

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Ádám Petra

Copyright (C) 2019, Ádám Petra,

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 Ádám, Petra	2019. november 27.	

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
0.0.5	2019-05-09	A könyv és a feladatok befejezve, fejezetek időben leadva, a leadás gyakorlatokon ellenőrizve/vezetve.	Á. Petra

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. Helló, Google!	15
2.7. 100 éves a Brun téTEL	17
2.8. A Monty Hall probléma	18
3. Helló, Chomsky!	20
3.1. Decimálisból unárisba átváltó Turing gép	20
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	21
3.3. Hivatkozási nyelv	22
3.4. Saját lexikális elemző	22
3.5. l33t.l	23
3.6. A források olvasása	26
3.7. Logikus	31
3.8. Deklaráció	32

4. Helló, Caesar!	35
4.1. double ** háromszögmátrix	35
4.2. C EXOR titkosító	38
4.3. Java EXOR titkosító	39
4.4. C EXOR törő	40
4.5. Neurális OR, AND és EXOR kapu	43
4.6. Hiba-visszaterjesztéses perceptron	45
5. Helló, Mandelbrot!	53
5.1. A Mandelbrot halmaz	53
5.2. A Mandelbrot halmaz a std::complex osztállyal	56
5.3. Biomorfok	58
5.4. A Mandelbrot halmaz CUDA megvalósítása	61
5.5. Mandelbrot nagyító és utazó C++ nyelven	62
5.6. Mandelbrot nagyító és utazó Java nyelven	65
6. Helló, Welch!	69
6.1. Első osztályom	69
6.2. LZW	71
6.3. Fabejárás	74
6.4. Tag a gyökér	77
6.5. Mutató a gyökér	85
6.6. Mozgató szemantika	94
7. Helló, Conway!	96
7.1. Hangyszimulációk	96
7.2. Java életjáték	97
7.3. Qt C++ életjáték	97
7.4. BrainB Benchmark	98
8. Helló, Schwarzenegger!	100
8.1. Szoftmax Py MNIST	100
8.2. Mély MNIST	100
8.3. Minecraft-MALMÖ	101

9. Helló, Chaitin!	102
9.1. Iteratív és rekurzív faktoriális Lisp-ben	102
9.2. Gimp Scheme Script-fu: króm effekt	102
9.3. Gimp Scheme Script-fu: név mandala	103
9.4. SMNIST passzolás feladat megoldás hozzáfűzés	103
10. Helló, Gutenberg!	105
10.1. Programozási alapfogalmak	105
10.2. Programozás bevezetés	109
10.3. Programozás	110
III. Második felvonás	115
11. Helló, Berners-Lee!	117
11.1. C++	117
11.2. Java és C++	118
11.3. Python	122
12. Helló, Arroway!	124
12.1. OO szemlélet	124
12.2. Homokozó	126
12.3. „Gagyí”	128
12.4. Yoda	129
12.5. Kódolás from scratch	131
13. Helló, Liskov!	134
13.1. Liskov helyettesítés sértése	134
13.2. Szülő-gyerek	136
13.3. Anti OO	137
13.4. Hello Android!	140
14. Helló, Mandelbrot!	143
14.1. Reverse engineering UML osztálydiagram	143
14.2. Forward engineering UML osztálydiagram	145
14.3. Egy esettan!	147
14.4. BPMN	149

15. Helló, Chomsky!	151
15.1. Encoding	151
15.2. 1334d1c4	152
15.3. Paszigráfia Rapszódia OpenGL full screen vizualizáció	153
15.4. Perceptron osztály	154
16. Helló, Stroustrup!	156
16.1. JDK osztályok	156
16.2. Változó argumentumszámú ctor és Összefoglaló összevonva	157
17. Helló, Gödel!	160
17.1. STL map érték szerinti rendezése	160
17.2. Alternatív Tabella rendezése	162
17.3. GIMP Scheme hack	164
18. Helló, Valami!	166
18.1. FUTURE tevékenység editor	166
18.2. SamuCam	167
18.3. BrainB	168
19. Helló, Lauda!	170
19.1. Port scan	170
19.2. AOP	172
19.3. Junit teszt	173
20. Helló, Calvin!	175
20.1. MNIST	175
20.2. Android telefonra a TF objektum detektálója	177
20.3. Minecraft MALMO-s példa	178
IV. Irodalomjegyzék	180
20.4. Általános	181
20.5. C	181
20.6. C++	181
20.7. Lisp	181

Táblázatok jegyzéke

13.1. Eredmények	138
----------------------------	-----

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:

Tanulságok, tapasztalatok, magyarázat...

Végtelen ciklusok írására számtalan lehetőség van. Nagyon egyszerűen el lehet készíteni, egy alapvetőt kell követni hozzá. While, illetve for ciklussal is előállítható, ezt kedv szerint dönti el a programozó. A lényeg, hogy olyan feltételt adjunk meg, ami minden teljesül, így a futás sem fog megállni. Ahhoz, hogy 0%-ban használjuk egy végtelen ciklusban a processzormagokat, sleep függvényt, valamint a #pragma omp parallel párhuzamos programozási eszközökkel kell használnunk. Az előbbi a magot nem használja, megállítja a ciklust, míg az utóbbi a több magért felelős. Futtatáskor a jó parancshasználat lényeges, mind a 0%, mind a 100%-os használatnál, a -fopenmp kapcsolót kell beírnunk. A 100%-os maghasználatnál szintén a feltétel folyamatos teljesülése szükséges. Az összes maghasználathoz ugyan az a párhuzamos programozási eszköz szükséges, mint a 0%-osnál, és a futtatásához -fopenmp kapcsoló szükséges. Ahol csak 1 magos futás kell, ott a fordítás: gcc vegtelen.c -o vegtelen és a futtatás: ./vegtelen .Tehát a példák pici programokkal:

Végtelen ciklus 1 magon:

```
#include <stdio.h>

int main() {
    while(1) {}
    return 0;
}
```

Végtelen ciklus minden magon:

```
#include <stdio.h>

int main() {
    #pragma omp parallel
```

```
while(1) {}  
return 0;  
  
//futtatas az osszes maghoz: gcc vegtelenmagok.c -o vegtelenmagok -fopenmp
```

Üres (0%) végtelen ciklus 1 magon:

```
#include <stdio.h>  
#include <unistd.h>  
  
int main(){  
while(1){  
    sleep(800);}  
return 0;  
}
```

Üres (0%) végtelen ciklus minden magon:

```
#include <stdio.h>  
#include <unistd.h>  
  
int main(){  
#pragma omp parallel  
while(1){  
    sleep(800);}  
return 0;  
}  
  
//futtatas az osszes maghoz: gcc vegtelenures.c -o vegtelenures -fopenmp
```

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 dönteneni, 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 for 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...

Ezen a programon már nem lehet túl sokat magyarázni, mert tanár úr részletesen leírta. A lényege az, hogy nem lehet ilyen programot írni, mert ez is lefagyna. Valamint ellentmondásba is ütközünk, tehát működésképtelen. A rekurzív hivatkozás miatt ha lefagy, akkor azt írja ki, hogy nem fagy, illetve ha nem fagy le a program, akkor pedig azt fogja kiírni, hogy lefagy. Szóval mindenképp rossz eredményt fogunk kapni, erre gondolok ellentmondás alatt.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Több megoldási lehetőség van, lehetne például függvényt is használni, de a feladat ezt a lehetőségeket kizára. Ezen kívül akár segédváltozó használatával teljesíthetjük a feladatot. Az eredetileg is deklarált 2 változónk mellé bevezetünk egy harmadik segédváltozót. A csere során ideiglenesen ebbe a segédbe töltjük be az egyik értéket, és 3 egyenlőséggel végrehajtjuk a cserét, majd kiíratjuk újra az értékeket. A fordításhoz és futtatáshoz nincs szükség semmi pluszra, normál módon gcc-vel lehet. Más megoldási lehetőségek például az összeadás, kivonás, szorzás, osztás művelet, ezeknek a segítségével is felcserélhetjük az értékeket, és így segédváltozóra sincs szükségünk. Fordítás: gcc valtozo.c -o valtozo és a futtatás: ./valtozo . Ezek ugyan úgy működnek, csak a műveleti jelekkel kell felcserélnünk. 2 példára kódcsipet:

```
#include <stdio.h>

int main(){
    int valtozo_1 = 2, valtozo_2 = 4;
printf("valtozo_1=%d valtozo_2=%d\n",valtozo_1, valtozo_2);
    valtozo_1 = ( valtozo_1 - valtozo_2 );
    valtozo_2 = ( valtozo_1 + valtozo_2 );
    valtozo_1 = ( valtozo_2 - valtozo_1 );
printf("valtozo_1=%d valtozo_2=%d\n",valtozo_1, valtozo_2);
return 0;
}
```

```
#include <stdio.h>

int main(){
    int valtozo_1 = 2, valtozo_2 = 4, c;
```

```
printf("Az eredeti valtozok: valtozo_1=%d valtozo_2=%d\n",valtozo_1, ←
    valtozo_2);
    c=valtozo_1;
    valtozo_1=valtozo_2;
    valtozo_2=c;
printf("Az eredmény: valtozo_1=%d valtozo_2=%d\n",valtozo_1, valtozo_2);
return 0;
}
```

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 videókon.)

Tutor (om volt): Tóth Attila

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

Megoldás forrása (kód): <https://bhaxor.blog.hu/2018/08/28/labdapattogas>

Tanulságok, tapasztalatok, magyarázat...

A labdapattogtatás során egy megadott karaktert, azaz a labdát mozgatjuk fokozatosan a pályályán. A léptetés egyszerűen működik, minden előre meghatározott lépésnél a labda előre meghatározott irányban halad. Az if-utasítás segítségével megadhatjuk a labda "pattogását", azaz az irányváltásait. Azt vizsgálhatjuk meg velők, hogy elérte-e a labda a pálya valamelyik szélét, tehát a bal, jobb, alsó vagy felső szélét. Az initscr függvény gondoskodik arról, hogy futtatáskor a program egy új (terminál)ablakban nyíljon meg. A for(;;) egy végtelen ciklus, így a program egészen addig fog futni, amíg a felhasználó meg nem szakítja azt egy parancssal. A usleep(nagy szám) függvény a késleltetésért felelős, így emberi szemmel is könnyen követhető. Nélküle a gépek gyors működése miatt szinte egybefolynak. If utasítások nélkül a konkrét pálya széleket is meg kell határozni, illetve egy-egy plusz deklarációt létrehozni, -1 értéket adva nekik. Ezekre akkor van szükségünk, amikor elérjük a pálya valamelyik szélét, és ha ezzel beszorzunk, akkor irányt vált a labda. Érdemes a labda kezdeti pozíciójának nem a 0;0 pontot megadni, mert így nem saroktól sarokig fog pattogni, ha jól határoztuk meg az értékeket. A usleep függvény itt is szükséges a lassításhoz, illetve használhatjuk még a system("clear") függvényt is, aminek a segítségével mindenkor csak a labda aktuális helyét fogjuk látni. Fordítás: gcc labda.c -o labda és a futtatás: ./labda

Labdapattogás if-fel:

```
#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;

for ( ; ; ) {

    getmaxyx ( ablak, my , mx );
    mvprintw ( y, x, "O" );

    refresh ();
    usleep ( 100000 );

    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;
    }

}

return 0;
}
```



Labdapattogás if nélkül:

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

static void gotoxy(int x, int y) //kurzor pozicionálása
{
    int i;
    for(i=0; i<y; i++) printf("\n"); //lefelé tolás
    for(i=0; i<x; i++) printf(" "); //jobbra tolás
    printf("o\n"); //labda ikonja
}

int main(){
int egyx=1;
int egyy=-1;
int i;
int x=10; //a labda kezdeti pozíciója
int y=20;
int ty[23]; //magasság // a pálya mérete
int tx[80]; //szélesség
```

```
//pálya széleinek meghatározás

for(i=0; i<23; i++)
    ty[i]=1;

ty[1]=-1;
ty[22]=-1;
for(i=0; i<79; i++)
    tx[i]=1;

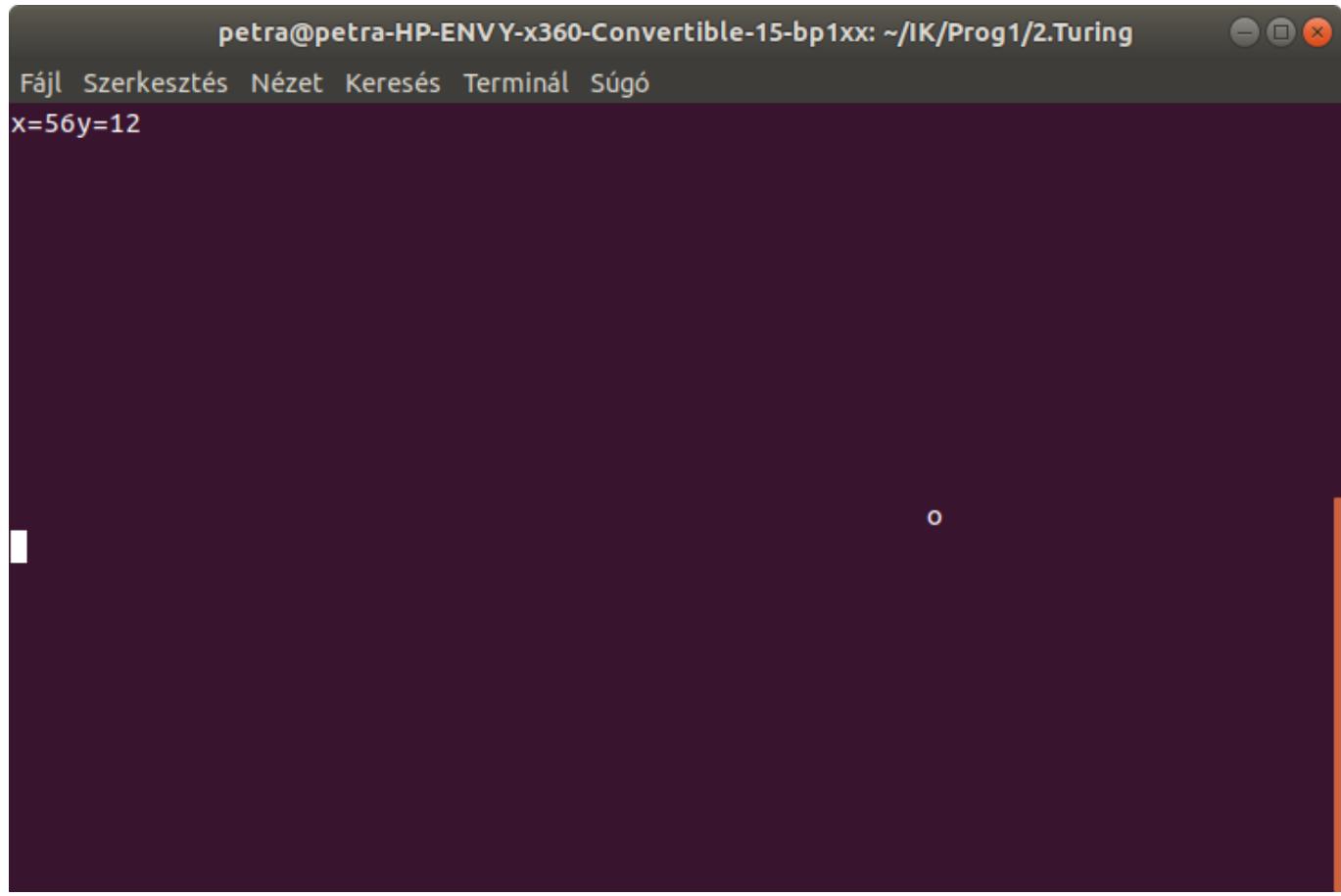
tx[1]=-1;
tx[79]=-1;

for(;;)
{
    //címsor és pozíció kijelzése
    printf("x=%2d", x);
    printf("y=%2d", y);

    (void)gotoxy(x,y);
    //printf("o\n"); Áthelyezve a gotoxy függvényre

    x+=egyx;
    y+=egyy;

    egyx*=tx[x];
    egyy*=ty[y];
    usleep (200000);
    system("clear");
}
```



A screenshot of a terminal window titled "petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog1/2.Turing". The window has standard Linux-style window controls (minimize, maximize, close) in the top right corner. The menu bar at the top includes "Fájl", "Szerkesztés", "Nézet", "Keresés", "Terminál", and "Súgó". Below the menu bar, the command "x=56y=12" is typed into the terminal. The terminal window has a dark background and light-colored text.

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: UDPORG csoport

Tanulságok, tapasztalatok, magyarázat...

A szóhossz vizsgálat bitenkénti léptetéssel történik, amennyit léptet, annyi lesz a szóhossz, az én esetben ez 32 bit. A BogoMIPS hasonló működési elvvel rendelkezik, mint a szóhossz. Lényege, hogy a processzoridőt felhasználva hajtja végre a bitenkénti léptetéseket, azaz bizonyos időn belül hány műveletet képes elvégezni. Nem a normál óra szerint mér, hanem a saját processzoridejét használja, hiszen az időt is ez alapján állapítja meg. A bitenkénti léptetés során a műveletek száma hatványozódik, azaz x^2 , minden 2 hatványa lesz az eredmény. Az órajel segítségével számol a BogoMips. A programon belül késleltetést is használ, de a fő funkciója az időmérés. A kommentek tartalmazzák a működés főbb lépéseihez leírását.

Fordítás: gcc bogomips.c -o bogomips és a futtatás: ./bogomips

A szóhossz a gépen:

```
#include <stdio.h>

int main(void) {
```

```
int h = 0;
int n = 0x01; //8 biten tarol
do
++h;
while(n<<=1);
printf("A szohossz ezen a gepen: %d bites\n", h);
return 0;
}

//bitenkenti leptetes 0 es 1 kozott
```

A BogoMips:

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

void
delay (unsigned long long int loops) //kesleltetés, parametertől függ
{
unsigned long long int i;
for (i = 0; i < loops; i++);

}

int
main (void)
{
unsigned long long int loops_per_sec = 1;
unsigned long long int ticks;

printf ("Calibrating delay loop..");
fflush (stdout); //kimenet kezelése

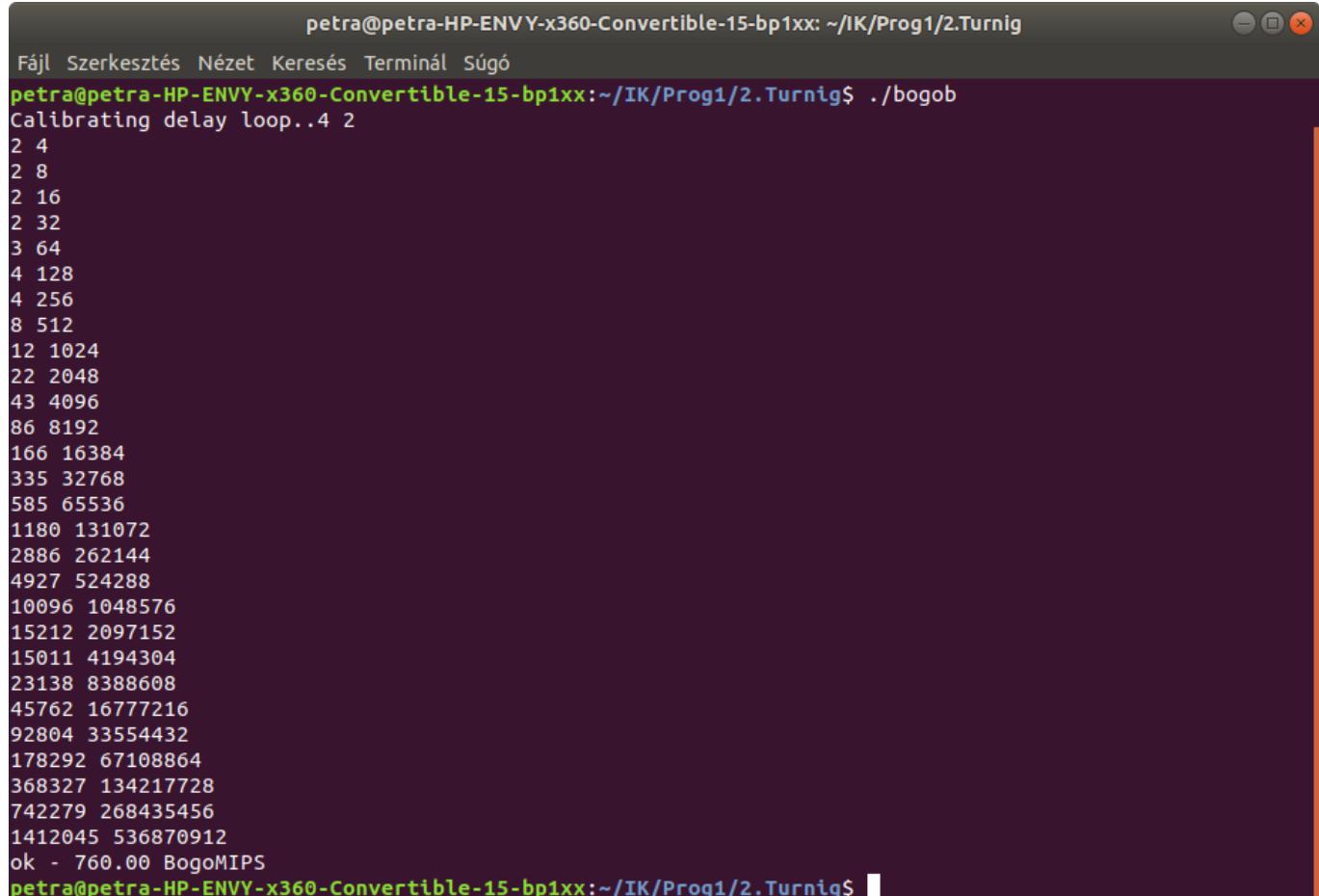
while ((loops_per_sec <<= 1)) //bitenkenti leptetes, leptetjük a ←
    kesleltetest eggyel balra
{
ticks = clock (); //orajel kivetele, a clock a program indítása óta eltelt ←
    processzoridovel ter vissza
delay (loops_per_sec);
ticks = clock () - ticks; //megegy orajel kivetel, ujra leolvassa az ←
    orajelet és kivonja belőle a kezdetet

printf ("%llu %llu\n", ticks, loops_per_sec); //aktuális orajel, llu = ←
    unsigned long long int

if (ticks >= CLOCKS_PER_SEC) //ha a vegjel nagyobb mint egy mp // ←
    CLOCKS_PER_SEC =az egy mp alatti orajelek szamaval
{
loops_per_sec = (loops_per_sec / ticks) * CLOCKS_PER_SEC; //a kesleltetest ←
    mennyi ideig tartja fent

printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec / 500000,
```

```
(loops_per_sec / 5000) % 100);  
  
return 0;  
}  
}  
  
printf ("failed\n");  
return -1;  
}
```



```
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/2.Turnig$ ./bogob  
Calibrating delay loop..4 2  
2 4  
2 8  
2 16  
2 32  
3 64  
4 128  
4 256  
8 512  
12 1024  
22 2048  
43 4096  
86 8192  
166 16384  
335 32768  
585 65536  
1180 131072  
2886 262144  
4927 524288  
10096 1048576  
15212 2097152  
15011 4194304  
23138 8388608  
45762 16777216  
92804 33554432  
178292 67108864  
368327 134217728  
742279 268435456  
1412045 536870912  
ok - 760.00 BogoMIPS  
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/2.Turnig$ █
```

2.6. Helló, Google!

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

Tutor (om volt): Tóth Attila

Megoldás videó:

Megoldás forrása: BHAX csatorna

Tanulságok, tapasztalatok, magyarázat...

A PageRankot eredetileg a Google találta ki, ez amit itt bemutat a kód, az egy egyszerűsített változat. A PageRank értékek célja az, hogy a különböző weboldalakat rangsorolja, egyfajta "fontossági" sorrend alapján.

Minél több oldal mutat egy másik oldalra, annál fontosabb lesz az adott oldal. Az alapelve a vektorszorzás. Az adott 4x4-es mátrix a 4 megadott weboldalra utal. "Szavazatokat" osztanak egymás között ezek alapján kapják meg a rangjukat, tehát egy oldal hány másikra utal. Ehhez szükséges a vektorizálás, vagyis a mátrix, mert az határozza meg, hogy ki kire ad le szavazatot. A műveletek végrehajtásához egy végleges és egy ideiglenes tömböt használhatunk. Az if-ben a 0.00001 érték csillapításként működik. Elindul egy for(;;) végzetlen ciklus, amit az if fog megtörni a break parancsal, amint teljesül az if-ben leírt feltétel. A fordítás: gcc pagerank.c -o pagerank és a futtatás: ./ pagerank

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

void
kiir (double tomb[], int db)
{
int i;
for (i=0; i<db; i++)
printf("PageRank [%d]: %lf\n", i, tomb[i]);
}

double tavolsag(double pagerank[], double pagerank_temp[], int db)
{
double tav = 0.0;
int i;
for(i=0;i<db;i++)
tav += fabs(pagerank[i] - pagerank_temp[i]);
return tav;
}

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 PR[4] = {0.0, 0.0, 0.0, 0.0}; //végleges tomb
double PRv[4] = {1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0}; //ideiglenes ←
//tomb

long int i, j, h;
i=0; j=0; h=5;

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

```
double temp=0;
for (j=0; j<4; j++)
temp+=L[i][j]*PR[j];
PRv[i]=temp;
}

if ( tavolsag(PR,PRv, 4) < 0.00001)
break;
}
kiir (PR,4);
return 0;

}
```

2.7. 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

Az R nyelv felépítése és szerkezete olyan mint a Matlabé, ez alapján megpróbálhatjuk bizonyítani a téTELt, Brun téTELét az ikerprímekről. Eszerint ha valamit felosztunk prím számú részekre, majd összeadjuk ezeket, akkor nem a végtelenségig fog nőni az érték, hanem a Brun által meghatározott konstanshoz tart. A Brun konstans a 2. A kérdés pedig amit a téTEL szeretne megválaszolni az az, hogy az ikerprímszámok száma véges sok, vagy végtelen?

```
library(matlab)

stp <- function(x) {

  primes = primes(x)
  diff = primes[2:length(primes)]-primes[1:length(primes)-1]
  idx = which(diff==2)
  t1primes = primes[idx]
  t2primes = primes[idx]+2
  rt1plust2 = 1/t1primes+1/t2primes
  return(sum(rt1plust2))
}

x=seq(13, 1000000, by=10000)
y=sapply(x, FUN = stp)
plot(x,y,type="b")
```

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

```
kiserletek_szama=10000000
kiserlet = sample(1:3, kiserletek_szama, replace=T)
jatekos = sample(1:3, kiserletek_szama, replace=T)
musorvezeto=vector(length = 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)]}

}

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(holvalt),1)]}

}

valtoztatesnyer = which(kiserlet==valtoztat)

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

Tanulságok, tapasztalatok, magyarázat...

A Monty Hall probléma egy valószínűségszámítási tételes. Aki megnézi a 21 Las Vegas ostroma című filmet, megértheti a problémát, ugyanis a választási esélyeket firtatja a tételes, hogy mennyi valószínűsséggel választunk jól, vagy rosszul. Az eredeti kérdés egy amerikai vetélkedő nyomán indult el. A józan észről szól. Ha nem gondolunk jobban utána, akkor úgy tűnik a legtöbb embernek, hogy nincs különbség aközött, hogy megváltoztatjuk e a döntésünket, vagy nem. Pedig valójában nagy különbség van, érdemes megváltoztatni a döntésünket a matematikai valószínűség szerint. Eredetileg minden választásnál $\frac{1}{3}$ -ad az esély, hogy azt választjuk, amelyikkal nyerünk. Viszont ha kiveszünk egyet a rosszakból, és előtte már választottunk, akkor a 3.-nak amelyet nem változtottunk és rossz, akkor a megmaradtnak $\frac{2}{3}$ -ad lesz az esélye, hogy jó, még az általunk választottnak továbbra is csak $\frac{1}{3}$ -ad.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

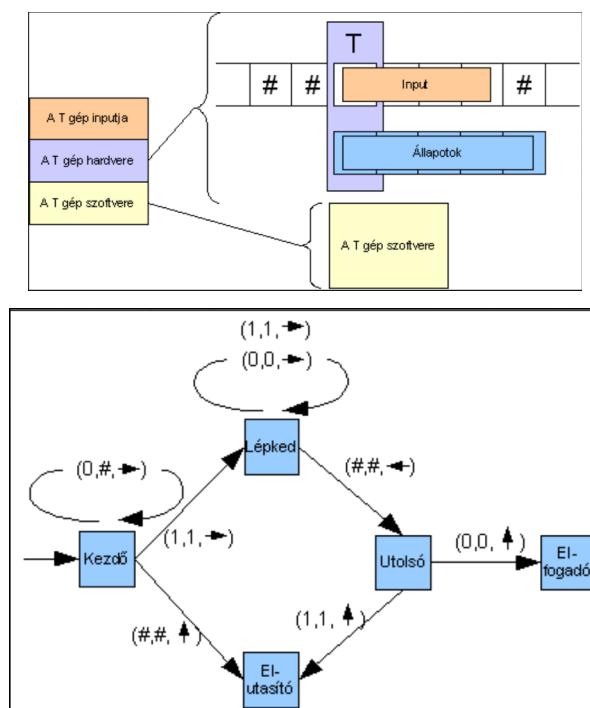
Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Tutoriált (am volt): Egyed Anna

Megoldás videó:

Megoldás forrása: előadás fóliák

Tanulságok, tapasztalatok, magyarázat...



A Turing gép Alan Turing nevéhez fűződik. Ez egyfajta absztrakt automata amellyel a digitális számítógépeket jeleníti meg egyszerűsített formában. Az átváltás folyamata nem túl bonyolult. Egész és pozitív számok ábrázolására vagyunk képesek így. A menete: 2 db 0 között annyi egyest írunk, hogy azoknak az összege megyegyezzen az általunk kigondolt számmal.

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

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

Tutor (om volt): Egyed Anna

Megoldás videó:

Megoldás forrása: előadás fóliák

Tanulságok, tapasztalatok, magyarázat...

A grammatika forrása az előadás fólia.

A, B, C legyenek változók

a, b, c legyenek konstansok

$A \rightarrow aAB, A \rightarrow aC, CB \rightarrow bCc, cB \rightarrow Bc, C \rightarrow bc$

A ($A \rightarrow aAB$)

aaAB ($A \rightarrow aAB$)

aaaaCB ($CB \rightarrow bCc$)

aaabbCcc ($C \rightarrow bc$)

aaaaaaaaabbbbbCCCCBcc ($cB \rightarrow Bc$)

aaaaaaaaabbbbbCBccc ($CB \rightarrow bCc$)

aaaaaaaaabbbbbCCCCcc ($C \rightarrow bc$)

S, X, Y legyenek változók

a, b, c legyenek konstansok

$S \rightarrow abc, S \rightarrow aXbc, Xb \rightarrow bX, Xc \rightarrow Ybcc, bY \rightarrow Yb, aY \rightarrow aaX, aY \rightarrow aa$

A grammatika forrása az előadás fólia.

S ($S \rightarrow aXbc$)

aXbcc ($Xb \rightarrow bX$)

aaabbXc ($Xc \rightarrow Ybcc$)

aabbYbcc ($bY \rightarrow Yb$)

aaaYbbbcc ($aY \rightarrow aaX$)

aaaaXbbcccc ($Xb \rightarrow bX$)

aaabbbXbcccc ($Xb \rightarrow bX$)

Ez egy generatív nyelv, amelyet Noam Chomsky nyelvész dolgozott ki. Főként formális nyelvek osztályozásához alkalmazzák. A nyelvtannak 4 fő összetevője van: a konstansok, a változók, egy kitüntetett szimbólum (itt S), és a helyettesítési szabályok. A helyettesítésekre ügyelnünk kell. A nyelv környezetfügő, tehát a szó hossza nem csökkenhet generálás közben.

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:

Tanulságok, tapasztalatok, magyarázat...

Kernighan Brian W. és Ritchie Dennis M., A C programozási nyelv, 1993-ban kiadott könyve alapján írom itt le az utasítás fogalmát. Az utasításokkal hajtjuk végre a feladatokat, amiket szeretnénk. Az utasítások után minden teszünk ; jelet, például: printf(...amit ki szeretnénk íratni...); . Léteznek utasítás tömbök, vagy más néven blokkok is, amelyek így egyetlen utasításként funkciolálnak. A tömböket "utasítás utána utasításoknak" is nevezhetjük. Az utasítástömbök minden { } zárójelek között vannak. Például egy for(...) vagy while(feltétel) után nyitjuk a { zárójelet, ahová beírjuk az összes végrehajtani kívánt utasítást, majd zárjuk a } zárójelet.

A C nyelvet többször is fejlesztették, és javítottak rajta, ezekre szolgálnak a szabványok. A C89 szabványt 1983-ban, míg a C99 szabványt 2000-ben adták ki. A C89-nek több hibája, elmaradottsága is van a C99-hez képest. Például a C89-ben a for ciklusfejben nem lehet változót deklarálni, és ez a szabvány a C99-cel ellentétben a // jelű kommentet sem ismeri. Az regyetlen komment típus amit ismer, az a /* */ . A fordítás C89 szabvány szerint: gcc -o hivatkozas -std=c89 hivatkozas.c . A C99 fordítása hasonló: gcc -o hivatkozas -std=c99 hivatkozas.c

3.4. Saját lexikális elemző

Ír 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 állunk és ne kispályázzunk!

Megoldás videó:

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/thematic_tutorials/bhax_textbook_IgyNeveldaProgram

Tanulságok, tapasztalatok, magyarázat...

A lexek felhasználásával lexikális elemző programot tudunk létrehozni. Jelen esetben egy adott szövegben szereplő számokat számolja meg. Erre szolgál a szamok_szama nevezetű változó. Amikor a szövegben talál egy számot, akkor megnöveli 1-gyel a változó értékét. Ctrl+D parancssal tudjuk megszakítani a program futását, ez az input végét jelenti, majd ezután kiírja, hogy hány darabot talált. A %% jelek elválasztási céllal működnek, azok darabolják fel a programot részekre. A program végén pedig kiírjuk a valós számok

számát. Először egy C nyelvű kódot kell generálnunk: lex -o lexszam.c lexszam.l . A fordításhoz van szükségünk a lexszam.c-re: gcc lexszam.c -o lexszam -lfl . A futtatás már a normál megsokott módon történik ./lexszam. Megszakítás Ctrl+D paranccsal. Egy működő leet, a bhax-ról:

A leet fájl, realnumber.l

```
% {  
#include <stdio.h>  
int realnumbers = 0;  
%}  
digit [0-9]  
%%  
{digit}*(\.{digit}+) ? {++realnumbers;  
    printf("[realnum=%s %f]", yytext, atof(yytext));}  
%%  
int  
main ()  
{  
    yylex();  
    printf("The number of real numbers is %d\n", realnumbers);  
    return 0;  
}
```

A lexszam.l, amelynek működéséről írtam kicsit:

```
% {  
#include <string.h>  
int szamok_szama = 0;  
%}  
%%  
[0-9]+ ++szamok_szama;  
[a-zA-Z] [a-zA-Z0-9]* ;  
%%  
int  
main()  
{  
    yylex();  
    printf("%d szam\n", szamok_szama);  
    return 0;  
}
```

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Tutorált (am volt): Ranyhóczki Mariann, Ignéczi Tibor

Megoldás videó:

Megoldás forrása: BHAX csatorna forrásai

Tanulságok, tapasztalatok, magyarázat...

A lexerrel elkészítjük a C nyelvű programunkat, a parancs amit Linuxon használnunk kell hozzá: lex -o leet.c leet.l . Ezután fordítjuk a programot, majd megkapjuk a futtatható fájlt: gcc leet.c -o leet -lfl . A leet.l fájlban a %% szakasztörést jelent. Az includeban szükségünk van a time.h fájlra, azaz az időre, ami a random számok generálásához kell. A generált számra a betű, illetve szám helyettesítéshez van szükségünk. A változtatni kívánt betűk és számok után újabb 4 karakter van írva {} zárójelek közé, ezekkel lesznek helyettesítve. És innentől van szükségünk a random számokra. A program generál egyet, ami maximum 100 lehet, és elkezdi vizsgálni. Ha a szám 91-től kisebb, akkor a 4 választható karakter "oszlop" közül az első fogja választani. Ha 91-95 között van, akkor a másodikat, ha 95-98 között, akkor a harmadikat, másképp, azaz 98-100 között a negyediket fogja választani a tömbből.

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

#define L337SIZE (sizeof 1337d1c7 / sizeof (struct cipher))

struct cipher {
    char c;
    char *leet[4];
} l337d1c7 [] = {

{'a', {"4", "4", "@", "/-\\"}}, {'b', {"b", "8", "|3", "|{}"}}, {'c', {"c", "(", "<", "{"}}, {'d', {"d", "|)", "|]", "|{}"}}, {'e', {"3", "3", "3", "3"}}, {'f', {"f", "|=", "ph", "|#"}}, {'g', {"g", "6", "[", "[+"}}, {'h', {"h", "4", "|-", "|[-"]}}, {'i', {"1", "1", "|", "!"}}, {'j', {"j", "7", "_|", "_/"}}, {'k', {"k", "|<", "1<", "|{"}}, {'l', {"l", "1", "|", "|_"}}, {'m', {"m", "44", "(V)", "|\\/|"}}, {'n', {"n", "|\\|", "/\\/", "/V"}}, {'o', {"0", "0", "()", "[]"}}, {'p', {"p", "/o", "|D", "|o"}}, {'q', {"q", "9", "O_", "(,)"}}, {'r', {"r", "12", "12", "|2"}}, {'s', {"s", "5", "$", "$"}}, {'t', {"t", "7", "7", "'|'"}}}, {'u', {"u", "|_|", "(_)", "[_]"}}, {'v', {"v", "\\\/", "\\\/", "\\\/"}}}, {'w', {"w", "VV", "\\\/\\\/", "(/\\)"}}, {'x', {"x", "%", ")(")}, {"y", {"y", "", ""}}, {"z", {"z", "2", "7_", ">_"}},
```

```
{'0', {"D", "O", "D", "O"}},  
{'1', {"I", "I", "L", "L"}},  
{'2', {"Z", "Z", "Z", "e"}},  
{'3', {"E", "E", "E", "E"}},  
{'4', {"h", "h", "A", "A"}},  
{'5', {"S", "S", "S", "S"}},  
{'6', {"b", "b", "G", "G"}},  
{'7', {"T", "T", "j", "j"}},  
{'8', {"X", "X", "X", "X"}},  
{'9', {"g", "g", "j", "j"}}

// https://simple.wikipedia.org/wiki/Leet
};

%}
%%%
. {

    int found = 0;
    for(int i=0; i<L337SIZE; ++i)
    {

        if(l337d1c7[i].c == tolower(*yytext))
        {

            int r = 1+(int) (100.0*rand() / (RAND_MAX+1.0));

            if(r<91)
                printf("%s", l337d1c7[i].leet[0]);
            else if(r<95)
                printf("%s", l337d1c7[i].leet[1]);
            else if(r<98)
                printf("%s", l337d1c7[i].leet[2]);
            else
                printf("%s", l337d1c7[i].leet[3]);

            found = 1;
            break;
        }

        if(!found)
            printf("%c", *yytext);

    }
}

%%%
int
main()
{
```

```
    srand(time(NULL)+getpid());
    yylex();
    return 0;
}
```

3.6. A források olvasása

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

```
if(signal(SIGINT, jelkezelo)==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 jelkezelo függvény kezelje. (Miután a **man 7 signal** lapon megismertem a SIGINT jelet, a **man 2 signal** lapon pedig a használt rendszerhívást.)

Ezzel a pici programmal kiíratjuk a megfogott signalt. Fordítás: gcc forras.c -o forras és a futtatás: ./forras .Megszakítás Ctrl+C billentyűkombinációval, ekkor írja ki, hogy A signal: 2.(jelen esetben)

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

void jelkezelo(int sig){
    printf("A signal %d\n", sig);
}

int main() {
    for(;;) {
        if(signal(SIGINT, jelkezelo)==SIG_IGN)
            signal(SIGINT, SIG_IGN);
    }
}
```



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, jelkezelo);
```

ii.

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

iii.

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

iv.

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

v.

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

vi.

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

vii.

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

viii.

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

Megoldás forrása:

Megoldás video:

Tanulságok, tapasztalatok, magyarázat...

A program működőképességéhez szükséges includolni a signal.h fájlt. Létrehozunk egy jelkezelő nevű függvényt, ebben lesz a kiíratás. A főfüggvényben szereplő végtelen ciklus miatt manuálisan kell megszakítani a futást, ugyanis a ciklus végig ellenőrzi. Amikor Ctrl+C parancssal megállítjuk a futást, akkor kiírja a program az elkapott signalt.

A Bugok, splint használatával:

Normál fordítással lefordul a program, de eredményül nem ad semmit, illetve létre kell hoznunk a jelkezelő függvényt. A hibaüzenet splint mellett:

```

Tevékenységek Terminal
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog1/2.Chomsky/Bugok
Fájl Szerkesztés Nézet Keresés Terminal Súgó
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/2.Chomsky/Bugok$ splint
t.c
Splint 3.1.2 --- 20 Feb 2018

t.c: (in function main)
t.c:10:2: Return value (type [function (int) returns void]) ignored:
    signal(SIGINT, j...
    Result returned by function call is not used. If this is intended, can cast
    result to (void) to eliminate message. (Use -fretvaltozero to inhibit warning)
t.c:11: Path with no return to function declared to return int
    There is nothing else of function declared to return int
    on which there
    is no return statement. This means the execution may fall through without
    returning a meaningful result to the caller. (Use -noret to inhibit warning)
t.c:46: Function exported but not used outside t: jelkezelo
    A declaration is exported, but not used outside this module. Declaration can
    use static qualifier. (Use -exportlocal to inhibit warning)
t.c:6: Definition of jelkezelo

Finished checking --- 3 code warnings
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/2.Chomsky/Bugok$
```

```

h 22.25
Mennyitás ▾ Ic Mentés
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/2.Chomsky/Bugok
#include<stdio.h>
#include<signal.h>

void jelkezelo(int sig){
    printf("A signal %d\n", sig);
}

int main(){
    if(signal(SIGINT, SIG_IGN)!=SIG_IGN)
        signal(SIGINT, jelkezelo);
}
```

Az ii. feladatban a for ciklus először megnöveli eggyel az i értékét, majd utána fut le a ++i miatt. A hiba splint mellett:

The screenshot shows a Linux desktop environment with a dark theme. On the left is a dock with various icons. In the center, there's a terminal window titled "Tevékenységek" and "Földi Szerkesztés Nézet Keresés Terminal Súgó". It displays the command "splint t iii.c" and its output, which includes a warning from Splint about a function returning void. To the right of the terminal is a code editor window titled "h 22.25" showing a C file named "II.c". The code contains a main function with a loop that does not return a value. Below the code editor, status bars show "C", "Tabulátorszélesség: 8", "1. sor, 1. oszlop", and "BESZ".

Az iii. feladat ugyan olyan mint az előző, annyi a különbség, hogy itt először lefut a program, majd az i-t csak utána növeli meg. A hiba splint mellett:

This screenshot is similar to the previous one, showing the same terminal and code editor setup. The terminal window now shows the command "splint t iii.c" and its output, including the warning from Splint. The code editor window shows the same "II.c" file. The status bars at the bottom are identical to the previous screenshot.

Az iv. feladatban normál fordítás mellett deklarálnunk kell egy i változót és egy tomb nevű tömböt. A hibaüzenet splint mellett:

The screenshot shows a Linux desktop environment with a dark theme. On the left is a dock with various icons. In the center-left is a terminal window titled "Tevékenységek" with the sub-titles "Fájl Szerkesztés Nézet Keresés Terminal Súgó". It displays the command "splint v.c" and its output, which includes several warnings from the Splint static code checker. On the right is a code editor window titled "h 22.28" showing a C file named "iv.c". The code contains a main function with a for loop that increments a variable "tomb[i]" by 1. A status bar at the bottom of the code editor shows "C ▾ Tabulátorszélesség: 8 ▾ 1. sor, 1. oszlop ▾ BESZ".

A v. feladatban már mutatókkal kell dolgozni, és ezeket is deklarálni kell a fordításhoz, lefordul, de valójában hibás. A hibaüzenet splittel:

The screenshot shows a Linux desktop environment with a dark theme. On the left is a dock with various icons. In the center-left is a terminal window titled "Tevékenységek" with the sub-titles "Fájl Szerkesztés Nézet Keresés Terminal Súgó". It displays the command "splint v.c" and its output, which includes several warnings from the Splint static code checker. On the right is a code editor window titled "h 22.28" showing a C file named "iv.c". The code contains a main function with a for loop that increments a variable "tomb[i]" by 1. A status bar at the bottom of the code editor shows "C ▾ Tabulátorszélesség: 8 ▾ 1. sor, 1. oszlop ▾ BESZ".

A vi. feladatban kiíratás szerepel egy a változó és f függvény segítségével. A `++a` jelentése, hogy először megnöveli az a-t mielőtt bármit csinálna. A hibaüzenet splittel:

A screenshot of a Linux desktop environment. On the left is a dark-themed application menu with various icons. In the center-left is a terminal window titled 'Szövegszerkesztő' (Text Editor) showing the output of the Splint static checker. It lists several warnings from file 'vii.c' at line 22.38, column 18, related to variable 'a' being used before it is defined. The terminal also shows the command 'splint vli.c' and the date 'Splint 3.1.2 --- 20 Feb 2018'. On the right is a code editor window titled 'vii.c' showing the C code for the program. The code includes a header '#include<stdio.h>' and a main function that prints the value of 'a' using printf("%d %d", f(a, a++), f(++a, a)). A warning message is visible in the status bar at the bottom right of the code editor.

A vii. feladatban újra kiíratás van printf függvényel, az a változóval. A hibaüzenet splittel:

A screenshot of a Linux desktop environment, similar to the previous one. The terminal window on the left shows the same Splint output as before, with 2 code warnings. The code editor window on the right shows the same C code for 'vii.c'. The status bar at the bottom right indicates 'Tabulátorszélesség: 8' (Tab width: 8). The overall layout is identical to the first screenshot.

A viii. feladatban továbbra is printf függvény, a változó és a mutató szerepel. A hibaüzenet splittel:

```

petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/2.Chomsky/Bugok$ splint viti.c
Fájl Szerkesztés Nézet Keresés Terminál Súgó
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/2.Chomsky/Bugok$ splint viti.c
Splint 3.1.2 ... 20 Feb 2018
viti.c: (in function main)
dm'viti.c:5:18: Unrecognized identifier: f
Identifier used in code has not been declared. (Use -unrecog to inhibit
warning)
viti.c:5:25: Variable a used before definition
An rvalue is used that may not be initialized to a value on some execution
path. (Use -usedef to inhibit warning)
Finished checking ... 2 code warnings
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/2.Chomsky/Bugok$ 
```

```

#include<stdio.h>
int main(){
    int a;
    printf("%d %d", f(&a), a);
    return 0;
}
```

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 (\exists y \text{ prim}) \leftrightarrow ) $  

$(\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 első kifejezés szerint bármelyik x mellett találunk olyan y-t, ahol y-tól kisebb az x, és az y prím szám. Eszerint az állítás szerint a prím számok száma végtelen sok.

A második szerint bármelyik x-hez van olyan y, amitől x kisebb, minden prím, valamint y+2 is prím.

A harmadik szerint létezik olyan y, hogy minden x-nél, ha az prím, akkor y nagyobb, mint x.

A negyedik szerint létezik olyan y, hogy minden x-nél, ha x nagyobb, mint y, abban az esetben az x szám nem prím. Eszerint a feltételezés szerint véges sok a prímek száma.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciajára
- egészek tömbje
- egészek tömbjének referenciajára (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;`
- `int *b = &a;`
- `int &r = a;`
- `int c[5];`
- `int (&tr)[5] = c;`
- `int *d[5];`
- `int *h();`
- `int *(*l)();`
- `int (*v(int c))(int a, int b)`

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

Megoldás video:

Megoldás forrása: [GitLab](#), [BHAX](#)

Tanulságok, tapasztalatok, magyarázat...

A deklarációkkal változókat vezetünk a programunkba. A C nyelvben a változókat mindig a program elején vezetjük be, ebben az esetben biztosan nem kapunk emiatt hibaüzenetet. A mostani programban a typedef parancssal egy új típust definiálunk. Ezt csak egyszer kell végrehajtanunk, a továbbiakban pedig sokkal könnyebben használhatjuk, nem kell újra kifejteni semmit sem, és a programunk is rövidebb lesz. A globális változók az egész programon keresztül élnek, tehát ha újra bevezetnénk valamit ugyan olyan néven, akkor hibásan működne a program. Ha függvényt hozunk létre a () zárójeleken belül is tudunk deklarálni, ezek a változók csak ezen a függvényen belül élnek, így később egy másik függvényben újra használhatjuk ugyan azt a változó nevet. Ebben a programban például a sum függvény szerepel, ahol az a és b változót a függvényen belül vezetjük be, majd a visszatérési értéke ennek a két változó értékének az összege lesz. Ettől függetlenül kedvünk szerint használhatunk tömböket és mutatókat is, attól függően, hogy mit kíván az adott feladat, mivel gondoljuk a megoldást a legcél szerűbbnek.

