

Univerzális programozás

Így neveld a programozód!

Ed. BHAX, DEBRECEN,
2019. május 12, v. 0.0.4

Copyright © 2019 Dr. Bátfai Norbert

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

Copyright (C) 2019, Takács Viktor, takiviko@gmail.com

Copyright (C) 2019, Rádi Dániel, radidani55@gmail.com

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert, Takács, Viktor, Ács Rádi, Dániel	2019. október 25.	

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	A Brun tételes feladat kidolgozása.	nbatfai
0.1	2019-03-12	Az első commit	takiviko
0.1.1	2019-05-12	A közel végleges commit	takiviko
1.0	2019-05-12	A teljesen végleges commit	takiviko
1.1	2019-09-27	Prog 2 Kezdetek	takiviko and raddanfea

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	7
2.4. Labdapattogás	7
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	9
2.6. Helló, Google!	10
2.7. 100 éves a Brun téTEL	12
2.8. A Monty Hall probléma	13
3. Helló, Chomsky!	15
3.1. Decimálisból unárisba átváltó Turing gép	15
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	15
3.3. Hivatalos nyelv	17
3.4. Saját lexikális elemző	18
3.5. l33t.l	19
3.6. A források olvasása	19
3.7. Logikus	20
3.8. Deklaráció	20

4. Helló, Caesar!	23
4.1. double ** háromszögmátrix	23
4.2. C EXOR titkosító	25
4.3. Java EXOR titkosító	27
4.4. C EXOR törő	28
4.5. Neurális OR, AND és EXOR kapu	30
4.6. Hiba-visszaterjesztéses perceptron	32
5. Helló, Mandelbrot!	34
5.1. A Mandelbrot halmaz	34
5.2. A Mandelbrot halmaz a std::complex osztállyal	38
5.3. Biomorfok	39
5.4. A Mandelbrot halmaz CUDA megvalósítása	40
5.5. Mandelbrot nagyító és utazó C++ nyelven	41
5.6. Mandelbrot nagyító és utazó Java nyelven	41
6. Helló, Welch!	42
6.1. Első osztályom	42
6.2. LZW	42
6.3. Fabejárás	43
6.4. Tag a gyökér	43
6.5. Mutató a gyökér	43
6.6. Mozgató szemantika	43
7. Helló, Conway!	45
7.1. Hangyszimulációk	45
7.2. Java életjáték	46
7.3. Qt C++ életjáték	47
7.4. BrainB Benchmark	47
8. Helló, Schwarzenegger!	48
8.1. Szoftmax Py MNIST	48
8.2. Mély MNIST	48
8.3. Minecraft-MALMÖ	48

9. Helló, Chaitin!	50
9.1. Iteratív és rekurzív faktoriális Lisp-ben	50
9.2. Gimp Scheme Script-fu: króm effekt	50
9.3. Gimp Scheme Script-fu: név mandala	51
10. Helló, Gutenberg!	52
10.1. Juhász István - Magas Szintű Programozási Nyelvek 1	52
10.2. B. W. Kernighan, D. M. Ritchie - A C programozási nyelv	53
10.3. Benedek Zoltán, Levendovszky Tihamér - Szoftverfejlesztés C++ nyelven	56
III. Második felvonás	58
11. Helló, Berners-Lee!	60
11.1. A C++ és a Java nyelvek összehasonlítása	60
11.2. A Python3 programozási nyelv	60
12. Helló, Arroway!	62
12.1. Objektum-Orientált Szemlélet	62
12.2. Homokozó	63
12.3. Gagyi	63
12.4. Yoda	63
12.5. Kódolás From Scratch	64
13. Helló, Liskov!	67
13.1. Liskov helyettesítés sértése	67
13.2. Szülő-Gyerek	67
13.3. Ciklomatikus komplexitás	69
13.4. Az SMNIST felélesztése	69
13.5. BPP algoritmus futási ideinek összevetése	70
14. Helló, Mandelbrot!	71
14.1. OOCWC Class Diagramms	72
14.2. BPMN	73
14.3. UML Reverse Engineering	75
14.4. UML Osztálydiagram From Scratch	75

15. Helló, Chomsky!	77
15.1. l334d1c4	77
15.2. Perceptron osztály	78
15.3. Encoding	79
15.4. FullScreen	79
16. Helló, Stroustup!	81
16.1. Objektumok másolása és mozgatása c++11-ben + esszé	81
16.2. RSA törés	83
16.3. Változó argumentumszámú ctor	84
IV. Irodalomjegyzék	86
16.4. Általános	87
16.5. C	87
16.6. C++	87
16.7. Lisp	87
16.8. Számítógépes Nyelvészeti Elmélete és Gyakorlata	87

Ábrák jegyzéke

3.1. A Chomsky-féle nyelvosztályok	16
4.1. Többelemű tömbre való hivatkozás pointerekkel	23
4.2. A program kimenete	25
5.1. A Mandelbrot halmaz a komplex síkon	37
5.2. Biomorph	40
7.1. UML	46
7.2. Sejtautomata	47
9.1. Mandala	51

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' does not exist.
Build bhax-textbook-fdl.pdf
'bhax-textbook-fdl.pdf' successfully built
```

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

Ne cifrázzuk: programok írása. Mik akkor a programok? Mit jelent az írásuk?

1.2. Milyen doksikat olvassak el?

- Kezd ezzel: <http://esr.fsf.hu/hacker-howto.html>!
- Olvasgasd aztán 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
- C kapcsán a [**KERNIGHANRITCHIE**] könyv adott részei.
- C++ kapcsán a [**BMECPP**] könyv adott részei.
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.
- Amiből viszont a legeslegjobban lehet tanulni, az a [The GNU C Reference Manual](#), mert gcc specifikus és programozókra van hangolva: szinte csak 1-2 lényegi mondat és apró, lényegi kódcsipete! Aki pdf-ben jobban szereti olvasni: <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf>
- Az R kódok olvasása kis általános tapasztalat után automatikusan, erőfeszítés nélkül menni fog. A Python nincs ennyire a spektrum magától értetődő végén, ezért ahhoz olvasd el a [**BMECPP**] könyv - 20 oldalas gyorstalpaló részét.

1.3. Milyen filmeket nézzek meg?

- 21 - Las Vegas ostroma, <https://www.imdb.com/title/tt0478087/>, benne a **Monty Hall probléma** bemutatása.
- Kódjátszma, <https://www.imdb.com/title/tt2084970>, benne a **kódtörő feladat** élménye.

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

Egy végtelen ciklus egy programban nem szerencsés, ha nem direkt van implementálva, hiszen erőforrást használunk anélkül, hogy akármilyen számolást végeznénk. Vannak esetek azonban, amikor egy véget nem érő, folyamatosan futó ciklus hasznos, sőt, elkerülhetetlen. A legegyértelműbb használata egy ilyen folyamatnak a rendszer erőforrásainak tesztelése. Ha a processzor órajelével játszadozunk, azaz overclock-olunk, a stabilitást egészen egyszerűen lehet tesztelni egy olyan végtelen ciklussal, mely 100%-ban dolgoztatja a processzorunkat.

Abban az esetben is hasznos lehet egy ilyen folyamat, ha egy inputot várunk egy felhasználótól, vagy másik programtól. Végtelen ciklus nélkül nem tudnánk bizonyos időközönként ellenőrizni, hogy a felhasználó megnyomott-e egy billentyűt, vagy nem. Láthatjuk tehát, hogy elkerülhetetlen egy végtelen ciklus használata, ha bizonyos időintervallumomként le kell futtatni egy kód részletet. Több mód is van egy végtelen ciklus létrehozására, lehet for, while, do-while ciklussal, vagy goto kifejezéssel is implementálni. Íme egy-egy példa mindegyikre:

For ciklussal:

```
Program T100
{
{
int main()
{
    for(;;)
    {
        return 0;
    }
}
}
```

While ciklussal:

```
Program T100
{
{
    while (TRUE)
```

```
    {}  
    return 0;  
}  
}
```

Do While ciklussal:

```
Program T100  
{  
int main()  
{  
    do  
        {}  
    while (1);  
    return 0;  
}  
}
```

Goto kifejezéssel:

```
Program T100  
{  
int main()  
{  
    cycle:  
    goto cycle;  
    return 0;  
}  
}
```

A C és C++ nyelvekben a többmagosítás (multithreading) a fork() függvényel valósítható meg, ezzel tudjuk elérni, hogy több szálat (thread) legyen képes használni a programunk. Egy fork() függvény két részre osztja a folyamatot, így ha n számú fork() függvényt helyezünk a programunkba, 2^n szálat fogunk tudni dolgoztatni. Ezt a következő példában láthatjuk:

Ha nem akarjuk, hogy a programunk lefoglalja a lehető legtöbb megengedett cpu-időt, bevezethetjük a sleep() függvényt, mely a zárójelek között megadott ideig (másodpercben) megállítja a folyamatunkat. Ezzel a függvényel el lehet érni a programjainkban, hogy csak bizonyos sebességgel fussanak, vagy csak bizonyos időközönként hajtsanak végre egy folyamatot. Ebben az esetben a végtelen ciklusunk nem fogja 100%-ban dolgoztatni a processzort, hiszen összesen 1 művelet hajtódik végre másodpercenként.

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

Egy minden informatikust érintő probléma a "Halting Problem", azaz az a kérdés, hogy lehet-e olyan programot írni, mely eldönti egy másik programról, hogy az végtelen ciklusba került-e. A problémával először Alan Turing foglalkozott 1936-ban, és arra a konklúzióra jutott, hogy nem lehetséges ilyen programot írni. Ezt az állítást a következőképp bizonyíthatjuk:

Tegyük fel, hogy sikerült olyan programot írni, mely el tudja dönteni bármilyen programról, hogy meg fog-e állni, vagy sem. Ha egy ilyen programnak bemenetül adjuk önmagát, ez a folyamat a végtelenségig fog futni, hiszen a rekurzió minden lépésében újra meghívja önmagát. Így a programunk, amelynek az lett volna a célja, hogy leállítsa a végtelenségig futó programokat, végtelen ciklusba került.

Erre a problémára sajnos nincs megoldás, így az informatikusoknak jelentős figyelmet kell fordítani arra, hogy az általuk írt programokban lezajló folyamatok egyszer véget érjenek.

2.3. Változók értékének felcserélése

Egy program futása közben sok esetben fordul elő, hogy két változó értékét egymás memóriacímére kell helyeznünk. Ez a folyamat az összes rendező algoritmus része, és a legtöbb adatszerkezet (pl. tömb, lista, stb) kezeléséhez szükséges. A legegyszerűbb megoldás a problémára egy harmadik, ideiglenes változó bevezetése, de ez értékes memóriát vehet el a rendszertől, valamint felesleges deklarációkhöz vezet. A legtöbb algoritmus Olyan módszereket használ, melyekhez nem szükséges egy új változót deklárnai.

Az első két módszer hasonlít egymáshoz, ugyanazon az alapelveken működnek. Mindkettő három lépésből áll, annyi a különbség, hogy míg az egyikben összeadást és kivonást használunk, a másikban szorzást és osztást. A folyamat első lépése az, hogy a két változó értékét összeadjuk és eltároljuk a változók egyikében. A következő lépés során a változatlan értékű változónak megadjuk az összegből levont értékét. Ekkor ennek a változónak az értéke pontosan annyi lesz, mint a másik változó értéke volt a folyamat megkezdése előtt. A harmadik, azaz utolsó lépésben Az első változónknak megadjuk az önmagából levont másik változó értékét. Ekkor a két változó értékének a cseréje megtörtént.

```
Program T100
{
    int main()
    {
        a=a+b;
        b=a-b;
        a=a-b;

        return 0;
    }
}
```

2.4. Labdapattogás

Sok programban, főleg játékokban szükség van arra, hogy valamilyen elemet mozgassunk a képernyőn. Mivel a képernyő pixelekből áll, melyeknek saját koordinátáik vannak, meg lehet oldani elemek mozgatását az elem pillanatbeli helyzetének változtatásával. Egy terminálablakban is tudunk hasonló módon mozgatni egy elemet, esetünkben egy karaktert, melynek megadhatjuk, hogy csak egy bizonyos kereten belül mozogjon.

A labda (esetünkben egy O szimbólum) ciklusonként kirajzolódik, valamint helyet változtat egy adott irányba. Ha eléri a pálya szélét, azaz az egyik for ciklusban teljesül a feltétel, a labda irányt változtat. Mindez egy végtelen (while) ciklusban fut, így a labda addig fog pattogni, amíg manuálisan le nem állítjuk a programot.

```
Program T100
{
#include <iostream>
#include <stdlib.h>
#include <unistd.h>

char bitzero(char x) {
    int i;
    char bitt = x&0x1;
    for (i=0; i<8; i++) {
        bitt |= (x>>i)&1;
    }
    return 1-bitt;
}

void rajzol(char width, char height) {
    int i;
    for (i=1; i<=height; i++) {
        std::cout << std::endl;
    }
    for (i=1; i<=width; i++) {
        std::cout << " ";
    }
    std::cout << "O" << std::endl;
}

int main() {
    char x=1, y=1, vx=1, vy=1;
    while(1) {
        system("clear");
        vx-=2*bitzero(79-x);
        vx+=2*bitzero(x);
        vy-=2*bitzero(24-y);
        vy+=2*bitzero(y);
        x+=vx;
        y+=vy;
        rajzol(x,y);
        usleep(100000);
    }
    return 0;
}
}
```

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

Mikor Linus Torvalds megírta a saját kerneljét, szüksége volt egy algoritmusra, mely nagyjából meg tudja állapítani a számítógép teljesítményét. Erre megoldásként megírta a BogoMIPS nevű programját. A neve beszédes, a "bogo" rész a bogus szóból ered, mely hamisat jelent, a MIPS (Million Instructions Per Second) pedig millió instrukciót másodpercenként. Azért lett ez a neve, mert bár a processzor teljesítménye függvényében általánosan nő a visszaadott szám. Ez az érték nem pontos, különböző architektúrák között összehasonlításra nem alkalmas, mert a memória, az I/O rendszer és a gyorsítótár sebessége és mérete nagyobb mértékben befolyásolja, mint a processzor órajele.

A program azt nézi meg, hogy hány utasítást tud a rendszer végrehajtani bizonyos időközönként (2^n tick-enként). Ha a végrehajtott utasításokra szánt idő túllépi az egy másodpercet, a folyamat leáll és visszaad egy értéket. Ez az érték úgy jön ki, hogy elosztjuk a végrehajtott utasítások számát az eltelt idővel, majd elosztjuk a CLOCKS_PER_TICK konstanssal, melynek értéke a clock() függvény másodpercenkénti tick-jeinek a száma (1.000.000).

```
Program T100
{
void delay(const unsigned long long int &loops);

int main() {

    unsigned long long int loops_per_sec = 1;
    unsigned long long int ticks;
    std::cout << "Calibrating delay loop.." << std::endl;
    while (loops_per_sec <= 1) {
        ticks = clock();
        delay (loops_per_sec);
        ticks = clock() - ticks;

        std::cout << ticks << " " << loops_per_sec << std::endl ;
        if (ticks >= CLOCKS_PER_SEC) {
            loops_per_sec = (loops_per_sec / ticks) * CLOCKS_PER_SEC;
            double result = static_cast<double> (((loops_per_sec / 5000) / 100) + ←
                (loops_per_sec / 5000) % 100);

            std::cout << "ok - " << std::fixed << std::setprecision(2) << result ←
                << " BogoMIPS" << std::endl;
            return 0;
        }
    }

    std::cerr << "failed" << std::endl;
    return -1;
}

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

```
}
```

A program 1993 óta a linux kernel része és bootoláskor azóta is minden alkalommal lefut. Bár benchmarkra nem alkalmas, ha valami probléma van valahol a rendszerben, a BogoMIPS segíthet szűrni annak a lehetséges okát.

Egy változó méretének lehetőleg elég nagynak kell lennie, hogy egy program futása során ne legyenek problémák abból, hogy az értéke lenullázódik. Ha viszont túl nagy a mérete, az ahhoz vezethet, hogy feleslegesen sok memóriát használ a program, hiszen túl nagy memóriaszeletet tulajdonít a változóknak, pedig kevesebb is bőven elég lenne. A c++ nyelv egy egyszerű egész számot 32 bittel tárol el, így 2^{32} , azaz több, mint négy milliárd különböző értéket vehet fel.

Ennek a bizonyítását a következő programmal végezhetjük el:

```
Program T100
{
int main()
{
    int a=1;
    int i;

    while (a!=0)
    {
        a <= 1;
        i++;
    }

    std::cout << "int: " << i << std::endl;

    return 0;
}
}
```

A programban az "a" változónak 1 értéket adunk, mely binárisan is 1. Ezután a számjegyeket folyamatosan toljuk el balra, egészen addig, amíg a rendszer nem képes tovább tolni a számot, mivel az "kicsúszik" a számára lefoglalt memóriából. Ekkor az 1 már nem lesz a szám része és csupa 0 értékkel fog rendelkezni az a változó. A lépések összeszámolva megkapjuk, hogy hánysor kellett balra tolni az első bitet, hogy az eltűnjön.

2.6. Helló, Google!

Mikor a 90-e évek elején fokozatosan egyre népszerűbbé vált az internet, sokszor nehéz volt navigálni rajta, mert az adatok halmazában nehéz volt eligazodni, megtalálni a keresett weboldalt. 1996-ban két egyetemista, Larry Page és Sergey Brin kidolgozott egy algoritmust, hogy a weboldalakat népszerűségeük és megbízhatóságuk alapján tudják csoportosítani. Az algoritmus azt nézte meg, hogy egy oldalra hány külső link mutat és az oldalról hány link mutat ki. Az oldalak értékét úgy számolták ki, hogy az összes oldalra mutató linkek számát elosztották a kimenő linkek számával (persze relatívan a többi oldalhoz képest). Ekkor egy tört érték jön ki, mellyel megismétlik a folyamatot. A folyamatos szorzások során az eredmény

nem lesz teljesen pontos, hiszen egy határérték felé fog haladni, de leállíthatjuk a folyamatot, ha már olyannyira kicsi lesz az eltérés az n-edik és az n-1-edik művelet közt, hogy az nem befolyásol semmit. Az algoritmusnak nagy sikere lett és azóta is használatban van más, komplexebb algoritmusok mellett

A PageRank forráskódja c ++-ban:

```
Program T100
{
    int main()
    void kiir(double tomb[], int db);
double tavolsag(double PR[], double PRv[], int n);
void pagerank(double T[4][4]);

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

    std::cout << std::endl << "Az oldalak PageRank értékei:" << std::endl;
    pagerank(L);

    std::cout << std::endl;

    return 0;
}

void kiir(double tomb[], int db) {
    for (int i = 0; i < db; ++i) {
        std::cout << std::fixed << std::setprecision(6) << tomb[i] << std::endl <<
    }
}

double tavolsag(double PR[], double PRv[], int n) {
    double osszeg = 0;

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

    return std::sqrt(osszeg);
}

void pagerank(double T[4][4]) {
    double PR[4] = { 0.0, 0.0, 0.0, 0.0 };
    double PRv[4] = { 1.0/4.0, 1.0/4.0, 1.0/4.0, 1.0/4.0 };
}
```

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

    if (tavolsag(PR, PRv, 4) < 1e-10)
        break;

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

kiir(PR, 4);
}
```

2.7. 100 éves a Brun téteI

Bár a prímszámok reciprokainak összegéből álló sort divergensnek tartják, 1919-ben Viggo Brun kidolgozott egy elméletet, mely szerint az ikerprímek reciprokainak összegét ha összegezzük, egy valós számot kapunk, azaz a sor konvergens. Abból, hogy a a prímek reciprokösszege divergens, következik, hogy a prímek száma végtelen, ám nem vonhatunk le hasonló állítást a Brun-konstanssal kapcsolatban. A Brun-konstanst meghatározó sor képlete a következő:

A konstanst a szitaelméletben lehet hasznosítani, mely egész számokból álló halmazok szűrt halmazai elemének a számát vizsgálja.

A Brun-konstanst megközelítő érték kiszámítása R-ben:

```
Program T100
{
library(matlab)

stp <- function(x) {

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

x=seq(13, 1000000, by=10000)
```

```
y=sapply(x, FUN = stp)
plot(x,y,type="b")
}
}
```

2.8. A Monty Hall probléma

Egy érdekes probléma merült fel 1975-ben egy Steve Selvin nevű emberben : „Képzelje el, hogy egy vetélkedőben szerepel, és három ajtó közül kell választania. Az egyik mögött kocsi van, a másik kettő mögött viszont kecske. Tegyük fel, hogy maga az 1. ajtót választja, mire a műsorvezető, aki tudja, melyik ajtó mögött mi van, kinyitja a 3. ajtót, megmutatván, hogy amögött kecske van. Ezután önhöz fordul, és megkérdezi: „Nem akarja esetleg mégis a 2. ajtót választani?” Vajon előnyére válik, ha vált?” - *Wikipédia, 2019*

Az első gondolat a legtöbb emberben az, hogy nem számít, hogy a másik ajtót választja-e a játékos, hiszen a maradék két ajtó bármelyike mögött lehet akár autó, akár a kecske. Ám ez a feltételezés nem helyes. Amikor a játékos választ egy ajtót, akkor $1/3$ esélye van arra, hogy helyesen dönt és $2/3$, hogy nem. Attól, hogy az egyik ajtót kizártuk, a $2/3$ -os esély nem vesz el, ezután egyszerűen arra az ajtóra fog vonatkozni, amely nem lett a játékos által először választva. Ebből tehát levonhatjuk azt a következtetést, hogy minden esetben nagyobb esélyünk van az autót megszerezni, ha megváltoztatjuk a döntésünket.

Íme egy szimuláció a problémáról random számokkal, láthatjuk, hogy a nyert játékok száma általában kétszer annyi, ha a játékos ajtót vált az egyik rossz ajtó eliminációja után.

