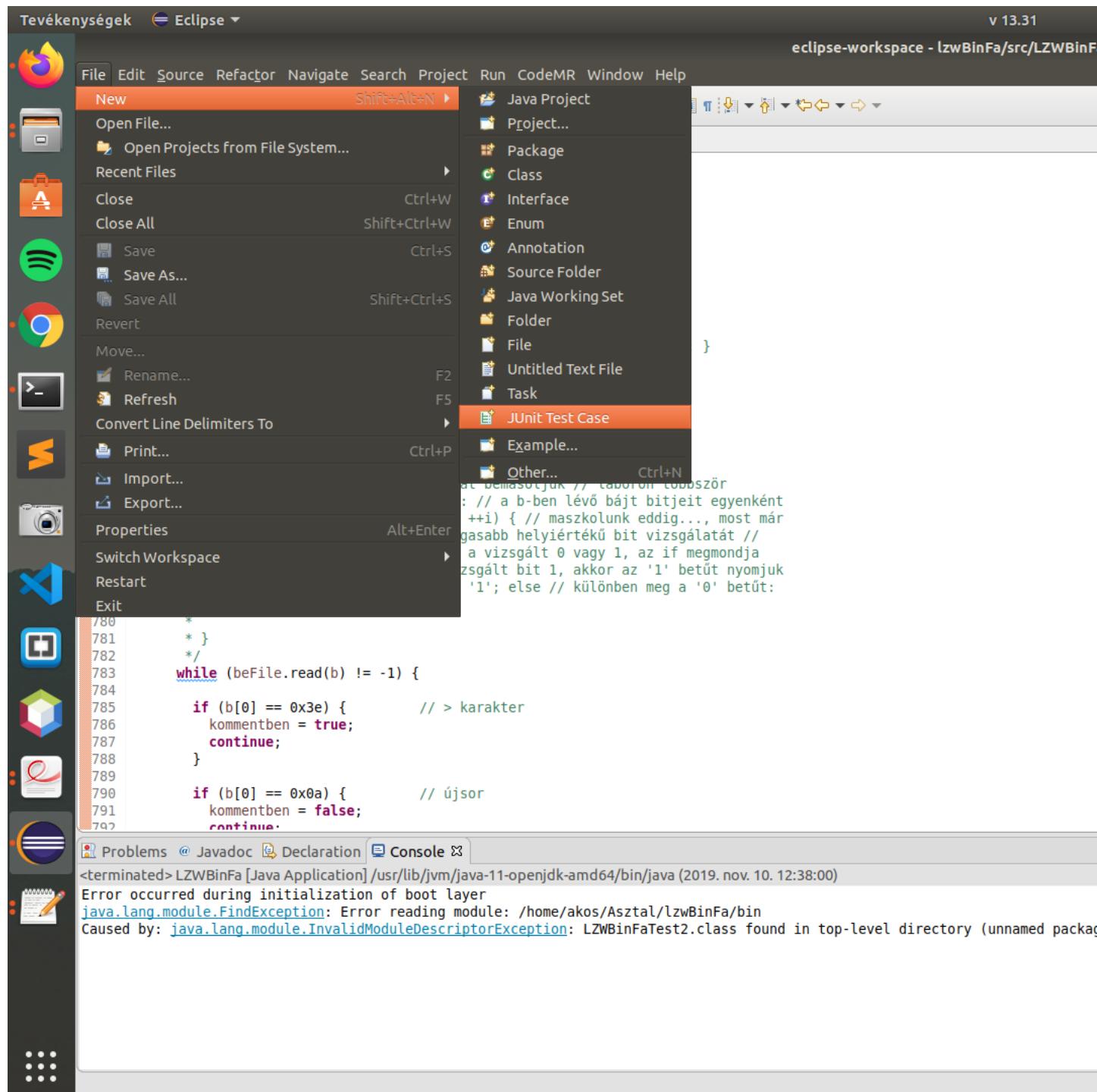
Nem olyan rég óta van lehetőségünk parametrizált teszteket is végezni, azonban ez már egy haladóbb technika, mi nem foglalkozunk vele.

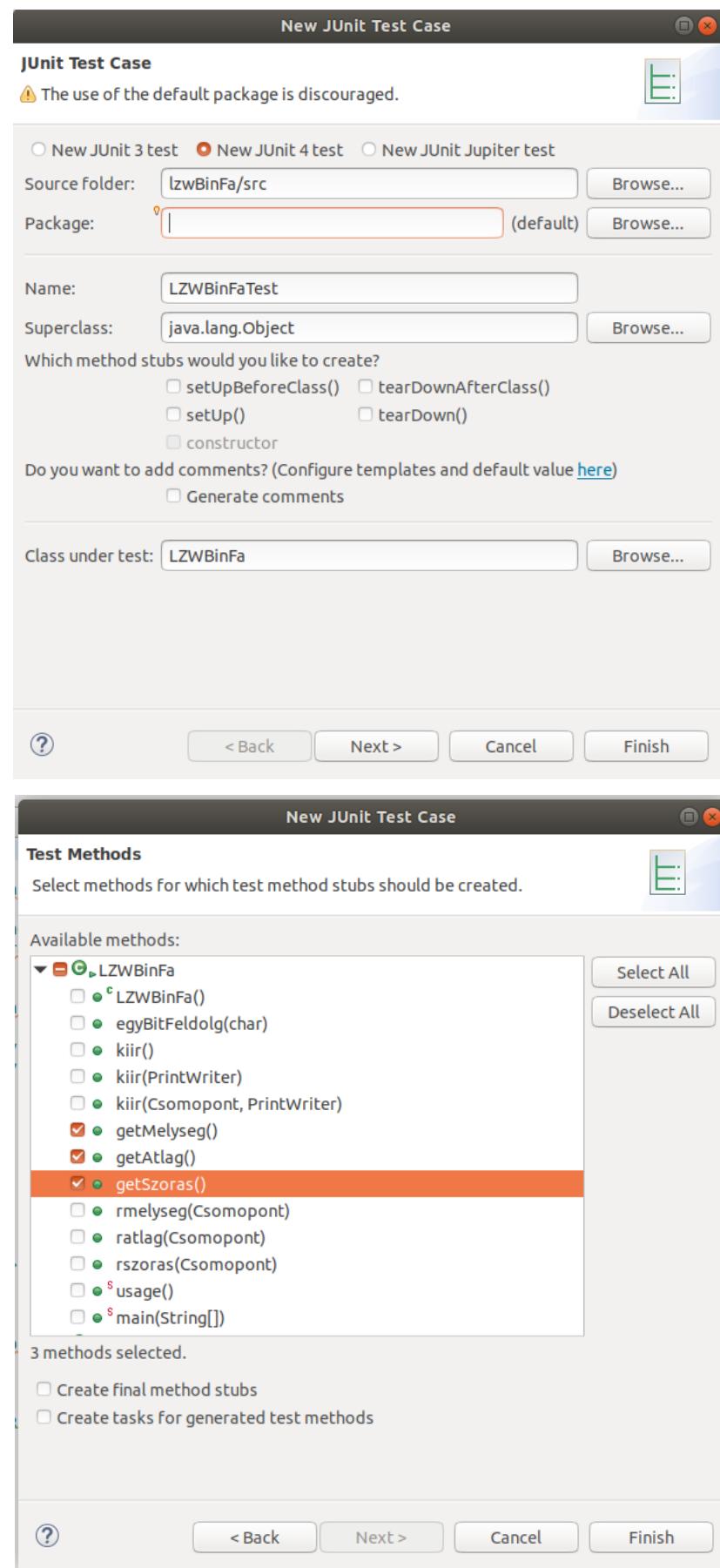
A törzsben található a `tesBitFeldolg` függvényünk, ami a teszesetünk lesz. Az `egyBitFeldolg` függvénnyel karakterenként dolgozzuk fel a megadott tömböt.

```
org.junit.Assert.assertEquals(4, binfa.getMelyseg(), 0.0);
 org.junit.Assert.assertEquals(2.75, binfa.getAtlag(), 0.001);
 org.junit.Assert.assertEquals(0.957427, binfa.getSzoras(), ←
 0.0001);
```

Az `assertEquals` függvénytel megvizsgáljuk, mennyi is az eltérés - vagyis inkább, hogy megegyezik-e a két érték - a várható mélység, átlag és szórás valamint ezek tényleges értékei között. A függvény rendre, mindenkor esetben három paraméterrel rendelkezik. Az első az az érték, amit 'kapunk kellene', a második a saját programunk által kapott érték, míg a harmadik a 'hibahatár' lényegében, vagyis hogy mennyi lehet a maximális eltérés az előbb említett két érték között.

Ennek ellenőrzésére az Eclipse-t illetve egy, a Marketplace-én megtalálható plugint használtam, a JUnit-Tools 1.1.0-t. Ennek telepítését követően a következőképp végezhetjük el a tesztjeinket.:





# 19. fejezet

## Helló, Calvin!

### 19.1. MNIST

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel,

[https://progpater.blog.hu/2016/11/13/hello\\_samu\\_a\\_tensorflow-bol](https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol) Háttérként ezt vetítsük le:

<https://prezi.com/0u8ncvvoabcr/no-programming-programming/>

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/calvin\\_codes/MNIST](https://gitlab.com/kincsa/bhax/tree/master/codes/calvin_codes/MNIST)

Tanulságok, tapasztalatok, magyarázat...