```
#include <stdio.h>

typedef int (*F) (int, int);
typedef int (*(*G) (int)) (int, int);

int
sum (int a, int b)
{
 return a + b;
}

int
mul (int a, int b)
{
 return a * b;
}

F sumormul (int c)
{
 if (c)
 return mul;
 else
 return sum;
}

int
main ()
{
 F f = sum;
```

```
printf ("%d\n", f (2, 3));
G g = sumormul;
f = *g (42);
printf ("%d\n", f (2, 3));
return 0;
}
```

## 4. fejezet

# Helló, Caesar!

### 4.1. double \*\* háromszögmátrix

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

Megoldás videó: <https://youtu.be/1MRTuKwRsB0>, <https://youtu.be/RKbX5-EWpzA>.

Megoldás forrása: [bhax/thematic\\_tutorials/bhax\\_textbook\\_IgyNeveldaProgramozod/Caesar/tm.c](https://bhax.com/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c)

Tanulságok, tapasztalatok, magyarázat...

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

int
main ()
{
 int nr = 5; //also haromszog matrix sorainak a szama
 double **tm; //helyfoglalas

 printf("%p\n", &tm); //pointer, cim kiiratas

 if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL) // ←
 // malloc pointert ad vissza a lefoglalt teruletrol (man 3-ban meg ←
 // lehet nezni), 40 bajtot foglal
 {
 return -1; //ha az if igaz, kiugrik a programbol
 }

 printf("%p\n", tm);

 for (int i = 0; i < nr; ++i)
 {
 if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL) ←
 // a double*-ra mutat, oda adj a cim nevet
```

```
{
 return -1;
}

}

printf("%p\n", tm[0]);

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");
}

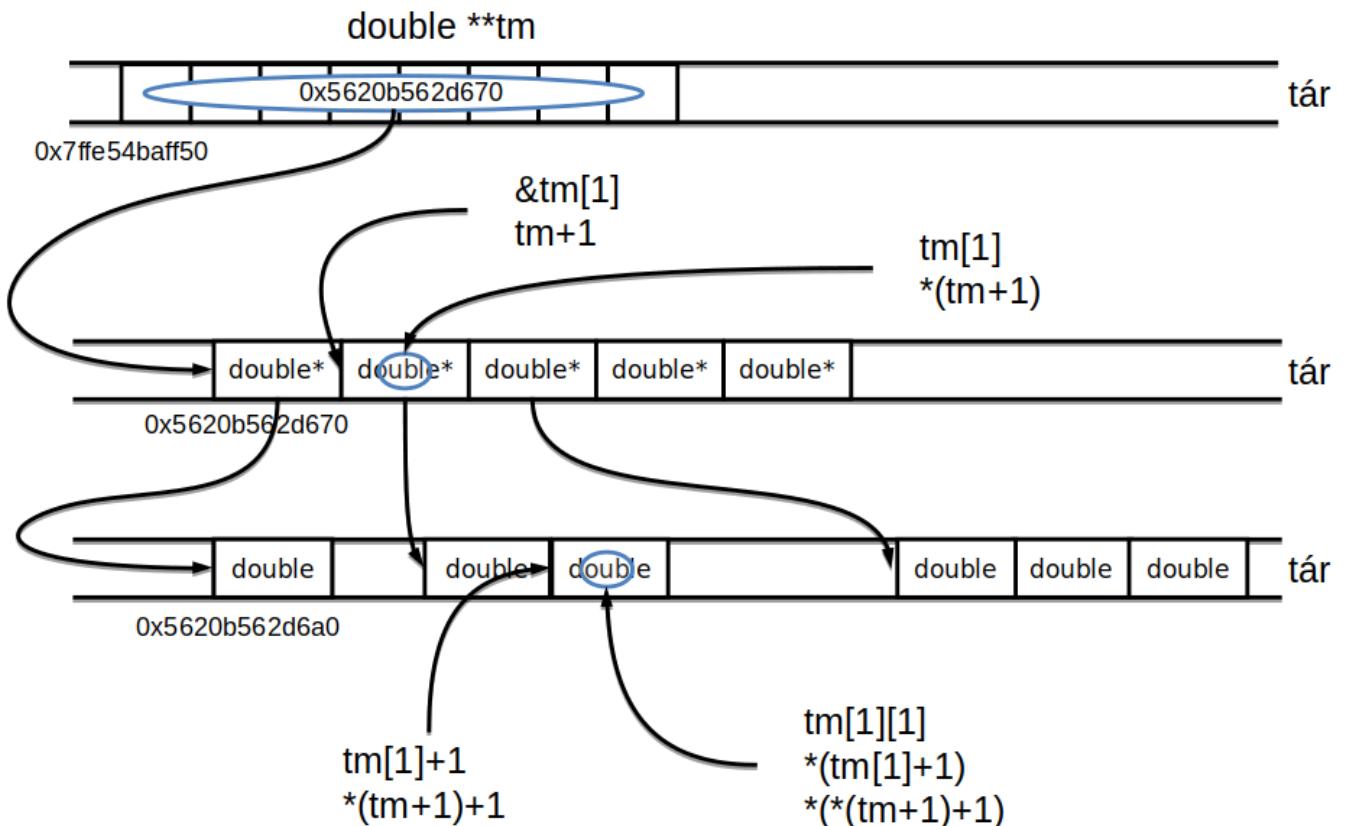
tm[3][0] = 42.0;
(* (tm + 3))[1] = 43.0;
*(tm[3] + 2) = 44.0;
* (* (tm + 3) + 3) = 45.0;

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

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

free (tm); //felszabadítja a tm számára lefoglalt memóriaterületet

return 0;
}
```



```

petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/4.Caesar
Fájl Szerkesztés Nézet Keresés Terminál Súgó
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/4.Caesar$./matrix
0x7ffd9e1d81d0
0x558028073670
0x5580280736a0
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
6.000000, 7.000000, 8.000000, 9.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
42.000000, 43.000000, 44.000000, 45.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/4.Caesar$

```

Az `double **` háromszögmátrixot más néven több dimenziós mátrixnak is nevezzük. A mátrixokról már

tanulhattunk órán is, tehát tudjuk, hogy egy alsó háromszögmátrixnál a főátlót kell figyelnünk, mert efölött csupa 0 számnak kell állnia. Ennek az alsó háromszögmátrix kiíratásának a folyamata: első sorban 1 elem, második sorban 2 elem, ..., és ötödik sorban már 5 elem van. Az elején kiírja a 3 memóriacímet is a program. A főbb elemek működési elvét kommentek formájában írtam bele a programba. 2 dolog maradt ki. A for ciklusok feladata az, hogy lépkedjenek a mátrix elemein soronként, illetve azok kiíratása. A program végén a free utasítás a malloc parancsot "teszi semmissé", tehát ez felszabadítja tm számára lefoglalt memóriaterületet. A kódba kommentek formájában hozzáfüztem a memória foglalás néhány részét, azt itt pluszban nem írnám le újra.

## 4.2. C EXOR titkosító

Tutor (om volt): Ignéczi Tibor

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: [progpater.blog.hu](http://progpater.blog.hu)

Tanulságok, tapasztalatok, magyarázat...

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

#define MAX_KULCS 100
#define BUFFER_MERET 256

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);

 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 az exor program szövegek titkosítására szolgál. A titkosítást adatvédelem céljából szoktuk használni, hogy illetéktelen személy ne tudhassa meg, mit írtunk. A titkosításhoz egy 8 számjegyű kulcsot használunk jelen esetben. A definiálásnál a kulcsra pont emiatt van szükségünk, illetve definiálunk egy buffert is, amire a tárolás miatt van szükségünk. A fordításnál és futtatásnál ügyelnünk kell, ha nem vagyunk pontosak, nem fog működni. Ahhoz, hogy elkezdhetünk, szükségünk van egy tiszta szövegre, vagyis az alapra, amit titkosítani akarunk. Én ezt tiszta.txt-nek neveztem el. A kódolás folyamata: gcc e.c -o e -std=c99. Az std=c99-re annyiból van szükség, hogy a c99-es szabvány szerint fordítsa le a programot. A szöveg titkosítása:

```
./e kercerece <tiszta.txt >titkos.szoveg
```

Ha kíváncsiak vagyunk a titkosított szövegre, akkor ki kell adnunk a more titkos.szoveg parancsot, így megnézhetjük a terminálon. Mappából nem tudjuk megnyitni, ha nincs hozzá külön programunk. Majd a kódolt szöveget is vissza tudjuk bontani, és kiíratni terminálban:

```
./e kercerece <tiszta.txt
```

## 4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: BHAX csatorna és forrásai

Tanulságok, tapasztalatok, magyarázat...

```
import java.io.InputStream;
import java.io.OutputStream;

public class Exor {

 public Exor(String kulcsSzöveg,
 java.io.InputStream bejövőCsatorna,
 java.io.OutputStream kimenőCsatorna)
 throws java.io.IOException {

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

 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);

}

}

public static void main(String[] args) {

 try {

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

 } catch(java.io.IOException e) {

 e.printStackTrace();

 }

}
```

Ugyan úgy működik futás közben, mint az exor titkosító C-ben, ami fentebb le van írva. A program osztálytallosan dolgozik. Megvizsgálja a bájtokat, majd elkezdi a titkosítást, ha minden stimmel. Adatvédelem szempontjából praktikus a program. A bejövő csatárán történik az adatok beolvasása, és utána a bufferben ideiglenesen tárolódnak az adatok. A kulcsnak pedig a törésnél van nagyobb szerepe. Fordítása: javac Exor.java . A futtatása pedig: ./Exor szavak > titkos.txt

## 4.4. C EXOR törő

Tutor (om volt): Ignéczi Tibor

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

Megoldás videó:

Megoldás forrása: BHAX csatorna és forrásai

Tanulságok, tapasztalatok, magyarázat...

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

```
#define _GNU_SOURCE

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

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;
}

int
tiszta_lehet (const char *titkos, int titkos_meret)
{
 // a tiszta szöveg valszeg tartalmazza a gyakori magyar szavakat
 // illetve az átlagos szóhossz vizsgálatával csökkentjük a
 // potenciális töréseket

 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");

}

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;

 }
}

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);
}

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

 // 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';

 // 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, ily nem kell egy ←
masodik buffer
exor (kulcs, KULCS_MERET, titkos, p - ←
titkos);
}

return 0;
}
```

A program működési elve szakaszosan bele van építve a programba kommentek formájában, illetve sok részében egyezik az exor titkosítóval, ezért nem írnám le itt is újra. A program a titkosított szövegek feltörésére szolgál, tehát visszaírja az eredeti formájába. Ha a titkosító e.c-t másképp futtatjuk le, és megadjuk neki a kulcsot, akkor lesz értelme ennek a programnak:

```
./e kercerece <tiszta.txt >titkos.szoveg
```

A more parancs itt is működik. A t.c program fordítása: gcc t.c -o t -std=c99 . Futtatása:

```
time ./t <titkos.szoveg |grep 12345678
```

Az std=c99-re amiatt van szükség, hogy c99 szabvány szerint forduljon a program, valamint a greppel megadjuk a 8 jegyű kulcsot. A futtatáskor kiírja a terminál, hogy mennyi időbe telt, még elvégezte a műveletet. Nem fut le azonnal, percekbe telik, de sokat csökkenthetünk a futási időn, ha másképp fordítunk, tehát így: gcc t.c -O3 -o t -std=c99 . A futtatás parancsa ugyan az, mint ahogyan a lassabnál írtam.

## 4.5. Neurális OR, AND és EXOR kapu

R

Tutor (om volt): Duszka Ákos Attila, Takács Viktor

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)

Tanulságok, tapasztalatok, magyarázat...

Ez a szimulációs program a neurális hálók segítségével dolgozik. A neurális hálók neuronokból állnak, amelyek agyi idegsejteknek felelnek meg, de programozásban is használják őket. Főleg a mesterséges intelligenciák létrehozásában van nagy szerepük. Ebben a szimulációban a neuronoknak megpróbálják megtanítani, hogy hogyan használjanak 3 logikai műveletet, az OR (megengedő vagy), AND (és) és EXOR(kizáró vagy). Az OR és AND műveletekkel nem kell sokat számolatni, viszonylag gyorsan és pontos eredményt kaphatunk így. Rejtett rétegeket, azaz hiddeneket is használ a program. Az OR rétegen az a1 és a2-nak adunk értéket, és végrehajtunk rajtuk logikai vagy műveleteket, amivel parancsot tanulnak meg. Az AND

már logikai műveletet is képes tanulni. Az EXORhoz már szükségünk van plusz rétegre. Ide kellenek a rejtett hálók, és a hidden értékek. A neuronok és neurális hálók eléréséhez telepíthetjük a neuralnet függvénycsomagot, pontosabban a neuralnetwork könyvtárra van szükségünk. Mivel mesterséges intelligencia az alapja, ezért ez a program is tanul. Ezt az első minta megadásával kezdhetjük meg. Súlyozást használ, de ezt jelenleg még nem nekünk kell megadnunk, hanem elvégzi magától a program. Amennyiben rejtett neuronokat használunk, manipulálhatjuk a programban a megoldásig vezető lépésszámot. Meg kell növelni ezeknek a mennyiségett, így kevesebb lépéssel is elvégezhető a feladat. A végén tudjuk ellenőrizni is, hogy mennyire sikerült megtanulnia a programnak az adott minta követését a számítások során.

```
Copyright (C) 2019 Dr. Norbert Bátfai, nbatfai@gmail.com
#
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
#
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
#
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>
#
https://youtu.be/Koyw6IH5ScQ
library(neuralnet)

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)

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])

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

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

nn.inand <- neuralnet(OR+AND~a1+a2, inand.data, hidden=0, linear.output= ←
```

```
 FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.operand)

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

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])

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=c(6, 4, 6), linear.←
 output=FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

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

## 4.6. Hiba-visszaterjesztéses perceptron

C++

Tutor (om volt): Duszka Ákos Attila

Megoldás videó:

Megoldás forrása, szöveg: <http://mialmanach.mit.bme.hu/neuralis/ch04>

Tanulságok, tapasztalatok, magyarázat...

A main.cpp:

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

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);

 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);

 std::cout << value << std::endl;

 delete p;
 delete [] image;
}
```

Az mlp.hpp amely a perceptron osztályt tartalmazza:

```
#include <iostream>
#include <cstdarg>
#include <map>
#include <iterator>
#include <cmath>
#include <random>
#include <limits>
#include <fstream>

class Perceptron
{
public:
 Perceptron (int nof, ...)
 {
 n_layers = nof;

 units = new double*[n_layers];
 n_units = new int[n_layers];

 va_list vap;

 va_start (vap, nof);
```

```
for (int i {0}; i < n_layers; ++i)
{
 n_units[i] = va_arg (vap, int);
}

if (i)
 units[i] = new double [n_units[i]];
}

va_end (vap);

weights = new double**[n_layers-1];

#ifndef RND_DEBUG
 std::random_device init;
 std::default_random_engine gen {init() };
#else
 std::default_random_engine gen;
#endif

std::uniform_real_distribution<double> dist (-1.0, 1.0);

for (int i {1}; i < n_layers; ++i)
{
 weights[i-1] = new double *[n_units[i]];

 for (int j {0}; j < n_units[i]; ++j)
 {
 weights[i-1][j] = new double [n_units[i-1]];

 for (int k {0}; k < n_units[i-1]; ++k)
 {
 weights[i-1][j][k] = dist (gen);
 }
 }
}

Perceptron (std::fstream & file)
{
 file >> n_layers;

 units = new double*[n_layers];
 n_units = new int[n_layers];

 for (int i {0}; i < n_layers; ++i)
 {
 file >> n_units[i];

 if (i)
```

```
 units[i] = new double [n_units[i]];
}

weights = new double**[n_layers-1];

for (int i {1}; i < n_layers; ++i)
{
 weights[i-1] = new double *[n_units[i]];

 for (int j {0}; j < n_units[i]; ++j)
 {
 weights[i-1][j] = new double [n_units[i-1]];

 for (int k {0}; k < n_units[i-1]; ++k)
 {
 file >> weights[i-1][j][k];
 }
 }
}

double sigmoid (double x)
{
 return 1.0/ (1.0 + exp (-x));
}

double operator() (double image [])
{
 units[0] = image;

 for (int i {1}; i < n_layers; ++i)
 {

#ifndef CUDA_PRCPS

 cuda_layer (i, n_units, units, weights);

#else

 #pragma omp parallel for
 for (int j = 0; j < n_units[i]; ++j)
 {
 units[i][j] = 0.0;

 for (int k = 0; k < n_units[i-1]; ++k)
 {
 units[i][j] += weights[i-1][j][k] * units[i-1][k];
 }
 }
 }
}
```

```
 }

 units[i][j] = sigmoid (units[i][j]);

 }

#endif

}

return sigmoid (units[n_layers - 1][0]);

}

void learning (double image [], double q, double prev_q)
{
 double y[1] {q};

 learning (image, y);
}

void learning (double image [], double y[])
{
 //(*this) (image);

 units[0] = image;

 double ** backs = new double*[n_layers-1];

 for (int i {0}; i < n_layers-1; ++i)
 {
 backs[i] = new double [n_units[i+1]];
 }

 int i {n_layers-1};

 for (int j {0}; j < n_units[i]; ++j)
 {
 backs[i-1][j] = sigmoid (units[i][j]) * (1.0-sigmoid (units[i][j])) * (y[j] - units[i][j]);

 for (int k {0}; k < n_units[i-1]; ++k)
 {
 weights[i-1][j][k] += (0.2* backs[i-1][j] *units[i-1][k]);
 }
 }

 for (int i {n_layers-2}; i >0 ; --i)
 {
```

```
#pragma omp parallel for
for (int j =0; j < n_units[i]; ++j)
{
 double sum = 0.0;

 for (int l = 0; l < n_units[i+1]; ++l)
 {
 sum += 0.19*weights[i][l][j]*backs[i][l];
 }

 backs[i-1][j] = sigmoid (units[i][j]) * (1.0-sigmoid (units ←
 [i][j])) * sum;

 for (int k = 0; k < n_units[i-1]; ++k)
 {
 weights[i-1][j][k] += (0.19* backs[i-1][j] *units[i-1][k] ←
);
 }
}

for (int i {0}; i < n_layers-1; ++i)
{
 delete [] backs[i];
}

delete [] backs;

}

~Perceptron()
{
 for (int i {1}; i < n_layers; ++i)
 {
 for (int j {0}; j < n_units[i]; ++j)
 {
 delete [] weights[i-1][j];
 }

 delete [] weights[i-1];
 }

 delete [] weights;

 for (int i {0}; i < n_layers; ++i)
 {
 if (i)
 delete [] units[i];
 }
}
```

```
 }

 delete [] units;
 delete [] n_units;

}

void save (std::fstream & out)
{
 out << " "
 << n_layers;

 for (int i {0}; i < n_layers; ++i)
 out << " " << n_units[i];

 for (int i {1}; i < n_layers; ++i)
 {
 for (int j {0}; j < n_units[i]; ++j)
 {
 for (int k {0}; k < n_units[i-1]; ++k)
 {
 out << " "
 << weights[i-1][j][k];
 }
 }
 }
}

private:
 Perceptron (const Perceptron &);
 Perceptron & operator= (const Perceptron &);

 int n_layers;
 int* n_units;
 double **units;
 double ***weights;
};
```

Ez egy kódcsipet a nahshon programból származik, jelenleg nekünk csak a perceptron osztályra van szükségünk a téma szerint, nem az egész programra. A program a neurális hálóra épül, vagyis a neuronokra, az agysejtekre amiknek a feladata az elektromos impulzusok kezelése. A neurális háló elve nélkül nem létezne a mesterséges intelligencia sem, és ezért van szükségünk perceptronokra, hogy létrehozhassuk a mesterséges intelligenciát. A perceptron feladata a tanulás, a függvény tanulás. A lineárisan elkülöníthető (vagyis szeparálható) függvényeket tanulja és használja könnyen. A perceptron lényegében egy hálózat, amit gyakorlati feladatok, problémák megoldására használnak. Ennek a hálónak a tanításakor alkalmazzák a hibavisszaterjesztéses (back-propagation) algoritmust legtöbbször. A perceptron osztály az általunk hoz-

záadott mandelbrot kép alapján dolgozik, a végén abból állít elő nekünk egy számot. A mandelbrot képünk rgb kódját adjuk át a neurális hálónak. Lényeges a rétegek száma. Fordítása: g++ mlp.hpp main.cpp -o perceptron -lpng -std=c++11 és a futtatása ./perceptron mandelbrot.png

## 5. fejezet

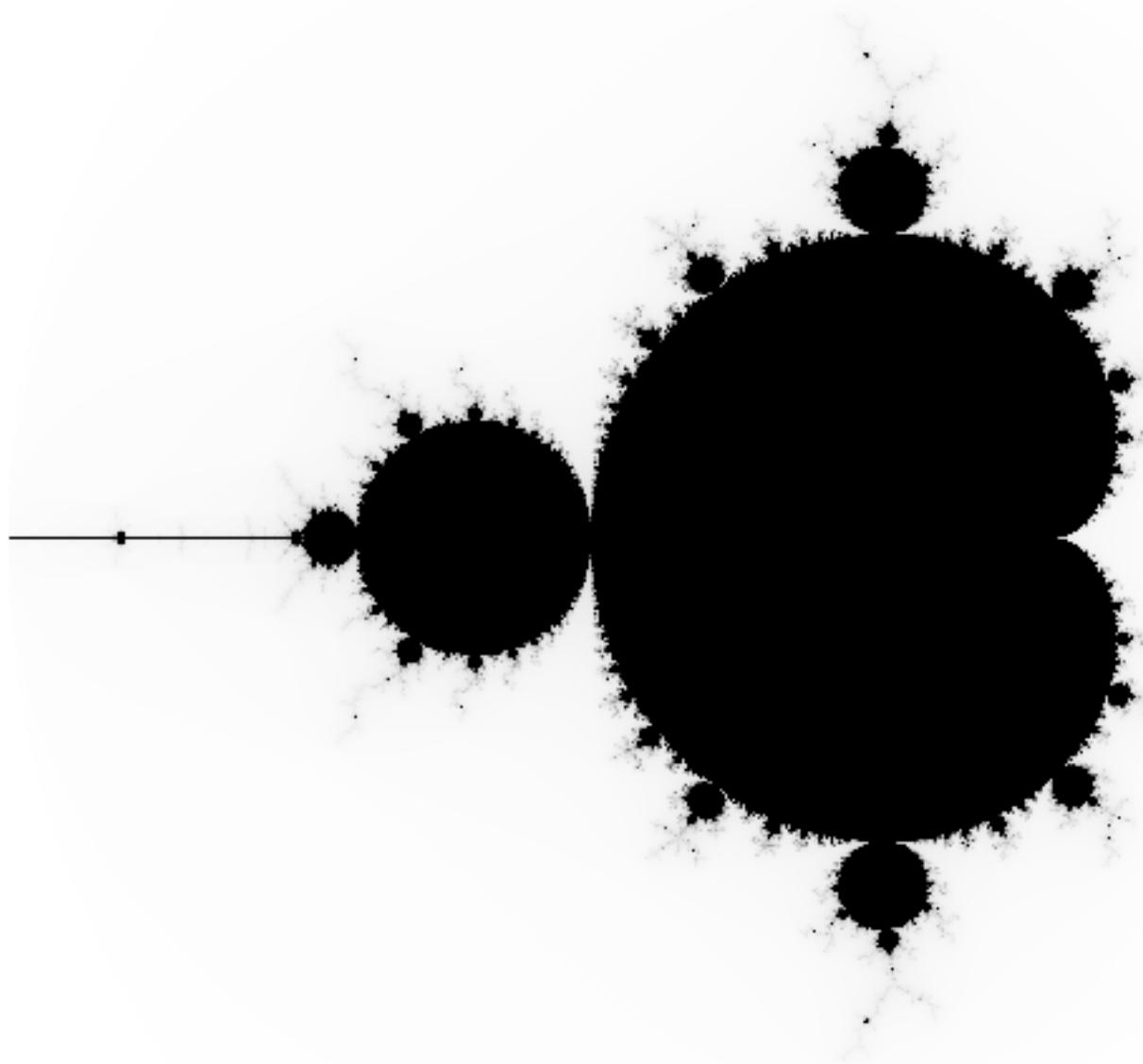
# Helló, Mandelbrot!

### 5.1. A Mandelbrot halmaz

Tutor (om volt): Egyed Anna

Megoldás videó:

Megoldás forrása: [https://gitlab.com/nbatfai/bhax/blob/master/attention\\_raising/CUDA/mandelpngt.c++](https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/CUDA/mandelpngt.c++)



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

#define N 500
#define M 500
#define MAXX 0.7
#define MINX -2.0
#define MAXY 1.35
#define MINY -1.35

void GeneratePNG(int tomb[N][M])
{
 png::image< png::rgb_pixel > image(N, M);
 for (int x = 0; x < N; x++)
```

```
{
 for (int y = 0; y < M; y++)
 {
 image[x][y] = png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y] ←
]);
 }
}
image.write("kimenet.png");
}

struct Komplex
{
 double re, im;
};

int main()
{
 int tomb[N][M];

 int i, j, k;

 double dx = (MAXX - MINX) / N;
 double dy = (MAXY - MINY) / M;

 struct Komplex C, Z, Zuj;

 int iteracio;

 for (i = 0; i < M; i++)
 {
 for (j = 0; j < N; j++)
 {
 C.re = MINX + j * dx;
 C.im = MAXY - i * dy;

 Z.re = 0;
 Z.im = 0;
 iteracio = 0;

 while(Z.re * Z.re + Z.im * Z.im < 4 && iteracio++ < 255)
 {
 Zuj.re = Z.re * Z.re - Z.im * Z.im + C.re;
 Zuj.im = 2 * Z.re * Z.im + C.im;
 Z.re = Zuj.re;
 Z.im = Zuj.im;
 }

 tomb[i][j] = 256 - iteracio;
 }
 }
}
```

```
 GeneratePNG(tomb) ;

 return 0;
}
```

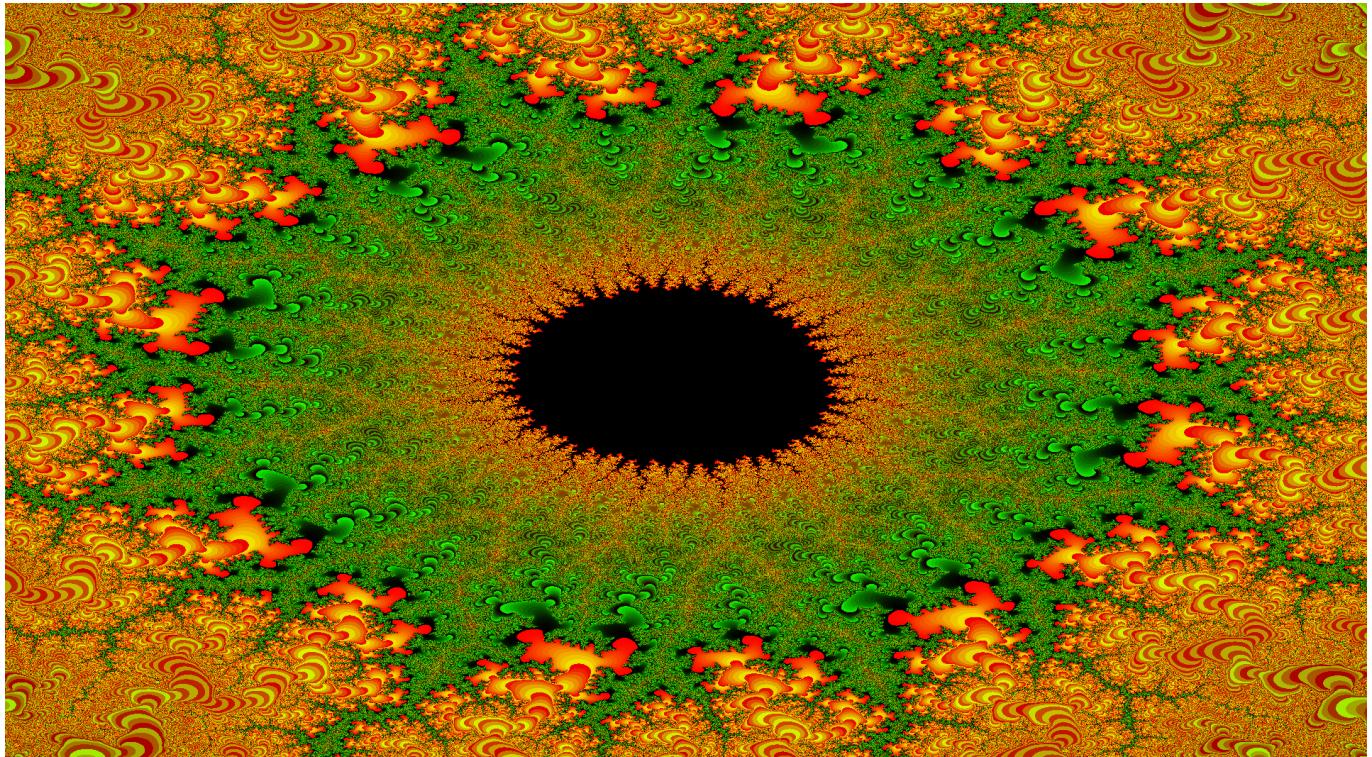
A Mandelbrot-halmaz atya Benoit Mandelbrot volt. 1980-ban fedezte fel a komplex számok adta lehetőségeket, például hogyan kaphatunk negatív számokat két másik szám szorzatából, ezt a célt szolgálja az i, mint "szám". A folyamatban a rácspontokat számoljuk a komplex számok halmazán. A képlete:  $z_{n+1} = z_n^2 + c$ , ahol a 0 nagyobb vagy egyenlő, mint n, a c pedig az aktuális rácspont, és az origóból indulunk. A megadott iteráció alapján számol, ez szorítja be a programot bizonyos keretek közé, ez az iterációs határ (255 itt). A program képpontonként számol, és így készíti el a képet, pontról pontra halad. A képlet amely alapján számol, a while cikluson belül van megfogalmazva. Csak véges sok elemet tudunk megvizsgálni, és az alapján színezzük a képet. A programban meg tudjuk adni a futtatáskor, hogy milyen néven mentse el a küpenket. Ha nem adjuk meg, akkor egy hibaüzenetet kapunk. Az image.write függvény készíti el a kimeneti képet, de a GeneratePNG void típusú függvény alkotja meg a tömbből.

## 5.2. A Mandelbrot halmaz a `std::complex` osztállyal

Tutorált (am volt): Egyed Anna, Ranyhóczki Mariann

Megoldás videó:

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



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

```
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;

 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: ./3.1.2 fajlnev szelesseg magassag n a b c d ←
 " << std::endl;
 return -1;
 }

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;

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

// j megy a sorokon
for (int j = 0; j < magassag; ++j)
{
 // k megy az oszlopokon

 for (int k = 0; k < szelesseg; ++k)

 // c = (reC, imC) a halo racspontjainak
 // megfelelo komplex szam
```

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

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

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));
}

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

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

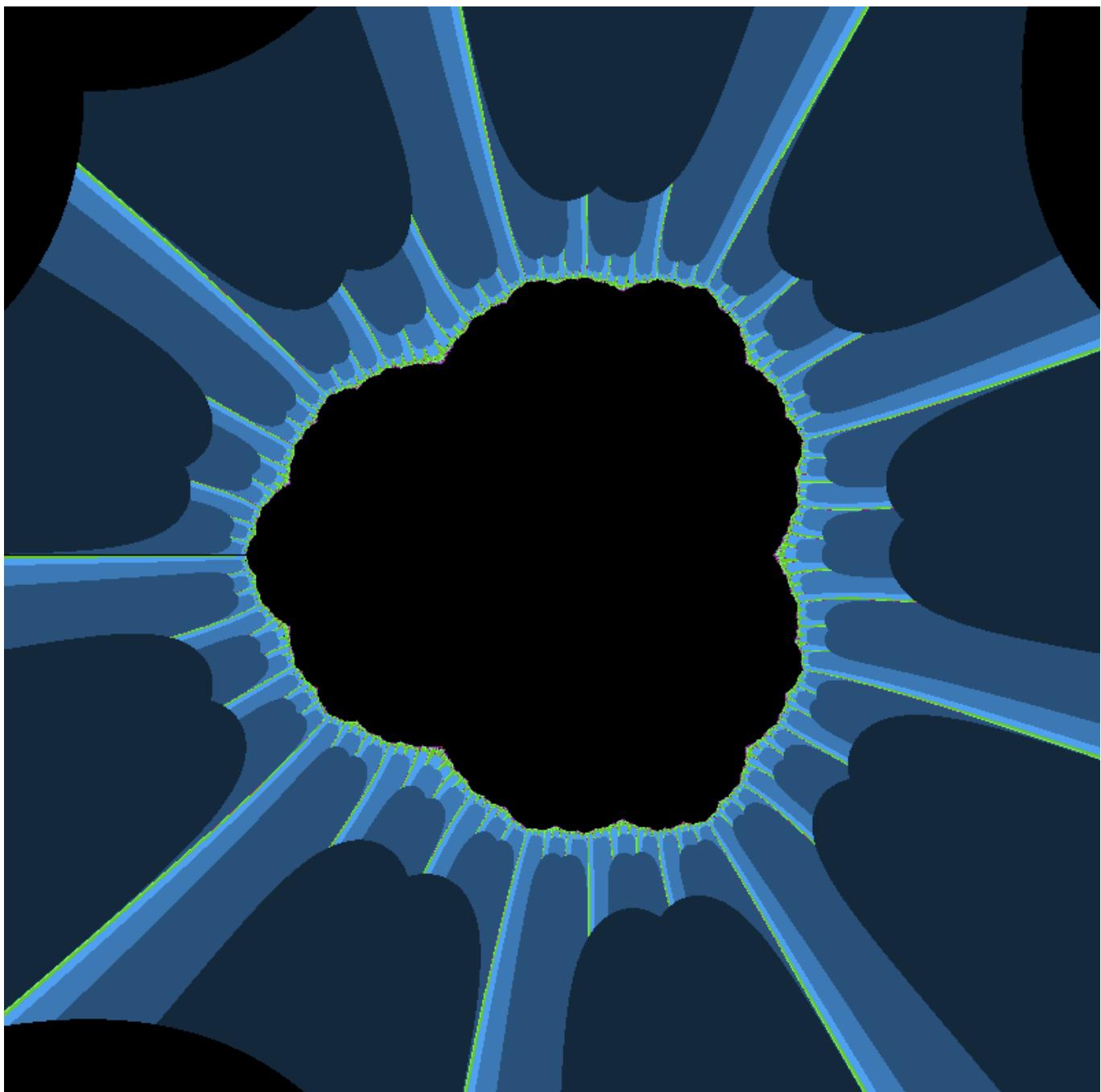
A működési elve ugyan az, mint az előző feladatban leírtaké, hiszen ez is Mandelbrot-halmaz. A különbség, hogy itt a komplex osztálytalálkozás dolgozik a program, amelyet már az include-ban is megjelenítünk. A while ciklusban leírt képlet alapján számol a program, valamint a színezést ezután végzi el. A fordítása: g++ mandel2.cpp -lpng -O3 -o mandel2 . Az #include "png++/png.hpp" miatt van szükség a -lpng kapcsolóra a fordításnál. A -O3 csak kényelmi célt szolgál, gyorsítja a folyamatot. Futtatás a programban kommentekként megadott formában. Például: ./mandel2 mandel2.png 1920 1080 1020 0.4127655418209589255340574709407519549131 0.4127655418245818053080142817634623497725 0.21353 0.2135387051804975289126531379224616102874 Ezekkel adjuk meg a programnak a "határokat", tehát a kép méretét, az iterációt, valamint azt a halmazt, amin ábrázoljuk a képet. Amint elindítjuk a futtatást megjelenik a terminálban a "Számítás" felirat, amely kiírja nekünk, hogy éppen hány %-on áll a kép létrehozása/feldolgozása. Majd ha kész, kiírja, hogy mentve.

## 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)

Tanulságok, tapasztalatok, magyarázat...



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

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;

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: ./bmorf fajlnev szelesseg magassag n a b c ←
 d reC imC R" << std::endl;
 return -1;
}

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";

// j megy a sorokon
for (int y = 0; y < magassag; ++y)
{
 // k megy az oszlopokon

 for (int x = 0; x < szelesseg; ++x)

 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));
}

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;
}
```

A Mandelbrot-halmaz tartalmazza az összes Julia halmazt. Az utóbbiból nagyon sok féle egymástól különböző létezik, és még a számítógépekkel sem lehet bármennyire nagyítani, korlátokba ütközünk, még a Mandelbrot halmazból csak egyet ismerünk. A fordítása: g++ bmorf.cpp -lpng -O3 -o bmorf és a futtatása szintén például /bmorf bmorf.png 1920 1080 1020 0.4127655418209589255340574709407519549131 0.4127655418245818053080142817634623497725 0.2135387051768746491386963270997512154281 0.21353 mert ugyan azokat a paramétereket kell megadnunk a programnak, mint az előző feladatban is. Ha nem ilyen formában próbáljuk futtatni, akkor hibaüzenetet fog dobni a program, ami az első if else ágában van megfogalmazva. Ekkor a return -1 miatt kilép a program. A számítás közben szintén kiírja a program %-ban (ezért kellett az int szazalek változó a programkód végén), hogy hol jár, illetve ha végzett, akkor azt, hogy mentve. Számítás közben sor, oszlop szerint halad, ezt írja le a 2 for ciklus, majd a matematikai számítások után kiszínezi a képet.

## 5.4. A Mandelbrot halmaz CUDA megvalósítása

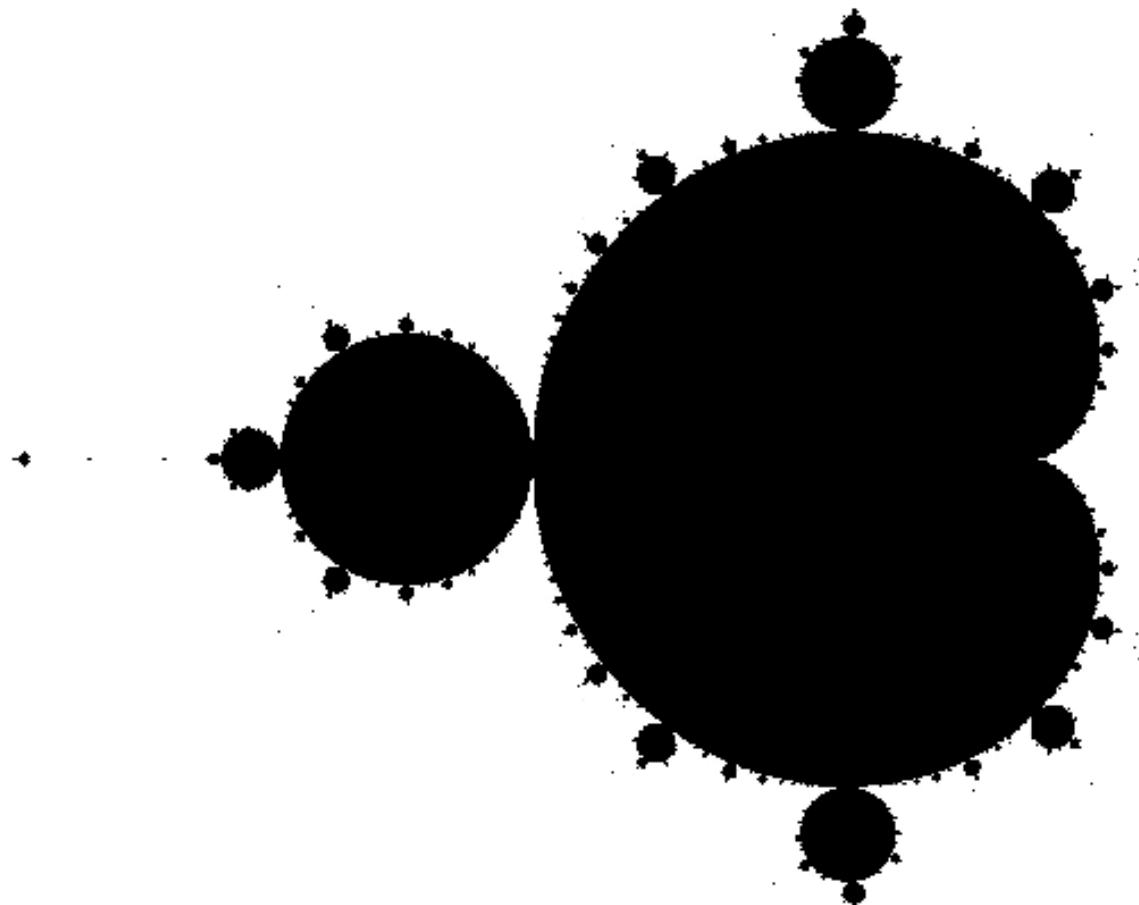
Megoldás videó:

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

A CUDA-s Mandelbrot halmaz az alapelveken megegyezik az előző Mandelbrot-halmazokkal, a számolása és a rajzolási elve szintén ugyan az. Ami a különbséget jelenti, az a futása, vagyis a képkotási módszere hardveresen. Még az eddigiek a processzor segítségével jöttek létre, és képpontról képpontra rajzolódtak meg, erre kellettek a for ciklusok, addig a CUDA-val a videókártya készíti el a képet. A sima processzoros módszert is lehetne párhuzamosítani, hogy egyszerre rajzolja a pontokat, de ez még mindig a

lassabb megoldás lenne, mert a CUDA-val a videókártya gyorsabban képes elvégezni a grafikai számításokat, ezért az sokkal gyorsabb. Tehát a CUDA párhuzamosítva rajzol, minden képpontot egy időben próbál meg elkészíteni. A parancs amit használtam: nvcc mandelpngc\_60x60\_100.cu -o mandelpngc -lpng ; g++ mandelpngt.c++ -o mandelpngt -lpng , majd a kép megalkotása: ./mandelpngt mandel.png .

A kép amelyet nekem a program készített:



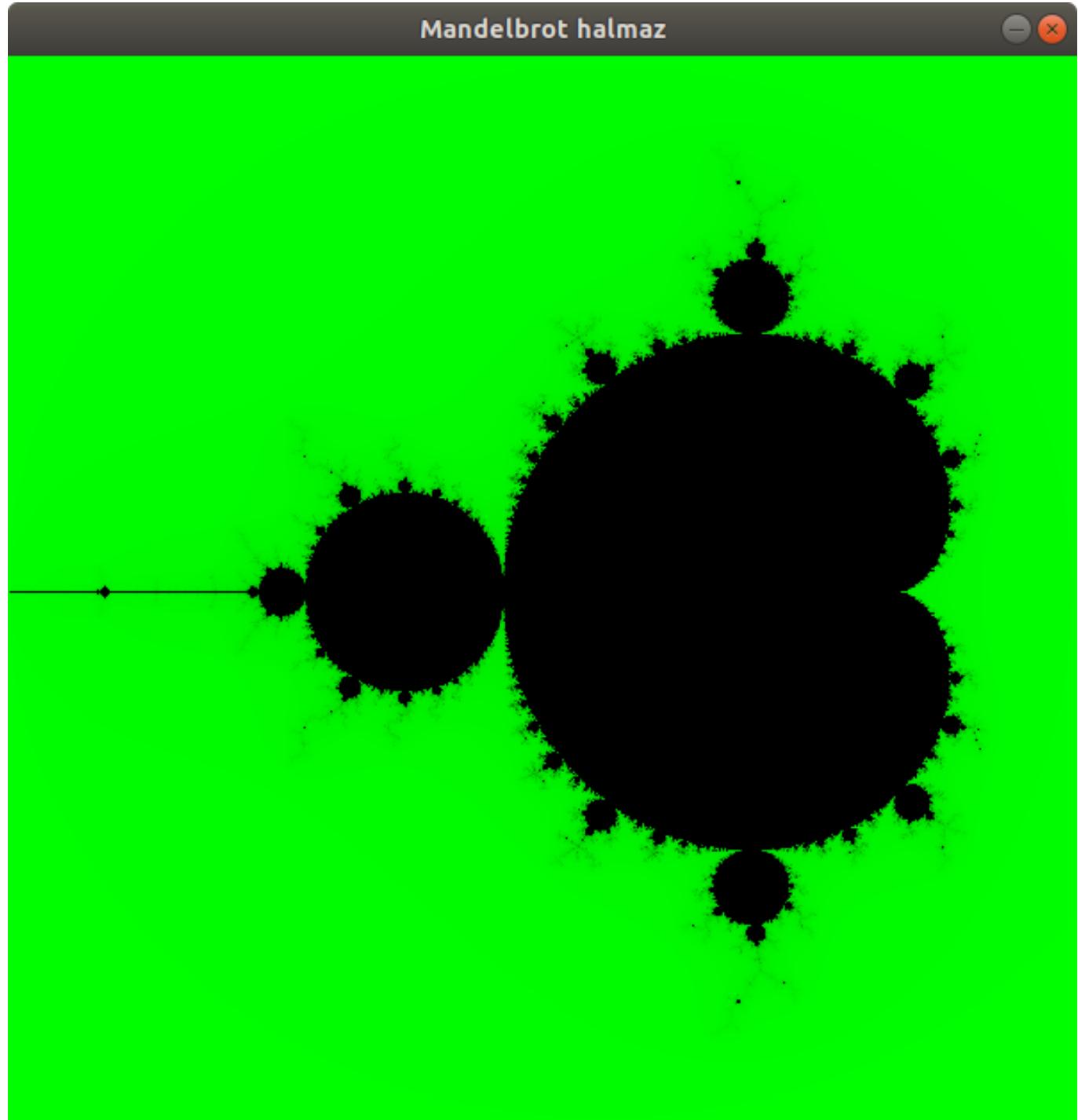
## 5.5. Mandelbrot nagyító és utazó C++ nyelven

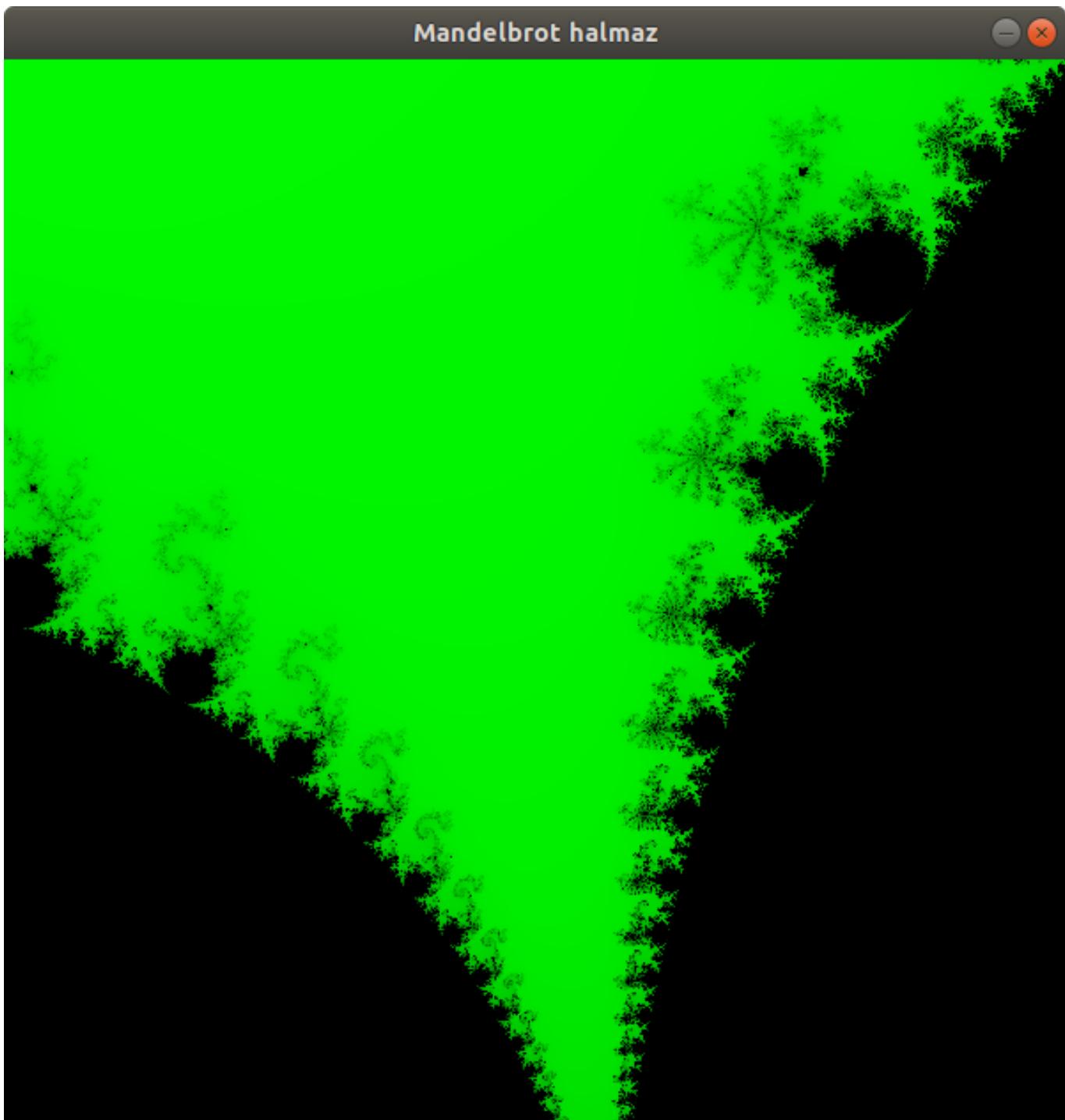
Építs GUI-t a Mandelbrot algoritmusra, lehessen egérrel nagyítani egy területet, illetve egy pontot egérrel kiválasztva vizualizálja onnan a komplex iteráció bejárta  $z_n$  komplex számokat!

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/)

Megoldás videó:

Tanulságok, tapasztalatok...





Ehhez a feladathoz több forrás fájlra van szükségünk, nem elég egy, és már szükségünk vannak a fraktálokra. A fraktálokat végtelen komplex matematikai, vagy geometriai alakzatoknak nevezzük, amelyekben matematikai ismétlődéseket fedezhetünk fel, egyfajta "szabályozást vagy szabályrendszert". A Mandelbrot-halmaz kiváló példa rájuk. Ha futtatjuk a programunkat, akkor ezzel a kiindulóképpel találjuk szemben magunkat. A nagyítónak és utazónak pedig ez a lényege, hogy ha egérrel kijelölünk egy területet, akkor azt nagyítási területként kezelje a program. Az include-ok miatt (pl: QMainWindow), szükségünk van egy plusz programra (Qt), amit telepítenünk kell ahoz, hogy fordíthassuk a programunkat. Miután a telepítéssel végeztünk, megvan a Makefile és lefordítottuk a programunkat, rögtön tudjuk futtatni a szokott módon, ./valami . A felhasznált fájlok: frakablak.cpp, frakablak.h, frakszal.cpp, frakszal.h, main.cpp és frak.pro Miután telepítettük a Qt programot, a fordítása így nézett ki: /home/user/Qt/5.12.2/gcc\_64/bin/qmake frak.pro

, ekkor elkészíti a Makefile-t. Ezután ki kell adni a make parancsot, amivel futtathatóvá válik a program. Innentől a futtatás ./frak .A program a nagyítást az egér lenyomásának koordinátáiból és a gomb felengedésének helyéből számolja, a szélesség és magasság segítségével. A fraksal.cpp felelős a számításokért és a halmazért, a frakablak.cpp pedig a nagyítást végzi el. Megnövelhetjük a képen az iterációs határt, ha lenyomjuk az n vagy m gpmbot a billentyűzeten, amivel változtatunk a képen valamennyit.

## 5.6. Mandelbrot nagyító és utazó Java nyelven

A Java-s nagyító működésében szinte ugyan olyan mint a C++-os nagyító. Ami különbséget jelent az az, hogy nem kell Qt, és egy program az egész, nem több különálló darabból áll össze. A kommentek a működés folyamatát írják le, vagyis a program alkotási sorrendjét. Osztályokkal készíti el a halmazt. A kijelölés terület meghatározása történik meg a program elején, valamint annak a területnek a kiszámítása. A kijelölés után újraszámolja a program a kijelölt területet, és újrarajzolja azt, valamint pillanatképek készítésével és megjelenítésével végzi el a nagyítást. A pillanatképeket menthetjük is, a név a tartománytól függ (a/b/c/d). Az n vagy m gomb lenyomásával növelhetjük az iterációs határt, amivel változtatunk kicsit a kép kinézetén. Végül ami a kód végén látható, példányosítást is végez a program. A program fordítása: javac MandelbrotHalmazNagyító.java (az osztály neve miatt a kód nevének is ennek kell lennie) és a futtatása: java MandelbrotHalmazNagyító

```
/*
 * MandelbrotHalmazNagyító.java
 *
 * DIGIT 2005, Javat tanítok
 * Bátfai Norbert, nbatfai@inf.unideb.hu
 *
 */
/**
 * A Mandelbrot halmazt nagyító és kirajzoló osztály.
 *
 * @author Bátfai Norbert, nbatfai@inf.unideb.hu
 * @version 0.0.1
 */
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;
 /**
 * Létrehoz egy a Mandelbrot halmazt a komplex sík
 * [a,b]x[c,d] tartománya felett kiszámoló és nagyítani tudó
 * <code>MandelbrotHalmazNagyító</code> objektumot.
 *
 * @param a a [a,b]x[c,d] tartomány a koordinátája.
 * @param b a [a,b]x[c,d] tartomány b koordinátája.
 * @param c a [a,b]x[c,d] tartomány c koordinátája.
 * @param d a [a,b]x[c,d] tartomány d koordinátája.
 * @param szélesség a halmazt tartalmazó tömb szélessége.
 * @param iterációsHatár a számítás pontossága.