```
Program T100
{
int main() {
    srand(time(0));

    int rounds = 1000;
    int rounds_won = 0;
    int won_without_swap = 0;

    for (int i=0; i<rounds; ++i)
    {
        int prize_door = rand() % 3 + 1;
        int player_choice = rand() % 3 + 1;
        bool door_switch = rand() % 2;

        if (!door_switch && (player_choice == prize_door))
        {
            ++won_without_swap;
            ++rounds_won;
        }
        else if (door_switch && (player_choice != prize_door))
        {
            ++rounds_won;
```

```
        }
    }

    cout << (rounds - rounds_won) << "\t" << "Number of games lost" << endl <-
        ;
    cout << rounds_won << "\t" << "Number of games won" << endl;
    cout << won_without_swap << "\t" << "Games won without door change" << endl;
    int winWithChange = (rounds_won - won_without_swap);
    cout << winWithChange << "\t" << "Games won with door change" << endl;
    cout << winWithChange * 100 / rounds_won << "\t" << "Percent of games won with door change" << endl;
    cout << won_without_swap * 100 / rounds_won << "\t" << "Percent won without door change" << endl;

    return 0;
}
}
}
```

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Minden programozónak tisztában kell lennie a tízesen (decimálison) kívül bizonyos számrendszerrel, melyek a számítástechnikában nagy szerepet játszanak. Ilyen például a kettes (bináris), a nyolcas (oktális) és a hexadecimális (hexadecimális). Előfordulhat olyan eset is, hogy ezeken felül is tisztában kell lenni más, kevésbé elterjedt reprezentációs módszerekkel. A számrendszer közül a legegyszerűbb az unáris, azaz az egyes számrendszer, melyet viszonylag kevés esetben használunk. Az oka annak, hogy széleskörben nincs elterjedve, annak köszönhető, hogy a komplexebb számítások (szorzás, hatványozás, stb.) relatívan bonyolultak benne, valamint az, hogy az ilyen módon eltárolt adatok feleslegesen sok memóriát igényelnek.

Az egyes számrendszerbe való átváltás nem egy bonyolult feladat, egy for ciklussal át tudunk váltani bár-milyen számot unárisba úgy, hogy ha n az átváltandó szám, n -szer futtatjuk a ciklust, minden iterációban egy számjegyet hozzáadva a számhoz. Ezt egészen addig ismétljük, ameddig el nem érjük a várt értéket. A reprezentáció érdekében minden szám mögött nulla áll, ha sok számot fűzünk egymásba, ez segít elkülöníteni őket, de a számok értékét nem változtatja meg.

Mivel az unáris számrendszerben csak egy karakter létezik, ezt szabadon kicserélhetjük, így akármilyen karakterekkel reprezentálhatjuk a kívánt számot. Vannak, akik egyesek sorozatával ábrázolják az unáris számokat, mások nullákat használnak, de kétségtelen, hogy a függőleges vonalakkal való ábrázolási mód a legősibb mind közül.

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

Noam Chomsky a 60-as években elkezdett formális nyelvekkel foglalkozni, azaz próbált olyan szabályosságokat keresni, melyek alapján létrehozhatók bizonyos nyelvi elemekből (terminális és nem terminális karakterekből) olyan, magasabb szintű elemek (szavak), melyek meghatározott tulajdonságokkal rendelkeznek. Fontos megemlíteni, hogy csak terminális karakterekhez rendelhetünk más értékeket, nem-terminális karakterekhez nem. Ilyen szabályok egy halmazát hívjuk egy formális nyelv nyelvtanának.

A Chomsky-féle hierarchia

0. típus: **frázis-struktúra nyelvtanok** (nincs más megkötés)

1. típus: **környezetfüggő nyelvtanok**

minden levezetési szabály $\alpha A\beta \rightarrow \alpha\psi\beta$, $\psi \neq e$ alakú
(környezetfüggő), azaz nem csökkentő

2. típus: **környezetfüggetlen nyelvtanok**

minden szabály $A \rightarrow \psi$ alakú

3. típus: **reguláris nyelvtanok**

kétféle szabálytípus lehetséges: $A \rightarrow a$, $A \rightarrow aB$

3.1. ábra. A Chomsky-féle nyelvosztályok

A nyelvtani szabályok hozzárendelések, melyek során egy karakterlánc elemeit kicseréljük más elemekre. Ez a folyamat lehet véges és végtelen is, ha elérünk egy olyan pontot, hogy kizárolag terminális karakterekből (az ábécénk betűi) áll a szavunk, megállunk. Ekkor egy környezetfüggetlen grammaikát alkottunk meg. Vannak viszont olyan nyelvek, melyek nem alkothatók meg olyan módon, hogy:

F → f

Azaz nem tudunk olyan környezetfüggetlen grammaikát létrehozni, mely az $a^n b^n c^n$ nyelvet generálná le.

Példa környezetfüggő grammaikákra, melyek ezt a nyelvet generálják, ha S a kezdő karakter:

S → aSBC
S → aBC
CB → BC
aB → ab
bB → bb
bC → bc
cC → cc

Ez a szabályhalmaz a következő nyelvet generálja:

```
S -> aSBC -> aaBCBC -> aaBBCC -> aabBCC -> aabbCC -> ←  
      aabbccC -> aabbcc
```

```
S -> aAbc  
Ab -> bA  
Ac -> Bbcc  
bB -> Bb  
aB -> aa/aaA
```

Ez a szabályhalmaz pedig ezt a nyelvet generálja:

```
S -> aAbc -> abAc -> abBbcc -> aBbbcc -> aaAbbcc -> ←  
      aabAbcc -> aabbAcc -> aabbBbcc -> aabBbbccc -> ←  
      aaBbbbccc -> aaabbccc
```

Azonban ez a grammaтика nem környezetfüggetlen, hiszen nem esik bele a környezetfüggetlen szabályhalmazba.

3.3. Hivatkozási nyelv

A C programozási nyelvben szinte minden program tartalmat van változónak való értékkedést, valamiféle ciklust, függvényeket, stb. Ezeket minden utasításoknak nevezünk, tehát mondhatjuk, hogy egy program nem más, mint utasítások sorozata. Ezeket pontosvesszővel választjuk el. A sorrend egyáltalán nem mindegy, hiszen ha két sort felcserélünk, valószínű, hogy nem helyesen működni a programunk, vagy akár olyan is lehet, hogy le sem fordul. Egy programozónak tisztaiban kell lenni minden utasítással, amit használ, hiszen csak akkor lesz képes helyesen működő programot írni, ha ismeri a szintaktikai és szemantikai megszorításait egy-egy kifejezésnek.

A C nyelv számos revíziótól érte el a mai formáját, a C18-as szabványt, de a jövőben várhatóak további verziók is. Mivel minden verzióval bővül a függvények könyvtára, vannak olyan esetek, hogy egy később írt kód nem lesz kompatibilis egy korábban kiadott fordítóval. Erre egy jó példa az snprintf() függvény, mely eltárolja egy bufferben a printf() függvény által a képernyőre kiírt adatokat.

Egy másik újdonság az egysoros kommentek engedélyezése, melyet a C99-es verziótól fogva '//' jelzéssel jelezünk.

```
#include <stdio.h>  
#include <math.h>  
  
int main ()  
{  
    char str[80];
```

```
    sprintf(str, "A Pi értéke: %f", M_PI);
    puts(str);

    return(0);
}      //Ez a program kiírja a standard kimenetre a pi ←
      értékét
```

3.4. Saját lexikális elemző

Sok munkahelyen, főleg, ahol írásos anyagokkal dolgoznak, fontos az, hogy egy adott szöveget gyorsan és kényelmesen lehessen elemezni. Akkor is fontos lehet egy elemzőt használni, ha valamilyen nyelvészettel kapcsolatos kutatást végünk. Ezekre az esetekre készült a python programozási nyelvhez egy nltk (Natural Language ToolKit) nevű könyvtár, mely nagymértékben megkönnyíti az írásos anyagok elemzését és feldolgozását. Ezt a könyvtárat számos kutatáshoz használják a mesterséges intelligencia, gépi tanulás és számítógépes nyelvészeti terén. Tartalmaz lehetőségeket tokenizálásra, tokenekkel való műveletekre, statisztikai adatok kinyerésére és grafikus ábrázolására, valamint szavak automatikus osztályozására is szófaj szerint.

Ezzel a lexikális elemzővel a következőképp lehet összeszámolni egy szövegben található számokat:

```
import nltk
from nltk.tokenize import word_tokenize

text = open('text.txt').read()

tokens = word_tokenize(text)

digit_count = len(set(word for word in tokens if word.isdigit() ←
))

print('The number of digits in the text: ')
print(digit_count)
```

Először megnyitjuk a text.txt fájlt, majd annak tartalmát odaadjuk a text változónak. Ez után szavakra bontjuk az így keletkezett string-et és végigjárjuk őket, kiszűrve azokat, amik csak számból állnak. (Fontos megemlíteni, hogy a set() függvény egy szólistát (szótárat) készít, mely a duplikált szavakat egynek veszi.) A kiszűrt list elemeinek számát a len() függvénytel kapjuk meg, majd a számot beleírjuk a digit_count változóba. Ezután már csak annyi a dolgunk, hogy kiírjuk a kimenetre az eredményt.

A program alapértelmezetten az indítási helyén keresi a text.txt fájlt, de ezt tetszés szerint lehet módosítani, ha igény van rá, hogy más fájlnéven, vagy más könyvtárból szeretnénk bekérni az elemzendő fájlt.

3.5. I33t.I

Egy új, emberek számára viszonylag könnyen olvasható írási mód terjedt el a 80-as években néhány hacker-társaságban. Kicseréltek bizonyos karaktereket szövegekben olyan más, nem ábécé-betűkre, melyek hasonlítottak a reprezentálni kívánt karakterekre. Ilyen például az 'A' betű kicserélése a '4' karakterre. A céljuk az volt, hogy kulcsszavas keresés során ne lehessen könnyedén a fájljaikra, weboldalaikra találni. Azóta széleskörben elterjedt ez az írásmód, melyet leetspeak-nek neveztek, (az angol 'elite' szóból). Manapság számítógépes játékokban és programozói körökben gyakori a használata.

Egy 1337-kódolót a következőképp lehet a python nyelv nltk könyvtára segítségével leprogramozni:

```
import nltk

text = open('text.txt').read()
text = text.lower()

text = text.replace('a', '4')
text = text.replace('b', '8')
text = text.replace('c', '(')
text = text.replace('e', '3')
text = text.replace('g', '6')
text = text.replace('i', '1')
text = text.replace('l', '|')
text = text.replace('o', '0')
text = text.replace('s', '5')
text = text.replace('t', '7')
text = text.replace('x', '8')
text = text.replace('z', '2')

print(text)
```

A program vesz egy inputot (ezesetben a text.txt fájlt), majd kicseréli benne az összes előfordulását a felso-rolt karaktereknek a megadott ekvivalensükre. Ezután elmenti az így keletkezett string-et egy változóba és kírja az output-ra.

3.6. A források olvasása

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

For ciklus, melyben a ciklusváltozó addig növekszik, amíg az kisebb, mint 5.

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

For ciklus, melyben a ciklusváltozó addig növekszik, amíg az kisebb, mint 5. Ugyanaz az eredmény, mint az előző példánál.

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

For ciklus, mely egy tömbbe bekerakja a ciklusváltozó értékét minden iteráció során. Mivel a műveletek sorrendje itt változhat fordítótól függően, kerüljük az ilyen ciklust.

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

For ciklus, mely n-szer fut le és a d pointer értékét odaadja az s pointernek, majd növeli a változók értékét.

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

Kiírunk a standard output-ra két darab kétargumentumú függvény visszatérési értékét. Mivel a két függvény ugyanazokkal a bemeneti adatokkal dolgoznak, nem szerencsés egyszerre futtatni őket, mert nem tudhatjuk, melyik fog előbb lefutni.

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

Kiírunk a standard output-ra egy függvény visszatérési értékét és egy változót.

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

Kiírunk a standard output-ra egy függvény visszatérési értékének a referenciaját és egy változót.

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

- I. minden x-re létezik olyan y, ami prím és nagyobb nála -> végtelen sok prímszám létezik.
- II. minden x-re létezik olyan y, hogy x kisebb, mint y, és y is és y+2 is prím -> végtelen sok ikerprím létezik.
- III. minden x-re létezik y , hogy x prím, ha x kisebb y-nál -> véges sok prímszám létezik.
- IV. minden x-re létezik y, hogy x nagyobb, mint y, ha x nem prím. -> véges sok prímszám létezik.

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;`
Egy egész típusú változó.
- `int *b = &a;`
Egy egész típusú változóra mutató pointer.
- `int &r = a;`
Egy egész típusú referencia, mely a-nak az értékét kapja.
- `int c[5];`
5 elemszámú, egészeket tartalmazó tömb.
- `int (&tr)[5] = c;`
Referencia egy ötelemű, egészeket tartalmazó tömbre, mely a 'c' tömb címére hivatkozik.
- `int *d[5];`
Ötelemű tömb, mely pointereket tartalmaz.
- `int *h();`
Függvény, mely egy pointert ad vissza visszatérési értékként.
- `int *(*l)();`
Pointer, mely egy pointerrel visszatérő függvényre mutat.

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

Függvény, mely bekér két argumentumot, visszaad egy egész típusú változót, majd visszatér egy, a változóra mutató pointerrel.

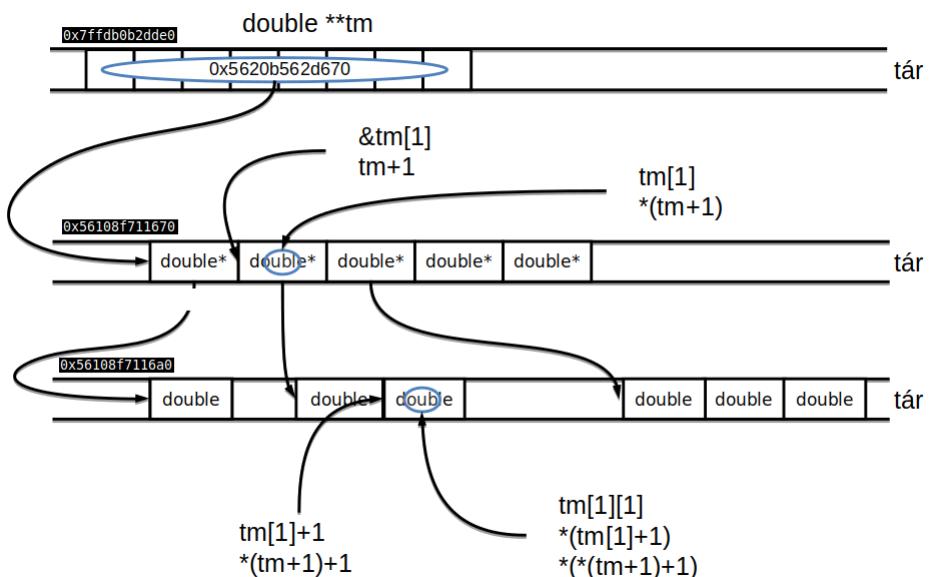
- ```
int (*(*z) (int)) (int, int);
```

Függvény, mely bekér két argumentumot, visszaad egy egész típusú változót, rámutat egy pointerrel, majd arra is rámutat.

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix



4.1. ábra. Többelemű tömbre való hivatkozás pointerekkel

A háromszög-mátrixok olyan kvadratikus mátrixok, melyeknek a főátlója alatt levő elemei csupa nullából állnak, így valamely oldalukat elhagyhatjuk. Ebből adódóan elég csak a számunkra értékes elemeket eltárolnunk. A tárolás módja lehet sor- és oszlopfolytonos is, az egyszerűség kedvéért most ragaszkodunk a sorfolytonos ábrázoláshoz.

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

int
main ()
```

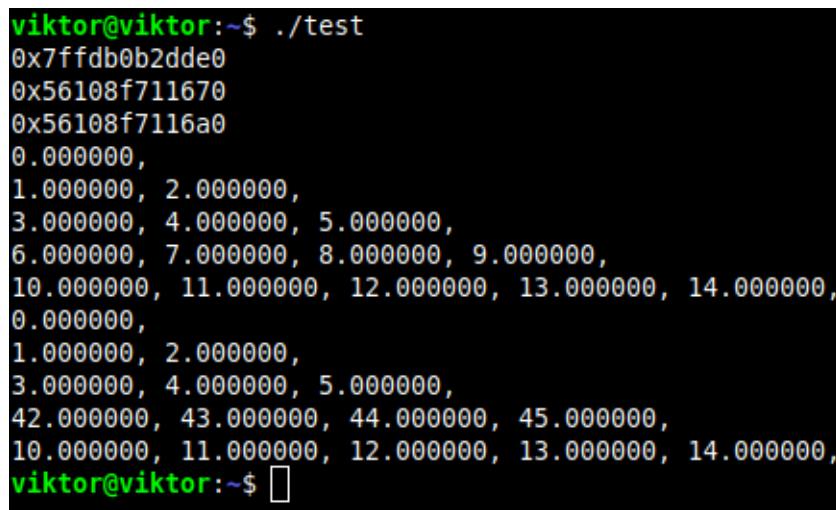
```
{  
    int nr = 5;  
    double **tm;  
  
    printf("%p\n", &tm);  
  
    if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)  
    {  
        return -1;  
    }  
  
    printf("%p\n", tm);  
  
    for (int i = 0; i < nr; ++i)  
    {  
        if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL)  
            )  
        {  
            return -1;  
        }  
  
    }  
  
    printf("%p\n", tm[0]);  
  
    for (int i = 0; i < nr; ++i)  
        for (int j = 0; j < i + 1; ++j)  
            tm[i][j] = i * (i + 1) / 2 + j;  
  
    for (int i = 0; i < nr; ++i)  
    {  
        for (int j = 0; j < i + 1; ++j)  
            printf ("%f, ", tm[i][j]);  
        printf ("\n");  
    }  
  
    tm[3][0] = 42.0;  
    (*tm + 3)[1] = 43.0; // mi van, ha itt hiányzik a külső ()  
    *(tm[3] + 2) = 44.0;  
    *(*tm + 3) + 3) = 45.0;  
  
    for (int i = 0; i < nr; ++i)  
    {  
        for (int j = 0; j < i + 1; ++j)  
            printf ("%f, ", tm[i][j]);  
        printf ("\n");  
    }  
  
    for (int i = 0; i < nr; ++i)  
        free (tm[i]);
```

```
    free (tm);  
  
    return 0;  
}
```

