

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert Ács Győrfi, Dániel	2019. szeptember 18.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	8
2.4. Labdapattogás	9
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	11
2.6. Helló, Google!	12
2.7. 100 éves a Brun tétel	13
2.8. A Monty Hall probléma	15
3. Helló, Chomsky!	17
3.1. Decimálisból unárisba átváltó Turing gép	17
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	17
3.3. Hivatkozási nyelv	18
3.4. Saját lexikális elemző	19
3.5. l33t.1	20
3.6. A források olvasása	22
3.7. Logikus	23
3.8. Deklaráció	24

4. Helló, Caesar!	26
4.1. double ** háromszögmátrix	26
4.2. C EXOR titkosító	28
4.3. Java EXOR titkosító	29
4.4. C EXOR törő	30
4.5. Neurális OR, AND és EXOR kapu	33
4.6. Hiba-visszaterjesztéses perceptron	36
5. Helló, Mandelbrot!	37
5.1. A Mandelbrot halmaz	37
5.2. A Mandelbrot halmaz a <code>std::complex</code> osztállyal	41
5.3. Biomorfok	43
5.4. A Mandelbrot halmaz CUDA megvalósítása	46
5.5. Mandelbrot nagyító és utazó C++ nyelven	46
5.6. Mandelbrot nagyító és utazó Java nyelven	47
6. Helló, Welch!	48
6.1. Első osztályom	48
6.2. LZW	50
6.3. Fabejárás	50
6.4. Tag a gyökér	51
6.5. Mutató a gyökér	53
6.6. Mozgató szemantika	54
7. Helló, Conway!	56
7.1. Hangyaszimulációk	56
7.2. Java életjáték	56
7.3. Qt C++ életjáték	57
7.4. BrainB Benchmark	57
8. Helló, Schwarzenegger!	58
8.1. Szoftmax Py MNIST	58
8.2. Mély MNIST	58
8.3. Minecraft-MALMÖ	58

9. Helló, Chaitin!	59
9.1. Iteratív és rekurzív faktoriális Lisp-ben	59
9.2. Gimp Scheme Script-fu: króm effekt	59
9.3. Gimp Scheme Script-fu: név mandala	60
10. Helló, Gutenberg!	61
10.1. Programozási alapfogalmak	61
10.2. Programozás bevezetés	62
10.3. Programozás	63
III. Második felvonás	65
11. Helló, Arroway!	67
11.1. OO szemlélet	67
11.2. Homokozó	67
11.3. „Gagyí”	67
11.4. Yoda	68
11.5. Kódolás from scratch	68
12. Helló, Liskov!	69
12.1. Liskov helyettesítés sértése	69
12.2. Szülő-gyerek	69
12.3. Anti OO	69
12.4. deprecated - Hello, Android!	70
12.5. Hello, Android!	70
12.6. Hello, SMNIST for Humans!	70
12.7. Ciklomatikus komplexitás	70
13. Helló, Mandelbrot!	71
13.1. Reverse engineering UML osztálydiagram	71
13.2. Forward engineering UML osztálydiagram	71
13.3. Egy esettan	71
13.4. BPMN	72
13.5. BPEL Helló, Világ! - egy visszhang folyamat	72
13.6. TeX UML	72

14. Helló, Chomsky!	73
14.1. Encoding	73
14.2. OOCWC lexer	73
14.3. l334d1c45	73
14.4. Full screen	74
14.5. Paszigráfia Rapszódia OpenGL full screen vizualizáció	74
14.6. Paszigráfia Rapszódia OpenGL full screen vizualizáció	74
14.7. Paszigráfia Rapszódia LuaLaTeX vizualizáció	74
14.8. Perceptron osztály	74
15. Helló, Stroustrup!	75
15.1. JDK osztályok	75
15.2. Másoló-mozgató szemantika	75
15.3. Hibásan implementált RSA törése	75
15.4. Változó argumentumszámú ctor	76
15.5. Összefoglaló	76
16. Helló, Gödel!	77
16.1. Gengszterek	77
16.2. C++11 Custom Allocator	77
16.3. STL map érték szerinti rendezése	77
16.4. Alternatív Tabella rendezése	78
16.5. Prolog családja	78
16.6. GIMP Scheme hack	78
17. Helló, !	79
17.1. FUTURE tevékenység editor	79
17.2. OOCWC Boost ASIO hálózatkézelése	79
17.3. SamuCam	79
17.4. BrainB	80
17.5. OSM térképre rajzolása	80
18. Helló, Schwarzenegger!	81
18.1. Port scan	81
18.2. AOP	81
18.3. Android Játék	81
18.4. Junit teszt	82
18.5. OSCI	82

19. Helló, Calvin!	83
19.1. MNIST	83
19.2. Deep MNIST	83
19.3. CIFAR-10	83
19.4. Android telefonra a TF objektum detektálója	84
19.5. SMNIST for Machines	84
19.6. Minecraft MALMO-s példa	84
20. Helló, Arroway!	85
20.1. C++ és Java összehasonlítás	85
20.2. Python	86
IV. Irodalomjegyzék	88
20.3. Általános	89
20.4. C	89
20.5. C++	89
20.6. Lisp	89

Ábrák jegyzéke

4.1. A neurális háló egységei	33
5.1. A Mandelbrot halmaz a komplex síkon	40

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 allokálni igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Mindenesetre 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 gyerekeknek, 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 gyerekeknek, 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 (hogyan lássuk mást is) példával.

Hogyan nyomjuk?

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

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

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

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

C végtelen ciklus, amely 100 százalékban dolgoztat egy magot:

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

Egy egyszerű `int main()` függvénybe beillesztünk egy `for` ciklust aminek argumentumaihoz nem adunk meg paramétereket. A paraméterek hiánya miatt a ciklus nem függ semmitől, ezért végtelen ciklust kapunk. /Különösebb `#include`-ok nem szükségesek/

C végtelen ciklus, amely 0 százalékban dolgoztat egy magot:

```
#include <unistd.h>

int main()
{
    int x=0;
    while (x<1)
    {
        sleep(5);
    }
}
```

A `sleep` beépített funkciót hívjuk segítségül. A "sleep" funkció adott másodpercig késlelteti a program végrehajtását. A `main()` függvényben található egy deklarálás, amely `x`-nek 0-át ad értékül. Található utána egy `while` ciklus, aminek feltétele a már deklarált `x`-hez kötődik. A feltétel alapján mindig igaz lesz aargumentum, így egy végtelen ciklust kapunk. A benne lévő "sleep" funkció így folyamatosan "altatja" a programot, ezáltal nem terheli a magot.

/Szükséges `#include` a `sleep` funkcióhoz: `unistd.h`/

Itt egy egyszerű kódcsipet ami demonstrálja a `sleep` funkció működését:

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

int main()
{
    printf("Várjon 5 másodpercet a kilépéshez.\n");
    sleep(5);
    return 0;
}
```

C végtelen ciklus, amely 100 százaléban dolgoztatja az összes magot:

```
#include <omp.h>

int main(void)
{
    int x = 0;
    #pragma omp parallel
    while (x<1) {}
}
```

A "#pragma omp parallel" segítségével futtatjuk párhuzamosan a magokon a ciklust. A függvény felépítése hasonló az előző feladatrészelethez található kóddal.

/Szükséges #include : omp.h/

A fordításhoz szükséges az -fopenmp kapcsoló! Lásd: gcc -fopenmp vegtelen_ciklus.c -o main

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

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

Megoldás videó:

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

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

    main(Input Q)
```

```
{
  Lefagy(Q)
}
```

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

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

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

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

Mit for kiírni erre a T1000 (T1000) futtatásra?

- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true

- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

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

Tanulságok, tapasztalatok, magyarázat...

2.3. Változók értékének felcserélése

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

Két változó felcserélése segéd változóval:

```
#include <stdio.h>

int main()
{
    int a = 3;
    int b = 5;
    printf("A változók felcserélés előtt: a=%d és b=%d\n", a, b);
    int seged = 0;
    seged = a;
    a = b;
    b = seged;
    printf("A változók felcserélés után: a=%d és b=%d\n", a, b);
}
```

Létrehozunk egy `int main()` függvényt. Deklarálunk két változót és értéket adunk nekik. Azért, hogy lássuk az alap felállást először kiiratjuk kimenetre. Deklarálunk egy "seged" változót, ami segéd változóként fog funkcionálni (bumm meglepetés). A segéd változónak átadjuk értékül az 'a' értékét. Majd az 'a' változónak átadjuk a 'b' értékét, így az egyik változó már megkapta a másik változó értékét anélkül, hogy bármelyiknek az értéke is elveszett volna. Mivel a segéd változó hordozza az 'a' értékét ezért azt átadjuk a 'b' változónak és kész is vagyunk. Biztonság képpen, kiiratjuk, hogy lássuk, tényleg véghez ment sikeresen a két változó cseréje.

Két változó felcserélése kivonás és összeadás segítségével:

```
#include <stdio.h>

int main()
{
    int a=5;
    int b=1;
    printf("A kezdő érték: a=%d és b=%d\n", a, b);
    a=a+b;
    b=a-b;
    a=a-b;
    printf("A csere utáni érték: a=%d és b=%d\n", a, b);
}
```

Létrehozunk egy `int main()` függvényt. Deklarálunk két változót és értéket adunk nekik. Azért, hogy lássuk az alap felállást először kiiratjuk kimenetre. Az 'a' változónak értékül adjuk a két változó jelenlegi értékeinek az összegét: $a=5+1$ azaz 6. A 'b' változónak értékül adjuk a két változó jelenlegi értékeinek a különbségét: $b=6-1$ azaz 5. A 'b' változó már meg is kapta az 'a' változó eredeti értékét. Az 'a' változónak értékül adjuk a két változó jelenlegi értékeinek a különbségét: $a=6-5$ azaz 1. Így tehát kész is vagyunk és még külön segéd változót sem kellett használnunk ezzel is kevesebb tárhelyet foglal el a programunk!

Két változó felcserélése szorzás és osztás segítségével:

```
#include <stdio.h>

int main()
{
    int a=4;
    int b=3;
    printf("A kezdő érték: a=%d és b=%d\n", a, b);
    a=a*b;
    b=a/b;
    a=a/b;
    printf("Csere utáni érték: a=%d és b=%d\n", a, b);
}
```

Létrehozunk egy `int main()` függvényt. Deklarálunk két változót és értéket adunk nekik. Azért, hogy lássuk az alap felállást először kiiratjuk kimenetre. Az 'a' változónak értékül adjuk a két változó jelenlegi értékeinek a szorzatát: $a=4*3$ azaz 12. A 'b' változónak értékül adjuk a két változó jelenlegi értékeinek a hányadosát: $b=12/3$ azaz 4. A 'b' változó már meg is kapta az 'a' változó eredeti értékét. Az 'a' változónak értékül adjuk a két változó jelenlegi értékeinek a hányadosát: $a=12/4$ azaz 3. Így tehát kész is vagyunk!