```

```
/*
public MandelbrotHalmazNagyító(double a, double b, double c, double d,
 int szélesség, int iterációsHatár) {
 // Az ōs osztály konstruktorának hívása
 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();
 }
 // Vonszolva kijelölünk egy területet...
 // Ha felengedjük, akkor a kijelölt terület
 // újraszámítása indul:
 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();
 }
 });
}
/**
```

```
* Pillanatfelvételek készítése.
*/
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("MandelbrotHalmazNagyítás_");
 sb.append(++pillanatfelvételszámláló);
 sb.append("_");
 // A fájl nevébe belelevesszük, 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();
 }
}
/**
 * A nagyítandó kijelölt területet jelző négyzet kirajzolása.
 */
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);
}
/**
 * Példányosít egy Mandelbrot halmazt nagyító obektumot.
 */
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);
}
```

## 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:

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

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!

Az első kód a polártranszformációs algoritmus C++-ban. Az algoritmus az OO-n, vagyis az is objektum orientált programozásban alapul. A PolarGen osztályt hozza létre a polargen.h fájlban, majd a polargen.cpp-ben ezt hivatkozza be include formájában. Találhatunk benne konstruktort és destruktort, a destruktur pedig miután már nem kell törli az osztályt, és felszabadítja a memóriát. A cpp fájlon belül pedig számításokat végez a program, amelyekhez random számokat is használ, illetve a tárolt adatokat vizsgálja, amelyekhez külön változót vezetett be. A számításokat addig végzi el, ameddig a w nagyobb mint 1, a while ciklus ezt írja le. Addig fog futni ameddig nem lesz tárolt adata. Ha van akkor kilép, ha nincs, akkor újra hívja (rekurzívan) az osztályt és újravizsgálja.

```
#include "polargen.h"

double
PolarGen::kovetkezo ()
{
 if (nincsTarolt)
 {
 double u1, u2, v1, v2, w;
 do
 {
 u1 = std::rand () / (RAND_MAX + 1.0);
 u2 = std::rand () / (RAND_MAX + 1.0);
 v1 = 2 * u1 - 1;
```

```
v2 = 2 * u2 - 1;
w = v1 * v1 + v2 * v2;
}
while (w > 1);

double r = std::sqrt ((-2 * std::log (w)) / w);

tarolt = r * v2;
nincsTarolt = !nincsTarolt;

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

A második kód a polártranszformációs algoritmus Java-ban. Jelentősége van annak, hogy páros vagy páratlan alkalmmal hívtuk meg a függvényt. A páratlanoknál nem kell számolni, csak az előző lépés számát kell visszaadni. A szükséges számításokat a következő() függvény végzi el, amelyet a nincsTárolt logikai változó jelöl. Ha igaz az érték, akkor a tárolt l.pontos változóban van eltárolva az a szám, amelyet vissza kell adni. Úgy működik mint a c++-os verzió, de itt a program 10 tárolt adatig megy.

```
public class PolarGen {

 boolean nincsTarolt = true;
 double tarolt;

 public PolarGen () {

 nincsTarolt = true;
 }

 public double kovetkezo () {

 if (nincsTarolt) {

 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);

 tarolt = r * v2;
 nincsTarolt = !nincsTarolt;

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

}

public static void main (String[] args) {
 PolarGen pg = new PolarGen();

 for (int i = 0; i < 10; ++i) {
 System.out.println(pg.kovetkezo());
 }

}
```

## 6.2. LZW

Tutorált (am volt): Egyed Anna, Ranyhóczki Mariann

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

Megoldás videó:

Megoldás forrása: tankonyvtar.hu, BHAX csatorna forrásai

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

typedef struct binfa
{
 int ertek;
 struct binfa *bal_nulla;
 struct binfa *jobb_egy;
} BINFA, *BINFA_PTR;
```

```
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 szabadit (BINFA_PTR elem);

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

 BINFA_PTR gyoker = uj_elem ();
 gyoker->ertek = '/';
 BINFA_PTR fa = gyoker;

 while (read (0, (void *) &b, 1))
 {
 write (1, &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);
extern int max_melyseg;
printf ("melyseg=%d", max_melyseg);
szabadit (gyoker);
}

static int melyseg = 0;
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);
 kiir (elem->bal nulla);
 --melyseg;
 }
}

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

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog1/6.Welch
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog1/6.Welch$./binsima <valami.txt
000111011100001111101010101100100010010001001010101010111111100100010001000101010101010111101010001
1110101110001
00111010101

-----1(6)
-----1(5)
-----1(4)
-----1(6)
-----0(5)
-----1(3)
-----0(4)
-----0(5)
-----1(2)
-----1(5)
-----0(6)
-----1(4)
-----1(7)
-----1(6)
-----0(5)
-----0(3)
-----1(5)
-----0(4)
-----0(5)
---/(1)
-----1(6)
-----1(5)
-----1(4)
-----1(3)
-----1(6)
-----1(5)
-----0(6)
-----0(4)
-----0(2)
-----1(4)
-----1(6)
-----0(5)
-----1(7)
-----0(6)
-----0(7)
-----0(3)
-----1(5)
-----0(6)
-----0(4)
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog1/6.Welch$
```

Ez a binfa inorder bejárású, tehát ha ábrázoljuk, akkor a gyökér középen helyezkedik el, a bal ága 0, a jobb ága pedig 1. A jobb és bal oldali szám elosztások a továbbiakban is ugyan így megy, a léptetések után. Az input szerint a már meglévő számokat léptetjük a fánk ágain, majd ha új számot találunk, ami nem léptethető már tovább, akkor beszűrjuk az új számot, majd visszaállítjuk a mutatót a gyökérre, és indulunk újra a lépegetéssel. Ha új elemet szúrtunk be, akkor azoknak a jobb és bal értékét NULL-ra állítjuk, ebből tudjuk, hogy az az utolsó, tehát a levélelem. Mielőtt beszűrünk egy új elemet, foglalunk neki memóriát, és utána töltjük bele az adatot. Függvények segítségével dolgozzuk fel a beszúrási módöt, amelynek a működési elvét az előbb írtam le. A kiír függvényben előbb a jobb egyes elemet, aztán a gyökeret, végül a bal nullás elemet írjuk ki. A for ciklus a léptetésért felelős, illetve a szintekért. A futtatáskor láthatjuk a 0 vagy egyes mellett, hogy mennyire vagyunk "mélyen", tehát hanyadik szintje a fának. Ennek a számolására szolgál a mélység változó, és a kiír függvényben minden egyes szintnél a ++melyseg-gel növeljük a számát, megadva ezzel a pontos értéket. A --melyseg pedig a kiir függvény végén történő visszaállítás az eredeti értékre, így amikor újra meghívódik továbbra sem fog hibás értéket visszaadni. Fordítása a megszokott módon történik, gcc binfa.c -o binfa .

### 6.3. Fabejárás

Tutor (om volt): Egyed Anna

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

Megoldás videó:

Megoldás forrása:

A preorder bejárásbeli különbség a kiíratásban: Az inorderhez képest a kiíratás változik meg. A preorder bejárás "elöl" történő bejárást jelent, tehát a gyökérrel kezdünk, és utána építjük fel a kiíratást. Gyökér, jobb elem, majd bal elem. A sorrend lényeges. Az elem != NULL kifejezéssel megvizsgálja az adott levélelementet, hogy mi a végső értéke. Ha az NULL, akkor levélelem, nem pedig ág, tehát ott végződik az adott ág.

```
void
kiir (BINFA_PTR elem)
{
 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->jobb_egy);

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

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
melyseg=7petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/6.Welch$./pre <valami.txt
000111011100001111101010110010001001000100010101101010111111100100010001000101010101010111101010010
1110101110001
00111010101

----/(1)
-----1(2)
-----1(3)
-----1(4)
-----1(5)
-----1(6)
-----0(5)
-----1(6)
-----0(4)
-----0(5)
-----0(3)
-----1(4)
-----1(5)
-----0(6)
-----0(5)
-----1(6)
-----1(7)
-----0(4)
-----1(5)
-----0(5)
-----0(2)
-----1(3)
-----1(4)
-----1(5)
-----1(6)
-----0(4)
-----1(5)
-----1(6)
-----0(6)
-----0(3)
-----1(4)
-----0(5)
-----1(6)
-----0(6)
-----1(7)
-----0(7)
-----0(4)
-----1(5)
-----0(6)
melyseg=7petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/6.Welch$
```

A posztorder kiírásbeli eltérés az inorderhez képest: Szintén a kiíratásban történik csak változás, az elosztási elve ugyan az, mint eddig. A posztorder bejárás "hátulról" történő bejárást jelent. Tehát kiíratjuk a jobb elemet, a balt és végül a gyökeret. Rekurzív függvény, mivel önmagát hívja meg futás közben.

```
void
kiir (BINFA_PTR elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 if (melyseg > max_melyseg)
 max_melyseg = melyseg;
 kiir (elem->jobb_egy);
 kiir (elem->bal nulla);
 for (int i = 0; i < melyseg; ++i)
 printf ("---");

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

}

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/6.Welch$./poszt <valami.txt
00011101110000111110101011011001000100100010001010101010111111100100100001001000101010101011100101010111101010010
1110101110001
00111010101

-----1(6)
-----1(5)
-----1(6)
----0(5)
---1(4)
---0(5)
---0(4)
--1(3)
---0(6)
---1(5)
---1(7)
---1(6)
---0(5)
---1(4)
---1(5)
---0(5)
---0(4)
---0(3)
---1(2)
-----1(6)
-----1(5)
---1(4)
-----1(6)
-----0(6)
-----1(5)
---0(4)
---1(3)
-----1(6)
-----1(7)
---0(7)
---0(6)
---0(5)
---1(4)
-----0(6)
-----1(5)
---0(4)
---0(3)
---0(2)
---/(1)
melyseg=7petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog1/6.Welch$
```

## 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! (Helyette \*gyökér)

## Megoldás videó:

## Megoldás forrása: BHAX csatorna

Tutor (om volt): Tóth Attila

Ez a fajta megoldás komplikáltabb, mint a struktúrás megoldás, de cserébe rögzíthetünk adatokat és változók értékei sem lesznek módosíthatóak az osztálykezelésnek köszönhetően. A private rész szolgál erre főként. A kész függvényeket megírhatjuk az osztályon belül, és a beágyazott osztállyal, illetve azokon kívül is használhatjuk a függvényeinket. A private részben megírt függvények csak ugyan annak az osztálynak a public részében lehet használni, ott tudjuk meghívni, máshol nem. Az adatvédelem szempontjából praktikus ez. Jelen esetben osztályba ágyazott osztállyal dolgozunk, a Binfa osztály private részében helyezik el a Csomópont osztályt. Ezekben az osztályokon belül készülnek el a függvények (pl kiíratás vagy feldolgozás). Például a kiir függvény felelős a kiíratásért, ebben meghatározzuk a kiíratás módját (pre-, in- vagy

posztorder) és ennek az argumentuma a gyökér. A végén szintén használunk destruktort, tehát ha már nincs szükség az adott memoriában tárolt dolgokra, akkor törlődik, itt az osztályt töröljük vele. Az osztályon belül lokális, azon kívül globális változókat hozunk létre. A programban van egy usage rész, amit futtatás-kor láthatunk, ha rosszul próbáljuk futtatni és ekkor kiírja a program, hogy hogyan írjuk be helyesen. Fájl beolvasással is képes vizsálni a számokat, majd a végzett, fájlba kiírja az eredményt. Ez a programkód jó a többi feladathoz is. Csomópontokkal jelöli meg a pontokat. A mutatók állításával követi nyomon a bemenő input adatok feldolgozását, minden az adott egyes vagy nullás gyermekre, vagy éppen csomópontra állítja. Ha új tag, vagyis gyermek került beírásra, akkor a mutató visszaáll a gyökérre.

```
#include <iostream>
#include <cmath>
#include <fstream>

class LZWBinFa
{
public:

 LZWBinFa ()
 {
 gyoker = new Csomopont();
 fa=gyoker;
 }

 ~LZWBinFa ()
 {
 szabadit (gyoker->egyesGyermek ());
 szabadit (gyoker->>nullasGyermek ());
 delete gyoker;
 }

 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);
}

private:
 class Csomopont
 {
 public:

 Csomopont (char b = '/') : betu (b), balNulla (0), jobbEgy (0)
 {
 };
 ~Csomopont ()
```

```
{
};

Csomopont *nullasGyermek () const
{
 return balNulla;
}

Csomopont *egyesGyermek () const
{
 return jobbEgy;
}

void ujNullasGyermek (Csomopont * gy)
{
 balNulla = gy;
}

void ujEgyesGyermek (Csomopont * gy)
{
 jobbEgy = gy;
}

char getBetu () const
{
 return betu;
}

private:

 char betu;

 Csomopont *balNulla;
 Csomopont *jobbEgy;

 Csomopont (const Csomopont &);
 Csomopont & operator= (const Csomopont &);
};

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 (Csomopont * elem)
{
 if (elem != NULL)
 {
 szabadit (elem->egyesGyermek ());
 szabadit (elem->>nullasGyermek ());
 delete elem;
 }
}

protected:
 Csomopont *gyoker;
 int maxMelyseg;
 double atlag, szoras;

 void rmelyseg (Csomopont * elem);
 void ratlag (Csomopont * elem);
 void rszoras (Csomopont * elem);

};

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 (Csmopont * 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;
}

int
main (int argc, char *argv[])
{
 try{

 if (argc != 5)
 {

 usage ();
 throw std::invalid_argument ("arg");
 return -1;
 }

 char *inFile = argv[1];

 if (argv[2][1] != 'o')
 {
 usage ();
 throw std::ios::failure ("Hibás bemenet");
 return -2;
 }
 }
}
```

```
}

std::fstream beFile (inFile, std::ios_base::in);

if (!beFile)
{
 std::cout << inFile << " nem letezik..." << std::endl;
 usage ();
 throw std::ios::failure("Hibás bemenet");
 return -3;
}

std::fstream kiFile (argv[3], std::ios_base::out);

unsigned char b;
LZWBinFa binFa;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
 if (b == 0x0a)
 break;

bool kommentben = false;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
{

 if (b == 0x3e)
 {
 kommentben = true;
 continue;
 }

 if (b == 0x0a)
 {
 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;
 }

}

if(argv[4][0]=='f') {
kiFile << binFa;

kiFile << "depth = " << binFa.getMelyseg () << std::endl;
kiFile << "mean = " << binFa.getAtlag () << std::endl;
kiFile << "var = " << binFa.getSzoras () << std::endl;
}
else if(argv[4][0]=='c')
{std::cout<< binFa;

std::cout << "depth = " << binFa.getMelyseg () << std::endl;
std::cout << "mean = " << binFa.getAtlag () << std::endl;
std::cout << "var = " << binFa.getSzoras () << std::endl;
}
kiFile.close ();
beFile.close ();

return 0;
}
catch (std::invalid_argument& e) {
 std::cout << "Hiba történt: ";
 std::cout << e.what() << std::endl;
}
catch (std::ios::failure& e) {
 std::cout << "Hiba történt: ";
 std::cout << e.what() << std::endl;
}
```

## 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: BHAX csatorna forrásai

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <algorithm>
#include <utility>

class LZWBinFa
{
public:
 LZWBinFa()
 {
 gyoker = new Csomopont();
 fa=gyoker;
 }
 ~LZWBinFa ()
 {
 szabadit (gyoker->egyesGyermek ());
 szabadit (gyoker->>nullasGyermek ());
 delete gyoker;
 }

 LZWBinFa (LZWBinFa && regi) {

 gyoker = nullptr;
 *this = std::move(regi);

 }

 LZWBinFa & operator= (LZWBinFa && regi) {

 std::swap(gyoker, regi.gyoker);

 return *this;
 }

 void operator<< (char b)
 {

 if (b == '0')
 {

 if (!fa->nullasGyermek ())
 {
 Csomopont *uj = new Csomopont ('0');
 fa->egyesGyermek = uj;
 }
 }
 }
}
```

```
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);
}

private:
 class Csomopont
 {
public:
```

```
Csomopont (char b = '/') : betu (b), balNulla (0), jobbEgy (0)
{
};

~Csonopont ()
{
};

Csonopont (const Csonopont& elem) {

 betu = elem.getBetu();
 balNulla = new Csonopont;
 jobbEgy = new Csonopont;
 *balNulla= *(elem.nullasGyermek());
 *jobbEgy= *(elem.egyesGyermek());
}

Csonopont & operator= (const Csonopont& elem) {

 betu = elem.getBetu();
 Csonopont* ujBal = new Csonopont();
 *ujBal = *(elem.nullasGyermek());
 delete balNulla;
 balNulla = ujBal;
 Csonopont* ujJobb = new Csonopont();
 *ujJobb = *(elem.egyesGyermek());
 delete jobbEgy;
 jobbEgy = ujJobb;

 return *this;
}

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;
 Csomopont *balNulla;
 Csomopont *jobbEgy;

};

Csomopont *fa;
int melyseg, atlagosszeg, atlagdb;
double szorasosszeg;
LZWBinFa (const LZWBinFa& binfa);

void kiir (Csomopont * elem, std::ostream & os)
{

 if (elem != NULL)
 {
 ++melyseg;
 kiir (elem->nullasGyermek (), os);
 for (int i = 0; i < melyseg; ++i)
 os << "---";
 os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
 kiir (elem->egyesGyermek (), os);
 --melyseg;
 }
}

void szabadit (Csmopont * elem)
{
 if (elem != NULL)
 {
 szabadit (elem->egyesGyermek ());
 szabadit (elem->nullasGyermek ());
 delete elem;
 }
}

protected:
 Csmopont *gyoker;
 int maxMelyseg;
 double atlag, szoras;

 void rmelyseg (Csmopont * elem);
 void ratlag (Csmopont * elem);
```

```
void rszoras (Csomopont * elem);

};

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 (Csmopont * 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 (Csmopont * elem)
{
 if (elem != NULL)
 {
 ++melyseg;
 rszoras (elem->egyesGyermek ());
 rszoras (elem->>nullasGyermek ());
 --melyseg;
 if (elem->egyesGyermek () == NULL && elem->>nullasGyermek () == NULL)
 {
 ++atlagdb;
 szorasossszeg += ((melyseg - atlag) * (melyseg - atlag));
 }
 }
}

void
usage (void)
{
 std::cout << "Usage: lzwtree in_file -o out_file" << std::endl;
}

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,binFa2;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
 if (b == 0x0a)
 break;

bool kommentben = false;

while (beFile.read ((char *) &b, sizeof (unsigned char)))
{
 if (b == 0x3e)
 {
 kommentben = true;
 continue;
 }

 if (b == 0x0a)
 {
 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;
binFa2=std::move(binFa);
kiFile<<"\n Mozgatás után binFa:"<< std::endl;
kiFile << binFa;
kiFile << "depth = " << binFa.getMelyseg () << std::endl;
kiFile << "mean = " << binFa.getAtlag () << std::endl;
kiFile << "var = " << binFa.getSzoras () << std::endl;
kiFile << "\nMozgatás után a binFa2"<< std::endl;
kiFile<<binFa2;
kiFile << "depth = " << binFa2.getMelyseg () << std::endl;
kiFile << "mean = " << binFa2.getAtlag () << std::endl;
kiFile << "var = " << binFa2.getSzoras () << std::endl;

kiFile.close ();
beFile.close ();

 return 0;
}
```

Az előzőhöz képest a csomópont működésében van nagyobb különbség, hiszen ez innentől pointer lesz. A gyökér átkerül a konstruktora. A működése nem változik az eddigiekhez képest, tehát nagy vonalakban: Ezzel a programmal képesek vagyunk fájlból kiolvasni adatot, és fájlba írni az eredményt, a használata benne van a programkódban: "Usage: lzwtree in\_file -o out\_file", tehát ha rosszul próbáljuk meg használni, ezt az üzenetet kapjuk. A számolások, és az új elemek beillesztése változás nélkül működik. Ameddig tud lépkedni a meglévő ágakon addig az input alapján lépked, aztán beszúrja az új elemet, a mutató pedig visszaáll a gyökérre. Az input vizsgálatát a beszúrt elemtől folytatja. A csomópont innentől kezdve pointer-ként funkcionál. A destrukturátor mellé most szükség van egy delete parancsra is, hogy teljesen felszabadítsuk a memóriát.

## 6.6. Mozgató szemantika

Tutoriált (am volt): Ignéczi Tibor, Egyed Anna

Írj az előző programhoz mozgató konstruktort és értékkadást, a mozgató konstruktor legyen a mozgató értékkadásra alapozva!

Megoldás videó:

Megoldás forrása:

A kód a 6.5-ösben, a mutató a gyökér feladatban benne van. A binfa működését és részleteit nem írnám le újból. Első lépésként megvizsgálja a megadott bemenetet, hogy az helyes e, és amennyiben nem, hibaüzenetet ad. Ha ez megfelelő, utána a fájlok tartalmát vizsgálja meg. Helyes paraméterek esetén pedig megtörténik a memória foglalás, másolás, majd a mozgatással történő másik fájlba való kiírás. Mozgató konstruktorral végezzük ezt, majd meghívuk a mozgatott objektumokra szintén a kiírató függvényt. Amennyiben a mozgatott fa csomópontjainak címei megegyeznek az első fáéval, akkor sikeresen elvégeztük a feladatunkat, és a program is helyesen működik. Tehát a mozgató konstruktornál a gyökér címe megváltozik, amíg az ágak és levelek címei nem. A move-val pedig még nem végzi el teljesen a mozgatást, hanem a

```
*this = std::move(regi)
```

hajta végre a feladatot. Ezután kiíratásnál láthatjuk az eredményt, hogy a mozgatás után az eredeti binfa eltűnt, és csak a mozgatott binfa2 maradt meg.

```
out.txt
- /K/Prog1/6.Welch
Megnyitás ▾ 0
Mentés ▾
-----0(5)
-----0(4)
-----0(3)
-----0(5)
-----1(4)
-----1(5)
-----0(2)
-----0(5)
-----0(4)
-----1(3)
-----0(5)
-----1(4)
-----0(1)
-----0(3)
-----1(2)
-----0(7)
-----0(6)
-----0(5)
-----0(4)
-----1(3)
-----0(2)
-----0(5)
-----0(6)
-----0(7)
-----0(6)
-----0(5)
-----0(4)
-----1(3)
depth = 7
mean = 4.33333
var = 1.65631

Mozgatás után binFa:
--- /()
depth = 0
mean = 0
var = 0

Mozgatás után a binFaz
-----0(5)
-----0(4)
-----0(3)
-----0(5)
-----1(4)
-----1(5)
-----0(2)
-----0(5)
-----0(4)
-----1(3)
-----0(5)
-----1(4)
-----0(1)
-----0(3)
-----1(2)
-----0(7)
-----0(6)
-----0(5)
-----0(4)
-----1(3)
```