A program egy ötsoros, ötoszlopos háromszögmátrixot fog kiírni a standard output-ra. A működése a következő: Lefoglalunk annyi helyet a memóriában, amennyi szükséges lesz a mátrix reprezentálásához, majd ellenőrizzük, hogy sikeres volt, kiírjuk a kétdimenziós tömbünk elemeinek az indexeit. Ezután kezdjük el feltölteni az adatainkkal a tömböt. Ezt többfélekében is megtehetjük, a forráskódban négyféle verzió látható a negyedik sor négy elemére:

1. A tömb egy elemének értékének explicit módon történő megadása a tömb különálló elemeire hivatkozva
2. A tömbre mutató mutató értékét növeljük annyival, amennyivel arrébblevő tömböt szeretnénk elérni, így a résztömb megfelelő elemére tudunk expliciten hivatkozni. Itt ha lehagyjuk a külső zárójeleket, a program azt fogja hinni, hogy a tm+3-ik sor 1-edik elemére szeretnénk hivatkozni, itt nagyfokú odafigyelés szükséges.
3. Mutatót állítva a harmadik résztömb után kettővel levő értékre
4. Mutatót állítva a tömbben levő harmadik tömbre és annak a harmadik elemére állítva a mutatót (Itt jön be a double** szerepe)

Ha feltöltöttük a mátrixot a kívánt elemekkel, már csak annyi dolgunk van, hogy kiírjuk azt a kimenetre, majd felszabadítsuk a tömbjeinknek lefoglalt tárhelyet.



```
viktor@viktor:~$ ./test  
0x7ffdb0b2dde0  
0x56108f711670  
0x56108f7116a0  
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,  
viktor@viktor:~$
```

4.2. ábra. A program kimenete

4.2. C EXOR titkosító

Sok olyan helyzet adódhat, amikor hasznos, vagy akár elengedhetetlen egy bitsorozat titkosítása. Ha egy fájlunk illetéktelen kézbe kerül, fontos adatok tudódhatsanak ki, ezért érdemes kriptográfiai módszerekkel

titkosítani azt. Erre egy egyszerű megoldás lehet egy xor-titkosítás. Bár nem a legkomplexebb módszer, így egy gyengén titkosított fájlból könnyen ki lehet nyerni a fontos adatokat, gyorsan dolgozik és megfelelő hosszúságú kulcs megadása esetén nehéz nyers erővel feltörni.

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

#define MAX_KULCS 100
#define BUFFER_MERET 256

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

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

    int kulcs_index = 0;
    int olvasott_bajtok = 0;

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

    while ((olvasott_bajtok = read (0, (void *) buffer, BUFFER_MERET)))
    {

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

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

        }

        write (1, buffer, olvasott_bajtok);

    }
}
```

Az algoritmus karakterenként végigfut a titkosítandó fájlon és egy előre megadott kulccsal egy kizáró vagy műveletet hajt végre, azaz a bitsorozatban a megfelelő helyen álló értékekkel összehasonlítja a kulcs megfelelő helyen álló értékével és ha ugyanaz a bitérték, 0-t ad vissza, ha különböző, akkor 1-et. Ezzel bebiztosítjuk, hogy minden megadott kulcs esetén más lesz a kimenet. Ami miatt az algoritmus jól tud működni, az az, hogy ha a kulccsal elvégezzük az xor műveletet a titkosított bitsorozaton, pontosan az eredeti, titkosítatlan adatainkat fogjuk visszakapni.

4.3. Java EXOR titkosító

A következő program egy java implementációja az előző exor-titkosítónak. A forráskód nagymértékben hasonlít első ránézésre a c programhoz, ám vannak bizonyos pontok, ahol eltérnek egymástól. A funkcionalitás ugyanaz, egy szövegen fogunk egy xor műveletet végrehajtani egy kulccsal.

```
public class ExorTitkosító {

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

        byte [] kulcs = kulcsSzöveg.getBytes();
        byte [] buffer = new byte[256];
        int kulcsIndex = 0;
        int olvasottBájtak = 0;
        while((olvasottBájtak =
               bejövőCsatorna.read(buffer)) != -1) {

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

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

            }

            kimenőCsatorna.write(buffer, 0, olvasottBájtak);

        }

    }

    public static void main(String[] args) {

        try {

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

        } catch(java.io.IOException e) {

            e.printStackTrace();

        }

    }

}
```

...

4.4. C EXOR törő

Ha egy xor-titkosított fájlból ki szeretnénk nyerni az eredeti bitsorozatot, csupán annyit kell tennünk, hogy a megfelelő kulccsal elvégezzük az xor műveletet. Ha nem tudjuk a kulcsot, természetesen próbálkozhatunk azzal is, hogy sorra vesszük az összes lehetséges kulcsot és mindegyikkel megpróbáljuk feltörni a titkosítást. Természetesen ehhez ismernünk kell bizonyos tulajdonságait a kulcsnak, pontosabban hogy hány és milyen karakterekből áll és a karakterkódolást. Ezután az összes lehetséges kombinációt végigfuttatjuk a fájlunkon, és a kimenetet figyeljük bizonyos mintákért, amik biztosan megjelennek a kívántt fájlban. Ilyen lehet egy karaktersorozat egy szövegben, vagy egy header valamelyen adatcsomagban.

A kulcs mérete exponenciálisan növeli a maximális műveletszámot, mely egy ilyen módon titkosított szöveg nyers erővel való feltörésére: n^k , ahol n a használt karakterek halmazának számossága, k pedig a kulcshossz. Ha nyolc alfanumerikus karaktert használunk kulcsként, a maximális műveletszám több, mint $2.82 \cdot 10^{12}$, azaz, több, mint kétféle millió. Ezt tovább lehet növelni több, speciális karakter használatával, vagy hosszabb kulcsok megadásával.

Bár a próbálkozások száma soknak tűnhet, ez nem elég olyan alkalmazási területekre, ahol az adatok védelme mindenkor fontosabb. Olyan helyeken azonban, ahol csak egyszerű felhasználók elől szükséges az adatok elrejtése, például játékok fájljainak titkosításakor, ez a módszer tökéletesen megfelel.

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

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

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

    return (double) titkos_meret / sz;
}

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

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

```
}

void exor (const char kulcs[], int kulcs_meret, char titkos[], int ←
    titkos_meret)
{
    int kulcs_index = 0;

    for (int i = 0; i < titkos_meret; ++i)
    {
        titkos[i] = titkos[i] ^ kulcs[kulcs_index];
        kulcs_index = (kulcs_index + 1) % kulcs_meret;
    }
}

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

    return tiszta_lehet (titkos, titkos_meret);
}

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

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

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

    for (int ii = '0'; ii <= '9'; ++ii)
        for (int ji = '0'; ji <= '9'; ++ji)
            for (int ki = '0'; ki <= '9'; ++ki)
                for (int li = '0'; li <= '9'; ++li)
```

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

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

                exor (kulcs, KULCS_MERET, titkos, p - titkos);
            }

        return 0;
}
```

4.5. Neurális OR, AND és EXOR kapu

Tutorált: Ádám Petra

Manapság meglehetősen elterjedt fejlesztési irány lett a gépi tanulás, avagy mesterséges intelligencia. Ez az algoritmus-halmaz a neurális hálókra épül, melynek a következő a lényege: Először a "tanítás" történik meg, mely során adatokat adunk meg, valamint annak tulajdonságait. Ezután a gép felépít magának egy algoritmust, mely alapján később el fogja dönten, hogy a további bemenő adatok is rendelkeznek-e azokkal a tulajdonságokkal. Ekkor kezdődhet el a tesztelés. Bemeneti adatokat szolgáltatunk a számítógépnek, melyeket az kategorizál egy bizonyos szempont alapján, majd eldönti, hogy milyen tulajdonságokkal rendelkeznek. Fontos lehet, hogy megadjunk szempontokat, amik alapján az algoritmus "értelmezi" a bejövő adatfolyamot, hogy a kívánt tulajdonságokra legyen kihegyezve a kategorizálás.

Három fő szintből áll a folyamat: Egy bemeneti szintből, ahol az érkező adatok még kategorizálatlanul vannak, egy rejtett részből, melynek több alszintje lehet, itt különböző kapcsolatok alakulnak ki a bizonyos bemeneti értékek közt, ezek alapján fog majd a kategorizálás történni. Ez a réteg meglehetősen bonyolult, komplexebb neurális hálók esetén már-már követhetetlen emberek számára, ezért lett a neve "rejtett réteg". A harmadik szinten már a kategorizált adatok vannak, melyek végigfutottak a rejtett rétegen, ennek a rétegneknak a neve a kimeneti réteg.

A háló felépítése során először random számokkal súlyozzuk a bemenő értékeket(0 és 1 között) és egy aktivációt függvényt (általánosan szigmoid, vagy tangens hiperbolikusz) alkalmazunk rá, majd végigvisszük

azokat a folyamatban, így olyan értékeket fogunk kapni eredményként, mely a bemeneti számokat valamilyen módon alterálta. Az eredmény alapján a súlyokat tudjuk módosítani, hogy az az adott alkalmazási területre megfelelőbb legyen, így növelte az algoritmus pontosságát.

Az alábbi R program három neurális hálót épít fel a három legtöbbet használt logikai kapu modellezésére, a bemeneti adatok ismeretében meg fogja tudni játszani, hogy az adott logikai kapu műveletének alkalmazása után mi lesz a kapott eredmény.

Az alábbi linken átláthatóan el van magyarázva a neurális kapuk felépítése és az, hogy milyen logika alapján lettek a súlyok megválasztva: <https://medium.com/@stanleydukor/neural-representation-of-and-or-not-xor-and-xnor-logic-gates-perceptron-algorithm-b0275375fea1>

```
library(neuralnet)

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

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

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

plot(nn.or)

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

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

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

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

plot(nn.orand)

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

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

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

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

plot(nn.exor)

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

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

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

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

plot(nn.exor)

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

4.6. Hiba-visszaterjesztéses perceptron

C++

A következő program egy képet vesz inputként, majd pixelenként veszi abban a pixelben a piros szín méretét. Ezt betölti egy neurális hálóba, melynek súlyozása random számokkal van inicializálva. Ez a bitsorozat végigfut a hálón, az visszaad egy számot 1 és -1 között. Ez a szám a súlyozástól függ, ami esetünkben random lett generálva, így minden futáskor más eredményt fogunk kapni. A hiba-visszaterjesztés lényege az, hogy a rendszer önmagát "tanítja", azaz a súlyokat az elérni kívánt eredmény alapján fogja állítgatni. Ezzel a módszerrel közel 99%-os pontosság érhető el bizonyos alkalmazási területeken.

```
#include <iostream>
#include "mlp.hpp"
#include "png++/png.hpp"
int main (int argc, char **argv)
{
    png::image<png::rgb_pixel> png_image (argv[1]);
    int size = png_image.get_width()*png_image.get_height();

    Perceptron* p = new Perceptron(3, size, 256, 1);
    double* image = new double[size];

    for(int i {0}; i<png_image.get_width(); ++i)
```

```
    for(int j {0}; j<png_image.get_height(); ++j)
        image[i*png_image.get_width()+j] = png_image[i][j].red;
    double value = (*p) (image);
    std::cout << value << std::endl;
    delete p;
    delete [] image;
}
```

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

1980-ban egy lengyel matematikus, Benoit B. Mandelbrot felfedezett egy érdekességet a komplex számok halmazán. Ha veszünk egy komplex számot -2 és 2 között, azt a négyzetre emeljük majd az így kapott eredménnyel megismételjük a műveletet, ezt a végtelenségig folytatva két kategóriába sorolhatjuk a számokat: Vagy konvergens, azaz az iterációk során egy adott számhoz tart az eredmény, vagy divergens, mely esetben a kapott szám a végtelenségig nő. Ha ábrázoljuk a komplex számok halmazán azokat a számokat, melyek négyzetes sora konvergens, egy érdekes alakzatot kapunk, melynek számtalan érdekessége van. Az egyik az, hogy a végtelenségig nagyítható, azaz bármilyen közel megyünk valamely elválasztóponthoz, minden fogunk új elágazásokat találni rajta. Az alakzat szimmetrikus az x, azaz a valós tengelyre, de nem az a képzetesre. Rengeteg furcsa tulajdonságot lehet felfedezni például az alakzatból kinyúló karokról, vagy a csomópontok elhelyezkedéséről.

Magunknak is tudunk egy mandelbrot-képet generálni, mely a következő program segítségével lehetséges:

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

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

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

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

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

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

```
if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL)
{
    return -1;
}

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

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

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

tm[3][0] = 42.0;
(*tm + 3)[1] = 43.0; // mi van, ha itt hiányzik a külső ()
*(tm[3] + 2) = 44.0;
*(*(tm + 3) + 3) = 45.0;

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

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

free (tm);

return 0;
}
</programlisting>
<![CDATA[
#include <stdio.h>
#include <iostream>
#include "png++/png.hpp"
#include <sys/times.h>
#define MERET 600
#define ITER_HAT 32000
void
mandel (int kepadat[MERET] [MERET]) {
```

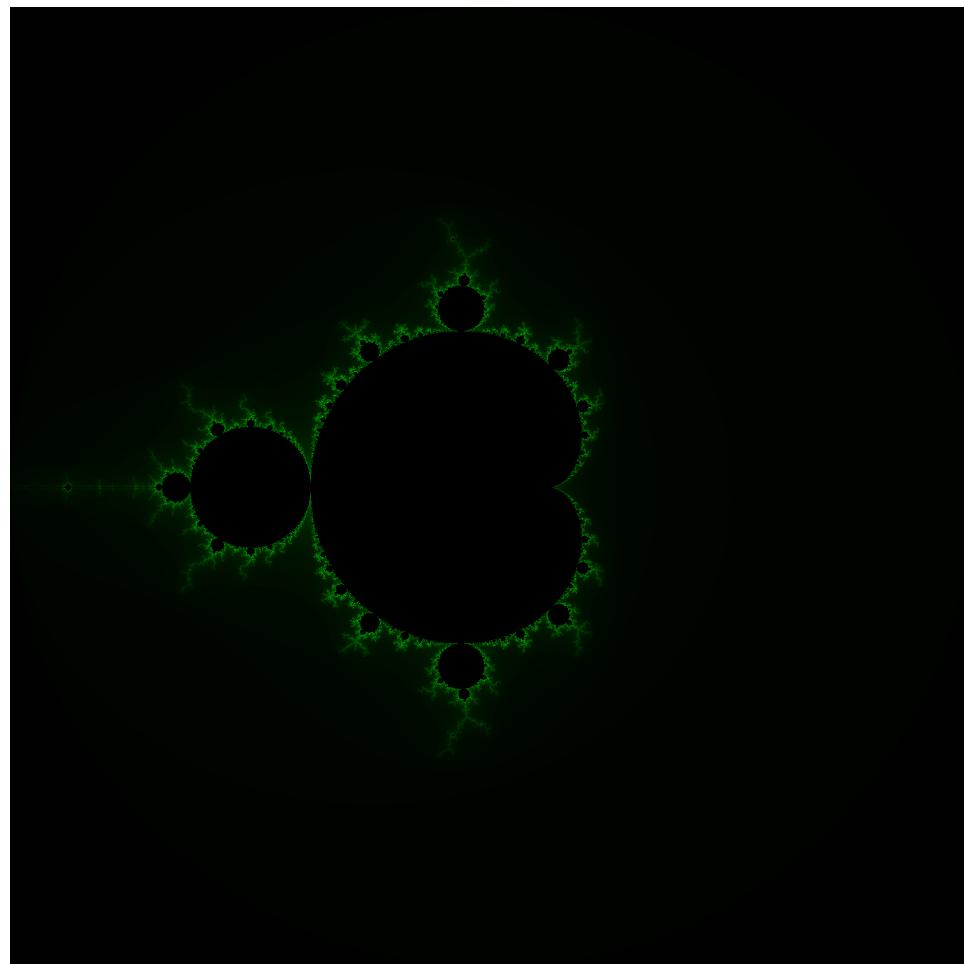
```
clock_t delta = clock ();
struct tms tmsbuf1, tmsbuf2;
times (&tmsbuf1);
float a = -2.0, b = .7, c = -1.35, d = 1.35;
int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;

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

int iteracio = 0;
for (int j = 0; j < magassag; ++j)
{
    for (int k = 0; k < szelesseg; ++k)
    {
        reC = a + k * dx;
        imC = d - j * dy;
        // z_0 = 0 = (rez, imZ)
        rez = 0;
        imZ = 0;
        iteracio = 0;
        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;
        }
        kepadat[j][k] = iteracio;
    }
}
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;
}

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

```
mandel(kepadat);
png::image < png::rgb_pixel > kep (MERET, MERET);
for (int j = 0; j < MERET; ++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;
}
```



5.1. ábra. A Mandelbrot halmaz a komplex síkon

5.2. A Mandelbrot halmaz a std::complex osztályal

Mivel a Mandelbrot-halmaz a komplex számsíkon van értelmezve, ezért egyszerűbb az std::complex osztályt használni a halmaz elemeinek a kiszámolására. Így nem szükséges egy komplex számnak két valós értéket adnunk, egy elemként kezelhetjük a valós és az imaginárius részt. Ezáltal megspórolhatunk felesleges változó-deklarációkat és műveleteket is, melyek során a kódunk egyszerűbb és átláthatóbb lesz.