Két változó felcserélése kizáró vagy(exor:'^') segítségével:

```
#include <stdio.h>

int main()
{
    int a=2;
    int b=3;
    printf("A kezdő érték: a=%d és b=%d\n", a, b);
    a = a ^ b;
    b = b ^ a;
    a = a ^ b;
    printf("Csere utáni érték: a=%d és b=%d\n", a, b);
}
```

Létrehozunk egy `int main()` függvényt. Deklarálunk két változót és értéket adunk nekik. Azért, hogy lássuk az alap felállást először kiiratjuk kimenetre.

2.4. Labdapattogás

Labdapattogtatás if-ekkel:

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

int
main ( void )
{
    WINDOW *ablak;
    ablak = initscr ();

    int x = 0; //oszlop
    int y = 0; //sor

    int deltax = 1; //lépésszám az x tengelyen
    int deltay = 1; //lépésszám az y tengelyen

    int mx; //oszlopok száma
    int my; //sorok száma

    for ( ;; ) { //végtelen ciklus segítségével futtatjuk a porgramot
        //hogya sose álljon meg (max ha lelőjük)

        getmaxyx ( ablak, my , mx );

        mvprintw ( y, x, "o" );

        refresh ();
        usleep ( 100000 );

        //clear(); //ha azt szeretnénk, hogy mindig csak 1
        //labda pattogjon ne hagyjon maga után nyomot
        //, akkor használni kell a clear()-t

        x = x + deltax; //léptetjük az x tengelyen a labdát
        y = y + deltay; //léptetjük az y tengelyen a labdát

        if ( x>=mx-1 ) { // elérte-e a jobb oldalt?
            deltax = deltax * -1;
            //ha elérte akkor megfordítjuk az irányát,
            //vagyis mínusz előjelet kap
        }
        if ( x<=0 ) { // elérte-e a bal oldalt?
            deltax = deltax * -1; //ugyanaz
        }
        if ( y<=0 ) { // elérte-e a tetejét?
            deltay = deltay * -1; //ugyanaz
        }
        if ( y>=my-1 ) { // elérte-e a alját?
            deltay = deltay * -1; //ugyanaz
        }
    }
}
```

```

    }
}
return 0;
}

```

Fordítás: gcc labdapat.c -o main -Incurses

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

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

A szóhossz méretének meghatározása:

```

#include <stdio.h>

int main()
{
    int i=1;
    int darab=0;
    while(i!=0)
    {
        i<<=1;
        ++darab;
    }
    printf("A gépünkön a gépi szó ilyen hosszú: %d\n",darab);
}

```

Fordítjuk, futtatjuk és a gépünk ki is adja, hogy 32.
Na de hogyan is működik ez:

Az 'i' integer-t 32 biten tárolja. 2-es számrendszeren van ábrázolva:

```
00000000 00000000 00000000 00000001
```

Ekkor az értéke 1, mivel 2^0 -on az 1. És arra vagyunk kíváncsi, hogy az 1-est hányszor kell balra shiftelni, hogy a végén csak nullák maradjanak. Hogy is néz ki ez a shiftelés: az összes bitet 1-el arrébb pakoljuk és az utolsó helyére egy 0 kerül.

1. shift: 00000000 00000000 00000000 00000010 Ekkor az értéke 2, mivel 2^1 -őn az 2
2. shift: 00000000 00000000 00000000 00000100 Ekkor az értéke 4, mivel 2^2 -on az 4
3. shift: 00000000 00000000 00000000 00001000 Ekkor az értéke 8, mivel 2^3 -on az 8

.
.
.

31. shift: 10000000 00000000 00000000 00000000 Ekkor az értéke 4.294967.296 , mivel 2^{31} -en az annyi

32. shift: 00000000 00000000 00000000 00000000 Ekkor az értéke 0, mivel csak nullák vannak.

Ha megfigyeljük, akkor az értéke, a shiftelések során mindig a kétszeresére nő. A while ciklusnak az a feltétele az, hogy addig menjen, ameddig az 'i' nem nulla. Vagyis addig megyünk ameddig 0 lesz, így a while hamis lesz, ezért megáll, nem növeli az 'i' értékét tovább.

2.6. Helló, Google!

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

A PageRank az informatikában egy olyan algoritmus, amely hiperlinkekkel összekötött dokumentumokhoz számokat rendel azoknak a hiperlink-hálózatban betöltött szerepe alapján. (Ezt a számot szintén PageRanknek nevezik.) A PageRank a Google internetes keresőmotor legfontosabb eleme. Larry Page és Sergey Brin (a Google alapítói) fejlesztették ki 1998-ban a Stanford Egyetemen. A Google arra a feltételezésre épít, hogy a weboldalak készítői általában azokra az oldalakra linkelnek a saját lapjukról, amiket jónak tartanak, vagyis minden hiperlink felfogható egy-egy szavazatként a céloldalra. Minél több szavazatot kap egy oldal, annál fontosabb, de azt is figyelembe kell venni, hogy a szavazatot leadó oldal mennyire fontos. (Ez egy rekurzív definíció: az a fontos oldal, amire fontos oldalak mutatnak.) A PageRank a fontosság számszerűsítése

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

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

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

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



```
double PR[4] = {0.0, 0.0, 0.0, 0.0};
double PRv[4] = {1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0};

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

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

    if ( tavolsag(PR,PRv, 4) < 0.0000000001)
        break;
}
kiir (PR,4);
return 0;

}
```

Megoldás forrása: <https://hu.wikipedia.org/wiki/PageRank>

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

A Brun tétel a ikerprímek reciprokaiknak sorozatával foglalkozik. Az iker prím két egymás követő prím, amelyeknek különbsége kettő. Ezeknek a számoknak vesszük a reciprokát, majd ha ezeknek vesszük a végtelen sorozatát akkor a Brun-tétel szerint konvergálni fog egy számhoz.

A következő programhoz szükséges még telepítjük a matlab csomagot. Lásd: `install.packages("matlab")`. Ezután meghívjuk : `library(matlab)`. Aztán dolgozhatunk is... /A kódot ha egy az egyben így átadjuk a gépnek, akkor utána az 'stp' funkcióként működni fog/

```
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
}
```

```

rtlplust2 = 1/t1primes+1/t2primes
return(sum(rtlplust2))
}

```

Na fussuk át a kódot:

1. A primes(x) visszaadja x-ig a prímszámokat

```

> primes(50)
[1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47

```

Ezt a vektort átadjuk a primes-nek, így tudunk rá hivatkozni.

2. Ezzel számoljuk ki a két egymást követő prím különbségét:

```

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

```

primes[2:length(primes)] : 2. tagtól a vektor hosszúságáig írja ki a vektorokat, vagyis az 1. elem kivételével az összeset.

```

> primes[2:length(primes)]
[1]  3  5  7 11 13 17 19 23 29 31 37 41 43 47

```

primes[1:length(primes)-1]: az összes tagot kiíratja az utolsó tag kivételével.

```

> primes[1:length(primes)-1]
[1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43

```

Ha így kivonjuk a két vektort akkor az n-edik tagból kivonjuk n-1-edik tagot és megkapjuk az egymást követő prímszámok különbségét. Azt az esetet keressük, ahol az érték 2.

```

idx = which(diff==2)

```

Ez lesz az 'idx' változó. Ezek helyén lévő prímekre van szükségünk.

```

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

```

A plusz kettő azért kell, mert ugye a prím plusz az iker párja kell, így egyszerűen csak hozzáadunk kettőt.

```

> t1primes
[1]  3  5 11 17 29 41
> t2primes
[1]  5  7 13 19 31 43

```

Utána nézzük a reciprokjainak összegét, amelyet letárolunk az 'rtlplust2' változóba.

```

> rtlplust2 = 1/t1primes+1/t2primes
> rtlplust2
[1] 0.533333333 0.34285714 0.16783217 0.11145511 0.06674082 0.04764606

```

És végül ezeket visszatérő értéknek összegezzük:

```
return(sum(rtlplust2))
```

Miután az egész programot beadtuk funkcióként a gépnek, ha következőt lefuttatva megláthatjuk

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

Ezt a parancssort lefuttatva egy ablak ugrik fel, amin egy grafikon lesz ábrázolva. Ez a grafikon az x tengelyen az összeadott elemek számát, az y tengelyen pedig az összeget ábrázolja. Láthatjuk, hogy a grafikon eléggé erősen konvergál.

Végtelen sok prím szám van? Nem tudhatjuk... Az összegük tényleg egy számhoz fognak tartani? Sejthetjük... ez mind mind nem biztos de egy majdnemnek megteszi.

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

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

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

A Monty Hall probléma egy régebben játszott "Áll az alku?" TV műsorból származott. A játékban volt 3 ajtó, ezen belül 1 ajtó mögött egy vadonatúj sportkocsi, 2 mögött meg nem volt nyeremény. A műsorvezetőt Monty Hallnak nevezték, innen jött a neve a problémának. A játékosnak választania kellett egy ajtót a 3 közül. Az ajtót még nem nyitották ki. Ezután a műsorvezető is választott egy ajtót, amelyet kinyitnak, és amögött az ajtó mögött nem volt a nyeremény. Ekkor a játékos választhatott, hogy marad-e az eredetileg választott ajtónál, vagy megváltoztatja a választását a még nem választott ajtóra.

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

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

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

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

```
holvált = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i]))
valtoztat[i] = holvált[sample(1:length(holvált),1)]}

valtoztatesnyer = which(kiserlet==valtoztat)

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

A program 10.000.000 eseten keresztül vizsgálja, hogy hány esetben nyer a játékos ha nem vált és hány esetben ha igen. Miután lefuttatjuk, megkapjuk, hogy:

```
> length(nemvaltoztatesnyer)
[1] 3334756
> length(valtoztatesnyer)
[1] 6665244
```

Mint látható, majdnem kétszer több esetben jön be az, ha változtatunk a választásunkon. De vajon miért van ez így? Az ajtókon 1/3 valószínűség van arra, hogy a nyeremény mögötte van. Azzal, hogy kiválasztunk egy ajtót, 1/3 esélyünk van arra, hogy a nyeremény ott van, és 2/3 esély arra, hogy a többi mögött van. Miután a műsorvezető kinyitja a másik két ajtó közül az egyiket - és az nem nyerő - ezért a 2/3 esély "összeszűkül" egy ajtóra. Ezért éri meg jobban, ha változtatunk és ez statisztikailag is kimutatható. Nem biztos, hogy a másik ajtó mögött lesz a nyeremény, de nagyobb az esély arra, hogy ott lesz mint hogy nem.