```
Megnyitás ⌂ out.txt -/lk/Program/ő.welch Mentés ⌂ ⌂ ⌂ ⌂
-----0(5)
-----0(4)
-----1(3)
-----0(5)
-----1(4)
-----0(1)
-----0(3)
-----1(2)
-----0(7)
-----0(6)
-----0(5)
-----0(4)
-----1(3)
---/(0)
-----0(2)
-----1(1)
-----1(2)
depth = 7
mean = 4.33333
var = 1.65831
Mozgatás után binFa:
---/(0)
depth = 0
mean = 0
var = 0
Mozgatás után a binFa2
-----0(5)
-----0(4)
-----0(3)
-----0(5)
-----1(4)
-----1(5)
-----0(2)
-----0(5)
-----0(4)
-----1(3)
-----0(5)
-----1(4)
-----0(1)
-----0(3)
-----1(2)
-----0(7)
-----0(6)
-----0(5)
-----0(4)
-----1(3)
---/(0)
-----0(2)
-----1(1)
-----1(2)
depth = 7
mean = 4.33333
var = 1.65831
Egyesítési szöveg ▾ Tabulátorszélesség: 8 ▾ 1. sor, 1. oszlop ▾ BESZ
```

## 7. fejezet

# Helló, Conway!

### 7.1. Hangyszimulációk

Tutorált (am volt): Ignéczi Tibor

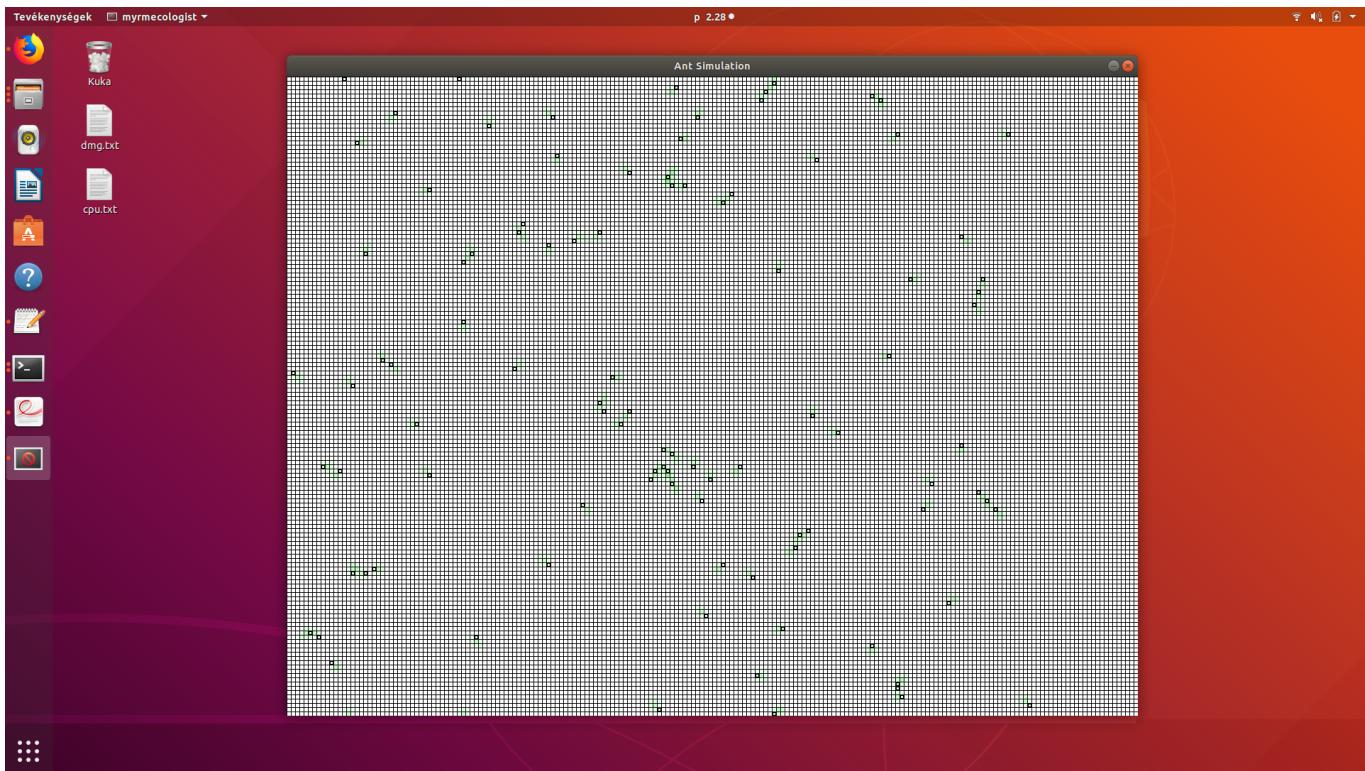
Írj Qt C++-ban egy hangyszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

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

Tanulságok, tapasztalatok, magyarázat...

A program futtatásakor apró pontok, vagyis hangyák kezdenek el véletlenszerűen mozogni a képen. Ez a program is Qt segítségével működik. A program verzióra ügyelni kell, a túl régiekkel nem fog működni. A programban több osztály is deklarálva van. Az Ant osztályban a hangyák vannak meghatározva, amelyek 3 tulajdonséggel rendeznek, 2 db koordináta pont, ami a pozíciót jelöli és még egy adat, ami az irányt adja meg. Ez az osztály többszörösítve van a programban, így lesz 1 hangyból egy egész hangyboly. A következő osztály, vagyis az AntWin, a hangyák megrajzolásáért felel. Ez felel a billentyűre történő progrsam megállításért, szüneteltetésért, és a programból történő kilépésért, a szimuláció megszakításáért. Az utolsó osztály, vagyis az AntThread a hangyák mozgásáért felel, bizonyos feltételek mellett változtatja a hangyák mozgását. A Qt-s fordítás és futtatás a következő: /home/user/QT/5.12.2/gcc\_64/bin/qmake myrmecologist.pro , ezután make parancs, majd a futtatás ./myrmecologist .



## 7.2. Java életjáték

Írd meg Java-ban a John Horton Conway-féle életjátékot, valósítsa meg a sikló-kilövőt!

Megoldás videó:

Megoldás forrása: <https://www.tankonyvtar.hu/en/tartalom/tkt/javat-tanitok-javat/apb.html?fbclid=IwAR3DPY0T>

Tanulságok, tapasztalatok, magyarázat...

A lényege ugyan az, mint ahogyan a következő feladatban le van írva. Az élet-halál szabály az időFejlődés() függvényben van meghatározva. A működése a kommentekben végig le van vezetve. Meg van fogalmava a kódban az életjáték szabályai, a kirajzolása, a sejtek, siklók és a siklókilövők megalkotása, az egérmozgás és kattintás feldolgozása, valamint a billentyűzetről érkező parancsok esetében is. Pillanatfelvételek készítésére is képesek vagyunk, a program is ez alapján dolgozik. Osztálykezeléssel készült el a program. A Java kód Sejtautomata.java néven kell lementenünk, így pedig a fordítása javac Sejtautomata.java, a futtatása pedig java Sejtautomata. Futtatáskor bizonyos billentyűk és az egérmutató használatával befolyásolhatjuk a program működését, "parancsot" adhatunk ki neki. Ezek a billentyűk: az s gomb lenyomásával pillanatképet készíthetünk, az n-nel megnövelhetjük a sejtek méretét, még a k-val csökkenthetjük azokat, a g billentyűvel gyorsíthatunk a program működésén, vagyis a szimulációt, az l-lel pedig lassíthatjuk azt. Tehát ahogyan az előbb írtam, az egérgombokkal is befolyásolhatunk, a jobb és bal gombjával a sejt állapotát változtathatjuk, magyaráról élőből halottat, vagy halottból élőt lehet csinálni. A mutató vonszolásával pedig élő sejteket hozhatunk létre.

## 7.3. Qt C++ életjáték

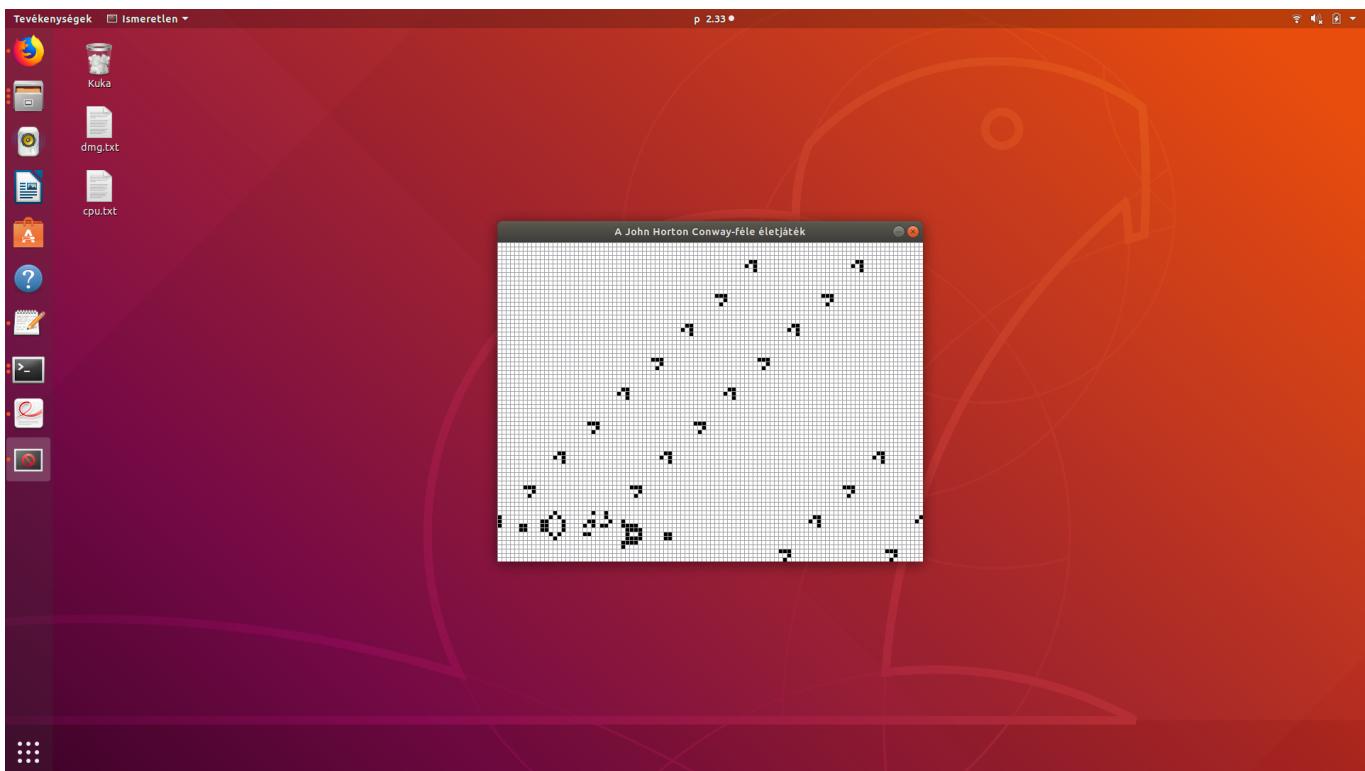
Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: <https://sourceforge.net/p/udprog/code/ci/master/tree/source/kozepes/Qt/sejtautomata/>

Tanulságok, tapasztalatok, magyarázat...

A szabály ami szerint működik az életjáték, az az, hogy van egy cella, amiben van egy sejt. A sejt csak bizonyos körülmények között marad életben, másképp meghal. A sejt csak abban az esetben marad életben, ha van élő szomszédja, és abból is 2-3. A halott sejt pedig halott marad, ha van 3 élő sejt szomszédja, máskülönben élő lesz. Ennek a működési elve az időFejlődés-ben van leírva. Léteznek a programban bizonyos élőlények, amelyeket siklónak nevezünk, ezeket a siklóKilövő indítja el, ami szintén egy osztály. A rácspontok és az élő sejtek segítségével működnek. A hangyaszimulációhoz hasonlóan itt is van egy osztály, ami a kiíratásért felelős, és a feltételek ellenőrzése után rajzol, valamint eldönti hogy életben hagyja vagy megöli a sejtet. A javashoz hasonlóan itt is befolyásolhatjuk a működését a programnak bizonyos billentyűgombok lenyomásával. A Qt életjátékot Qt program segítségével tudjuk fordítani: /home/user/QT/5.12.2/gcc\_64/bin/qmake Sejtauto.pro, majd ezután make parancs kiadásával létrehozzuk a szükséges fájlokat, a futtatása pedig: ./Sejtauto .



## 7.4. BrainB Benchmark

Tutor (om volt: Ranyhóczki Mariann)

Megoldás videó:

Megoldás forrása: <https://github.com/nbatfai/esport-talent-search>

Tanulságok, tapasztalatok, magyarázat...

Itt már van egy kicsi különbség az előző 2 programhoz képest. Egy kirajzoló osztályunk itt is van. A program lényegében az esportolók mérésére szolgál, követni kell egy bizonyos pontot, vagyis a hősünket. A hősünket vagyis a karakterünket egy külön osztály segítségével határozzuk meg a programban. Azt

teszteli, hogy mennyire tudják az emberek a zavaró tényezők ellenére is követni a hősünket. A zavaró tényező például az, hogy plusz karakterek kerülnek be, a program pedig a követni kívánt karaktert is egyre zavaróbb módon mozgatja. A program futása közben adatot gyűjt, amit a futás után, ha bezártuk vagy ha végigvittük a játékot, és ezt eredményként kirakja nekünk egy külön fájlba/mappába, a programkód mellé. Mivel ehhez is szükséges a Qt program, így a fordítás futtatásra ügyelnünk kell. A fordítása: /home/user/QT/5.12.2/gcc\_64/bin/qmake BrainB.pro, ezután make parancs, majd a futtatása ./BrainB .



## 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...

A program képes tanulni, vagyis a tanítás után képesek vagyunk a programmal felismertetni egy kézzel írt számjegyet, amely az MNIST kézzel írt számjegyes adatbázisából származik. Itt még nem kell túlbonyolítni a dolgot, ez egy kisebb példa, mint ahogyan a linken megadott példa, ha megnézzük. Az előbb leírtak alapján, itt rajzolt számok felismerése történik, tehát az adott képet ismeri fel. A program először tanul, majd ezt követően elkezdi tesztelni magát, kiírja a várható pontosságát, végül a megadott képre kiírja, hogy minek ismeri azt fel a hálózat. A program tanulásához van szükség a TensorFlow programra, ez egy könyvtár, amely szoftvereket tartalmaz, amelyek a tanulási és véghajtási szoftvereket foglalják magukba. Ezen kívül a Python3 telepítése is szükséges. A példa szerint a Tanár Úr és Gimp által készített 8-ast is felismeri a program. Ezt azután írja ki eredményül, ha megvizsgálta a képet, és már bezártuk azt. A classification tömb szolgál arra, hogy a könyvtárból kivett képre készített tippet tárolja. A TensorFlow programot a tensorflow hivatalos honlapjáról tudjuk telepíteni.

### 8.2. Mély MNIST

Turor (om volt): Takács Viktor

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Lényegében ugyan az, mint az előző feladat, csak apróbb különbségek vannak. A tanulási folyamat és a megjelenítés szintén azonos. A korábban leírt programok telepítése itt is szükséges, valamint a működése

is azonos, tehát az MNIST adatbázisa szerint ismer fel. Az előző feladathoz képest a különbség, az a hatékonyságban rejlik. Ebben a feladatban a program több rejtett réteggel dolgozik, ez pedig megnöveli a felismerés pontosságát. Az eredeti MNIST algoritmussal kb 90% a program pontossága, de ha a rejtett rétegek számát megnöveljük, akkor a pontosság is nő (például 6 rejtett réteggel a pontosság kb 99,7%). Itt a hálóban vannak olyan rétegek, ahol nem mi határozzuk meg a súlyozást, hanem az algoritmus magának készíti el.

## 8.3. Minecraft-MALMÖ

Megoldás videó: <https://youtu.be/bAPSu3Rndl8>

Megoldás forrása: <https://github.com/Microsoft/malmo>

[https://bhaxor.blog.hu/9999/12/31/minecraft\\_steve\\_szemuvege](https://bhaxor.blog.hu/9999/12/31/minecraft_steve_szemuvege)

Tanulságok, tapasztalatok, magyarázat...

Itt a probléma az, hogy hogyan irányítsuk a figuránkat. Ez úgy oldható meg, ha megvizsgáljuk, hogy merre van szabad út, és bizonyos szögekben (elég néhány fokban) elforgatjuk Stevet. Nem csak lapos síkban, vagyis 9 kockát kell vizsgálnunk, hanem 27-et, mivel akár felfelé vagy lefelé (levegő, víz) is lehet út. Ha van szabad útja, akkor kiírja a terminálban, hogy szaad az út, amennyiben nincs, akkor kiírja, hogy milyen akadály van előtte (pl water, grass, air), valamint irány, rácsindex, megtett táv és az éppen látott objektumok kiírása is megtörténik, ha a videóban vett példát vizsgáljuk. Amennyiben körbefordulás után sem tud egyenesen haladni, akkor (például) átugorja az előtte álló akadályt. A játékban az irányok, észak, dél, kelet és nyugat fix szögekből meg vannak adva. A programon belül pedig Steve mozgásához több változó van deklarálva, amelyek a koordinátákat határozzák meg, ezeknek a módosításával tudjuk rávenni az irányváltásra, illetve az akadályon történő átjutásra. A kockákat, amiket említettem, blokkokként kell kezelni a program szempontjából, így hidalhatóak át a problémák.

## 9. fejezet

# Helló, Chaitin!

### 9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó: <https://youtu.be/z6NJE2a1zIA>

Megoldás forrása:

Lispben nagyon sok zárójelet kell használni, illetve a műveletek megadásának formája is változik, itt "fordított lengyel jelölést" használnak, tehát más a műveleti sorrend, először megadjuk az elvégezni kívánt műveletet (összeadás, kivonás például), majd utána adjuk meg a számokat, amelyekkel végre szeretnénk ezt hajtani. Függvények megadására, meghatározására is képes, ehhez először kiadjuk a define parancsot (define (függvény szám)(művelet szám szám)) formában például, ha 2 számmal szeretnénk kezdeni valamit. A függvények és műveletek halmozhatóak, csak a zárójelezésre kell nagyon figyelnünk, mivel sok van belőlük. Beépített függvényeket is tudunk használni, például feltételvizsgálathoz if. A zárójeleken kívül a karakterek közé szóköz kell (pl műveletek, függvények és számok közé), valamint a feldolgozása balról jobbra történik. A faktoriális számolása: a faktoriálisnak megadott számot megszorozzuk az összes előtte álló egész számmal egészen 1-ig. Pl  $3! = 3*2*1$ . Itt a faktoriális definiálásánál if használatával tudjuk megadni a feltételt, hogy a csökkentés során ne menjen 1 alá a szám. A számításhoz pedig rekurzívan kell megadunk a függvényt, például egy faktoriális számítás Lispben: (define (fakt x)(if(itt.jobbra.nyíló.kacsacsőr n 1)1(\* x(fakt(- x 1))))).

### 9.2. Gimp Scheme Script-fu: króm effekt

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely megvalósítja a króm effektet egy bemenő szövegetre!

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)

Tanulságok, tapasztalatok, magyarázat...

Az alapjai ugyan azok, mint az előző feladatban, hiszen a kód Lisp-ben van megírva, de itt már bonyolódik a helyzet. A programkódban az image-scale parancs a skálázásra szolgál. A Gimpben a Script-Fu-konzolba kell beilleszteni a szkriptet, hogy használhatóvá váljon a programban. Jelen esetben ennek a feltöltése után

válik elérhetővé aprigramban a króm effektus. A Lisp kódon belül létezik varlist nevű rész, és ezen a részen belül deklaráljuk a különböző változókat. A kódban szerepel egy car nevű függvény, amely a megadott lista első elemét adja vissza. A létrehozáshoz pedig megadjuk az általunk kiválasztott paramétereket, pl háttérszín, effekt, a kép mérete. Exportálással tudjuk menteni az általunk létrehozott képeket. A króm effekt meghatározása a forráskódokban ki van fejtve, ehhez szükség van egy kis háttértudásra, másnéven nem lehetne megfogalmazni egy színt.

### 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)

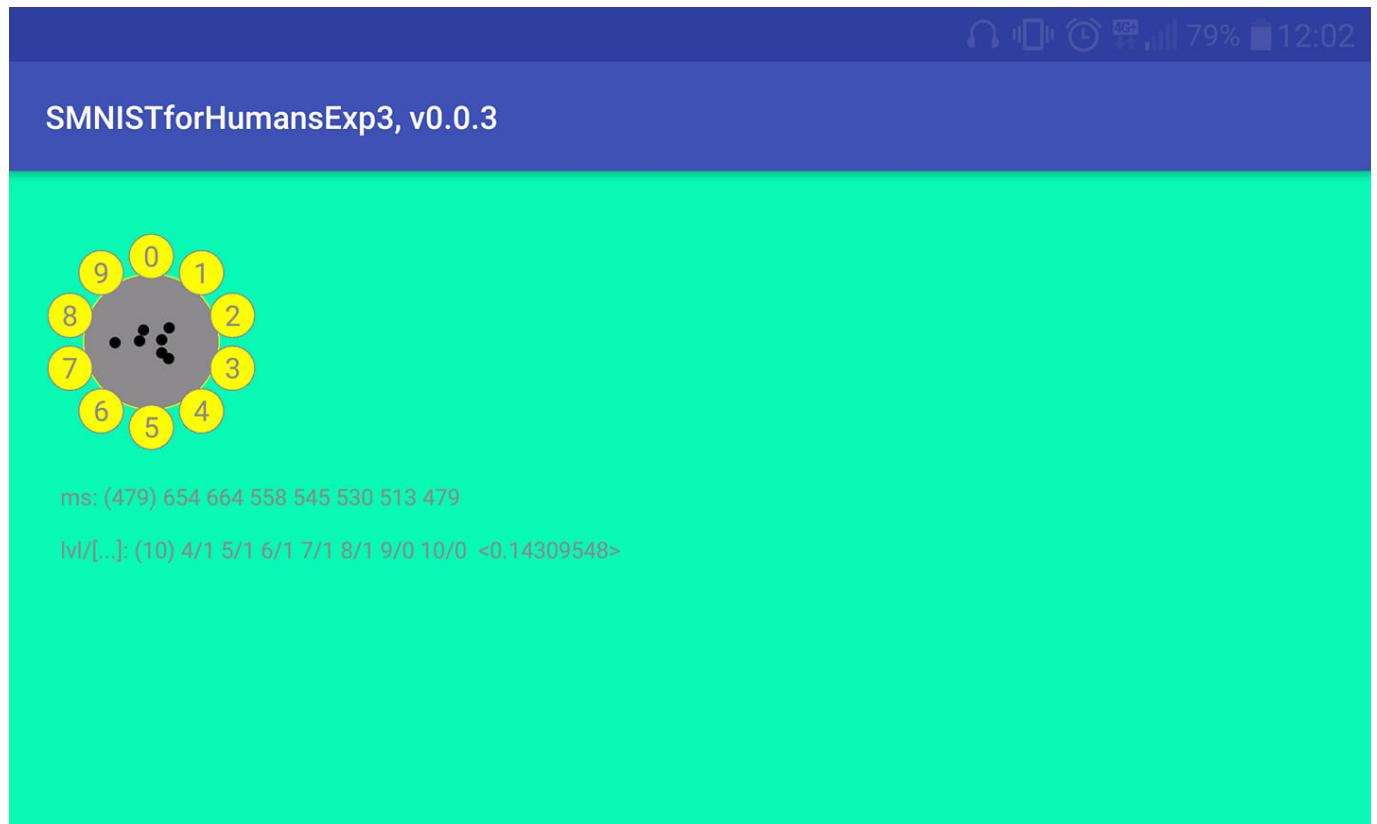
Megoldás forrása: [https://gitlab.com/nbatfai/bhax/tree/master/attention\\_raising/GIMP\\_Lisp/Mandala](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Mandala)

Tanulságok, tapasztalatok, magyarázat...

A Lisp nyelvnek több különböző fajtáját ismerjük, ebben a feladatban Scheme-ben lesz megírva a programkód. Mivel az eredeti Python kódot elég nehéz lenne átírni, ezért az egyszerűsítés kedvéért egy kész mandala elkészítésének a lépései kerülnek megfogalmazásra ebben a feladatban. Első lépésként érdemes a szövegforgatást meghatározni, és beleenni azt a GIMP programunkba, hogy bármikos használhassuk azt. Ez is az API programozás része, tehát erre is ügyelnünk kell. Amire még szintén külön figyelmet kell fordítanunk, az a program felépítése és logikája, valamint annak a kipróbálása és telepítése. A script-ben előre meghatározhatunk formátumot, tehát ha elindítjuk a programot GIMP-ben, akkor azokat fogja venni alapbeállításként, amiket tudunk módosítani is. Megtehetjük például azt is, hogy letöltsük más betűtípus meghatározását tartalmazó fájlt, és azt becsatolva használjuk a programban, azáltal, hogy bemásoljuk először a GIMP-et tartalmazó mappák közül a megfelelőbe, hogy onnan felhasználhatóvá váljon. A forráskódon belül megtehetjük, hogy úgy fogalmazzuk meg azt, hogy egyszerre több színt használhassunk, azaz több RGB kódot építünk bele, de ez már megfogalmazás kérdése. Ezekhez a függvények tökéletes eszközök (pl gimp-image-new). A car függvénynek van egy ellentéte, a cdr, amely az első elemen kívül alista többi elemét írja ki.

### 9.4. SMNIST passzolás feladat megoldás hozzáfűzés

A feladattal kapott 2 passzolási lehetőséget nem használtam fel, de fel szeretném tüntetni a könyvben a játékokat, ami lvl 10-ig ment.



# 10. fejezet

## Helló, Gutenberg!

### 10.1. Programozási alapfogalmak

[?]

Ez az olvasónapló a "Pici könyv" előadások alkalmával meghatározott részeit dolgozza fel. Az első fő egység az alapfogalmak. A programozási nyelveknek 3 különböző szintet állapítottak meg, ezek a gépi nyelv, assembly szintű nyelv, és a magas szintű nyelv. A magas szintű nyelvvel foglalkozunk az órákon. A forrásprogram a programozási nyelven megírt programot jelenti, a "nyelvtani" szabályokat fedi le a szintaktika, valamint az értelmezésért és a tartalomért a szemantikai szabályok a felelősek. A fordítóprogramos vagy interpreteres technika felelős a magas szintű forráskódok gépi nyelvre történő fordításáért, és ezzel előállítja a tárgyprogramot, amit különböző sémák mentén elemez, majd legenerálja a kódot. Léteznek fordítóprogramok, amelyek lényegében bármilyen nyelvről fordít át egy másikra. A hivatkozási nyelv a programnyelvek saját szabványa, ahol a szintaktikai és szemantikai szabályok pontosan meg vannak határozva. Az implementációk realizált interpretek vagy fordítóprogramok. Az implementációk nem kompatibilisek más implementációkkal és hivatkozási nyelvekkel. Ez a hordozhatóság problémája, amit napjainkig nem tudtak megoldani. Manapság már léteznek integrált fejlesztői környezetek (IDE), amelyek nagyban megkönnyítik a programozó dolgát, mivel szinte minden tartalmaznak, amire szükség lehet. A nyelvek osztályozásában több dolgot ismerünk. Az első az imperatív nyelv, ami algoritmus alapú, és képes értékek közvetlen manipulására. 2 alcsoporthoz van: eljárásorientált és objektumorientált nyelvek. A deklaratív nyelvek nem algoritmussosak, és beléük van építve a megoldás keresésének módja. Alcsoporthoz a funkcionális és logikai nyelvek. Azok a nyelvek amik ide nem sorolhatóak a máselvű nyelvek közé sorolhatóak.

A következő amiről írni szeretnék egy kicsit, azok az adattípusok. Az adattípus maga is egy programozási eszköz, amely absztrakt. Az adattípusok név alapján azonosíthatóak. Nem minden programnyelv ismeri ezt, ezért típusos és nem típusos nyelv formájában különböztetjük meg. Az adattípusok a programozási eszközök fölvezető értékeit szabályozza, ide tartoznak a végrehajtható műveletek és minden egyes adattípus mögött áll egy meghatározott, megfelelő ábrázolási mód (pl. bájtok és bitkombinációk). A típusos nyelvek rendelkeznek standard, vagyis állandó beépített típusokkal. Léteznek olyan programnyelvek, ahol saját adattípushoz hozhatunk létre (műveletekkel, mindenkel együtt). Két nagy csoportja van az adattípusoknak. Az egyik a skalár vagy egyszerű adattípusok, amelyek atomi értékekkel rendelkeznek, tehát műveletekkel nem bonthatóak tovább. A másik a strukturált vagy összetett adattípus, aminek az elemei már rendelkeznek valamilyen típussal.

A nevesített konstans egy programozási eszköz, amely 3 részből áll: név, típus és érték. A konstans deklarációjakor megadjuk az értékét, és ezen a program további részében nem lehet módosítani. Praktikus

beszélő nevekkel ellátni, amivel biztosan tudni fogjuk, hogy miért vezettük be. Ha egy értéket többször használunk a programon belül érdemes használni, mert így ha változtatni akarunk, elég egy módosítást végezni a deklarációban, nem kell minden előfordulást megkeresni. A változó szintén egy programozási eszköz, amely 4 komponenssel rendelkezik: név, attribútumok, cím és érték. A név azonosítóként szolgál, amellyel azonosíthatjuk a másik 3 komponenst. Az attribútumok a változók futás során történő viselkedését határozzák meg, a változó által felvehető értékek körét határolja be. A deklaráció segítségével tudjuk a változókhöz rendelni, így különböző típusokat tudunk megkülönböztetni: explicit deklaráció, implicit deklaráció és automatikus deklaráció.

(alapelemek az egyes nyelvekben) A C nyelv típusrendszere aritmetikai (egész, karakter, felsorolásos, valós), származtatott(tömb, függvény, mutató, struktúra, union) és void típusból áll. Az aritmetikai egyszerű, a származtatott összetett típus. Az aritmetikai típussal aritmetikai műveletek végezhetőek, még a származtatott elemeit belső kódok adják. A C az int 0-át hamisként ismeri, a többi számot (int 1 pl) igazként, ilyen a logikai felépítése, mert külön logikai típusa nincs. A karakter és egész típus előtt szerepelhet unsigned típusminősítő, ami előjel nélkülit jelent, illetve lehet signed, ami előjeles ábrázolást jelent. A struktúra fix szerkezetű rekord, míg a void típus tartománya üres, tehát nincsenek műveletei. A felsorolásos típus tartományai nem egyezhetnek meg, és elemei int típusú konstansok. Értékük egész literálokkal beállíthatóak, ha pedig ez nem történik meg explicit értékadással, 0-ról elindul az értékük, és eggyel nő az értékük a felsorolás sorrendjében haladva. A C nyelv csak egydimenziós tömböket kezel. A szabványok változásával folyamatosan történtek módosítások, pl C89, C99. A C a tömböt mindig mutató típusként kezeli, és ismeri az automatikusdeklarációt. A C nyelvben tudunk saját típust definiálni, struktúrát deklarálni és uniont is deklarálhatunk.

A C nyelvben a kifejezések szintaktikai eszközök, valamint értékük és típusuk van. Egy kifejezés 3 összetevőből állhat: az első az operandus, amely értékét ad vissza, a második az operátorok, amelyek a műveleti jelek és a harmadik a kerek zárójelek, amelyekkel pedig a végrehajtás sorrendjét tudjuk befolyásolni. Az operátorok egy-, két- vagy háromoperandusúak lehetnek. Az operátor és az operandus sorrendje lehet prefix, infix vagy postfix. Amikor kiértékelünk, akkor először végrehajtjuk a műveleteket, megkapjuk az értéket és egy típust rendelünk hozzá. Többféle sorrendet tudunk megkülönböztetni, a könyv szerint 3-at. Az operandusok meghatározásának sorarendezésére a C nyelv azt mondja, hogy tetszőleges, azaz implementációfüggő. A zárójelezésre ügyelni kell. A logikai operátort tartalmazó kifejezések speciális kiértékelésűek. Ennek a meghatározásánál a nyelvek 2 elv közül választanak: típusegyenértékűséget vagy típuskényszerítést. A C nyelv néhány numerikus típusnál megenged bizonyos típuskényszerítést, ilyenkor beszélünk bővítséről vagy szűkítésről.

A konstans kifejezések kiértékelését a fordító végzi el, az értéke még a fordítási időben eldől. A C egy kifejezésorientált nyelv, és a típuskényszerítést használja többnyire. A mutató típus elemeivel összeadást és kivonást tudunk végrehajtani. A C nyelvben használhatunk rekurzív függvényeket. Valamint a könyvben le van írva a C precedencia táblázata, amit ide nem másolnék át, (könyv 51.oldal). A következő 3 oldalon az operátorok értelmezéséről olvashatunk. Pl: a [] zárójel a tömboperátor, vagy a % jel a maradékképzés operátora.

Az utasításokról a Kernighan and Ritchie könyvben részletesen írtam, ezért itt csak a hiányzó részeket írnám, mivel csak a C nyelvről van szó, más nyelvekről nem. Az előírt lépésszámú ciklusokban megadunk kezdőértékét és végértékét is a fejben, valamint állíthatjuk a lépésközöt is. A változó irányára változhat, tehát csökkenhet vagy növekedhet, attól függően, hogy merre járja be a tartományt. A felsorolásos ciklusban a ciklusváltozó több értéket is felvehet, ebben az esetben pedig a mag minden egyes értéknél lefut. A változót és az értékeket a ciklusfejben adjuk meg, még az értékeket kifejezésekkel adjuk meg. Létezik összetett ciklus, amely a többi ciklusfajta kombinációjából épül fel, és nagyon bonyolultra is lehet készíteni.

Az eljárásorientált nyelvekben a program szövege programegységekre bontható. Léteznek olyan nyelvek, ahol az egységek külön-külön is fordíthatóak, néhányban csak egy egységek ként lehetséges ez, és van egy harmadik lehetőség ami ennek a kettőnek a kombinációja. Ezekben az eljárásorientált nyelvekben 4 féle programegység létezik: alprogram, blokk, csomag és taszk. Az alprogram a procedurális absztraktió első megjelenési formája, de programozási eszközök ként is funkcionálhat, az újrafelhasználás eszköze. Az alprogramot elég egyszer megírni, újra felhasználható, csak hivatkozni kell rá, bizonyos helyeken meghívható. A formális felépítése: fej, törzs, vég; és 4 komponensből áll: név, formális paraméterlista, törzs és környezet. A formális paraméterlistát kerek zárójelek között találjuk meg. A törzsben a deklarációkat, azok utasításait és a végrehajtó utasításokat jelenítjük meg. Az alprogramban létrehozott/deklarált programozási eszközöket lokális eszközöknek nevezzük, és ezek az alprogram lokális nevei. A környezetben a globális változók helyezkednek el. Az alprogram két fajtája az eljárás és a függvény. Az eljárás tevékenységet hajt végre, még a függvény célja, hogy egyetlen értéket adjon vissza. A függvény paramétert vagy környezetet módosít, akkor azt mellékhatásnak nevezzük, amit többnyire károsnak gondolunk. Az eljáráshívás elhelyezésének annyi kikötése van, hogy olyan helyen kell elhelyeznünk, ahol utasítás is szerepelhet. Az eljárás vagy külön utasításra fejeződik be, vagy akkor ha parancssal megszakítjuk. Illetve néhány nyelv megengedi a GOTO-val történő megszakítást, és átugorva más címre folytatja a futást. Egy függvény többféleképpen meghatározhatja a visszatérési értéket, C-ben a leggyakoribb az, amikor külön utasítás szolgál a visszatérési érték meghatározására, ami ekkor a függvényt befejezi. Befejezzük a függvényt szabályosan, ha elérjük a végét és van visszatérési érték, vagy befejező utasítás és van már visszatérési érték/az utasítás ad vissza visszatérési értéket. Ha nincs visszatérési érték vagy GOTO utasítást használunk, az nem szabályos. Az utóbbi utasítás bizonyos körlmények között C-ben nem jelent gondot. Ezekben a(z eljárásorientált) nyelvekben lennie kell főprogramnak, ami elengedhetetlen. A betöltő a főprogramnak adja át a vezérlést és a többi programrész működéséért is ez felel. Ha ez befejeződik visszatérünk az operációs rendszerhez.

A blokk programegység, ami egy másik programegységen belül található, de nem lehet kívül. Kezdete (ami speciális karaktersorozat vagy alapszó), törzse (ahol a deklarációs és végrehajtó utasítások vannak) és vége van. A blokkoknak nincs paramétere, és bárhol elhelyezhetőek, ahol utasítás is szerepelhetne. GOTO utasítással be- és kiléphetünk belőle.

Néhány program megengedi, hogy az alaprogram meghívása a törzsben is megtörténhessen, ezeknek a kialakításához szükségesek a másodlagos belépési pontok, és annak a nevével lehet az alaprogramra hivatkozni. Függvények esetében a típusnak egyeznie kell, és a másodlagos belépési pontnál a törzsnek csak egy része hajtódnak végre. A paraméterkiértékelés során az alprogram hívásakor egymáshoz rendelődnek a paraméterek, és meghatározzák a paraméterátadásnál a kommunikációhoz szükséges információkat. A paraméterkiértékelésben a formális paraméter aktuális paraméterhez történő hozzárendelése kétféleképpen következhet be: sorrendi kötéssel vagy név szerinti kötéssel. Ha a formális paraméterek száma fix, akkor az aktuálisnak azonos számúnak, vagy kevesebbnek kell lennie. Ha a formálisok száma tetszőleges, akkor az aktuálisoké is. A paraméterátadás az alaprogram és a programegységek közötti kommunikáció, ahol mindenig van egy hívó és egy hívott. Az átadási módok érték (az érték meglesz a paraméterkiértékelés során, és átadódik a hívott alaprogram címkomponensére), cím (a meghívott alaprogram a hívó területén dolgozik, és az infirmációátadás kétirányú), eredmény (saját területén dolgozik, futás közben nem használja az aktuális paraméter címét, viszont ha végzett, átmásolja a formális paraméter értékét erre a címkomponensre), érték-eredmény (az aktuális paraméternek kell cím és értékkomponens, ha végez átmásolódik a formális paraméter értéke az aktuális címére, a kommunikáció kétirányú és kétszer másol), név (az aktuális paraméter egy tetszőleges szimbólumsorozat, ami a formális paraméter minden előfordulását felülírja az alprogram szövegében, és utána fut) és szöveg (a név szerintihez képest annyi a különbség, hogy hívás után elkezd futni, és akkor ír felül, ha először fordul elő a szövegen a formális paraméter) szerinti lehet. A C csak egyetlen paraméterátadási módot ismer. Az alprogramok formális paramétereit 3 csoportra tudjuk bontani,

az információ mozgási iránya szerint.

A hatáskör, vagyis a láthatóág a program szövegének azon része, ahol a név ugyan azt a (programozási) eszközöt hivatkozza. A név hatásköre az eljárásorientált nyelvekben a program- és fordítási egységekhez kapcsolódik. A lokális név a programegységben van, ami pedig azon kívül deklarálunk, de behivatkozzuk, az a szabad név. Dinamikus és statikus határkörkezelést ismerünk. A statikus a fordítási időben történik, és a fordító program készíti el. Ilyenkor a lokális név hatásköre a programegység. A hatáskör csak befelé terjed. A név ami a programegységben nem lokális, de onnan látható, az a globális név. A dinamikus határkörkezelésnél a név hatásköre a programegység, az az abból kiinduló hívási láncok. A dinamikus-nál a hatáskör futási időben futásunként változhat. Az eljárásorientált nyelvek statikus határkörkezelést használnak.

Az programnyelvek közötti legnagyobb különbséget az input és output területe jelenti. A nyelvek különbözőképpen kezelik, néhányban nincs is erre meghatározott eszköz, hanem implementációfüggő. Az I/O feladata a perifériákkal történő kommunikáció, adatok küldése és fogadása és a középpontja az állomány. A logikai állományoknak neve és állományjellemzői vannak, a fizikai állomány pedig a perifériákon jelenik meg és adatokat foglal magába. Ha funkció szerint vizsgáljuk, akkor 3 féle állományt tudunk megkülönböztetni. Az első az input, ami már létezik a feldolgozás előtt, nem módosul közben, és csak olvasni tudjuk. A második az output, amely a feldolgozás során keletkezik, és írni lehet bele. Az utolsó pedig az input-output, ami már létezik a feldolgozás előtt és után is, de módosul, valaint írni és olvasni egyaránt lehet benne. Az I/O során adatmozgás történik, az adatok a periféria és a tár között mozog. A könyv 2 féle adatátviteli mód ról ír, az egyik a folyamatos és bináris, a másik pedig a rekord módú. Az első esetben az adatátvitel fogalma alatt az egyedi adatok átvitelét értjük konverzióval. A karaktersorozatot határozza meg ezzel, illetve annak akezelését, írását és olvasását. Ahhoz, hogy ezeket meg tudjuk adni, 3 különböző eszközrendszer alakult ki: a formátumos módú adatátvitel (a karakterek kezeléséhez szükséges a darabszám és a típus megadása), a szerkesztett módú adatátvitel (egy maszk kell az egyedi adatok átviteléhez) és a listázott módú átvitel (a karaktersorozatban már szerepelnek azok a speciális karakterek, amelyek a tördelésért felelősek). Bináris átvitel esetén a periféria és a tár ugyan úgy jeleníti meg a adatokat, de ekkor csak a háttértárral történő kommunikációról beszélhetünk. Amennyiben állományokat akarunk használni a programunkban, akkor néhány "feladatot" el kell végezni: deklaráció (név és attribútumadás), összerendelés (a logikai állománynak meg kell feleltetni egy fizikai állományt, ami többnyire a programban, nyelvi eszközök segítségével történik meg), az állomány megnyitása (ahhoz, hogy dolgozhassunk egy állománnyal, előbb meg kell nyitnunk, ez az operációs rendszer rutinjaival történik meg, amely valójában ellenőrzi, hogy a logikai állomány attribútumai és a fizikai jellemzői megfelelnek e), feldolgozás (megnyitás után írhatunk és olvashatunk is a megadott szabályok szerint) és a lezárás (elengedhetetlen folyamat, amelyet szintén az operációs rendszer rutinjaival végeznek, ekkor aktualizálódnak, vagyis frissülnek a könyvtárak adatai, információi, lezáráskor szűnik meg a kapcsolat a logikai és a fizikai állomány között, valamint a szabály szerint az output és input-output állományokat muszáj lezárnai, az inputot pedig illik). A programozási nyelvek mára már megengedik nekünk, hogy az írás-olvasás során ne állományokban gondolkozzunk, hanem elképzelhetjük olyannak, mintha közvetlenül kerülne a perifériára. Ezt nevezzük implicit állománynak, tehát a fizikai és logikai állomány kap standard neveket és jellemzőket, így megoldható az automatikus kezelés. A C nyelvben nincs I/O eszközrendszer, vagyis nem része annak, helyette könyvtári függvényeket használhatunk.

A kivételkezelést a programozó az operációs rendszertől veszi át, valamint ezek megszakítást okoznak és maga a program végzi el. A nyelvek használnak bizonyos beépített kivételkezelőket, de megtörténhet, hogy megszakításkor a program nem veszi figyelembe és fut tovább, de ennek következményei lehetnek, amik csak utólag derülnek ki. A kivételeknek 2 fő tulajdonsága van, a név és a kód. Kivételt tud létrehozni a programozó is, ha condition(név) formában deklarálja azt. A kivételkezelés (Ada-ban) egy adott időpontban kezdődik, és a kezelés valamilyen formában történő lezárásáig tart. A PL/I határkörkezelése

ellentmondásba ütközhet a nevek (statikus) és a kivételkezelő (dinamikus) kezelése miatt.

## 10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

Online könyv forrás: <http://lidi.uw.hu/krc/index.html>

Ebben az olvasónaplóban a változókkal és az adattípusokkal szeretnék kezdeni. Különböző változó típusokat, azaz adattípusokat tudunk megkülönböztetni, amelyek meghatározzák az adott változó formáját, ezzel befolyásolva a program működését. Az int, teljes névén az intiger típusban a változó egész szám értéket lehet fel, ellentétben a float típussal, amely lebegőpontos értéket tartalmaz, tehát lehet tört szám. A változók nagysága függ a számítógép típusától is, de a leggyakoribb 16 bites int típus -32768 és +32767 között van. Létezik 32 bites int és float is. További adattípusokat, változó típusokat is ismer a C nyelv, ezek a char, ami karaktert jelöl egy bájton ábrázolva, a short rövid egész típust jelöl, a long hosszú egész típus, valamint a double típust ismerjük még, amely kétszeres pontosságú (2 tizedes jegy) lebegőpontos valós szám ábrázolására képes. A bithosszok módosíthatóak a short és a long előjellel. Például az int esetében, a short legalább 16, még a long legalább 32 bites. A short nem lehet hosszabb sem az inttől, sem a longtól. Hasonló elven működik a signed és az unsigned minősítők is, amelyeket a char vagy bármely egész típusnál használunk. Számok esetén az unsigned a típus előtt azt jelenti, hogy előjel nélküli, tehát a szám nagyobb vagy egyenlő, mint 0. A változók érvényességi tartománya változó. Léteznek lokális változók, amelyek pl csak egy adott függvényen belül léteznek, és ha lezártuk a függvényt, használhatjuk újra ugyan azt a változó nevet. A másik típus a globális változó, amely az egész programkódon átível, tehát nem használhatjuk ugyan azt a változónevet többször. Léteznek konstans változók is. Ezeknek is adunk egy nevet, de az értékük fix marad, nem változnak meg a programon belül sehol sem. Kiterjedésük globális vagy lokális.

A vezérlési szerkezetek című fejezetben néhány utasításról olvashatunk. A vezérlési szerkezetek végre-hajtási sorrendet határozzák meg. Egy kifejezés, akkor válik utasítássá, ha utána írjuk a ; jelet. Ezzel zárnak le az utasítást, pl: printf(...); . A {} jelek közötti utasítások és deklarációk egyetlen utasítással érnek fel, és blokknak nevezzük. Ide több utasítást is írhatunk, amelyeket a program egyként kezel. A záró } jel után soha nem rakunk ; jelet. A blokkokban lehet deklárálni is. Ilyen blokkok vannak pl az if, else, while vagy for, de saját függvények esetében is így működik. Ha döntéseket szeretnénk kifejezni, akkor az if-else utasításpárt használhatjuk. Az if után írjuk a kifejezést amit vizsgálni szeretnénk, és utána az utasítást, majd az else után a második utasítást. Az else rész opcionális. Ha az if-ben szereplő feltétel értéke igaz, akkor az első utasítás hajtódiék végre, ha az hamis, akkor a program tovább ugrik az else ágra és a 2. utasítást hajtja végre. Amennyiben több if-et írtunk a programba, és van else ág, akkor az egyértelműség végett {} zárójeleket kell használnunk az if után, így biztosan tudni fogja a program, hogy mit tartalmaz az első utasítás. Az else-if utasítás szerkezet adja a többágú, vagyis a többszörös döntésű programozás legáltalánosabb lehetőségét. Itt az if, és az else if utasítások mögött is van egy kifejezés, amit megvizsgál. Amennyiben talál egy olyan kifejezést, ami igaz/teljesül, akkor végrehajtja azt és kilép az utasításból. A {} zárójelek közötti blokkok itt is ugyan úgy léteznek, mint az if-else esetben. A végén van egy if nélküli else ág is, ami akkor lép működésbe, ha a fenti feltételekből egy sem teljesül. Ez továbbra is csak opcionális. A switch utasítás szintén a többirányú programelágaztatás eszköze. A switch-ben megadjuk a kifejezést, amit meg kell vizsgálni, utána a case-ben pedig egy állandó kifejezést, amivel össze lehet hasonlítani a switch-ben szereplő értéket. Létrehozhatunk egy default ágat is, ami opcionális. Ez abban az esetben hajtódiék végre, amennyiben a case kifejezései/értékei közül egyikkel sem egyezik meg a switch kifejezésével. A break utasítás feladata az, hogy megszakítsa az adott folyamatot. Ez az utasítás while, for vagy do utasításokból

összeállított ciklusokból való kilépésre alkalmazható. A while és for ciklusokban a program először kiérte a kifejezést, és utána végrehajtja azokat, amennyiben az érték nem nulla. Ha 0 lesz az érték, az azt jelenti, hogy a kifejezés hamis, és megáll a program. A 3 kifejezés amit használunk a ciklusokban kihagyhatóak. Ha nincs kifejezés, akkor a program minden igaznak tekinti, tehát végtelen ciklustkapunk. A break és return parancsokkal megállíthatóak. Ezekben a ciklusokban közös, hogy a ciklus tetején/fejben, tehát a ciklusmagba lépés előtt vizsgál. A do-while ezzel ellentétben legalább 1-szer végrehajtódik, és majd csak utána vizsgál. A do után leírjuk az utasítást, majd while(kifejezés), amit utána megvizsgál. Amennyiben igaz, újra lefut, ha hamis, akkor megáll és átlép a következő utasításra. Tehát a leállás feltételét a ciklusmag végrehajtása után ellenőrzi. A tapasztalatok szerint sokkal ritkábban használják a do-while ciklust, mint a for vagy while ciklust. Kényelmes a használata, {} zárójeleket nem kell használni, mivel a while rész nem téveszthető össze, egyértelműek a részek. A break utasítás kényelmi célt szolgál, így ellenőrzés nélkül is kiléphetünk a ciklusokból, valamint a switch utasításból. Utóbbinál vagy a belső utasítás vagy a teljes switch fejeződik be. A continue utasítást csak ritkán használják. Hatására a kifejezést figyelmen kívül hagyva vég-bemegy az utasítás/megkezdődik a következő iterációs lépés. Akkor használjuk, ha a ciklus további része nagyon bonyolult. A goto utasítás vitatott múltú. Ezzel az utasítással a megadott címkékre lehet ugrani. Ritkán használják, könnyen lehet olyan programot írni amiben nincs rá szükség. Gyakorlati haszna, amikor egymásba ágyazott szerkezetek belsejében szeretnénk félbe hagyni a feldolgozást. Ilyenkor a break utasítás nem használható, hiszen az csak a legbelső utasításból/ciklusból lép ki. A goto utasítást érdemes a lehető legritkábban használni, mert azok a programok, amiben szerepel, többnyire nehezebben átláthatóak.

Az utasítások amennyiben nem jelezzük külön, a leírásuk sorrendjében hajtódnak végre. Az utasítások nem rendelkeznek értékkal, hanem "hatással vannak" a programra. Több csoportjukat ismerjük. Az első amiről írnék, azok a címkézett utasítások. Ezekhez az utasításokhoz előtagként megadott címkék tartozhatnak. A címkék csak goto utasítás célpontjául használhatók. Nem tartozik hozzájuk tárterület, és az aktuális függvényeken belül érvényesek csak. A kifejezésutasítások közül a legtöbb függvényhívás vagy értékkadás. Ha hiányzik belőle az utasítás, null-utasításnak nevezzük. Utasítások üres ciklusmagjának vagy címkék helyének jelölésére használhatják. Összetett utasításokat olyan helyeken használunk, ahol a fordítóprogram csak egyetlen utasítással dolgozik, csak annyit fogad el. A blokk ezt a célt teljesíti. Ilyen lehet például egy függvénydefiníció. A blokkon kívüli deklarációk nem érvényesek a blokkon belül, addig megszűnnek. A kiválasztó utasítások egy lehetséges végrehajtási sorrendet választanak ki. If, if-else vagy switch használatával lehetségesek. A switch utasítás összetett utasítás, de ezeknek a működését korábban kifejtettem a könyv alapján, ugyan ez a működési elv itt is. Az iterációs utasítások ciklusok felhasználásával keletkeznek, tehát while, do-while és for ciklussal. Itt szintén a ciklusok működését kellene kifejteni, de korábban ez már szerepel. Az utolsó csoport a vezérlés nélküli utasítások, amelyek a vezérlés feltétel nélküli átadására képesek. Ezek az utasítások a goto, continue, break és return. A return kivételével a többöt kifejtettem korábban. A függvények a hívások hatására a return hívó függvény hatására térnek vissza. Utána kifejezés áll, amelyet a hívó függvénynek ad vissza. Ha nincs return, akkor nincs definiálva visszatérési érték.

Megoldás videó: <https://youtu.be/zmfT9miB-jY>

## 10.3. Programozás

[BMECPP]

A C++ alkalmazás objektumorientált és generikus programozásra is, azonban alacsony nyelvi szintű programnyelveket is támogat. Bjarne Stroustrup a "kitaláló atya". A C nyelvre épül, és azt terjeszti ki. A különböző fordító programok segítségével a C programok működnek C++-ban is.

Különbségek a C és C++ között. Az utóbbi nyelven az üres paraméterlista a void függvény megadásával azonos, amelyek jelentése az, hogy a függvényeknek nincs paramétere. Ugyanakkor C++-ban is van lehetőség számunkra szabadon választható számú paraméterrel hívható függvények meghatározására. Ezek a nyelvek a függvény visszatérési típusának meg nem adásától függően különböző módon viselkednek. A C++-ban a main függvényt kétféleképpen határozhatjuk meg. A return használata nem kötelező a főfüggvényben, ebben az esetben automatikusan return 0-ként kezeli, azaz sikeres futásként. Újdonság a bool típus is, amely egy logikai típus, és true vagy false, tehát igaz-hamis értéket képes felvenni. Ennek a típusnak az előnye, hogy olvashatóbb kódokat eredményez. Létezik automatikus konvenzió a túlterhelés ellen, a 0 int értéket a nyelv falsenak, még minden más értéket truenak vesz. Ezek a szavak a C++ kulcsszavai közé tartoznak. A C++-ban már könnyebben használhatóak a Unicode karakterek mint a C-ben, mert itt már előre definiálták őket. A változódeklarálás is másabb a C++-ban. Itt minden olyan helyen lehet változódeklaráció, ahol utasítás állhat. Ott érdemes deklarációt létrehoznunk, ahol azt használni szeretnék használni (vagyis előtte), így csökkenthetjük a hibalehetőséget és átláthatóbbak a kódok. A változók érvényessége a deklarációtól kezdődik, és a blokk végéig tart (kivéve if, for). Míg a C nyelven nem lehetett 2 azonos nevű függvény, addig a C++-ban lehet, ahol a nevekhez az argumentumaikat társítják, és azzal különböztetik meg. Ha C++-ban akarunk C-ként fordítani, akkor az extern "C" deklarációt kell a függvény deklarációja elő írni. A C++ a névelferdítés technikáját használja. A C++-ban lehetőségünk van arra, hogy a függvény-argumentumoknak kezdeti értéket adjunk, amennyiben ez nem történik meg, akkor alapértelmezett értéket kap. Az alapértelmezett argumentumok megadásának többnyire a függvénydeklarációnál van értelme a használat miatt. A C programok csak az érték szerinti paraméterátadást ismerik. A paraméter átadásának érdekében mechanikus átalakítást kell végeznünk. A paraméterlistában a változó helyett pointert veszünk át, tehát a név elő régi helyen áll. A változó helyett annak a címét kell átadni, ezért az átadandó változó neve elő régi helyen "end/és" jelet. Így az átadott érték maga a pointer lesz, tehát könnyebb lesz a módosítás. Ezt cím szerinti paraméterátadásnak nevezzük. Az érték szerintivel ellentétben, itt már végezhetünk módosításokat, míg az utóbbi másolatot készít, de nincs hatással az eredeti változóra. A C++-ban bevvezették újabb könnyítésként a referenciátipust, ezzel feleslegessé téve a pointer szerepét a referenciaátadásban. Az "end" jel a C-ben operátorként funkcionált, és ezt tovább hozta magával, és ezt a C++ fel tudja használni referencia deklarálásához. Csak megváltoztatható kifejezést kaphat a referencia, pl nem lehet konstans, kivéve konstansreferencia. Lokális változókra nem szabad referenciaival hivatkozni, mert ezek felszabadulnak, így üres memóriaterületre hivatkoznának.

Az objektumorientáltság alapelveinél az áttekinthetőség szerepe kulcsfontosságúra nőtt. Az egységbe zárás az egyik alapelve, ami az adatstruktúra tulajdonságait és a rajta végzett műveleteket tartalmazza. Az egységbe záró struktúra neve osztály, ami egyfajta kategóriaként funkcionál. A példányokat, vagyis önálló "egyedeket" objektumoknak nevezzük. Ha a program többi része hozzáfér egy bizonyos tulajdonséghoz, akkor azon változtatni is tud. Ezért kell adatvédelmet biztosítanunk, hogy az objektum "tudjon vigyázni önmagára", és a program belsejéhez ne férjen hozzá a program többi része, ezért ezt az egyfajta védekezést hívjuk adatrejtésnek. A könyv példája a személy-alkalmazott viszony, amivel a tulajdonságok örökösítetlenül mutatja be. Egy másik tulajdonság a behelyettesítés, vagyis az alkalmazottat kezelhetjük személyként is, tehát egy speciális osztály objektuma bárhol felhasználható, a speciális helyettesíti az általánost. Ez az OO = objektumorientált programozás 3 alapelve. A típusámagatás már nem tartozik ezek közé, de a felhasználásakor a felhasználó által definiált típusként úgy viselkedik mint a beépített típusok. Az OO programozásban ha analízisről vagy tervezésről beszélünk, akkor objektumokban és osztályokban gondolkodunk, ami jobban tükrözi az emberi gondolkodásmódot. Az OO miatt teljesítménycsökkenés tapasztalható, ezért lényeges az átláthatóbbá tételek. Az előnye viszont, hogy összetettebb problémákat is meg lehet oldani. Az egységbe zárára példa a koordináta rendszer. Mivel egy ponthoz az x és y koordináta összetartozik, ezért egy struktúrába lehet őket zárni. A függvények ugyan úgy szorosan kapcsolódnak a koordinátákhoz, mint azok egymáshoz, tehát érdemes

a függvényeket a struktúra tagfüggvényeivé tenni. A tagváltozót nevezzük attribútumnak, a tagfüggvény pedig metódus vagy művelet más néven. Tehát a tagváltozó a struktúra adattagja. Ahányszor változót hozunk létre, annyiszor foglalódik hely a struktúra tagváltozónak. A tagfüggvényeket osztálydefinícióban, és a struktúradefiníció kívül is megadhatjuk. A struktúradefiníciót belül a prototípust adhatjuk meg. A hatókör operátor az ütközések elkerülésére szolgál. A tagfüggvény csak egyetlen példányban jön létre a memóriában. Az adatok és a függvények így a struktúrában egy helyre kerültek. Az adatrejtéssel a változó adattagjait az adatfüggvényből lehet így csak elérni, és kívülről nem. Erre van a private kulcsszó. Az ez után felsoroltak az az adott osztályon belül lesznek láthatóak. Ellentéte a public, ami osztályon kívülről is elérhető és módosítható. A struktúra nem tükrözi ezt az elvet ennyire szabályosan. Az osztály egy típus, amelyből ha fel szeretnénk használni, változót kell deklárnunk. Ezt az osztály példányosításának nevezzük. A létrejött változó az objektum. Az osztálydefiníció többször is be lehet építve a forrásállományba, ide kell az #ifndef direktíva, vagyis ezt ilyenkor használjuk. A konstruktur az objektumok létrejötténél az inizializálásra lett kitalálva, hogy az objektum inicializálhassa önmagát. Ez egy speciális tagfüggvény és a neve megegyezik az osztályéval, aminek a példányosításakor automatikusan meghívódik. A konstruktur is túlterhelhető. Üres zárójelek nem megengedettek. Ha nem írunk konstruktort, akkor az osztály létrehoz egyet, ami alapértelmezett, de nem csinál semmit. Viszont ha mi írtunk, akkor az osztály csak azzal példányosítható. Az objektumok által birtokolt erőforrások felszabadítását a destruktur végzi el, és egy ~ jelrel kezdődik, majd az osztály neve jön, ezzel felszabadítunk. A dinamikus memóriakezelés arra szolgál, hogy ha nem adtuk meg a tároló maximális méretét, akkor addig tehetünk bele elemeket, ameddig tud nekik memóriát foglalni. A C-hez képest már továbbfejlesztették a biztonságát. C++-ban a new operátorral lehet létrehozni, amire a paraméterátadás miatt van szükség, tehát ezért nem függvényre va szükség. A new a lefoglalt típusra mutató pointerrel tér vissza. Az üres zárójellel alapértelmezett konstruktort hívunk. Használat után a delete operátor segítségével szabadítunk fel. A delete meghívja a felszabadításra váró objektum destruktörét. A tömbök lefoglalása és felszabadítása new[] és delete[] operátorral történik. Ha ezeket nem tartjuk be, memóriaszivárgás, vagy más következménye is lehet. A FIFO dinamikus osztálytal a tároló egész típusú értékeit egy pointerrel megjelölt dinamikus foglalású adatterületen tároljuk el. Az új elemnél dinamikus területet növelünk, és hozzáfűzzük az adatot. Ha elveszünk egy elemet, akkor a területet csökkentjük és átmásoljuk a többi elemet. Óvatosnak kell lennünk, mert utána a memóriaterületet fel is kell szabadítani. A nyilvántartásra nem elég egy pointer, tudnunk kell az elemeink számát. A memóriaterület a beletett és kivett elemktől függően fog változni. Ha új elemet helyezünk kell, 2 dologra kell figyelnünk: meghaladja, vagy nem haladja meg a tároló mérete az előre foglalt területet. Az első esetben nagyobb méretű tömböt kell foglalnunk, áthelyezni a tartalmat, az új elemet hozzáfűzni, felszabadítani a tömböt és a pointert át kell irányítani. A másik esetben az elemet csak a végére fűzzük. A másolókonstruktur rendelkezik az összes többi konstruktur lehetőségeivel, azaz létrehozáskor az objektumot inicializálhatjuk. A másoló esetén a meglévő konstruktur alapján inicializáljuk, cél a másolat készítés. A másolás beépített típusoknál egyszerű, mert ismeri (méretét, helyét a memóriában, stb...), és így csak bitenként átmásolja. Ez a művelet struktúrára és objektumokra egyaránt elvégezhető. A másoláshoz függvényt kell írnunk, másképp bitenként másol. A bitenként másolást sekély másolásnak nevezzük, még a dinamikus másolást mély másolásnak. Ha nem írnánk másoló konstruktort, akkor a példában a main függvényben a fifo azt hinné, hogy minden a régi, de a param objektum függvényből való kilépéskor destruktur hívódik, ami felszabadítja a data által mutatott területet. Emiatt a fifo tartalmát használó lekérdező és módosító függvények, és a destruktur érvénytelen adatterülettel dolgozna. Ez az érték szerinti paraméterátadásnál vagy inicializálásnál probléma. Ha mégis így adnánk át, kiderülne a hiba. C++-ban az osztály adhat jogosultságot globális valamint más osztályok tagfüggvényeinek ahoz, hogy hozzáférjenek a saját védett részeihez. Itt a friend függvény szolgál erre a célra. Attól még ugyan úgy globális függvény marad, csak speciális jogai vannak. A friend osztály hasonló ehhez, csak az egy másik osztályt jogosít fel arra, hogy hozzáférjen a védett tagjaihoz. A friend tulajdonság nem öröklődik, és nem tranzitív. Ez nem mond ellent az adatvédelemnek, mert szabályozott módon történik.

nik, de nem érdemes túl gyakran használni. Az inicializálás jelenthet konstruktorthívást, vagy az "==" jellel értékadást. A tagváltozókat a konstruktur inicializálási listában tudjuk inicializálni. Az argumentumok után ":" jelet teszünk, és utána írjuk az inicializálni kívánt tagváltozókat. A statikus tagokváltozók speciális változók, amelyek definiálásra van lehetőség. Ezek a változók közös értéket vesznek fel. Ha az osztálynak nincs objektuma, akkor is használhatóak. A deklarálásukhoz static kulcsszóra van szükség. Ezeket az osztálydefiníció kívül is definiálni kell. A szintaktika hasonlít a globális változókhoz, de hatókör (::) operátor szükséges. Statikus tagfüggvények definiálására is van lehetőség. Ezek a statikus változókkal dolgoznak. A statikus tagfüggvényekből a nem statikus változók és tagfüggvények nem érhetőek el. A statikus függvényekben a this mutató nem használható/értelmezhető. A közös statikus változó itt a count. Az adatréjtés miatt private. Akkor célszerű a használata, ha az adott osztály minden objektumára közös változóra van szükség. Mindig az induláskor inicializálódnak. Az osztálydefiníció belül adhatunk meg enumerációt, osztály, struktúra vagy típusdefiníciót, ezt hívjuk beágyazott definíciónak. A beágyazott enumeráció definíció az objektum viselkedését befolyásolja. Más osztályok tagfüggvényeiből vagy globális függvényekből csak a minősített nevükkel érhetőek el a beágyazott típusdefiníciók. A beágyazás nélkül a globális megvalósítással a névütközések száma valószínűleg több lenne. Ha private szakaszba vannak beágyazva ezek a definíciók, akkor csak az osztályon belüli tagfüggvényekből lennének elérhetőek. A beágyazott definíciók használatával sem érdemes túlzásba esni.

Az operátorok az argumentumaikon hajtanak végre műveleteket, amelynek az eredményét visszatérési értékként történő feldolgozásával használhatjuk. Például: a c++ kifejezésben a c legyen a változó, a ++ operátor argumentuma. Az operátorok kiértékelése bizonyos sorrend, egy speciális szabályrendszer szerint történik meg. Érdemes zárójelekkel használni. Még a C-ben csak érték szerinti paraméterátadás volt, addig a C++-ban már nincs ez a korlátozottság. C++-ban az operator egy kulcsszó, amit speciális függvényekhez adunk meg. Az operátorok (Pontosabban speciális függvények és függvénynevek) is túlterhelhetőek a függvényekhez hasonlóan. Az osztály típus miatt érdemes tagfüggvényként definiálni. Definiálhatunk ilyen operátorokat pl összeadáshoz, kivonáshoz, szorzáshoz... A könyvben van rá konkrét példa is.

Az I/O kezelés: A C nyelvben a program miután elindult, 3 létrehozott és megnyitott állományt és leíróját tudjuk felhasználni, pontosabban az stdin, stdout, és stderr, amelyek a szabályos bemenet, kimenet és hibakimenet. Ezek magas szintű állományleírók és FILE\* típusúak, az stdin csak olvasható, még az stdout és stderr pedig csak írható. Ezzel ellentétben a C++-ban már objektumokban kell gondolkozni, mert ez adatfolyamatokban, vagyis streamekben gondolkozik, amelyek irányítására 2-2 egymás mellé írt jobbra vagy balra nyíló kacsacsőr szolgál. A könyvben találhatunk erről egy táblázatot, ami jól összefoglaljaaz előbb leírtakat. Ezeknek a használatához szükséges includolni az iostream állományt, valamint érdemes a következő sorba beírnunk, hogy using namespace std, mert így nem kell kiírnunk mindehová hogy std:: , azaz azt, hogy standard névteret használunk. Adatáramláskor a kacsacsőrök olyanok mint a nyilak, tehát az adatáramlás irányába mutatnak. A cin nagyon hasonlít a scanf-re, és addig halad ameddig meg nem kapta a megfelelő bemenetet, vagy el nem ért egy záró karaktert vagy a bemenet végét. C-ben volt az fflush(stdin) függvény, amely megakadályozta hogy megálljon a beolvasás. A rendszerhívások nagy költséggel járnak, ezért az adatfolyamok buffert kapnak, és a program egy rendszerhívással több cout kiírást is kezel, kiír, amennyiben viszont törölnénk a buffert, flush "kiíratással" megtehetjük, cout.flush(). A memória fogyás esetére, haezt közölni kellenemár a felhasználóval, a cerr nem rendelkezik bufferrel. C++-ban létezik egy iostate tagváltozó, amely az adatfolyam állapotának jelzésére szolgál, és 4 konstanssal állítható az értéke: eofbit, failbit, badbit, goodbit, és ezeket a clear tagfüggvénytelhet beállítani. Azt, hogy a beolvasás sikeres e, függvény helyett whileciklussal is megoldhatjuk. A könyv 79. oldalán található egy táblázat, amely leírja a beolvasás és kiíratás fontosabb tagfüggvényeit, és ugyan azokat C-ben is. C++-ban már van string osztály, így beolvasáskor az automatikusan növeli a saját méretét, azonban ha szóközt kap, ez a beolvasás is megállna, erre a megoldás pedig a getline függvény, amellyel sorokat és szóközt tartalma-

zó szöveg is beolvasható. Léteznek I/O manipulátorok, amelyek speciális objektumok és az adatfolyamot módosítják, valamint a megszokott ki- és bemeneti operátorok argumentumaként kell alkalmazni. Szükséges a használatukhoz az iomanip állományt includálni. Manipulátor például az endl, a noskipws, a ws vagy a setw. Néhány manipulátor rendelkezik paraméterrel, néhány pedig nem. Léteznek jelzőbitek, vagy más néven flags, illetve maszk, amely egy bináris szám, amit abban az esetben kapunk meg, ha egyszerre több tulajdonságot szeretnénk tárolni, és a tulajdonságot meghatározó bitek 1-esek. Állománykezelésnél C++-ban a leírót egy objektum zárja körbe, és ennek a tagfüggvényeinek a felhasználásával tudunk különböző műveleteket elvégzni. A leíró tagváltozóként érhető el. A C++ állománykezeléshez adatfolyamot használ, ehhez szükség van az ifstream és ofstream osztályokra, mivel ezekkel valósítja meg, illetve az fstream foglalja ezeket magába, mivel ez kétirányú áramlást biztosít. A korábban leírt, adatfolyamokon végzett műveletek itt is érvényesek, mivel ezek is lényegében adatfolyamok. A megnyitásáért konstruktorkor, a lezárássáért pedig destruktorkor felelősek, ígybiztonságosabb a folyamat. A használatához szükség van az std névterre (tehát érdemes a using namespace std) és az fstream includálására. Az üzemmódok használatához jelzőbitekre van szükség, smelyeket ios:: előtaggal kell használni. A konstruktur és destruktur helyett használhatunk függvényeket, pontosabban az open és a close függvényeket, valamint létezik az is\_open tagfüggvény, amellyel megvizsgálhatjuk, hogy meg van-e nyitva az állomány. A pozícionálásra 2 függvényt tudunk felhasználni, bemeneti adatfolyam esetén olvasási pozícionálásról (get) beszélhetünk, kimenetnél pedig írásiról (put). A programozó felelőssége, hogy jól pozícionáljon, vagyis érvényes pozícióval dolgozzon. Használhatunk átirányítást is, amely jelen volt már C-ben is, de C++-ban már csak módosításokat végeztek rajta (megnyitjuk az állományt és az irányítandó adatfolyam tulajdonságait, majd átmásoljuk azokat a megnyitott adatfolyamunkba).

A C-ben történő hibakezeléshez és kivételezéshez képest a C++ egy fejlettebb megoldást nyújt, mert amíg a C hibakódokkal dolgozott, addig a C++ a kivételezés segítségével felhasználóbarátabb és könnyebb a hibakezelése. A hiba- és kivételkezelés fogalma nem egy és ugyan az, mert kivételeket alkalmazhatunk bármikor, amikor szeretnénk, attól függetlenül, hogy helyes vagy hibás, ilyenkor a program a kivételkezeléshez lép. A 190.oldal példájának megoldása szerint egy try catch blokk alkalmas a vizsgálatra, ha a catch rész tartalmazza a hibakezelést. Amennyiben a try, vagyis a védett részben nem találkozunk semmilyen hibával, akkor fut tovább a program, és a catch következik, ami viszont csak akkor fut le, ha előtte hiba volt. Ha nem volt, akkor a catch lényegében kimarad, és folytatódik a futás a következő résszel. A throw utasítás a kivételkezelést szolgálja, és emiatt rögtön jön a catch ág, aholá a kivétel bekerül. Ezek után pedig kiíródik a catch utána rész, jelen esetben a "Done." szöveg. A továbbiakban a könyvben függvények kivételkezeléséről olvashatunk, és azokra láthatunk 1-1 példát. A try catch blokkokkal megtehetjük, hogy egymásba ágyazzuk őket, tehát szinteket hozunk létre, erre a 195. oldalon láthatunk példát. Illetve, ha a throw függvény másképp használjuk, vagyis pontosabban nem adunk meg neki paramétert, akkor elérhetjük vele, hogy a megfogott kivételt újradobja. A 197. oldal példája a verem visszacsévélését mutatja be, azaz a dobás és elkapás között léteznek olyan függvények, amelyeknek a lokális változói felszabadulnak. Egy dobás és elkapás között nem lehet újabb kivétel dobás, mert az összességében a program futásának leállásához fog vezetni. A 211. oldalon látható kódcsipet az erőforrás kezelésre mutat egy példát. Ez bemutatja, hogy a memóriát mikor kell felszabadítani, illetve az ez után történő kivétel újradobás kulcsfontosságú, hiszen amennyiben eltekintenének tőle, akkor a hiba nem derülne ki.

## **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, Berners-Lee!

### 11.1. C++

C++: Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven

Megoldás video:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Nem szeretnénk túl sok bevezetéssel kezdeni, hiszen ennek a könyvnek egy részéről már írtam a prog1-es részben, illetve a következő Java olvasónaplóban is vannak részek, ahol kiemelem a Java és a C++ különbségeit röviden, de külön is le szeretném írni, itt az általánosság mellett inkább C++-ra kihagyevze a történetet. A C++ egy objektumorientált nyelv, amelynek kialakulásához az egyre bonyolultabb programok igénye vezetett. Itt az áttekinthetőség és a hatékonyúság volt a legfontosabb. Az osztály lényegében egy egységbe záró adatstruktúraként fogalmazható meg. Az osztály példányokkal, vagy más néven egyedekkel rendelkezik, ezeket nevezzük objektumoknak. Az osztály egyik feladata az adatrejtés, amelyet az objektumokkal lehet megvalósítani. Célja, hogy (például) csak azok az osztályok férjenek hozzá, amelyek dolgoznak az adattal, de más ne férjen hozzá, így ne legyen akárhogyan kinyerhető egy adat. Az öröklődés során egy osztály objektumai "öröklik" és másik, általánosított osztály tulajdonságait. A C++ OO, tehát objektumorientált nyelv. Emiatt létezik benne típustámogatás, tehát a programozó, vagy felhasználó által definiált típusok úgy viselkednek mint a beépített típusok. Ez az objektum orientáltság jellemző Javara is. Az egységbe záráskor tagváltozókat (objektumok), vagyis attribútumokat, valamint tagfüggvényeket, vagyis metódusokat hozhatunk létre. C++-ban és Javaban is létezik a "hatókör" fogalma, ha az osztályokból szeretnénk például valamelyik tagfüggvényt használni, mivel lehetnek különböző osztályokban azonos nevű és paraméterlistájú tagfüggvények. Ennek a megkülönböztetésére szolgál. A Javaban történő használatról írtam a következő olvasónaplóban. C++-ban viszont ebben az esetben a hatókör, azaz scope operátort használjuk, amelyet :: (dupla kettősponttal) jelölünk. Itt vissza térnék még egy kicsit az adatrejtésre (data hiding). A lényegéről írtam, de a megvalósításáról is szeretnék egy kicsit. Az osztályokon és struktúrákon belül kulcsszóval tudjuk befolyásolni, hogy az adatot használhatja bármi más ami az osztályon kívül van. (Igazából 3, mert ott van a protected is, de azt későbbre hagynám még.) Ez a 2 pedig a private és public. A private és public egymás ellentétei. Még előbbi azt éri el, hogy az adott tagváltozók és tagfüggvények csak az adott osztályon belül legyenek elérhetőek, addig az utóbbi pont az ellenkezőjét, tehát ezek az osztályon kívül és belül is láthatóak lesznek. A C++-ban szintén jelen van a dinamikus memóriakezelés, de itt már nem függvény (mint C-ben), hanem operátor felelős érte. A new operátor szolgál a memória foglalásra, és a new a lefoglalt típusra mutató

pointerrel tér vissza. A lefoglalt helyet a delete operátorral tudjuk felszabadítani, amennyiben már nincs rá szükségünk. C++-ban nagykülönböző van az értékadás és az inicializálás között. Inicializálásra akkor van szükség, ha változókat vagy objektumokat szeretnénk létrehozni, értékadást viszont az "=" jellel történik, amivel a már meglévő változóknak vagy objektumoknak egy értéket adunk. A folytatás a továbbiakban a következő részben, illetve ott a kettő összevonva vagy általánosítva, az esetleges különbségekkel kiemelve.

## 11.2. Java és C++

Java: Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A Java nyelv jelölésrendszere nagy hasonlóságot mutat a C++-szal, mivel több minden átvett a nyelvből. Objektumorientált, valamint az osztálykezelés az alapja. A Java nyelv szintaxisa a C és a C++ nyelv felhasználásával alakult ki. Az utóbbi két nyelvvel ellentétben itt muszáj megjelölni a visszatérési típust. Az appleteknek az internetes felhasználás miatt lett nagy jelentősége, ugyanis ezek a HTML oldalba beágyazva futtathatóak. A változók deklarálása, típusai is hasonlóan történik a 3 említett nyelvben. Javaban szintén jelen vannak az operátorok, pl az = jel, amelynek egy felhasználása pl az értékadás egy változónak. A konstansok, vagyis állandók, illetve a megjegyzések jelölése szintén azonos a nyelvekben. C-ben nincs, de Javaban és C++-ban van osztálykezelés. Javaban a karakterláncok, vagyis a nem csak egy karakterből álló tömb kezelésére új osztályt hoznak létre (string). A new operátorral lehet objektumot létrehozni, ugyanis ez az operátor foglalja le a szükséges méretű helyet, majd inicializálja azt, és innentől kezdve referenciaiként használja, azzal tér ide vissza. Egyszerű típusok esetében lehet a new operátor nélkül is inicializálni. Azok az elemek, amelyeket static-kal deklaráltunk, azok az adott osztályhoz tartoznak, nem az objektumhoz. Javában nincs eszközünk ahhoz, hogy megszüntessük egy objektumot. Amikor Javaban egy metódust szeretnénk alkalmazni, akkor a szerkezetet egy class szóval kell bevezetnünk. A metódusok létrehozásánál is vannak bizonyos szabályok, amelyeket muszáj betartanunk. A kivételkezelés szintén fontos szerepet tölt be a nyelvekben, ugyanis ezek a megbízhatóság növelésére szolgálnak. Például ekkor érdemes használni a try-catch blokkot. Az AWT, vagyis az Abstract Window Toolkit-nek egy felhasználói felület összeállításában, megjelenítésében lehet például szerepe. Lehetőségünk van azonban (ha az AWT-vel nem vagyunk elégedettek) saját grafika írására is. Ha felhasználjuk a párhuzamosítási lehetőségeinket, akkor változó/mozgó képet érhetünk el a használt értékek szabályszerű változtatásával, illetve a metódusokon történő változtatásokkal. A párhuzamosítási szál a thread. A könyvben szereplő példa ezzel a szállal végzi az ábra forgatását. A Swing csomag az AWT-hez hasonlít, de sokkal több választási lehetőséget tartalmaz. A nyelvek karakterkészlete szintén sok hasonlóságot mutat, ugyanis az ASCII táblázatot használják, illetve a Unicode karaktereket. Javaban a típusok között lényeges különbség van, ugyanis nem minden a használat szempontjából, hogy primitív típusú a változó (pl int változó) vagy sem (pl az Integer típusú objektumhivatkozást tartalmaz). Literálokat kell használnunk, ha egyszerű típusokat vagy objektumokat szeretnénk inicializálni, erre több féle literál szolgál. Amikor változót deklarálunk mindenki a nevet kell adni annak, ez igaz a C++-ra és a Java-ra is. Értéket adni neki nem muszáj azonnal. Javaban a tömb egy típust jelöl, ebben különbözik a C++-tól, ugyanis nem csak a mutató jelölése másképp, de több dimenziós tömb viszont nem létezik, ha azt szeretnénk, akkor a tömbbe újabb tömböt kell tenni. Léteznek felsorolási típusok, amiket a tömbökhez tudnánk hasonlítani működés szempontjából (indexelés), sorszám szerint tudunk értéket adni ezeknek. Az operátorok szerepe az, hogy megadja a kiértékelés sorrendjét. A Java nyelv erősen

típusos, ezért van szükség típuskonverzióra, amely során megvizsgálja, hogy léteznek-e összegyeztethető típusok, vagy konverzióval azonos típusra lehet-e alakítani a különböző elemeket. Javaban egy struktúra egyik részelemének/tagjának eléréséhez a "." karaktert használhatjuk az elemek közötti elhelyezéssel. Itt nincs megkülönböztetve az osztályok elérése mint C++-ban, ott erre a ":" operátor szolgál. Az utasításokon belül 2 fajta utasítást tudunk megkülönböztetni. Ezek a kifejezés- és deklaráció-utasítás. Mindkettőt azonosan ; zárja, előbbi értékadásra, postfix vagy prefix képezésre, metódushívásra vagy példányosításra szolgál, még utóbbi egy lokális változó létrehozására/inicializálására. Ha Javaban elágazási szerkezetet szeretnénk készíteni, azt kétféleképpen tehetjük meg, az egyik az egyszerű elágazás, amelyhez egy if-szerkezet szükségez, még a másikhoz, az összetett elágazáshoz switch-szerkezet szükséges. A Java összesen 4 féle ciklust ismer, amelyekben a vizsgálati helyek változnak. Ezek az elől-, hátultesztelő ciklusok, a léptető (for/while) és bejáró (for) ciklus. Léteznek utasítások, amelyekkel a program vezérelhető, pl bizonos körülmények között megállítható, folytatható, vagy az adott ponton átugrik egy másik pontra (utasításra). Ezek a: cím-kék, break és continue utasítások, a visszatérés és a goto. A Java alapja az osztálykezelés, a legkisebb önálló egység benne az osztály. Az osztályok egyfajta modellezési szerepet töltnek be, egy rendszert épít fel. Példányok, objektumok és egyedek szerepelnek bennük, bár ezek a kifejezések ugyan azt jelentik, de lényeges tudnunk, hogy melyik osztályban vannak és ezt jelezni valahogyan. Az osztályoknak saját változó készlete van, így akár 2 különböző osztályban is használhatunk azonos változónevét pl. Az osztály egyik szerepe az egységbe zárás, tehát az adatok és műveletek összefogása. Biztonsági funkciója is van, pl adatrejtés. Ha osztály változójára hivatkozunk, akkor a változó neve előtti ponttal elválasztva megadhatjuk, hogy melyik más objektum változójára hivatkozunk. Ennek akár a metódusok működésében is lehet fontos szerepe. Ezeket a metódusokat metódusdefiníciók írják le, valamint fej és törzs részből áll. A fej előtt a módosítók állnak, a törzs pedig az utasításblokk. Ebben azonos a Java és a C++, hogy a törzs és a fej nincs külön választva. Ha metódushívást alkalmazunk ügyelnünk kell a paraméterlistában megadott dolgokra, ugyanis ezek kötött tényezők a használat során (típusoknak egyezni kell, és a paraméterszámnak). Ha példányra szeretnénk alkalmazni, akkor a metódusnévvel ellátott objektummal kell hívunk. A metódus zárójelezésére szintén ügyelni kell, ugyanis ha nincs paraméterlista is szükséges a () zárójelezés. Egy metódusnevet többször is használhatunk, csak a paraméterlistájuk ne egyezzen meg (paraméterek száma/típusa), ezt nevezzük metódusnevek túlterhelésének. A helyes használatot utána a Java fordító tudni fogja, a jót választja ki még a lefordítás idejében. Az objektumok a példányosítással hozhatók létre a new operátor használatával. A folyamat közben a new operátor memoriát foglal le, ahol az objektum változóit elhelyezi, és a memória kezdőcímét adja vissza. Az ezzel kapott referenciát a megfelelő osztály típusú változónak adható csak át. Ha már nincs szükségünk egy objektumra, akkor érdemes törölni azt. Itt különbség van a C++ és Java között, ugyanis mint sok nyelvben a C++-ban a programozónak kell törölni azt (így könnyen ütközhet hibába a használat miatt), még a Javaban viszont a program futás közben vizsgálja, és ha már nincs rá szükség akkor törli magától, ez a szemétygyűjtő mechanizmus. Az osztályoknál figyelnünk kell a hozzáférhetőségekre. Erre szolgál pl a private, public vagy protected hozzáférési kategória, tehát a láthatóságra. Ez alapján több csoportra bonthatjuk szét: félnyilvános, nyilvános, privát vagy leszármazottban hozzáférhető tagok. Az osztály szintű tagokat példányváltozóknak nevezzük. Az osztályváltozók és metódusok más néven statikus tagok elő a static módosítót helyezzük. Ezek az osztályokhoz kapcsolódnak. Az osztályváltozók és a példányváltozók inicializálása is az előfordulási sorrend szerint történik, de az előbbi kezdőértéket csak egyszer kap, még utóbbi minden meghíváskor inicializálódik. Ugyan úgy hivatkozunk rájuk. Az osztálymetódus az az osztályon belüli műveletet takarja, ehhez csak az osztályváltozók használhatóak fel. Az osztályokon belül konstruktorkat és destruktorkat használunk. A destruktorknak feladata a törlés, amiről nemrég írtam, hogy C++-ban a programozók (röviden és tömörén) ennek a segítségével szüntethetnek meg meg. A konstruktur lényegében példányosítás, de a hibalehetőségek száma kisebb. A konstruktordefiníció leginkább a metódusdefinícióhoz hasonlítható. Ha példányosítani szeretnénk, a new operátor számára paraméterek is megadhatóak, ezek a paraméterek azok amelyeket a konstruktornak sze-

retnénk adni. Öröklődést pedig úgy tudnám jellemezni, hogy egy osztályt egy másik osztállyal bővítsük. Ezután a szülő osztályon keresztül elérhető a gyermek osztály, valamint a gyermek osztály öröklí a szülő tulajdonságait a létrehozásakor. A gyermek osztály lényegében a szülőnek a kibővítése. Viszont a szülő megadhatja, hogy egy metódusa például public vagy private. Ami private, azt a gyermek már nem látja (ha public vagy protected, akkor látja a szülő metódusait). Ez igaz a Javára és C++-ra is. Az osztályhierarchiáról, metódustúlterhelésről és hatáskörről korábban már írtam úgy, hogy az minden nyelvre igaz legyen, így áttérnék a többire. A polimorfizmusban létezik egy szülő osztály, és azon belül lehet bármennyi gyermek osztály, amelyek öröklík a szülő tulajdonságait. Ha a szülő osztályt példányosítjuk, akkor utána polimorfizmussal a szülőn keresztül elérhetőek a gyermekek. Például a példányosított szülő osztályban létrehozunk egy objektumot, és a szülőn belül vannak (extend), leszármaztatott gyermek osztályok. Ezeket a gyermeket pedig akár öbbet is bele tölthetünk az említett adott objektumba. Emiatt nevezzük a polimorfizmust más szóval többletölgységnek, vagyis inkább ha lefordítjuk a szót. A polimorfizmus során a gyermeket kaphatnak még azonos metódusokat, de ezeket külön-külön kezelik, hiába azonos a metódus (mintha a két gyereknek ugyanaz a jellemzője lenne más tulajdonságú). Ha absztrakt osztályokról és az interfészkről beszélünk, akkor azonnal eszünkbe juthat, hogy vannak absztrakt függvények és metódusok is. Ha ezeket szeretnénk használni, akkor a definiálásuk során használhatjuk az abstract vagy public kulcsszót, de ezek nem kötelezők. Javaban az interfész nem tartozik már az osztályokhoz. Ez lényegében az előbb említett absztrakt metódusokat és konstansokat használja. Az interfész lényegében egy felület, és a bennatalálható metódusok csak deklarálva vannak, kifejtve nincsenek. A interfészeken belül is létezik öröklődés, a szülő-gyermek kapcsolat. Az absztrakt és az osztályok metódusai között annyi különbség van a deklarációjukban, hogy az absztraktnak nincs törzse. C++-ban az absztrakt osztály pedig tulajdonképpen az interfészben megjelenő műveletek és objektumok felsorolása. Ahogy olvastam a 2 könyvet én úgy vettet észre, hogy a lényege és a főbb összetevők (pl absztrakt metódusok) azonosak, a megvalósításukban látok különbséget (szerintem a C++ itt bonyolultabb). Az egységbe zárás pedig a date hiding, vagyis rejtés lényegében, de arról korábban már írtam. A következő fejezetről nem beszélnék sokat, hiszen az egyik feladat lényegében az egészet feldolgozza (14.3). Az alapján írok egy rövid összefoglalót. Tovább mennék, és röviden fogalmaznák. Az első a karakterkészletek. Általában ASCII karakterkészletet használunk, bár létezik néhány ékezes betű amit nem tartalmaz a készlet (mivel 8 biten tárol). Ha ezt szeretnénk kikerülni, mert szükségünk van azokra is (pl angol nyelvezetnél felesleges), akkor a unicode-ot érdemes használni. Következőként a lexerek jönnének, de ezt nem szeretném itt kirészletezni, mert a 1334d1c4 feladat (15.2) tartalmazza a választ. A kifejezések-ről és utasításokról pedig írtam az előzőekben, még valahol az olvasónapló eleje körül, ezért nem fejteném ki azt is újra. Ha jól emlékszem korábban erről isvolt már szó, de inkább írom most. Javában léteznek primitív és nem primitív változók. Ami nem primitív, az már lényegében itt osztályként alkalmazható, a primitívnek pedig összesen 8 típusa létezik, ezek: int, long, boolean, float, char, double, short, byte. Tehát így nézne ki a Java típusrendszere. A következő a konstansok. Ez megtalálható Javában és C++-ban is egyaránt, különbség, hogy C++-ban osztályon kívül is alkalmazható. Amennyiben viszont osztályon belül van, akkor meg kell határoznunk, vagyis inicializálni kell. Ezzel adhatunk a konstansnak egy kezdeti értéket. A nevéről is lehet következtetni, a konstans egy állandó változót jelent lényegében (bár ponyola fogalmazással). Az, hogy osztályon belül hol hoztuk létre mindegy, a program tudni fogja a sorrendet. Használhatunk még operátorokat, ezek pedig akkor jutnak szerephez, ha műveleteket szeretnénk végrehajtani. Léteznek bizonyos esetek, amikor a kívánt cél eléréséhez muszáj túlterhelni az operátorokat. Az indexelőkről nem először a tömbök jutnak az eszembe, hiszen a [ ] zárójelek között hivatkozhatunk akár egy tömb index számára. Ami itt újdonság lehet (szerintem), az az osztályok és objektumok indexelhetősége, ugyanis erre is van lehetőség, szintén [ ] zárójelek között. Ha jól emlékszem szintén volt már szó a konstruktorkról és destruktorkról, de mivel nem vagyok benne biztos, inkább írom itt is. Mindkettőre az osztályoknál van szükségünk, és bizonyos szabályokat követnek. A konstruktornál ezek a bizonyos szabályok megmondják, hogy mik a feladatai, ezek: a konstruktort az osztály elején kell meghívni, és ezzel megadunk egy

kezdeti állapotot, ezt nevezzük példányosításnak, amikor az osztály meghívódik. A neve pedig azonos az osztályéval, amiben szerepeltek jükk. Lényegében ez erőforrást és memóriát foglal le, ezért kell meghívni a destrukturát, ha már nincs rá szükség. Ezzel megszüntetjük az előbbieket, illetve az objektumokat (aminek pl hely volt foglalva), használata pedig egy hullámvonal (~) amit az adott osztály neve követ. (A rossz használata pl memória folyást okozhat.) A beágyazott típusok pedig a beágyazott osztályok lényegében, tehát egy osztályba beültetett/beágyazott másik osztály. Ezt szebben nem tudom megfogalmazni. Térjünk át a következő fontosabb témakörre. Az interfések majdnem egyenlőek az absztrakt osztályokkal, nincs nagy különbség. A normál osztályokhoz képest viszont különbség, hogy nem tartalmaz tagváltozókat és a metódusok kifejtését, vagyis implementációját sem. Használni pedig az implementáció keretében lehet. A következő téma a kollekciók. Más néven gyűjtemények. Ezek a legtöbb nyelvben használhatóak. Javaban létezik egy kiindulója, ami az összes többi alapjául szolgál. A feladata az objektumokkal való dolgozás, tehát például a lekérdezések, és ezeket a típusai szerint (1/több) elhelyezze a gép memoriájában. Ha meg kellene említeni a leggyakrabban gyakorlatban is felmerülő példát, akkor akkor az a összekapcsolás, vagyis tömbök és objektumok összekapcsolása referencia vagy pointerrel (Java/C++). A funkcionális nyelvek lényegében egyenlőek egy speciális függvényvel, mivel a programot egy darab függvényként kezeli, amit végül kiértékel. Egyszerű, de a kapott paraméterek alapján ad vissza nekünk egy értéket végeredményként. A lambda kifejezések pedig olyan metódusok, amelyeknek nincs neve, és csak ott szerepeltek jükk, ahol aktuálisan szükség van rá, nem kell előre deklarálni, de emiatt csak egyszer felhasználható. Imperatív nyelvekben megtalálható, ami itt minket érdekel az a Java és C++, de a feladatokban lesz példa rá. Akkor jöhét megint a következő témakör. Első az adatfolyamok kezelése és a streamek. C++-ban ezek lényegében a byte sorozatokat fedik. Az iostreamet emiatt szinte minden meghívjuk, hiszen írás olvasáshoz szükséges. Ez fedi az input és az output streamet. Az írás olvasáshoz pedig erre a két operátorra van szükségünk:

```
<< és >>
```

. Javaban a különbség, hogy itt ezek is objektumok. Emiatt pedig 3 különböző osztálytalálkozás lehet használni ezeket, a felhasználási módok szerint elkülönítve, ezek pedig: adattípusok szerint, az adatfolyam irányá szerint és funkciójuk/feladataik szerint. Állománykezelésnél a C++ most is adatfolyamokban gondolkozik, emiatt pedig az ifstreamre és az ofstreamre van szükségünk, ezek tehát a be- és a kimeneti adatfolyamok, de létezik egy fstream is, ami kétirányú "forgalmat" biztosít. Javaban ez teljesen más, alapjáraton hasonló implementációt nem is ismer. Ha ilyesmit szeretnénk, akkor egy rendszerfájl kell használnunk, a random acces filet. Ami előnye viszont, hogy nincs határozott sorrend, bárhonnan ír és olvas, ezt viszont logikus, hogy egy mutatóval végzi, hiszen anélkül nem lehetne bárhonnan írni olvasni. Szerializációjánál az adatfolyamokba objektumokat ültethetünk. Ennek során pedig bájtsorozatokat tudunk képezni ezekből. Osztályokat is lehet szerializálni, de akkor szükséges, hogy az tartalmazza a "serializable interfész". Viszont ha ez így szerializálható, akkor a gyermekje is. A következő szakasz a kivételkezeléssel folytatnám. Ez lehetőséget nyújt arra, hogy ha hibába ütközik a program, akkor átugorjon a hibakezelésre. Azonban a nevéről is kikövetkeztethetjük, hogy más célja is van, ugyanis szándékosa is hozhatunk létre ilyen helyzeteket, ha azt szeretnénk, hogy egy bizonyos esemény bekövetkeztekor (ami nem hiba) is a hibakezelő részre ugörjen. Ezt szintén megtalálhatjuk mindenkit tárgyalt programnyelvben, csak a leírásukban, vagyis megírásukban van pici különbség a nyelvek különbségéből adódóan, de funkcióban azonosak. Ezt exceptionnek nevezzük. Javaban eddig a try-catch szerkezettel találkoztam, ami a hibakezelés leírására szolgál. Érdemes megjegyezni, hogy hiba esetén a hibakezelő rész futása után nem ugrik vissza korábbra segova, hanem folytatódik azt azt követő részzel. A nyelvi elemek közül találkozhatunk (legalábbis eddig) annotációkkal és attribútumokkal. Annotációra a példa, amikor egy kuckac jel mögött láthatjuk az override-ot. Gyakorlatilag ennek nincs befolyásoló hatása, csak plusz adatokkal szolgál. Tehát a kód működésén nem módosít, egyfajta megjegyítések tudnám nevezni ezeket az adatokat. Az attribútumokról pedig még a napló elején volt szó, így azt nem venném elő újra (még a szétszedett részben is, illetve az összevonásban szintén). Ezzel pedig az utolsó

szakaszhoz érdeztünk, a multiparadigmás nyelvekhez, és az ebben történő programozáshoz. A paradigmák határozzák meg azt, hogy egy programozási nyelv milyen is magában. Például eszközök, fogalmak, a nyelv jellemzői, stb... Ezek általában önmagukban működnek jól, de a gyakorlatban viszont sokszor többre is szükség lenne egyszerre. Ennek a kiküszöbölésére szolgálnak a multiparadigmális nyelvek, amelyek ezt képesek megoldani. A programozásban történő felhasználása terén vannak bizonyos szabályok, amelyeket követni kell, de eközben sokszor mégis lehetőséget ad arra, hogy a programozók a saját stílusuk szerint írják meg a programjaikat, és a hibák elkerüléséhez pedig a paradigmákat is fel tudják használni.

## 11.3. Python

Python: Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba. Gyors prototípus-fejlesztés Python és Java nyelven (35-51 oldal)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A Python fejlesztők számára alakították ki, általános célú programnyelv, amely magas szintű, dinamikus, objektumorientált és platformfüggetlen. A Python alkalmas összetettebb feladatok illetve problémák megoldására, mert sok csomagot és beépített eljárást is tartalmaz és támogatja a magas szintű típusokat. A C, C++ és Java nyelvekkel ellentétben, itt nincs szükség külön fordításra, az értelmezőnek csak a Python kódot kell megadni a futtatáshoz. Inkább nevezhetjük ezek miatt programozási nyelvnek, mint szkript nyelvnek. A kódkönyvtár rengeteg előre megírt modult tartalmaz, ezzel sok időt spórolva a programozóknak (pl: fájlkezelés, hálózatkezelés, rendszerhívások). A nyelv jellemzője, hogy könnyű vele rövid, de tartalmas, velős kódot írni (gyorsítja pl: magas szintű adattípusokkal rövid összetett kifejezések írása, egyszerűbb a tagolás mint C++-ban vagy Javaban, nincsenek zárójelek és nem szükséges a változók és az argumentumok definiálására). A zárójelek helyett a tagolása behúzásokkal történik meg, annak az egységére viszont figyelni kell. Például egy adott blokk végét egy üres sorral tudjuk jelölni. A Python értelmezője a sorokat ún. tokenkre bontja, amelyek lehetnek azonosítók, kulcsszavak, operátorok, delimiterek vagy literálok. Pythonban is vannak előre lefoglalt kulcsszavak, ezeket a könyv egy táblázatban felsorolja. Itt a típusok ábrázolása is másnéhány működik, az adatok objektumként tárolódnak, és a műveletek is azoknak a típusától függ. A Python csak számokat (azon belül egész, lebegőpontos vagy komplex), sztringeket, enneseket, listákat és szótárakat ismer adattípusként. Ha sztringeket szeretnénk írni, akkor az " " idézőjelek között tehetjük meg, de a unicode-ot is ismeri. Itt is van NULL érték, de itt None-nak nevezzük, de tökéletesen alkalmazható boolenként. Ha egy gyűjtemény None elemű, akkor True, másnéhány False. Pythonban objektumokra mutató referenciakat jelentenek a változók, de az értékkedésük szintén "=" jellel történik meg, akárcsak más magas szintű programozási nyelvekben, és ilyen módon vannak jelen a lokális és globális (igaz utóbbinál lényeges, hogy a függvény előtt legyen global kulcsszóval, de működés szempontjából azonos) változók. A szekvenciákon különböző műveleteket is végezhetünk, valamint indexeléssel érhetjük el az elemeit. Ha ":" jelet használunk, akkor azzal a szekvencia számára egy intervallumot tudunk megadni. Pythonban szükségünk van kulcsokra, ha a szótárakról beszélünk, ugyanis annak elemeit a szögletes "[ ]" zárójelekbe beleírt kulcsokkal lehet elérni. Pythonban a print metódus szolgál kiíratásra (tehát konzolra, stdout), de a tokeneket amiket szeretnénk megjeleníteni vesszővel el kell választani. Az if elágazás és a for ciklus szintén hasonló működésű a többi nyelvhez, bár a for ciklussal itt a kulcsokon is tudunk műveletet végrehajtani, vagy kikeresni egy kulcsot például a hozzá tartozó értékkel együtt. Még 2 függvényt érdemes megemlíteni, a range függvényt (alapesetben a range(x) 0-tól x-ig számol, de intervallum is megadható neki) és az xrange,

amely hasonló, de kevesebb memóriát foglal. A while ciklus pedig a feltétel teljesüléséig működik. Léteznek címkék, amelyeket a label kulcsszóval tudunk megadni, és a goto utasítással pedig a ponthoz ugorhatunk. Amennyiben magunk szeretnénk egy függvényt definiálni, akkor azt a def kulcsszóval lehetjük meg. Ezeket nézhetjük úgy is, mintha értékek lennének, hiszen továbbadhatóak. Azonban paraméterekkel is elláthatjuk őket, és a paraméterek is átadhatóak, csak (kivétel a mutable típus) érték szerint kell kezelni. Pythonban is van lehetősége a programozóknak objektumorientált fejlesztésre, programozásra. Ennek következménye, hogy használhatunk osztályokat, amelyeknek az objektumok a példányai, attribútumnak pedig az objektumokat vagy függvényeket (tagfüggvény/metódus más néven) nevezzük. Ha egy osztály attribútumát megváltoztatjuk, akkor az az összes többi példány attribútumára hatással lehet, ha csak nem módosítottunk azokon is. A Pythonban sok modul áll a rendelkezésünkre, tehát nem csak a szabványosak, hanem a mobilfejlesztéshez is tartalmaz modulokat (pl mobilhálózat, kamera, stb). Létezik kivételkezelés is, ami viszont különbözik a Javától és C++-tól. A try kulcsszó után kerül a blokk, majd ezután az except blokk következik, ahova hiba után átkerül az irányítás. Az elágazásokhoz fűznék még annyit, hogy Pythonban létezik az "elif" kulcsszó, ami az else if itteni verziója.

## 12. fejezet

# Helló, Arroway!

### 12.1. 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) Ugyanezt írjuk meg C++ nyelven is! (lásd még UDPROG repó: source/labor/polargen)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A magas szintű programozási nyelvekre jellemző az OO szemlélet, vagyis az objektum orientáltság. Ez jelen van a C++-ban és a Javaban is, melyek az osztályokban objektumokkal dolgoznak. A polartranszformációs algoritmus lényegében a tárolt adatokról szól. A program generál 2 számot, majd elkezdi vizsgálni a nincsTarolt változót. Ez egy boolen típusú változó, amely true vagy false, tehát igaz hamis értékeket tárol. Ami miatt szükség van rá: ha nincs tárolt szám, akkor a 2 generált értékből az egyiket letárolja, a másikat pedig visszaadja. Amennyiben pedig ha már korábban is volt tárolt érték, akkor azt adja vissza a program. Ezek után pedig a program minden negálja a nincsTarolt változót, tehát az ellenkezőjére állítja (true -> false, vagy false -> true), így minden változni fog, hogy melyik értéket adja éppen vissza. A negálás művelet miatt viszont csak minden második alkalommal fog lefutni a program (ezt a következő metódus hajtja végre), ez a true érték miatt történik, tehát ha a nincsTarolt éppen true, az if ág akkor fog elindulni, másképp az else ágra ugrik át.

Ha érdekel minket, akkor a JDK-ból is kinyerhető a program egy kicsit másabb formában, de a lényege ugyan az: /jdk/src/share/classes/java/util és itt a Random.java fájl.

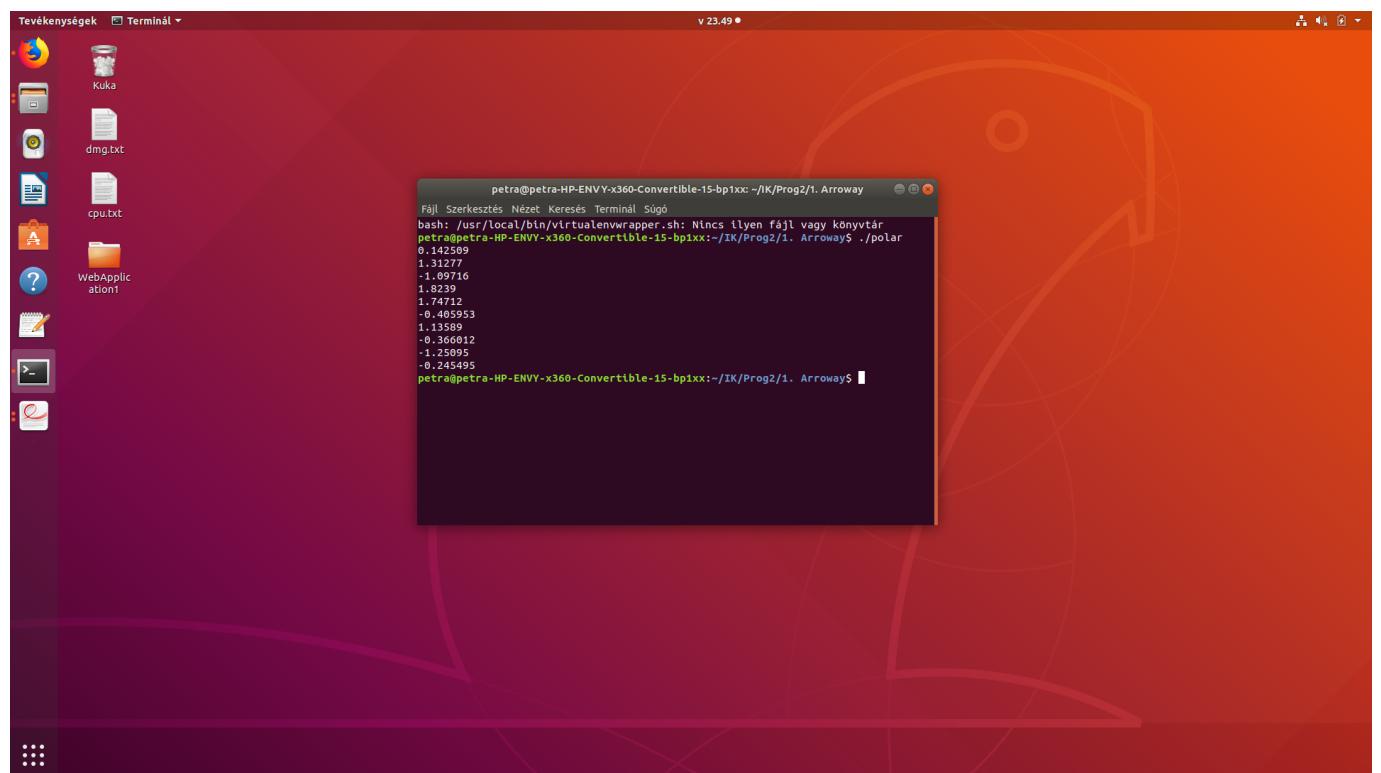
A Java csipetek amiről írtam: az elsőben a minden második futás, illetve a 2 random szám generálása, illetve a negálás és az érték amit visszakapunk.

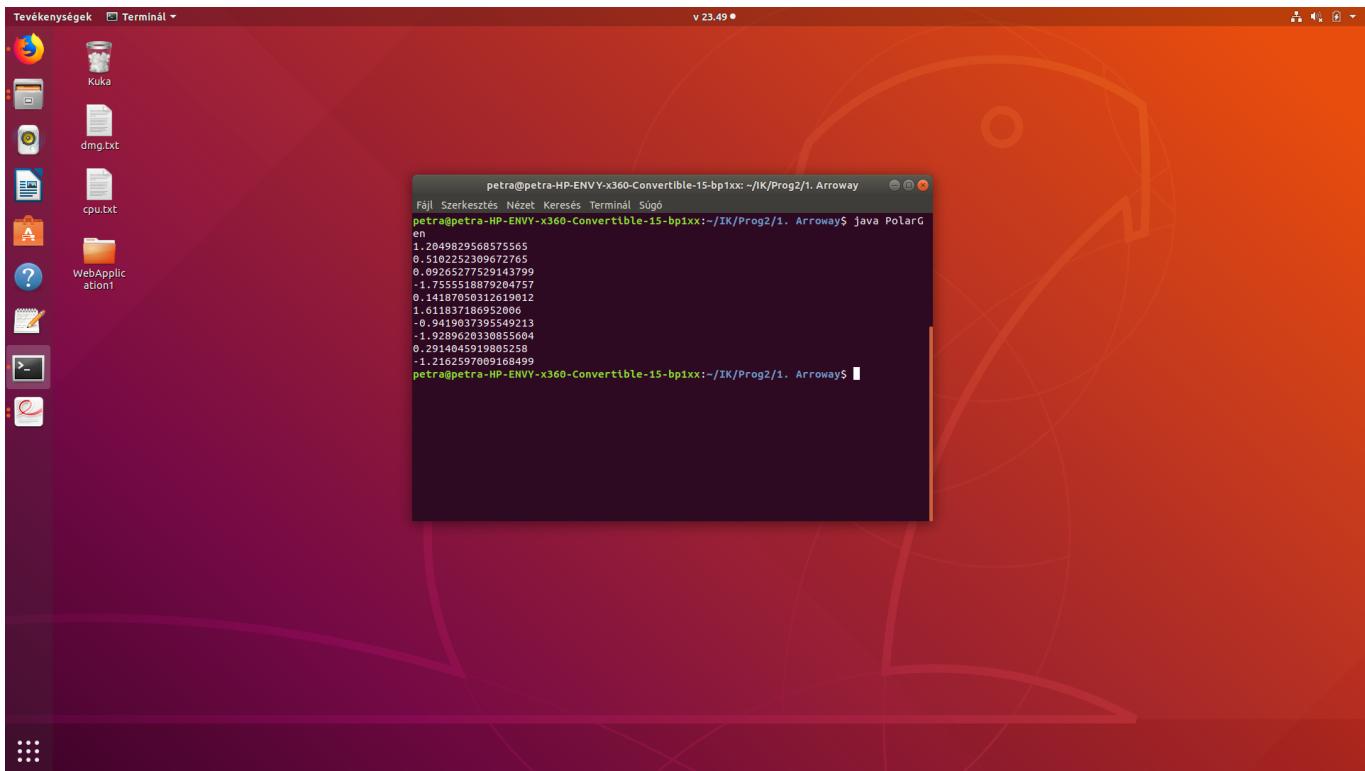
```
if (nincsTarolt) {

 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;
}
```

```
else
{
 nincsTarolt = !nincsTarolt;
 return tarolt;
}
```





## 12.2. Homokozó

A feladat megoldásánál tutorom volt: Tóth Attila

Írjuk át az első védési programot (LZW binfa) C++ nyelvről Java nyelvre, ugyanúgy működjön! Mutassunk rá, hogy gyakorlatilag a pointereket és referenciákat kell kiirtani és minden másik működik (erre utal a feladat neve, hogy Java-ban minden referencia, nincs választás, hogy mondjuk egy attribútum pointer, referencia vagy tagként tartalmazott legyen). Miután már áttettük Java nyelvre, tegyük be egy Java Servletbe és a böngészőből GET-es kéréssel (például a böngésző címsorából) kapja meg azt a mintát, amelynek kiszámolja az LZW binfáját! (Tavalyi prog2 első védés volt.)

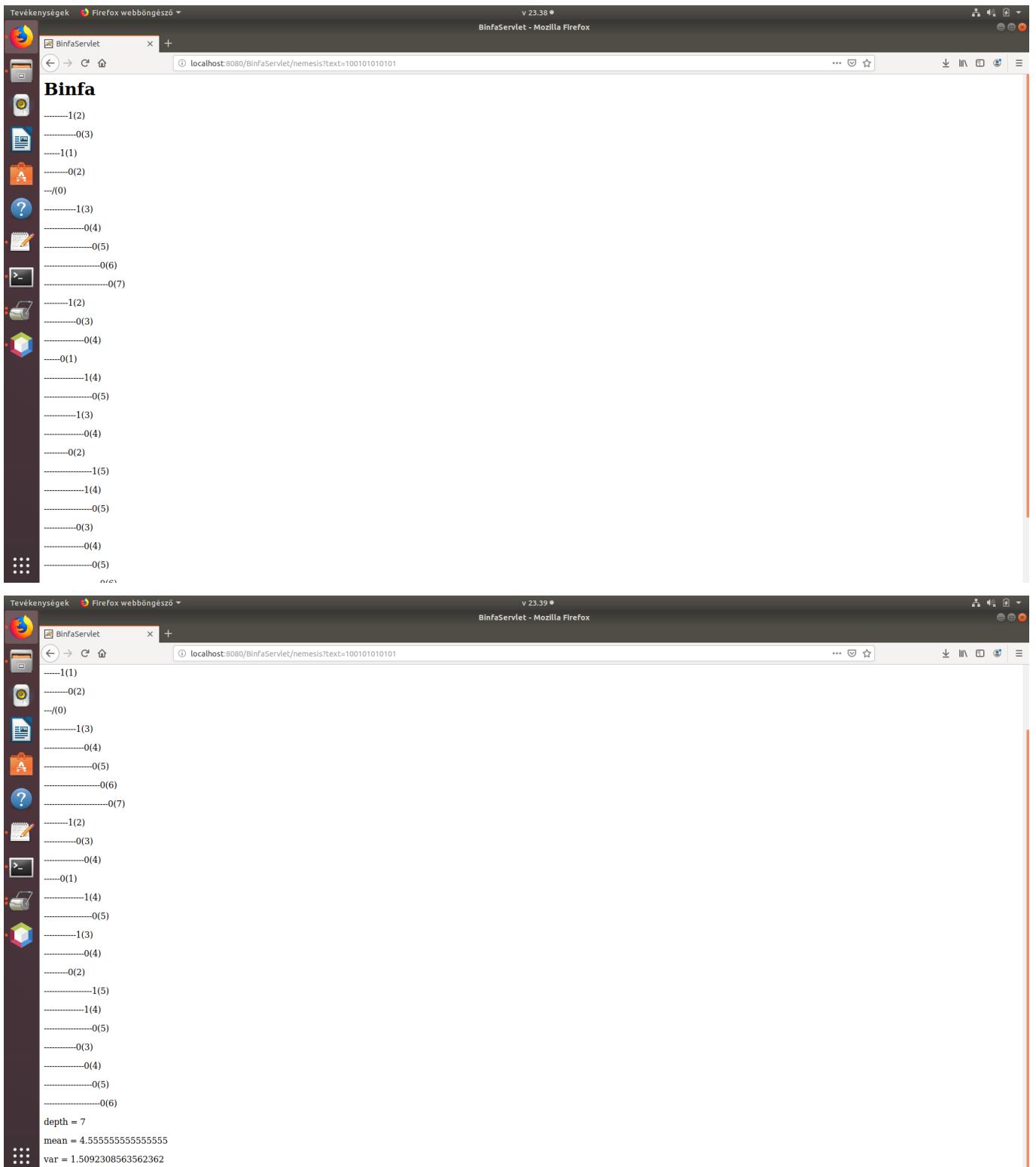
Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A feladat alap egysége megegyezik a Welch fejezet feladatával, vagyis az LZWBinfa felépítésével. Az akkori feladat aprogram C++-beli megalkotása volt, a mostani Javával ellentétben. A Java nyelv osztálykezelésére épül, és az objektumorientáltságot is megfigyelhetjük, akár csak a többi magas szintű programozási nyelvben, valamint szerintem az olvasónaplóban is erről olvashattunk. A két nyelv közti különbség miatt lehet érdekes a program átírása egyik nyelvről a másikra, ugyanis Javában nincsenek pointerek. A Java objektumokat ismer, és az értékeket csak referenciaként tudja átadni. A binfa működését nem írnám le újra, mert a lényege és a feldolgozás alap módszere nem változott a korábbihoz képest. Az átírás során viszont az osztályokra és a függvényekre ügyelni kell, hogy minden helyesen legyen megadva (private, public például). Itt a binfa gyökere, vagyis az az objektum ami fix marad az egész program alatt, az a root lesz. Mindig ezt fogja visszaadni a Node függvény miután új értéket szűrt be. Az érték beszűrését pedig a write függvény végzi el. Ahhoz, hogy a csomópontok által megadott értékeket megtudjuk, get függvényt kell használni, az

értékek beállításához pedig set-et. Szükségünk van egy doGet függvényre a http servlet miatt. A feladat megoldásához tomcatre volt szükség. A feladat megoldásához pedig a böngésző címsorából kell beolvasni a bemenetet. A szöveget ahogyan a képen is látszik a "?text=..." részről olvassa be. Emiatt volt szükség a html-re is. A kódokat pedig a hosszúságuk miatt nem szúrnám be ide, linkről elérhetőek lesznek.



## 12.3. „Gagyí”

Az ismert formális

```
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 referenciaja) „mélyebb” választ, írj Java példaprogramot mely egyszer végtelen ciklus, más x, t értékekkel meg nem! A példát építsd a JDK Integer.java forrására 3, hogy a 128-nál inkluzív objektum példányokat poolozza!

Megoldás videó:

Megoldás forrása:

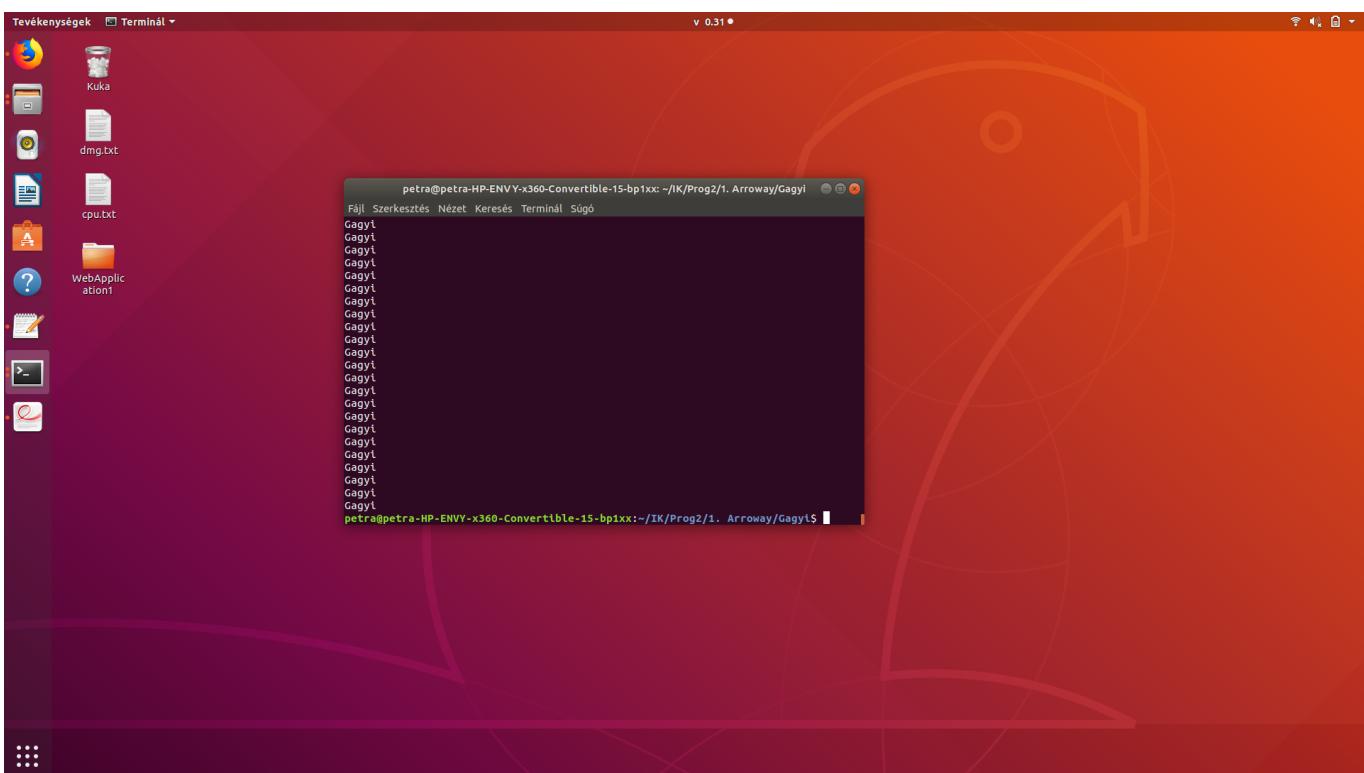
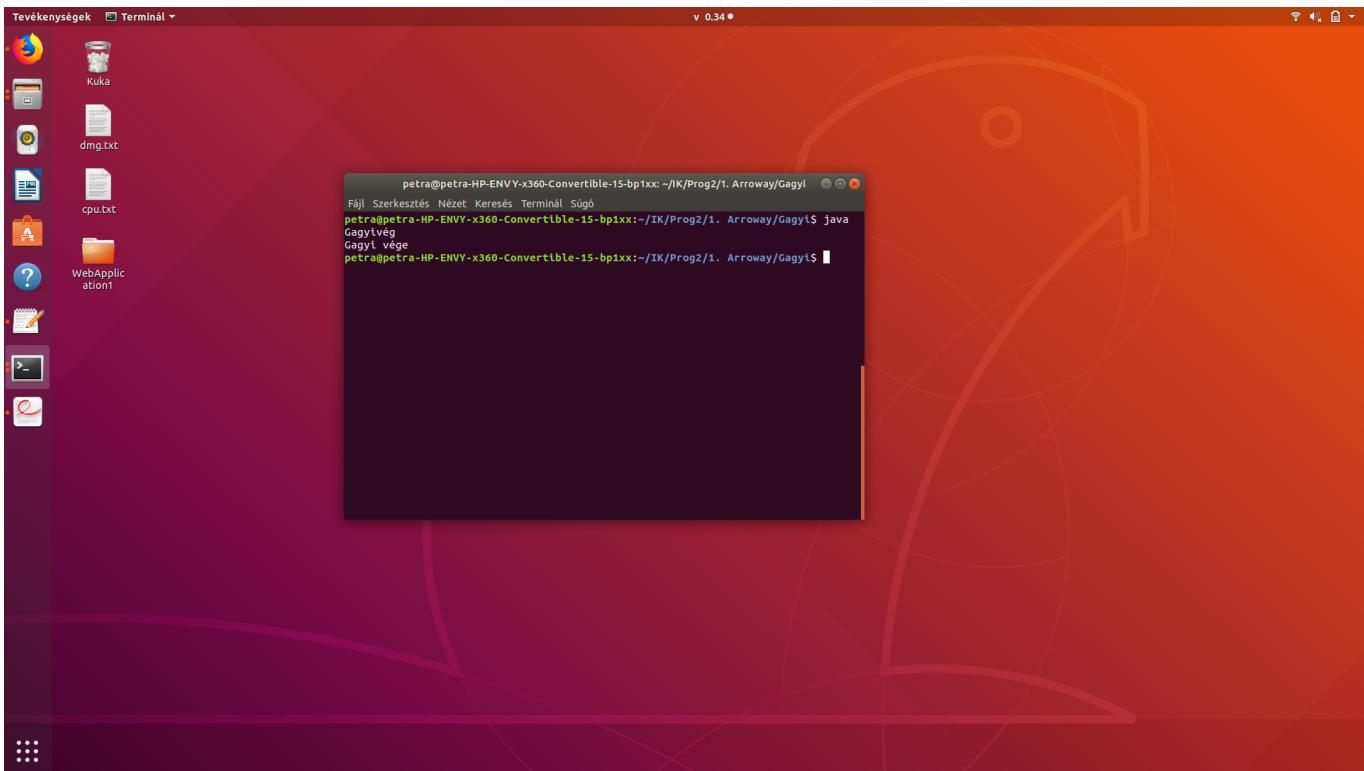
Tanulságok, tapasztalatok, magyarázat...

Ahogy látom ez a feladat a Java osztálykezelésére és az objektumaira helyezi a hangsúlyt, mint ahogyan azt az olvasónaplóban is olvashattuk. Az Intiger osztály -128 és 127-es érték között pooloz, vagyis ezt készítí elő a poolban. A Java nyelv készítői úgy gondolták, hogy ez elég lesz a fejlesztők számára, itt fog megmutatkozni a 2 bemutatott pici program közötti különbség. Amennyiben az említett érték közötti értéket adunk meg, akkor a program simán lefut, mivel a ciklus feltétele teljesül, kilép, és visszakapjuk a kívánt eredményt. Amennyiben viszont ha az érték kívül esik a -128 és 127 közötti tartományon, akkor az Intiger osztály egy új objektumot fog létrehozni. Ennek a következménye pedig, hogy a while ciklus false értéket fog kapni, ekkor nem fut le a program, hanem átcsap egy végtelen ciklusba. A következő kódcsipet pedig a "new" operátor értelmét mutatja meg, ugyanis ha poolozunk, akkor két különböző objektum azonos referenciát kapna, viszont ha használjuk a new-t, akkor a probléma megszüntethető.

Ha érdekel minket, akkor a JDK-ból is kinyerhető a program egy kicsit másabb formában, de a lényege ugyan az: /jdk/src/share/classes/java/lang és ebben a mappában az Intiger.java fájl tartalmazza.

```
Integer x = new Integer(-127);
Integer t = new Integer(127);
while (x <= t && x >= t && t != x)
 { System.out.println("Gagyí"); }
System.out.println("Gagyí vége");
```

A képek pedig prezentálják, hogy ha megfelelő a tartomány akkor lefut a program, viszont ellenkező esetben végtelen ciklust kapunk.



## 12.4. Yoda

Írunk olyan Java programot, ami java.lang.NullPointerException-ot leállít, 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:

Tanulságok, tapasztalatok, magyarázat...

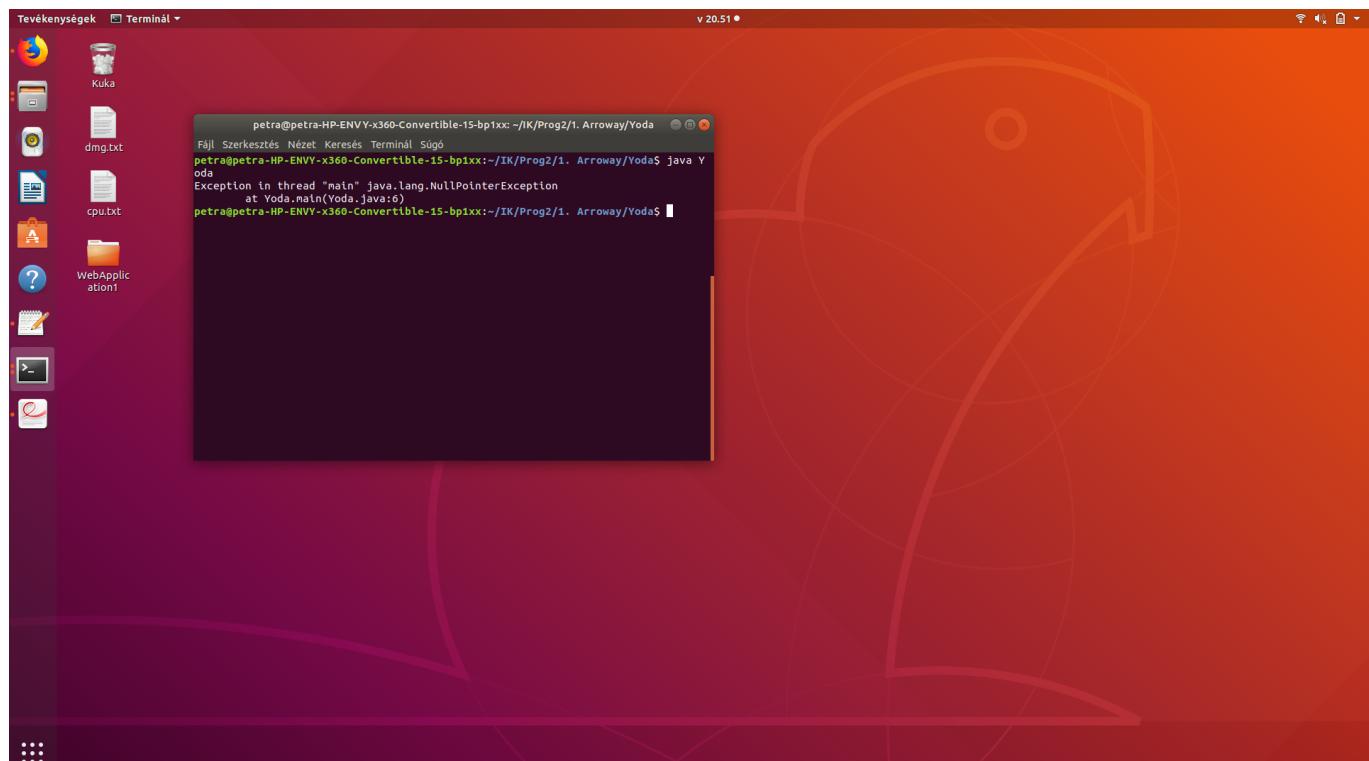
A képek segítségével szeretném bemutatni, hogy mi történik, ha nem követjük a Yoda conditionst, amelyet akár egyfajta szabálynak is nevezhetünk. Ez ténylegesen egy programozási stílus, ahol a fordított sorrendben van a hangsúly. A neve szerintem találó, hiszen a filmekben Yoda is tényleg fordított sorrendben beszélt. A programban használunk egy sztringet, aminek értéket adunk, jelen esetben null értéket. Ezt követően pedig ha az érték, vagyis a sztring a bal oldalra kerül és egy konstans pedig a jobb oldalra, akkor a program java.lang.NullPointerException-t generál, ha pedig a konstans a bal oldalon van, és a jobb oldalon az equals()-on (vagyis értéken belül) áll a null (vagyis a sztring), akkor lefut a program a Yoda conditions szerint, itt az if egy 0-ás értéket hasonlítana össze és leáll.