A következő program komplex számokkal dolgozik, majd egy ppm fájlba írja ki a kimenetet, mely pixelenként tömörítetlenül tárolja el a szükséges színkódokat. Ezáltal magunk is könnyedén tudunk egyszerű, átlátható műveletekkel képeket létrehozni, melyek később természetesen tömöríthetők, valamint átkonvertálhatóak bármilyen más képformátumba.

```
#include <iostream>
#include <fstream>
#include <complex>

using namespace std;

float width = 4096;
float height = 4096;

int value (int x, int y) {
    complex<float> point((float)x/width*4-2, (float)y/height*4-2);
    complex<float> z(0, 0);
    int nb_iter = 0;
    while (abs (z) < 2 && nb_iter <= 128) {
        z = z * z + point;
        nb_iter++;
    }
    if (nb_iter < 128)
        return (255*nb_iter)/128;
    else
        return 0;
}

int main () {
    ofstream my_Image ("mandelbrot.ppm");

    if (my_Image.is_open ()) {
        my_Image << "P3\n" << width << " " << height << " 255\n";
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                int val = value(i, j);
                my_Image << val << ' ' << val/2 << ' ' << val << "\n";
            }
        }
    }
}
```

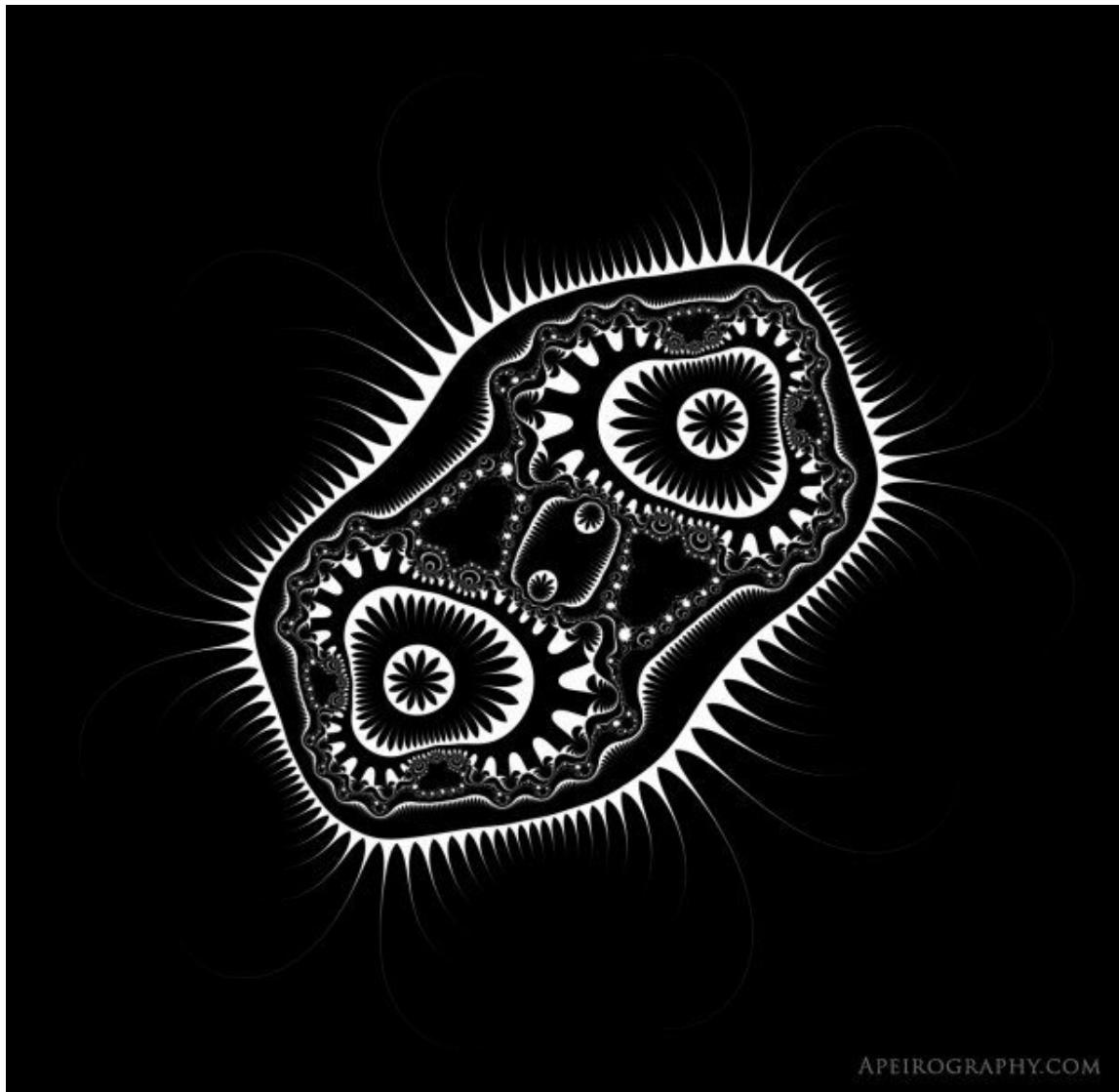
```
    my_Image.close();
}
else
    cout << "Could not open the file";

return 0;
}
```

5.3. Biomorfok

Végtelen sok, a mandelbrot halmazhoz hasonló kinézetű fraktált lehet készíteni különböző függvények megadásával, melyek érdekes alakzatokat rajzolnak ki. Csupán annyi a dolgunk, hogy módosítunk egy-két adatot az előbb bemutatott programunkban és lefuttatjuk az algoritmust.

A 90-es években Clifford Pickover érdekes alakzatokat hozott létre, mikor Julia-halmazokat próbált kirajzolni a számítógépen, de elfelejtette a konstansokat a függvénye végén iterálni. Életszerű alakok jöttek ki eredményként, melyek kinézete nagyban hasonlított az egysejtűekrére. Mikor felfedezte őket, azt hitte, hogy rátalált "az élet bölcsőjére", de mint utólag kiderült, nem ez volt a helyzet. Ezek az alakzatok is "csupán" a komplex síkon való konvergenciavizsgálat eredményei, jelenlegi kutatások szerint nincs szoros összeköttetés különböző életformák felépítéséhez.



5.2. ábra. Biomorph

5.4. A Mandelbrot halmaz CUDA megvalósítása

A CUDA, azaz Compute Unified Device Architecture egy Nvidia által 2007-ben bevezetett technológia, melyet elsősorban videokártyákba építettek be. A chipek egyszerű aritmetikai műveletekre vannak optimalizálva és nagy sebességgel képesek végrehajtani őket. Talán a legnagyobb előny a hagyományos, cpu-alapú számításokhoz képest az egyszerű parallelizálhatóság. Az Nvidia által szolgáltatott toolkit segítségével sokkal egyszerűbb a műveletek több szálra való végrehajtása, így a programozók munkája könnyebb és egyszerűbb lesz. A legjobb példa erre egy for ciklus, melyet Nvidia videokártyák segítségével felbonthatunk külön szálakra, így a műveletek egyszerre képesek végrehajtódni, ezzel gyorsítva a teljes folyamat sebességét. Sajnos én magam nem rendelkezem CUDA technológiával felszerelt kártyával, de amint tudom, kipróbálom a módszert valaki más számítógépénen.

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!

A programot a QApplication könyvtár segítségével írjuk meg, mely lehetőséget ad számunkra egy GUI felépítésére. A cél, hogy a felhasználó saját kedvére tudjon nagyítani a mandelbrot-halmazban.

A programunk több részből fog állni: A main.cpp fájlban inicializáljuk a többi elemhez szükséges programrészeket, valamint megnyitjuk az ablakot, melyben a program fog futni. A frakszal.cpp-ben történik a számolás nagyja, itt fogjuk az inputot vizsgálni, valamint a halmaz értékeit módosítani, a kívánt nagyításnak megfelelően.

Megoldás forrása: https://github.com/takiviko/prog1/tree/master/bhax_textbook_IgyNeveldaProgramozod/Mandel/mandel_nagyito/

5.6. Mandelbrot nagyító és utazó Java nyelven

Ehhez a java GUI implementációhoz három fájl szükséges, az első magát a halmazt fogja számunkra kirajzolni. Ez valójában ugyanaz a program, mint amit már korábban láthattunk, csak java kódba átírva. A második fájl, a MandelbrotIterációk.java, ugyanezen az algoritmuson alapszik, de csak egy szeletét fogja kirajzolni a Mandelbrot-halmaznak. A harmadik, MandelbrotHalmazNagyító.java program fogja majd az egér pozícióját vanni bemenetként és annak pozíciója alapján kirajzolni a megfelelő pozícióban levő értéket a halmaznak. Nagyítani úgy tudunk, hogy az egérrel kijelölünk egy téglalapot a képen, ekkor a program a kijelölt alakzatban elhelyezkedő pontokra fogja kirajzolni a Mandelbrot-halmaz-elemeket.

A három fájlt nem írunk bele a könyvbe, hiszen már az első is négy egész oldalt felöllel, de íme egy link a három program forráskódjához:

Fontos megjegyezni, hogy a fordítás sorrendje nem mindegy először a MandelbrotHalmaz.java, majd a MandelbrotIteráció.java, és végül a MandelbrotHalmazNagyító.java kódot kell fordítani.

https://github.com/takiviko/prog1/tree/master/konyv/mandelbrot/mandelbrot_java

6. fejezet

Helló, Welch!

6.1. Első osztályom

Az osztályok az objektum-orientált programozás legfontosabb elemei. Ezeknek a paradigmáknak a segítségével a három alapfogalmat, az öröklődést, az egyezárást és a többalakúságot is megvalósíthatjuk. Egy osztály deklarációja a class kulcsszó segítségével történik, és hullámos zárójellel zártjuk egybe az elemeit. Egy osztályban több változót és függvényt helyezhetünk el és kezelhetünk egy egysékként. Egy osztály elemeire az [osztálynév].[elemnév] szintaxisossal hivatkozunk. Egy osztályban lehetnek publikus, privát és védett részek is, melyek azt határozzák meg, hogy az adott egység elérhető-e az osztályon kívüli elemek által.

A következő program egy polártranszformációs algoritmus, melynek a header fájljában deklaráljuk az osztályt és egy for ciklussal 10 tagot hozunk létre. Ezután következik a matematikai rész, minden egyes tagban lefut az algoritmus a többitől függetlenül és így 10 különböző eredményt kapunk.

6.2. LZW

Tutor: Ádám Petra

Az LZW algoritmus egy, a 80-as években kifejlesztett módszer bináris adatok tömörítésére. A bejövő inputot bináris alakban meggvizsgálja és egy bináris fa adatszerkezetet épít fel belőle úgy, hogy minden alkalommal, amikor egy olyan substring következik, mely még korábban nem szerepelt a fában, a fa egyik ágát növeli csak, méghozzá azzal az elemmel, amely a már létező substring után következik. Ha a bejövő elem '0' értékű, a bal oldali ágba fog kerülni, ha '1', akkor a jobb ágba. Érdemes megemlíteni, hogy ez egy szigorú értelemben vett bináris fa lesz, tehát vagy 0, vagy 2 ága lesz minden elemnek.

Az alábbi program a standard input-ról veszi az adatokat és azokat rendezи egy bináris fába. minden elem egy struktúra, melyben mutatók vannak a fa következő elemeire, valamint az adott elem értéke, mely 0, vagy 1 lehet.

6.3. Fabejárás

Egy bináris fa bejárásának 3 módja van: Preorder, Inorder és Posztorder. A neveiket a gyökérelem bejárásának az időpontja adja. Mindhárom bejárási módszerrel elérjük a fa összes elemét, de más sorrendben. A preorder módszerrel először a fa gyökérelemezt dolgozzuk fel, majd a bal- és végül a jobb oldali részfáját. Ezzel szemben az inorder bejárási móddal a bal oldali részfát járjuk be először, majd a gyökérelemet és végül a jobb oldali részfát. A posztorder módszer a gyökérelemet utoljára dolgozza fel, míg a részfákat balról jobbra veszi sorra. A kódunkban csupán annyi a dolgunk, hogy egyfajta bejárási módot lekódolunk, majd a többi a sorok megfelelő sorrendbe helyezésével megkapjuk.

6.4. Tag a gyökér

Ha a gyökeret nem csak egy mutatóként, hanem tagként szeretnénk elhelyezni a fában, akkor annyit kell tennünk, hogy a gyökérnek kinevezünk egy elemet és abból építjük tovább a fát, mintha az egy részfa gyökere lenne. Ekkor az értékét egy speciális karakterre állítjuk, ez esetben a "/" jelre. Ez a jel többször nem fog megjelenni a fában hiszen csak 0 és 1 értékeket olvasunk be.

6.5. Mutató a gyökér

Ha mutatóként szeretnénk a gyökér elemet elhelyezni az osztályban, azt úgy tudjuk megvalósítani, hogy a konstruktorkba írjuk bele a gyökér pointer deklarációját. Ekkor a fa mutatójának referenciajával kell azt átadnunk. Ekkor egyes elemeket nem az osztály elemeiként fogjuk tudni elérni, hanem mutatók által. A programunk átírása nem nehéz, a gyökérelemet "csomopont *gyoker" -ként deklárljuk, majd erre építjük fel a fát. Ekkor az eddig "." operátorral hivatkozott jobb és bal oldali mutatókat a "->" operátorral fogjuk tudni elérni.

6.6. Mozgató szemantika

Ha egy, ár felépített fát szeretnénk lemasolni valahova, az első gondolatunk az lenne, hogy a pointerek értékét egyszerűen másoljuk bele egy új fa struktúrába. Azonban ez nem egy jó megoldás, hiszen ha az eredeti fa törlődik, az új fa mutatói nem fognak felhasználható adatra mutatni. Ezért az a helyes megoldás, ha egy teljesen új fát építünk fel és új elemeknek foglalunk helyet a memóriában minden egyes alkalommal, amikor egy új levélhez érünk. Ez talán nem a leg helytakarékosabb megoldás, de szükséges ahhoz, hogy az adataink ne vesszenek el az egyik fából, ha a másikat töröljük.

```
LZWBInFa & operator= ( LZWBInFa && regi )
{
    std::cout << "LZWBInFa move assign" << std::endl;
    std::swap ( gyoker, regi.gyoker );
    std::swap ( fa, regi.fa );
    return *this;
}
```

A mozgató értékkadást az std::move() függvényel tudjuk meghívni. Fontos megemlíteni, hogy ez nem mozgat semmit sehova, csak jobbérék-referenciát csinál a neki átadott struktúrából.

7. fejezet

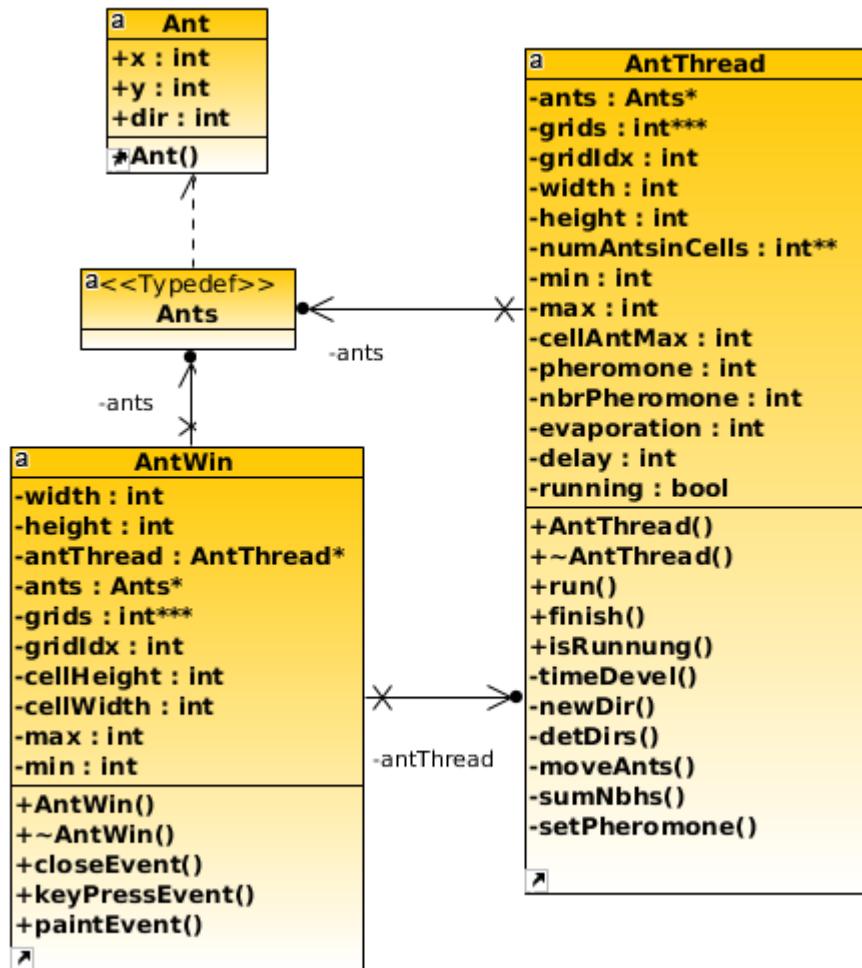
Helló, Conway!

7.1. Hangyszimulációk

A program lényege a valós világban élő hangyák mozgásának szimulációja. Egy adott hangya egy random irányba elindul, feromonokat bocsátva ki magából, majd az alapján dönti el a következő lépés irányát, hogy melyik rácsponton a legerősebb a szag. A feromonok, miután kibocsátja őket egy hangya, folyamatosan párolognak, így nagy eséllyel for egy olyan helyzet kialakulni, hogy a hangyák körbe-körbe "kergetik" egymást.

C++-ban a GUI létrehozásához ismét a QT könyvtárait használjuk. Hat fájlunk lesz, melyekből három header fájl, melyekben a szükséges osztályokat definiáljuk és három 'hajtó' program. Az ant.h-ban lesz a hangya objektum definiálása, annak x és y koordinátái, valamint jelenlegi iránya. Az irányt random inicializáljuk a konstruktőrben. Az Antwin.cpp-ben hozzuk létre a játékteret, itt fognak megjelenni a hangyák egy külső osztályból, valamint a rácsszerkezet, mely a pálya felépítését adja meg. Itt tudjuk megadni a hangyák számát, a pálya méretét, a párolgás mértékét, stb. Az AntThread-ben történik a hangyák mozgásának a vezérlése, valamint a kezdéskor az elhelyezése. A hangyák mozgása, hogy ne legyen túl kiszámítható, kis mértékben randomizálva van. Az utolsó részben (AntThread::run()) elindítjuk a program futását, és egy végtelen ciklussal oldjuk meg, hogy folyamatosan fusson a program. Ha megfelelő inputot kapunk, csak akkor állunk le (esc, vagy p).

UML osztálydiagram a programról:



7.1. ábra. UML

7.2. Java életjáték

John Horton Conway 1970-ben bemutatott egy érdekes algoritmust, melyet sejtsimulátorak, vagy életjátéknak hívott.

A játék menete a következő: egy rácson helyzünk el pontokat (sejtek), majd elindítjuk az algoritmust, mely ciklusokban végigmegy a pályán és előírja, hogy egy cellában lesz-e a következő körben pont, vagy nem. Három szabály van:

1. Ha egy sejtnek kettő vagy három szomszédja van, a sejt túléli a kört.
2. Ha egy sejtnek nulla, egy vagy négy szomszédja van, a sejt meghal.
3. Ha egy cella pontosan három szomszédjában van sejt, ott új sejt születik.

A konstruktorban egy kétdimenziós mátrixot hozunk létre, majd ennek az elemeit fogjuk a szabályok alapján módosítani. Az inputot egy pár beépített függvény segítségével tudjuk vizsgálni, melynek csak közvetlenül indítás után kell megtennünk, hiszen ha fut a játék, onnantól nincs beleszólásunk a dolgok menetébe.

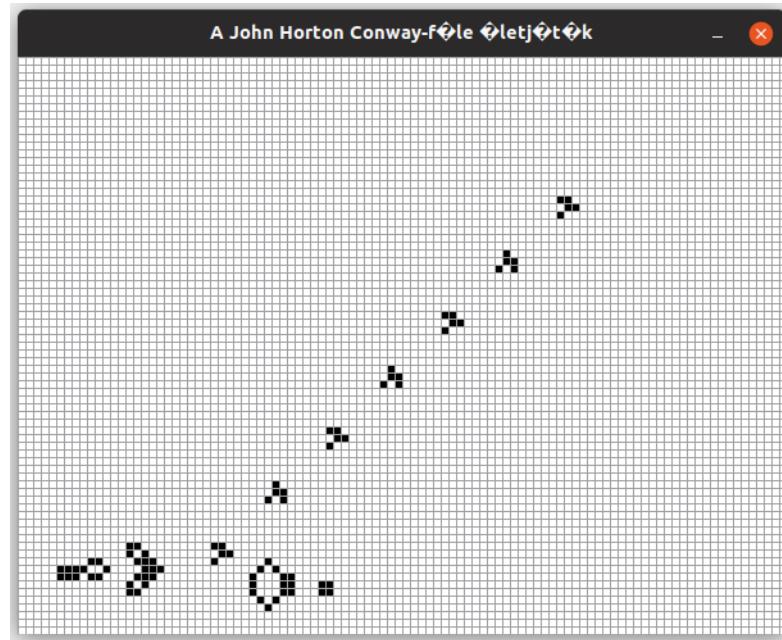
A siklókilövőt bele tudjuk hard-code-olni a programba, hogy később legyen egy alapértelmzőezett indítási helyzet.

