

Univerzális programozás

:)

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert ÁCs Szuhai, Dávid	2019. október 4.	

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	2019-05-08	44 feladatból 38 megoldva +2 human mnist lvl10; (4-es szint)	sz_d

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?	7
2.3. Változók értékének felcserélése	8
2.4. Labdapattogás	10
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	12
2.6. Helló, Google!	13
2.7. 100 éves a Brun téTEL	15
2.8. A Monty Hall probléma	16
3. Helló, Chomsky!	19
3.1. Decimálisból unárisba átváltó Turing gép	19
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	20
3.3. Hivatalos nyelv	22
3.4. A források olvasása	23
3.5. Logikus	24
3.6. Deklaráció	25

4. Helló, Caesar!	28
4.1. double ** háromszögmátrix	28
4.2. C EXOR titkosító	30
4.3. Java EXOR titkosító	31
4.4. C EXOR törő	33
4.5. Neurális OR, AND és EXOR kapu	36
4.6. Hiba-visszaterjesztéses perceptron	37
5. Helló, Mandelbrot!	40
5.1. A Mandelbrot halmaz	40
5.2. A Mandelbrot halmaz a std::complex osztállyal	42
5.3. Biomorfok	45
5.4. A Mandelbrot halmaz CUDA megvalósítása	48
5.5. Mandelbrot nagyító és utazó C++ nyelven	49
5.6. Mandelbrot nagyító és utazó Java nyelven	52
6. Helló, Welch!	54
6.1. Első osztályom	54
6.2. LZW	57
6.3. Fabejárás	63
6.4. Tag a gyökér	65
6.5. Mutató a gyökér	73
6.6. Mozgató szemantika	75
7. Helló, Conway!	78
7.1. Hangyszimulációk	78
7.2. Java életjáték	79
7.3. Qt C++ életjáték	81
7.4. BrainB Benchmark	82
8. Helló, Chaitin!	84
8.1. Gimp Scheme Script-fu: króm effekt	84
8.2. Gimp Scheme Script-fu: név mandala	85
9. Helló, Gutenberg!	88
9.1. Programozási alapfogalmak	88
9.2. Programozás bevezetés	89
9.3. Programozás	90

III. Második felvonás	91
10. Helló, Berners-Lee!	93
10.1. C++ vs Java	93
10.2. Bevezetés a mobilprogramozásba	94
11. Helló, Arroway!	96
11.1. OO szemlélet	96
11.2. Gagyi	99
11.3. Yoda	100
12. Helló, Liskov!	101
12.1. Liskov helyettesítés sértése	101
12.2. Szülő-gyerek	104
12.3. Anti OO	106
12.4. Hello, Android!	107
12.5. Ciklomatikus komplexitás	109
13. Helló, Mandelbrot!	110
13.1. Reverse engineering UML osztálydiagram	110
13.2. Forward engineering UML osztálydiagram	110
13.3. Egy esettan	110
13.4. BPMN	110
13.5. TeX UML	111
14. Helló, Chomsky!	112
14.1. Encoding	112
14.2. OOCWC lexer	112
14.3. Full screen	112
14.4. Paszigráfia Rapszódia OpenGL full screen vizualizáció	112
14.5. Paszigráfia Rapszódia LuaLaTeX vizualizáció	113
14.6. Perceptron osztály	113
15. Helló, Stroustrup!	114
15.1. JDK osztályok	114
15.2. Másoló-mozgató szemantika	114
15.3. Hibásan implementált RSA törés	114
15.4. Változó argumentumszámú ctor	114
15.5. Összefoglaló	115

16. Helló, Arroway!	116
16.1. FUTURE tevékenység editor	116
16.2. OOCWC Boost ASIO hálózatkezelése	116
16.3. SamuCam	116
16.4. BrainB	116
16.5. OSM térképre rajzolása	117
17. Helló, Schwarzenegger!	118
17.1. Port scan	118
17.2. AOP	118
17.3. Android Játék	118
17.4. Junit teszt	118
18. Helló, Calvin!	119
18.1. MNIST	119
18.2. Deep MNIST	119
18.3. CIFAR-10	119
18.4. Android telefonra a TF objektum detektálója	119
18.5. SMNIST for Machines	120
18.6. Minecraft MALMO-s példa	120
IV. Irodalomjegyzék	121
18.7. Általános	122
18.8. C	122
18.9. C++	122
18.10Lisp	122

Táblázatok jegyzéke

12.1.	107
-------	-----

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

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

Egy szálon futó végtelen ciklus:

```
int main() {
    for(;;) {

    }
}
```

Ezzel egy végtelen ciklust hhozunk létre, úgy, hogy programunknak egy egyszerű main függvénye van, amiben egy for ciklus helyezkedik és semmi más.

A for ciklus feltételeként kettő pontosvesszőt adunk meg, ezzel azt a feltételelt tesszük, hogy "semmi", mivel a vesszők közé nem adunk meg semmit. Ezzel azt hozzuk létre, hogy a függvény "addig megy, amíg semmi", mivel a semmi mindig semmi ezért a függvény nem áll le, míg manuálisan le nem állítjuk. Ez a program csak egy magot "foglal" el.

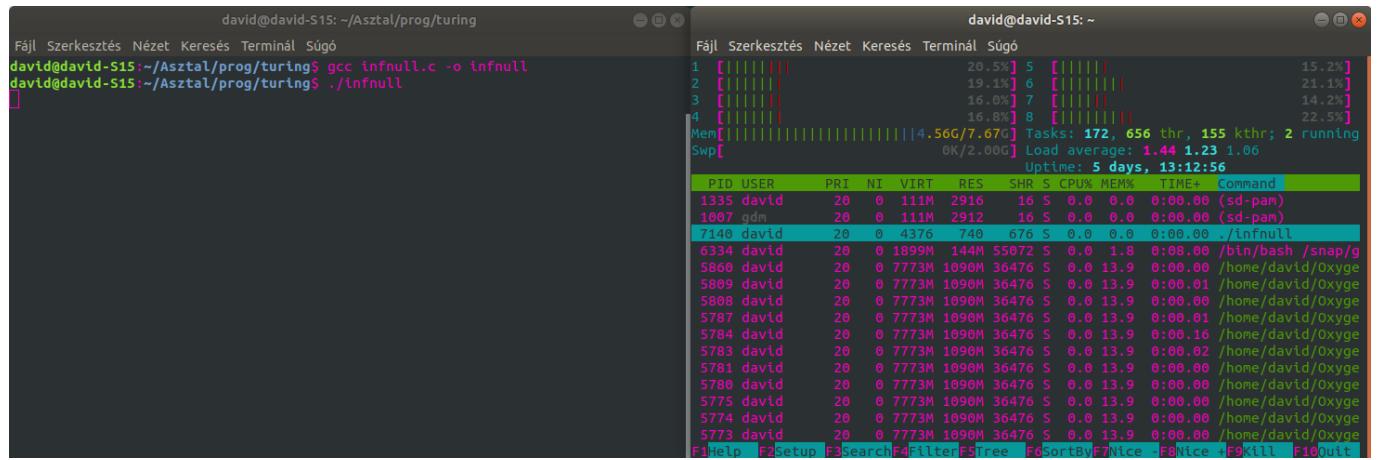
```
david@david-S15: ~/Asztal/prog/turing
Fájl Szerkesztés Nézet Keresés Terminál Súgó
david@david-S15:~$ cd Asztal/prog/turing/
david@david-S15:~/Asztal/prog/turing$ ./inf
david@david-S15:~$ top
Tasks: 169, 617 thr, 155 kthr; 2 running
Mem[ 1.3% 100.0% ] 13.99G/7.67G Tasks: 169, 617 thr, 155 kthr; 2 running
Swp[ 0K/2.00G ] Load average: 0.76 0.79 0.88
Uptime: 5 days, 13:03:07
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
6231 david 20 0 4186M 572M 724 R 99.3 0.0 0:23.07 ./inf
1513 david 20 0 1181M 283M 97M S 5.9 7.3 49:24.16 /usr/bin/gnome-shell
15106 david 20 0 1181M 283M 97M S 1.3 3.6 5:57.85 /opt/google/chrome
1354 root 20 0 654M 173M 79340 S 1.3 2.2 29:07.63 /usr/lib/xorg/Xorg
6187 david 20 0 785M 39720 29140 S 1.3 0.5 0:00.72 /usr/lib/gnome-terminal
1535 david 9 -11 2436M 14244 10524 S 0.7 0.2 15:03.89 /usr/bin/pulseaudio
4176 david -6 0 2436M 14244 10524 S 0.7 0.2 1:36.93 /usr/bin/pulseaudio
6222 david 20 0 42688 5360 3848 R 0.7 0.1 0:00.77 htop
15107 david 20 0 1181M 283M 97M S 0.7 3.6 1:14.25 /opt/google/chrome
13869 david 20 0 927M 177M 77584 S 0.7 2.3 9:36.12 /opt/google/chrome
1994 david 20 0 1849M 244M 121M S 0.7 3.1 30:30.02 /opt/google/chrome
6090 david 20 0 1849M 244M 121M S 0.7 3.1 0:07.19 /opt/google/chrome
1554 david 20 0 369M 11348 8428 S 0.7 0.1 1:06.85 ibus-daemon --xim
29094 david 20 0 1251M 101M 74864 S 0.7 1.3 0:08.44 gnome-control-center
1496 david 20 0 215M 6952 6240 S 0.7 0.1 0:08.79 /usr/lib/at-spi-2.0-aspiservice
F1=help F2=Setup F3=Search F4=Filter F5=Tree F6=SortByF7=Nice F8=Nice F9=Kill F10=Quit
```

0 százalékon futó végtelen ciklus:

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

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

Azonos a működésének elmélete, mit az egy szálat 100%ban használóval, viszont itt a "sleep" használatával a ciklus fut, de nem terheli a cput.



Mindent 100%-on futtató program:

A "#pragma omp parallel" használatával lehetővé tesszük, hogy az adott program egyszerre több magon is futtatható legyen. A mellékelt képen látszik is, hogy minden 100%-on megy és a program minden szálban fut, külön-külön pid-ekkel.

```
#include <omp.h>

int main(){
    #pragma omp parallel
    {
        for(;;);
    }
    return 0;
}
```

The screenshot shows the htop terminal application running on a Linux system. The top part displays system statistics: CPU usage (100.0% for all cores), memory usage (4.92G/7.67G), tasks (177 total, 692 threads, 157 kernel threads, 8 running processes), load average (2.26, 1.40, 1.32), and uptime (5 days, 13:45:25). Below this is a detailed process list with columns for PID, USER, PRI, NI, VIRT, RES, SHR, S, CPU%, MEM%, TIME+, and Command. The list includes several instances of the 'inffull' command running under the 'david' user. At the bottom of the screen, there is a menu bar with F1 through F10 keys corresponding to various functions like Help, Setup, Search, Filter, Tree, SortBy, Nice, and Kill.

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1007	gdm	20	0	111M	2912	16	S	0.0	0.0	0:00.00	(sd-pam)
1335	david	20	0	111M	2916	16	S	0.0	0.0	0:00.00	(sd-pam)
8121	david	20	0	68360	880	784	R	96.8	0.0	0:07.15	./inffull
8122	david	20	0	68360	880	784	R	87.7	0.0	0:06.86	./inffull
8123	david	20	0	68360	880	784	R	90.3	0.0	0:06.90	./inffull
8124	david	20	0	68360	880	784	R	99.4	0.0	0:07.50	./inffull
8125	david	20	0	68360	880	784	R	94.2	0.0	0:06.92	./inffull
8126	david	20	0	68360	880	784	R	93.5	0.0	0:07.15	./inffull
8127	david	20	0	68360	880	784	R	88.3	0.0	0:06.94	./inffull
8128	david	20	0	68360	880	784	R	749.	0.0	0:57.03	./inffull
5737	david	20	0	7775M	1172M	36476	S	0.0	14.9	0:06.35	/home/david/Oxyge
5738	david	20	0	7775M	1172M	36476	S	0.0	14.9	0:00.44	/home/david/Oxyge
5739	david	20	0	7775M	1172M	36476	S	0.0	14.9	0:00.43	/home/david/Oxyge
5740	david	20	0	7775M	1172M	36476	S	0.0	14.9	0:00.39	/home/david/Oxyge
5741	david	20	0	7775M	1172M	36476	S	0.0	14.9	0:00.40	/home/david/Oxyge

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!

Egy program, ami eldönti más programokról, hogy azok le fognak fagyni vagy nem, másképp a Turing megállási probléma. Alan Turing, az Enigma feltörője, az első "számítógép" készítője, a névadója a problémának. Ugyanis először ő mondta meg, hogy efféle program nem létezik.

Turing gondolatmenete a problémával kapcsolatban:

Tegyük fel, hogy van egy programunk, ez be fog fogadni "imputokat" és ad nekünk "outputokat". Ez a program megválaszolja a kérdést, hogy a feltételezések magukkal vonják-e a következtetést. Felteszünk egy kérdést, amit "ő" megválaszol, igen vagy nem. A kérdés az, hogy egy program, aminek egy bizonyos bemenettel adunk, ad-e választ, vagy a végtelenségig fog futni. Tehát megáll vagy sem. Továbbá egy program valami bemenettel fog-e valaha lefagyni?

Turing feladata az volt, hogy bebizonyítsa, hogy lehetetlen olyan programot létrehozni, ami megoldja a "megállási problémát". Vajon egy adott gép, egy adott bemenettel megáll vagy sem?

Turing válasza a kérdésre:

Tegyük fel, hogy meg van a gépünk, ami eldönti, hogy egy adott program leáll-e vagy sem, tehát a működése elvétől függetlenül, az adott gép választ ad, igen leáll, vagy nem nem áll le.

Ha megvan ez a gép át tudjuk alakítani egy másik géppé, úgy hogy ha a válasz "igen", akkor egy végtelen ciklusba kerül, ha a válasz "nem" azonnal leáll.

Ekkor jön az újabb kérdés, mi van ha ezt a bővített programot "imputként" beadjuk önmagának? Ebben az esetben, ha a válasz "igen", azaz megáll, akkor egy végtelen ciklusba kerül, azaz mégsem áll le. De ha nem áll le, a válasz "nem", aztán leáll.

Tehát az adott program, ha leáll, akkor nem áll le, valamint ha nem áll le, akkor leáll. Így egy ellentmondásba kerülünk.

Tehát feltételeztük, hogy létezik megoldás a problémába, de végül egy ellentmondásba ütközünk. Azaz nem lehetséges olyan gépet/programot létrehozni, ami megoldja a megállási problémát, vagyis eldönti egy adott gépről/programról, hogy az le áll-e vagy sem.



(Kép forrása: <https://cs.stackexchange.com/questions/65401/proof-of-the-undecidability-of-the-halting-problem>
2019.04. 17. 18:55)

"Turing & The Halting Problem - Computerphile" alapján

(https://www.youtube.com/watch?v=macM_MtS_w4 2019. 03. 05. 22:35)

2.3. Változók értékének felcserélése

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

Két váltató cseréjét számos módon lehet kivitelezni, jelen esetben a feladatnak megfelően, 3 mód létezik. Összeadással, szorzással és kizárával vagy való csere.

(A programok fix értékekkel dolgoznak a könnyeb és gyorsabb futás érdekében.)

Az első program az összeadás/kivonás módszert használja.

```
#include <stdio.h>

int main(){
    int a=3, b=1;
    printf("%d ",a); //kiirjuk az a-t
    printf("%d\n",b); //kiirjuk a b-t
    a= (a+b); //itt az a=4, b=1
    b= (a-b); //itt az a=4, b=3
    a= (a-b); //itt az a=1, b=3
    printf("%d ",a); //kiirjuk az a-t
    printf("%d\n",b); //kiirjuk a b-t

}

/* három egyszerű művelettel kicseréljük két szám értékét segédváltozók ←
nélkül*/
```

A következő a szorzás illetve osztás műveleteket használja, az előzőhöz képest csak az írásjelekben van különbség.

```
#include <stdio.h>

int main(){
    int a=3, b=1;
    printf("%d ",a); //kiirjuk az a-t
    printf("%d\n",b); //kiirjuk a b-t
    a= (a*b); //itt az a=3, b=1
    b= (a/b); //itt az a=3, b=3
    a= (a/b); //itt az a=1, b=3
    printf("%d ",a); //kiirjuk az a-t
    printf("%d\n",b); //kiirjuk a b-t

}
```

A harmadik pedig a kizáró vagy (azaz exor).

```
#include <stdio.h>

int main(){
    int a=3, b=1;
    printf("%d ",a); //kiirjuk az a-t
    printf("%d\n",b); //kiirjuk a b-t
    a= (a^b);
    b= (a^b);
    a= (a^b);
    printf("%d ",a); //kiirjuk az a-t
    printf("%d\n",b); //kiirjuk a b-t
```

```
}
```

Az exor bitenkét összehasonlítja a két adott számot, jelen esetben inteket használva 32 bitet (azaz 32 db 0 vagy 1). Az eredményben adott bit értéke 1 ha az összehasonlítottak közöl az egyik 0 a másik meg 1, különben az érték 0.

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés használata nélkül írj egy olyan programot, ami egy labdát pattogtat a karakteres konzolon!

A labda (esetünkben egy 'x'[lenti képen látható]) pattog a képen, azaz mozog a kijöllt ablakban. Alább a két megoldás if használatával és anélkül.

Labdapattogás if használatával:

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

int main(){
    WINDOW *ablak;
    ablak=initscr();
    int x=0,y=0; //oszlop és sor
    int px=2,py=4; //"pattogás gyakorisága"
    int mx,my; //oszlopok, sorok száma
    for(;;){ //végtelen ciklus a for ban
        getmaxyx(ablak, my, mx);
        mvprintw(y, x, "x");
        refresh();
        usleep(75000);
        clear();
        x=x+px;
        y=y+py;
        if(x<=0)
            px=px*-1;
        if(x>=mx-1)
            px=px*-1;
        if(y<=0)
            py=py*-1;
        if(y>=my-1)
            py=py+-1;

    }
}
```

Az elején létrehozzuk az ablakot valamint megadunk néhány értéket (oszlopok, sorok záma, pattogás gyakorisága), majd indítunk egy végtelen ciklust. A végtelen cikluson belül iratjuk ki a "x" karaktert valamimnt

a pattogás gyorsaságát adjuk meg és a törtélesek is itt végezzük (hogy egyszerre csak egy abda legyen a képen). Az ifekben nézzük meg, hogy hol jár a labda és ha elérte valamelyik szélet ahoz viszonyítva indítjuk másik irányba.

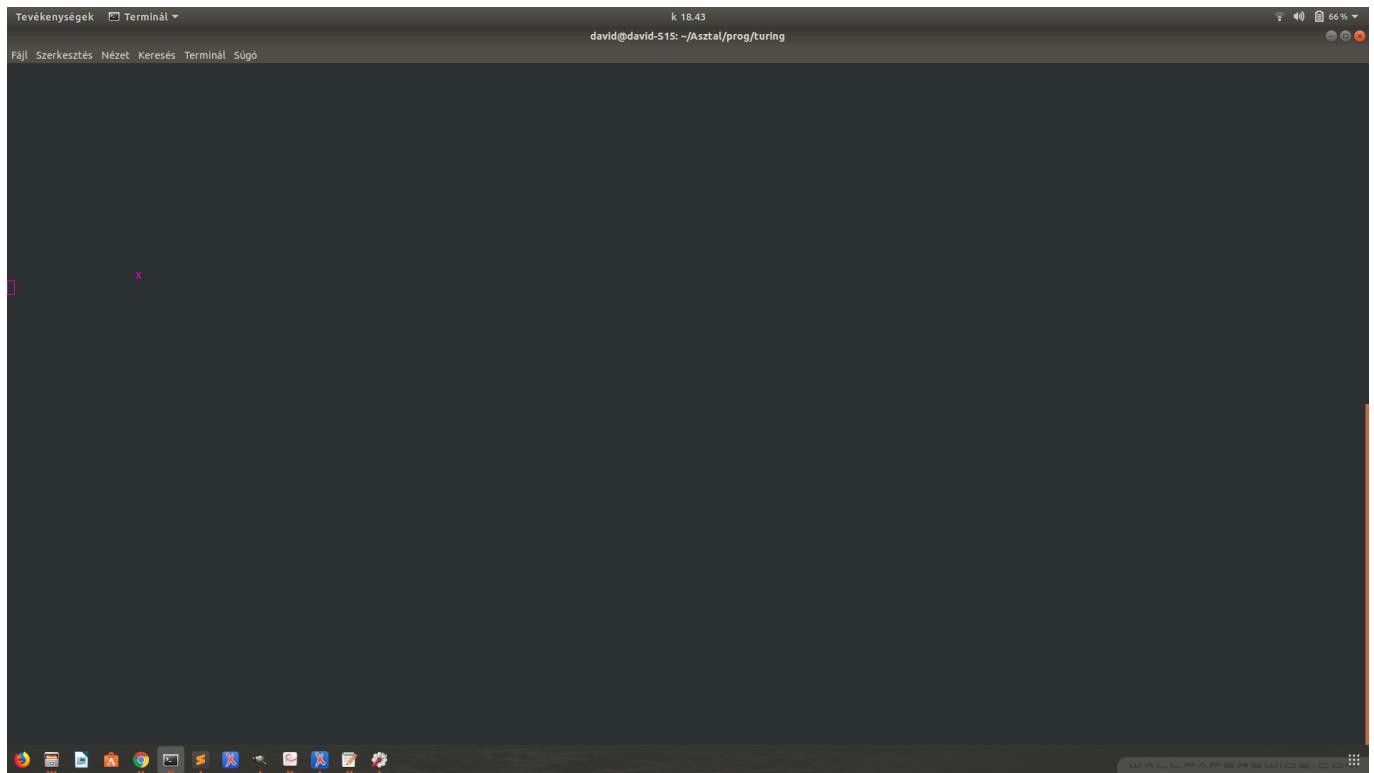
Labdapattogás if nélkül:

```
#include<stdio.h>
#include<math.h>
#define SZEL 78    //meghatározzuk az ablak méreteit
#define MAG 22

int putX(x,y)
{
    int i;
    for(i=0;i<x;i++) //megindítjuk az i-t novekedni, amíg kisebb mint az x
        printf("\n"); //kiiratunk sortöréseket
    for(i=0;i<y;i++) //itt addig növeljük, míg az y-nél kisebb
        printf(" "); //kiir "spaceket"
    printf("X\n"); //kiir egy X-et (ez jelképezi a labdát) és egy sortörést
return 0;
}

int main()
{
    long int x=0,y=0; //létrehozzuk az xet és yont, értékeinek 0-át adunk
    while(1) //végzetlen ciklus a while-ban
    {
        system("clear"); //letöröljük a képernyőt
        putX(abs(MAG-(x++%(MAG*2))),abs(SZEL-(y++%(SZEL*2)))); //meghívjuk a "←
            putX"-et
        usleep(50000); //késlelteti a folyamat lefutását
    }
return 0;
}
```

Sortörésekkel és spacekkal tesszük lehetővé a labda mozgását. A mainben szintén van egy végzetlen ciklus, abban meghívjuk a kiirató fügvényt valamint egy sleep is van itt, amivel a labda gyorsaságát tudjuk szabályozni.



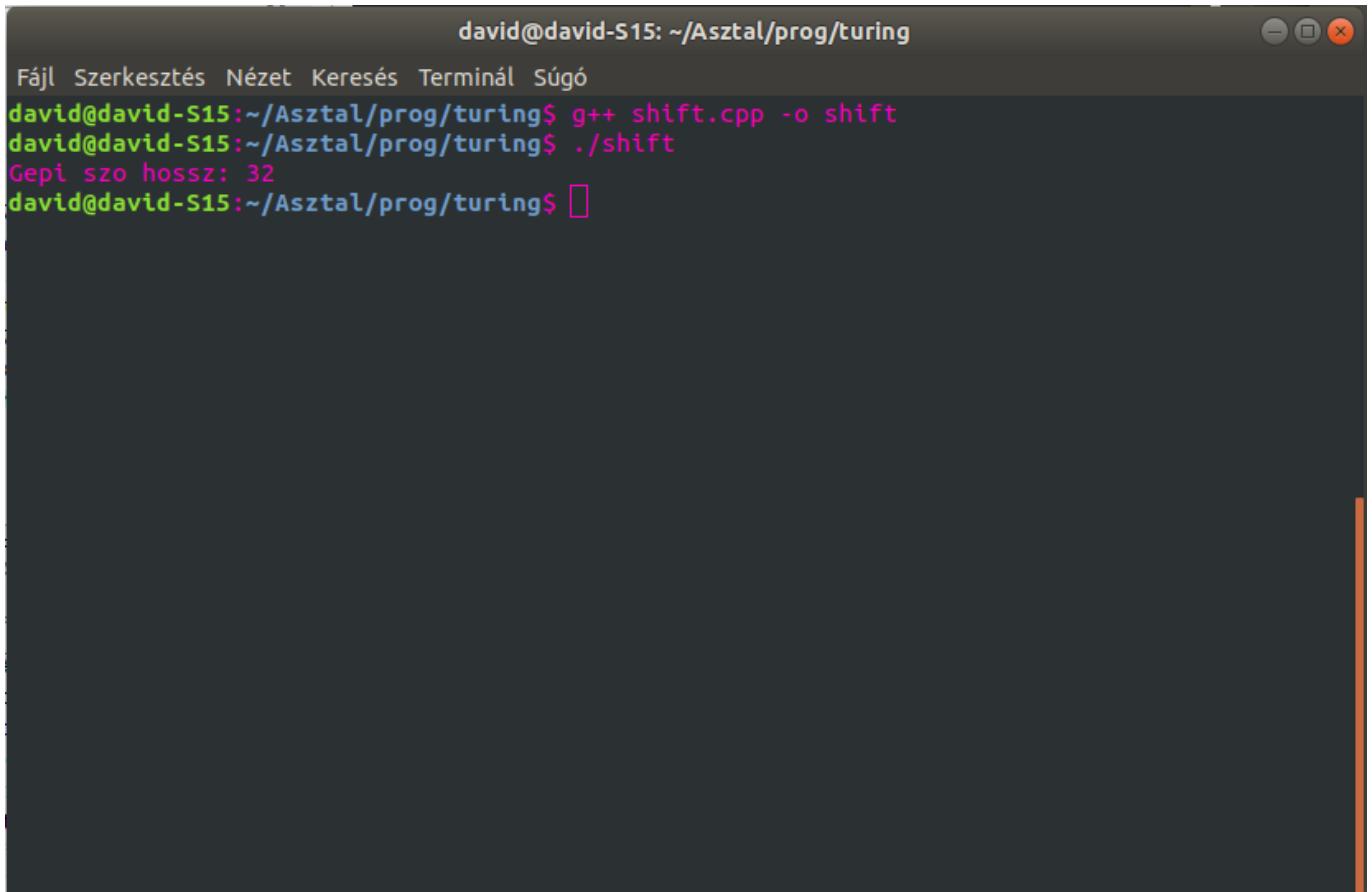
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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!