Az első képen a hibaüzenet látható, a második képen pedig a helyesen működő program.

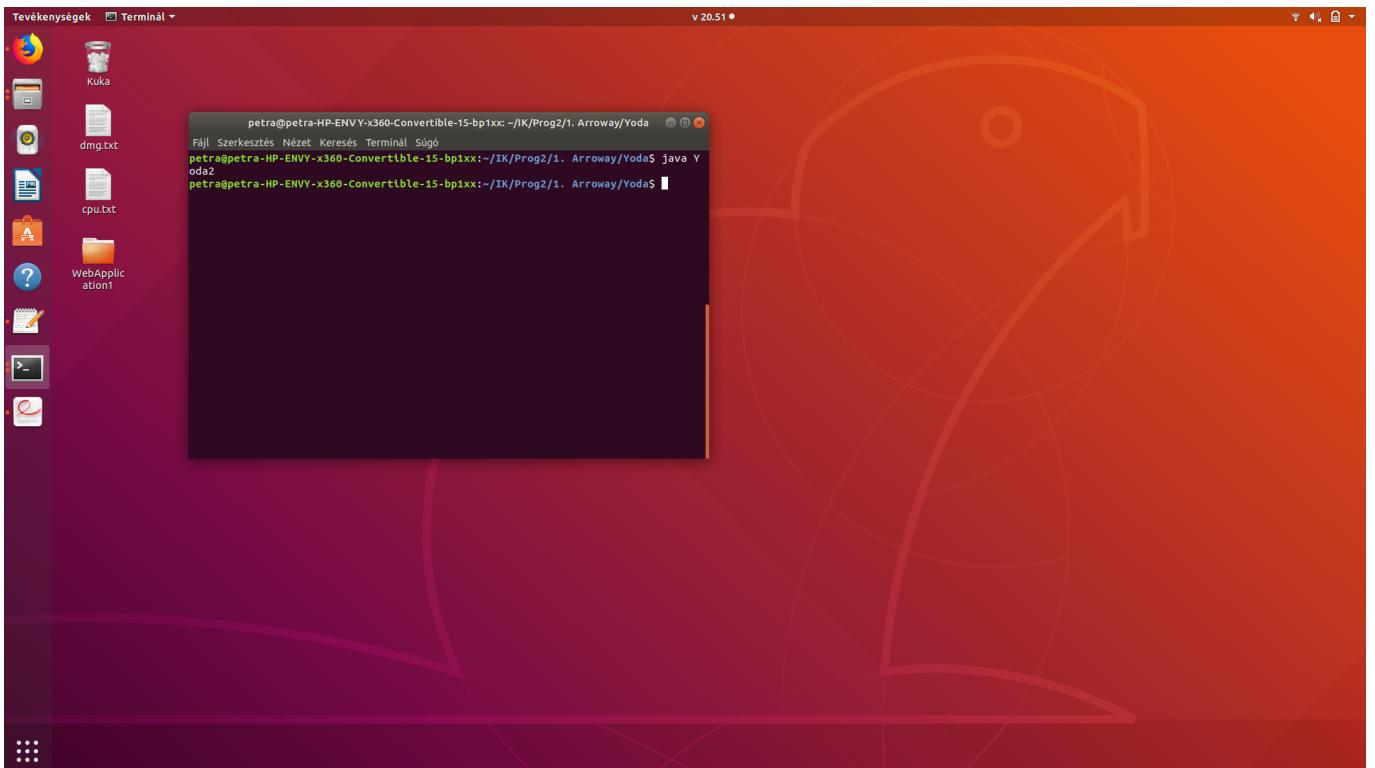
```
public class Yoda {

 public static void main(String[] args) {

 String r = null;
 if(r.equals("yodapélda")) {
 System.out.println(r);
 }
 }
}
```



```
String r = null;
if("yodapélda".equals(r));
}
```



## 12.5. 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 szeretném, ha átélnél).

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Sajnos számonra még nehéznek számított a feladat, ezért segítségül vettetem Tanár Úr kódját. Szükséges a feladathoz matematikai háttértudás, de ennek a pótlása megoldható. A program lényege, hogy miután lefut, kiírja a pi szám néhány számjegyét, azt viszont hexadecimális formában kell tennie. A BBP algoritmus szolgál arra, hogy a kívánt számokból hexadecimális számok legyenek. Az algoritmus a PiBBP osztályon belül van, ahol a BBP létrehoz egy objektumot, ami a Pi-től függ. A számolás során szükség van maradékos osztásra, erre szolgál a mod (vagyis modulo). Az eredmény visszaadására lebegőpontos típusú változó szolgál. Itt az első 6 pontosat kapjuk vissza, de ha módosítjuk a 1000000 számot a példányosításnál, vagyis a d változót, ami PiBBP konstruktorban van, akkor változtathatunk a visszaadott hexadecimális eredményen. Jelen esetben a megadott d értékkel a pi 1 millió+1-edik elemét számíthatjuk ki.

A következő kódcsipet a moduloval történő számításokat tartalmazza, vagyis a maradékos osztást. Ezeket az n16modk metódus tartalmazza.

```
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;

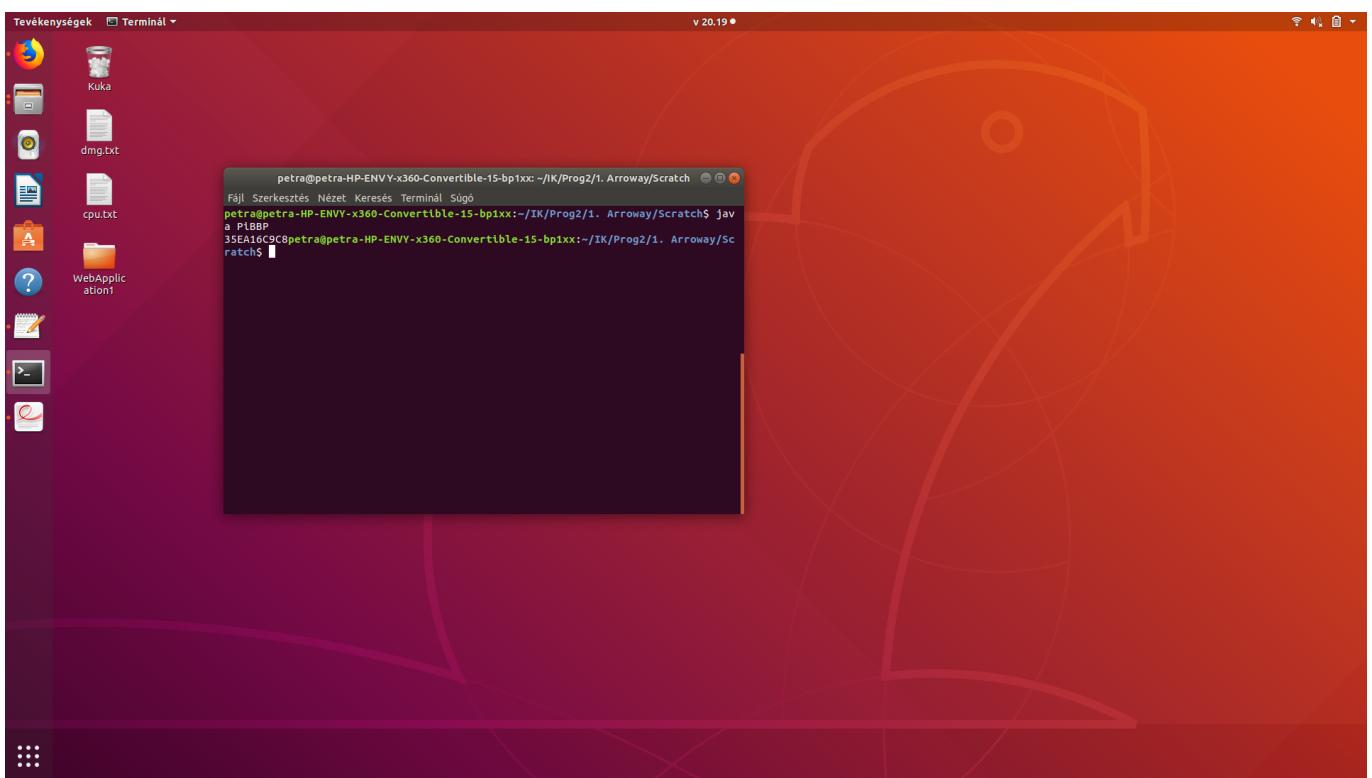
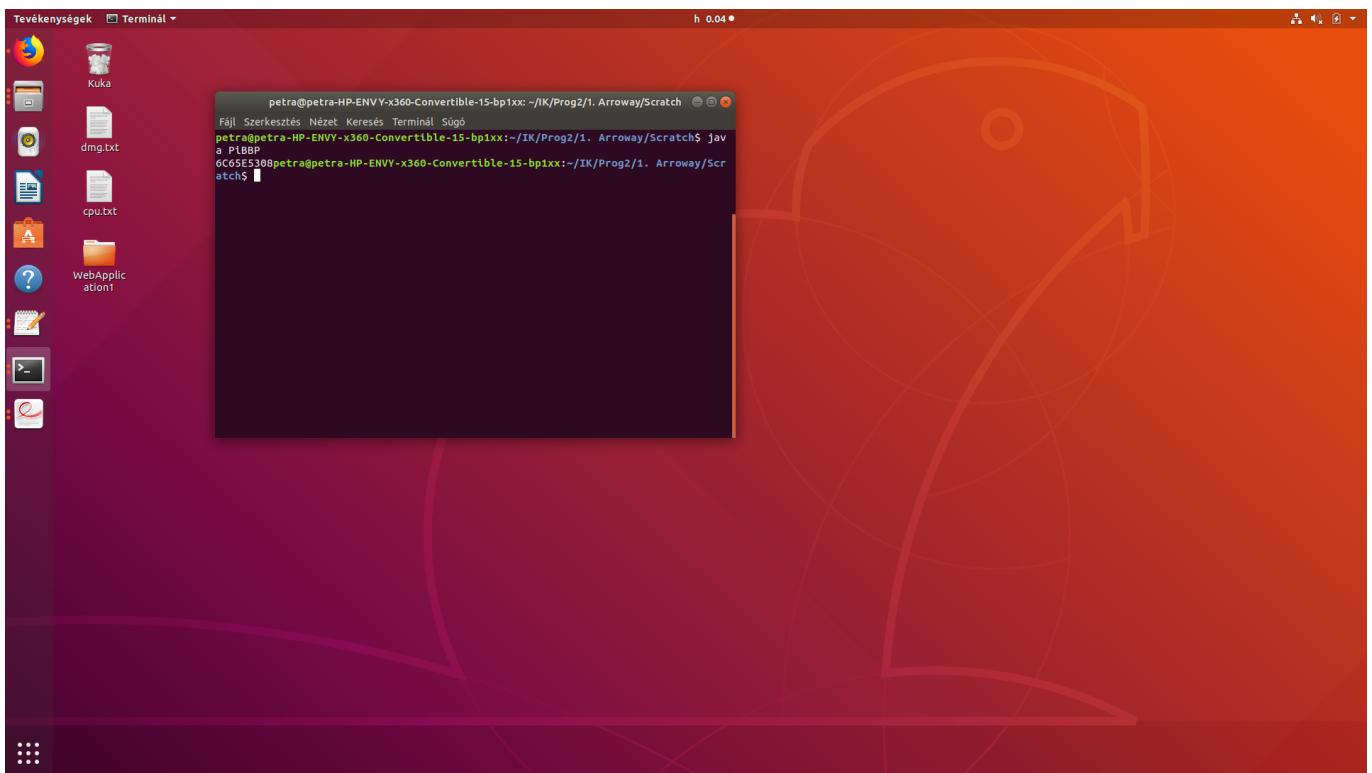
 r = (r*r) % k;
 }

 return r;
}
```

Itt pedig a példányosítás történik meg:

```
public static void main(String args[]) {
 System.out.print(new PiBBP(1000000));
```

Az első képen egymilliós d értékkel van kiszámolva a hexadecimális szám, a másodikon pedig százezressel.



## 13. fejezet

# Helló, Liskov!

### 13.1. 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: (a Liskov-elv forrása) <https://reiteristvan.wordpress.com/2011/07/05/s-o-l-i-d-objektum-orientált-tervezési-elvek-3-lsp/>

Tanulságok, tapasztalatok, magyarázat...

A feladat a Liskov behelyettesítési elvének a bemutatása volt. Szó szerint, vagyis csak magyarra lefordítva az elv a következő:

Ha  $S$  altípusa  $T$ -nek, akkor minden olyan helyen ahol  $T$ -t felhasználjuk  $S$ -t  $\leftrightarrow$  is minden gond nélkül behelyettesíthetjük anélkül, hogy a programrész  $\leftrightarrow$  tulajdonságai megváltoznának.

Tehát az elv lényegében az öröklődést és a polimorfizmust is bemutatja, amelyről az olvasónaplóban már írtam. Az elv kimondja, hogy ha az  $S$  leszármazottja, vagyis gyermeké a  $T$  (szülő) osztálynak, akkor a gyermek behelyettesíthető minden olyan helyre, ahová eredetileg a  $T$ -t várunk (paraméterek például). Ez az elv él végig a program során és feltételezi, hogy teljesül is. Gyakorlatilag viszont lehetséges, hogy a program igazként kezeli az adott esetet, ám a valóságban erre nincs lehetőség. Ennek a szemléltetésére szerintem jól érthető példa amit Tanár Úr írt le, így én is annak az elemzésénél maradnék (valamint a forráskód is erre épül, de akár lehetett volna téglalap-négyzet, vagy kör-ellipszis példa). Tehát ha a Liskov-elv szerint járunk el, és a madaras példánál maradunk, akkor a sasok és a pingvinek is repülhetnek. Ennek a megjelenítése a programban polimorfizmussal történik meg (repül a Madarakon, valamint a Sason és Pingvinen belül), valamint az öröklődés, ahol a szülő osztály a Madarak és a Sas és a Pingvin osztályok pedig a gyermekeket jelképezik (extends, leszármaztatás).

```
class Madar
{
```

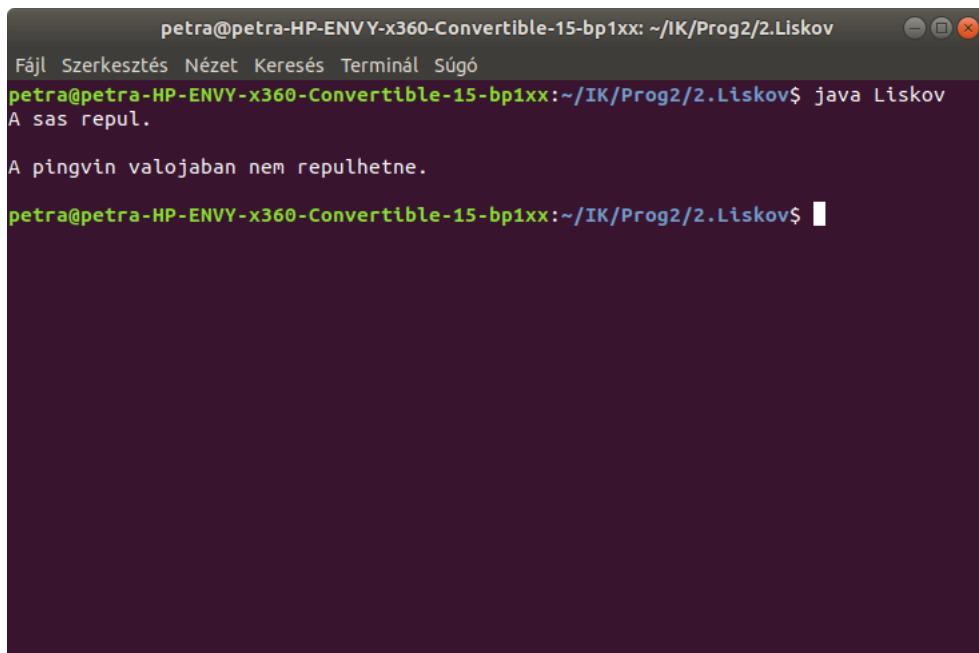
```
public void repul()
{
 System.out.println("A madar repul.\n");
}

class Sas extends Madar
{
 public void repul()
 {
 System.out.println("A sas repul.\n");
 }
}

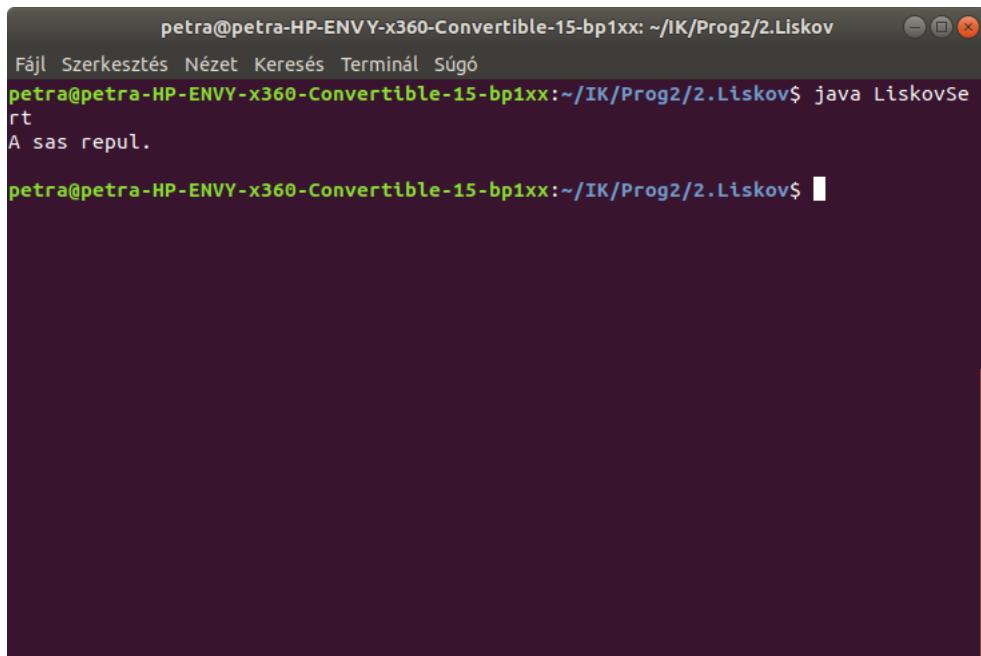
class Pingvin extends Madar
{
 public void repul()
 {
 System.out.println("A pingvin valójában nem repulhetne.\n");
 }
}
```

A probléma itt az, hogy ez az állítás nem igaz. Ahhoz, hogy igazzá tegyük, külön kell választanunk a madarakat, hogy melyek tudnak, és melyek nem tudnak repülni. Itt jönnek a képbe az absztrakt osztályok, és az interfések. (Az objektum orientáltság miatt nem nézhetjük meg igaz-hamis módszerrel.) Az interfésekkel választhatjuk szét két csoportra a madarakat, repülőkre, és nem repülőkre. A repül metódust pedig csak a repülő madarak használhatják (mivel csak itt van deklarálva a metódus), ezzel a módszerrel pedig továbbra is helyes marad a Liskov-elv. (A források mellékelve, nem szeretném teljes kódokkal tölteni a könyvet.)

A képeken pedig a program működése (most csak Java, de a C++ kódok mellékelve vannak), előbbin a megkülönböztetés nélkül, utóbbin megkülönböztetéssel.



The screenshot shows a terminal window with a dark background and light-colored text. At the top, the window title is "petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/2.Liskov". Below the title, there are standard window control buttons (minimize, maximize, close). The terminal prompt is "petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/2.Liskov\$". The user types "java Liskov" and presses Enter. The terminal then displays two lines of output: "A sas repul." and "A pingvin valójában nem repulhetne.". The cursor is visible at the end of the second line of output.



A screenshot of a terminal window titled "petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/2.Liskov". The window shows the following text:

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/2.Liskov$ java LiskovSe
rt
A sas repul.

petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/2.Liskov$
```

## 13.2. 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:

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban szintén előjönnek azok a dolgok amikről az olvasónaplónak megadott könyvekben is olvashattunk, tehát az öröklődést, hatáskört és a polimorfizmust figyelhetjük meg. Az öröklődésre próbálunk ezzel a feladattal rácáfolni, vagyis pontosabban arra, hogy a szülőn keresztül nem küldhetjük el a gyermek üzeneteit, tehát ilyen formában agyermek üzenetei nem érhetők el referenciával sem, mivel a szülőben ez nem volt definiálva. A láthatósági szintekről nem írnék, hiszen itt public-ként használtam minden, nem volt rejtés. A hibaüzeneteket pedig már fordításkor jelzi a fordító, hogy a gyermek metódusa a szülőkön keresztül nem meghívhatók, csak fordítva lehetne, ha a gyermek örökölte volna a szülő metódusait. A programkódok mellékelve vannak (nem szúrnám be plusz helyfoglalás céljából).

Valamint a fordítási hibákról a képek:

The image displays two terminal windows side-by-side, both titled "petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/2.Liskov".

**Top Terminal (C++):**

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/2.Liskov$ g++ szulo2.cpp
p -o szulo
g++: error: szulo2.cpp: Nincs ilyen fájl vagy könyvtár
g++: fatal error: no input files
compilation terminated.
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/2.Liskov$
```

**Bottom Terminal (Java):**

```
Fájl Szerkesztés Nézet Keresés Terminál Súgó
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/2.Liskov$ javac szulogyerek.java
szulogyerek.java:5: error: cannot find symbol
 osok.uzenet();
 ^
symbol: method uzenet()
location: variable osok of type Szulo
1 error
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/2.Liskov$
```

### 13.3. Anti OO

A BBP algoritmussal 4 a Pi hexadecimális kifejtésének a 0. pozíciótól számított  $10^6$ ,  $10^7$ ,  $10^8$  darab je-gyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket! <https://www.tankonyvtar.hu/hutanitok-javat/apas03.html?id=561066> <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#>

Megoldás video:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A program megegyezik az Arroway csokorban elkészített Pi-vel dolgozó feladattal, a programkód megírásában van egy minimális különbség csak, a matematikai alapja ugyan az. A laptopomban amin a számításokat

lefuttattam Intel Core i5-8250U processzor, és 8 GB Ram van. A C# futtatását Ubuntu alatt pedig dotnet segítségével oldottam meg. Amit kiemelnék, hogy a fordítás futtatásnál a C kódos programra figyeljünk, ugyanis használni kell egy -lm kapcsolót (gcc BPP.c -o bpp -lm).

```
for (d = 1000000; d < 1000001; ++d)
{
 d16Pi = 0.0;

 d16S1t = d16Sj (d, 1);
 d16S4t = d16Sj (d, 4);
 d16S5t = d16Sj (d, 5);
 d16S6t = d16Sj (d, 6);

 d16Pi = 4.0 * d16S1t - 2.0 * d16S4t - d16S5t - d16S6t;

 d16Pi = d16Pi - floor (d16Pi);

 jegy = (int) floor (16.0 * d16Pi);
}
```

A hatványok növeléséhez a for ciklus fejében kell a d változót módosítani, ez jelzi, hogy 10 a hanyadikont vizsgálunk. Ami most a példában szerepel az 10 a hatodikon.

| Nyelv es hatvany | C (sec)    | C++ (sec)  | C# (sec)    | Java (sec) |
|------------------|------------|------------|-------------|------------|
| 6                | 1.725870   | 1.736117   | 2,1079615   | 1.547      |
| 7                | 19.924895  | 19.974519  | 21,6759024  | 17.89      |
| 8                | 230.134115 | 231,377710 | 243,9283811 | 206,013    |

13.1. táblázat. Eredmények

A screenshot of the Visual Studio Code interface. On the left is the sidebar with icons for file operations, search, extensions, and recommended extensions. The main area shows a C# file named bpp.cs with code related to calculating pi. Below the editor are tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active, displaying a terminal session with the command `dotnet run` followed by several lines of numerical output. The status bar at the bottom indicates the file is 20.34 kB and the line count is 112.

A second screenshot of the Visual Studio Code interface, identical to the first one. It shows the same C# file (bpp.cs), terminal session with the command `dotnet run`, and the same numerical output. The status bar at the bottom indicates the file is 21.15 kB and the line count is 112.

The screenshot shows a Visual Studio Code interface running on a Linux desktop. The left sidebar has 'Tevékenységek' and 'Visual Studio Code' tabs. The main area shows a C# file named 'bpp.cs' with code for calculating pi. The terminal window on the right shows the output of the program, which calculates pi to 206.013 digits.

```
home > petra > IK > Prog2 > 2.Liskov > bpp.cs
93 double d16S6t = 0.0d;
94
95 int jegy = 0;
96
97 System.DateTime kezd = System.DateTime.Now;
98
99 for(int d=1000000; d<1000001; ++d)
100
101 d16Pi = 0.0d;
102
103 d16S1t = d16Sj(d, 1);
104 d16S4t = d16Sj(d, 4);
105 d16S5t = d16Sj(d, 5);
106 d16S6t = d16Sj(d, 6);
107
108 d16Pi = 4.0d*d16S1t - 2.0d*d16S4t - d16S5t - d16S6t;
109
110 d16Pi = d16Pi - System.Math.Floor(d16Pi);
111
112 jegy = (int)System.Math.Floor(16.0d*d16Pi);

petra@petra-HP-ENVY-x360-Convertible-15-bpxx:~/IK/Prog2/2.Liskov$./bpp
12
230.134115
petra@petra-HP-ENVY-x360-Convertible-15-bpxx:~/IK/Prog2/2.Liskov$./bpp
12
231.377710
petra@petra-HP-ENVY-x360-Convertible-15-bpxx:~/IK/Prog2/2.Liskov$ java PiBBPBen
ch
12
206.013
petra@petra-HP-ENVY-x360-Convertible-15-bpxx:~/IK/Prog2/2.Liskov$
```

## 13.4. Hello Android!

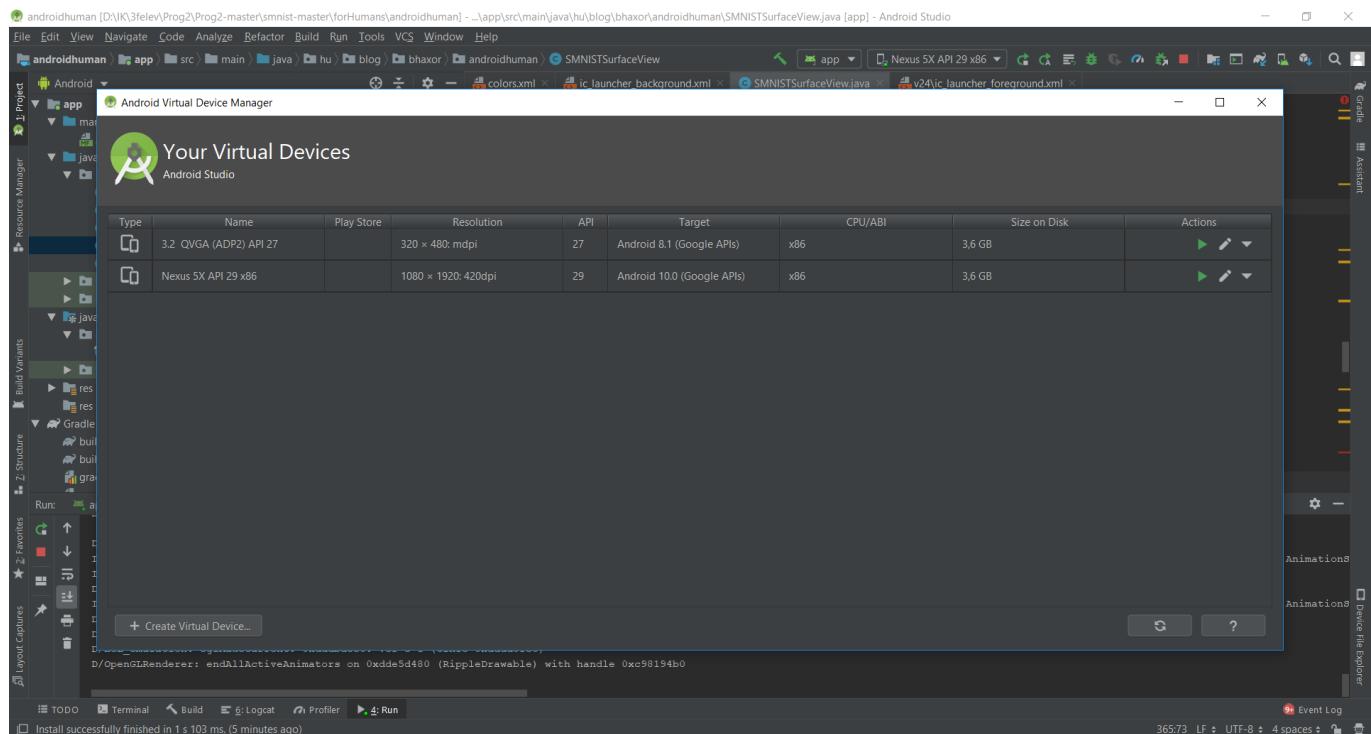
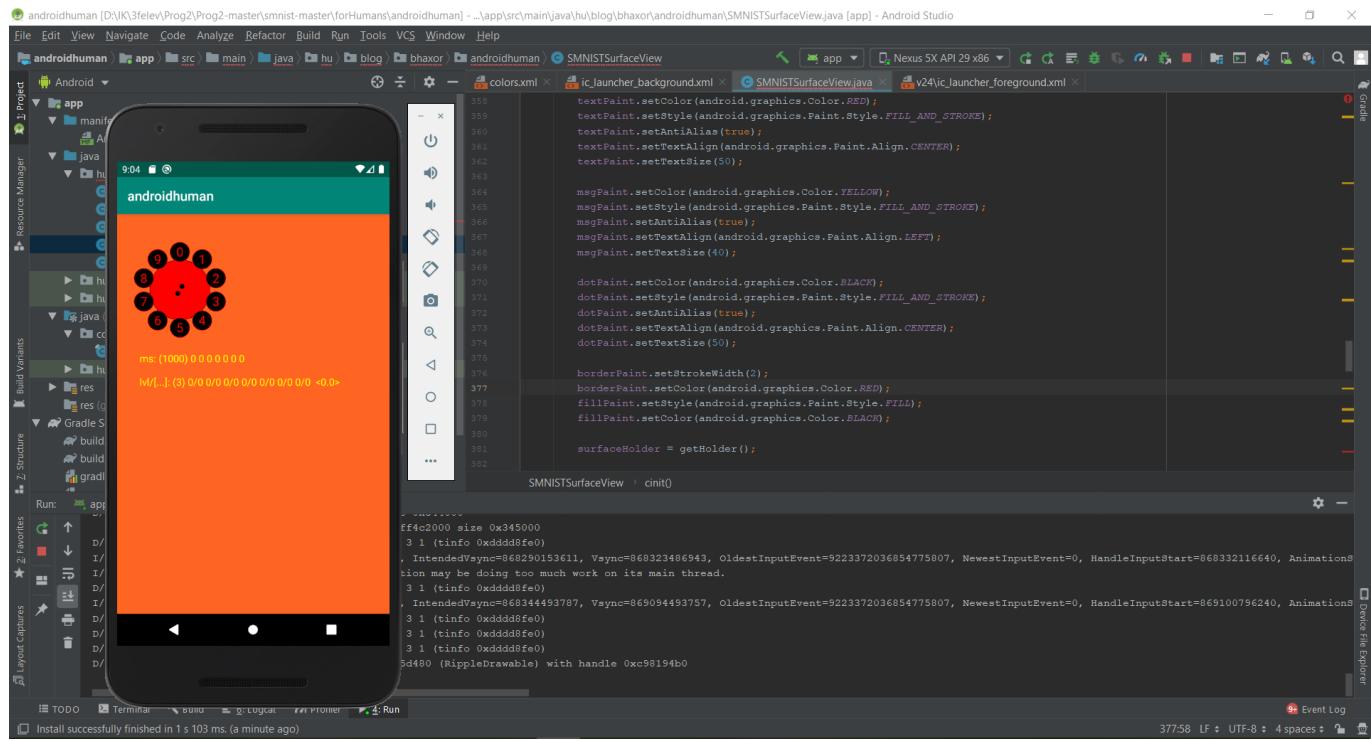
Élesszük fel az SMNIST for Humans projektet! <https://gitlab.com/nbatfai/smnist/tree/master/forHumans/SMNIST>  
Apró módosításokat eszközölj benne, pl. színvilág.

Megoldás videó:

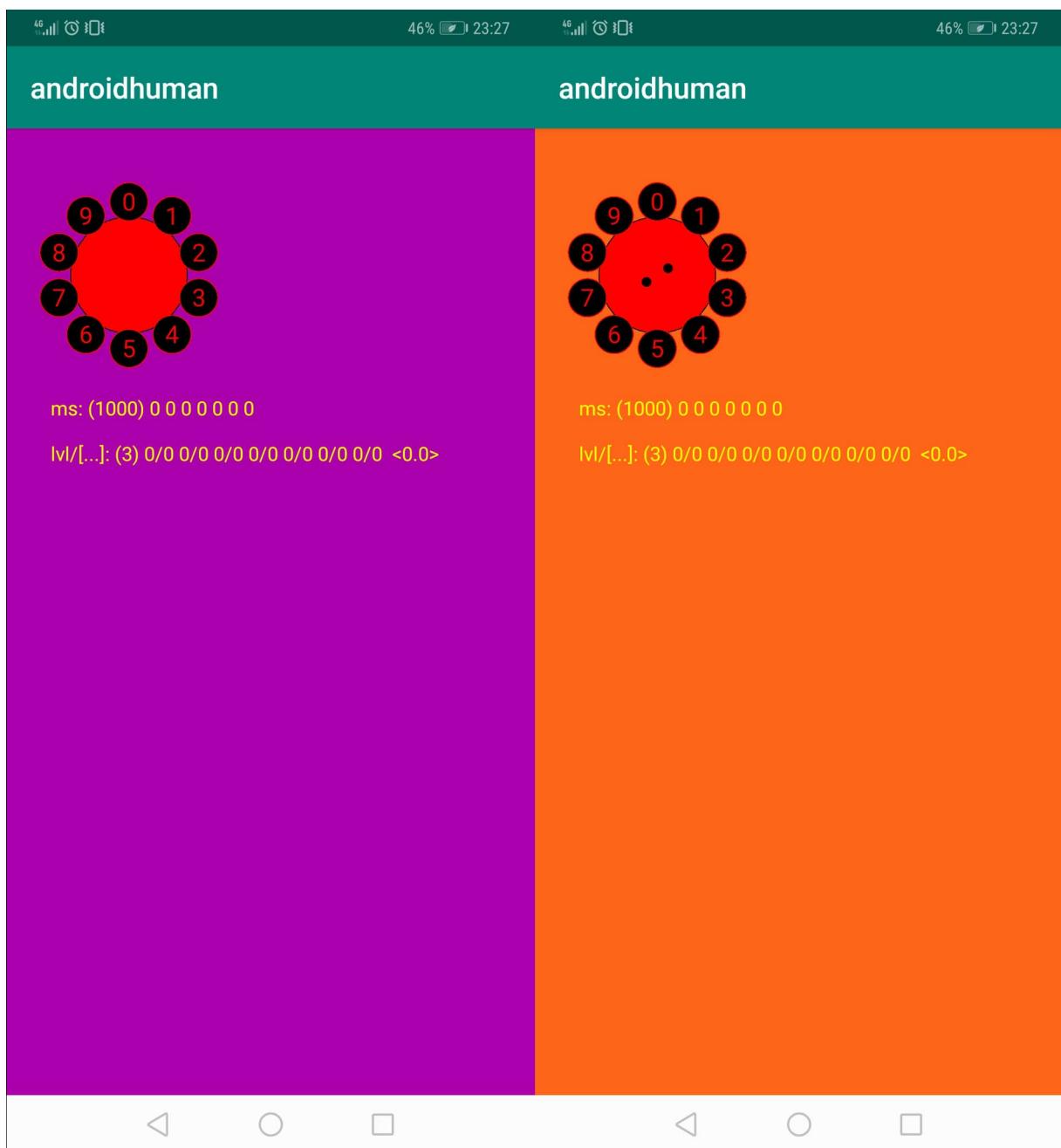
Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Ahhoz, hogy hozzá kezdhessünk a feladathoz először is szükségünk lesz valamire, ami képes android programot futtatni (valamilyen IDE), én az android studiot választottam erre a célra. Még a program telepítése közben érdemes megjelölni, hogy telepítse fel a virtuális android eszközt is. Miután először megpróbáltam feléleszteni a progit, felbukkanthat az első hiba. A .grade fájlt kellett pótolni ekkor, majd ezután a konfigurációt kellett helyesen megcsinálni, miután a fájlokat beimportáltuk. Itt még a konfiguráció során utána kell nézni, hogy a forráson mit kell módosítani (hiszen gyorsan változik SDK). Végül az aktuálisan legújabb api verziós androiddal futtattam le a progit (api 29, Android Q). A programban a színvilágban végeztem módosítást, ami már nem volt nehéz feladat (az elején az indítás okozott némi fejvakarást). Ki kellett keresni a megfelelő java forrást, amelyekben a színvilág van definiálva, majd az RGB kódöt átírhattuk a kívánt színre, illetve a szöveg színénél (például) a konkrét szín nevét kellett átírni (BLACK -> RED). Ezek pedig pontosan a SMNISTSurfaceView.java fájlban találhatóak meg. Az android studio lehetőséget ad nem csak az android verzió kiválasztására, de a felbontást is tudjuk befolyásolni. A második kép pont ezt mutatja be, hogy hol választhatunk.



Az android studio ahogyan a fenti képeken is látszik, virtuális eszközökkel kezdi el futtatni a programot amikor kipróbáljuk, de ha ki szeretnénk próbálni a saját mobilunkon, azt is megtehetjük. Az android studioban van egy build apk-s funkció, amellyel megkapunk egy .apk kiterjesztésű fájlt. Ezt csak át kell húznunk a mobilra és telepíteni, aztán játszhatunk is.



## 14. fejezet

# Helló, Mandelbrot!

### 14.1. Reverse engineering UML osztálydiagram

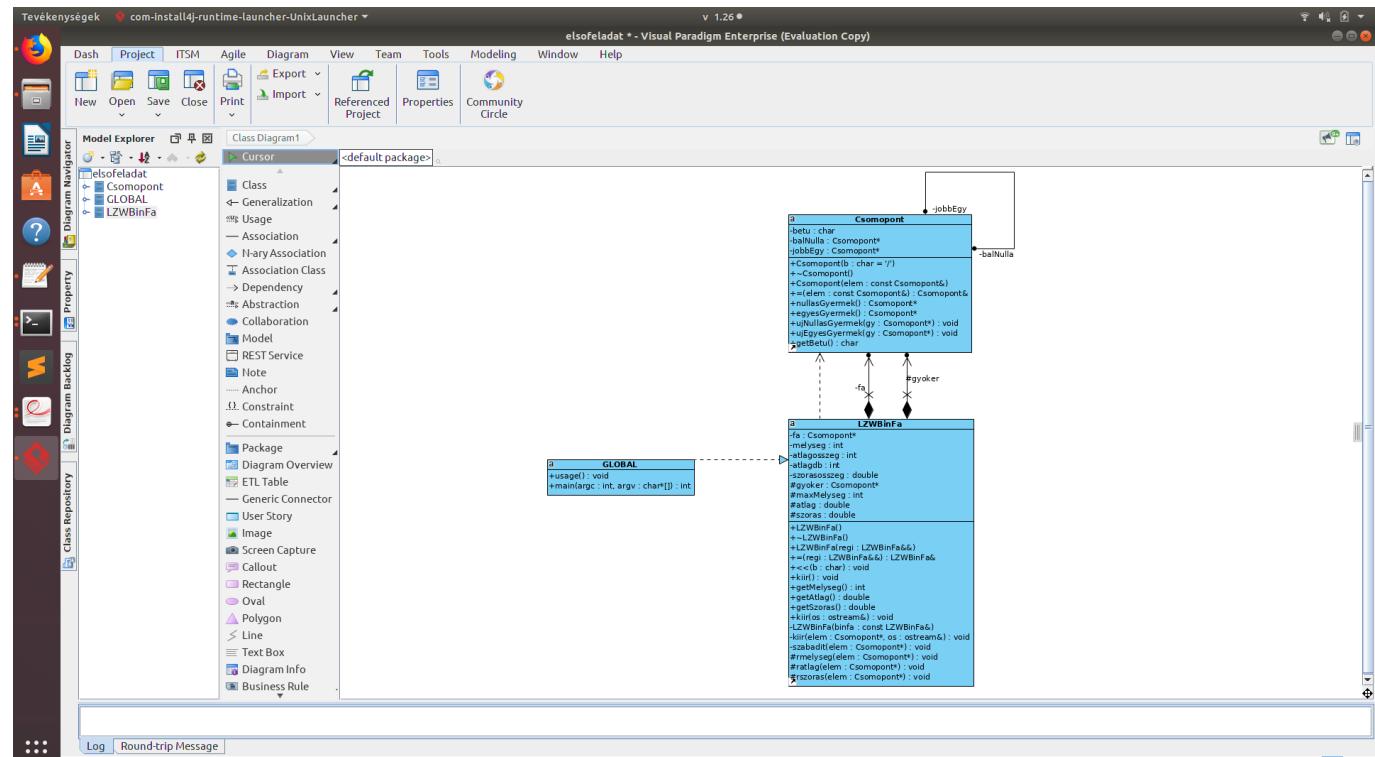
UML osztálydiagram rajzolása az első védési C++ programhoz. Az osztálydiagramot a forrásokból generáljuk (pl. Argo UML, Umbrello, Eclipse UML) Mutassunk rá a kompozíció és aggregáció kapcsolatára a forráskódban és a diagramon, lásd még: [https://youtu.be/Td\\_nIERIEOs](https://youtu.be/Td_nIERIEOs). <https://arato.inf.unideb.hu/batfai.norbert/UD> (28-32 fólia)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A feladat egy UML osztálydiagram elkészítése volt. Ehhez a Visual Paradigm programot vettet segítségül. A feladat még kérte, hogy a forráskód, amiről a diagram készül, az az első C++ védési feladat legyen, ami a binfa volt. A program ezt 3 fő egységre bontotta szét, a globálisan elérhető dolgokra (változók, függvények, main), a Csomópont és az LZWBInFa osztályra. Feladat még a kompozíció és az aggregáció kapcsolata, erről írnék még egy kicsit. Az olvasónaplóban részletesebben ki van fejtve. Az aggregációt egy üres rombusz jelöli az uml diagramon azon az oldalon, ahol tartalmazás található (mivel az aggregáció tartalmazást jelent), a kompozíció pedig egy tömött rombusszal van jelölve, de a lényege hasonló, vagyis ennyi a különbség a kettő között. Jelen esetben, vagyis a mi példánknál is kompozícióról beszélhetünk, amely 2 helyen jelent meg. Ha kompozíciót van, akkor csak együtt hozhatunk létre és szüntethetük meg azt, vagyis a tartalmazót és a tartalmazottat. Ebben a példában a gyökér jelenti a kompozíciót. Illetve a későbbi összehasonlításhoz beszúrnám a binfa csomópont osztályát.



```
class Csomopont
{
public:
Csomopont (char b = '/') : betu (b), balNulla (0), jobbEgy (0)
{
};
~Csomopont ()
{
};
Csomopont (const Csomopont& elem) {

 betu = elem.getBetu();
 balNulla = new Csomopont;
 jobbEgy = new Csomopont;
 *balNulla= *(elem.nullasGyermek());
 *jobbEgy= *(elem.egyesGyermek());
}

Csomopont & operator= (const Csomopont& elem) {

 betu = elem.getBetu();
 Csomopont* ujBal = new Csomopont();
 *ujBal = *(elem.nullasGyermek());
 delete balNulla;
 balNulla = ujBal;
 Csomopont* ujJobb = new Csomopont();
 *ujJobb = *(elem.egyesGyermek());
 delete jobbEgy;
```

```
 jobbEgy = ujJobb;

 return *this;
 }

 Csomopont *nullasGyermek () const
 {
 return balNulla;
 }

 Csomopont *egyesGyermek () const
 {
 return jobbEgy;
 }

 void ujNullasGyermek (Csomopont * gy)
 {
 balNulla = gy;
 }

 void ujEgyesGyermek (Csonopont * gy)
 {
 jobbEgy = gy;
 }

 char getBetu () const
 {
 return betu;
 }

private:

 char betu;
 Csonopont *balNulla;
 Csonopont *jobbEgy;

};
```

## 14.2. Forward engineering UML osztálydiagram

UML-ben tervezünk osztályokat és generálunk belőle forrást!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A kódot egy Téglalap-Négyzet példából generáltam le, vagyis forrásként egy téglalap-négyzet kapcsolatot valósítottam meg osztályokkal. Amikor diagramról építünk fel egy kódot, vagyis generálunk le, akkor

lényegében egy vázat kapunk vissza. Ebben a headereket látjuk jobban megjelenítve, a többi csak egy váz, az osztályok, amelyek a metódusokat tartalmazzák, ezek a metódusok viszont üresek (tehát a függvény törzse), nincsenek kifejtve. (Itt szintén Visual Paradigmot használtam.) És ezeket a vázlatokat kaptam vissza a programtól:

A Square.java:

```
public class Square extends Rectangle {
 public Rectangle _unnamed_Rectangle_;

 public Square(int aSide) {
 throw new UnsupportedOperationException();
 }

 public int getArea() {
 throw new UnsupportedOperationException();
 }
}
```

Az Rectangle.java:

```
public class Rectangle {
 public int _m_width;
 public int _m_height;
 public Square _unnamed_Square_;

 public Rectangle(int aWidth, int aHeight) {
 throw new UnsupportedOperationException();
 }

 public int getWidth() {
 throw new UnsupportedOperationException();
 }

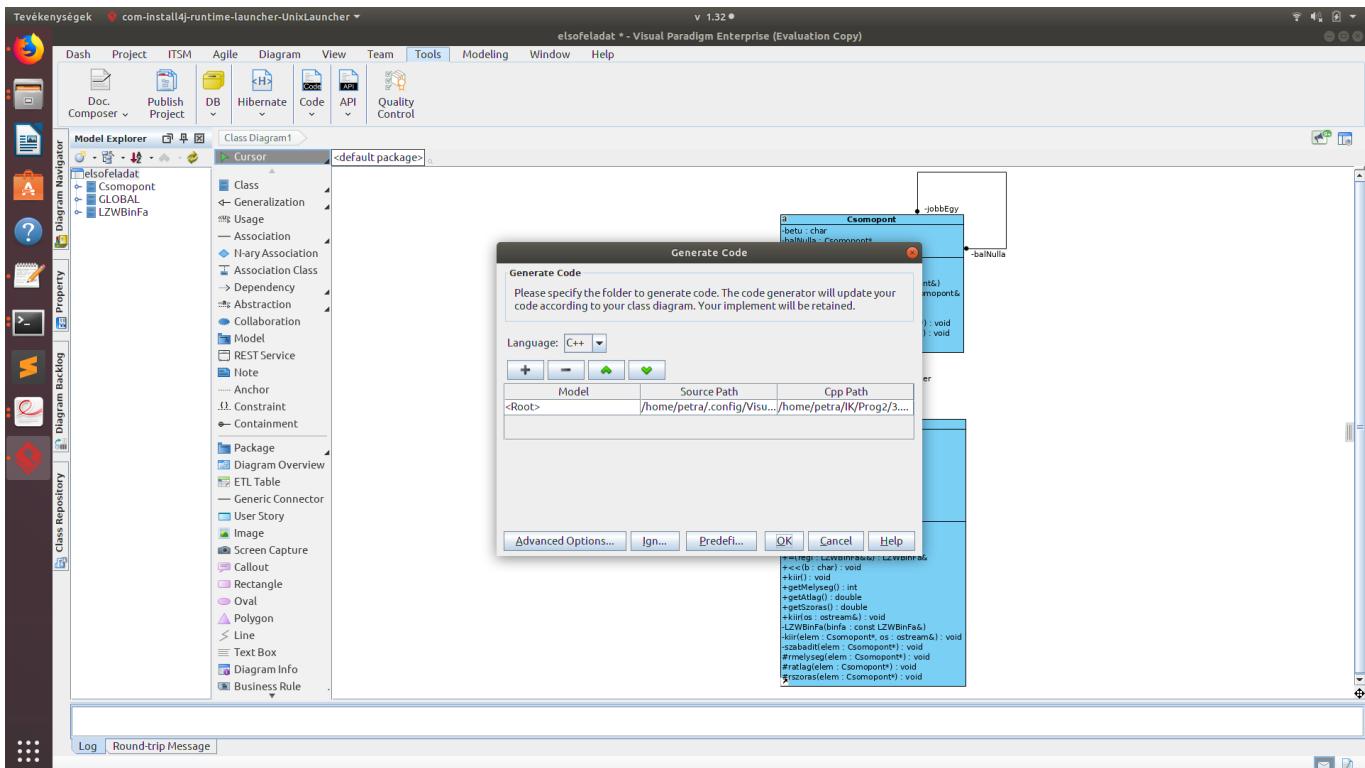
 public int getHeight() {
 throw new UnsupportedOperationException();
 }
}
```

És a GLOBAL.java:

```
public class GLOBAL {

 public int main() {
 throw new UnsupportedOperationException();
 }
}
```

Valamint a programon belül itt tudjuk a visszafejtést megcsinálni:



### 14.3. Egy esettan!

A BME-s C++ tankönyv 14. fejezetét (427-444 elmélet, 445-469 az esettan) dolgozzuk fel!

