

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Dr. Bátfai Norbert

Copyright © 2019 Fürjes-Benke Péter

Copyright (C) 2019, Norbert Bátfai Ph.D., batfai.norbert@inf.unideb.hu, nbatfai@gmail.com, Fürjes-Benke Péter, fupn26@hotmail.hu

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>

DRAFT

COLLABORATORS

	<i>TITLE :</i>		
	Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert ÁCs Fürjes-Benke, Péter	2019. május 7.	

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	2019-03-01	Turing feladatsor kidolgozva	fupn26
0.0.6	2019-03-08	Chomsky feladatsor kidolgozva	fupn26
0.0.6	2019-03-15	Cesar feladatsor kidolgozva	fupn26

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.7	2019-03-22	Mandelbrot feladatsor kidolgozva	fupn26
0.0.8	2019-03-29	Welch feladatsor kidolgozva	fupn26
0.0.9	2019-03-30	Kernighan olvasónapló hozzáadva	fupn26
0.1.0	2019-04-05	Conway feladatsor kidolgozva	fupn26
0.1.1	2019-04-08	Olvasónapló K&R fejezete frissítve	fupn26
0.1.2	2019-04-10	Olvasónapló Pici könyv fejezete frissítve	fupn26
0.1.3	2019-04-13	Schwarzenegger feladatsor kidolgozva	fupn26
0.1.4	2019-04-15	Olvasónapló K&R és BME fejezete frissítve	fupn26
0.1.5	2019-04-16	Olvasónapló Pici fejezete frissítve	fupn26
0.1.6	2019-04-17	Olvasónapló BME fejezete frissítve	fupn26
0.1.7	2019-04-19	Olvasónapló BME fejezete frissítve	fupn26
0.1.8	2019-04-20	Olvasónapló Pici fejezete frissítve	fupn26
0.1.9	2019-04-21	Olvasónapló BME fejezete frissítve	fupn26
0.2.0	2019-04-25	Olvasónapló BME fejezete frissítve	fupn26

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.2.1	2019-04-26	Olvasónapló Pici fejezete frissítve	fupn26
0.2.2	2019-04-30	Chaitin fejezet kidolgozva	fupn26
0.2.3	2019-05-02	Mandelbrot fejezet Java zoom része kiegészítve	fupn26
0.2.4	2019-05-03	Mandelbrot fejezet Qt zoom része kiegészítve	fupn26
0.2.5	2019-05-03	Schwarzenegger fejezet Mély MNIST része passzolva	fupn26
0.2.6	2019-05-04	Conway ÉLetjátékának Java része kiegészítve	fupn26
0.2.7	2019-05-04	Hibák javítása	fupn26
0.2.8	2019-05-04	Szoftmax feladat kiegészítve	fupn26
0.2.9	2019-05-04	BrainB passzolva	fupn26
0.3.0	2019-05-04	Conway Qt siklókilövő kiegészítve	fupn26
0.3.1	2019-05-07	Háromszögmátrix és Mozgató konstruktör feladat javítva	fupn26

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]

DRAFT

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	9
2.4. Labdapattogás	10
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	12
2.6. Helló, Google!	14
2.7. 100 éves a Brun téTEL	16
2.8. A Monty Hall probléma	18
3. Helló, Chomsky!	21
3.1. Decimálisból unárisba átváltó Turing gép	21
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	22
3.3. Hivatalos nyelv	24
3.4. Saját lexikális elemző	26
3.5. l33t.l	28
3.6. A források olvasása	30
3.7. Logikus	33
3.8. Deklaráció	34

4. Helló, Caesar!	39
4.1. double ** háromszögmátrix	39
4.2. C EXOR titkosító	45
4.3. Java EXOR titkosító	49
4.4. C EXOR törő	52
4.5. Neurális OR, AND és EXOR kapu	56
4.6. Hiba-visszaterjesztéses perceptron	61
5. Helló, Mandelbrot!	67
5.1. A Mandelbrot halmaz	67
5.2. A Mandelbrot halmaz a std::complex osztállyal	72
5.3. Biomorfok	75
5.4. A Mandelbrot halmaz CUDA megvalósítása	80
5.5. Mandelbrot nagyító és utazó C++ nyelven	87
5.6. Mandelbrot nagyító és utazó Java nyelven	103
6. Helló, Welch!	117
6.1. Első osztályom	117
6.2. LZW	120
6.3. Fabejárás	126
6.4. Tag a gyökér	127
6.5. Mutató a gyökér	136
6.6. Mozgató szemantika	137
7. Helló, Conway!	140
7.1. Hangyszimulációk	140
7.2. Java életjáték	147
7.3. Qt C++ életjáték	155
7.4. BrainB Benchmark	163
8. Helló, Schwarzenegger!	170
8.1. Szoftmax Py MNIST	170
8.2. Mély MNIST	176
8.3. Minecraft-MALMÖ	176

9. Helló, Chaitin!	184
9.1. Iteratív és rekurzív faktoriális Lisp-ben	184
9.2. Gimp Scheme Script-fu: króm effekt	186
9.3. Gimp Scheme Script-fu: név mandala	195
10. Helló, Kernighan!	203
10.1. Pici könyv	203
10.2. K&R könyv	210
10.3. BME C++ könyv	213
III. Második felvonás	225
11. Helló, Arroway!	227
11.1. A BPP algoritmus Java megvalósítása	227
11.2. Java osztályok a Pi-ben	227
IV. Irodalomjegyzék	228
11.3. Általános	229
11.4. C	229
11.5. C++	229
11.6. Lisp	229

Ábrák jegyzéke

2.1. Megállási probléma	9
2.2. PageRank	15
2.3. Brun-tétel	18
3.1. Decimálisból unárisba	21
3.2. 1. Splint kép	26
3.3. 1. Splint kép	32
3.4. 2. Splint kép	33
4.1. Háromszögmátrix	40
4.2. Pointerek a memóriában	41
4.3. $(*(tm + 3))[1] = 43$	44
4.4. $*(tm + 3)[1] = 43$	45
4.5. Titkosítandó szöveg	47
4.6. Fordítás és futtatás	48
4.7. Titkosított szöveg	49
4.8. Titkosított szöveg	51
4.9. Törés	55
4.10. Törés	56
4.11. Neuron	57
4.12. OR	58
4.13. AND	59
4.14. EXOR első próba	60
4.15. EXOR második próba	61
4.16. Perceptron bemenet	62
4.17. mandelpng.cpp fordítása és futtatása	63
4.18. Mandelbrot-halmaz	64

4.19. Fordítás és futtatás	66
5.1. Mandelbrot halmaz	68
5.2. Program fordítás, futtatása	70
5.3. Mandelbrot halmaz	71
5.4. Program fordítása és futtatása	74
5.5. Mandelbrot halmaz	75
5.6. Program fordítása és futtatása	78
5.7. Biomorf	79
5.8. Indexelés	82
5.9. Két program hasonlítása	84
5.10. CUDA variáns	85
5.11. C++ megvalósítás, párhuzamosítás nélkül	86
5.12. 1. lépés	88
5.13. 2. lépés	88
5.14. 3. lépés	89
5.15. 4. lépés	89
5.16. Alapállapot	90
5.17. Nagyítva	91
5.18. Tovább nagyítva	92
5.19.	104
5.20.	105
5.21.	106
5.22.	107
6.1. LZW algoritmus	121
7.1. Kapcsolók nélkül	141
7.2. Kapcsolókkal	142
7.3. UML diagram	143
7.4. Siklókilövő	154
7.5. Siklókilövő	163
7.6. Program	164
7.7. Eredmény	165
8.1. Tutorial_3	180

9.1. Rekurzió	186
9.2. Alap	190
9.3. Blur effect	191
9.4. Clamp	192
9.5. Színetmenet alkalmazása	193
9.6. Végeredmény	195
9.7. Alap mandala	200
9.8. Végső mandala	202

DRAFT

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

DRAFT

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 csemegeznek 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

DRAFT

**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.

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Végtelen ciklusnak olyan ciklusokat hívunk, melyek soha nem érnek véget, általában valamilyen logikai hiba miatt. Pl:

Program pelda

```
{  
int main()  
{  
    int i = 0;  
    while (i<=0)  
    {  
        i = i-1;  
    }  
}
```

Ebben jól láthatod, hogy a while cílusban lévő feltétel folyamatosan teljesül, theát a program végtelen ciklusba kerül. Viszont, ha kifejezetten végtelen ciklust szeretnél írni, ennek a legelegánsabb módja a következő:

Program vegtelen.c

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

Ez a végtelen ciklus csak egy magot dolgoztat, de azt 100%. A lényege annyi, hogy a for ciklus nem kap semmilyen argumentumot, ennek következtében a ciklus előtti teszt folyamatosan igazat fog adni, tehát a ciklus nem fejeződik be.

De nem elégünk meg az egy maggal, hiszen ma már a legtöbb számítógép legalább 4 maggal rendelkezik. Tehát találni kell egy megoldást, hogy az összes mag dolgozzon 100%-on. Ezt oldja meg az OpenMP.(Bővebben [itt](#) olvashatsz erről.) A lényege annyi, hogy program több szálón dolgozhat, így kihasználva a rendelkezésre álló erőforrásokat. Ráadásul ez könnyen implementálható az előző kódunkba:

```
Program vegtelen_all.c
{
    int main()
    {
        #pragma omp parallel
        for(;;);
    }
}
```

Amint látod, csak a #pragma omp parallel sort kellett hozzáadni. Természetesen ezt bármelyik programnál használhatod, sőt javasolt is, köszönhetően a hardverek gyors fejlődésének.

Még egy dologgal adós vagyok. Már láttad, hogy hogyan lehet megoldani, hogy egy végtelen ciklus 100%-ban használjon egy magot, majd azt is, hogyan használjon 100%-ban a processzort. Itt az idő megnézni, hogyan lehet elérni, hogy egy végtelen ciklus egyáltalán ne használjon (vagyis nagyon keveset) a CPU által biztosított erőforrásból. Ehhez a sleep függvényre lesz szükségünk. A kód a következő:

```
Program vegtelen_s.c
{
    int main()
    {
        for(;;)
        {
            sleep(1);
        }
    }
}
```

A sleep függvény lényegében minden meghívásánál "aludni" küldi azt a szálat, amit program használja, jelen esetben 1 másodpercig altatja.

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

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

Megoldás videó:

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

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

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

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

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

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

}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

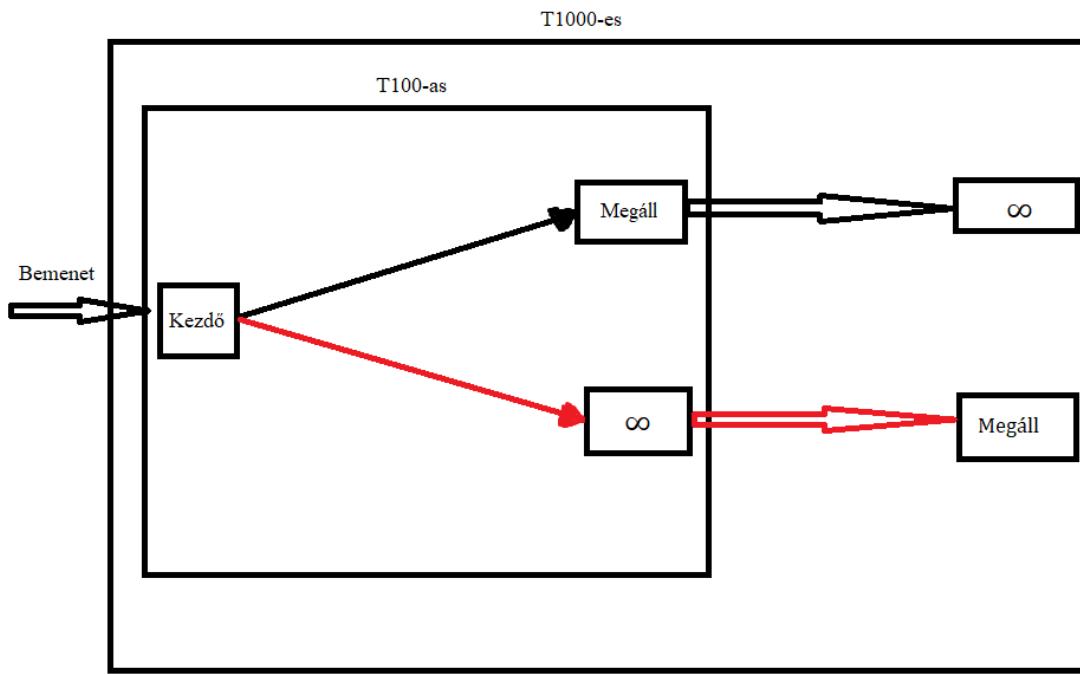
- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true
- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

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

Tanulságok, tapasztalatok, magyarázat...

Alan Turing a XX.század egyik legjelentősebb brit matematikusa, a modern számítógép-tudomány atya. Az ő nevéhez fűződik a Turing-gép, mint fogalom, ezt 1936-ban dolgozta ki. De egy másik fontos eredménye is van, hiszen az ő segítségével sikerült a szövetségeseknek feltörni a Náci Németország titkosító berendezését, az Enigmát. Enélkül talán soha nem sikerült volna véget vetni a II. világháborúnak, de az biztos, hogy sokkal több emberveszteséggel járt volna. Ezt a történetet dolgozza a fel a [Kódjátszma](#) című film.

De visszatérve a Turing-géphez, ez egy 3 főbb fizikai egységből áll: egy cellákra osztott papírszalagból, egy vezérlőegységből és egy író-olvasó fejből. A működése nagyjából abból áll, hogy az író-olvasó fej a szalagon beolvass egy cellát, módosítja, majd tovább mozog. Ez folytatódik minden iterációban. Ennek két-féle kimenetele lehet, vagy megáll a "program", vagy végtelen ciklusba kerül. Egy másik fontos fogalom az Univerzális Turing-gép, melynek lényege az, hogy egy bemenetre ugyan azt az eredményt adja mindegyik Turing-gépen. Ezzel eljutottunk a megállási problémához, mely mind a mai napig megoldhatatlan probléma elő állította a számítógép-tudományt. A megállási probléma azt mondja ki, hogy nem tudunk olyan programot írni, amely meg tudja mondani egy másikról, hogy az le fog-e fagyni, azaz végtelen ciklusba kerül-e, vagy sem. Ez az ábra szemlélteti a forrásban leírtakat:



2.1. ábra. Megállási probléma

Tehát a T100-as program kap egy programot bemenetként, és arról eldönti, hogy az megáll-e vagy sem. Ebben az esetben a bemeneti programot úgy kell elképzelni, mint egy szöveges fájlt. A program beolvasása és eldönti, hogy van-e benne végtelen ciklus. A T100-as visszaad egy igaz/hamis értéket. Ezt átadjuk a T1000-es programnak, mely ha a bemenet igaz, akkor megáll, ha az érték hamis, akkor pedig végtelen ciklusba kerül. Itt azt hihetnén, hogy minden rendben, hiszen a program működik, de mi történik akkor, ha a T1000-es program bemenete önmaga. Ha úgy érzékeli, hogy nincs önmagában végtelen ciklus, akkor végtelenciklusba kerül, ha pedig van benne végtelen ciklus, akkor pedig megáll. Itt jól láthatod az ellentmondást. Ez az oka annak, hogy mind a mai napig nem tudott senki ilyen programot létrehozni.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ha valaki két változónak az értékét fel akarja cserélni, akkor a legegyszerűbb megoldásnak egy segédváltozó bevezetése tűnhet. De természetesen ennél sokkal kifinomultabb eszközök is vannak erre a célra. Az egyik ilyen megoldás, hogy valamelyen matematikai műveletet használunk. Egyik megoldás az, hogy a két

változó értékét összeadjuk, majd ebből az összegből kivonjuk a változók régi értékét úgy, hogy a értékük felcserélődjön. Tehát:

```
int a = 4;
int b = 5;
a = a+b;
b = a-b;
a = a-b;
```

Ez szín tiszta matematika, viszont ennél egy sokkal érdekesebb dolg ugyan ennek a feladatnak EXORral való megvalósítása. A lényeg annyi, hogy a számítógép a változó értékét 2-es számrendszerben tárolja ennek következtében a szám 0-kból és 1-kból áll. A XOR (kizáró vagy) minden esetben 1-et ad vissza, azaz igaz értéket, kivéve ha a művelet jobb és bal oldalán azonos érték van, mert ilyenkor 0-t ad vissza. Ezt használjuk most ki a következő példában:

```
int a = 4; //2-es számrendszerben: 0100
int b = 5; //2-es számrendszerben: 0101
a = a^b; // 0100 ^ 0101 = 0001
b = a^b; //0001 ^ 0101 = 0100
a = a^b //0001 ^ 0100 = 0101
```

A komment szekcióban láthatjátok, hogy mi is történik a háttérben.

2.4. Labdapattogás

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

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

Megoldás forrása:[itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ahogy a videóban láthattátok, a labdapattogás annyiból áll, hogy a terminálon belül egy karakter pattog a az ablak teljes méretében. Fontos, hogy az ablak méretét állíthatjuk, és a program ezt érzékeli.

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

Az initscr() beolvassa az ablak adatait, melynek segítségével megtudjuk az ablak méretét. Ezután létrehozunk változókat, melyekben a lépésközt, a pozíciót, és az ablak méretét fogjuk eltárolni.

```
int x = 0; //aktuális pozíció x-tengelyen
int y = 0; //aktuális pozíció y-tengelyen

int xnov = 1; //lépésköz az x-tengelyen
int ynov = 1; //lépésköz az y-tengelyen
```

```
int mx; //ebben lesz eltárolva az ablak szélessége  
int my; //ebben pedig az ablak magassága
```

Ezután létrehozunk egy végtelen ciklust a már megszokott módon:

```
for ( ; ; )  
{  
}
```

És ebbe a végtelen ciklusba fogjuk "pattogtatni" a labdát. Ehhez elsőnek meghívjuk a getmaxyx() függvényt melynek átadjuk paraméterként a az ablakban eltárolt értékeket, és azt, hogy meylik változóba tárolja el az ablak hosszúságát és szélességét. És az mvprintw() függvény fogja az általunk megadott koordinátákba a karaktert "mozgatni".

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

Mostmár tudjuk az ablak méretét. Az x és az y változót folyamatosan 1-el növelve a karakter el kezd mozogni a terminálban. Azt, hogy ez milyen gyorsan történjen, azt a usleep() függvénytellyel tudjuk beállítani. A usleep mikroszekundumba számol, tehát az másodperc egymiliomod részében. Ha azt akarjuk, hogy a labda másodpercenként menjen 1-et, akkor 1000000-et kell beírnunk a usleep-be. Így:

```
usleep(1000000);
```

Most, hogy a labda már mozog, már csak meg kéne állnia az ablak határainál. Ezt pedig if-el fogjuk elsősorban megoldani.

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

Tehát, ha a labda eléri valamelyik szélét az ablaknak, akkor a lépésközöt megszorozzuk -1-el, így elérve, hogy visszapattanjon.

Egy másik megoldás is létezik ehhez, mégpedig az, ahol nem használunk if-et. Ennél a for-cikluson belül a következő írjuk:

```
getmaxyx(ablak, my, mx);
xj = (xj - 1) % mx;
xk = (xk + 1) % mx;

yj = (yj - 1) % my;
yk = (yk + 1) % my;

//clear ();

mvprintw (abs (yj + (my - yk)),  

          abs (xj + (mx - xk)), "X");

refresh ();
usleep (150000);
```

Ennél a maradékos ozstást használjuk ahhoz, hogy a labda egy bizonyos érték után "visszapattanjon". A lényeg annyi, hogy a modulóval való osztás, mindenkorának a szának az értékét adja vissza, amit elosztunk egészen addig, ameddig egyenlő nem lesz az ablak szélességével/hosszával, mert akkor újra visszaáll 1-re(vagy -1-re), ebben a pillanatban a labda elindul visszafelé, és ez folytatódik végtelen ciklusban. A program futását Ctrl+c-vel tudjátok megállítani. Jelenleg ehhez további magyarázatot nem tudok fűzni, mivel én nekem eszembe se jutott volna ez a megoldás módszer.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó:

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ahogy láthattad a forrásban, a gépeden egy szó 32 bites. Hogy ezt kiszámold, arra rengeteg megoldás létezik. Az egyik iylen a bitshiftelés, melynek lényege az, hogy addig léptetjük a számot, ameddig nulla nem lesz.

```
int a = 1; //kettes számrendszerben: 00000000 00000000 ←  

           00000000 00000001
while (a != 0){
    a <<= 1; //itt léptetjük eggyel:00000000 00000000 ←  

              00000000 00000010
              // újra: 00000000 00000000 00000000 00000100
              //...
}
```

Ha ezt megismételjük 32-ször, akkor végén csak nullakból fog állni, mivel ez nem körkörös folyamat, ha az egy elér az elejére, akkor nem fogvisszaugrani a végére. A linkelt forrásban az 1 hexadecimálisan lett megadva, de nyugodtan használhatod az általam írt példát, mivel ugyan azt az eredményt adja.

Ennek a résznek a másik fontos témája a BogoMIPS. Ez lényegében egy Cpu tesztelő program, melyet Linus Torvalds írt, és a linux kernel része mind a mai napig. A lényege az, hogy a program méri, hogy mennyi idő alatt fut le, ezzel megmondva, hogy a CPU-d milyen gyors. Persze, ha CPU vásárlás előtt állsz, ne pont ez alapján dönts egyik-másik processzor mellett. Számunkra azért érdekes ez a program, mert ennek a while fejléce ugyan azt a megoldást hasznája, amit mi a szóhossz számításához.

```
while (loops_per_sec <= 1)
{
    ;
}
```

Most, hogy láttátok, hogy mi a kapcsolat a mi kis programunk és a Linus Torvalds féle BogoMIPS között, akkor lássuk is, hogy hogyan működik pontosan. Elsőnek deklarálnunk kell 2 változót, az első a loops_per_sec, melynek definiálása során az értékét egyre állítjuk. A bitshiftelésnek köszönhetően ebbe 2 hatványokat fogunk tárolni. A ticks pedig a CPU időt fogja tárolni.

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

    ...
}
```

A while ciklus addig tart, ameddig a loops_per_sec le nem nullázódik. A ciklusba belépve, minden iterációban, lekérdezzük az aktuális CPU időt, és eltároljuk a ticks változóban. Ezután pedig meghívjuk a delay függvényt.

```
void delay (unsigned long long loops)
{
    unsigned long long i;
    for ( i=0; i<loops; i++);
}
```

Ez a függvény egy hosszú egész számot kér paraméterül, és amint látod, csak egy for ciklus megy végig 0-tól paraméter-1-ig. Ezután a while cikluson belül újra lekérjük az aktuális processzort időt és kivonjuk belőle a kezdeti étéket. Így megkapjuk, hogy mennyi ideig tartott a cpu-nak befejeznie a delay függvényben lévő for ciklus befejezéséig. Ezt egészen addig iteráljuk, ameddig nem teljesül az if-ben lévő feltétel.

```
while (loops_per_sec <= 1 )
{
    ...
}
```

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

printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec/500000,
       (loops_per_sec/5000) % 100);
    return 0;
}
}
```

A CLOCKS_PER_SEC a POSIX szabvány szerinti értéke 1.000.000, tehát akkor teljesül a feltétel, ha a processzor idő ezzel egyenlő, vagy nagyobb. Ezután pedig kiszámoljuk, hogy CLOCKS_PER_SEC idő alatt milyen hosszú ciklust képes végrahajtani a gép. Ennek az eredménye egy viszonylag nagy szám lesz, de a végeredmény megadásához használhatjuk a `log` függvényt, mely sokkal lassabban növekszik, így egy jóval kisebb számot kapunk eredményül. Ehhez csak egy kicsit kell módosítani az előbbi forrást:

```
printf("ok - %lld %f BogoMIPS\n", loops_per_sec, log( ←
    loops_per_sec));
```

2.6. Helló, Google!

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

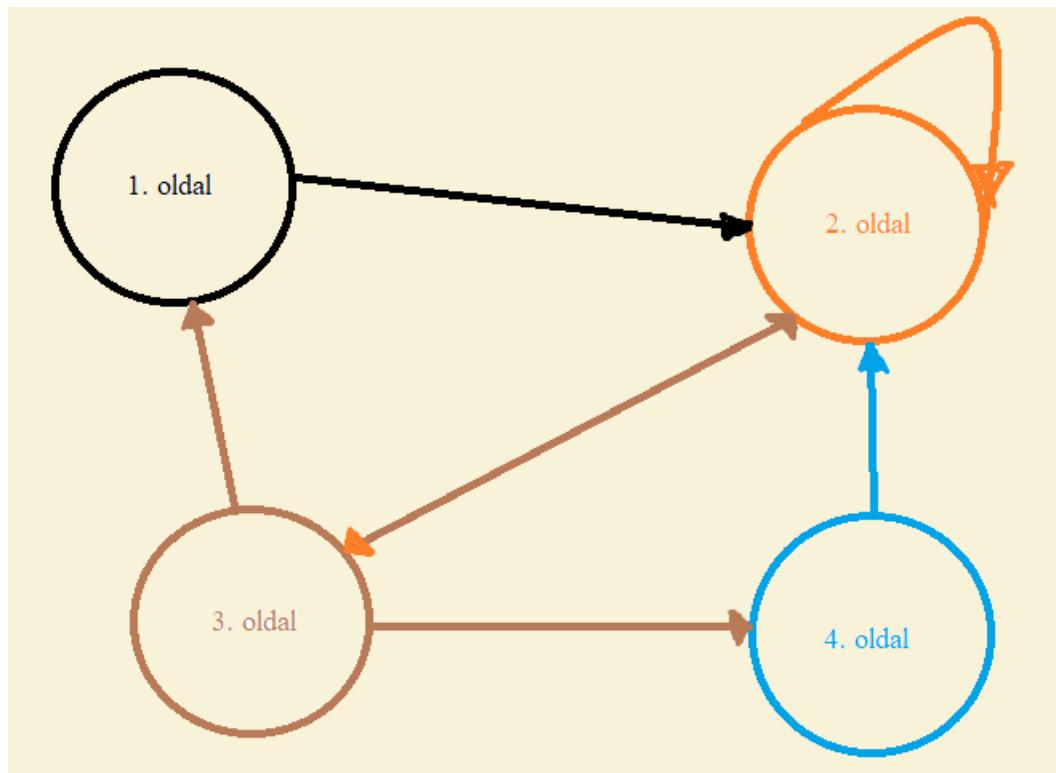
Megoldás videó:

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

A PageRank-et Larry Page és Sergey Brin fejlesztette ki 1998-ban. Az algoritmus az alapja mind a mai napig a Google keresőmotorjának. A lényege az, hogy osztályozza az oldalakat az alapján, hogy hány link mutat rájuk és a rájuk mutató oldalakra hány oldal mutat. Tehát általában az az oldal lesz az első találat, amelyikre a legtöbb oldal hivatkozik. Ez az ötlet megjelenésekor hatalmas forradalmat hozott az internethoz közelítők világában, hiszen a pontos találatok száma jelentősen megnőtt. Az előtt csak álom volt, hogy már az első találat az lesz, amit éppen keres, de a PageRank-kel ez valósággá vált. Ezt a forradalmi algoritmusnak láthatod a forrásban is, egy 4 honlapos hálózatra leszűkítve.

Az első iterációban mindegyik oldal PageRank-je 1/4, hiszen 4 oldalunk van. Ezután kezdjük el vizsgálni, hogy ezek között milyen kapcsolatvan.



2.2. ábra. PageRank

A kapcsolatokat egy mátrixban tároljuk:

```
double L[4][4] = {
    /*1.*/   /*2.*/   /*3.*/   /*4.*/
    /*1.*/{0.0,      0.0,   1.0/3.0,   0.0},
    /*2.*/{1.0,   1.0/2.0, 1.0/3.0,   1.0},
    /*3.*/{0.0,   1.0/2.0,      0.0,   0.0},
    /*4.*/{0.0,      0.0,   1.0/3.0,   0.0}
};
```

Az sorok és oszlopok metszetében láthatod, hogy milyen kapcsolatban van az oldalak között. Ezt a mátrixot adjuk át argumentumként a pagerank () függvénynek, mely így néz ki:

```
void
pagerank(double T[4][4]){
    double PR[4] = { 0.0, 0.0, 0.0, 0.0 }; //ebbe megy az ←
                                                //eredmény
    double PRv[4] = { 1.0/4.0, 1.0/4.0, 1.0/4.0, 1.0/4.0}; // ←
                                                               //ezzel szorzok

    int i, j;

    for(;;){
```

```
// ide jön a mátrix művelet

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

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

// ide meg az átpakolás PR-ből PRv-be

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

kiir (PR, 4);
}
```

A PRv[] tömbben tároljuk az oldalak első iterációbeli értékét, és a PR tároljuk el a mátrixszorzás eredményét. A mátrixszorzást a L és PRv tömb között hajtjuk végre, pontosan ebben a sorrendben, mivel ez a művelet nem kommutatív, csak bizonyos esetekben. A szorzás eredménye lesz egy 4x1-es oszlopvektor lesz, mely a mi "kis" hálózatunkban lévő 4 oldal PageRank-jét tartalmazza. A kiir() függvényel pedig kiíratjuk az egyes oldalakhoz tartozó eredményeket, mely a következő képpen zajlik:

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

    int i;

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

Tehát egy for ciklussal bejárjuk a tömböt, és egyenként kiírjuk az értékeket.

2.7. 100 éves a Brun téTEL

Írj R szimulációt a Brun téTEL demonstrálására!

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

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

Tanulságok, tapasztalatok, magyarázat...

Viggo Brun, XX. századi norvég matematikushoz kapcsolódik a Brun téTEL kidolgozása, melyet 1919-ben bizonyított be. A téTEL azt mondja ki, hogy az ikerprímek reciprokokösszege egy véges értékhez, úgynevezett Brun-konstanshoz konvergál. Ikerprímeknek nevezük azokat a prímpárokat, melyek különbsége 2. A konvergálás pedig azt jelenti, hogy az reciprokokösszeg soha nem halad meg egy bizonyos értéket, csak tetszőleges mértékben megközelíti. Ezt bitonyítottuk be egy R programmal, amit az előbbi linken láthatsz. Az R nyelv kifejezetten matematikai számításokhoz, statisztikák és grafikonok készítéséhez lett kifejlesztve. Ezért ideális ennek a téTELnek a bizonyításához is. Magát az ehhez szükséges programot itt tudod letölteni. A feladat megoldásához a matlab kiegészítőre lesz szükséged, melyet ezzel a parancsal tudsz telepíteni:

```
install.packages("matlab")
```

Ha megnézed a stp.r forráskódot, láthatod, hogy az R nyelv szintaktikája eltér a C nyelvétől, de könnyen tanulható. Mivel az R nyelv iteratív nyelv, ezért nem kell lefordítanod az egész fájlt gépi kódá, hanem sorról sorra írod be, és folyamatosan lefordul, majd végrehajtódik. A program futtatásához elsőnek be kell tölteni a matlab függvénykönyvtárat, majd létrehozzuk a stp függvényt.

```
library(matlab)

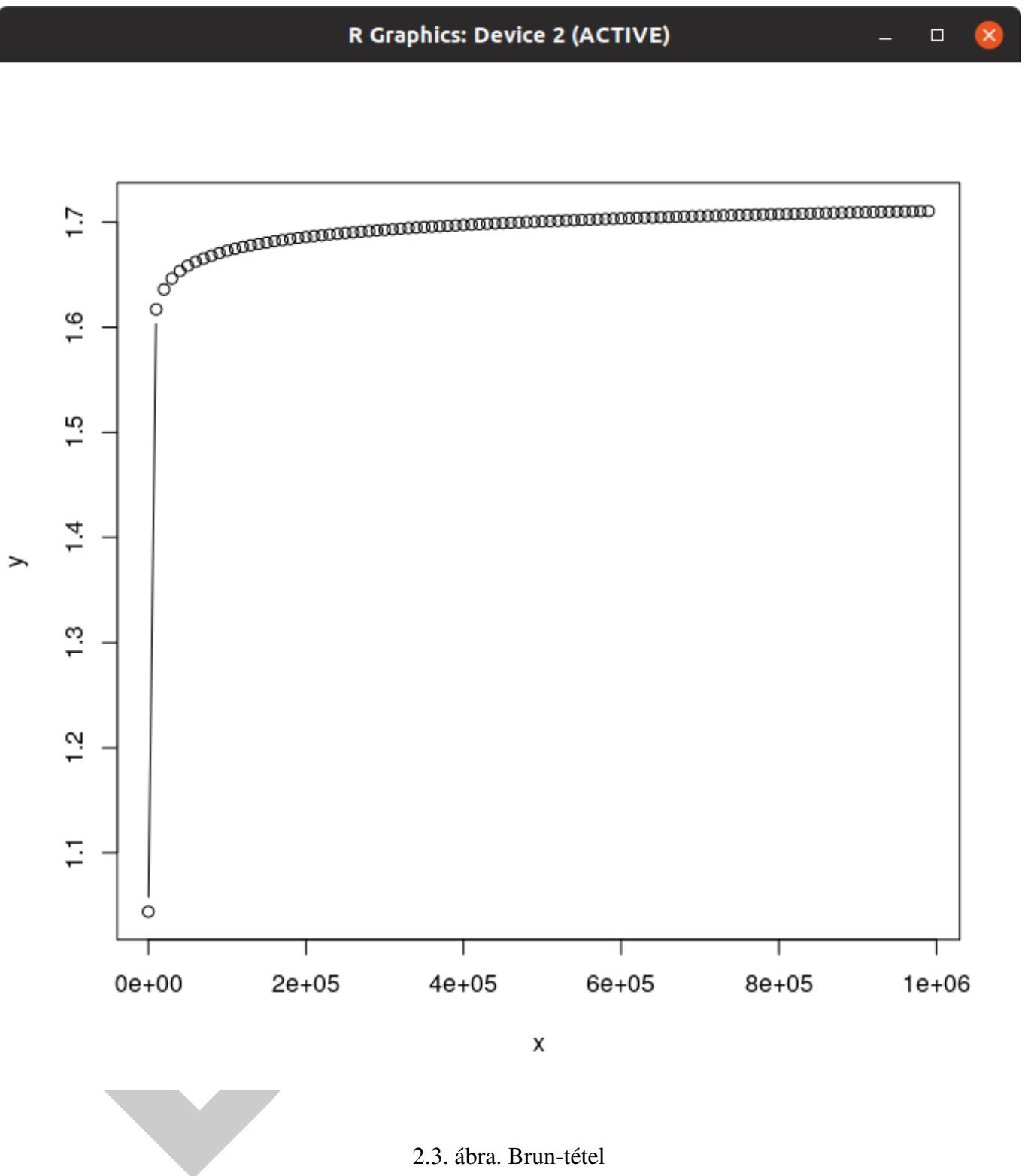
stp <- function(x) {

    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))
}
```

Ez a függvény 1 paramétert kér, és ezt adja tovább a primes beépitett matlab függvénynek. Ez a függvény a megadott x-ig kiír minden prím számot egy vektorba. A diff változóban eltároljuk a primes vektorban lévő egymás melletti prímek különbségét. Ezután az idx vektorban pedig diff vektor elemeinek indexét tároljuk el, ahol amelyek értéke 2. Ezután index alapján megnézzük, hogy melyek ezek párok, ahol a különbség kettő. Majd felhasználjuk a Brun-téTELt, tehát vesszük a prímeknek a reciprokát, és azt adjuk össze. Mivel egy függvényt akarunk kirajzolni, ezért meg kell adnunk egy x és egy y értéket, hogy ki tudjuk rajzolni az ábrát.

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

A seq() függvénytel megadjuk, hogy x-tengelyen mettől-meddig haladjunk, és milyen lépésközzel. A következő sort, pedig így kell érteni: y = stp(x). Tehát minden y-hoz hozzárendeljük az stp(x) értékét. A plot() függvény pedig kirajzolja az (x,y) értékeket egy grafikonon. Ha majd MATLAB-ot kell használnod, akkor jól megfogod ismerni a plot(), plot3() és a hasonló kiírató függvényeket. Tehát, ha ezt lefuttatod, akkor a következő ábrát fogod látni:



2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

paradoxon_kapcsan

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

Tanulságok, tapasztalatok, magyarázat...

A Monty Hall probléma lényege a következő. Van 3 ajtó, az egyik mögött van a nyeremény, a többi mögött semmi. A játékos választ egyet a 3 ajtó közül, majd a játékmester kinyit egy olyan ajtót, amit a játékos nem választott, és nincs mögötte semmi. Ezután megkérdezi a játékost, hogy szeretne-e változtatni a döntésén, vagy marad az általa elsőnek kiszemelt ajtónál. Kérdés az, hogy megéri-e váltania? Ahogy a belinkelt videóban is láthatadt ez még a legjobb matematikusoknak is fejfájést okozott. Tehát a válsz, a mindenkit foglalkoztatónak kérdésre az, hogy igen, megéri váltani.

Ennek oka nagyon egyszerű. Az első tippednél $1/3$ az esélye annak, hogy eltaláltad a helyes ajtót. Ekkor ha váltasz, akkor buksz, ha maradsz a döntésednél, akkor viszont nyersz. De tegyük fel, hogy nem találtad el a megfelelő ajtót. Ebben az esetben te rámutatsz egy üres ajtóra, a játékmester kinyitja neked a másik üres ajtót. Tehát, ha ebben az esetben váltasz, akkor nyersz. Ennek az esély pedig $2/3$ -hoz. Hiszen $2/3$ esélyteljesítés esetén váltasz, akkor nyersz. Egy szemléltető ábra, a fent linkelt oldalról: <https://m.blog.hu/bh/bhaxor/image-montyhall2.png>.

Ezt próbáljuk szimulálni egy R programban.

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

Első lépésként megadjuk, hogy mennyi legyen a kísérletek száma. Ha ez meg volt, akkor véletlenszerűséget szimulálunk a sample() függvénnyel. Ahogy látjátok ennek 3 argumentuma van, az első megadja, hogy mettől-meddig generáljon random számokat. Majd megadjuk, hogy háynszor tegye meg ezt, és a replace=T(rue) pedig azt engedi meg, hogy lehessen ismétlődés a számok között. Lényegében itt egy tömböt adunk a sample()-nek át, amit átrendez. Tehát a kísérletek tömbben tároljuk, hogy az egyes esetekben hol van a nyeremény, a játékos tömbben a játékos tippjeit. A műsorvezető változó pedig szintén egy tömb/vektor, melynek jelenleg csak a méretét adjuk meg, mivel az ő döntése függ a játékosétől és a nyeremény helyétől is.

```
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 ciklusban az i változót futtajuk 1-től a kísérletek számáig. Két esetet kell szétbontani itt, a játékos elatlálta a megfelelő ajtót, vagy sem. Ezt vizsgáljuk az if-ben. Tehát, ha az a kiserlet tömb első elem és a jatekos tömb első eleme megegyezik, akkor a játékvezető csak azt az egy ajtót nem választhatja. A mibol tömbben, pont ezt tároljuk el. Ehhez a setdiff() függvényt használjuk ,mely a megadott tömbből kiveszi a kiserlet tömb első elemével megegyző értéket. Ugyen ez történik az else ágon is, csak ott mind a kiserlet, minden a jatekos tömb első elemét ki kell vonni a meagdott tömbből. Ez alapján pedig fel tudjuk tölteni a musorvezeto vektort a megfelelő értékekkel, ehhez megint a sample() függvényt használjuk.

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

A nemvaltozatesnyer vektorba a which() függénnyel betöltjük azon elemek indexét melyek a a kiserlet és a jatekos tömbben azonos pozícióban vannak, és megegyeznek. Ezután létrehozzuk a valtoztat vektort, melynek mérete megegyezik a kísérletek számával.

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

    holvalt = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i ←
        ]))
    valtoztat[i] = holvalt[sample(1:length(holvart),1)]

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

A for cikluson belül feltöljtük a valtoztat vektort minden esetben azzal az ajtóval, amire a játékos muat, ha újra választ. Ezutána a valtoztatesnyer vektorbe betöltjük azonak az elemeknek az indexét, melyeknek az értéke megegyezik minden a kiserlet, minden a valtoztat tömbben. Ha ezzel megvagyunk, már csak ki kell print-elní a valtoztatesnyer és a nemvaltoztatesnyer vektorok hosszát, és ezzel megtudjuk, hogy melyik lesz a nagyobb. Természetesen az előbbi, ahogymár azt említettem.

3. fejezet

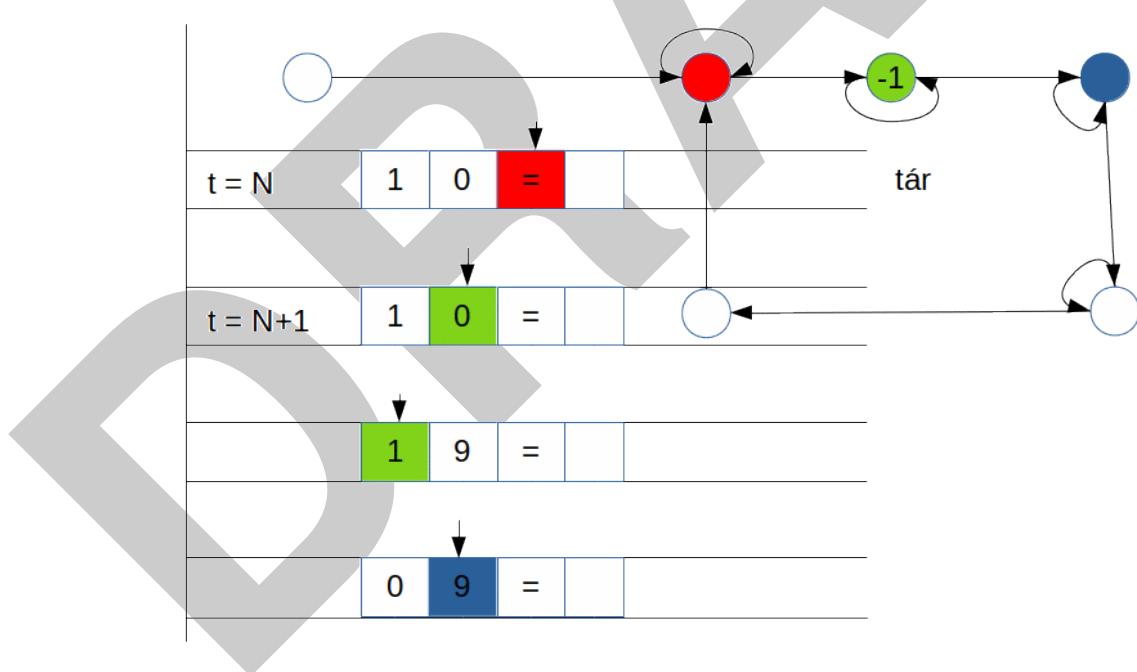
Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...



3.1. ábra. Decimálisból unárisba

A unáris számrendszer a létező legegyszerűbb az összes számrendszer közül. Lényegében megfelel annak, amikor az ujjaink segítségével számolunk, tehát a számokat vonalakkal reprezentáljuk. A szám ábrázo-

lása pont annyi vonalból áll, amennyi a szám. A könnyebb olvashatóság érdekében minden ötödik után rakhattunk helyközt, vagy az ötödik vonalat lényegében rakhattuk keresztbé az őt megelőző 4-re.

Az ábrán látható Turing gép az egyes számrendszerbe való átváltást végzi, de itt a 1 helyett 1-eseket írnak. A gép működése abból áll, hogy beolvassa a szalag celláiban tárolt számokat, ha a talál egyenlőség jelet, akkor az előtte lévő számból kivon 1-et. Ezt egészen addig teszi, ameddig az előtte lévő szám le nem nullázódik. Jelen esetben ez pont 0, de mivel előtte áll egy 1-es, ezért itt a 0-ból 9 lesz, és az 1-esből 0. Miközben folyamatosan vonja ki a egyeseket a számból, a kivont egyeseket kiírja a tárolóra. Így a végén egy 1-eskből álló sorozatot kapunk, melyek száma megegyezik a kiindulási szám értékével.

A forrásként megadott program lényegében egy átváltó, mely függőleges vonalakat ír ki a bemenettől függően. Mivel ez egy C++ program, ezért ennek a fordításához a g++-t érdemes használni, a szintaxisa teljesen megegyezik a gcc-nél megszokottakkal. Ha lefuttatod, akkor a következő fogod látni:

```
$ g++ unaris.cpp -o unaris
$ ./unaris
Adj meg egy számot decimálisan!
10
Unárisan:
||||| |||||
```

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Generatív nyelvtan jelSORozatok átalakítási szabályait tartalmazza. A nyelvnek megfelelő szavak létrehozásához szükség van egy kezdő értékre, és ezután már csak a szabályokat kell alkalmazni az átalakítások elvégzéshez. Tehát a kezdő szimbólumot más szimbólumokkal helyettesítjük a nyelvtani szabályok figyelembevételével.

Noam Chomsky, XX. századi amerikai nyelvész volt az, aki elsőnek javasolta a generatív nyelvtanok formalizálását. Az 1950-es években publikálta munkásságát, mely alapján egy nyelvtannak a következő elemeiből kell állnia:

- i. nem-terminális szimbólumokból
- ii. terminális szimbólumokból
- iii. produkciós szabályokból
- iv. előállítási szabályokból
- v. kezdő szimbólumból

Chomsky nevéhez fűződik még a generatív nyelvtanok csoportosítása. Négy osztályba sorolta őket ezek a következők:

- 0. típus (rekurzíve felsorolható nyelvtanok)
- 1. típus (környezetfüggő nyelvtanok)
- 2. típus (környezetfüggetlen nyelvtanok)
- 3. típus (reguláris nyelvtanok)

Ebben a feladatban az 1. típussal foglalkozunk. Az környezetfüggő nyelvtanok szabályait kétféle képpen lehet felírni. Itt a képzési szabály minden oldalon szerepelhetnek terminális szimbólumok, ellentétben a környezetfüggetlen nyelvtanokkal ahol a produkciós szabályok bal oldalán csak nemterminális szimbólum állhat.

Most hogy tudjuk, miből épül fel egy generatív nyelvtan lássuk a feladat megoldását.

S, X, Y „változók” – nemterminálisok

a, b, c „konstansok” – terminálisok

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

Jelen esetben a kezdő szimbólumunk az S lesz.

A képzési szabályokat alkalmazva a következőket kapjuk:

S ($S \rightarrow aXbc$)
 $aXbc$ ($Xb \rightarrow bX$)
 $abXc$ ($Xc \rightarrow Ybcc$)
 $abYbcc$ ($bY \rightarrow Yb$)
 $aYbbcc$ ($aY \rightarrow aaX$)
 $aaXbbcc$ ($Xb \rightarrow bX$)
 $aabXbcc$ ($Xb \rightarrow bX$)
 $aabbXcc$ ($Xc \rightarrow Ybcc$)
 $aabbYbcc$ ($bY \rightarrow Yb$)
 $aabYbbccc$ ($bY \rightarrow Yb$)
 $aaYbbbccc$ ($aY \rightarrow aa$)
 $aaabbccccc$

Tehát meg kell határozni változókat, terminálisokat és szabályokat. Amint látod a képzési szabályt formailag a nyíl operátorral jelöljük, tehát miből mi lesz. Ugyanennek a nyelvnek egy rövidebb reprezentációja a következő:

S ($S \rightarrow aXbc$)
 $aXbc$ ($Xb \rightarrow bX$)
 $abXc$ ($Xc \rightarrow Ybcc$)
 $abYbcc$ ($bY \rightarrow Yb$)
 $aYbbcc$ ($aY \rightarrow aa$)
 $aabbcc$

Fentebb említettem, hogy mi is teszi környezetfüggővé a nyelvtant, erre kiváló példa több is a képzési szabályokból. Például $bY \rightarrow Yb$, ahol tisztán látszik, hogy a nyíl bal oldalán is van terminális szimbólum, ez egy környezetfüggetlen nyelvben nem lenne lehetséges.

Egy másik nyelvtan, amely a feladatban szereplő nyelvet generálja a következőképpen néz ki:

A, B, C „változók”

a, b, c „konstansok”

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

A-ból indulunk ki, ez lesz a kezdőszimbólum.

A szabályokat követve:

```

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

Amint láthatod nem sokat változtattunk, elég volt csak a változókat átírni, és a szabályokat, ezzel meg is kaptunk egy másik generatív nyelvtant, mely szintén az $a^n b^n c^n$ nyelvet generálja.

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

A Backus-Naur-forma(vagyis BNF) segítségével környezetfüggetlen grammaatikákat tudunk leírni. Széles-körben alkalmazzák a programozási nyelvek szintaxisának leírására. Ahogy a neve is mutatja, a jelölés rendszer John Backus nevéhez fűződik, aki 1959-ben a párizsi WCC-n mutatta be. Majd Peter Naur egyszerűsített a jelöléseken, csökkentette a felhasznált karakterek számát. Tevékenységeért kiérdezelte, hogy az ő neve is szerepeljen a forma nevében.

Ahhoz, hogy lássuk hogyan is néz ki egy BNF leírás, vegyük egy egyszerű példát:

```

<postai_cím> ::= <név_rész> "," <irányítószám_rész> " " <cím_rész>
<név_rész> ::= <személyi_rész> <keresztnév> | <név_rész> <keresztnév>
<személyi_rész> ::= <titulus> "." "<vezetéknév>" | <vezetéknév>" "
<cím_rész> ::= <kerület> <elnevezés> <közterület_tipus> <szám> <EOL>
<közterület_tipus> ::= "utca"|"tér"|"körút"|"lépcső"|"u."|"krt."
<irányítószám_rész> ::= <ország_kód>"-"<irányítószám_belső>| <↔
    irányítószám_belső>
<irányítószám_belső> ::= <irányítószám> " "<városnév> ", "
  
```

Forrás: [wiki](#).

Nézzük például az első sort. Itt a postai címet határozzuk meg, ami áll egy név részből, irányítószámból és címből. Közéjük vesszőt rakunk, vagy csak szóközt. A második rész már egy kicsit érdekesebb, mert ebből kiderül, hogy a BNF jelölések rekurzívak is lehetnek. A név_részben hivatkozunk a személyi_részre, és önmagára is. A | jel vagy -ot fejez ki, az értékkadás pedig a ::= operátorral történik.

Mostmár van fogalmunk arról, mi is az a BNF, hát akkor próbáljuk meg átültetni ezt a tudást a C utasítás leírására. Mielőtt tovább haladnál a könyv olvasásával érdemes elolvasni a feladatban említett részletet a K&R könyvből.

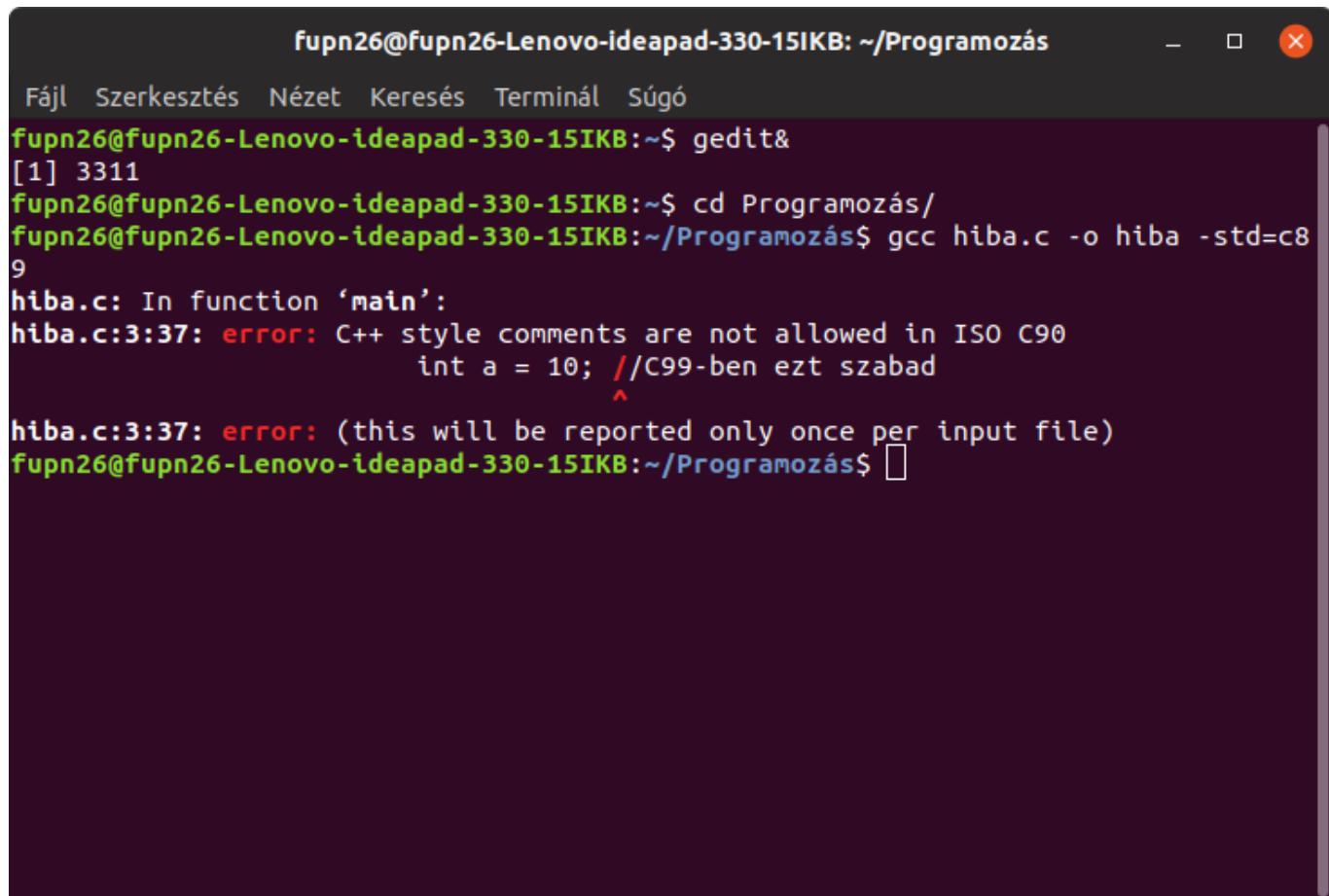
```
<utasítás> ::= <kifejezés_utasítás>|<összetett_utasítás>|
               <feltételes_utasítás>|<while_utasítás>|
               <do_utasítás>|<for_utasítás>|<switch_utasítás>|
               <break_utasítás>|<continue_utasítás>|<return_utasítás>|
               <goto_utasítás>|<címkézett_utasítás>|<>nulla_utasítás>
<kifejezés_utasítás> ::= <kifejezés>;
<összetett_utasítás> ::= {<deklarációlistaopc> <utasításlistaopc>}
<feltételes_utasítás> ::= if (<kifejezés>)<utasítás>|<feltételes_utasítás> ←
    else <utasítás>
<while_utasítás> ::= while (<kifejezés>) <utasítás>
<do_utasítás> ::= do <utasítás> while (<kifejezés>)
<for_utasítás> ::= for (<1._kifejopc>;<2._kifejopc>;<3._kifejopc>) < ←
    utasítás>
<switch_utasítás> ::= switch (<kifejezés>) <utasítás>| <switch_utasítás> ←
    case
        <állandó_kifejezés>:|<switch_utasítás> default:
<break_utasítás> ::= break;
<continue_utasítás> ::= continue;
<return_utasítás> ::= return; | <return_utasítás> <kifejezés>
<goto_utasítás> ::= goto <azonosító>;
<címkézett_utasítás> ::= <azonosító>:
<>nulla_utasítás> ::= ";"
```

A C programnyelvet az 1970-es években alkották meg, mely azóta 3 főbb verziót ért meg. Az első volt a C89, ez a klasszikus C nyelv, amit a K&R könyvből ismerhetünk meg. Ezután jött a C99, mely újdonságai között szerepelt többek között az inline függvények támogatása, új adattípusok jelentek meg, mint a long long int, vagy a complex. Egészen eddig nem volt támogatott az egysoros komment // jelölése, és a dinamikus tömb sem. Egy újabb verziója pedig 2011-ben jelent meg C11 néven, mind a mai napig ez a legfrissebb sztenderd. Ez főleg a C99-es sztenderd továbbfejlesztésének tekinthető.

Fontos megjegyezni, hogy a gcc lap esetben a C89-es szabványt használja, tehát, ha ezen módosítanál, akkor a -std=c99 kapcsolót kell használnod.

```
int main()
{
    int a = 10; //C99-ben ezt szabad
    return 0;
}
```

Ránézésre semmi probléma nem lehetne vele, de a gcc-t a -std=c89 kapcsolóval használva hibát fog kiírni.



The screenshot shows a terminal window titled "fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás". The terminal content is as follows:

```
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~$ gedit&
[1] 3311
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~$ cd Programozás/
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ gcc hiba.c -o hiba -std=c89
hiba.c: In function 'main':
hiba.c:3:37: error: C++ style comments are not allowed in ISO C90
    int a = 10; //C99-ben ezt szabad
                                ^
hiba.c:3:37: error: (this will be reported only once per input file)
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ 
```

3.2. ábra. 1. Splint kép

Igen, jól látod, az egysoros komment jelölésére használt // a C99 előtt, csak a C++ támogatta. Ennek a támogatásának bevezetése a C nyelvbe, a C99-es sztenderd egy nagy újdonsága volt. Ha C89-es szabvány szerint szeretnél kommentelni, akkor az előző példa, így módosul:

```
int main()
{
    int a = 10 /*ez már lefordul -std=c89-el*/
    return 0;
}
```

Ennek a kommentelésnek az előnye, hogy lehetőséget biztosít a többsoros komment létrehozására.

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása:[itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ehhez a feladathoz a lex programot kell használni, melyel egy lexikális lemezőt lehet készíteni. Szövegfájlok ból olvassa be a lexikális szabályokat, és egy C forráskódot készít, melyet a gcc-vel tudunk fordítani. A lex forráskód 3 részből áll, az első a definíciós rész, amely lényegében bármilyen C forrást tartalmazhat, itt lehet include-álni a header fájlokat. A második rész a szabályoknak van fenntartva. Ez 2 részből áll, reguláris kifejezésekkel, és az azokhoz tartozó C utasításokból. Tehát, ha a program futása során a bemenetként kapott string illeszkedik valamelyik reguláris kifejezésre, akkor végrehajtja a hozzá tartozó utasítást. A harmadik rész pedig egy C-kód, amely lényegében a lexikális elemzőt hívja meg. Ez a rész, és az első, teljes mértékben átmásolódik a lex által generált C forrásba. A részeket %%-jellel különítjük el.

Most hogy már tudod mi is az a lexer, itt az idő végig futni a belinkelt forráson. Az első rész a következő:

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

Ahogy említettem, itt importáljuk a header fájlokat, és a szükséges változókat is itt deklaráljuk. Ezután következik a második rész:

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

Itt a bal oldalon vannak a reguláris kifejezések, jobb oldalon pedig az, amit a C program végrehajt. Ez a regex azokra a bemenetekre illeszkedik, amelyek számmal kezdődnek, és a * jelöli, hogy többször is előfordulhat. Majd a zárójelben lévő csoportból valamelyik tag 0-szor, vagy 1-szer fordul elő. Ezt jelöli a ?-jel. Az atof() függvény pedig az argumentumként kapott stringet double-lé alakítja. Vessünk egy pillantást a harmadik részre:

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

Ez tartalmazza a yylex() lexikális elemző függvény hívását, és itt íratjuk ki az eredményt is.

Most, hogy láttad, hogyan épül fel a lexer, akkor már csak ki kéne próbálni.

```
$ lex -o lexikalis.c lexikalis.l  
$ gcc lexikalis.c -o lexikalis  
$ ./lexikalis  
12 34 54 12  
The number of real numbers is 4
```

A bevitelt a Ctrl+D parancssal lehet megállítani. Szöveges fájlt beleírányítani pedig a

```
./lexikalis <fajl_nev
```

parancssal lehet.

3.5. l33t.l

Lexelj össze egy l33t cipher!

Megoldás videó: [itt](#)

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Elsőnek nézzük meg, hogy mi is az a leet nyelv. Ennek a nyelvnek az a lényege, hogy a szavakban lévő bizonyos betűket, számokkal, vagy egyéb más karakterekkel helyettesítjük. Az egyes betűket közmegállapodás szerinti karakterekre cserélhetjük. Erről a teljes listát [itt](#) találod. Természetesen a linkelt programunk is ebből indul ki, de nem minden opciót vettünk bele.

Az előző feladatban már ismertettem veled a lexer felépítését, és ugyan ezt használjuk itt is. Természetesen az első részben most is importálva vannak különböző header fájlok, de ami érdekesebb az utána jön.

```
#define L337SIZE (sizeof l337d1c7 / sizeof (struct cipher))

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

{'a', {"4", "4", "@", "/-\\"}}, 
{'b', {"b", "8", "|3", "|}"}, 
{'c', {"c", "(", "<", "{"}}, 
    ...
};
```

A `#define` segítségével lényegében egy makróhelyettesítést hajtunk végre, ezt úgy kell elképzelni, hogy ha a programban valahol hivatkozunk az `L337SIZE`-ra akkor a mellette található értékkel fogja helyettesíteni. Ezután létrehozzuk a `cipher` struktúrát, mely egy `char c`-ből, és egy 4 elemű karakterekből álló tömbre mutató `char *` típusú mutatóból áll. Ez alapján a struktúra alapján létrehozzuk az `l337d1c7 []` tömböt. Ez a tömb tartalmazza az egyes betűket, és a hozzájuk tartozó lehetséges helyettesítő karaktereket. Tehát a a tömb minden eleme áll egy `char c`-ből, és egy `char *leet[4]`-ből.

A második részben ezúttal nem használunk különféle regex-eket, tehát minden karakter esetén végrehajtjuk a C utasítást.

```
. {

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

```
{  
  
    if(l337d1c7[i].c == tolower(*yytext))  
    {  
  
        int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0)); //random szám ←  
        generálás  
  
        if(r<91)  
            printf("%s", l337d1c7[i].leet[0]);  
        else if(r<95)  
            printf("%s", l337d1c7[i].leet[1]);  
        else if(r<98)  
            printf("%s", l337d1c7[i].leet[2]);  
        else  
            printf("%s", l337d1c7[i].leet[3]);  
  
        found = 1;  
        break;  
    }  
  
}  
  
if(!found)  
    printf("%c", *yytext);  
  
}
```

Tehát a .-t használjuk, ami minden karakterre illeszkedik. A for ciklus fejében lekérjük a L337SIZE konstans értékét, ami a l337d1c7[] tömbnek és a cipher struktúra méretének a hányadosa. Mivel a l337d1c7[] egy struktúrált tömb, ezért a l337d1c7[i].c tudunk hivatkozni a tömb i-dik elemének a char c részére. A tolower() függvény segítségével az esetleges nagybetűket kicsivé változtatjuk. Majd ezután képzünk egy random számot 1-100 között. Ha 91-nél kisebb számot "dob" a gép, akkor a char *leet[4] tömb első elemét printeljük, és így tovább, ahogy a forrásban látod. A int found változót azért vezettük be, hogy jelezzük, benne van-e tömbünkbe a beolvasott karakter. Ha nincs, akkor visszadujuk az eredeti karaktert, módosítás nélkül.

Ezután megint a C forrás jön, melyben elindítjuk a lexelést.

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

Itt található egy érdekesség, az srand() függvény adja meg a kiindulási értéket a rand() függvény számára. Jelen esetben az aktuális időt és a szülő folyamat PID-jának összegét adja át.

3.6. A források olvasása

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

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

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



Bugok

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

i.

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

ii.

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

iii.

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

iv.

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

v.

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

vi.

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

vii.

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

viii.

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

Megoldás forrása:

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat...

Programozó pályafutásunk során rengetegszer kell mások kódjában tájékozódni, ez a feladat ebben nyújt segítséget, hogyan értelmezzük helyesen a kódokat. Lássuk is az elsőt. A bevezetőben már láthattad, hogy melyik manuál oldalakat kell átnézni ehhez a kódcsipethez, szóval ezzel most nem rabolnám az időt.

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

Ez azt jelenti, hogyha eddig nem volt figyelmen kívül hagyva a SIGINT jel, akkor a jelkezelő függvény kezelje. Ellenkező esetben hagyjuk figyelmen kívül.

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

Ez egy for ciklus, amelynél az első iterációban az i nulla, majd megnézzük, hogy kisebb-e, mint 5, és minden iterációban növeljük 1-el.

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

Ez ebben az esetben megegyezik az előzővel, ugyan úgy 0,1,2,3 majd 4 lesz az i értéke. Viszont ez nem minden iterációban növeljük 1-el.

```
int a = 5;
int b = a++; //itt a b értéke 5 lesz, majd növeljük az a értékét 1-el
int c = ++a; //c értéke már nem 6 lesz, hanem 7, mivel itt előbb ↪
növelünk
```

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

Ez már viszont egy bugos program, mivel egyszerre inkrementáljuk az i-t, és hivatkozunk a tomb i. elemére. Az a baj, hogy nem ismerjük a végrahajtás sorrendjét, emiatt nem kiszámítható az eredmény.

```
furjes-benke@fupn26: ~/Tanulmányok/Prog1
Fájl Szerkesztés Nézet Keresés Terminál Lapok Súgó
furjes-benke@fupn26: ~/Tanulmányo... furjes-benke@fupn26: ~/bhax/themati...
different places not separated by a sequence point constraining evaluation
order, then the result of the expression is unspecified. (Use -evalorder to
inhibit warning)

Finished checking --- 1 code warning
furjes-benke@fupn26:~/Tanulmányok/Prog1$ splint gyak.c -evalorder
Splint 3.1.2 --- 20 Feb 2018

Finished checking --- no warnings
furjes-benke@fupn26:~/Tanulmányok/Prog1$ splint gyak.c
Splint 3.1.2 --- 20 Feb 2018

../../../../Tanulm\377\377nyok/Prog1/gyak.c: (in function main)
../../../../Tanulm\377\377nyok/Prog1/gyak.c:8:34:
    Expression has undefined behavior (left operand uses i, modified by right
    operand): tomb[i] = i++
Code has unspecified behavior. Order of evaluation of function parameters or
subexpressions is not defined, so if a value is used and modified in
different places not separated by a sequence point constraining evaluation
order, then the result of the expression is unspecified. (Use -evalorder to
inhibit warning)

Finished checking --- 1 code warning
furjes-benke@fupn26:~/Tanulmányok/Prog1$
```

3.3. ábra. 1. Splint kép

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

Ez a kódcsípet színtén bugos, a probléma az, hogy az értékadó operátort használjuk, az összehasonlító operátor helyett, ennek következtében a `&&` operátor jobb oldalán nem egy logikai operandus áll.