A program implementációja java-ban: Conway/eletjatek_java/Sejtautomata.java

7.3. Qt C++ életjáték

A QT könyvtárai segítségével meg tudjuk írni a sejtautomatánkat, mely hat fájlból áll. Az implementáció nagyon hasonlít a java programhoz, de az osztályokat külön fájlban helyezzük el és a java megjelenítője helyett a QT könyvtárakat használjuk.

A main() függvényben elindítjuk a programot, valamint a paraméterlistában átadott számokkal inicializáljuk a pálya méretét. Egy kétdimenziós mátrixot hozunk létre, melyben a számítások alapján fogjuk tudni az értékeket változtatni.



7.2. ábra. Sejtautomata

7.4. BrainB Benchmark

/home/viktor/prog1/bhax_textbook_IgyNeveldaProgramozod/Conway/BrainB

Utólag felhasználom egy passzolási lehetőségem erre a feladatra.

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

A neurális hálók a gépi tanulás alapkövei, és ezáltal egyre nagyobb figyelem fordul feléjük. Többféle módszer is van különböző neurális hálót létrehozó algoritmusok létrehozására, ebben a feladatban a Softmax regressziót használjuk. A célunk, hogy kézzel írt számjegyeket minél nagyobb pontossággal tudjon felismerni a számítógép.

Az MNIST (Modified National Institute of Standards and Technology) egy adatbázis, melyben 28x28 pi- xeles képek, valamint a hozzájuk tartozó címkék vannak kézzel írt számokról. 60.000 kép tartozik az alap adatbázisba, és mellé van 10.000 további, melyet nem használnak fel az algoritmusokban, hanem annak a tesztelésére hagyák meg.

A következő python script ezt a neurális hálót építi fel. A pontossága 90% körül van, ami jól hangozhat, de bőven lehet még rajta finomítani.

8.2. Mély MNIST

Tutoriált: Ádám Petra

A neurális hálók pontosságának javítása érdekében több "titkos" réteget vezethetünk be, melyek, bár exponenciálisan fogják növelni az algoritmus komplexitását, meglehetesen nagy javulást okozhatnak a hibaszá- zákok csökkentésében. Ilyen titkos réteggel már találkoztunk, mikor egy xor kaput próbáltunk neurális hálóval szimulálni. Minél több ilyen réteget rakunk a programunkba, annál finomabban tudjuk majd a rétegek súlyozását alakítani.

DEEP MNIST PROGRAM

8.3. Minecraft-MALMÖ

A Minecraft Malmö projekt egy kezdőknek szóló, Microsoft által elindított neurális háló-fejleszto program. A célja, hogy játékosan ismertesse meg a leendo fejlesztőkkel egy háló felépítését és működését.

A projekt egyáltalán nem kötött, mindenki saját magának szabhat meg egy célt, amit megpróbál megvalósítani. Ilyenek lehetnek egyszerű feladatok, például egy szobában a leheto legrövidebb úton ejlutni A-ból B-be, vagy a generált világban tájékozódni.

A projekthez biztosítanak egy python api-t, melynek segítségével könnyen lehet a játékbeli paramétereket vizsgálni. A szükséges források ehhez a <https://github.com/Microsoft/malmo> címen találhatóak.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

A Lisp az egyik legrégebbi mai napig is használt programnyelv, még a 60-as években fejlesztették ki. Érdekkessége, hogy minden kódöt listaként kezel, így a kezdetekben ideális volt a lyuksalagos háttértár elterjedt használata miatt. A függvények prefix alakban írandóak, így például két szám összeadását a következő módon kell elvégezni:

(+ 2 3)

A LISP nyelvnek több dialektusa is van, ilyen a Clojure, vagy a Scheme, melyet a GIMP is használ script-ek formájában.

Egy faktoriálist kiszámoló rekurzív függvény deklarációja a következőképpen néz ki:

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))) ) )
```

Vegyük észre, hogy a műveletek és a függvények is prefix alakban írandóak.

9.2. Gimp Scheme Script-fu: króm effekt

A GIMP képszerkesztő program a LISP Scheme nyelvjárását használja script-ek kezelésére, melyekkel effekteket sorozatát tudjuk egy képhez hozzáadni. A folyamat menete nagyon egyszerű, annyi a teendőnk, hogy felsoroljuk, milyen effekteket szeretnénk milyen elemekre alkalmazni. A szintaxis talán kissé furcsa lehet, de ha sikerül követni a nyelv szabályait, a programozás része nem lesz nehéz. A script folyamán létrehozunk egy képet, beállítjuk az alapértelmezett színeit, majd egy új réteget adunk hozzá, melyen a szöveg fog elhelyezkedni. Ezután Gauss-féle elmosást alkalmazunk rá, beállítjuk a megfelelő függvényeket a szöveg effektjeire, végül megjelenítjük a képet. A script végén információkat találhatunk önmagáról, ha kiválasztjuk a menüből, milyen információkat tudhatunk meg.

9.3. Gimp Scheme Script-fu: név mandala

A következő script hasonlít az előzőhöz, de több, komplexebb lépésből áll. Itt is beállítjuk az alapértelmezett háttérszíneket, majd beszúrjuk az egyik szöveg-réteget. Ide beillesztjük a kívánt szöveget több alkalommal, minden beszúráskor elforgatva azt. Az effekteket rárakjuk a rétegre, majd beszúrjuk az utolsó réteget, mely a második szövegrész lesz és a kép közepén helyezkedik majd el.



9.1. ábra. Mandala

10. fejezet

Helló, Gutenberg!

10.1. Juhász István - Magas Szintű Programozási Nyelvek 1

[?]

A könyv 2008-as kiadású és a Debreceni Egyetem tantervi hálója alapján lett felépítve. Az első fejezetben az informatika céljáról esik szó, bár érdekes módon egy objektum-orientált szemlélettel tekint a világra. Az absztrakció paradigmáját részletesen megfogalmazza, hiszen erre épül a programozási (és nem programozási) nyelvek mindegyike. A programozási nyelveket három kategóriába sorolja: Gépi nyelv (számunkra nem olvasható), assembly szintű nyelv (nehezen, de átlátható) és magas szintű nyelvek (emberek számára olvasható, itt is előjön egyfajta absztrakció). Egy fordítóprogram magas szintű nyelvet fordít le gépi nyelvre, ezáltal tudjuk majd futtatni a számunkra átlátható nyelven megírt programunkat. A magas szintű nyelvet is lehet alcsoportokra bontani, ezek lesznek az imperatív (C, C++, Java, stb.) és a deklaratív (Prolog, SQL, stb.) nyelvek. A imperatív nyelvekben a programozónak magának kell a saját algoritmusait megírni, míg a deklaratív nyelvekben általában meg vannak adva előre a nyelv lehetőségei. A második fejezetben a nyelvek alapelemeit ismerhetjük meg. Szó lesz egy nyelv ábécéjéről: a változók nevében használható karakterektől, a szimbolikus karakterekről, konstansokról. A szerző bevezeti az adattípusok (egyszerű és összetett), a tömbök, mutatók, nevesített konstansok, és változók fogalmát. A harmadik fejezetben lesz szó a kifejezésekről, melyek operandusokból, operátorokból és zárójelekből állnak. A három írásmódról is szó esik, bemutatásra kerül a prefix, infix és postfix írásmód is. A C nyelvben vannak egyoperandusú operátorok is, mint a '++' operátor, mely eggyel növeli egy változó értékét egy adott utasítás előtt, vagy után attól függően, hogy az operátor a változó előtt vagy után helyezkedik el. Megismerkedhetünk a műveletek végerhajtási sorrendjével, majd mindegyikről egy rövid leírást kapunk. A negyedik fejezet az utasításokról szól, azaz a különböző értékkedásokról, ugrásokról, elágazási lehetőségekről, ciklusokról, hívásokról, i/o utasításokról, stb. egy programban. Itt nem lesz konkrét nyelvi példa C -ben, hiszen a könyv 4 különböző nyelvet fed le, és rengeteg oldal csak kódiból állna. A három kulcsszó, amivel megismertet a könyv, az a 'continue', a 'break' és a 'return'. A programok szerkezetéről szól az ötödik fejezet, a programmegségekre bomlás logikáját magyarázza el. Itt lesz világos, hogy az eljárásvagyorientált nyelkeknek mi az előnye. Itt ismerkedhetünk meg az alprogramok fogalmával. Ez csak egy különböző név a saját magunk által létrehozott eljárásvagyoknak és függvényeknek. Ezeknek négy komponense van: név, paraméter-lista, törzs és környezet. A rekurzió fogalmával is megismertet a könyv, ez az a programozási eszköz, amikor egy program saját magát hívja meg(általában megváltozott paraméterekkel). Többféle paraméter-átadási típus létezik. Egy alprogramnak adhatunk érték szerint, cím szerint, eredmény szerint, érték-eredmény szerint, név szerint és szöveg szerint. A C nyelv csak egy paraméter-átadási módot ismer, a paraméter típusától függ, hogy

melyik fajta fog történni. A következő részben hosszasan kifejti minden egyes nyelvben a blokk és hatáskör fogalmát, de a C nyelvnek csak két oldalt szán. Itt lesz szó a különböző szabályokról és az ezek kikerülésére használható kulcsszavakról (extern, auto, register, static). Az absztrakt adattípusokról nem tudunk meg sokat, csupán annyit, hogy egy módszer az információ elrejtésére, és hogy egyre népszerűbb az alkalmazása. A kivételkezelésnek is külön fejezet van szentelve, ez a paradigma segíthet kiküszöbölni a programban levő futási, valamint észrevenni az esetleges szemantikabeli hibákat. A könyv nem hoz a C nyelvből példát, hiszen a kivételkezelés nem része a nyelvnek, de megoldást nyújthatnak egy bizonyos mértékig egyszerű if-else utasítások is. A párhuzamos programozás a C nyelvbe be van építve, egy új szálat a fork() utasítással tudunk nyitni. Ennek a segítségével tudunk például több magon futtatni egy ciklust egy programban (lásd: végetlen ciklus több magon). Az I/O kezeléséhez is új fejezetet nyitott a szerző, ez talán az egyik legfontosabb, bár nem a legnehezebb eleme a programozásnak, hiszen valamilyen módon tudnunk kell kommunikálni a programunkkal. Kétféle adatátviteli mód létezik: folyamatos (konvezióval) és rekord módú (binárisan történő). Ez csupán annyit jelent, hogy például a printf() függvény típuskonverziót hajt végre egy szám kiírásakor (innen jön a név - print formatted). Lesz itt szó az állománykezelésről (nyitás, feldolgozás, zárás) és a különböző nyelvi eszközökkről is. A könyv végén még szó esik röviden a memóriakezelésről, de itt már nem találkozunk egy nyelvben sem konkrét példával, kizárálag elméleti szinten mutatja be a nyelvek ezen elemeit.

10.2. B. W. Kernighan, D. M. Ritchie - A C programozási nyelv

Az első fejezetben azonnal rátér a könyv a programozásra és egy "hello world" program elkészítésében segít az olvasónak. Megadja, hogy mire kell odafigyelni, valamint hogyan kell lefuttatni a programot egy UNIX rendszeren. Itt már előjönnek olyan dolgok, amiket még csak később fog elmagyarázni a szerző, de egy gyors áttekintést ad róluk a teljesség kedvéért. A változókat sajátos módon vezeti be, nem magyarázza el azonnal, hogy mik azok, hanem rögtön egy példaprogramot ad a felhasználó kezébe. Miután a működését értelmezte, azután fog csak plusz információt kapni az olvasó a kifejezések jelentéséről. Itt bevezetésre kerülnek az int, float, short, long, double és a char típusú változók. Ekkor a while ciklus is be lesz vezetve, melyek segítségével megírhatjuk az első valóban hasznos programunkat. v /* FahrenheitCelsius táblázat kinyomtatása f = 0, 20, . . . , 300 értékekre */ #include <stdio.h> int main () { int lower, upper, step; float fahr, celsius; lower = 0; /* A hőmérséklettáblázat alsó határa */ upper = 300; /* felső határ */ step = 20; /* lépésköz */ fahr = lower; while (fahr <= upper) { celsius = (5.0 / 9.0) * (fahr - 32.0); printf ("%4.0f %6.1f\n", fahr, celsius); fahr = fahr + step; } } A következő bemutatott része a C nyelvnek a karakterek ki- és bevitelle lesz, méghozzá a putchar() és a getchar() függvényeken keresztül. Itt ismerkedünk meg a ++i és az i++ kifejezésekkel is. Ezekkel kapunk pár példát különböző inkrementálási megoldásokra, és végül szó esik az 'or' logikai operátor használatáról. A következő téma kör a tömbökkel foglalkozik, deklarálásukkal, rajtuk végezhető műveletekkel. Ezután a függvényekről lesz szó, melyek talán a legfontosabb elemei egy programozási nyelvnek. Ehhez kapunk példákat is, és hamar megismerkedhetünk az argumentumokkal és az érték alapján való hívással. A 9. részben a szerző egyesíti az eddig megismert részeit a nyelvnek és egy példaprogramot tár elénk, mely tartalmaz függvényeket, tömböket, relációs operátorokat és ciklusokat is.

```
#define MAXLINE 1000 /*A beolvasott sor maximális mérete*/
/*Sor beolvasása s be, a hosszát adja vissza*/
getline (char s[], int lim)
{
```