Megoldás videó:

Megoldás forrása:

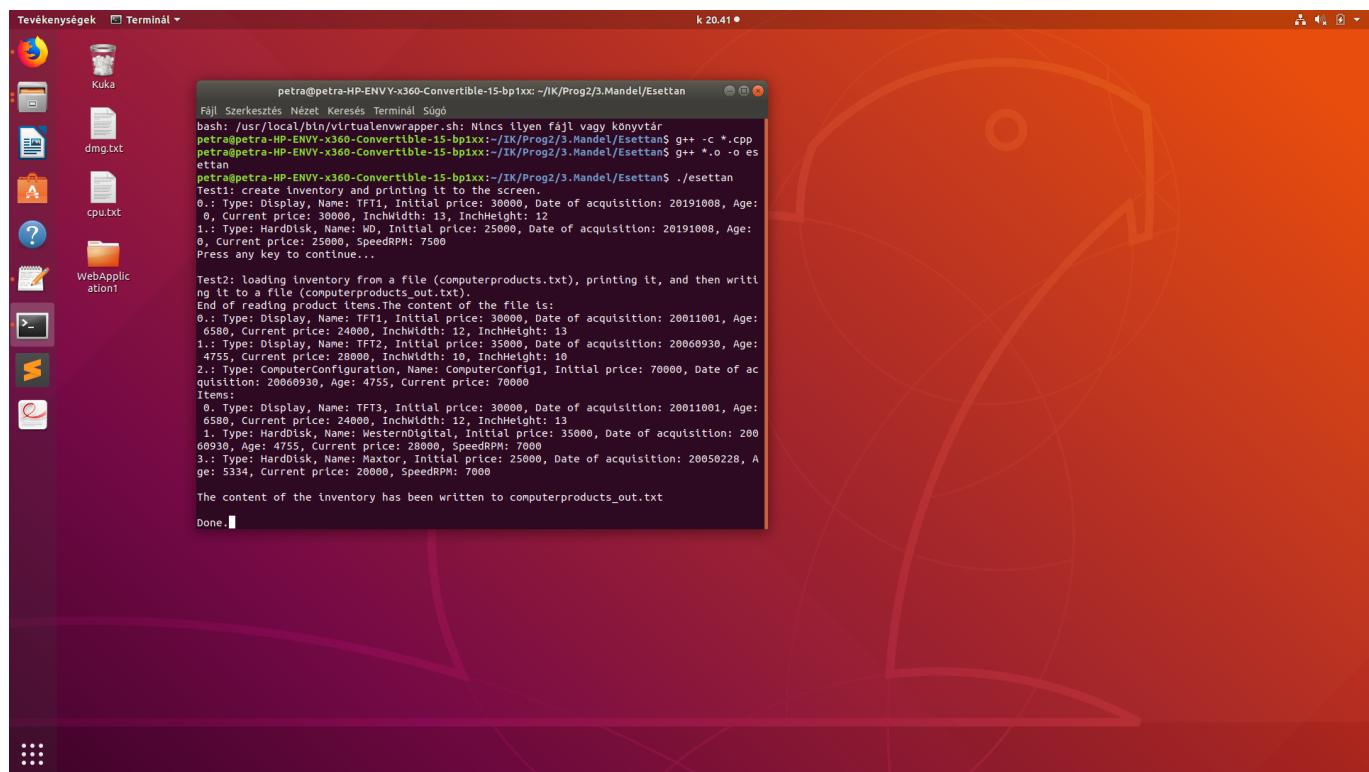
Tanulságok, tapasztalatok, magyarázat...

Az egységes modellezőnyelv, vagy más néven UML, a mai állás szerint az egyedüli (szabványos) modellező nyelv a szoftverfejlesztés terén. Emiatt viszont nagy támogatottsággal rendelkezik. Az osztálydiagram az osztályok, interfések és más típusokat mutat be, valamint ábrázolja a köztük lévő kapcsolatot. Az általa létrehozott diagrammokat grafikus felületen mutatja be. Alkalmas például osztályok és az öröklődéseik bemutatására is, erre a későbbiekben fog példát mutatni a könyv. Egy ilyen osztálydiagram 3 részre oszlik: az első téglalapban az osztály nevét fedezhetjük fel, a következőben az attribútumait, a harmadikban pedig a műveleteit. Ha az utolsó két téglalapot nem szeretnénk látni, akkor létezik olyan funkciója a programnak, ami elrejti azt. Az attribútumok és műveletek előtt jelöléseket láthatunk (+ - # ~), ezek a láthatóságot jelölik, tehát public, private, protected vagy package (C++-ban nincs). Ha esetleg szükség lenne egy változó megadására, mert lényeges, akkor azt komment formájában jelölhetjük meg. Ha az attribútum után : jelet látunk, akkor az után a típusa van megadva. A multiplicitással intervallumot is megadhatunk ha szeretnénk, ezekre is különböző jelölések szolgálnak (könyv leírja ezeket). Ha nem adtunk meg, akkor 1-et fog alapértelmezettként kezelni. A tömbökre is kétféle jelölést használhatunk, az egyikkel megadva a konkrét számértéket, tehát statikus tömbünk lesz, a másikkal pedig dinamikus tömböt készíthetünk. A paraméterlistánál az irányokat is megadhatjuk, tehát in, out vagy inout (ki-, be- vagy kimenő és bemenő egyszerre). Ha egy osztály absztrakt, akkor azt is meg tudjuk jelölni a diagrammon. De ugyanígy az interfész, struktúrát vagy az enum típust is meg tudjuk jelölni. Amikről eddig írtam, azok voltak csomópontként jelölve.

A könyv itt egy szállásfoglalás menetét hozza fel példának, ahol a kapcsolatokat is részletesen bemutatja. Itt beszélhetünk asszociációkról, amik lényegében a kapcsolatok, és ehhez akár 2 szerepnév is tartozhat. (Itt pl munkavállaló és munkáltató, de nem feltétlen ilyen kapcsolat lehet, vagy lehet kevesebb szerepnév is). A C++ forráskódban amivel ezt a diagrammot létrehozták egy oda-vissza hivatkozás figyelhető meg, amely megoldás néha tényleg praktikus, de alapvetően nem. Az összerendelések megvalósításához pedig tömböket használnak, viszont lényeges, hogy a Reservationnél a pbill és a GetBill is vehet fel null értéket. A példában itt már látszik, hogy vannak privát dolgok, ezt jelzi a szerepnév előtt a "-" jel. Létezhet akár nulla multiplicitás is, nincs kizárvva a lehetőségek közül. Ebben a példában a statikus tömb és a pointer nem elég, dinamikus és pointerek tömbje szükséges. Például ha új foglalást hozunk létre, akkor egy objektumlistára történő feliratkozással eleget tehetünk a kérésnek, de eltávolításra is van a programban függvény megírva. Ha pedig a láthatóságon szeretnénk változtatni, akkor privátból publikussá kell tenni ("+" jelre változik a "-"). A diagramon az abszakt osztály dőlt betűvel lett megjelölve. Az asszociációk implementálása általában C++ nyelven pointerekkel történik meg. A minősített asszociáció az asszociatív tömb és a kulcsának a használatát mutatja be. A minősítőn keresztül a minősített asszociáció hoz létre kapcsolatot osztályok között. Az asszociációs osztály és az asszociáció kapcsolatát szaggatott vonallal jelölik. Az asszociációnak pedig 2 különböző fajtája van, az aggregáció (rombusz, tartalmazást jelöl), és a kompozíció (különbség, hogy itt a rombusz nem üres). Az UML diagrammoknál lehetőségünk van néhány esetben (művelet)sablonok létrehozására. Az osztálysablonok paramétereit a jobb felső sarokban jeleníthetjük meg. A példányosítás illetve a kifejtésre két lehetőség van, amelyre a könyv példát is hoz. Illetve lehetőségünk van arra, hogy mi magunk hozunk létre egy UML diagrammot, majd abból generáljuk le a forráskódot (az már kérdés mennyire hatékony). Más néven kódvisszafejtésnek is nevezik. Ha kódot hasonlítunk már meglévő kóddal, és csak a különbségre van szükségünk, akkor beszélünk modell-kód szinkronizálásról., illetve ennek ellentéte kód-modell szinkronizálás. Ezeknek az előállítására több program is segítséget nyújt, néhány sok terhet levesz a felhasználóról.

Az esettanulmány a főbb nyelvi elemekre, vagyis pontosabban az objektumorientáltság eszközeire fog jobban koncentrálni, ami az öröklés és a virtuális függvényeket takarja. Az esettanulmány egy számítógép-alkatrészek és konfigurációjával foglalkozó kereskedés tervét dolgozza fel. Céljuk egy keretrendszer felépítése, amely támogatja a termékcsaládfaikat, és egy alkatrész kezelő alkalmazást szeretnének még ennek a segítségével létrehozni. A jövőbeli terv pedig az, hogy erre a keretrendszerre építve a lehető legkevesebb munkával készítsenek más termékcsaládot támogató alkalmazásokat is. Az elvárások pedig a következők: a forráskód kiadása nélkül is használható legyen, adatfolyamok kezelését támogatnia kell, a termék attribútumai és az azokból történő árszámítás, a termék összetettsége, tehát elemi vagy összetett (több termékből áll). Az alkalmazásnak is van követelménye, a kijelző, merevlemez egység és a konfiguráció. Itt most a tesztalkalmazás fog elkészülni. Az osztálykezeléshez van egy előre elkészített C++ könyvtár, hogy csak a fejlécfájlokat kelljen behúzni, definálni ne kelljen minden. A termék bemutatására készítettek UML diagrammot. A termékeket pedig amennyire lehet megpróbálják egységesen kezelni. A Product osztály tagváltozói védettek, tehát külsősök számára csak olvashatók. A termék korának meghatározására a GetAge függvény szolgál. Az adott termék árát is mindenki tudjuk számítani aktuálisan, ehhez leszármaztatást használnak. A terméktípusokat megkülönböztetik a számítások során, külön Product-leszármazottként kezelik. A HardDisk osztály felüldefiniál a programban, és specifikus termékosztályt ad ebben az esetben. A Print függvény pedig megjelenítésreszolgál, bemutatja a különböző termékeket. A termékek adatait adatfolyamba írják, ezt a writeParamsToStream tagfüggvény végzi el, amely a Product osztályban van. Az azonos, vagyis a közös paramétereket/tulajdonságokat gyűjti ki az adatfolyamba írás során. Az aktuális árat felesleges tárolni, hiszen az folyamatosan változik. Az adatfolyamba írás összetett feladat a példában, valamint a beírást és kiolvasát a globális operátorok túlterhelésével oldják meg. Az összetett termékek bemutatására hozták létre a CompositeProduct osztályt. Ha összetett terméket szeretnénk beolvasni, akkor

újra felüldefiniálásra van szükségünk. A termékek reprezentálására létrehozták a ProductInventory osztályt, ez betölt az adatfolyamból, beleír, memóriában tárol, megjelenít, stb... A termékeket, vagyis az objektumokat adatfolyamból beolvasással hozzuk létre, amelyeket a CompositeProduct és a ProductInventory-ból olvasunk be. Ennek célja a megfelelő Productból leszármaztatott objektum megalkotása és a termékkód beolvasása. Ezzel viszont sok probléma merül fel, amelyet a virtuális függvények és az indirekciónak oldhat meg, amennyiben jól használjuk azokat. A CreateProduct tagfüggvény a példányosítást végzi el (termékosztályt), és a pointerezést. Ennek a megfelelő működés érdekében absztrakt függvénynek kell lennie. Ezután még egy felülírásra van szükség, amit a ProductFactory végez, és a termékek példányosítást ez végzi. A ProductFactory osztályunkat csak egyszer kell példányosítani, tehát csak egyre lenne szükségünk, ehhez viszont több helyről szeretnénk elérni. Ehhez kellett definiálni egy globális változót, majd inicializálni kell és egy statikus változóként be kell ágyazni a ProductFactory osztályba, és hozzáférést kell neki adni. Ezután pedig lehet tesztelni, mert a program összeállt. Rövidíthetünk a kódon, ha előre megírt függvényeket vagy sablonokat használunk, ezeket a könyvben az Algoritmusok fejezetben megtalálhatjuk. Gyorsíthatunk, vagy átláthatóbbá tehetjük még a programot, ha rendezünk például menet közben, így a kezelést is megkönnyebbítjük, vagy jobban tudjuk fejben tartani a dolgokat és kezelní azokat. De az előre definiált függvények a legpraktikusabbak ténylegesen. A formátumok okozhatnak problémát ha nem azonosak, erre kell megoldást keresni. Érdemes kutakodni ingyenesen letölthető megoldások között is, sok új funkciót érhetünk el ilyen módon, amivel a problémákat is könnyebb sokszor megoldani.



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/3.Mandel/Esettan". The terminal displays the following output:

```
Fájl Szerkesztő Nézet Keresés Terminál Súgó
bash: /usr/local/bin/virtuaenvwrapper.sh: Nincs ilyen fájl vagy könyvtár
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/3.Mandel/Esettan$ g++ -c *.cpp
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/3.Mandel/Esettan$ g++ *.o -o esetan
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/3.Mandel/Esettan$./esetan
Test1: create inventory and printing it to the screen.
0.: Type: Display, Name: TFT1, Initial price: 30000, Date of acquisition: 20191008, Age: 0, Current price: 30000, InchWidth: 13, InchHeight: 12
1.: Type: Harddisk, Name: WD, Initial price: 25000, Date of acquisition: 20191008, Age: 0, Current price: 25000, SpeedRPM: 7500
Press any key to continue...
Test2: loading inventory from a file (computerproducts.txt), printing it, and then writing it to a file (computerproducts_out.txt).
End of reading product items. The content of the file is:
0.: Type: Display, Name: TFT1, Initial price: 30000, Date of acquisition: 20011001, Age: 0580, Current price: 24000, InchWidth: 12, InchHeight: 13
1.: Type: Display, Name: TFT2, Initial price: 35000, Date of acquisition: 20060930, Age: 4755, Current price: 28000, InchWidth: 10, InchHeight: 10
2.: Type: ComputerConfiguration, Name: ComputerConfig1, Initial price: 70000, Date of acquisition: 20060930, Age: 4755, Current price: 70000
Items:
0.: Type: Display, Name: TFT3, Initial price: 30000, Date of acquisition: 20011001, Age: 0580, Current price: 24000, InchWidth: 12, InchHeight: 13
1.: Type: Harddisk, Name: WesternDigital, Initial price: 35000, Date of acquisition: 20060930, Age: 4755, Current price: 28000, SpeedRPM: 7000
3.: Type: Harddisk, Name: Maxtor, Initial price: 25000, Date of acquisition: 20050228, Age: 5334, Current price: 20000, SpeedRPM: 7000
The content of the inventory has been written to computerproducts_out.txt
Done.
```

## 14.4. BPMN

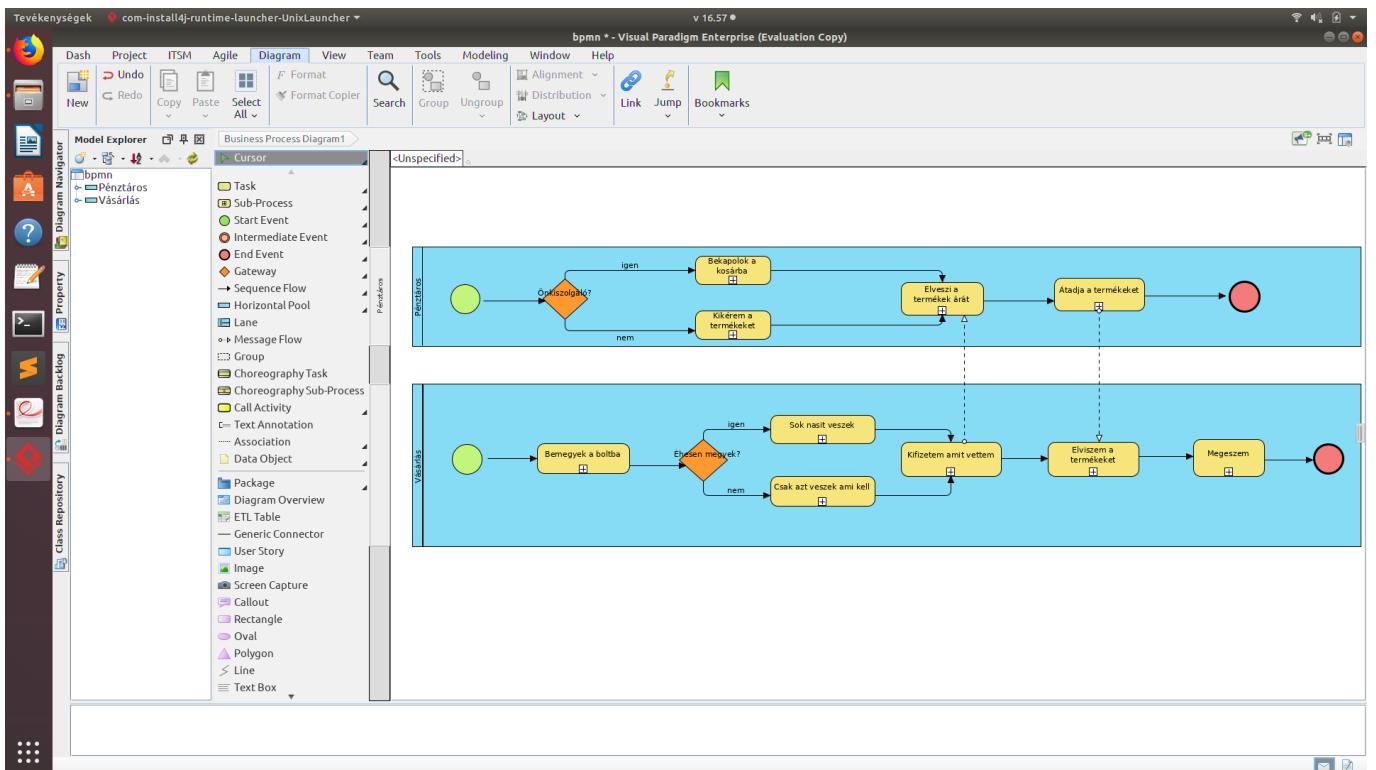
Rajzoljunk le egy tevékenységet BPMN-ben! [https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog \(34-47 fólia\)](https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog (34-47 fólia))

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A feladat nem volt túl bonyolult ebben a formában, hiszen egy a saját fejünk ből kipattanó ötletet kellett megvalósítani. Szerintem hasonlít is valamennyire az UML diagrammhoz, amivel eddig dolgoztunk, hiszen minden kettő egyfajta működést/felépítést mutat be. Ahogyan a BPMN (Business Process Model and Notation) nevéről is következik, főleg üzleti dolgok modellezésére használják, elvégre ez is egy modellező eszköz, amivel grafikus felületen alakíthatjuk ki a megállmodott folyamatábrát. Én egy bevásárlós példát hoztam, ahol a pénztáros és a vásárlás kapcsolatát próbáltam megjeleníteni, illetve egy pincit belevittem egy rossz szokásomat is a megjelenítésbe. Az ábra pedig a következő:



## 15. fejezet

# Helló, Chomsky!

### 15.1. Encoding

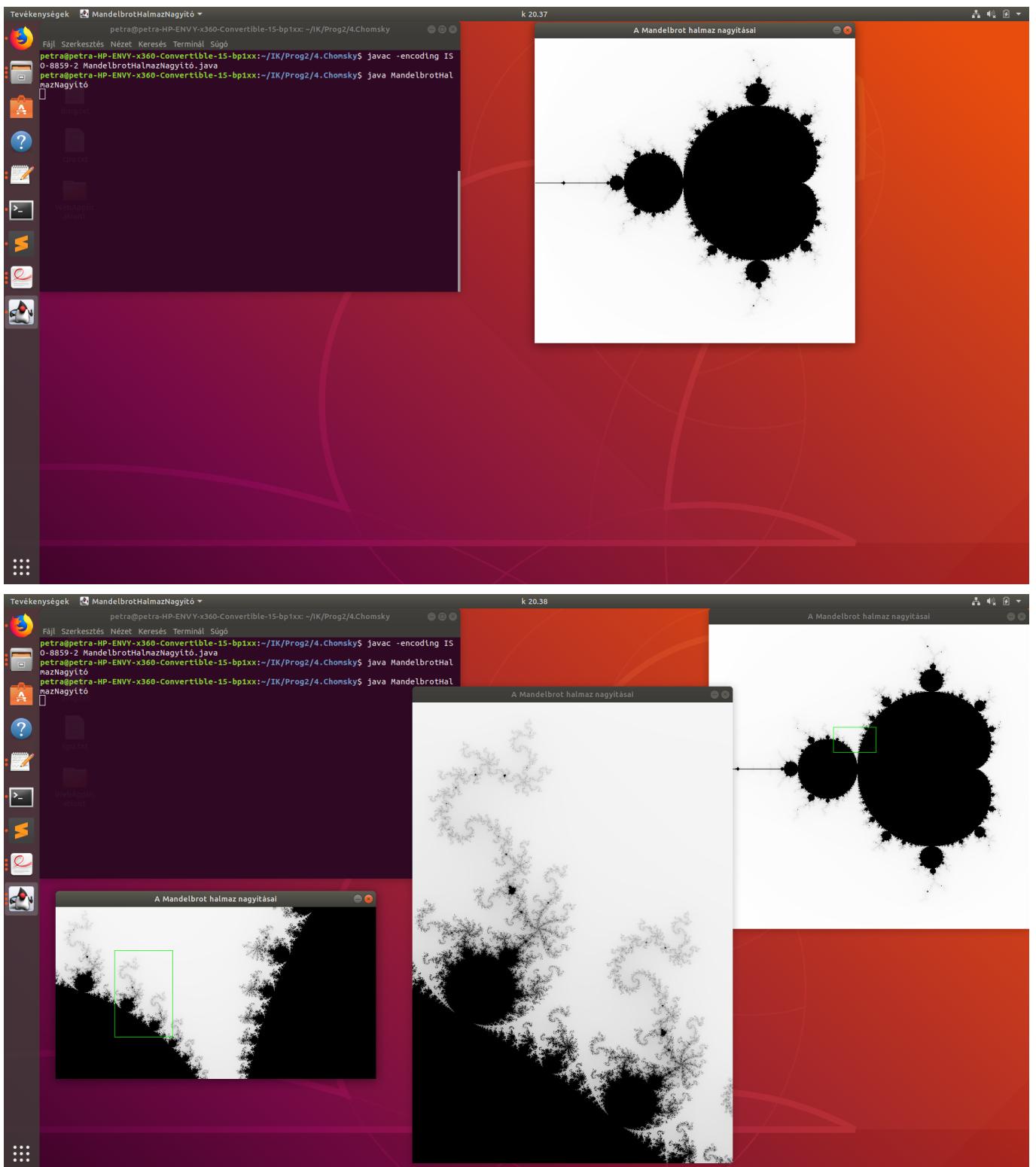
Fordítsuk le és futtassuk a Javat tanítok könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezes betűket! <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/adatok.html>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban a forráson nem kellett módosítani a megoldás érdekében, viszont tartalmaz olyan karaktereket (a forrás és a forrás neve), amely a fordító nem ismer fel alapjáraton, és rengeteg hibaüzenetet kapunk ha simán javac-cal próbálnánk fordítani. Az alap karakterkészletünk az ami nem jó, ugyanis az angol megnevezéseket nem érdemes vegyíteni a magyarral, tehát ha valamit nevesíteni szeretnénk, azt nem érdemes magyarul megtennünk. (A forrásunk egy tömörített állomány volt, és ebben találhattunk 3 Mandelbrot java fájl ha kibontottuk. Futtatáshoz szükséges mind a 3.) És a lényeg: Az ISO-8859 szabványra van szükség, hogy ne zavarjanak be a magyar nevek. Ha belegondolunk, végül is az olvasónaplóban már olvashattunk ezekről a karakterkészletekről, ez a feladat pedig jól szemléltette egynek a működését.



## 15.2. I334d1c4

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 tettek meg, akkor írasd ki és magyarázd meg a használt struktúratömb memória foglalását!)

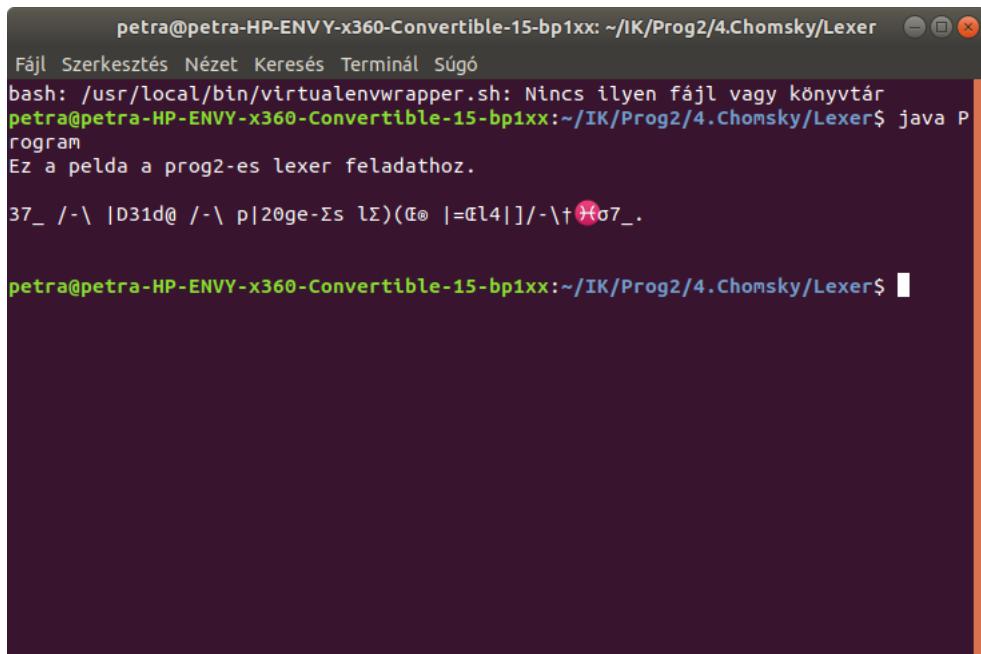
Tutorom volt: Tóth Attila

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Prog1-en már volt egy hasonló feladatunk, csak ott leetteléssel volt megoldva a feladat, nem egy oo nyelven. A Lexer osztályba kerültek be sztringek, amelyek tartalmazzák, hogy az adott betű mit vehet fel alakként (ez private, hiszen nem kellene rajta módosítani semikor). Majd aszöveg feltöltés után azt, hogy a megadott 4 választási lehetőségből melyikkel történik meg a helyettesítés, azt egy randommal választjuk ki. Tehát ebből is látszik, hogy eléggyé hasonlít a korábbi feladathoz, bár a megjelenítése azért változott. A feladatban a java Scanner osztálya is felhasználásra került, ezzel tudjuk a terminálból bekérni a bemenetet. Ha nem kap bemenetet a program, tehát simán enteröt tűünk, akkor megáll. Előtte viszont egy végtelen ciklussal kér be folyamatosan szöveget. A Lexer osztály pedig lex néven lett példányosítva. Egy pici gyorsítás volt a nagybetűk miatt, a 'text.toLowerCase()' átalakítja azokat kicsire, így nem kell külön vizsgálni a nagy és a kis betűket, hanem mehet egyben.



A screenshot of a terminal window titled "petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/4.Chomsky/Lexer". The terminal shows the following command and its output:

```
bash: /usr/local/bin/virtualenvwrapper.sh: Nincs ilyen fájl vagy könyvtár
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/4.Chomsky/Lexer$ java Program
Ez a pelda a prog2-es lexer feladathoz.

37_ /-\ |D31d@ /-\ p|20ge-Σs lΣ)(€® |=€l4|]/-\†€o7_.

petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/4.Chomsky/Lexer$
```

### 15.3. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Lásd vis\_prel\_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, textúrázás, a szintek jobb elkülönítése, kézreállóbb irányítás.

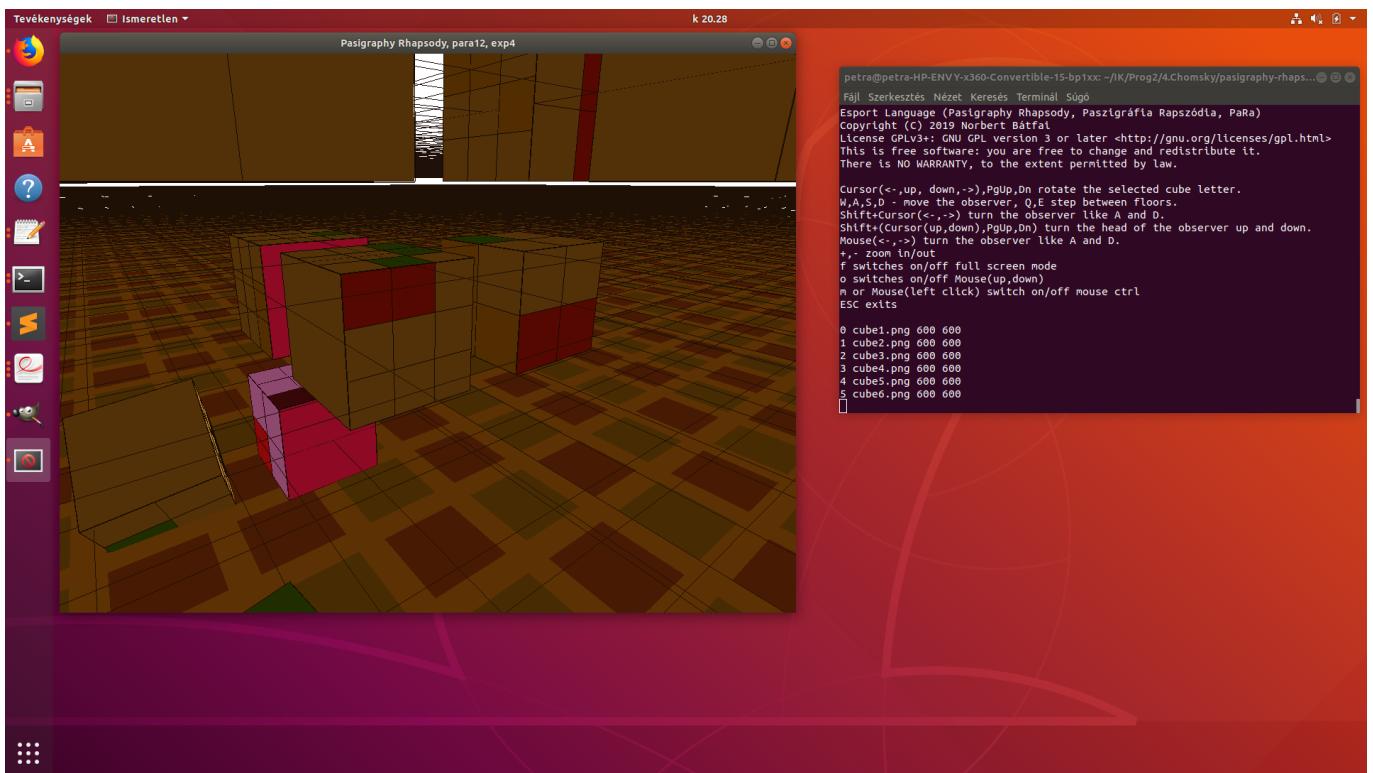
Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Arról, hogy mi is ez a Pasziográfia Rapszódia, a vis\_prel\_para doksiban olvashatunk részletesebben. Röviden összegezve pedig az esport mesterségesen kialakított nyelve, mivel a humunkulusz és a mesterséges humunkulusz között próbál meg kapcsolatot/kommunikációt kialakítani. Ez egy harmadlagos fejlesztendő

nyelv pontosan, amely az esport jáékba van beágazva. A feladatban pedig konkrétan a vizualizációval tudunk szórakozni. Visszatérve a humunkulusra pedig, ezt Neumann nevéhez köthetjük, és ennek a létrejöttéhez szükséges a központi idegrendszer, valamint a donaldi külső memória, illetve az ezekből kiinduló elméleti háttér. Ez az egész elmélet lényegében a komplexitás méréséről szól, tehát az idegrendszer és a külső memória, vagy akár a humunkulusz és a mesterséges humunkulusz komplexitását is össze tudja hasonlítani, és ami fontos neki, az az, hogy a humunkulusz által létrehozott mesterséges humunkulusz összehasonlítható legyen, vagy akár a mesterséges meghaladja a létrehozóét is. Nos ez lett volna az elméleti háttér röviden. Ha pedig ki szeretnénk próbálni, akkor először telepítenünk kell az opengl-t, és a további szükséget csomagokat. A fordítás futtatás a forrásban már szintén megtalálható, utóbbihoz még 6 darab képre van szükségünk, másra már nem. És valami ilyesmi eredményt kapunk ekkor.



## 15.4. Perceptron osztály

Dolgozzuk be egy külön projektbe a projekt Perceptron osztályát! Lásd <https://youtu.be/XpBnR31BRJY>

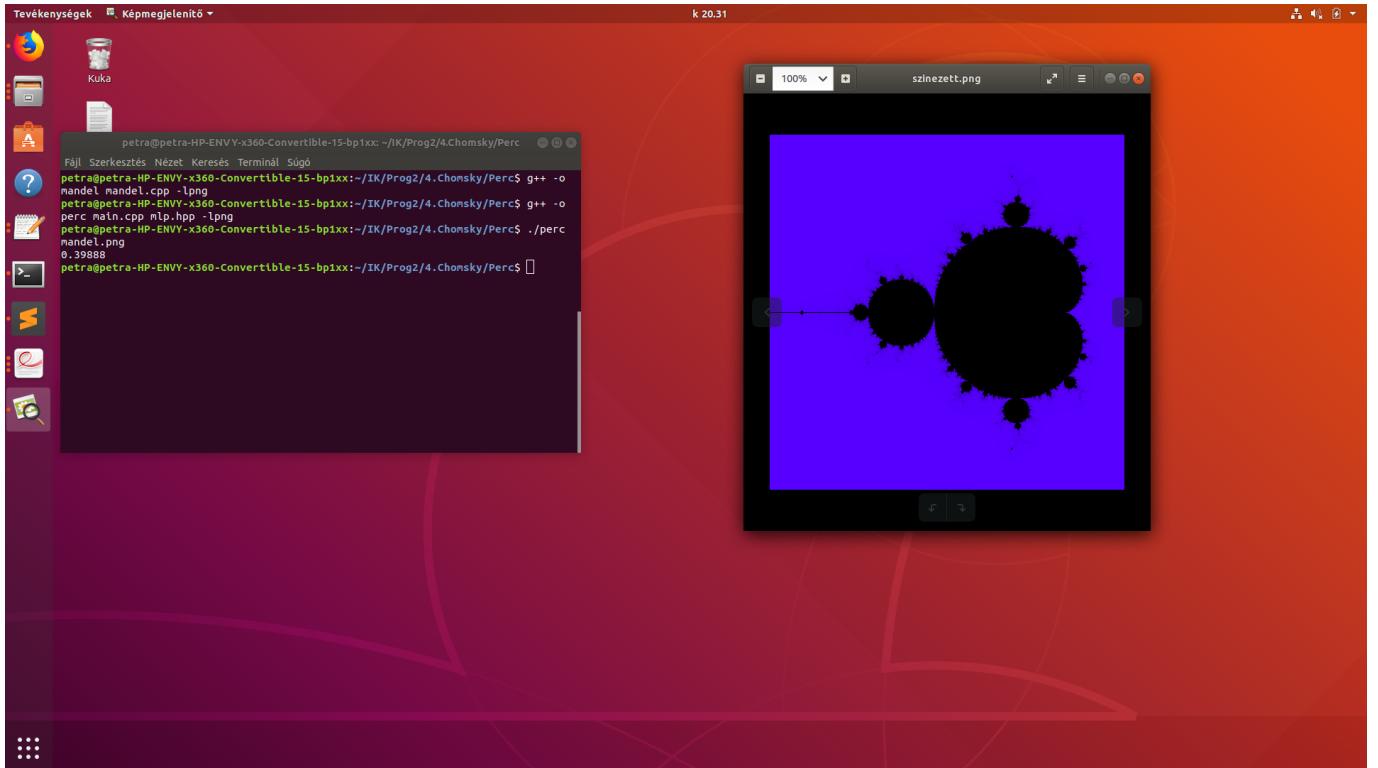
Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A perceptron osztály szintén egy Prog1-ről visszatérő feladat, vagyis részben. Ez egy neurális háló lényegében, és egy függvény értékkel dolgozik, de a működését akkor leírtam, nem részletezném újra. Most annyi módosítást kellett rajta végezni, hogy a számításokkal módosítani is tudjon. A számításokat a neuronok hajtják végre. Ez a módosítás pedig az RGB színek módosítása, vagyis az átszínezés. Ha tudjuk a színek kódjait, akkor nemcsak a 3 alapszínt tudjuk vele elérni, hanem másat is. A neuronok pedig minden bemenetinél van még jelentőség, ugyanis ezek között kapcsolat van (mint az idegek között),

ez a súlyösszeköttetés. Ezen pedig lehet módosítani. A végzett módosítások után pedig ha elindítom a perceptront (jelen esetben egy mandelbrotból generált képre), akkor ilyen eredményt kapok.



## 16. fejezet

# Helló, Stroustrup!

### 16.1. 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:

A program kiindulási alapja a fénykard.cpp volt, ugyanis mindenki tartalmazza a boost állományokat, vagyis azt, amit a feladat szeretne bemutatni. A boostoláshoz ebben a példában a Boost.Filesystem könyvtárra volt szükség. Mivel a JDK folyamatosan frissül, így a végeredmények is változhatnak verziótól függően. Én a 8-asat használtam. Amint a képen is látszik, elég sok osztályt tartalmazott az src. A futás végén a szám ezeknek a darabszámát jelöli, hiszen egyesével megszámolni egy embernek elég sokáig tartana. A forrásra kitérve kicsit pedig egy rekurzív bejárással lett megoldva. A bejárás során felhasználunk egy vektort is, szerintem elengedhetetlen elem a megoldáshoz. A vizsgálat során megnézi a program, hogy mappa vagy fájl az adott elem. Ha mappa, akkor belemegy, és annak a tartalmát vizsgálja, ha pedig fájlt talált, akkor annak a kiterjesztését fogja megnézni. Ha ".java" végződésű fájlt talált, akkor az kell nekünk, hiszen az biztosan egy java fájl. Ezután pedig lehet tovább a vizsgálat. Itt van benne egy számláló, hogy ha talált ilyen fájlt, akkor a változó értékét növelte meg eggyel, így a futás végén megkapjuk pontosan hány ilyen fájlt talált.

```
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/5.Stour/JDK
Fájl Szerkesztés Nézet Keresés Terminál Súgó
"ExtendedTextSourceLabel.java"
"TextSourceLabel.java"
"StandardTextSource.java"
"Underline.java"
"TextRecord.java"
"FontSubstitution.java"
"Script.java"
"FontStrikeDesc.java"
"FontResolver.java"
"LayoutPathImpl.java"
"XMap.java"
"AttributeValues.java"
"GlyphList.java"
"ScriptRunData.java"
"Type1GlyphMapper.java"
"FontDesignMetrics.java"
"NullFontScaler.java"
"CoreMetrics.java"
"FcFontConfiguration.java"
"CompositeFontDescriptor.java"
"FontScaler.java"
"BiDiUtils.java"
"StrikeCache.java"
"PhysicalFont.java"
"DelegatingShape.java"
"FontScalerException.java"
"FileFont.java"
"FontAccess.java"
"NativeStrike.java"
"SunFontManager.java"
"NativeFont.java"
"CharToGlyphMapper.java"
"Font2DHandle.java"
"TextLabel.java"
"FontRunIterator.java"
"FileFontStrike.java"
"FontManager.java"
"CompositeStrike.java"
"StandardGlyphVector.java"
"NativeStrikeDisposer.java"
"TrueTypeFont.java"
"FreetypeFontScaler.java"
"FontStrike.java"
"FontFamily.java"
"FontManagerFactory.java"
"ExtendedTextLabel.java"
"XRGlyphCache.java"
"X11TextRenderer.java"
"SunLayoutEngine.java"
"DelegateStrike.java"
"FontLineMetrics.java"
"CMap.java"
"FontstrikeDisposer.java"
17405
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/5.Stour/JDK$
```

## 16.2. Változó argumentumszámú ctor és Összefoglaló összevonva

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:

A feladat első verziója még prog1-re nyúlik vissza. Illetve a már meglévő csokor elkészítésemmel is fog egyezésekkel mutatni, mivel már azt a feladatot is úgy készítettem el, hogy a kép méret azonos maradjon. A változás pusztán annyi, hogy itt már a Perceptron osztály által visszaadott értékre nincs szükségünk, hanem csak egy átalakított képet szeretnénk visszakapni. Az összehasonlíthatóság végett most is a Mandelbrot halmaid által elkészített képet használom majd fel. Illetve itt jobban ki szeretném fejteni a feladatot, és ezáltal összevonni az "Összefoglaló" feladattal is. Ha szeretnénk, akkor szerintem még itt érdemes összekötni a feladatot az olvasónaplóval, ugyanis itt szintén dolgozunk az osztály miatt konstruktőrrel és destruktőrrel, valamint a változó argumentumszámmal is, de annak a kódcsipetét később megjelölöm.

A Mandelbrot-halmazról itt nem mesélnék feleslegesen, hiszen ez is egy prog1-es feladatból indult ki, illetve a múlthati csokorban az encoding által is újra szerephez juthatott. A feladatban a Perceptron osztály a halmaz által generált képet fogja átszínezni az rgb kódok manipulálásával, ami a forrásban látszódni is fog.

A Perceptron alapját a neurális hálók adják, vagyis lényegében ebből áll az egész. Ha hasonlítanom kellene valamihez, akkor hozzánk, az emberek működéséhez tudnám, hiszen nekünk is az idegrendszerünkben rengeteg ilyen neuron és neurális háló található, de a biológiai részletekbe inkább nem mennék bele. Ennek viszont a súlyozásban látom az egyik fő értelmét. A Perceptronban megjelenő neuronokat ugyanis össze kell kötni, másképp nem áll össze belőle a hálózat, nem maradhatnak szabad végződések. A súlyozásnak még a számításban van nagy szerepe, ugyanis a súlyok változtatásával módosul a számítások pontossága is, a hibák lehetőségének a csökkentése. Emiatt használják fel a Perceptron a gépi tanuláshoz.

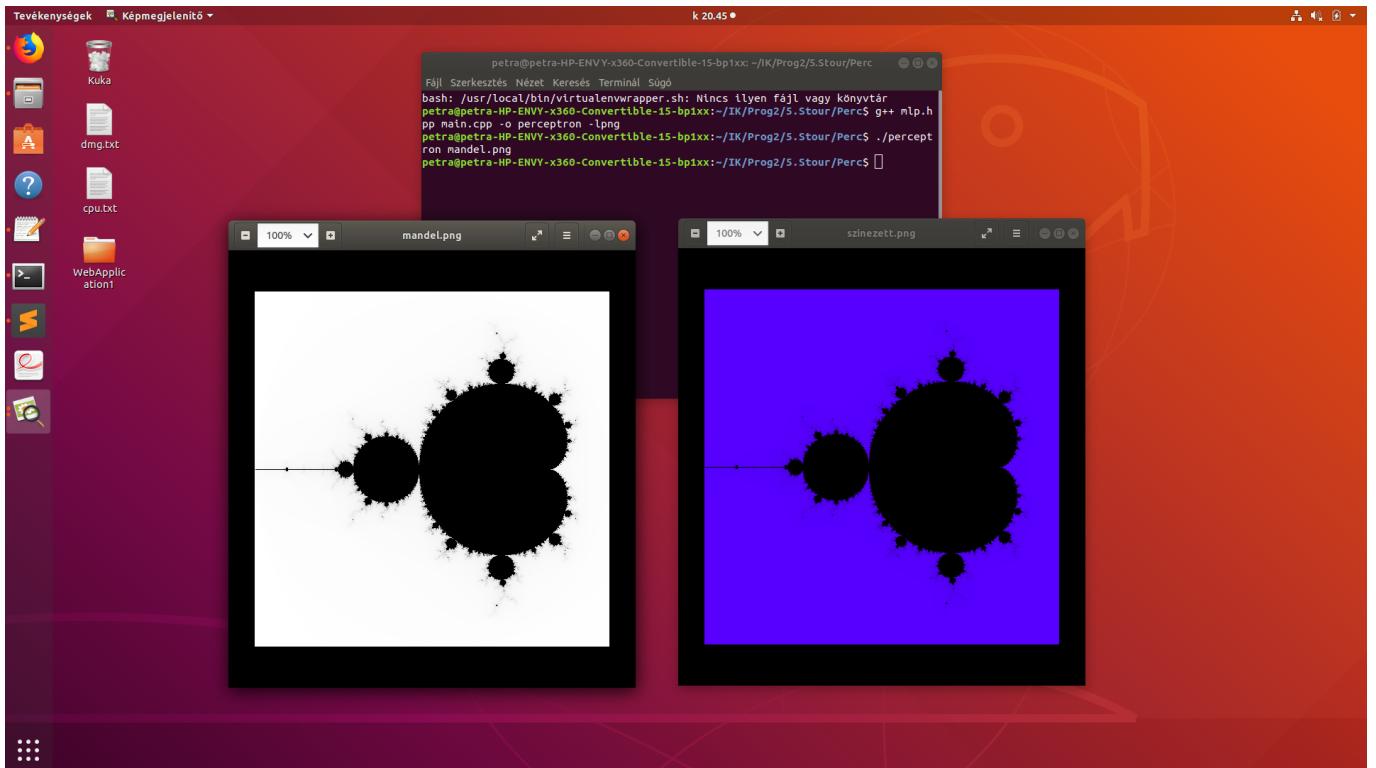
Ha jobban megvizsgáljuk, találhatunk benne egy sigmoid függvényt:  $1.0/(1.0+\exp(-x))$ . Ez a függvény több számításnál is szerepet kap. Szerintem ami lényegesebb, az a kép méreténél végzett számítása, hiszen azzal is dolgozik (szélesség\*magasság). A függvény értéke a  $[0,1]$  intervallumon van meghatározva, más eredményt nem is adhat, emiatt alkalmas valószínűség számításhoz is. Ezután foglalkoznánk következőnek a main.cpp-vel. Ebben lesz egy mlp.hpp includolva, de erre visszatérünk később. A másik include pedig az aktuális feladat miatt szükséges, hiszen egy képpel kell foglalkoznunk, ehhez pedig elengedhetetlen forrás a libpng csomag, amelyet az alábbi módon hívunk meg: (nélküle a program nem tudna képeket kezelní)

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

Ezután kezdhetünk foglalkozni a képpel. A meghatározás után a kép méretének létrehozunk egy size változót, amelyben rögtön 2 függvényel is találkozhatunk és fel is használjuk. Ezek a get.width() és get.height() függvények, tehát a szélesség és a magasság meghatározása, majd ezeknek a szorzata megadja a kép méretét. Ezután jön a példányosítás, amely new Perceptronnal, és ez visszamat a pointerrel a Perceptronra (ezért láthatjuk ott a \*-ot), majd a double-lel szintén. Ha itt nem használnánk pointert, működésképtelen lenne a program, hoszen nem adná át megfelelően az értékeket. A Perceptron most 3 réteggel fog dolgozni, ezt jelöli a példányosításkor a 3-as, ahol size van, vagyis az 1. és 3. réteg a kép méreteivel fog dolgozni (ezért kapunk vissza azonos méretű képet), a második, 256 értéket felvevő érték pedig az rgb kód miatt van. Majd ezek után for ciklusokkal járjuk be a képet, lényegében a pixelekből egy mátrix készül: png\_image[i][j]. Majd az átszínezés is ugyan ezen az elven fog végig menni, tehát a kép pixelein mátrix-ként megy végig, és egyesével átszínezi a meghatározott színű pixeleket. Ezért volt fontos a szélesség és magasság meghatározása az elején, mert a mátrix 2 for ciklusa ezek alapján lépked pontról pontra. Majd a memória felszabadítása előtt még látjuk a write függvényt, amelyet szintén tartalmaz a program elején meghívott csomag, és ezzel kapjuk vissza a módosított képet.

Az mlp.hpp fájlból még kiemelnék egy két dolgot, ami szerintem lényeges lehet a Perceptron megisméréséhez. Ha elkezdjük nézegetni, rögtön szemet szűrhet a Perceptron ( int nof, ... ) meghatározása. Ez lenne a változó számú paraméterlista, amelyről a feladat a nevét kapta, illetve az olvasónaplóban találkozhattunk vele. Előbbi a számukat fogja meghatározni, hiszen ez fogja a rétegek számát is megadni, mint ahogyan az látszott a példányosításnál (pár sorral ezelőtt, jelen esetben 3 volt), utóbbi pedig maga a paraméterlista (szintén látszott a példányosításnál, illetve leírtam melyik mi). Valamint amit még kiemelnék, azok a változó paraméterlistához szükséges dolgok, tehát a va\_list típus, mivel enélkül a va\_start() és a va\_args() is elérhetetlen lenne. A va\_start()-ra az inicializálásnál van szükség, a va\_args() pedig mint ahogyan az elnevezése is sugallja az argumentumokkal dolgozik. A lezárasuk (vagyis a lista lezárása) pedig a va\_end(), amelyet nem szabad kihagyni (egyfajta szabálynak is nevezhetném).

Megjegyzésként fűzném még ide, hogy a fordításnál még figyeljünk, ne ott csússzon be egy ügyetlen hiba. Vagyis eközben a -lpng kapcsolót ne hagyjuk le, mert fontos. A futtatásnál pedig ne felejtsük el megadni a módosítani kívánt képet.