```
furjes-benke@fupn26: ~/Tanulmányok/Prog1$ splint gyak.c
Splint 3.1.2 --- 20 Feb 2018

.../.../Tanulm\377\377nyok/Prog1/gyak.c: (in function main)
.../.../Tanulm\377\377nyok/Prog1/gyak.c:11:25:
    Right operand of && is non-boolean (int): i < meret && (*d++ = *s++)
    The operand of a boolean operator is not a boolean. Use +ptrnegate to allow !
    to be used on pointers. (Use -boolops to inhibit warning)
.../.../Tanulm\377\377nyok/Prog1/gyak.c:10:18:
    Variable tomb declared but not used
    A variable is declared but never used. Use /*@unused@*/ in front of
    declaration to suppress message. (Use -varuse to inhibit warning)

Finished checking --- 2 code warnings
furjes-benke@fupn26:~/Tanulmányok/Prog1$
```

3.4. ábra. 2. Splint kép

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

Ez is hibás kód, mivel az f függvény két int-et kap, de azok kiértékelésének sorrendje nincs meghatározva.

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

Ennél a kódcsipetnél nincs probléma, kiírjuk az a értékét és az a függvény által módosított értékét.

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

Ennek a kódnak szintén a kiértékelés sorrendjével van baja, mivel az f() függvény módosítja az a értékét, emiatt nem tudhatjuk, hogy az önmagában kiprintelt a az eredeti értékét, vagy a módosított értékét fogja kiírni.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

Tanulságok, tapasztalatok, magyarázat...

Az aritmetika nyelve a logikai nyelvek közül az elsőrendűek közé tartozik. A legalapabb logikai nyelv a nulladrendű logika, ez csak ítéletváltozókat tartalmaz, és 3 műveletet: a konjunkciót, diszjunkciót és implikációt. Erre épül rá az elsőrendű logika, kiegészülve függvényszimbólumokkal, változókkal, kvantorokkal. Ha részletesebb tájékozódnál, akkor a forrásban több könyv is meg van nevezve, ami a feladathoz szükséges ismereteket tartalmazza.

Akkor térjünk rá a feladatra. Elsőnek elmondom, hogy melyik kifejezés mit jelent. A `forall` fogja jelölni az univerzális kvantort, és az `exist` pedig az egzisztenciális kvantort. A `wedge` alatt az implikációt értjük, és a `supset` pedig a konjunkciót. Ehhez a feladathoz érdemes még ismerni az Ar nyelv rakkövetkező függvényét, amit S-el jelölünk. Most, hogy ezeket tudjuk, már csak le kell fordítani a emberi nyelvre a fentebb látható formulákat.

1. minden x-hez létezik olyan y, amelynél ha x kisebb, akkor y prim.
2. minden x-hez létezik olyan y, amelynél ha x kisebb, akkor y prim, és ha \leftrightarrow y prim, akkor annak második rakkövetkezője is prim.
3. létezik olyan y, amelyhez minden x esetén az x prim, és x kisebb, mint y \leftrightarrow .
4. létezik olyan y, amelyhez minden x esetén az x nagyobb, és x nem prim.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciája
- egészek tömbje
- egészek tömbjének referenciája (nem az első elemé)
- egészre mutató mutatók tömbje

- egészre mutató mutatót visszaadó függvény
- egészre mutató mutatót visszaadó függvényre mutató mutató
- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

Mit vezetnek be a programba a következő nevek?

- `int a;`
- `int *b = &a;`
- `int &r = a;`
- `int c[5];`
- `int (&tr)[5] = c;`
- `int *d[5];`
- `int *h();`
- `int *(*l)();`
- `int (*v(int c))(int a, int b);`
- `int (*(*z)(int))(int, int);`

Megoldás videó:

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

A feladat első részében készíteni kell egy programot, ami tartalmazza a fentebb felsorolt elemeket. Menjünk végig a forráson. Maga a forrás C++ nyelven íródott mivel a referenciakat a sima C nem támogatja.

```
int* fakt(int szam){  
    static int a = 1;  
    if (szam < 2)  
        return &a;  
    while (szam>1){  
        a = a*szam;  
        --szam;  
    }  
    return &a;  
}
```

Az első függvény egy számnak a faktoriálisát számolja ki, és visszaad egy erre az értékre mutató mutatót. Mivel a függvényen belül deklarált változók lokálisak, és törlődnek a függvény lefutása után, ezért nem tudunk visszatérni annak a memóriacímével. Ezért kellett használni a static kulcszót, mely lehetővé teszi, hogy a lokális változó benne maradjon a memóriában, akkor is ha vezérlés már továbblépett a függvényen.

```
int* sum(int egyik, int masik) {
    static int sum = egyik + masik;
    return &sum;
}

int szorzat(int egyik, int masik) {
    return egyik*masik;
}

int osztas(int egyik, int masik) {
    return egyik/masik;
}
```

Ezek a függvények lényegében segítségként jöttek létre, hogy a függvényre mutató mutatókat használni tudjuk a programban. Ebben a dologban a legnagyobb problémát a fakt függvényben lévő statikus változó okozta. Csinálni kell egy egészre mutató mutatót visszadó függvényre mutató mutatót. Ez a pointer mutathatna a fakt-ra, csak az a probléma, hogyha a pointeren keresztül meghívja a program a függvényt, akkor a sum értéke megváltozik. Ezért definiáljuk a sum függvényt, ami szintén a egészre mutató mutatót ad vissza. A szorzat és osztas függvények pedig, ahogy említettem, csak segéd függvények.

```
int (*pfgv (int a)) (int,int) {
    if (a){
        return &szorzat;
    }
    else
        return &osztas;
}
```

A pfgv egy olyan függvény, ami függvényre mutató pointert ad vissza, egy olyan függvényre mutató pointert, ami két egész kér paraméteréül. A pfgv egy egészet kér paraméteréül, és ennek függvényében ad vissza egy pointert, vagy a szorzat vagy a osztas függvényekre.

```
int main()
{
    int a = 10;
    int b = 5;
    int* pa = &a;
    int& ra = a;
    int tomb[a];
    int(& rtomb)[a] = tomb;
    int* ptomb[2];
    ptomb[0] = &a;
    ptomb[1] = &b;
    int* fakt_a = fakt(a);
    int*(psum)(int,int) = &sum;
    int (*(*p_pfgv) (int valami))(int, int) = &pfgv;
```

```
    std::cout << "a és b szorzata " << (pfgv(1)) (a,b) << std::endl;
    std::cout << "a és b hányadosa " << (pfgv(0)) (a,b) << std::endl;
    std::cout << "a és b hányadosa " << (p_pfgv(0)) (a,b) << std::endl;
    std::cout << "a értéke "<< a << '\t' << "a! értéke "<< *fakt_a << std::endl;
    std::cout << "a és b összege " << *psum(a,b) << std::endl;
}
```

Az main függvény tartalmazza a feladat további megoldásait. A a és b változók egészek. A pa egy egész számra mutató pointer. A pointer egy memóriacímre mutat, emiatt az a változó címét kell képeznünk, amit a & operátorral lehetünk meg. A referencia nagyon hasonlít a pointerre, ez is egy változóra hivatkozik. Lényegében az érekéül adott változó aliasa. Előnye a pointerrel szemben, hogy nem foglal külön helyet, hanem a változó memóriacímét használja. Ilyen referencia a ra. Itt a & nem címképző operátor, hanem ezzel jelezzük a fordítónak, hogy egy referenciát deklaráltunk. Az tomb egy egészket tartalmazó tömb, aminek az elemszáma megegyezik az a változó értékével. A rtomb pedig a tomb tömbnek a referenciaja, nem csak az első elemé, hanem az egész tömböt. A ptomb tömb egészekre mutató mutatók tömbje. Ahogy látható az elemeihez hozzá tudjuk rendelni a változók memóriacímét. A fakt_a pointer pedig a fakt függvény által visszaadott sum változóra mutat. A psum szintén egy pointer, de nem egy változóra mutat, vagy egy tömbre, hanem egy függvényre. Jelen esetben egy olyan függvényre tud mutatni, ami int* ad vissza, azaz egészre mutató pointert, és paraméterül két egész kér. Ennek a leírásnak pont megfelel a sum függvény. A p_pfgv szintén egy függvénymutató, ahogy a nevéből látszik a pfgv függvényé. Vagyis ez a pointer egy olyan függvényre mutat, ami paraméterül kér egy egész, és egy olyan függvényre mutató pointert ad vissza, ami két egészet kér be, és egy egészet ad vissza. A cout segítségével pedig kiírjuk a képernyőre a következő sorokat. Érdekesség, hogy a pointere kereszttüli függvényhívásoknál nincs szükség a * használatára, ellentétben a változókkra mutató pointerekkel. Például a fakt_a értékét csak a * operátor segítségével kaphatjuk meg, ha ellenkező esetben a memóriacímet adja vissza. Viszont a psum pointeren kereszttüli függvényhívásnál már használtunk * operátort, mivel a sum függvény egy egészre mutató pointert ad vissza.

Mostpedig lássuk a feladat másik részét, ahol le kell írni, hogy mit vezetnek be a programba az alábbi kifejezések. Az feladat első részében már minegyikkel kellett dolgoznunk, szóval könnyű dolgunk lesz.

```
int a;
```

egy egész vezet be a programba.

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

egy egészre mutató mutatót vezet be. A mutató, vagy elterjedtebb nevén pointer lényegében egy olyan változó, ami egy változó, egy tömbre esetleg egy függvényre memóriacímre mutat. Érdemes megfigyelni, hogy a &a vagyunk képesek átadni az a változó memóriacímét.

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

egy egésznek a referenciaját vezeti be. Láthatod, hogy a mutatót a * operátor segítségével deklarálunk, míg a referenciait a & jelkel. A kettő közti különbség az, a referencia, lényegében egy alias egy másik változóhoz. Nem foglal külön helyet a memóriában, mint a pointer, hanem osztozik a területen a referált változóval.

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

egy egészkből álló, 5 elemű tömböt deklarál.

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

az egészek tömbjének referenciáját vezeti be a programba. Fontos látni, hogy ez nem az első elemek a referenciája, hanem az összes elemnek.

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

egészre mutató mutatók tömbje.

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

egészre mutató mutatót visszadó függvény.

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

egészre mutató mutatót visszaadó függvényre mutató mutató.

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

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

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

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

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

Megoldás forrása: [itt](#)



Megjegyzés

A feladat megoldásában tutoráltként részt vett: Országh Levente.

Tanulságok, tapasztalatok, magyarázat...

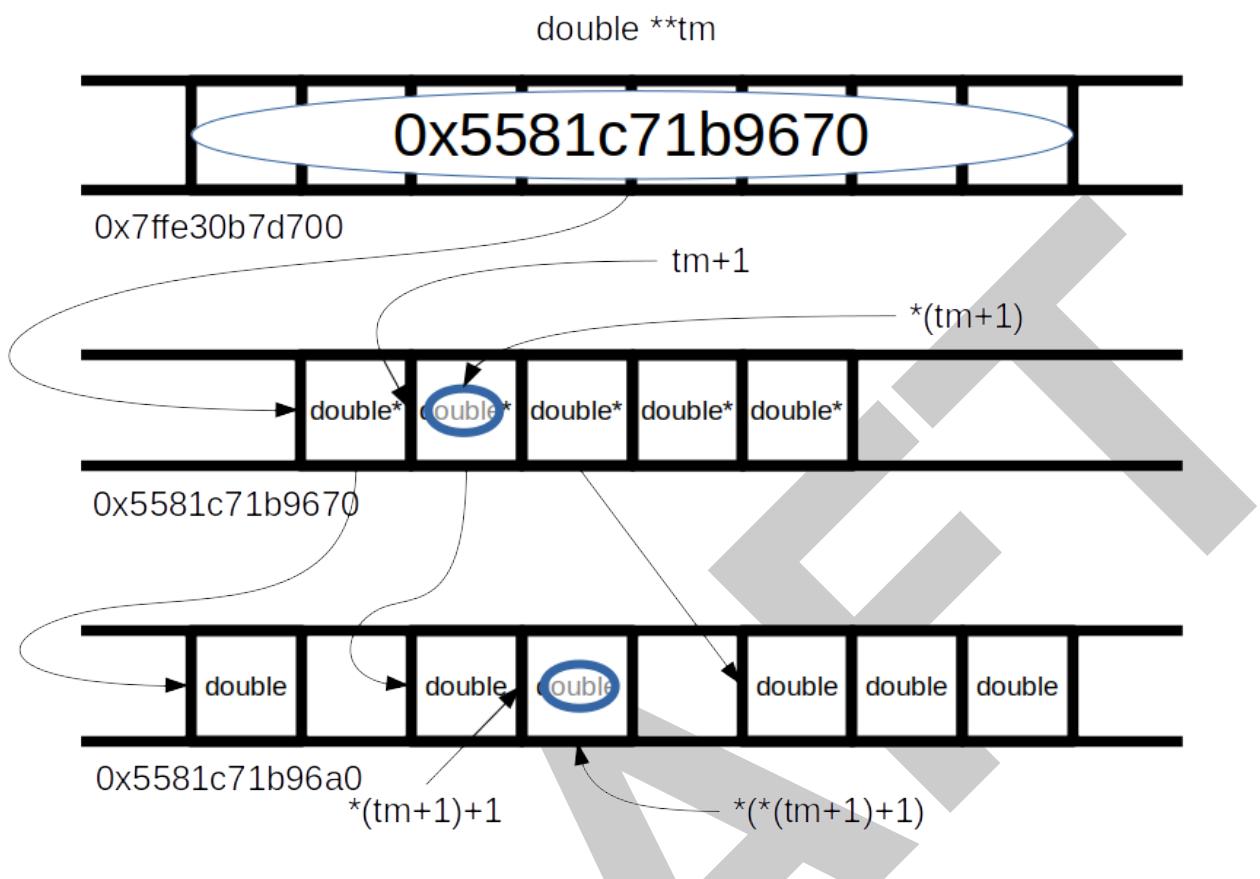
Ebben a feladatban a pointerek használatával fogunk egy kicsit jobban megismerkedni. De mielőtt rátérnék a forrásra, előtte tisztázzuk mi is az a háromszögmátrix. Ez 2 tulajdonsággal rendelkezik, az első, hogy négyzetes, tehát sorai és oszlopai száma megegyezik, a másik pedig az, hogy a főátlója alatt(a mi programunkban felett) csupa nulla szerepel. A program ezt a mátrixot fogja elkészíteni:

Főátló →

		Oszlopok				
		0	0	0	0	0
Sorok	1	2	0	0	0	
	3	4	5	0	0	
	6	7	8	9	0	
	10	11	12	13	14	

4.1. ábra. Háromszögmátrix

Most, hogy tudjuk, mit várunk el a programtól, nézzük meg, hogyan lehet ezt megvalósítani. Ahogy a feladat kiírásában is láthatjátok, ezt egy egészre mutató mutatatónak a mutatójával fogjuk létrehozni. Gondolom ez most egy kicsit bonyolultnak hangzik, de itt egy ábra a program egyszerű megértéséhez:



4.2. ábra. Pointerek a memóriában

A könnyebb megértés érdekében vegyük sorra, hogy mi is van az ábrán. Legfelül látjuk az double $\star\star$ tm mutatót, ez a deklaráció. Azért áll 4 négyzetből, mert az double típusú változók 8 bájtosak. És ez a mutató egy double \star -ra mutat, vagyis annak a memóriacímére, a memóriacím kerül be a kék oválisba. Ezután láthatod a tm+1-et, ez azt jelenti, hogy a tm mutató "eggyel" arrébb mutat, vagyis 4 bájttal árrébb. A \star (tm+1)-el pedig a benne lévő értéket kaojuk meg. Ezzel ekvivalens jelölés, mely talán jobban érthető, ha úgy képzeljük el ezt, mintha egy tömb lenne. Tehát a tm+1 értelmezhető így: &tm[1], vagyis a tm tömb második elemének memóriacímeként. Ebből következik, hogy a \star (tm+1) pedig a tm[1], mely nem más, mint a tm tömb második elemének az értéke. Mivel még mindig a pointereknél tartunk, ezért ez az érték szintén egy memóriacím lesz, mégpedig egy double típusú változójé. És ezzel megérkeztünk a változók szintjére, ahol már nincsenek pointerek. Ha azt akarod tudni, hogy mi a \star (tm+1) által mutatott double értéke, akkor egyszerű a dolgunk, csak elé rakunk még egy csillagot, tehát \star (* (tm+1)). Értelem szerűen, itt úgy tudsz a következő elemre mutatni, hogy +1-et hozzáadsz, vagyis \star (tm+1)+1, ennek az értékét pedig \star (* (tm+1)+1). Ebben az esetben is használhatod a tömbös analógiát. Mivel a mátrix lényegében egy két dimenziós tömb, ezért ábrázolhatod így is &tm[1][1] a memóriacímet, és tm[1][1]-el az értékét. Összességében rajta áll, hogy melyiket szeretnéd használni, kezdetben talán a tömbös megoldás érthetőbb, de érdemes hozzászokni a másikhoz, mert az az elterjedtebb.

Most elemezzük a programot sorról sorra.

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

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

Az elején, ahogy már megszoktad includálj a megfelelő header fájlokat, az stdio.h ismerős lehet, ez kell a printf használatához, és most megismерkedünk az stdlib.h-val, mely a malloc utasítás használatához lesz szükséges. Az nr tartalmazza az oszlopok számát, és itt deklarájuk a **tm pointert is.

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

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

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

Ebben a részben a printf kiírja a tm memóriacímét, majd az if feltételén belül, a tm-et ráállítjuk a malloc által foglalt 5*8 bájtnyi táterületre. A malloc egy pointert ad vissza, ami a lefoglalt tárra mutat, void * típusút, tehát bármely típust vissza tud adni típuskényszerítéssel. Jelen esetben ez double ** visszaadására kényszerítjük. Majd az if feltételeként megvizsgáljuk, hogy tudott-e lefoglalni területet a malloc, ha nem, akkor visszatérünk hibával. Ha a tárfoglalás sikerült, akkor kíírjuk a lefoglalt tár címét. Ha az ábrát visszanézed, most tartunk a második sorban.

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

A for cikluson belül "létrehozzuk" az ábra szerinti második sor elemeit, melyek double * típusúak. A ciklusban 0-tól megyünk 4-ig, egyesével lépkedve. A tm mutatót itt úgy kezeljük, mint egy tömböt, és a tm által mutatott mutatókhoz foglalunk tártetületet, és ráállítjuk őket. Éredemes megfigyelni, hogy mindenekhez i+1-szer 4 bájtot foglalunk le, és a malloc double *-ot ad vissza. Itt is megvizsgáljuk, hogy sikerült-e a foglalás, hanem hibával térünk vissza. Most kész a második sor, és mindenek double * egy harmadik sorban lévő double-ek csopoortjának első elemére mutat, mindenek más csoporthoz.

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

Kiíratjuk a harmadik sor első double csoportjának első elemének a memóriacímét. Majd a for cikluson belül értéket adunk a harmadik sori double-eknek. Az i-vel megyünk a 4-ig, vagyis nr-1-ig, j-vel pedig minden 0-tól i-ig. Az i jelöli a sorok számát, a j pedig az oszlopokat. Mártrix minden eleméhez a sorszám*(szorszám+1)/2+oszlopszám, és ezzel megkapjuk a feladat legelején felvázolt mátrixot, amit a következő for -ban már csak elemenként kiíratunk.

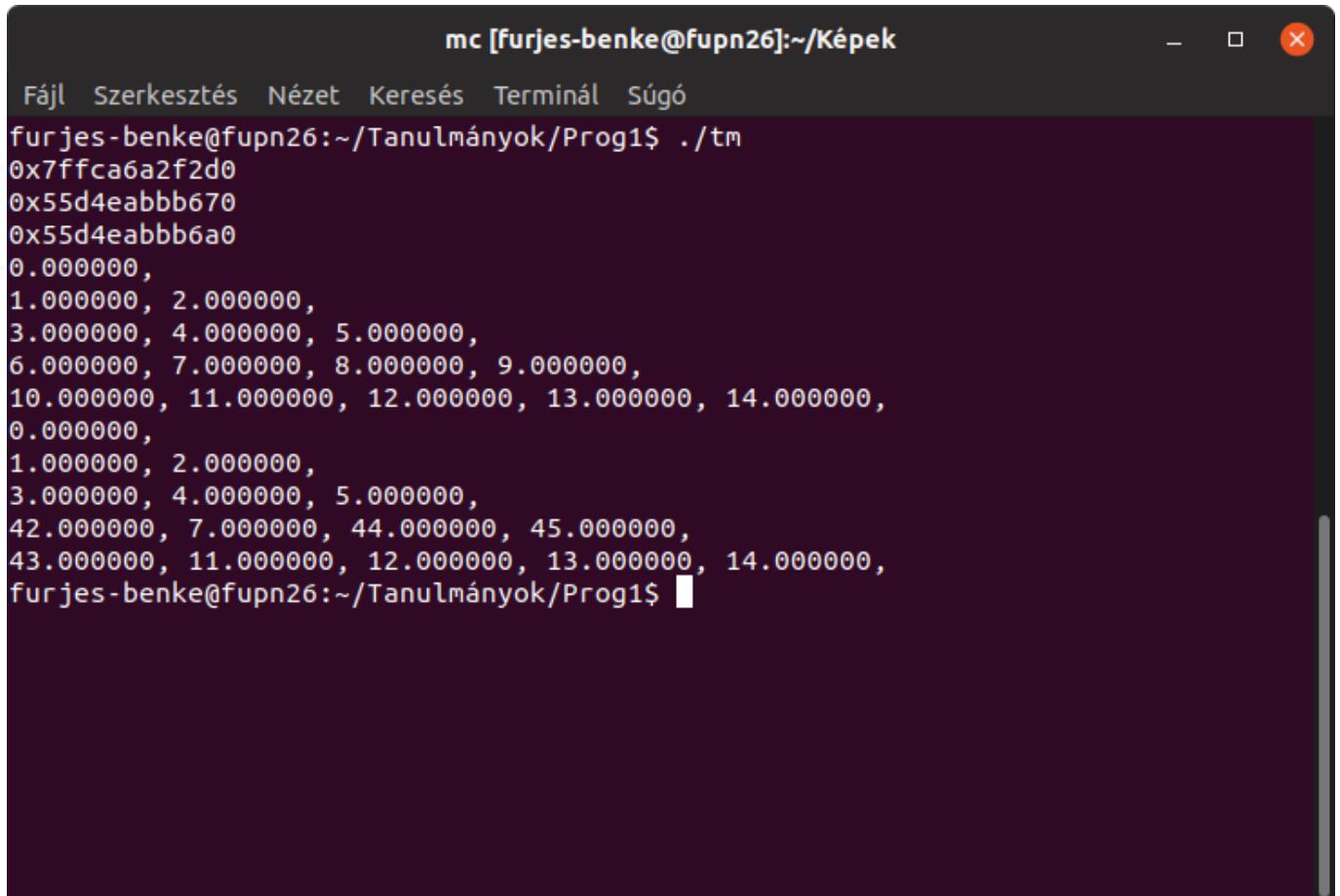
Hogy egy kicsit szokja a szemed a többféle jelölést, nézzük meg az előbb megadott mátrix néhány elemének módosítását.

```
tm[3][0] = 42;
(* (tm + 3))[1] = 43; // mi van, ha itt hiányzik a külső ←
()
* (tm[3] + 2) = 44;
* (* (tm + 3) + 3) = 45;
```

Az elsőt már láttad a gyakorlatban, hogy működik, mivel a programban eddig ezt használtuk, tehát a tm 3. sorának első elemének értékét 42-re módosítjuk. Utána a harmadik sor második elemének az értékét változtatjuk, majd a harmadik sor harmadik elemét, végül pedig a harmadik sor negyedik elemét. Itt is lát-hatod, hogy az első verzió sokkal egyszerűbb a többinél, főleg azoknak, akik már sokat használtak tömböket C-ben. A második lehetőségnél felmerül a kérdés, hogy elhagyható-e a külső zárójel. Elhagyható viszont, így nem a harmadik sorba lesz a módosítás, hanem a 4. sor első eleménél, mivel $* (tm + 3)[1]$ azzal ekvivalens, hogy $* (tm+4)$.

```
mc [furjes-benke@fupn26:~/BHAX/attention_raising/Source/Haromszogmatrix - □ ×
Fájl Szerkesztés Nézet Keresés Terminál Súgó
furjes-benke@fupn26:~/Tanulmányok/Prog1$ ./tm
0x7ffe30b7d700
0x5581c71b9670
0x5581c71b96a0
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
6.000000, 7.000000, 8.000000, 9.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
42.000000, 43.000000, 44.000000, 45.000000,
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
furjes-benke@fupn26:~/Tanulmányok/Prog1$ █
```

4.3. ábra. $(*(tm + 3))[1] = 43$



The screenshot shows a terminal window titled "mc [furjes-benke@fupn26]:~/Képek". The window contains a memory dump of the variable "tm" from a C program. The dump shows the memory address 0x7ffca6a2f2d0, followed by several floating-point numbers: 0x55d4eabbb670, 0x55d4eabbb6a0, 0.000000, 1.000000, 2.000000, 3.000000, 4.000000, 5.000000, 6.000000, 7.000000, 8.000000, 9.000000, 10.000000, 11.000000, 12.000000, 13.000000, 14.000000, 0.000000, 1.000000, 2.000000, 3.000000, 4.000000, 5.000000, 42.000000, 7.000000, 44.000000, 45.000000, 43.000000, 11.000000, 12.000000, 13.000000, 14.000000.

4.4. ábra. $*(tm + 3)[1] = 43$

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása:[itt](#)

Tanulságok, tapasztalatok, magyarázat...

Az EXOR titkosító lényegében a logikai vagyra, azaz a XOR műveletre utal, mely bitenként összehasonlítja a két operandust, és minden 1-et ad vissza, kivéve, amikor az összehasonlított 2 Bit megegyezik, mert akkor nullát. Tehát 2 operandusra van szükségünk, ez jelen esetben a titkosítandó bemenet, és a titkosításhoz használt kulcs. Ideális esetben a ketté mérete megegyezik, így garantálható, hogy szinte feltörhetetlen kódot kapunk, mivel túl sokáig tart annak megfejtése. A mi példánkban természetesen nem lesz ilyen hosszú a kulcs, mivel ki is szeretnénk próbálni a programot. Viszont ha a kulcs rövidebb, mint a titkosítandó szöveg, akkor a kulcs elkezd ismétlődni, ami biztonsági kockázatot rejt magában.

Nézzük is meg ennek a titkosító algoritmusnak a C-beli implementációját, melynek majd a törő verzióját is elkészítjük a későbbiekbén.

```
#define MAX_KULCS 100
#define BUFFER_MERET 256
```

Elsőnek a kulcs méret és a buffer méretének maximumát konstansban tároljuk el, ezek nem módosíthatóak. A szintaxisa is másabb, mint egy változó definiálásánál, itt lényegében azt adjuk meg, hogy mivel helyettesítse a megadott nevet a program a forrásban. Az előre definiált konstansok nevét általában nagy betűvel írjuk, ezzel is elkülönítve a vátozóktól. Nem csak számokat használhatunk konstansként, hanem stringeket, és kifejezéseket is. Érdekessége még, hogy nem program futtatásakor történik meg a behelyettesítés, hanem a már a fordítás alatt, tehát a gépi kód már behelyettesített értékeket tartalmazza.

```
int
main (int argc, char **argv)
```

Újabb érdekesség, hogy a main () definiálása itt egy kicsit másképp zajlik, mivel jelen esetben argumentumokat adunk át neki, az argumentumokat általában a terminálon keresztül adjuk át, amikor futtatjuk. Az argc-vel adjuk át az argumentumok számát, és az argumentumokra mutató mutatókat pedig az argv tömbben tároljuk el.

```
char kulcs[MAX_KULCS];
char buffer[BUFFER_MERET];
```

A main () belül deklarálunk két tömböt, egyikbe a kulcsot tároljuk, a másikban pedig a beolvasott karaktereket, mind a kettőnek a mérete korlátozott, melyet még a #define segítségével adtunk meg.

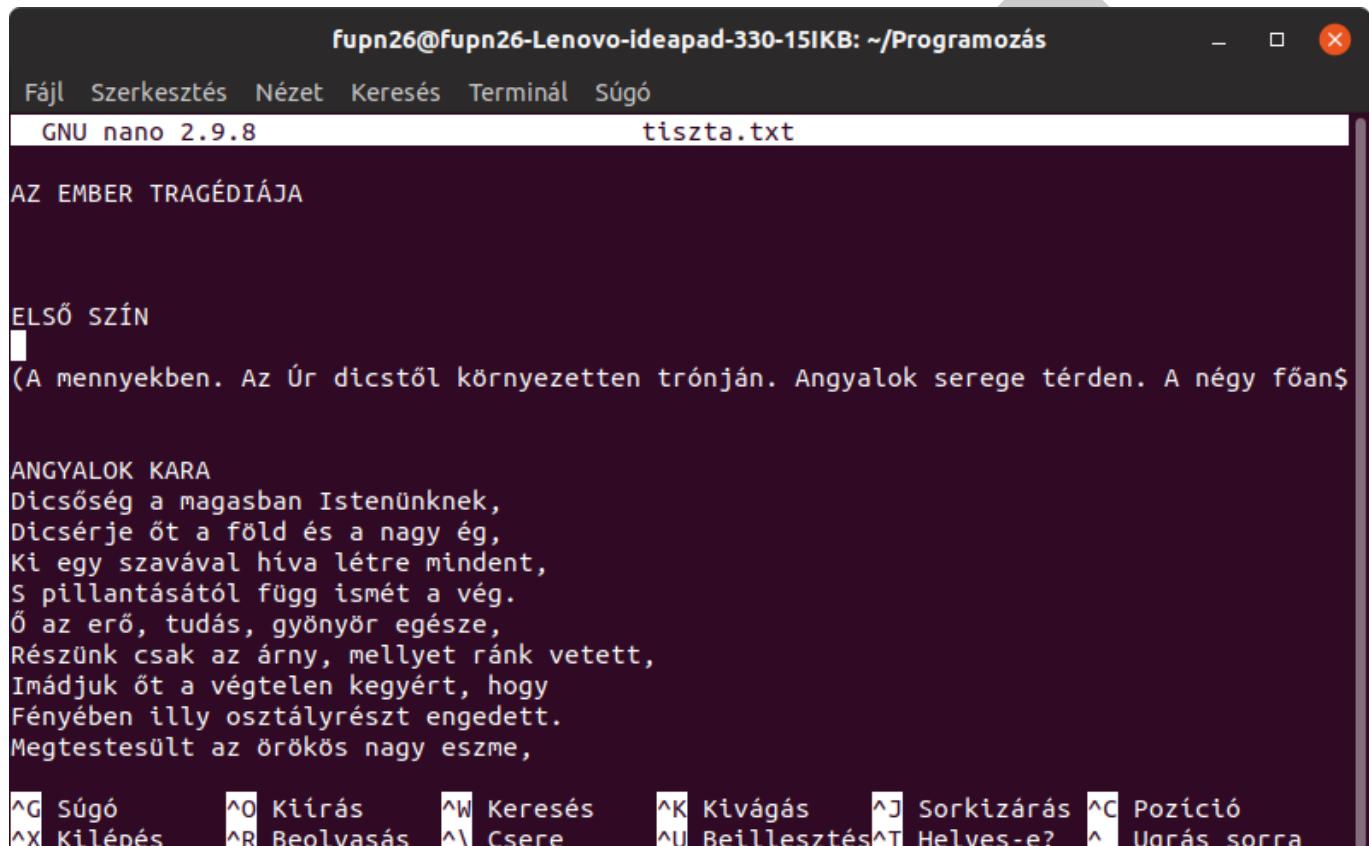
```
int kulcs_index = 0;
int olvasott_bajtok = 0;

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

Definiálunk számlálókat, melyek segítségével bejárjuk majd a kulcs tömböt, és számoljuk a beolvasott bajtokat. A kulcs méretét a strlen () függvénykel kapjuk meg, mely jelen esetben visszadja a második parancssori argumentum hosszát. Ezután a strncpy () függvényel átmásoljuk az argv [1]-ben tárolt sztringet karakterenként a kulcs tömbe "másolja", lényegében mindegyikhez visszaad egy pointert. A MAX_KULCS-csal pedig meghatározzuk, hogy mennyi karaktert msáoljon át.

```
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 cikluson feltétele addig lesz igaz, ameddig a read parancs beolvassa a megadott mennyisésgű bájtokat. A read 3 argumentumot kap, az egyik a file descriptor, ami megadná, hogy honnan szeretnénk beolvasni a bájtokat, jelen esetben a standard inputról olvasunk, a bájtokat a buffer-ben tároljuk egészen addig, ameddig el nem érjük a megadott mennyiséget, amit BUFFER_MERET definiál. A beolvasott bájtok számát adja vissza. Ezután pedig végigmegyünk elemenként a bufferben eltárolt karaktereken és össze EXOR-ozzuk a kulcs tömb megfelelő elemével, majd inkrementáljuk a kulcs_index-et 1-el, mely egészre addig nő, ameddig el nem érjük a kulcs_meret-et, ekkor lenullázódik. Végezetül pedig kiírjuk a buffer tartalmát a standard outputra.



The screenshot shows a terminal window titled "fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás". The window has a dark theme. The title bar includes standard window controls. The menu bar shows "Fájl Szerkesztés Nézet Keresés Terminál Súgó" and "GNU nano 2.9.8" above the file name "tiszta.txt". The main area contains the following text:

```
AZ EMBER TRAGÉDIÁJA

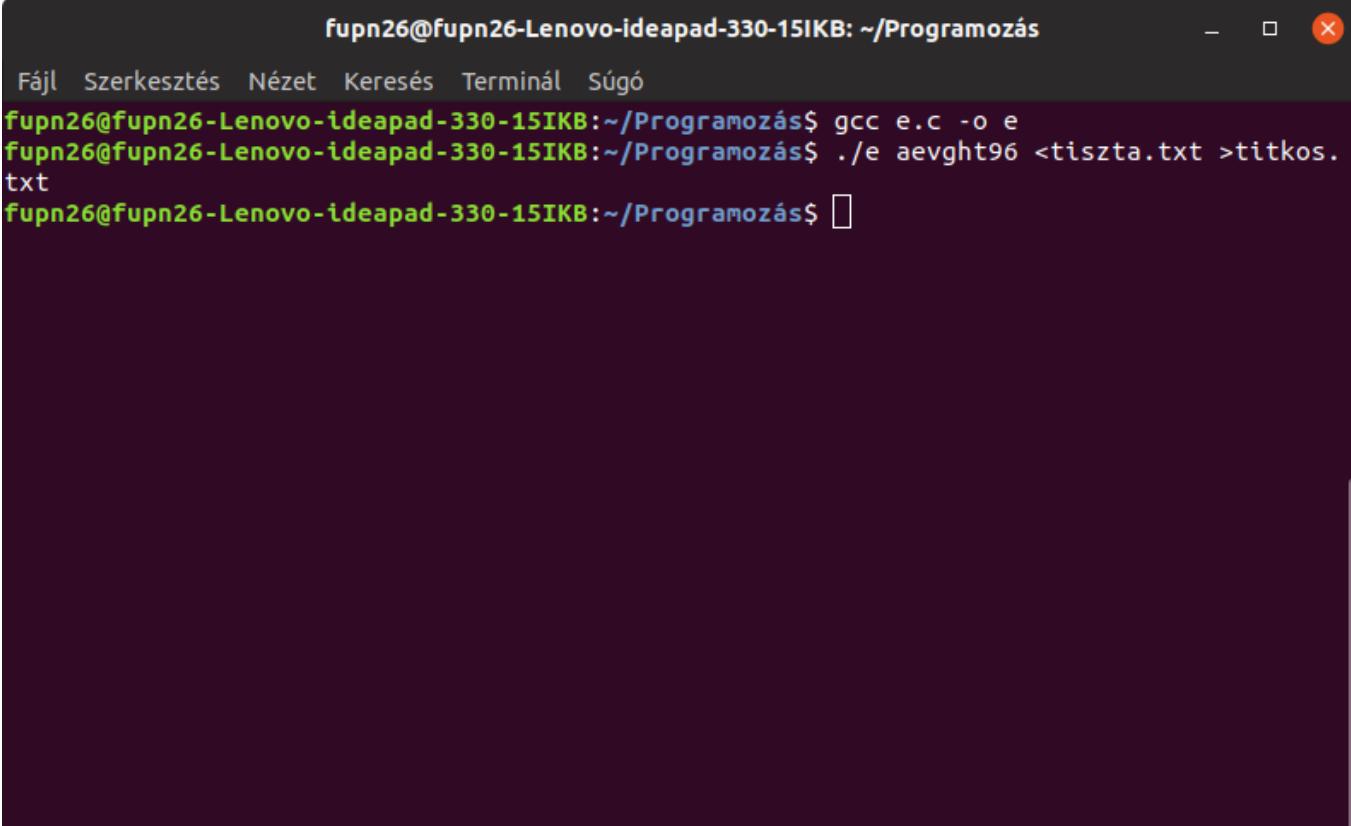
ELSŐ SZÍN
(A mennyekben. Az Úr dicstől környezetten trónján. Angyalok serege térdén. A négy főan$)

ANGYALOK KARA
Dicsőség a magasban Istenünknek,
Dicsérje őt a föld és a nagy ég,
Ki egy szavával híva létre minden,
S pillantásától függ ismét a vég.
Ő az erő, tudás, gyönyör egésze,
Részünk csak az árny, mellyet ránk vetett,
Imádjuk őt a végtelen kegyért, hogy
Fényében illy osztályrészt engedett.
Megtestesült az örökös nagy eszme,
```

The bottom of the terminal shows the nano editor's command line:

```
^G Súgó      ^O Kírás      ^W Keresés      ^K Kivágás      ^J Sorkizáras ^C Pozíció
^Kilépés    ^R Beolvasás  ^\ Csere       ^U Beillesztés ^T Helyes-e? ^Ugrás sorra
```

4.5. ábra. Titkosítandó szöveg



Fájl Szerkesztés Nézet Keresés Terminál Súgó

```
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ gcc e.c -o e
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ ./e aevght96 <tiszta.txt >titkos.txt
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$
```

4.6. ábra. Fordítás és futtatás

The screenshot shows a terminal window with the following details:

- Title bar: fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás
- Menu bar: Fájl, Szerkesztés, Nézet, Keresés, Terminál, Súgó
- Version: GNU nano 2.9.8
- File name: titkos.txt
- Content: The file contains a large amount of encoded text. A tooltip above the cursor indicates "9 sor beolvasva" (9 lines read).
- Bottom status bar: ^G Súgó, ^O Kiírás, ^W Keresés, ^K Kivágás, ^J Sorkizáras, ^C Pozíció, ^X Kilépés, ^R Beolvasás, ^\ Csere, ^U Beillesztés, ^T Helyes-e?, ^ Ugrás sorra.

4.7. ábra. Titkosított szöveg

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása:



Megjegyzés

Ebben a feladatban tutoráltként részt vett: Molnár András Imre, Pócsi Máté.

Tanulságok, tapasztalatok, magyarázat...

Az előző feladatát kell most java-ban megoldani. Mivel a Java a C és a C++ alapján készült, ezért az előző kód implementációja nem nehéz feldata, de természetesen vannak különbségek, melyeket érdemes megemlíteni.

Az első és legfontosabb dolog, hogy a Java a C-vel ellentétben Objektum-orientált programozási nyelv, azaz létre tudunk hozni objektumokat, úgynevezett class-okat melyekkel bizonyos utasításokat tudunk végrehajtani. A C++-ban kicsit arra hasonlít, mintha egy változót hoznánk létre, csak nem egy típust írnunk elé,

hanem a class nevét. De ott a class elkülönül a main() függvénytől, míg a Java-ban szerves részét képezi a classnak, tehát az egész program egy nagy class-ból áll.

```
public class ExorTitkosító {  
    ...  
    ...  
}
```

Tehát ezen belül kell minden deklarálni és definiálni. A class-nak lehet public és private elemei, ezt a függvény deklaráció elé írjuk, jelen esetben nem alkalmazzunk private elemeket, de a lényeg az, hogy azokhoz nem férünk hozzá közvetlenül a classon kívül, csak akkor ha a class tartalmaz egy public függvényt, amivel ki tudjuk nyerni az értéket.

Elsőnek nézzük meg a main tartalmát:

```
public static void main(String[] args) {  
  
    try {  
  
        new ExorTitkosító(args[0], System.in, System.out);  
  
    } catch(java.io.IOException e) {  
  
        e.printStackTrace();  
  
    }  
}
```

Az main fejléce egy kicsit furán néz ki, mivel egy kicsit bonyolultabb, mint azt C-ben megszoktuk. A public annyit tesz, hogy elérhető a class-on kívül is. A static azt jelenti, hogy a a main része a class-nak, de egy külön álló objektum, nem része egyik beágyazott objektumnak sem. A void-ot már ismerjük, tehát ennek a main-nek nincs visszatérési értéke. Ráadásul itt is képesek vagyunk parancssori argumentumokat átadni a programnak a String[] args segítségével. Egy másik újdonság a try és a catch használata, mely lényegében a Java-ban és a C++-ban a kivételkezeléshez nélkülözhetetlen. A try blokk tartalmazza az utasításokat, ha valami hiba történik, akkor dobu k egy hibát, a catch "elkapja" és visszaad egy hibaüzenetet a terminálba. Jelen esetben ha nem adunk meg kulcsot, akkor kapunk hibát. A try-ban tárhelyet foglalunk az ExorTitkosító függvénynek, melynek átadjuk a kulcsot, a bemenetet és a kimenetet.

```
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 olvasottBájtak = 0;  
  
    while((olvasottBájtak =  
           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);

    }

}

```

Az ExorTitkosító függvény a már említett argumentumokat kéri, és dobja throws a hibát, ha ez nem teljesül. A függvényen belül pedig a C kódban megismert utasításokat hajtjuk végre. Itt látható egy primitív adattípus, a byte, mely 8-bit-ból áll. Jelen esetben egy byte-okból álló tömböt hozunk létre kulcs és buffer néven. Az első argumentumból a getBytes() függvény segítségével olvassuk be a kulcsot a kulcs tömbbe. A buffer tömbnek pedig foglalunk egy 256 bájt-ból álló területet a memóriában. Majd a definiáljuk a már jól ismert változókat a kulcs tömb bejárásához, és a beolvasott bájtok számlálására. A while ciklus addig megy, ameddig be nem olvasunk a buffer méretű karakterSORozatot, vagy már nem tudunk többet beolvasni. Majd a beágyazott for ciklussal elemenként összeexorozzuk a buffer tartalmát a kulccsal, és növeljük a kulcsindexet a már megszokott módon a % operátorral, mely a maradékos osztást jelenti. Ennek következtében ha elérjük a kulcs tömb hosszát, akkor lenullázódik.

Végezetül lássuk, hogy hogyan kell futtatni ezt a programot: (A képeken létható parancsok futtaása előtt telepíteni kell ezt: sudo apt-get install openjdk-8-jdk)

```

fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás
Fájl Szerkesztés Nézet Keresés Terminál Súgó
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ javac ExorTitkosító.java
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ java ExorTitkosító 00000001 <tiszta.txt >titkos.txt
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ 
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ 
Fájl Szerkesztés Nézet Keresés Terminál Súgó
GNU nano 2.9.8      titkos.txt
[10 sor beolvasva]
^G Súgó   ^O Kijárás   ^W Keresés   ^K Kivágás
^K Kilépés  ^R Beolvásás  ^A Cseré  ^U Bellesztés

```

4.8. ábra. Titkosított szöveg

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...



Megjegyzés

Ebben a feladatban tutorként és tutoráltként részt vett: Imre Dalma, György Dóra.

Az előbbi feádatokban láthattad, hogy hogyan lehet titkosított szövegeket készíteni az EXOR titkosítás segítségével. Most ennek az ellentétét kell megcsinálnunk, ami egy kicsit trükkösebb, és talán nem is tökéletes, de az előző feladatban generált titkos szöveg feltörésére alkalmas lesz.

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

Ezúttal is meghatározunk bizonyos konstansokat, ebből a kulcs_meret érdekes, mert feltételezzük, hogy a kulcs 8 elemből áll, már itt látni, hogy ez nem lenne túl hatékony a való életben.

```
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;  
}
```

Az `atlagos_szohossz` függvényel kiszámítjuk a bemenet átlagos szóhosszát, argumentumként adjuk egy tömböt, és annak a méretét. Majd egy `for` ciklussal bejárjuk, és minden elem után hozzáadunk 1-et az `sz` változóhoz. Visszatérési értékként pedig a tömb méretének és a számlálónka a hányadosát adjuk.

```
int  
tiszta_lehet (const char *titkos, int titkos_meret)  
{  
    // a tiszta szöveg valszeg tartalmazza a gyakori magyar ←  
    // szavakat  

```

```
        double szohossz = atlagos_szohossz (titkos, ←
            titkos_meret);

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

    }
```

A tiszta_lehet függvény az átlagos szóhossz segítségével vizsgálja, hogy a fejtésben lévő kód tiszta-e már. Itt meg kell felelni az átlagos magyar szóhossznak, és a leggyakoribb szavakat tartalmaznia kell. Felmerül a kérdés, hogy mi történik akkor, ha ezeknek nem felel meg a törni kívánt szöveg? Sajnos akkor nem tudjuk feltörni, tehát ez egy újabb gyengesége a programunknak.

```
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;

    }

}
```

Az exor függvény ugyan azt csinálja, mint az EXOR titkosító, mivel ha valamit kétszer EXOR-ozunk, akkor visszakapjuk önmagát. Lényegében ezzel állítjuk vissza a tiszta szöveget. Ez argumentumként megkap egy lehetséges kulcsot, annak méretét, és magát a titkosított szöveget, annak méretével együtt.

```
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);

}
```

Az exor_tores hívja a korábban definiált függvényeket, és 0-át vagy 1-et ad vissza, attól függően, hogy tiszta-e már a szöveg.

```
char kulcs[KULCS_MERET];
char titkos[MAX_TITKOS];
char *p = titkos;
int olvasott_bajtok;
```

Mostmár átérhetünk a main belüli deklarációkra, definíciókra. Elsőnek dekláralunk egy kulcs[] tömböt, és egy titkos[] tömböt. Ezek mérete a fentebb már rögzített értékekkel lesz azonos. Majd definiálunk egy mutatót, mely a titkos[] tömbre mutat, és deklaráljuk a beolvasott bajtok számlálóját.

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

A while ciklussal addig olvassuk a bajtokat, ameddig a buffer be nem telik, vagy a bemenet végére nem érünk.

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

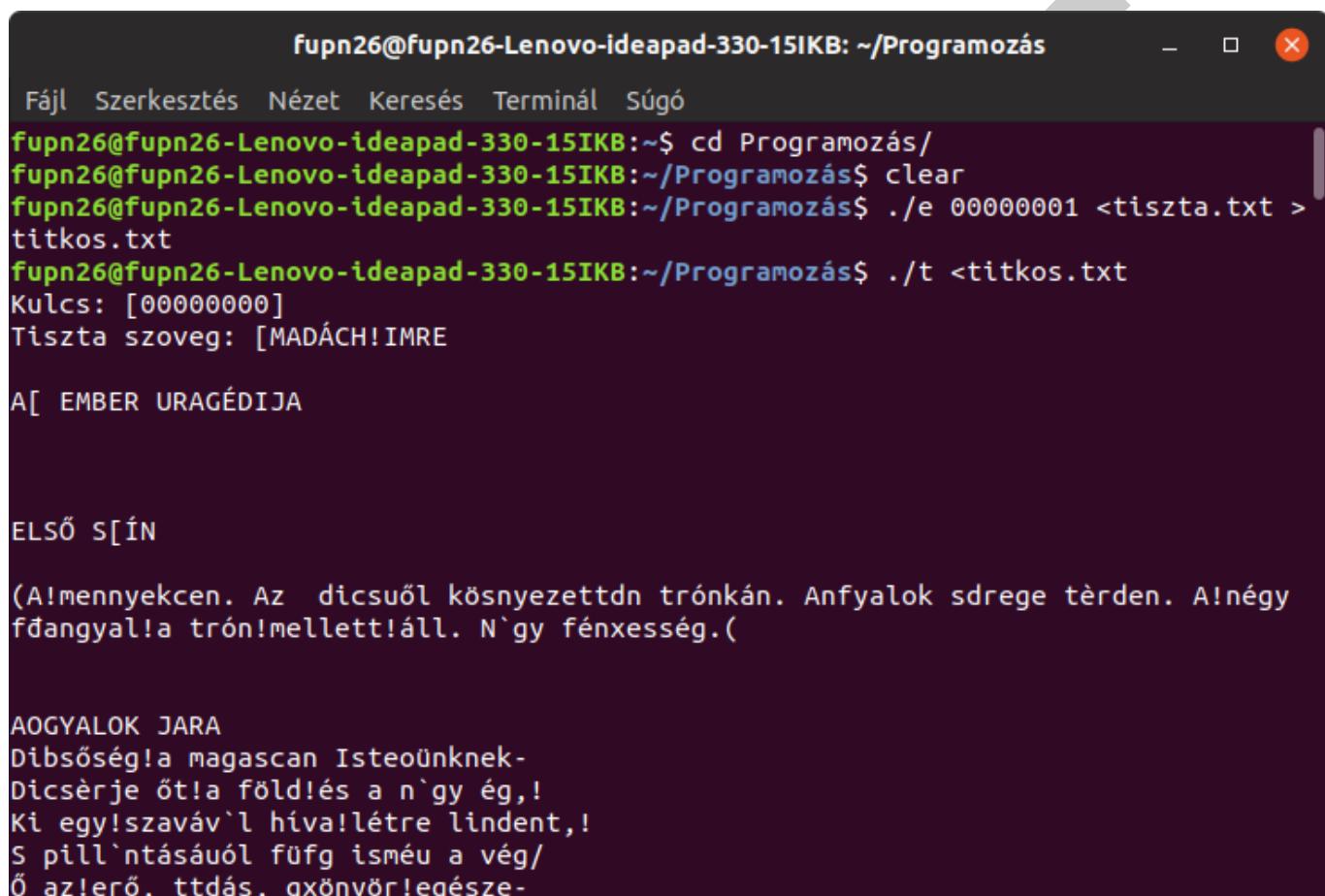
Ezzel a for ciklussal kinullázzuk a buffer megmaradt helyeit, és utána pedig előállítjuk az összes lehetséges kulcsot:

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

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

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

Végig futatjuk az összes lehetőségen a `for` ciklust, majd meghívjuk az `exor_tores` függvényt. Ha ez igazat ad, tehát a visszatérési érték 1, akkor kiíratjuk az aktuális kulcsot és a feltört szöveget. Ahogy látod ez csak olyan kódokat tud feltörni, amit számokkal kódoltunk. Ezután pedig újra meghívjuk az `exor` függvényt, ezzel elkerülve a második buffer létrehozását. Az előző feladatban én a betűket és számokat is használtam, ezt is ki lehetne bővíteni, hogy fel tudjuk törni azt a kódot, de az a baj, hogy nagyon sokáig tartana. Tehát ennél a megoldásnál maradva újratitkosítottam az `tiszta.txt`-t



The terminal window title is `fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás`. The command history shows:

```
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~$ cd Programozás/
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ clear
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ ./e 00000001 <tiszta.txt >
titkos.txt
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás$ ./t <titkos.txt
Kulcs: [00000000]
Tiszta szöveg: [MADÁCH!IMRE

A[ EMBER URAGÉDIJA

ELSŐ S[ÍN

(A!mennyekcen. Az dicsuől kösnyezettdn trónkán. Anfyalok sdrege tèrden. A!négy
f!dangyal!a trón!mellett!áll. N`gy fénxesség.(

AOGYALOK JARA
Dibsőség!a magascan Isteoünknek-
Dicsérje öt!a föld!és a n`gy ég,! 
Ki egy!szaváv`l hiva!létre lindent,! 
S pill`ntásauól füfg isméu a vég/
Ő az!erő, ttdás, gxönyör!egésze-
```

4.9. ábra. Törés

A `.c` fordítva és futtatva a képen látható módon folyamatosan kapjuk a lehetséges megfejtéseket, fontos hogy nem kell végig várni a folyamatot, mert az sokáig tart, `Ctrl+c`-vel meg tudod állítani. Jelen esetben láthatod, hogy sikerült megtalálnia a megfelelő kódot.

fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás

Fájl Szerkesztés Nézet Keresés Terminál Súgó

Ér meg nel únod wéges vøgtelen-
Hogy `z a nöua mindif úgy mdgyen.
Léltó-d illyen!aggasty;nhoz e! wøs♦]
Kulcs: [00000001]
Tiszta szoveg: [MADÁCH IMRE

AZ EMBER TRAGÉDIÁJA

ELSŐ SZÍN

(A mennyekben. Az Úr dicstől környezetten trónján. Angyalok serege térdén. A négy főangyal a trón mellett áll. Nagy fényesség.)

ANGYALOK KARA

Dicsőség a magasban Istenünknek,
Dicsérje őt a föld és a nagy ég,
Ki egy szavával híva létre minden,
S pillantásától függ ismét a vég.
Ő az erő, tudás, gyönyör egésze,
Részünk csak az árny, mellyet ránk vetett,
Imádjuk őt a végtelen kegyért, hogy

4.10. ábra. Törés

4.5. Neurális OR, AND és EXOR kapu

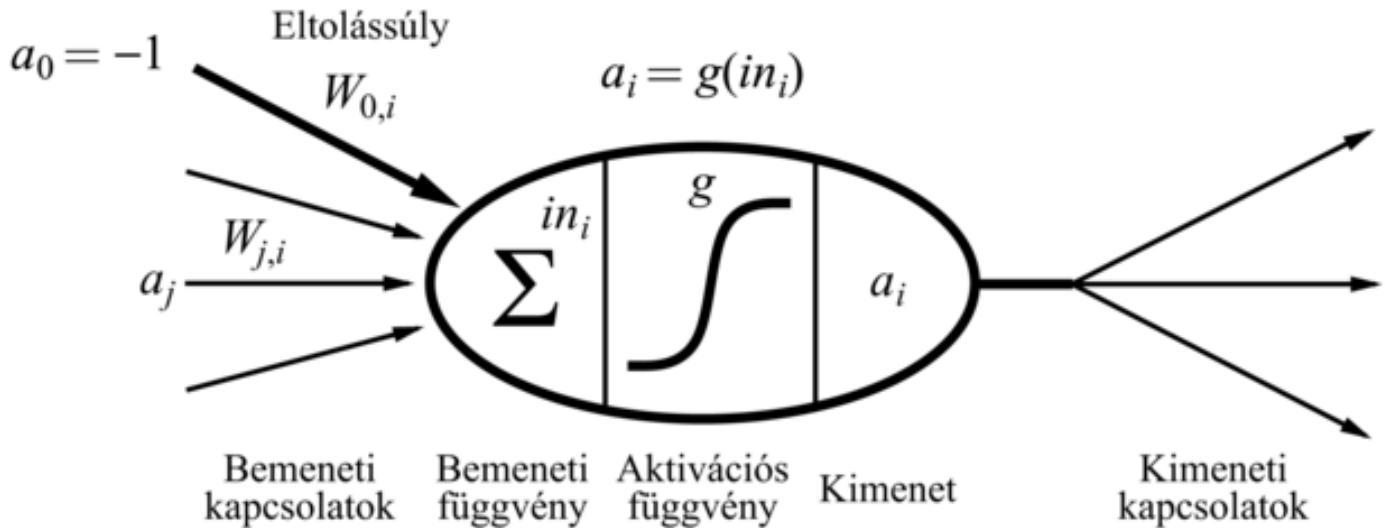
R

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

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

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban újra visszatérünk a Monty Hall problémánál megismert R nyelvhez. Segítségével neurális hálózatot fogunk létrehozni, mely képes "tanulni", és megközelíteni az átalunk megadott megfelelő értékeket. A hálózat a nevet a neuronról kapta, mely gyunk egy sejtje. Feladata az elektromos jelek összegyűjtése, feldolgozás és szétterjesztése. Az a feltételezés, hogy az agyunk információfeldolgozási képességét ezen sejtek hálózata adja. Éppen emiatt a mesterséges intelligencia kutatások során ennek a szimulálást tüzték ki célul. A neuron matematikai modeljét McCulloch és Pitts alkotta meg 1943. Ezt mutatja a következő ábra:



4.11. ábra. Neuron

A lényeg, hogy a neuron akkor fog tüzelni, ha a bemenetek súlyozott összege meghaladnak egy küszöbot. Az aktivációs függvény adja meg a kimenet értékét.

Ezt a modellt fogjuk implementálni egy R programba. Az első részben a logikai vagyot tanítjuk meg a neurális hálózatnak, mely 1-et ad vissza, kivéve, ha mind a két operandusa 0, mert akkor 0-át.

```
library(neuralnet)

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

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

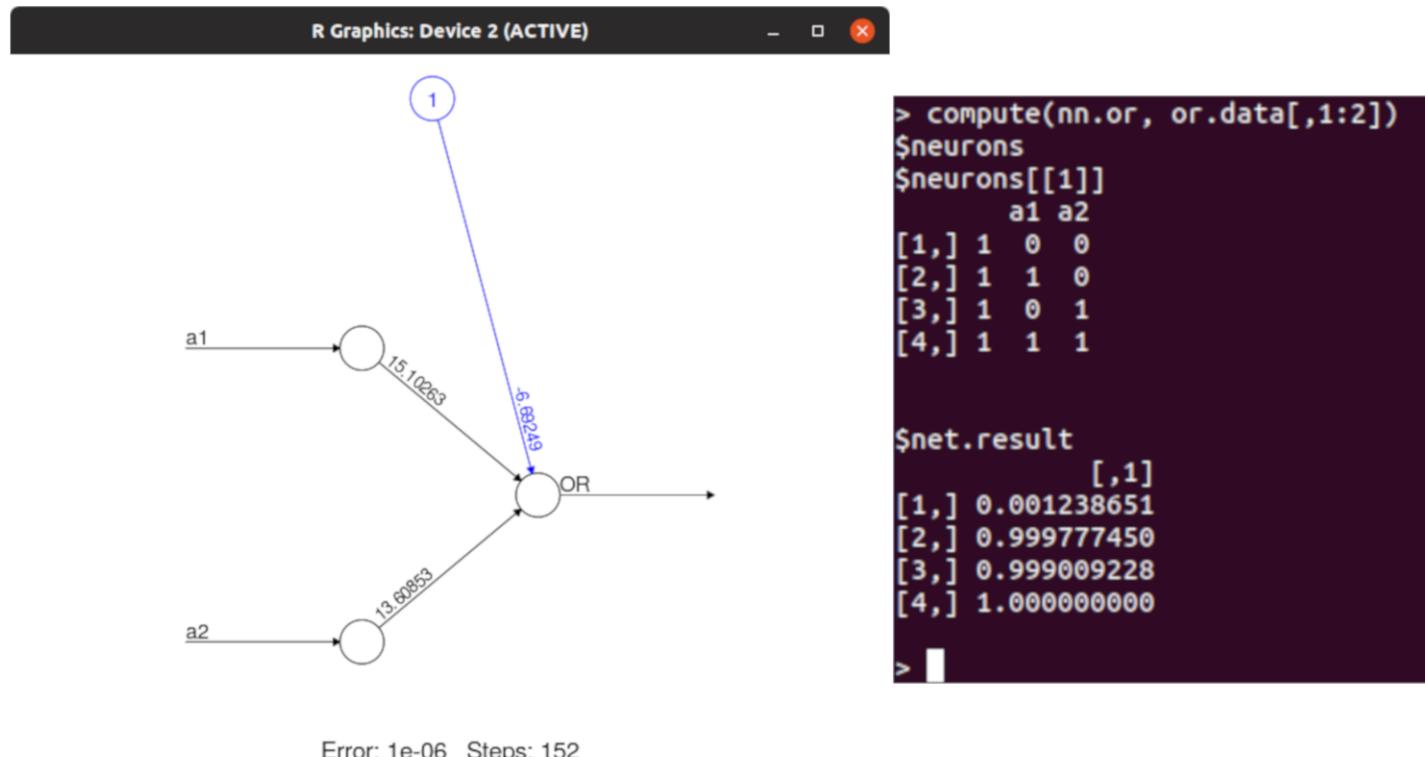
nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE, ←
                   stepmax = 1e+07, threshold = 0.000001)

plot(nn.or)

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

A neuránet könyvtárat kell használni a feladat megoldásához. Az a1, a2, OR változókhöz hozzárendeljük a megfelelő értékeket, amit szeretnénk betanítani a programnak. Majd az or.data-ban tároljuk el ezeket a változókat, melyek a betanítás alapjául fognak szolgálni. Az nn.or változó értékéül pedig a neuralnet függvény visszatérési értékét adjuk. Ennek a függvénynek több argumentuma van, ezeket vegyük gyorsan sorra. Az első elem maga a formula, amit meg kell tanulni a programnak, majd jön egy minta, mely alapján a tanulást végzi, a harmadik argumentum a rejtegt neuronok számát adja meg. A linear.output egy logikai változó, melynek értékét TRUE-ra kell állítani, ha azt szeretnénk, hogy az aktívációs függvény ne fusson le a kimeneti neuronokra. A stepmax adja meg a maximum lépésszámát a neuron háló tanulásának, mely befejeződik, ha elérjük ezt az értéket. A threshold pedig számérték, mely meghatározza a hiba részleges deriváltjainak küszöbértékét, a tanulás megállási kritériumaként funkcionál. Ezután pedig plot

kirajzoltatjuk a neurális háló tanulási folyamatának egy állapotát. A neuralnet minden bemenethez kiszámol egy súlyozást, amellyel megszorozza a bemenet értékét. Majd pedig kiszámítjuk a logikai művelet neurális háló szerinti értékét, és összevetjük a referencia értékkel.