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

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

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfiával megadva írd meg ezt a gépet!

```
#include "std_lib_facilities.h"

int main()
{
    int dec = 0;
    cout << "Szám decimálisban: ";
    cin >> dec;
    cout << "Szám unárisban: ";
    for ( dec ; dec>0 ; --dec)
    {
        cout << "1";
    }
}
```

Decimális számrendszert használunk a mindennapokban. Az unáris számrendszer az 1-es számrendszer. Minden számrendszernek annyi eleme van amennyi maga a számrendszer. Például: kettesszámrendszer: 0, 1; hármasszámrendszer: 0, 1, 2. Az egyesszámrendszernek egyetlen egy eleme van ami az 1. Decimálisból úgy váltunk unárisba, hogy annyi 1-est írunk le amennyi az értéke a számnak.

A kód egy egyszerű átváltást hoz létre. A main függvényben bekérjük a decimális számot amelyet elmentünk egy dec nevű integer változóban. Majd egy for ciklusban a szám maga a feltételadó. Mindig mikor leírunk egy egyest csökkentjük a számot, és ezt addig csináljuk amíg maga a szám el nem éri a 0-t

Tutoriál: Mátravölgyi Adrián

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

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

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

- Első verzió:

S -ből indulunk ki

$S (S \rightarrow aXbc)$

$aXbc (Xb \rightarrow bX)$

$abXc (Xc \rightarrow Ybcc)$

$abYbcc (bY \rightarrow Yb)$

$aYbbcc (aY \rightarrow aa)$

$aabbcc$

- Második verzió:

$S (S \rightarrow aXbc)$

$aXbc (Xb \rightarrow bX)$

$abXc (Xc \rightarrow Ybcc)$

$abYbcc (bY \rightarrow Yb)$

$aYbbcc (aY \rightarrow aaX)$

$aaXbbcc (Xb \rightarrow bX)$

$aabXbcc (Xb \rightarrow bX)$

$aabbXcc (Xc \rightarrow Ybcc)$

$aabbYbcc (bY \rightarrow Yb)$

$aabYbbccc (bY \rightarrow Yb)$

$aaYbbbbccc (aY \rightarrow aa)$

$aaabbbccc$

3.3. Hivatkozási nyelv

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

```
#include <stdio.h>

int main()
{
    for(int i=1; i<10; i++){
        printf("Lefutott ");
    }
}
```

Fordítás: gcc c89_c99.c -o main -std=c99 vagy gcc c89_c99.c -o main -std=c89 függően attól, hogy melyik szabványt szeretnénk megnézni.

Az alábbi kódcsipetben egy for ciklusban egyszerűen megmutatható, hogy van olyan kódfajta amely adott szabvánnyal nem, de másik szabvánnyal lefordul. A lényege a kódban az i változó deklarálása. A c99 szabványban megengedett, hogy a for ciklus argumentumaiban deklaráljuk a változót, ami paraméterként szolgál, azonban a c89 szabványban nem megengedett, csak is ha a for ciklus előtt deklaráljuk.

3.4. Saját lexikális elemző

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

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

Lexer használatával a fent látható kód megíródik C-ben. Ez a program egy ömlesztett szövegből kiszűri a valós számokat. A kód 3 részből áll.

Az első részben (ami a két kapcsos zárójel közt található) a C forráskódba kerülő részek vannak. Szükséges az #include mivel printf-elni szeretnénk. Mivel szeretnénk számolni, hogy hány számot olvas be a program, ezért egy számlálót deklarálunk. Az első résznek a végén található a definíciók. A digit azt határozza meg, az intervallumban meghatározott számjegyek lesznek a digitek.

A második részben következnek a fordítási szabályok. A csillag jelöli, hogy az előtte álló "digit"- amelyet már definiáltunk - tetszőleges számban szerepelhet. Mivel szükségünk van magára a '.' karakterre ezért elé kell helyeznünk egy védő karaktert, ami azt funkcionálja, hogy a joker karaktert nem tekinti joker karakternek. Világos. A következő digit után szereplő '+' jel pedig azt jelenti, hogy akármennyi digit szereplhet de legalább egy. Azért van a második tag bezárójelezve, mivel lehet, hogy van rá szükség és lehet, hogy nincs. Ezután található egy tabulátorral elválasztott C forráskódjel, amely azt jelenti, hogy a valósszám számlálót növeljük eggyel. Azután printf segítségével kiiratjuk stringként (%s) és számként (%f) is egyaránt. Az atof függvény, konvertálja stringet double lebegőpontos számmá.

A kód harmadik része maga a program. Az yylex-el hívjuk a lexikális elemzőt, majd ha lefutott kiiratjuk a valósszámok számát.

Szükséges ehhez a programhoz telepítenünk a flexhez tartozó csomagokat: `sudo apt install flex`

Lexelés: `lex -o realnumber.c realnumber.l`

Fordítás: `gcc realnumber.c -o realnumber -lfl`

Például:

```
abc3.57de43fghi24.68jkl
abc[realnum=3.57 3.570000]de[realnum=43 43.000000]fghi[realnum ←
    =24.68 24.680000]jkl
```

3.5. l33t.l

Lexelj össze egy l33t ciphert!

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

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

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

{'a', {"4", "4", "@", "/-\\\"}},
{'b', {"b", "8", "|3", "|"}},
{'c', {"c", "(", "<", "{"}},
{'d', {"d", "|)", "|]", "|"}},
{'e', {"3", "3", "3", "3"}},
{'f', {"f", "|=", "ph", "|#"}},
{'g', {"g", "6", "[", "[+"}},
{'h', {"h", "4", "|-|", "[-"}},
{'i', {"1", "1", "|", "!"}},
```



```
{'j', {"j", "7", "_|", "_/"}}},
{'k', {"k", "|<", "1<", "|{"}}},
{'l', {"l", "1", "|", "|_"}},
{'m', {"m", "44", "(V)", "\\|"}},
{'n', {"n", "\\|", "/\\/", "/V"}},
{'o', {"o", "0", "()", "[]"}},
{'p', {"p", "/o", "|D", "|o"}},
{'q', {"q", "9", "O_", "(,)"}}},
{'r', {"r", "12", "12", "|2"}},
{'s', {"s", "5", "$", "$"}},
{'t', {"t", "7", "7", "'|"}},
{'u', {"u", "|_|", "(_)", "[_]"}},
{'v', {"v", "\\|/", "\\|/", "\\|/"}}},
{'w', {"w", "VV", "\\|\\|/", "(/\\|)"}}},
{'x', {"x", "%", ")(", ")("}},
{'y', {"y", "", "", ""}}},
{'z', {"z", "2", "7_", ">_"}}},
```

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

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

```
%}
```

```
%%
```

```
. {
```

```
int found = 0;
for(int i=0; i<L337SIZE; ++i)
{
    if(l337d1c7[i].c == tolower(*yytext))
    {
        int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0));

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

```
        printf("%s", l337d1c7[i].leet[2]);
    else
        printf("%s", l337d1c7[i].leet[3]);

    found = 1;
    break;
}

}

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

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

Lexelés: `lex -o l337d1c7.c l337d1c7.l`

Fordítás: `gcc l337d1c7.c -o l337d1c7 -lfl`

Egy olyan struktúrát képez a kód, amely betűket különböző, hasonló karakterekre cserél át. A kód megint három részből áll.

A kód első részében található egy struktúra amelyben deklarálunk egy `char c` betűt. Ebből készítünk egy `l337d1c7 []` tömböt. Nem adunk meg neki elemszámot hanem feltöltjük az utána következő elemekkel. Ezek voltak a definíciók.

A kód második részében található egy for ciklus, amelyben a nagybetűket átváltja kisbetűkre. Aztán megnézi, hogy egyenlő-e a tömb `i`-edik elemének a `.c` részével. Az `i`-vel végigmegyünk a tömböt feltöltő sorokon, és keresem a `.c` részét vagyis magát a betűt. Ha feltétel teljesült akkor megtörténik egy random szám generálás, amely 0-tól 100-ig terjedhet. Ha 91-nél kisebb akkor az első elemet fogja választani, ha 95-nél kisebb akkor a második elemet fogja választani, és így tovább. Ha megtalálta akkor `found = 1`, ha nem akkor pedig visszaírja az eredeti betűt

A kód harmadik részében következik a forráskód. Inicializál egy random számot. Aztán indítja a parcolást, a lexikális elemzést és ugye akkor indul el a szabály rész.

3.6. A források olvasása

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

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

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



Bugok

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

i.

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

ii. Egy for ciklus amely i=0-tól, ameddig i kisebb mint 5, i-t növeli eggyel.

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

iii. Hibás az i növelése.

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

iv. Hibás az i növelése. Egyébként for ciklus, amely i=0-tól indul és addig megy míg i el nem éri az 5-öt, a tömb i-edik eleme egyenlő lesz i+1-el.

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

v.

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

vi. Az f függvény két operandosú. Először kiírja az 'a' és az 'a'-ra rákövetkező elem együttesével meghatározott függvényértéket, majd fordítva.

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

vii. Kiírja az f függvény 'a'-n kapott értékét és magát az 'a' változót.

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

viii.

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

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

Minden x-hez van olyan y, ahol x kisebb mint y és y prím

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

Minden x -hez van olyan y , ahol x kisebb mint y és y prím és y -ra rákövetkező 2. szám prím.

```
$(\texttt{forall } x \texttt{ exists } y ((x < y) \wedge (y \texttt{ text\{ prím\}}) \wedge (Ssy \texttt{ text\{ prím\}})) \leftrightarrow )$
```

Létezik y , ahol minden x prím vagy x kisebb mint y

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

Létezik y , ahol minden x nagyobb mint y , vagy x nem prím.

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

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

/forall x : minden x

/exist x : létezik olyan x

/wedge: és

/supset: vagy

/S x : x -re rákövetkező szám, innen jön, hogy ahány 'S'-t írunk a változó elé, a ahanyadik rákövetkező tagra vagyunk kíváncsiak.

/neg: nem, azaz tagadás

/text: szöveg formátum

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész

```
int a;
```

- egészre mutató mutató

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

- egész referenciája

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

- egészek tömbje

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

- egészek tömbjének referenciája (nem az első elemé)

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

- egészre mutató mutatók tömbje

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

- egészre mutató mutatót visszaadó függvény

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

- egészre mutató mutatót visszaadó függvényre mutató mutató

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

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

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

- függénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

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

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

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

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

Megoldás forrása: [bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c](https://github.com/bhax/thematic_tutorials/blob/master/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c)

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

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

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

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

    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;
}

```

Megadjuk a mátrix sorainak a számát 'int nr = 5;'. Aztán egy Pointerre mutató pointer 'double **tm' amely lefoglal 8 bájtot. Majd kiírjuk a 'tm' címét a következő kód segítségével:

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

. A 'malloc' segítségével lefoglalunk 5-ször 8 bájtot, ami egy void mutatót ad vissza. Ha ez mégsem sikerül a 'NULL' és 'return -1' -el kilép a program. Újra kiírjuk tm címét. A for ciklusban 'nr'-nek megfelelő alkalommal, avagy jelen esetben 5-ször, a 'malloc' segítségével 'tm[i]'-nek lefoglalja a helyet. Az elsőnek 8-bájtot, a másodiknak 16-bájtot és így tovább egészen az 5-ödik elemig. Ezután ismét kiírjuk 'tm' címét. Létrehozunk for ciklussal egy mátrixot és kiírjuk az értékeit. A 'free'-vel felszabadítjuk a pointereket és bezárjuk a programot.