## 17. fejezet

# Helló, Gödel!

### 17.1. STL map érték szerinti rendezése

Például: <https://github.com/nbatfai/future/blob/master/cs/F9F2/fenykard.cpp#L180>

Megoldás videó:

Megoldás forrása:

A feladatban rendezést kell végrehajtanunk, bár nem teljesen olyan formában, mint eddig. Itt a map-ra lesz szükségünk, amely lényegében kulcs-érték párként szolgál. Az elején abc szerint van rendezve, tehát a kulcsok lettek abc sorrendbe rendezve, majd ezen szeretnénk változtatni, és a végén az értékek szerinti csökkenő sorrendet szeretnénk kapni. Lényegében most ennek a bemutatásáról szól ez a feladat. Akkor most jöhet a forrás elemzése is kicsit. Amikor létrehozzunk az új mapunkat, akkor megjelenítünk egy insertet is, amely értékkadásra szolgál, a map létrehozásában pedig a make\_pair van a segítségünkre. Illetve az elején láthatjuk, hogy vektorokra is szükség van a rendezéshez, itt ezek a pair vektorok lesznek. A p1 illetve p2.second pedig a rendezésben vesz részt, ezekkel tudjuk érték szerint rendezni. A beginnel megadjuk, hogy az it legyen az első érték, amit vizsgál a rendezés során. A while ciklus fogja bejárni a mapokat, addig, ameddig a rendezés meg nem történt. A rendezett vektor szolgál arra, hogy a már elrendezett elemeket a megfelelő sorrendben letárolja. Majd a már említett p1 és p2.second tényleges rendezéséhez még felhasználunk egy lambda kifejezést, ez fogja véglegesen rendezni. Ez a p1 és p2-öt hasonlíta össze folyamatosan a vizsgálat alatt, és megnézi, hogy melyik a nagyobb. Ha a p1 a nagyobb, akkor jó, ha a p2, akkor nem. És ezt végig vizsgálja és rendezi egészen addig, ameddig minden érték a megfelelő sorrendbe nem kerül. Majd a kiíratásnál látszik ez pontosabban, a kezdeti kiinduló (abs szerinti rendezés), majd a végén az érték szerinti rendezés, és a kettő között láthatjuk a p1 és p2 folyamatos összehasonlítását.

```
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/6.Gödel/Map
Fájl Szerkesztés Nézet Keresés Terminál Súgó
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/6.Gödel/Map$./map
earth :: 4
jupiter :: 10
mars :: 7
moon :: 2
saturnus :: 9
uranus :: 3

10
4
=====
7
10
=====
7
10
=====
2
10
=====
2
4
=====
9
10
=====
9

Rendezés után:
jupiter - 10
saturnus - 9
mars - 7
earth - 4
uranus - 3
moon - 2
```

## 17.2. 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 szerepét!

Megoldás video:

Megoldás forrása:

Ennek a feladatnak elég erős focus kötődése van. Lényegében rangsorolja a csapatokat, és becslést végez a várható végkimenetlrol. Először is a Wiki2Matrix-szal kellett foglalkozni. Az eredeti forráshoz képest annyit szerintem érdemes volt átírni, hogy aktualizáljuk kicsit a progit. Ebben ugyanis szerepel egy mátrix, amely a csapatok egymás elleni eredményét tartalmazza 0,1,2,3 formában. A végeredmény is akkor fogja az aktuális állást mutatni, ha ez át van írva, hiszen azt a mátrixot, amit ez generál, át kell ültetni az alternatív tabellába. Ez volt a kereszt mátrix, amiről beszéltem. Ezt nem is részletezném tovább, hiszen a feladat a comparable szerepét kérdezi, ez pedig az alternatív tabellában fog szerepelni. Ennek nagy szerepe lesz a rendezésben, ugyanis ez végzi az összehasonlítást és a rendezést. Ebben szerepel egy compareTo (metódus), ez végzi a hasonlítást, ugyanis paraméterként megkap egy objektumot, és ezt összehasonlítja az aktuálissal, majd ezt adja vissza visszatérési értékként.

```
public int compareTo(Csapat a) {
 if (this.getErtek() < a.getErtek()) {
 return 1;
 } else if (this.getErtek() > a.getErtek()) {
 return -1;
 } else {
 return 0;
 }
}
```

Tehát megnézi, hogy melyik a nagyobb, vagy 0 értéket ad ha egyenlő a 2. A végleges rendezést a sort fogja elvégezni tehát: Collections.sort(uj). A interfészre pedig amiatt van szükségünk, hogy minden (típush) meg tudunk vizsgálni, amit abban deklaráltunk. A sortnál 3 rendezés megy végbe végül is, ha jobban megnézzük. A csapatok név szerinti rendezése, illetve utána a hozzájuk tartozó értékek. Majd a végén duplázva látjuk az eredményt, ezért a regi és az újnev felel.

```
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/6.Gödel/Alternatív
Fájl Szerkesztés Nézet Keresés Terminál Súgó
A "link" matrix

[0.0, 0.11764705882352941, 0.09090909090909091, 0.0, 0.11764705882352941, 0.13333333333333, 0.07692307692307693, 0.0, 0.15384615384615385, 0.05555555555555555555, 0.06666666666666666667, 0.11111111111111111111,], [0.11111111111111111111, 0.0, 0.09090909090909091, 0.15384615384615385, 0.11764705882352941, 0.06666666666666666667, 0.07692307692307693, 0.125, 0.0, 0.05555555555555555555, 0.1333333333333333, 0.05555555555555555555,], [0.0555555555555555, 0.058823529411764705, 0.0, 0.15384615384615385, 0.058823529411764705, 0.06666666666666666667, 0.07692307692307693, 0.0625, 0.0, 0.11111111111111111111, 0.06666666666666666667, 0.05555555555555555555,], [0.11111111111111111111, 0.058823529411764705, 0.09090909090909091, 0.0, 0.1176470582352941, 0.1333333333333333, 0.15384615384615385, 0.125, 0.15384615384615385, 0.11111111111111111111, 0.06666666666666666667, 0.11111111111111111111,], [0.11111111111111111111, 0.11764705882352941, 0.09090909090909091, 0.07692307692307693, 0.0, 0.06666666666666666667, 0.0, 0.125, 0.15384615384615385, 0.11111111111111111111, 0.06666666666666666667, 0.11111111111111111111,], [0.11111111111111111111, 0.058823529411764705, 0.09090909090909091, 0.07692307692307693, 0.11764705882352941, 0.1333333333333333, 0.11111111111111111111, 0.0555555555555555, 0.11764705882352941, 0.09090909090909091, 0.07692307692307693, 0.11764705882352941, 0.0, 0.0, 0.125, 0.15384615384615385, 0.11111111111111111111, 0.06666666666666666667, 0.11111111111111111111,], [0.11111111111111111111, 0.11764705882352941, 0.09090909090909091, 0.07692307692307693, 0.058823529411764705, 0.1333333333333333, 0.0, 0.0, 0.15384615384615385, 0.0555555555555555, 0.06666666666666666667, 0.05555555555555555555,], [0.11111111111111111111, 0.11764705882352941, 0.09090909090909091, 0.07692307692307693, 0.058823529411764705, 0.1333333333333333, 0.15384615384615385, 0.0625, 0.0, 0.11111111111111111111, 0.06666666666666666667, 0.11111111111111111111,], [0.0, 0.058823529411764705, 0.09090909090909091, 0.07692307692307693, 0.058823529411764705, 0.1333333333333333, 0.07692307692307693, 0.0625, 0.0, 0.11111111111111111111, 0.1333333333333333, 0.0,], petra@petra-HP-ENVY-x360-Convertible-15-bp1xx
```

```
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/6.Gödel/Alternatív
Fájl Szerkesztés Nézet Keresés Terminál Súgó
1 Budapest Honvéd 0.09917779791056482
1 Ferencváros 74.0
2 Diósgyör 0.09774347861387919
2 MOL Vidi 61.0
3 Mezőkövesd 0.0902789115133693
3 Debrecen 51.0
4 Újpest 0.08481739948545247
4 Budapest Honvéd 49.0
5 Puskás Akadémia 0.08453522013872308
5 Újpest 48.0
6 Kisvárda 0.08364630123971446
6 Mezőkövesd 44.0
7 MOL Vidi 0.08274348300580465
7 Puskás Akadémia 40.0
8 MTK Budapest 0.0821323626320022
8 Paks 39.0
9 Paks 0.07778621132385563
9 Kisvárda 38.0
10 Ferencváros 0.07711681612199368
10 Diósgyör 38.0
11 Szombathelyi Haladás 0.07085175651800024
11 MTK Budapest 34.0
12 Debrecen 0.06917026149664002
12 Szombathelyi Haladás 30.0
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/6.Gödel/Alternatív$
```

## 17.3. 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:

Megoldás forrása:

Ebben a feladatban a mandalát választottam. Ide a végeredményt fogom beszűrni. A gimpben van egy külön konzol (Script-Fu), amely a lispet használja saját nyelvként. Ennek segítségével akár egyszerűbb számításokat, de akár érdekes képeket is készíthetünk. Ilyen például a króm effektes keret, vagy egy mandala szövegből elkészítve. Nos, hogyan is készül ilyen formában egy mandala. Mivel egy egyszerű néhány betűs szövegre van csak szükség, így ha ezt körbe forgatjuk (360 fokban), akkor meg is kapjuk. Ezt egy függvény hajtja végre, ez pedig a gimp-item-transform-rotate-simple. A forrásban tudjuk állítani a betűk méretét, a szöveg szélességét és magasságát, valamint a betűtípusát. Például:

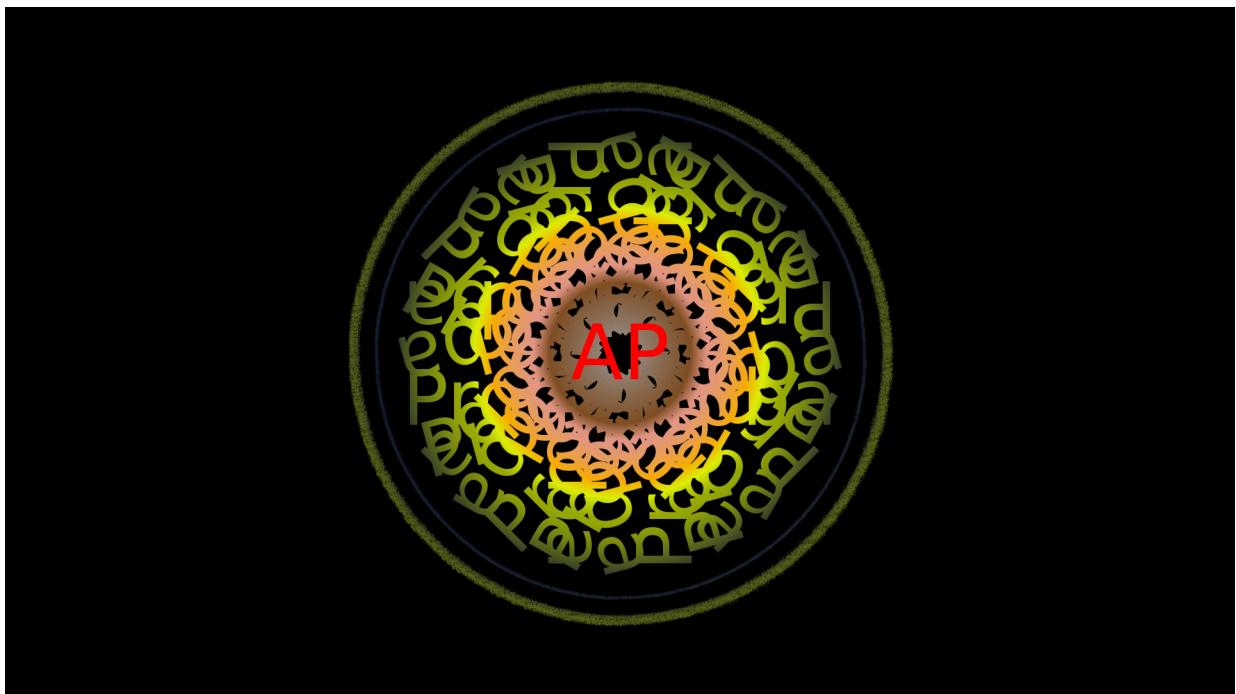
```
(set! textfs-width (- textfs-width 70))
(set! textfs-height (- textfs-height 70))
```

Létezik egy drawable függvény is, ami a kirajzolásra szolgál. Itt pixelekben kell gondolkozni, amikor rajzolni szeretnénk valamit. Egy példa a drawable-re, de ez persze többször is szerepel a forrásban, a meghatározások szerint változón.

```
(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)))
```

Ezek után nem sokkal találkozhatunk a gimp-context-set-brush-size függvénytel, aminek az értéke az én esetemben 22. Ez szintén a rajzolásnál kap szerepet, ugyanis ez jeleníti meg az ecset méretét amivel elkészül a kép, valamint a keret is hasonló módon van megadva, egy függvénytel, valamint az annak adott érték állítja be a vastagságát. Mivel két keret van, ez kell 2-szer, és érdemes különböző értékeket adni, így más lesz a vastagságuk. A futtatás pedig: Tehát megadjuk a szöveget amit megjeleníteni szeretnénk, valamint azok mérete, betűtípusa és a színek, amilyenre színezni szeretnénk. Pontosabban színskála a színátmennet miatt. Amúgy szerintem érdemes lementeni a forrást, így bármikor használható, és egy grafikus felületen tudjuk ezeket beállítani, nem a forrásban kellene minden módosítani.

```
(script-fu-bhax-mandala "forgatottszöveg" "középenschöveg" "Ruge Boogie" 120 ←
1920 1080 '(255 0 0) "Shadows 3")
```



# 18. fejezet

## Helló, Valami!

### 18.1. FUTURE tevékenység editor

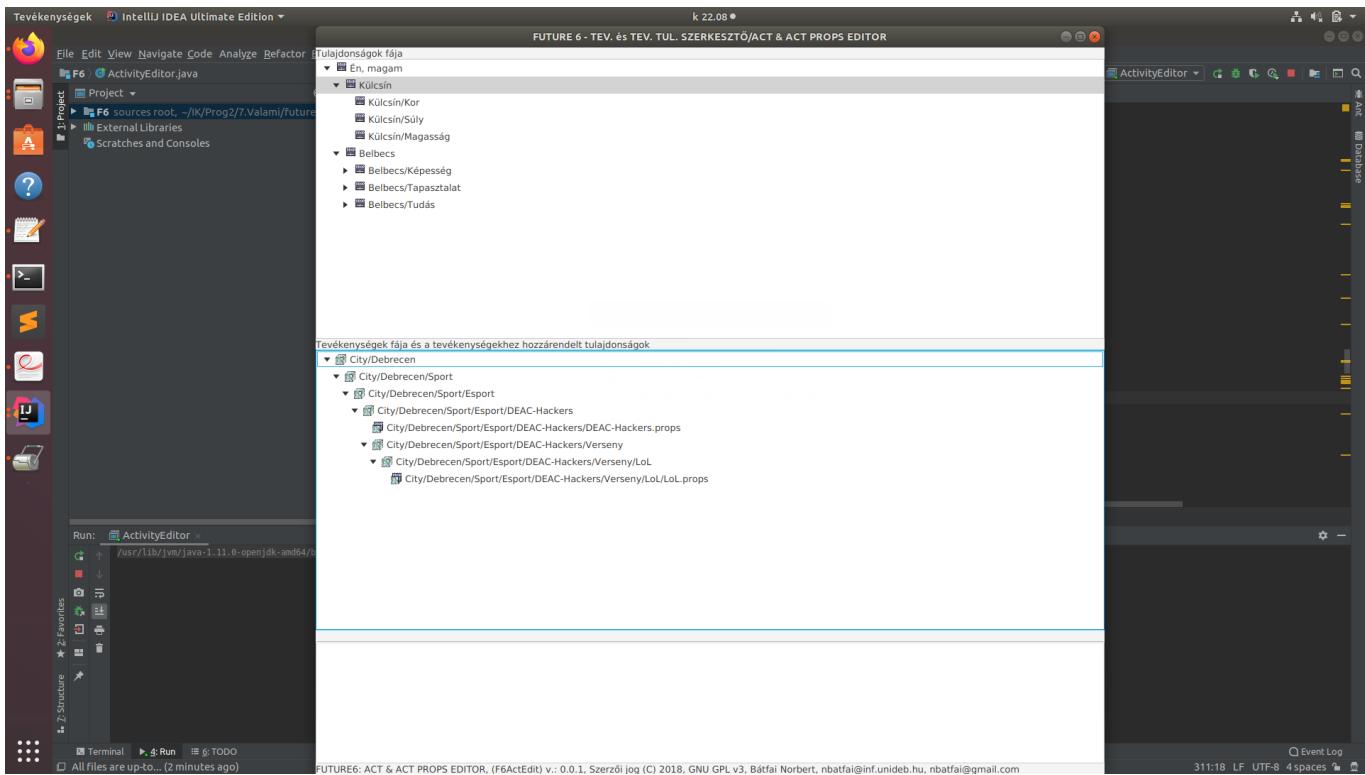
Javítsunk valamit a ActivityEditor.java JavaFX programon! <https://github.com/nbatfai/future/tree/master/cs/F6>  
Itt láthatjuk működésben az alapot: <https://www.twitch.tv/videos/222879467>

Megoldás videó:

Megoldás forrása:

A feladat hiba javítás, vagy új funkció hozzáfűzése volt, én az előbbivel szeretnék foglalkozni. A feladathoz szükséges volt a javafx telepítése, majd a kipróbáláshoz egy IDE-t, pontosabban az IntelliJ-t használtam. Az egyik hiba a fájlok létrehozásánál volt, nálam például az új fájlok amik létrehozásra kerültek a mappán belül bezárás után eltűntek. Illetve egy másik pedig, hogy azonos fájl névvel és kiterjesztéssel nem szerepelhet, erre lehetne megoldás ha a fájl nevében megjelenne egy plusz valami, ami mutatja, hogy más. Például egy zárójelben egy szám vagy karakter. Illetve még órán is megfigyelhettük tetszetől közben, hogy ha létrehoztunk egy filet, és azt átnevezzük, akkor a könyvtárában az nem neveződik át ténylegesen, tehát bezárás után szintén elveszik. A javításra a próbálkozások a forrásban figyelhetők meg. És az egyik hiba kiküszöbölése ami a feladat volt:

```
try {
 if (oldf.isDirectory()) {
 oldf.renameTo(newf);
 } else {
 oldf.renameTo(newf);
 }
} catch (Exception e) {
 System.err.println(e.getMessage());
}
```



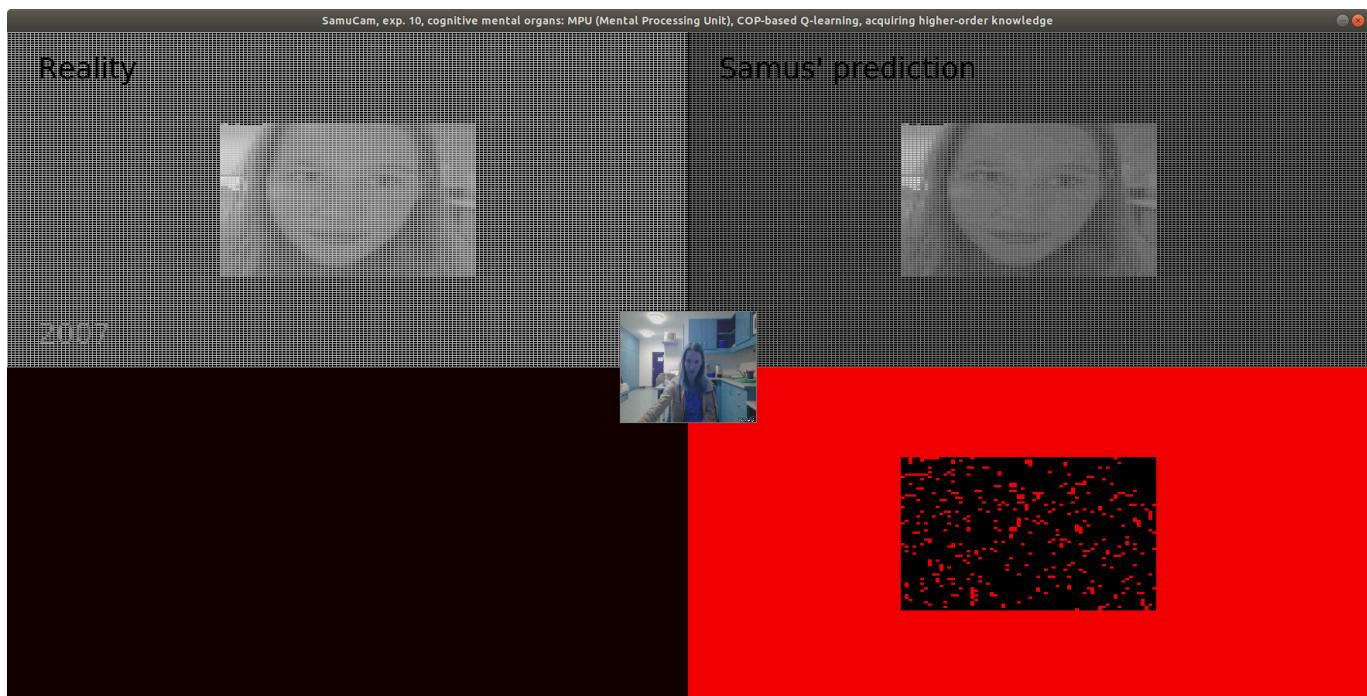
## 18.2. SamuCam

Mutassunk rá a webcam (pl. Androidos mobilod) kezelésére ebben a projektben: <https://github.com/nbatfai/SamuCam>

Megoldás videó:

Megoldás forrása:

Annyiban hasonlít a feladat a BrainB-hez, hogy itt is Qt-ra van szükségünk, illetve az opencv-t is telepíteni kellett. Akkor röviden kezdenék a Qt használatával a terminálosfuttatáshoz. Először is qmakeal kell, ez pedig a következő parancssal hajtható végre: `~/Qt/5.12.2/gcc_64/bin/qmake SamuLife.pro`. Ezután nyomjunk egy make parancsot, ezután pedig már tényleg minden fájlunk megesz a futtatáshoz, ami `./SamuCam --ip http://127.0.0.1:8081/ 2>out`, tehát localhoston futtatunk. Még ez is most olyan, mintha csak `./SamuCam` parancs lett volna, mert a kamera még nem üzemképes. Laptopon csináltam a feladatot, tehát ip webkamera nélkül. Tehát ezután telepíteni kell a motion-t a gépre, és a config fájlját beállítani. Majd futtatásnál egy terminálban indítsuk el a kamerát, a másikban pedig a Samut, és így már működik. A forrásba egy stringen keresztül be van építve hogy a terminálból bekapott bemenetből olvassa be a kamera ip címét, és hogyan használja azt. A kamera beállításánál fontos, hogy nem csak 1-1 képre van szükségünk, hanem egy streamre, tehát azt kell továbbítania a kamerának a programhoz. Még szintén a forráson belül is tudjuk állítani a kamera beállításait, nem csak a motion configurálásánál. Ilyen például a szélesség és a magasság, tehát ahogyan megjelennek a képek a samuban. Ha pedig kipróbáljuk, megfigyelhetjük, hogy csak a tényleges arcokat ismeri fel, mást nem, és azt kezdi el elemezni, megtanulni, később pedig kipróbálhatjuk, hogy felismer e minket például újra.



### 18.3. BrainB

Mutassuk be a Qt slot-signal mechanizmust ebben a projektben: <https://github.com/nbatfai/esport-talent-search>

Megoldás videó:

Megoldás forrása:

A prográmmal már találkozhattunk prog1-en is, és mivel már ott is foglalkoztam vele kicsit ezért bevezetőként csírnék belőle. A lényege viszont még mindig ugyan az, az esportolók képességeinek felmérése, vagy akár átlag emberekkel is lehet tesztelni, érdekes eredmények születhetnek belőle. Egy kirajzoló osztályunk van. A program lényegében az esportolók mérésére szolgál, követni kell egy bizonyos pontot, vagyis a hősünket. A hősünket vagyis a karakterünket egy külön osztály segítségével határozzuk meg a programban. Azt teszteli, hogy mennyire tudják az emberek a zavaró tényezők ellenére is követni a hősünket. A zavaró tényező például az, hogy plusz karakterek kerülnek be, a program pedig a követni kívánt karaktert is egyre zavaróbb módon mozgatja. A program futása közben adatot gyűjt, amit a futás után, ha bezártuk vagy ha végigvittük a játékot, és ezt eredményként kirakja nekünk egy külön fájlba/mappába, a programkód mellé. Mivel ehhez is szükséges a Qt program, így a fordítás futtatásra ügyelnünk kell. A fordítása: /home/user/QT/5.12.2/gcc\_64/bin/qmake BrainB.pro, ezután make parancs, majd a futtatása ./BrainB . Nos akkor jöhet is a kérdés, a Qt slot-signal mechanizmus, erről írnék még kicsit röviden. A signalt a Q\_Objectben találhatjuk meg bemutatásképp. Ennek a segítségével működtethetők a c++-os dolgok Qt-ben is. A fájlok létrehozásában is ez segít (qmake, hiszen ekkor generálódik le még több fájl is). A korábban ezek által legenerált kód szükséges a megfelelő működéshez, hiszen a jelek és a slot mechanizmus is igényli, ez lenne a slot-signal mechanizmus. A forrásban szerencsére az író most utaló neveket használt, ebből tudjuk, hogy a signalok mikor kerülnek használatba. Van egy connect parancs is, ennek a jelentősége pedig, hogy összekapcsolja a signalokat a slottal. Például:

```
connect (brainBThread, SIGNAL (heroesChanged (QImage, int, int)) ,
```

```
this, SLOT (updateHeroes (QImage, int, int)));
```



## 19. fejezet

# Helló, Lauda!

### 19.1. Port scan

Mutassunk rá ebben a port szkennelő forrásban a kivételkezelés szerepére! <https://www.tankonyvtar.hu/hu/tartalom/tanitok-javat/ch01.html#id527287>

Megoldás videó:

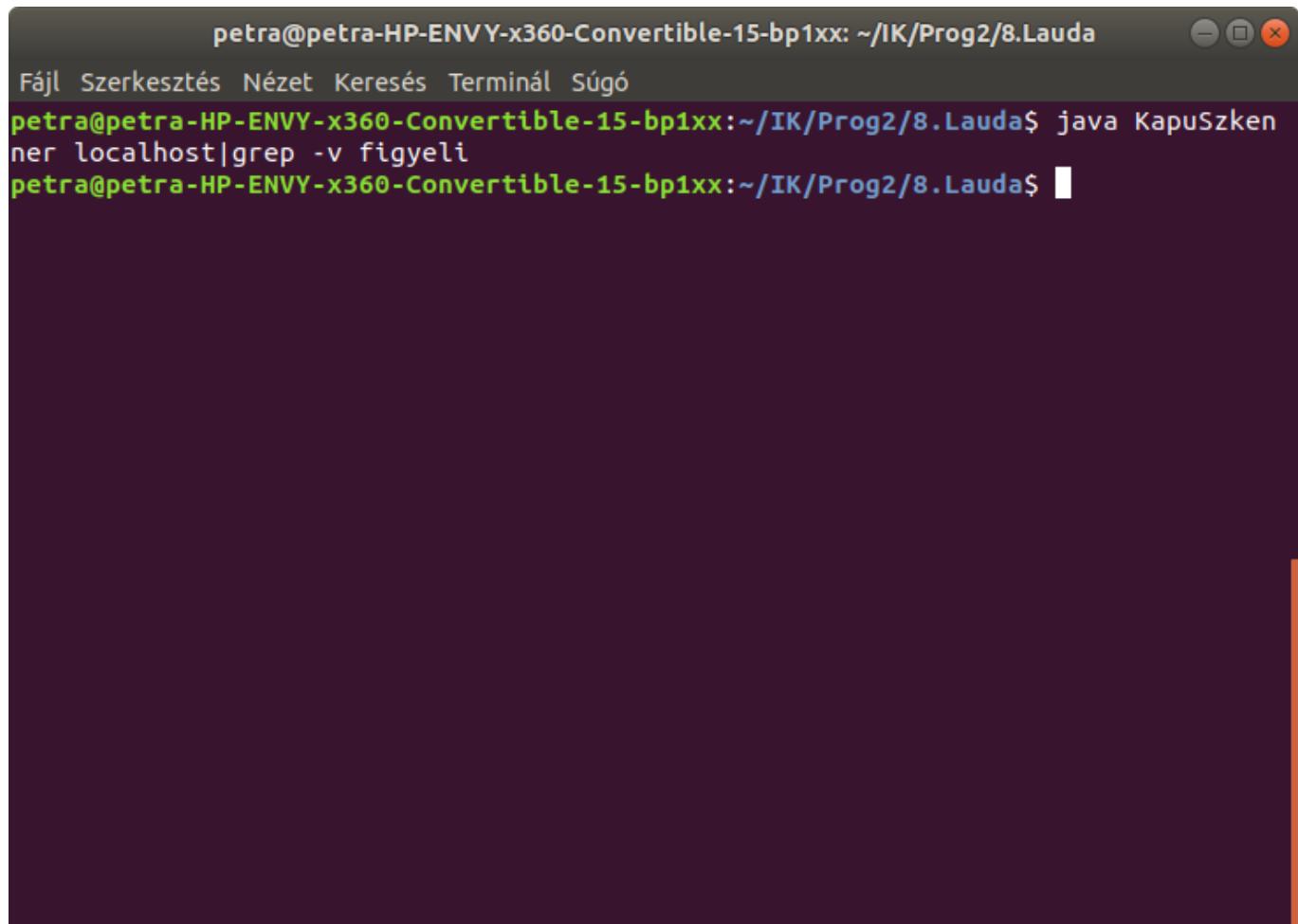
Megoldás forrása:

A kivételkezelésről már az olvasmányokból is hallhattunk. Ez lényegében egy pici példa a megvalósítására. Ehhez szükséges a try-catch. A kód elején van megadva egy argumentum, ez azt a célt szolgálja, hogy a bemenetre a program várjon egy argumentumot, ami lényegében egy címet takar. A teszteléshez érdemes localhosttal kipróbálni, ezzel pedig végig nézzük egy for ciklussal a portjainkat egészen 1024-ig, majd eredményül megkapjuk, hogy vannak-e nyitott portjaink, pontosabban a tcp kapcsolatokat vizsgálja végig. Ha sikerül ezzel új kapcsolatot létrehoznia, akkor találhatunk ott socket folyamatot, ha nem, akkor exceptiont, vagyis kivételt kapunk, tehát ekkor kapjuk azt, hogy "nem figyeli". Ez pedig ahogy a program megpróbál kapcsolatot létesíteni: Illetve láthatunk egy try-catch szerkezetet is, amely a kivételkezelés. Ha átugrik a hibakezelő részre, ergo ha nem talált kapcsolatot, akkor nem jön a nem figyeli felirat.

```
java.net.Socket socket = new java.net.Socket(args[0], i)
```

Ha pedig kíváncsiak vagyunk, de nem akarjuk végig vizsgálni azt az 1024 kiírt eredményt, akkor greppel le is szűrhetjük.

```
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/8.Lauda
Fájl Szerkesztés Nézet Keresés Terminál Súgó
993 nem figyeli
994 nem figyeli
995 nem figyeli
996 nem figyeli
997 nem figyeli
998 nem figyeli
999 nem figyeli
1000 nem figyeli
1001 nem figyeli
1002 nem figyeli
1003 nem figyeli
1004 nem figyeli
1005 nem figyeli
1006 nem figyeli
1007 nem figyeli
1008 nem figyeli
1009 nem figyeli
1010 nem figyeli
1011 nem figyeli
1012 nem figyeli
1013 nem figyeli
1014 nem figyeli
1015 nem figyeli
1016 nem figyeli
1017 nem figyeli
1018 nem figyeli
1019 nem figyeli
1020 nem figyeli
1021 nem figyeli
1022 nem figyeli
1023 nem figyeli
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/8.Lauda$
```



A screenshot of a terminal window titled "petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/8.Lauda". The window shows the following command being run:

```
java KapuSzken
ner localhost|grep -v figyeli
```

## 19.2. AOP

Tutorom volt: Tóth Attila

Szőj bele egy átszövő vonatkozást az első védési programod Java átiratába! (Sztenderd védési feladat volt korábban.)

Megoldás videó:

Megoldás forrása:

A feladat alapja a binfa és az aspectj. A fánk alapjároaton inorder bejárású, most pedig kapnunk kellene egy pre- és egy posztorder bejárást is. Itt jön képbe az aspectj, amivel megoldható a probléma, tehát a többi eredmény az eredeti forrás módosítása nélkül. Először telepíteni kell az aspectj-t, majd azután jöhet a kínlódás. Az aspectj fájlban kell ezek után megírni a kétféle bejárást, de még igazán nem is ez a fogós része, hanem a "becsatolások", hiszen valahogyan meg kell oldani, hogy a binfa java fájlja alapján dolgozza fel, tehát az itt megírt részekhez valahogyan hozzá kellene társítani a fát. Erre szolgál a pointcut, amiben meghatározzuk magát a pointcutot, a nevét és az argumentumait, valamint lehetőségünk van arra is, hogy meghatározzuk, hogy ez mikor és pontosan hol fusson le egy metódushoz viszonyítva. Ezt próbáltam az előbb magyarázni a becsatolással. De akkor a példa a működésre:

The screenshot shows a dual-pane text editor interface. The left pane displays the content of the file `postorder.txt`, which contains the following text:

```
-----1(3)
----0(2)
----0(3)
----1(3)
---1(2)
--1(1)
```

The right pane displays the content of the file `preorder.txt`, which contains the following text:

```
-.../(1)
...-0(2)
...-1(3)
...-1(2)
...-0(3)
...-1(3)
```

The interface includes a vertical toolbar on the left with various icons, and status bars at the bottom of each pane indicating file paths and line numbers.

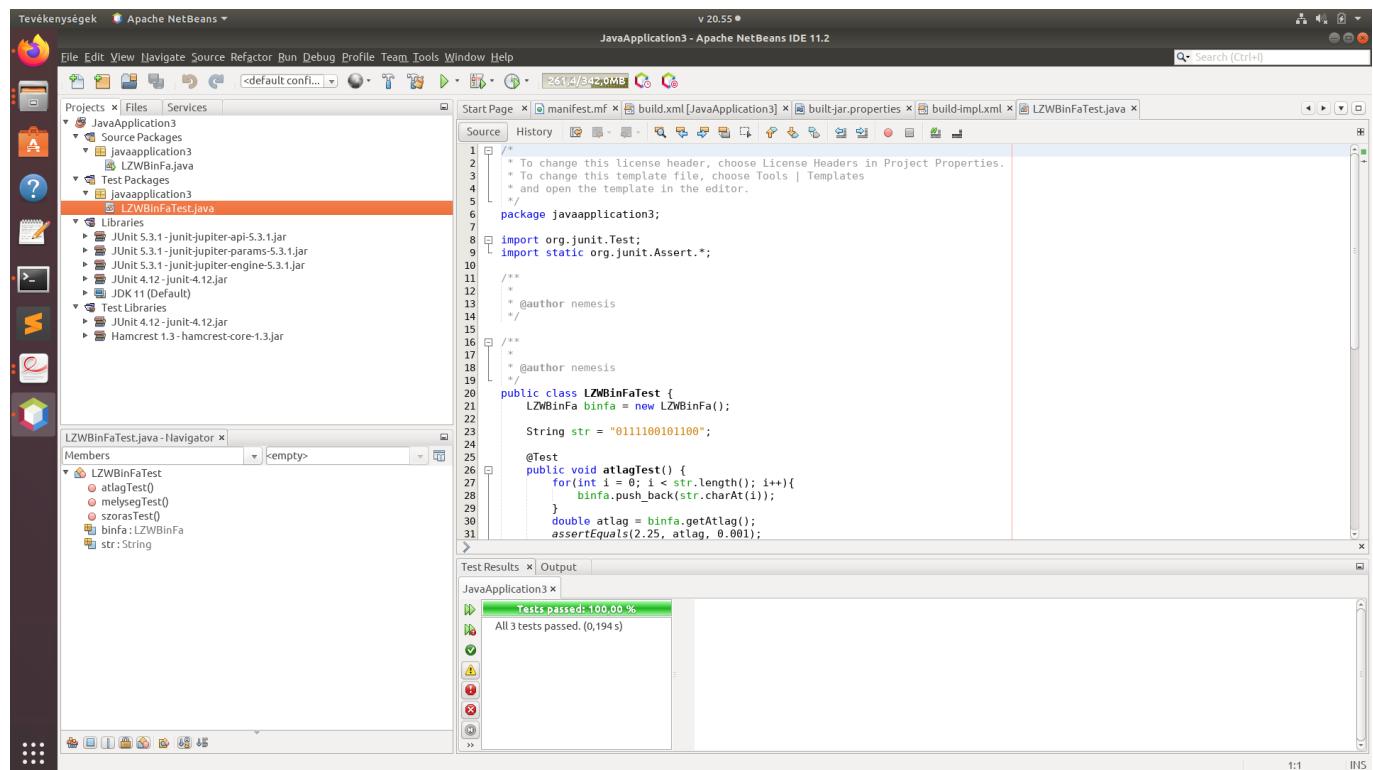
## 19.3. Junit teszt

A [https://progpater.blog.hu/2011/03/05/labormeres\\_otthon\\_avagy\\_hogyan\\_dolgozok\\_fel\\_egy\\_pedat poszt k  zzel sz  m  tott m  lyseg  t   s sz  r  s  t dolgozd be egy Junit tesztbe \(sztenderd v  d  si feladat volt kor  b  ban\).](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).)

Megold  s vide  :

Megold  s forr  sa:

A Junit teszt l  nyeg  ben pontoss  g  t m  r, illetve   sszehasonl  t  st v  gez. Java programokkal hajt v  gre unit tesztet. A feladat is meghat  rozta, hogy a binfa java v  ltozat  t tesztelj  k le. Amennyiben a f  ggv  nyek helyesek az alapk  nt haszn  lt f  ban,   s az eredm  nyt   sszehasonl  tjuk a k  zzel levezetettben, akkor egyeznie kell a kett  nek. Nos a teszt ezt ellen  rzi le. A megold  shoz netbeanst haszn  ltam, pontosabban a junit tesztel  t, ami ide be van   p  tve, teh  t itt r   is lehet engedni a tesztel  st. A binf  r  l pedig nem kezden  k el megint   rni, azzal kor  b  ban m  r tal  lkohattunk.



## 20. fejezet

# Helló, Calvin!

### 20.1. MNIST

Tutorom volt: Tóth Attila

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel, [https://progpater.blog.hu/2016/11/13/hello\\_sbol](https://progpater.blog.hu/2016/11/13/hello_sbol) Háttérként ezt vetítsük le: <https://prezi.com/0u8ncvvoabcr/no-programming-programming/>

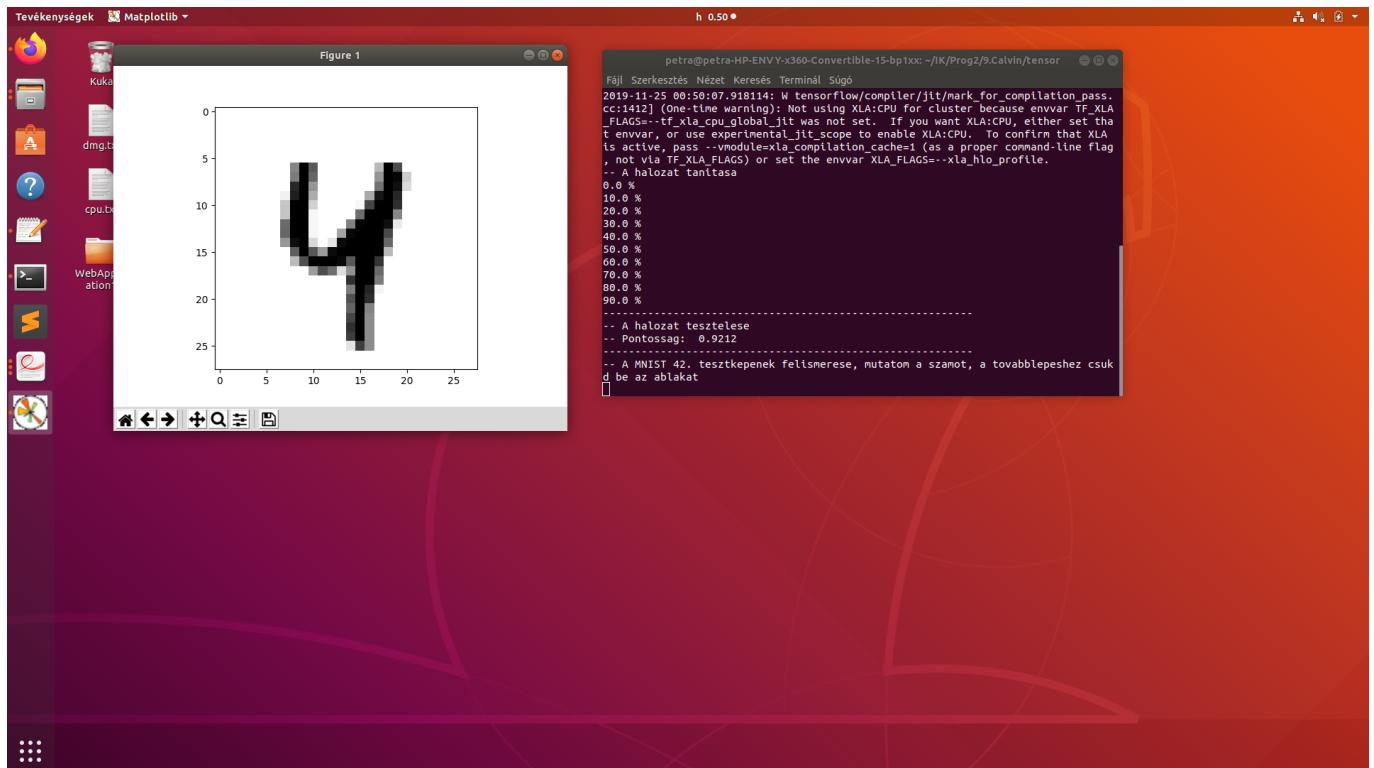
Megoldás videó:

Megoldás forrása:

A feladat megoldásához első lépésként telepítenünk kell a tensorflowt. Ez egy Google által fejlesztett program, ami a tanulást szolgálja, pontosabban a gépek betanításában próbál segítséget nyújtani. A mostani feladatban számokat kell tudnia felismerni képről, ehhez viszont nekünk kell betanítani a programot. A felismerési folyamat során a program súlyokat használ, és az alapján próbálja meghatározni, hogy mennyire egyezik az adott kép azokkal, amelyekkel korábban már találkozott. Első lépés a hálózat betantása, majd ezután meghatározza a program a számítások várható pontosságát. A képeket a következő képpen olvassa be, ha saját képet kap:

```
img = readimg()
```

A tanítás után először egy tesztképet kap, egy 4-est (sikeresen felismerte), majd ezután jött a saját kép, amit szintén jól ismert fel. Viszont mivel ez lényegében egy neurális háló, a pontossága befolyásolható például a rétegek (vagy hiddenek) számának a módosításával, bár meg kell jegyezni, hogy ez egy módosított neurális háló, kicsit másabb a működése az általunk eddig megismertéhez képest.



```
petra@petra-HP-ENVY-x360-Convertible-15-bp1xx: ~/IK/Prog2/9.Calvin/tensor
Fájl Szerkesztés Nézet Keresés Terminál Súgó
-- A halozat tanitása
0.0 %
10.0 %
20.0 %
30.0 %
40.0 %
50.0 %
60.0 %
70.0 %
80.0 %
90.0 %

-- A halozat tesztelese
-- Pontosság: 0.9143

-- A MNIST 42. tesztkepenek felismerése, mutatom a számot, a továbblepeshez csukd be az ablakat
-- Ezt a halozat ennek ismeri fel: 4

-- A saját kezi 8-asom felismerése, mutatom a számot, a továbblepeshez csukd be az ablakat
-- Ezt a halozat ennek ismeri fel: 8
```

petra@petra-HP-ENVY-x360-Convertible-15-bp1xx:~/IK/Prog2/9.Calvin/tensor\$

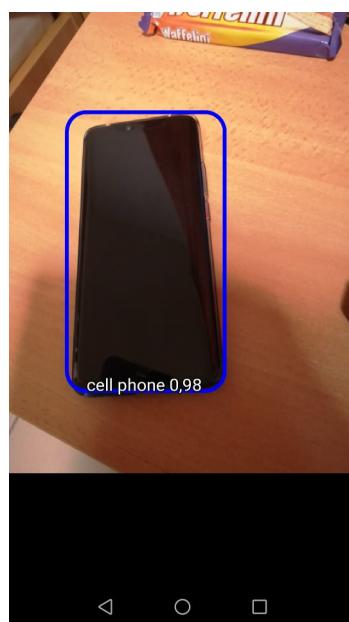
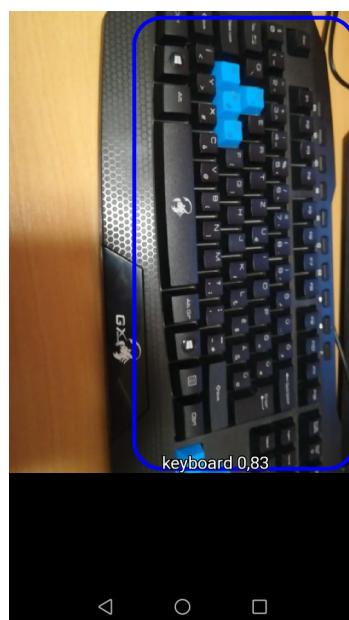
## 20.2. Android telefonra a TF objektum detektálója

Telepítsük fel, és próbáljuk ki!

Megoldás videó:

Megoldás forrása:

Jelen esetben tesztelés volt a feladat. A forrás a tensorflow által készített detektáló volt, amelyet a tensorflow github oldalán találhatunk meg és mindenki számára elérhető. Mivel az egész projekthez hozzá lehet férni, így a legcélszerűbb megoldásnak azt láttam, ha az android studiot hívom segítségül, és telefonon keresztül futtatom. Maga a program hagy kívánnivalót maga után, hiszen ha megnézzük elég érdekes teszteredményeket produkál le. Igaz, vannak tárgyak amiket felismer, de azokat is sokszor úgy, hogy abszolút nem biztos abban, hogy helyes volt e. Néhány példa (ahol a billentyűzetet és telefont felismeri, utóbbit végre határozottan, de a harmadik példánál a csoki és a hotdog szerintem kicsit más):





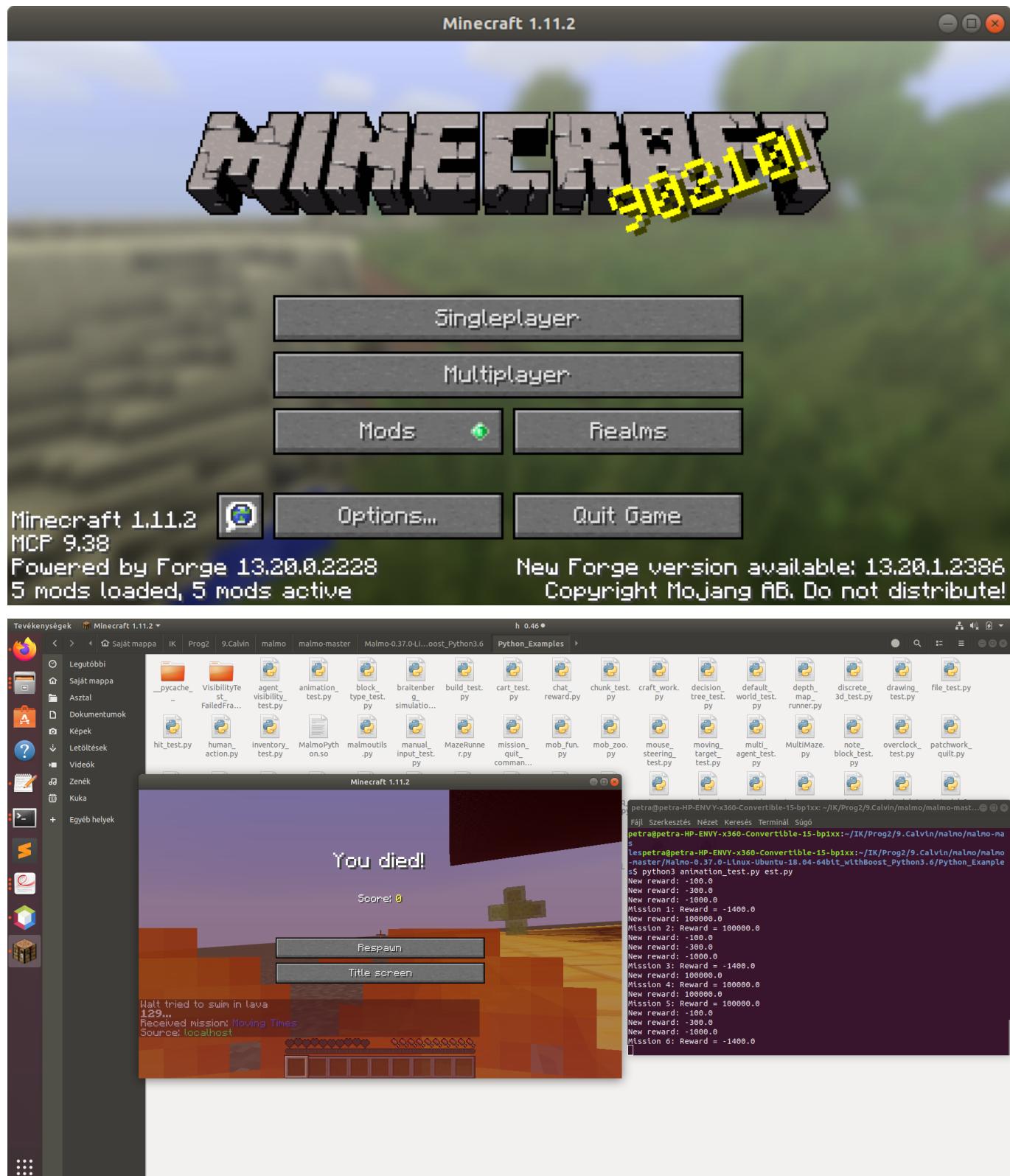
### 20.3. Minecraft MALMO-s példa

A <https://github.com/Microsoft/malmo> felhasználásával egy ágens példa, lássd pl.: <https://youtu.be/bAPSu3Rndi8>, [https://bhaxor.blog.hu/2018/11/29/eddig\\_csaltunk\\_de\\_innentol\\_mi](https://bhaxor.blog.hu/2018/11/29/eddig_csaltunk_de_innentol_mi), <https://bhaxor.blog.hu/2018/10/28/minecraft>

Megoldás videó:

Megoldás forrása:

A játékban most a főszereplő Steve lesz (bár erről már a cím is árulkodhat). Az példákkal pedig be lehet mutatni, hogy különböző módosításokkal hogyan irányítható automatikusan is. Mivel a Minecraft világa lényegében kockákból épül fel, ezért amikor az irányítást próbálják megtervezni, akkor is kockákban kell gondolkodni, hogy Stevet melyik kockára léptetik tovább. Ezek alapján tehát 26 kockát kell egyszerre figyelembe venni. A forrást megtalálhatjuk a Microsoft githubos oldalán a malmo projektben. Letöltés után még nem tudjuk rögtön indítani, előbb telepíteni kell a sudo apt-get install libboost-all-dev libpython3.5 openjdk-8-jdk ffmpeg python-tk python-imaging-tk parancsal. Ezután pedig még 2 parancs végrehajtása szükséges volt nálam, anélkül szintén nem indult: export JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64/ és egy export MALMO\_XSD\_PATH=~/MalmoPlatform/Schemas. Ezután pedig már tényleg futtathatjuk a klienst, és el is indul a játék. Itt még csak az alap játékkal találkozunk, ahol mi építünk világot, mi irányítunk, stb... Viszont ha megnézzük az examples mappát, akkor onnan aki tesztelni szeretne kedve szerint tud próbálgatni a korábban említett minták közül. Ekkor a más futó alap játék mellé nyissunk még egy terminált, és ott kezdjük el futtatni valamelyik példát. Ez magától be fog tölteni (néhányhoz még plusz telepítés szükséges, de a legtöbkhöz nem kell). És a kép a feladatról, vagyis a példa ágens futtatásáról:



## **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.

A könyvben a források a feladatok előtt meg vannak jelölve, ezek a képekre is vonatkoznak, amelyek saját készítésűek, az UDPORG csoportból vagy Bátfai Norbert előadás fóliáiról származnak.

Külön köszönet az alapért és a forrásokért Bátfai Norbert Tanár Úrnak! Illetve a Tutoroknak is, akik segítettek a feladatok feldolgozásában: Tóth Attila (feladatok: 2.4; 2.6; 6.4), Egyed Anna (feladatok: 3.2; 5.1; 6.3), Ignéczi Tibor (feladatok: 4.2; 4.4), Duszka Ákos Attila (feladatok: 4.5; 4.6), Takács Viktor (feladatok: 4.5; 8.2)

Tutoráltam volt: Egyed Anna (feladatok: 3.1; 5.2; 6.2; 6.6), Ignéczi Tibor (feladatok: 3.5; 6.6; 7.1), Takács Viktor (6.2), Ranyhóczki Mariann