Tutorok: Harmati Norbert, Szegedi Csaba

Érdemes lenne azzal kezdeni, mi is az MNIST. Lényegében kézzel írott arab számjegyek adatbázisa, mely összesen 60000 darab 28x28 pixel méretű, greyscale PNG képállományt tartalmaz. Ez a 60000 állomány azonban alapvetően két részre bontható, hiszen 50000 darab tanítási és 10000 darab tesztelési képet tartalmaz. Előbbiek alapján tanulja meg a gép számjegyeket, majd a tanulásának eredményességét ellenőrzi 10 ezer képre nézve.

Lássuk a gépi tanulást megvalósító kódunkat!

```
import keras
from keras.datasets import fashion_mnist
from keras.layers import Dense, Activation, Flatten, Conv2D, ←
 MaxPooling2D
from keras.models import Sequential
from keras.utils import to_categorical, np_utils
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import os
```

Kezdődik az egész azzal, hogy a programunk legelején importáljuk a megoldásunkhoz szükséges könyvtárat. Számos alkalommal fordult elő, hogy ezen könyvtárak közül valamelyik nem volt megtalálható a gépen. Ezeket mindenkor egy pip install 'library\_neve' parancs segítségével orvosolni tudjuk. Példának okáért ha a matplotlib könyvtárral még nem rendelkezünk, akkor: pip install matplotlib.

Ennek említését azért tartottam fontosnak, mert nem egyszer jött velem szemben ez a probléma a feladat megoldása során.

```
(train_X,train_Y), (test_X,test_Y) = tf.keras.datasets.mnist. ←
 load_data()
```

Az előző sor azért felel, hogy tudjunk dolgozni az adatbázisunkkal, itt töltjük be a `load.data()` függvény segítségével. Most már rendelkezésünkre áll a több tízezer képállomány, mellyel dolgozni tudunk majd. Ezek vektorokban kerülnek tárolásra.

```
train_X = train_X.reshape(-1, 28,28, 1)
test_X = test_X.reshape(-1, 28,28, 1)
```

Elkészítjük a tanításra és tesztelésre használatos számok tárolására alkalmas vektorokat, melyeket a `reshape` függvény segítségével számunkra és a feladatnak megfelelő formájúra hozunk. Ez esetünkben azt jelenti, hogy 28 db., 28 db. elemet tartalmazó vektorra bontjuk adatainkat. Az első paraméter -1, vagyis ezt minden tagra eljátsszuk. Ha más szám állna ott, például 5, akkor nyilván csak a vektor első 5 tagja került volna ilyen formában átalakításra.

```
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255
test_X = test_X / 255
```

A vektorok típusát átállítjuk, majd osztjuk a látható számértékkel azért, hogy a képeket alkotó képpontok értéke megfelelő legyen a lehető leggyorsabb tanítási folyamathoz.

```
train_Y_one_hot = to_categorical(train_Y,10)
test_Y_one_hot = to_categorical(test_Y,10)
model = Sequential()
```

Ebben a pár sorban megjelenik az úgynevezett one hot encoding is. Ez azért lényeges, mert a modell nem tud kategorikus adatokkal (a `to_categorical` függvénytel kerül ilyen formájúra) dolgozni. Ezért 0-sok és 1-esek sorozatára kerül felbontásra az adott szám. Erről az encodingról több [itt](#) és [itt](#) olvasható. Ezen túl beállításra kerül a modellünk típusa is.

```
model.add(Conv2D(64, (3,3), input_shape=(28, 28, 1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(64, (3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())
model.add(Dense(64))

model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(loss=keras.losses.categorical_crossentropy, optimizer ←
 =keras.optimizers.Adam(), metrics=['accuracy'])
```

Rendre az `add` függvényt használjuk arra, hogy modellünkhez újabb rétegeket adjunk hozzá. Láthatunk példát az első sorban arra, hogy milyen az, mikor egy konvolúciós réteget adunk hozzá a modellünkhez. Amit erről tudni kell: első paramétere a neuronok száma, a második az úgynevezett detektor, a harmadik pedig az input shape, mely esetünkben a 28x28-as greyscale állomány lesz. Tehát egyértelműen egy úgynevezett konvolúciós neurális hálózatunk lesz (CNN angolul), amely a képanalitika területén igen népszerű modell. A következő sorban aktiválunk egy úgynevezett ReLU-t (Rectified Linear Unit), amit ha nagyon le szeretnénk fordítani magyarra, talán a "helyesbített lineáris egység" lenne a legmegfelelőbb. Az azt követő sorban a `pool_size`-zal azt adjuk meg, mennyi adat kerüljön feldolgozásra, ennek két argumentuma van, egy függőleges és egy vízszintes érték. A `compile` függvénytel megindul a tanítási folyamat.

```
model.fit(train_X, train_Y_one_hot, batch_size=64, epochs=1)

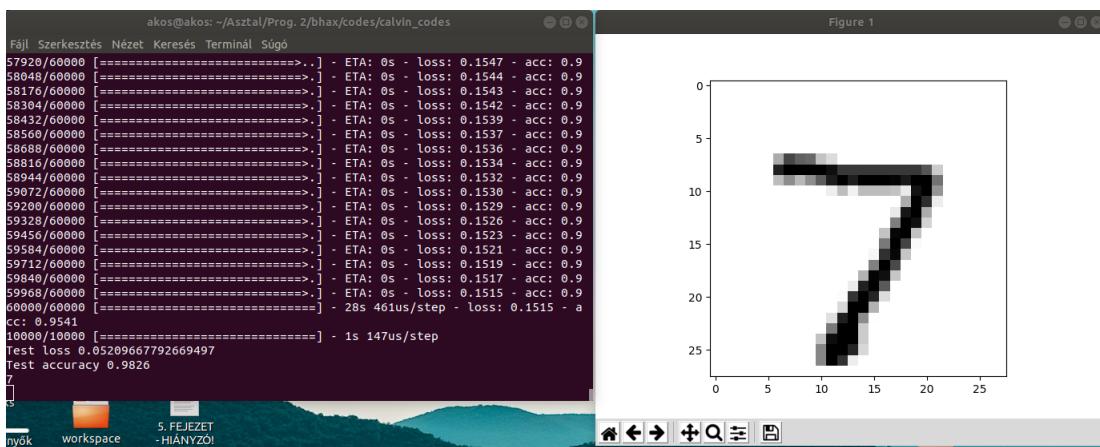
test_loss, test_acc = model.evaluate(test_X, test_Y_one_hot)
print('Test loss', test_loss)
print('Test accuracy', test_acc)

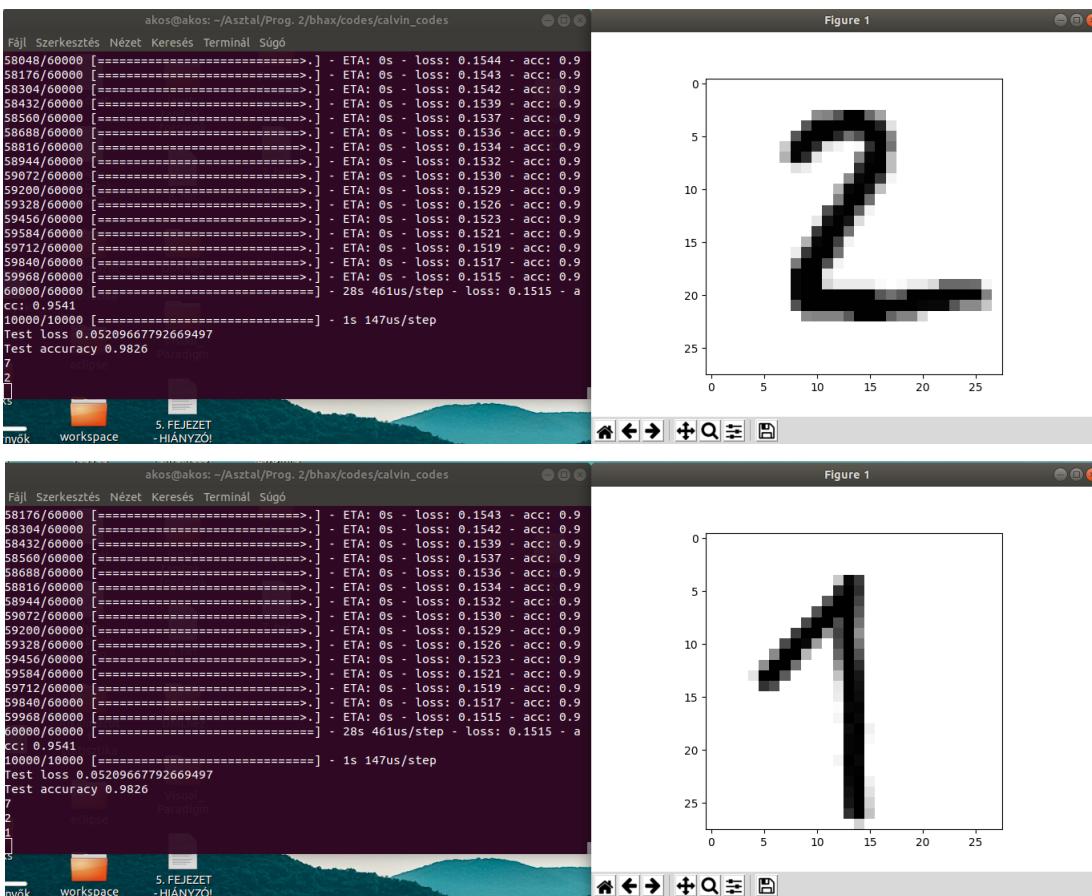
predictions = model.predict(test_X)

print(np.argmax(np.round(predictions[0])))
plt.imshow(test_X[0].reshape(28, 28), cmap = plt.cm.binary)
plt.show()
print(np.argmax(np.round(predictions[1])))
plt.imshow(test_X[1].reshape(28, 28), cmap = plt.cm.binary)
plt.show()
img = Image.open('one.png').convert("L")
img = np.resize(img, (28,28,1))
```