A program egy alsó háromszögmátrixot hoz létre, aminek a lényege, hogy a főátló alatt találhatóak csak elemek. Miután fordítjuk és futtatjuk ezt kapjuk:

```

0x7ffc55f9fb00
0x557a4625b670
0x557a4625b6a0
0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
6.000000, 7.000000, 8.000000, 9.000000,

```

```
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,  
0.000000,  
1.000000, 2.000000,  
3.000000, 4.000000, 5.000000,  
42.000000, 43.000000, 44.000000, 45.000000,  
10.000000, 11.000000, 12.000000, 13.000000, 14.000000,
```

4.2. C EXOR titkosító

```
#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);  
  
    }  
}
```

Fordítás: gcc exor_tor.c -o main

Futtatás:

```
./main 56789012 < tiszta.txt > titkos.txt
```

Befile:

A processz és a szál olyan absztrakciók, amelyeket egy program teremt meg számunkra, az operációs rendszer, azaz a kernel. A konkrétabb tárgyalás kedvéért gondoljunk most egy saját C programunkra! Ha papíron, vagy a monitoron egy szerkesztőben nézegetjük a forrását, akkor valami élettelen dolgot vizsgálunk, amelyben látunk lexikális és szintaktikai egységeket, utasításokat, blokkokat, függvényeket; nem túl érdekes. Ha lefordítjuk és futtatjuk, akkor viszont már valami élő dolgot vizsgálhatunk, ez a processz, amely valahol ott van a tárban. Ennek a tárterületnek az elején a program környezete, a parancssor argumentumai, a lokális változóterülete és a paraméterátadás bonyolítására szolgáló veremterület található, amelyet a dinamikusan foglalható területe, a halom követ. Majd jön az inicializált globális és statikus változókat hordozó adat szegmens és az inicializált BSS. Végül jön a kódszegmens, majd a konstansai. Ennek a tárterületnek a kernelben is van egy vetülete, ez a PCB.hogya

Kifile:

A titkosított szöveg láthatóan emberi fogyasztásra alkalmatlan. A dekódoláshoz nem kell mást tennie az Olvasónak, mint újra futtatni a programot ugyanazzal a kulccsal, de most bemenetként az iménti futtatáskor elkészített titkosított szöveget beleirányítva:

```
/main 56789012 < titkos.txt > tiszta.txt
```

Az ASCII jelkészlet szövegkaraktereket előjel nélküli egész számokra képez le. Minden jelhez tartozik egy bináris 8 biten tárolt érték. Ezt az értéket ha exorral törjük akkor egy másik értéket kapunk és ez már egy másik karakterhez fog tartozni. Ezt minden karakterrel megteesszük és így titkosítva is van a szövegünk.

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux/ch05s02.html

Tanulságok, tapasztalatok, magyarázat...

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

A kizáró vagyos titkosítás során a titkosítandó szöveg bájtjait lefedjük a titkosító kulcs bájtjaival és az egymás alá eső biteken végrehajtunk egy kizáró vagy műveletet. A kizáró vagy 1 értéket ad, ha a két bit különböző és 0 értéket, ha megegyező.

```
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 olvasottBajtok = 0;

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

    for(int i=0; i<olvasottBajtok; ++i) {
        buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]);
        kulcsIndex = (kulcsIndex+1) % kulcs.length;
    }

    kimenőCsatorna.write(buffer, 0, olvasottBajtok);
}

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

A külső while ciklus buffer tömbönként addig olvassa a bemenetet, amíg csak tudja. A belső for ciklusban helyezzük rá a kulcsot a beolvasott bájtokra a kulcsIndex változó segítségével, majd végrehajtjuk a kizáró vagy műveletet, az eredmény a buffer tömbben keletkezik, amit végül a kimenetre írunk.

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

4.4. C EXOR törő

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

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

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

```
    }

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

    double szohossz = atlagos_szo_hossz (titkos, titkos_meret);

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

void
xor (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
xor_tores (const char kulcs[], int kulcs_meret, char titkos[], int ←
    titkos_meret)
{
    xor (kulcs, kulcs_meret, titkos, titkos_meret);

    return tisztalehet (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;

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

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

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

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

    return 0;
}

```

Fordítás: gcc vissza_exor.c -o main -std=c99

Futtatás:

```
time ./main < titkos.txt |grep 5678012
```

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux/

[ch05s02.html](#)

Ebben az esetben végig nézi az összes exorolási lehetőséget. Majd a kódban megadott szavakat keresve állapítja meg, hogy mely exor törés a helyes. Ilyen szavak a: "hogya", "az", "ha", "nem".

4.5. Neurális OR, AND és EXOR kapu

A neurális hálóról lesz szó.

4.1. ábra. A neurális háló egységei

A képen látható egy idegsejt modellje. A neurális hálók irányított kapcsolatokkal összekötött csomópontokból vagy egységekből állnak. Az elipszisbe mutató nyilak a más idegsejtekből beáramló axonok, vagy magából az inputból. A j -edik egységtől az i -edik felé vezető kapcsolat hivatott az a_j aktivációt j -től az i -ig terjeszteni. Az a_0 jelen esetben -1 , de folytatódik a_1 , $a_2 \dots$ és ezeknek is van egy értéke. Ezek a bemeneti értékek. Minden egyes kapcsolat rendelkezik egy hozzá tartozó $W_{j,i}$ numerikus súllyal. Ez határozza meg a kapcsolat erősségét és előjelét. Minden egyes i egység először a bemeneteinek egy súlyozott összegét számítja ki:

Egy neuron akkor tüzel, ha a beérkezett súlyozott összeg átlép egy küszöböt.

A kimenetét úgy kapja, hogy ezek után egy g aktivációs függvényt alkalmaz a kapott összegre:

Az aktivációs függvénnyel szemben két elvárásunk van. Először az, hogy az egység legyen $+1$ körüli kimenet, ha a „helyes” bemeneteket kapja, és 0 körüli kimenet, ha „rossz” bemeneteket kap. Másodszor az, hogy az aktiváció legyen nemlineáris, különben az egész neurális háló egy egyszerű lineáris függvénnyé fajul.

A forráskódokat 'R'-ben fogjuk működtetni.

Neurális OR:

```
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])
```

Betöltjük a neuralnetes csomagot (`library(neuralnet)`). Az 'a1'-nek beadjuk a 0,1,0,1-et ; 'a2'-nek a 0,0,1,1-et mert ha megnézzük ha az egymás alatti elemeket 'OR' logikai műveletezzük (0=hamis, 1=igaz), akkor megkapjuk az 'OR'-t.

Ebből adatot csinálunk :

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

Aztán a neuralnet függvény csinálja meg a neurális hálót R-ben, majd átadjuk neki az adatokat. A tanítás pedig úgy történik, hogy az előtte megadott a1 és a2 segítségével már megtanítjuk, hogy hogyan működik az OR művelet. Azután elkezdi tanítani magát és beállítja a súlyokat. Kijönnek a súlyok és amikor már futtatjuk, befejeződik a tanítás.

A `compute` paranccsal beadjuk neki, hogy ellenőrizze és adja be neki tényleg az a1 és a2 értékeket:

```
> compute(nn.or, or.data[,1:2])
$neurons
$neurons[[1]]
      a1 a2
[1,]  1  0  0
[2,]  1  1  0
[3,]  1  0  1
[4,]  1  1  1
```

Ha lejjebb tekerünk, akkor láthatjuk, hogy nem logikai számítást végez, hanem a tört számokkal számol, összead és alkalmazza a 'g' függvényt.

```
$net.result
[,1]
[1,] 0.00133027
[2,] 0.99900454
[3,] 0.99961042
[4,] 1.00000000
```

Láthatjuk, hogy első esetben hamist kellett kapnunk, és tényleg 0-hoz közelítő értéket kapunk. Második és harmadik esetben igazat és 1-hez közelítő értéket kapunk. Negyedik esetben is igazat és ott már konkrétan 1-et kapunk.

Kiiratjuk az 'nn.or'-t:

```
> print(nn.or)
```

Ebben megtaláljuk az aktivációs függvényt:

```
$act.fct
function (x)
{
  1/(1 + exp(-x))
}
```

Ha ezt a függvényt ábrázoltatjuk a -10 és 10 intervallumon:

```
>curve(1/(1 + exp(-x)), from=-10, to=10)
```

Ezt a függvényt kapjuk:

Itt láthatjuk, hogy ha a súlyozott összegek, vagyis a szummák, mondjuk -10, akkor láthatjuk, hogy az x-hez tartozó y, amelyet az aktivációs függvény csinál az összegből, leviszi 0-ra. Ha mondjuk 0 jön ki, akkor láthatjuk, hogy olyan 0,5 körüli érték jön ki, ha pedig 10 akkor meg majdnem 1. Láthatjuk, hogy 0 és 1 közé transzformálja be a súlyozott összeget.

Neurális AND:

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

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

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

plot(nn.and)

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

Neurális EXOR:

```
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])
```

Hasonlóan járunk el ezekkel a kódokkal is.

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

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/NN_R és https://www.tankonyvtar.hu/tartalom/tamop425/0026_mi_4_4/ch20s05.html

Tanulságok, tapasztalatok, magyarázat...

4.6. Hiba-visszaterjesztéses perceptron

Mi is az a perceptron? Gyakorlatilag egy egy rétegből álló neurális háló. Működése nagyon egyszerűen leírható. A megadott input adatokat megszorozzuk a hozzájuk rendelt súllyal. Ezekkel a súlyokkal tudjuk meghatározni hogy az output szempontjából mely inputok lesznek meghatározóbbak és melyek kevésbé. Ezután a súlyozott input értékeket összegezzük. Az így kapott súlyozott összegen végül alkalmazzuk az aktivációs függvényt. Ez a lépés teszi lehetővé hogy az adatok könnyebben kiértékelhetőek legyenek. Leggyakrabban olyan függvényeket alkalmazunk, amelyek gyorsan 0 vagy 1 felé tartanak, így az outputot egy kis kerekítés után bináris értékben kapjuk.

Megoldás videó:

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

Tanulságok, tapasztalatok, magyarázat...

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

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

#define MERET 600
#define ITER_HAT 32000

void
mandel (int kepadat[MERET][MERET]) {

    // Mérünk időt (PP 64)
    clock_t delta = clock ();
    // Mérünk időt (PP 66)
    struct tms tmsbuf1, tmsbuf2;
    times (&tmsbuf1);

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

    // a számítás
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, reZ, imZ, ujreZ, ujimZ;
    // Hány iterációt csináltunk?
    int iteracio = 0;
    // Végigzongorázzuk a szélesség x magasság rácsot:
    for (int j = 0; j < magassag; ++j)
    {
        //sor = j;
        for (int k = 0; k < szelesseg; ++k)
        {
            // c = (reC, imC) a rács csomópontjainak
```

```
// megfelelő komplex szám
reC = a + k * dx;
imC = d - j * dy;
// z_0 = 0 = (reZ, imZ)
reZ = 0;
imZ = 0;
iteracio = 0;
// z_{n+1} = z_n * z_n + c iterációk
// számítása, amíg |z_n| < 2 vagy még
// nem értük el a 255 iterációt, ha
// viszont elértük, akkor úgy vesszük,
// hogy a kiindulási c komplex számra
// az iteráció konvergens, azaz a c a
// Mandelbrot halmaz eleme
while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
{
    // z_{n+1} = z_n * z_n + c
    ujureZ = reZ * reZ - imZ * imZ + reC;
    ujimZ = 2 * reZ * imZ + imC;
    reZ = ujureZ;
    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)
{
    //sor = j;
    for (int k = 0; k < MERET; ++k)
    {
        kep.set_pixel (k, j,
                        png::rgb_pixel (255 -
                                         (255 * kepadat[j][k]) / ITER_HAT ↔
                                         '
                                         255 -
                                         (255 * kepadat[j][k]) / ITER_HAT ↔
                                         '
                                         255 -
                                         (255 * kepadat[j][k]) / ITER_HAT ↔
                                         ));
    }
}

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

Futtatás: `g++ mandelpngt.c++ -lpng16 -o mandelpngt`

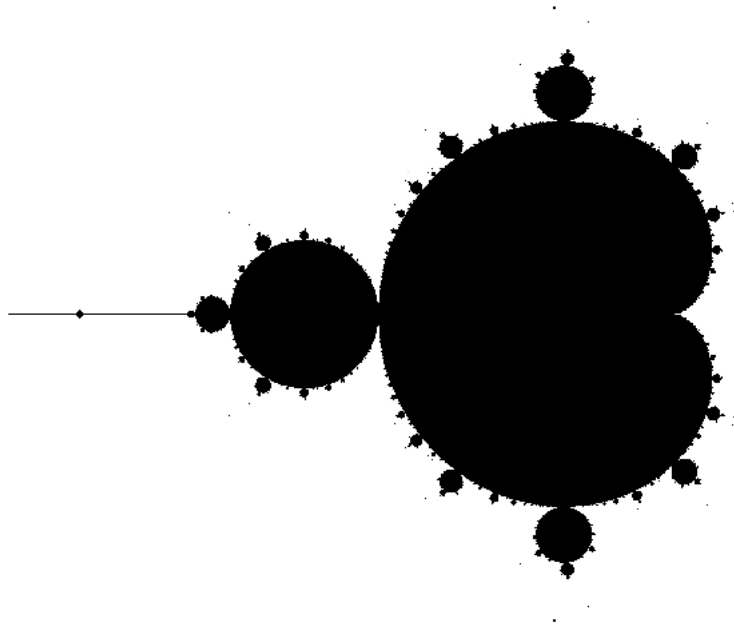
Fordítás és kimentés a t.png fájlként: `./mandelpngt t.png`

A következőt kapjuk:

```
1954
19.5484 sec
t.png mentve
```

Ez csak megmutatja mennyi időbe telt lefutnia a programnak.

Majd megnyitjuk a képet: `eog t.png`



5.1. ábra. A Mandelbrot halmaz a komplex síkon

A Mandelbrot halmazt Benoit Mandelbrot 1980-ban találta meg a komplex számsíkon. Komplex számok azok a számok, amelyek halmazán értelmezni lehet az olyan kérdéseket, mint: Melyik az a két szám, amelyet összeszorozva -4 -et kapunk, mert ez a szám például a $2i$ komplex szám.

A Mandelbrot halmazt úgy láthatjuk meg, hogy a sík origója középpontú 4 oldalhosszúságú négyzetbe lefektetünk egy, mondjuk 800×800 -as rácsot és kiszámoljuk, hogy a rács pontjai mely komplex számoknak felelnek meg. A rács minden pontját megvizsgáljuk a $z_{n+1} = z_n^2 + c$, ($0 \leq n$) képlet alapján úgy, hogy a c az éppen vizsgált rácspont. A z_0 az origó. Alkalmazva a képletet a

- $z_0 = 0$
- $z_1 = 0^2 + c = c$
- $z_2 = c^2 + c$
- $z_3 = (c^2 + c)^2 + c$
- $z_4 = ((c^2 + c)^2 + c)^2 + c$
- ... s így tovább.

Azaz kiindulunk az origóból (z_0) és elugrunk a rács első pontjába a $z_1 = c$ -be, aztán a c -től függően a további z -kbe. Ha ez az utazás kivezet a 2 sugarú körből, akkor azt mondjuk, hogy az a vizsgált rácspont nem a Mandelbrot halmaz eleme. Mivel nem tudunk végtelen sok z -t megvizsgálni, ezért csak véges sok z

elemet nézünk meg minden rácsponthoz. Ha közben nem lép ki a körből, akkor feketére színezzük, hogy az a c rácspont a halmaz része.

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

Megoldás forrása: [bhax/attention_raising/CUDA/mandelpngt.c++](https://github.com/bhax/attention_raising/CUDA/mandelpngt.c++) nevű állománya.

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

A **Mandelbrot halmaz** pontban vázolt ismert algoritmust valósítja meg: [bhax/attention_raising/Mandelbrot/3.1.2.cpp](https://github.com/bhax/attention_raising/Mandelbrot/3.1.2.cpp)

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

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

    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double a = -1.9;
    double b = 0.7;
    double c = -1.3;
    double d = 1.3;

    if ( argc == 9 )
    {
        szelesseg = atoi ( argv[2] );
        magassag = atoi ( argv[3] );
        iteraciosHatar = atoi ( argv[4] );
        a = atof ( argv[5] );
        b = atof ( argv[6] );
        c = atof ( argv[7] );
        d = atof ( argv[8] );
    }
    else
    {
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d ↵
        " << std::endl;
        return -1;
    }

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

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

```

int iteracio = 0;

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

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

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

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

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

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

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

            ++iteracio;
        }

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

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

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

```

Forditas: g++ 3.1.2.cpp -lpng -O3 -o 3.1.2

Futtatás: ./3.1.2 mandel.png 1920 1080 2040 ↵
 -0.01947381057309366392260585598705802112818 ↵
 -0.0194738105725413418456426484226540196687 ↵
 0.7985057569338268601555341774655971676111 ↵
 0.798505756934379196110285192844457924366

Ezután ha megnyitjuk a mandel.png-t ezt a csodát kapjuk.

5.3. Biomorfok

A különbség a Julia halmazok között az, hogy a komplex iterációban az előbbiben a c változó, utóbbiban pedig állandó. A következő Mandelbrot csipet azt mutatja, hogy a c befutja a vizsgált összes rácspontot.

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

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

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

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

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

            ++iteracio;
        }
    }
}
```

Ezzel szemben a Julia halmazos csipetben a c nem változik, hanem minden vizsgált z rácspontra ugyanaz.

```
// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    // k megy az oszlopokon
    for ( int k = 0; k < szelesseg; ++k )
    {
        double reZ = a + k * dx;
        double imZ = d - j * dy;
        std::complex<double> z_n ( reZ, imZ );

        int iteracio = 0;
        for (int i=0; i < iteraciosHatar; ++i)
        {
            z_n = std::pow(z_n, 3) + cc;
            if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
            {
                break;
            }
        }
    }
}
```

```
        {
            iteracio = i;
            break;
        }
    }
```

A teljes kód:

```
// Verzio: 3.1.3.cpp
// Forditas:
// g++ 3.1.3.cpp -lpng -O3 -o
// Futtatas:
// ./3.1.3 bmorf.png 800 800 10 -2 2 -2 2 .285 0 10

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

int
main ( int argc, char *argv[] )
{
    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double xmin = -1.9;
    double xmax = 0.7;
    double ymin = -1.3;
    double ymax = 1.3;
    double reC = .285, imC = 0;
    double R = 10.0;

    if ( argc == 12 )
    {
        szelesseg = atoi ( argv[2] );
        magassag =  atoi ( argv[3] );
        iteraciosHatar =  atoi ( argv[4] );
        xmin = atof ( argv[5] );
        xmax = atof ( argv[6] );
        ymin = atof ( argv[7] );
        ymax = atof ( argv[8] );
        reC = atof ( argv[9] );
        imC = atof ( argv[10] );
        R = atof ( argv[11] );
    }
    else
    {
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c ↵
            d reC imC R" << std::endl;
```



```
        return -1;
    }

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

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

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

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

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

        for ( int x = 0; x < szelesseg; ++x )
        {
            double reZ = xmin + x * dx;
            double imZ = ymax - y * dy;
            std::complex<double> z_n ( reZ, imZ );

            int iteracio = 0;
            for (int i=0; i < iteraciosHatar; ++i)
            {
                z_n = std::pow(z_n, 3) + cc;
                //z_n = std::pow(z_n, 2) + std::sin(z_n) + cc;
                if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
                {
                    iteracio = i;
                    break;
                }
            }

            kep.set_pixel ( x, y,
                           png::rgb_pixel ( (iteracio*20)%255, (iteracio *
                           *40)%255, (iteracio*60)%255 ));
        }

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

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

```
Forditas: g++ 3.1.3.cpp -lpng -O3 -o 3.1.3
Futtatas: ./3.1.3 bmorf.png 800 800 10 -2 2 -2 2 .285 0 10
```

Megoldás videó: <https://youtu.be/IJMbgRzY76E>

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

5.4. A Mandelbrot halmaz CUDA megvalósítása

A CUDA (Compute Unified Device Architecture) – az NVIDIA grafikus processzorainak általános célú programozására használható környezet. Napjainkban minden területen fontos szempont, hogy a kitűzött feladatokat a lehető legrövidebb idő alatt meg lehessen oldani. Így nagy szerepe van a már meglévő algoritmusok párhuzamosításának. Ennek köszönhetően egyre elterjedtebbek a többprocesszoros gépek, melyek egyszerre számos folyamatot képesek kezelni. Amíg a CPU-k csupán néhány szálal tudnak valós időben egymástól függetlenül futtatni, addig a GPU-k nagyságrendekkel nagyobb számú feladatot végeznek el egyszerre. Az eredetileg grafikus feladatokra szánt hardver modellezési és képalkotási feladataiból adódóan nagy számítási teljesítménnyel rendelkezik, amit ma már az általános célú algoritmusok megvalósítása esetén is fel lehet használni. A GPU-k felépítése a parallel működésüknek köszönhetően összetettebbek a hagyományos CPU-khoz képest. A tényleges számítási műveleteket az úgynevezett multiprocesszorok (MP) végzik, melyek mindegyike önálló regiszterekkel és osztott memóriával rendelkezik. Az itt található memória akár 100-szor is gyorsabb lehet a videokártya globális memóriájától. Az MP-ken belül további CUDA magok találhatók. A párhuzamos feldolgozás érdekében az MP-ken warpok vannak definiálva, melyek meghatározott számú folyamatot képesek egyszerre lefuttatni.

Megoldás forrása: [bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu](https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/CUDA/mandelpngc_60x60_100.cu) nevű állománya.

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!

Ehhez a feladathoz szükséges telepíteni a qt4 csomagot.