```
int c, i;
for (i = 0; i < lim    1 && (c = getchar ()) != EOF
&& c != '\n'; ++i)
s [i] = c;
if (c == '\n') {
s [i] = c;
++i;
}
s [i] = '\0';
return (i);
}
/*s1 másolása s2-be; s2-t elég nagynak feltételezi*/
copy (s1 [], s2 [])
{
int i;
i = 0;
while ((s2 [i] = s1 [i]) != '\0')
++i;
}
/*A leghosszabb sor kiválasztása*/
main ()
{
int len; /*A pillanatnyi sor hossza*/
int max; /*Az eddigi maximális hossz*/
char line [MAXLINE]; /*A pillanatnyilag olvasott sor*/
char save [MAXLINE]; /*A leghosszabb sor mentésére*/
max = 0;
while ((len = getline (line, MAXLINE)) > 0)
if (len > max) {
max = len;
copy (line, save);
}
}
if (max > 0) /*Volt sor*/
printf ("%s", save);
```

Ezután még egy kicsit elmélyedhetünk a különböző változók "élettartamában", hogy pontosabban megértsük a fordító működését és hogy a későbbi félreértéseket elkerüljük. Fontos megjegyezni azonban, hogy bár van rá lehetőségünk, a globálisan deklarált változókat erősen ellenzi a legtöbb programozó, hiszen könnyen megváltoztathatja az értéküket egy olyan függvény vagy eljárás, melynek nem lenne szabad elérne az adott változót. Ezután már csak az összefoglalás marad hátra, itt egy pár olyan feladatot ad a könyv, melyhez az eddig tanult tudás elegendő. A típusokról és a kifejezések ról a második fejezetben esik szó részletesen, majd megtudhatjuk, hogy bizonyos architekrúrákon milyen módon vannak tárolva. A különböző változókon elvégzett műveleteket 8 fejezetben át tárgyalja a könyv, itt szó esik az aritmetikai és logikai operátorokról, relációkról, inkrementálásról és típuskoverzióról is. A fejezet végén egy precedencia-táblát mutat be, melyben fentről lefelé haladva vannak rendezve a különböző elvégzhető műveletek, azoknak "erőssége" alapján. A harmadik fejezetben részletesen megismertet a könyv a különböző feltételes utasításokkal,

melyek szinte minden programban elengedhetetlenek. Itt szó esik a következő utasításokról: if-else else-if switch while, for do-while break, continue, goto Bár már látott és írt is az olvasó ilyen utasításokat, ebben a fejezetben részletesen be van mutatva a használatuk és lényegük. A negyedik fejezet a függvényeknek van szentelve, itt bemutatja az író az összes olyan lehetőséget, melyekkel dolgozhatunk egy program írásánál a függvényekben. Itt lesz szó a blokkstruktúráról, a változók kezeléséről majd végül a rekurzióról is. A mutatók talán a c nyelv legjellegzetesebb elemei, segítségükkel rengeteg lehetőség áll a rendelkezésünkre, viszonylag kis erőforrás-igénnyel. Nem csoda tehát, hogy a teljes ötödik fejezet ezeknek a különleges változóknak van szentelve. Itt leginkább a tömbök kezeléséről, az elemek között való műveletekről lesz szó. Itt láthatunk példákat egyszerű változók címével való műveletekre, tömbök bejárására mutatókkal, valamint a parancssori argumentumokkal is találkozhatunk. A fejezet végén röviden megtárgyaljuk a függvénymutatók célját, használatát. A hatodik fejezet a struktúrákról szól. Először a könyv bevezeti a fogalmat, majd egy pár egyszerűbb példát hoz a lehetséges alkalmazási módokra, miközben a szintaxist bemutatja. Szót ejt az önhivatkozó, azaz rekurzív struktúrákról, majd az unionok és típusnévdefiníciók fogalmát vezeti be. Ezután lesz csak szó a be- és kimenetről, azaz a tárhellyel és általában az I/O-val foglalkozó függvényekkel. Itt részletesen tájékozódhat az olvasó a printf(), scanf(), getchar(), putchar(), stb. függvények használatáról, működéséről. Itt esik szó a rendszerhívásokról, melyek a UNIX alapú operációs rendszerek fontos elemei, valamint a tárhelykezelésről, melyet minden programozónak helyesen kell tudnia használni. Ide tartoznak az alloc, calloc, malloc függvények. Az utolsó, fejezetben az UNIX operációs rendszerrel való kommunikációról esik szó, itt leginkább a tárhellyel való kommunikálás problémáinak megfelelő kezelése a cél, az open(), read(), close() függvényekkel. A RAM és a háttértár eléréséről sok részlet kimaarad, de az olvasó, aki idáig eljutott, valószínűleg más forrásból is szerzett tudást, így ezzel a téma körrrel valószínűleg nem lesz problémája a karrierje során. A könyv végén a függelékben egy majdnem ötvensoros összefoglalót találunk a megtárgyaltyelvi eszközökről, melyet gyorsan, könnyen át tud nézni az olvasó, ha valami probléma adódik programozás közben, miután elolvasta a könyv többi részét. Itt összegyűjt a szerző a fontos kifejezéseket, utasításokat és a szintaxisukat. A könyv lezárás nélkül marad, talán azt szimbolizálva, hogy a nyelv folyamatos fejlődése miatt nem lehet teljes, minden egybefoglaló olvasmányt készíteni.

Alapismeretek

Ebben a fejezetben az olvasó megírja az első programját, mely egy egyszerű "Hello World" típusú program. Itt szó esik a különböző speciális kifejezésekéről, mint a "\n". Ezután megismerkedünk a programozás legfontosabb elemével, a változó-értékkedással. Ekkor találkozunk először a változó-típusokkal és a rajtuk végzett műveletekkel. Itt már a while ciklus is előjön, de még nincs részletesen elmagyarázva. A harmadik alfejezetben a for utasítás jön elő, melyet a már megismert while ciklushoz hasonlít a könyv. Ezután láthatunk példákat szimbolikus állandókra, és láthatjuk a változóktól való különbségüket. Láthatjuk továbbá a getchar() és a putchar() függvények használatát, valamint kicsit specifikusabb példákat velük kapcsolatban. A hatodik alfejezetben a tömbökkel ismerkedhetünk meg, valamint az if/else kifejezéssel. Itt is találhatunk példákat különböző alkalmazásaira ezeknek a nyelvi elemeknek. A következő téma a függvényekkel kapcsolatos, itt láthatunk példát egyszerű hatványozó függvényre, valamint a függvények tulajdonságairól is szerezhetünk ismereteket (érték szerinti hívás). A kilencedik alfejezet a karaktertömbökről szól, bár még a string-ekről nem esik szó. Az utolsó részben a külső változókról esik szó, azoknak a használatáról és céljukról. Az összefoglalás részben néhány gyakorló-feladatot kap az olvasó, melyek az eddig megismert kifejezésekhez, paradigmákhoz kapcsolódnak.

Típusok, operátorok, kifejezések

A fejezetet a különböző változótípusok leírásával, jellemzésével kezdik a szerzők, majd a karakterláncokról, állandókról lesz szó. Miután ezeknek a deklarációjáról is esett szó, a különböző operátorok és használatuk van bemutatva.

10.3. Benedek Zoltán, Levendovszky Tihamér - Szoftverfejlesztés C++ nyelven

[BMECPP]

A könyv bevezetőjében megtudhatjuk a c++ nyelv leglényegesebb különbségeit a többi objetum-orientált nyelvtől, valamint rövid összefoglalót kapunk a c nyelvből örökolt elemekről. Feltételezi a szerző, hogy az olvasónak már van valamennyi tapasztalata a C nyelvvel, és azonnal a különbségeket kezdi el részletezni. Az első valóban a nyelv elemeiről szóló fejezetben a függvényekkel foglalkozik (itt is látszik a különbség a Kernighan-Ritchie könyvtől: Ott a változók, adattípusok voltak az első téma), és a C-vel való viszonyát fejti ki, valamint az újdonságokról ejt szót, mint például az üres (void) paraméterről. Bemutat egy új változótípust, a "bool"-t, mely csak 1, vagy 0 értéket vehet fel. A függvények túlterhelését is fontos megemlíteni, két függvénynek lehet azonos neve, ha más a paraméterlistájuk. A referencia szerinti paraméter-átadás is újdonság, ennek a segítségével egyszerűbben, átláthatóan kezelhetjük a változóinkat. A harmadik fejezetben az objektumorientáltságról esik szó, mely szintén egy olyan elem, mely a C nyelvben nem elérhető. Ennek a meglehetősen népszerű programozási szemléletnek az alapja, hogy minden univerzumelem felfogható egy adathalmaznak, melyen függvényekkel és eljárásokkal végezhetünk műveleteket. A paradigma három fő alapfogalmáról is szó esik, az egységbézárásról, az öröklődésről és a többalakúságról. A konstruktorkal és destruktorkal a 3.4-es alfejezetben ismerkedhetünk meg részletesebben, ezek azok a részei egy osztálynak, melyek az osztály minden meghívásakor, illetve felszabaduláskor lefutnak. Ezekben tudunk változókat deklarálni, vagy különböző műveleteket végezni már meglévő struktúrákkal (pl. mozgató szemantika - LZWBinFa). A dinamikus tagok osztálybeli kezeléséről is lesz szó, ezekkel vigyázni kell, sok eset lehetséges, amikor dupla felszabadítás, vagy többszörös adatmódosítás történik. Erre jó példát hoz a szerző, melyben egy változóra több pointer is mutat, és ezáltal több módon lehet elérni. Ezzel az a baj, hogy könnyen "elromolhat" a változónk. A "Friend" típusú függvényekről és osztályokról szól a következő alfejezet, melyek olyan függvények és osztályok, melyek el tudják érni egy másik osztály privát tagjait is. A statikus tagok lesznek a következő téma, melyek olyan konstans változók, melyek egy osztály minden objektumában azonos értéket vesznek fel. Érdekesség, hogy a memóriában egy változónyi hely van nekik foglalva a helytakarékos céljából. A negyedik fejezetet a konstans változók ismertetésével kezdi a szerző, ezek olyan változók, melyeknek a program futása során nem változhat meg az értékük. Konstans pointerek deklarálására is van lehetőségünk, ezeknek sem lehet megváltoztatni az értékét, azaz minden esetben ugyanarra a memóriacímre fog mutatni. Konstans változókat egy osztályban az inicializási listában kell létrehozni. Konstans tagfüggvények is léteznek, ezek paraméterül átadott változóit nem lehet módosítani. A C++ nyelv Input/Output függvényei kissé máshogy működnek, mint C-ben, itt a bitshift operátorokkal történik a ki- és beolvasás. (Talán hasznos lehet megemlíteni, hogy van lehetőségünk a c nyelv függvényeit használni, ezek egy külön header fájlban vannak elhelyezve, pl. 'cstdio.h'). Az állománykezelés hasonlóan működik, min C-ben, bár itt is van lehetőségünk az imént megismert operátorokat használnunk, ha szípaticusak. Az operátoroknak egy teljes fejezetet szánt a szerző, hiszen velük kapcsolatban sok újdonság van a C++ nyelvben. Az egyik legerősebb eleme a nyelvnek az operátor-túlerhelés (operator-overload), mellyel egy operátor viselkedését tudjuk megváltoztatni (ismét, lásd: mozgató szemantika). A típuskonverzió is egy hasznos eleme a nyelvnek, Bár a C nyelvhez képest itt nincs sok különbség, itt fordítva működött a fejlődés irányába: először lettek a c++ nyelvben megalkotva az újítások, és azután vette át a C nyelv is ezeket az elemeket. Névterekkel el tudjuk választani az azonos nevű és paraméterlistájú függvényeket, így azok egyértelműek lesznek. Ez egy magányos programozó számára nem feltétlenül hasznos, de egy cégen, vagy vállalaton belüli munkát képes megkönnyíteni. Én személyesen egyszerűbb programoknál szívesen hívom meg az std névteret a 'using' kulcsszóval, így később nem kell majd minden egyes i/o műveletnél használnom. A kivételkezelés bonyolultabb programoknál hasznos lehet, ennek a segítségével könnyebben lehet

megalálni az esetleges hibákat a programunkban. Ezeket a 'try', a 'catch' és a 'throw' kulcsszavakkal tuduk használni. A könyv ezután A sablonokat fejti ki a maradék ~100 oldalban. Ezek olyan elemei a nyelvnek, melyek segítségével egy adott osztályt vagy függvényt több adattípus esetén is tudjuk használni. Ilyenkor csak egyszer kell deklarálnunk az osztályt/függvényt, és később az átadott paraméterek fogják meghatározni a viselkedését. A könyv végén egy gyors összefoglaló van az operátor-precedenciáról, a különböző ajánlott fejlesztői környezetekről (Visual C++, linux os), valamint egy táblázatot is találhatunk a különböző C utasítások C++ megfelelőiről.

III. rész

Második felvonás

**Bátf41 Haxor Stream**

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

11. fejezet

Helló, Berners-Lee!

11.1. A C++ és a Java nyelvek összehasonlítása

A c++ és a Java nyelvek szintaxisa között első ránézésre lehet, hogy nem sok különbséget fogunk látni, de felépítésükben és működésükben sokféleképpen eltér a két programozási nyelv. Más célból alkották meg őket, a java nyelv lényege, hogy egyszer kell megírnia a programozónak a kódot és minden platformon fogja tudni futtatni a JVM, azaz Java Virtual Machine segítségével. Ezzel ellentétben a c++ nyelv a C nyelv kiterjesztése, az objektum-orientáltság lehetőségével, referenciákkal és egyéb gyakran használt elemekkel bővítették ki a nyelvet, de meghagyták a "régi", alacsonyabb szintű paradigmákat is, mint a mutatók. Az objektum-orientáltság használata a c++ nyelvben a programozó szabad döntése, ám a java nyelvben csak objektum-orientált szemléettel lehet programozni. Erre jó példa az, hogy egy egyszerű "hello world" programhoz is egy main osztályt kell létrehozni, melybe az egész program be lesz ágyazva. A c++ nyelvben olyan fogalmakkal is találkozhatunk, melyeket manapság nem sok helyen használnak, de a C nyelvből lett örökölve, így benne maradt a nyelvben, ilyen például az unió és a struktúrák, melyeknek egyszerűen helyettesíthetők osztályokkal. A java nyelv, mivel egy virtuális környezetben fut, "biztonságosabbnak" mondható, ez annyit jelent, hogy a memóriakezelés automatikusan történik, a programozónak nincs lehetősége annyi memóriát allokálnia például egy struktúrának, amennyit ő szeretne. Ebből adódóan a "memory leak" sokkal kevésbé fenyegető jelenség, viszont lényegesen kevesebb szabadsága van a programozónak, ezáltal sok kód kevésbé optimalizált és lassabban fut. A biztonságossága a java nyelvnek a futási időben bekövetkezett hibákra is kiterjed, például 0-val való osztáskor hibaüzenetet fogunk kapni a rendszertől, míg a c++ nyelv nem tudja megfelelően kezelni a helyzetet. A több szálon való programozásra is ad lehetőséget a java, ezt elég nehézkes a c++ nyelvben megvalósítani, mely ezen a téren a C nyelv sajátosságait örökölte.

11.2. A Python3 programozási nyelv

A python egy interpreteres, általános használatú magas szintű programozási nyelv, melyet sokan szeretnek az egyszerű, szabados szintaxisa miatt. Sokan ajánlják kezdőknek, mivel a fejlesztés során elsődleges cél volt a kód olvashatósága, ugyanekkor sokan el is vetik emiatt, mondván, hogy nem elég specifikus, sok dolog nem egyértelmű. Erre egy jó példa az, hogy változókban nem kell a programozónak deklarálnia, ezt automatikusan elvégzi a rendszer. Ennek az az előnye, hogy nem kell ismerni a különböző változótípusokat, ám lehet, hogy egy haladó programozó nem fogja tudni pontosan azt leprogramozni, amit szeretne. Több verziója is van a python-nak, a legújab és legelterjedtebben használt a python 3, de sokan ragaszkodnak a

python 2-höz. Vannak programok, amik csak a python 2-ben működnek, emiatt vannak olyanok, akik nem szívesen váltanak. A legújabb csomagok azonban csak python 3-on érhetőek el, így ha a régi programokkal való kompatibilitás nem fontos, érdemes a 3-as verzióval kezdeni.

12. fejezet

Helló, Arroway!

12.1. Objektum-Orientált Szemlélet

Az objektum-orientált programozás azok számára kissé idegen lehet, akik azelőtt kizárolag procedurális programozással foglalkoztak, mert olyan magas szintű paradigmákat vezet be, melyek elsőre talán nem tűnnek hasznosnak, vagy könnyen implementálhatónak. Azonban ezeket a feltevéseket semmi nem cátfolja jobban, mint a tény, hogy a legtöbb manapság írt program objektum-orientált szemlélettel íródik. Ez szerves része olyan elismert és elterjedt nyelveknek, mint a Java, C#, Python, Ruby, és a lista folytatódik látszólag a végtelenséggel. A kissé absztraktabb felfogás sokak számára könnyebbéget jelent, főleg a kezdő programozóknak, vagy azoknak, akik nem kívánnak teljesen belemerülni a számítástudomány rejtelmeibe. Az alapvető koncepciókat, mint változók, függvények, stb. megtartja, de kiegészíti osztályokkal, virtuális függvényekkel és absztrakt típusokkal. Három alapötlet mentén tudunk objektum-orientált programokat írni: Egybezárás, többalakúság és öröklődés. Ha ezek mentén írjuk a programjainkat és betartjuk a hozzájuk kapcsolódó szabályokat, hamar rájöhetünk, hogy tulajdonképpen nem is olyan nehéz átláttni ezt az erősen absztrakt gondolatmenetet még egy újoncnak sem.

Az alábbi kód talán a legegyszerűbb olyan objektum-orientált program, mely hasznos is. A neve polár-transzformációs generátor, mely kissé ijesztőnek hangzik, de csupán annyiból áll, hogy két random változót hoz létre, majd az egyiket visszaadja. A belső függvény újból meghívása során megvizsgálja, hogy van-e nem visszaadott random szám generálva, ha van, visszaadja azt, ha nincs, akkor pedig két újat generál. Ezzel valamennyire fel lehet gyorsítani a generálási folyamatot, mely sok generálás esetén elég erőforrásigényes lehet.

[kód]

Mint látjuk, a program forráskódja meglehetősen rövid, nem tart sokáig kibogozni. Ha a megírása után bemenetünk a Sun Microsystems által kiadott dokumentációba, azt láthatjuk, hogy szinte teljesen megegyezik az általunk írt program, valamint az általuk implementált randomszám-generátor. Ha át szeretnénk írni egy objektum-orientált programot egy másik nyelvre, általában csupán annyi a teendőnk, hogy átírunk egy pár kulcsszót és a két nyelv közti kisebb szintaktikai eltéréseket figyelembe véve átmásoljuk a programot egy új fájlba és odaadjuk a fordítónak. Az imént megvitatott Java forráskódot nem tart több, mint 10 percebe átírni C++ nyelvre úgy, hogy ugyanúgy működjön, mint a Java program.

[kód]

12.2. Homokozó

Ahogy az előző fejezetben megtárgyaltuk, adott objektum-orientált szemlélettel írt programokat átírni egy másik objektum-orientált nyelvbe általában nem nehéz, általában csak szintaktikailag különböznek a forráskódok. Ez a helyzet a tavaly c++-ban megírt binFa programmal is, melybe nem szükséges sok időt belefektetni, hogy átírjuk Java nyelvre. A konverzió első lépése az, hogy átmásoljuk a szükséges osztályokat egy külön fájlba, amit .java kiterjesztéssel látunk el. Az a legjobb, ha abból a verzióból indulunk ki, amelyben referenciaiként kezeljük a fa gyökerét, hiszen a Java nyelvben minden értékátadás referenciaiként működik. Ezen kívül át kell még írnunk az osztályok és függvények deklarációit is, melynek a szyntaxa kicsit más Javában, mert minden osztály és függvény előtt meg kell mondanunk, hogy az publikus legyen-e, valamint ha származtatunk egy osztályt egy másikból, akkor a ":" karakter helyett az "extends" kulcsszóval kell ezt megtenni. Hasonló apróságokat kell átírnunk, és ha minden jól csináltunk, a .java kiterjesztésű fájt már futtathatjuk is a JVM-ben.

[kód]

12.3. Gagyi

A modern programnyelvek tervezői mindenet elkövetnek azért, hogy az írt programkód átlátható, olvasható, egyértelmű és helyes legyen. Ezen kívül fontos az is, hogy gyorsan fusson, ezért a fordítókba sokféle optimalizációt implementálnak. Ezek akkor lesznek megfelelők, ha a programok ugyanúgy fordulnak le az optimalizációval, mint nélküle. Ez általában így van, de nem minden. Az interpretes nyelveket sokkal nehezebb optimalizálni, hiszen a futásidőben fordulnak a programok, így ha közben sokat analizálunk, a gépünk ugyanúgy lelassulhat, mint egy rosszul megírt kód esetén.

Egy jó példa lehet optimalizációra a Java nyelvbe beépített "pool-ozás". Ez annyit jelent, hogy ha kis számokkal számolunk a programunkban (-128-tól 127-ig), akkor az egy előre megírt adatbázisra fog referenciakat létrehozni, nem pedig teljesen új objektumokat. Ezzel lényegesen javul a futásidő, ám valóban ugyanúgy fog a prgramunk futni? A válasz: Attól függ. Általában igen, hiszen ha a számokkal egyszerű aritmetikát végzünk, amit általában számokkal szoktunk, nem változik az objektumok funkcionalitása, viszont lényegesen gyorsabban futnak le a programjaink, mert nem kell folyamatosan új objektumokat létrehozni a rendszernek. Viszont ha két szám memóriacímét vizsgáljuk, akkor beleütközhetünk egy olyan problémába, hogy két számnak ugyanaz lehet a memóriacíme. Emiatt lehetséges az, hogy a következő feltételre, ami normális körülmények között nem lehetne semmilyen módon igaz, azt kapjuk, hogy a [-128, 127] tartományon belüli számok megfelelnek.

Az ilyenfajta, nem tökéletesen egyértelmű implementációkkal pontosan ez a gond: egy nem szokványos helyzetben a programjaink félrecsúszhatnak, rossz eredményt adhatnak, esetenként lehet, hogy le sem futnak. Talán az ilyen furcsaságok miatt tart egy-egy felvetődő problémára megoldást találni, hiszen nem tudhatjuk kezdőként, hogy mi hibáztunk, vagy egyszerűen a fordító máshogyan, talán rosszul értelmezi a kódunkat. Ezért fontos, hogy elolvassuk a nyelv adott dokumentációját, és ha kérdésünk van, fordulunk valaki olyanhoz, aki teljes mértékben ismeri a nyelv adottságait.

12.4. Yoda

Feltesszük az alábbi esetet:

```
string nuku = null;
```

A Yoda írási módban a kifejezésben az összehasonlítás sorrendjét megcseréljük. Amennyiben ismerjük A-t, emberi módon általában rákérdezhetünk arra hogy A egyenlő-e B-vel.

```
if( nuku.equals("imaword")
```

Yoda kóddal viszont megkérdezük hogy B az egyenlő-e A-val.

```
if("imaword".equals(nuku)
```

Az azért fontos mert amennyiben A null-al egyenlő, NullPointerException hibát kapunk amely a programunk futását gátolja. Amennyiben Egy ismert változót/stringet hasonlítunk össze null-al, azt a Java gond nélkül összehasonlítja.

Ennek az okozója az hogy amikor null-t dekraláltuk a Java egy semmiré se mutató pointert hozott létre, nincs hozzá tartozó objectum. Amennyiben alkalmazzuk a Yoda módszert, a nyelv paraméterként érkezi a null-t és nem fordul elő a hiba.

Az elnevezés a Star Wars filmsorozatból egy híres karater nevéből származik, aki a szavak sorrendjét megcserélve beszél.

12.5. Kódolás From Scratch

A feladat BBP (Bailey-Borwein-Plouffe) a Pi hexa jegyeit meghatározó algoritmus David H. Bailey 2006-os írása alapján. A feladat ennek Java nyelvre való átírása.

(kép az algoritmus matematikai formátumáról)

Először is létrehozunk egy számítást az alábbi részre:

```
{16^d Pi} = {4*{16^d S1} - 2*{16^d S4} - {16^d S5} - {16^d S6}}
```

```
public PiBBP(int d) {  
  
    double d16Pi = 0.0d;  
  
    double d16S1t = d16Sj(d, 1);  
    double d16S4t = d16Sj(d, 4);  
    double d16S5t = d16Sj(d, 5);  
    double d16S6t = d16Sj(d, 6);
```

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

d16Pi = d16Pi - StrictMath.floor(d16Pi);

StringBuffer sb = new StringBuffer();

Character hexaJegyek[] = {'A', 'B', 'C', 'D', 'E', 'F'};

while(d16Pi != 0.0d) {

    int jegy = (int)StrictMath.floor(16.0d*d16Pi);

    if(jegy<10)
        sb.append(jegy);
    else
        sb.append(hexaJegyek[jegy-10]);

    d16Pi = (16.0d*d16Pi) - StrictMath.floor(16.0d*d16Pi);
}

d16PiHexaJegyek = sb.toString();
```

A $\{16^d S_j\}$ részletet a következő módon számíthatjuk ki:

```
public double d16Sj(int d, int j) {
    double d16Sj = 0.0d;
    for(int k=0; k<=d; ++k)
        d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);
    return d16Sj - StrictMath.floor(d16Sj);
}
```

A forrás papír megadott egy bináris hatványozási módszert a A $16^n \bmod k$ kiszámítására, ezt fel is használjuk.

```
public long n16modk(int n, int k) {
    int t = 1;
    while(t <= n)
        t *= 2;
    long r = 1;
    while(true) {
        if(n >= t) {
            r = (16*r) % k;
            n = n - t;
        }
    }
}
```

A {16^d Sj} részletet a következő módon számíthatlyuk ki:

```
public double d16Sj(int d, int j) {
    double d16Sj = 0.0d;
    for(int k=0; k<=d; ++k)
        d16Sj += (double)n16modk(d-k, 8*k + j) / (double)(8*k + j);
    return d16Sj - StrictMath.floor(d16Sj);
}
```

A forrás papír megadott egy bináris hatványozási módszert a A 16ⁿ mod k kiszámítására, ezt fel is használjuk.

```
public long n16modk(int n, int k) {
    int t = 1;
    while(t <= n)
        t *= 2;
    long r = 1;
    while(true) {
        if(n >= t) {
            r = (16*r) % k;
            n = n - t;
        }
        t = t/2;
        if(t < 1)
            break;
        r = (r*r) % k;
    }
    return r;
}
```

A main() függvényben példányosítsuk az objectumot. A megadott d = 1000000, ezért a Pi hexadecimális kifejtése a d+1 tehát 1000001. hexa jegyetől számítunk. Az eredmény: 6C65E5308. Decimálisan: 29097874184.