4.12. ábra. OR

A jobb oldali kis ablakban láthatod, hogy egészen jó közelítéssel sikerült visszaadnia a megfeleő értékeket a programnak. Ugyan ezt megismétljük az AND logikai operátorral is, mely akkor lesz igaz, azaz 1, ha mind a 2 operandusa 1, amúgym hamis.

```

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

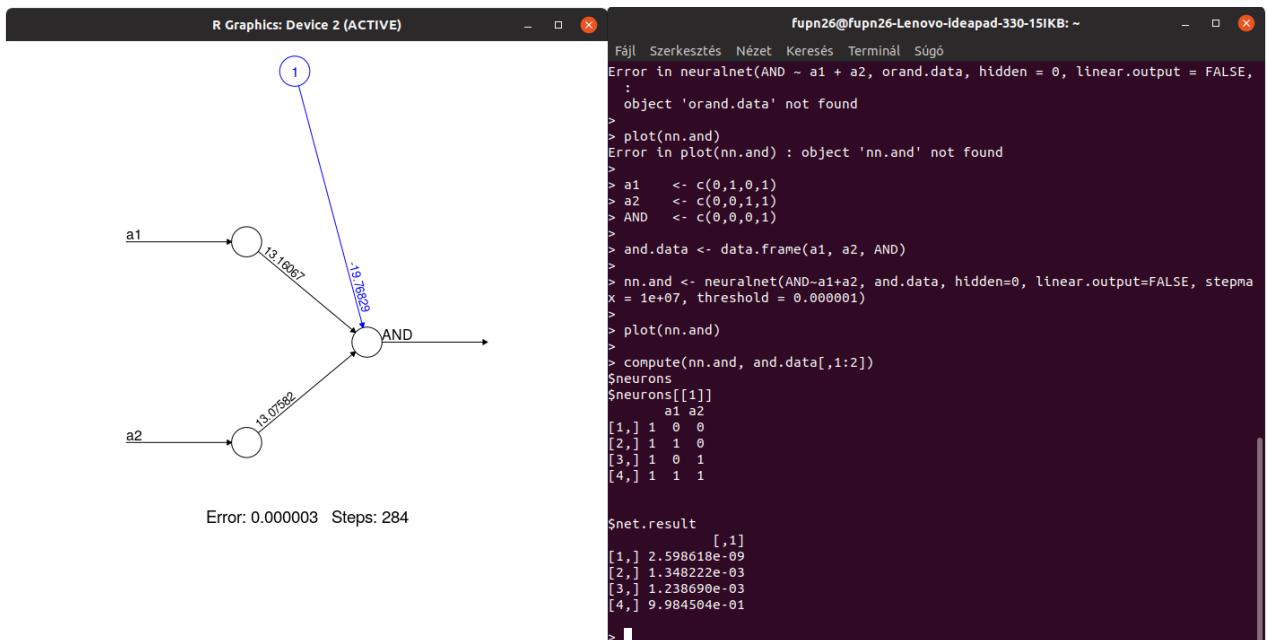
and.data <- data.frame(a1, a2, OR, AND)

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])

```



4.13. ábra. AND

Ezután pedig jönne az EXOR, viszont ez már nem annyira egyszerű. Amikor régen ezt a technológiát kitalálták, és az EXOR nem működött, sokan elpártoltak tőle. Majd a kor nagy matematikusai megfejtették, hogy nem lehetetlen feladat, csak egy apróságra van szükség, létre kell hozni a rejtett neuronokat, melyek segítik a tanulást.

```

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

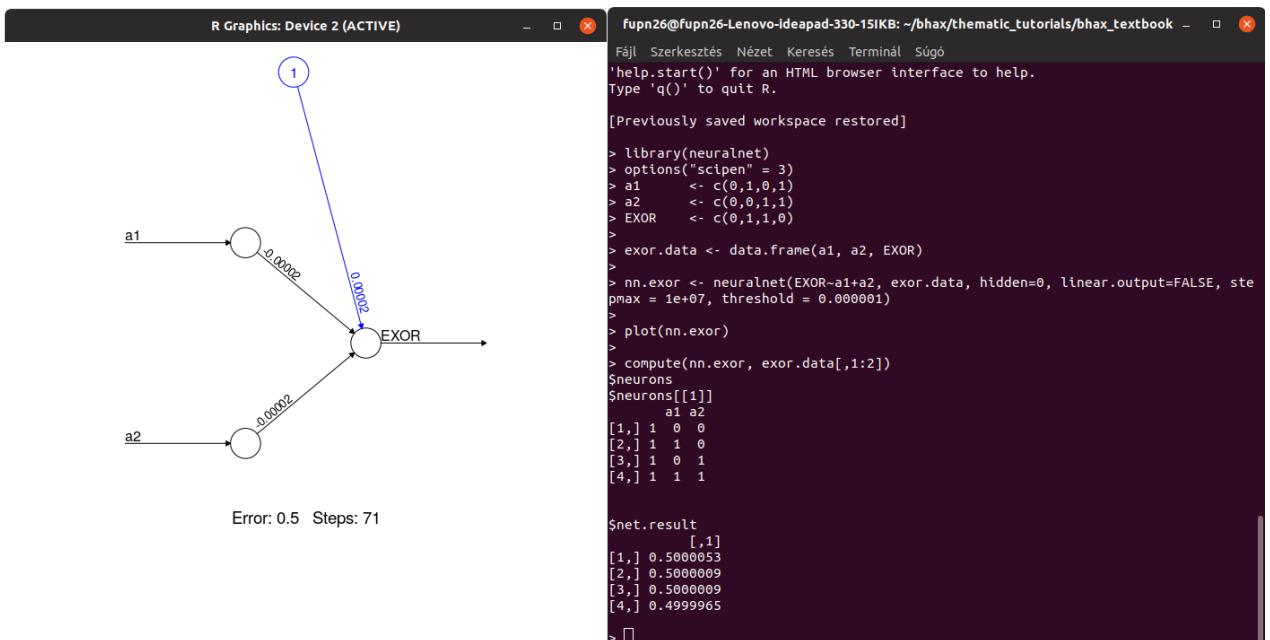
exor.data <- data.frame(a1, a2, EXOR)

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])

```



4.14. ábra. EXOR első próba

Látható, hogy nagyon nagy a hibaarány, és még az eredmények is tévesek lettek, mindegyik 0,5 körül volt, az 1 és a 0 helyett. A `neuralnet` `hidden` argumentuma 0-ra volt állítva ebben az esetben, tehát nem használtunk rejtett neuronokat. Ha ezt átállítjuk, akkor a következőt kapjuk:

```

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

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

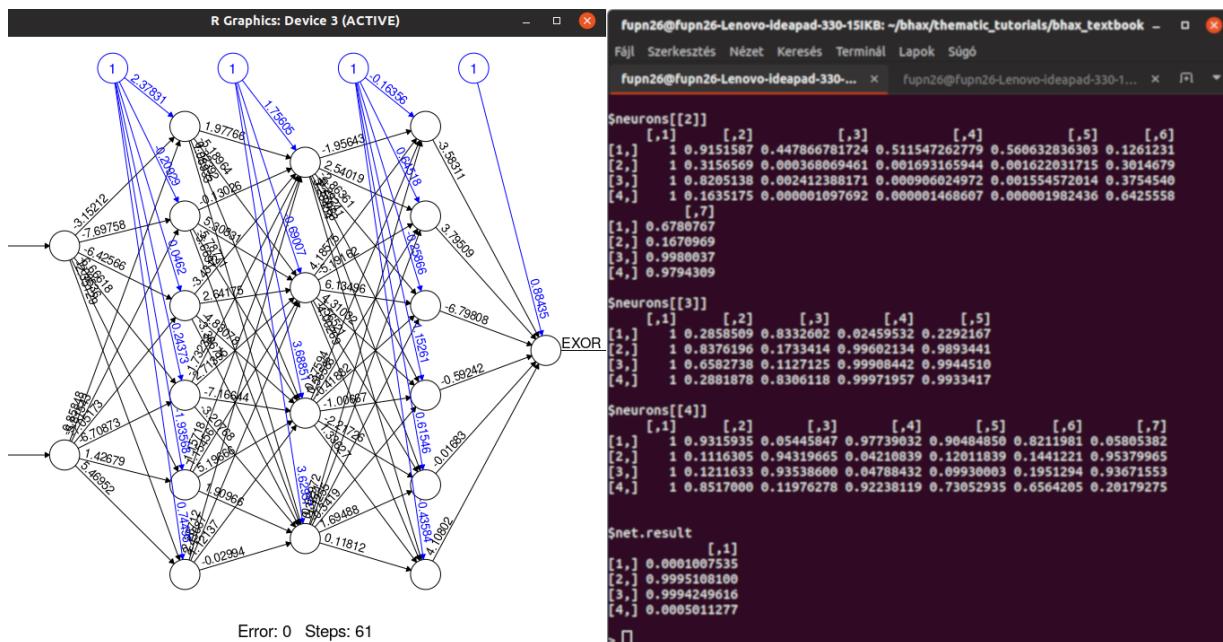
nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=c(6, 4, 6), linear. ←
  output=FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

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

```

Módosítottuk a rejtett neuronok számát, elsőnek 6, majd 4 és végül megint 6 neuront hozunk létre. Lássuk az eredményt:



4.15. ábra. EXOR második próba

Láthatod, hogy sikerült, megkaptuk a helyes eredményeket, bár az ábra jóval bonyolultabb lett a rejtett neuronok miatt.

4.6. Hiba-visszaterjesztéses perceptron

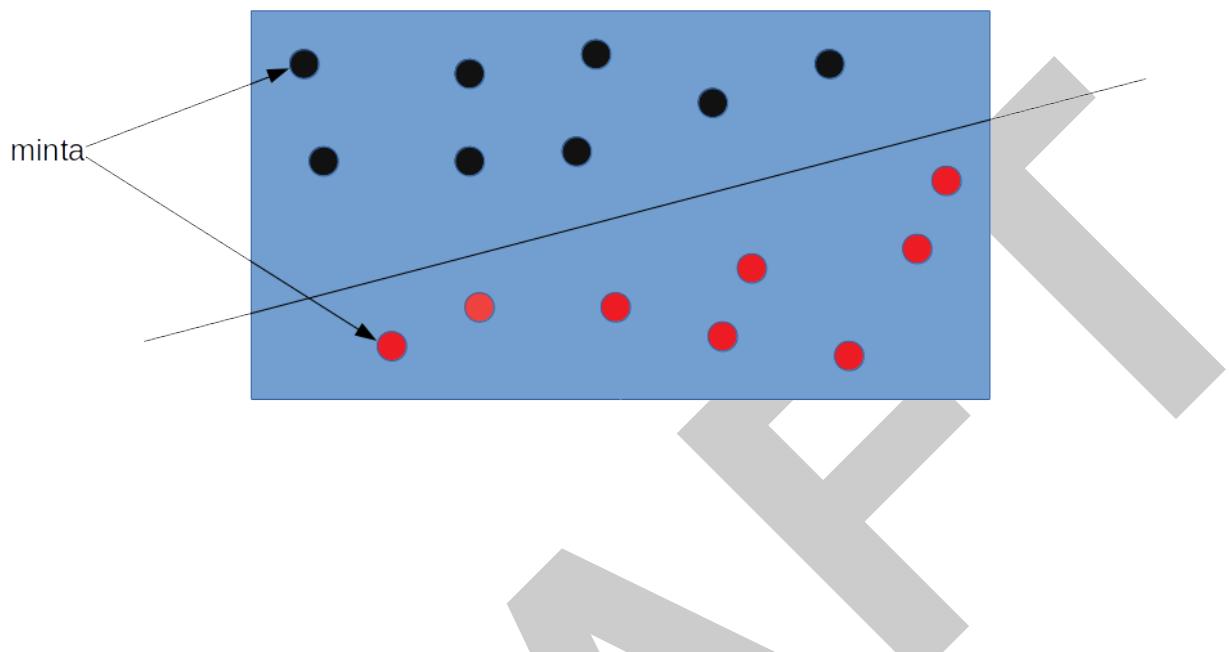
C++

Megoldás videó: <https://youtu.be/XpBnR31BRJY>

Megoldás forrása: <https://github.com/nbatfai/nahshon/blob/master/ql.hpp#L64>

Tanulságok, tapasztalatok, magyarázat...

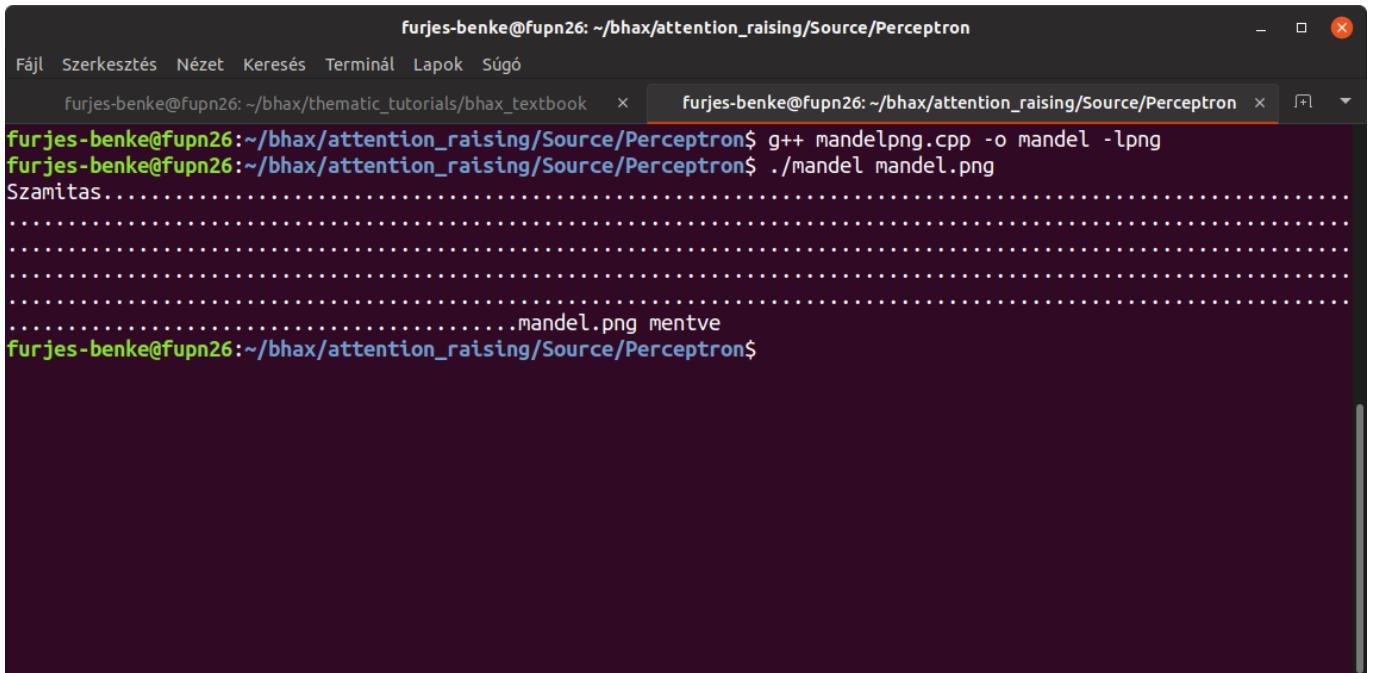
Ebben a feladatban folytatjuk a elmélkedést a neuron hálózatokra, ezen belül perceptronokról lesz szó. Ez egy algoritmus amely a számítógépnek "megtanítja" a bináris osztályozást. Ide is berakhatnám az előző fejezetben lévő képet a neuronról, ami egy bemenetet kap, és egy bizonyos pontot elérve "tüzel", ad egy kimenetet. Itt is hasonló dologról van szó:



4.16. ábra. Perceptron bemenet

Tehát van egy halmaz amiben vannak piros és fekete pontok, a fekete pontok a vonal felett vannak, a pirosak pedig alatta. Adok a perceptronnak bemenetként egyet-egyet minden a kettőből, és a képes lesz megmondani a többite, hogy a vonal felett van-e, vagy alatta. Ezért nevezzük bináris osztályozásnak, mert van a vonal felettiek osztálya és az alattiaké, ez a kapcsolat könnyen reprezentálható eggyel és nullával.

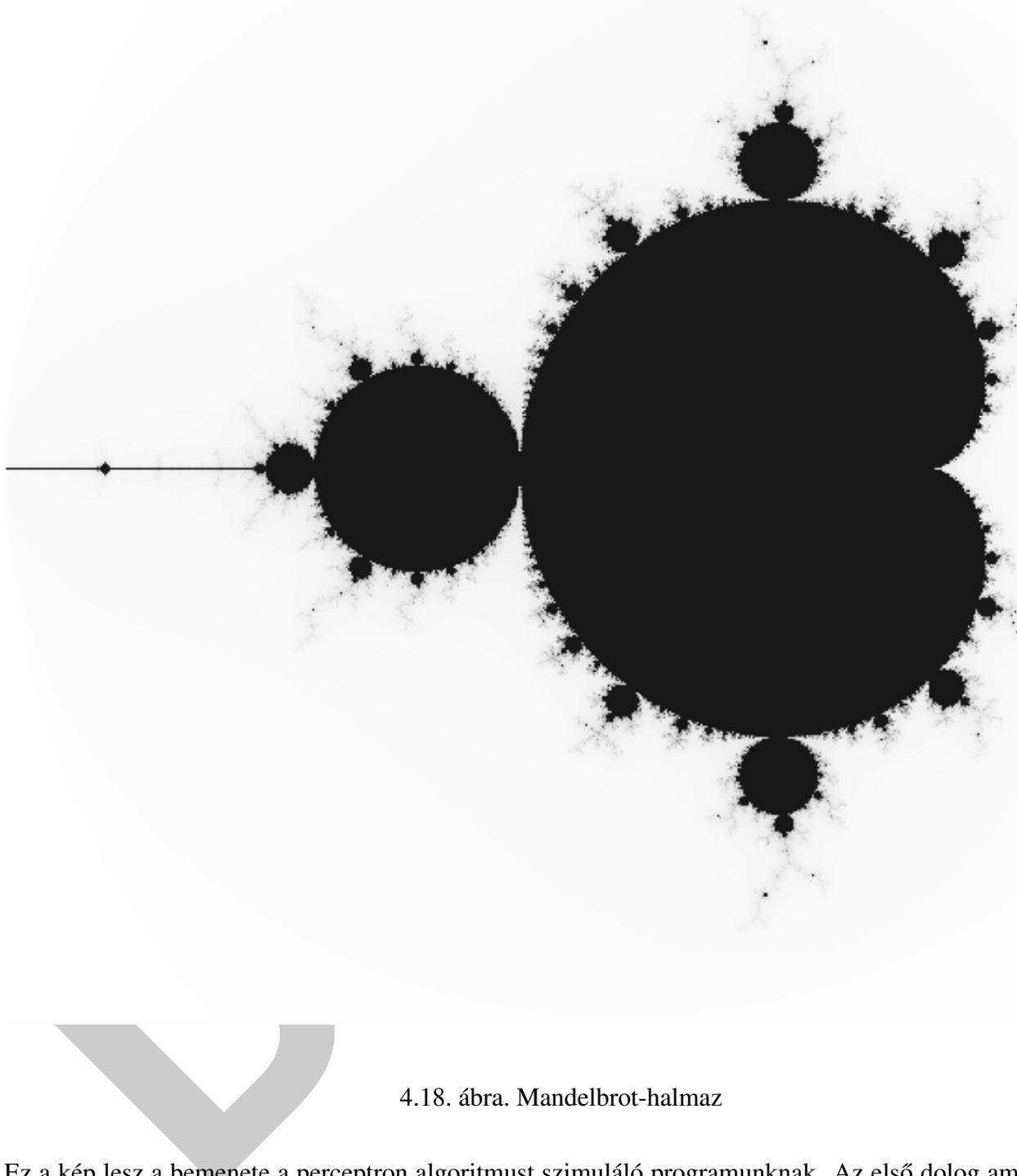
Akkor lássuk magát a programot. Most C++-ban fogunk dolgozni, melyről tettem már említést a Java-s feladatban, szóval annyit már tudsz, hogy ez is egy objektum-orientált nyelv, szóval a class-okat képtelenség lesz elkerülni. 3 fájlre lesz szükség, abból az egyikkel, `mandelpng.cpp`-vel most nem foglalkoznánk, hiszen a következő fejezetben pont erről lesz szó. Egyelőre legyen elég annyi, hogy ezzel tudunk készíteni egy képet, ami a Mandelbrot halmazt ábrázolja.



The screenshot shows a terminal window with two tabs. The left tab is titled 'furjes-benke@fupn26: ~/bhax/thematic_tutorials/bhax_textbook' and the right tab is titled 'furjes-benke@fupn26: ~/bhax/attention_raising/Source/Perceptron'. The user runs the command 'g++ mandelpng.cpp -o mandel -lpng' in the right tab, followed by './mandel'. The output shows the file 'mandel.png' being created and saved. The terminal window has a dark background with light-colored text.

4.17. ábra. mandelpng.cpp fordítása és futtatása

Mivel a program tartalmazza a png.hpp header fájlt, ezért van szükség a -lpng kapcsolóra. A png.hpp-t a **sudo apt-get install libpng++-dev tudod telepíteni.**



4.18. ábra. Mandelbrot-halmaz

Ez a kép lesz a bemenete a perceptron algoritmust szimuláló programunknak. Az első dolog amit nagyon fontos, hogy most 2 fájlból áll a programunk. A `ml.hpp` és `main.cpp`-re lesz szükségünk. Az `ml.hpp` tartalmazza a Perceptron class-t, ezzel a `main.cpp` sokkal átláthatóbb. Ezt technikát gyakran használják, szóval érdemes megtanulni. Magát a class-t most nem vesézzük ki, helyette vessünk egy pillantást a `main`-re.

```
#include <iostream>
#include "ml.hpp"
```

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

int main (int argc, char **argv)
{
    ...
}
```

Itt láthatod, hogyan kell az ml.hpp-t includálni, maga a main fejrésze pedig az EXOR-nál már jól megszokott felépítést követi.

```
    png::image<png::rgb_pixel> png_image (argv[1]);
```

Ezzel létrehozzunk egy üres png-t, melynek mérete megegyezik a bemenetként kapott fájl méretével. Ehhez van szükség a png.hpp headerre. Egy fontos dolog még, hogy miért van szükség a dupla kettőpontra. A C++-ban létezik egy olyan fogalom, hogy névterek. Erről még fogunk szót ejteni, de gyelőre annyi elég lesz róla, hogy a png.hpp-ben használt függvények, változók elé oda kell rakni a png:: -ot. Ugyan így van ez az iostream-ben lévő cout-tal is, mely a standard kimenetre írja ki azt amit szeretnénk. Előtte az std:: prefixet használjuk. Lényegében ez azért hasznos, mert ha lenne a png.hpp-ben és az iostream-ben is cout , akkor ezzel meg tudjuk őket különböztetni. Persze kicsit hosszú mindegyik elé odaírni, és lesz is rá majd megoldás, de ennek használatát egyelőre kerüljük.

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

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

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

A kép méretét eltároljuk egy változóban, majd létrehozunk felhasználó által definiált típust, ez a Perceptron, melyet a ml.hpp Perceptron osztályában találunk. Lényegében itt adjuk meg a rétegek számát, jelen esetben ez 3, majd azt adjuk meg, hogy hány darab neuront szeretnénk az egyes rétegekben. Az utolsóba csak 1-et rakunk, mely az eredményt adja majd. Definiálunk még egy double* pointert, melyet size-zal megegyező memóriaterületre állítunk rá.

```
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;
```

Az egymásba ágyazott for cílusok segítségével az újonan lefoglalt tárba másoljuk bele a beolvasott kép pixeleinek piros komponensét.

```
double value = (*p) (image);
```

Itt meghívjuk a Perceptron class () operátorát, mely vissza fogja adni az nekünk szükséges eredményt. Végezetül ezt már csak kiíratjuk a cout-tal.

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

Futassuk:

The screenshot shows a terminal window with two tabs. The left tab is titled 'furjes-benke@fupn26: ~/bhax/thematic_tutorials/bhax_textbook' and the right tab is titled 'furjes-benke@fupn26: ~/bhax/attention_raising/Source/Perceptron'. The command entered in the right tab is 'g++ ml.hpp main.cpp -o perc -lpng -std=c++11'. The output shows the compilation was successful, and the command './perc mandel.png' was run, resulting in a value of '0.73102'. The prompt 'furjes-benke@fupn26:~/bhax/attention_raising/Source/Perceptron\$' is visible at the bottom.

4.19. ábra. Fordítás és futtatás

Fontos figyelembe venned, hogy nem minden fogja ugyanazt az értéket adni, szóval nem kell kétségbe esni, ha nem ugyan az jön ki, mint a képen.



5. fejezet

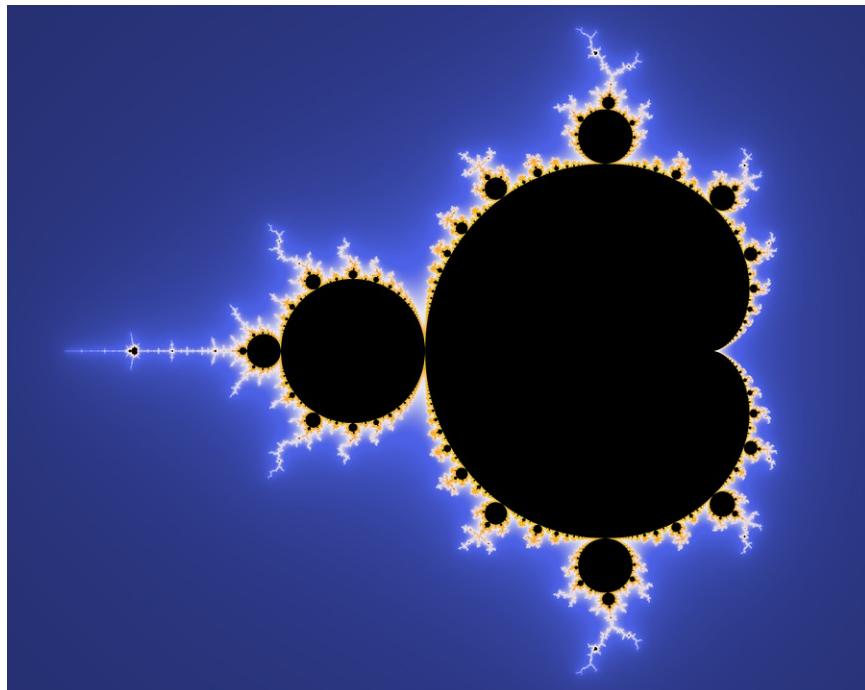
Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Írj olyan C++ programot, amely kiszámolja a Mandelbrot halmazt!

Megoldás forrása: [itt](#)

Mielőtt a Mandelbrot halmazzal foglalkoznánk, elsőnek tisztázzuk, hogy mik is a fraktálok, és mi a kapcsolatuk a Mandelbrot-halmazzal. A fraktálok lényegében olyan alakzatok, melyek végtelenül komplexek. Két fő tulajdonságuk van, az egyik, hogy a legtöbb geometria alakzattal ellentétben a fraktálok szélei "szakadozottak", nem egyenletesek. A másik tulajdonságuk pedig, hogy nagyon hasonlítanak egymásra. Ha egy kör határfelületét folyamatosan nagyítjuk, egy idő után kisimul(a csúcsokat leszámítva), megkülönböztethetetlen válik egy egyenestől. Ezzel szemben a fraktálok első tulajdonsága, mi szerint határfelületük szakadozott, megmarad, függetlenül a nagyítás mértékétől. A Mandelbrot halmaz is a fraktálok közé tartozik. Ezt és a hozzá tartozó szabályt Benoit Mandelbrot fedezte fel 1979-ben. A halmaz komplex számokból áll, melyek az alábbi sorozat elemei: $x_1 := c$, $x_{n+1} := (x_n)^2 + c$, és ez a sorozat konvergens, azaz korlátos. Ezeket a számokat ábrázolva a komplex számsíkon kapjuk meg a Mandelbrot-halmaz híres farktálját.



5.1. ábra. Mandelbrot halmaz

Ezt az ábrát fogjuk mi megalkotni a C++ programunkkal. Ehhez a png++ header fájlra lesz szükségünk, mely nincs alapból telepítve. A sudo apt-get install libpng++-dev parancssal tudjuk feltelepíteni, és a g++ fordító használatánál szükségünk lesz a -lpng kapcsolóra. Most vegyük szépen végig a programot.

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

int main (int argc, char *argv[])
{
    ...
}
```

Tehát, ahogy már említettem, szükségünk lesz a p++/png.hpp header-re. Parancssori argumentum segítségével adjuk meg, hogy melyik fájlba szeretnénk elmenteni a képet.

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

Ha nem adjuk meg az argumentumot, akkor dobunk egy hibaüzenetet, amely tartalmazza, a helyes használat leírását. Ha megadtuk az argumentumot, akkor elkezdődik a lényeg.

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

Első lépésként megadjuk a függvény értékkészletét és értelmezésitartományát. Majd meghatározzuk a létérehozandó kép méretét, és az iterációs határt.

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

Ezzel az utasítással létrehozunk egy üres png-t, melybe majd betölthjük a Mandelbrot halmaz ábráját.

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

Megadjuk a lépésközt, amellyel majd végig megyünk a koordináta-rendszer rácspontjain. Deklaráljuk a változókat, amikben a c és a z komplex számok valós és imaginárius részét fogjuk tárolni. Ezután pedig végigmegyünk a rácson 2 egymásba ágyazott for ciklus segítségével.

```

for (int j=0; j<magassag; ++j) {
    for (int k=0; k<szelesseg; ++k) {
        reC = a+k*dx;
        imC = d-j*dy;
        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;
}

```

A valós számokat képesek vagyunk egy számegyesen ábrázolni, de a komplex számokat már nem, szükségünk van ugyanis egy másik tengelyre amelyen a képzetek részleteket ábrázoljuk. Így lényegében egy koordináta rendszert kapunk, ahol minden számnak van egy x és egy y koordinátája, jelen esetben egy valós és egy képzetes rész. Tehát elkezdünk végig lépkedni az értelmezési tartományon, és minden iterációban megadjuk a c számot, melyhez kiszámoljuk a z értékeit. Ehhez van szükség egy while ciklusra, melyben egészen addig számoljuk a halmaz következő elemeit, ameddig a z komplex szám négyzete kisebb, mint 4 és még nem értük el az iterációs határt. Ha elértek az iterációs határt, az iteráció konvergens, tehát a c eleme a Mandelbrot halmaznak.

```
kep.set_pixel(k, j, png::rgb_pixel(255-iteracio%256,  
                                255-iteracio%256, 255-  
                                iteracio%256));
```

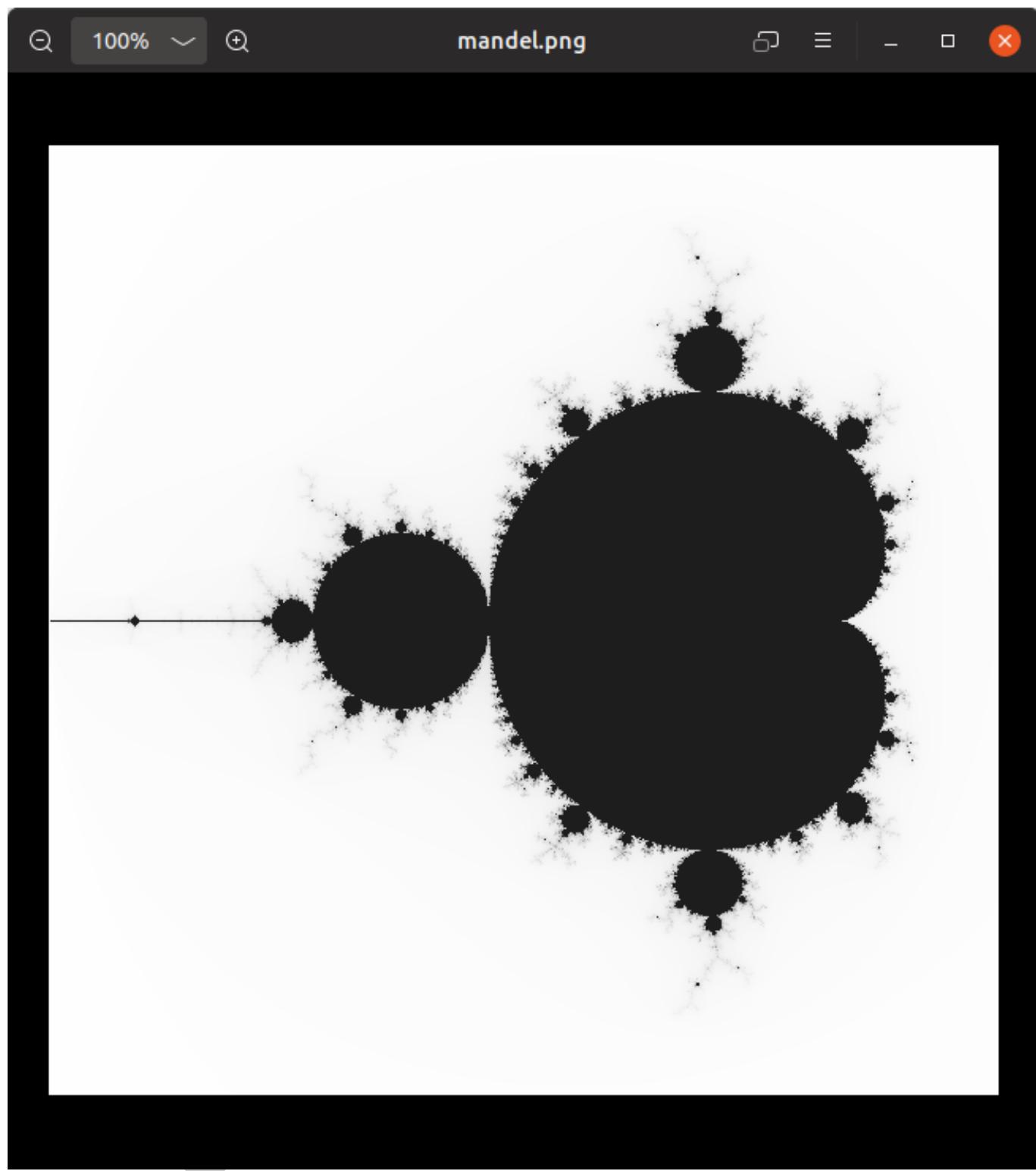
Megadjuk létehozott png képunk egyes pixeleinek a megfelelőt színt, ezzel kirajzolódik a Mandelbrot-halmaz ábrája.

```
kep.write (argv[1])
```

Végezetül pedig a létrehozott képunk tartalmát beleírjük abba fájlba, amit a felhasználó megad argumentumként.

```
Fájl Szerkesztés Nézet Keresés Terminál Lapok Súgó
fupn26@fupn26-Lenovo-Ideapad-330-15IKB:~/Programozás/Mandelbrot
fupn26@fupn26-Lenovo-Ideapad-330-15IKB:~/Programozás/Mandelbrot$ g++ mandelpng.cpp -o mandelpng -lpng
fupn26@fupn26-Lenovo-Ideapad-330-15IKB:~/Programozás/Mandelbrot$ ./mandelpng mandel.jpg
Számítás.....mandel.jpg mentve
fupn26@fupn26-Lenovo-Ideapad-330-15IKB:~/Programozás/Mandelbrot$
```

5.2. ábra. Program fordítás, futtatása



5.3. ábra. Mandelbrot halmaz

5.2. A Mandelbrot halmaz a std::complex osztályal

Megoldás forrása: [itt](#)

Az előző feladatot fogjuk megoldani, csak most egy kicsit másképpen. Ahogy láttad, az előbb a komplex számokat két változóban tároltuk, egyikben a valós, másikban pedig a képzetes részét. De az infomatiskusok lusták, mindenek használnánk 2 változót, ha lehet egyet is. Ezt teszi számunkra lehetővé a complex library, melynek segítségével a gép képes kezelni ezeket a számokat.

Ahogy az előbb, most is végigfutunk a forráson. Az első különbség ott van, hogy a felhasználó adhatja meg a létrehozandó kép attribútumait, de ezt ki is hagyhatjuk, akkor az alapértelmezett értékeket használja a program.

```
int szelesseg = 1920;
intmagassag = 1080;
intiteraciosHatar = 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] );
}
```

Az atoi és atof segítségével tudjuk átalakítani a parancssori argumentum stringet int és double típusra. Ezután létrehozzuk az üres png-t, mint legutóbb, majd a szükséges változókat.

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

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

A két egymásba ágyazott for ciklus segítségével bejárjuk a rácsot, és beállítjuk a kép pixeleit a megfelelő értékre, mely most eltér az előzőhöz képest. Itt egy színesebb ábrát fogunk kapni, az előzőhöz képest.

```
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;

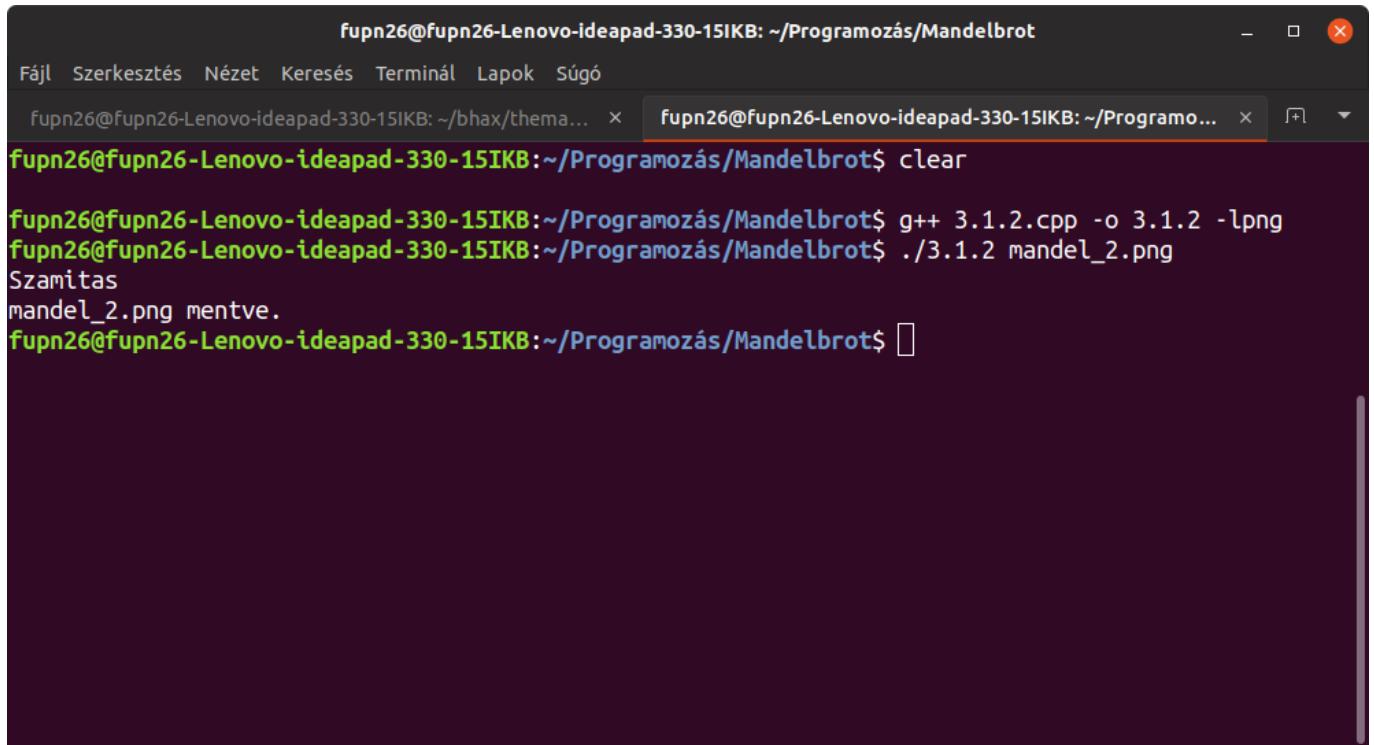
while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )
{
    z_n = z_n * z_n + c;

    ++iteracio;
}

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

int szazalek = ( double ) j / ( double ) magassag * 100.0;
//kiírja, hogy hány százaléknél tart a képgenerálás
std::cout << "\r" << szazalek << "%" << std::flush;
}
```

Na itt néhány dolg eltér az előző feladathoz képest. Itt használjuk elsőnek a complex típust, ami double-ket tartalmaz, és két részből áll, a valós és az imaginárius részből. Ennek a segítségével definiáljuk a c és a z_n változókat. Majd, innen már ismerős lehet, kiszámoljuk minden c esetén a z_n-eket, és ha elérjük az iterációs határt akkor, tudhatjuk, hogy az iteráció konvergens. Ebből következik, hogy a c eleme a Mandelbrot halmaznak. A while fejrészében látható abs () függvény az abszolút értékét adja meg az bemenetként kapott argumentumának. A halmazt lértehozó sorozat képzési szabálya egy az egyben beírható a programba, nincs szükség semmilyen szétbontásra, mint az előző programnál volt, köszönhetően annak, hogy képesek vagyunk kezelni a komplex számokat. Pusz dolg, hogy a a program a futása azt is látjuk, hogy hány százalékát végezte el a számításoknak a gép. Végezetül pedig itt is kiírjuk a png fájlt a parancssori argumnetumként megadott fájlba a write segítségével.

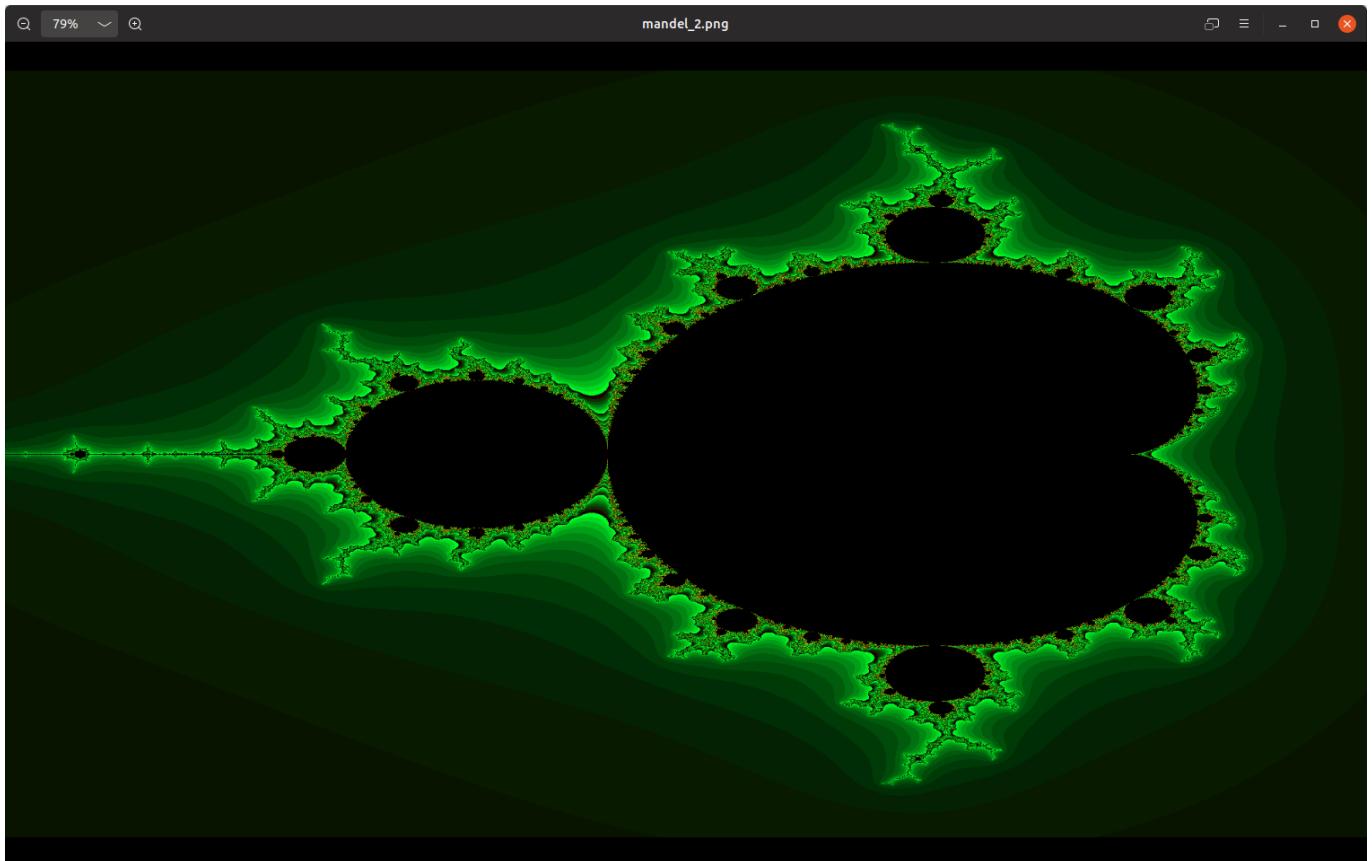


Fájl Szerkesztés Nézet Keresés Terminál Lapok Súgó

fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás/Mandelbrot

```
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot$ clear
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot$ g++ 3.1.2.cpp -o 3.1.2 -lpng
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot$ ./3.1.2 mandel_2.png
Szamitas
mandel_2.png mentve.
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot$
```

5.4. ábra. Program fordítása és futtatása



5.5. ábra. Mandelbrot halmaz

5.3. Biomorfok

Megoldás videó: <https://youtu.be/IJMbgrZy76E>

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

Tanulságok, tapasztalatok, magyarázat...

Tovább folytatjuk a Mandelbrot-os témaüket. Ezzel a feladattal nem egészen direkt a kapcsolata a Mandelbrot halmaznak, inkább a Julia halmazokkal lesz szoros rokonságban. A Mandelbrot halmaz tartalmazza az összes Julia halmazt. Ez abból következik, hogy a Julia halmaz esetén a c konstans, és a z -vel járjuk be, viszont a Mandelbrot halmazban a c változóként szerepel, melyhez kiszámoljuk a z értékeit. Szóval minden újabb és újabb Julia halmazt számolunk ki vele.

A Biomorfokra Clifford Pickover talált rá, méghozzá egy Julia halmazt rajzoló prgramjának írása közben. A programja rejtegett egy bugot, és emiatt egészen furcsa dolgokat produkált a program, melyre azt hitte, hogy valami természeti csodára lelt rá. Magáról a Biomorfokról, és a Pickover történetéről részletesebben olvashatsz [itt](#). Mi is ez alapján készítettük el a biomorf rajzoló programunkat, mely a Mandelbrot-os programra alapul, annak egy továbbfejlesztett verziója.

A program eleje teljesen megegyezik a Mandelbrot halmazos programunkkal, azzal a kivétellel, hogy most a felhasználótól kérjük be a c konstans érékét, és a küszöbszámot. Ezek az adatok megtalálhatóak a cikkben, minden biomorphhoz külön-külön.

```
#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] );
    }
}
```

Ahogy látható a parancssori argumentumok száma 10-re nőtt, melyeket nem kötelező megadni, ilyenkor az alapértelmezett értékeket használja. Ezt követően létrehozzuk az üres png-t, a lépésközöt a rácsok között, és most a cc szám deklarációja is a cikluson kívülre kerül, mivel az jelen esetben konstans.

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

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

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

Egy kisebb átalakításra volt szükség a cilusok tekintetében, melyet ezen algoritmus alapján módosítottunk:

```
1 for x = xmin to xmax by s do
2     for y = ymin to ymax by s do
3         z = x + yi
4         ic = 0
```

```
5      for i = 1 to K do
6          z = f(z) + c
7          if |z| > R then
8              ic = i
9              break
10         PrintDotAt(x, y) with color ic
//forrás: https://bit.ly/2HCbCYs
```

Ennek a C++ implementációja a következő:

```
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 < iteraciosHatar; ++i)
        {

            z_n = std::pow(z_n, z_n) + std::pow(z_n, 6) + 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*60)%255, (iteracio *
                        *100)%255, (iteracio*40)%255 ) );
    }

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

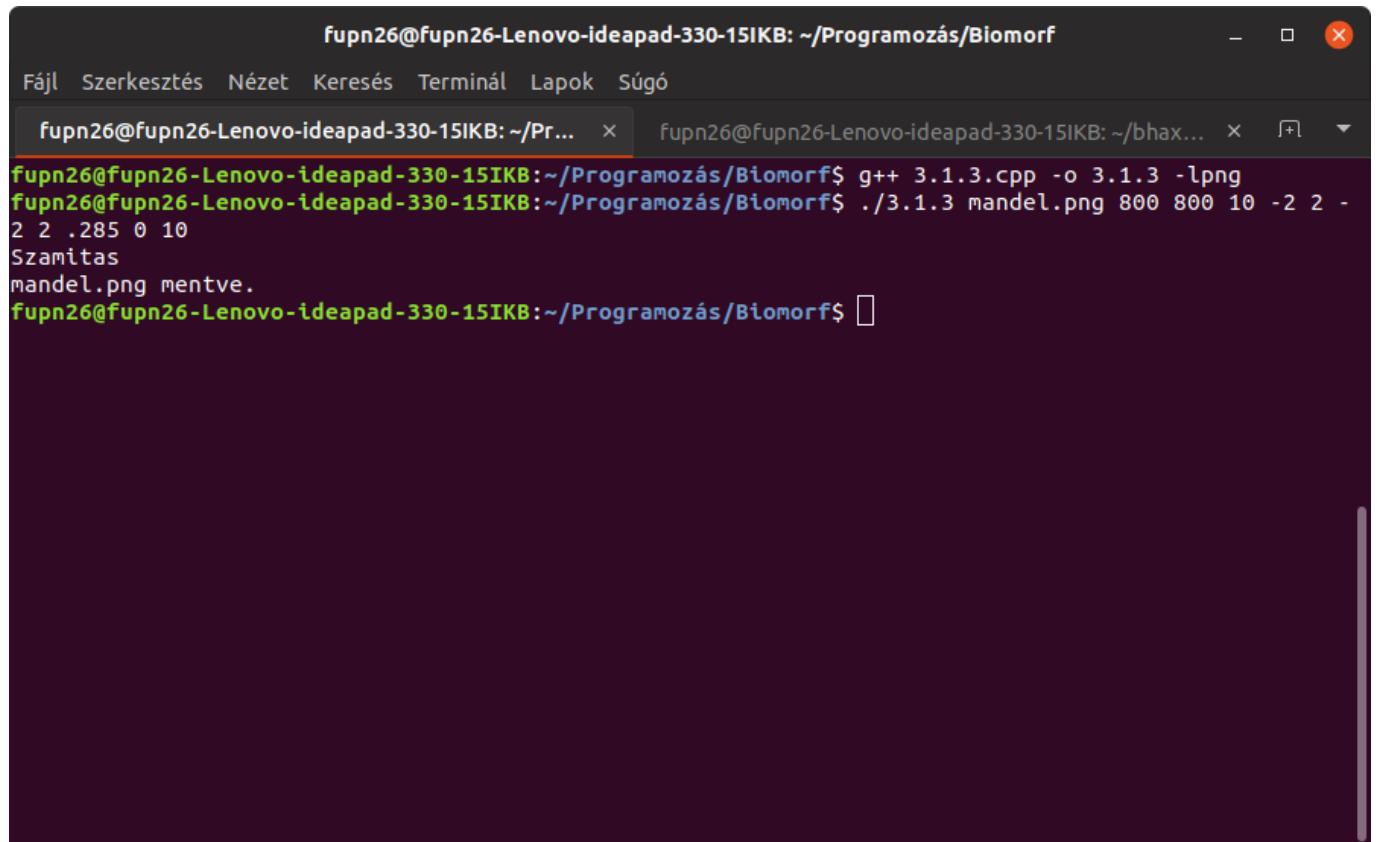
A két egymásba ágyazott for ciklus segítségével végigmegyünk a rácspontokon és egy harmadik for ciklus-
sal pedig kiszámoljuk a függvényértékeket, egészen addig, amíg el nem érjük az iterációs határt, vagy nem
teljesül ez a feltétel:

```
if (std::real ( z_n ) > R || std::imag ( z_n ) > R)
```

Ez volt az a bug, amit Clifford Pickover programja tartalmazott, egy sor, ami nélkül ez a feladat lehet, hogy soha nem készült volna el. Végezetül pedig beállítjuk az egyes pixelek színét, makd kírjuk egy fájlba a

tartalmát.

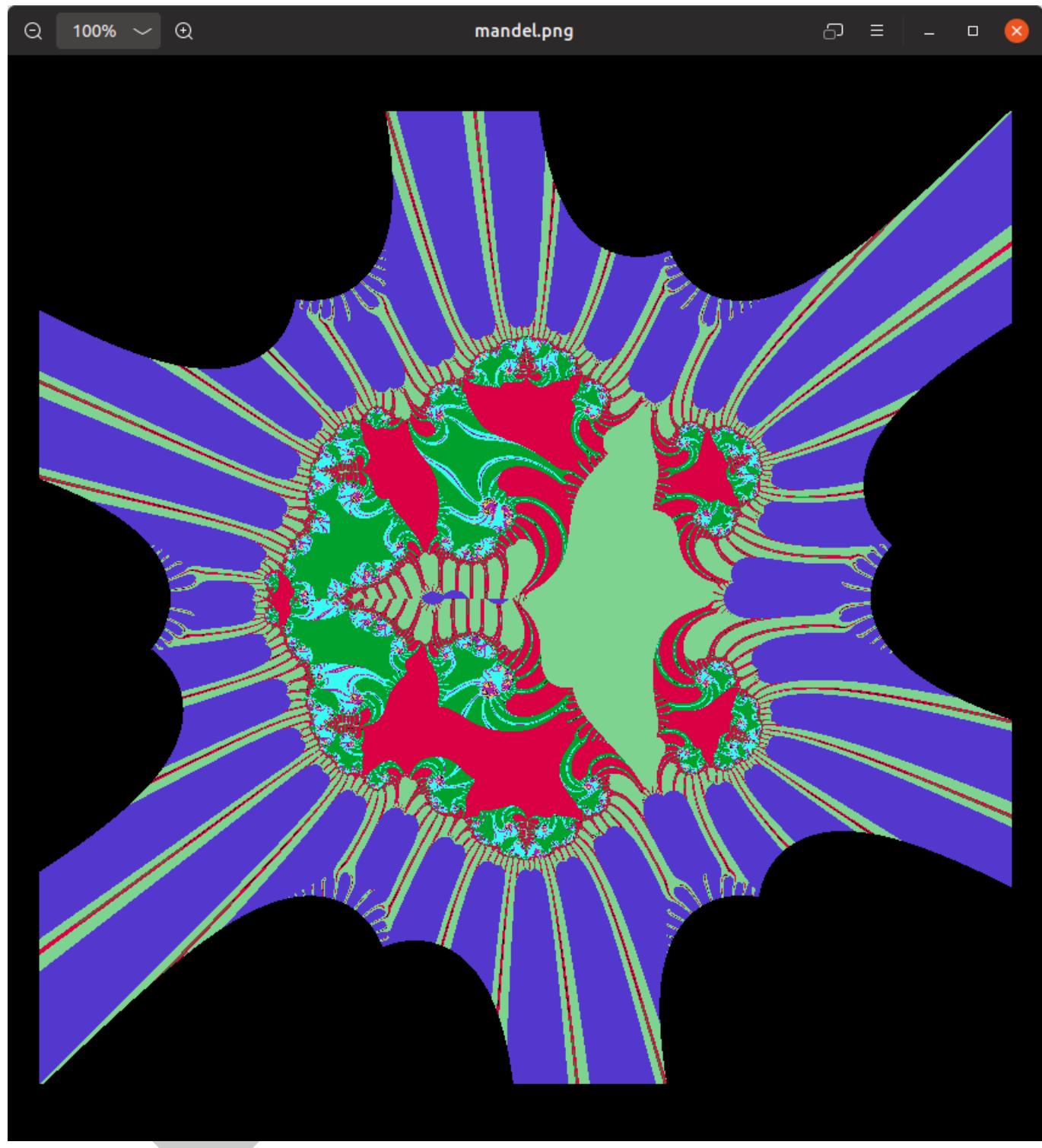
```
kep.write ( argv[1] );
```



The screenshot shows a terminal window titled "fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás/Biomorf". The window has tabs for "fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Pr..." and "fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/bhax...". The terminal content is as follows:

```
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Biomorf$ g++ 3.1.3.cpp -o 3.1.3 -lpng
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Biomorf$ ./3.1.3 mandel.png 800 800 10 -2 2 -
2 2 .285 0 10
Szamitas
mandel.png mentve.
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Biomorf$
```

5.6. ábra. Program fordítása és futtatása



5.7. ábra. Biomorf

Sajnos nem sikerült elérni ugyanazt a színt, mint a cikkbén, de lényeg ezen is látható.

5.4. A Mandelbrot halmaz CUDA megvalósítása

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

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Tovább folytatjuk a Mandelbrot-halmazos programunk fejlesztését, ezúttal az Nvidia CUDA technológiáját hívjuk segítségül, mellyel jelentősen fel tudjuk gyorsítani a kép generálását. A teknika lényege, hogy egy 600x600 darab blokkból álló gridet hozunk létre, és minden blokhoz tartozik egy szál. Ezzel sikerül a program futását párhuzamosítani.



Megjegyzés

A CUDA használatához nvidia GPU-ra van szükség, és telepíteni kell a nvidia-cuda-toolkit-et.

Lássuk akkor a forrást:

```
#define MERET 600
#define ITER_HAT 32000

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

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

    // a számítás
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, reZ, imZ, ujreZ, ujimZ;
    // Hány iterációt csináltunk?
    int iteracio = 0;

    // c = (reC, imC) a rácson csomópontjainak
    // megfelelő komplex szám
    reC = a + k * dx;
    imC = d - j * dy;
    // z_0 = 0 = (reZ, imZ)
    reZ = 0.0;
    imZ = 0.0;
    iteracio = 0;
    // z_{n+1} = z_n * z_n + c iterációk
    // számítása, amíg |z_n| < 2 vagy még
```

```
// nem értük el a 255 iterációt, ha
// viszont elértük, akkor úgy vesszük,
// hogy a kiinduláció c komplex számra
// az iteráció konvergens, azaz a c a
// Mandelbrot halmaz eleme
while (reZ * reZ + imZ * imZ < 4 && iteracio < iteracionsHatar)
{
    // z_{n+1} = z_n * z_n + c
    ujreZ = reZ * reZ - imZ * imZ + reC;
    ujimZ = 2 * reZ * imZ + imC;
    reZ = ujreZ;
    imZ = ujimZ;

    ++iteracio;

}
return iteracio;
}
```

Előre definiáljuk a méret és az iterációs határ értékét. A `mandel` függvényel hozzuk létre a Mandelbrot halmazt. Ez teljes mértékben megegyezik az első Mandelbrot-os feladatunkkal, ahol nem használtuk a `complex` típust. Érdekesség lehet, hogy a függvény visszatérési értéke előtt megtalálható a `__device__` kifejezés, mellyel azt jelezzük, hogy CUDA-val fogjuk számolni. Amikor az `nvcc`-vel fordítunk, akkor a fordító két részre osztja a programot, egy eszközhöz kapcsolódó részre, amelyet az NVIDIA fordító készít el, és egy host részre, mely a `gcc`-vel fordul. Amelyik delkaráció elé odaírjuk a `__device__` vagy a `__global__` kifejezést, azt az NVIDIA fordító fogja gépi kóddá alakítani. A régebbi feladatban a számok `double` típusúak voltak, itt áttértünk a `float` típusra, mivel a fordító úgyis erre konvertálta volna őket.

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

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

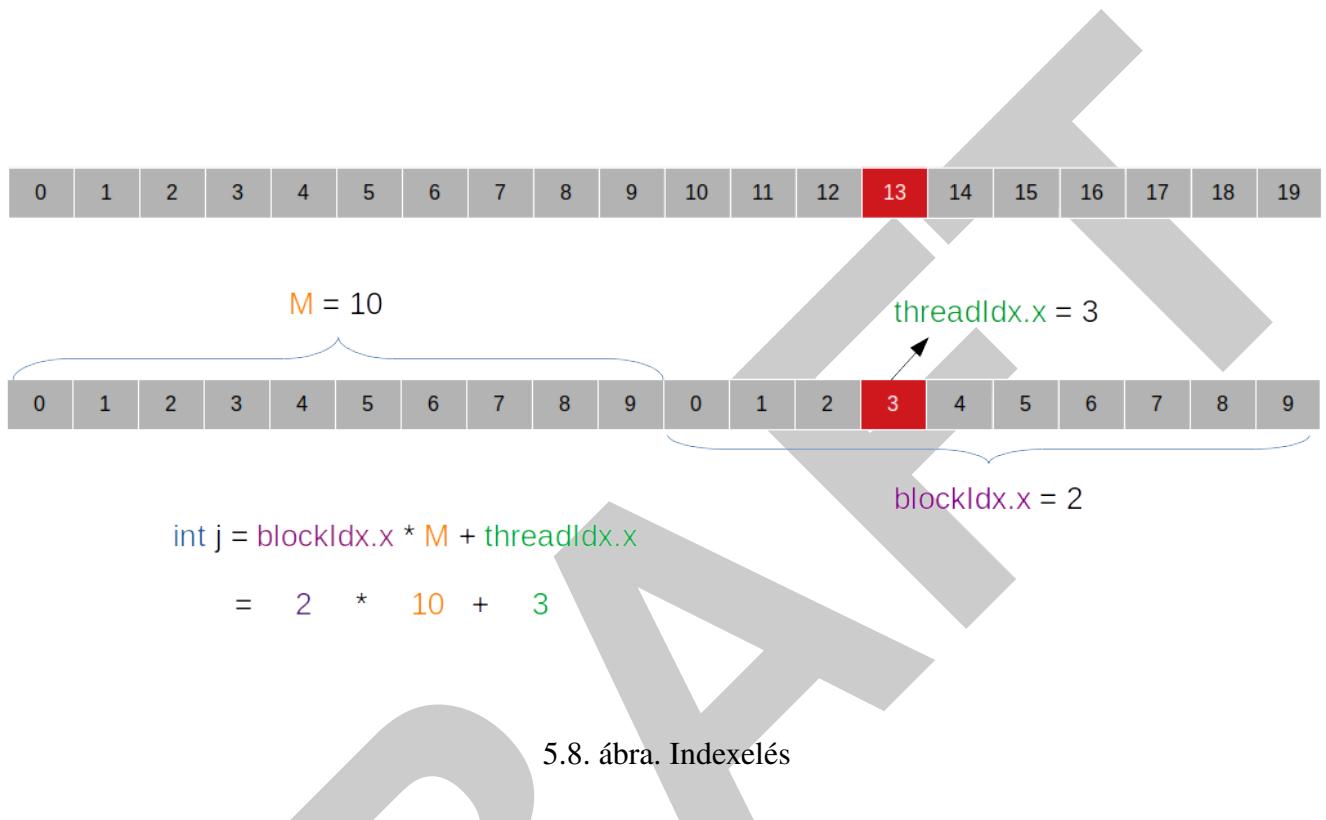
    int j = blockIdx.x * 10 + tj;
    int k = blockIdx.y * 10 + tk;

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

}
```

Ahogy említettem a `__global__` előtag jelzi, hogy ezt is a GPU-val fogjuk elvégeztetni. Ebben a részben adjuk át a `mandel` függvénynek az aktuálisan feldolgozás alatt álló érték indexét, amelyhez kiszámoljuk az összes lehetséges z értéket. A `threadIdx.x/y` jelöli, hogy melyik szálon fut az aktuális x és y számhoz tartozó érték kiszámítása. Ahhoz, hogy ezekenek a pontos indexét meg tudjuk adni, tudnunk kell, hogy melyik blokkba van épbenne a szám, és hogy mekkora a blokk mérete, ezek segítségével a fent látható

módon eltudjuk tárolni a koordinátákat a j és k változókban. Egy ábra, hogy könnyebben megértsd, hogyan működik ez az indexelés.



```

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

// dim3 grid (MERET, MERET);
// mandelkernel <<< grid, 1 >>> (device_kepadat);

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

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

```

A cudamandel függvénynek átadunk egy 600x600-as tömböt. Létrehozunk egy pointert, és a memóriában foglalunk neki egy az átadott tömmbel megegyező méretű területet, melyre ráállítjuk. Majd ezt pointert fogjuk átadni a mandelkernel függvénynek. A függvényhívásnál észrevehetsz egy furcsaságot. A <<<a, b>>> kifejezésben lévő 2 érték közül az a jelöli, hogy hány blokkot akarunk létrehozni, a b pedig a blokkokhoz tarto zó szálak számát. A előbbi értéke jelen esetben 3600, míg az utóbbié 100 lesz. Ha függvény lefutott, akkor átmásoljuk az értékeket a argumentumként megadott tömbbe, és felszabadítjuk a lefoglalt területet.