```
#include <iostream>

int main(){
    int s=1;
    int i=1;
    while (s >=0) {
        s<<=1;
        i++;
    }
    std::cout <<"Gepi szo hossz: "<<i<<"\n";
}
```

Addig fut, amíg az s értéke nagyobb/egyenlő, mint 0 a while ciklus. Azon belül az első sor balra shiftel egyel, vagyis a "szó" végére egy 0-át rak, amíg a "szó" csak 0-ból áll, azaz a bináris értéke =0. A második sor egyel növeli az i értékét, ezzel számolja, hogy hányszor shiftel a program, vagyis hány 0-át rak a szó végére. És a végén kiirjuk az i értékét vagyis, hogy hány shift volt.



```
david@david-S15: ~/Asztal/prog/turing
Fájl Szerkesztés Nézet Keresés Terminál Súgó
david@david-S15:~/Asztal/prog/turing$ g++ shift.cpp -o shift
david@david-S15:~/Asztal/prog/turing$ ./shift
Gepi szó hossz: 32
david@david-S15:~/Asztal/prog/turing$
```

A bogomips egyfajta eszköz, mértékegység, ami meghatározza a processzor sebességét, de mivel ez nem hovatalos, azért használjuk előtte a "bogo" szócskát, mi 'bogus' szóból származik, minek jeletése hamis.

```
#include <iostream>

int main(){
    int s=1;
    int i=0;
    while ((s<<=1)) {
        s<<=1;
        i++;
    }
    std::cout <<"Gepi szó hossz: "<<i<<"\n"; //kiírja, az i értékét, azaz, ←
        hogy hány shift volt
}
```

2.6. Helló, Google!

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

A PageRank lényege, mit a neve is mondja osztájozza az oldalakat "hasznosságuk" alapján pontosan, hogy egy oldalra hány másik mutat valamint az adott oldal hány másikra mutat. Ezen az alapon nyugszik a Google keresőmotora és mondhatni ennek köszönheti sikereit. Ez a program 4 adott oldalnak határozza meg a page rank értékéét

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

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

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

Ebben a függvényben kiírjuk a kapott értékeket, deklaráció szerint a függvény egy többe írja az értékeket, valamint kéri az oladalk számát. Majd alatta a távolság függvény visszadja az összeg négyzetét.

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

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

    return sqrt(osszeg);
}

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

    int i, j;

    for (;;)
    {
        for (i = 0; i < 4; ++i)
        {
            PR[i] = 0.0; //nullázuk a PR tömbnek az i-edik elemét
            for (j = 0; j < 4; ++j)
                PR[i] += (L[i][j] * PRv[j]);
        }
        if (tavolsag (PR, PRv, 4) < 0.00000001)
            break;
    }
}
```

```
    for (i = 0; i < 4; ++i)
PRv[i] = PR[i]; //másolást hajtunk végre a két tömbben

}

kiir (PR, 4);

return 0;
}
```

A mainban először megadjuk a mátrixot, ami a kapcsolatok számát tartalmazza. A PR tömb szolgál majd a pagerank értékek tárolására, ez alapból nullákkal van feltöltve. Indítunk egy végtelen ciklust, ebben végigmegyünk a 4 elemen mindegyiknek kiszámoljuk az értékét a mgefelelő elemek összeszorzásával. A végtelen ciklus akkor áll le, ha a távolság fügvény értéke 0.00000001-nál kisebb értéket ad vissza. A végé pedig meghívjuk a kiir fügvényt a PR tömb 4 elemére.

```
david@david-S15:~/Asztal/prog/turing$ gcc pagerank.c -o pr -lm
david@david-S15:~/Asztal/prog/turing$ ./pr
0.090909
0.545455
0.272727
0.090909
david@david-S15:~/Asztal/prog/turing$ 
```

A compile-nál használnunk kell a -lm kapcsolót a math library miatt. Miután lefutott a program kitűnően látszik a 4 oldal pagerank értéke.

2.7. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

Brun téTEL szerint az ikerprímek (olyan prímpár, melynek különbsége=2) reciprokának összege egy véges értékhez terül, ezt nevezzük "Brun-konstans"-nak. A téTEL bizonyítása (1919) Viggo Brun-hoz kötődik, aki a téTEL névadója is lett.

```
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")
```

A programhoz használjuk a matlab libraryt. A harmadik sor meghatározza a prímeket "x"-ig, a következőben a kivonás a-b, a:a meghatározott primek közül az első kivételével mind, b: az utolsó kivételével mind. Aztán azokat, ahol a kivonás eredménye=2 azokat eltárolja az idx vektorban, a köviben az össze prímek közül kivesszük azonat, ahol a kivonás eredménye 2 volt ez lesz az ikerpár első tagja. Ezt követően azonos művelet, csak itt hozzáadunk 2öt a helyekhez, így a párok második tagjait kapjuk meg. A fügvény utolsó két sorában előbb vesszük az ikerprímek reciprokait és összeadjuk azokat, majd a korábban kiszámolt összegeket összeadjuk. AZ utolsó három sorban miután kiszámmoltuk az értékeket, hogy hova tartanak az ikerprímek egyszerűen kiiratjuk őket.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Monty Hall probléma/paradoxon: Egy amerikai TV-s vetélkedőből ered. Röviden, három ajtó közül egy möggött van nyeremény kettő mögött pedig nem. Kiválasztunk egyet, majd egy a kiválasztottól különböző ajtót, mi möggött nincs nyeremény kinyitnak. Ekkor adódik lehetőségünk maradni a korábban választott ajtónál, vagy megváltoztathatjuk döntésünket. A nyerésnek akkor van esélye, ha változtatunk, vagyis minig megéri változtatni.

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

Meghatározzuk, hogy az adott kísérletben, hol van a díj, majd a játékos választása és a műsorvezető választása a for ciklusban.

```
for (i in 1:kiserletek_szama) {

  if(kiserlet[i]==jatekos[i]){
    mibol=setdiff(c(1,2,3), kiserlet[i] )

  }else{
    mibol=setdiff(c(1,2,3), c(kiserlet[i], jatekos[i])) }

  musorvezeto[i] = mibol[sample(1:length(mibol),1) ]

}
```

A for ciklus minden kísérleten "végig megy". Az ifben ha a nyeremény helye és a játékos választása egyezik, ezok közül választ, ami nem a játékos választása. Else ágon, a három közül csak azt választja,

ami nem a játékos választása és nem az ajándék helye. Végül a korábbi események fügvényében választ a műsorvezető.

```
nemvaltoztatesnyer= which(kiserlet==jatekos)
valtoztat=vector(length = kiserletek_szama)
```

Az első sor az a lehetőség ha nem változtat az ember és nyer, a második pedig, mi történik, ha változtat...

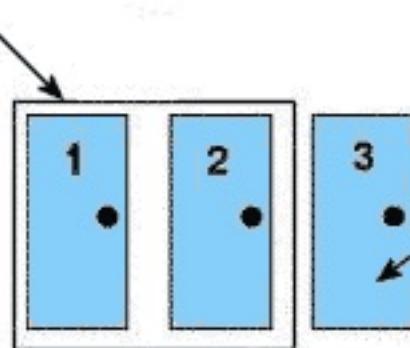
```
for (i in 1:kiserletek_szama) {
  holvolt = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i]))
  valtoztat[i] = holvolt[sample(1:length(holvolt),1)]
}
```

A forban az az ajtó marad csak, ami nem a korábban választott és nem a már kinyitott. (lent)Eztán jön az az eset, ahol a jóra vált az ember. Eztán a kiiratás: hány kísérelt volt, hányszor nyert váltás néélkül, hányszor azzal. Legvégül pedig a két lehetőség hányadosa és összege (=kísérletek száma).

```
valtoztatesnyer = which(kiserlet==valtoztat)

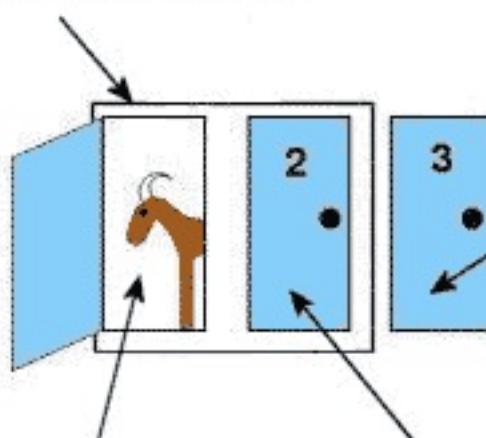
sprintf("Kiserletek szama: %i", kiserletek_szama)
length(nemvaltoztatesnyer)
length(valtoztatesnyer)
length(nemvaltoztatesnyer)/length(valtoztatesnyer)
length(nemvaltoztatesnyer)+length(valtoztatesnyer)
```

2/3 eséllyel itt a kocsi



1/3 eséllyel itt

2/3 eséllyel itt a kocsi



1/3 eséllyel itt

0 eséllyel itt, tehát 2/3 eséllyel itt

(Kép forrása: <https://www.karpatinfo.net/cikk/tudomany/2562270-hallott-e-mar-arr>
2019. 04. 17. 18:50)

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

A decimális számrendszer az "általánosan használt" rendszer, azaz a tízes rendszer.

Az unáris pedig az egyes számrendszer, ezt a program vonalakkal fogja jelezni, minden vonal értéke egy. Ha összeszámoljuk a vonalakat kijön a decimálisan megadott szám.

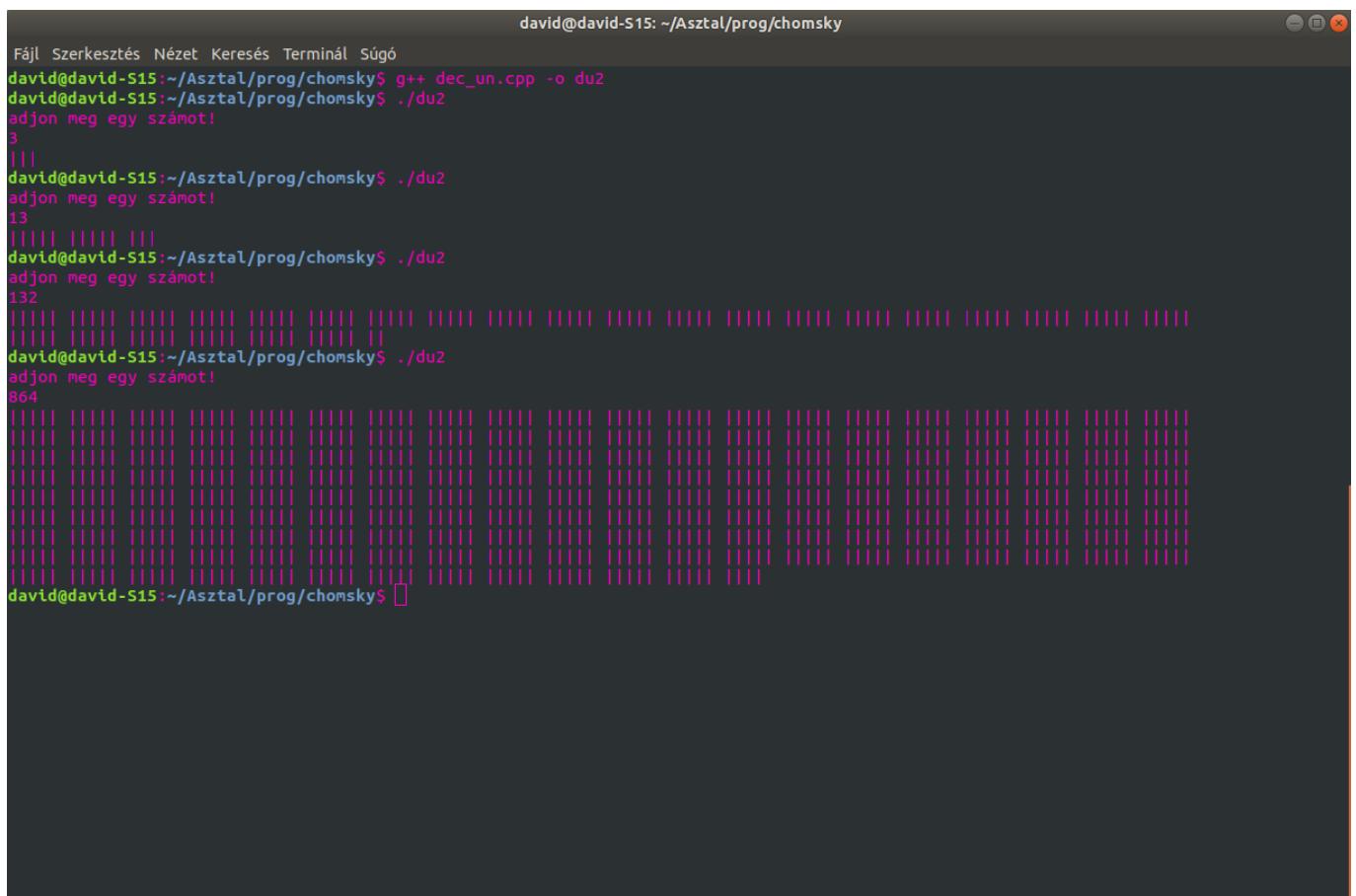
```
#include <iostream>
using namespace std;

int main() {
    int a, s=0, h=0;

    cout<<"adjon meg egy számot!\n";
    cin>>a;      t
    for(int i=0; i<a; i++) {
        cout<<"| ";
        ++s;
        ++h;
        if(s==5) {
            cout<<" ";
            s=0;
        }
        if(h==100) {
            cout<<"\n";
            h=0;
        }
    }
    cout<<"\n";
}
```

Egy egyszerű main ciklusból áll a program, az elején létrehozunk három int változót, ezeket fogjuk használni a programban. Bekérünk egy számot, amit át fogunk alakítani unárisba. A for ciklus addi megy, amíg

el nem éri a korábban megadott számot. Egyessével kiiratunk "!" karaktereket. A for ciklus számolja, hogy hány vonalat írtunk ki, ha ez a szám eléri a ötöt, kiirunk egy space-t, ha pedig a százat egy sortörést. Ez azért van, hogy könnyebben lehessen olvasni a kimenetet.



```
david@david-S15:~/Asztal/prog/chomsky$ g++ dec_un.cpp -o du2
david@david-S15:~/Asztal/prog/chomsky$ ./du2
adjon meg egy számot!
3
!!!
david@david-S15:~/Asztal/prog/chomsky$ ./du2
adjon meg egy számot!
13
!!!!
david@david-S15:~/Asztal/prog/chomsky$ ./du2
adjon meg egy számot!
132
!!!!!!
david@david-S15:~/Asztal/prog/chomsky$ ./du2
adjon meg egy számot!
864
!!!!!!
```

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

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

Generatív nyelvtanok

Egy generatív nyelvtan adott szabályok végesek alkalmazásával létrehozott jelsorozatokból áll. Ha egy adott jelsorozatot töbféleképp is le lehet képezni, az esetben az adott nyelv kétértermű.

Olyan környezetfüggő generatív grammatika, a kért nyelvet generálja:

1,

S, X, Y „változók”

a, b, c „konstansok”

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

S-ből indulunk ki

S (S \rightarrow aXbc)

aXbc ($Xb \rightarrow bX$)

abXc ($Xc \rightarrow Ybcc$)

abYbcc ($bY \rightarrow Yb$)

aYbbcc ($aY \rightarrow aa$)

aabbcc

vagy

S ($S \rightarrow aXbc$)

aXbc ($Xb \rightarrow bX$)

abXc ($Xc \rightarrow Ybcc$)

abYbcc ($bY \rightarrow Yb$)

aYbbcc ($aY \rightarrow aaX$)

aaXbbcc ($Xb \rightarrow bX$)

aabXbcc ($Xb \rightarrow bX$)

aabbXcc ($Xc \rightarrow Ybcc$)

aabbYbcc ($bY \rightarrow Yb$)

aabYbbcc ($bY \rightarrow Yb$)

aaYbbbcc ($aY \rightarrow aa$)

aaabbccc

.

2,

A, B, C „változók”

a, b, c „konstansok”

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

S-ből indulunk ki

A, B, C „változók”

a, b, c „konstansok”

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

S-ből indulunk ki

vagy

A ($A \rightarrow aAB$)

aAB ($A \rightarrow aAB$)

aaABB ($A \rightarrow aAB$)

aaaABBB ($A \rightarrow aC$)

aaaaCBBB (CB → bCc)
aaaabCcBB (cB → Bc)
aaaabCBcB (cB → Bc)
aaaabCBBc (CB → bCc)
aaaabbCcBc (cB → Bc)
aaaabbCBcc (CB → bCc)
aaaabbbCccc (C → bc)
aaaabbbbcccc

3.3. Hivatkozási nyelv

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

A c89 és c 99 közti egyik különbözőség, hogy a c99-ben 'hozták be' a "://" komment jeletezt a c89 nem tudja értelmezni.

```
#include <stdio.h>

int main(){
    int a=1,b=3;
    int y;
    y = a+b;
    //y = y+a
    printf("%d", y);
}
```

Ha ezen kódot c89-ben compileoljuk hibakódöt kapunk.

```
david@david-S15:~/Asztal/prog/chomsky$ gcc c89_99.c -o c89 -std=c89
c89_99.c: In function ‘main’:
c89_99.c:7:2: error: C++ style comments are not allowed in ISO C90
  //y = y+a
  ^
c89_99.c:7:2: error: (this will be reported only once per input file)
```

Azonban, ha c99-et használunk márfordul

```
david@david-S15:~/Asztal/prog/chomsky$ gcc c89_99.c -o c99 -std=c99
david@david-S15:~/Asztal/prog/chomsky$
```

További funkciók, amik a c99-el jöttek be:

- komplex aritmetika támogatása
- a long long int típus
- változások a printf és scankf funkciókban

-új könyvtárak és headerek

-...

3.4. A források olvasása

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

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

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



Bugok

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

i.

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

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);
```

i. Ha nem volt figyelmen kívül hagyva a `ctrl+c` általi megszakítás, akkor beállítja a jelkezelésre a "jelkezelő" függvényt. Ha figyelmen kívül volt hagyva, akkor továbbra is figyelmen kívül lesz hagyva, hisz az if fejében meghívtuk a `signal` függvényt, és beállítottuk a `SIG_IGN`-t, ami nem csinál mászt, mint figyelmen kívül fogja hagyni a `ctrl+c` általi megszakítás lehetőségét.

ii. $i=0$ -tő növeljük i értékét, míg az 5-nél kisebb

- iii. ugynaugy növeljük az értékét, de az i mindenkor előtt növelés előtti értékét adja, azaz ez egyel többször fog lefutni, mit az 'ii.'
- iv. szintén i=0 tól megy, míg kisebb mint 5 és az adott tömb i-edik elemének az i++-t adja értékül azaz i mindenkor előtt értékét. Azaz első körben a tömb 1. eleme 0 lesz, majd a tömb 2. eleme 1 és így tovább.
- v. i 0-ról indul és minden körben növeljük egyel az értékét (már ezt a növelt értéket adja vissza), addik megy, míg az i kisebb mint 'n' és a d-re mutató pointert növeljük (postfix módon) ésrt megkapja az s pointer (postfix módon) növelt értékét.
- vi. A 'printf' kiiratás, a rész ""%d %d"" azt jelzi, hogy 2 integer formátumot fogunk kiiratni. Az f() egy fügvényt hív meg, aminek bemenetként először megadjuk az a-t, majd az a egyel növelt értékét. Másodszorra pedig az a eggyel növelt értékét, majd az a-t (ami már korábban meg volt növelte).
- vii. Ismét két számot iratunk ki, megint meghívunk egy függvényt, minek bemenetképt az a-t adjuk, továbbá kiiratjuk az a-t.
- viii. Ismét kiiratunk két számot, jelen esetben az f függvények egy a-ra mutató pointert adunk és kiiratjuk mellé az a-t.

3.5. Logikus

Turotiált: Oláh Sándor

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

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

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

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

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

= minden x-nél van nagyobb y és van olyan y, ami prím.

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

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

= minden x-nél van nagyobb y és van olyan y, ami prím és van olyan y, ami prím.

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

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

= minden x prím, ebből következik, hogy minden x-nél van nagyobb y.

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

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

= Van olyan y, aminél minden x nagyobb, ebből következik, hogy nem minden x prím.

3.6. Deklaráció

Oláh Sándor segítségével(tutor)

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);`

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

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

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

int (*(d) (int a)) (int, int) {
if (a==1)
return szor;
else if(a==0)
return sum;
else
return NULL;
}

int (*(*p)(int))(int,int);

int main(){
int a=1,b,c;//egészek
int* m=&a;//egészre mutató mutató
int &r =b;//egész referenciája
int tomb[5]={0};//egészek tömbje
int (&tr)[5]=tomb;//egészek tömbjének referenciája.
int* tombm[5]={0};//egészre mutató mutatók tömbje
int* valami();//egészre mutató mutatót visszaadó függvény.

cout<<"Szorozni szeretnél(1) vagy összeadni(0)?\n";
cin>>a;
cout<<"Melyik az a két szám?\n";
cin>>b>>c;

int* (*v)();//egészre mutató mutatót visszaadó függvényre mutató;
int (*(d)(int x))(int c, int d);//egészet visszaadó és két egészet kapó ↵
függvényre
```

```
//mutató mutatót visszaadó egészet kapó függvény.  
cout<<(* (d) (a)) (b, c)<<"\n";  
int ok;  
p=d;  
//függvényremutató egy egészet visszaadó két egészet kapó függvényre ↪  
//mutató  
//mutatót visszaadó egészet kapó függvényre.  
cout<<(* (*p) (a)) (b, c)<<"\n";  
}  
  
//Oláh Sándor segítségével
```

- i. int a;
=deklaráljuk az a integrert
- ii. int *b = &a;
=dc a b pointert, ami az a ra mutat
- iii. int &r = a;
=cd az r, ami az a referenciája
- iv. int c[5];
=dc a c 5 elemű integer alapú tömböt
- v. int (&r)[5] = c;
=a c 5 elemű tömbök referenciája
- vi. int *d[5];
=d 5 elemű tömb, ami egészre mutató mutatók tömbje
- vii. int *h();
=dc a h függvényt egészre mutató mutatót visszaadó függvény
- viii. int *(*l)();
=egészre mutatót visszaadó függvény mutató
- ix. int (*v (int c)) (int a, int b);
= egészet adó két egészet kapó függvény mutatót visszaadó egy egészet kapó függvény
- x. int (*(*z) (int)) (int, int);
=egészet adó két egészet kapó függvény mutatót visszaadó egy egészet kapó függvény mutató

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

A feladat értelmezése. A "double ** háromszögmátrix" alatt olyan mátrixot értünk, ami double értékekből áll. Magának a mátrixnak sok fajtája van, a mátrix alapvetően számok halmaza, mely megjelenhet háromszög vagy négyzet alakban. Esetünkben egy alsó háromszögmátrixot készítünk, ami azt jelenti, hogy a double értékek a felső sorról lefelé, számuk növekszik, azaz az első sorban egy, a másodikban kettő és így tovább.