A `fit` függvényel elkezdjük beállítani a tanítás jellemzőit. Itt inkább az utolsó két paraméter érdekes. A `batch_size` a tanulási sebesség, míg az `epochs` az lesz, hányszor hajtsa végig a folyamatot.

Majd kiiratjuk a tanulási folyamat során való veszeséget majd a pontossági értéket is. Eltároljuk a prediction-öket majd megjelenítjük az első, majd második feldolgozott képállományunkat a gép általi prediction-nel egyetemben. A harmadik képet mi adjuk- és nyitjuk meg az `Image.open` függvény segítségével. Mi jelen esetben egy 1-es számmal teszteltünk, lássuk, mire sikerült jutnia a modellünknek!





Látható, hogy igen jó százalékos pontossággal sikerült neki eltalálni a számokat. A kézzel írott 1-esünket is sikerült felismernie.

## 19.2. CIFAR-10 -deprecated

Az alap feladat megoldása, +saját fotót is ismerjen fel,

[https://progater.blog.hu/2016/12/10/hello\\_samu\\_a\\_cifar-10\\_tf\\_tutorial\\_peldabol](https://progater.blog.hu/2016/12/10/hello_samu_a_cifar-10_tf_tutorial_peldabol)

Megoldás videó:

Megoldás forrása: [https://gitlab.com/kincsa/bhax/tree/master/codes/calvin\\_codes/CIFAR](https://gitlab.com/kincsa/bhax/tree/master/codes/calvin_codes/CIFAR)

Tanulságok, tapasztalatok, magyarázat...

Tutorok: Harmati Norbert, Szegedi Csaba

A feladat hasonlít az első, MNIST-es feladatunkhoz. Az alepető különbség az, hogy most számjegyek helyett tárgyakat, élőlényeket (összefoglalóan mondhatjuk azt, objektumokat) kell majd felismernie. Szintén 60 ezer darab képpel fog dolgozni, ugyanúgy 50 ezer tanítási és 10 ezer tesztelési képet tartalmaz. A kép-állományok azonban most már nem greyscale-ek, hanem RGB-ek lesznek illetve az előző méret helyett 32x32-ek. Alapvetően 10 féle dologról találhatók meg képek az adatbázisban, ezekre néhány példa: autó, szarvas vagy éppen madár.

Azonban ahogy arra már a feladat elején utaltam, nagyrészt egyezik a megoldás logikája az első feladatunkkal. Ezzel együtt jár az is, hogy a kód sem sokban tér el tőle. Ezért ezen feladat esetében is elemzésre kerül a kód, azonban az újdonságokat az előző kódhoz képest félkövér betűvel fogom jelezni. Lássuk tehát!

```
import keras
from keras.datasets import fashion_mnist
from keras.layers import Dense, Activation, Flatten, Conv2D, ←
 MaxPooling2D
from keras.models import Sequential
from keras.utils import to_categorical,np_utils
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import os
```

Ismét azzal kezdünk, hogy a programunk legelején importáljuk a megoldásunkhoz szükséges könyvtákat. A hiányzó könyvtárakat még mindig egy pip install 'library\_neve' parancs segítségével orvosolni tudjuk.

```
(train_X,train_Y), (test_X,test_Y) = cifar10.load_data()
```

**Értelemszerű, most egy másik adatbázis képeivel dolgozunk majd, ezeket töltjük be.**

```
train_X = train_X.reshape(-1,32,32,3)
test_X = test_X.reshape(-1, 32,32, 3)
```

**Az első paramétert leszámítva minden változik az előzőhez képest, a lényeg viszont ugyanaz. A második és harmadik 32 lesz, hiszen 32x32-es képallokányokkal lesz dolgunk, a negyedik paraméterként megadott 3-as pedig arra utal, RGB képekkel fogunk dolgozni.**

```
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255
test_X = test_X / 255
```

A vektorok típusát átállítjuk, majd osztjuk a látható számértékkel azért, hogy a képeket alkotó képpontok értéke megfelelő legyen a lehető leggyorsabb tanítási folyamathoz.

```
train_Y_one_hot = to_categorical(train_Y,10)
test_Y_one_hot = to_categorical(test_Y,10)
model = Sequential()
```

Ebben a pár sorban megjelenik az úgynevezett one hot encoding is. Ez azért lényeges, mert a modell nem tud kategorikus adatokkal (a to\_categorical függvényel kerül ilyen formájúra) dolgozni. Ezért 0-sok és 1-esek sorozatára kerül felbontásra az adott szám. Erről az encodingról több [itt](#) és [itt](#) olvasható. Ezen túl beállításra kerül a modellünk típusa is, ami szekvenciális lesz.

```
model.add(Conv2D(64, (3, 3), activation='relu', input_shape=(32, ←
 32, 3)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(256))
```

```
model.add(Activation('relu'))

model.add(Dense(10))
model.add(Activation('softmax'))
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss=keras.losses.categorical_crossentropy, optimizer ←
 =sgd, metrics=['accuracy'])
img = Image.open('allat.png').convert("L")
img = np.resize(img, (32, 32, 3))
im2arr = np.array(img)
im2arr = im2arr.reshape(1, 32, 32, 3)
```