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

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

    int kepadat [MERET] [MERET];

    cudamandel (kepadat);

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

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

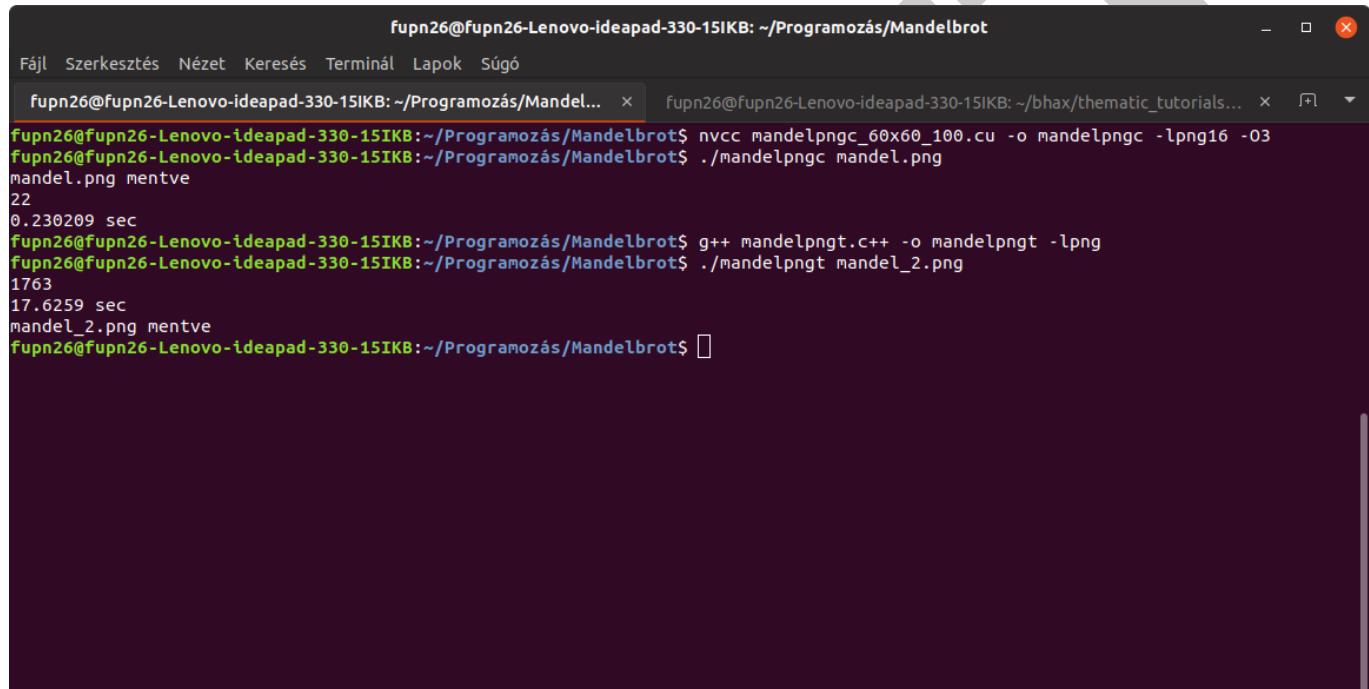
    kep.write (argv[1]);

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

    times (&tmsbuf2);
    std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
```

```
+ tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;  
  
delta = clock () - delta;  
std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << std::endl;  
}
```

A main függvény nem tartalmaz nagy újdonságokat az előző feladatokhoz képest, annyi különbséggel, hogy most a futási időt is mérjük, hogy össze tudjuk hasonlítani a CUDA-s verziót a simával. Itt is végig-megyünk az egyes pixleken, és beállítjuk a megfelelő színeket, melyhez felhasználjuk a képadatban tárolt értékeket.

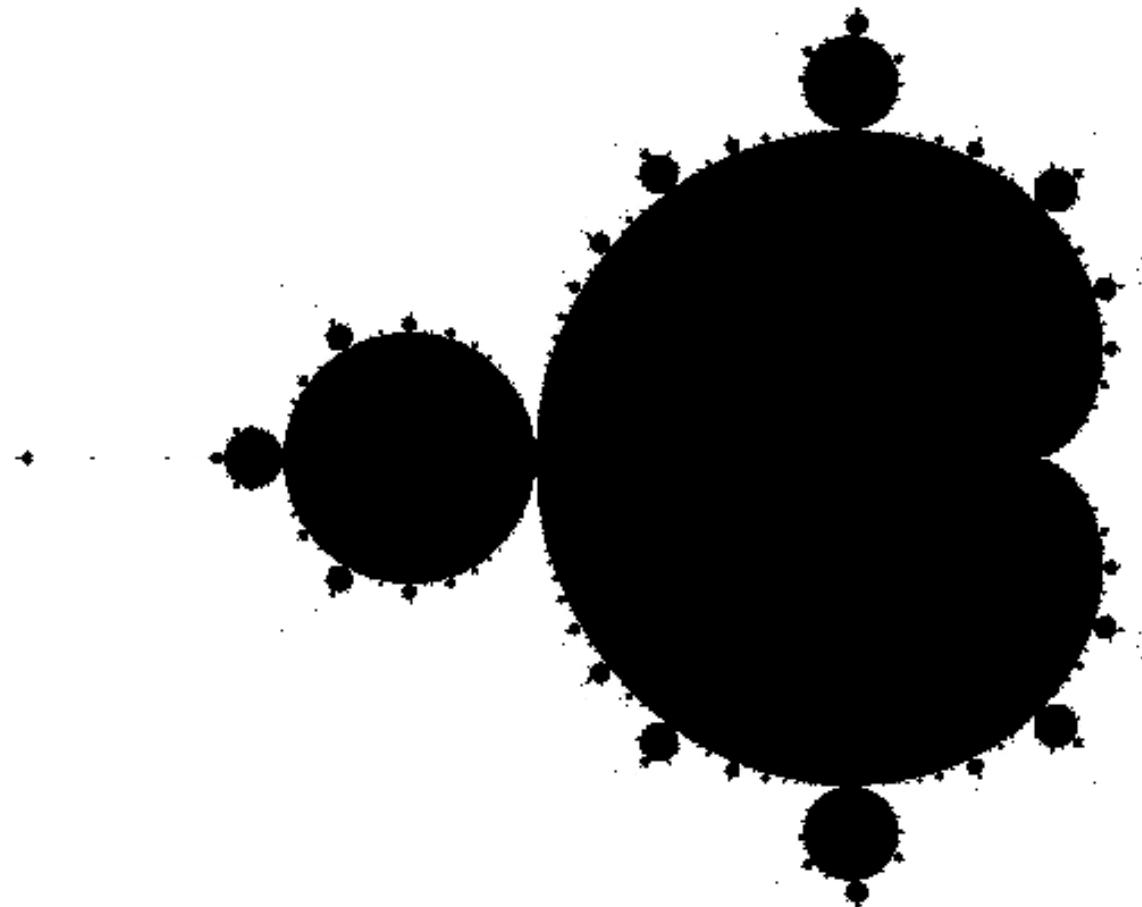


The screenshot shows a terminal window titled 'fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás/Mandelbrot'. The terminal displays the following command-line session:

```
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot$ nvcc mandelpngc_60x60_100.cu -o mandelpngc -lpng16 -O3  
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot$ ./mandelpngc mandel.png  
mandel.png mentve  
22  
0.230209 sec  
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot$ g++ mandelpngt.cpp -o mandelpngt -lpng  
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot$ ./mandelpngt mandel_2.png  
1763  
17.6259 sec  
mandel_2.png mentve  
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot$
```

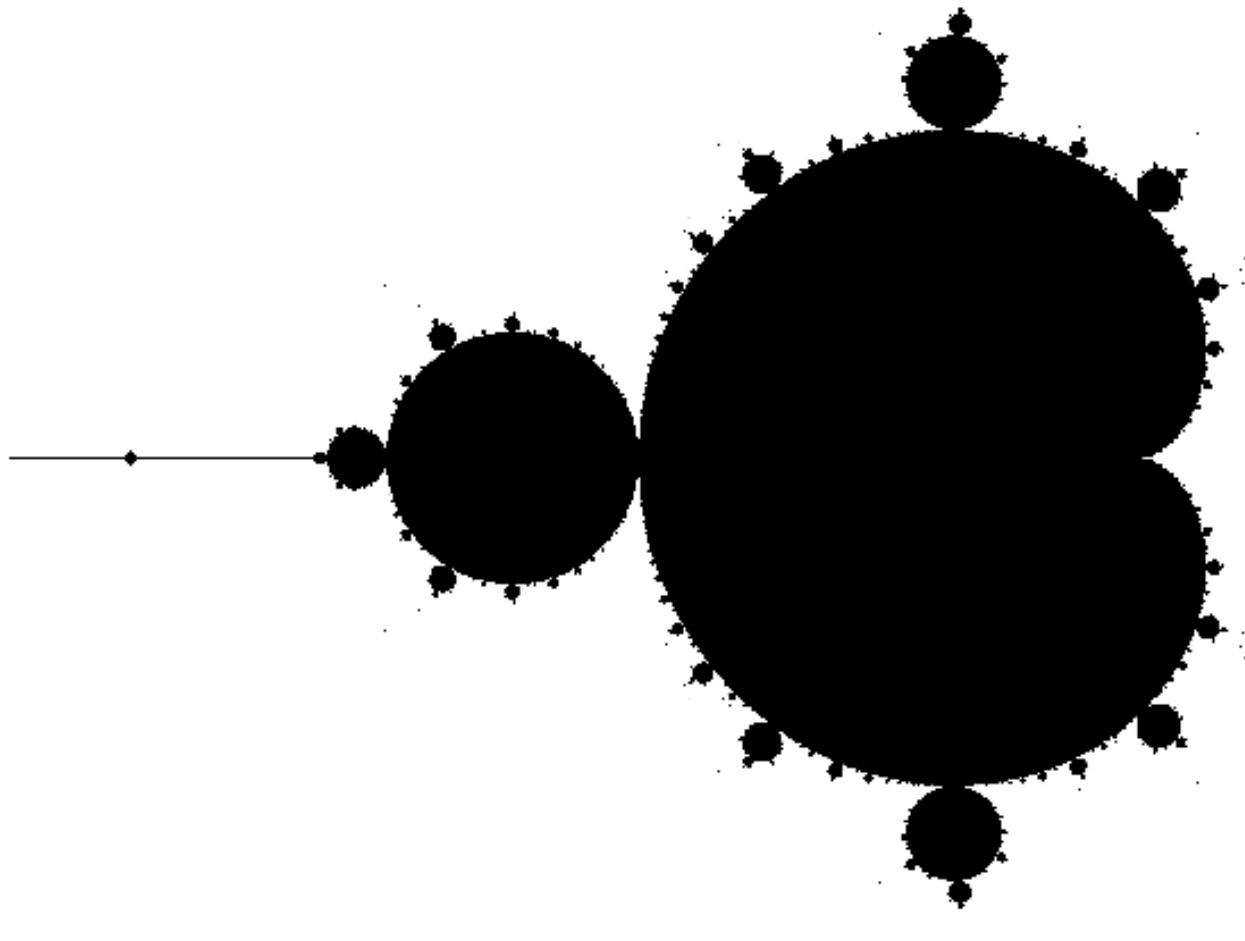
5.9. ábra. Két program hasonlítása

Hogy a két programot összetudjuk hasonlítani, a régi C++-os Mandelbrot programunkat alakítottuk ást, hogy az is számolja a futási időt. Ahogy a képen is láthatod, a különbség letaglózó. Kevesebb mint az egy tizenhetede a CUDA-s implementáció, az eredetinek. Hogy lássuk, nem árulok zsákbamacskát, megmutatom az egyik és a másik által készített képeket is.



5.10. ábra. CUDA variáns





5.11. ábra. C++ megvalósítás, párhuzamosítás nélkül

Összességében elmondhatjuk, hogy a CUDA egy nagyon hasznos technológia, mely jelentős gyorsulást érhetünk el, főleg olyan programokban, ahol képet kell generálni. De videók renderelésnél is nagyon hasznos.

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

Megoldás forrása:[itt](#)

Megoldás videó:



Megjegyzés

A feladat megoldásában tutorként részt vett Racs Tamás. Tutorálként pedig Halász Dávid.

Tanulságok, tapasztalatok, magyarázat...

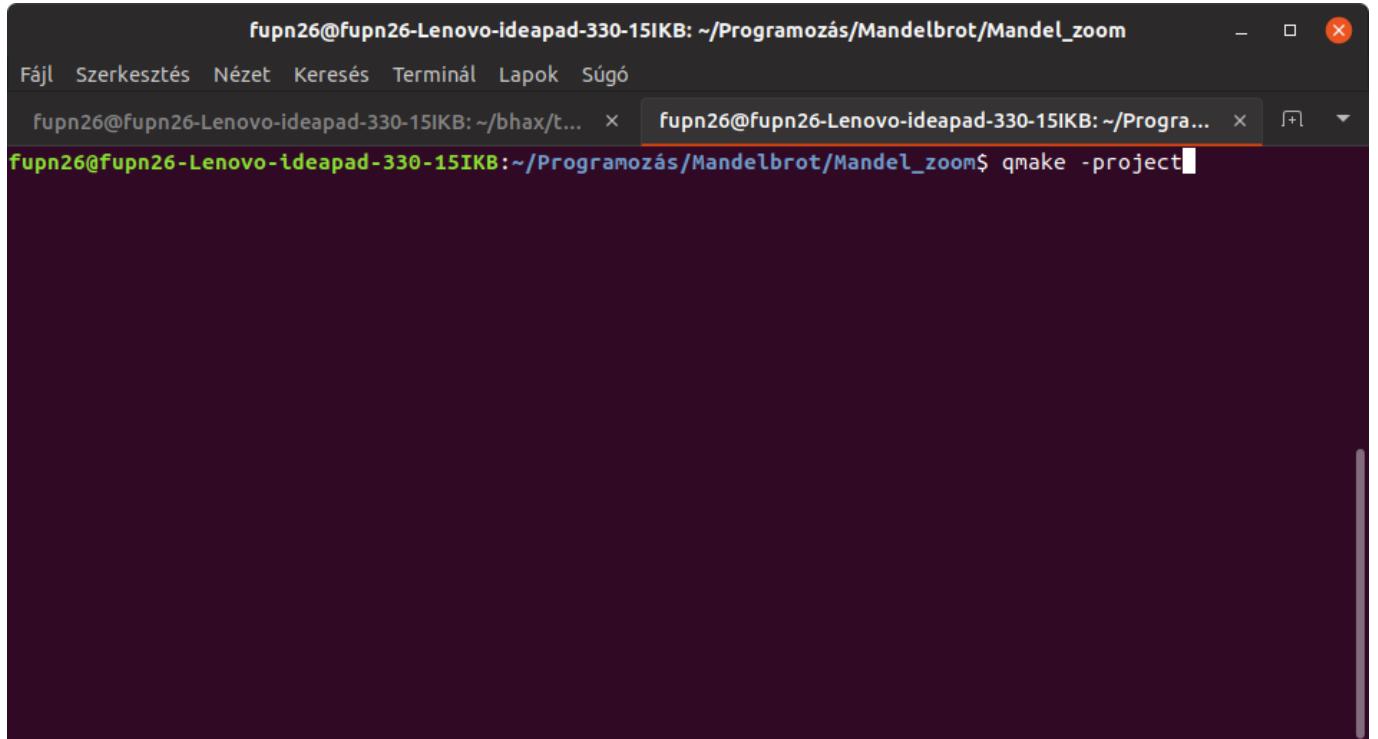


Használata

Telepíteni: sudo apt-get install libqt4-dev

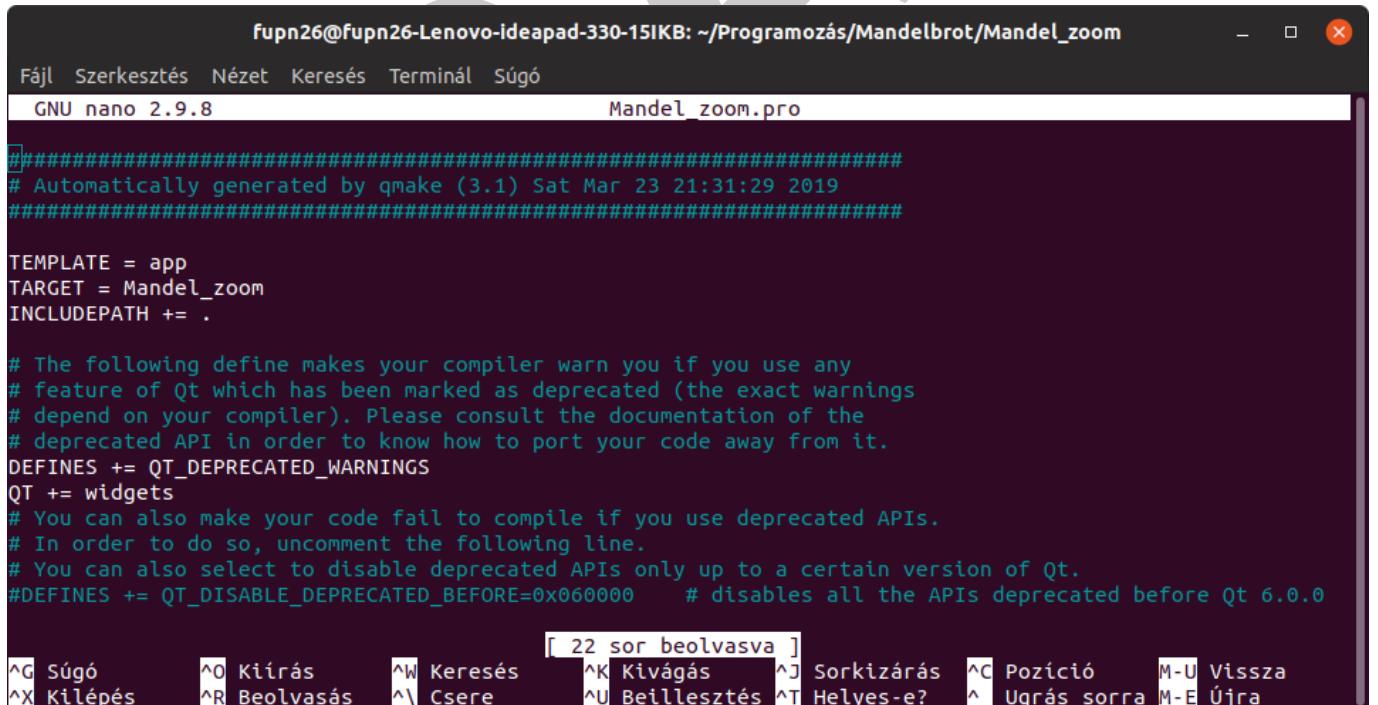
A program a QT GUI-t használja, ennek segítségével tudjuk elkészíteni a Mandelbrot halmazt beutazó programunkat. Ez a GUI az egyik legertékkeltebb grafikus interfésze a C++-nak, rengeteg tutorial van rólá fent a neten.

Fordítás: Az szükséges 4 fájlnak egy mappában kell lennie. A mappában futtatni kell a qmake -project parancsot. Ez létre fog hozni egy *.pro fájlt. Ebbe a fájlba be kell írni a következő: QT += widgets sort. Ezután futtatni kell a qmake *.pro. Ezután lesz a mappában egy Makefile, ezt kell majd használni. Ki adjuk a make parancsot, mely létrehoz egy bináris fájlt. Ezt pedig a szokásos módon futtatjuk.



```
fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás/Mandelbrot/Mandel_zoom
Fájl Szerkesztés Nézet Keresés Terminál Lapok Súgó
fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/bhax/t... × fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás/Mandelbrot/Mandel_zoom$ qmake -project
```

5.12. ábra. 1. lépés



```
fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás/Mandelbrot/Mandel_zoom
Fájl Szerkesztés Nézet Keresés Terminál Súgó
GNU nano 2.9.8                               Mandel_zoom.pro

#####
# Automatically generated by qmake (3.1) Sat Mar 23 21:31:29 2019
#####

TEMPLATE = app
TARGET = Mandel_zoom
INCLUDEPATH += .

# The following define makes your compiler warn you if you use any
# feature of Qt which has been marked as deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS
QT += widgets
# You can also make your code fail to compile if you use deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000 # disables all the APIs deprecated before Qt 6.0.0

[ 22 sor beolvasva ]
^G Súgó      ^O Kiírás      ^W Keresés      ^K Kivágás      ^J SorkizáráS  ^C Pozíció      M-U Vissza
^X Kilépés   ^R Beolvasás   ^\ Csere       ^U Beillesztés ^T Helyes-e?   ^_ Ugrás sorra M-E Újra
```

5.13. ábra. 2. lépés

```
fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás/Mandelbrot/Mandel_zoom
Fájl Szerkesztés Nézet Keresés Terminál Súgó
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot/Mandel_zoom$ qmake Mandel_zoom.pro
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot/Mandel_zoom$ make
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -fPIC -DQT_DEPRECATED_WARNINGS -DQT_NO_DEBUG -DQT_WIDGETS_LIB -DQT_GUI_LIB -DQT_CORE_LIB -I. -I. -isystem /usr/include/x86_64-linux-gnu/qt5 -isystem /usr/include/x86_64-linux-gnu/qt5/QtWidgets -isystem /usr/include/x86_64-linux-gnu/qt5/QtGui -isystem /usr/include/x86_64-linux-gnu/qt5/QtCore -I. -isystem /usr/include/libdrm -I/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++ -o frakablak.o frakablak.cpp
frakablak.cpp: In member function ‘virtual void FrakAblak::mouseReleaseEvent(QMouseEvent*)’:
frakablak.cpp:67:48: warning: unused parameter ‘event’ [-Wunused-parameter]
 void FrakAblak::mouseReleaseEvent(QMouseEvent* event) {
                                         ^
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -fPIC -DQT_DEPRECATED_WARNINGS -DQT_NO_DEBUG -DQT_WIDGETS_LIB -DQT_GUI_LIB -DQT_CORE_LIB -I. -I. -isystem /usr/include/x86_64-linux-gnu/qt5 -isystem /usr/include/x86_64-linux-gnu/qt5/QtWidgets -isystem /usr/include/x86_64-linux-gnu/qt5/QtGui -isystem /usr/include/x86_64-linux-gnu/qt5/QtCore -I. -isystem /usr/include/libdrm -I/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++ -o frakszal.o frakszal.cpp
g++ -c -pipe -O2 -Wall -W -D_REENTRANT -fPIC -DQT_DEPRECATED_WARNINGS -DQT_NO_DEBUG -DQT_WIDGETS_LIB -DQT_GUI_LIB -DQT_CORE_LIB -I. -I. -isystem /usr/include/x86_64-linux-gnu/qt5 -isystem /usr/include/x86_64-linux-gnu/qt5/QtWidgets -isystem /usr/include/x86_64-linux-gnu/qt5/QtGui -isystem /usr/include/x86_64-linux-gnu/qt5/QtCore -I. -isystem /usr/include/libdrm -I/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++ -o main.o main.cpp
g++ -pipe -O2 -Wall -W -dM -E -o moc_prelude.h /usr/lib/x86_64-linux-gnu/qt5/mkspecs/features/data/dummy.cpp
/usr/lib/qt5/bin/moc -DQT_DEPRECATED_WARNINGS -DQT_NO_DEBUG -DQT_WIDGETS_LIB -DQT_GUI_LIB -DQT_CORE_LIB -

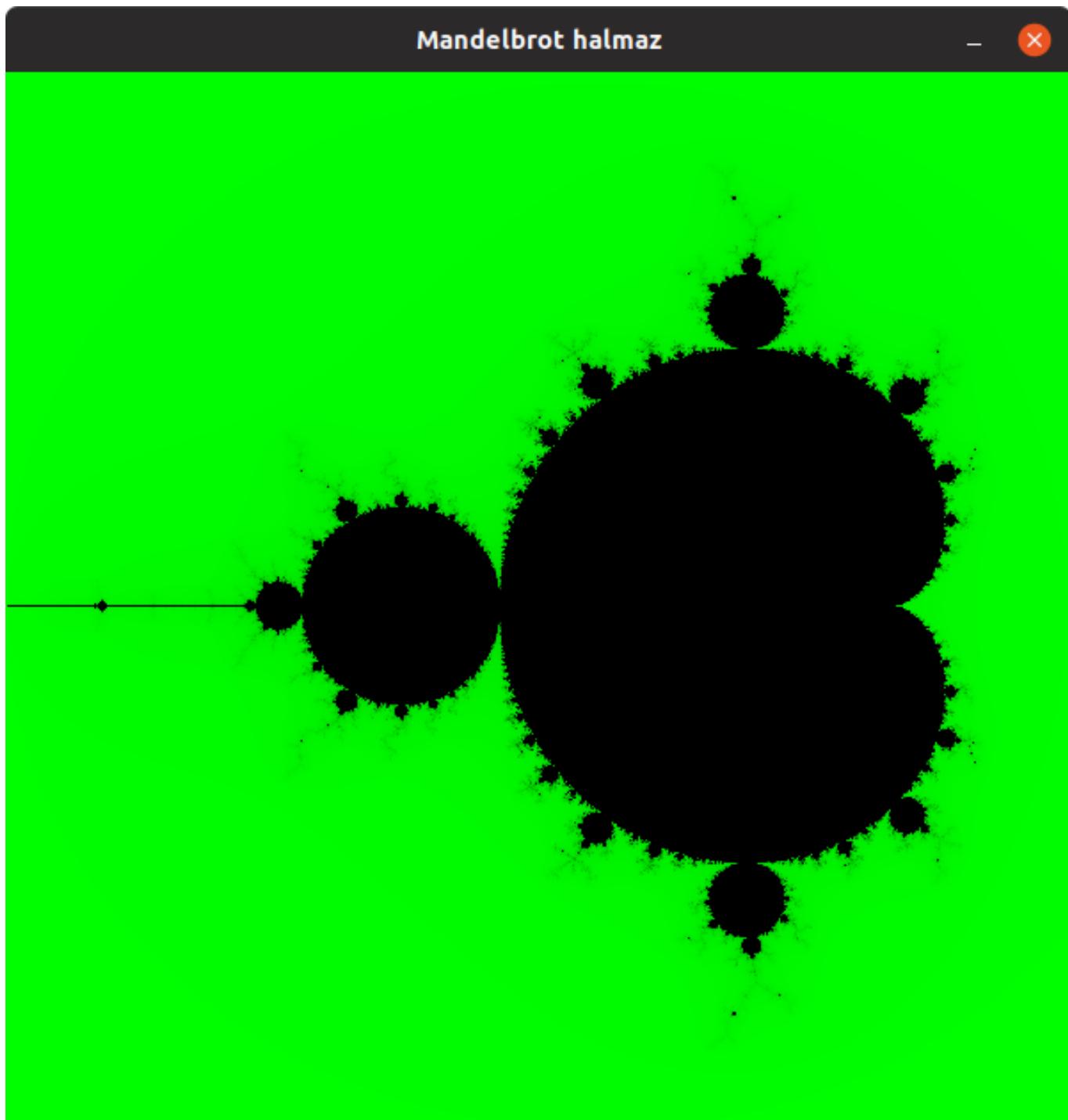
```

5.14. ábra. 3. lépés

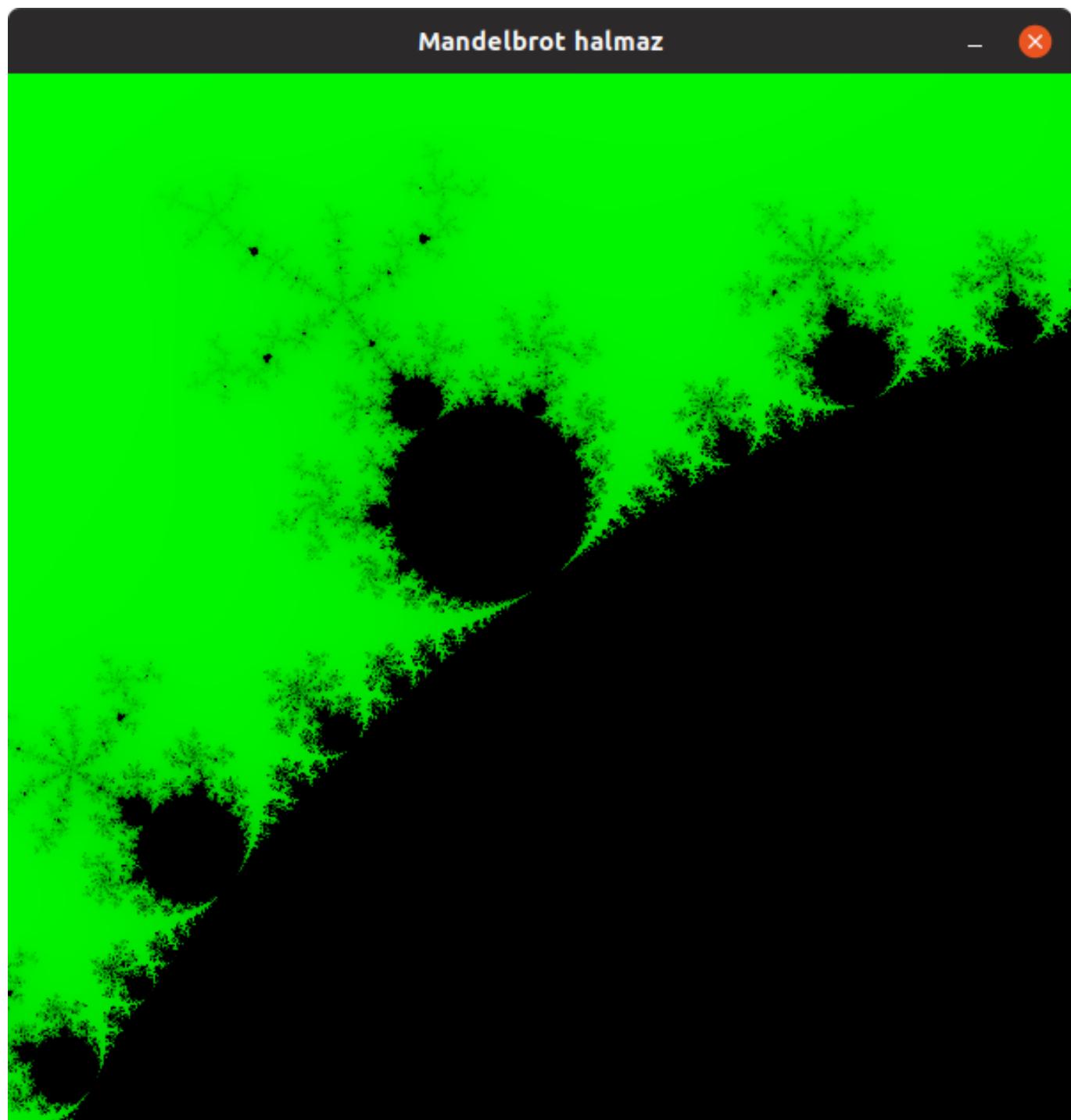
Rengeteg figyelmeztetést ad vissza, de ezzel most nem kell törődni, hiszen a bináris fájl elkészült, melyet futtatunk, és elindul az utazásunk a végtelenbe.

```
fupn26@fupn26-Lenovo-ideapad-330-15IKB: ~/Programozás/Mandelbrot/Mandel_zoom
Fájl Szerkesztés Nézet Keresés Terminál Súgó
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot/Mandel_zoom$ ./Mandel_zoom
fupn26@fupn26-Lenovo-ideapad-330-15IKB:~/Programozás/Mandelbrot/Mandel_zoom$ 
```

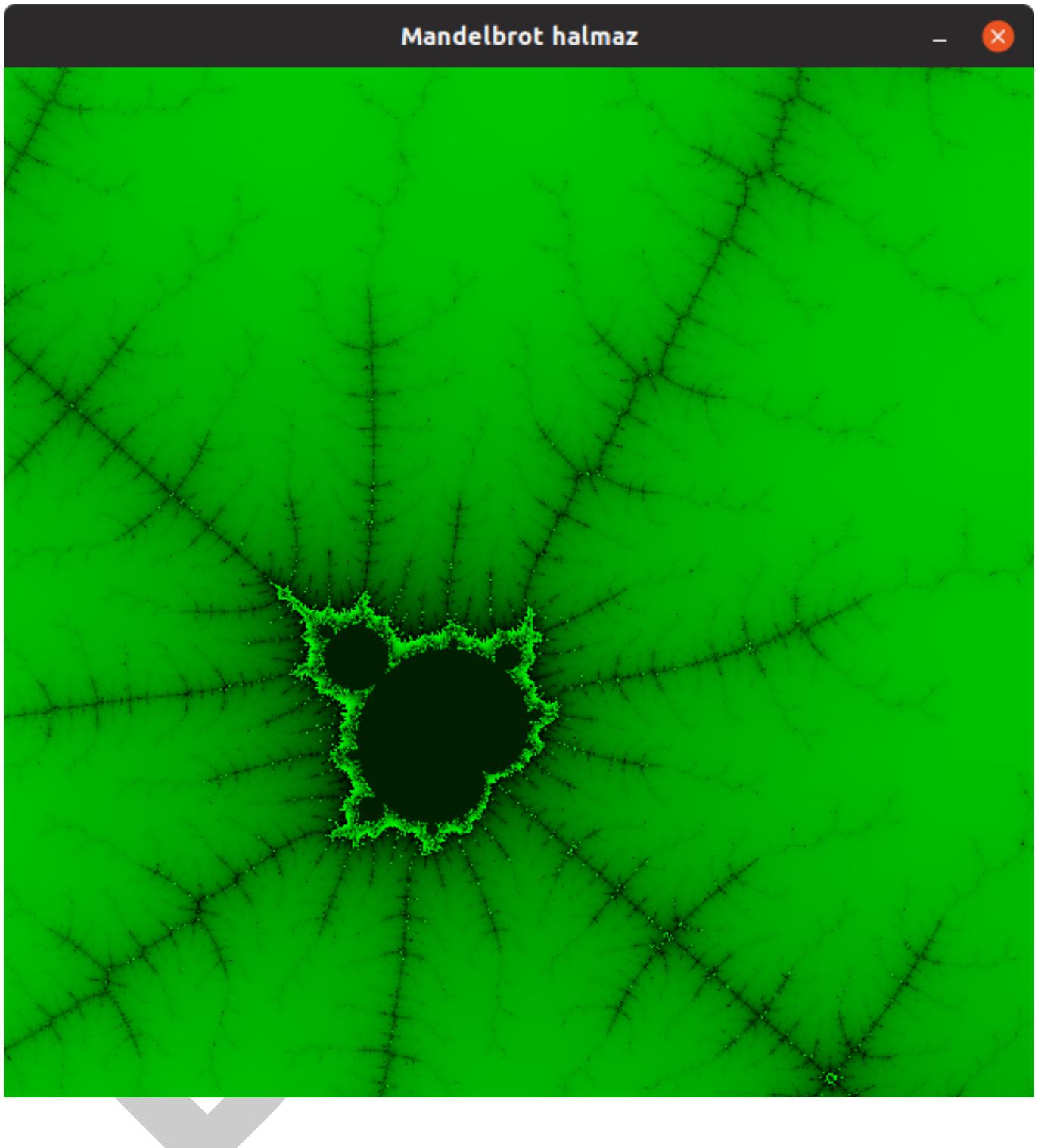
5.15. ábra. 4. lépés



5.16. ábra. Alapállapot



5.17. ábra. Nagyítva



5.18. ábra. Tovább nagyítva

Ahhoz, hogy részletesebb képet kapj a ránagyított területről, az "n" billentyűt kell lenyomnod, mely kiszámolja a z-ket a megadott területen. Ahogy folyamatosan nagyítjuk, észre vehetjük, hogy a újra meg újra Mandelbrot halmazokat kapunk.

Most hogy láttad, hogyan működik a program, nézzük meg a forrást. A szokásostól eltérően, most több fájlból áll a forrás. Ez azt a célt szolgálja, hogy átláthatóbb legyen.

Elsőnek a `frakablak.h`-t dolgozzuk fel. Ez egy header fájl. Ezeket szokás a `#include` kulcsszóval a programunkhoz fűzni, viszont eddig kizárolag előre elkészített header-öket használtunk, de most egy sajátot írtunk. Legfontosabb dolog, hogy a forrás szövegét egy keretbe kell foglalnunk, ez a keret így néz ki:

```
#ifndef FRAKABLAK_H
#define FRAKABLAK_H

...
#endif
```

Az ezek között megjelenő szöveg viszont semmiben nem különbözik az eddig megszokott C++-os forrá-suktótól.

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

Természetesen most is a header fájlok importálásával kezdünk. Itt észreveheted, hogy egyik sem láttuk még egyik forrásban sem. Ezek mind a Qt headerjei. A `QMainWindow` tartalmazza a programunk ablakának létrehozásához szükséges elemeket. A `QImage` segítségével tudunk létrehozni képeket, erőssége, hogy a pixeleket közvetlenül tudjuk elérni. A `QPainter` arra szolgál, hogy a képet színezhesse. A `QMouseEvent` és a `QKeyEvent` segítségével tudjuk megoldnai, hogy az egér vagy billentyű használatával folyamatokat indítsunk el a programban. Például, hogy az egérrel való kijelölés hatására a program ránagyítson a halmazt. Az imént felsorolt Qt header-ök minden osztályt tartalmaznak, melyek neve megegyezik a header-ök nevével. A `farktszal.h`-ről pedig a későbbiekben részletesen beszélünk.

```
class FrakSzal;

class FrakAblak : public QMainWindow
{
    Q_OBJECT

public:
    FrakAblak(double a = -2.0, double b = .7, double c = -1.35,
               double d = 1.35, int szelesseg = 600,
               int iteraciosHatar = 255, QWidget *parent = 0);
    ~FrakAblak();
    void vissza(int magassag, int * sor, int meret) ;
    void vissza(void) ;
    // A komplex sík vizsgált tartománya [a,b]x[c,d].
    double a, b, c, d;
    // A komplex sík vizsgált tartományára feszített
    // háló szélessége és magassága.
    int szelesseg, magassag;
    // Max. hány lépésig vizsgáljuk a z_{n+1} = z_n * z_n + c iterációt?
    // (tk. most a nagyítási pontosság)
    int iteraciosHatar;
```

```
protected:  
    void paintEvent (QPaintEvent *);  
    void mousePressEvent (QMouseEvent *);  
    void mouseMoveEvent (QMouseEvent *);  
    void mouseReleaseEvent (QMouseEvent *);  
    void keyPressEvent (QKeyEvent *);  
  
private:  
    QImage* fraktal;  
    FrakSzal* mandelbrot;  
    bool szamitasFut;  
    // A nagyítandó kijelölt területet bal felső sarka.  
    int x, y;  
    // A nagyítandó kijelölt terület szélessége és magassága.  
    int mx, my;  
    std::vector<int> zX, zY, zX2, zY2;  
};
```

A header állományok deklarációkat tartalmaznak, tehát a függvények és változók definiálása nem itt történik. A Frakszal osztály teljes deklarációját a frakszal.h tartalmazza. Csak azért vol szükség arra, hogy deklárájuk itt, hogy az elemeit a FrakAblak osztály fel tudja használni. Ennek az osztálynak a szülő osztály a QMainWindow, ami annyit jelent, hogy minden elemét a QMainWindow osztálynak, ami public részben van eltudjuk érni egy FrakAblak objektumon keresztül is. A Q_OBJECT makró teszi lehetővé, hogy a program kezelni tudja a Qt C++ kiegészítőit. Ezután pedig a public elemeit dekláraljuk az osztálynak. A public azt jelenti, hogy ezeket az osztályon kívül el tudjuk érni. Első, amit dekláralni kell, az a konstruktur. Ez egy függvény, ami akkor hívódik meg, amikor egy objektumot létrehozunk, neve mindenig megegyezik az osztály nevével. Ennek segítségével a változók értékét inicializáljuk alap értékkel, ha a felhasználó nem adna meg paramétereket. Ezt követ a destruktur, ami a ~ jelről imerszik meg. A többi eleme a public résznek pedig változók és függvények. Ami érdekesebb az a protected rész, ez olyan elemet taratlmaz, amik csak az osztályból érehtőek el, vagy a gyermek osztályokból. Itt vannak dekláralva azok a függvények, amivel a billentyű és egér által küldött jeleket feldolgozzuk. A private rész pedig kizárolag az osztályból érhető el. Ezt azért alkalmazzák a programozók, hogy megvédjék az osztály fontosabb elemeit attól, hogy valaki az osztályon kívül módosítsa. Ezeknek az osztályon kívüli elérésére általában egy függvény segítségével nyújtanak lehetőséget a programozók, mely csupán az értéküket adja vissza. Ezzel a header végére értünk, nézzük, hogy milyen definíciók tartoznak az imént deklarált változókhöz.

```
FrakAblak::FrakAblak(double a, double b, double c, double d,  
                      int szelesseg, int iteraciosHatar, QWidget *parent)  
    : QMainWindow(parent)  
{  
    setWindowTitle("Mandelbrot halmaZ");  
  
    szamitasFut = true;  
    x = y = mx = my = 0;  
    this->a = a;  
    this->b = b;  
    this->c = c;
```

```
this->d = d;
this->szelesseg = szelesseg;
this->iteraciosHatar = iteraciosHatar;
magassag = (int)(szelesseg * ((d-c) / (b-a)));

setFixedSize(QSize(szelesseg, magassag));
fraktal= new QImage(szelesseg, magassag, QImage::Format_RGB32);

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

Elsőnek a konstruktur nézzük meg. A FrakAblak osztály konstruktora a :: hatóköroperátorral tudjuk elérni. Egy érdekesség, hogy a FrakAblak konstruktora a QMainWindow osztály konstruktora is meghívja. A setWindowTitle függvénytel van lehetőségeink beállítani a QMainWindow által létrehozott ablak nevét beállítani. Amikor létrehozzuk a FrakAblak objektumot a szamitasFut logikai értéket igazra állítjuk. Mivel kezdetben nincs kijelölt terület, ezért az x, y változókat, melyek a kijelölt terület bal felső sarkát adják meg, nullára állítjuk. Ugyanígy járunk el a mx, my változókkal is, melyek a terület szélességét adják meg. Hogy jelöljük melyik változó név tartozik az objektumhoz, és melyik a paraméterhez, ezért this-> operátort használjuk. Ez jelöli, hogy az objektum saját változójáról van szó. Tehát ezen a részen átadjuk a paramétereik értékeit a megfelelő változóknak. Többek között paraméterként adjuk meg a komplex számsík vizsgálandó területét, azt, hogy erre a területre milyen széles hálót feszítsünk ki. A magasságot viszont már ezek alapján adjuk meg, és típuskényszerítést alkalmazunk, hogy egész számot kapjunk. Ha már tudjuk a magasságot és a szélességet, az ablakot fix nagyságúra állítjuk a setFixedSize függvénytel. A fraktal pointer ráállítjuk egy QImage objektumra, melyhez a new operátort használjuk. A mandelbrot pointer pedig egy FrakSzal objektumot fog címezni. Majd a start() függvény segítségével elindítjuk a szál végrehajtását.

```
FrakAblak::~FrakAblak()
{
    delete fraktal;
    delete mandelbrot;
}
```

A destruktornban pedig a fraktal és mandelbrot pointerek által mutatott területet szabadítjuk fel.

```
void FrakAblak::paintEvent(QPaintEvent*)
{
    QPainter qpainter(this);
    qpainter.drawImage(0, 0, *fraktal);
    if(!szamitasFut)
    {
        qpainter.setPen(QPen(Qt::white, 1));
        qpainter.drawRect(x, y, mx, my);
        if(!zX.empty()) //excuse me
        {
            for(int i=0; i<zX.size(); i++)
            {
                qpainter.drawLine(zX[i], zY[i], zX2[i], zY2[i]);
            }
        }
    }
}
```

```
    }

    qpainter.end();
}
```

A paintEvent függvény, ahogy a neve is mutatja a kép kirajzolásáért felelős. Átadunk egy pointert, ami a kirajzolni kívánt doogra mutat. Majd példányosítunk egy QPainter objektumot, melynek átadjuk a paraméterként kapott pointert. Ezután a drawImage függvény segítségével kirajzoltatjuk a farktal által mutatott kép objektumot az ablak (0,0)-ás koordinátájától kezdve. Ha nem fut éppen számítás, akkor a toll színét fehérre állítjuk, és kirajzolunk egy téglalapot vele a megadott (x,y) koordinátából, a megadott mx*my-os méretben. Abban az esetben, ha a zX vektor nem üres, akkor végig megyünk a vektoron és vonalakat rajzolunk ki az (x,y) és (x2,y2) koordináták között. Végezetül pedig, mivel a QPainter objektumot minden fel kell szabadítani, ezért az end() tagfüggvénye segítségével ezt meg is tesszük.

```
void FrakAblak::mousePressEvent (QMouseEvent* event) {

    if (event->button() == Qt::LeftButton)
    {

        // A nagyítandó kijelölt területet bal felső sarka:
        x = event->x();
        y = event->y();
        mx = 0;
        my = 0;
    }
    else if(event->button() == Qt::RightButton)
    {
        double dx = (b-a)/szelesseg;
        double dy = (d-c)/magassag;
        double reC, imC, reZ, imZ, ujreZ, ujimZ;

        int iteracio = 0;

        reC = a+event->x()*dx;
        imC = d-event->y()*dy;

        reZ = 0;
        imZ = 0;
        iteracio = 0;

        while(reZ*reZ + imZ*imZ < 4 && iteracio < 255) {
            // z_{n+1} = z_n * z_n + c
            ujreZ = reZ*reZ - imZ*imZ + reC;
            ujimZ = 2*reZ*imZ + imC;
            zX.push_back((int)((reZ - a)/dx));
            zY.push_back((int)((d - imZ)/dy));
            zX2.push_back((int)((ujreZ - a)/dx));
            zY2.push_back((int)((d - ujimZ)/dy));
            reZ = ujreZ;
            imZ = ujimZ;
            iteracio++;
        }
    }
}
```

```
    imZ = ujimZ;

    ++iteracio;
}

update();
}
```

A mousePressEvent függvény azt határozza meg, hogy mi történjen, ha felhasználó csak rákattint az egérrel képre. Paraméteréül egy QMouseEvent objektumra mutató pointert kap. Ennek az objektumnak a button() függvényével tudjuk lekérdezni, hogy melyik gombot nyomta le a felhasználó. Ha a balt, akkor egy pixelre nagyítunk rá, vagyis vagy teli zöld, vagy teli fekete lesz az ablak. Ezt onnan tudjuk, hogy a nagyítandó terület méretét meghatározó változók nullára vannak állítva. Ha jobb gomb kerül lenyomásra, akkor a memgadott pontból kezdődően kiszámoljuk az z_n komplex számokat, és ezeket berakjuk egy tömbbe. Addig megyünk a while ciklusban, ameddig a z_n értéke kisebb, mint 2, vagy ameddig nem érjük el az iterációs határt. Az itt kiszámolt pontokat fogja összekötni a paintEvent() függvény for ciklusa futása során. Az update() segítségével pedig frissítjük a kirajzolt képet, hogy azonnal változzon a felhasználó behatására.

```
void FrakAblak::mouseMoveEvent(QMouseEvent* event) {

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

    update();
}
```

Ahhoz, hogy az egér mozgását kezeljük a mouseMoveEvent függvényt definiáljuk. Ahogy mousePressEvent függvénynél említettem, abban az esetben, ha csak a bal egérgombra nyomunk rá, akkor csak egy pixelre nyagyítunk rá. Akkor nem tettem hozzá, hogy ez csak akkor igaz, ha mozgatás nélkül engedjük fel az egérgombot. Ha viszont mozgatjuk, akkor a mousePressEvent függvényben megkapott x érték és a az utolsó egér pozíció x koordinátája alapján meghatározzuk a nagyítandó terület szélességét. Itt is meghívjuk a update() függvényt, hogy egyből látható legyen a behatás. Hogy teljes legyen a kép még van egy függvény, ami azt kezeli, amikor elengedjük az egeret.

```
void FrakAblak::mouseReleaseEvent(QMouseEvent* event) {

    if(szamitasFut)
        return;

    szamitasFut = true;

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

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

```
double d = this->d-y*dy;

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

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

update();
}
```

Ha éppen fut a halmaz kiszámítása, akkor nem dolgozzuk fel gombelengedést, egyébként pedig beállítjuk az egér mozgatásával meghatározott terület mérete alapján meghatározott síktartományt, amit vizsgálunk. Az éppen kirajzolt ábrát pedig dobjuk, azaz a `mandelbrot` pointer által mutatott objektumot töröljük. Majd foglalunk helyet egy újnak, aminek átadjuk a megújult paramétereket. Végezetül pedig elindítjuk a szálat, frissítünk.

Ezt most egy kicsit nehezen állhat össze a fejekben, de összegezzük, hogy mi is történik. Amikor rákattintunk a bal egérgombra, akkor meghívódik a `mousePressEvent` függvény, mely az x és y koordináタkat beállítja az egér aktuális pozíójába. Abban az esetben, ha nem mozgatjuk az egeret, csak elegedjük a gombot, meghívódik a `mouseReleaseEvent` függvény, az kiszámolja a megadott koordináták és szélesség alapján a Mandelbrot halmazt, ami lényegében azt eredményezi, hogy egy pixelre ránagyít a program. Ha mozgatjuk az egeret, akkor meghívódik a `mouseMoveEvent`, és ezzel párhuzamosan a `paintEvent` függvéyn, ami folyamatosan rajzolja azt a méretű négyzetet, amit az egér mozgatásával megadunk. Ha felengedjük a gombot, `mouseReleaseEvent` ránagyít az általunk meghatározott területre. Ugyan ezek a folyamatok zajlanak le, akkor is, ha a jobb egérgombot nyomjuk le, azzal a különbséggel, hogy akkor a z_n számokat összeköti az adott ponttól kezdve. Azt fontos látni, hogy nem konkrétan a `mouseReleaseEvent` függvény nagyítja ki a képet, hanem a vissza függvény a `FraktalSzal` objektum által visszadott értékek alapján.

A program abban az esetben, ha lenyomjuk az n billentyűt, akkor a program újraszámolja a Mandelbrot halmazt. Ennek a kezeléséért a `keyPressEvent` függvény felelős. Ha a halamaz számítása fut, akkor nem csinál semmit, amíg újra kiszámolja a halmazt. Ha az n billentyűt lenyomjuk, akkor az iterációs határt a duplájára növeli. Ezzel pontosabb lesz a halmaz kiszámítása.

```
void FrakAblak::keyPressEvent (QKeyEvent *event)
{
    if (szamitasFut)
        return;

    if (event->key() == Qt::Key_N)
        iteraciosHatar *= 2 //bit shift: iteraciosHatar = iteraciosHatar ←
            << 1;
    szamitasFut = true;
```

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

}
```

Ahhoz, hogy megjelenítsük az halmazt, a vissza függvényt fogjuk használni, ahogy már korábban említem.

```
void FrakAblak::vissza(int magassag, int *sor, int meret)
{
    for(int i=0; i<meret; ++i) {
        QRgb szin = qRgb(0, 255-sor[i], 0);
        fraktal->setPixel(i, magassag, szin);
    }
    update();
}
```

A függvény megkapja paraméteréül az éppen kirajzolandó sor y koordinátáját, egy tömbre mutató pointert, ami az éppen aktuális iteráció számát mutatja a Mandelbrot halmaz számolása során, és az ablak szélességét. A for ciklus a kép pixeleit beállítja a megfelelő színekre. Ehhez létrehoz egy QRgb objektumot, amivel egy színt tárolunk el, majd pedig ezt a színt, és az aktuális (x,y) koordinátákat adjuk a setPixel() függvénynek.

```
void FrakAblak::vissza(void)
{
    szamitasFut = false;
    x = y = mx = my = 0;
}
```

Van a programunkban egy másik vissza függvény. Ez nem zavarja meg a fordítót, mert a paraméterlista alapján meg tudja őket különböztetni. Ez a függvény nem kér paramétert, csak visszaállítja a kijelölés értékeit nullára, és a szamitasFut változó értékét hamisra változtatja. Ez akkor hívódik meg, amikor megtörténik a Mandelbrot halmaz kirajzolása.

Most nézzük meg, hogy a már emlegetett FrakSzal osztály milyen feladatokért felelős. Ennek is elsőnek a deklarációját nézzük meg, amit a frakszal.h tartalmaz.

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

class FrakAblak;

class FrakSzal : public QThread
{
    Q_OBJECT

public:
    FrakSzal(double a, double b, double c, double d,
```

```
        int szelesseg, int magassag, int iteraciosHatar, FrakAblak * ←
        frakAblak);
~FrakSzal();
void run();

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

};
```

Az első header állományról tegyük emléket, a többi már úgyis ismerős. A QThread szintén egy osztály, ami azt teszi lehetővé, hogy programszálakat kezeljünk a programunkban. Ez az osztály szülőosztálya a FraktalSzal osztályunknak. Itt is alkalmazzuk a Q_OBJECT makrót, ami lehetővé teszi a fordító számára a Qt-s kiegészítők felismerését. A public részben megtalálható a konstruktur és a destruktur, valamint a run() függvény. A protected részben lévő változók pedig, már ismertek a FrakAblak osztályból. Kivéve a frakAblak pointert, ami egy pointer egy FrakAblak objektumra. Az egySor pedig egy egészre mutató mutató. Lássuk ezeknek a definiálását:

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

    egySor = new int[szelesseg];
}
```

A konstruktornban inicializáljuk a változók értékét a paraméterként kapott értékekkel, és létrehozunk egy szelesseg értékével megyegyező méretű, egész számokból álló tömböt, amire ráállítjuk az egySor pointert.

```
FrakSzal::~FrakSzal()
```

```
{  
    delete[] egySor;  
}
```

A destruktörben pedig szokás szerint felszabadítjuk az imént említett pointer által mutatott tömböt. Azt, hogy tömbről van szó, explicit módon ki kell írni.

```
void FrakSzal::run()  
{  
    // A [a,b]x[c,d] tartományon milyen sűrű a  
    // megadott szélesség, magasság háló:  
    double dx = (b-a)/szelesseg;  
    double dy = (d-c)/magassag;  
    double reC, imC, rez, imZ, ujrez, ujimZ;  
    // Hány iterációt csináltunk?  
    int iteracio = 0;  
    // Végigzongorázzuk a szélesség x magasság hálót:  
    for(int j=0; j<magassag; ++j) {  
        //sor = j;  
        for(int k=0; k<szelesseg; ++k) {  
            // c = (reC, imC) a háló rácspontjainak  
            // megfelelő komplex szám  
            reC = a+k*dx;  
            imC = d-j*dy;  
            // z_0 = 0 = (rez, imZ)  
            rez = 0;  
            imZ = 0;  
            iteracio = 0;  
            // z_{n+1} = z_n * z_n + c iterációk  
            // számítása, amíg |z_n| < 2 vagy még  
            // nem értük el a 255 iterációt, ha  
            // viszont elértek, akkor úgy vesszük,  
            // hogy a kiinduláci c komplex számra  
            // az iteráció konvergens, azaz a c a  
            // Mandelbrot halmaz eleme  
            while(rez*rez + imZ*imZ < 4 && iteracio < iteraciosHatar) {  
                // z_{n+1} = z_n * z_n + c  
  
                ujrez = rez*rez - imZ*imZ + reC;  
                ujimZ = 2*rez*imZ + imC;  
  
                rez = ujrez;  
                imZ = ujimZ;  
  
                ++iteracio;  
            }  
            // ha a < 4 feltétel nem teljesült és a  
            // iteráció < iterációsHatár sérülésével lépett ki, azaz  
            // feltesszük a c-ről, hogy itt a z_{n+1} = z_n * z_n + c
```

```
// sorozat konvergens, azaz iteráció = iterációsHatár
// ekkor az iteráció %= 256 egyenlő 255, mert az esetleges
// nagyítások során az iteráció = valahány * 256 + 255

iteracio %= 256;

// a színezést viszont már majd a FrakAblak osztályban lesz
egySor[k] = iteracio;
}

// Ábrázolásra átadjuk a kiszámolt sort a FrakAblak-nak.
frakAblak->vissza(j, egySor, szelesseg);
}

frakAblak->vissza();

}
```

A `run()` függvény számolja ki a Mandelbrot halmazt, és minden sor kiszámolása után meghívja a `vissza()` függvényt, ami kirajzolja azt a sort az ablakra. A `for` ciklus addig fut, ameddig az ablakban megjelenő összes sort ki nem számolta, vagyis ameddig a ciklusváltozó eléri a magasságot. Ezután pedig meghívódik a `vissza()` függvény, ami a kijelölés értékeit állítja vissza, és jelzi a `FrakAblak` objektumnak, hogy a számítás befejeződött.

Ezen hosszú elemzés végeztével pedig lássuk, mint is rejt magában a `main()` függvény, mely a `main.cpp`-ben található.

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

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

    FrakAblak w1;
    w1.show();

    return a.exec();
}
```

Ehhez a fájlhoz importáltuk a `QApplication` headert, amire azért volt szükség, mert a `QApplication` osztály teszi lehetővé, hogy grafikus felületű programokat kezeljünk. A `main` függvényben példányosítunk egy objektumot, ami az imént említett osztályú. Létrehozunk egy `FrakAblak` típusú objektumot `w1` néven. Majd meghívjuk ennek az objektumnak a `show()` függvényét. Jogosan merül fel a kérdés, hogy ez a függvény hol is van definiálva. A `QWidget` osztályban, melyet azért tudunk elérni, mert a szülő osztálya a `QMainWindow`, ami a `FrakAblak` osztályunknak a szülő osztálya is. Ez a függvény jeleníti meg az ablakot, amiben a program fut. Az a objektum `exec()` függvénye pedig addig fut, ameddig a program be nem záródik. Ha minden rendben lefutott, akkor nullával tér vissza, ellenkező esetben valamilyen hibakódval.

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

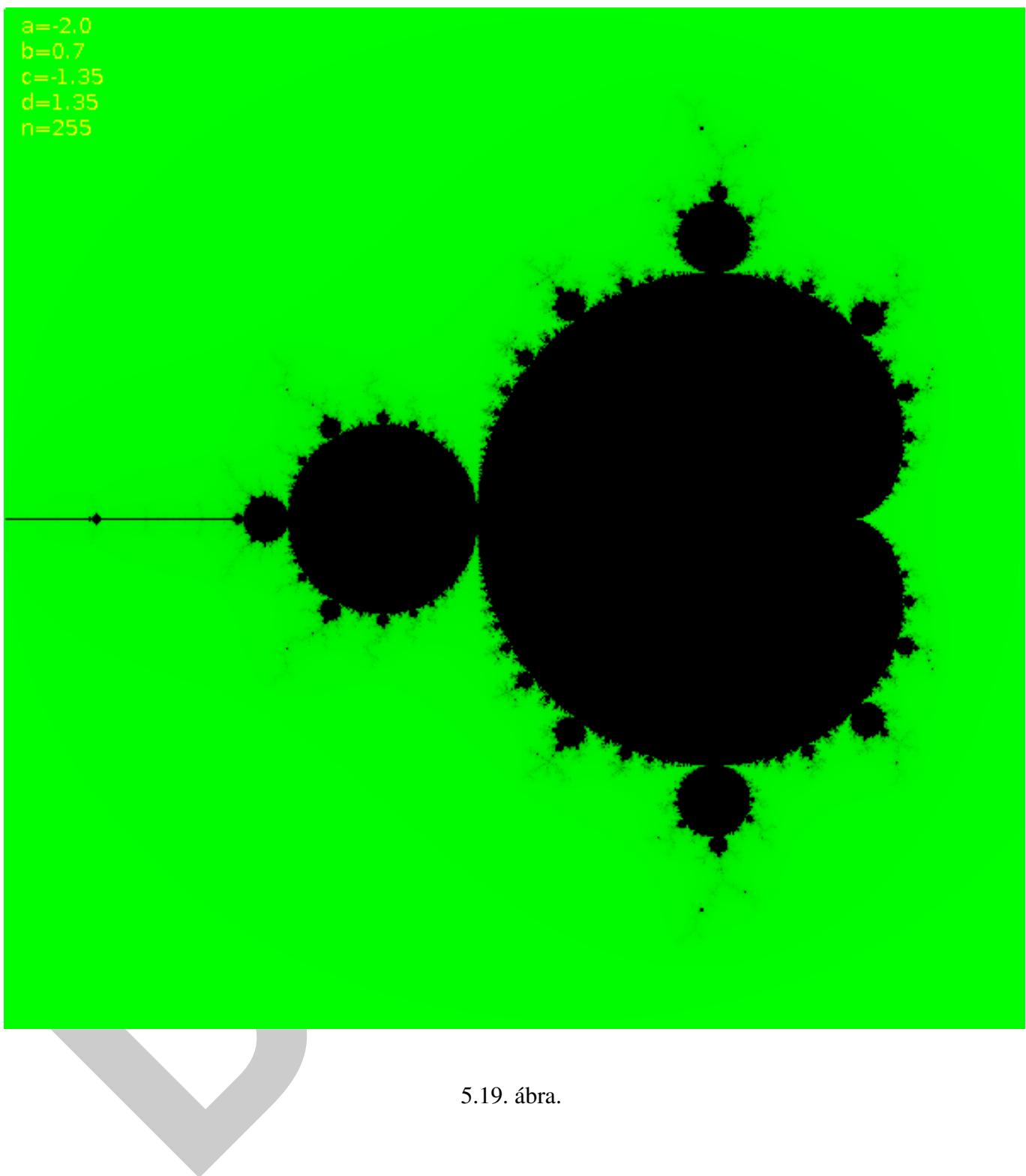
Megjegyzés

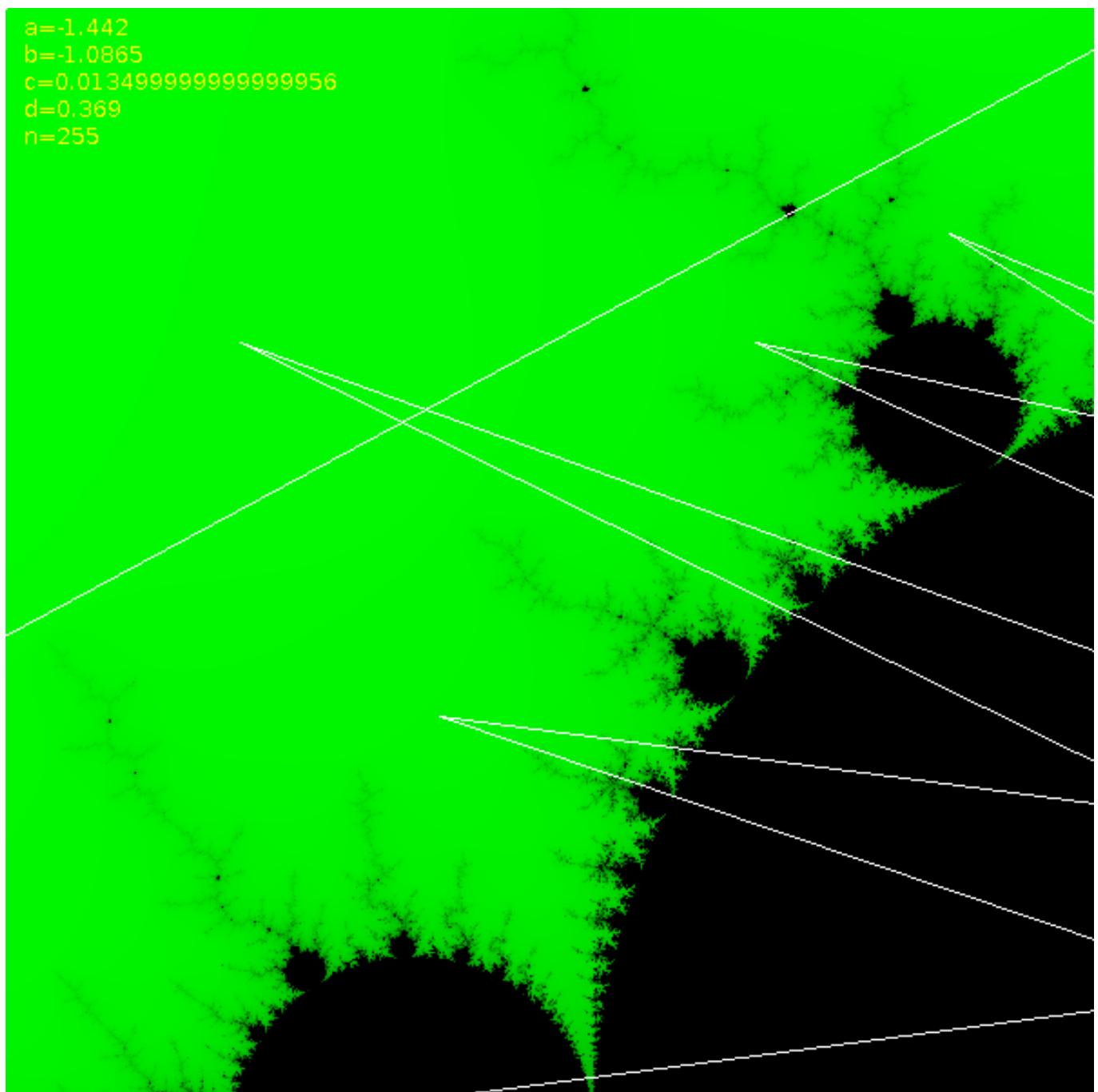
Telepíteni: sudo apt-get install openjdk-8-jdk

Fordítás, futtatás:

```
javac MandelbrotHalmazNagyító.java  
java MandelbrotHalmazNagyító
```

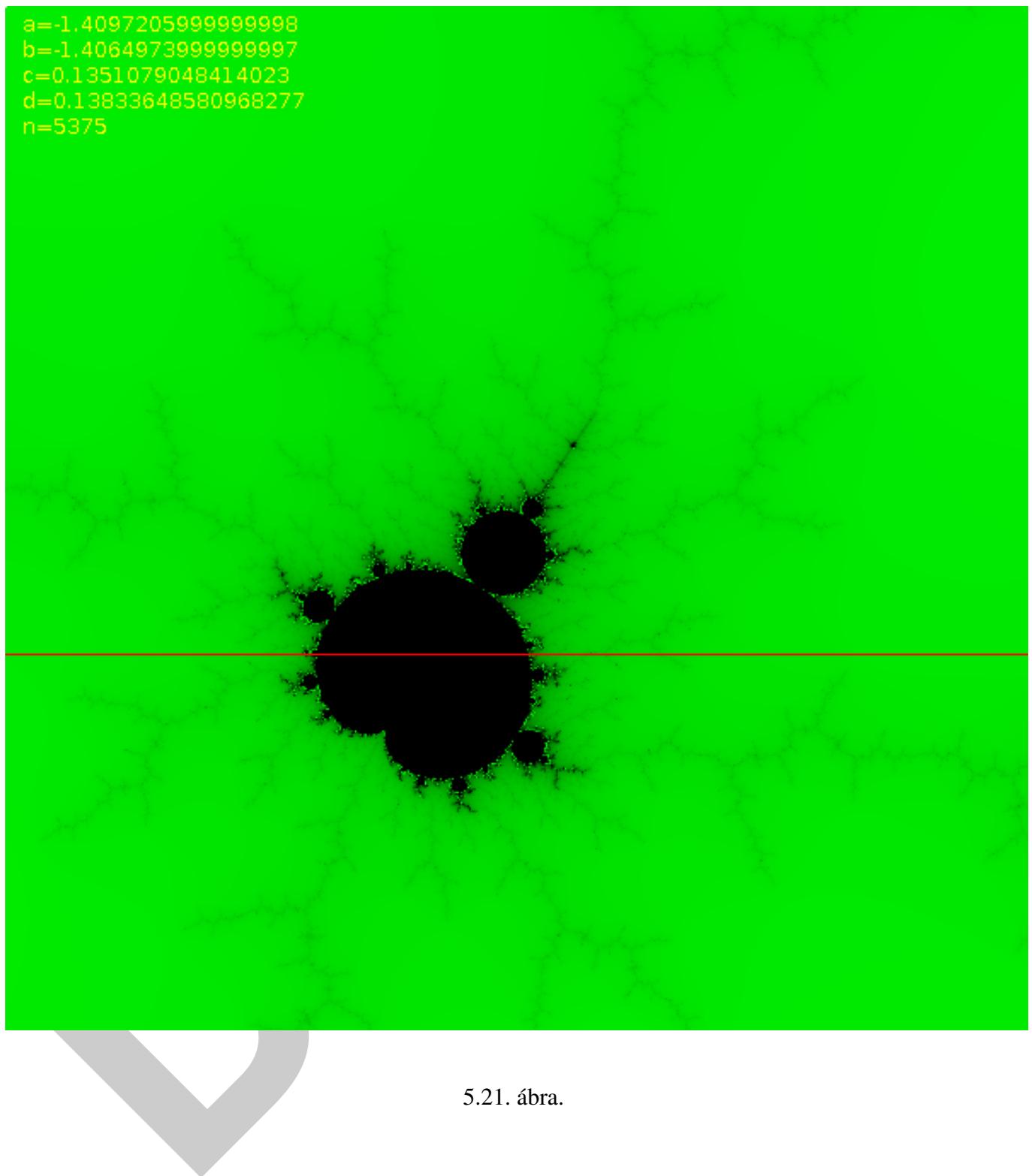
Az előző feladatban készített nagyító Java implementációját kell most elkészíteni. A program teljesen ugyanazokkal a funkciókkal rendelkezik, mint a fentebb megismert C++-os verzió, annyi különbséggel, hogy itt az "s" gomb lenyomásával lehetőségünk van pillanatfelvételt készíteni a programról. A kép bel felső sarkában láthatjuk az éppen vizsgált tartomány koordinátáit, és az iterációs határt is. Az iterációs határ növelésére ez a program egy másik lehetőséget is nyújt az "n" billentyűn kívül. Az "m" gomb lenyomásával megtízszeri a program az éppen aktuális iterációs határt, mellyel a program újra számolja halmazt. Egy másik újdonság az előzőhöz képest, hogy itt egy piros vonal jelöli azt, hogy melyik sorban tart éppen a rajzolás. Nézzünk is meg néhány képet a programról:(A beépített pillanatfelvétő funkcióval készültek.)





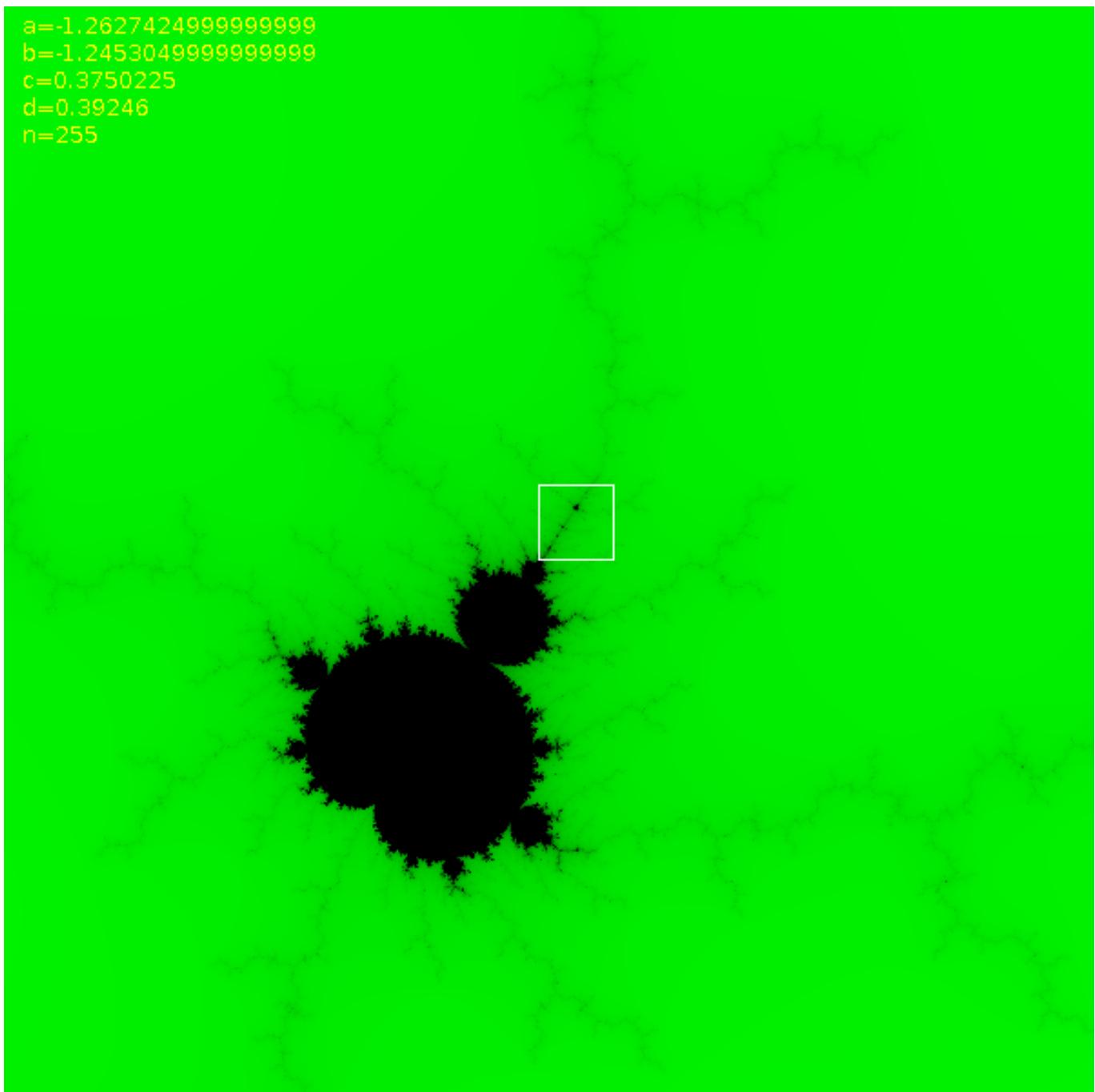
5.20. ábra.