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

int main(){
    int nr = 5;
    double **tm;

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

    if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
        return -1;

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

    for (int i = 0; i < nr; ++i)
        if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL)
            return -1;

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

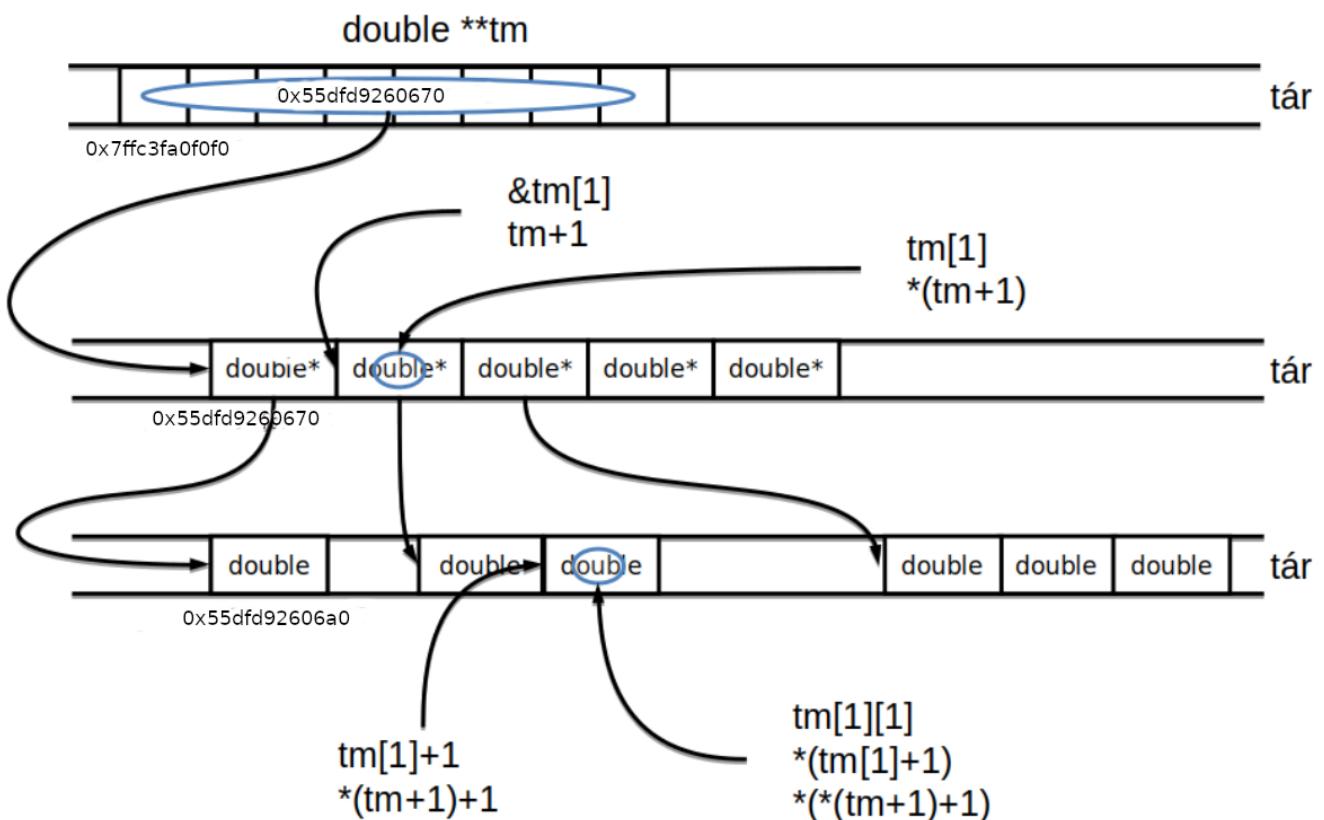
    for (int i = 0; i < nr; i++)
        for (int j = 0; j < i + 1; j++)
            tm[i][j] = i * (i + 1) / 2 + j;

    for (int i = 0; i < nr; i++)
        {for (int j = 0; j < i + 1; j++)
```

```

        printf ("%f, ", tm[i][j]);
        printf ("\n");
    }
}

```



Először megadjuk a sorok számát, majd egy double pointerre mutató pointert, aztán ennek a pointernek a memóriacímét kiiratjuk. Az első if-ben a malloc (mely fügvény egy akármire mutóta pointerrel tér vissza, viszont ezt double**-ra kényszerítjük) értékként $5*8=40$ -et kap, ennek az if-nek a feltétele teljesül, vagyis nem tudunk helyet lefoglalni, kilép. Ismét kiirunk egy címet, mégpedig azt, hogy a tm hová mutat, eztán memóriát foglalunk a tm tömbe i-edik elemei számára. Itt is használjuk a korábban említett malloc fv-t, ha pedig nem sikerül memóriát foglalni ismét kilép. Ha viszont nem lépünk ki, azaz sikeresen foglaltuk a tm[0] címét (vagyis az első elemének címét).

A végén a tömb elemeinek értéket adunk (az alábbi módon: (sor * ((sor+1)/2) + oszlop)). Eztán egyessével végighaladunk a tömbön és kiirjuk elemeit.

```
david@david-S15:~/Asztal/prog/caesar$ gcc 3szög.c -o 3
david@david-S15:~/Asztal/prog/caesar$ ./3
0x7ffc3fa0f0f0
0x55dfd9260670
0x55dfd92606a0
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,
david@david-S15:~/Asztal/prog/caesar$ █
```

Kiválóan látszik a kimenten az első sorokban a kódba említett memóriacímek, utánuk a szemléletes háromszögmátrix (a sor sorszámával meggyező elemszámmal, azaz első sor=egy elem, a második=kettő és így tovább).

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

(A gyorsabb futás érdekében a kulcs csak 3 jegyű, de ez 'bármeddig' növelhető.)

```
#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);
```

Definiálunk két konstansot, 100-as illetve 256-as értékekkel, továbbá deklaráljuk az elemeket az elején. Először 2 char alapú tömböt, amik értékeinek a a korábban definiált konstansokat adjuk. Eztán 2 egész

alapú változót, amik 0 értéket veszik fe. Majd a "kulcs_meret" felveszi a paracssorban megadott értékből számított számot.

```
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);
}

}
```

A while ban, haladunk bájtonként, míg van, a karaktereket összexorozzuk a klucs adott elemével és a végén a már titkosított szöveget kiírjuk.

```
david@david-S15:~/Asztal/prog/caesar$ gcc toro.c -o toro
david@david-S15:~/Asztal/prog/caesar$ ./toro <secret.txt
Kulcs: [374]
Tiszta szöveg: [A Boston Celtics csapatát 1946-ban alapították, az első tulajdonos Walter A. Brown volt. Először még csak a BAA-ban (Amerikai Kosárlabda Szövetség) játszottak, de 1949 őszén bekerültek az NBA-be a BAA és az NBA egyesülése miatt. 1950-ben a Celtics volt az első csapat, amely egy afro-amerikai játékost igazolt, Chuck Cooper személyében. A Kelták az első éveiben nem volt túl jó csapat, egészen Red Auerbach edzéséig leszerződtetéséig. Egy kiváló játékos is, Bob Cousy is csatlakozott a Bostonhoz, bár Auerbach edzés kezdetben nem akarta, hogy ilyen legyen. Az 1955-56-os szezon után Auerbach egy meglepő cserével módosította a játékoskeretet. Cliff Haganért és a válogatott Ed Macaulayért cserébe a San Francisco-i center, Bill Russell neve lett a csapatnávsor legújabb bejegyzése. Ha ide kell írni Ha-t, hogy jo legyen a ha.]
david@david-S15:~/Asztal/prog/caesar$ 
```

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Ezen program azonos működési elven alapszik, mint a c-s verzió, azonban ezúttal java nyelven.

```
public class ExorTitkosító {

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

    } catch(java.io.IOException e) {

        e.printStackTrace();

    }

}
```

```
david@david-S15:~/Asztal/prog/caesar$ java ExorTitkositó alma > titkosított.szöveg
A Boston Celtics csapatát 1946-ban alapították, az első tulajdonos Walter A. Brown volt. Először még csak
a BAA-ban (Amerikai Kosárlabda Szövetség) játszottak, de 1949 őszén bekerültek az NBA-be a BAA és az NBA
egyesülése miatt. 1950-ben a Celtics volt az első csapat, amely egy afro-amerikai játékost igazolt, Chuck Cooper személyében. A Kelták az első éveiben nem volt túl jó csapat, egészen Red Auerbach edző leszerző
dtetéséig. Egy kiváló játékos is, Bob Cousy is csatlakozott a Bostonhoz, bár Auerbach edző kezdetben nem
akarta, hogy így legyen. Az 1955-56-os szezon után Auerbach egy meglepő cserével módosította a játékosker
etet. Cliff Heaganért és a válogatott Ed Macaulayért cserébe a San Francisco-i center, Bill Russel neve l
ett a csapatnévsor legújabb bejegyzése. Ha ide kell irni Ha-t, hogy jo legyen a ha.
^C david@david-S15:~/Asztal/prog/caesar$ more titkosított.szöveg
[[[<-->]]]
@M Tovább--(5%)
M# - @[[[<-->]]]
@M [[[[<-->]]]XUUM<-->[[[[<-->]]]]
[[[[<-->]]]Tovább--(33%)
[[[[<-->]]]^L^L[[[[<-->]]]4
[[[[<-->]]]     [[[[<-->]]]L<-->[[[[<-->]]]BM A'
[[[[<-->]]]     [[[[<-->]]]?[[[[<-->]]],[[[[<-->]]]
[[[[<-->]]]     [[[[<-->]]]A
[[[[<-->]]]     [[[[<-->]]]..[[[[<-->]]]D[[[[<-->]]]L/[[[[[<-->]]]..[[[[<-->]]]D[[[[<-->]]]
[[[[<-->]]]     [[[[<-->]]]A
[[[[<-->]]]     [[[[<-->]]]BM A"
[[[[<-->]]]     [[[[<-->]]]L[[[[<-->]]]4
[[[[<-->]]]     [[[[<-->]]]BM)
[[[[<-->]]]
```

Ha fel akarja törni a szövegünket, azaz vissza kívánjuk kapni az eredeti szövegünket, nincs más dolgunk, mint azonos kulccsal újra lefuttatni a programunkat, azonban a titkosított szöveget már bemenetként adjuk meg. Ekkor kimenetként a program visszaadja az eredeti szövegünket.

```
david@david-S15:~/Asztal/prog/caesar$ java ExorTitkositó alma < titkosított.szöveg
A Boston Celtics csapatát 1946-ban alapították, az első tulajdonos Walter A. Brown volt. Először még csak
a BAA-ban (Amerikai Kosárlabda Szövetség) játszottak, de 1949 őszén bekerültek az NBA-be a BAA és az NBA
egyesülése miatt. 1950-ben a Celtics volt az első csapat, amely egy afro-amerikai játékost igazolt, Chuck Cooper személyében. A Kelták az első éveiben nem volt túl jó csapat, egészen Red Auerbach edző leszerző
dtetéséig. Egy kiváló játékos is, Bob Cousy is csatlakozott a Bostonhoz, bár Auerbach edző kezdetben nem
akarta, hogy így legyen. Az 1955-56-os szezon után Auerbach egy meglepő cserével módosította a játékosker
etet. Cliff Heaganért és a válogatott Ed Macaulayért cserébe a San Francisco-i center, Bill Russel neve l
ett a csapatnévsor legújabb bejegyzése. Ha ide kell irni Ha-t, hogy jo legyen a ha.
david@david-S15:~/Asztal/prog/caesar$
```

4.4. C EXOR törő

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

(A gyorsabb futás érdekében a kulcs csak 3 jegyű, de ez 'bármeddig' növelhető.)

```
#define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 3
#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)  
{  
    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");  
}
```

Fontos, a tiszta_lehet függvény vizsgálja a szövegünköt. Itt ha nem talál benne gyakori szavakat (hogy az, nem, ha), akkor hibás kimenetet kapunk (üres lesz a kimenet), ezért fontos, hogy az eredeti szövegünkbe legyenek az előbb említett "gyakori" szavak!

```
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;

    while ((olvasott_bajtok =
        read (0, (void *) p,
        (p - titkos + OLVASAS_BUFFER <
        MAX_TITKOS) ? OLVASAS_BUFFER : titkos + MAX_TITKOS - p)))
        p += olvasott_bajtok;

    for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
        titkos[p - titkos + i] = '\0';

    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]\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 main-ben behúzzuk a titkos fájlt, a for-ban a titkos bufferben a maradék helyet nullázzuk. A nagy for halmazban előállítjuk a kulcsot (annyi for szükséges, ahány elemű a kulcs, jelen esetben három). AZ

ezt követő if-ben megvizsgáljuk, hogy jó kulcsot találtunk-e, ha igen kiirjuk, ha nem eldobjuk és addig próbálkozunk új kulcsokkal, míg meg nem találjuk a helyeset.

Miután a titkosító programunkkal a titkosított szöveget a secret.txt fájlba mentettük, nincs más dolgunk, mint compileolni a törő programot, majd futtatni, bemenetként adva neki a secret.txt-t. Ekkor a program miután feltörte (a feltörési idő a kulcs hosszától függ, minél hoszab a kulcs annál tövább tart feltörni a titkosított szöveget) a szövegünket a kimenetre kiírja a kulcsot valamint az eredeti szöveget.

```
david@david-515:~/Asztal/prog/caesar$ gcc toro.c -o toro
david@david-515:~/Asztal/prog/caesar$ ./toro <secret.txt
Kulcs: [374]
Tiszta szöveg: [A Boston Celtics csapatot 1946-ban alapították, az első tulajdonos Walter A. Brown volt. Először meg capot a Bostonban (Amherst Kollégiuma) Szovjetországban, 1947-ben olcsón bekerült az NBA-be. Az NBA előbbi részlegében a Boston Celtics volt az első csapat, amely sikeresen sportkari játékokat igazolt. Chuck Cooper személyében. A Keltak az első éveiben nem volt túl jó csapat, egészén Red Auerbach edző vezetésével leszerződötték. Egy kiváló játékos is, Bob Cousy is csatlakozott a Bostonhoz, bár Auerbach edző vezetésében nem akarta, hogy így legyen. Az 1955-56-os szezon után Auerbach egy meglepően módosította a játékoskeretet. Cliff Meaganert és a válogatott Ed Macaulayert cserébe a San Francisco-i center, Bill Russell nevű lett a csapatnevőrég legújabb bejegyzése. Ha ide kell trafi ha-t, hogy jó legyen a ha.]
```

4.5. Neurális OR, AND és EXOR kapu

R

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

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

A mesterséges neutrális háló biológiai alapokon nyugszik, az ottani neuronok/idegekből eredeteztethető. Fő használati területe az informatikában a gépi tanulás. Gráf alapú modell, ebben mesterséges neuronok kommunikálnak, melyek rétegekbe rendezettek.

Minden részben használnunk kell a 'neutralnet' könyvtárat.

Neurális or/vagy művelet:

```
library(neuralnet)

a1      <- c(0,1,0,1)
a2      <- c(0,0,1,1)
OR      <- c(0,1,1,1) #a 3 sorban 'megmondjuk' hogy a1 és a2-ből mit kell ←
                     kapni
#defineíáljuk az OR/Vagy szabályát

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])
```

A kódban az 'a1' és 'a2'-nek megadjuk a vagy művelet minden(4) lehetséges kombinációját, majd az 'or'-ba megadjuk a vagy művelet elvégzése utáni eredményt. Az 'or.data'-ba bevisszük a 'data.frame' eredményét. A végén pedig rajzoltatunk.

And/és:

```
library(neuralnet)

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

and.data <- data.frame(a1,a2,AND)

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

plot(nn.and)

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

A program szinte azonos az előbbivel, azonban értelelem szeűen itt nem vagy, hanem az és művelet van a progiban. Ugyanúgy az 'AND' sorban az és művelet eredményeit adjuk.

Exor/kizáró vagy:

```
library(neuralnet)

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])
```

Szintén azonos a működés, csak ezáltal a kizáró vagy műveletének eredményeit adjuk meg.

Azonban ezen paraméterekkel a tanítás nem igazán eredményes, azaz nagy hibaszázalékkal dolgozik, azonban, ha két rejtett neuront adunk a proginak a tanítás már sikeres lesz és a hibaszázalék is kellőképp kicsi lesz.

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó: "youtube.com/watch?v=XpBnR31BRJY&feature=youtu.be"

Megoldás forrása:

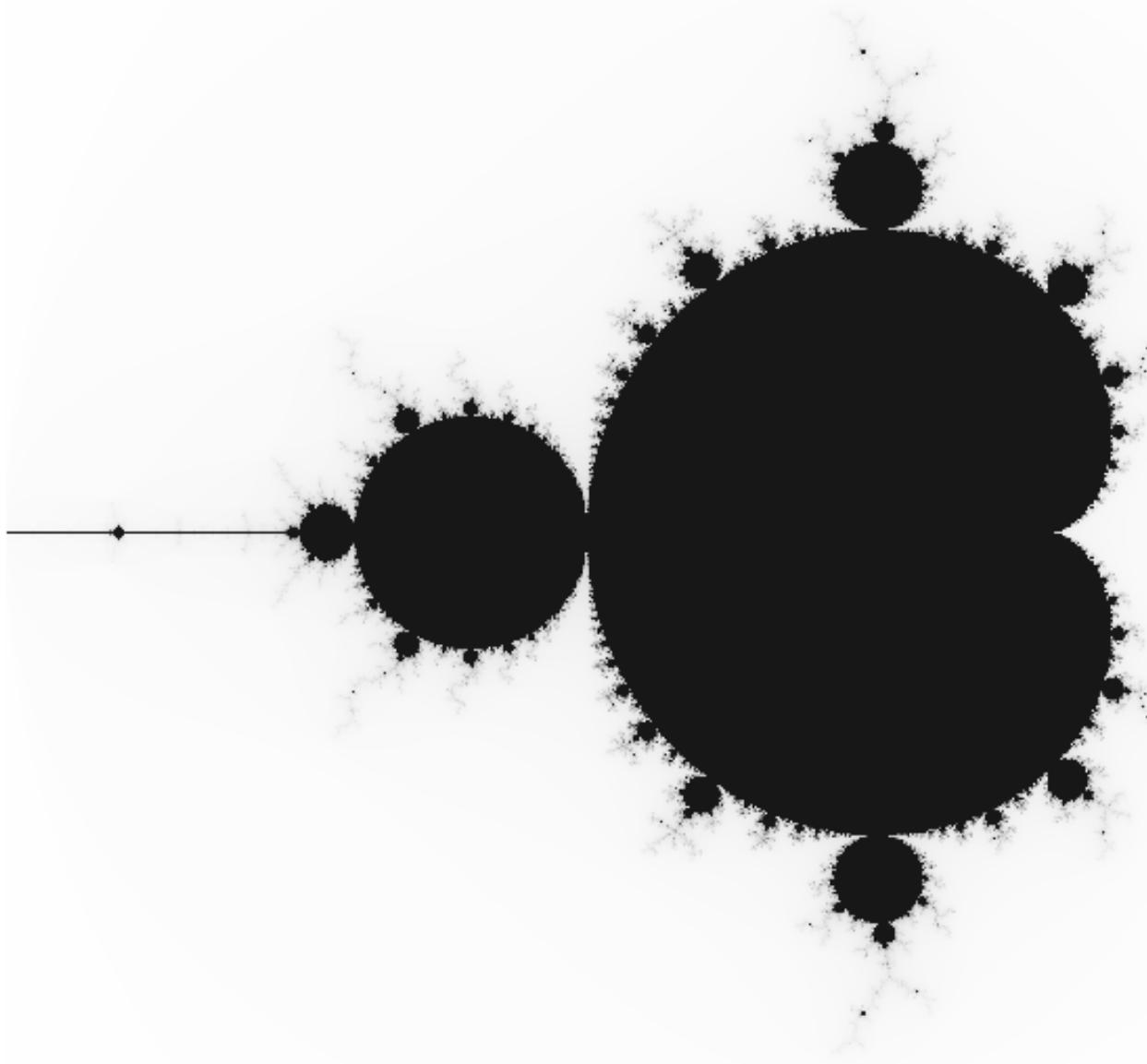
<https://github.com/szuhi27/prog1/blob/master/caesar/mandelpng.cpp>

<https://github.com/szuhi27/prog1/blob/master/caesar/mlp.hpp>

```
#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;
}
```

A mandelpng.cpp programra lesz szükségünk, ami futtatása után kapjuk ezen képet.



Eztán tudjuk futtatni a main programot, a megszokott módon.

```
david@david-S15:~/Asztal/prog/caesar$ g++ mlp.hpp main.cpp -o hat -lpng
david@david-S15:~/Asztal/prog/caesar$ ./hat fajl
0.730615
david@david-S15:~/Asztal/prog/caesar$ █
```

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Tutoriált: Oláh Sándor

A halmaz felfedezője és névadója Benoit Mandelbrot, 1980.

Mandelbrot halmaz áll azon 'c' komplex számokból, amikre az x_n rekurzív sorozat:

$x_1 := c$

$x_{n+1} := \text{sqr}(x_n) + c$

nem tart a végelenbe, vagyis abszolút értékben korlátos.

Mandelbrot halmaz a komplex számsíkon ábrázolva egy nevezetes fraktálhalmaz jön létre.

Kirajzoltatást tekintve a gép minden pixelre kiszámolja, hogy az adott pontosrozat a végelebe tart-e vagy sem. Szabély szerint, ha az adott sorozat egy tagának abszolútértéke nagyobb, mint 2, akkor a végelebe tart. Bár ezen határ növelésével a kapott kép is változik.

Az adott programot használtuk az előző feladatban is, a kép készítésére.

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

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

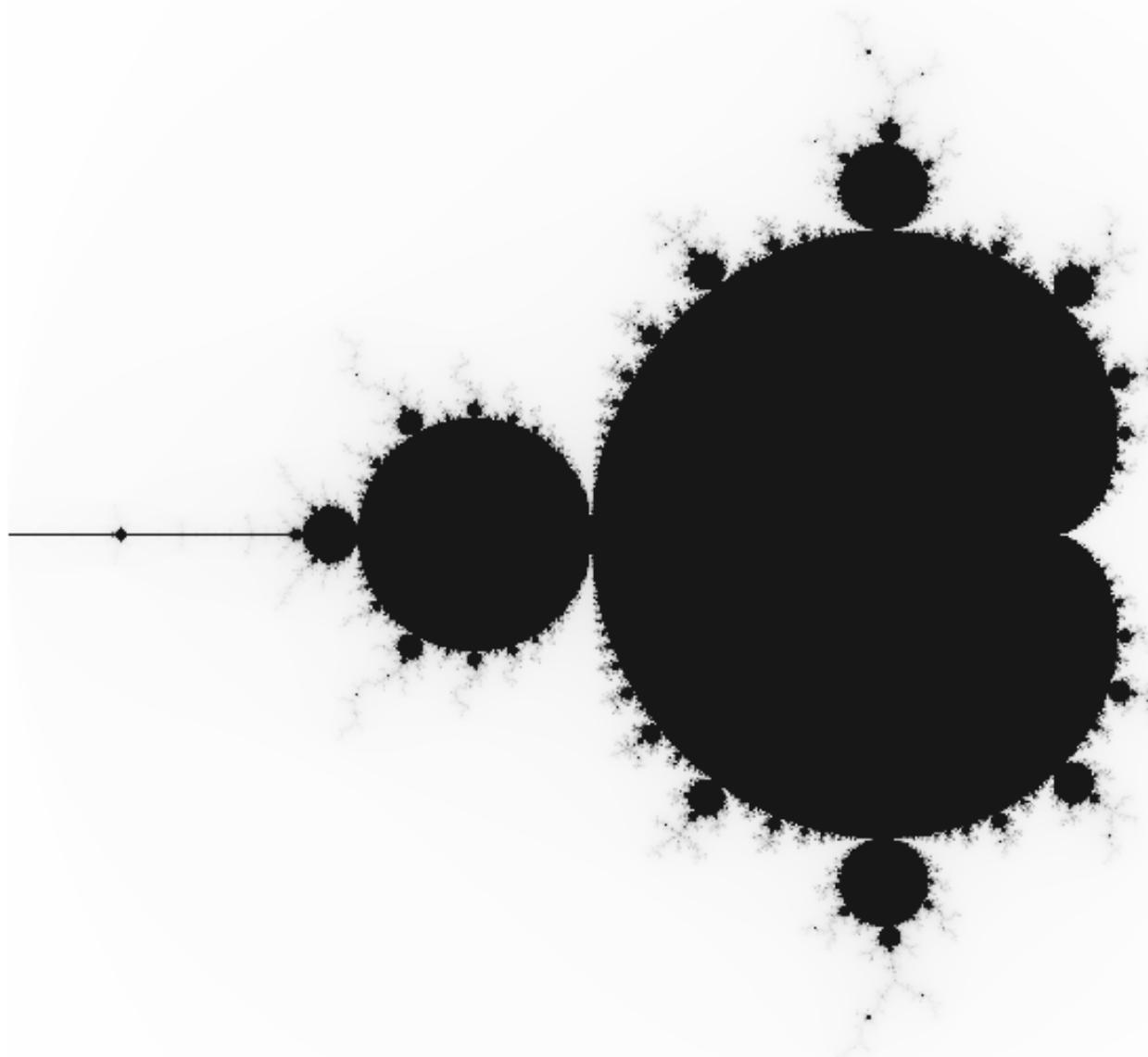
    // számítás adatai
    double a = -2.0, b = .7, c = -1.35, d = 1.35;
    int szelesseg = 600, magassag = 600, iteraciosHatar = 1000;

    // png-t készítünk a png++ csomaggal
    png::image<png::rgb_pixel> kep (szelesseg, magassag);
```

```
// a számítás
double dx = (b-a)/szelessseg;
double dy = (d-c)/magassag;
double reC, imC, rez, imZ, ujrez, ujimZ;
// Hány iterációt csináltunk?
int iteracio = 0;
std::cout << "Szamitas";
// Végigzongorázzuk a szélesség x magasság rácsot:
for (int j=0; j<magassag; ++j) {
    //sor = j;
    for (int k=0; k<szelessseg; ++k) {
        // c = (reC, imC) a rács csomópontjainak
        // megfelelő komplex szám
        reC = a+k*dx;
        imC = d-j*dy;
        // z_0 = 0 = (rez, imZ)
        rez = 0;
        imZ = 0;
        iteracio = 0;
        while (rez*rez + imZ*imZ < 4 && iteracio < iteraciosHatar) {
            // z_{n+1} = z_n * z_n + c
            ujrez = rez*rez - imZ*imZ + reC;
            ujimZ = 2*rez*imZ + imC;
            rez = ujrez;
            imZ = ujimZ;

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

Az első if-ben vizsgáljuk, hogy megfelően futtatuk-e a programot, ha nem kiírja hogy lenne helyes. Elvégezzük a szükséges számításokat, majd a for ciklusban végigmegyünk a pixelrácson. minden helyre adunk egy megfelelő értéket, majd a pixeleket az értékeknek megfelelően beszínezzük. A végén, ha minden jó kiirjuk , hogy mentve és az alábbi képet kapjuk.



5.2. A Mandelbrot halmaz a std::complex osztályval

Feladatunk szintén egy mandelbrot halmaz készítő program, ám ezúttal az std::complex osztály használatával.

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

int
```

```
main ( int argc, char *argv[] ) {
    int szelesseg = 1920;      //a készítendő kép szélessége pixelekben
    int magassag = 1080;      //a készítendő kép magassága pixelekben
    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;
    }
}
```