Rendre az add függvényt használjuk arra, hogy modellünkhez újabb rétegeket adjunk hozzá. Láthatunk példát az első sorban arra, hogy milyen az, mikor egy konvolúciós réteget adunk hozzá a modellünkhez. Amit erről tudni kell: első paramétere a neuronok száma, a második az úgynevezett detektor. **Mivel teljesen más jellemzőjű képállományokkal dolgozunk majd, értelemszerűen megváltoznak az input\_shape függvény paraméterei is hiszen most már 32x32-es, ráadásul RGB-s képeket szeretnénk feldolgoztatni majd megtanítatni. Továbbá több neuront használunk fel ez alkalommal.** A következő sorban ismételten aktiválunk egy úgynevezett ReLU-t (Rectified Linear Unit). Ezt követően a pool\_size-zal megadjuk, mennyi adat kerüljön feldolgozásra, ennek két argumentuma van, egy függőleges és egy vízszintes érték. A compile függvényel megindul a tanítási folyamat.

```
model.fit(train_X, train_Y_one_hot, batch_size=64, epochs=1)

test_loss, test_acc = model.evaluate(test_X, test_Y_one_hot)
print('Test loss', test_loss)
print('Test accuracy', test_acc)

predictions = model.predict(test_X)
cifar_classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', '←
 dog', 'frog', 'horse', 'ship', 'truck']
print(cifar_classes[np.argmax(np.round(predictions[0]))])
plt.imshow(test_X[0].reshape(32, 32,-1), cmap = plt.cm.binary)
plt.show()
print(cifar_classes[np.argmax(np.round(predictions[526]))])
plt.imshow(test_X[526].reshape(32, 32,-1), cmap = plt.cm.binary)
plt.show()

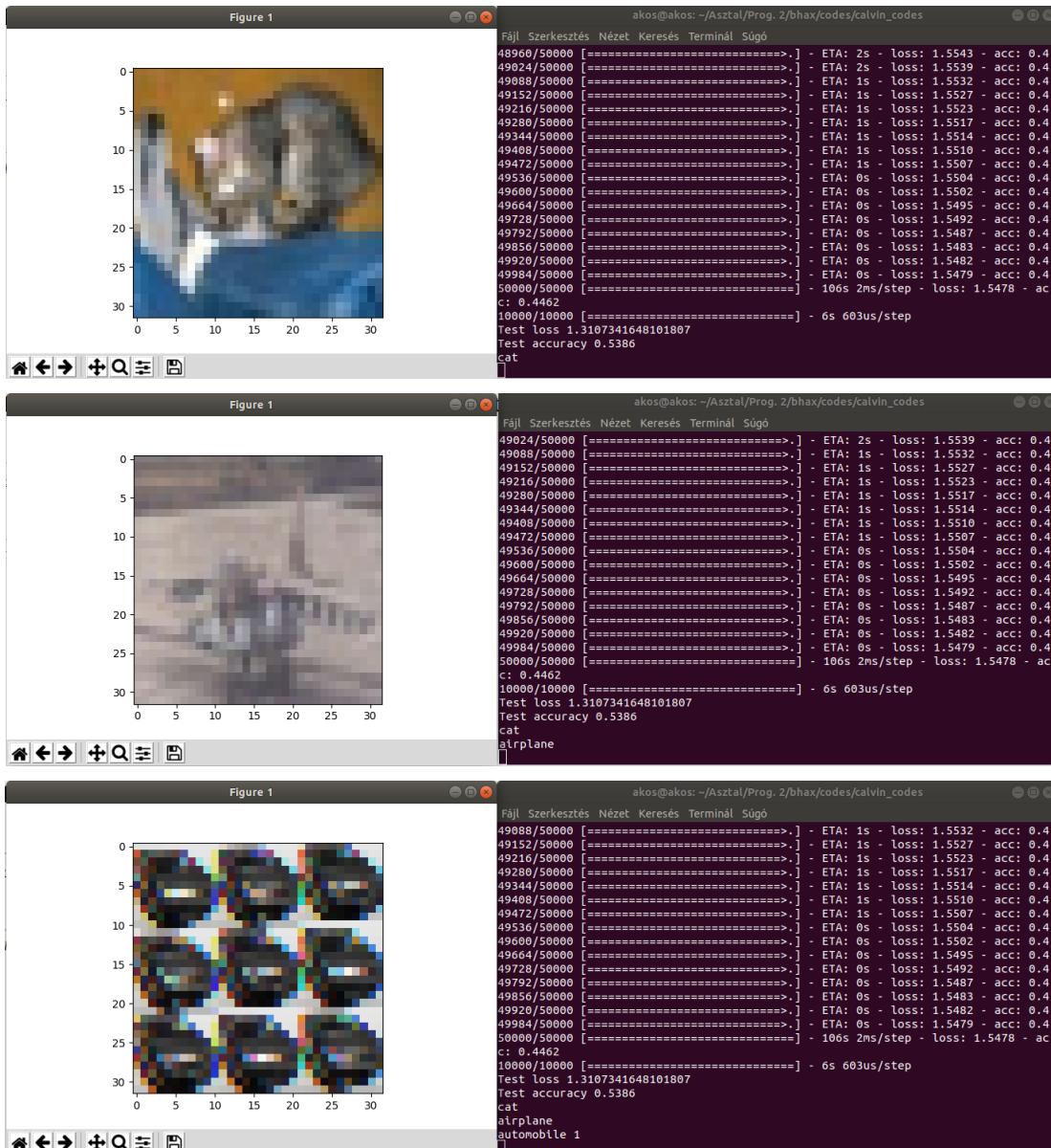
print(cifar_classes[np.argmax(np.round(model.predict(im2arr)))], np. ←
 argmax(np.round(model.predict(im2arr))))
plt.imshow(im2arr[0].reshape(32, 32, 3), cmap = plt.cm.binary)
plt.show()
```

A fit függvényel elkezdjük beállítani a tanítás jellemzőit. Itt inkább az utolsó két paraméter érdekes. A batch\_size a tanulási sebesség, míg az epochs az lesz, hányszor hajtsa végig a folyamatot.

Majd kiiratjuk a tanulási folyamat során való veszteséget majd a pontossági értéket is. Eltároljuk a prediction-öket. **Fontos megemlíteni az utolsó, egyik legszembetűnőbb különbséget és egyben érdekességet az MNIST feladathoz képest. Most meg kell adnunk kézzel egy tömböt, mely tartalmazza azokat a osztályokat, lényegében tárgyainkat, melyekről képeket találhatunk adatbázisunkban. Ez lesz a**

**cifar\_classes tömb.** Majd megjelenítjük az első, majd második feldolgozott képállományunkat a gép általi prediction-nel egyetemben. Utolsóként pedig az a kép kerül megjelenítésre amit mi adtunk be neki és remélhetőleg értelmezte, megtanulta. Lássuk, mire jutott:

Virtuális környezetben futtatjuk is:



Hát...látható, ha a számokat nézzük, nem a leg pontosab... Fontos hozzá tenni azért azt is, hogy ha többször futtattam volna le a tanítást, sokkal pontosabb lett volna a felismerés. Amit viszont elég impresszívnek tartottam az az, hogy még ilyen pontossági mutatók mellett is képes volt értelmezni és felismerni minden háróm képet, beleértve az általam letöltött autós képet is:

A letöltött kép forrása: <https://m.blog.hu/pr/progpater/image/matchbox.png>



## 19.3. Android telefonra a TF objektum detektálója

Telepítsük fel, próbáljuk ki!

Megoldás videó:

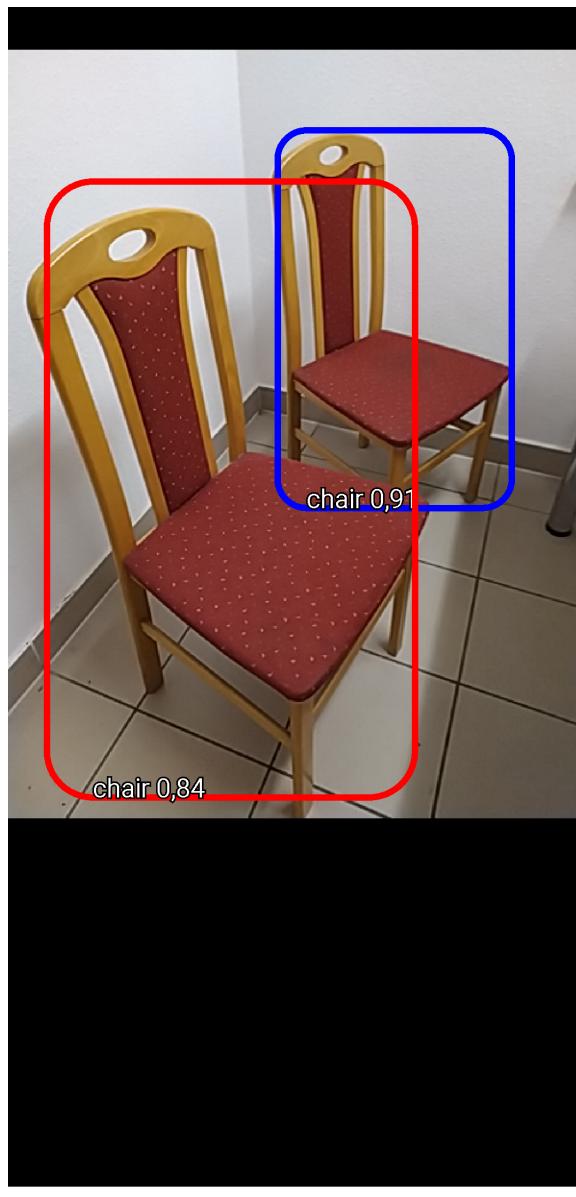
Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

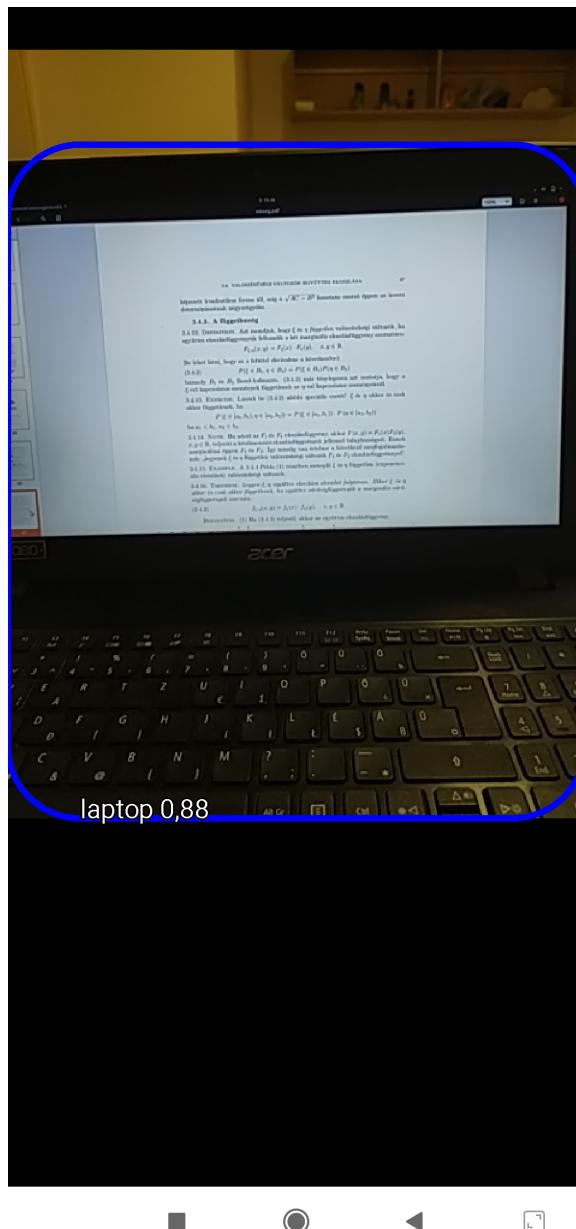
A TensorFlow github repóbjából sikerült lekapnom a megfelelő fájlokat, majd azokat Android Studioban egy apk-ként kimentve már kipróbálható volt az alkalmazás. A továbbiakban az látható hogy a TF Detector hogyan működött, amikor a koliban kipróbtam néhány random tárgyon. Azt kell mondani, az esetek többségében igen pontos volt, habár természetesen bőven vannak még hiányosságai. Egyébként az okostelefonunk áruházát böngészve rátaláltam objektum-detektáló TF-alkalmazásra, így azoknak is elérhető kipróbálásra, akik nem rajonganak az előbb említett Android Studio-s megoldásért..