```
a=-1.4097205999999998  
b=-1.4064973999999997  
c=0.1351079048414023  
d=0.13833648580968277  
n=5375
```



5.21. ábra.

```
a=-1.2627424999999999  
b=-1.2453049999999999  
c=0.3750225  
d=0.39246  
n=255
```



5.22. ábra.

Most pedig nézzük meg a forrást, mely ismét több fájlból áll, egészen pontosan 2-ből. A MandelbrotHalmaz.java egy önmagában is működő program, ellentétben az előző feladatban megismert fájlokkal, mivel azok teljesen egymásra épültek. Ez a forrás tartalmazza a MandelbrotHalmaz kiszámításához szükséges algoritmust. Lássuk a forráskódját:

```
import java.awt.Color;  
public class MandelbrotHalmaz extends java.awt.Frame implements Runnable {  
    protected double a, b, c, d;  
    protected int szélesség, magasság;
```

```
protected java.awt.image.BufferedImage kép;
protected int iterációsHatár = 255;
protected boolean számításFut = false;
protected int sor = 0;
protected static int pillanatfelvételSzámláló = 0;
```

Az import kulcsszó segítségével tudjuk a szükséges osztályokat hozzáadni a programunkhoz. Java-ban lényegében az egész program egy osztálynak fele meg. Ennek megfelelően létrehozunk deklarálunk egy osztályt MandelbrotHalmaz néven. Az extends segítségével az osztályunkat kiegészítjük a Frame osztálytal. Lényegében ez ahhoz hasonlít, mint a C++ forrásban egy szülő osztályt adtunk meg. A Frame osztály felelős azért, hogy megalkossuk a programablakot. Az implements kulcszsalval pedig implementáljuk a Runnable interfészt. Ez abban különbözik egy osztálytól, hogy ebben nem tudjuk a deklarált elemeit definiálni, csak az osztályon keresztül, ami implementálja. A Runnable interfész amúgy a programszálakér felelős. minden osztálynak implementálnia kell, amelyeket egy szalon szeretnénk futtatni. Láthatjuk, hogy itt is megjennek a már ismert változók, amik meghatározzák a komplex számsík vizsgált [a,b]x[c,d] tartományát, ennek a tartománynak a magasságát, szélességét és a Mandelbrot halmaz számítási pontosságát meghatározó iterációs határt is. Itt is van egy logikai változónk, a számításFut, mely azt tárolja, hogy fut-e a halmaz számítása, vagy sem. Alapértelmezetten hamisra van állítva. Különleges lehet, hogy Java-ban nem lehet csoportosítani a public, private, protected elemeket. Mindegyik elő külön ki kell írni. Most nézzük meg az osztály konstruktörét:

```
public MandelbrotHalmaz(double a, double b, double c, double d,
    int szélesség, int iterációsHatár) {
    this.a = a;
    this.b = b;
    this.c = c;
    this.d = d;
    this.szélesség = szélesség;
    this.iterációsHatár = iterációsHatár;
    this.magasság = (int)(szélesség * ((d-c) / (b-a)));
    kép = new java.awt.image.BufferedImage(szélesség, magasság,
        java.awt.image.BufferedImage.TYPE_INT_RGB);
    addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosing(java.awt.event.WindowEvent e) {
            setVisible(false);
            System.exit(0);
        }
    });
}
```

Hasonlóan a C++-os forráshoz, itt is paraméterként kéri a számításhoz szükséges változók értékét. Mivel Java-ban nincsenek pointerek, csak referenciák, ezért az osztályunk változóit a . operátorral érjük el, nem ->. A magasságot kiszámítjuk a vizsgált terület és a megadott szélesség alapján. Majd példányosítunk egy BufferedImage objektumot a new operátorral, mely területet foglal a megadott objektumnak, és visszatér egy referenciával. A BufferedImage osztály konstruktörának meg kell adnunk a kép méretét és típusát. Jelen esetben a képünk típusa TYPE_INT_RGB, ami azt jeenti, hogy RGB színeket használunk, és az egyes színkomponenseket 8-bites számokkal reprezentáljuk. Végül pedig meghívjuk a Frame osztály tagfüggvényét, a addwindowListener () -t. Ez a függvény azért felelős, hogy kezelní tudjuk az ablakon történő behatásokat, most kifejezetten azt, hogy az ablak záródjon be, ha az "x"-re kattintunk. A billentyűzetről érkező bemenetket is a konstruktörban kezeljük.

```
addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyPressed(java.awt.event.KeyEvent e) {
        if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)
            pillanatfelvétel();
        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {
            if(számításFut == false) {
                MandelbrotHalmaz.this.iterációsHatár += 256;
                számításFut = true;
                new Thread(MandelbrotHalmaz.this).start();
            }
        } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_M) {
            if(számításFut == false) {
                MandelbrotHalmaz.this.iterációsHatár += 10*256;
                számításFut = true;
                new Thread(MandelbrotHalmaz.this).start();
            }
        }
    }
});
```

A feladat bevezetőjében már említettem, hogy ebben a forrásban több billentyűt is kezelünk, egészen pontosan hármat. Ahhoz, hogy a billentyűk lenyomását érzékelje a program a addKeyListener függvényre van szükségünk. A konkrét lenyomásokat a program a KeyAdapter() osztály segítségével dolgozza fel. A KeyListener osztály tagfüggvényét, a keyPressed-et használva el tudjuk érni, hogy pontosan melyik billentyű lett lenyomva, és annak függvényében utasításokat végrehajtani. Ha az "s" billentyűt nyomjuk le, akkor meghívódik a pillanatfelvétel() függvény, "n" esetén duplázzuk az iterációs határt, majd újra számoljuk a halmazt. Ugyan ezt tesszük az "m" lenyomásakor is, csak akkor az iterációt a tízszeresére növeljük.

```
// Ablak tulajdonságai
setTitle("A Mandelbrot halmaz");
setResizable(false);
setSize(szélesség, magasság);
setVisible(true);
// A számítás indul:
számításFut = true;
new Thread(this).start();
}
```

A konstruktur végén pedig az ablakot állítjuk be, amiben a program megjelenik. Majd elindítjuk a halmaz számítását.

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

```
    }  
}
```

A `paint()` egy metódus, mely arra szolgál, hogy az aktuális halmazt kirajzoljuk az ablakba. Ez úgy zajlik, hogy paraméterként kap egy `Graphics` objektumot. Ennek a `drawImage` függvényének megadjuk, hogy milyen képet szeretnénk kirajzolni, hova az ablakon belül, és melyik ablakba. Egy piros vonallal pedig ábrázoljuk a számítás előrehaladását. Ehhez elsőnek be kell állítani a színt, amit használnánk, ezt a `setColor` tagfüggvénytel tudjuk állítani. A `drawLine` függvény pedig a megadott két pont közé húz egy egyenest.

```
public void update(java.awt.Graphics g) {  
    paint(g);  
}
```

Az alapértelmezett `update()` függvényt pedig felül definiáljuk, annak érdekében, hogy az újraszámolás során ne fehéredjen ki a kép egy pillanatra. Az előző feladatban látott programhoz lépest újdonság a `pillanatfelvétel()` eljárás. Ennek segítségével maga a program készít egy képet, nincs szükség a `PrintScreen` használatára.

```
public void pillanatfelvétel() {  
    // Az elmentendő kép elkészítése:  
    java.awt.image.BufferedImage mentKép =  
        new java.awt.image.BufferedImage(szélesség, magasság,  
            java.awt.image.BufferedImage.TYPE_INT_RGB);  
    java.awt.Graphics g = mentKép.getGraphics();  
    g.drawImage(kép, 0, 0, this);  
    g.setColor(java.awt.Color.YELLOW);  
    g.drawString("a=" + a, 10, 15);  
    g.drawString("b=" + b, 10, 30);  
    g.drawString("c=" + c, 10, 45);  
    g.drawString("d=" + d, 10, 60);  
    g.drawString("n=" + iterációsHatár, 10, 75);  
    g.dispose();  
    // A pillanatfelvétel képfájl nevének képzése:  
    StringBuffer sb = new StringBuffer();  
    sb = sb.delete(0, sb.length());  
    sb.append("MandelbrotHalmaz_");  
    sb.append(++pillanatfelvételszámláló);  
    sb.append("_");  
    // A fájl nevébe belelevesszük, hogy melyik tartományban  
    // találtuk a halmazt:  
    sb.append(a);  
    sb.append("_");  
    sb.append(b);  
    sb.append("_");  
    sb.append(c);  
    sb.append("_");  
    sb.append(d);  
    sb.append(".png");  
    // png formátumú képet mentünk
```

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

A kép elkészítéséhez szükség van arra, hogy létrehozzunk egy új képobjektumot. Ezt nevezzük mentképnak. Majd a képből készítünk egy grafikus objektumot a `getGraphics()` függvény segítségével. Ezzel lehetővé tesszük, hogy a ki tdujuk rá írni a kép-ben tárolt halmazt, a szükséges értékeket. Szöveget a képre a `drawString` függvénnyel tudjuk megtenni. A `dispose()` pedig bezára az ablakot, amiben a kép tartalmának a kiírását végezzük. Majd a kép nevét adjuk meg. Ahogy a forrásban is látható, ez tartalmazza a halmaz nevét, hogy hanyadik pillanatképet készítjük, és azt is, hogy melyik tartományban találtuk a halmazt. Végezetül a `javax.imageio.ImageIO.write()` függvény segítségével kiírjuk a `mentKép` tartalmát egy png fájlba. A célállományt a `java.io.File()` segítségével hozzuk létre.

```
public void run() {
    double dx = (b-a)/szélesség;
    double dy = (d-c)/magasság;
    double reC, imC, reZ, imZ, ujreZ, ujimZ;
    int rgb;
    int iteráció = 0;
    for(int j=0; j<magasság; ++j) {
        sor = j;
        for(int k=0; k<szélesség; ++k) {
            // c = (reC, imC) a háló rácspontjainak
            // megfelelő komplex szám
            reC = a+k*dx;
            imC = d-j*dy;
            // z_0 = 0 = (reZ, imZ)
            reZ = 0;
            imZ = 0;
            iteráció = 0;
            while(reZ*reZ + imZ*imZ < 4 && iteráció < iterációsHatár) {
                // z_{n+1} = z_n * z_n + c
                ujreZ = reZ*reZ - imZ*imZ + reC;
                ujimZ = 2*reZ*imZ + imC;
                reZ = ujreZ;
                imZ = ujimZ;

                ++iteráció;
            }

            iteráció %= 256;
            Color bg = new Color(0, 255-iteráció, 0);
            rgb = bg.getRGB();
            kép.setRGB(k, j, rgb);
        }
    }
}
```

```
        repaint();
    }
    számításFut = false;
}
```

A Mandelbrot halmaz kiszámolását a megadott tartományban a `run()` függvény végzi. Ez a Runnable interfészen lett deklarálva, de nekünk kell definiálni. Maga az algoritmus a teljes másolata a fejezetben már többször leírtaknak. Egyedüli érdekesség a színezés beállítása. Deklarálunk ehhez egy `rgb` nevű változót. A példányosítunk egy `Color` objektumot `bg` néven, melyben eltároljuk a szükséges színt. Ennek a színnek az RGB felépítése teljesen megyegyezik a C++-os verziójával. Ebből az objektumból a színt a `getRGB()` függvénykel kapjuk meg, ami egy RGB értékkel tér vissza, azaz egy egész számmal. Végül a `setRGB()` függvénykel állítjuk be az egyes pixelek színét. A `repaint()` függvény azért felelős, hogy újraraajzoljuk a halmazt a megváltozott értékek alapján. Ha végeztünk a halmaz számolásával, a `számításFut` értékét hamisra állítjuk.

```
public static void main(String[] args) {
    new MandelbrotHalmaz(-2.0, .7, -1.35, 1.35, 600, 255);
}
```

A `main` függvényben pedig nem teszünk mást, csak példányosítunk egy `MandelbrotHalmaz` objektumot. Ezzel a `MandelbrotHalmaz.java` forrás végére értünk, láthatjuk, hogy a nagyítás során végrehajtandó utasítások nagyrésze itt hajtódi végre.

Nézzük meg, mit tartalmaz a `MandelbrotHalmazNagyító.java` forrás fájl.

```
import java.util.*;
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {
    private int x, y;
    private int mx, my;
    private List<Integer> zX = new ArrayList<>();
    private List<Integer> zX2 = new ArrayList<>();
    private List<Integer> zY = new ArrayList<>();
    private List<Integer> zY2 = new ArrayList<>();
```

A `MandelbrotHalmazNagyító` osztályt kiegészítjük a már kkorábban taglalt `MandelbrotHalmaz` osztállyal. Erre azért van szükség, mert a halmaz számolása abban az osztályban zajlik, de maga a programablakot is ott kezeljük. Az `x`, `y` változók a nagyítandó terület bal felső sarkát határozzák meg. A `mx`, `my` pedig a terület méretét. Az `zX`, `zX2`, `zY`, `zY2` listákban pedig a z_n számokat tároljuk el.

```
public MandelbrotHalmazNagyító(double a, double b, double c, double d,
    int szélesség, int iterációsHatár) {
    super(a, b, c, d, szélesség, iterációsHatár);
    setTitle("A Mandelbrot halmaz nagyításai");
    // Egér kattintó események feldolgozása:
    addMouseListener(new java.awt.event.MouseAdapter() {
        // Egér kattintással jelöljük ki a nagyítandó területet
        // bal felső sarkát:
        public void mousePressed(java.awt.event.MouseEvent m) {
            if (m.getButton() == java.awt.event.MouseEvent.BUTTON1) {
                // A nagyítandó kijelölt területet bal felső sarka:
                x = m.getX();
```

```
        y = m.getY();
        mx = 0;
        my = 0;
    }
    else if(m.getButton() == java.awt.event.MouseEvent.BUTTON3)
    {
        double dx = (b-a)/szélesség;
        double dy = (d-c)/szélesség;
        double reC, imC, reZ, imZ, ujreZ, ujimZ;

        int iteracio = 0;

        reC = a+m.getX()*dx;
        imC = d-m.getY()*dy;

        reZ = 0;
        imZ = 0;
        iteracio = 0;

        while(reZ*reZ + imZ*imZ < 4 && iteracio < 255) {
            // z_{n+1} = z_n * z_n + c
            ujreZ = reZ*reZ - imZ*imZ + reC;
            ujimZ = 2*reZ*imZ + imC;
            zX.add((int)((reZ - a)/dx));
            zY.add( (int)((d - imZ)/dy));
            zX2.add((int)((ujreZ - a)/dx));
            zY2.add((int)((d - ujimZ)/dy));
            reZ = ujreZ;
            imZ = ujimZ;

            ++iteracio;
        }
    }
    repaint();
}
```

Az osztály konstruktora érdekesnek tűnhet első látásra. A `super()` függvény segítségével meg tudjuk hívni a MandelbrotHalmaz osztály konstruktörét. Ezután hozzá tudunk férni az ablak beállításaihoz, tehát át tudjuk állítani a nevét a `setTitle()` függvénytellyel. Ebben a részben valósítjuk meg az egér kezelését a programban. Ha a bal egérgombot megnyomjuk, akkor a kurzor aktuális pozíciója beletöltődik a a kijelölendő terület bal felső sarkát meghatározó változókba, a terület méretét pedig nullázzuk. Ha pedig a jobb egérgombot nyomjuk le, akkor a program kiszámolja attól a ponttól kezdve a z_n számokat, egészen addig, míg el nem érjük az iterációs határt, vagy a z_n értéke nem haladja meg a 2-őt. A kapott értékeket pedig betöltekük a megfelelő listába. Végül meghívjuk a `repaint()` függvényt.

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

```
        - MandelbrotHalmazNagyító.this.c)
        /MandelbrotHalmazNagyító.this.magasság;
// Az új Mandelbrot nagyító objektum elkészítése:
if(m.getButton() == java.awt.event.MouseEvent.BUTTON3) {
    return;
}
dispose();
new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+ ←
    x*dx,
    MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
    MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
    MandelbrotHalmazNagyító.this.d-y*dy,
    600,
    MandelbrotHalmazNagyító.this.iterációsHatár);
}
});
```

A konstruktur része még a egérgomb felengedését kezelő függvény is. Ekkor állítjuk be a vizsgált tartomány hálósűrűségét meghatározó változókat, vagyis a dx és dy-t. Különbséget kell tennünk abban, hogy melyik gombot engedtük fel, mert, ha a jobbot, akkor nem kell új Mandelbrot halmazt létrehozni, vagyis return kiléünk a függvényből. Ellenkező esetben bezárjuk a régi ablakot a dispose() segítségével, majd létrehozunk egy új MandelbrotHalmazNagyító objektumot, ami nyit egy új ablakot. Az új objektumot a nagytásra kijelölt terület alapján határozzuk meg.

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

Ha az egerrel nem csak kattintunk, hanem húzunk is, akkor a kattintáskor lenullázott méret változókat az első és az utolsó x érték különbségére módosítjuk. Mind a magasságot, mind a szélességet, ezzel garantálva, hogy minden négyzetet jelöljünk ki. Ennél is hívjuk a repaint() függvényt. Ennek az osztálynak is van pillanatfelvétel() függvényt.

```
public void pillanatfelvétel() {
    // Az elmentendő kép elkészítése:
    java.awt.image.BufferedImage mentKép =
        new java.awt.image.BufferedImage(szélesség, magasság,
            java.awt.image.BufferedImage.TYPE_INT_RGB);
    java.awt.Graphics g = mentKép.getGraphics();
    g.drawImage(kép, 0, 0, this);
    g.setColor(java.awt.Color.YELLOW);
    g.drawString("a=" + a, 10, 15);
    g.drawString("b=" + b, 10, 30);
    g.drawString("c=" + c, 10, 45);
```

```
g.drawString("d=" + d, 10, 60);
g.drawString("n=" + iterációsHatár, 10, 75);
if(számításFut) {
    g.setColor(java.awt.Color.RED);
    g.drawLine(0, sor, getWidth(), sor);
}
g.setColor(java.awt.Color.WHITE);
g.drawRect(x, y, mx, my);
if (!zX.isEmpty())
{
    for (int i = 0; i<zX.size(); ++i){
        g.drawLine(zX.get(i), zY.get(i), zx2.get(i), zY2.get(i));
    }
}
g.dispose();
// A pillanatfelvétel képfájl nevének képzése:
StringBuffer sb = new StringBuffer();
sb = sb.delete(0, sb.length());
sb.append("MandelbrotHalmazNagyitas_");
sb.append(++pillanatfelvételszámláló);
sb.append("_");
// A fájl nevébe belelevesszük, hogy melyik tartományban
// találtuk a halmazt:
sb.append(a);
sb.append("_");
sb.append(b);
sb.append("_");
sb.append(c);
sb.append("_");
sb.append(d);
sb.append(".png");
// png formátumú képet mentünk
try {
    javax.imageio.ImageIO.write(mentKép, "png",
        new java.io.File(sb.toString()));
} catch(java.io.IOException e) {
    e.printStackTrace();
}
}
```

Ez csak annyiban különbözik az előzőtől, hogy a számítás állapotát jelző piros vonalat is kirajzolja, és a z_n számokat összekötő vonalakat is fehér színnel. Erre azért volt szükség, mert pillanatfelvételt készítünk, így ha éppen fut a számítás, akkor meg kell jeleníteni a piros vonalat is. Vagy ha kirajzoltattuk a z_n számokat az ablakra, akkor a képen is jelenjen ez meg.

```
public void paint(java.awt.Graphics g) {
    // A Mandelbrot halmaz kirajzolása
    g.drawImage(kép, 0, 0, this);
    // Ha éppen fut a számítás, akkor egy vörös
    // vonallal jelöljük, hogy melyik sorban tart:
```

```
if(számításFut) {  
    g.setColor(java.awt.Color.RED);  
    g.drawLine(0, sor, getWidth(), sor);  
}  
// A jelző négyzet kirajzolása:  
g.setColor(java.awt.Color.WHITE);  
g.drawRect(x, y, mx, my);  
if (!zX.isEmpty())  
{  
    for (int i = 0; i<zX.size(); ++i){  
        g.drawLine(zX.get(i), zY.get(i), zX2.get(i), zY2.get(i));  
    }  
}
```

Az osztály paint függvény felüldefiniálása is azt a célt szolgálja, hogy az imént a pillanatfelvételnél kirajzolt vonalakat az ablakban is kirajzoljuk. Egyrészt ki kell rajzolni a számítás előrehaladását jelző vonalat, ha éppen fut a halmaz számítása, ehhez piros színt használunk. Amikor kijelöljük a területet az egérrel, akkor ki kell rajzolni a programnak egy fehér négyzetet. Ezt a drawRect () függvény hajtja végre. Végül abban az esetben, ha már legalább egy z_n számot kiszámoltunk, akkor az ezeket a pontokat összekötő egyeneseket is ábrázoljuk.

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

A main függvény pedig most is csak példányosít egy objektumot, jelen estében egy MandelbrotHalmazNagyító osztályt.

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás forrása: [itt](#)

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

A feladatban szereplő polártranszformáció segítségével random számokat tudunk kiszámolni. Maga az algoritmus olyannyira eltrejedt, hogy a Java random szám generátor függvénye is ezt használja.

Elsőnek vegyük végig a C++ forrást, majd rátérünk a java-ra, mely, látni fogod, sokkal letisztultabb.

```
class PolarGen {  
  
public:  
  
    PolarGen(); //konstruktor  
  
    ~PolarGen() {} //destruktur  
  
    double kovetkezo(); //random lekérés  
  
private:  
  
    bool nincsTarolt;  
    double tarolt; //random értéke  
  
};
```

Szükségünk van egy osztályra, melyben a random számokat fogjuk elkészíteni. Alapvetően ez 2 részből áll, a `public` és a `private` részből. A `public` részben szereplő elemek elérhetőek a class-on kívül, ezzel

szemben a private csak a class-on belül. A konstruktor egy olyan "függvény", ami akkor hajtódik végre amikor létre hozzuk a PolarGen típusú objektumunkat. Fontos, hogy csak egyszer hajtódik végre, és ezt, függetlenül attól, hogy a public részben van, már nem tudjuk meghívni. Hasonló igaz a destruktora, mely a program futása végén hajtódik végre. A nevöknek meg kell egyeznie a class nevével. Használata olyan esetekben nélkülözhetetlen a class-on belül foglaltunk tárteületet, mivel ebben tudjuk felszabadítani ezeket. A kovetkezo() függvény segítségével pedig a random számokat fogjuk kiszámolni.

```
PolarGen::PolarGen() { //a konstruktor kifejtése
    nincsTarolt = false;
    std::srand (std::time(NULL)); //random inicializálás
}
```

A konstruktor jelen esetben, ad egy alapértelmezett értéket a nincsTarolt változónak és meghívja az srand() függvényt, ami a random számokat fogja generálni.

```
double PolarGen::kovetkezo() { //random lekérő függvény kifejtése
    if (nincsTarolt)
    {
        double u1, u2, v1, v2, w;

        do{
            u1 = std::rand () / (RAND_MAX + 1.0); //innentől jön az algoritmus
            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; //idáig tart az algoritmus
    }

    else
    {
        nincsTarolt = !nincsTarolt; //ha van korábbi random érték, akkor azt ←
            adja vissza
        return tarolt;
    }
};
```

A kovetkezo() függvény pedig tartalmazza az algoritmust, mellyel most részletesen nem foglalkozunk. A lényeg annyi, hogy ellenőrizzük, hogy van-e tárolt random számunk, ha nincs, akkor generálunk kettőt, az egyiket visszadajuk, a másikat pedig eltároljuk.

```
int main()
{
```

```
PolarGen rnd;

for (int i = 0; i < 10; ++i) std::cout << rnd.kovetkezo() << std::endl; ←
    //10 random szám generálása

}
```

Végezetül a main-ben létrehozzuk a PolarGen típusú változónkat, és generálunk 10 random számot. Fentebb említést tettem a destruktorról, de, ahogy feltűnt, azt nem definiáltuk, ugyanis jelen esetben arra nincs szükség.

A java forrás nagyon hasonló az imént elemzett C++ forráshoz. A könyebb értelmezhetőség érdekében a változó és osztálynevek azonosak.

```
public class PolarGenerator
{
    boolean nincsTarolt = true;
    double tarolt;

    public PolarGenerator()
    {
        nincsTarolt = true;
    }

    public double kovetkezo()
    {
        if(nincsTarolt)
        {
            double u1, u2, v1, v2, w;
            do{
                u1 = Math.random();
                u2 = Math.random();
                v1 = 2* u1 -1;
                v2 = 2* u2 -1;
                w = v1*v1 + v2*v2;
            } while (w>1);

            double r = Math.sqrt((-2 * Math.log(w) / w));
            tarolt = r * v2;
            nincsTarolt = !nincsTarolt;
            return r * v1;
        }
        else
        {
            nincsTarolt = !nincsTarolt;
            return tarolt;
        }
    }

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

```
{  
    PolarGenerator g = new PolarGenerator();  
    for (int i = 0; i < 10; ++i)  
    {  
        System.out.println(g.kovetkezo());  
    }  
}
```

Láthatóan, maga a forrás is sokkal rövidebb. A Java-ban az egész forrás egy nagy class része. Ebben szerepel a main-is, de azt nem tekintjük a class részének. C++-ban tudja tömbösi íteni a private és public elemeket, itt mindegyik elő ki kell írni. Itt is van egy konstruktorunk, ami a nincsTárolt értékét igazzá definiálja. A következő függvény pedig a már megszokott módon generálja a random számokat. Egy érdekesség, hogy a Java-ban a random szám generálás függvény szintaxisa sokkal egyszerűbb, mely a Math library része.

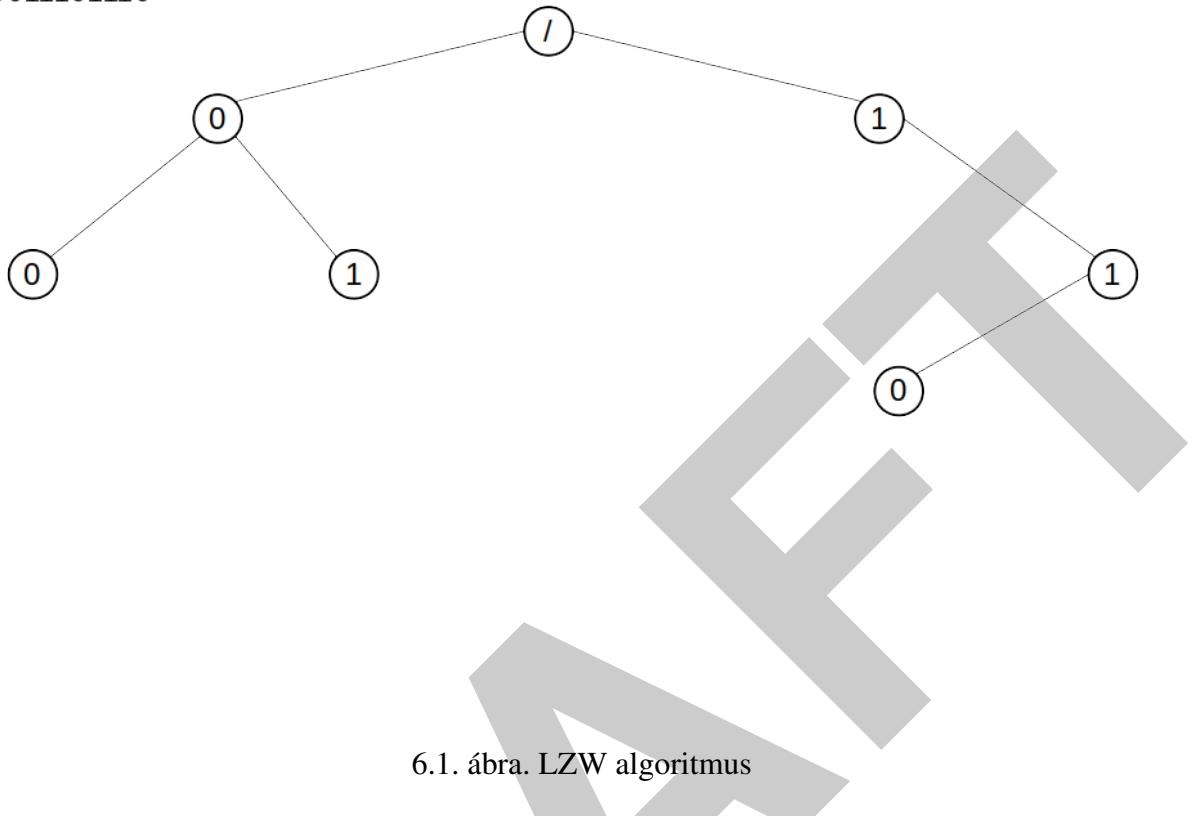
6.2. LZW

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

Megoldás forrása: [itt](#)

Az LZW algoritmus egy tömörítő eljárás, melyet többek között a gif formátuma, de sok tömörítő is, mint zip, gzip ezt használja. Az algoritmus lényege annyi, hogy a bemeneti 1-ekből és 0-ákból egy bináris fát épít. Mondjuk ezt még nem neveznénk tömörítésnek, tehát most jön a lényeg. Úgy építi fel a fát, hogy minden ellenőrzi, hogy van-e már 0-ás vagy 1-es gyermek, ha nincs akkor létrehoz egyet, és visszaugrik a gyökérre. Ha van, akkor a 0- és vagy 1-es gyermekre lép, és addig lépked lefelé a fában, ameddig nem talál egy olyan részfát, ahol létre kellene hozni egy új gyermeket, a létrehozás után visszaugrik a gyökérre.

00011101110



Az ábrán látható, hogy a 11 bites bemenetből, a végére csak 6 bitet tartottunk meg. Ezt az algoritmust használja a C programunk.

```
typedef struct binfa
{
    int ertek;
    struct binfa *bal_nulla;
    struct binfa *jobb_egy;

} BINFA, *BINFA_PTR;
```

Első lépésként létrehozunk egy struktúrát, amely 3 részből áll, egy értékből, és a gyermekire mutató mutatókból. Maga a struktúra segítségével egy új típust definiálunk, mely segítségével az összetartozó adatokat tudjuk együtt kezelni. A `typedef` segítségével pedig meg tudunk adni más nevet, amivel hivatkozhatunk a struktúrára.

```
BINFA_PTR
uj_elelem ()
{
    BINFA_PTR p;

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

```
    }
    return p;
}
```

Az `uj_elem()` függvény segítségével foglalunk helyet a BINFA típusú változóknak, majd visszaadunk egy erre a területre mutató pointert.

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

Deklaráljuk a szükséges függvényeket, melyeket a későbbiekben majd definiálunk, de most előtte jöjjön a `main`. Jelen esetben ez most elég hosszú, tehát szedjük e darabraira.

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

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

Elsőnek létrehozzuk a gyökeret. Az értékét beállítjuk a '/' jelre, ezzel fogjuk jelölni ebben a feladatban, és az előkötkezőkben a gyökeret. Mivel még nincs se bal, se jobb oldali gyermekek, ezért a mutatóknak NULL értéket adunk. A fa mutatót pedig a gyökérre állítjuk.

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

```
    fa->jobb_egy->bal nulla = fa->jobb_egy->jobb_egy = NULL;
    fa = gyoker;
}
else
{
    fa = fa->jobb_egy;
}
}
```

A while ciklusban alkotjuk meg a binfánkat. A standard inputról olvassuk a bemenetet, bitenként. Ha a bemenet 0, akkor ellenőrizzük, hogy van-e nullás gyermekek, ha nincs létrehozunk egyet, és a fa mutatót visszaállítjuk a gyökérre. Ellenkező esetben a fa mutatót a bal oldali gyermekre állítjuk. Ha a bemenet nem 0, akkor ellenőrizzük, hogy van-e jobb oldali gyermek, ha nincs, akkor létrehozunk, és a fa mutatót visszaállítjuk a gyökérre. Ha viszont van, akkor a fa mutatót a jobb oldali gyermekre mutat.

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

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

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

/* Átlagos ághossz kiszámítása */
atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
ratlag (gyoker);
// atlag = atlagosszeg / atlagdb;
// (int) / (int) "elromlik", ezért casoljuk
// K&R tudatlansági védelem miatt a sok () :)
atlag = ((double)atlagosszeg) / atlagdb;

/* Ághosszak szórásának kiszámítása */
atlagosszeg = 0;
melyseg = 0;
atlagdb = 0;
szorasosszeg = 0.0;

rszoras (gyoker);

double szoras = 0.0;

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

```
    szoras = sqrt (szorasosszeg);

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

    szabadit (gyoker);
}
```

A main függvény végén íratjuk ki a binfát, és számolunk ki hozzá néhány érdekes adatot. De koncentráljunk a kiir és a szabadit függvényekkel.

```
void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;
        kiir (elem->jobb_egy);
        // ez a postorder bejáráshoz képest
        // 1-el nagyobb mélység, ezért -1
        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ↔
                ,
                melyseg-1);
        kiir (elem->bal nulla);
        --melyseg;
    }
}
```

A kiir segítségével fogjuk a bináris fánkat kiírni a standard outputra. Ehhez most az inorder bejárást alkalmazzuk, ahol elsőnek feldolgozzuk a jobb oldali gyermeket, majd a gyökeret, és végül a bal oldali gyermeket .

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

A destrukturban meghívott rekurzív szabadító függvény pedig itt látható. A lefoglalt tártelületet a free () függénnyel tudjuk felszabadítani. De mielőtt felszabadítanánk az átadott elemet, előtte meghívjuk ugyan ezt a függvényt a gyerekeire is, legalábbis ha van.

A lényeg már elhangzott, most tegyük egy kis említést a ratlag és a rszoras függvényekről.

```
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 függvény egy pointert kap, ha a pointer nem NULL, akkor növeljük a melyseg nevű globális változónkat, majd meghívjuk a pointer által mutatott elem gyermekéire is a ratlag függvényt. Ha rekurzív hívások során elérkezünk az utolsó jobb és bal oldali gyermekhez, akkor az atlagdb változót növeljük 1-el, és az atlagosszeg-hez hozzáadjuk a melyseg értékét.

```
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));

        }
    }
}
```

}

Az `rszoras` függvény nagyon hasonlít az `ratalag`-hoz. Lényegi különbség az `if` utasításon belül van. Ebben a szórásösszeget adjuk meg, mely a mélység és az átlag különbségének a négyzete.

6.3. Fabejárás

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

Megoldás forrása: [itt](#)

Az előző feladatban taglalt program módosítását fogjuk ebben a részben elvégezni. Kezdjük azzal, hogy mit is jelent az inorder, posztorder, preorder bejárás. Az inorder bejárásnál elsőnek dolgozzik fel a bal oldali gyermeket, majd a gyökérelemet, és legvégül pedig a jobb oldali gyermeket. Ezt használta eredetileg a programunk. A posztorder bejárásnál elsőnek a bal oldali gyermeket, majd a jobb oldali gyermeket, végül pedig a gyökérelem kerül feldolgozásra. A preorder bejárás pedig a gyökérelemet dolgozza fel először, majd a bal és a jobb oldali gyermeket.

Most, hogy ismertettem veled a fabejárásokat, kezdhetjük is megírni a programot. Nézziük elsőnek a posztorder bejárást. Mivel a program lényegében ugyan az, ezért csak a `kiir` függvényt kell módosítani.

```
void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;
        kiir (elem->jobb_egy);
        // ez a postorder bejárásra képest
        // 1-el nagyobb mélység, ezért -1
        kiir (elem->bal nulla);
        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ←
                ,
                melyseg-1);
        --melyseg;
    }
}
```

A `for` ciklustól kezdődik az függvénynek átadott elem feldolgozása. Ha visszaemlékezel az előző feladatban tárgyalt inorder bejárásra, ott a `for` ciklus a két gyermek feldolgozása között szerepelt. Itt a postorder bejárásnak megfelelően az utolsó helyre kerül, előtte pedig a bal és a jobb oldali gyermeket dolgozza fel a program.

Természetesen nem siklunk el a preorder bejárás felett sem.

```
void
```

```
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;
        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ↔
                ,
                melyseg-1);
        kiir (elem->jobb_egy);
        // ez a postorder bejáráshoz képest
        // 1-el nagyobb mélység, ezért -1
        kiir (elem->bal nulla);
        --melyseg;
    }
}
```

Itt a `for` ciklus került legelőre, tehát a paraméterként átadott elemet dolgozzuk fel, és csak ezután a bal, majd a jobb oldali gyermeket.

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!

Megoldás forrása: [itt](#)

Megjegyzés

A feladat megoldásában tutoráltként részt vett: Halász Dávid

C-ben már megismertük az LZW algoritmust, most pedig át kell azt írni C++-ra. Maga az átalakítás nem nagyon bonyolult, de azért nézzük végig sorról sorra.

Az első újdonság az osztály lesz, mely lényegében a C forrásban megismert struktúrának a továbbgon-dolása. Az osztályban már nem csak változók egy csoportját tudjuk együtt kezelni, hanem írhatunk bele függvényeket is.

```
class LZWBinFa
{
public:

    LZWBinFa () :fa (&gyoker)
    {
```

```
    }
~LZWBinFa ()
{
    szabadít (gyoker.egyesGyermek ());
    szabadít (gyoker.nullasGyermek ());
}
```

Az osztályt, ahol a Polártranszformációtól, itt is a konstruktorkkal kezdünk. A konstruktur csakis annyit csinál, hogy a fa mutatót a gyökér elem memóriacímére állítjuk. A destrukturban pedig a C-ben már megírt szabadít függvényt hívjuk.

```
void operator<< (char b)
{
    // Mit kell betenni éppen, '0'-t?
    if (b == '0')
    {
        if (!fa->nullasGyermek ()) // ha nincs, hát akkor csinálunk
        {
            Csomopont *uj = new Csomopont ('0');
            fa->ujNullasGyermek (uj);
            fa = &gyoker;
        }
        else // ha van, arra rálépünk
        {
            fa = fa->nullasGyermek ();
        }
    }

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

A C++ egyik érdekességét láthatod itt. Ezt hívjuk operátor túlterhelésnek, melynek segítségével tudjuk módosítani, hogy egy operátor mit csináljon. Természetesen ezt nem úgy kell elkövetni, hogy az operátor működését teljesen meg tudod változtatni, csak arra biztosít lehetőséget, hogy a saját típusainkat tudja kezelni. Jelen esetben a `operator <<` segítségével a bemenetként kapott elemeket beleshifteljük a fába, az LZW algoritmusnak megfelelően. Ha nullát kap paraméterként, és nincs még nullás gyermek, akkor létrehozzuk. C++-ban ezt a `new`-val tudunk tárterületet lefoglalni. Tehát a létrehozás abból áll, hogy foglalunk területet az új csomópontnak, majd az `ujNullasGyermek ()` függvény segítségével bele fűzzük a fába,

és a fa mutató a gyökérre állítjuk. Ugyan ezt az eljárást hajtjuk végre 1 esetén is, természetesen ehhez is meg van az ujEgyesGyermek () függvény. Mind 0 és 1 esetén is ha már létezik az adott csomópontnak 0-ás vagy 1-es gyermek, akkor a fa mutatót az adott gyermekre állítjuk. Az egyes-/nullasGyermek () függvényt hívjuk ilyenkor, mely az adott csomópont nullás vagy egyes gyermekére mutatott mutatót ad vissza.

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

Ez is a C++ egy érdekessége, mely abból áll, hogy itt nem a `kiir` függvényt hívjuk kétszer, hanem a 2 `kiir` függvényünk van, és ezt a C++ képes kezelni, mivel a fordító meg tudja őket különböztetni a paraméterlista eltérése miatt. Tehát ez a `kiir` nem kap paramétert, csak meghívja a másik `kiir` függvényt, és a `melyseg` értékét 0-ra.

```
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);
}
```

A `get*` függvényekre azért lesz szükség, mert a mélységet, átlagot és szórást eltároló változóink az osztály privát részében találhatóak, tehát csak a függvényen belül tudjuk elérni. Ezek után a globális `operator<<-t` terheljük túl. Mellyel kimenetet adunk vissza, és kimenetet és egy `LZWBInFa` objektum referenciát adunk át. Meghívjuk az objektumhoz tartozó `kiir` függvényt, amelyet alatta definiáltunk. Ez abban különbözik az előzőtől, hogy itt megadjuk, hogy mi legyen a kimenet.

Az eddig tárgyalt részek mind az osztály `public` részében szerepeltek. Nézzük meg, hogy mi is van a `private` részben.

```
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 &); //másoló konstruktor  
    Csmopont & operator= (const Csmopont &);  
};
```

A Csomópont osztályt a az LZWBInFa osztályba integráltuk, annak is a private részébe, tehát Csmopont típusú objektumot nem tudunk létrehozni az LZWBInFa osztályon kívül. A konstruktőrben a Csmopont alapértelmezett értékét '/'-re állítjuk, a gyermeket pedig kinullázzuk. A destruktur jelein esetben nem csinál semmit. A nullasGyermek() és az egyesGyermek() pointert ad vissza a bal és a jobb oldali gyerekre. Az ujEgyesGyermek és ujNullasGyermek pedig a gyermeket mutatóját állítja a paraméterként átadott csomópontra. A getBetu() segítségével olvassuk be a bemenetet. A másoló konstruktor, másoló értékkopírozás letiltva, mivel helyes működésük nincs megoldva, ezért raktuk privát részbe.

```
Csmopont *fa;  
int melyseg, atlagosszeg, atlagdb; /* A fában tagként benne van egy ←  
    csomópont, ez erősen ki van tüntetve, ō a gyökér: */  
double szorasosszeg;  
LZWBInFa (const LZWBInFa &);
```

```
LZWBInFa & operator= (const LZWBInFa &);
```

A már említett fa mutatót itt deklaráljuk, tehát a fa mutató mindenkorra egy csomópontra mutat. Az utána lévő változók már ismertek a C kódiból. Viszont az utolsó két elem a másoló konstruktőr és másoló értékkopírozás. Ezek azért vannak privátban, mert a működésük nincs megoldva jelenleg, tehát letiltjuk, hogy a felhasználó ne tudja használni.

```
void kiir (Csomopont * elem, std::ostream & os)
{
    if (elem != NULL)
    {
        ++melyseg;
        kiir (elem->egyesGyermek (), os);
        for (int i = 0; i < melyseg; ++i)
            os << "---";
        os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
        kiir (elem->>nullasGyermek (), os);
        --melyseg;
    }
}
void szabadit (Csmopont * elem)
{
    if (elem != NULL)
    {
        szabadit (elem->egyesGyermek ());
        szabadit (elem->>nullasGyermek ());
        delete elem;
    }
}
```

Ebben a részben a már jól megismert `kiir` és a `szabadit` függvényt láthatjuk. Működése lényegében megegyezik a C forrásban leírtakkal. A fabejárás itt is inorder módon történik.

`protected:`

```
Csmopont gyoker;
int maxMelyseg;
double atlag, szoras;

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

};
```

Legvégül pedig lássuk mi is van az `LZWBInFa` osztály `protected` részében. Ez abban különbözik a private résztől, hogy olyan osztályok el tudják érni, amik az `LZWBInFa` osztályból vannak származtatva. Ebben a

részben találjuk a feladat címében szereplő gyoker változót, amely tagja az osztálynak. Itt vannak deklarálva továbbá a szórás, átlag és mélység kiszámításához szükséges függvényeket is. Ezzel az LZWBInFa osztály végére értünk, érezhetően sokkal hosszabb volt, mint a C forrás struktúrája, de sokkal több dolgot tudunk egy objektumként kezelni.

Akkor nézzük meg, hogy mi is történik az LZWBInFa class-on kívül.

```
int
LZWBInFa::getMelyseg (void)
{
    melyseg = maxMelyseg = 0;
    rmelyseg (&gyoker);
    return maxMelyseg - 1;
}

double
LZWBInFa::getAtlag (void)
{
    melyseg = atlagosszeg = atlagdb = 0;
    ratlag (&gyoker);
    atlag = ((double) atlagosszeg) / atlagdb;
    return atlag;
}

double
LZWBInFa::getSzoras (void)
{
    atlag = getAtlag ();
    szorasosszeg = 0.0;
    melyseg = atlagdb = 0;

    rszoras (&gyoker);

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

    return szoras;
}

void
LZWBInFa::rmelyseg (Csomopont * elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > maxMelyseg)
            maxMelyseg = melyseg;
        rmelyseg (elem->egyesGyermek ());
        // ez a postorder bejáráshez képest
```

```
// 1-el nagyobb mélység, ezért -1
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));
        }
    }
}
```

Az osztályon belül deklarált függvényeket az osztályon kívül definiáljuk, ezzel leegyszerűsítve a már így is bonyolult osztályunkat. Az osztályon belüli függvényeket a `LZWBinFa::_nev_` előtaggal tudjuk elérni. A `get*` függvények azt a célt szolgálják, hogy a private részben lévő elemeket elérjük, a többi pedig a C-ben megismert értékeket számolja ki.

```
void
usage (void)
{
```

```
    std::cout << "Usage: lzwtree in_file -o out_file" << std::endl;
}
```

Ha a felhasználó nem a megfelelő módon futtatja a programot, akkor hibaüzenetet dobunk. Majd következik a main.

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

A main elején ellenőrizzük, hogy a felhasználó elég argumentumot adott-e meg, ha nem, akkor a usage függvény segítségével kiírjuk a program helyes használatát, végül pedig hibával térünk vissza, azaz a program futása megáll.

```
char *inFile = *++argv; //argv[1]

if (*((++argv) + 1) != 'o') //argv[2][1]
{
    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); //argv[3]

unsigned char b;
```

LZWBinFa binFa;

Ha megbizonyosodtunk rólunk, hogy a felhasználó elég argumentumot adott meg, akkor bemeneti fájlra ráálítunk egy pointert. Ellenőrizzük, hogy az argumentum tömb harmadik elemének a második eleme 'o'-e, ha nem, akkor újra kiírjuk a felhasználónak a helyes működést, és jelezük az operációs rendszernek, hogy a program futása során hiba lépett fel. Ezután az fstream osztályú beFile-t definiáljuk, az első argumentum a bemenetre mutató pointer, míg a másodikkal jelezük, hogy beolvassuk az első argumentum tartalmát. Ha olyan bemeneti fájlt adunk meg, ami nem létezik, akkor szintén kiírjuk a program helyes használatát, és hibával térünk vissza. Ezt követően definiáljuk a kiFile fstream osztályú objektumot, első paraméterként átadjuk az argumentum tömb 4. elemét, és ebbe fogja kiírni a program a bináris fát. A b változóba fogjuk beolvasni bájtonként a bemenetet, és létrehozzuk az LZWBinFa objektumunkat.

```
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)
    {      // > karakter
        kommentben = true;
        continue;
    }

    if (b == 0x0a)
    {      // újsor
        kommentben = false;
        continue;
    }

    if (kommentben)
        continue;

    if (b == 0x4e)      // N betű
        continue;

for (int i = 0; i < 8; ++i)
{

    if (b & 0x80)
        binFa << '1';
    else
        // különben meg a '0' betűt:
        binFa << '0';
    b <<= 1;
}

}
```

Az első while ciklus segítségével elkezdjük beolvasni a bemenetet a b változóba, és ha találunk sortörést, akkor megállítjuk a ciklust. A kommentben változó értékét hamisra állítjuk, majd a második while ciklusban elkezdjük feldolgozni a bemenetet, és felépíteni a bináris fát. Ha karaktert olvasunk be, akkor a kommentben változó értékét igazzá tesszük, ha pedig sortörést, akkor pedig hamissá. Ha a beolvasott karakter nagy betű, akkor folytatjuk a ciklust, ez nem okoz problémát. A for ciklusban a beolvasott karakter bitjein végig fut a program egyesével, ha a b bitenként éselve egyet kapunk, akkor a fába belenyomjuk az 1-es étéket, ha nem, akkor 0-át, végezetül pedig minden iteráció végén shifteljük a b értékét 1-el.

```
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;
}
```

Az utolsó lépés pedig az, hogy a kimenetként fájlba beleshifteljük a bináris fát, és a hozzá tartozó mélység, áltla és szórás értékeket.

6.5. Mutató a gyökér

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

Megoldás forrása: [itt](#)

Az előző feladatban részletesen taglaltuk, hogy hogyan is épül fel a C++-os LZW algoritmust alkalmazó program, most ezt kell módosítani, mivel az előzőben a `gyoker` tagja volt az osztálynak. Ebben a feladatban pointerre fogjuk átírni, ami nem is annyira nehéz feladat, csak néhány dologra oda kell figyelni.

Első lépésként csak át kell írni a `gyoker`-t mutatóra, melyet a 316. sorban találunk.

```
Csomopont *gyoker;
```

Ha most fordítjuk a forrás, akkor rengeteg hibát fog találni, de ez nem baj, folytatni kell az átalakítást. A következő módosítást a konstruktörben kell végrahajtani.

```
LZWBinFa ()
{
    gyoker = new Csomopont ('/');
    fa = gyoker;
}
```

A `gyoker` mutatókat ráállítjuk egy újonan lefoglalt tárterületre, és a `fa` mutatót is erre állítjuk. Mivel foglaltunk területet, ezért azt fel is kell szabadítani, tehát a destruktur a következőképpen alakul:

```
~LZWBinFa ()
{
    szabadit (gyoker->egyesGyermekek ());
    szabadit (gyoker->>nullasGyermekek ());
    delete (gyoker);
}
```

Több módosítás van, egyszerűen a gyökér mostmár mutató, tehát a gyökér által mutatott csomópont nullás és egyes gyermekére kell meghívunk a `szabadit()` függvényt. Szemben az előző feladatban megismert forrással, ahol ezek a gyokerhez tartoztak, azaz ott írtuk:

```
    szabadít (gyoker.egyesGyermek ());
```

A gyoker mutató által mutatott területet pedig a `delete()` függvényel szabadítjuk fel. Ha most fordítjuk a programot, akkor már kevesebb hibűt kapunk, és megtudjuk, hogy pontosan hol vannak még hibák. Ezenken a helyeken annyi a feladat, hogy már nem a gyökér memóriacímét kell átadni, hanem csak a gyökeret, mint mutatót. Tehát mindenhol törölni kell az & címképző operátort a gyökér elől.

6.6. Mozgató szemantika

Írj az előző programhoz mozgató konstruktort és értékadást, a mozgató konstruktor legyen a mozgató értékadásra alapozva!

Megoldás forrása: [itt](#)



Megjegyzés

A feladat megoldásában tutoráltként részt vett Országh Levente és Stigi Máté.

A feladatban ahhoz a programhoz nyúlunk vissza, amely esetén a gyökét tag volt. Ezen elvégezve az apróbb módosításokat, már készen is van a mozgatókonstruktor.

```
LZWBinFa (LZWBinFa&& forras) //mozgató konstruktor
{
    std::cout<<"Move ctor\n";
    gyoker = nullptr;
    *this = std::move(forras); //ezzel kényszerítjük ki, hogy a mozgató ←
                                értékadást használja

}
LZWBinFa& operator= (LZWBinFa&& forras) //mozgató értékadás
{
    std::cout<<"Move assignment ctor\n";
    std::swap(gyoker, forras.gyoker);
    return *this;
}
```

Ehhez meg kell csinálnunk a mozgató konstruktort és a mozgató értékadást. Mind a kettőnek hasonló a szintaktikája, ami annyi tesz, hogy a paraméterként átadott fa gyökerének az elemeit átadjuk az üres fának, majd a mozgatott fa elemeit kinullázuk.

```
LZWBinFa binFa2 = std::move(binFa);

kiFile << binFa2;
kiFile << "depth = " << binFa2.getMelyseg () << std::endl;
kiFile << "mean = " << binFa2.getAtlag () << std::endl;
kiFile << "var = " << binFa2.getSzoras () << std::endl;
```

Végezetül a move függvényel jobbértékké tesszük az átadott argumentumot, mellyel meghívjuk a mozgató értékadást, és végül kiírjuk az új fát. Fontos látni, hogy ha ezután az eredeti binFa-t is ki akarnánk nyomni, akkor nem tudnánk, mert azt kinulláztuk.

Ha már tudjuk, hogy hogyan kell megcsinálni a mozgató konstruktort úgy, hogy a gyoker kompozícióban van, lássuk ennek a feladatnak a megoldását mutatóval. Észrevehetően sokkal egyszerűbb lesz.

```
LZWBinFa (LZWBinFa&& forras)
{
    std::cout<<"Move ctor\n";
    gyoker = nullptr;
    *this = std::move(forras); //ezzel kényszerítjük ki, hogy a mozgató ←
        értékadást használja

}
LZWBinFa& operator= (LZWBinFa&& forras)
{
    std::cout<<"Move assignment ctor\n";
    std::swap(gyoker, forras.gyoker);
    return *this;
}
```

Ebben már kifejezetten a feladatban előírt a mozgató konstruktort a mozgató értékadásra alapozós analógiát követjük. Tehát a mozgató konstruktorban meghívjuk a mozgató értékadást. Azt már említettem feljebb, hogy a mozgatást úgy oldjuk meg, hogy a fát átmásoljuk, és az eredetit pedig kinullázzuk. Ezt egy kicsit most más megközelítéssel használjuk fel. A fának, ahova az eredeti fánkat akarjuk mozgatni, a gyökér mutatóját kinullázzuk, majd pedig meghívjuk a mozgató értékadást, úgy hogy az argumentumként átadott fát egyenlővé tesszük a cél fával, természetesen jobbértékké alakítva. A jobbértékké alakítást oldja meg a move () függvény. A mozgató értékadásban pedig megcseréljük a cél fa gyökerének és a forrás fa gyökerének az értékét, melyben a swap van segítségünkre. Ezzel megoldjuk a forrás fa kinullázását egy lépésben. Ha csak ennyit módosítasz az előző feladatban tárgyalt programon, akkor memóriakezelési problémákba fogsz ütközni. Mégpedig a destruktur tartalma miatt.

```
~LZWBinFa ()
{
    szabadít (gyoker->egyesGyermekek ());
    szabadít (gyoker->>nullasGyermekek ());
    delete (gyoker);

}
```

Jelenleg ez destrukturunk, de ez hibás. Gondoljunk arra, hogy a forrásunk gyökerét kinulláztuk, és ezután meg szeretnénk hívni a szabadít függvényt annak a gyökérnek az egyes és nullás gyermekére, aki nem is mutat már semmire. Tehát ezeket az utasításokat csak akkor szabad végrehajtani, ha a gyökér mutató nincs kinullázva, így a konstruktur a következőképpen módosul:

```
~LZWBinFa ()
{
    szabadít (gyoker);
```

{}

Hiszen a szabadít () függvény csak akkor kezdi el felszabadítani az egyes és nullás gyermeket, ha az átadott pointer nem null pointer.

DRAFT

7. fejezet

Helló, Conway!

7.1. Hangyszimulációk

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

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

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...



Használat előtt

`sudo apt-get install qtbase5-dev`

A program lényegében a hanygák feromonokkal történő kommunikációját szimulálja. A képernyőt cellákra osztjuk, és a cellákban lévő hangyák megkeresik azt a szomszédjukat, akinek a legerősebb a feromonja, és arra lép tovább. A cellák fermonon értékei folyamatosan csökkennek, viszont ha valamelyik hangya bele lép az egyikbe, akkor ott megnő a fermomon szint. Ezeket az értékeket minden parancssori argumentumok formájában tudjuk megadni.

A `main.cpp`-ben megtalálható futtatási javaslat a következő:

```
./myrmecologist -w 250 -m 150 -n 400 -t 10 -p 5 -f 80 -d 0 ←  
-a 255 -i 3 -s 3 -c 22
```

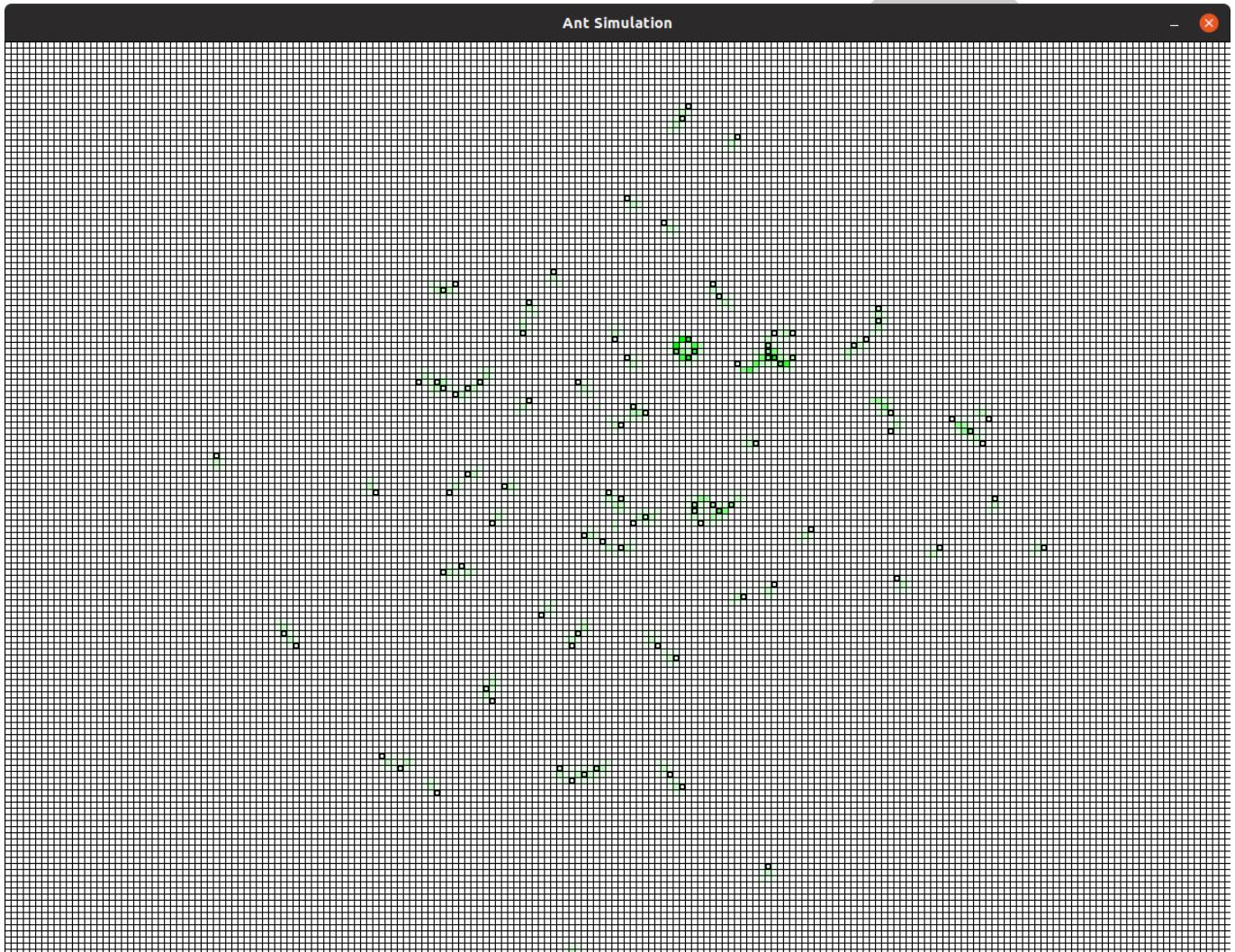
A `w` kapcsolóval állítjuk be, hogy egy cella hány oszlobból álljon, vagyis, milyen széles legyen, a `m`-el pedig a cellák magasságát. Az `n` kapcsoló a hangyák számát állítja be, a `t`-vel pedig a lépések gyakoriságát állítjuk, melyet milliszekundumba adunk meg. A `p`-vel a feromonok párolgásának a gyorsaságát állítjuk be, az `f`-el pedig azt, hogyha egy hangya belelép egy cellába, akkor mennyivel növeli meg annak a feromonértékét. A `d` kapcsolóval meg tudjuk adni, hogy mi legyen a cellák feromonértékének a kezdőértéke. Az `a` és a `i` kapcsolókkal meg tudjuk adni a maximális és a minimális fermonoértékeket. Az `s` kapcsoló azt állítja, hogy a hangyák mennyi feromont hagyjanak a szomszédos cellákban. Végezetül pedig lehetőségünk van beállítani azt is, hogy az egyes cellákban maximum hány hangya lehessen egyszerre. Természetesen ezeket nem muszáj megadni, mivel vannak a programban alapértelmezett értékek a futtatáshoz.

Futtatás

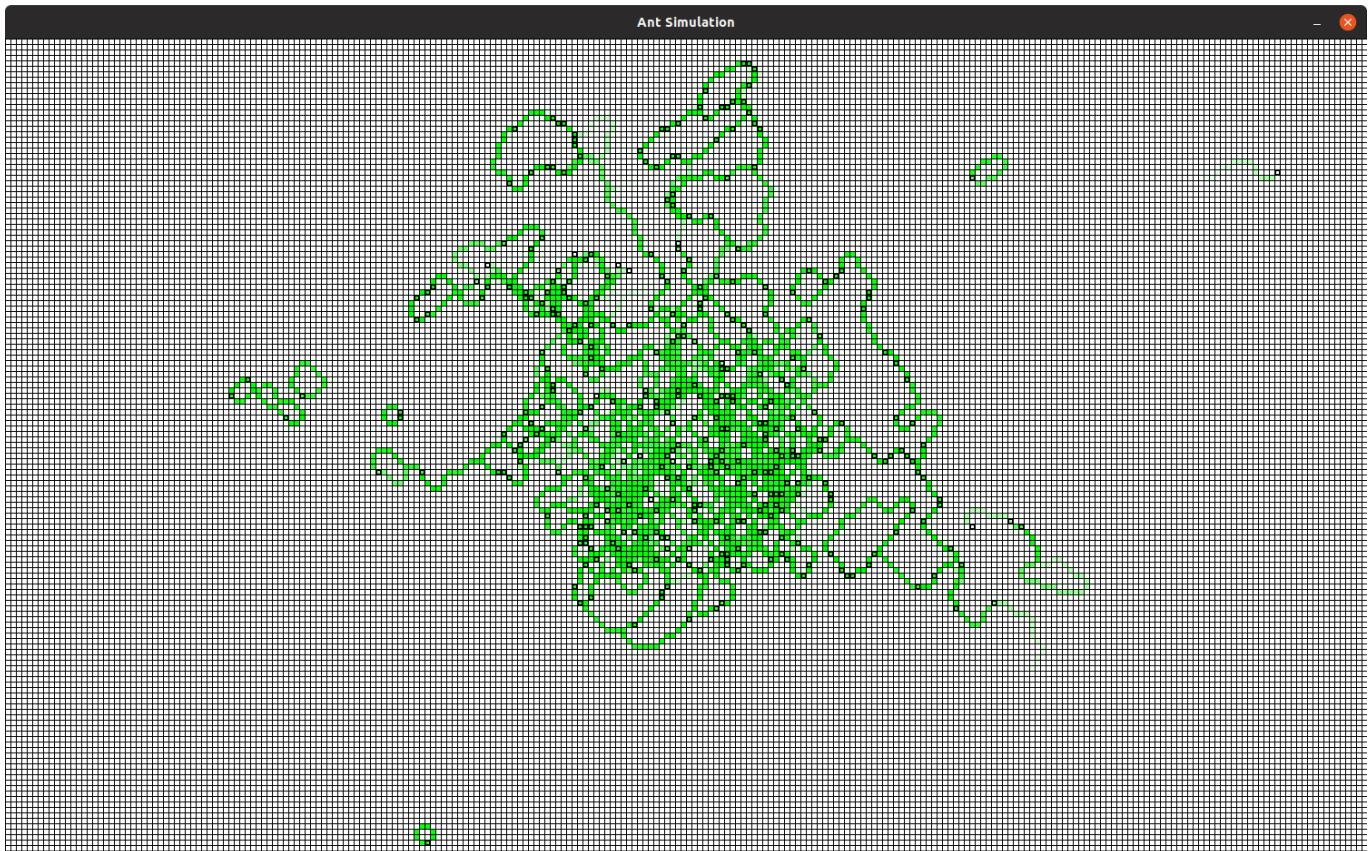
Mindegyik fájlnak egy mappában kell lennie. Majd:



```
qmake myrmecologist.pro  
make  
../myrmecologist //+ a kapcsolók, ha szeretnéd ←  
állítani
```

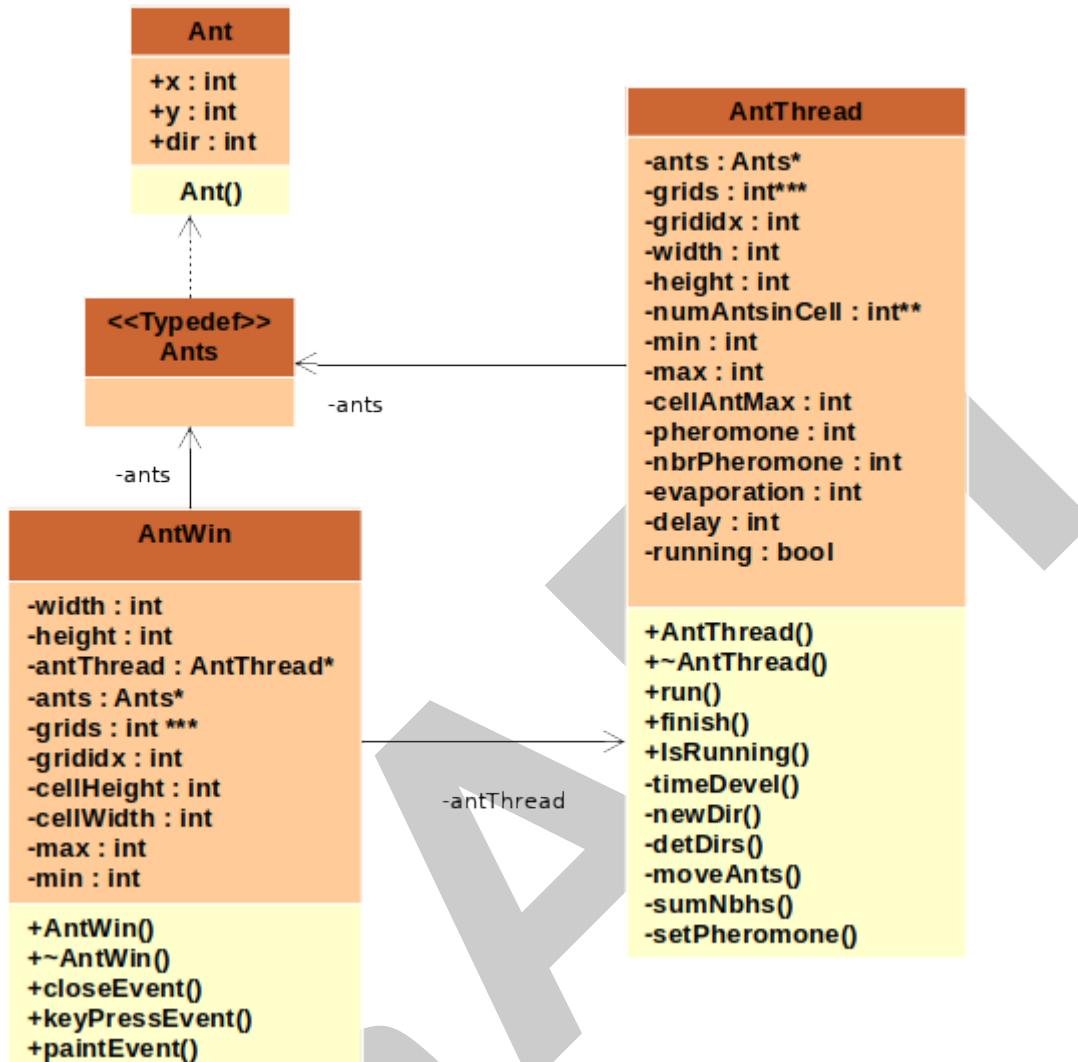


7.1. ábra. Kapcsolók nélkül



7.2. ábra. Kapcsolókkal

Most, hogy láttuk, hogyan is működik a program, lássuk magát a forrást. Mivel most több fájlból áll a forrás, ezért egy kicsit forradalmibb módon próbálom neked demonstrálni a program alapvető felépítését. Ezt nevezzük UML osztálydiagramnak. Segítségével könnyebben átláthatod a bonyolult programokat.



7.3. ábra. UML diagram

Az ábrán láthatjuk a programunk egyes osztályait. Azon elemek, melyek előtt a "-" jel szerepel, nem érhetők el az osztályon kívül, míg a "+" jelek igen. Tehát ezekkel különböztetjük meg az osztályok public és private tagjait. A legkisebb osztály az `Ant`, melyet a `ant.h` header tartalmazza.