Eddig bevittük a később használt értékeket, mint szélesség, magasság és társai, vagyis adtunk nekik egy-egy alap értéket. Majd vizsgáljuk, hogy bemeneten megfelelő értékeket kapott-e ha igen elvégezzük a megfelelő értékadásokat. Ha nem kiírjuk a helyes használatot.

```
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 ) );
}

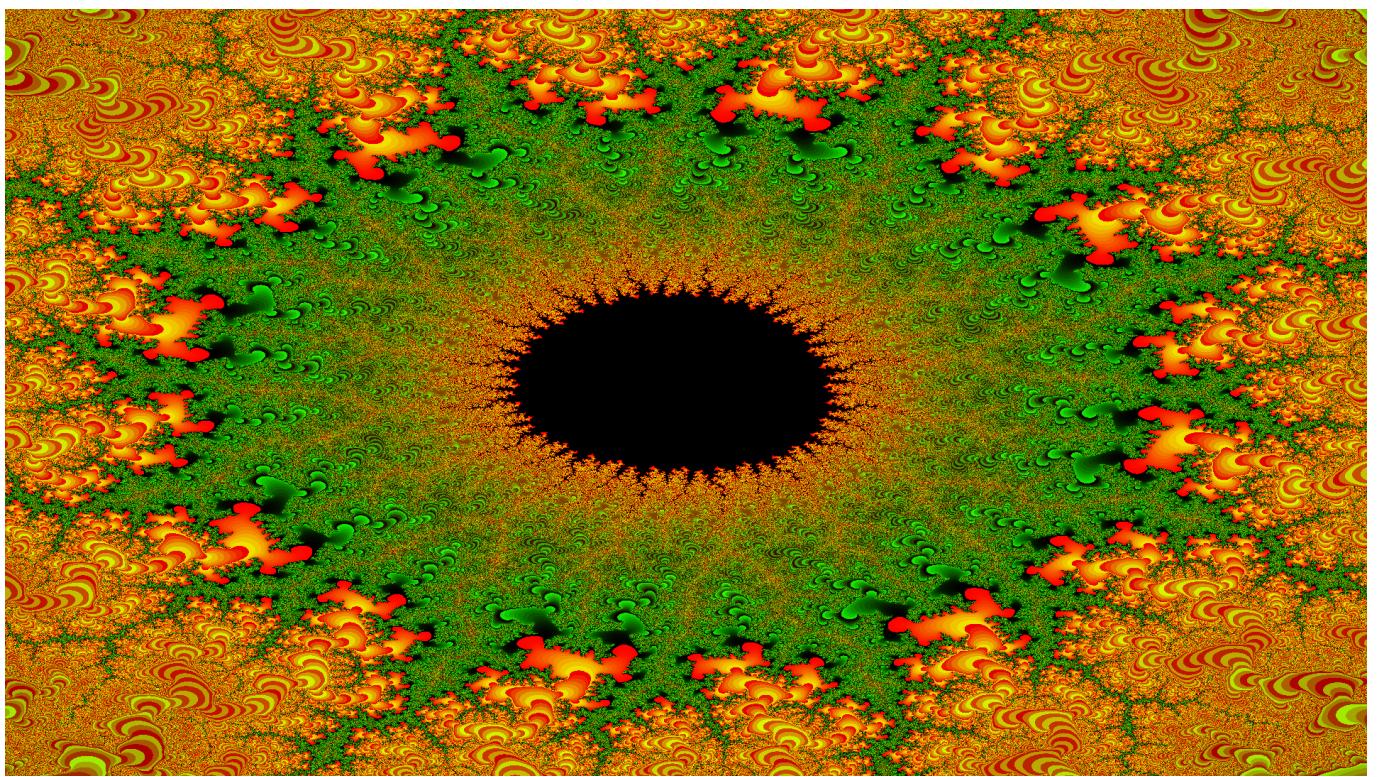
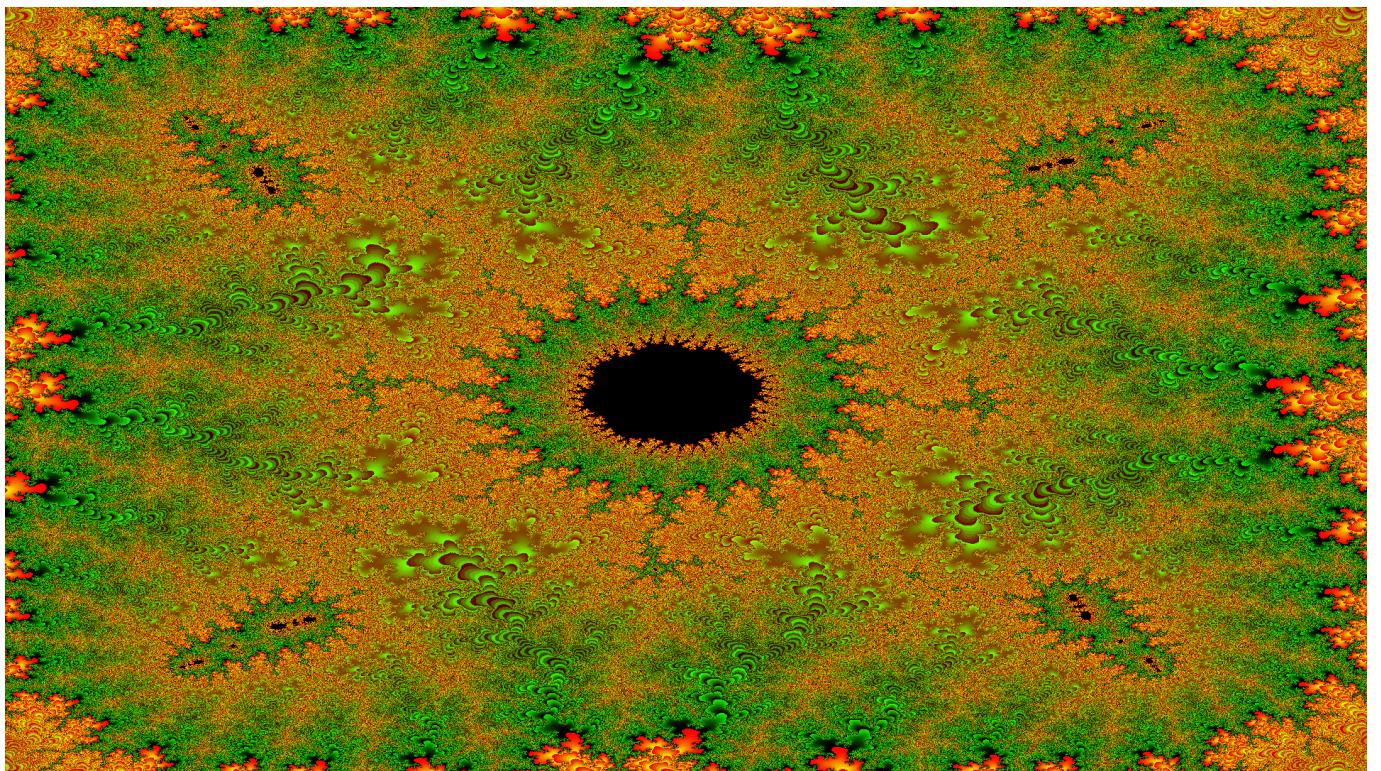
int szazalek = ( double ) j / ( double ) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}

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

Szintén végigmegyünk a rácson értékeket számolunk az elemeknek, majd azokhoz egy-egy színt rendelünk az elem értékének megfelően. Lentebb láthatunk kettőt a program által létrehozott képek közül, ezek egy-egy pontos bemenettel jöttek létre, ahogy pontosan meg volt határozva, hogy milyen értékeket számoljon a program. Ha megpróbálunk véletlenszerű számokat beadni nagy valószínűséggel egy "értelmetlen" képet fogunk kapni.

```
david@david-S15:~/Asztal/prog/mandelbrot$ g++ 3.1.2.cpp -lpng -O3 -o 3.1.2
david@david-S15:~/Asztal/prog/mandelbrot$ ./3.1.2 mandel.png 1920 1080 2040 -0.01947381057309366
392260585598705802112818 -0.0194738105725413418456426484226540196687 0.7985057569338268601555341
774655971676111 0.798505756934379196110285192844457924366
Szamitas
mandel.png mentve.
david@david-S15:~/Asztal/prog/mandelbrot$ 
```

A program által készített képek:



5.3. Biomorfok

A biomorfok, más néven Juli halmaz nagyon hasonlítnak biológiai sejtekre, innen ez az elnevezés. A felfeledzője azt hitte nagy dologra jött rá, de végül kiderült, hogy csak egy véletlennek köszönheti, mondhatni

egy hibának.

A Mandelbrothoz képest Julia-halmazban nem a z a változó, hanem a c .

```
#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: ./3.1.2 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 < szelessseg; ++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 < iteracionsHatar; ++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;
}

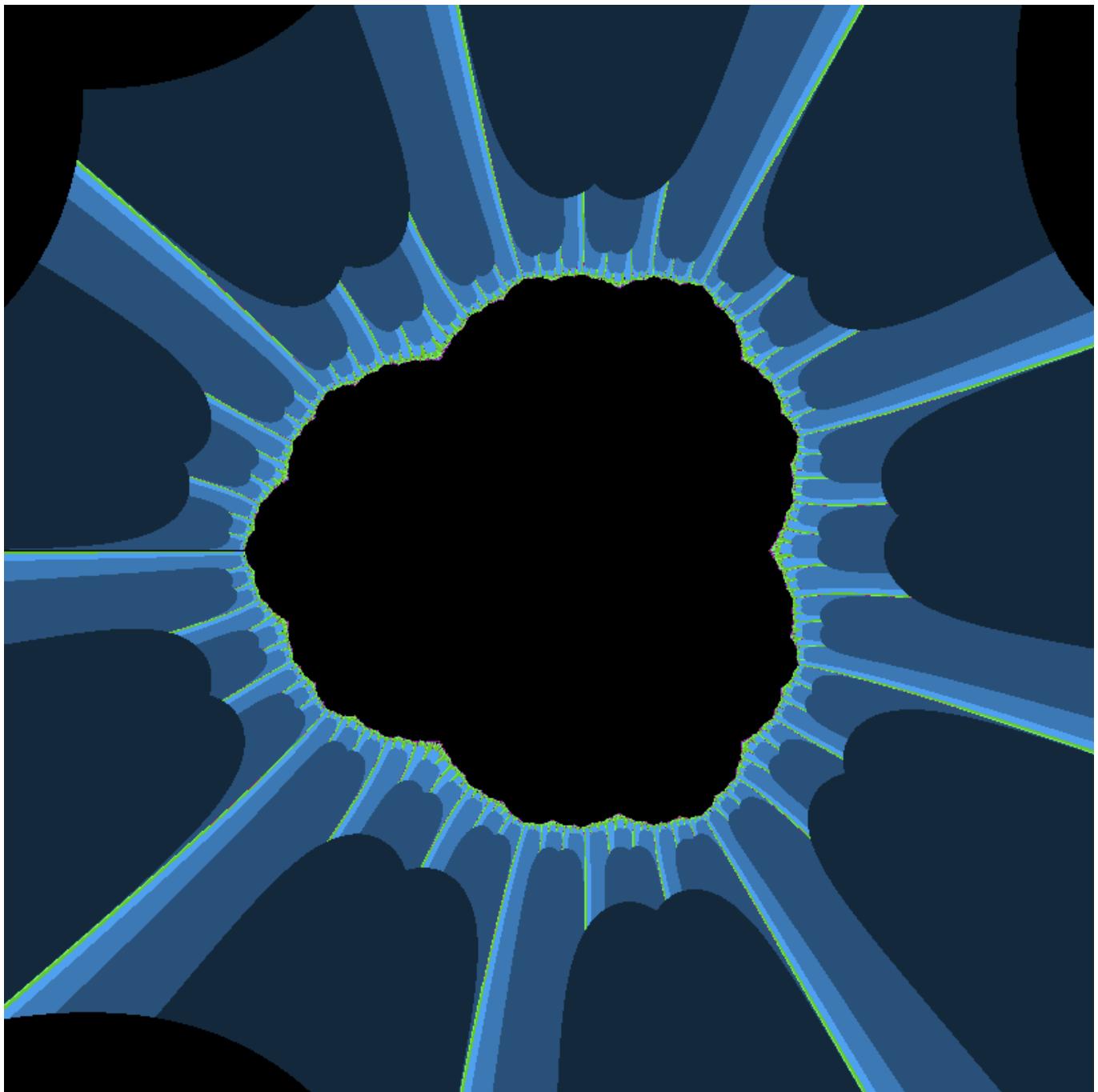
kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;

}
```

Ezen program nagyon hasonlatos az előzőhez, csak néhány sorban tér el. Lényegesebb különbség a kettő között, hogy az előzőben egy rövidebb while ciklus van, itt egy komplexebb for ciklus található. A 'nagy' for ciklus zintén végig megy az egész képen és kiszámolj aez értékeit. A belső for az iterációs határig megy el , azon belül szímitásokat végezünk, itt határozza meg a készükő képen levő színek tulajdonságait.

Az utolsó műveletben pedig kimentjük a képet, a parancssorban meghatározott néven, majd kiirjuk, hogy el vna mentve, majd megtekinthetjük a képet. A lenti kép ezzel a programmal készült. Jól látszik rajta a sejt szeű kinézet.

```
david@david-S15:~/Asztal/prog/mandelbrot$ g++ 3.1.3.cpp -lpng -O3 -o 3.1.3  
david@david-S15:~/Asztal/prog/mandelbrot$ ./3.1.3 biom.png 800 800 10 -2 2 -2 2 .285 0 10  
Szamitas  
biom.png mentve.  
david@david-S15:~/Asztal/prog/mandelbrot$
```



5.4. A Mandelbrot halmaz CUDA megvalósítása

Nvidia kártya hiányában nincs CUDA.

5.5. Mandelbrot nagyító és utazó C++ nyelven

Célunk hogy létrehozzunk egy mandelbrot halmazt, majd, ahova kattintunk oda a program ránagyítson, ezzel újab értékeket számolva. Ilyenfajta nagyítás mondhatni a végtelenségig lehetséges, a jelenleg ismert legnagyobb nagyítás 750.000.000-os. Érteleм szerűen minél tovább nagyítunk a programnak annál többet kell számolnia, vagyis minden nagyítás egyre tovább tart. A lent látható linken egy .gif van, amit a nagyító által adott képekből készítettem.

Programunk futásához szükségünk lesz az SFML-re, ezért ha futtatni akarjuk előtte töltsük le azt, operációs rendsérőkhöz megfelelően. A compileolás illetve futtatás alább meglátható.

```
#include "SFML/Graphics.hpp"

//a készítendő ablak méreteit adja meg(pixelben), ezt változtatható kedv ←
    szerint
const int width = 1920;
const int height = 1080;

//komplex számokhoz használatos
struct complex_number
{
    long double real;
    long double imaginary;
};
```

Az alábbi "generate_mandelbrot_set", nevéből egyértelműen generálja magát a halmazt. Ahogy a korábbi programokban itt is végigmegy a pixeleken és kiszámolja értékeiket. Azonban itt figyelembe kell venni, azt is hogy nagyítunk ezzel változtatva az alap értékeket, ezért is van egy változónk, ami a nagyítások számát figyeli. Valamint a pixlek színei is függnek a nagyításuktól, ezért erre is külön rész szolgál.

```
void generate_mandelbrot_set(sf::VertexArray& vertexarray, int ←
    pixel_shift_x, int pixel_shift_y, int precision, float zoom)
{
#pragma omp parallel for
    for(int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {

            long double x = ((long double)j - pixel_shift_x) / zoom;
            long double y = ((long double)i - pixel_shift_y) / zoom;
            complex_number c;
            c.real = x;
            c.imaginary = y;
            complex_number z = c;
            int iterations = 0; //a nagyítások számát tartja számmal
            for (int k = 0; k < precision; k++)
            {
                complex_number z2;
                z2.real = z.real * z.real - z.imaginary * z.imaginary;
```

```
        z2.imaginary = 2 * z.real * z.imaginary;
        z2.real += c.real;
        z2.imaginary += c.imaginary;
        z = z2;
        iterations++;
        if (z.real * z.real + z.imaginary * z.imaginary > 4)
            break;
    }
//'megszínezi' az adott pixeleket, a nagyítások száma alapján
if (iterations < precision / 4.0f)
{
    vertexarray[i*width + j].position = sf::Vector2f(j, i);
    sf::Color color(iterations * 255.0f / (precision / 4.0f), ←
        0, 0);
    vertexarray[i*width + j].color = color;
}
else if (iterations < precision / 2.0f)
{
    vertexarray[i*width + j].position = sf::Vector2f(j, i);
    sf::Color color(0, iterations * 255.0f / (precision / 2.0f) ←
        , 0);
    vertexarray[i*width + j].color = color;
}
else if (iterations < precision)
{
    vertexarray[i*width + j].position = sf::Vector2f(j, i);
    sf::Color color(0, 0, iterations * 255.0f / precision);
    vertexarray[i*width + j].color = color;
}
}
}
}
```

A main-ben természetesen meghívjuk a halmaz készítő fügvényt, valamint figyeljük az ablakot, hogy fenn van-e. Illetve természetesen, hogy kattintunk és, hogy a kattintás hova történik. Ezen dolgok változásával adunk más-más értékeket a halmaz készítőnek. A végén persze letörüljük az ablakot és újra kirajzoltatjuk a már ráközelített halmazról.

```
int main()
{
    sf::String title_string = "Mandelbrot Set Plotter";
    sf::RenderWindow window(sf::VideoMode(width, height), title_string);
    window.setFramerateLimit(30);
    sf::VertexArray pointmap(sf::Points, width * height);

    float zoom = 300.0f;
    int precision = 100;
    int x_shift = width / 2;
    int y_shift = height / 2;
```

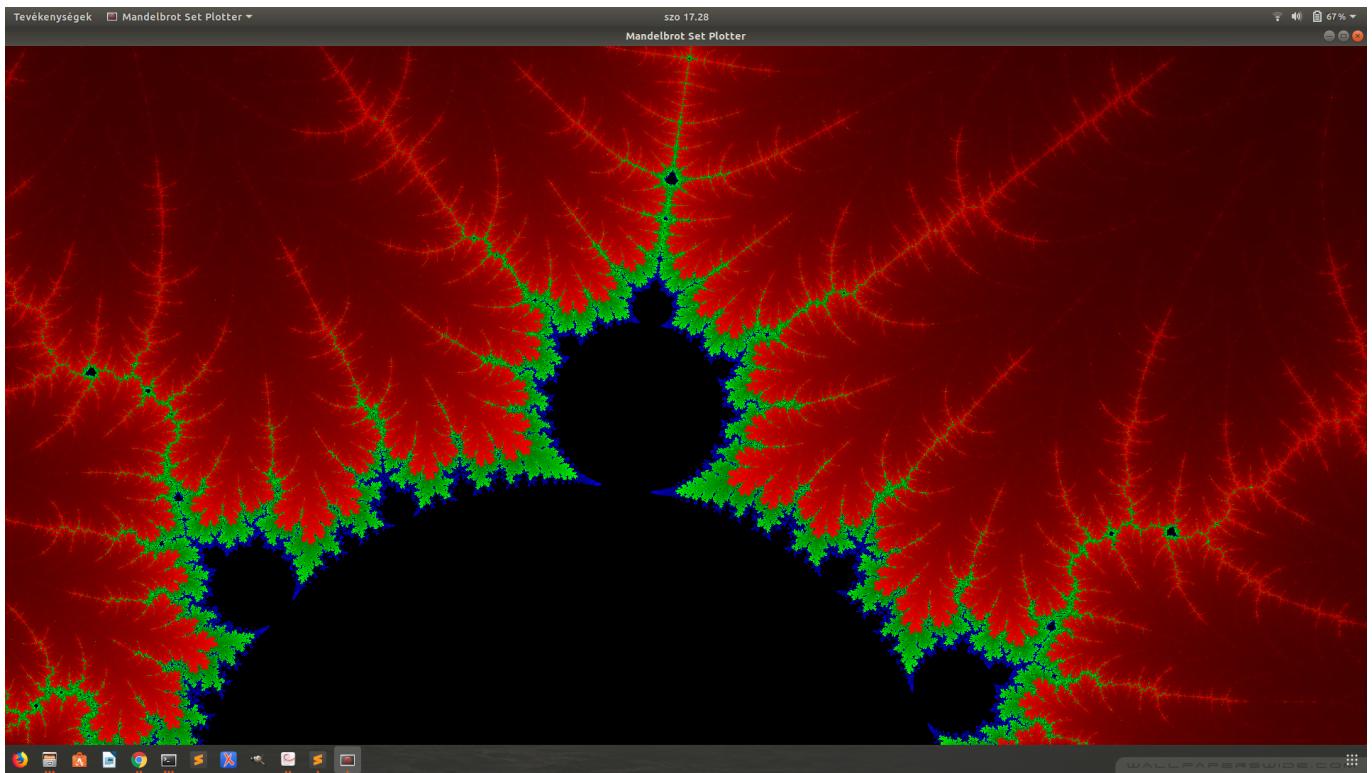
```
generate_mandelbrot_set(pointmap, x_shift, y_shift, precision, zoom);

while (window.isOpen())
{
    sf::Event event;
    while (window.pollEvent(event))
    {
        if (event.type == sf::Event::Closed)
            window.close();
    }

    //zoomol arra a területre ahova kikkelve lett
    if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
    {
        sf::Vector2i position = sf::Mouse::getPosition(window);
        x_shift -= position.x - x_shift;
        y_shift -= position.y - y_shift;
        zoom *= 2;
        precision += 200;
    }
    #pragma omp parallel for
    for (int i = 0; i < width*height; i++)
    {
        pointmap[i].color = sf::Color::Black;
    }
    generate_mandelbrot_set(pointmap, x_shift, y_shift, precision, ←
                           zoom);
}
window.clear();
window.draw(pointmap);
window.display();
}

return 0;
}
```

```
david@david-S15:~/Asztal/prog/mandelbrot/zom/z$ g++ -c zoom.cpp
david@david-S15:~/Asztal/prog/mandelbrot/zom/z$ g++ zoom.o -o sfml-app -lsfml-graphics -lsfml-window -lsfml-system
david@david-S15:~/Asztal/prog/mandelbrot/zom/z$ ./sfml-app
```



A program által létrehozott zoom-ot az alábbi linken meg lehet tekinteni (nem sikerült xml-be megjeleníttetnem): <https://github.com/szuhi27/prog1/blob/master/mandel.gif>

5.6. Mandelbrot nagyító és utazó Java nyelven

<https://github.com/szuhi27/prog1/blob/master/mandel/MandelbrotHalmazNagyító.java>

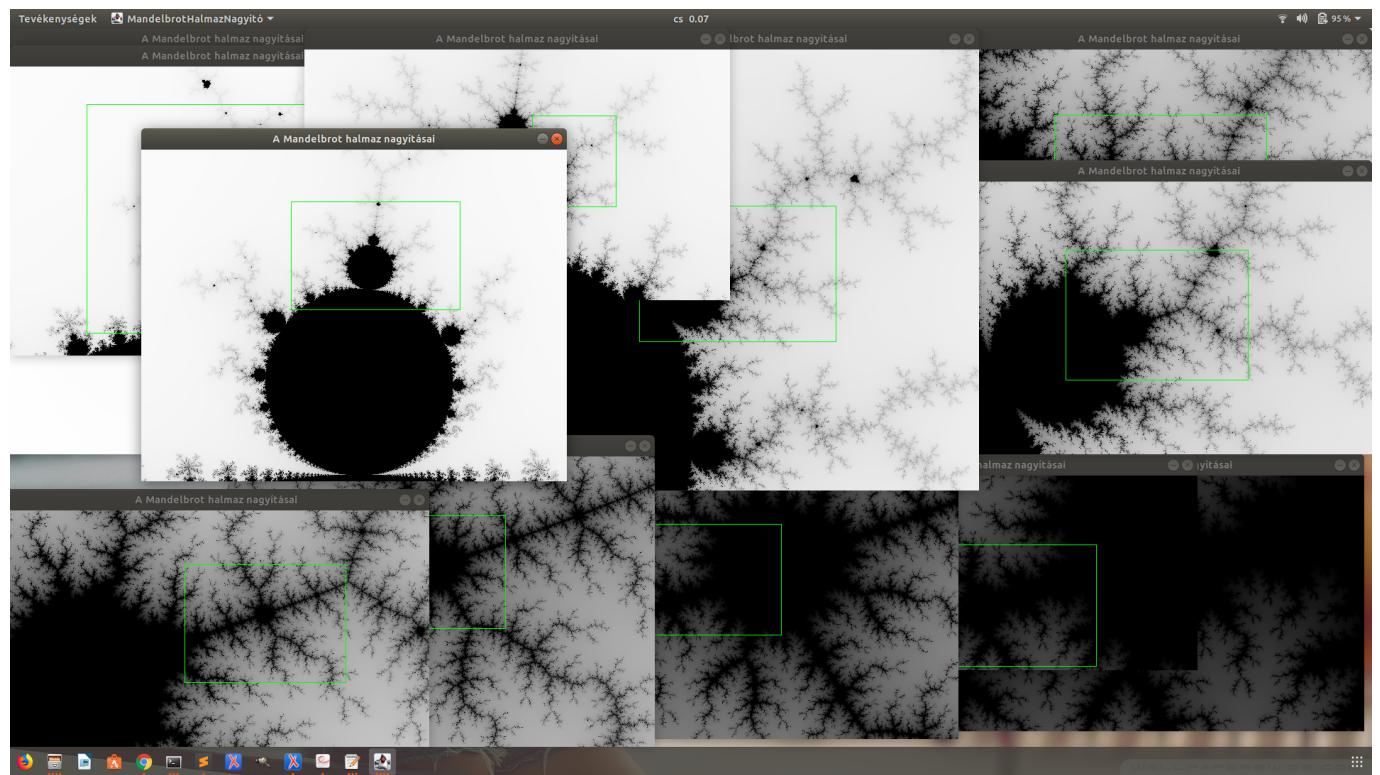
<https://github.com/szuhi27/prog1/blob/master/mandel/MandelbrotHalmaz.java>

<https://github.com/szuhi27/prog1/blob/master/mandel/MandelbrotIterációs.java>

Feladatunk azonos az előzővel, azonban ezúttal javában. Ez a program nem kattintások után zoomol, hanem ki kell jelölnünk egy területet, amire rázoomol a program. Ahogy a képeken is látszik itt nincsenek színek, csak fekete és fehérből áll a halmaz. Viszont itt minél beljebb nagyítunk annál sötétebb lesz a kép, míg végül egyszer csak teljesen fekete képet kapunk.

Fordítás könnyen megy, a 3 fájlt be kell rakni egy mappába és a lent látható módon fordítani és futtatni.