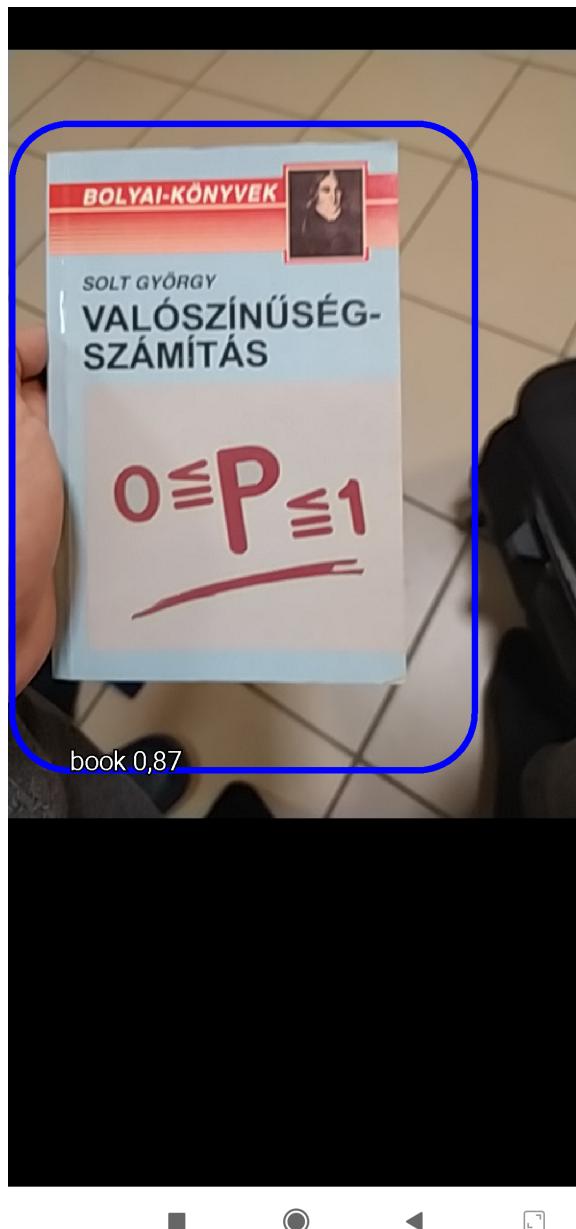
Most, hogy már felvezettem, lássuk, miként is teljesített:



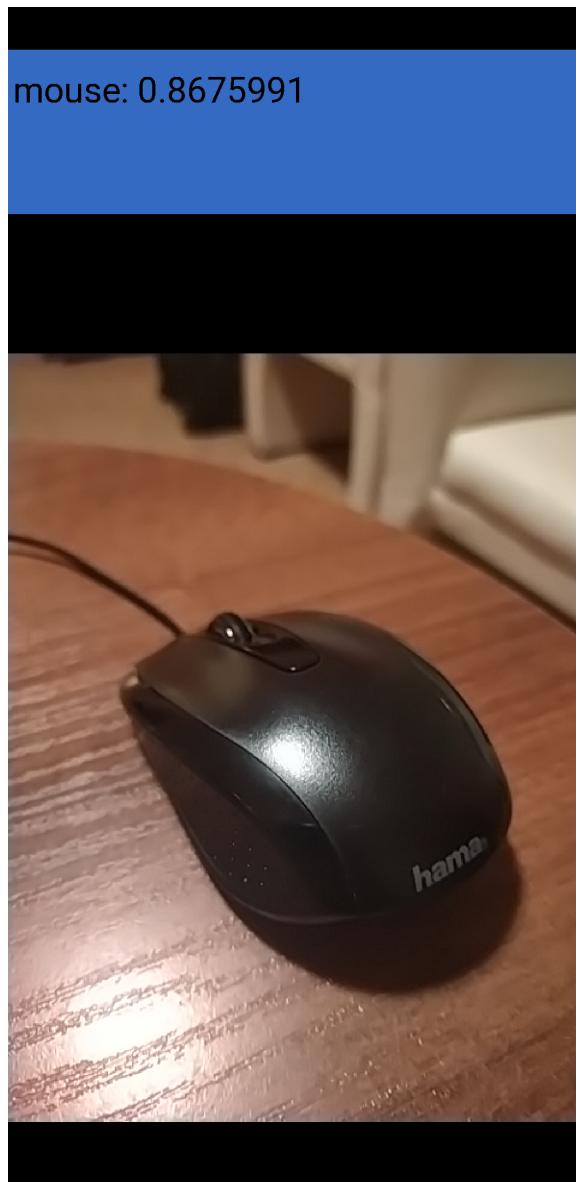




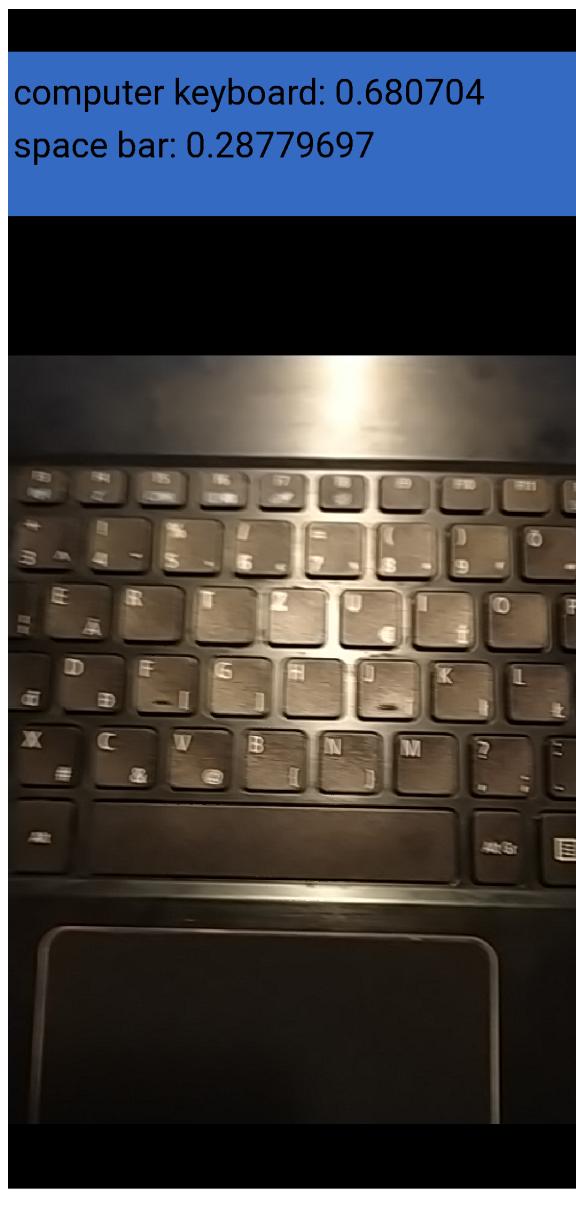




Majd kipróbáltam a csomag második tagját, a TF Classify-t is. Annak működése:







## 20. fejezet

# Helló, Berners-Lee!

### 20.1. Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba

A Python programozási nyelv történeti hátterével kezdi a könyv. A '90-es években került megalkotásra a mai napig, elsősorban prototípusok elkészítésére népszerű, objektumorientált és platformfüggetlen nyelv. Viszonylag könnyen tanulható nyelvként hivatkozik rá az író. A nyelv érdekessége, hogy az eddig tanult C, C++ vagy akár Java nyelvekkel ellentétben nincs szükség fordításra, az interpreter elboldogul csupán a forrással.

A könyv a nyelv bemutatását az alapvető szintaxissal kezdi. Ebből a pár sorból megtudhatjuk, hogy behúzásalapú a szintaxisa vagyis itt elfelejthető az eddigi tanulmányaink során használt kapcsos-zárójelezés, sőt mi több, az utasítások a sor végéig tartanak, így az utasításokat eddig lezáró pontosvesszőt sem használja a nyelv. Példával kerül szemléltetésre, hogyan is néz ki minden a gyakorlatban. Megkapjuk a kulcsszavak listáját is és megtudjuk, hogy a megjegyzéseket a kettőskereszt karakterrel jelölhetjük.