```
sudo apt-get install libqt4-dev
```

Fájlok amik szükségesek: `frakablak.h` ; `frakablak.cpp` ; `frakszal.h` ; `frakszal.cpp` ; `main.cpp`

Miután megvagyunk a fájlokat a qmake4 segítségével elkészítjük a programot. A mappában lehetőleg csak ezek a fájlok szerepeljenek.

```
qmake-qt4 -project
```

Ezután kapunk egy olyan fájlt, ami a mappa neve, amiben a fájlok vannak. Ezt a:

```
qmake-qt4 QtMandelbrot.pro
```

Ezután legenerálja a fájlokat. Kapunk egy Makefile-t. Le"make"eljük

```
make
```

Futtatjuk:

```
./QtMandelbrot
```

Feldobja ezt az ablakot:

Ezen az ablakon tudjuk nagyítani a képet. Úgy tudjuk megtenni, hogy az egérrel kijelölünk egy területet, amit nagyítani szeretnénk:

Fontos, hogy várjuk meg míg betölt a teljes kép és ne egyből nagyítsunk a következő területre mert buggos lesz. Akövetkező képeken a nagyításokat láthatjuk.

Amikor már újra akarjuk számoltatni, akkor nyomunk egy "n" betűt, ezzel növeljük a halmaz kiszámolásának pontosságát, 256-al megnöveljük az iterációk számát. Az "m" billentyű lenyomásával nagyobb ugrással növeljük a halmaz kiszámolásának pontosságát, 10*256-al megnöveljük az iterációk számát.

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

Megoldás forrása már legeneráltan: <https://github.com/gyorfi-daniel/prog1/tree/master/QtMandelbrot>

5.6. Mandelbrot nagyító és utazó Java nyelven

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Fordítás:

```
g++ polargen.cpp polargen.h polargenteszt.cpp -o polargen
```

Futtatjuk majd ezt kapjuk:

```
./polargen
1.52988
0.754152
1.66029
1.39486
-0.958656
0.526459
-0.923094
-0.767898
-0.397263
0.367265
```

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

Javában:

```
public class PolárGenerátor {

    boolean nincsTárolt = true;
    double tárolt;

    public PolárGenerátor() {
```

```
nincsTárolt = true;
}

public double következő() {

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

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

            w = v1*v1 + v2*v2;
        } while(w > 1);

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

        tárolt = r*v2;
        nincsTárolt = !nincsTárolt;

        return r*v1;

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

public static void main(String[] args) {

    PolárGenerátor g = new PolárGenerátor();

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

Az algoritmus matematikai háttere most számunkra lényegtelen, fontos viszont az eljárás azon jellemzője, hogy egy számítási lépés két normális eloszlású számot állít elő, tehát minden páratlanadik meghíváskor nem kell számolnunk, csupán az előző lépés másik számát visszaadnunk. Hogy páros vagy páratlan lépésben hívtuk-e meg a megfelelő számítást elvégző következő() függvényt, a nincsTárolt logikai változóval jelöljük. Igaz értéke azt jelenti, hogy tárolt lebegőpontos változóban el van tárolva a visszaadandó szám. (Jelen tárgyalásunkban a következő() függvényben implementált matematikai eljárás, a módosított polár-módszer további tulajdonságai teljesen érdektelenek.)

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html#OO_vilag 1.18. példa

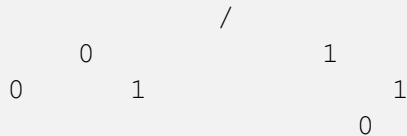
6.2. LZW

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

Jön a 0-1 sorozat, betűnként olvassuk, ha olyan rész jön, ami még "nincs a zsákban", akkor "letörjük és be a zsákba":

```
00011101110 -> 0 00 1 11 01 110
```

A tördelést egy bináris fával könnyen meg tudjuk valósítani:



Fordítás: gcc z.c -o z -std=c99

```
> more b.txt
01111001001001000111
> ./z < b.txt
01111001001001000111

-----1 (4)
-----1 (3)
-----1 (2)
-----0 (3)
-----0 (4)
-----0 (5)
---/ (1)
-----1 (3)
-----0 (2)
-----0 (3)
melyseg=5
```

Megoldás forrása:https://progpater.blog.hu/2011/02/19/gyonyor_a_tomor

6.3. Fabejárás

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

A program 'kiir' részéhez tekerünk:

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

Ez a függvény rekurzívan hívja önmagát a függvényen belül. Az ebben a részben található kiir függvény áthelyezésével érhetjük el, hogy in-, post- vagy preorder kiíratást kapjunk. A következő három esetben láthatjuk, hogy mely változatok, milyen átrendezéssel jönnek ki.

Inorder esetben a fa először kiíratja az 1-es ágat, azután következik a gyökér (/) majd a 0-ás ág:

```

    ++melyseg;
    kiir (elem->egysegGyermeke (), os);
    for (int i = 0; i < melyseg; ++i)
        os << "---";
    os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
    kiir (elem->nullasGyermeke (), os);
    --melyseg;
    break;

```

Postorder esetben a fa először kiírja az 1-es ágat, aztán a 0-ás ágat és azután következik a gyökér (/):

```

    ++melyseg;
    kiir (elem->egysegGyermeke (), os);
    kiir (elem->nullasGyermeke (), os);
    for (int i = 0; i < melyseg; ++i)
        os << "---";
    os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
    --melyseg;
    break;

```

Preorder esetben a fa először kiírja az 0-ás ágat, aztán a 1-es ágat és azután következik a gyökér (/):

```

    ++melyseg;
    for (int i = 0; i < melyseg; ++i)
        os << "---";
    os << elem->getBetu () << "(" << melyseg - 1 << ")" << std::endl;
    kiir (elem->egysegGyermeke (), os);
    kiir (elem->nullasGyermeke (), os);
    --melyseg;
    break;

```

Megoldás forrása:

6.4. Tag a gyökér

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

Megoldás forrása: <https://github.com/gyorfi-daniel/prog1/blob/master/z3a7.cpp>

Az LZW binfa C++ implementációja azzal kezdődik, hogy az egész fát egy **LZWBinFa** osztályba ágyazzuk. Az osztály definíciójába beágyazzuk a fa egy csomópontjának jellemzését, ez lesz a beágyazott Csomópont osztály. Azért ágyazzuk be, mert külön nem szánunk neki szerepet, csak a fa részeként számolunk vele.

```

class LZWBinFa {
public:
    LZWBinFa () : fa (&gyoker) {} ~LZWBinFa ()

```

```

{
    szabadit (gyoker.egyGyermek ());
    szabadit (gyoker.nullasGyermek ());
}

```

Ebben az osztályban a fa gyökere nem pointer, hanem a '/' betűt tartalmazó objektum; a fa viszont már pointer, mindig az épülő LZW-fa azon csomópontjára mutat, amit az input feldolgozásakor az LZW algoritmus diktál. Ez a konstruktor annyit csinál, hogy a fa mutatót ráállítja a gyökérre. Tagfüggvényként túlterheljük a

```
<< operátort
```

, mert ekkor úgy nyomhatjuk a fába az inputot, hogy

```
binFa << b
```

; ahol a b egy '0' vagy '1'-es karakter. Mivel ezen operátor tagfüggvény, így van rá "értelmezve" az aktuális (this "rejtett paraméterként" kapott) példány, azaz annak a fának amibe éppen be akarjuk nyomni a b betűt a tagjai ("fa", "gyoker") használhatóak a függvényben.

```

void operator<< (char b) { // Mit kell betenni éppen, '0'-t?
    if (b == '0') {
        /* Van '0'-s gyermeke az aktuális csomópontnak? megkérdezzük Tőle, a " ←
           fa" mutató éppen reá mutat */
        if (!fa->nullasGyermek ())
            // ha nincs, hát akkor csinálunk
        {
            // elkészítjük, azaz példányosítunk a '0' betű akt. parammal
            Csomopont *uj = new Csomopont ('0');
            // az aktuális csomópontnak, ahol állunk azt üzenjük, hogy
            // jegyezze már be magának, hogy nullás gyereke mostantól van
            // küldjük is Neki a gyerek címét:
            fa->ujNullasGyermek (uj);
            // és visszaállunk a gyökérre (mert ezt diktálja az alg.)
            fa = &gyoker; } else
            // ha van, arra rálépünk
        {
            // azaz a "fa" pointer már majd a szóban forgó gyermekre mutat:
            fa = fa->nullasGyermek (); } }
        // Mit kell betenni éppen, vagy '1'-et?
        else {
            if (!fa->egyGyermek ())
            {
                Csomopont *uj = new Csomopont ('1');
                fa->ujEgyGyermek (uj);
                fa = &gyoker; }
            else
            {
                fa = fa->egyGyermek ();
            }
        }
    }
}

```



```
}  
}
```

6.5. Mutató a gyökér

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

Aggregáció:

Rész-egész kapcsolat, a részek fogják alkotni az egészet. Például a kerékpár váz, kerekek, és kormány aggregációja, itt a részek túlélhetik az egészet. Akkor valósul meg ha egy objektumnak része, tulajdona egy másik. A tartalmazó objektum megszűnésével a tartalmazott objektum tovább élhet.

Kompozíció:

A kompozíció egy speciális aggregáció, itt a rész szorosan hozzátartozik az egészhez, a rész nem éli túl az egészet, például az emberi agy szorosan hozzátartozik az emberhez. Itt a tartalmazott objektum nem lehet egyszerre több objektum része, önmagában nem létezhet, mindig van tartalmazott objektuma és az élettartama azonos a tartalmazó objektumével, a tartalmazó objektum konstruktora hozza létre és a destruktora törli.

Tehát a jelenlegi objektumunkban a gyökér csomópont kompozícióban van a fával, de ha átírjuk mutatóvá akkor már az aggregációs kapcsolat fog megvalósulni.