```
david@david-S15:~/Asztal/prog/mandelbrot/manzj$ javac MandelbrotHalmazNagyító.java
david@david-S15:~/Asztal/prog/mandelbrot/manzj$ java MandelbrotHalmazNagyító
david@david-S15:~/Asztal/prog/mandelbrot/manzj$ █
```



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.

C++ megoldás:

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

class PolarGen{
public:
    PolarGen();
    double next();
    ~PolarGen() {}
private:
    bool nincsTarolt;
    double tarolt;
};

PolarGen::PolarGen() {
    nincsTarolt = true;
    std::srand (std::time(NULL));
}
```

A PolarGen osztályt létrehozzuk benne definiálunk néhány változót illetve eljárást. Aztán a PolarGen-ben megadjuk a nincsTarolt boolean tipusu változónak a true értéket.

```
double PolarGen::next() {
    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 'next()' eljárás a PolarGen-hez tartozik, ezt vezetjük be. Ezen belül, ha a nincsTarolt értéke igaz létrehozunk 5 double típusú váltpzót és ezekkel számításokat végezünk. Az u1 és u2-nek adunk egy számot 0-1-ig, a v1 és v2-nek adjuk az előbb generált értékek kétszerese-1-et ($u1 \cdot v1; u2 \cdot v2$), majd a w-ben berakjuk a v1 és v2 négyzetéinek összegeit. Ezen műveleteket addig csináljuk, amíg a w értéke nagyob, mint 1, amint az már kiseb, mint 1 továbblépünk. Ha továbbléptünk az r-be eltároljuk a w logaritmusának -2-szeresét, amit osztunk w-vel majd ennek vesszük még a gyökét. A tarolt-nak megadjuk az r kétszereség és a nincsTároltat ellentétére állítjuk, azaz false-ra. Végül viisszadajuk az r v1-szeresét. Ha viszont a nincsTarolt értéke eredetileg false, akkor azt megváltoztatjuk és visszaadjuk a tarolthat.

```
int main(){
    PolarGen rnd;
    for (int i = 0; i < 10; ++i)
        std::cout << rnd.next() << std::endl;
}
```

A mainben meghívjuk a PolarGen-t és az rnd-t egy for ciklusban 10 szer futtatjuk. A kimeneten láthatjuk a program által generált 10 értéket.

```
david@david-S15:~/Asztal/prog/welch$ g++ polár.cpp -o pol
david@david-S15:~/Asztal/prog/welch$ ./pol
0.138666
-0.485073
-1.28815
-1.06906
-1.43296
-0.988425
0.0608496
0.905193
-0.99888
-0.107419
david@david-S15:~/Asztal/prog/welch$ □
```

A java megoldás:

```
public class PolárGenerátor {

    boolean nincsTárolt = true;
    double tárolt;

    public PolárGenerátor() {

        nincsTárolt = true;

    }

    public double következő() {

        if(nincsTárolt) {

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

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

                w = v1*v1 + v2*v2;

            } while(w > 1);

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

            tárolt = r*v2;
            nincsTárolt = !nincsTárolt;

        }

    }

}
```

```
        return r*v1;

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

public static void main(String[] args) {

    PolárGenerátor g = new PolárGenerátor();

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

}
}
```

A program azonos elven működik, mint a c++, ugyanazt csinálja csak ezúttal java nyelvet alkalmazunk a megoldásra. Láthatjuk, hogy a kimeneten itt is megkapjuk a 10db generált számot.

```
david@david-S15:~/Asztal/prog/welch$ javac PolárGenerátor.java
david@david-S15:~/Asztal/prog/welch$ java PolárGenerátor
0.7721945025717902
-0.15887838310331975
0.43451819737941677
1.4529576652755365
0.4581313358832149
0.473255558892007
1.3672219004078954
-0.2465153160559716
-1.8978726777141997
0.42106242342332734
david@david-S15:~/Asztal/prog/welch$ █
```

6.2. LZW

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

Az LZW algoritmus egy veszteségmentes tömörítési algoritmus. 1984-re datálható és a feladatok névadójához, Terry Welch-hez kötődik. Jelen feladatban ezen algoritmus fa építésének a megvalósítását kell elvégezni. Ez a fa egy bináris fa, vagyis minden szülőnek 0, 1 vagy 2 gyereke lehet, ezek a gyerekek 0 vagy 1-esek. A szülők jobb oldalán "állnak" az egyesek gyerekek, bal oldalukon pedig a nullák. Így épül fel a fa,

minden szülőnek és gyereknek van egy mélysége, ami gyakorlatilag azt mutatja meg, hogy hány lépéssel tudunk eljutni a gyökérig. A binfát számos egyéb algoritmus használja, ilyen például a kupac rendezés.

Minden következő "fás" feladatnál az alábbi tartalmú input fájlt adjuk a programnak: "LIDUS". Itt természetesen hoszab szöveget is használhatunk, de a könnyeb értelmezés érdekében ilyen rövid.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <fcntl.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;
}
```

Létrehozzuk a struktúrát, majd vizsgéljük, hogy a fának le tudjuk e foglani a memóriát, na nem hiírjuk hibáként.

```
extern void kiir (BINFA_PTR elem, FILE * of);
extern void ratlag (BINFA_PTR elem);
extern void rszoras (BINFA_PTR elem);
extern void szabadit (BINFA_PTR elem);

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

    if (argc != 4 || argv[2][0] != '-' || argv[2][1] != 'o' || argv[2][2] != '\0'){
        printf ("\nUsage: ./futtathato_fajl input_fájl_neve -o kimeneti_fájl_neve\n\n");
        return -1;
    }
    int inputfile = open (argv[1], O_RDONLY);
    if (inputfile == -1){
        printf ("\nA bemeneti fájl nem létezik!\n\n");
    }
```

```
        return -1;
    }

FILE *outputfile;
outputfile = fopen (argv[3], "w");
if (!outputfile){
    printf ("\nA kimeneti fájl nem jött létre!\n\n");
    return -1;
}

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

while (read (inputfile, (void *) &b, sizeof (unsigned char))) {
    for (i = 0; i < 8; i++){
        egy_e = b & 0x80;
        if ((egy_e >> 7) == 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;
            }
        }
        b <= 1;
    }
}
```

Mainben vizsgáljuk, hogy jól futtatuk-e le a programot, ha nem, kiírja a helyes futtatási módot. Majd vizsgáljuk, hogy a bemeneti fájt helyes-e, létezik -e, illetve a kimeneti fájl létrejött-e, ha bármelyik meghiúsul kiírja az adottat hibaként. Meghívjuk az új elem függvényt, ami lefoglalja a memóriát. A gyökérnek értékül adjuk a "/" jelet, majd a 'bal nulla' és a 'jobb_egy' mutatókat NULL-ra állítjuk. While ciklust indítunk, ami hétszer fut le, ebben létrejön a binfa készítése, vizsgáljuk, hogy egyes vagy nullás az éppen vizsgált elem értéke. Ha egyes akor a helyére rakjuk majd visszamegyünk a gyökérre, ha épp a gyökéren állunk

addig megyünk, amíg heylet nem találunk neki. Ugyanezt elvégezzük, ha nulla az értéke.

```
fprintf (outputfile, "\n");
kiir (gyoker, outputfile);

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

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

atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
ratlag (gyoker);
atlag = ((double) atlagosszeg) / atlagdb;
atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
szorasosszeg = 0.0;

rszoras (gyoker);

double szoras = 0.0;

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

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

szabadit (gyoker);

close(inputfile);
fclose(outputfile);

printf ("\nA kimeneti fájl lásstrejäsent!\n\n");
}
```

Kiiratunk egy sortörést, majd a kiir fügvényt meghívjuk a gyökérre, eztán kiirjuk a mélységet, átlagot és szórást a fának. Mjad meghíjuk a gyökérre a szabadít függvényt, eztán bezárjuk az imput és output fájlt s kiírjuk, hogy létrejött a fájl, azaz a binfa.

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

void
ratlag (BINFA_PTR fa) {

    if (fa != NULL) {
        ++melyseg;
        ratlag (fa->jobb_egy);
```

```
ratlag (fa->bal_nulla);
--melyseg;

if (fa->jobb_egy == NULL && fa->bal_nulla == NULL) {

++atlagdb;
atlagosszeg += melyseg;

}

}

}
```

A ratlag függvény, ha a fa mutató mutat valahova, lemeg a bal és jobb ágakon, ha leért az atlagdb-t növeli egyel, majd az atalgosszeg-hez hozzáadja a melyseget.

```
double szorasosszeg = 0.0, atlag = 0.0;

void
rszoras (BINFA_PTR fa) {

if (fa != NULL) {
    ++melyseg;
    rszoras (fa->jobb_egy);
    rszoras (fa->bal_nulla);
    --melyseg;

    if (fa->jobb_egy == NULL && fa->bal_nulla == NULL) {

++atlagdb;
szorasosszeg += ((melyseg - atlag) * (melyseg - atlag));

    }

}

}
```

Ha a fa mutat valahova, növel ia mélységet, majd meghívja magát a bal és jobb oldalra, ha laeért az aljára, csökkenti a mélyéset. Ha a job és bal oldal is null, növeli az atlagdb-t illetve a szorasosszeghez hozzáadja a (melyseg-atlag) négyzetét.

```
int max_melyseg = 0;

void
kiir (BINFA_PTR elem, FILE * of) {
    int i;
    if (elem != NULL) {
        ++melyseg;
        if (melyseg > max_melyseg)
```

```
max_melyseg = melyseg;
    kiir (elem->jobb_egy, of);
    for (i = 0; i < melyseg; ++i)
        fprintf (of, "---");
        fprintf (of, "%c(%d)\n",
            elem->ertek < 2 ? '0' + elem->ertek : elem->ertek,
            melyseg - 1);
        kiir (elem->bal nulla, of);
        --melyseg;
    }
}
```

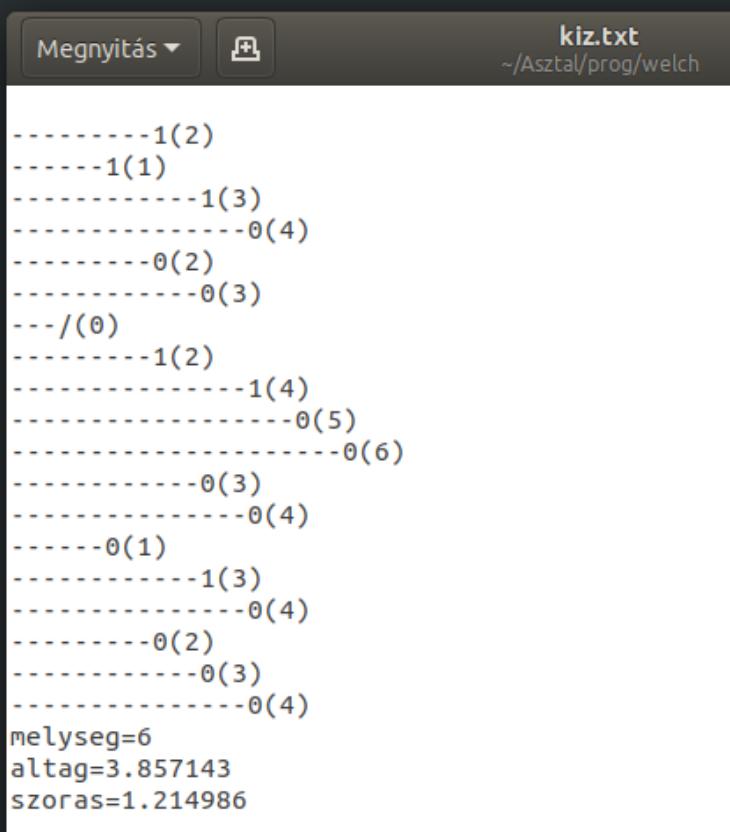
Értelemben szerűen a kiir függvény végzi a kiíratást, akképp, hogy végigmegy a szárakon és megkeresi a végét, ha "mélyebben" van, mint az eddigi legmélyebb möveli a max_melyseget. A kiiratás úgy történik, hogy kiirja az értéket (0 v. 1) kiír mélységnyi '---'-t az érték előtt és az érték mögött zárójelbe annak a mélységét.

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

A szabadit függvény végigmegy a szárakon és felszabadítja a lefoglalt memóriát.

A program által készített bináris fa:

```
david@david-S15:~/Asztal/prog/welch$ gcc z.c -lm -o z
david@david-S15:~/Asztal/prog/welch$ ./z bifa.txt -o kiz.txt

A kimeneti fájl látható!
david@david-S15:~/Asztal/prog/welch$ 
```

A képen jól látható a fa jellege, hogy a szülők jobb oldalán az egyesek bal oldalukon pedig a nullások. Az elemek mögötti zárójelben levő számok a mélységet mutatják, vagyis, ahogy korábban is említettem, azt, hogy mennyi lépéssel jutunk el a gyökérig. A féjl alján pedig jól látjuk, hogy az adott fa "legmélyebb" pontja 6 mélységű.

6.3. Fabejárás

Oláh Sándor segítségével(tutor)

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

Feladatunk ugyanaz mint előzőleg, csupán a kiiratást kell megváltoztatnunk. Ezt a program futtatásánál adjuk meg, mégpedig: 0= inorder, 1= postorder, minden más preorder.

Különbségek az előzőhöz képest:

```
enum Bejar{inorder =0, postorder=1,preorder=2};
Bejar bejar;
class LZWBInFa
{
public:
    LZWBInFa () :fa (&gyoker)
    {
```

```
    }
~LZWBinFa ()
[...]
```

A kiiratás fügvény, amiben eldöntődik, hogy milyen módon végződik a kiiratás.

```
void kiir (Csomopont * elem, std::ostream & os, Bejar bejar)
{
switch(bejar) {
case inorder: {if (elem != NULL)
{
    ++melyseg;
    kiir (elem->nullasGyermek (), os, bejar);
    for (int i = 0; i < melyseg; ++i)
        os << "---";
    os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
    kiir (elem->egyesGyermek (), os, bejar);
    --melyseg;
} break;
}
case postorder: {if (elem != NULL)
{
    ++melyseg;
    kiir (elem->nullasGyermek (), os, bejar);
    // ez a postorder bejáráshoz képest
    // 1-el nagyobb mélység, ezért -1
    kiir (elem->egyesGyermek (), os, bejar);
    for (int i = 0; i < melyseg; ++i)
        os << "---";
    os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
    --melyseg;
}break;
}

case preorder: {if (elem != NULL)
{
    ++melyseg;
    for (int i = 0; i < melyseg; ++i)
        os << "---";
    os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
    kiir (elem->nullasGyermek (), os, bejar);
    // ez a postorder bejáráshoz képest
    // 1-el nagyobb mélység, ezért -1

    kiir (elem->egyesGyermek (), os, bejar);
    --melyseg;
}break;
}
```

}

A kiir függvények maguk majd azonosak az alap binfához képest, csak attól függően mit adunk meg neki futtatáskor azt fogja a program meghívni.

A kiiratások post-, in- és preorder módon. A különféle kiiratások minden részfát egy-egy adott módon irat ki. A postorder mód először a bal, majd a jobb oldalt írja ki és végüll a gyökeret. Az inorder a "normál" kiiratás, azaz először a bal oldal, majd a gyökér és végül a jobb oldal. A preorder pedig először írja ki a gyökeret, majd a bal oldalt és legvégül a jobb oldalt.

Jelen esetben az imput fájl az alábbi szöveget tartalmazta: "LIDUSFHGÉKAJ"

6.4. Tag a gyökér

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

A C++ segítségével a C vel szemben, itt vannak már osztályok így a pointerek használatát ki tudjuk váltani velük.

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

class LZWBInFa
{
```

```
public:
    LZWBinFa () :fa (&gyoker)
    {
    }
    ~LZWBinFa ()
    {
        szabadit (gyoker.egyesGyermek ());
        szabadit (gyoker.nullasGyermek ());
    }
    void operator<< (char b)
    {
        if (b == '0')
        {
            if (!fa->nullasGyermek ())
            {
                Csomopont *uj = new Csomopont ('0');
                fa->ujNullasGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->nullasGyermek ();
            }
        }
        else
        {
            if (!fa->egyesGyermek ())
            {
                Csomopont *uj = new Csomopont ('1');
                fa->ujEgyesGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->egyesGyermek ();
            }
        }
    }
    void kiir (void)
    {
        melyseg = 0;
        kiir (&gyoker, std::cout);
    }
    int getMelyseg (void);
    double getAtlag (void);
    double getSzoras (void);
    friend std::ostream & operator<< (std::ostream & os, LZWBinFa & bf)
    {
        bf.kiir (os);
        return os;
    }
}
```

```
    }
    void kiir (std::ostream & os)
    {
        melyseg = 0;
        kiir (&gyoker, os);
    }
```

Az LZWbinfa osztályban a public részen, lényegében itt vizsgáljuk a fát, hogy a nullások és egyesek hogy helyezkednek. Itt megy végbe, na nincs gyerek létrehoz egyet és visszalép a gyökérre, egyébként rálép a megfelelő gyerekre. A kiir megkapja a gyökeret, elé írjuk a '&' jelet mert a class része.

```
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 (Csmopont * gy)
    {
        balNulla = gy;
    }
    void ujEgyesGyermek (Csmopont * gy)
    {
        jobbEgy = gy;
    }
    char getBetu () const
    {
        return betu;
    }

private:
    char betu;
    Csmopont *balNulla;
    Csmopont *jobbEgy;
    Csmopont (const Csmopont &);
    Csmopont & operator= (const Csmopont &);
};
```

A private rész ahogy neve is mondja, privát azaz az osztály többi 'tagja' nem fér hozza, ami a priváton belül van. Ezenn a privát részen belül van egy újabb osztály, a 'Csomopont' néven, ez ugyanúgy működik, mint az LZWbinfa, csak azon belül egy újabb class.

```
Csomopont *fa;
    int melyseg, atlagosszeg, atlagdb;
    double szorasosszeg;
LZWBinFa (const LZWBinFa &);
LZWBinFa & operator= (const LZWBinFa &);
void kiir (Csonopont * elem, std::ostream & os)
{
    if (elem != NULL)
    {
        ++melyseg;
        kiir (elem->egyesGyermek (), os);
        for (int i = 0; i < melyseg; ++i)
            os << "---";
        os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
        kiir (elem->>nullasGyermek (), os);
        --melyseg;
    }
}
void szabadit (Csonopont * elem)
{
    if (elem != NULL)
    {
        szabadit (elem->egyesGyermek ());
        szabadit (elem->>nullasGyermek ());
        delete elem;
    }
}

protected:
    Csonopont gyoker;
    int maxMelyseg;
    double atlag, szoras;

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

};
```

AZ LZWbinfába beírjuk a kiir és azabadit fügvényeket. Itt megfigyelhetjük, hogy a kiir azonos módon működik, mint a c-s verzióban, azaz kiír '---'-okat, mögéjük az elemeket, majd a mélységeket. Ahogy a c-ben is az alapértelmezett kiiratási mód inorder, azaz bal oldal először, majd a gyökér és utána következik a jobb oldal.

```
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;  
}
```

Itt vannak a melyseg, atlag és szórások, amik szontén benne voltak a c-ben, itt a lentebbi függvényeket hívják meg ezen függvények. Az alábbi függvények azonosak a c-s verzióval, ezeket hívják meg a fenti függvények. Így számolódnak ki a megfelelő értékek és kerülnek majd kiiratásra a kimenet végére.

```
void  
LZWBinFa::rmelyseg (Csomopont * elem)  
{  
    if (elem != NULL)  
    {  
        ++melyseg;  
        if (melyseg > maxMelyseg)  
            maxMelyseg = melyseg;  
        rmelyseg (elem->egyesGyermek ());  
        rmelyseg (elem->>nullasGyermek ());  
        --melyseg;  
    }  
}
```

```
void
LZWBinFa::ratlag (Csomopont * elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        ratlag (elem->egyesGyermek ());
        ratlag (elem->>nullasGyermek ());
        --melyseg;
        if (elem->egyesGyermek () == NULL && elem->>nullasGyermek () == NULL ↔
            )
        {
            ++atlagdb;
            atlagosszeg += melyseg;
        }
    }
}

void
LZWBinFa::rszoras (Csmopont * 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;
}
```

Van még itt egy usage függvény, ami, ha rosszul futtatjuk a programot kiírja nekünk, hogy hogy helyes. A használata egyébként megegyezik a c-s programmal.

```
int
main (int argc, char *argv[])
{
    if (argc != 4)
    {
        usage ();
        return -1;
```

```
}

char *inFile = *++argv;

if ((*((++argv) + 1) != 'o')
{
    usage ();
    return -2;
}
std::fstream beFile (inFile, std::ios_base::in);
if (!beFile)
{
    std::cout << inFile << " nem létezik..." << std::endl;
    usage ();
    return -3;
}

std::fstream kiFile (*++argv, std::ios_base::out);

unsigned char b;
LZWBinFa binFa;
while (beFile.read ((char *) &b, sizeof (unsigned char)))
    if (b == 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;

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

return 0;
}
```

A main elején egyből vizsgáljuk, hogy helyesen idítottuk-e el a programot, ha nem, meghívja az usage-t és az elmonja nekünk. A mainben megyünk végig a bemenetin és hivogatjuk meg a megfelelő fügvényeket. Végül kiirjuk a kimeneti fájlba magát a fát, aztán a mélységet, atlagot és a szórást. A legvégén bezárjuk a fájlokat és végezér a programunk.

Szintén itt van a futtatás illetve, amit a programunk létrehozozza, a fa, mélység illesmi.

```
david@david-S15:~/Asztal/prog/welch$ g++ binfa.cpp -o binfa
david@david-S15:~/Asztal/prog/welch$ ./binfa bif.a.txt -o kibi.txt
david@david-S15:~/Asztal/prog/welch$
```

```
-----1(2)
-----1(1)
-----0(2)
---/()
-----1(3)
-----1(2)
-----1(4)
-----0(5)
-----0(3)
---0(1)
-----1(3)
-----1(5)
-----0(4)
-----0(2)
-----0(3)
depth = 5
mean = 3.33333
var = 1.36626
```

6.5. Mutató a gyökér

Oláh Sándor segítségével(tutor)

Írd át az előző forrást, hogy a gyökér csomópont ne kompozícióban, csak aggregációban legyen a fával!

Kifejezetten kevés módosítást kell végezni

Korábban referenciaiként adtuk a függvényeknek, ez esetben pedig a gyökér egy pointer a classban. További egyéb hibák kiküszöbölése végett itt töröljük a gyökeret.

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

```
    delete gyoker;
}
```

Itt is át kell írni néhány dolgot, mivel a gyökérből pointer lett. Illetve fel kell szabadítanunk a memóriát. És nyilván pointerként hivatkozni a gyökérre.

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

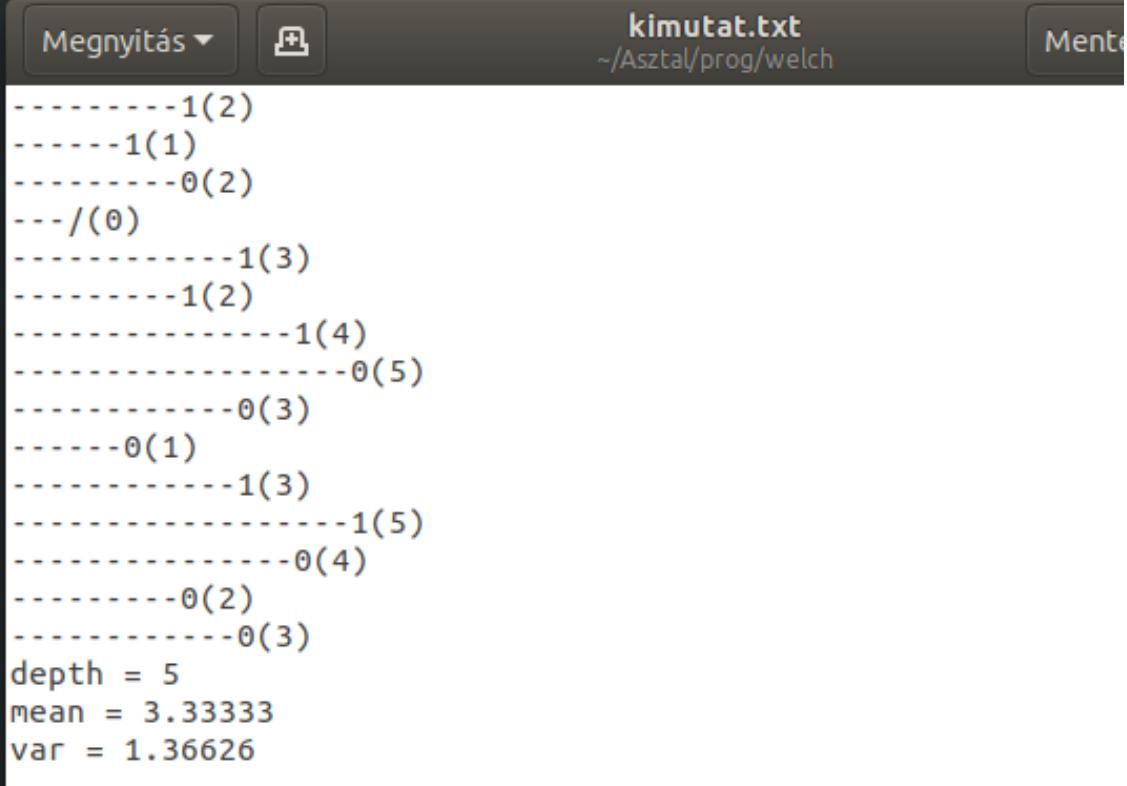
[...]

if (!fa->egyesGyermek ())
{
    Csomopont *uj = new Csomopont ('1');
    fa->ujEgyesGyermek (uj);
    fa = gyoker;
}

[...]

void kiir (void)
{
    melyseg = 0;
    kiir (gyoker, std::cout);}
```

```
david@david-S15:~/Asztal/prog/welch$ g++ mutato.cpp -o mutato
david@david-S15:~/Asztal/prog/welch$ ./mutato bifa.txt -o kimutat.txt
david@david-S15:~/Asztal/prog/welch$ 
```



```
kimutat.txt
~/Asztal/prog/welch
```

```
Megnyitás ▾
```

```
kimutat.txt
~/Asztal/prog/welch
```

```
Mente
```

```
-----1(2)
----1(1)
---0(2)
--/(0)
----1(3)
---1(2)
----1(4)
----0(5)
---0(3)
---0(1)
----1(3)
----1(5)
----0(4)
----0(2)
----0(3)
depth = 5
mean = 3.33333
var = 1.36626
```

6.6. Mozgató szemantika

Oláh Sándor segítségével(tutor)

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

```
~Binfa() {
    torol(gyoker);
}
Binfa (Binfa&& regi)
{
    gyoker = nullptr;
    *this =std::move(regi);
}
Binfa& operator= (Binfa&& regi)
{std::swap(gyoker, regi.gyoker);
    return *this;
}
```

Fentébb látható a mozgató konstruktur. Ez egy jobbértéket vár paraméternek(vagyis egy érték, amit majd egy adott változónak adni akarunk), az első sorában a gyökér pointert nullpointerre állítjuk. A 'regi' fára meghívjuk az 'std::move' fügvényt, ezzel jobbértéket kapunk a korábbi (paraméterként kapott) változóból. A 'Binfa' opreátor= ben (a ==-vel jelezzük az opreator overloading-ot) az std::swap-ot hívjuk meg ami értelelm szerűen felcserél két értéket, mégpedig a gyoker-et és a regi.gyoker-et. Így pedig mikor nullponterre állítjuk a gyökeret, akkor már a regi.gyoker a gyökér, mert a = jelnél meghívódott a korábban leírt operátor, ahol kicsérélődött a két gyökér valamint visszadata a gyoker változót, amiben a régi fa volt. A *this-be mozgattuk a régi fát.