```

class Ant
{
public:
    int x;
    int y;
    int dir;

    Ant(int x, int y) : x(x), y(y) {
        dir = qrand() % 8;
    }
}

```

```
    }

};

typedef std::vector<Ant> Ants;
```

Ebben csak a hangyák pozícióját határozzuk meg az x és az y koordináták segítségével, plusz az irányát is itt számoljuk ki, melyet a dir változóban tárolunk. Az utolsó sor az ábrán már külön búbrikban szerepel. A **typedef** segítségével az Ant típusú elemkeből álló vektorokra az Ants alias-szal tudunk hivatkozni. Tehát:

```
Ants elso;
```

Ebben a kódcsipetben tehát egy Ant típusú elemekből álló vektort deklaráltunk.

A következő osztályunk az AntWin, melyet az antwin.h és az antwin.cpp-ben találunk. A header fájlban vannak az egyes elemek deklarációi, míg a C++ forrás tartalmazza azok definícióját. Ez viszonylag gyakori eljárás a C++ programozásban, mivel így letisztultabb kódot kapunk.

```
#include <QMainWindow>
#include <QPainter>
#include <QString>
#include <QCloseEvent>
#include "antthread.h"
#include "ant.h"

class AntWin : public QMainWindow
{
    Q_OBJECT

public:
    AntWin(int width = 100, int height = 75,
            int delay = 120, int numAnts = 100,
            int pheromone = 10, int nbhPheromon = 3,
            int evaporation = 2, int cellDef = 1,
            int min = 2, int max = 50,
            int cellAntMax = 4, QWidget *parent = 0);

    AntThread* antThread;

    void closeEvent ( QCloseEvent *event ) {

        antThread->finish();
        antThread->wait();
        event->accept();
    }

    void keyPressEvent ( QKeyEvent *event )
    {

        if ( event->key() == Qt::Key_P ) {
```

```
    antThread->pause();
} else if ( event->key() == Qt::Key_Q
            || event->key() == Qt::Key_Escape ) {
    close();
}

}

virtual ~AntWin();
void paintEvent (QPaintEvent*);

private:

int ***grids;
int **grid;
int gridIdx;
int cellWidth;
int cellHeight;
int width;
int height;
int max;
int min;
Ants* ants;

public slots :
void step ( const int &);

};
```

A Qt egyes funkcióinak a használatához nélkülözhetetlen a fenti header-ök csatolása. A QMainWindow osztályal tudjuk megalkotni a programunk ablakját, amiben a hangyaszimuláció kirajzolódik. Kirajzolódik, ezért van szükség a QPainter osztályra. A QString osztály segítségével tudunk Unicode karakterkódulású sztringeket tárolni, a QCloseEvent segítségével pedig a program bezárását tudjuk szabályozni paraméterek segítségével. Ez a header fájl kapcsolja össze az Ant és a AntThread osztályokat. Összeségében elmondható, hogy ebben az osztályban történik meg a programablaknak a létrehozása, itt állítjuk be az egyes paramétereket, amiket parancssori argumentumok segítségével a programnak át tudunk adni. A program futásának szüneteltetéséért és bezárásáért is ez az osztály felelős, itt vannak meghatározva az egyes billentyűkhöz rendelt tevékenységek.

Maga az érdemi számítást pedig az antthread.h és az antthread.cpp fájlok tartalmazzák. Az előzőhez hasonlóan itt is csak a headert nézzük meg:

```
#include <QThread>
#include "ant.h"

class AntThread : public QThread
{
    Q_OBJECT

public:
    AntThread(Ants * ants, int ***grids, int width, int height,
```

```
        int delay, int numAnts, int pheromone, int nbrPheromone,
        int evaporation, int min, int max, int cellAntMax);

~AntThread();

void run();
void finish()
{
    running = false;
}

void pause()
{
    paused = !paused;
}

bool isRunning()
{
    return running;
}

private:
    bool running {true};
    bool paused {false};
    Ants* ants;
    int** numAntsinCells;
    int min, max;
    int cellAntMax;
    int pheromone;
    int evaporation;
    int nbrPheromone;
    int ***grids;
    int width;
    int height;
    int gridIdx;
    int delay;

    void timeDevel();

    int newDir(int sor, int oszlop, int vsor, int voszlop);
    void detDirs(int irany, int& ifrom, int& ito, int& jfrom, int& jto );
    int moveAnts(int **grid, int row, int col, int& retrow, int& retcol, ←
                 int);
    double sumNbhs(int **grid, int row, int col, int);
    void setPheromone(int **grid, int row, int col);

signals:
    void step ( const int &);

};
```

A Qthread osztályra azért van szükség, hogy kezelni tudjuk a program egyes szálait. Az AntThread osztély konstruktora megkapja azokat az értékeket, amiket az AntWin osztályban megkapott a program, majd elkezdi "mozgatni" a hangyákat. Kiszámolja az irányukat a feromonszintek és a cellatelítettség alapján, állítja az egyes cellák feromonértékeit, és megadja az AntWin osztálynak, hogy mely cellákat kell átszínezni.

7.2. Java életjáték

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

Megoldás forrása: [itt](#)

Megjegyzés

A feladat megoldásában tutorként részt vett Petrus József Tamás. Tutoráltként pedig Szabó Benedek.

Használat

```
sudo apt-get install openjdk-8-jdk
javac game_of_life.java
java game_of_life
```

Tanulságok, tapasztalatok, magyarázat...

Melvin Conway 1970-ben alkotta meg saját sejtautomatáját. A sejtautomaták lényege, hogy megadunk néhány feltételt az életben maradásra, a születésre, és a halálozásra, majd figyeljük, hogy az általunk "teremtett" lények, hogyan "élnek". Conway 3 egyszerű szabályt adott meg, és mind a mai napig ez a legnépszerűbb sejtautomata.

A szabályok a következők:

1. szabály: Egy sejt csak akkor marad életbe, ha 2 vagy 3 szomszédja van.
2. szabály: Ha egy sejtnak 3-nál több szomszédja van, túlnépesedés miatt meghal. Ha pedig 2-nél kevesebb a szomszédai száma, akkor pedig az elszigetelődés miatt hal meg.
3. szabály: Egy sejt megszületik, ha az üres cellának pontosan 3 élő sejt található a szomszédos celláiban.

Ezen szabályok megadásával elképesztő animációkat kapunk, de mi most kifejezetten egyre, a siklókilövőre koncentrálunk. Ehhez fix cellákba helyezzük a kezdeti sejteket, és utána folymatosan indulnak el felfelé a siklók.

```
public class game_of_life extends JFrame {
    RenderArea ra;
    private int i;
```

```
public game_of_life() {
    super("Game of Life");
    this.setSize(1005, 1030);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    this.setVisible(true);
    this.setResizable(false);
    ra = new RenderArea();
    ra.setFocusable(true);
    ra.grabFocus();
    add(ra);
    int[][] siklokilovo = {{6,0},{6,1},
                           {7,0},{7,1},
                           {3,13},
                           {4,12},{4,14},
                           {5,11},{5,15},{5,16},{5,25},
                           {6,11},{6,15},{6,16},{6,22},{6,23},{6,24},{6,25},
                           {7,11},{7,15},{7,16},{7,21},{7,22},{7,23},{7,24},
                           {8,12},{8,14},{8,21},{8,24},{8,34},{8,35},
                           {9,13},{9,21},{9,22},{9,23},{9,24},{9,34},{9,35},
                           {10,22},{10,23},{10,24},{10,25},
                           {11,25}};
    int min_o = 5;
    int min_s = 85;
```

A game_of_life osztálynak a JFrame a szülőosztálya. A JFrame osztályra azért van szükség, hogy az ablak, amiben a programunk fut, megjelenjen. Deklarálunk egy RenderArea objektumot, mellyel a hálót és a sejteket fogjuk kirajzolni. A game_of_life() konstruktörben pont ezeket állítjuk be. A super() függvénytel meghívjuk a szülő osztály konstruktőrét, ezután be tudjuk személyre tudjuk szabni a programablakot. Megadjuk az ablak nevét, a méretét. Beállítjuk azt, hogy hogyan záródjon be a program alapértelmezetten, és az átméretezhetőséget kikapcsoljuk, mivel fix abalkmérettel dolgozunk. Az ra objektumot hozzáadjuk az ablakhoz, így tudjuk kirajzolni a sejtautomatát. Ezután pedig megalkotjuk a siklókilövőhöz szükséges fix pontok koordinátáit. Végül pedig meghatározzuk, hol legyen az origó, ahonnan a koordinátákat kirajzoljuk.

```
for (int i = 0; i < siklokilovo.length; ++i)
{
    ra.entities.get(min_o + siklokilovo[i][1]).set(min_s+ siklokilovo ←
        [i][0],!ra.entities.get(min_o + siklokilovo[i][1]).get((min_s+ ←
        siklokilovo[i][0])));
    this.update(this.getGraphics());
}

ra.edit_mode = false;
ra.running = true;
```

A for ciklusban az ra objektum entities listájának átadjuk a kiinduló sejtek pozícióját. Majd meghívjuk az update függvényt, amely frissíti az ablak tartalmát. Ez a függvény egy tagfüggvénye a JComponent osztálynak. Azért tudjuk elérni, mert a RenderArea osztályunknak az őse a JPanel osztály, aminek az őse a JComponent. Abban különbözik a miénktől, hogy paraméterül kér egy Graphics osztályú

objektumot. Ilyen objektumot a `getGraphics()` függvénytel tudunk készíteni az ablakunkból. Az `edit_mode`-ra azért van szükség, mert a program képes lenne egyedi ábrák használatára is, de ez most nem része a feladatnak. A `running` pedig jelzi, hogy a program fut.

```
public void update() {
    ArrayList<ArrayList<Boolean>> entities = new ArrayList<ArrayList<Boolean>>(); // = ra.entities;
    int size1 = ra.entities.size();
    int size2 = ra.entities.get(0).size();
    for(int i=0;i<size1;i++)
    {
        entities.add( new ArrayList<Boolean>());
        for(int j=0;j<size2;j++)
        {
            int alive = 0;

            if(ra.entities.get((size1+i-1)%size1).get((size2+j-1)%size2) == true) alive++;
            if(ra.entities.get((size1+i-1)%size1).get((size2+j)%size2) == true) alive++;
            if(ra.entities.get((size1+i-1)%size1).get((size2+j+1)%size2) == true) alive++;

            if(ra.entities.get((size1+i)%size1).get((size2+j-1)%size2) == true) alive++;
            if(ra.entities.get((size1+i)%size1).get((size2+j+1)%size2) == true) alive++;

            if(ra.entities.get((size1+i+1)%size1).get((size2+j-1)%size2) == true) alive++;
            if(ra.entities.get((size1+i+1)%size1).get((size2+j)%size2) == true) alive++;
            if(ra.entities.get((size1+i+1)%size1).get((size2+j+1)%size2) == true) alive++;

            /*for(int k=-1;k<2;k++)
            {
                for(int l = -1; l < 2 ;l++)
                {
                    if(! (k==0 && l == 0))
                    {
                        if(ra.entities.get((size1+i+k)%size1).get((size2+j+l)%size2) == true) alive++;
                    }
                }
            }*/
        }
    }
}
```

```
{  
    if(alive < 2 || alive > 3)  
    {  
        //ra.entities.get(i).set(j,false);  
        entities.get(i).add(false);  
    }  
    else  
    {  
        entities.get(i).add(true);  
    }  
}  
else  
{  
    if(alive == 3)  
    {  
        //ra.entities.get(i).set(j,true);  
        entities.get(i).add(true);  
    }  
    else  
    {  
        entities.get(i).add(false);  
    }  
}  
}  
}  
ra.entities = entities;  
}  
}
```

Az update() függvény felelős a Conway által meghatározott szabályok implementálásáért. minden sejt-hez kiszámoljuk a szomszédjai számát, majd a szabályok alapján eldöntjük, hogy életben marad-e, meghal, vagy születik.

```
class RenderArea extends JPanel implements KeyListener {  
    public ArrayList<ArrayList<Boolean>> entities;  
  
    public int diff;  
    public boolean edit_mode;  
    public boolean running;  
    public RenderArea() {  
        super();  
        setSize(1000, 1000);  
        setVisible(true);  
        setBackground(Color.WHITE);  
        setForeground(Color.BLACK);  
        setLocation(0, 0);  
  
        diff = 10;  
    }  
}
```

```
this.addKeyListener(this);
entities = new ArrayList<ArrayList<Boolean>>();
for(int i=0;i<1000/diff;i++)
{
    entities.add(new ArrayList<Boolean>());
    for(int j=0;j<1000/diff;j++)
    {
        entities.get(i).add(false);
    }
}

@Override
public void keyTyped(KeyEvent e) {
    //System.out.println(e.getKeyChar());
}

@Override
public void keyReleased(KeyEvent e) {
    System.out.println("Key pressed:"+e.getKeyChar());
    if(e.getKeyChar()=='e')
    {
        edit_mode = !edit_mode;
    }
    else if(e.getKeyChar()=='q')
    {
        this.running = false;
    }
    else if(e.getKeyChar()=='c')
    {
        if(edit_mode)
        {
            for(int i=0;i<this.entities.size();i++)
            {
                for(int j=0;j<this.entities.get(1).size();j++)
                {
                    this.entities.get(i).set(j,false);
                }
            }
            this.update(this.getGraphics());
        }
    }
}
```

```
}

@Override
public void keyPressed(KeyEvent e) {
    //System.out.println(e.getKeyChar());
}

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    for(int i=0;i<1000;i+=diff)
    {
        g.drawLine(i, 0, i, 1000);
    }
    for(int j=0;j<1000;j+=diff)
    {
        g.drawLine(0, j, 1000, j);
    }
    for(int i=0;i<1000;i+=diff)
    {
        for(int j=0;j<1000;j+=diff)
        {
            if(entities.get(i/diff).get(j/diff))
            {
                g.setColor(Color.BLACK);
            }
            else
            {
                g.setColor(Color.WHITE);
            }

            g.fillRect(i+2, j+2, diff-3, diff-3);
        }
    }
}

private static final long serialVersionUID = 1L;

}
```

A RenderArea osztály a JPanel osztály gyermeke, és implementálja a KeyListener interfést. Az interfész és az osztály között annyi a különbség, hogy az interfészben nincsenek definiálva a tagfüggvények. Azokat nekünk kell definálni a saját osztályunkon belül. Az osztály konstruktorában meghívjuk a JPanel konstruktorát. Eközött és a JFrame között annyi a különbség, hogy a JFrame osztály a fő ablakot alakítja ki, a JPanel pedig ebbe az ablakra illeszkedik bele. Beállítjuk a panel szélességét, amely a fő ablakkal egyenlő, a setSize() segítségével. A setVisible(true) láthatóvá teszi a panelt. Beállítjuk a az

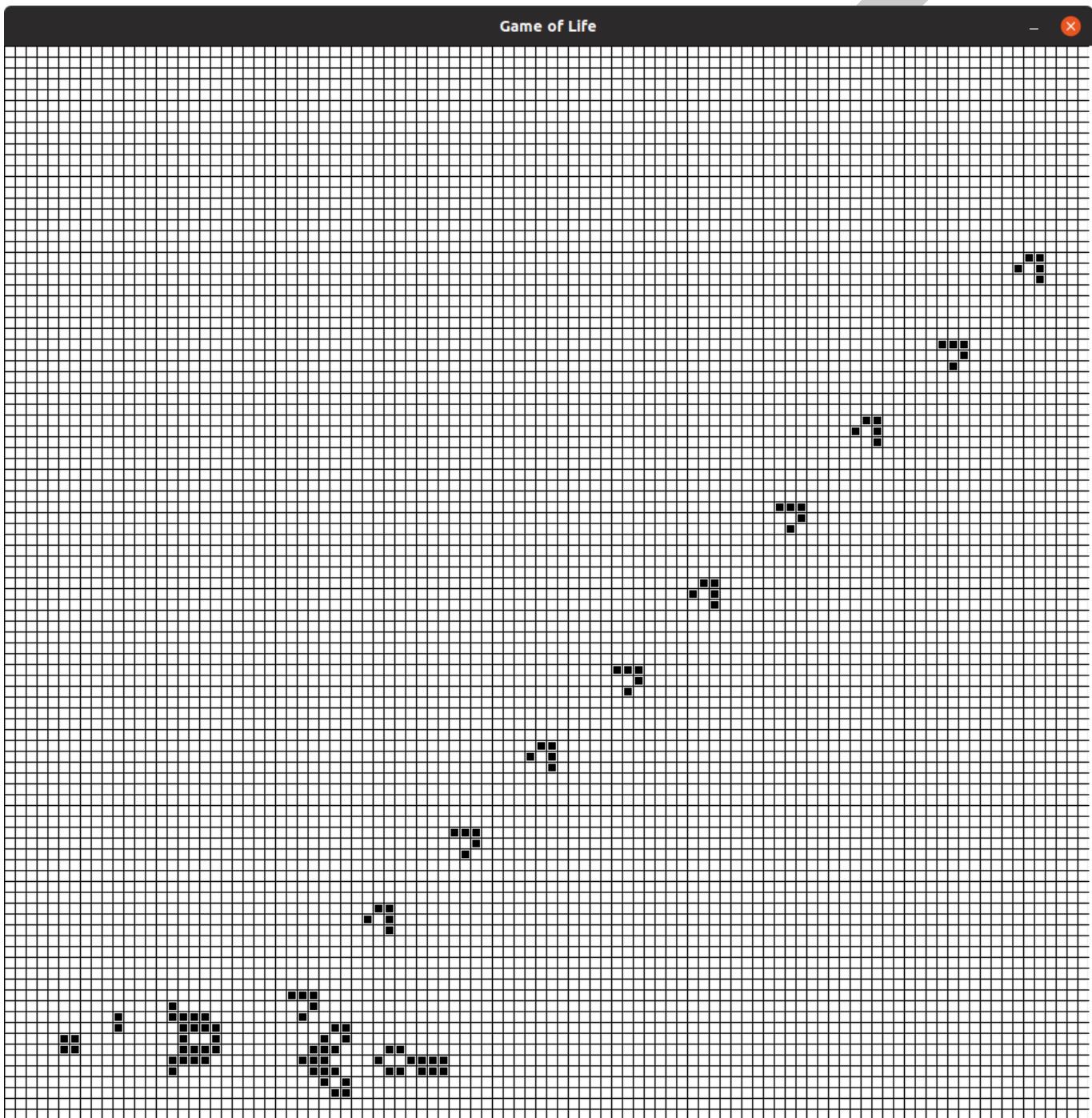
előtér és a háttér színét feketére és fehérre. A `diff` a kirajzolandó háló sűrűségét határozza meg. Hozzáadjuk a panelhez a billentyűzet kezelését, és a kétdimenziós listánkban minden elem logikai értékét hamisra állítjuk. Tehát alap esetben minden sejt halott. A `@Override` címkével azt jelöljük, hogy a függvény nem az osztályban van deklarálva, hiszen a `KeyListener` deklarálja azokat. Mivel implementáltuk ezt az interfész, ezért minden tagfüggvényét kötelezően definiálnunk kel, ezért láthatod, hogy néhány függvény nem csinál semmit. Mi azt kezeljük, amikor a felhasználó felengedi az egyik billentyűt, erre szolgál a `KeyReleased` függvény. Ezek közül a "q" billentyű kezelése lényeges, mivel a program jelenleg nem képes iteraktívításra. Ha képes lenne, akkor az "e" billentyűvel lépnénk ki a szerkesztő módból. A "c" pedig a kijelölt négyzeteket tisztította volna, ha előről szeretnénk kezdeni a szerkesztést. A "q" billentyűvel ki tudunk lépni a programból. A `paintComponent` szintén egy felüldefiniált függvény, ez a `JPanel` osztály tagfüggvénye. A függvény elsőnek a `super.paintComponent` függvényt hívja meg. Ez újra rajzolja a panelt úgy, mintha mi nem módosítottunk volna a tagfüggvényen. Ez azért hasznos, mert így nem kell nekünk az egészet kirajzolni, csak azt a részt, amit módosítani szeretnénk. Tehát ha mi az 1000x1000-es panelból csak egy 500x500-as rész újrarendezésére szeretnénk használni a `paintComponent`-et, akkor a maradék részt az alapértelmezett `paintComponent` kirajzolja nekünk. Mi ezt a függvényt a rácsháló kirajzolására használjuk, plusz a sejteket is ezzel ábrázoljuk. A hálót simán a `drawLine()` függvénytel ábrázoljuk. A sejteket viszont úgy, hogy nem a teljes négyzetet töltjük ki, hanem adunk neki egy fehér keretet.

```
if(entities.get(i/diff).get(j/diff))  
{  
    g.setColor(Color.BLACK);  
  
}  
else  
{  
    g.setColor(Color.WHITE);  
}  
  
g.fillRect(i+2, j+2, diff-3, diff-3);
```

Végezetül a main-ben példányosítunk egy `game_of_life` objektumot, majd addig jelenítjük meg az újabb állapotokat, ameddig a felhasználó le nem nyomja a "q" billentyűt, vagy be nem zárja a programot.

```
public static void main(String args[])  
{  
    game_of_life gol = new game_of_life();  
    while(gol.ra.running)  
    {  
        if(!gol.ra.edit_mode)gol.update();  
        try{Thread.sleep(120);}  
        catch(Exception ex)  
        {  
  
        }  
        gol.ra.update(gol.ra.getGraphics());  
    }  
    gol.dispose();  
}
```

A `while` cikluson belül hívjuk meg a saját `update()` függvényünket, mellyel újraszámoljuk a sejtek állapotát, és pozícióját. Majd a szálat altatjuk 120 miliszekundumig. Ez az utasítás azért került egy kivételekkel kezelő blokkba, mert enélkül a Java fordító hibát dobott. A ciklus utolsó utasítása, hogy meghívjuk a másik `update()` függvényt, melynek segítségével frissítjük az ablak tartalmát. A program futásának végeztével, pedig a `dipose()` függvényrel bezárjuk az ablakot. Ez akkor hívódik meg, amikor a "q" gomb lenyomásával a `running` változó hamisra változik, vagyis kilépünk a `while` ciklusból.



7.4. ábra. Siklókilövő

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás forrása: [itt](#)

Használat



```
sudo apt-get install libqt4-dev
qmake-qt4 -project
qmake-qt4 *.pro
make
./QT (vagy a bináris neve, ahogy létrejött)
```

Tanulságok, tapasztalatok, magyarázat...

Az előző feladatban taglalt Conway-féle életjátékot fogjuk most megcsinálni a Qt grafikus felületet használva C++ nyelven. A szabályok továbbra is adottak. A sejtek meghalnak, ha 2-nél kevesebb a szomszédjuk vagy 3-nál több szomszédja van. Megszületik, ha pontosan 3 szomszédja van, és ha még abban a cellában nem él. Végezetül, ha 2-nél több, de 3-nál kevesebb a szomszédok száma, akkor a életben marad a sejt. Ebben a feladatban is a siklókilövőt kell megvalósítani, de ehhez már tudjuk a koordinátákat, tehát ez nem lesz gond. A program nem képes önálló alakzatok megvalósítására, még a forrása sem tartalmazza az ehhez szükséges kódokat.

A program forrása 4 fájlból áll, erre azért van szükség, hogy általáthatóbbak legyenek az egyes osztályok definíciója.

```
#include <QtGui/QMainWindow>
#include <QPainter>
#include "sejtszal.h"

class SejtSzal;

class SejtAblak : public QMainWindow
{
    Q_OBJECT

public:
    SejtAblak(int szelesseg = 100, int magassag = 75, QWidget *parent = 0);
    ~SejtAblak();
    // Egy sejt lehet élő
    static const bool ELO = true;
    // vagy halott
    static const bool HALOTT = false;
    void vissza(int racsIndex);

protected:
    bool ***racsok;
    bool **racs;
```

```
int racsIndex;
// Pixelben egy cella adatai.
int cellaSzelesseg;
int cellaMagassag;
// A sejttér nagysága, azaz hányszor hány cella van?
int szelesseg;
int magassag;
void paintEvent (QPaintEvent* );
void siklo(bool **racs, int x, int y);
void sikloKilovo(bool **racs, int x, int y);

private:
SejtSzal* eletjatek;

};
```

A `sejtablak.h` fájlban előre deklaráljuk a `SejtSzal` osztályt, erre azért van szükség, hogy használni tudjuk az elemeit a `SejtAblak` osztályban. A `SejtAblak` osztály a `QMainWindow` osztály leszármazottja, ezért elérheti annak a tagfüggvényeit. A `Q_OBJECT` makróval tesszük lehetővé, hogy a fordító kezelni tudja a Qt-s kiegészítéseket. A `public` részben található a konstruktur és a destrukturációja. Itt definiáljuk a sejteket állapotát jelző változókat is, és a vissza függvényt. Ezek az osztályon kívül is elérhetőek. A `protected` rész az, amelyet a leszármazott osztályok is elérhetnek. Két logikia tipusra mutató mutatóink van, `racsokra` azért van szükség, mert 2 rácsot használunk, egyik tartalmazza az aktuális állapotot, a másik pedig az egygel későbböt. Ezek valamelyikére mutata a `racs` mutató. A `racsIndex` azt jelöli, hogy melyik rácsot használjuk éppen. A cellák méretét a `cellaSzelesseg` és `cellaMagassag` változók tárolják. A `paintEvent` egy `QMainWindow`-beli tagfüggvény, mellyel az aktuális rácsot rajzoljuk ki. A `siklokilovo` és a `siklo` függvények azt adják meg, hogy melyik pontjai legyenek élő sejtek a rácsnak, ahhoz, hogy a program kirajzoljon egy siklót, vagy egy siklókilövőt. A `private` részben pedig egy `SejtSzal` objektumra mutató pointert deklarálunk. A `sejtablak.cpp` fájl tartalmazza az osztály tagfüggvényinek a definiálását. Elsőnek lássuk a konstruktort:

```
SejtAblak::SejtAblak(int szelesseg, int magassag, QWidget *parent)
: QMainWindow(parent)
{
    setWindowTitle("A John Horton Conway-féle életjáték");

    this->magassag = magassag;
    this->szelesseg = szelesseg;

    cellaSzelesseg = 6;
    cellaMagassag = 6;

    setFixedSize(QSize(szelesseg*cellaSzelesseg, magassag*cellaMagassag));

    racsok = new bool**[2];
    racsok[0] = new bool*[magassag];
    for(int i=0; i<magassag; ++i)
        racsok[0][i] = new bool [szelesseg];
    racsok[1] = new bool*[magassag];
    for(int i=0; i<magassag; ++i)
```

```
    racsok[1][i] = new bool [szelesseg];  
  
    racsIndex = 0;  
    racs = racsok[racsIndex];  
    // A kiinduló racs minden cellája HALOTT  
    for(int i=0; i<magassag; ++i)  
        for(int j=0; j<szelesseg; ++j)  
            racs[i][j] = HALOTT;  
    // A kiinduló racsra "ELOlényeket" helyezünk  
    //siklo(racs, 2, 2);  
    sikloKilovo(racs, 5, 60);  
  
    eletjatek = new SejtSzal(racsok, szelesseg, magassag, 120, this);  
    eletjatek->start();  
  
}
```

A konstruktur egy magasságot, egy szélességet, és egy QWidget objektumra mutató pointert kér paramétreként. A QWidget osztály adja az alapját a felhasználói felületet létrehozó objektumoknak. Érdekesség, hogy meghívja a QMainWindow osztály konstruktorát is, aminek paraméterül átadja a pointert. Ennek köszönhetően be tudjuk állítani az ablak nevét a setWindowTitle függvénnel. A this az aktuális objektumra mutató pointer. Ezért ennek segítségével adjuk meg, hogy minék a változóját szeretnénk definiálni. Ha nem használnánk, akkor nem tudna a fordító különbséget tenni az azonos nevű változók között. A cellaSzelesseg és cellaMagassag változóknál erre nincs szükség, azok értékét 6-ra állítjuk. A setFixedSize függvénnyel beállítjuk az ablak méretét. Ehhez QSize osztályt használjuk, ami egy kétdimenziós vektornak a méretét adja vissza. Ezután létrehozzuk a két rácsot, a racsIndexet nullára állítjuk, és a racs mutatót ráállítjuk a index által meghatározott rácusra. Ezt a rácsot feltöltjük csupa halott sejttel. Meghívjuk a sikloKilovo függvényt és példányosítunk egy Sejtszal objektumot, és elindítjuk a programszálat a start() függvényel. Ezt azért lehetjük meg, mert a Sejtszal osztály leszármazottja a QThread osztálynak.

```
void SejtAblak::paintEvent(QPaintEvent*) {  
    QPainter qpainter(this);  
  
    // Az aktuális  
    bool **racs = racsok[racsIndex];  
  
    // racsot rajzoljuk ki:  
    for(int i=0; i<magassag; ++i) { // végig lépked a sorokon  
        for(int j=0; j<szelesseg; ++j) { // s az oszlopok  
            // Sejt cella kirajzolása  
            if(racs[i][j] == ELO)  
                qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,  
                                   cellaSzelesseg, cellaMagassag, Qt::black) ↔  
;  
            else  
                qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,  
                                   cellaSzelesseg, cellaMagassag, Qt::white) ↔  
;
```

```
    qpainter.setPen(QPen(Qt::gray, 1));

    qpainter.drawRect(j*cellaSzelesseg, i*cellaMagassag,
                      cellaSzelesseg, cellaMagassag);
}
}

qpainter.end();
}
```

A paintEvent függvény felelős azért, hogy a sejteket kirajzoljuk az ablakba. Elsőnek példányosítunk egy QPainter objektumot, ez az osztály felelős azért, hogy rajzolni tudjunk. A racsIndex függvényében eldöntjük, hogy melyik rácsban rajzolunk. Elsőnek egy négyzetet kitölünk fehérrrel, ha halott, és feketével, ha élő az ott lévő sejt. Ehhez a fillRect függvényt használjuk. Majd a setPen függvénytel a átállítjuk a rajzoló eszköz színét szürkére, és 1 pixeles vastagságra. Ezeket az értékeket a QPen osztály segítségével adjuk át a függvénynek. Végül szürke színnel körbe rajzoljuk a négyzetet.

```
SejtAblak::~SejtAblak()
{
    delete eletjatek;

    for(int i=0; i<magassag; ++i) {
        delete[] racsok[0][i];
        delete[] racsok[1][i];
    }

    delete[] racsok[0];
    delete[] racsok[1];
    delete[] racsok;
}

}
```

A destruktur segítségével felszabadítjuk a pointerek által mutatott tárterületet, vagyis a rácsokat és a Sejtszal objektumot, amire az eletjatek mutat.

```
void SejtAblak::vissza(int racsIndex)
{
    this->racsIndex = racsIndex;
    update();
}
```

A vissza függvény segítségével állítjuk át a racsIndex változót. Majd frissítjük az ablak tartalmát.

```
void SejtAblak::sikloKilovo(bool **racs, int x, int y) {

    racs[y+ 6][x+ 0] = ELO;
    racs[y+ 6][x+ 1] = ELO;
    racs[y+ 7][x+ 0] = ELO;
    racs[y+ 7][x+ 1] = ELO;
```

```
racs[y+ 3][x+ 13] = ELO;  
  
racs[y+ 4][x+ 12] = ELO;  
racs[y+ 4][x+ 14] = ELO;  
  
racs[y+ 5][x+ 11] = ELO;  
racs[y+ 5][x+ 15] = ELO;  
racs[y+ 5][x+ 16] = ELO;  
racs[y+ 5][x+ 25] = ELO;  
  
racs[y+ 6][x+ 11] = ELO;  
racs[y+ 6][x+ 15] = ELO;  
racs[y+ 6][x+ 16] = ELO;  
racs[y+ 6][x+ 22] = ELO;  
racs[y+ 6][x+ 23] = ELO;  
racs[y+ 6][x+ 24] = ELO;  
racs[y+ 6][x+ 25] = ELO;  
  
racs[y+ 7][x+ 11] = ELO;  
racs[y+ 7][x+ 15] = ELO;  
racs[y+ 7][x+ 16] = ELO;  
racs[y+ 7][x+ 21] = ELO;  
racs[y+ 7][x+ 22] = ELO;  
racs[y+ 7][x+ 23] = ELO;  
racs[y+ 7][x+ 24] = ELO;  
  
racs[y+ 8][x+ 12] = ELO;  
racs[y+ 8][x+ 14] = ELO;  
racs[y+ 8][x+ 21] = ELO;  
racs[y+ 8][x+ 24] = ELO;  
racs[y+ 8][x+ 34] = ELO;  
racs[y+ 8][x+ 35] = ELO;  
  
racs[y+ 9][x+ 13] = ELO;  
racs[y+ 9][x+ 21] = ELO;  
racs[y+ 9][x+ 22] = ELO;  
racs[y+ 9][x+ 23] = ELO;  
racs[y+ 9][x+ 24] = ELO;  
racs[y+ 9][x+ 34] = ELO;  
racs[y+ 9][x+ 35] = ELO;  
  
racs[y+ 10][x+ 22] = ELO;  
racs[y+ 10][x+ 23] = ELO;  
racs[y+ 10][x+ 24] = ELO;  
racs[y+ 10][x+ 25] = ELO;  
  
racs[y+ 11][x+ 25] = ELO;  
  
}
```

A siklokilovo függvény pedig előre állítja `racs` mutató által mutaott rács megfelelő tagjait.

```
#include <QThread>
#include "sejtablak.h"

class SejtAblak;

class SejtSzal : public QThread
{
    Q_OBJECT

public:
    SejtSzal(bool ***racsok, int szelesseg, int magassag,
              int varakozas, SejtAblak *sejtAblak);
    ~SejtSzal();
    void run();

protected:
    bool ***racsok;
    int szelesseg, magassag;
    // Megmutatja melyik rács az aktuális: [rácsIndex][][][]
    int racsIndex;
    // A sejttér két egymást követő t_n és t_n+1 diszkrét időpíllanata
    // közötti valós idő.
    int varakozas;
    void idoFejlodes();
    int szomszedokSzama(bool **racs,
                         int sor, int oszlop, bool allapot);
    SejtAblak* sejtAblak;

};
```

A `sejtszal.h` headerben definiáljuk a `SejtSzal` osztályt, de a `SejtAblak` osztályt is előre definíáljuk. A `SejtSzal` osztály a `QThread` osztály gyermeke, mely segítségével programszálakat kezelhetünk. Itt is használjuk a `Q_OBJECT` makrót. A `protected` részben deklarálunk egy változót a rácsok kezelésére, a rács szélességét és magasságát is eltároljuk ebben az osztályban. A `racssIndex`-re szintén szükség van. Hogy ne legyen túl gyors a sejtautomata, ezért beállítunk egy bizonyos várakozási időt, ezt az értéket a `vvarakozas` változóban tároljuk. Az `idoFejlodes` függvény segítségével implementáljuk a Conway-féle szabályokat, és az alapján eldöntjük a sejt állapotát. Hogy megkapjuk az adott sejt szomszéda-inak számát, deklaráljuk a `szomszedokSzama` függvényt. Végül pedig deklarálunk egy `SejtAblak` objektumra mutató pointert.

```
SejtSzal::SejtSzal(bool ***racsok, int szelesseg, int magassag, int ←
    varakozas, SejtAblak *sejtAblak)
{
    this->racsok = racsok;
    this->szelesseg = szelesseg;
    this->magassag = magassag;
    this->varakozas = varakozas;
    this->sejtAblak = sejtAblak;
```

```
    racsIndex = 0;  
}
```

A Sejtszal osztály konstruktora paramétereként kéri a kér rácsot, azok szélességét, és magasságát, valamint a várkozási időt és egy pointert egy SejtAblak objektumra. A paraméterként kapott értékeket átadjuk az osztály tagváltozóinak, és a racsIndex változót nullázzuk.

```
int Sejtszal::szomszedokSzama(bool **racs,  
                                int sor, int oszlop, bool allapot) {  
    int allapotuSzomszed = 0;  
    // A nyolcszomszédok végigzongorázása:  
    for(int i=-1; i<2; ++i)  
        for(int j=-1; j<2; ++j)  
            // A vizsgált sejtet magát kihagyva:  
            if(!((i==0) && (j==0))) {  
                // A sejttérből szélénk szomszédai  
                // a szembe oldalakon ("periódikus határfeltétel")  
                int o = oszlop + j;  
                if(o < 0)  
                    o = szelesseg-1;  
                else if(o >= szelesseg)  
                    o = 0;  
  
                int s = sor + i;  
                if(s < 0)  
                    s = magassag-1;  
                else if(s >= magassag)  
                    s = 0;  
  
                if(racs[s][o] == allapot)  
                    ++allapotuSzomszed;  
            }  
  
    return allapotuSzomszed;  
}
```

A szomszedokSzama függvény egy adott rács adott pontjának szomszádait vizsgálja. Paraméterként egy logikai változót is kap, ezzel tudjuk meghatározni, hogy élő vagy halott szomszédokat keresünk. Egy sejtnak nyolc szomszédja lehet, tehát ezeken megyünk végig. Magát a sejet nem vizsgáljuk, ezért van az if feltétel. Egyéb esetben, ha az o változó értéke kisebb, mint 0, akkor átugrunk az utolsó oszlopra. Ha az o nagyobb, mint nulla, akkor pedig az első oszlopra ugunk. Ugyanezt a megoldást alkalmazzuk a sorokra is. Majd megvizsgáljuk, hogy a rács s. sorában és o. oszlopában milyen állapotú sejt található. Ha ez megegyezik a keresett allapot-tal, akkor növeljük a allapotuSzomszed változó értékét 1-el. A függvény például ennek a változónak a végső értékével fog visszatérni.

```
void Sejtszal::idoFejlodes() {  
  
    bool **racsElotte = racsok[racsIndex];  
    bool **racsUtana = racsok[(racsIndex+1)%2];
```

```
for(int i=0; i<magassag; ++i) { // sorok
    for(int j=0; j<szelesség; ++j) { // oszlopok

        int elok = szomszedokSzama(racsElotte, i, j, SejtAblak::ELO);

        if(racsElotte[i][j] == SejtAblak::ELO) {
            /* Elő élő marad, ha kettő vagy három élő
            szomszedja van, különben halott lesz. */
            if(elok==2 || elok==3)
                racsUtana[i][j] = SejtAblak::ELO;
            else
                racsUtana[i][j] = SejtAblak::HALOTT;
        } else {
            /* Halott halott marad, ha három élő
            szomszedja van, akkor élő lesz. */
            if(elok==3)
                racsUtana[i][j] = SejtAblak::ELO;
            else
                racsUtana[i][j] = SejtAblak::HALOTT;
        }
    }
}
racsIndex = (racsIndex+1)%2;
}
```

Magát a konkrét szabályokat az `idoFejlodes` eljárás tartatja be, itt dől el, hogy egy sejt milyen állapotba kerül. Elsőnek meghatározzuk, hogy melyik rácsból váltunk melyikbe. Majd ezalapján a `racsUtana` rácsba betöltyük a sejtállapotokat. Ahhoz, hogy ezt megtegyük, tudnunk kell, hogy az egyes sejteknek a `racsElotte` rácsban milyen szomszédai voltak. Ehhez meghívjuk az imént taglalt `szomszedokSzama` függvényt. A szabályokat a Java-s feladatban már tisztáztuk, szóval azt most nem részletezzük. A végén a `racsindexet` átállítja az éppen aktuálisan frissített rácsra.

```
void SejtSzal::run()
{
    while(true) {
        QThread::msleep(varakozas);
        idoFejlodes();
        sejtAblak->vissza(racsIndex);
    }
}
```

A `run()` függvényt felüldefiniáljuk. Egy végtelen ciklust hozunk létre. Ebben elsőnek altatjuk az éppen futó programszálat, a `varakozas` változóban megadott ideig, majd meghívjuk a `idoFejlodes` függvényt. Végezetül pedig a `sejtAblak` objektum `vissza` függvényének átadjuk az éppen frissített rács indexét, hogy az megjenelítse az ablakban az aktuális állapotot.



7.5. ábra. Siklókilövő

7.4. BrainB Benchmark

Megoldás forrása: [itt](#)



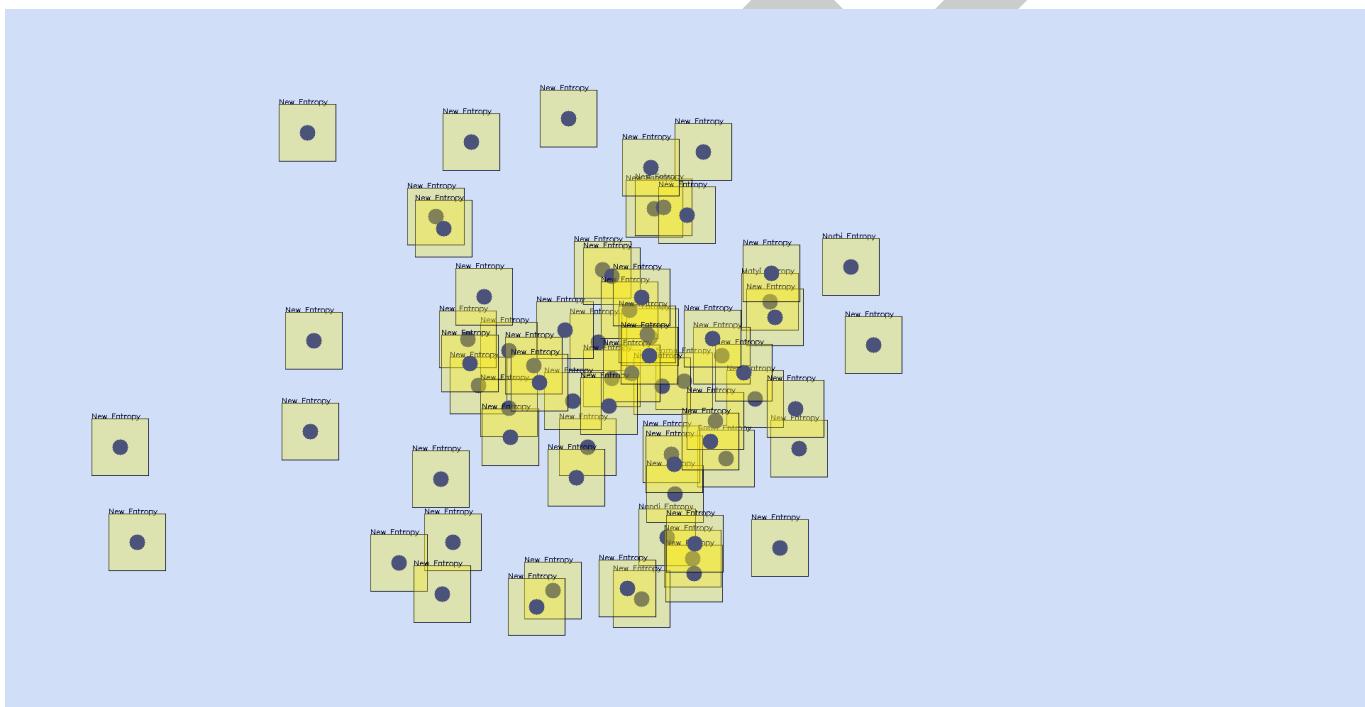
Passzolva

Használata előtt

```
sudo apt-get install libqt4-dev  
sudo apt-get install opencv-data  
sudo apt-get install libopencv-dev
```

Tanulságok, tapasztalatok, magyarázat...

A BrainB Benchmark feladata az esport tehetségek felkutatása lenne, úgy, hogy feltérképezi az agy kognitív képességeit, és az elért pontszámok alapján össze lehet hasonlítani az egyes egyéneket. Maga a benchmark a "karakterelvesztést" teszteli, vagyis ha a játékban elveszítjük a karakterünket, mennyi ideig tart megtalálnunk, és ha megtaláltuk, mennyi ideig tart elveszítenünk. Ideális esetben rövidebb ideig tart megtalálnunk, mint elveszteni. A program azt is figyeli, hogy az egyes karakterelvesztésekhez milyen bit/sec képernyő-váltások tartoznak.



7.6. ábra. Program

A program így néz ki, a feladat annyi, hogy a kurzort rajta kell tartani a Samu Entropy-n, minnél tovább tartod rajta, annál több New Entropy jelenik meg, ezzel nehezítve a dolgod. Ha elveszted, akkor a szaporodás lelassul, hogy könyebben megtalálhasd a karakteredet. A benchmark összesen 10 percig tart, és minél komplikáltabb lesz a kép a végén, annál jobb vagy.

```
Megnyitás ▾ [+]
Test-6000-stats.txt [Csak olvasható]
~/esport-talent-search/NEMESPOR BrainB Test 6.0.3 - Szó apr. 6 20191554569708327
Mentés ⌂ - X
NEMESPOR BrainB Test 6.0.3
time      : 6000
bps       : 62840
noc       : 70
nop       : 0
lost      :
18260 18870 33730 64770 31910 36900 20420 89350 52660 66800 66460 70760 57690 44510 40420 43460 24750
46680 53930 65550 78360 68520 69790 61330 35600 27820 96340 73260 76840 60130 23780 47210 55930
mean     : 52205
var      : 20879.4
found    : 2880 15780 20910 28390 26020 6760 19520 22440 31580 29580 41820 66700 35050 62790 48390 38470
24230 38370 18150 17190 23020 25660 28120 16620 36410 29840 34530 54400 39400 31140 33890 69750 58730
66480 83580 42980 67440 23650 23640 43250 48930 60280 68420 54140 60270 45280 54190 52450 79520 51080
49780 29980 28560 36430 57140 44500 45890 59950 62280 62620 61700 72900 71830 70310 21510 26530 46340
35580 43490 56480 51460 61970 64390 65110 71180 69420 75540 80320 56040 70140 68390 74020 70910 37530
33210 39850 43940 56470 69090 77770 46710 17580 15380 43650 54670 38440 50360 63610
mean     : 46459
var      : 19183.1
lost2found: 6760 31580 38470 18150 39400 42980 23650 29980 44500 61700 71830 21510 69420 37530 15380
43650 54670
mean     : 38303
var      : 18601.2
found2lost: 33730 64770 31910 89350 66460 44510 46680 65550 78360 69790 61330 96340 76840 23780 47210
55930
mean     : 59533
var      : 20624.7
mean(lost2found) < mean(found2lost)
time      : 10:0
U_R about 5.97144 Kilobvtes
```

Egyszerű szöveg ▾ Tabulátorszélesség: 8 ▾ 1. sor, 1. oszlop ▾ BESZ

7.7. ábra. Eredmény

A végén pedig megkapod ezt a fájlt, ami tartalmazza az elért eredményt. Hogy te is le tudd futatni, fontos, hogy minden fájl egy mappában legyen, majd pedig a következő parancsokat kell kiadni: qmake BrainB.pro, make, ./BrainB. Ha nem szeretnéd az egész benchmarkot végig várni, Esc-el ki tudsz lépni bármikor, és az addig elérte eredményedről is kapsz pontos leírást.

A main.cpp nézzük át elsőnek.

```
#include <QApplication>
#include <QTextStream>
#include <QtWidgets>
#include "BrainBWin.h"

int main ( int argc, char **argv )
{
    QApplication app ( argc, argv );

    QTextStream qout ( stdout );
    qout.setCodec ( "UTF-8" );
```

Mivel a program a Qt grafikus felületét használja, ezért a hozzá tartozó osztályokat is szükséges hozzáfűzni a forráshoz. Természetesen a BrainBWin.h-ra is szükségünk van, mert ezzel biztosítjuk, hogy a main-ból el tudjuk éerni a többi fájlt. A main a már megszokott formában köszön vissza, viszont első lépésként deklarálunk egy QApplication típusú objektumot, melynek paramétereként adjuk át a pa-

rancssori argumentumok számát, és tömbüket. A qout segítségével pedig a standard kimenetre írunk. Ez is egy objektum, amely QTextStream típusú. A qout objektumnak része a setCodec függvény, mellyel a karakterkódolást tudjuk beállítani.

```
qout << "\n" << BrainBWin::appName << QString::fromUtf8 ( " ↵
Copyright (C) 2017, 2018 Norbert Bátfai" ) << endl;
```

A std::cout-hoz hasonlóan, a qout-ba is beleshifteljük a kimenetet. A BrainBWin osztályról majd később beszélünk, de itt az osztály appName változóját érjük el és írjuk ki. A QString::fromUtf8 függvény segítségével UTF-8-as stringet írunk ki a parancsorra.

```
QRect rect = QApplication::desktop()->availableGeometry();
```

A QRect osztály segítségével egy téglalapot tudunk rajzolni a síkra, melynek méretét a kijelző méretével tesszük egyenlővé. A kijelző méretét úgy kapjuk meg, hogy a desktop() függvény visszaadja magát az asztalt, az availableGeometry() pedig annak a méretét.

```
BrainBWin brainBWin ( rect.width(), rect.height() );
```

Létrehozzuk a BrainBWin objektumunkat, melynek átadjuk a képernyő méretét. Ekkor meghívódik a BrainBWin konstruktora, amely a következőképpen néz ki:

```
BrainBWin::BrainBWin ( int w, int h, QWidget *parent ) : QMainWindow ( ↵
    parent )
{
    statDir = appName + " " + appVersion + " - " + QDateTime::currentDate() ↵
        .toString() + QString::number ( QDateTime::currentMsecsSinceEpoch() );
    brainBThread = new BrainBThread ( w, h - yshift );
    brainBThread->start();

    connect ( brainBThread, SIGNAL ( heroesChanged ( QImage, int, int ) ) ↵
        ,
        this, SLOT ( updateHeroes ( QImage, int, int ) ) );
    connect ( brainBThread, SIGNAL ( endAndStats ( int ) ),
        this, SLOT ( endAndStats ( int ) ) );
}
```

A QMainWindow osztály segítségével hozzuk létre a fő ablakot a programnak, és ezen belül készítjük el a BrainBWin ablakot. A hierarchikus sorrendet úgy érjük el, hogy a BrainBWin konstruktora meghívja a QMainWindow konstruktörét, és jellezzük neki, hogy ő a szülő. A QString típusú statDir változóban tároljuk a későbbiekben létrehozandó mappa nevét. Mjd a BrainBWin.h-ban deklarált BrainBThread objektumra mutató muutatót ráállítjuk egy újonan lefoglalt tárterületre. Ehhez meghívódik a BrainBThread konstruktora.

```
BrainBThread::BrainBThread ( int w, int h )
```

```
{  
  
    dispShift = heroRectSize+heroRectSize/2;  
  
    this->w = w - 3 * heroRectSize;  
    this->h = h - 3 * heroRectSize;  
  
    std::srand ( std::time ( 0 ) );  
  
    Hero me ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - ←  
              100,  
              this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ) - ←  
              100,  
              255.0 * std::rand ←  
              () / ( RAND_MAX ←  
              + 1.0 ), 9 );  
  
    Hero other1 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←  
              ) - 100,  
              this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←  
              ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), ←  
              5, "Norbi Entropy" );  
    Hero other2 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←  
              ) - 100,  
              this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←  
              ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), ←  
              3, "Greta Entropy" );  
    Hero other4 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←  
              ) - 100,  
              this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←  
              ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), ←  
              5, "Nandi Entropy" );  
    Hero other5 ( this->w / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←  
              ) - 100,  
              this->h / 2 + 200.0 * std::rand() / ( RAND_MAX + 1.0 ←  
              ) - 100, 255.0 * std::rand() / ( RAND_MAX + 1.0 ), ←  
              7, "Matyi Entropy" );  
  
    heroes.push_back ( me );  
    heroes.push_back ( other1 );  
    heroes.push_back ( other2 );  
    heroes.push_back ( other4 );  
    heroes.push_back ( other5 );  
  
}  

```

Paramáterként átadjuk a magasságot és a szélességet. A dispShift alapértelmezett értéke 40, és a heroRectSize-é is. AZ első sorban a dispShift-hez hozzáadjuk a heroRectSize másfélszeresét. Ezután pedig random számokat generálunk, és létrehozzuk az öt alapértelmezett "karaktert", ami meg fog jelenni a képernyőn. A Hero class a BrainBThread.h-ban van definiálva. A konstruktorának

5 paraméterre van szüksége, de ha nem adunk meg, akkor használja az alapértelmezett értékeket:

```
Hero ( int x=0, int y=0, int color=0, int agility=1, std::string name = " ← Samu Entropy" )
```

Az mē-nek nem adtunk meg nevet, ezért ō lesz Samu Entropy, akit majd követni kell az egérrel. A létrehozott hősököt pedig a heroes vektorba rakjuk. A hősök mozgásáért a move függvény lesz a felelős.

```
void move ( int maxx, int maxy, int env ) {

    int newx = x+ ( ( double ) agility*1.0 ) * ( double ) ( std::rand ← () / ( RAND_MAX+1.0 ) )-agility/2 ;
    if ( newx-env > 0 && newx+env < maxx ) {
        x = newx;
    }
    int newy = y+ ( ( double ) agility*1.0 ) * ( double ) ( std::rand ← () / ( RAND_MAX+1.0 ) )-agility/2 ;
    if ( newy-env > 0 && newy+env < maxy ) {
        y = newy;
    }
}
```

Ezzel új pozíciót adunk meg a hősöknek, annak a függvéynében, hogy hol van hozzá képest a másik hős, ezt az env változóban tároljuk. Az új pozíció meghatározásában fontos szerepe van az agilitásnak is, hiszen az előző x-hez az agilitás random számszorosának és az agilitás felének a különbségét adjuk. Ha a kapott szám és a környezet különbsége nagyobb, mint nulla, és összegük kisebb, mint a max x érték, akkor ez lesz az új x koordinátája a hősnek. Ugyan ezt hajtjuk végre az y koordinátával is.

A BrainBThread osztály definíciója a következő:

```
class BrainBThread : public QThread
{
    Q_OBJECT

    //Norbi
    cv::Scalar cBg { 247, 223, 208 };
    cv::Scalar cBorderAndText { 47, 8, 4 };
    cv::Scalar cCenter { 170, 18, 1 };
    cv::Scalar cBoxes { 10, 235, 252 };
```

A fejrészben láthatjuk, hogy a BrainBThread osztály az alosztályaa lesz a QThread osztálynak. A cv:: használatához volt szükség az openCV telepítésére. A Scalar osztály konstruktora 4 paramétert kér, mi ebből csak 3-at adunk meg, a negyediket alapértelmezett értéken hagyjuk. A változók neveiből sejthető, hogy jeen esetben RGB kódot tárolunk bennük, ezzel határozzuk meg a hősök színét.

```
Heroes heroes;
int heroRectSize {40};

cv::Mat prev {3*heroRectSize, 3*heroRectSize, CV_8UC3, cBg };
int bps;
long time {0};
long endTime {10*60*10};
```

```
int delay {100};

bool paused {true};
int nofPaused {0};

std::vector<int> lostBPS;
std::vector<int> foundBPS;

int w;
int h;
int dispShift {40};
```

Az első sorban deklaráljuk a `heroes` vektort. Majd adunk a `heroRectSize` értékét 40-re állítjuk, ez lényegében a hösökt ábrázoló téglalap mérete. A `cv::Mat` osztály segítségével több dimenziós tömböt hozunk létre, amiben megadjuk a sorok, oszlopok számát, egy típust, és egy `Scalar` vektort. A típussal jelöljük, hogy a `Scalar` vektorban RGB színeket tárolunk. Majd létrehozzuk, a `bps` változót, melyben a pillanatnyi bps értékeket fogjuk tárolni. Megadjuk a benchmark idejét, ahogy láthatod, 10 percre van állítva. A `lostBPS` és a `foundBPS` vektorokban fogjuk tárolni a az elvesztés és a megtalálás pillanatában fennálló bps-t.

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

Megoldás videó: <https://youtu.be/j7f9SkJR3oc>

Megjegyzés

A feladat megoldásában turorként részt vett: Halász Dávid A feladat megoldásában tutoráltként részt vett: Stiegemayer Máté

Megoldás forrása: <https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0> (/tensorflow-0.9.0/tensorflow/exa...
https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol

Tanulságok, tapasztalatok, magyarázat...

Az MNIST egy nagy adatbázis, melyben kézze írott számjegyek vannak eltárolva. Ezt sokszor képfelisemrő programok tanítására szokták felhasználni, ahogy mi is tesszük ebben a feladatban. Magát a programot python nyelven fogjuk megírni, de szükségünk lesz még a TensorFlow telepítésére is. A TensorFlow egy olyan szoftverkönyvtár, amit gépi tanáshoz használnak. Magát a számításokat egy irányított gráfkként lehet elközpelní. Vannak csúcsok, amik műveleteket szimbolizálnak, rendelkezhetnek több bemenettel is. Az adatok a csúcsokba az élek mentén történik, ezen haladnak a tensorok, melyek tetszőleges dimenziójú vektorok. A gép tanításához pedig a softmax függvényt fogunk felhasználni.

TensorFlow telepítése

```
sudo apt update  
sudo apt install python3-dev python3-pip  
sudo pip3 install -U virtualenv  
sudo apt-get install python3-tk
```



```
#virtuális környezet létrehozása  
virtualenv --system-site-packages -p python3 ./venv  
source ./venv/bin/activate  
#ekkor megjelenik egy (venv) felirat a terminalban  
pip install --upgrade tensorflow  
python -c "import tensorflow as tf; tf.enable_eager_execution(); ←  
    print(tf.reduce_sum(tf.random_normal([1000, 1000])))"  
pip install matplotlib
```

Részletesebb leírásért látogass el a [TensorFlow hivatalos oldalára](#)

Megjegyzés

A program azon része, hogy a saját képünket ismerje fel, jelenleg nem működik, ezért a forrás módosítva lett, úgy, hogy helyette egy MNIST mintát olvasunk be.

Módosított forrás:[itt](#)

A python interpreteres nyelv, ezért nincs szükség rá, hogy lefordítsuk, elég kiadni a `python fajlnev.py` parancsot, és a program lefut. Nézzük meg a forrást.

```
from __future__ import absolute_import  
from __future__ import division  
from __future__ import print_function  
  
import argparse  
  
# Import data  
from tensorflow.examples.tutorials.mnist import input_data  
  
import tensorflow as tf  
old_v = tf.logging.get_verbosity()  
tf.logging.set_verbosity(tf.logging.ERROR)  
  
import matplotlib.pyplot
```

Az elején, ahogy a C/C++ nyelvekben már megszokhattuk, importáljuk a szükséges könyvtárakat. Itt egy apróbb módosítást láthattok az alap forráshoz képest.

```
import tensorflow as tf  
old_v = tf.logging.get_verbosity()  
tf.logging.set_verbosity(tf.logging.ERROR)
```

Ezt a részt azért kellett kiegészíteni az utolsó két sorral, mert enélkül a program hibát dobott. Ahogy látható, ezen a részen importáljuk a TensorFlow szoftver könyvtárat. A `matplotlib.pyplot` könyvtárra azért van szükség, hogy a képet ki tudjuk írni egy fájlba.

```
# def readimg():
#     file = tf.read_file("sajat8a.png")
#     img = tf.image.decode_png(file)
#     return img
```

Ez a függvény azért felelne, hogy saját képeket tudunk beolvasni. Most ez a fentebb említett okok miatt ki lett kommentezve.

```
def main(_):
    mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

    # Create the model
    x = tf.placeholder(tf.float32, [None, 784])
    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))
    y = tf.matmul(x, W) + b

    # Define loss and optimizer
    y_ = tf.placeholder(tf.float32, [None, 10])

    # The raw formulation of cross-entropy,
    #
    # tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y))),
    #                 reduction_indices=[1]))
    #
    # can be numerically unstable.
    #
    # So here we use tf.nn.softmax_cross_entropy_with_logits on the raw
    # outputs of 'y', and then average across the batch.
    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
        labels=y_, logits=y))
    train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

    sess = tf.InteractiveSession()
    # Train
    tf.initialize_all_variables().run(session=sess)
    print("-- A halozat tanitása")
    for i in range(1000):
        batch_xs, batch_ys = mnist.train.next_batch(100)
        sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
        if i % 100 == 0:
            print(i/10, "%")
    print("-----")

    # Test trained model
    print("-- A halozat tesztelese")
```

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print("-- Pontosság: ", sess.run(accuracy, feed_dict={x: mnist.test.images,
                                                       y_: mnist.test.labels}))
print("-----")

print("-- A MNIST 42. tesztképenek felismerése, mutatom a számot, a továbblepeshez csukd be az ablakat")

img = mnist.test.images[42]
image = img

matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm.binary)
matplotlib.pyplot.savefig("4.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

print("-- Ezt a hálózat ennek ismeri fel: ", classification[0])
print("-----")

#print("-- A saját kezi 8-asom felismerése, mutatom a számot, a továbblepeshez csukd be az ablakat")
print("-- A MNIST 11. tesztképenek felismerése, mutatom a számot, a továbblepeshez csukd be az ablakat")
# img = readimg()
# image = img.eval()
# image = image.reshape(28*28)
img = mnist.test.images[11]
image = img
matplotlib.pyplot.imshow(image.reshape(28,28), cmap=matplotlib.pyplot.cm.binary)
matplotlib.pyplot.savefig("8.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

print("-- Ezt a hálózat ennek ismeri fel: ", classification[0])
print("-----")

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/mnist/input_data',
                        help='Directory for storing input data')
    FLAGS = parser.parse_args()
    tf.app.run()
```

Maga a main függvény a hivatalos TensorFlow péda programja az MNIST-re. Néhány módosítást pesrsze ehhez is végre kellett hajatni.

```
#a régi verzió
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, ←
    _y))
#a módosított
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(←
    labels = y_, logits = y))
```

Az újabb TensorFlow verzió más szintaktikát alkalmaz, ezért volt szükség az átalakításra.