Átírjuk a gyökeret hogy inentől kezdve mutató legyen.

```
Csomopont *gyoker;
```

Ezután javítjuk a következő szintaktikai hibákat:

```
szabadit (gyoker.egyGyermek ());  
szabadit (gyoker.nullasGyermek ());
```

Mivel a gyökerünk inentől kezdve pointer ezért a '.' helyett itt a nyíl operátort fogjuk használni ami a pointer által mutatott tagra hivatkozik:

```
szabadit (gyoker->egyGyermek ());  
szabadit (gyoker->>nullasGyermek ());
```

Valamint ahol a következő kifejezés szerepelt:

```
&gyoker
```

Kitöröljük az előtte lévő 'and' jelet, hiszen a pointer tartalmazza a címét az objektumnak, nem pedig a pointer címére van szükségünk:

```
gyoker
```

Valamint a konstruktort átírjuk a következőképpen:

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

Itt helyet foglalunk a memóriában, a gyoker által mutatott csomópont számára, valamint a fa mutatót ráállítjuk a gyökérre.

Megoldás forrása: <https://github.com/BorvizRobi/vegyes/blob/master/z3a7mutatogyoker.cpp>

Megoldás videó:

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása: https://gitlab.com/Mark018/prog1/blob/master/Welch/mozgatas_masolas.cpp

```
LZWBinFa & operator= (const LZWBinFa & cp) {
    if(&cp != this)
        rekurzioIndutasa(cp.gyoker);
    return *this;
};
```

Létrehozunk egy operátort, ami ha nem a gyökeret tartalmazza, akkor lemásolja saját magát. A másolás egy rekurzióval végződik, ami a fa minden ágát újra létrehozza egy másik gyökérre.

```
void rekurzioIndutasa(Csomopont csm){
    if(csm.nullasGyermeke()){
        fa = &gyoker;
        Csomopont *uj = new Csomopont ('0');
        fa->ujNullasGyermeke (uj);
        fa = fa->>nullasGyermeke();
        std::cout << "GYOKER: nullas van" << std::endl;
        rekurzioAzAgakon(csm.nullasGyermeke());
    }
    if(csm.egyGyermeke()){
        fa = &gyoker;
        Csomopont *uj = new Csomopont ('1');
        fa->ujEgyGyermeke (uj);
        fa = fa->egyGyermeke();
        std::cout << "GYOKER: egy van" << std::endl;
        rekurzioAzAgakon(csm.egyGyermeke());
    }
}
```

```
void rekurzioAzAgakon(Csomopont * csm){
    if (csm->nullasGyermek()) {
        std::cout << "====van nullas" << std::endl;
        Csomopont *uj = new Csomopont ('0');
        fa->ujNullasGyermek(uj);
    }
    if (csm->egyenesGyermek()){
        std::cout << "====van egyenes" << std::endl;
        Csomopont *uj = new Csomopont ('1');
        fa->ujEgyenesGyermek(uj);
    }
    Csomopont * nullas = fa->nullasGyermek();
    Csomopont * egyenes = fa->egyenesGyermek();
    if(nullas){
        fa = nullas;
        rekurzioAzAgakon(csm->nullasGyermek());
    }
    if(egyenes){
        fa = egyenes;
        rekurzioAzAgakon(csm->egyenesGyermek());
    }
}
```

A rekurzioIndutasa függvény indítja el a rekurziót, ha van nullás gyermeke akkor azon fut tovább, ha van egyes gyermeke akkor arra is meghívásra kerül. A fő eljárást maga a rekurzioAzAgakon függvény végzi, ez fut át az összes ágon, és létrehozza az új csomópontokat.

```
LZWBinFa binFa2;
    binFa2 = binFa;
```

A másolás már csak az egyenlőség jel operátorral meghívva történik, így az alap binFa átmásolódik a binFa2-be.

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

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

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

Megoldás forrása: <https://github.com/gyorfi-daniel/prog1/tree/master/Myrmecologist>

A program futtatásához szükségünk van a libqt4-dev csomagra.

Az argumentumokat a main függvény kéri be, és deklarálja a bemeneti argumentumokkal az AntWin osztályt. Megjelenítésért az AntWin osztály a felelős. Az AntThread osztály végzi a 'hangyák' lehelyezését és a számolást. Itt az AntThread::newDir kalkulálja, hogy melyik irányba haladjon a hangya. Az AntThread::moveAnts függvényben történik a kiszámolása ezeknek a műveleteknek. Az AntThread megállítható és újra indítható művelet, mivel ez külön szálon fut, A keyPressEvent-nél be van bindelve a 'P' betű erre.

7.2. Java életjáték

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

A program futtatásához szükségünk van: `sudo apt-get install libqt4-dev`

A program elkészítéséhez szükség van egy pár parancs lefuttatásához. Először is egy mappába kell szedni az összes fájlt amire szükségünk van. A terminálon belépünk abba a mappába (nekem a mappám neve sejtauto volt) majd:

```
qmake-qt4 -project
qmake-qt4 sejtauto.pro
make
./sejtauto
```

Conway a sejtautomata-tervét a minimumig igyekezett egyszerűsíteni, amikor ezeket a „genetikai szabályokat” igen átgondoltan, hosszú kísérletezés után kiválasztotta. Röviden: a szabályok olyanok legyenek, hogy a népesség viselkedése lehetőleg ne legyen előre megjósolható.

Túlélés: minden sejt, amelynek kettő vagy három szomszédja van, életben marad a következő generációra.

Halál: minden sejt, amelynek négy vagy több szomszédja van, meghal a túl-népesedéstől! Ugyancsak minden sejt meghal a következő generációra, amelynek kevesebb mint két egy szomszédja van – a halál oka: elszigetelő-dés.

Születés: ha egy üres cellának pontosan három élő sejt van a szomszédjaiban, akkor ott új sejt születik.

Kapunk egy ablakot, amelyen bal alul található 2 'blokk', amelyek egymás felé küldenek egy-egy 'csomót' amelyek összeütöknék és létrehoznak kis 'sikló'-kat, amik átlósan felfelé haladnak. Ez a folyamat addig tart, amíg ezek a kis mozgó siklók el nem érik az 'ágyúkat', aztán meghalnak. A 2x2-es fekete négyzetek nem mozognak és nem változnak.

Megoldás részletesebb áttanulmányozásáért nagyon szépen és átláthatóan le van írva: <http://becskeha.blogspot.com/2015/04/game-of-life-eletjatek.html>

Pár illusztráció a történetekről:

7.3. Qt C++ életjáték

Most Qt C++-ban!

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

Az előzőekhez hasonlóan minden ugyanúgy történik.

Kapunk egy ablakot, amelyen bal alul található 2 sejt, amelyek egymás felé küldenek egy-egy 'csomót' amelyek összeütöknék és létrehoznak kis 'sikló'-kat, amik átlósan felfelé haladnak. Ez a folyamat addig tart, amíg ezek a kis mozgó siklók el nem érik a az 'ágyúkat', aztán meghalnak. A 2x2-es fekete négyzetek nem mozognak és nem változnak.

7.4. BrainB Benchmark

A Samu Entrophy-n belül lévő kék körben kell a játékosnak kurzort lenyomva tartani, ez eleinte nem nehéz, de az bizonyos idő elteltével egyre jobban elkezdnek mozogni a pontok, így egyre nehezebb követni, hogy hol jár. A konzolon kiírja ha hibázunk. Folyamatosan méri a reakció időnkét bit/sec-ban.

10 percig fut a játék, de akár mikor lehetőségünk van kilépni akár az idő lejártá előtt is. Ekkor is láthatjuk az eredményeinket.

Fordítani és futtatni úgy tudjuk, ha a szükséges fájlok mind egy mappába vannak és a szokásos qmake-s eljárást kell alkalmaznunk itt is

Megoldás forrása: <https://github.com/nbatfai/SamuBrain>

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python PASSZ

Megoldás videó: <https://youtu.be/j7f9SkJR3oc>

Megoldás forrása: <https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0> (/tensorflow-0.9.0/tensorflow/exa
https://progpater.blog.hu/2016/11/13/hello_samu_a_tensorflow-bol

Tanulságok, tapasztalatok, magyarázat...

8.2. Mély MNIST

Python PASSZ

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Minecraft-MALMÖ

PASSZ

Megoldás videó: <https://youtu.be/bAPSu3Rndi8>

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

A Lisp programozási nyelv általános célú programnyelvnek tervezték az 50-es évek legvégén, de hamarosan a mesterséges intelligencia kutatás előszeretettel alkalmazott nyelvévé vált. Ma a Lisp nyelveket számos területen alkalmazzák, és közkedvelt a számításelmélet oktatásában is. A Lisp név az angol „List Processing” (listafeldolgozás) kifejezésre vezethető vissza (maga a lisp szó angolul pöszét, pöszítést jelent). A Lisp nyelvek fő adatstruktúrája ugyanis a láncolt lista. Az alapvető listakezelő műveletek az összes nyelvjárásban megegyeznek. Emellett közös sajátosság a zárójelezett prefix-jelölés, a futásidejű típusosság, a funkcionális programozásra jellemző jegyek (pl. rekurzív függvények, Lambda-kalkulus), és az, hogy a programkód adatként manipulálható, tehát végrehajtható. A változatok többsége értelmezőprogrammal fut, de sok közülük tartalmaz beépített fordítót is. Tetszőleges Lisp nyelven írt program azonnal felismerhető a jellegzetes szintaktikájáról. A programkód ugyanis egymásba ágyazott listák, azaz zárójelezett, ún. S-kifejezések (S-expression, sexp) sorozata. Ennek a primitív szintaktikának köszönhetően a Lisp nyelveken írt programokhoz nagyon egyszerű elemzőt (parser-t) és kódot generáló (meta-) programokat írni, ugyanakkor emberi szemmel könnyű elveszni a nyitó- és csukózárójelek erdejében. A tömör kifejezőerején kívül a könnyű gépi elemezhetőség volt a másik oka a nyelv népszerűségének a 60-as években, amikor még sokszor monitor nélküli terminálokra vagy lyukkártyával, lyukszalaggal történt a bevitel, jellemzően nagygépes kötegelt végrehajtási (batch) környezetben, és nem mindenhol volt elegendő számítási kapacitás összetett, többmenetes fordító- és értelmezőprogramok futtatásához sem. Formális specifikációjának első, 1958-ban készített változatával a Lisp a második legöregebb magas szintű programozási nyelv (a Fortran után). A megalkotása óta elmúlt bő 50 évben a nyelv sokat változott és számos nyelvjárással gazdagodott. A ma legelterjedtebb változatai az általános célú Common Lisp és Scheme nyelvek.

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

Megoldás forrása: [https://hu.wikipedia.org/wiki/Lisp_\(programoz%C3%A1si_nyelv\)](https://hu.wikipedia.org/wiki/Lisp_(programoz%C3%A1si_nyelv))

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

Ehhez a feladathoz a GIMP nevezetű képszerkeztő programra lesz szükségünk. A GIMP (GNU Image Manipulation Program) egy rendkívül nagytudású bittérképes vagy más néven pixelgrafikus képszerkesztő program, amely elérhető Windows, Linux és Mac OS X változaton is. A programon belül nem a grafikus felületet fogjuk használni, hanem a programban található konzolon kell dolgoznunk, amely megtalálható: Szűrők/Script-fu/Konzol. A feladathoz szükséges, hogy a Gimp program mappájában lévő 'scripsts' mappába bemásoljuk a fájlokat, amikkel dolgozunk, ezesetben a 'Chrome' mappában lévő 'bhax_chrome3.scm' és 'bhax_chrome3_border2.scm' fájlokat. A fájlok alján megtalálható, hogy mit kell beírni a konzolba az adott program futásához és melyik argumentuma mit jelent.