```
kiFile << "Eredeti" << endl << binFa;

kiFile << "depth = " << binFa.getMelyseg () << std::endl;
kiFile << "mean = " << binFa.getAtlag () << std::endl;
kiFile << "var = " << binFa.getSzoras () << std::endl;
Binfa binFa1 = std::move(binFa);

kiFile << endl << "regi mozgatas utan:" << endl << endl << binFa;

kiFile << "depth = " << binFa1.getMelyseg () << std::endl;
kiFile << "mean = " << binFa1.getAtlag () << std::endl;
kiFile << "var = " << binFa1.getSzoras () << std::endl;

kiFile << endl << "Mozgatott fa:" << endl << binFa1;

kiFile << "depth = " << binFa1.getMelyseg () << std::endl;
kiFile << "mean = " << binFa1.getAtlag () << std::endl;
kiFile << "var = " << binFa1.getSzoras () << std::endl;

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

return 0;
}
```

A program további része azonos a korábbival. A kiiratásban azonban értelelm szerűen van változás, mégpedig először kiiratjuk az eredeti fát(illetve enneka melységét, átlagát, szórását). Ezt követően hívjuk meg a mozgatást és újabb kiiratásokat végezünk az eredeti, majd már a mozgatott fára.

```
Eredeti
-----0(4)
-----0(3)
-----0(5)
-----1(6)
-----1(4)
----0(2)
----0(4)
-----0(6)
-----1(5)
----1(3)
-----1(4)
---/(1)
-----0(3)
-----1(2)
-----1(3)
depth = 4
mean = 3.33333
var = 1.36626

regi mozgatas után:

depth = 0
mean = -nan
var = 0

Mozgatott fa:
-----0(4)
-----0(3)
-----0(5)
-----1(6)
-----1(4)
----0(2)
----0(4)
-----0(6)
-----1(5)
----1(3)
-----1(4)
---/(1)
-----0(3)
-----1(2)
-----1(3)
depth = 4
mean = 3.33333
var = 1.36626
```

```
david@david-S15: ~/Asztal/prog/welch$ Fájl Szerkesztés Nézet Keresés Terminál Súgó
david@david-S15:~/Asztal/prog/welch$ g++ LZWbinfamozgat.cpp -o mozog
david@david-S15:~/Asztal/prog/welch$ ./mozog bifa.txt .o kímezog.txt
david@david-S15:~/Asztal/prog/welch$ █
```

7. fejezet

Helló, Conway!

7.1. Hangyszimulációk

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

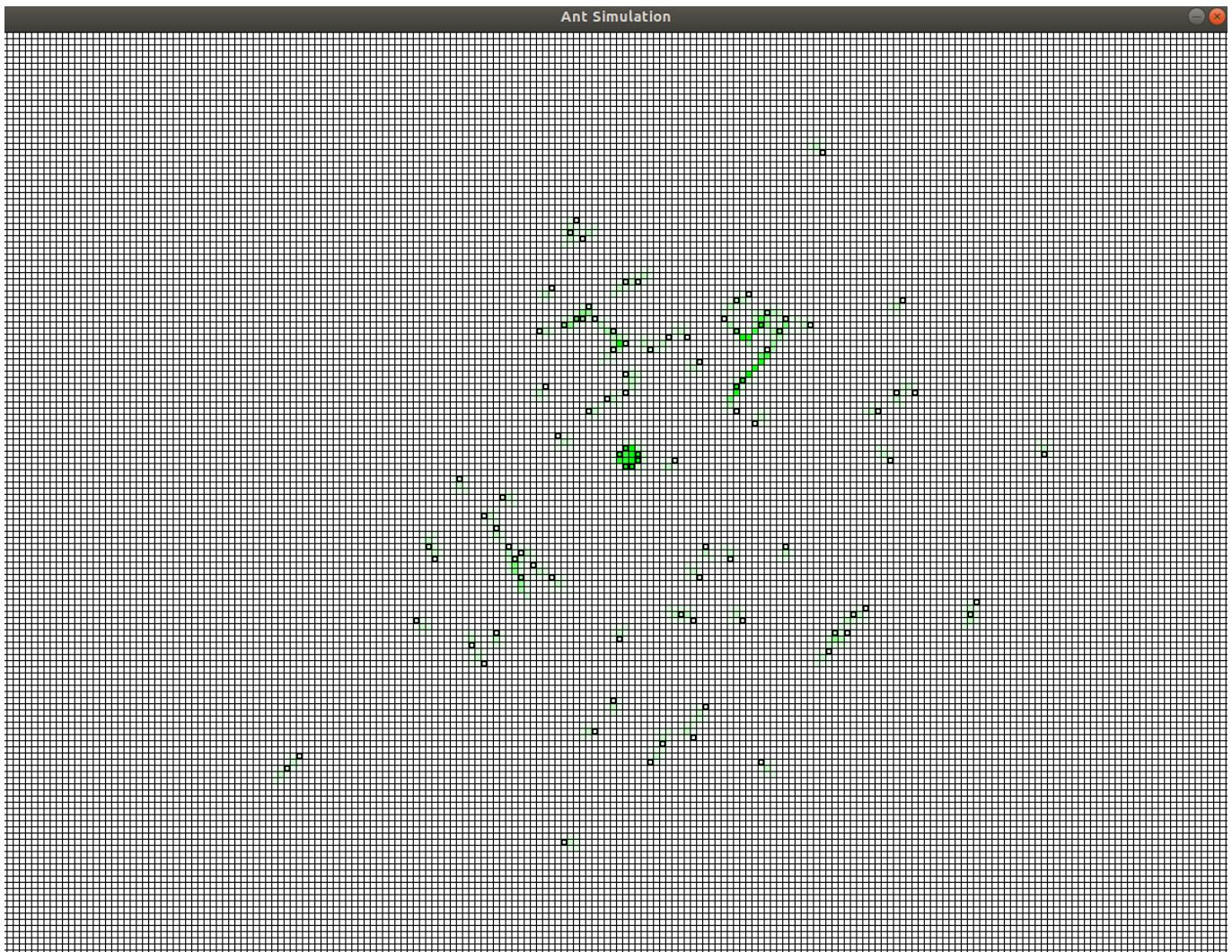
Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

források: <https://github.com/szuhi27/prog1/tree/master/hangyas>

Ezen program a hangyák viselkedését/mozgását hivatott szimulálni. Tudnáló ugyanis, hogy egy adott kajátoz a hangyán egy uton közlekednek, ami a legrövidebb illetve legbiztonágosabb, ezt a mozgás van a programba implementálva. Vagyis adott pontok közti legrövidebb, legjobb utakat figyelhetjük meg, mint a prédera menő hangya munkás sereg.

A programunkhoz van három header fájlunk, amiben eltároljuk a classokat. Ezek az 'Ant', 'AntThread' és az 'AntWin' osztályok. Ezekben vannak eltárolva a függvények és a változók, amiket a hangyáink használni fognak.

Egy kis módosítást végre kellet hajtani az egyik header file privát szekcióján (azt publikká tenni), de ezt megtéve kitűnően fut a programunk és a mellékelt képen láthatjuk, hogy szépen ftkosnak a hangyák.



7.2. Java életjáték

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

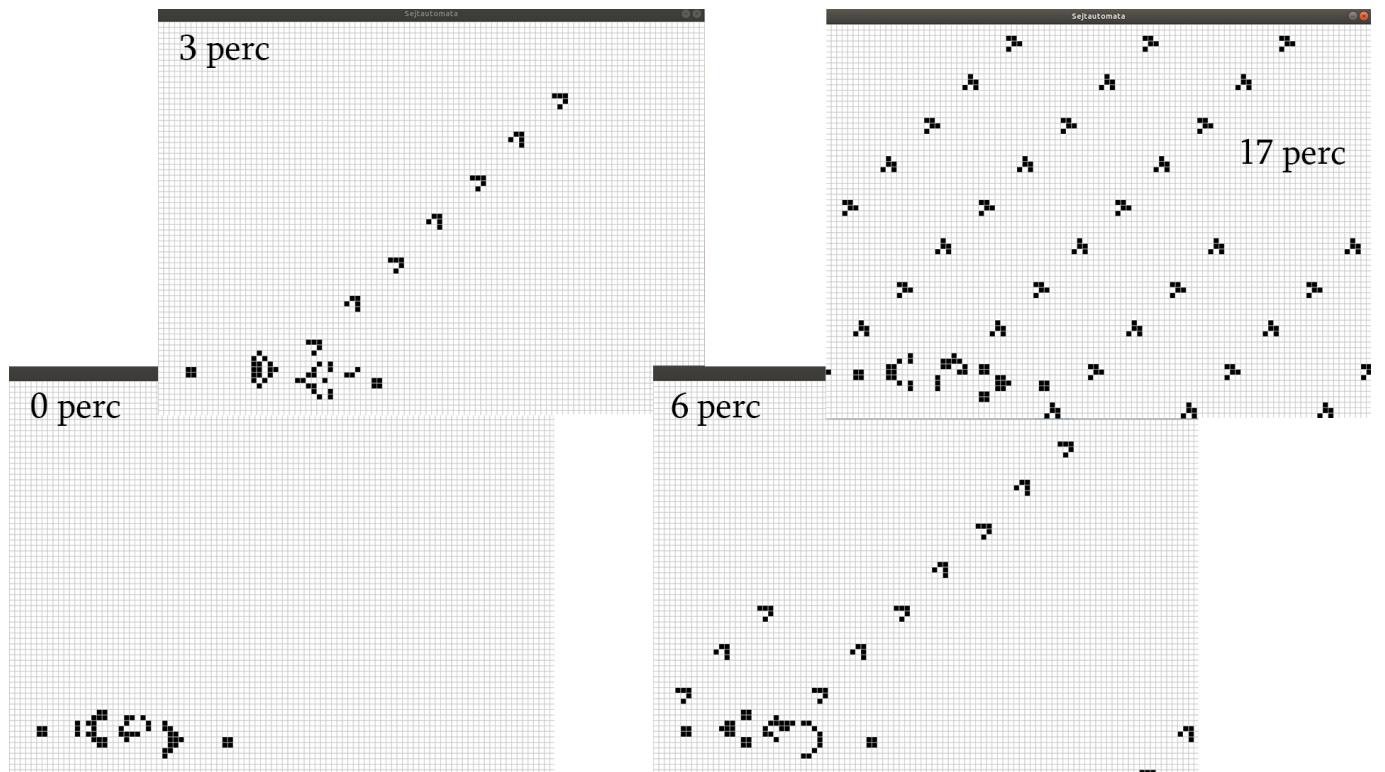
kód: <https://github.com/szuhi27/prog1/upload/master/javaelet>

A Conway-féle életjáték, Game of Life(GOF) egy cellás automata. A GOF univerzuma egy végtelen két-dimenziós, ortogonális négyzetrácsos háló, minden nyézetnek két állapota lehet, élő vagy halott. minden cella interakcióba lép az őt körülvevő nyolc másik cellával. Az idő elteltével(lépésenként, maik a generáció váltást jelképezik), a következő változások léphetnek életbe:

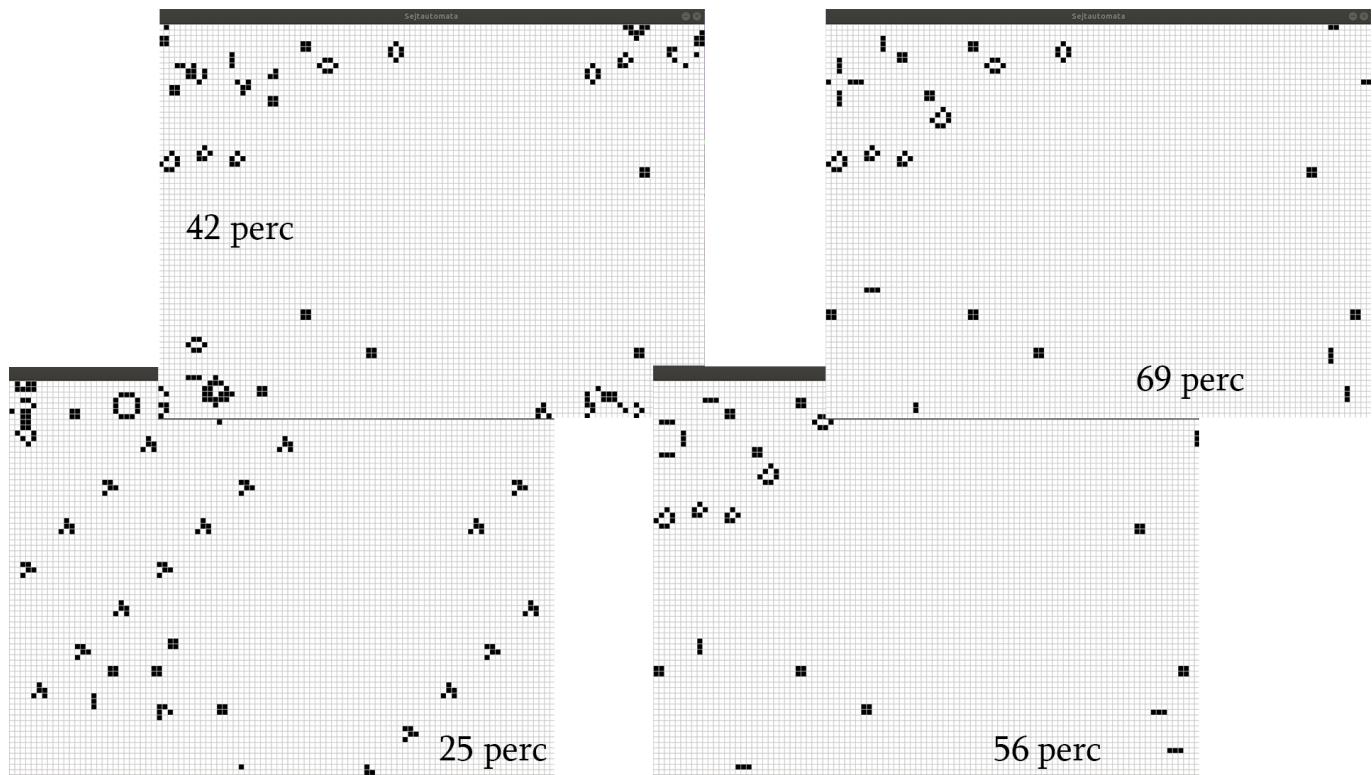
- Bármely élő cella, minek kettőnél kevesebb élő szomszédja van meghal (népességhiány állapota).
- Bármely élő cella kettő vagy három élő szomszéddal tovább él.
- Bármely élő cella három vagy több szomszéddal meghal (túlnépesedés).
- Bármely halott cella pontosan három szomszéddal egy élő cellává alakul(reprodukció).

Ezen szabályok alapján működik a programunk, melyet én közel hetven percig futtam az alábbi eredményeket figyelhetjük meg.

Az első négy képen egyfajta szabályosság figyelhető meg. A "generátorok" kis "csapatokat" indítanak útnak észak-keleti irányba, azok kifutnak a képből, majd újra megjelennek és ismét kifutnak és így tovább. Amíg ezek a "csapatok" nem ütköznek addig nincs baj, azonban amint ez megtörténik, összeomlik a rendszer, amit a következő négy kép mutat.



A 25. percben készített képen láthatjuk, hogy a "csapatok" beleütköztek a "generátorokba", így azok megsemmisültek. Így csak a "csapatok" tudtak tovább menni és nem indulnak újak. Az 56. percben készített képen már látható egyfajta végleges állapot, amiből a cellák nem tudnak kimozdulni, mert nem indulnak útra újak. Így két állapot között váltogatnak, a két állapot az 56. és a 69. perceb készített kép.



7.3. Qt C++ életjáték

Most Qt C++-ban!

kód: <https://github.com/szuhiz27/prog1/tree/master/qtelet>

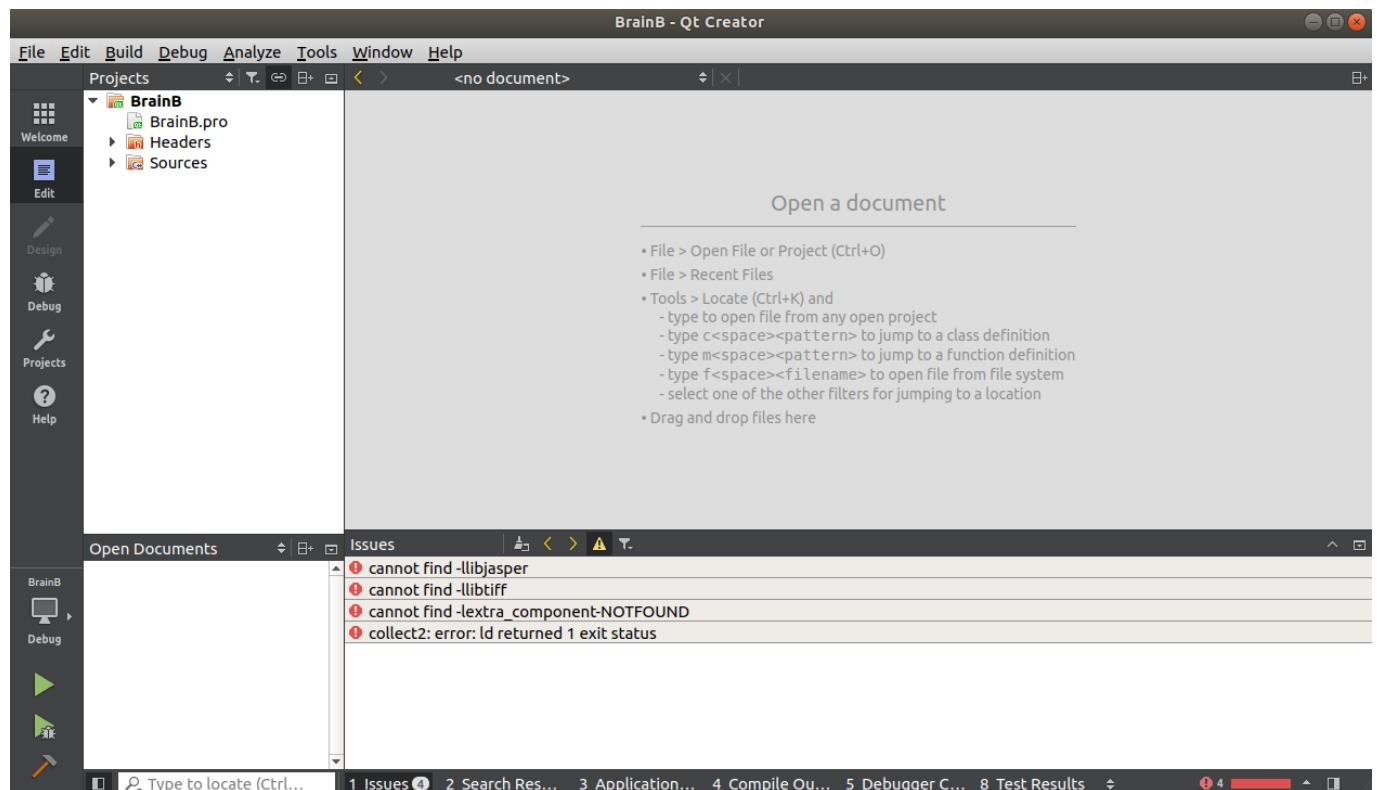
A feladat azonos az előzővel csak ezúttal c++-ban qt-val megvalósítva. A program működése és elmélete szintén azonos. Ugyan azon szabály szerint jönnek létre a sejtek a programban és ugyanúgy pusztulnak el.



7.4. BrainB Benchmark

A program lényege, hogy egy adott pontot kövessük a képernyőn, miközben újabb pontok jellenek meg, hogy megzavarjanak minket, ha elveszítjük a pontunkat, a többi elkezd eltűnni. Célunk, hogy minél tovább tudjuk követni a kis pontunkat, amit samunak hívnak. Ezen programhoz ismételten a qt kell használnunk.

Futtatni sajnos nem sikerült, az alábbi hibát küldte a qt. A neten se találtam megoldást a hibára, segítséget kértem másoktól, sajnos ők se tudtak benne segíteni. A qt-t többször újratelepítettem, rengeteg 'bővítményt' töltöttem le hozzá, de nem sikerült a hibákat eltávolítani a programból.



8. fejezet

Helló, Chaitin!

8.1. Gimp Scheme Script-fu: króm effekt

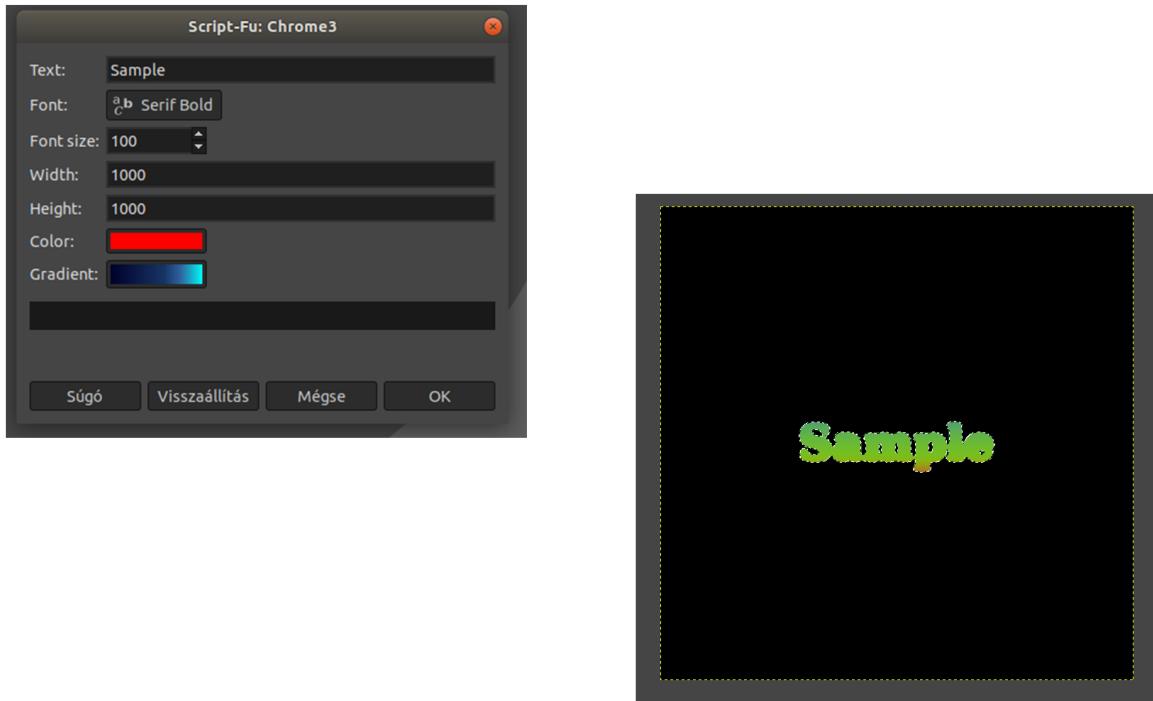
Írj olyan script-fu kiterjesztést a GIMP programhoz, amely megvalósítja a króm effektet egy bemenő szövegre!

Megoldás videó: https://youtu.be/OKdAkI_c7Sc

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

A feladat megoldásához szükségünk lesz a Gimp képszerkesztő program legalább 2.10-es verziójára. Ezt könnyedén beszerezhetjük (ubuntu rendszeren), az 'Ubuntu szoftverek'-ből. Ha letöltöttük, le kell mellé szedni a fenti linken található kódot (scriptet), majd azt bemásolni az alábbi helyre: [/home/{user id}/snap/gimp/165/.comfig/GIMP/2.10/scripts].

Ha ezt megtettük futtatjuk a Gimpet, majd a Fájl/Létrehozás menüpontnál kiválasztjuk a "BHAX" fület, majd a beillesztett scriptet. Azt futtatván megnyílik az alább látható kis ablak, ott kiválasztjuk a mekkunk tetsző paramétereket, majd az 'OK' gombra kattintva létrehozzuk a szövegünkeet. Jelent esetben ez a szöveg a "Sample", ezt aztán ki tudjuk exportálni png képként és szabadon felhasználhatjuk bárhol.



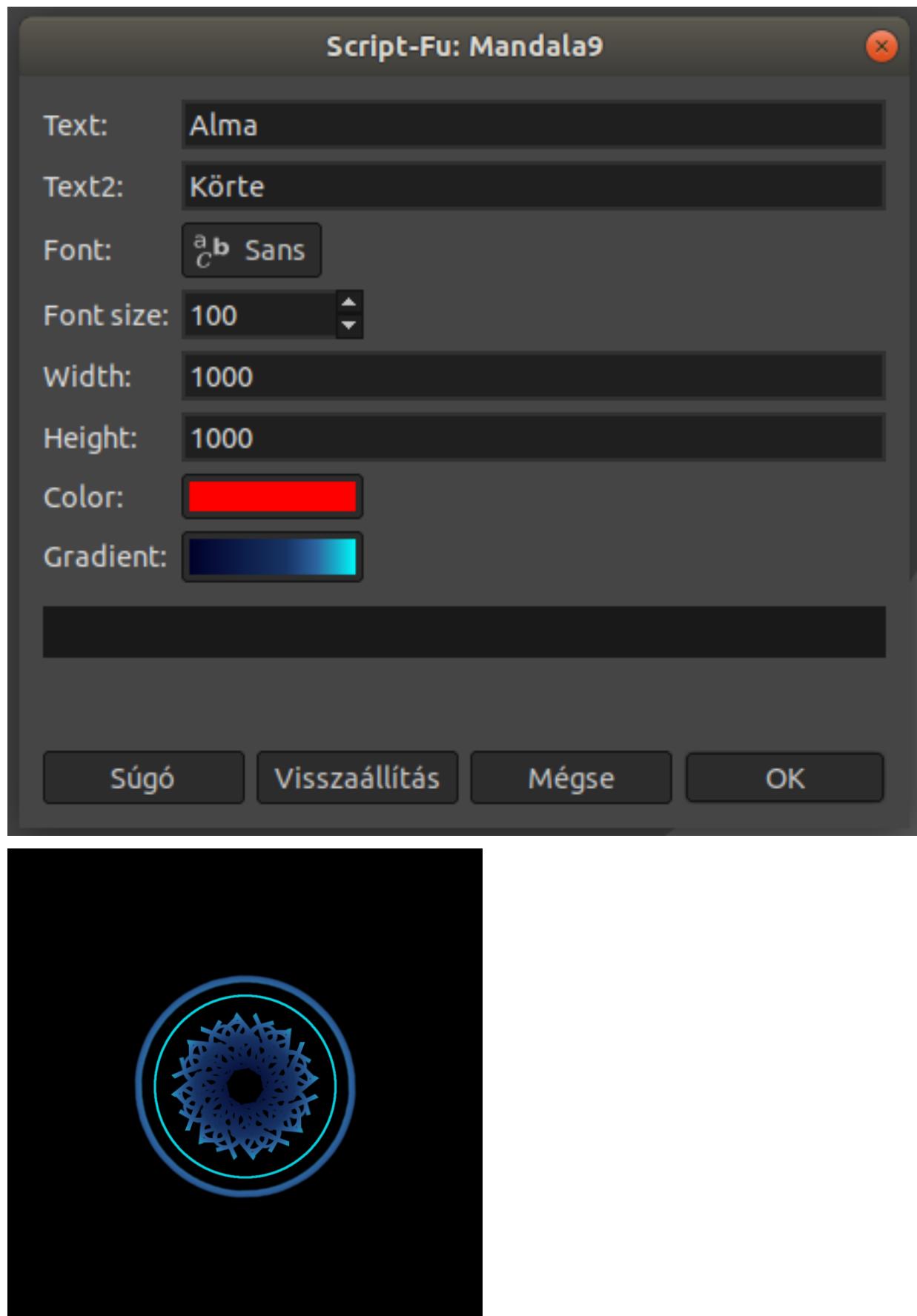
8.2. Gimp Scheme Script-fu: név mandala

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely név-mandalát készít a bemenő szövegből!

Megoldás video: 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

A fealdatunkat megoldani az előzőhöz hasonlóan tudjuk. Letölthjük a kódot, majd az bemásoljuk a megfelelő helyre. Majd a korábban ismertetett módon futtatjuk a scriptet, kitölthjük a felugró ablakot a megfelelő paraméterekkel, majd nyomunk egy ok-t. Ekkor kapunk egy mandalát, amit kimenthetünk a Gimp-ből és pl ide lentsebb be is illeszthetjük.



Itt az "Alma" szövegből készített nekünk egy mandalát, ha jól megnézzük a képen látszanak a bezűk. De ha egy hosszabb szöveget adunk a scriptnek, pl.: "Futballpálya", az már nagyobb is lesz és jobban észrevehők rajta a betűk.



9. fejezet

Helló, Gutenberg!

9.1. Programozási alapfogalmak

[?]

A proramozási nyelvek 3 szinttel léteznek, assembly, magas szintű és gépi nyelv. MAgas nyelven írt programokat nevezünk forráskódoknak, ezeket különféle fordítóprogik segítségével fordítjuk le és utánna futtathatóvá válnak. Ezen fordítások során a kód gépi nyelvvé alakul.

Minen különféle programozási nyelvnek más-más szemantikája van, azaz a nyelvek más szabályok szerint működnek, épülnek fel. Ebből ered, hogy jelenleg a kódok nem hordozhatók, azaz ha azt egy adott nyelven írjuk meg, akkor egy attól különböző nyelv nem tudja értelmezni, és más más nyelvet kell használni más más platformokhoz. Ezen a problémán jeleleg is dolgoznak, hogy kialakuljon egy platformfüggetlen nyelv. A nyelveket osztályokba osztjuk, imperatív, objektumorientált, logikai.

Az adattípusoknak két fő csoportja van, skalár vagy struktúrált. Az egész típusúkat fixpontosan, míg a valóaskat lebegőpontosan ábrázoljuk. Ezenk numerikus típusok, avagyis számok, ezen feül vannak a karakter típusuk is, logikai, speciális és sorszámozott típusok. Ezek egyszerűek, természetesen vannak összetett típusok is, mint például a tömb vagy a rekord. A tömb elemei lehetnek különféle egyszerű típusok. Létezik még mutató típus, mely címek elérését mutatja. A deklarációknak is több fajtája van, vagyis, amikor a különféle típusú változókat veszünk fel a programba, ezek: explicit, implicit és automatikus.

A kifejezések vagyis műveletek a programozásban háromféleképp definiálhatók, prefix (oerátor-operandus), infix(operand.-operát.-operand.) és postfix(operand.-operát.). AZ infix nem egyértelmű, itt a műveleti sorrend nem egyértelmű, azaz ezt zárójelezni kell a helyes érték kapása érdekében. A való életben mi infix módon írjuk le a műveleteket, például a matematikában, de nekünk is kell zárójelekt használni az egyértelműség érdekében. A logikai műveleteknél a nyelv érzékelheti automatikusan az eredményeket, vagyis ha van egy hosszú logikai művelet, azonban egy és a fő összekötő es annak egyik oldalán hamis áll, akkor már nem szükséges a másik oldalt kiszámolni, mert a művelet mindenki által hamis lesz.

Az utasítások két fő csoporttal büszkélkednek, deklarációs és végrehajtó. Ezek szerkezete nyelvekben eltérhet egymástól, de az egyes fajtáknak a működése ettől független megegyezik. Az utasításokat általában mi adjuk a programnak, hogy mit is végezzen pontosa. VAnnak a ciklusos utasítások, itt van előtesztelő és utótesztelő, előbbi először megnézi a feltételt, ha teljesül végrehajtódik a benne levő utasítás. Az utóbbi először végrehajta a bele írt utasítást(okat), majd aztán megnézi, hogy teljesül-e a feltétel, ha igen, akkor újra fut.

Ezen ciklusoknak megmondhatjuk hogy pontosan hányszor fussenak, le ezt általában feltételként adjuk meg nekik. Itt van még a végtelen ciklus, amivel a könyv elején találkoztunk, ennek a tulajdonsága, hogy amint ebbe belelépünk, ha csak egy belső feltétellel nem állítjuk le a ciklus a végtelenségig fut. Leállítani egy végtelent egy vezérlő utasítással, a breank-al például tudjuk.

A programok szerkezete egységekre bontható, mint pl az alprogram, blokk, csomag és task. AZ alprogram igazán egyértelű, maga is egy kisebb program, ami a nagyon belül létezik és rá hivatkozunk és az végrehajtódi. Néhány nyelvben lehetőség van ezen alprogramok fejen kívüli meghívására is, ezt másodlagos belépései pontnak nevezzük. Ha ezen módon hívjuk meg akkor a programtörzs csak adott része fut le, nem az egész.

Az alprogram és egyéb prog.egységek közti komunikáció egy formája a paraméteradás, természetesen ennek is több fajtája van, nevesítve, érték-, eredmény-, érték-eredmény-, cím-, név- és szöveg szereinti paraméteradás. A blokk három egységből áll, kezdet, törzs és vég(, mint egy jó szöveg). A törzsön belül különféle utasítások foglalhatnak helyet. Felmerült még a hatáskör vagyis láthatóság fogalma, aminek kezelésére két fajta van, a dinamikus és statikus..

Az I/O vagyis imput/output téren jellenek meg a jeletős eltérések egyes programnyelvek között. Az I/O felelős a különféle perifériák közötti kommunikációjáért. Ez a rendzser állományokkal dolgozik, ezezekhez mindenhoz hozzá rendelhetjük az egyik vagy akár minden két betűt. Az állományok használatához először deklarálnunk kell azokat a használt nyelv szerint.

9.2. Programozás bevezetés

[**KERNIGHANRITCHIE**]

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

A C programozási nyelvben nincsen sok adattípus, de ezekhez rekhatalunk előszavakat, vagyis, amivel pontosítunk a jelentésen. Az alap adattípusok plédául a float, int, double vagy char és a korábban említett előszavak mint például, hogy short vagy long, amiket pl az int elé rakhatunk. Ezzel kibővíthetjük jelentését, mégpedig a short előjellel az int 16, míg long előjellel 32 bitet foglal. Real típusnál a longgal például a pontosságot növelhetjük. Ilyen előjeleket nem csak változók elé rakhatunk, hanem állandók elé is, és hasonlatos eredményt érhetünk el velük, mint a váltózóknál.

A C nyelvben a változók elnevezésére is van pár szabály, miszerint a változók neveinek első karaktere betű kell legyen. Valamint a nevek nem lehetnek a nyelv "kulcsszavai", vagyis pl. nem nevezhetünk el egy változót "if" néven, mert azt a nyelv alapértelmezetten használja. Az egyes változók köszt lehetőség van átalakításra, kapcsolatra köztük, például intet double ba alakítani, de más esetekben már nem feltétlen jöhetszre írni.

Ezért kell figyelnünk a deklarációról vagy a függvényekben, hogy azok milyen típust adnak vissza. Fontos, hogy c-ben (mint egyébként sok másik nyelvben is) az utasítások végére, mint mpéldául egy változó deklarálása pontosvesszót kell tenni, ezzel lezárva azt. HA ezt elhalasztjuk a forítás hibával fog visszatérni.

Az is és az else, gyakran használt formula. Az if-ben megadunk egy feltételt, ami ha teljesül, akkor az azt követő utasítás hajtódi végre, ezt kiegészíthetjük egy ezt követő else-el. Ekkor ha az if nem teljesül, akkor mindenkiéppen lefut az else-t követő utasítás. Ha nem ezt nem akarjuk, akkor használhatjuk az "else if" formát, ekkor ha az első if nem teljesül, akkor átlép ez esle-re eztán vizsgálja a következő if-et és ha ez teljesül akkor lefut az utasítás. Ha egyszerre több utasítást is végre akarunk hajtattni egy if-et követően, akkor használunk kell "{}" karaktereket, ekkor a két zárójel közé írhatunk több utasítást is.

Az előbb vázolt szituáció egyirányú, de létezik többirányú utasítás is. Ebbe esik a "switch", egy switchen belü adhatunk meg úgynevezett "case"-eket, ekkor amit megadunk egy case nek, akkor, ha az teljesül vagyis az az eset jön létre, akkor az azon belüli utasítás hajtódiik végre. És it hagyhatunk egy defaultot, ami ha egyis csae se teljesül, akkor rölegen valami, ami mindenkiépp fut.

A következő alanyunk a while és a for. A while után adunk egy feltételt és amíg az igaz addig ismételgeti meg az azon belüli utasítást. A for ciklusban pedig adunk egy induló pontot, majd egy vég pontot és egy utasítást, plédául "for(i=0, i<3, ++i)", ha ezt megnézzük, akkor az i-t nulláról indítjuk, majd minden futásnál növeljük az i-t eggyel, ezt addig csináljuk, amíg az i értéke kisebb mint 3. Ezeket használhatjuk végtelen ciklus létrehozására is, ami szintén előfordult a feladatok között.

A do-while egy úgynevezett hátróltesztelő ciklus vagyis azon belül először végrehajtódiik az utasítás, majd eztán vizsgálja meg, hogy teljesül-e a felétel. Ezt addig csinálja, amíg nem teljesül az adott feltétel.

9.3. Programozás

[BMECPP]

Ezen könyv az előbbihez képest már nem a c-vel, hanem a c++-az foglalkozik. Ez a nyelv fő jellemzője az objektumorientáltság. Ez a nyelv nehéznek van tartva, azonban a c++-ban sok lehetőség van és sok dolgot ezzel jobban meg lehet oldani, például játékfejlesztésben népszerű nyelv. Az objektumorientáltság segít abban, hogy a programunk átláthatóbb legyen, ez például jól jön, csoportos projekthez használjuk.

Az adatrejtés dolgában bizonyos részeit elérhetetlenné tesszük, erre a private szavat használhatjuk, így amit a privaton belül írunk az úgymond ott is marad. HA viszont public-nak adjuk, akkor értelelem szerűen publikus lesz, vagyis mindenkinél elérhető. Az osztájok tagjai apaból privátok, míg a struktúráké publikok.

A friend kulcsszavas függvények speciális esetet képeznek, mert ami a frienden belül van az a privát részhez is hozzáfér, nem csak a publikokhoz.

A bool típus a c++ ban benne van, míg a c-ben nincs, a bool két értéket vehet fel, true vagy false, vagyis igaz vagy hamis. C-ben ez int típusban volt jelen, a 0 vagy 1 értékekkel. A c++-ban a deklarálást a felhasználása előtt szoktuk megoldani, vagyis pl egy függvény előtt vagy azon belül. Nem pedig például a program elején.

A cppben az I/O is másiképp működik, például a kiíratás még a cben a printf függvényt kellett használnunk, cpp-ben elés a "<<" ">>" karaktereket használni, kiíratáshoz és beolvasáshoz. Ehhez azonban szükségünk van az 'iostream' állományra, mit a program tetején inculdeolhatjuk hozzá és márás működik a kiíratásunk.

Valamint cpp ben a változók típusainak átalakítása is bővült a c blei egy-ről itt már 4 esetben jöhet létre. Hbakezelésre van a try catch, ami nevénből adódó a hibákat kapja el, ezen keresztül vihetünk ki hibaüzeneteket is ha szügséges. Természetesen a c-ben használt függvények itt is jene vannak és ugyanúgy működnek.

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.

10. fejezet

Helló, Berners-Lee!

10.1. C++ vs Java

A C++-t Bjarne Stroustrup "készítette", a nyelv tulajdonképp a C nyelv továbbá a Simula67 egy kombinált, továbbfejlesztett változata. A nyelv vezette be az osztály és az objektum fogalmát. A java egy objektum orientált nyelv, ami a c++-hoz hasonló (de ahoz nem kompatibilis) szintaxist használ.

Osztály és fájl név közti kapcsolat. C++-ban nincs kikötés, nem számít, hogy hogy nevezzük el a fájlunk, attól bármi lehet az osztályok neve. Ezzel szemben javában az osztály nevének meg kell egyezni a fájl nevével, például ha van egy "Helo" nevű osztályunk, akkor a fájl nevének Helo.java-nak kell lennie, különben hibás lesz.

I/O. C++ egyszerű megvalósítást használ, ha be akarunk kérni valamit a "cin>>[vmi];"-t míg ha ki akarunk valamit iratni a "cout<<[vmi];"-t használjuk. Meglehetősen egyszerű módszer, és egyértelmű. A kiírás a javánál is hasonlóképp egyszerű "System.out.println([vmi]);", azonban a bekérés kissé bonyolultabb, ugyanis a java egyszerre csak egy bájtot olvas be.

A c++ csak a compilert támogatja, míg a Java az interpretert is támogatja.

Más nyelvekkel való kompatibilitás. A C++ kompatibilis C kódokkal, néhány kivételtől eltekintve. A Java bár C "inspirált" nyelv sem a C sem C++ kóddal nem kompatibilis.

Programozási módok. C++, eljárással valamint objektum orientált programozást is támogatja, míg a Java az utóbbiit.

Memóriakezelés. C++ a programozó tudja kezelni, figyelni kell a memória folyásra, a folyamatokat, ha végeznek le kell állítani és törlni kell a memóriából. Java a rendszer kezeli a memóriát.

Root Hierarchia. A java "single root hierarchy"-val működik, ez a legtöbb objektum orientált nyelvre jellemző, azaz minden osztály valamilyen módon származik egy adott gyökértől. Bár a C++ részben objektum orientált, mégsem efféle hierarchia szerint működik.

Többszörös öröklés. A C++-ban van többszörös öröklés, azaz egy objektum vagy osztály tulajdonságai származhatnak több, mint egy szülő objektumtól, osztálytól. A származtatott osztályok a származás sorrendjében is hívódnak meg. Ezen esetben merült fel az úgynevezetű a program futása alatt hibába "gyémánt probléma", azaz amikor egy osztály két szuperosztályának közös az alap osztálya. Erre a megoldás egy virtuális kulcsszó alkalmazása. A Javában efféle származás, nincs, ott a származás csak egy bizonyos objektumtól vagy osztálytól származhat.

Hatókör-felbontás operátor (::). A C++-ban a :: azza szolgál, hogy elérést nyújt egy globális változóhoz, egy adott egységen belül, ahol már létezik egy változó azon a néven. A Javában ez nem található meg, mert itt a method definíciók az osztályon kell megtörténniük.

Try and cathc. A Java kivételkezelésénél jelenik meg. Alap esetben, ha java kódban elkövetünk egy hívát, vagy felüti a fejét egy hiba, akkor a program leáll és egy kidob egy hibát. Ha a try and catch metódust használjuk a try-al kitudunk jelölni egy kódcsipetet, amit vizsgálunk, amíg a fut a program. A catch-el pedig, futtatni tudunk egy kódrészletet, ha a hiba üti fel a fejét a try-os részletben.

Overloading. Javában megtalálható a method túlterhelés (overloading), ez megengedi, hogy egy osztályban több mint egy feladatnak legyen azonos neve, ha az argumentum listájuk különbözik. Ám C++-ban e mellett megtalálható az operátor túlterhelés is, vagyis hogy általunk definiált osztályokban működjenek. Tehát speciális jelentést tudunk adni egy operátornak adott adattípusra, ezt híjuk operátor túlterhelésnek. Például, a '+' operátort túl tudjuk terhelni egy osztályban, hogy két stringet egyesíteni tudunk vele.

Platform. A c++ platformfüggő, azaz a kódot le kell compilálni adott platformra. Ezzel szemben a java platformfüggetlen, a programokat JVM-en írjuk (Java Virtual Machine), így nem kel újra kompileálni ha platformot váltunk.

Pointer. A c++ ban teljes támogatás van rájuk, még a javában nem igazán támogatott. A pointerek adott memómória címekre mutatnak, így a program a memóriából olvassa be a változó értékeit. Egy pointert úgy tudunk deklarálni, ha a változó neve elé helyezünk egy '*'-ot, természetesen előtte meg kell adni, hogy azon a címen milyen típusú adatot találunk. Ide kapcsolódik a '&' jel is, ami ha egy változó elé rakjuk, akkor megmondja, hogy az a változó hol van, vagyis a címét a memóriában.

Struktúrák. A struktúra hasonló az osztályhoz, ez csak a c-ben található meg, javában nincs. Itt is elnevezhetjük kedvünk szerint a struktúrát és abba elemeket tárolhatunk, nem azonos adattípusokkal. A struktúra egy felhasználó által definiált adattípus, ebbe helyezhetünk akár különböző típusú adatokat. Egy struktúra létrehozása:

```
struct név{  
    adat1;  
    adat2;  
    ...  
    adatn;  
};
```

Union,, szintén csak c-ben van. Ezzel egy memóriacímre tudunk rendelni több különböző adattípust, de egy időben csak egynek lehet valós értéke. Tehát ez hasonló a struktúrához de egyszerre csak egy értéket tud tárolni.

10.2. Bevezetés a mobilprogramozásba

35-51:

2.4: Kezdés képpen a Symbian OS-ről olvashatunk, ami egy mára mondhatni kihalt mobilknál alkalmazott operációs rendzser. Az os C++ ra épült, de annak egy korai verziójára, eképp több fejlesztés még nem található meg benne (2008). A rendszer UI-kat használt a készülékekken (S60, UIQ), ezeken keresztül készíthetünk programokat a készülékekre.

2.5: Windows mobile, a Microsoft mobilokra szánt operációs rendszere, ez a .NET keretrendzsert használja programozási oldalról. Ám ez az oprendszer sem akkor se manapság nem fedi le a piac jelentős részét.

2.6: Maemo, Linuxra épülő operációsrendszer, sok manapság alap eszközt tartalmazott, mint pl a Bluetooth, GPS. A fejlesztői környezet is Linux alapú, a rendszer fejlesztése is Linuxon zajlott. Az Android kihívója volt a maga idejében, de tudjuk, hogy "nem ő nyerte a harcot".

2.7: Az Android is egy Linux alapokra épülő nyílt forráskódú rendszer, mely a Google szárnyai alatt képzült, sikerét is köszönheti ennek. A Google szolgáltatásai alap tartozékai a rendszernek. Az alapvető programozási nyelve Java, de ma már más nyelveken is írhatunk rá programokat, más kisegítő programok segítségével (pl C#-Xamarin).

2.8: iPhone OS, vagy iOS, a manapság létező egyik legnagyobb mobil operációs rendszer, az Apple terméke, és akár a PC-s verzió ez is Unix alapokra épül. A rendszer sokkal kötöttebb, mint az Android, sikere az Apple innovatív megoldásainak köszönhető.

Python:

1990-ben alkotódott meg, magas szintű, dinamikus, objektum orientált és platform független programozási nyelv. Egy könnyen és gyorsan tanulható nyelv, főképp egyszerű problémák megoldására, de bonyolultabbak is megoldhatók vele. Symbian S60 platformjára készült Python S60 implementációja.

A C, C++, Java nyelvekkel ellentétben a Python (Py) nem szükség fordítani, elegendő csupán az értelmezőnek a forrást megandi, az futtatja is az alkalmazást. A nyelv platformok széles választékára elérhető, főkkép prototípusok készítésére érdemes alkalmazni.

A Py egy nagyon magas szintű programozási nyelv. A kódok rövidebbek, mint más nyelveknél, mert itt nincs szükség nitó/záró jelekre, valamint változó/argumentum definiálására.

A nylev szintaxisa behúzás alapó, azaz nincs szükség pl. "{ }" jelekre, mint C++-ban, hanem a külön részeket behúzások jelölik. Azaz "space" vagy "shift", fontos, hogy ezt egségesen használjuk. Továbbá adott utasítás a sor végéig tart, ezáltal nincs szükség pontosvessző használatára, ha mégis több sort szeretnénk adni egy utasításnak azt a sor végi "\"- el jelezhetjük.

Típusok; Py-ben minden adatot objektumok reprezentálnak, azok típusa határozza meg az adaton végezhető műveleteket. A Py kitalálja a változók típusát, értékük alapján. Az adattípusok lehetnek számok, stringek, ennesek, listák, szótárak.

Változók az egyes objektumokra mutatú referenciai. Léteznek lokális és globális változók is.

11. fejezet

Helló, Arroway!

11.1. OO szemlélet

A módosított polártranszformációs normális generátor beprogramozása Java nyelven. Mutassunk rá, hogy a mi természetes saját megoldásunk (az algoritmus egyszerre két normálist állít elő, kell egy példánytag, amely a nem visszaadottat tárolja és egy logikai tag, hogy van-e tárolt vagy futtatni kell az algot.) és az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua.!

Ugyanezt írjuk meg C++ nyelven is!

A programok során "generálunk" 10 számot, de ezt matematikai műveletek segítségével végezzük el. Egy futás alatt 2 szám készül, viszont az egyiket eltároljuk.

A feladat Java megvalósítása:

```
public class polar {  
  
    boolean nincsTárolt = true;  
    double tárolt;  
  
    public polar() {  
  
        nincsTárolt = true;  
  
    }  
  
    public double következő() {  
  
        if(nincsTárolt) {  
  
            double u1, u2, v1, v2, w;  
            do {  
                u1 = Math.random();  
                u2 = Math.random();  
  
                v1 = 2*u1 - 1;
```

```
v2 = 2*u2 - 1;

w = v1*v1 + v2*v2;

} while(w > 1);

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

tárolt = r*v2;
nincsTárolt = !nincsTárolt;

return r*v1;

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

public static void main(String[] args) {

    polar g = new polar();

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

}
```

Egy boolean típusú változóban rögzítjük, hogy van-e már tárolt számunk, és egy double típusúban pedig az eltárolt számokat. A következő fügvényen készítjük a számokat. A mainben, pedig adig hozunk létre számokat, mígy 10 nem lesz, ezeket írjuk ki a kimenetre.

A kimeneten az alábbi jelenik meg:

```
-1.1528050422003104
-0.20329761893903386
1.4226106126036109
2.169456853683861
-0.3198825156658214
0.45254984083693306
0.986461222838455
-1.800794753564929
0.06661740405329619
0.7626316658495963
```

Majd alább a C++ megvalósítás:

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

```
#include <ctime>

class PolarGen{
public:
    PolarGen();
    double next();
    ~PolarGen() {}
private:
    bool nincsTarolt;
    double tarolt;
};

PolarGen::PolarGen() {
    nincsTarolt = true;
    std::srand (std::time(NULL));
}

double PolarGen::next () {
    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;
    }
}

int main(){
    PolarGen rnd;
    for (int i = 0; i < 10; ++i)
        std::cout << rnd.next() << std::endl;
}
```

A C++ verzióban is természetesen ugyanaz történik, mint javában, csak itt C++ környezetben.

A kimenet:

```
david@david-S15:~/Asztal/prog/welch$ ./"polár"
1.50101
-0.390099
```

```
-1.03249  
-0.838285  
0.435722  
1.07054  
-0.282963  
-0.0108148  
-0.704234  
1.46202
```

11.2. Gagyí

Az ismert formális 2 „`while (x <= t && x >= t && t != x);`” tesztkérdéstípusra adj a szokásosnál (miszerint x , t az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciaja) „mélyebb” választ, írj Java példaprogramot mely egyszer végtelen ciklus, más x , t értékekkel meg nem! A példát építsd a JDK Integer.java forrására, hogy a 128-nál inkluzív objektum példányokat poolozza!

Tehát létrehozunk egy programot, amiben 2 integernek adunk értéket és azokat tesztejük le a megadott while ciklusban.