A nyelvben minden adatot objektum reprezentál. Érdekesség, hogy nincs szükség egy változó típusának konkrét megadására, van olyan okos és fejlett a rendszer, hogy kitalálja, "mire gondolt a költő". Az adattípusok egyébként a következők lehetnek: számok, sztringek, ennesek, listák, szótárak, melyek közül az utolsó 2-ről még nem is hallottam. Mindegyikre mutat gyakorlati példát is a könyv egy pár soros kódcsípetben. A változók a Pythonban az egyes objektumokra mutató referenciák, értékadásuk a már megszokott egyenlőségjellel történik. A "del" kulcsszó szerepét is taglalja az olvasmány, ezzel változó hozzárendelést törölhetünk. A már előbb említett adattípusok, pontosabban a listákkal és szótárakkal végezhető műveleteket is megtalálhatjuk több táblázatban.

Ezután a nyelv eszközei rész következik a könyvben, ami megemlíti a `print` metódust, mellyel a standard kimenetre írhatunk tetszőleges változókat, szöveget. Természetesen elágazásokkal is operál a nyelv, csak úgy, mint ciklusokkal. Természetesen a `for` és a `while` ciklusok itt is megtalálhatók. Említésre kerül a `range` és az `xrange` függvények is, amelyek két, általunk megadott alsó- és felső határ közötti értékeket sorol fel alapvetően 0-tól kezdődően, megadott lépésközzel. Érdekesség, hogy a C++ nyelvvel ellentétben a Java-ban minden függvény virtuális.

Létrehozhatunk címkeket is melyekre a `goto` parancsral ugorhatunk. A `comefrom` az előző parancs ellenére. A függvények definiálása a `def` kulcsszóval történik.

A Python lehetőséget ad az objektumorientált fejlesztési eljárások használatára is, ezek is példákon keresztül kerülnek szemléltetésre.

Mivel a Python a mobiltelefonra való fejlesztői munka (egyik) éollovása, modulok kerültek létrehozásra annak érdekében, hogy ez a munka könnyebb legyen. Ilyen modulok például: camera amely segítségével a készülék kamerájához férhetünk hozzá, audio, ami felelős a hangfelvételekért, messaging, ami többek között az SMS-üzenetekért felel vagy épp a sysinfo, ami a telefonnal kapcsolatos infók lekérdezését teszi lehetővé.

A nemkívánatos esetek előfordulásakor a kivételkezelés lép életbe. A Java-val és C/C++ -szal ellentétben itt nem try-catch, hanem try-except szerkezetről beszélhetünk.

A fejezet az előbb bemutatott dolgokat mutatja be részletesebben, egy-egy szemléletes példán keresztül.

A könyvet (részletét) alapvetően korrektnak tartottam, érthetően magyarázza a dolgokat. Összességében tetszett, és örültök, hogy egy újabb (szimpatikus) programozási nyelvbe nyerhettem ezzel egy kis betekintést.

## 20.2. Java és C++ nyelv összehasonlítása Nyékyné Dr. Gaizler Judit Java 2 útikalauz programozóknak 5.0 I-II. segítségével

### 1. fejezet - Nyolc lap alatt a nyelv körül

Kezdjük a legelején. A Java a C++ nyelvhez hasonlóan egy objektumorientált nyelv. Lényegében ez nem jelent mást, minthogy egy Java nyelvben írt program objektumokból és a "tervrajzok" szerepét betöltő osztályokból áll. Az osztályok pedig változókból és metódusokból állnak. A C++ esetében is ugyanez a helyzet.

A könyv szerzői egy egyszerű Hello, World!-ön keresztül mutatják be a Java fordítójának működését. Ebből a szempontból eltér a C++ és a Java, ugyanis utóbbi esetében az ún. Java Virtuális Gép (interpreter) fogja értelmezni a fordítóprogram által előzőleg kreált ún. bajtkódot.

A Java nyelv sikerét alapvetően a világhálón fellelhető oldalakon betöltött funkcióinak köszönheti. Ez alapvetően számomra újdonság, a C++ során nem találkoztam hasonlóval. Egy HTML oldalba beépített Java programot hoz példaként a könyv, mely programokat gyűjtönéven appleteknek nevezzük. Az applet forráskódjában megjelenik az import kulcsszó (ezzel eddig tanulmányaink során is találkoztunk már), amivel a feladat végrehajtásához szükséges könyvtárakat teszünk elérhetővé számunkra.

A következő 3 alfejezet rendre a változókkal, konstansokkal és a megjegyzésekkel foglalkozik. (ebben a sorrendben) A változók típusa lehet: boolean (logikai true vagy false értékkel), char (16 bites Unicode karakter), byte (8 bites előjeles egész szám), short (16 bites előjeles egész szám), int (32 bites előjeles egész szám), long (64 bites előjeles egész szám), float (32 bites lebegőpontos racionális szám), double (64 bites lebegőpontos racionális szám). A konstansok szerepe ugyanaz, mint C++ esetén, megadásuk a final kulcsszóval történik. A megjegyzések Java esetén is lehetnek egy- vagy akár többsorosak, sőt dokumentációsak is, utóbbi. Mindegyikre hoz példát a könyv.

Java nyelvben a class kulcsszóval hozható létre osztály. Tetszőleges sorrendben felsorolhatók az adattagjai illetve metódusai, amik az eddigi tanulmányaink során tanult változóknak és függvényeknek nevezhetünk lényegében. Az adattagok és metódusok láthatóságát egyenként lehet beállítani. A könyv által hozott példában megjelenik egy új típus, melynek neve String. Igen, a Java-ban a karakterláncok kezelésére létre lett hozva egy új osztály, ez merőben eltér az eddig használt és megszokott (akár) C++-os megoldástól, ahol a String típus lényegében karakterek tömbje volt. Ha van osztályunk, természetesen objektumot is létre hozhatunk belőle, mindezt a new operátor segítségével. Ez az operátor helyet foglal le az új objektum számára és erre a területre vonatkozó referenciaval tér vissza. Megjelenik a static kulcsszó szerepe

is, ha ezzel deklarálunk elemeket, akkor azok az elemek magához az osztályhoz fognak tartozni. Ez utóbbi is újdonság az eddigi tanulmányainkhoz képest.

Nyilvánvalóan a Java nyelvben is kifejezetten fontos, hogy egyes, nem elvárt helyzetben hogyan reagáljon a program. Ez jellemzően `try-catch` szerkezetben valósul meg, amelyet most nem ragozok, már találkoztunk vele tanulmányaink során, ugyanígy működik C++ alatt is. Ez hivatalosabban fogalmazva nem más, mint a kivételkezelés.

Érdemes még pár szót ejteni a nyelv biztonsági lehetőségeiről, amely rugalmasan az adott igényekhez igazítható. Segítségével meghatározható, hogy egy adott kód részlet mihéz is férhet hozzá.

Az AWT (Abstract Window Toolkit) az adott program felhasználói grafikai felület megvalósításában van nagy segítségünkre. Ha már grafika, érdemes megemlíteni a Swing könyvtárat is amely az AWT-hez hasonló, azonban sokkal gazdagabb lehetőségeket kínál a programozó számára. C++ esetén nem volt hasonló eszközzel dolgunk.

## 2. fejezet - Az alapok

A legtöbb számítógépen két karakterkészlet terjedt el az idők során: az ASCII illetve az EBCDIC. Mindkettő 8 bites mérettel rendelkeznek, ezért többek között a magyar nyelv különleges karakterei (pl. ö, ú) nem jeleníthetők meg. Erre nyújt megoldást az Unicode karakterkészlet, ami több, mint 8 biten tárolja karaktereit, ezzel lebontva az előző karakterkészletek korlátait. A pozitívum az, hogy a Java forráskódjában bármilyen tetszőleges Unicode karakterek szerepelhetnek.

Ami az azonosítókat illeti: a Java nyelvben az azonosítóknak betűvel kell kezdődniük és betűvel vagy számmal kell, hogy folytatódjanak. Érdekesség, hogy a betűk nem csak az angol-, hanem akár a hindu karakterkészletből is érkezhetnek. Az azonosítók hossza tetszőleges, azonban vannak olyan szavak, amelyek ilyen célra fel nem használhatók, ugyanis a nyelv kulcsszavai.

A nyelv 8 db. primitív típussal rendelkezik, melyeket előzőleg már ismertettem. A primitív változók magukat az értékeket tárolják, míg a változók másik típusa (referenciatípus) objektumhivatkozásokat tartalmaznak, amely így azt írja le, hogy az adott érték hol lelhető fel a memóriában.

A literálokról annyit érdemes megjegyezni, hogy egyszerű típusok és objektumok inicializálásához használatosak. Többek között használjuk őket speciális karakterek (escape karakterek) megadásához is.

Egy változó deklarációjához szükségünk van legalább egy típusra és egy változónévre. A változókhöz az egyenlőségjel opeátorral rendelhető kezdeti érték ami akár már deklarációkor is megtehető. Nyilván a csak deklárált de értékkel még nem rendelkező változók alapértelmezett értékkel rendelkeznek.