A programunk azon alapszik, hogy beolvassunk 28x28-as képeket, és ezekről eldönti, hogy milyen szám van rajtuk. Ezeket a képeket vektorrá alakítjuk, így kapunk 784 dimenziós álló vektorokat. Az MNIST adatbázisban található képekhez tartozik egy címke, ami azt tartalmazza, hogy milyen szám van a képen. Ez a címke, egy 10 elemű vektor, amiben nullák vannak és egy 1-es. Az 1-es pozíciója határozza meg, hogy milyen számról van szó. Azt, hogy miylen valószínűséggel találja el a programunk a számot, a softmax függvény segítségével tudjuk meg, ez egy listát ad, az egyezés valószínűségéről. Az értékek nulla és egy között váltakoznak, összességében 1 az összegük. A softmax regresszió 2 lépésből áll. Elsőnek ki kell számolnunk egy evidenciát. Ez egy szám, amely az bemeneti adatnak az osztályba tartozását reprezentálja. Végezetül az evidencia képlete:

$$\sum_j W_{i,j} x_j + b_i$$

A képletben a W jelöli az i -dik osztály súlyát, b a torzítási értéket, x pedig a bemenetet. Majd az így kiszámított összeget adjuk át a softmax függvénynek. Ezeket az utasításokat hajtjuk végre a main első részében.

```
x = tf.placeholder(tf.float32, [None, 784]) # Nx784 mátrix
W = tf.Variable(tf.zeros([784, 10])) #784x10 mátrix
b = tf.Variable(tf.zeros([10])) #10x1 mátrix (vektor)
y = tf.matmul(x, W) + b           # Nx10 mátrix
```

Tehát a bemenetet reprezentáló x -nél létrehozunk egy tensort. Megadjuk a beolvasandó képek mind 784 dimenziós vektork, a None pedig azt jelöli, hogy tetszőleges számú bemenetünk lehet. A súlyokat és a hibaértékeket pedig nullára állítjuk elsőnek. A `matmul` segítségével megvalósítjuk a mátrix szorzást és a mátrix összeadást. Az így kapott több dimenziós vektort pedig az y változóban tároljuk. Ezzel elkészítettük a modellünket. A modell tanításához a keresztnatrópia hibafüggvényt foguk használni. Erre azért van szükség, mert meg kell határoznunk a programnak, hogy mi a jó modell. Ennek pont az ellentétét adjuk meg a keresztnatrópia segítségével. A függvény a következő:

$$H_y'(y) = \sum_i y'_i \log(y_i)$$

Tehát az y' jelöli, hogy modellünk által jóvoltal valószínűség eloszlást, a y' pedig a valódit.

```
y_ = tf.placeholder(tf.float32, [None, 10]) # Nx10 mátrix
```

A $y_$ egy újabb tensor, ami pedig a valódi valószínűségeket fogja tárolni. Ezutá pedig már implementálhatjuk a keresztnatrópia hibafüggvényt:

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(←
    labels = y_, logits = y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(←
    cross_entropy)
```

A `train_step` változóban pedig egy folymatot tárolunk, amit a `tf.train.GradientDescentOptimizer` ad vissza. A folyamat itt egy csomópontot jelent, ami tensorokon műveleteket hajt végre. Ez egy grádiens módszeren alapuló optimalizálás, mely segítségével csökkenthetjük a hibaszázalékot. Ezek után interaktív session-be léptetjük a modellt.

```
sess = tf.InteractiveSession()
# Train
tf.initialize_all_variables().run(session=sess)
print("-- A halozat tanitasa")
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
    if i % 100 == 0:
        print(i/10, "%")
print("-----")
```

A `tf.initialize_all_variables().run(session=sess)` segítségével inicializáljuk a változókat. Majd `for` cikluson belül 1000-szer végrehajtjuk a tanítást a pontosabb eredmény érdekében, folyamat előrehaladását a felhasználó a terminálban láthatja. Ha pedig ezzel végeztünk, kezdhetjük a modell tesztelését.

```
print("-- A halozat tesztelese")
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print("-- Pontosság: ", sess.run(accuracy, feed_dict={x: mnist.test.←
    images,
    y_: mnist.test.labels}))
print("-----")

print("-- A MNIST 42. tesztképenek felismerése, mutatom a szamot, a ←
    továbblepeshez csukd be az ablakat")

img = mnist.test.images[42]
image = img

matplotlib.pyplot.imshow(image.reshape(28, 28), cmap=matplotlib.pyplot.cm.←
    .binary)
matplotlib.pyplot.savefig("4.png")
matplotlib.pyplot.show()

classification = sess.run(tf.argmax(y, 1), feed_dict={x: [image]})

print("-- Ezt a halozat ennek ismeri fel: ", classification[0])
print("-----")
```

Első lépésként megtudjuk, hogy a modell mennyire pontos. Ehhez elsőnek összehasonlítjuk a legnagyobb értékeinek az indexét az `y` és `y_` vektoroknak. Az indexeket a `tf.argmax()` függvény adja meg. Ennek az értékét adjuk át paraméterként a `tf.equal()` függvénynek. Ez egy listát ad vissza, hamis és igaz értékkel. Ezeket az értékeket lebegőpontos számmá alakítjuk a `tf.cast()` segítségével, majd kiszámoljuk a lista eleminek átlag értékét a `tf.reduce_mean()` függvény segítségével, ami a pontosságát adja

a modellünknek. Ezt a felhasználóval tudatjuk, úgy hogy kiszámoljuk a pontosságot a tesztelő adatokra nézve, mely eredményt a `sess.run()` függvény adja vissza. Majd megnyitjuk az első képet, és ha a felhasználó bezárja az ablakot, akkor megtudjuk a program melyik számra tippel. Eddig tesztelésem során nem hibázott, szóval elég nagy eséllyel eltalálja. A program, amúg 90%-os ponotossággal dolgozik, ez teszi lehetővé a jó eredményt. A képeket az MNIST adatbázisból töltjük be, a `matplotlib` könyvtár segítségével rajzoltatjuk ki a képet a felhasználó elé, majd elmentjük a képet egy fájlba. Azt, hogy minek ismerte fel a program a képet, a `classification` listában tároljuk el.

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/ ←
        mnist/input_data',
                        help='Directory for storing input data')
    FLAGS = parser.parse_args()
    tf.app.run()
```

A program elején importáltuk az `argparse` könyvtárat. Ezt arra tudjuk használni, hogy kapcsolókat tudunk létrehozni a programunk számára. Jelen esetben a `--data_dir` kapcsoló használatával meg tudjuk adni, hogy hol tárolja a program a tanuláshoz szükséges állományokat.

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...



Passzolva

8.3. Minecraft-MALMÖ

Megoldás videó: <https://youtu.be/bAPSu3Rndi8>

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

A Project Malmo-t a Microsoft indította el a mesterséges intelligenciában való kutatás éredkében. A lényege, hogy a programozók beprogramozhatják a Minecraft karakter mozgását, de nem csak azt, hogy hogyan mozognak, hanem azt is, hogy felismerjék a környezetüket. Tehát képesek vagyunk a gépi tanulást szimulálni a játékban. A gép megtanulja, hogyan maradjon életben a virtuális világban. Ebben a feladtból azt fogjuk megnézni, hogy ezt hogyan valósítja meg a program. Ebben a feladatban is a Python programozási

nyelvet fogjuk használni, lényegében a projekt python-os példáit nézzük át. A Malmo projekt Github-on megtalálható csomagjai tartalmaznak tutorial feladatokat, melyek segítségével a kezdők megismerkedhetnek a programmal. Lássuk az első példát:

```
# Tutorial sample #1: Run simple mission

from builtins import range
import MalmoPython
import os
import sys
import time

if sys.version_info[0] == 2:
    sys.stdout = os.fdopen(sys.stdout.fileno(), 'w', 0) # flush print ←
        output immediately
else:
    import functools
    print = functools.partial(print, flush=True)

# Create default Malmo objects:

agent_host = MalmoPython.AgentHost()
try:
    agent_host.parse( sys.argv )
except RuntimeError as e:
    print('ERROR:',e)
    print(agent_host.getUsage())
    exit(1)
if agent_host.receivedArgument("help"):
    print(agent_host.getUsage())
    exit(0)

my_mission = MalmoPython.MissionSpec()
my_mission_record = MalmoPython.MissionRecordSpec()

# Attempt to start a mission:
max_retries = 3
for retry in range(max_retries):
    try:
        agent_host.startMission( my_mission, my_mission_record )
        break
    except RuntimeError as e:
        if retry == max_retries - 1:
            print("Error starting mission:",e)
            exit(1)
        else:
            time.sleep(2)

# Loop until mission starts:
print("Waiting for the mission to start ", end=' ')
```

```
world_state = agent_host.getWorldState()
while not world_state.has_mission_begun:
    print(".", end="")
    time.sleep(0.1)
    world_state = agent_host.getWorldState()
    for error in world_state.errors:
        print("Error:",error.text)

print()
print("Mission running ", end=' ')

# Loop until mission ends:
while world_state.is_mission_running:
    print(".", end="")
    time.sleep(0.1)
    world_state = agent_host.getWorldState()
    for error in world_state.errors:
        print("Error:",error.text)

print()
print("Mission ended")
# Mission has ended.
```

Ez a példa azt mutatja meg, hogy hogyan tudunk elindítani egy küldetést a Malmo mod segítségével. Első lépésként létre kell hoznunk egy `agent_host` objektumot. Majd az ehhez tartozó függvények segítségével elindítjuk a küldetést, anélkül, hogy a futó Minecraft programba mi belenyúlnánk. A küldetés 10 másodpercig tart, majd a küldetés befejeződik. Ezután a felhasználó irányíthatja a Minecraft karaktert. Lehetőség van arra is, hogy XML szerűen tároljuk el az utasításokat. Erre ad példát a második példa.

```
missionXML='''<?xml version="1.0" encoding="UTF-8" ?>
<Mission xmlns="http://ProjectMalmo.microsoft.com" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <About>
        <Summary>Hello world!</Summary>
    </About>

    <ServerSection>
        <ServerHandlers>
            <FlatWorldGenerator generatorString="3;7,220*1,5*3,2;3;,biome_1"/>
            <ServerQuitFromTimeUp timeLimitMs="30000"/>
            <ServerQuitWhenAnyAgentFinishes/>
        </ServerHandlers>
    </ServerSection>

    <AgentSection mode="Survival">
        <Name>MalmoTutorialBot</Name>
```

```
<AgentStart/>
<AgentHandlers>
    <ObservationFromFullStats/>
    <ContinuousMovementCommands turnSpeedDegs="180"/>
</AgentHandlers>
</AgentSection>
</Mission>'''
```

Tehát a missionXML változóban tároljuk az utasításokat. A pprogram kiírja, hogy Hello world!, generál egy világot, majd 30 másodpercig fut, végül átadja az irányítást a felhasználónak. Mivel lehetőséünk van befolyásolni a világot, ezért rajzolhatunk is benne tártygakat. Ezt mutatja be a harmadik példa. A rajzoláshoz szükségünk van elsőnek egy függvényre:

```
def Menger(xorg, yorg, zorg, size, blocktype, holetype):
    #draw solid chunk
    genstring = GenCuboid(xorg,yorg,zorg,xorg+size-1,yorg+size-1,zorg+size ←
        -1,blocktype) + "\n"
    #now remove holes
    unit = size
    while (unit >= 3):
        w=old_div(unit,3)
        for i in range(0, size, unit):
            for j in range(0, size, unit):
                x=xorg+i
                y=yorg+j
                genstring += GenCuboid(x+w,y+w,zorg, (x+2*w)-1, (y+2*w)-1, ←
                    zorg+size-1,holetype) + "\n"
                y=yorg+i
                z=zorg+j
                genstring += GenCuboid(xorg,y+w,z+w,xorg+size-1, (y+2*w) ←
                    -1,(z+2*w)-1,holetype) + "\n"
                genstring += GenCuboid(x+w,yorg,z+w, (x+2*w)-1,yorg+size-1, ( ←
                    z+2*w)-1,holetype) + "\n"
        unit = w
    return genstring
```

Ez a függvény felelős azért, hogy rajtoljunk egy nagy kocka, és abban pedig apró lyukakat végejunk. A végeredmény pedig ez lesz:



8.1. ábra. Tutorial_3

A 4. példa mutatja be azt, hogyan lehet irányítani a karaktert. Jelen esetben a az előbb létrehozott lyukacsos test belsejébe juttat el. Ezt a következő parancsokkal érjük el:

```
agent_host.sendCommand("hotbar.9 1")
agent_host.sendCommand("hotbar.9 0")
agent_host.sendCommand("pitch 0.2")
time.sleep(1)
agent_host.sendCommand("pitch 0")
agent_host.sendCommand("move 1")
agent_host.sendCommand("attack 1")
```

Terhát szintaktikailag így lehet átadni parancsot a karakternek. De ez még nem az a szint, amit a bevezetőben belengetttem, amikor már a karakterünk érzékeli a környezetét. Viszont az ötödik példában már ilyet is láthatunk. Na jó, persze ezt is be kell programozni, nem a gép jön rá magától, de már egy fokkal intelligensebb, mint az előző feladatban.

```
if world_state.number_of_observations_since_last_state > 0:
    msg = world_state.observations[-1].text
    observations = json.loads(msg)
    grid = observations.get(u'floor3x3', 0)
    if jumping and grid[4]!=u'lava':
        agent_host.sendCommand("jump 0")
        jumping = False
    if grid[3]==u'lava':
        agent_host.sendCommand("jump 1")
```

```
jumping = True
```

A `json.loads(msg)` függvény segítségével tesszük lehetővé, hogy a környezetet tudjuk vizsgálni. A grid változónak átadjuk a talajt, majd azt vizsgálva, ahol láva van a karakter bizonyos környezetében, akkor a karakter ugrik egyet. A 6-os példa hozza el az igazi áttörést gép tanulás tekintetében. Itt a Q-learning technológiát használja a program. A küldetés többször lefut, és feljegyzi, hogy mennyit tudott mozogni a karakter az egyes irányokba, ameddig meghalt. Ezzel azt éri el, hogy életben tudja tartani a gép a karaktert.

```
class TabQAgent(object):
    ...

```

A lényeget a TabQAgent osztály tartalmazza. Az osztály konstruktora a következő:

```
def __init__(self):
    self.epsilon = 0.01 # chance of taking a random action instead of the ←
                      best

    self.logger = logging.getLogger(__name__)
    if False: # True if you want to see more information
        self.logger.setLevel(logging.DEBUG)
    else:
        self.logger.setLevel(logging.INFO)
    self.logger.handlers = []
    self.logger.addHandler(logging.StreamHandler(sys.stdout))

    self.actions = ["movenorth 1", "movesouth 1", "movewest 1", "moveeast 1"]
    self.q_table = {}
    self.canvas = None
    self.root = None
```

Itt meg vannak adva a lehetséges parancsok, amiket a karakter végrehajthat. Ezek közül fog választani majd a program félre random, félre pedig a megismert információk alapján.

```
def updateQTable( self, reward, current_state ):
    """Change q_table to reflect what we have learnt."""

    # retrieve the old action value from the Q-table (indexed by the ←
    # previous state and the previous action)
    old_q = self.q_table[self.prev_s][self.prev_a]

    # TODO: what should the new action value be?
    new_q = old_q

    # assign the new action value to the Q-table
    self.q_table[self.prev_s][self.prev_a] = new_q

def updateQTableFromTerminatingState( self, reward ):
    """Change q_table to reflect what we have learnt, after reaching a ←
    terminal state."""
```

```
# retrieve the old action value from the Q-table (indexed by the ←
# previous state and the previous action)
old_q = self.q_table[self.prev_s][self.prev_a]

# TODO: what should the new action value be?
new_q = old_q

# assign the new action value to the Q-table
self.q_table[self.prev_s][self.prev_a] = new_q
```

A program futása során megjelenik egy táblázat, amiben a megismert információkat követheti a felhasználó. Az előbbi két függvény ennek a táblázatnak a frissítését végzi.

```
def act(self, world_state, agent_host, current_r):
    ...

    rnd = random.random()
    if rnd < self.epsilon:
        a = random.randint(0, len(self.actions) - 1)
        self.logger.info("Random action: %s" % self.actions[a])
    else:
        m = max(self.q_table[current_s])
        self.logger.debug("Current values: %s" % ", ".join(str(x) for x ←
            in self.q_table[current_s]))
        l = list()
        for x in range(0, len(self.actions)):
            if self.q_table[current_s][x] == m:
                l.append(x)
        y = random.randint(0, len(l)-1)
        a = l[y]
        self.logger.info("Taking q action: %s" % self.actions[a])
```

A karakter által elvégzett mozgásokat a `act` függvény határozza meg. Ahogy látható, ha a random szám kisebb, mint a konstruktörben a véletlen lépés esélyére adott értéknél, akkor az alapértelmezetten megadott mozgások egyikét választja a program, anélkül, hogy figyelembe venné a már megismert adatokat. Ellenkező esetben pedig az aktuális állapot alapján választ egy random lépést.

```
def run(self, agent_host):
    ...
    # main loop:
    world_state = agent_host.getWorldState()
    while world_state.is_mission_running:

        current_r = 0

        if is_first_action:
            # wait until have received a valid observation
            while True:
                time.sleep(0.1)
                world_state = agent_host.getWorldState()
                for error in world_state.errors:
```

```
        self.logger.error("Error: %s" % error.text)
        for reward in world_state.rewards:
            current_r += reward.getValue()
        if world_state.is_mission_running and len(world_state.observations)>0 and not world_state.observations[-1].text=="{}":
            total_reward += self.act(world_state, agent_host, current_r)
            break
        if not world_state.is_mission_running:
            break
    is_first_action = False
else:
    # wait for non-zero reward
    while world_state.is_mission_running and current_r == 0:
        time.sleep(0.1)
        world_state = agent_host.getWorldState()
        for error in world_state.errors:
            self.logger.error("Error: %s" % error.text)
        for reward in world_state.rewards:
            current_r += reward.getValue()
    # allow time to stabilise after action
    while True:
        time.sleep(0.1)
        world_state = agent_host.getWorldState()
        for error in world_state.errors:
            self.logger.error("Error: %s" % error.text)
        for reward in world_state.rewards:
            current_r += reward.getValue()
        if world_state.is_mission_running and len(world_state.observations)>0 and not world_state.observations[-1].text=="{}":
            total_reward += self.act(world_state, agent_host, current_r)
            break
        if not world_state.is_mission_running:
            break
```

A run függvény felelős a karakter mozgatásáért. Az aktuális állapot itt tárolódik, itt történik az act függvény meghívása. Ellenőrzi, hogy az első akciót hajtjuk-e végre.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása: [itt](#)

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban a Lisp nyelvet fogjuk megismerni. A nyelvet John McCarthy alkotta meg 1958-ban. Ez nevezhető a második magas szintű programozási nyelvnek, melynél csak a Fortran régebbi. Ez az alapján gondolhatnánk azt, hogy nem sok értelme van róla beszélni, tekintve korát, de a mesterséges intelligencia kutatások egyre nagyobb előretörése újra előtérbe hozta. Most nem a mestreséges intelligenciával foglalkozunk, és nem is a Lisp nyelvvel önmagában, hanem a GIMP képszerkesztő programjának egy különlegességével. Ezt nevezik Script-fu-nak, mely lehetőséget ad, hogy a Windows rendszereken megismert makrókhoz hasonló dolgokat alkossunk meg. Ehhez pedig a Lisp nyelvet fogjuk használni. A nyelv szintaktikailag nem bonyolult, viszont van pár dolog, amire oda kell figyelni. Ilyen az, hogy nem infix alakban kell megadni a műveleteket, hanem prefix-ben. A másik fontos dolog, hogy a zárójelek használatára is oda kell figyelni, ugyanis a Lisp teljesen zárójelezett programozási nyelv. Nézzük is, hogy hogyan néz ki a feladat megoldása.

Megjegyzés

Magát a gimpet így tudod letölteni (terminálon keresztül):

```
sudo apt-get install gimp
```

Maga a faktoriális kiszámítása nyelvtől függetlenül ugyan úgy kell megoldani. Persze vannak eltérő megoldások, de az alapja ugyanaz. Az $n!$ faktoriális értéke a következő:

```
n * (n-1) * (n-2) ... * 2 * 1
```

Az iteratív megoldásnál adná magát, hogy egy `for` vagy egy `while` ciklust kéne használni. De ez Lispben nem egészen így van. Ciklust fogunk használni, ami nagyon hasonlít a `while` ciklushoz, viszont ezt expliciten nem írjuk ki.

```
(define (fakt n)
  (if (= 0 n)
      (set! n 1)
    )
  (let loop ((variable (- n 1)))
    (if (> variable 1)
        (begin
          (set! n (* n variable))
          (loop (- variable 1) )
        )
      )
    )
  n
)
```

Tehát definiálunk egy függvényt, aminek a neve fakt, és egy paramétert kér. Az első if-ben ellenőrizzük, hogy a megadott paraméter nulla-e. Ha igen, akkor az n értékét 1-re állítjuk, mivel ez a 0! értéke. A let operátor teszi lehetővé, hogy változókat hozzunk létre, és azokkal művelteket hajtsunk végre. Jelen esetben a ciklusváltozót hozzuk létre, melynek értéke paraméterül kapott szám és az 1 különbsége. Aztán megfogalmazunk egy feltételt, ha a változó nagyobb, mint 1, akkor a paraméter értékét megváltoztatjuk a faktoriális szabályának megfelelően. Tehát a set segítségével adhatunk új értéket egy vékozónak. Ha ezt megtettük, akkor a csökkentjük a ciklusváltozót 1-el. Végezetül pedig visszadjuk az n értékét, amikor a ciklus lefutott.

Az iteratív megoldással szemben a rekurzív sokkal rövidebb, viszont átlátni talán egy kicsit nehezebb, hogy mi is történik a háttérben.

```
(define (fakt n)
  (
  if (< n 1)
    1
    (* n (fakt (- n 1))) ;else ag
  )
)
```

A függvény fejrésze teljesen megegyezik az előző példával, de utána láthatjuk a rekurzív megoldás erősséget. Nincs szükségünk ciklusokra. Eztazzal érjük el, hogy a függvény önmagát hívja meg, abban az esetben, ha a paraméter értéke nagyobb, mint 1. Ellenkező esetben pedig 1-et adunk vissza. Matematikai szempontból ez a megoldás teljesen helyes és pontos. Viszont fontos látni, hogy programozói szempontból a ez tartalmaz egy felesleges függvényhívást. Hiszen a vezérlés akkor is az else ágra ugrik, amikor az n 1, pedig tudjuk 1!-nak az értéke 1. A programot így módosíthatjuk:

```
(define (fakt n)
  (
  if (< n 2)
    1
    (* n (fakt (- n 1))) ;else ag
  )
)
```

Ezzel a módosítással elérjük, hogyha az $n = 1$, akkor ne ugorjon a vezérlés az `else` ágra. Hogy jobban lásd, mi is történik a háttérben ennél a megoldásnál, vess egy pillantást az alábbi ábrára:



9.1. ábra. Rekurzió

A képen láthatod, hogyan is zajlik az $5!$ -nak a kiszámolása. Tehát folyamatosan az `else` ágra ugrik a vezérlés, egészen addig, ameddig el nem érünk az $1!$ -ig. Ekkor visszatérünk eggyel, és fokozatosan haladunk vissza a hívási láncba, minden visszaadva a megfelelő értéket, vagyis az éppen aktuális n és az $(n-1)$ szorzatát.

9.2. Gimp Scheme Script-fu: króm effekt

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

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

Megoldás forrása: [itt](#)

Használat

Ha már telepítette a faktorálisos feladat alapján a GIMP-et, akkor a következőt kell tenned, hogy használhasd a scriptet:

```
cp *.scm /home/<user_name>/snap/gimp/165/.config/GIMP/2.10/scripts/
```

Tanulságok, tapasztalatok, magyarázat...

Az előző feladatban megismert Script-fu alrendszert fogjuk használni ebben a feladatban is. Ahogy a feladatban is látható, egy szövegre kell alkalmazni a króm effektet. Nézzük is meg, hogyan lehet ezt megvalósítani.

```
(define (color-curve)
  (let* (
    (tomb (cons-array 8 'byte))
  )
    (aset tomb 0 0)
    (aset tomb 1 0)
    (aset tomb 2 50)
    (aset tomb 3 190)
    (aset tomb 4 110)
    (aset tomb 5 20)
    (aset tomb 6 200)
    (aset tomb 7 190)
  tomb)
)
```

Első lépésként definiálunk egy függvényt, amely létrehoz egy 8 elemből álló tömböt, majd az elemeket beállítjuk a megfelelő értékekre.

```
(define (elem x lista)
  (if (= x 1) (car lista) (elem (- x 1) (cdr lista) ) )
)
```

Az elem függvény segítségével azt oldjuk meg, hogy elérjük egy lista x. elemét. Ehhez rekurziót használunk. A car függvény egy lista első elemét adja vissza, ezért abban az esetben, ha az x 1, akkor ezt hívjuk meg. A cdr pedig a lista első elemét törli. Tehát, ha a második elemet akarjuk megkapni, akkor a program törli a lista első elemét, majd visszadja a korábban még másodiknak számító elemet a car függvénnyel.

```
(define (text-wh text font fontsize)
(let*
  (
    (text-width 1)
    (text-height 1)
  )
  (set! text-width (car (gimp-text-get-extents-fontname text fontsize ←
PIXELS font)))
```

```
(set! text-height (elem 2 (gimp-text-get-extents-fontname text ←
    fontsize PIXELS font)))

(list text-width text-height)
)
)
```

A `text-wh` függvény egy listát ad vissza a paramétereként kapott szöveg szélességéről és magasságáról. A `let*` lehetővé teszi, hogy változókat hozhassunk létre. A szöveg méretének meghatározásához a `gimp-text-get-extents-fontname` függvényre van szükség, amely meghatározza a szöveget bekerítő "doboz" méretét. Egy listát ad vissza, melynek első paramétere a szélesség, a második a magasság. Ahhoz, hogy a `text-height` változó értékét megadjuk az `elem` függvény segítségével kell kinyernünk a viszszaadott list a második elemét. Ez a folyamat a `text-width` esetén könnyű, mivel elég a `car` függvényt használnunk. Most, hogy a szükséges segédfüggvényeket definiáltuk, jöhet a feladat lényegi megoldása a Chrome effekt. A `script-fu-bhax-chrome` egy szöveget, egy betűtipust, egy betűméretet, a létrehozandó kép méretét, egy színt és egy színátmenetet kér paraméteréül. Mint a többi függvénynél, itt is a változók definiálásával kezdünk:

```
(define (script-fu-bhax-chrome text font fontsize width height color ←
    gradient)
(let*
  (
    (image (car (gimp-image-new width height 0)))
    (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" 100 ←
        LAYER-MODE-NORMAL-LEGACY)))
    (textfs)
    (text-width (car (text-wh text font fontsize)))
    (text-height (elem 2 (text-wh text font fontsize)))
    (layer2)
  )
)
```

Az `image` változóban tároljuk el a `gimp-image-new` függvény által létrehozott képet, pontosabban annak az azonosítóját. A kép a felhasználó által megadott értékeknek megfelelő méretű lesz. A harmadik paraméter pedig azt jelöli, hogy RGB képet hoztunk létre. A `layer` változó fogja tartalmazni az újonan létrehozott réteg azonosítóját. A réteget a `gimp-layer-new` függvénytel alkotjuk meg. Paramétereként meg kell adni azt a képet, amihez hozzá szeretnénk adni a réteget, a réteg méretét, azt, hogy milyen színkezelést használjon. A réteget el is kell nevezni, meg kell határozni a kép átlátszóságát és a kombinációs módját. Tehát jelen esetben egy a képpel megyegyező méretű réteget hozunk létre, ami RGB színeket használ, a neve `bg` és teljesen átlátszó. A `text-width` és `text-height` változók értékét a `text-wh` függvény segítségével adjuk meg. Mivel ez a függvény egy listát ad vissza, ezért ismét alkalmaznunk kell az `elem` függvényt, ahhoz, hogy a szöveg magasságát meghatározzuk. A változókat definiáltuk, most jöhetnek az érdemi módosítások.

```
(gimp-image-insert-layer image layer 0 0)
(gimp-context-set-foreground '(0 0 0))
(gimp-drawable-fill layer FILL-FOREGROUND)
(gimp-context-set-foreground '(255 255 255))

(set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) ←
  ))
```

```
(gimp-image-insert-layer image textfs 0 0)
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (- (/ ←
height 2) (/ text-height 2)))

(set! layer (car(gimp-image-merge-down image textfs CLIP-TO-BOTTOM-←
LAYER)))
```

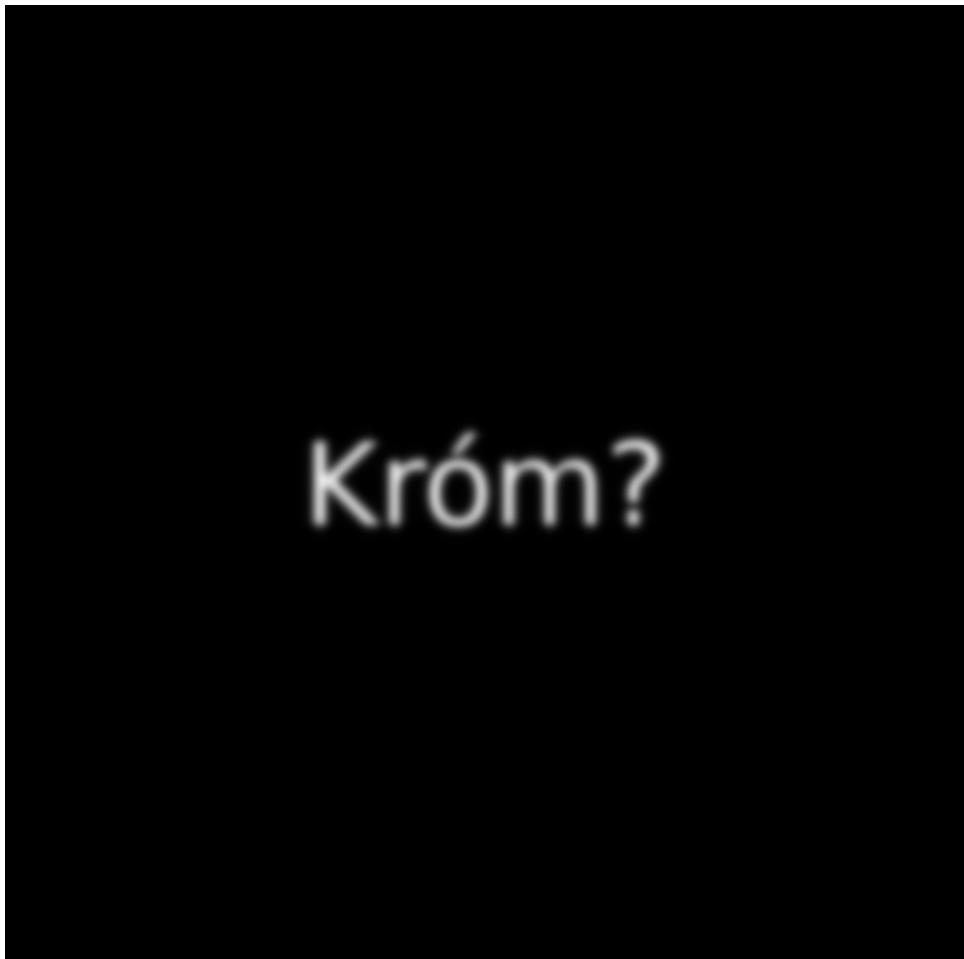
Elsőnek a `layer` változóhoz társított réteget beszúrjuk a létrehozott képbe. Az utolsó két paraméter azt jelölné, hogy van-e szülője a rétegnak, és listában elfoglalt helyét adják meg. Jelen esetben nincs szülőréteg, és a lista legtetejére rakjuk. Itt a listát úgy kell elképzelni, hogyha van sok réteg, akkor ebben az esetben minden egyik fölér kerülne az új réteg. A `gimp-context-set-foreground` beállítja az előtér színét, melyet a réteg színezéséhez használunk. Ezt a `gimp-drawable-fill` függvénytel oldjuk meg. Az első paramétere azt adja meg, hogy mit szeretnénk kitölteni, a második pedig azt, hogy előtér- vagy háttérszínnel. Ezután pedig az előtér színét megváltoztatjuk fehérrre. Ezzel a színnel fog kiíródni a szöveg addig, ameddig nem alkalmazzuk rá a Chrome effektet. A `textfs` változóban fogjuk tárolni a szöveges réteg azonosítóját. Szöveges réteget a `gimp-text-layer-new` függvény hoz létre. Természetesen meg kell adnunk a szöveget, a stílust, a méretet, és azt is, hogy a méret milyen mértékegységben van. Ezt a réteget is ráillesztjük a képre a már ismert módon, vagyis a lista elejére. A `gimp-layer-set-offsets` függvény segítségével be tudjuk állítani a réteg eltolását. Jelen esetben ezt arra használjuk, hogy a szöveget a kép közepére mozgassuk. Alapesetben a szöveg a bal felső sarokban lenne, ezért a középre való eltolás a kép szélességének és magasságának a fele lenne, viszont akkor a szöveg bal felső sarka lenne középen, emiatt még ki kell vonni a szöveg szélességét és magasságát. Végül pedig a szövegréteget lefelé az első aktív réteggel összevonjuk. Erre használjuk a `gimp-image-merge-down` függvényt. Ezzel meg van a kép, amit alapjául szolgál a feladat megoldásához.



9.2. ábra. Alap

(plug-in-gauss-iir RUN-INTERACTIVE image layer 15 TRUE TRUE)

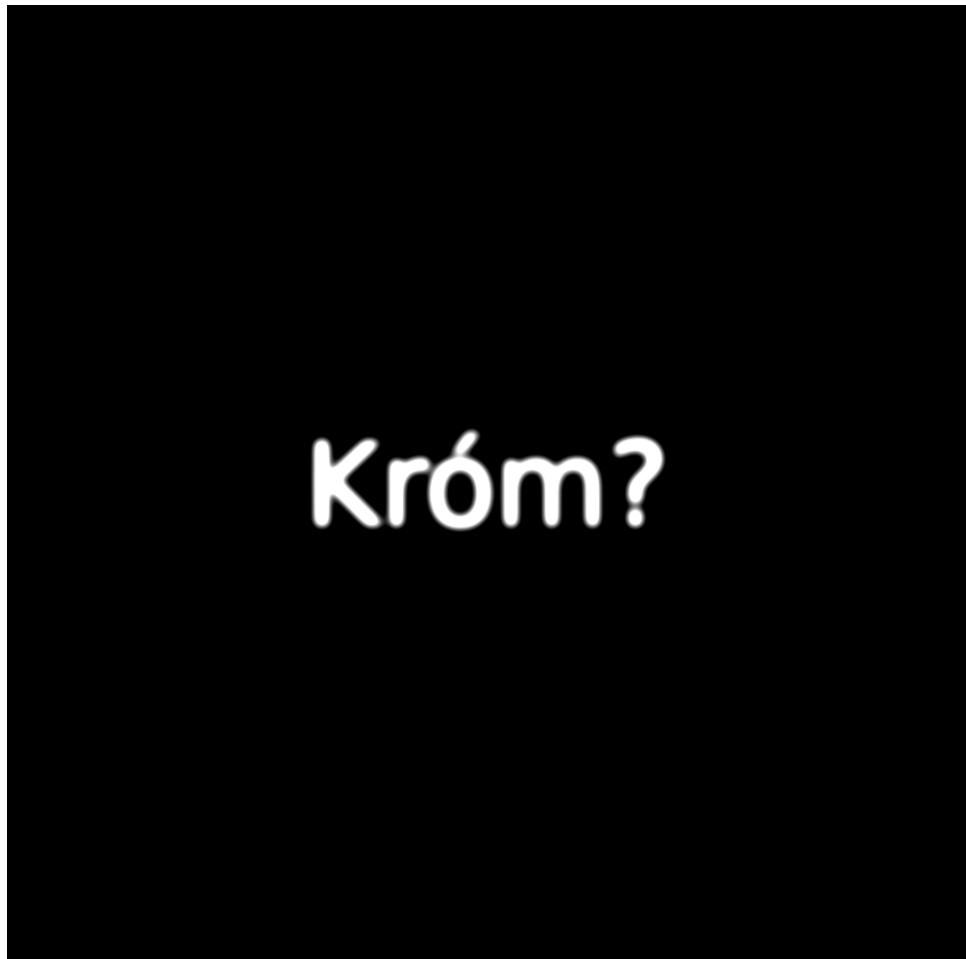
A plug-in-gauss-iir függvény segítségével a réteget elhomályosítjuk. A haramdik paraméterrel tudjuk meghatározni a homályosítás mértékét. Az utolsó két érték pedig azt szabályozza, hogy vízszintesen vagy függőlegesen homályosítson, most ezt mind a két irányból alkalmazzuk.



9.3. ábra. Blur effect

```
(gimp-drawable-levels layer HISTOGRAM-VALUE .11 .42 TRUE 1 0 1 TRUE)
```

Hogy ez a függvény mit csinál az már egy kicsit bonyolultabb. A histogram lényegében egy diagram lenne, ami azt mutatja, hogy az egyes színek árnyalatai hogyan oszlanak el a képen. Tehát ez a függvény befolyásolja a kép hisztogramján előforduló érékeket. A legalacsonabb értéknek az intenzitását kisebbre állítjuk, mint a legmagasabbét. Ezt láthatkuk a 3. és 4. paramétereknél. Az 5. paraméter a clampolást állítja be. A clampolásra azért van szükség, mert a sima képek élei túl recések lehetnek. Ez megoldható, ahogyan mi is csináltuk, homályosítással. Ezzel viszont az a baj, hogy a kép tartalam így nehezebben látható. A clampolás abban segít, hogy az elhomályosított kép részkleteit visszakapjuk, de az élek finomabbak maradjanak. A 6. paraméter a fényerősséget állítja be. A 7. azt jelenti, hogy a kimeneti értékek közül a legalacsonyabbat teljesen kiveszzük a képből, mert annak intenzitását 0-ra állítjuk. A legmagasabb értéket pedig maximális intenzitásra állítjuk. Végezetül pedig engedélyezzük a clampolás eredményeként kapott érékek visszaadását.



9.4. ábra. Clamp

(plug-in-gauss-iir RUN-INTERACTIVE image layer 2 TRUE TRUE)

A clampolás után kapott képen ismét alkalmazunk homályosítást, de azzal előbbinél jóval kisebb mértékben.

```
(gimp-image-select-color image CHANNEL-OP-REPLACE layer '(0 0 0))  
(gimp-selection-invert image)
```

A gimp-image-select-color függvény segítségével a rétegen minden fekete pixelt kijelölünk. Majd pedig a gimp-selection-invert inverzévé fordítja az előbbi kijelölést, tehát minden pixel, ami nem fekete, ki lesz jelölve.

```
(set! layer2 (car (gimp-layer-new image width height RGB-IMAGE "2" 100 ←  
LAYER-MODE-NORMAL-LEGACY)))  
(gimp-image-insert-layer image layer2 0 0)
```

A kijelölés után létrehozunk egy új réteget, amelyet kép rétegeinek a legtetejére rakunk. Ezt a réteget a layer2 változóban tároljuk.

```
(gimp-context-set-gradient gradient)
```

```
(gimp-edit-blend layer2 BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY GRADIENT- ←  
LINEAR 100 0 REPEAT-NONE  
FALSE TRUE 5 .1 TRUE width (/ height 3) width (- height (/ height ←  
3)) )
```

A hsználni kívánt színátmennetet a gimp-context-set-gradient függvény segítségével tudjuk beállítani. Jelen esetben a függvénynek a felhasználó által megadott színátmennetet adjuk paraméterül. Ezzel még nem alkalmaztuk a színátmennetet a szövegre, ahhoz a gimp-edit-blend függvényre van szükség. Ennek nem megyünk végig mindegyik paraméterén, de nézzük a fonotosabbakat. Elsőnek meg kell adni, hogy melyik rétegre szeretnénk alkalmazni, utána a színkeverést egyéniire állítjuk. A színátmennet típusa lináris lesz, tehát vizszintesen változik a színe a szövegnek. Ami fontos még a koordináták, meg kell adni a színezés kezdő és végpontját.



9.5. ábra. Színetmenet alkalmazása

```
(plug-in-bump-map RUN-NONINTERACTIVE image layer2 layer 120 25 7 5 5 0 ←  
0 TRUE FALSE 2)
```

A plug-in-bump-map függvény lehetővé teszi, hogy a szöveget dombornyomat szerűvé tegyük. Ehhez szükség van egy másik rétegre, ami kiindulásként szolgál folyamathoz. Lényegében az szolgáltatja a bump map-et.

```
(gimp-curves-spline layer2 HISTOGRAM-VALUE 8 (color-curve))
```

Végezetül a második réteg színeinek intenzitását módosítja egy görbe alapján. A görbe pontjait pedig a color-curve függvény tartalmazza. Lehetőségünk van alapértelmezett értékeket megadni a felhasználónak. Erre a szolgál a programkód következő része:

```
(script-fu-register "script-fu-bhax-chrome"
  "Chrome3"
  "Creates a chrome effect on a given text."
  "Norbert Bátfai"
  "Copyright 2019, Norbert Bátfai"
  "January 19, 2019"
  ""
  SF-STRING      "Text"        "Bátf41 Haxor"
  SF-FONT        "Font"        "Sans"
  SF-ADJUSTMENT  "Font size"   '(100 1 1000 1 10 0 1)
  SF-VALUE       "Width"       "1000"
  SF-VALUE       "Height"      "1000"
  SF-COLOR       "Color"       '(255 0 0)
  SF-GRADIENT    "Gradient"    "Crown molding"
)
```

Nem csak alapértelmezett adatok adhatók meg, hanem a script nevét is itt adhatjuk meg, amit a GIMP mutat majd, a script leírását és a szerzői jogokat is ide írjuk. Azt, hogy a GIMP menüjében hol található a script azt pedig a következő sorok határozzák meg:

```
(script-fu-menu-register "script-fu-bhax-chrome"
  "<Image>/File/Create/BHAX"
)
```

Végezetül pedig lássuk, hogy milyen lett a programunk által elkészített Chrome effekt.





9.6. ábra. Végeredmény

9.3. Gimp Scheme Script-fu: név mandala

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

Megoldás videó: https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelete_a_scheme_programozasi_nyelv

Megoldás forrása: [itt](#)

Használat

Ha már telepítetted a faktorálisos feladat alapján a GIMP-et, akkor a következőt kell tenned, hogy használhasd a scriptet:

 cp *.scm /home/<user_name>/snap/gimp/165/.config/GIMP/2.10/scripts/

Ha új betűstílusokat akarsz hozzáadni a GIMP-hez, akkor pedig ezt:

cp *.ttf /home/<user_name>/snap/gimp/165/.config/GIMP/2.10/fonts/

Tanulságok, tapasztalatok, magyarázat...

A mandala lényegében egy alakzat, amit a neved forgatásával, állítasz elő. Magát az alakzatot a GIMP-ben bárki meg tudja csinálni, erről egy [tutoriál](#) található az interneten. A mi programunk is nagyjából ezt valósítja meg, azzal a különbséggel, hogy a folyamatok automatizálva vannak, tehát nem kell kézzel előállítani a saját mandaládat.

Az előző feladatokban már megismerkedtünk a Lisp programozási nyelvvel, szóval a szintaktika már nem lesz újdonság, lássuk is a forrást:

```
(define (elem x lista)
  (if (= x 1) (car lista) (elem (- x 1) (cdr lista) ) )
)
```

Az első függvényünk egy listának az `x`. elemét adja vissza. Ehhez a rekurziót használjuk, tehát ha az első elemet keressük a listának, akkor a `car` függvénysegítségével visszaadjuk, de ha nem, akkor lesz szükség további hívásokra. A megoldás lényege, hogy minden egyet kisebb `x`-re hívjuk meg az `elem` függvényt, és a lista első elemét töröljük. Ha egy 5 elemű listából ki szeretnénk venni az utolsó elemet, akkor tehát 4-szer hívja meg önmagát a fófüggvény, majd visszaadja a lista első elemét, ami ekkor már az eredeti lista utolsó eleme lesz.

```
(define (text-width text font fontsize)
(let*
  (
    (text-width 1)
  )
  (set! text-width (car (gimp-text-get-extents-fontname text fontsize ←
    PIXELS font)))
    text-width
  )
)
```

A `text-width` függvény paraméteréül egy szöveget kér, annak a betűtípusát és a méretét. A `let*` függvény segítségével definiálunk egy változót. A `set!` függvény beállítja a `text_width` változó értékét az általunk megadott értékre. Jelen esetben ez a `gimp-text-get-extents-fontname` függvény által visszaadott lista első eleme. Ez a függvény egy szövegdobozt hoz létre az általunk megadott szöveggel, és ennek a szövegdoboznak a szélességével, magasságával tér vissza. A visszaadott listában van még két elem, de azok a feladat megoldásának szempontjából nem fontosak. Tehát a függvény visszaadja a szövegünk szélességét.

```
(define (text-wh text font fontsize)
(let*
  (
    (text-width 1)
    (text-height 1)
  )
  ;;;
  (set! text-width (car (gimp-text-get-extents-fontname text fontsize ←
    PIXELS font)))
)
```

```
;;; ved ki a lista 2. elemét
(set! text-height (elem 2 (gimp-text-get-extents-fontname text ←
    fontsize PIXELS font)))
;;
(list text-width text-height)
)
)
```

Ez a függvény nagyon hasonlít az előzőre, de ebben az a különbség, hogy nem csak a szélességet, de a magasságot is vissza kell adnunk. Az eredeti mandalához elég lenne az előző függvény, de a mi programunk a mandala közepére kiír egy másik szöveget is. Ennek a helyes pozícionálásához nélkülözhetetlen a `text-wh` függvény. Mivel a `gimp-text-get-extents-fontname` által visszaadott lista második eleme a magasság, ezért használjuk a már korábban megírt `elem` függvényt, ezzel a `text-height` változó értékét is meg tudjuk határozni. Végezetül visszadajuk a szövegdoboz szélességéből és magasságából álló listát.

Az eddig tárgyalt függvények csak segédfüggvények voltak, a mandala készítő függvény most jön.

```
(define (script-fu-bhax-mandala text text2 font fontsize width height ←
    text2_color gradient)
(let*
  (
    (image (car (gimp-image-new width height 0)))
    (layer (car (gimp-layer-new image width height RGB-IMAGE "bg" 100 ←
        LAYER-MODE-NORMAL-LEGACY)))
    (textfs)
    (text-layer)
    (text-width (text-width text font fontsize))
    ;;
    (text2-width (car (text-wh text2 font fontsize)))
    (text2-height (elem 2 (text-wh text2 font fontsize)))
    ;;
    (textfs-width)
    (textfs-height)
    (gradient-layer)
  )
)
```

A neve `script-fu-bhax-mandala`. Alap esetben csak egy szöveget kérne paraméteréül, de most kettőt kérünk a felhasználótól. Megadható a betűtípus, a betűméret, a kép mérete, a második szöveg színe és a kép elkészítéséhez szükséges grádiens. A szokásos módon létrehozunk változókat. Az `image` változóban tároljuk a képet, amit a `gimp-image-new-val` létrehozunk. Ennek képnek a mérete az általunk megadott paraméterek. Ezután létrehozunk egy új réteget, amelynél megadjuk, hogy melyik képhez adj a hozzá az új réteget, meagdjuk a réteg méretét, a típusát, ami jelen esetben RGB, tehát a réteg képes lesz kezelni az RGB színeket. Ezeken felül megadható a réteg neve, az átlátszósága és a réteg kombináció módja. A új réteget a `layer` változóban tároljuk. Ezen a részen érdemes még megemlíteni a `text-width`, `text2-width` és `text2-height` változókat. Ezeknek az értékét a már fentebb taglalt függvényekkel számoljuk ki. A többi változóról majd a későbbiekben lesz szó.

```
(gimp-image-insert-layer image layer 0 0)
```

```
(gimp-context-set-foreground '(0 255 0))
(gimp-drawable-fill layer FILL-FOREGROUND)
(gimp-image-undo-disable image)

(gimp-context-set-foreground text2_color)
```

A `gimp-context-insert-layer` függvény azt teszi lehetővé, hogy egy réteget hozzáadjunk egy képhez. Jelen esetben az `image` változóban tárolt képhez adjuk hozzá a korábban létrehozott rétegünket. Az utolsó 2 paraméter a szülő réteg számát és a beillesztés helyét adja meg. Tehát a réteg a kép többi rétegével egyenrangú lesz, mivel egyik sem a szülője, és ő kerül legfelülről. A `gimp-context-set-foreground` segítségével lehet az előtérszínt módosítani, de ez nem azt jelenti, hogy ezzel a parancssal a kép ilyen színű lesz, csak ezután bármit írnánk, rajzolnánk a képre, az ilyen színű lenne. A `gimp-drawable-fill` segítségével színezzük be a rétegünket az előbb megadott előtérszínnel. A `gimp-image-undo-disable` arra szolgál, hogy ne lehessen visszavonni az előző módosításokat az `image`-en. Végezetül újra meghívjuk a `gimp-context-set-foreground`, ezúttal a második szöveg színe lesz ilyen. Ha már elolvastad a belinkelt tutoriált, akkor tudhatod, hogy elsőnek maga az ábra is ilyen lesz, csak utána alkalmazunk rá grádienst, ami átszínezi a többit.

```
(set! textfs (car (gimp-text-layer-new image text font fontsize PIXELS) -->
    ))
(gimp-image-insert-layer image textfs 0 -1)
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text-width 2)) (/ -->
    height 2))
(gimp-layer-resize-to-image-size textfs)
```

A `textfs` változóban tároljuk el a `gimp-text-layer-new` által létrehozott szöveges rétegünket, amelhez meg kell adni, hogy melyik képre szeretnénk rögzíteni, milyen szöveggel, milyen betűtípus-sal és betűmérettel. Ezen kívül, még azt is meg kell adni, hogy a betűméretet milyen mértékegységben adtuk meg, jelen esetben pixeleken. Ezt a réteget beszűrjük az `image`-be, egészen pontosan az éppen aktív réteg fölé. A `gimp-layer-set-offsets` függvény felelős a réteg eltolásának a meghatározásában. Tehát a szöveget úgy toljuk el, hogy az x-tengelyen középen legyen, viszont az y tengelyen kicsit lejjebb, mint a középerték. Végezetül pedig a réteget a `kép` ,méretével megegyezővé tesszük a `gimp-layer-resize-to-image-size` függvénytel. Ezután létrehozunk egy új réteget.

```
(set! text-layer (car (gimp-layer-new-from-drawable textfs image)))
(gimp-image-insert-layer image text-layer 0 -1)
(gimp-item-transform-rotate-simple text-layer ROTATE-180 TRUE 0 0)
(set! textfs (car (gimp-image-merge-down image text-layer CLIP-TO-BOTTOM -->
    -LAYER)))
```

Az új réteget úgy kapjuk, hogy lényegében lemásoljuk az `textfs` réteget. Ehhez a `gimp-layer-new-from-` függvényre van szükség. Majd a réteget ugyan úgy beillesztjük a képbe, mint előzőleg, tehát az éppen aktív réteg fölé. Majd elforgatjuk 180°-al, ehhez a `gimp-item-transform-rotate-simple` függvényt használjuk. A paraméterek közül egyértelmű, hogy meg kell adni, hogy mit forgatunk, és hány fokban. Viszont az utolsó 3 már érdekesebb, a `TRUE` azt jelenti, hogy a automatikusan a kijelölés közepe körül forgasson, az utolsó kettő pedig a forgatás középpontjának koordinátái. A `gimp-image-merge-down` függvény a képnek a megadott réteget és az alatta lévő első látható réteget vonja össze. A `CLIP-TO-BOTTOM-LAYER` paraméter azt adja meg, hogy a az eredményül kapott réteget olyan méretűre kell állítani, mint a legalsó ré-

teg. Ugyanezeket a lépéseket végzi el a program még 90° -al, 45° -al és 30° -al. Ha kész van a forgatásokkal előállított alakzat, akkor következik annak szépítése.

```
(plug-in-autocrop-layer RUN-NONINTERACTIVE image textfs)
(set! textfs-width (+ (car(gimp-drawable-width textfs)) 100))
(set! textfs-height (+ (car(gimp-drawable-height textfs)) 100))
```

A plug-in-autocrop-layer körbevágja az aktív réteget, vagyis azt, amelyiken az alakzatunk van. A textfs-width és a textfs-height változóknak átadjuk a textfs réteg 100x100 pixellel növelt méretét.

```
(gimp-layer-resize-to-image-size textfs)

(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width 2) ←
(/ textfs-width 2)) 18)
(- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) (+ ←
textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)
```

A textfs méretét visszaálltjuk a képpel megegyező méretre, majd ellipszis kijelölést alkalmazunk a képen. Ehhez meg kell adnunk az ellipszist magába foglaló téglalap bal felső sarkának koordinátáit, és magának a téglalapnak a szélességét és magasságát. Ezt az ellipszis kijelölést a plug-in-sel2path függvény segítségével görbévé alakítjuk.

```
(gimp-context-set-brush-size 22)
(gimp-edit-stroke textfs)
```

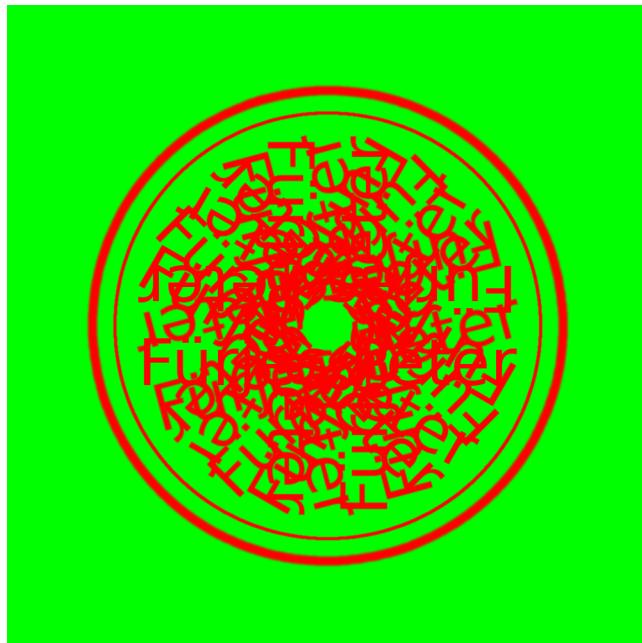
A létrehozott görbe vonalvastagságát a gimp-context-set-brush-size függvénytel állítjuk be. A gimp-edit-stroke pedig a textfs rétegen kijelült területet, vagyis a görbét az előtér színére színezi. Ezekkel az utasításokkal létrehoztuk a külső gyűrűt, ugyanígy hozzuk létre a belsőt is.

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

(gimp-image-select-ellipse image CHANNEL-OP-REPLACE (- (- (/ width 2) ←
(/ textfs-width 2)) 18)
(- (- (/ height 2) (/ textfs-height 2)) 18) (+ textfs-width 36) (+ ←
textfs-height 36))
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)

(gimp-context-set-brush-size 8)
(gimp-edit-stroke textfs)
```

Anyia különbség, hogy csökkentjük a textfs-width és a textfs-height változók értékét 70-nel. Egy másik különbség, hogy a vonalvastagsága is vékonyabb lesz a belső gyűrűnek. Ha nem alkalmaznánk gradienst, az a szín átmenetet, akkor itt véget is érhetne a program, a következő eredménnyel:



9.7. ábra. Alap mandala

Ennél még befolyásolhatjuk mandala színét a `text2_color` néven átadott paraméter segítségével, ahogy azt a feladat elején is említettem. De ezzel nem elégünk meg, nézzük, hogyan lehet tovább javítani a kinézetét.

Következő lépésként a színátmenetet oldjuk meg, mellyel sokkal látványosabb mandalákat lehet készíteni.

```
(set! gradient-layer (car (gimp-layer-new image width height  
RGB-IMAGE "gradient" 100 LAYER-MODE-NORMAL-LEGACY)))
```

Első lépésként létrehozunk egy új réteget, melynek szélessége és magassága megyegyezik a kép méretével, ez is RGB színeket használ, és teljesen átlátszó.

```
(gimp-image-insert-layer image gradient-layer 0 -1)  
(gimp-image-select-item image CHANNEL-OP-REPLACE textfs)  
(gimp-context-set-gradient gradient)  
(gimp-edit-blend gradient-layer BLEND-CUSTOM LAYER-MODE-NORMAL-LEGACY ←  
GRADIENT-RADIAL 100 0 REPEAT-NONE FALSE TRUE 5 .1 TRUE 500 500 (+ (+ ←  
500 (/ textfs-width 2)) 8) 500)
```

A szokásos módon az új réteget, amit a `gradient-layer` változóban tárolunk, a kép utolsó aktív rétege fölé. A `gimp-image-select-item` függvény a megadott réteget kijelöléssé alakítja, vagy annak az alfa csatornáját. A kijelölt területre a `gimp-edit-blend` függvény segítségével tudunk színátmenetet beállítani, melynek paraméteréül a felhasználó által megadott grádiens nevét, a színezés módját, a színátmenet típusát és azt a területet, ahol színezni szeretnénk, kell megadni. Rengeteg más paramétere van, de ezek a legfontosabbak.

```
(plug-in-sel2path RUN-NONINTERACTIVE image textfs)
```

A kijelölést görbévé alakítjuk a szokásos módon.

```
(set! textfs (car (gimp-text-layer-new image text2 font fontsize PIXELS ←
    )))
(gimp-image-insert-layer image textfs 0 -1)
(gimp-layer-set-offsets textfs (- (/ width 2) (/ text2-width 2)) (- (/ ←
    height 2) (/ text2-height 2)))
```

A második szöveget, amit meg tudunk adni, ezekkel a sorokkal helyezzük al a kép közepére.