```
class gagyi{  
    public static void main(String[] args){  
        Integer t = 456;  
        Integer x = 456;  
        while(x <= t && x>=t && t != x)  
            System.out.println("igaz");  
    }  
}
```

Ezt a progit ha futtatjuk egy végtelen ciklust kapunk, vagyis a while feltétele igaznak bizonyul és mivel az értékek sehol sem változnak, az mindig igaz is marad. A while-on belül minden összehasonlításnak igaz kell lenni, hogy a while igaz legyen. Az első két feltétel a matematika szabályai miatt lesz igaz, mert $456=456$. A harmadik, tehát, hogy " $t \neq x$ ", azért lesz igaz, mert t és x kölön kölön objektumok, különbözik a memóriacímük, tehát ezért nem is egyenlőek. Így egy végtelen ciklus generálódik, amit a megszokott módon tudunk megszakítani.

De meg tudjuk csinálni, hogy ne legyen végtelen ciklus, vagyis hogy a " $t \neq x$ " feltétel hamis értéket adjon. Mégpedig úgy, hogy a számokat az integer osztály hatókörén belül adjuk meg, vagyis -128 és 127 között.

```
class gagyi{  
    public static void main(String[] args){  
        Integer t = 125;  
        Integer x = 125;  
        while(x <= t && x>=t && t != x)  
            System.out.println("igaz");  
    }  
}
```

Ekkor ahogy korábban kifejtettem ez esetben a while hamis lesz, vagyis a program gyakorlatilag azonnal megáll, vagyis végig megy, így nincsen végtelen ciklusunk.

11.3. Yoda

Írunk olyan Java programot, ami java.lang.NullPointerException-t leáll, ha nem követjük a Yoda conditions-t! https://en.wikipedia.org/wiki/Yoda_conditions

A Yoda condition névadója a Star Wars karaktere, aki a filmekben nyelvtanilag helytelenül beszél. Ez a helytelenség van általánosan a programozási nyelvbe. Mégpedig aképp, hogy egy feltételben felcseréljük a két oldalt, vagyis az érték kerül a bal, míg az aminek a viszonyát akarjuk tudni az értékhez képest kerül a jobb oldalra.

Normál esetben egy feltétel az alábbi módon néz ki:

```
if (price == 20) {} //Vagyis, ha a price egyenlő 20
```

A Yoda condition szerint pedig:

```
if (20 == price) {} //Vagyis, ha 20 egyenlő a price
```

De ez így nem jó, nem véletlenül nem használjuk ekképp. NullPointerException-t úgy tudunk előidézni, ha olyan objektum metódusát akarjuk használni, aminek értéke null.

```
class yoda
{
    public static void main (String[] args)
    {
        String st = null;

        try
        {
            if (st.equals("yoda"))
                System.out.print("Same");
            else
                System.out.print("Not Same");
        }
        catch(NullPointerException e)
        {
            System.out.print("NullPointerException Caught");
        }
    }
}
```

Output: NullPointerException Caught

Ebben a kis kódban láthatjuk, hogy megkapjuk a hibát. Az st stringnek null-t adunk értéklő, majd egy try-catch-ben teszteljük. Alapvetően egy egyenlőséget vizsgálunk, de mivel az st null, ezért ez a keresett hibát generálja, amit a catch-ben ki is iratunk.

12. fejezet

Helló, Liskov!

12.1. Liskov helyettesítés sértése

Írunk olyan OO, leforduló Java és C++ kódcsipetet, amely megséríti a Liskov elvet! Mutassunk rá a megoldásra: jobb OO tervezés.

(számos példa szerepel az elv megsértésére az UDPORG repóban, lásd pl.<https://sourceforge.net/p/udprog/code/ci/master/tree/source/binom/Batfai-Barki/madarak/>)

A Liskov elv, "S egy altípusa T-nek, akkor T típus objektumai kicserélhetők S típus objektumaival." Ha már rendelkezésünkre áll, haladjunk a fenti minta mentén. Tehát a madarakon keresztül fogjuk bemutatni, hogy megsérül a Liskov elv, mert a fügvényünk olyan helyen is meg fog hívódni, ahol nem kellene neki.

```
#include <iostream>

class Madar{
public:
    virtual void repul() {
        std::cout << "Repul\n" ;
    }
};

static void repulomadar(Madar& rep) {
    rep.repul();
}

class Sas : public Madar{
public:
    void repul() override{
        std::cout << "A Sas repul\n";
    }
};

class Pingvin : public Madar { };

int main(){
    Sas sasy;
```

```
Pingvin pingviny;
repulomadar(sasy);
repulomadar(pingviny);
return 0;
}
```

```
Kimenet:
david@david-S15:~/Asztal/prog2/liskov$ ./"madar"
A Sas repul
Repul
david@david-S15:~/Asztal/prog2/liskov$
```

Ezen kis c++ példa futtatása során a kimeneten kapjuk, mjad, hogy "A Sas repul", de a repülés fügvény a pingvinre is meghívódik, ami pedig a való életből tudjuk hogy a pingvin nem repül legalábbis a hétköznapi értelemben nem.

Ha ezt a kódot átültetjük Java nyelvre, nem meglepően azonos kimenetet kapunk (ha minden jól csináltunk).

```
class Madar{
    public void repul(){
        System.out.println("Repul\n");
    }
}

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

class Pingvin extends Madar{}

class Lmadar{
    public static void repulomadar(Madar rep) {
        rep.repul();
    }

    public static void main(String[] args) {
        Madar sasy = new Sas();
        Madar pingviny = new Pingvin();
        repulomadar(sasy);
        repulomadar(pingviny);
    }
}
```

Mivel ez nem megfelelően működik, ki kell javítani. Tehát a madarakból kivesszük a repülést és két új osztályt vezetünk be. Az új osztályok lesznek a Repülő valamint a NemRepül, a repülést pedig értelem szerűen elhelyezzük. Innentől minden madárhoz a tulajdonságainak megfelelően rendelünk hozzá osztályokat, ekképp a program már a Liskov elvnek megfelelően fog futni. Lent a módosított c++ kód helyezkedik.

```
#include <iostream>

class Madar{};

class Repulo : public Madar{
    public:
        virtual void repul() = 0;
};

class NemRepul : public Madar{};

static void repulomadar(Repulo& rep) {
    rep.repul();
}

class Sas : public Repulo{
    public:
        void repul() override{
            std::cout << "A Sas repul\n";
        }
};

class Pingvin : public NemRepul{};

int main(){
    Sas sasy;
    Pingvin pingviny;
    repulomadar(sasy);
    //repulomadar(pingviny);
    return 0;
}
```

Latható a két új osztály, amik segítségével már helyes a futás. A kód végén egy sor ki van kommentelve, ha az nem lenne ott, futási hiba lenne, mert olyan függvényteljesítőt hívjuk meg ami rá nem igaz. Ez is igazolja hogy helyes a működés.

```
Kimenet:
david@david-S15:~/Asztal/prog2/liskov$ ./"maradjo"
A Sas repul
david@david-S15:~/Asztal/prog2/liskov$
```

És természetesen ez is megvan Javában is, csak fontos, hogy nem classokat, hanem interfaceket hozunk létre, valamint a Madar-ból is az lesz.

```
interface Madar{};

interface Repulo extends Madar{
    public void repul();
}
```

```
interface NemRepul extends Madar{ }

class Sas implements Repulo{
    public void repul(){
        System.out.println("A Sas repul\n");
    }
}

class Pingvin implements NemRepul{}

class jo{
    public static void repulomadar(Repulo rep) {
        rep.repul();
    }

    public static void main(String[] args) {
        Sas sasy = new Sas();
        Pingvin pingviny = new Pingvin();
        repulomadar(sasy);
        //repulomadar(pingviny);
    }
}
```

12.2. Szülő-gyerek

Írunk Szülő-gyerek Java és C++ osztálydefiníciót, amelyben demonstrálni tudjuk, hogy az ősön keresztül csak az ős üzenetei küldhetőek!

Ahogy a feladat is írja azt kell megmutatnuk, hogy a szülő csak saját metódusait tudja használni, de a gyerekét nem. A gyereke pedig értelem szerűen a sajátját és szülejéét is. Én MSCode-ot használok a programok elkészítésekor, ha íyen módon járunk el, mikor a main-ben adjuk át a metódusokat a résztvevőknek, mikor a szülőnek akarjuk adni a gyerekét a program nem is ajánlja fel az a metódust, míg amikor az helyes fejalánlya azokat, amik elérhetők. Már ez is mutatja a helyes alkalmazást.

Nézzük a C++ kódot:

```
#include <iostream>
using namespace std;

class Szulo{
public:
    void szol() {
        std::cout<<"Gyerekem!\n";
    }
};

class Gyerek: public Szulo{
public:
    void szolketo () {
```

```
        std::cout<<"Szulom!\n";
    }
};

int main(){
    Szulo* szulo = new Szulo();
    Gyerek* gyerek = new Gyerek();
    szulo->szol();
    gyerek->szolketo();
    gyerek->szol();
    //szulo->szolketo();
    return 0;
}
```

Kimenet:

```
david@david-S15:~/Asztal/prog2/liskov/szulo$ ./"szulo"
Gyerekem!
Szulom!
Gyerekem!
david@david-S15:~/Asztal/prog2/liskov/szulo$
```

Egy egyszerű kódot használunk, de az tökéletesen teljesíti a feladatot. A Szülőnek a szol metódust adjuk, amiben "kimondja" hogy "Gyerekem", a gyerek pedig "Szulom"-et mond. A mainben lathatjuk hogy először a szülő kapja meg a sajátját, majd a gyerek a sajátját és a szülejéét. Ám amikor a szülőnek akarjuk adni a gyerekét és úgy akarjuk fordítani, hibát fogunk kapni, mert nem lehet kivitelezni azt az "átadást".

És természetesen ebből is van java verzió, szintén csak átirjuk amit át kell, de működésben azonos módon működik, mint a C++.

```
class Szulo{
    public void szol(){
        System.out.println("Gyerem!");
    }
}

class Gyerek extends Szulo{
    public void szolketo(){
        System.out.println("Szulom!");
    }
}

public class Atad{
    public static void main(String[] args){
        Szulo szul = new Szulo();
        Gyerek gyer = new Gyerek();
        szul.szol();
        gyer.szolketo();
        gyer.szol();
        //szul.szolketo();
    }
}
```

Természetesen a kimeneten is ugyanazt kapjuk, valamint, ha kivesszük a kikommentelt sort ugyanúgy hibát fogunk kapni.

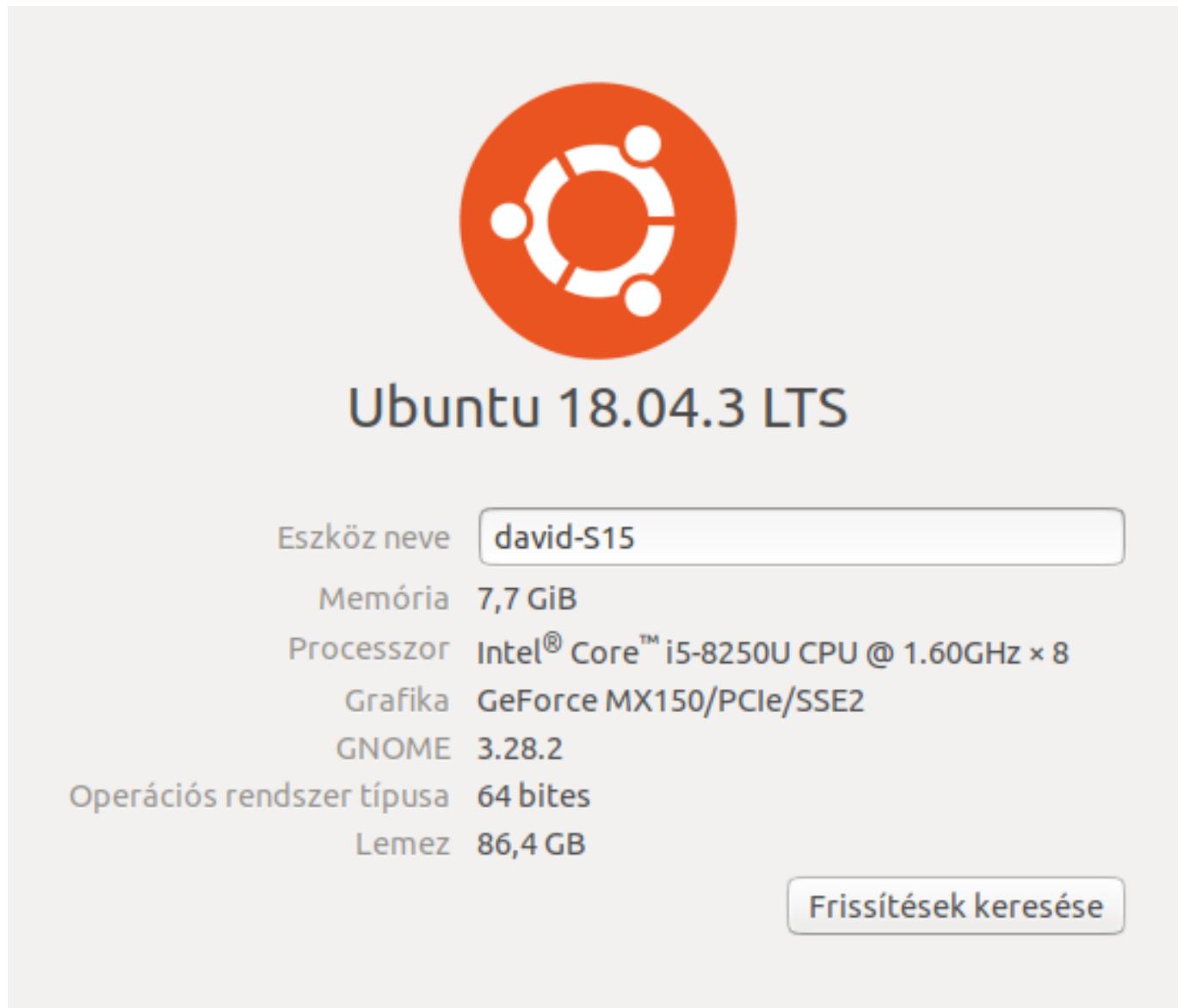
12.3. Anti OO

BPP algoritmussal a Pi hexadecimális kifejtésének a 0. pozíciótól számított 10 6, 107, 108 darab jegyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket!

A kódokat az alábbi linken: <https://github.com/szuhi27/prog2/tree/master/anti>

Futtatni egyszerűen lehet, miután le compileoltuk őket a "./BPP 6[7/8]"-at beírván fut a program. A futási időt úgy kapjuk meg, hogy az előbb említett szövet elé a terminálba beírjuk, hogy "time", ezáltal, mikor befejeződik a futás megkapjuk annak idejét. Ezeket a számokat láthatjuk majd lentebb.

A compileokat gcc (v7.4.0), g++(v7.4.0), javac(v11.0.4) és msc (mono v4.6.2.0) végzi. A programokat a laptopomon futtattam, melynek specifikációi az alábbi képen tekinthetők meg.



10^x	C	C++	C#	Java
10^6	1,723s	1,735s	1,573s	1,624s
10^7	20,002s	20,020s	18,109s	18,038s
10^8	3m 49,946s	3m 49,807s	3m 27,807s	3m 26,554s

12.1. táblázat.

Nézzük a számokat:

Mit mondunk a számok?

A két legjobb nyelvnek a C# és a Java bizonyult, míg a C és C++ minden futás során alul maradnak. A C és C++ szinte azonos eredményeket adnak, C-ben a 6 és 7 egy hajszállal gyorsabb, de a 8-as már a C++-nak tart egy tizeddel kevesebb ideig.

A két gyorsabb nyelv viszont jelentős előnnyel rendelkezik a 3. és 4. helyezetekkel szemben. A versenyt a Java nyeri 2-1-re azonban a C# nem sokkal marad le, ráadásul a 6-os futásban fél tizeddel gyorsabban bizonysul. Viszont a két hoszabb futásban már a Java győzedelmeskedik.

Azt megállapíthatjuk, hogy a futási eredmények nagyságrendekben megegyeznek a nyelvek között. Azt viszont észre kell venni, hogy ahogy haladunk előre a leggyorsabb és leglassabb idő közti különbség növekszik. Első futásra ~6,8%, másodikra ~10,9% és harmadikra ~11,3% a gyorsabb nyelv előnye.

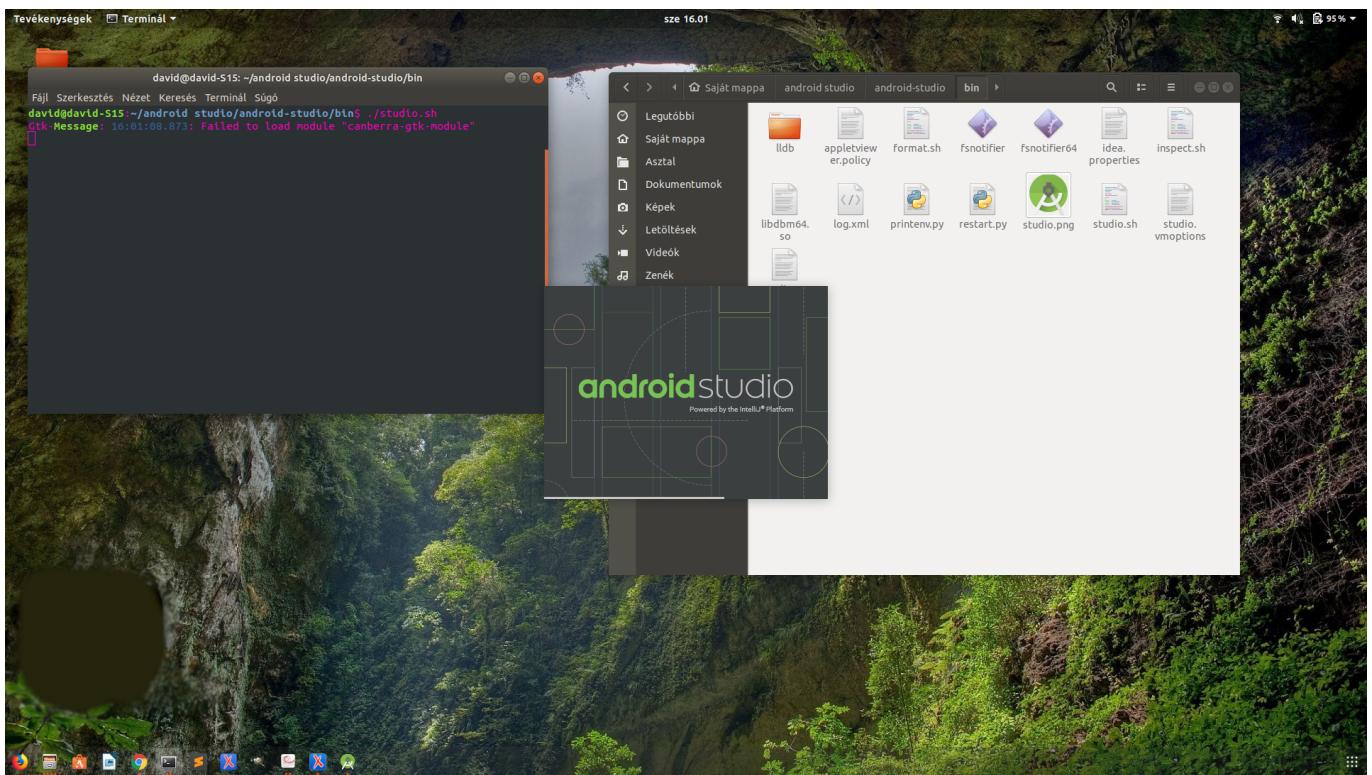
Természetesen ezek a mérések nem "professzionálisak", tehát ha úgy tekintjük nem adunk minden nyelvnek azonos terepet. Valamint a mért számok az első futásra adott eredmények, hogy még pontosabb eredményt kapunk több futás során vett átlagot kellene vennünk. De ettől függetlenül egy alapvető ötletet adhat nekünk a különböző nyelvekről.

12.4. Hello, Android!

Feladat: Élesszük fel az SMNIST for Humans projektet! Apró módosításokat eszközölj benne, pl. színvilág.

A project az alábbi linken található meg: <https://gitlab.com/nbatfai/smnist/tree/master/forHumans/SMNISTforHumansExp3/app/src/main>

Első dolgunk egy programot letölteni, amiben szerkeszteni illetve futtatni tudjuk majd. Jelen esetben az Android Studio lesz a választás. Azt az alábbi linkről le tudjuk tölteni: <https://developer.android.com/studio>, majd a letöltött fájlt egyszerűen csomagoljuk ki, majd a "bin" mappát a terminálba megnyitva csak futassuk a studio.sh-t és már indul is programunk.



Ha fut a programunk indításunk egy új "Empty Activity"-t. Először a projektben levő "layout" mappában levő xml-t másoljuk be az újonnan létrehozott projektünk "layout" mappájába. Ezt követően új projektünkben a "manifest" mappában levő "AndroidManifest.xml" fájl tartalmát írjuk át a következőre:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"    ↵
    package="com.example.forhumans">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".SMNISTE3Activity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Majd a "java/com.example.forhumans" mappát ürítsük ki, majd másoljuk be a 4 fájlt amit a gitlabos projektból a "[...]/main/java/[..]/smnistforhumansexp3" helyen találunk.

12.5. Ciklomatikus komplexitás

Számoljuk ki valamelyik programunk függvényeinek ciklomatikus komplexitását!

Feladatunk egyértelmű, a választott kódként viszonylag kézen van a prog1-es védési program az LZWBinafa. Annak a kódját fogjuk használni és kiszámolni a ciklomatikus komplexitását. Egy weboldal segítségével fogjuk a feladatot megoldani, a weboldal a "lizard.ws" lesz. Ez a program c++-ban van megírva tehát be kell állítanunk a megfelelő nylevet, majd analizálni.

Mi is a ciklomatikus komplexitás, a fogalom egy szoftvermetrika 1976-ból. Ez a metrika egy szoftver forráskódja alpján meghatározza annak komplexitását, amit egy számértékben jelenít meg. Ezen számítás gráfelméleten alapul. A számítás menete $M = E - N + 2P$, hol E a gráf élek száma, N a gráf csúcsainak száma és a P az összefüggő komponensek száma.

Tehát miután analizáltuk a kódot kapunk egy nagy adag adatot.

The screenshot shows a browser window with the URL www.lizard.ws. The page title is "Try Lizard In Your Browser". On the left, there is a code editor window containing C++ code. On the right, there is a detailed analysis report table.

Code analyzed successfully.

Function Name	NLOC	Complexity	Token #	Parameter #
LZWBinafa::LZWBinafa	3	1	11	
LZWBinafa::~LZWBinafa	5	1	23	
LZWBinafa::operator <<	29	4	108	
LZWBinafa::xir	5	1	23	
LZWBinafa::operator <<	5	1	28	
LZWBinafa::xir	5	1	27	
LZWBinafa::Csomopont::Csomopont	3	1	24	
LZWBinafa::Csomopont::~Csomopont	3	1	5	
LZWBinafa::Csomopont::nullasGyermekek	4	1	9	
LZWBinafa::Csomopont::egyesGyermekek	4	1	9	
LZWBinafa::Csomopont::ujNullasGyermekek	4	1	12	
LZWBinafa::Csomopont::ujEgyesGyermekek	4	1	12	
LZWBinafa::Csomopont::getBetu	4	1	9	
LZWBinafa::xir	38	10	296	
LZWBinafa::szabadt	9	2	37	
LZWBinafa::getMelyseg	6	1	25	
LZWBinafa::getAttag	7	1	36	
LZWBinafa::getsZoras	12	2	68	
LZWBinafa::rmelyseg	12	3	52	
LZWBinafa::rattag	15	4	69	
LZWBinafa::rszoras	15	4	81	
usage	4	1	18	
main	68	16	385	

A második oszlopból láthatjuk egyes folyamatok komplexitását, egyértelmű, hogy a leg komplexebb feladat a main függvény, ott történik a legtöbb meghívás.

13. fejezet

Helló, Mandelbrot!

13.1. Reverse engineering UML osztálydiagram

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.2. Forward engineering UML osztálydiagram

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.3. Egy esettan

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.4. BPMN

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.5. TeX UML

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14. fejezet

Helló, Chomsky!

14.1. Encoding

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.2. OOCWC lexer

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.3. Full screen

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.4. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.5. Paszigráfia Rapszódia LuaLaTeX vizualizáció

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.6. Perceptron osztály

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15. fejezet

Helló, Stroustrup!

15.1. JDK osztályok

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.2. Másoló-mozgató szemantika

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.3. Hibásan implementált RSA törés

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.4. Változó argumentumszámú ctor

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.5. Összefoglaló

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

XInclude: bhax-textbook-motto.xml file not found

16. fejezet

Helló, Arroway!

16.1. FUTURE tevékenység editor

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.2. OOCWC Boost ASIO hálózatkezelése

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.3. SamuCam

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.4. BrainB

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.5. OSM térképre rajzolása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17. fejezet

Helló, Schwarzenegger!

17.1. Port scan

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.2. AOP

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.3. Android Játék

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.4. Junit teszt

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18. fejezet

Helló, Calvin!

18.1. MNIST

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.2. Deep MNIST

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.3. CIFAR-10

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.4. Android telefonra a TF objektum detektálója

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.5. SMNIST for Machines

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.6. Minecraft MALMO-s példa

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

18.7. Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

18.8. C

[KERNIGHANRITCHIE] Kernighan, Brian W. & Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

18.9. C++

[BMECPP] Benedek, Zoltán & Levendovszky, Tihamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

18.10. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, 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.