A nyelvben a [] jelöléssel lehet egy tömb típust megadni, ami érdekesség, hogy ez tényleges típus, nem úgy mint C++-ban, ahol ugyanis csak a mutató típus egy megjelenési formája volt. Indexelése 0-val kezdődik.

Beszélhetünk felsorolás típusokról is, melyeket az enum kulcsszóval vezethetünk be, kapcsos zárójelek között a megfelelő értékeket feltüntetve. Az indexelés itt is 0-val kezdődik.

Az operátorok természetesen csakúgy, mint a legtöbb programozási nyelvben, így Java-ban is nagy jelenlősséggel bírnak a kiértékelés szempontjából, ugyanis először a legbelőző zárójelen belüli részkifejezés kerül kiértékelésre, ha nincs zárójel, akkor először a nagyobb prioritású operátor lesz végrehajtva; ha több operátor prioritása egyenlő, akkor balról jobbra, illetve néhány esetben jobbról balra lesznek azok kiértékelve. Beszélhetünk többek között postfix- vagy akár prefix operátorokról is.

A Java mivel egy erősen típusos nyelv, ezért a kifejezésekben szinte minden esetben megvizsgálja, hogy az automatikus konverziókat végrehajtva azonos típusra alakíthatók-e az elemek. Ha az ellenőrzési fordítási időben elvégezhető, akkor hiba esetén a fordítás megszakad. Ha egy konverziós hiba csak futási időben

deríthatő ki, akkor hibaüzenetet a futtató rendszer generál. Létezik ezen kívül viszont explicit konverzió is, ez azonban általában nem biztonságos, adatvesztéssel járhat. Szövegkonverzióról is beszélhetünk.

Egy struktúra részelemének eléréséhez a . (pont)-tal minősített név használható.

### 3. fejezet - A vezérlés

Ha az utasítás szóval kerülünk szembe, alapvetően két fajtáról beszélhetünk. Az egyik ilyen utasításfajta a kifejezés-utasítás még a másik a deklarációs-utasítás. Közös jellemzőjük, hogy minden kettőt pontossá visszavezeti. Kifejezés-utasítás a következő kifejezésfajtból képezhető, például: értékadás, postfix illetve prefix operátorokkal képzett kifejezések, metódushívások vagy példányosítás. A deklaráció-utasítás egy lokális változó létrehozását jelenti. Utasítások sorozatát {} jelek között blokknak nevezzük.

Természetesen az elágazások fellelhetők Java-ban is. Megkülönböztetünk egyszerű- és összetett elágazásokat. Előző alatt az if-szerkezetet, utóbbi alatt pedig a switch-szerkezetet értjük.

Négyféle ciklust ismer a Java nyelv: előtesztelőt, hátultesztelőt, léptetőt és bejáró ciklust. Ezek szinte mindenkorral találkoztunk már C/C++-os tanulmányaink során, talán a bejáró ciklust érdemes viszon megemlíteni.

A bejáró ciklus a Java 5-ös verziójában jelent meg és egy adatszerkezet bejárását végezhetjük vele, mely adatszerkezet lehet tömb, sorozat vagy akár halmaz is.

Egy egészeket tartalmazó tömb bejárása így néz ki a használatával:

```
int [] tomb = {1, 5, 7, 9, 31, 87};
int osszeg = 0;
for(int n: tomb)
 osszeg += n;
```

A Java címkéről csak ennyit: bármely utasítás elő írható címke, mely az utasítás egyértelmű azonosítását teszi lehetővé feltétlen vezérlésátadás érdekében. Formai kinézete: címke: utasítás.

A most következő pár sor sem lesz ismeretlen számunkra. A break utasítás egy blokkból való kilépésre szolgál. Formája: break [címke].

A continue utasítással egy ciklus magjának hátralevő részét lehet átugrani. Kinézete: continue [címke];

Egy metódussal a return utasítással lehet visszatérni. A metódus visszatérési értéke így a return után írt kifejezés értéke lesz.

És most essen pár szó a goto-ról! Többek között a C++-szal ellentétben a Java-ban már nincsen goto utasítás. Helyette alternatív megoldásokat kínál a Java.

### 4. fejezet - Osztályok

A Java-ban írt programok legkisebb önálló egységei az osztályok. Egy osztály olyan, mint egy tervrajz, amely azonos típusú "dolgok" modelljét írja le. A program a működése során példányosítja az osztályokat vagyis konkrét példányokat hoz létre a modellek séma szerint. Az osztályokat két részből írják le, ez az osztálydefiníció. Az egyik rész deklarálja a változókat, míg a másik az objektum viselkedését meghatározó metódusokat tartalmazza.

Az osztály változóit változódeklarációk adják meg. Egy deklaráció egy vagy több azonos típusú változót vezet be. A deklaráció a változó típusával kezdődik majd mögötte vesszővel elválasztva változó nevek állnak. Kezdőérték megadása is lehetséges.

Egy osztály metódusait metódusdefiníciók írják le. Egy definíció fejből és törzsből áll. A fej a metódus visszatérési típusát, azonosítóját és formális paramétereit tartalmazza. Első 2 alkotja a metódus szignatúráját. Abban az esetben, ha a visszatérési típus `void` típusú, azt jelenti, a metódus nem ad vissza értéket, csupán végrehajt. Ez C++ esetén is ugyanígy volt. A metódus törzsében a működést definiáló utasításblokk található. A törzsben használható a `this` ún. pszeudováltozó is, amivel az aktuális példányra hivatkozhatunk.

A metódushívásban meg kell adni a metódus definíciójában előírt számú és típusú paramétert. Fontos megemlíteni, hogy a nem osztálymetódusok esetében egy metódus kizárolag egy konkrét példányra hívható meg.

Túlterhelt metódusnév esetén (több metódusnak is ugyanaz a neve) az aktuális paraméterek száma és típusa alapján dönti el a Java fordító, melyik metódust szerettük volna meghívni.

A Java egy egész intelligens nyelv. Például hibaüzenetet kapunk abban az esetben, amikor a metódus törzsében deklarált lokális változók nem kerülnek inicializálásra és a program így akarja kiolvasni azokat. Ezt elkerülve van benne a nyelvben egyfajta ellenőrző-mechanizmus, vagyis a fordító elemzi a metódustörzseket, hogy az egyes változók olvasása előtt megtörtént-e az értékadása.

Egy objektum a példányosítás során születik meg, melyet a `new` operátorral hajtunk végre. A `new` operátor mögött azt adjuk meg, melyik osztályt példányosítjuk. Ekkor az operátor memóriát foglal le, ami az objektum változóit fogja tárolni és visszaadja a lefoglalt terület kezdőcímét. Ez dinamikus módon történik, a C++-os megoldással ellentétben. A Java megkülönbözteti magát az objektumot és a referenciát ami mintegy "odamutat" az objektumra. A referencia abban különbözik a C++-os tanulmányaink során már megismert pointerektől (mutatóktól), hogy velük ellentétben a mutatott objektumot jelentik, nem pedig a címet. A `null` egy speciális referenciaérték, ez értékül adható minden referencia típusú változónak, olyan referencia, ami nem mutat egyetlen objektumra sem. Ha egy objektumra továbbá nincs szükségünk, érdemes tőle megszabadulni. A C++ nyelvvel ellentétben Java-ban ez nem a programozó feladata. Ha egy objektummra már nincs több hivatkozás, biztosan nincs rá szükség, ezér törlésre kerül. Ezt a `garbage collector`, az ún. szemetgyűjtő mechanizmus végzi el.

A hozzáférési kategóriák téma köre sem elhanyagolható. Ezek szabályozása a `public`, `private` és a `protected` módosítókkal megvalósítható. Ez számunkra nem újdonság, találkozhattunk már ezzel C++ esetén is.

Egy Java program az elindítás szempontjából egy osztálytalálkozásból indító szerepet töltön be, rendelkeznie kell egy `main` nevű osztálymetódussal. A program indításakor a felhasználónak meg kell adnia az indítandó osztály nevét, a megnevezett osztályt a Java Virtuális gépe inicializálja majd megkezdi a `main` metódus végrehajtását. Egy példán keresztül szemlélteti a szerző.

A konstruktur olyan programkód aminek végrehajtása a példányosításkor automatikusan megtörténik. Ne vüknek meg kell egyezniük az osztály nevével. Közvetlenül azonban nem meghívhatók, csupán példányosításon keresztül. Visszatérési típusuk nincsen, még csak `void` sem.

A `new` példányosító operátor számára paramétereket is meg lehet adni. Ezek azok az aktuális paraméterek amiket a konstruktornak szán.

Blokkokról az előzőekben már volt szó, azonban az inicializáló blokkokról még nem esett szó. Ez egy olyan utasításblokk, amelyet az osztálydefiníciót belül helyezünk el a változódeklarációk és a metódusdefiníciók között. Vannak osztály- és példányinicializátorok is. Az osztályinicializátor előtt a `static` kulcsszó áll.

A Java szemetgyűjtő mechanizmusa is meg lett említve már az előbb. Azonban vannak olyan szituációk, amikor szeretnénk, ha tájékoztatva lennének arról, ha az objektum megsemmisült. Ezt egy `finalize` nevű metódussal kiegészítve az osztályt tehetjük meg.