13. fejezet

Helló, Liskov!

13.1. Liskov helyettesítés sértése

Források: UDPORG repó, Bátfai Norbert, madarak

A Liskov féle behelyettesítési elv az objekt orientált programozás egyik fontos alapköve. Az elv kimondja, hogy a program futásának eredménye nem változhat meg attól hogy az ős osztálya helyett annak gyermeké osztályában használom.

Ilyen akkor fordulhat elő amikor nem megfelelő sorrendben kezeljük az osztályokat, például az alábbi forrásban:

```
class Madar {  
public:  
    virtual void repul() {};  
};
```

Ez a class alapján az öröklődési sorrend az alábbi: MadárFaj - Madár - repül

Bármilyen maradat hívunk meg, ez programunk szerint képes a repülésre, még akkor is ha nem kellene.

Hogyan és kene akkor? Így:

```
class RepuloMadar : public Madar {  
public:  
    virtual void repul() {};  
};
```

Az öröklődési sorrendet ágakra bontjuk az alábbi meghívási sorrendek szerint:

MadárFaj - Madár

MadárFaj - RepülőMadár - Madár

13.2. Szülő-Gyerek

Megoldás forrása:

```
class szulo
{
    private String sztring;
    public void setOsTulajdonsag(String sztring)
    {
        this.sztring = sztring;
    }
    public String getOsTulajdonsag()
    {
        return this.sztring;
    }
}
class gyerek extends szulo
{
    public String getNemSzuloTulajdonsag()
    {
        return "Nope.";
    }
}
public class Liskov
{
    public static void main(String[] args)
    {
        szulo szulominta = new gyerek();
        System.out.println("Liskov: " + szulominta.getNemSzuloTulajdonsag());
    }
}
```

! Error:(25, 51) java: cannot find symbol
symbol: method getNemSzuloTulajdonsag()
location: variable szulominta of type szulo

```
#include <iostream>
class szulo
{
private:
    unsigned int stuff;
public:
    virtual void setStuff(int m)
    {
        stuff = 1;
    }
    virtual int getStuff()
    {
        return stuff;
```

```
    }
};

class gyerek : public szulo
{
    int getGyStuff()
    {
        return 0;
    }
};

int main()
{
    szulo *szulo = new gyerek();
    std::cout << szulo->getGyStuff() << std::endl;
    return 0;

}
```

```
szulogyerek.cpp: In function ‘int main()’:
szulogyerek.cpp:26:23: error: ‘class szulo’ has no member named ‘getGyStuff’; did you mean ‘getStuff’?
    std::cout << szulo->getGyStuff() << std::endl;
                           ^~~~~~
                           getStuff
```

13.3. Ciklomatikus komplexitás

A feladat megoldásához a McCabe ciklomatikus komplexitás alánizáló programot használtam amely részletes elemzést is biztosít. Jelen feladatban csak a második oszlopban található számok érdekelnek minket. Kódforrás a korábban felhasznált PHB generátor. (III/Arroway/Kódolás From Scratch)

```
fater@fater-VirtualBox:~/repos/prog2fix/prog2fix/prog2-master/arroway$ pmccabe PolarGen.cpp
2      2       6      4      10      PolarGen.cpp(5): main
3      3       15     16      29      PolarGen.cpp(17): PolarGen::kovetkezo
```

13.4. Az SMNIST felélesztése

A tavalyi félévben már foglalkoztunk egy keveset a Bátfai Norbert által írt SMNIST for Humans v3 androidos programmal, melynek a célja az, hogy a mesterséges intelligencia határait valamilyen módon összeérje az emberi képesség határaival.

A felmérés folyamata a következő: a képernyón megjelenő pontokat kell összeszámolni és megnyomni az adott számozású gombot a mobiltelefonunk képernyőjén. A játék egyre nehezedik és gyorsul, és egy idő után azon kapja magát az ember, hogy véletlenszerűen nyomkodja a gombokat, reménykedve, hogy eltalálja a helyes számot. Itt jön be az a tény, hogy a hasonló, mintafelismeréshez kapcsolódó mesterséges intelligencia algoritmusok sokkal jobban teljesítenek az ilyen feladatokban. Míg egy ember (tapasztalatom szerint) 5-6 pontig azonnal meg tudja mondani, hogy hány pont van a képernyőn, egy erre a célra betanított algoritmusnak talán csak az idő és erőforrások szabnak határt.

Ha elkezdjük a feladatot, a legelső teendőnk az lesz, hogy beimportáljuk a fájlokat az Android Studio-ba. Ezután szükséges lesz átírnunk egy-két dolgot a programunkban és a konfigurációs fájlokban, hogy

működjön. Itt nem tudok seemi konkréthat írni, mert két platformon próbáltam megoldani a problémát és mindenkorban más volt a probléma. Egy kis interneten való kutakodás után nem okozhat problémát a dolog. Fontos lehet, hogy a szükséges fájlok elérési útvonalát helyesen adjuk meg a konfigurációs fájlokban. Ezután ha sikerült az összes problémát elhárítani, amit a fordító feldobott, elkezdhetjük a programon belül a módosításokat elvégezni. A legtöbb dologhoz elég annyit tennünk, hogy egy-egy változó értékét átírjuk, például a háttérszín megváltoztatásához a bgColor[] tömbön belül kell átírnunk az rgb értékeket.

13.5. BPP algoritmus futási ideinek összevetése

Az előző fejezetben megtárgyalt BPP algoritmus segítségével ki lehet számolni a pi matematikai konstans számjegyeit adott pontossághoz. A programunkat Java nyelven írtuk meg, ami azt jelenti, hogy egy interpreter fordította a programunkat valós időben. De vajon ez mennyire befolyásolja a futási időt?

Vessük össze a Java kódunkat egy, szintén a bpp algoritmust implementáló c++ és c# programmal! Először is gondoljuk át, hogy milyen tényezők állhatnak a futási idők mögött - ezek csak feltevések, nem a valós teszt eredményei:

Egy interpreteres nyelv (Java és C#) lassabb lesz, mint egy fordított (C++), hiszen az utóbbinál nem szükséges valós időben fordítani semmit, egyedül a már kész algoritmust kell végigjárnia a rendszernek.

Egy objektum-orientált nyelv, ahol nem lehet elhagyni az osztályokat (Java és C#) lassabb lesz, mint egy tiszta, alacsony(abb) szintű nyelv (mint a C és a C++)

Azonban lehetséges, hogy valamelyik nyelv jobban van optimalizálva, mint egy másik

Most pedig futtassuk le a tényleges állományokat!

I. 10^6 : C++: 2.03 Java: 2.46 C#: 2.34

II. 10^7 : C++: 25.29 Java: 24.84 C#: 26.78

III. 10^8 : C++: 4.35.19 Java: 3.58.98 C#: 4.37.64

10^6 : C++: 2.03 Java: 2.46 C#: 2.34

10^7 : C++: 25.29 Java: 24.84 C#: 26.78

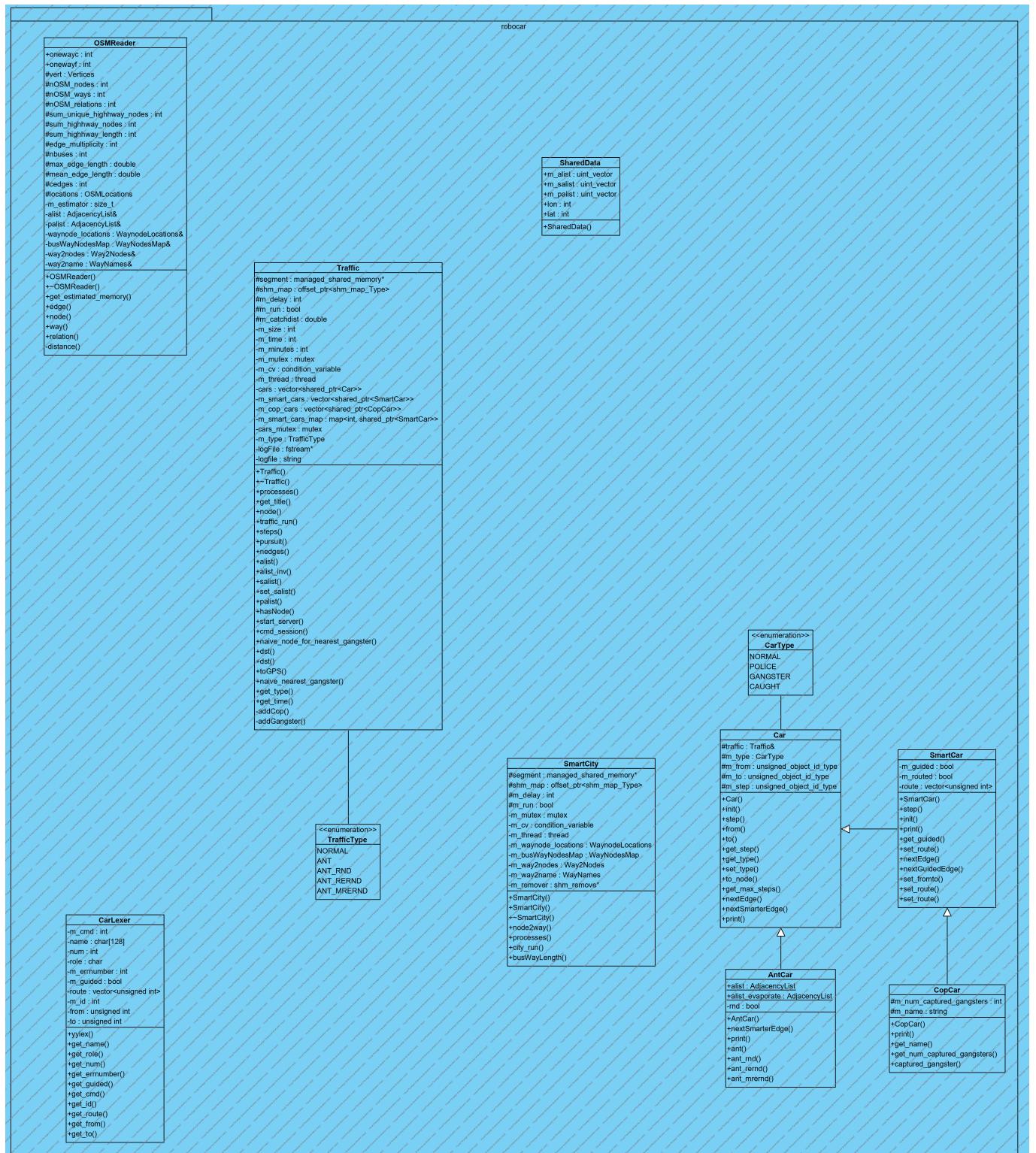
10^8 : C++: 4.35.19 Java: 3.58.98 C#: 4.37.64

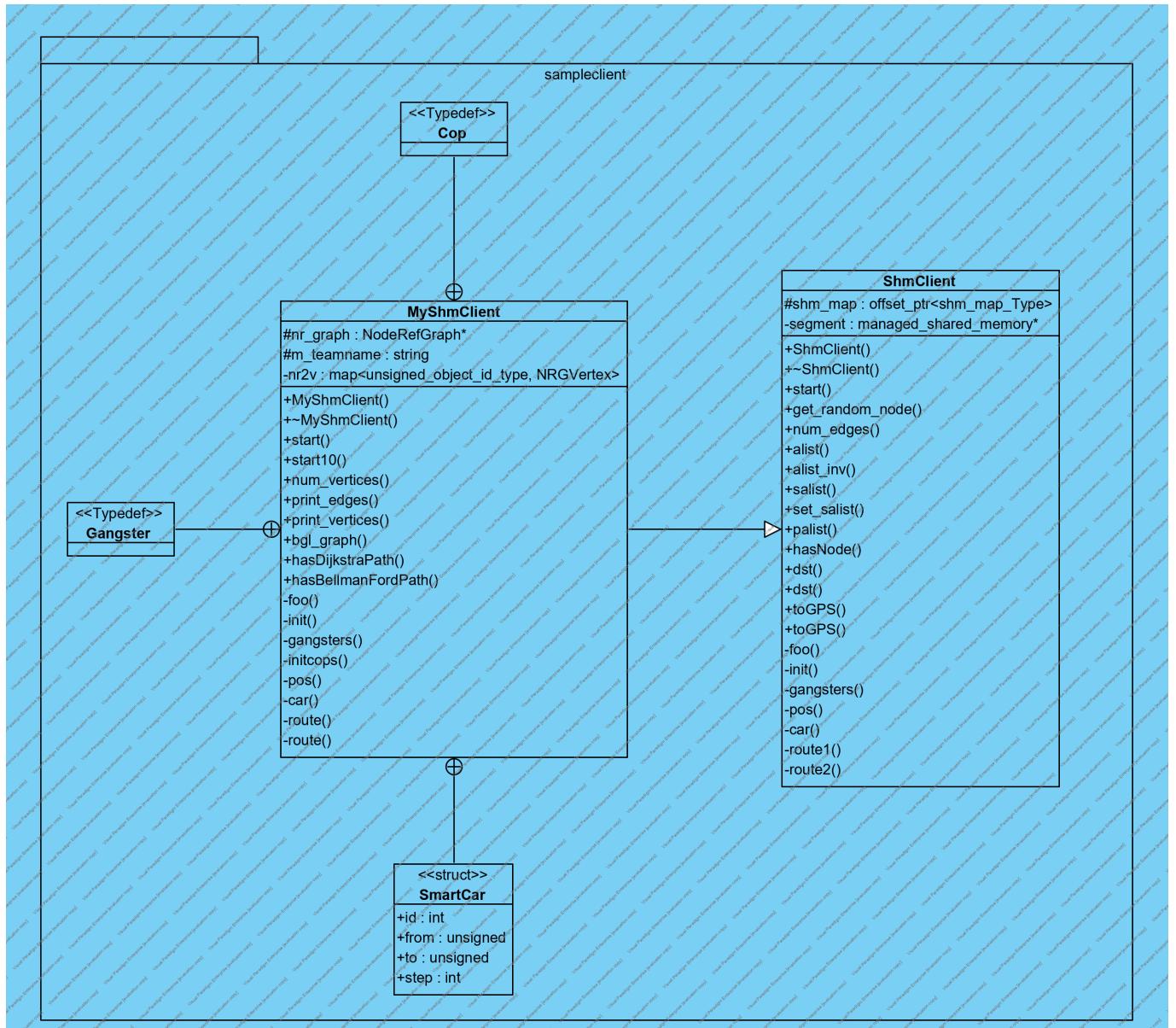
Meglepetésként érhet, hogy a Java program gyorsabban futott le, mint a másik kettő, ha több számítás történt, ez minden bizonnal a sok optimalizáció miatt van. A C# kód nagyjából olyan gyors volt, mint a C++, bár valószínűleg abban is sok az optimalizáció, hiszen egy interpreteres nyelvről beszélünk.

14. fejezet

Helló, Mandelbrot!

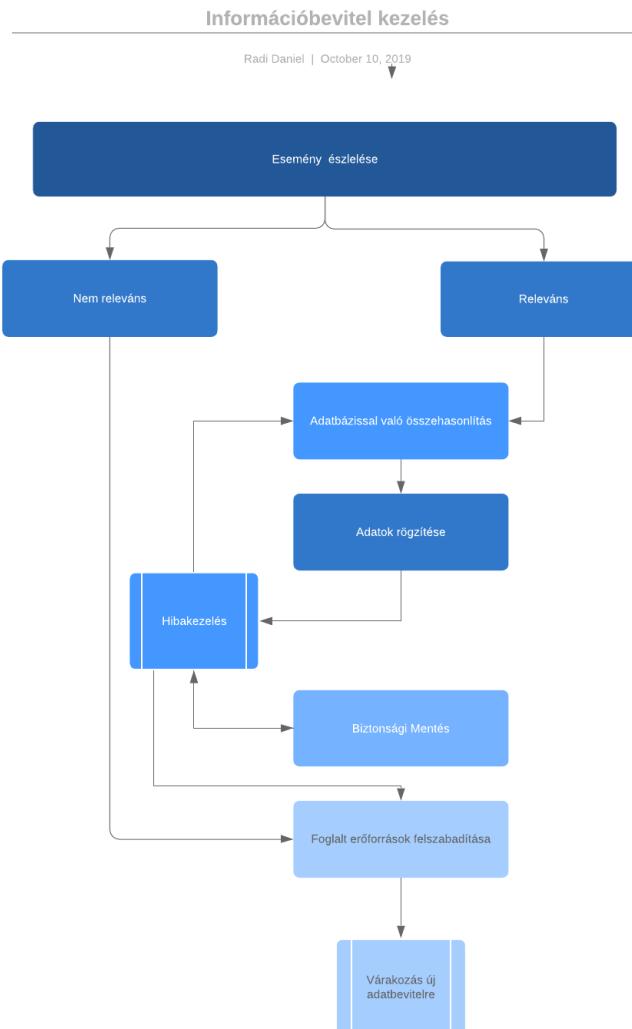
14.1. OOCWC Class Diagramms





14.2. BPMN

A BPMN a Business Process Model and Notation rövidítése. Ez egy grafikus ábrázolás hasonló az UML-hez, az üzleti folyamatok modelljének meghatározására szolgál.



A bevitelt rengeteg módon kezelhetjük programjainkban, viszont figyelnünk kell arra hogy a lehetséges problémákat elkerüljük.

Az felül található diagram ennek egy módját mutatja be.

A bevitel fogadásakor egy alapvető hivakeresési eljárást folytatunk, hogy a bevitel módtípusa és az adat típusa megfel-e az elvártnak. Maennyiben nem felel meg, nem is kezeljük, különben egy hibást program vagy egy támadás nagyon könnyen túlterhelné programunkat.

Összehasonlíjtuk a bemenetet az ismert adatokkal és felhasznájuk az adatokat szükség szerint.