```
(gimp-selection-none image)

(gimp-display-new image)
(gimp-image-clean-all image)
)
```

Végezetül pedig megszüntetjük a kijelölést, a képet új ablakban jelenítjük meg a GIMP-ben, ehhez a `gimp-display-new` függvényt használjuk. A `gimp-image-clean-all` függvénytel pedig eltünítjük a képből a szükségtelen információkat. Ahhoz, hogy alaértelemzett értékeket állítsunk be a felhasználóknak a következő utsasításokat kell elvégezni:

```
(script-fu-register "script-fu-bhax-mandala"
  "Mandala9"
  "Creates a mandala from a text box."
  "Norbert Bátfai"
  "Copyright 2019, Norbert Bátfai"
  "January 9, 2019"
  ""
  SF-STRING      "Text"        "Bátf41 Haxor"
  SF-STRING      "Text2"       "BHAX"
  SF-FONT        "Font"        "Sans"
  SF-ADJUSTMENT   "Font size"   '(100 1 1000 1 10 0 1)
  SF-VALUE        "Width"       "1000"
  SF-VALUE        "Height"      "1000"
  SF-COLOR        "Color"       '(255 0 0)
  SF-GRADIENT     "Gradient"    "Deep Sea"
)
```

Ebben a részben adhatunk nevet, magyarázó szöveget a scriptünknek. Beállíthatjuk, hogy a script megnyitásakor mik legyenek az alap értékek, amiből a felhasználó kiindulhat. Azt pedig, hogy honnan legyen elérhető a script a GIMP menüből a következőképpen tudjuk meghatározni:

```
(script-fu-menu-register "script-fu-bhax-mandala"
  "<Image>/File/Create/BHAX"
)
```

Végezetül pedig lássuk, hogy milyen képet tudunk készíteni ezzel a scripttel.



9.8. ábra. Végső mandala

10. fejezet

Helló, Kernighan!

10.1. Pici könyv

Alapfogalmak

Ha a számítógépek programozási nyelveiről beszélünk, akkor fontos említeni róluk, hogy milyen szintekre tudjuk őket osztani. Az első a gépi nyelv, mely lényegében 0-kból és 1-kból álló bináris kód. Ezt követi az assembly szintű nyelv, amely már egy kicsit közelebb van az emberi nyelvekhez, de még alacsony szintű programozási nyelvnek minősül. Legvégül pedig tegyük emlékeztetni a könyv fő témájáról a magas szintű programozási nyelvekről. Ezek már közel állnak az emberek által is értelmezhető nyelvekhez, főleg az angolra épülneké.

A magas szintű programozási nyelven írt programot forrásprogramnak nevezünk. Ennek előállításához be kell tartani bizonyos az adott nyelvre jellemző formai, szintaktikai szabályokat és a tartalmi, szemantikai szabályokat.

A processzorok saját gépi nyelvvel rendelkeznek, és csak az ezen a nyelven írt programokat képesek végrehajtani. Tehát a forráskódokat át kell alakítani a gép által értelmezhető gépi kódra. Erre két analógia létezik, az egyik a fordítóprogramos megoldás, a másik az interpreteres.

A fordítóprogram egyetlen egységeként kezeli a forrást, és lexikai, szintaktikai, szemantikai elemzést hajt végre, majd legenerálja a gépi kódot. Ez még nem futtatható, ebből a kapcsolatszerkesztő állít elő futtatható programot, melyet a betöltő behelyez a tárba, és a futtató rendszer felügyeli a futását. Bizonyos esetekben lehetőség van arra, hogy nem nyelvi elemeket használunk egy forrásprogramban, de ilyenkor szükség van egy előfordítónak.

Az interpreteres megoldás nem készít tárgykódot, viszont a fentebb említett 3 elemzést végrehajtja. Utasításonként sorra veszi a forrásprogramot, értelmezi, és végrehajtja. tehát rögtön kapjuk meg eredményt. Bizonyos nyelvek esetén mind az interpreteres, mind a fordítóprogramos megoldást alkalmazzák.

Minden programozási nyelvhez tartozik egy hivatkozási nyelv, mely a szemantikai és szintaktikai szabályokat határozza meg. Emellett léteznek még implementációk. Az egyes rendszereken több fordítóprogram és interpreter létezik, és ennek következtében az implementációk nem kompatibilisek egymással, ez pedig meggátolja a programok tökéletes hordozását a platformok között.

A programozó dolgának megkönnyítése érdekében létrejöttek az integrált grafikus felületek(IDE), melyek egy csomagban tartalmaznak minden szükséges eszközt a programok megírásához, és futtatásához.

A programnyelvek osztályozása

Összeségében 2 fő csoportra oszthatjuk a programozási nyelvezetet: az imperatív nyelvezetekre és a deklaratív nyelvezetekre.

Az imperatív nyelvezek algoritmikus nyelvezek, tehát a programozó maga kódolja le az algoritmust, amit a processzor majd végrehajt. A program utasításokból épül fel. Legfőbb eszköz a változó, mely segítségével el tudunk érni tárterületet, és tudjuk módosítani annak tartalmát. Jelen esetben az algoritmusok teszik ezt. Az imperatív nyelvezeknek 2 alcsoportja van: az Eljárásorientált nyelvezek és az Objektumorientált nyelvezek.

Ezzel szemben a deklaratív nyelvezek nem algoritmikusak, a programozó csak felvázolja a problémát, és a nyelvi implementációkban be van építve a megoldás megkeresésének módja. Memóriaműveletekre nincs lehetőség, vagy nagyon korlátozott módon. Ennek is 2 alcsoportja van: a Funkcionális nyelvezek és a Logikai nyelvezek.

Léteznek olyan nyelvezek is, amelyek nem sorolhatóak be egyik osztályba sem, nincs egységes jellemzőjük. Általában az imperatív nyelv valamelyik tulajdonságát tagadják.

Karakterkészlet

A programok legkisebb építőelemei a karakterek. A forrásszöveg írásakor alapvető a karakterkészlet ismerete, mivel ezek jelenhetnek meg a kódban, és belőlük állítható elő összetett nyelvi elem. Az eljárásorientált nyelvezek lexikális, szintaktikai egységek, utasítások, programegységek, fordítási egységek és végül maga a program épül fel a nyelvi elemekből.

Minden nyelv definiálja a saját karakterkészletét, de a legtöbb programnyelv a karaktereket 3 kategóriába sorolja: betűk, számok, egyéb karakterek. A legtöbb nyelvben a betűk az angol ABC 26 betűjét tartalmazzák, viszont az már nyelv függő, hogy értelmezve van-e a nagy és kis betű, vagy különbséget tez a kettő között. Egyes újabb nyelvezek már nemzetiségi betűket is tartalmaznak, ennek következtében "magyarul" is programozhatunk. A számokat nagyjából mindegyik nyelv hasonlóan kezeli, a decimális számjegyeket veszik számokjegy karaktereknek. Egyéb karakterek közé soroljuk a műveleti jeleket (+, -, *, /), elhatároló jeleket ([,], .., {, }, ;), írásjeleket és speciális karaktereket. A hivatkozási nyelv és az implementáció karakterkészlete egyes esetekben eltérő lehet.

Lexikális egységek

A lexikális egységek azok, amelyeket a fordítóprogram felismer, és tokenizál. Lehetnek többkarakteres szimbólumok, szimbolikus nevek, címkek, megjegyzések, literálok.

Többkarakteres szimbólumoknak a nyelv tulajdonít jelentést, gyakran operátorok vagy elhatárolók, mint C-ben a ++, --, &&.

Szimbolikus neveket 3 csoportra bontjuk, az azonosítókra, a kulcsszóra és a standard azonosítóra. Az azonosító betűvel kezdődik és betűvel vagy számmal folytatódik, segítségével programozói eszközöket nevezünk meg, és ezzel a névvel tudunk rájuk hivatkozni a programon belül. A kulcsszónak az adott nyelv tulajdonít jelentést, ez nem változtatható meg. C-ben iylenek az if, for, case, break. A standard azonosító pedi egy kicsit mind a kettő, mivel ezt is az adott nyelv látja el jelentéssel, de a programozó megváltoztathatja. C-ben sztenderd azonosítónak számít a NULL.

A címkeknek főleg az eljárásorientált nyelvezekben van nagy jelentősége, mivel a segítségével képesek vagyunk megjelölni a végrehajtó utasításokat, így tudunk rájuk hivatkozni a program más részein.

A megjegyzések szerepe az évek során eléggé megkopott, de természetesen minden nyelv biztosít lehetőséget kommentek hozzáadásához. A comment lényegében egy karakterszorozat, amelyet a fordító ignorál. Több implementáció létezik a kommentek alkalmazására, lehet teljes komment sorokat elhelyezni, vagy

sor végi kommenteket, de lehetőség van több soros, egybefüggő kommentek elhelyezésére is. A C-ben mind a 3-ra van lehetőség. (Egysoros komment://, többsoros: /* */)

Végezetül tegyük említést a literálokról is, mely programozási eszközök segítségével fix értékek építhetők be a programba. A literáloknak van típusuk és értékük. A literálrendszernek egyes programozási nyelvekben eltérő lehet.

Adattípusok

Az adattípus egy absztrakt programozási eszköz, mely kronikát eszközök komponenseként jelenik meg. Mindig rendelkezik egy névvel, ami azonosító. Nem minden nyelv ismeri ezt, azt eszközt, főleg az eljárás-orientált nyelvekben van fontos szerepe. Az adattípust a tartomány, a műveletek és a reprezentáció határozzák meg.

A tartomány az adattípus által felvehető értékek halmazát adja meg. Mindegyikhez hozzáartoznak műveleteket, amit az adott típusossal végre tudunk hajtani. A reprezentáció pedig a beslő ábrázolást írja le, vagyis a tárban a típus hány bájtra képződik.

Bizonyos nyelvek lehetőséget biztosítanak arra, hogy a programozó saját adattípusokat adjon meg, mindezt általában a beépített típusok segítségével. Meg tudjuk adni a tartományt, a műveleteket, és a reprezentációt is. Viszont ezt nagyon kevés nyelv támogatja.

Az adattípusokat 2 nagy csoportra osztjuk: skalár adattípus és összetett adattípus.

A **saklár** típus tartománya atomi értékeket tartmaz. Skalár típus a minden nyelvben létező egész, melynek belső ábrázolása fixpontos. Vannak még a valós típusok, ezek reprezentációja lebegőpontos, tartománya implementációfüggő. Ezt a két típust együtt numerikus típusoknak nevezzük, melyeken numerikus és összehasonlító műveletek hajthatók végre.

A karakteres típus elemei karakterek, míg karakterlánc típusai karakterszorozatok. Ezeken szöveges és hasonlító műveleteket hajthatunk végre. Ábrázolásuk karakteres.

Bizonyos nyelvek támogatják a logikia típusokat is, melyek értéke igaz vagy hamis lehet, logikai és hasonlító műveleteket tudunk velük végrehajtani.

Érdekességeként meg lehet említeni a felsorolás típust vagy a sorszámozott típust, de ezek kevésbé elterjedtek, csak néhány nyelvben fordulnak elő.

Az **összetett típusok** közül a két legfontosabb a tömb és a rekord. A tömb homogén, statikus összetett típus, meghatározzák a dimenzióinak száma, az indexkészletének típusa és tartománya, és az elemeknek a típusa. Léteznek több dimenziós tömbök, melyeket lehet oszlopfolytonosan vagy sorfolytonosan ábrázolni. A tömb elemeit indexek segítségével tudjuk bejárni.

A rekord típus heterogén, tartományában különböző típusok szerepelhetnek. Az egyes elemeket mezőknek nevezzük, melyeknek van saját neve és típusa. A mezőneveket minősített nevekkel tudjuk ellátni, ezzel segítve az azonos nevű mezők megkülönböztetését.

A **mutató típus** egyszerű típus, de érdekessége az, hogy tartomány tárcímekből áll. Segítségével indirekt módon el tudjuk érni a mutatott elem értékét. Fontos a szerepe az absztrakt adatszerkezetek szétszort reprezentációjában.

Nevesített konstans

3 komponensből áll: név, típus, érték. Mindig deklarálni kell, mely során megkapja az értékét, és azt nem lehet módosítani a futás folyamán. A nevével tudunk hivatkozni az értékkomponensére. Ennek köszönhetően, elég a deklarációtól, módosítani az értéket, és nem kell minden egyes előfordulásnál. C-ben ezt a következőképpen használjuk:

```
# define név literál
```

Ez az előfordítónak szól, és a név minden előfordulását helyettesíti a literállal.

Változó

4 komponensből áll: név, attribútum, cím, érték. A név egy azonosító, a programban a változó minden a nevével jelenik meg. Attribútumnak olyan jellemzőket nevezünk, amik befolyásolják a változó viselkedést. Attribútum például a változó típusa, mely a deklaráció során rendelődik a változóhoz.

Létezik explicit deklaráció, implicit deklaráció és automatikus deklaráció. Az explicit deklaráció a változó nevéhez rendel attribútumot, míg az implicit betűkhöz. Pl. megoldható, hogy az azonos kezdőbetűvel rendelkező változók azonos attribútumúak lesznek. Az automatikus pedig azt jelenti, hogy a fordítóprogram rendel attribútumot a változóhoz, ha az még nem volt deklarálva.

A változó címkomponense határozza meg a változó helyét a tárban. Egy változó élettartamának a futási idő zon részét nevezzük, ameddig a változó rendelkezik címkomponenssel. A címet lehet statikus vagy dinamikus tárkiosztással rendelni változóhoz. Létezik még programozó által vezérelt tárkiosztás is, a futási időben a programozó rendel címkomponenst a változóhoz.

A változó értékkomponense bitkombinációként jelenik meg a címen, melynek felépítését a típus határozza meg. A változónak értéket értékadó utasítással tudunk adni, mely a C-ben így néz ki:

```
változónév = kifejezés;
```

Alapelemek C-ben

A C tipusrendszere 3 részre bontható, vannak az aritmetikai típusok, a származtatott típusok és a void típus. Az aritmetikai típusok közé tartoznak az egészek, a karakter, a felsorolás, a valósok. A származtatott típusok a tömb, a függvény mutató, struktúra, union.

Az aritmetikai típusok egyszerű, mig a származtatottak összetett típusok.. Az előbbivel aritmetikai műveletek végezhetőek. Logikai típus nincs, az int 0 felel meg hamisnak, az összes többi érték igaz értékkel rendelkezik. A logikai műveletek int 1-et adnak vissza igaz esetén. A egészek és karakterek előtt használható unsigned/signed típusminősítők segítségével tudjuk beállítani vagy letiltani az előjeles ábrázolást. A struktúra fix szerkezetű rekord, ezzel szemben a union, csak változó részt tartalmazó rekord. A void típus tartomány üres, tehát nincs reprezentációja, sem műveletei.

Utasítások

Az utasítások segítségével tudjuk megadni az algoritmusok egyes lépéseit, és a fordítóprogramok ezek segítségével hozzák létre a tárgykódot. 2 csoportra oszthatjuk őket: deklarációs és végrehajtó utasítások. A deklarációs utasítások kifejezetten a fordítóprogramnak szólnak, mögöttük nem áll tárgykód. Összességeben a tárgykódot befolyásolják, de nem fordulnak le. A végrehajtó utasítások összességeből áll Ezekből többféle létezik, melyeket most sorravezünk.

Az értékadó utasítások beállítják vagy módosítják az egyes változók értékét.

Az üres utasítások az eljárásorientált nyelvekre jellemzők. De vannak olyan nyelvek, ahol hasunálatuk nélkülözhettek. Hatására a processzor egy üres gépi utasítást hajt végre.

Régebbi programnyelvek előszeretettel alkazmaták az **ugró** utasítást. De ezt manapság már nem használják, mivel felelőtlen alkalmazása értelmezhetetlen kódot eredményez. Alakja: GOTO címke. Lényegében az adott címkével ellátott utasításra/utasítás csoportra adhatjuk át a vezérlést.

Az **elágaztató** utasítások két csoportra oszthatjuk, a kétirányú és a többirányú elágaztatásra. A kétirányú alatt igaz/hamis feltételt kell érteni. Van egy fejrész, abban tároljuk a feltételt, ami teljesülése esetén végrehajtjuk hozzá tartozó utasításokat. Lehet még else ágat is használni, ahova akkor igrik a vezérlés, ha a feltétel nem teljesül. Ha egyszerre több utasítást kakrunk végrehajtani, akkor a kapcsos zárójellel tudjuk összekapcsolni őket, mely így egy utasításnak számít. Alakja: `if feltetel then tevekenyseg [else tevekenyseg]` Ha van else ág is, akkor hosszú IF-utasításról beszélünk, ellenkező esetben rövidről. Lehetőségünk van egymásba ágyazni több IF-utasítást, de ekkor értelmezésbeli problémák adódnak, mivel nem egyértelmű, hogy az else ág, melyik if-hez tartozik. Általában a belülről kifelé haladó kiértékelés az elterjedt, szóval minden az else-hez "legközelebbi" if-hez tartozik.

Ezzel szemben a többirányú utasítás esetén több egymást kizáró tevékenység közül egyet választunk ki, ha a fejrészben lévő kifejezés megegyezik az konsatnsok valamelyikével. A konstansokhoz rendelünk tevékenységeket, melyeket végrehajtuk az előbb említett esetben. Ez C-ben általában a következő alakú: `switch kifejezés case1: tevekenyseg default: tevekenyseg` Több case-t is használhatunk, és a végére, ha egyik case se teljesülne, akkor használhatunk default tevékenységet, amit végrehajtunk ilyenkor.

Vannak még **ciklusvezérlő** utasítások, melyek segítségével utasításokat tudunk megismételni. Általános felépítése fejből, meagból és végből áll. A mag tartalmazza a végrehajtandó utasításokat, a fej vagy a vég pedig a ismétlést meghatározó információk helye. Két véglet az üres ciklus és a végtelen ciklus. Míg az előbbi nem hajt végre semmit, addig az utóbbi végtelenséggel ismétlődik. Ezektől eltekintve létezik feltételes, előírt lépésszámú és felsorolásos.

A feltételes ciklus a feltétel igaz/hamis volta alapján hajt végre utasításokat. A feltétel lehet a fejrészben, vagy a végrészben. Az előbbi eset C-ben a `while`, mely esetén addig oismétlődik a cílus, ameddig a feltétel igaz, utána befejeződik. Utóbbi eset a `do while`, mely egyszer mindenkorban végrahajtja a ciklusmagot, és utána teszeli a feltételt, ha hamis, akkor kilépünk a cílusból.

Az előírt lépésszámú ciklus, lényegében a ciklus fejrészében megadott számban ismétli az utasításokat. Ehhez szükség van egy ciklus változóra, melyet addig léptetünk, ameddig a feltétel igaz, majd kiléünk. Egyes nyelvek engedélyezik a cílusváltozó deklarálását a fejben, másoknál (C-ben), a ciklus előtt kell deklarálni. A lépésköz is tudjuk állítnai, illetve azt is, hogy csökkenően, vagy növekvően haladjunk a ciklusban. Ezt a ciklusfajtát nevezzük `for` cíklusnak.

Felsorolásos ciklus az előbbi ciklus egy általánosítása. A cílusváltozó és az értékek megadása a fejben törrténik. A cílus minden egyes értékre lefut. A cílusváltozó típusa szabadon választható, viszont ez a ciklus fajta nem lehet se üres, se végtelen.

Végtelenb ciklus esetén sem a fej, sem a vég nem tartalmaz információkat arról, hogy mikor kéne befejeződni a cílusnak. Ezeknél általában a magban adunk meg valamilyen utasítást, amely kiugratja a vezérlést a ciklusból. Használata eseményvezérelt alkalmazásoknál jön jól.

Az összetett ciklusok pedig az eddig felsorolt cílusokból épülnek fel. A cílusfejükben tetszőleges mennyiségű információt tárolhatunk el az ismétlődésre vonatkozóan.

Ahogy már a végtelen cíklusnál említettem, vannak utasítások, amik segítségével a cílusok működését befolyásolhatjuk. Ezeket nevezzük **cilusszervező** utasításoknak.

Programok szerkezete

A program forrását programegységekre tudjuk osztani. Vannak nyelvek, ahol ezek teljesen függetlenek egymástól, másoknál egyetlen egységeként kell lefordítani az egészet, de léteznek olyanok is, amelyek ezt a kettőt kombinálják. Tehát bizonyos programegységek összetettek, bizonyosak pedig szimplák. Az eljárásortorientált nyelvek 4 programegységből állnak. Van egy alprogram, egy blokk egy csomag és egy taszk.

Az alprogramnak az a lényege, hogyha egy programrész többször ismétlődik, akkor ezt külön megírjuk, és csak hivatkozunk rá. Két fő típusa van az alprogramoknak: eljárás és függvény. Az eljárás a paraméterként átadott elemekkel végrahajt valamilyen utasítást, viszont nem tér vissza semmilyen értékkel, ellentétben a függvénnyel. A függvénynél a visszatérési érték típusát a függvény nevével együtt adjuk meg, ha nem adunk meg visszatérési értéket a függvény végén, akkor szintaktikai hibát kapunk. Ha az egyes programegységek újabb és újabb programegységet hívnak meg, akkor alakul ki a hívási lánc. Bizonyos esetekben egy programegység önmagát is hívhatja, ezt nevezzük rekurzív hívásnak. Bizonyos nyelvek támogatják a törzsben szereplő alprogramokat is, ezeket másodlagos belépései ponttal oldja meg. A blokk egy programegység, mely önállóan nem állhat, csak egy másik programegység belsejében. Fromálisan 3 részből áll: kezdet, törzs, vég. A kezdetet és véget C-ben kapcsoszárójellel jelöljük, nem tartozik hozzá név. A törzsben szerepelhet bármilyen utasítás. A vezérlés vagy szekvenciálisan kerül a blokkra, vagy egy GOTO utasítással mi ugratjuk rá.

Paraméterkiértékelés, paraméterátadás

A paraméterkiértékelés az alprogram híváskor egymáshoz rendeli a formális és aktuális paraméter listát. Formális paraméter listából minden csakis csak egy van, melyet az alprogram specifikációja tartalmaz, viszont aktuálisból annyi létezik, ahány függvényhívás. Mindig a formálishoz rendeljük az aktuális, de ennek megvalósítása többféleképpen lehetséges. Lehet sorrend szerint, tehát elsőt az elsővel és így tovább, vagy név szerint is. Ezt úgy kell elképzelni, hogy az aktuális paraméter listában meg tudjuk adni, hogy a formális paraméter lista melyik eleméhez mit rendelünk. Az is eltérő lehet, hogy hány paramétert kell megadni. Ezt minden a formális paraméter lista határozza meg, vagyis ha fix számú, akkor az aktuálisnak is annyit kell tartalmaznia, vagy lehet kevesebb is, de ez kizárolag érték szerinti paraméterátadás esetén lehetséges. Ha a formális paraméter lista tetszőleges méretű, akkor az aktuális is. Nyelvenként eltérő annak a kezelése, ha az aktuális paraméter lista elemei nem egyeznek meg a formálisan lévő paraméterek típusával. Ilyenkor vagy hibát kapunk, vagy egyes nyelveknél a típuskényszerítést alkalmazzák. A paraméterátadásának 6 módja van. Az első, a már említett érték szerinti paraméterátadás. Ebben az esetben a formális paramétereknek van címkomponensük az alprogram területén. Az átadot érték lemásolódik, ezzel dolgozik tovább az alprogram. Ennek a gyengesége a másolás, mely bizonyos adatszerkezetek esetén sok időbe telhet, és sok helyet foglal. A cím szerinti paraméterátadás esetén a formális paramétereknek nincs külön címkomponense, hanem az alprogram a hívó területén dolgozik. Ebben az esetben az alprogram olvashat és írhat is értékeket a hívó területén. Az eredmény szerinti paraméterátadás abban tér el ettől, hogy az alprogram a saját területén dolgozik, míg be nem fejezi a feladatát, ezután pedig a hívó területére kiírja az eredményt. Az érték-eredmény szerinti átadás ezt gondolja tovább. Itt már azv értéket is átadjuk a cím mellett, az alprogram az átadott értékkel végrahajtódik a saját területén, majd pedig az eredményt kiírja a hívó memóriaterületére. Létezik még név szerinti átadás is, ahol a paraméter egy szimbólum. Ekkor az alprogram formális paraméterének minden előfordulása átíródik a megadott szimbólumra, és az alprogram végrehajtódik. A szöveg szerinti abban különbözik, hogy az átirás csak az alprogram futásának megkezdése után, az első előfordulástól kezdve hajtódi végre.

Hatóskör

A hatóskör, vagy láthatóság a programban előforduló nevekhez kapcsolódó fogalom. Egy név hatókörének azt programrész nevezzük, ahol ugyan azt a programozási eszközöt hivatkozza. A programegységekben deklarált nevek lokális nevek, azokon kívül deklaráltak pedig szabad nevek. A hatókörkezelésnek két ismert megvalósítása van, az egyik a statikus, a másik a dinamikus. Az eljáráorientált nyelvek a statikus verziót használják. A statikus megoldás lényege, hogy minden név hatóköre az adott programegységig terjed, ahol deklarálták. Ebbe bele tartoznak a beágyazott programegységek is. Fontos kikötés, hogyha az egyik beágyazott programegység úradeklarálja a nevet, akkor már nem érhető el ott. A dinamikus hatókörkezelés

is hasonló ehhez, azzal a különbéggel, hogy a név hatóköré kiterjed azokra a programegységekre amik a saját programegysége után vannak a hívási láncban, hacsak ott nem deklarálják újra.

INPUT/OUTPUT

Az I/O az, amiben a legnagyobb divergencia figyelhető meg az egyes programnyelvek között. Ez a programoknyelveknek azon része, melynek segítségével a perifériákról tudunk beolvasni, és perifériára tudunk kiírni állományokat. Állományokon belül különbéget teszünk a logikai állományok és a fizika állományok között. A logikai állomány a programban jelenik, ezzel szemben a fizikai állomány valamelyen periférián megjelenő adatállomány. Egy állomány lehet input, output és input-output. A különbé aból fakad, hogy az első fajtából csak olvasni tudunk, a másodikba csak írni, viszont a harmadik lehetővé teszi mindenkitőt. Tehát a tárban és a perifériákon valamelyen módon ábrázolva vannak az állományok. Annak a függvényében, hogy történik-e konverzió a tár és a perifériák közötti adatmozgás során, két módot különböztetünk meg: a folyamatos és a bináris átviteli módot. A folyamatos átviteli mód esetén a tárban és a periférián eltérő az adatok reprezentációja. Ennek következtében konverziót kell alkamazni. Háromféleképpen alakult ki ennek a konverziónak a megvalósítása az egyes nyelvekben. A formátumos módban adatátvitel esetén minden adathoz meg kell adni a karakterszámot, és azok típusát. A szerkesztett módban átvitel-nél maszkot kell megadni, ezen keresztül tudjuk beállítani a fromátumot, és a karakterek típusát is. A listázott átvitel pedig a karaktersorozatban tárolj a töredék karaktereket, a karaktertípus expliciten nem kerül meghatározásra. A bináris átvitelnél pedig minden a periférián és a tárolón ugyan úgy jelennek meg az adatok, ezért nincs szükség konverzióra, ebben az esetben rekordokként történik az adatok továbbítása. Ahhoz, hogy állományokat tudunk kezelní szükséges néhány lépést végrehajtanunk a program írása során. Az első a deklaráció. Itt a logikai állományt deklaráljuk, hiszen a programunkban ezzel dolgozunk, nem a fizikai állománnyal, legalábbis nem közvetlenül. Ahogy már megszokhattuk a deklaráció során nevet adunk, és jelen esetben szükség van attribútumokra is. Az összerendelés során a logikai állományhoz hozzárendelünk egy fiziki állományt. Ezzel érjük el azt, hogy miközben a logikai állománnyal dolgozunk, aközben lényegében a fizikaival. Ha megtörtént az összerendelés, akkor megnyitjuk az állományt. Ezután eddig történhet az adatok feldolgozása, mely alatt ríthatunk írást vagy olvasást. Végezetül pedig ba kell zární az állományt. Ez output és input-output állományokra kötelező, input állományok esetén pedig csak javasolt. De ezek elmulasztása sem okoz halászatot, mert a megnyitott állományok automatikusan bezáródnak a program futásának befejezével. Az imént említett módszeren kívül egyes nyelvek lehetőséget biztosítanak arra, hogy nem állományokkal dolgozunk, hanem az írást-olvasást közvetlenül a perifériával végezzük. Ennek a neve az implicit állomány. Ilyen állomány példáiul a sztenderd bemenet, és a sztenderd kimenet. Ezekhez pedig lehetőségünk van hozzárendelni más állományokat. A Linux terminálban például lehetőségünk van átfirányítani a program kimenetét egy fájlba, vagy a bemenetet egy fájlból olvasni. A C nyelvben az I/O műveletekhez könyvtári függvényekre van szükségük. Átviteli módok közül mind a folyamatos, mind a bináris fajtát ismeri. Írás és olvasás során a mnimálisan egy karaktert, egy bájtot tudunk kezelní.

KIVÉTELKEZELÉS

Az egyes programnyelvek lehetőséget biztosítanak a programozónak, hogy a megszakítások kezelését átvegye az operációs rendszertől, és program szintjén oldja meg. A kivételkezelő egy programrész, mely akkor fut le, ha egy kivétel bekövetkezik. Lehetőség van arra, hogy egyes kivételeket figyelmen kívül hagyjunk, tehát a kivételek figyelését letiltathatjuk. A kivételek felépítése 2 elemből áll: névből, kódból.

JAVA KIVÉTELKEZELÉSE

Egy Java program a futása során módszereket hajt végre. Ha valamelyik módszer esetén speciális esemény bekövetkezik, akkor jön létre a kivétel-objektum. A módszer eldobja a kivételt, és a Java virtuális gép

foglalkozik vele tovább. A JVM feladata, hogy megtalálja a megfelelő kivételkezelést. Megfelelőnek akkor hívjuk, ha a kivétel típusával megyegyezik a kivételkezelő típusa vagy esetleg a kivételkezelő típusa őse a kivétel típusának. Itt megjelenik a Java nyelv öröklődés tulajdonsága, mely jól mutatja, hogy ez a nyelv is objektumorientált. Egy kivételkezelőnek fontos tulajdonsága a láthatósága, melyet saját magának definiál. A kivételkezelőre úgy gonolhatunk, mint egy blokkra. A kódban bárhol elhelyezhetők, és tetszőlegesen egymásba ágyazhatóak. Ilyen esetben a JVM a blokk belséjéből kifelé haladva, keresi a szükséges kivételkezelést, ha megtalálja, a vezérlést átadja, lefut a kivételkezelő, majd pedig a program futása onnan folytatódik. Javaban vannak ellenőrzött és nem ellenőrzött kivételek. Általánosan elmondható, hogy érdekes ellenőrzött kivételeket használni. Nem ellenőrzött kivételek csak akkor alkalmaznak a programozók, ha túl sok kódsort eredményezne, vagy képtelen kezelni. Azokat a kivételeket, amik egy módszer futása során keletkezhetnek, de nem ellenőrizzük, a módszer fejében fel kell sorolni. A hibát a Throw kulcsszóval tudunk dobni. A catch kulcsszó jelzi a kivételkezelést, mely a hibát "elkapja". A kivételkezelés Java-ban így néz ki:

```
TRY
    {utasítások}
CATCH (típus változónév)
    {utasítások}
[CATCH (típus változónév) {Utasítások}] ...
[FINALLY {Utasítások}]
```

A TRY blokkban dobjuk az esetleges kivételeket, amelyeket a CATCH blokkban, blokkokban kezeljük. Ha JVM nem talál megfelelő CATCH blokkot, akkor fut le a FINALLY blokk. Fontos tulajdonsága ennek a blokknak, hogy mindenkorban lefut, akkor is, ha nem volt kivétel.

10.2. K&R könyv

Vezérlési szerkezetek

A C nyelvben létező vezérlésátadó utasítások segítségével vagyunk képesek meghatározni a számítások sorrendjét. Lássuk a C leggyakoribb vezérlésátadó utasításait.

Elsőként említsük a **utasításokat és blokkokat**. Az egyes kifejezések akkor válnak utasítássá, ha utánuk rakunk egy pontovesszőt, ezzel jelezük az utasítás végét. Képesek vagyunk az utasításokat blokkba foglalni, melyet a {}-jel segítségével tudjuk megtenni. Ezt a jelülést használjuk a függvény definíciójánál, a for, while, if utasítás által végrehajtott utasítások csoportosításra. Fontos, hogy ha kapcsoszárójelet használunk, akkor utána tilos pontovesszőt rakni.

```
int a;
for (a = 0; a < 5; a++)
{
    /*utasítások*/
}
```

Ha valamit el szeretnénk dönteni a programunkban, akkor jön nagyon jól az **if-else** utasítás. A fejlécben minden megadunk egy kifejezést, mely alapján eldöntjük, hogy egy utasítás végre legyen-e hajtva, vagy sem. Az else ágra nem feltétlenül van szükség, az akkor hajtódiék végre, ha nem teljesül az if feltétele. Az if numerikus értéket viszgál ezért lehetőségünk van rövidíteni a kifejezést. Pl.:

```
if (kifejezés)
if (kifejezés != 0)
```

A két leírás ekvivalens egymással, az első használata viszont néha nehezen érthetővé teszi a kódot. Az else ággal is lehetnek értelmezésbeli nehézségek, mivel nem minden könnyű az egymásba skatulyázott ifek közül eldönteni, hogy melyikhez tartozik az else ág. C nyelben ez úgy van megoldva, hogy minden a hozzá legközelebbi if-hez tartozik. Az esetleges kétértelműségek elkerülése érdekében érdemes használni a kapcsos zárójeleket, melyel egyértelműen meghatározható az utasítás vége. De ha az if/else csak egy utasítást hajt végre, akkor nincs szükség a kapcsos zárójelre, elég a ; használta.

```
if (kifejezés)
{
    /* több utasítás */
}
if (kifejezés)
    utasítás;
```

Az if-else utasítás szorosan összefügg az **else-if utasítás** használatával. Ennek a segítségével több elágazásos if utasítássorozatot hozhatunk létre. Ha az egyik if teljesül a sorozat többi tagja már nem, ha egyik se teljeül, akkor itt is az else ágra kerül a vezérlés, ami akár el is hagyható, ha a maradék esetben semmit sem kell csinálni.

```
if (1.kifejezés)
    1. utasítás;
else if (2.kifejezés)
    2. utasítás;
else
    3. utasítás;
```

Az else-if utasítás kiváló arra, hogy pár esetet elkülönítsünk, de mi van akkor, ha több tíz darab eset van. Erre találták ki a **switch** utasítást, amely megvizsgálja, hogy a kifejezés megegyezik-e valamelyik esettel, és ahhoz az esethez továbbítja a vezérlést.

```
switch (kifejezés) {
    case '1.lehetőség':
    case '2.lehetőség':
        1. utasítás;
    case '3.lehetőség':
    case '4.lehetőség':
    case '5.lehetőség':
        2. utasítás;
}
```

Amint látod, nem muszály mindegyik esethez külön megadni az utasítást, lehet őket csoportosan megadni, ezzel megkönnyítve a programozó dolgát. A switch magját pedig {} jellel határoljuk. A case-en kívül használhatunk default címkét is, amivel egy alapértelmezett utasítást adhatunk meg, arra az esetre ha egyik eset se teljesülne. Fontos a case-ek között nem fordulhat elő azonos. Ha egy case utasítása végrehajtódiak akkor a vezérlés a következőre ugrik. Ez viszont nem minden szerencsés, ezért break-et kell használni mindegyikesetén, de ha nem is rakunk mindegyikhez, akkor is az utolsó eset után érdemes rakni egy break-et, ezzel biztosítva, hogy a vezérlés nem fog szétesni.

Ha a programozási nyelvek ciklusairól beszélünk van 2 alapvető, amelyik mindegyikben megjelenik, így C-ben is, ezek a **for** és a **while** utasítások. Arra használjuk őket, hogy bizonyos utasításokat ismételjünk, egészen addig, ameddig a fejrészükben lévő kifejezés teljesül.

```
while (kifejezés)
    utasítás
```

A while így néz ki zsintaktikailag, egészen addig hajtja végre az utasítást, ameddig a kifejezés nem nulla. Itt egy kifejezés van, ezzel szemben a for ciklus nyelvtanilag 3 kifejezésből áll.

```
for (kif1; kif2; kif3)
    utasítás

/*ennek a while átírata*/

kif1;
while (kif2)
    utasítás
kif3
```

Amint látod, a két fajta ciklus átírátható egymásba, de bizonyos esetekben az egyik könnyebben használható mint a másik. A for ciklusnál nem muszály minden kifejezést kiírni, sőt el is hahatóak, ekkor végétlen ciklushoz jutunk, amelyből vagy break, vagy return utasítással tudunk kiugrani. A fentebbí példában láttad, hogy a for kifejezései közé ; kellett rakni, de ez nem feltétlenül kötelező, csak akkor más értelemmel rendelkezik.

```
int i, j;
for (i = 0, j = 10; i!=j; ++i, --j)
    pritnf("Még nem egyenlő");
```

Ilyen esetben az i-t és az j-t is léptejük egyszerre. Az i és a j definiálása és az értékük változtatása is egy-egy kifejeznek felel meg.

A for és a while cílusok a kiugrás feltétel teljesülését a cílus elején viszgálja, de bizonyos esetekben jól jöhét a **do-while** utasítás. Lényege az, hogy legalább 1-szer biztos, hogy belépünk a ciklusba, ezután pedig a gép kiértékeli a kifejezést.

```
do
    utasítás
while (kifejezés)
```

Már többször szó esett a ciklusokból való kilépés módjáról, ezt valósítja meg a **break** utasítás. Az utasítás hatására a vezérlés kiugrik a legbelübb ciklusból még az előtt, hogy a kifejezést kiértékeltük volna az újabb iterációban.

Ha van olyan utasítás, ami megállítja a ciklust, akkor lennie kell olyannak a folytatja azt. Na nem pont ilyen értelemben, de hasonlót csinál a **continue** utasítás. Ez azt jelenti, hogy a vezérélés visszatér a ciklus fejéhez, tehát abban az iterációban nem csinálunk semmit, csak továbblépünk. Ezt általában valamilyen feltételhez szokták kötni.

```
int i
for (i = 0; i < 20; ++i)
```

```
{  
    if (i%2 != 0) continue;  
    printf("%d", i);  
}
```

Ebben az esetben csak a páros számokat írjuk ki, a többi számot kihagyjuk.

A függvények esetén a break-hez hasonló a **return** utasítás. Ez a függvény hívójához tér vissza, általában valmilyen értékkel, de előfordulhat, például void típusú függvények esetén, hogy nem ad vissza semmilyen értéket.

```
return;  
return kifejezés;
```

Ahogy említettem a break utasítással mindenkor csak a legbelőző ciklusból tudunk kilépni, de mi van akkor, ha mi az egymásba skatulyázott ciklusokból, ekkor kerül előtérbe a **goto** utasítás. Szintaxisát tekintve:

```
for (...)  
    for (...)  
        if (kifejezés)  
            goto megoldás;  
  
megoldás:  
    tedd hamissá a kifejezést
```

Tehát egy címkére ugrunk, amit a goto utasítás után megnevezünk, és a címkének megadjuk, hogy milyen utasítást hajtson végre. A goto használata viszont abszolút nélkülözhető, és használata nem is javasolt.

Az előző paragrafusban látott megoldás egy **címkézett utasítás**, mely a goto célpontja. Érvényessége arra a függvényre szól, amelyben előfordul.

A létezik még a **nulla** utasítás. Ezt a használhatjuk címkék hordozására összetett utasításokat lezáró } előtt, vagy jelölhet üres ciklusmagot is.

10.3. BME C++ könyv

C++ nem objektum-orientált újdonságai

A C++ lényegében a C programozási nyelv objektum-orientált változata, melynek első változata 1983-ban jelent meg. A nyelv atyjának Bjarne Stroustrup-ot tekintjük, aki AT&T Bell Laboratories-nak dolgozott. Első verzió neve C with class, ezzel uralta az objektum-orientáltságra. A C++ 1998-ban lett szabványosítva, azóta szabványt is kiadtak, a 2003-ast és a 2011-est. A C++ programok érdekessége, hogy többségük lefordul a C fordítókkal, sőt kezdetben az első C++ fordítók C kódot generáltak. Tehát minden C program C++ program, de nem minden C++ program C program.

Ebben a fejezetben pont azokat a különbségeket fogjuk sorra venni, amik C++-ban használhatóak, de a C-ben nem. Itt olyan funkciókra kell gondolni, amik segítettek kijavítani a C nyelv esetleges gyengeségeit.

Az első lényegi különbség a függvényparaméterekben és a visszatérési értékben rejlik. A C-ben, ha nem írtunk paramétert, akkor korláthatlan számú paramétert használhattunk, ezzel szemben a C++-ban ez annyit jelent, hogy nincs paraméter, vagyis mintha a paraméternek void típust adtunk volna meg. Ahhoz, hogy a korláthatlan számú paramétermegadást engedélyezzük C++-ban, a paraméterlistéba ...-t kell beírni.

```
//C-ben
void f()
{
}

//C++-ban
void f(...)
```

Egy másik különbség, hogy a C nyelvben kihagyhatjuk a visszatérési érték típusának meghatározását, azt int-nek értelmezi. Ezzel szemben C++-ban fordítási hibát kapunk, nem szabad kihagyni a visszatérési érték típusának meghatározását.

```
void f(void); //nincs hiba
g(void); //fordítási hiba
```

A másik különbség a main függvényben keresendő. Az argumentum listát hagyhatjuk üresen, vagy használhatjuk az argc, argv[] argumentumokat, mellyel a parancssori argumentumok számát és tömbjét adja át a main-nek. Ez eddig hasíónló a C-hez, de a main végére nem szükséges oda írni a return-t, azt a fordító automatikusan hozzáilleszti a kódhoz.

Sokáig a C++ nyelv sajátja volt a bool típus, mellyel logikai változókat deklarálhatunk, értéke true és false lehet. Ez már a C-ben is elérhető, de előtte csak az int vagy az enum típusokkal fejezhettük ezt ki. Int esetén 0 számít hamisnak és minden más érték igaz.

A C++-ban megjelet a wchar_t, mint beépített típus, mely segítségével Unicode karakter stringeket tudunk reprezentálni. Ez a C-ben is használható, viszont ehhez #include-olni kellett a bizonyos header fájlokat. Az ilyen típusú változókat a következőképpen definiálhatjuk:

```
wchar_t = L 's';
wchar_t* = L "sss";
```

C++-ban a változókat olyan helyeken is deklarálhatjuk, ahova utasításokat is írhatunk. Ilyen például, hogy a for ciklus-ban tudjuk definiálni az indexet, nem kell azt előtte deklarálni. Ettől függetlenül elmondható, hogy a változókat érdemes pont ott deklarálni, ahol a szeretnénk használni, ezzel áttekinthetőbb kódot kapunk. A változó csak a deklarációja után érhető el. A for ciklus fejlécében deklarált változókat nem tudjuk használni a cikluson kívül.

A C nyelvben nem tudunk azonos nevű függvényeket megadni, mivel azonosításként a linker csak a függvény nevét használta. Ennek következtében a programozóknak rengetegszer kellett erőltett függvényneveket kitalálni. Ezt a problémát oldja meg a C++ függvény túlterhelés funkciója, mely annyit tesz, hogy a linker a függvényt a neve és az argumentum listája alapján azonosítja. Tehát létre tudunk hozni azonos nevű függvényeket, ha azok argumentumlistája eltér. Ezt a technikát név felderítésnek nevezzük. Mivel ez a C-ben nem értelmezett ez a funkció, kitaláltak egy módszert arra, hogy a C és C++ közötti kompatibilítást megoldjuk, ehhez a függvény deklaráció előtt kell írni, hogy extern int. Ezzel letiltjuk a név felderítő mechanizmust.

A függvényekkel kapcsolatos másik különbség, hogy a C++ kódban lehetősségünk van alapértelmezett értéket adni az argumentumoknak. Ezzel ki tudjuk küszöbölni azt, hogyha nem adunk meg elég argumentumot a függvényhívásnál.

```
int f( int a, int b = 100, int c = 200) {
```

```
        return a*b-c;
    }
    int main()
{
    int egyik = 10;
    int masik = 20;
    int harmadik = 30;
    f(egyik, masik, harmadik); //érték = 10*20*30
    f(egyik) // érték = 10*100*200
    f() // hiba: nincs elég argumentum
}
```

A C++ egyik legnyagobb újdonsága pedig a referencia típus bevezetése. A C-ből tudjuk, hogy ha egy függvénynek paraméterül adunk egy változót, akkor a azt lemásolja a függvény, és nem módosítja a változó értékét. Ha azt szeretnénk, hogy a függvény módosítson az értéken, ahhoz kell használni a pointert, ami a változó memóriacímére mutat. Ennek segítségével közvetlenül tudjuk átadni a függvénynek a változót, nem másoljuk az értékét. Ennek használata elég nehezen érhetővé teszi a kódot, és rengeteg hibázási lehetőséget rejti magában. Hogy ezt orvosolják, bevezették a referencia típust. A referencia típusú változót & jellel jelöljük. Ez a jel a C-ben egyoperandusú operátor, ami a változó memóriacímét adja vissza. A pointer és a referencia közötti különbség:

```
int x = 5;
int* a = &x;
int& b = x;
```

Míg a pointerek egy memóriacímét kell értékül adni, addig a referenciánál csak meg kell adni, hogy melyik változóra referáljon. Ezután mindegy, hogy a b-t vagy az x-et módosítjuk, mind a kettőn végrehajtódik a változtatás. Érdekesség még, hogy a referencia és a változó memóriacíme megyegyezik, tehát a referencia nem foglal a memóriában területet, ellentétben a mutatóval. A programokban nagyon ritkán használjuk a referenciakat a fentebb látott módon, mivel nehéz lenne olvasni egy olyan kódot, ahol ugyan arra 2 változó is hivatkozik. Ezért ennek a fő használati értéke a cím szerinti paraméterátadásban rejlik, ezt szokták referencia szerinti paraméterátadásnak. A cím szerinti paraméterátadást általában akkor szoktuk használni, ha változtatni akarunk az értéken, ezért a refenciának csak olyan változót adhatunk értékül, ami módosítható.(Létezik konstans típusú referencia is, de ez speciális.) Egy fontos dolog még ebben a témaörben az, hogy lehet-e egy függvény visszatérési értéke pointer vagy referencia. A válasz igen, lehet, de csak korlátozott esetekben, főleg cím szerinti paraméterátadás esetén. Összességében elmondható, hogy nem szabad visszaadni pointert és referenciát lokális változókra, vagy érték szerinti paraméterekre, mivel előfordulhat, hogy érvénytelen memóriacímre fognak hivatkozni.

Objektumok és osztályok

Az objektumorientáltságnak 3 alapelve van. Az egyik az egységbézárás, mely egységet nevezünk osztályoknak. Ezek olyan struktúrák, melyek az adatokat és a velük végrehajtható műveleteket is tárolják. minden osztálynak lehet olyan eleme, amire önéálló egyedként tekinthetünk, ezeket objektumoknak nevezzük. Egy másik fontos alapelve az adatrejtés, tehát nem szabad hagyni, hogy az osztályon kívülről olyan műveleteket hajthassunk végre, mely inkonzisztenssé teszi az osztályt. Ennek fontos szerepe van a komplexitás szempontjából is, mivel ha elrejtünk bizonyos elemeket, akkor csak elég az elérhető elemeket ismerni az osztály használatához. Az objektum orientált programozásban lényeges tényező még az öröklés, mely lehetővé teszi, hogy egy speciálisabb osztály örökölje az általánosabb osztály tulajdonságait. Fontos említeni az

ezzel összefüggő behelyettesítésről is, mely lehetőve teszi, hogy ahol speciálisabb osztályt használunk ott ahol amúgy általánosabb osztályt használtunk.

Az **egységezés** C++-ban a struktúrák már nem csak a tagváltozókat tartalmazhatnak, hanem tagfüggvényeket is. Egy másik különbség, hogy a C-ben a struktúra szerinti változót akartunk létrehozni, akkor muszáj volt kiírni elő a `struct` kulcsszót, vagy meg kellett adni egy `alias-t` amivel tudunk rá hivatkozni. C++-ban viszont már erre nincs szükség, szimplán a struktúra nevével létrehozhatunk változót. A tagváltozókat a `".` és a `"->"` operátorokkal tudjuk elérni. A tagfüggvényeket struktúrán belül kell deklárnai, viszont a definiálásukra 2 módszer is létezik. Az egyik az, hogy struktúrán belül definiáljuk is, vagy a kívül, de ebben az esetben szükség van a `::` hatókör operátorra, mivel előfordulhat, hogy több struktúrának is ugyan olyan nevű tagfüggvénye van. minden tagfüggvénynek van egy láthatatlan paramétere, ami módosítandó struktúrára mutat. Ezt a láthatatlan pointert a `this` szóval tudjuk elérni. A struktúrával egy nagy baj van, minden tagja publikus, tehát a programozónak nincs eszköze ahhoz, hogy szabályozza a struktúra elemeinek a hozzáférését. Ezt hivatott megoldani az osztály, mely minden tagja alaértelmezetten private, de megadhatunk `oylan` elemeket is, amik elérhetők az osztályon kívülről. De ezt implicit módon is kiírhatjuk, `private:` és `public:` címkkel. Az adott osztály használatához egy változót kell deklárnunk, ezt az osztály példányosításának nevezzük, a változót pedig objektumnak.

Az objektumok létrehozásánál felmerül egy kis probléma, mégpedig az, hogy a tagváltozók értékét alapból nem adjuk meg, így az azokkal számoló függvények véletle értéket adnak. Ezt küszöböli ki a konstruktur használata, amivel képesek vagyunk kezdőértéket adni a változóknak, azaz inicializálni őket. Ha nem írunk konstruktort, akkor az alapértelmezett konstruktur hívódik meg, ami nem csinál semmit. Lehetőségünk van több konstruktort is megadni, annak függvényében, hogyan szeretnénk példányosítani. A konstruktur arról ismerszik meg, hogy neve megegyezik az osztály nevével, és csak egyszer hajtódik végre, amikor létrehozzuk az objektumot. Ennek az ellentéte a destruktur, ami az egyes elemek által lefoglalt erőforrásokat szabadítja fel a program futásának végeztevel. Szerepe főleg a dinamikus adattagokat tartalmazó osztályokban jelentős.

A **dinamikus adattagok** memóriakezelését C-ben a `malloc` és `free` függvényekkel oldottuk meg. A `malloc` függvény hátránya, hogy csak a legoglalandó terület méretét tudja, magáról a típusról nem tud semmit, azaz nem tudunk vele objektumot inicializálni. Ehhez a C++ új operátorát kell használni, a `new`-t, ennek az ellentéte a `delete`, mely a `free` megfelelője. A `new` segítségével példányosíthatunk osztályokat, paraméterket adhatunk át, eredményként egy pointerrel tér vissza. Ha tömbnek szeretnénk helyet foglalni, ahhoz a `new []` operátorra lesz szükség. Ennek a használata esetén fontos odafigyelni arra, hogy a felszabadítást a `delete []` operátorral tegyük.

Ha egy objektum tartalmát másolni szeretnénk, akkor van szükségünk a **másoló konstruktorra**. Lényegében egy új objektumot inicializálunk egy már létező objektum alapján. A C++ nyelvben van beépített másoló függvény, ami kiválóan átmásolja az egyik változó tartalmát a másikba, de ez a mi esetünkben hiányoz vezetne. A probléma lényege az, hogyha pointert másoltatunk vele, akkor csak szimplán lemásolja az értékét, vagyis lesz 2 azonos helyre mutató mutatónk. Ezért nekünk kell létrehozni az másolást végző függvényt. A neve, mint a többi konstruktornak, ennek megegyezik az osztály nevével, paraméterként pedig egy objektum referenciát kap.

Ahogy egy korábbi paragrafusban már megbeszéltek, az adatrejtés nagyon fontos tulajdonsága az objektum-orientált programozási nyelveknek. Természetesen vannak speciális megoldások a rejtegett tagok kinyerésére, ezzel kapcsolatban kerül elő a `friend` kulcsszó. A **friend függvények** ugyanolyan jogokkal rendelkeznek, mint az osztály tagfüggvényei, viszont nem kell az osztályhoz tartozniuk. Lehetőleg érdemes ketülni a használatát, de vannak esetek amikor szükség lehet rájuk. A `friend` függvények mellett fontos szót ejteni még a **friend osztályok**-ról is. Ezen osztályok tagfüggvényei teljes hozzáférést kapnak a barát osztály

tagváltozóihoz.

A konstruktorok és destruktorkapcsán szó volt a **tagváltozók inicializálásáról**, most ezt tisztázzuk egy kicsit. Amikor egy objektumot létrehozunk, akkor a konstruktor hívódik meg, ekkor történik meg az inicializálás. Ha utólag értéket adunk, akkor viszont "=" operátor hívódik meg, nem a kontrsuktor. A tagváltozókat az inicializálási listában tudjuk inicializálni, ezt a konstruktor fejléce után írjuk ":" elválasztva. A inicializálandó tagokat pedig vesszővel választjuk el.

Az osztályoknak lehetnek **statikus tagjai**. Az osztály statikus tagváltozói nem objektumhoz tartoznak, hanem magához az osztályhoz. Tehát ezek minden objektumnál azonosak, nem foglalódnak a memóriában új terület. Ezeket a tagokat az osztályon kívülről a hatókör operátorral lehet elérni, már azelőtt, hogy bármilyen példányosítást végrehajtottunk volna. A statikus tagváltozók mellett létezik még **statikus tagfüggvény**. Ezek kizárolag a statikus tagváltozókkal tudnak "dolgozni", főleg olyan esetekben használjuk, amikor a statikus tagváltozó privátban van, és szeretnénk elérni osztályon kívülről.

A C++ nyelvben lehetőség van beágyazott definíciókat alkalmazni, ezalatt azt értjük, hogy osztályokat, struktúrákat, típusdefiníciókat lehetőségeink van osztályon belül definiálni. A beágyazott osztályok függvényeit a hatókör operátor többszöri alkalmazásával tudjuk elérni, lényegében a teljes elérés utat kell megadni a vezérlésnek. Ezekre is érvényes, hogy ha `private` részben definiáljuk őket, akkor csak a publikus tagfüggvények segítségével tudjuk elérni a fő osztályon kívülről. Fontos megjegyezni, hogy attól, hogy egy osztály egy másiknak a része, nem ad jogokat a fő osztálynak, és ez fordítva is igaz. Ez úgy oldható meg, ha az osztályok egymást `friend`-nek deklarálják.

Operátorok és túlterhelésük

A beépitett típusokkal rengeteg művelet végrehajtható, de ha olyanakra is szeretnénk ezeket értelmezni, ami nem része a nyelvnek, akkor kell operátor túlterheléshez fordulnunk. Az operátorok az argumentumaikon hajtanak végre műveleteket, némelyik még az értéküket is megváltoztatja, ezt hívjuk az operátor mellékhatásának. Az operátorok kiértékelésének sorrendjét a C++ nyelv precedencia táblázata határozza meg. A C nyelvben a függvényeknek nincs mellékhatása, kivéve akkor, ha pointerként adjuk át a paramétert. A C++-ban létező referenciaiának köszönhetően ez a problma nem áll fent. Tehát az operátorok felfoghatók speciális függvényekként, azzal a különbséggel, hogy a kiértékelés szabályrendszer alapján történik.

C++ sablonok

A C++ nyelv egy érdekes megoldása a sablonok használata. Ezekre azért van szükésg, hogy az osztályokat vagy a függvényeket általánosabb módon tudjuk definiálni. Például van egy függvényünk, amit szeretnénk, hogy működjön `int`, és `double` típussal is. Ebben az esetben választhatnánk azt a megoldást, hogy szimplán lemasoljuk, és ahol eddig `int` volt, oda `double` írunk. Ez nem túl hatékony, ezért helyette készítünk egy sémát, és az éppen használni kívánt típust sablonparaméterként adjuk át. Ugyanez érvényes az osztályokra is, ott is megoldható, hogy különböző típusokat használunk, az átadott paraméter alapján. A fordítónak a `template` kulcsszóval jelezük, hogy ez egy sablon, a sablonparamétereit pedig `<>` jelek közé írjuk, vesszővel elválasztva. PL.:

```
template <class tipus> tipus novelo (tipus nev)
{
    return nev+1;
}
```

A `class` kulcsszóval jelezük, hogy egy típust szeretnénk átadni paraméterként. A példányosítást megtehetjük implicit és explicit módon is.

```
int a = novelo(5); //implicit
int b = novelo(6.3) //hiba
int c = novelo<double>(6.3); //explicit, nincs hiba
```

Ahogy látható, az implicit megadás lehetségteljesebb, viszont bizonyos esetekben hibához vezet, mert csak egy típust tud kezelni. Viszont az explicit megoldásnál már nincs ezzel gond, mivel megadjuk, hogy milyen típust szeretnénk használni, tehát az automatikus konverzió segítségével az double-t int-é tudjuk alakítani. A függvények mellett osztályokból is készíthetünk sablonokat. A működési elve nagyon hasonló. Itt is sablonparaméterként adjuk meg a típust, a tagváltozók pedig ilyen típusúak lesznek. Szintaktikailag így néz ki:

```
template <class tipus> class Pelda
{
    ...
};
```

Ha ennek az osztálynak egy tagfüggvényét az osztályon kívül szeretnénk definiálni, akkor a következő módon kell eljárni:

```
template <class tipus>
void Pelda<tipus>::novelo(tipus adat)
{
    ...
}
```

Tehát a definíció előtt a template kucszsó után fel kell sorolni a sablonparamétereket. Ha ez megvolt, az osztály neve után fel kell sorolnunk a sablonparaméterek nevét. A konstruktörököt is ugyanígy kell definiálni. Régebbi fordítóknál problémát okozhat, hogy csak az osztály nevét adjuk meg, amikor egy változó típusát adjuk meg. Ilyen esetekben meg kell adni a sablonparaméterek nevét is.

```
template <class tipus> class Pelda
{
    Pelda(const Pelda & objektum) //ez régen hibát dobott
    {
    }
    Pelda(const Pelda<tipus> & objektum) //régen így ←
        kellett
    {
    }
};
```

Argumentumfüggő névfeloldás

Az argumentumfüggő névfeloldást Koenig féle névfeloldásnak is nevezik. A lényege az, hogy ha van egy névterünk:

```
namespace Pelda
{
    class X
    {
    ...
}
```

```
};  
  
void f(X& x) { ...}  
}
```

Ebben az esetben ahhoz, hogy példányosítsunk egy X osztáylú objektumot, ki kell írnunk hatókör operátorral, hogy melyik névterből szeretnénk ezt létrehozni. Ezzel szemben a f függvény esetén erre nincs szükség. Ez annak köszönhető, hogy amikor paraméterként átadjuk neki az objektumot, akkor a fordító a Pelda névterben keres, és ott az f függvényt is megtalálja.

```
int main()  
{  
    Pelda::X alma;  
    Pelda::f(alma); //ez rendben van  
    f(alma); // ez is  
}
```

Ennek a legnagyobb előnye az operátorok használatánál ütközik ki. Vegyük példának a már jól ismert Hello, World! programot.

```
#include <iostream>  
int main()  
{  
    std::cout<<"Hello, World!\n";  
}
```

Ebben a példában mind a cout, mind a << operátor az std névterben van definiálva. Ha nem létezne névfeloldás, akkor ezt a következőképpen kéne írnunk:

```
std::operator<<(std::cout, "Hello, World!");
```

Típuskonverziók

A C++ típuskonverziói nagyban hasonlítanak a C-ben megismertekre, viszont ezek használata biztonságosabb. Ráadásul megjelentek objektumorientáltságot is figyelembe vevő konverziók.

Elsőnek nézzük meg a **beépitett típusok** konverzióját. A C nyelvvel ellentétben a enum és a int típusok között implicit konverzió nem lehetséges, az explicit variánst kell használni. A double és int típusok között van automatikus konverzió, de más a helyzet a referenciaikkal. A referenciák egy memóriaterületre hivatkoznak, emiatt a típus konverzió hibákhoz vezethet. A típuskényszerítést csak olyan esetben érdemes használni referenciaikkal, ha azok konstansok. Tehát:

```
double négyzet(const double& d)  
{  
    return d*d;  
}  
  
int main()  
{  
    int i = 2;  
    square((double)i);  
}
```

Ebben az esetben az `i`-t a fordító `double` tudja alakítani, azaz nem ezt alakítja át, hanem létrehoz egy ideiglenes konstanst, és ennek a referencia szerinti másolását teszi lehetővé.

A **felhasználói típusok** konverziója 2 részre osztható, a független típusok közötti, és hierarchikus típusok közötti konverzióra. Előbbi esetben a konverziós konstruktorra van szükségünk, ha egy másik típusról a saját osztályunkra szeretnénk konvertálni. Fordított esetben pedig konverziós operátorra van szükség. A konverziós konstruktor egy egyparaméteres konstruktor, a paramétere pedig olyan típusú, amilyenről szeretnénk végrehajtani a konverziót. Alapesetben csak egyargumentumú konstruktort használjuk, ahonnan a konverzó automatikusan végrehajtódik, de lehetőség van többargumentumú konstruktor definiálására is. A programozónak lehetősége van elkerülni a véletlen konverziókat azzal, hogy a konstruktor deklarációja elő az `explicit` kulcsszót írja. Ezzel az `implicit` konverzió letiltásra kerül, csak expliciten lehet megadni. Bizonyos esetekben a fordító nem tudja eldönteni, hogy melyik konverziót használja, az operátor minden oldalán lévő elemet át tudná alakítani a másikba. Ekkor explicit módon meg kell adni, hogy melyiket konvertálja, ellenkező esetben fordítási hibát kapunk.

A másik fontos része a felhasználói típusok konverziójának az egymással hierarchiában lévő típusok konverziója. Ha leszármazottról a szülőre történő konverzió esetén a másolókonstraktor hívódik meg. Azon tagváltozók, amik nem részei az ősosztálynak, a ásolás során kímaradnak, ezt nevezik **slicing-on-the-fly**-nak. Mivel másolókonstruktornak mindenkor kell lennie, ezért ez a konverzió automatikusan végrehajtódik. Ezzel szemben a szüéő-gyermekek konverzió már nem ilyen egyszerű. Ebben az esetben a leszármazottban kell megírni a konverziós konstruktort.

A **C++ típuskonverziós operátorai**-ról is érdemes szót ejteni. 4 operátor létezik erre a célra: a `static_cast`, a `const_cast`, a `dynamic_cast` és a `reinterpret_cast`. Ezek közül az első ismert lehet, lényegében ezt használtuk eddig.

```
double d = (int)3.14;
double d = static_cast <int> 3.14;
```

A kettő ekvivalens egymással. Összességében elmondható, hogy ez a leggyakrabban használt konverzó. A `const_cast` segítségével vagyunk képesek konstansokat nem konstanssá tesz, kizárálag ezzel lehet megoldani ezt a konverziót. Semmiyen más konverzióra nem használható. A `dynamic_cast` konverzióknak az a nagy előnye, hogy futási időben ellenőrzi, hogy a konverzió végrahajtható-e, ha nem, akkor hibával tér vissza. A fentebb említett leszármazottra történő konverzió esetén szükséges a használata. A `reinterpret_cast` konverziót pedig általában pointerknél használjuk, amikor meg akarjuk változtatni a pointer típusát.

A C++ I/O alapjai

A C nyelv állomány leírókat használ I/O műveletek végrehajtására. Ezzel szemben a C++ objektumokkal dolgozik. Ezeket nevezik adatfolyamnak, angolul `stream`-nek, mely bájtok sorozatából áll. A `istream` objektumok csak olvasatók, az `ostream` objektumok pedig csak írható. A beolvasást és a kiíratást a `<<`, `>>` operátorok végezik. Az adatfolyamok automatikusan felismerik a feldolgozásra szánt adatok típusát, ezért, ellentétben a C-vel, nem kell megadni `explicit`, ezt az információt. Beolvasásra a `cin` objektumot használjuk, ez alapértelmezetten a billentyűzetről olvas, a hozzá szükséges operátor: `>>`. Kiíratásra több alapértelmezett objektum is van. Ha valamit ki akarunk íratni a képernyőre, akkor a `cout`-ot használjuk. Ez adatfolyambufferbe tárolja a kiírandó adatokat. Ha hibát akarunk kiírni, akkor a `cerr` objektumot használjuk. Ez nem bufferel, ellentétben a szintén hibák kiírására használatos `clog`-gal. Mivel a bufferek tárolása erőforrás igényes, ezért lehetőség van a bufferek ürítésére, erre a `flush` kulcsszó használható, a következő formában:

```
cout << flush;  
cout.flush();
```

Az adatok beolvasása során fontos minden ellenőrizni az `cin` állapotát, ha valami hiba történt, akkor azt jelezni a felhasználónak. Az állapotot az `iostate` tagfüggvény adja meg. Ezt mi is állíthatjuk. 4 jelzőbit létezik ennek az állapotnak a kifejezésére: `eofbit`, `failbit`, `badbit`, `goodbit`. Az `eofbit` jelzi, ha az állomány végére ért az írás, vagy az olvasás. A `failbit` olyan hibát jelöl, amely valamelyen formátumbeli problémákból ered. Ezzel szemben a `badbit` már fatális hibát jelöl, melynek oka adatvesztés, vagy ehhez hasonló probléma. Fontos, hogy a `failbit` akkor is igazzal tér vissza, ha a `badbit` van igazra állítva. A `goodbit` esetén pedig az egyik előbb említett hibabit sincs beállítva. Ha egy adatfolyam valamelyik hibabitje beállítódik, onnantól kezdve használhatatlanná válik, minden írás és olvasás hatás-talan. Ezen jelzőbitek állapotát külön-külön le tudjuk kérdezni a hozzájuk tartozó tagfüggvényekkel, tehát `cin.good()`, `cin.bad()`, `cin.fail()`, `cin.eof()`. Természetesen ez igaz az összes eddig említett adatfolyam objektumra is. Lehetőségünk függvények használatára is, mellyel az írást és az olvasást végezhetjük. `get()` segítségével karaktereket olvashatunk be, a `getline()` függvény a sor végéig olvassa a bemenetet, a `read` segítségével bináris adatokat olvashatunk be, az `unget`, `putback` pedig az utolsó karaktert visszahelyezi az adatfolyamba. A kiíratásra a `write` függvényt használhatjuk, ha a bináris adatokat akarunk kiírni, a `put` karaktert ír ki.

Az adatfolyamokat lehetőségünk van manipulátorokkal tudjuk beállítani az egyes tulajdonságait, vagy tagfüggvények használatával. Mind a kettőre láttunk példát, buffer ürítésével kapcsolatban:

```
cout << flush; //manipulátor  
cout.flush(); //tagfüggvény
```

A manipulátorok közé tartozik még a `endl`, ami egy sortörést szűr be. A `cin` például bolvasáskor el-hagyja a whitespace karaktereket, ezt a `noskipws` manipulátorral tudjuk megakadályozni. Manipulátorokkal tudunk például lebegőpontos számok kiírásának pontosságát beállítani, ilyen a `setprecision()`. Használhatunk jelzőbiteket a tulajdonságok beállításának és törlésének a megkönyítése érdekében. Ilyen jelzőbit a `ios::fixed`. Ha ezek tárolására nem elég egy bit, akkor pedig maszkot használunk. Manipulárokkal képesek vagyunk mezőszélességet állítani, kitöltő karaktereket beállítani, igazítani a szöveget, számlarendszerek között váltani, a lebegőpontos számok formátumát állítani, számok előjelenék muatását pozitívak esetén is, helykitöltő nullákat megjeleníteni. Tehát rengeteg minedre képesek ezek a manipulátorok, és nem is kell mindegyiket fejben tartani, mert a minden megtaláljuk a fordítók dokumentációjában. Fontos, hogy a `setw` manipulátor, melyet a mezőszélesség beállítására használunk, az egyetlen, ami csak a megadott adatok kiírására érvényes. A többi hatályba lép az utána következő írásokra vagy olvasásokra, tehát ezeket minden vissza kell állítani alapértelmezettre, ha már nincs rájuk szükségünk.

Ha állományokba akarunk írni, vagy állományokból akarunk olvasni, akkor az `ofstream` és az `ifstream` osztályokat kell használnunk. Az előbbi a kimenetet használt állomány kezelésére szolgál, utóbbi pedig a bemeneti állományokat kezeli. Ha kétirányú adatfolyamra van szükségünk, akkor pedig a `fstream` osztályra lesz szükség. Mivel az osztályokhoz tartozik konstruktur és destruktor, ezért az állományok megnyitása, és bezárása elvégezhető ezekben. Itt is használhatunk jelzőbiteket, mint a `formázánál`, melyeket az `ios::` előtaggal látunk el. Ilyen jelzőbit az `ios::in`, mellyel megadjuk, hogy csak olvasásra nyitottunk meg az állományt, természetesen az `ios::out` pedig azt jelenti, hogy írásra szeretnénk használni az állományt. Ha nem szeretnénk felülríni a célállomány tartalmát, akkor használhatunk hozzáfűzést az `ios::app` kapcsolóval. Az `ios::ate` jelzőbit segítségével az állomány végére ugrunk a megnyitás után, a `ios::trunc` törli az adatokat az állományból, a `binary` pedig az állomány hozzáférést binárisra állítja. Természetesen ezeket a jelzőbiteket lehet kombinálni is egymással a `"|"` operátorral.

Kivételkezelés

C nyelvben a hibák kezelésére a `return` kulcsszót használtuk, de ez egy hosszabb híváslista esetén átláthatatlanná teszi a hibakezelést. Erre nyújt megoldást a C++ kivételkezelése. A neve jól mutatja, hogy nem csak hibakezelésre alakalmas, hanem bármilyen kivételes esetet tudunk vele kezelní, ami előfordul a programban. A kivételt a "eldobjuk" és a vezérlés a megfelelő kivételkezelésre ugrik, annak lefutásának következtével pedig a vezérlés a kivételkezelő után ugrik. A könyvben látható egy egyszerű példa a C++ kivételkezelésre, mely a bekért szám reciprokát adja vissza.

```
//BME tankönyv 190.o.  
#include <iostream>  
using namespace std;  
int main()  
{  
    try{  
        double d;  
        cout << "Enter a nonzero number: ";  
        cin >> d;  
        if (d==0)  
            throw "The number can not be zero.";  
        cout << "The reciprocal is: " << 1/d << endl;  
    }  
    catch (const char* exc)  
    {  
        cout << "Error! The error text is: " << exc << endl;  
    }  
    cout << "Done." << endl;  
}
```

A `throw` kulcsszót használjuk a kivétel dobására. Jelen esetben azt kell kezelní, ha a felhasználó nullát ad meg. Ez a részt, amin a kivételkezelést szeretnénk alkalmazni a `try` blokkba zárjuk. Amikor eldobjuk a hibát a vezérlés megkeresi a megfelelő `catch` blokkot, amellyel elkapjuk a kivételt. Akkor megfelelő egy `catch` blokk, ha a paraméter típusa megyegyezik a kivéteként dobott elem típusával. A kivétel beíródik a `catch` blokk paraméterének, a blokkon belül elérhető. Jelen esetben kiíratjuk a hibaként dobott kódot. Már említettem, hogy a kivételkezelés nagy előnye, hogy egész hosszú hívási lánc végéről dobott kivételek kezelésére is könnyen alkalmazható. Hiszen a vezérlés a hívási láncon visszafelé lépkedve megeskeresi a megfelelő `catch` blokkot. Vannak esetek, amikor közvetlenebbül kell kezelnünk valamilyen kivételt. Ebben az esetben a függvények belsejében kell létrehozni egy `try-catch` blokkot. Arra is lehetőség van, hogy a függvényen belül elkapott kivételt továbbítsuk egy magasabb szintű kivételkezelőnek, tehát a szükséges utasításokat a lokális `catch` blokk végrehajtja, majd továbbítja a kivételt. Erre főleg azért van szükség, hogy a felhasználó kapjon információt arról, hogy hiba történt. Ellenkező esetben csak a lokális `catch` blokk tudna a kivételről, a `main` függvény nem. Fontos tudni, hogy amikor a vezérlés visszafelé halad a hívási láncon, akkor a lokális változók destrukturálódnak, tehát azokat már később nem érjük el. Nem szabad a kivétel dobása és elkapása között hibát dobni. A lokális változók felszabadulásának folyamatát ebben az esetben verem visszacsévélésnek nevezzük. Erre ad példát a következő példa:

```
//BME 197.o.  
int main()  
{  
    try
```

```
{  
    f1();  
}  
catch(const char* errorText)  
{  
    cerr << errorText << endl;  
}  
}  
  
void f1()  
{  
    Fifo fifo;  
    f2();  
}  
  
void f2()  
{  
    int i = 1;  
    throw "error'";  
}
```

Tehát a program a `try` blokkban meghívja az `f1()` függvényt. Ez a függvény példányosít egy `Fifo` objektumot, majd meghívja az `f2()` függvényt. Az `f2()` függvény definiálja az `i` változót, és a dob egy kivételt, mely `string`, azaz `char*` típusú. Ekkor kezdődik meg a verem visszacsévélés. Az `i` változó felszabadul, a `Fifo` osztály destruktora meghívódik, végül pedig lefut a `main` függvény `catch` blokkja. Fontos aspektusa a kivételkezelésnek az erőforrások kezelése, azon belül is a memóriája. Vegyük azt az esetet, mikor még az előtt dobunk kivételt, hogy a egy objektumot felszabadítanánk. Ebben az esetben a memória elfolyik, mivel a vezérlés már nem fog ráugrani a megefelelő utasításra, hanem a `catch` blokk után folytatódik. Ezt úgy tudjuk megoldani, hogy ilyen esetekben a `catch` blokkon belül szabadítjuk fel az objektumot. Ezt láthatjuk a következő kódban:

```
class MessageHandler  
{  
public:  
    void ProcessMessages(istream& is)  
    {  
        Messages *pMessage;  
        while(pMessage = readNextMessage(is)) != NULL)  
        {  
            try  
            {  
                pMessage -> Process();  
                delete pMessage;  
            }  
            catch(...)  
            {  
                delete pMessage;  
                throw;  
            }  
        }  
    }  
}
```

```
    }  
    ...  
}
```

Tehát ebben a kódrészleteben üzeneteket feldolgozó függvényt nézzük meg. A függvényben példányosítunk egy `Messages` objektumra mutató mutatót. Majd ebbe olvassuk be az üzeneteket. Az üzenetek feldolgozását a `Process()` függvény végzi, ebben a példában ez a függvény dobja a hibát. De ha hibát dob, akkor az őt követő `delete` hívása. Ezért van szükség a `catch` blokkban lévő `delete` hívásra. Egy másik fontos dolgot megfigyelhetünk ebben a példában. A `catch` blokk paraméterlistája nem nevezi meg konkrét paramétertípusot, ennek következtében minden kivétlt elkap.

DRAFT

III. rész

Második felvonás

DRAFT

**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.

DRAFT

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

Az előző feladat kódját fejleszd tovább: vizsgáld, hogy Vannak-e Java osztályok a Pi hexadecimális kifejtésében!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

DRAFT

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. 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/DEACHackers> (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.