Az objektumorientált programozás lényege: az adatabsztraktiós, az öröklődés és a polimorfizmus. Ezek mindegyikével már találkoztunk C++-os tanulmányaink során is.

## 5. fejezet - Interfészek

Az interfészok az osztályok mellett a Java nyelv egyik legfontosabb eleme. Az interfész egy újfajta referencia típus. Egyik alapvető tulajdonsága, hogy a benne metódusok csak deklarálódnak, vagyis megvalósítás nélkül szerepelnek. A nevéből is adódik, hogy egy felületet definiál. Használata implementációján keresztül történik. Egy osztály implementál egy interfészt, ha az összes, az interfész által specifikált metódushoz implementációt ad. Köztük is létezik öröklődés, vagyis szülő-gyermek kapcsolat, melynek neve kiterjesztés. Egy interfész akkor közvetlen kiterjesztése egy másik interfésznek, ha azokat deklarációjukor explicit módon kiterjeszti. A kiterjesztést az extends kulcsszóval kell jelezni.

Ha a nyelv nem tud igényeinknek megfelelő interfészt kínálni, akkor nekünk kell cselekednünk és létrehoznunk egyet. Ezt nevezzük interfészdeklarációnak. A folyamat a következőket tartalmazza: opcionális interfézmódosítók (lehet abstract vagy public), az interface kulcsszó, az interfész neve, interfész által kiterjesztett interfészek és interfésztörzs.

Az interfész egyébként egy új referenciátípust vezet be, így minden olyan helyen használható, ahol egy osztály is.

Az interfész törzse egy kapcsos zárójelek közti felsorolás. A felsorolás elemei csak konstansdefiníciók és absztrakt metódusok deklarációi lehetnek. Az interfész az így deklarált tagokból és az interfész által kiterjesztett interfészek tagjaiból tevődik össze.

A konstansdefiníciók az interfészek konstansait vezetik be. A definíció szintaktikája a következő: módosítók, konstans típusa, konstans egyedi azonosítója, incializáló kifejezés és pontosvessző. A konstansok deklarációjához egyébként módosítókat lehet használni.

Az interfész absztrakt metódusainak deklarációja nagyon hasonlít az osztályok metódusainak deklarációjához, az eltérés csupán a metódus törzsének elhagyása. Erre példát a könyv 62. oldala hoz.

Az interfészek használata az azokat megvalósító osztályokkal történik. Egy osztály csak akkor tud implementálni egy interfészt, ha minden implementáló metódus szignatúrája és visszatérési típusa megegyezik az interfézbeli absztrakt metódus szignatúrájával és visszatérési típusával. Az interfészek implementálását az implements kulcsszóval lehet deklární. A kulcsszó után kell felsorolni az implementált interfészeket, vesszővel elválasztva. Erre hoz példát a könyv:

```
interface Gyumolcs{
 void egyedMeg (int i);
}

interface Alma extends Gyumolcs{
 int szine();
 void szinAllit(int i);
}

interface Korte{
 int merete();
 void meretAllit (int i);
}

class Jonatan implements Alma{
 int szin;
 public void szinAllit (int i){szin = i; }
 public int szine() {return szin;}
 public void egyedMeg(int i) { System.out.println("Finom volt"); }
```

```
}

class VilmosKorte implements Korte{
 int meret;
 public void meretAllit(int i) { meret = i; }
 public int merete() { return meret; }
}
```

## 20.3. Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven

### 2.1 A main függvény

Két formája létezik C++-ban, mindenkorre példát látunk a 4. oldalon.

#### 2.1.3 A bool típus

A C++ nyelvben bezetésre került a bool típus is, ami logikai értéket hivatott tárolni, így búcsút lehet inteni az eddigi, intben tárolt logikai igaznak/hamisnak.

A C++-ban alapból, beépített típusként kaptak helyet a C-ben megismert, több-bájtos sztringek reprezentálását megvalósító wchar\_t típus. Szó esik még a változók deklarálásáról, azok hatóköréről is.

#### 2.2 Függvények túlterhelése

A C nyelvben egy függvényt egyértelműen a neve azonosítja. C++-ban ez másképp van, a név és az argumentumlista együttesen azonosít egy függvényt, ellenben a visszatérési értékkel, amivel egyértelműen nem azonosítható egy függvény. Jellemzésre kerül a linker névelferdítési technikája melyet azonos nevű függvények esetén alkalmaz. Szó esik az extern kulcsszó használatáról is, amit akkor kell használnunk ha egy C++ függvényt akarunk C-ben meghívni.

#### 2.3 Alapértelmezett függvényargumentumok

A C++ nyelv lehetőséget ad ehhez is, függvényeinknek meg lehet adni alapértelmezett értékeket is, így a hívás sokkal egyszerűbb is lehet. Példákon szemlélteti a könyv a 9. oldalon.

#### 2.4 Paraméterátadás referenciatípussal

C nyelvben csakis érték szerinti paraméterátadás lehetséges, ez a C++ nyelv esetén nem így van, megvalósítható a referenciával való paraméterátadás is. Ehhez csupán annyit kell tennünk, hogy a paraméterlistában mutatóra írjuk át az eddigi változót. Ez a cím szerinti paraméterátadás. Így működik C esetében. De a C++ ilyen téren (is) megkönnyíti a dolgunkat hiszen bevezette a referenciatípust, melyet a változó elé tett & jellet tudunk létrehozni. Ezt részletezi a könyv a 12.-16. oldalakon, kódokkal szemléltetve.

### Objektumok és osztályok

#### 3.1 Az objektumorientáltság alapelvai

A programok összetettségének növekedése miatt sokszor már lehetetlen volt a programkódokat kézben tartani. Megnövekedett az igény az átláthatóság iránt, emiatt terjedt el a '90-es években az objektumorientált programozás.

Fontos fogalom az egységbe zárás (enkapsuláció), az egységbe záró adatstruktúra neve osztály. Az osztálynak lehetnek "egyedei", ezek az objektumok. Az objektumok "védekező mechanizmusa" az adatrejtés,

ekkor az objektum a program többi része számára nem teszi elérhetővé tartalmát. Megjelenik továbbá az öröklődés és az egységebe zárás fogalma is. Ezek az objektumorientált programozás alapelvei.

Fontos látni, hogy az éppen aktuális feladat dönti el minden, érdemes-e az OOP-t (Objektumorientált Programozás) használni.

### 3.2 Egységebe zárás a C++-ban

Átveszi a tagváltozók és tagfüggvények szerepét, használatát. A tagfüggvények esetében a függvény rendelkezik egy láthatatlan, első paraméterrel, ami alapján tudni fogja, melyik struktúrát kell módosítania. A `this` kulcsszó szerepe.

### 3.3 Adatrejtés

A `private` kulcsszó és szerepe: az utána álló változók és függvények csak osztályon belül láthatóak. Ellentéte a `public`, ami azt jelenti, hogy az adott tag struktúrán kívül is látható

A változó létrehozást osztályból példányosításnak hívjuk, a példány más néven objektum.

### 3.4 Konstruktorok és destruktorkor

A konstruktor speciális tagfüggvény, példányosításkor hívódik meg. A destruktur az objektumok által esetlegesen lefoglalt erőforrások felszabadításáért felel, automatikusan meghívódik mikor az objektum megsűnik.

### 3.5 Dinamikus adattagot tartalmazó osztályok

A dinamikus memóriakezelésért felelős operátor a `new`, a lefoglalt helyet a `delete` operátorral szabadíthatjuk fel. Tömbök esetén a `new []` és a `delete []` használhatók.

#### 3.5.3 Másoló konstruktor

Ismérve, hogy egy referenciát vár, aminek típusa megegyezik az osztály típusával. Esetében az újonnan létrehozott objektumot egy már létező objektum alapján inicializáljuk. Megjelenik a sekély- (shallow) és a mély (deep) másolás fogalma, ezek közti különbségek tisztázása.

### 3.6 Friend függvények és osztályok

#### 3.6.1 Friend függvények

A `friend` kulcsszó feljogosíthatnak globális függvényeket, illetve más osztályok függvényeit arra, hogy hozzáférhessék az osztály védett tagjaihoz.

#### 3.6.1 Friend osztályok

Koncepciójukban már-már megegyeznek a `friend` függvényekkel. Az osztályunk egy másik osztályt jogosít fel a védett tagjaihoz való hozzáférésre. A friend osztály tagfüggvényei olyan hozzáférési jogokkal rendelkeznek mintha alapból az adott osztály tagfüggvényei lennének. A friend tulajdonságról azonban tudni kell, hogy: nem öröklődik és nem tranzitív.

## **IV. rész**

### **Irodalomjegyzék**

## 20.4. Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

## 20.5. C

[KERNIGHANRITCHIE] Kernighan, Brian W. & Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

## 20.6. C++

[BMECPP] Benedek, Zoltán & Levendovszky, Tihamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

## 20.7. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, [http://arxiv.org/PS\\_cache/math/pdf/0404/0404335v7.pdf](http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf) , 2004.

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPORG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEAHCackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésekért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPORG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségen született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.