Itt egy külön hibakezelő lefut a folyamat végekor amely kezeli a biztonsági mentéseket, korruptálási hibákat, kivételeket és a rosszindulatú támadásokat. Ennek hatékonysága nagyban függ a programozó képességeitől.

Amennyiben minden rendben van a folyamattal, ne felejtsük el felszabadítani a szükségtelenné vált erőforrásokat.

14.3. UML Reverse Engineering

A nagyobb cégeknél dolgozó programozók számára fontos az, hogy átlátható legyen az adott projekt felépítése és a programrészek között húzódó rés könnyen áthidalható legyen. Ehhez pontos tervezés és ugyanilyen megvalósítása szükséges a programoknak. Ezt nem elég szóban, vagy akár írásban egyeztetni, meg kell minden egyes programelem helyét és szerepét határozni a hatékony munka érdekében. minden egyes osztálynak pontosan tudni kell a függvényeit, visszatérési értékeit, public és protected változót, konstruktőrök, stb., még mielőtt elkezdődik a munka. Erre a legegyszerűbb és leghatékonyabb megoldás egy UML diagram készítése. Ebben az SQL adatbázisoknál használt kapcsolat-ábrázolásokhoz hasonlóan vannak megjelenítve program osztályai, az osztályokhoz tartozó függvények, stb.

Hogy ne csak a levegőbe beszéljünk, mutassunk egy példát!

```
 ::=LZWBinFa
-fa = null: Csomopont
-melyseg, atlagosszeg, atlagdb: int
-szorasosszeg: double
#gyoker = new Csomopont('/'): Csomopont
~maxMelyseg: int
~atlag, szoras: double
+LZWBinFa(): ctor
+egyBitFeldolg(char b): void
+kiir(): void
+kiir(java.io.PrintWriter os): void
+kiir(Csomopont elem, java.io.PrintWriter os): void
+getMelyseg(): int
+getAtlag(): double
+getSzoras(): double
+rmelyseg(Csomopont elem): void
+ratlag(Csomopont elem): void
+rszoras(Csomopont elem): void
+usage(): void
+main(String args[]): void
```

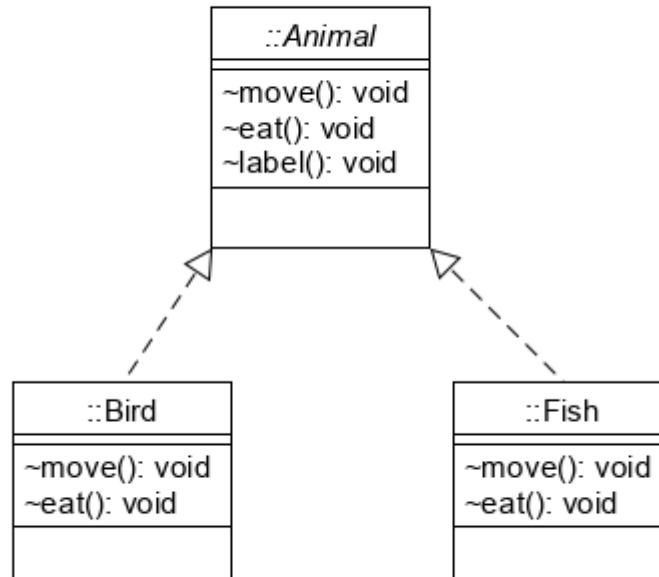
Ezen a képen a már jól ismert LZWBinFa java implementációjából generált UML diagramot láthatjuk. Ez talán nem a legjobb példa, hiszen nem tartalmaz öröklőést, sem absztrakt függvényeket, csak egy csomópont és egy binfa osztályt. A konstruktőrök, függvények visszatérési értékét és a bevett változótípusokat jól láthatjuk.

Ezek alapján sokkal egyszerűbb lehet egy adott munkafolyamatba belépni és ha valami probléma adódik, kiküszöbölni azt.

14.4. UML Osztálydiagram From Scratch

Az előző feladatban megismerkedhettünk Az UML diagramokkal és már egy létező forrásból le is generáltunk egyet. Ám az UML lényege pontosan az ellenkezője annak, amit mi csináltunk, tehát először készítünk egy diagramot, melyben megtervezzük a programunk felépítését, végiggondolva minden egyes változótípushat, függvényt (vagy metódust, ki hogy szereti hívni őket) és osztályt, valamint a köztük levő relációkat.

Az előző példa alapján lehet, hogy nem jutnánk sokra, hiszen az nem tartalmazott egyetlen öröklődést vagy absztrakt metódust sem, de ha elkezdünk egy egyszerű objektum-orientált példaprogramot végiggondolni, hamar rájöhettünk, hogy nem is olyan nehéz a dolog.



A képen egy Animal osztályt láthatunk, melyből örökli a Fish és a Bird osztály a move() és az eat() metódusokat. Ennél egyszerűbb példát nem igazán lehet hozni, de ezen is jól látszik az UML osztálydiagramok lényege és célja, valamint az, hogy mennyivel egyszerűbb lehet egy ilyen eszköz segítségével a programtervezés.

15. fejezet

Helló, Chomsky!

15.1. I334d1c4

```
import java.io.*;
import java.util.*;

public class Main
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        String OUP = "placeholder";
        while(OUP.charAt(0) != 'Q') {
            OUP = sc.nextLine();
            for(int i = 0; i < OUP.length(); i++) {
                switch (OUP.charAt(i)) {
                    case 'e':
                        OUP = OUP.substring(0, i) + '3' + OUP.substring(i + ↵
                            1); break;
                    case 'E':
                        OUP = OUP.substring(0, i) + '3' + OUP.substring(i + ↵
                            1); break;
                    case 'a':
                        OUP = OUP.substring(0, i) + '4' + OUP.substring(i + ↵
                            1); break;
                    case 'A':
                        OUP = OUP.substring(0, i) + '4' + OUP.substring(i + ↵
                            1); break;
                    case 'o':
                        OUP = OUP.substring(0, i) + '0' + OUP.substring(i + ↵
                            1); break;
                    case 'O':
                        OUP = OUP.substring(0, i) + '0' + OUP.substring(i + ↵
                            1); break;
                }
            }
        }
    }
}
```

```
        }
        System.out.println(OUP);
    }
}
```

Az alábbi l33t cypher Javában fut és sorokat fogad be, melyeket lefordít l33t nyelvre.

A sort egy stringbe menti majd azt karakterenként átnézi lecserélendő karakterekért. Amikor cserélendő karaktert talál, a stinget kettévágja, a jelenlegi karaktert egy annak megfelelő l33t karakterre cseréli, majd az előtte és autána lévő résszel egyesíti azt.

Amennyiben Q karaktert magában írjuk, a program kilép.

15.2. Perceptron osztály

A perceptron az emberi agyi neuronokhoz mintázva épül fel.

A perceptron az egyik legegyszerűbb előrecsatolt neurális hálózat. Létrehozzuk a p nevű Perceptron osztály objektumot, a konstruktörának segítségével. Ez a perceptron 3 rétegű és egy kimeneti csomópontból áll, az egyes rétegek size változó értéke, 256, 1 szálakból állnak. Majd a számításhoz szükséges mátrixba mozgatjuk az adatokat, melyeket a heapen tárolunk, majd ezeket a helyeket felszabadítjuk a delete függvény segítségével manuálisan. A double típusú változóba mentjük a perceptron számítás eredményét, amit megjelnítünk szabványos kimeneten.

```
#include <iostream>
#include "png++/png.hpp"
#include "mlp.hpp"
using namespace std;
using namespace png;
int main ( int argc, char *argv[] )
{
image<rgb_pixel> png_image ( argv[1] );
int size = png_image.get_width() * png_image.get_height();
Perceptron* p = new Perceptron(3, size, 256, 1);
double* image = new double[size];
for (int i{0}; i < png_image.get_width(); i++)
for (int j{0}; j < png_image.get_height(); j++)
image[i * png_image.get_width() + j] = png_image[i][j].red;
double value = (*p)(image);
cout << " " << value << endl;
delete p;
delete [] image;
}
```

15.3. Encoding

Sok kezdő és idegen nyelvet nem jól ismerő programozó kezdi el a programjait a saját nyelvén írni, annotálni. Ám ezt sok fordító és IDE nem támogatja. Emiatt nem csak ajánlott, de egyenesen szükséges az, hogy a programjainkat angolul írjuk, angol változó- és függvényneveket, kifejezéseket és kommenteket, a megfelelő kompatibilitás biztosításának érdekében.

De mit tehetünk abban az esetben, ha nem vagyunk hajlandók limitálni magunkat a szabványos, programozásban használt karakterhalmazhoz?

Először is: Ne csinálunk ilyet!

Másodszor: Lehetőségünk van megadni praméterként a szövegkódolást a fordítónak. Ez a következő paranccsal lehetséges UTF-8 kódolás esetén:

```
javac -encoding utf-8 <fájlnév>
```

Itt az utf-8 helyett természetesen a saját fájlunk kódolási módszerét kell beírnunk, amit a következő paranccsal tudhatunk meg:

```
file <fájlnév>
```

Ez a parancs ki fogja írni az adott fájllal kapcsolatos tudnivalókat, köztük a szöveges fájlok karakterkódolását.

Ha lefordítottuk a programunkat, nincs más dolgunk, mint futtatni, itt már nem szabad komplikációnak lennie, hiszen ez már egy bináris fájl lesz, melyben nem maradnak meg a változónevek, kommentek.

15.4. FullScreen

Egy program írásakor el kell döntenünk, hogy milyen módon fogjuk a kimenetet megjeleníteni a felhasználó számára. Ezt több szempont alapján lehetjük meg:

Ha annyi a cél, hogy egy kimeneti értéket, például egy számot kiírunk, vagy csak saját használatra írunk egy programot minél egyszerűbben, a legcélsoberűbb egy terminálban futó programot írni.

Ha el szeretnénk adni a programunkat, talán az a legjobb, ha egy grafikus felületet készíteni az egyszerű használat és vonzó kinézet miatt.

Ha egy grafikailag szofisztikált program írása a cél, mit egy képmegjelenítő, vagy egy játék, talán a legjobb ötlet egy teljes képernyős program készítése.

Az előző feladatban bemutatott java mandelbrot-implementáció kimenete egy képfájl, amit megnyitunk egy ablakban, így a legelegánsabb megoldás az, ha egy teljes képernyős ablakban jelenítjük meg azt.

Ha a MandelbrotHalmaz.java fájl 60. sorától beillesztjük a következő 3 sort, teljes képernyős kimenetet fogunk kapni.

```
this.setUndecorated(true);  
this.setVisible(false);  
GraphicsEnvironment.getLocalGraphicsEnvironment().  
    getDefaultScreenDevice().setFullScreenWindow(this);
```

16. fejezet

Helló, Stroustup!

16.1. Objektumok másolása és mozgatása c++11-ben + esszé

A c++ nyelv szerves részét képezik a mutatók. Ezek memóriacímet tartalmazó változók, melyek segítségével saját magunknak viszonylag alacsony szinten tudunk adatszerkezeteket definiálni. Ha olyan területen dolgozunk, melyen fontos a sebesség és személyre szabhatóság, ezek használata valószínűleg elkerülhetetlen. Ám ennek ára van. Pointerekkel sokszor nehéz dolgozni, átlátni, a felépítését egy adott adatszerkezetnek. Ha mutatókra állítunk mutatókat, majd arra is egyet, esetleg belerakunk a programba egy pár referenciát is, hamar olyan összetettségű problémával fogunk találkozni, amit saját magunk is nehezen látnunk át. Általában ha nem dolgozunk bonyolultabb adatszerkezetekkel, a mutatók használata nem nehéz, csupán kis gyakorlást igényel. Azonban egy fa, lista, vagy akár egy osztály implementálásakor exponenciálisan nőhet az átláthatóság nehézsége. A legtöbb magas szintű programozási nyelv manapság nem is engedi meg a mutatókkal való babrálást, helyette automatikusan végzi a memória allokálását és felszabadítását. Viszont ha nem akarunk, vagy nincs lehetőségünk egy előre elkészített API-val dolgozni, kénytelenek leszünk magunknak implementálni az adott probléma megoldásához szükséges erőforrásokat.

Ezekre vannak jól bevált módszerek, melyek alapján könnyebben tudjuk hibamentesen megoldani a problémát. Ilyenek a másoló és mozgató értékadás és konstruktur. Ezekre a fogalmakra már láttunk példát az előző fejezetben, az LZWBInFa algoritmus során. Itt az idő tehát, hogy részletesebben is megnézzük, hogy mi történik pontosan.

Másoló konstruktur:

```
LZWBInFa ( const LZWBInFa & regi )
{
    std::cout << "LZWBInFa copy ctor" << std::endl;
    gyoker = masol ( regi.gyoker, regi.fa );
}
```

Itt a konstruktur bemeneti értékként megkapja a régi fa objektumot, a masol() függvénytellemásolja annak tartalmát, majd ráállít egy mutatót, ami a gyökérmutatója lesz az új fának. Fontos megemlíteni, hogy konstans értéket kap a konstruktur, ezzel biztosítjuk, hogy az eredeti fa nem változik.

Mozgató konstruktur:

```
LZWBinFa ( LZWBinFa && regi )
{
    std::cout << "LZWBinFa move ctor" << std::endl;
    gyoker = nullptr;
    *this = std::move ( regi );
}
```

Ez a kódcsipet azt meséli el, hogy lenullázzuk az eredeti fa értékét, majd az újonnan létrehozott BinFa típusú példány gyökerének odaadjuk a régi fa szerkezetét. Ez az std::move() függvénytől történik, mely jobbértékké alakítja a bemenetként kapott értéket, ezzel megteremtve a lehetőséget, hogy egy egyenlőségjel jobb oldalára írjuk azt.

Másoló értékadás:

```
LZWBinFa & operator= ( const LZWBinFa & regi )
{
    std::cout << "LZWBinFa copy assign" << std::endl;
    Csomopont * ujgyoker = masol( regi.gyoker, regi.fa );
    szabadit (gyoker);
    gyoker = ujgyoker;
    return *this;
}
```

Itt egy "operator overloading" varázslat történik, mellyel az egyenlőségjel jelentését írjuk felül. Létrehozunk egy új Csomopont példányra mutató pointert ujgyoker néven, majd rámásoljuk a bal oldali értékként kapott fa szerkezetét. Eltöröljük a régi gyökérből induló fát, majd a régi mutatónak (itt: gyoker) odaadjuk a az új gyökér értékét. Ezután visszaadunk egy erre a fára mutató pointert, ami mutatni fog az imént lemasolt fa gyökerére, ez az érték fog odaadódnai az egyenlőség al oldalán levő változónak.

Mozgató értékadás:

```
LZWBinFa & operator= ( LZWBinFa && regi )
{
    std::cout << "LZWBinFa move assign" << std::endl;
    std::swap ( gyoker, regi.gyoker );
    std::swap ( fa, regi.fa );
    return *this;
}
```

Ezt talán a legegyszerűbb átlátni a négy konstruktur közül: Itt is az egyenlőségjelet írjuk felül, de ebben az esetben a jobb oldali értéket az std::move() függvény keretein belül kell megadni. Csupán annyi történik, hogy az új fa és a régi fa gyökerét, majd adataikat kicseréljük egymással. Ezzel azt érjük el, hogy az eredeti fára mutató pointer ki lesz nullázva, hiszen null értékű pointerrel cseréltük ki, az új pointer értéke pedig a fa gyökere lesz. Itt is visszaadunk egy erre az objektumra mutató pointert, ami jobbértékként az eredeti mutató helyén lesz.

Ezzel a négy konstrukcióval kisebb módosításokkal bármilyen adatszerkezetet biztonságosan és gyorsan fogunk tudni mozgatni és duplikálni, ha szükség lesz rá, hiszen csak pointereket pakolunk ide-oda.

16.2. RSA törés

```
import java.util.Scanner;
public class RSA {
    public static void main(String[] args) {
        KulcsPar jszereplo = new KulcsPar();
        String szoveg;
        Scanner sc = new Scanner(System.in);
        System.out.println("Adja meg a titkosítani kívánt szöveget: ");
        szoveg=sc.nextLine();
        byte[] buffer = szoveg.getBytes();
        java.math.BigInteger[] titkos = new java.math.BigInteger[buffer. ←
            length];
        for (int i = 0; i < titkos.length; ++i) {
            titkos[i] = new java.math.BigInteger(new byte[]{buffer[i]}));
            titkos[i] = titkos[i].modPow(jszereplo.e, jszereplo.m);
        }
        for (java.math.BigInteger t : titkos) {
            System.out.print(t);
            System.out.println();
        }
        for (int i = 0; i < titkos.length; ++i) {
            titkos[i] = titkos[i].modPow(jszereplo.d, jszereplo.m);
            buffer[i] = titkos[i].byteValue();
        }
        System.out.println("\n" + new String(buffer));
    }
}
class KulcsPar {
    java.math.BigInteger d,e,m;
    public KulcsPar() {
        int meretBitekben = 700 * (int) (java.lang.Math.log((double) 10)
            / java.lang.Math.log((double) 2));
        java.math.BigInteger p = new java.math.BigInteger(meretBitekben, ←
            100, new java.util.
                Random());
        java.math.BigInteger q = new java.math.BigInteger(meretBitekben, ←
            100, new java.util.
                Random());
        m = p.multiply(q);
        java.math.BigInteger z = p.subtract(java.math.BigInteger.ONE). ←
            multiply(q.subtract(java.
                math.BigInteger.ONE));
        do {
```

```
        do {
            d = new java.math.BigInteger(meretBitekben, new java.util.←
                Random());
            } while (d.equals(java.math.BigInteger.ONE));
        } while (!z.gcd(d).equals(java.math.BigInteger.ONE));
        e = d.modInverse(z);
    }
}
```

Az RSA (Rivest – Shamir – Adleman) az egyik első nyilvános kulcsú kriptoszisztemája, amelyet széles körben használnak a biztonságos adatátvitelhez. Egy ilyen kriptoszisztemában a titkosítási kulcs nyilvános, és különbözik a titkosított (privát) titkosítási kulcsotól. Az RSA-ban ez az asszimmetria a két nagy prímszám szorzatának faktorizálásának gyakorlati nehézségén alapszik, a "faktoring probléma".

Az RSA betűszó Ron Rivest, Adi Shamir és Leonard Adleman vezetékneveinek kezdőbetűiből áll, akik elsőként 1977-ben jelentették be az algoritmust. Clifford Cocks, a brit hírszerző ügynökség kormányzati kommunikációs központjában (GCHQ) dolgozó angol matematikus. 1973-ban kifejlesztett egy megfelelő rendszert, de ezt csak 1997-en nyilvánosították.

16.3. Változó argumentumszámú ctor

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

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

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

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

    double* image = new double[size];

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

    double* newPNG = (*p) (image);

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

    png_image.write("output.png");
```

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

A main-ben beolvassuk a képet és annak adataiból létrehozzuk a perceptron osztályt. A változó méretű konstruktor segítségével a méretek akkorák lesznek mint amire szügünk van. Végigmegyünk a kép szélességén és magasságán, a megfelelő szín értékeket kiírjuk az image tömbbe, majd kimentjük az új képet és felszabadítjuk a memóriát.

XInclude: bhax-textbook2-gödel.xml file not found

IV. rész

Irodalomjegyzék

16.4. Általános

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

16.5. C

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

16.6. C++

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

16.7. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf , 2004.

16.8. Számítógépes Nyelvészeti Elmélete és Gyakorlata

[1] Szekrényes, István, *Számítógépes Nyelvészeti*, <http://lingua.arts.unideb.hu/hu/pages/study/compling.php> , 2019.

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.