A 'bhax_chrome3.scm' scriptben található parancs:

```
(script-fu-bhax-chrome "Bátf41 Haxor" "Sans" 120 1000 1000 '(255 0 0) " ←  
Crown molding")
```

A 'bhax_chrome3_border2.scm' scriptben található első parancs:

```
(script-fu-bhax-chrome-border "Bátf41 Haxor Stream" "Sans" 160 1920 1080 ←  
400 '(255 0 0) "Crown molding" 7)
```

A 'bhax_chrome3_border2.scm' scriptben található második parancs:

```
(script-fu-bhax-chrome-border "Programozás" "Sans" 110 768 576 300 '(255 0 ←  
0) "Crown molding" 6))
```

9.3. Gimp Scheme Script-fu: név mandala

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

Megoldás videó: https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelese_a_scheme_programozasi_nyelv

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

Az előző feladathoz hasonlóan a Gimp programot használjuk. Minden ugyan az, csak most Mandalát kell csinálni a nevünkéből. A feladathoz itt is szükséges, hogy a Mandala mappából a 'bhax_mandala9.scm' átmásoljuk a Gimp program scriptjei közé. Ennek a scriptnek is megtalálható a végén is megtalálható a parancs, amivel futtatni tudjuk a mandala készítését. Az argumentumok megmagyarázása is ott található.

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

Tutoriál: Mátravölgyi Adrián

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

Modellezés:

Objektumok, egyedek – tulajdonságaik és viselkedésmódjuk van. Az egyedek osztályozhatóak, kategorizálhatóak. A valós világ túl bonyolult, ezért az emberi gondolkodás az absztrakción alapul, ezért modellekben gondolkodunk. Az absztrakció lényege a közös és lényeges dolgok kiemelése.

Számítógépen az egyedek tulajdonságait az adat, viselkedésmódjaikat a programok szemléltetik. Ezt adatmodellnek nevezzük.

Alapfogalmak:

A magas szintű nyelven megírt programot forrásszövegnek nevezzük, a rá vonatkozó formai követelményeket a szintaktikai szabályok, a tartalmi szabályokat a szemantikai szabályok definiálják. Minden processzor saját gépi nyelvvel rendelkezik, így csak az adott gépi nyelven írt programokat tudja végrehajtani. Tehát a magas szintű nyelven írt forrásszövegeket gépi nyelvű programmá kell alakítani. Ennek két módja van: fordítóprogramos és interpreteres.

Tárgyprogramot csak a szintaktikailag helyes forrásprogramból lehet előállítani. Az előállított program gépi nyelvű, de még nem futtatható. Belőle futtatható programot a kapcsolatszerkesztő állít elő. Majd a futtatható programot a betöltő helyezi a tárba és adja át neki a vezérlést. A futó program vezérlését a futtató rendszer felügyeli.

A magas szintű programozási nyelvek között létezik olyan, amelyben olyan forrásprogramot lehet, amely tartalmaz nem nyelvi elemeket is. Ilyenkor egy előfordító segítségével először a forrásprogramból egy adott nyelvű forrásprogramot állítunk elő, majd ezt feldolgozzuk a nyelv fordítójával. Az interpreteres technikai esetén nem készül tárgyprogram, utasításonként sorra veszi a forrásprogramot, értelmezi, s végrehajtja.

Minden nyelvnek megvan a saját szabványa, amit hivatkozási nyelvnek nevezünk, ebben vannak definiálva a szintaktikai és szemantikai szabályok. A hivatkozási nyelvek mellett léteznek még implementációk. Ezek egy adott platformon realizált fordítóprogramot vagy interpreterek.

Az implementációk nagy problémája mai napig is a hordozhatóságban rejlenek. Gyakorlatilag lehetetlen megoldani, hogy az egyik implementációban működő program a másik implementációban is ugyan azt az eredményt adja. Napjainkban a programok írásához grafikus integrált fejlesztői környezetek állnak rendelkezésünkre. Ezek tartalmazznak szövegszerkesztőt, fordítót, kapcsolatszerkesztőt, betöltőt, futtató rendszert és belövőt is.

A programnyelvek osztályozása:

Imperatív nyelvek (eljárásorientált nyelvek, objektumorientált nyelvek), Deklaratív nyelvek (funkcionális nyelvek, logikai nyelvek) és máselvi nyelvek.

Karakterkészlet:

A forrásszöveg legkisebb része a karakter. Egy nyelvben megjelenhető karaktereket karakterkészlet tartalmazza. A karakterekből állíthatóak elő a bonyolultabb nyelvi elemek.

Minden nyelvnek saját karakterkészlete van, általában a következő kategóriák alapján csoportosítva a karaktereket: betűk, számok, egyéb karakterek. Minden nyelvben betű az angol ABC 26 nagybetűje. Egyes nyelvekben (FORTRAN) a kisbetűk nem betűk. A Pascal szerint a kis- és nagybetű azonos, a C szerint nem. A legtöbb nyelv a nemzeti betűket nem tartja betűnek. A számokat minden nyelv egységesen kezeli, a decimális számok mindenhol számok. Egyéb karakterek a műveleti jelek, elhatároló jelek, írásjelek és a speciális karakterek (pl.: ~). A szóköz kitüntetett szerepű.

Lexikális egységek:

- Többkarakteres szimbólum (általában operátorok, pl.: ++).
- Szimbolikus név (azonosító: betűvel kezdődik, számmal vagy betűvel folytatódhat, kulcsszó: a nyelv tulajdonít neki jelentést – IF, FOR, CASE, standard azonosító: a nyelv tulajdonít neki jelentést, de ez a jelentés megváltoztatható - NULL).
- Címke (a végrehajtható utasítások megjelölésére szolgál, hogy a program egy másik pontjából hivatkozni tudjunk rá, COBOL: nincs, Pascal: max. 4 jegyű e.n. e.sz., FORTRAN: max. 5 jegyű e.n. e.sz., PL/I,C,Ada: azonosító).
- Megjegyzés (olyan rész helyezhető el segítségével, amelyet a fordító program a lexikális elemzés során amúgy is töröl, így nem befolyásolja a programot. Három féle megjegyzés létezik: egész soros megjegyzés, sor végi megjegyzés, szóközők közötti megjegyzés).
- Literál (fix értékkel rendelkező programozási eszköz, 2 komponense van: típus, érték).

A forrásszöveg összehasonlításának általános szabályai:

Kötött formátumú nyelvekben alapvető szerepet játszik a sor, ugyanis azt vallják, hogy egy sor egy utasításnak felel meg. Ha egy utasítás nem fér el egy sorban, azt külön jelölni kell. Több utasítás egy sorban soha nem állhat. Szabad formátumú nyelvekben a sor és az utasítás között nincs kapcsolat, egy sorban akárhány utasítás lehet, és egy utasítás is állhat több sorból. Mivel nem a sor vége jelöl az utasítás végét, így egy speciális karakter jelöli az utasítás végjelét, ez általában egy pontosvessző. Az eljárásorientált nyelvekben a lexikális egységeket valamilyen elhatároló jellel (vessző), vagy szóközzel kell elválasztani egymástól. Az elhatároló jelek tetszőlegesen ismételhettek.

Adattípusok:

Léteznek típusos nyelvek és nem típusos nyelvek. Vannak egyszerű, összetett és mutató adattípusok.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

Szintaktikai egységek:

6 szintaktikai egységet különböztetünk meg: azonosítók, kulcsszavak, állandók, karakterláncok, operátorok és egyéb szeparátorok. a C fordító nem veszi figyelembe a szóközöket, tabulátorokat, újsorokat, megjegyzéseket (közös nevükön üres helyeket).

Az azonosítók értelmezése:

A C nyelv az azonosítók értelmezését a tárolási osztályára és a típusára alapozza. A tárolási osztály az azonosítóhoz rendelt tárhely elhelyezkedését és élettartamát, a típus az azonosítóhoz rendelt tárterületen talált értékek jelentését határozza meg. Négy deklarálható tárolási osztály van: automatikus, statikus, külső és regiszterosztály.

Objektumok és balértékek:

Az objektum a tár valamely műveletekkel kezelhető része; a balérték (lvalue) objektumra hivatkozó kifejezés. A balérték kifejezésre kézenfekvő példa az azonosító. Bizonyos operátorok balértékeket eredményeznek: ha E mutató típusú kifejezés, akkor *E olyan balérték kifejezés, amely arra az objektumra hivatkozik, amire az E mutat. A balérték elnevezés az E1 =E2 értékadó kifejezésből származik, amelyben az E1 bal oldali operandusnak balérték kifejezésnek kell lennie. Az egyes operátorok alább következő ismertetése során közöljük hogy az adott operátor balérték operandusokat vár-e és hogy balértéket ad-e eredményül.

Konverziók:

Operandusuktól függően számos operátor válthatja ki valamelyik operandusa értékének egyik típusból valamilyen másik típusba történő átalakítását.

Kifejezések:

A legmagasabb precedencia az első. így pl. azokat a kifejezéseket, amelyekre mint a + operandusaira hivatkozunk. Az operátorokra bal-, ill. jobbirányú asszociativitás vonatkozik. A kifejezésekben alkalmazott operátorok precedenciáját és asszociativitását nyelvtan foglalja össze. Egyéb esetekben a kifejezések kiértékelésének sorrendje határozatlan. A fordítóprogram a részkifejezéseket saját megítélése szerint abban a sorrendben számítja ki, amit leghatékonyabbnak vél, még abban az esetben is, ha a részkifejezéseknek mellékhatásaik vannak. A mellékhatások előfordulásának sorrendje meghatározott. Kommutatív és asszociatív operátorokat (*, +, &, |, ~) tartalmazó kifejezések tetszés szerint rendezhetők még zárójelek jelenlétében is; ha adott sorrendben végzendő kiértékelést kívánunk előírni, explicit ideiglenes változót kell használnunk. A kifejezések kiértékelése során a túlcsordulás és az osztás ellenőrzésének kezelése gépfüggő. A C nyelv minden létező megvalósítása figyelmen kívül hagyja az egészek túlcsordulását; a 0-val való osztás kezelése, ill. a lebegőpontos kivételek gépről gépre változnak, és általában valamilyen könyvtári függvénnyel módosíthatók.

Deklarációk:

A deklarációk segítségével határozzuk meg, hogyan értelmezze a C fordító az egyes azonosítókat; a deklarációk nem feltétlenül jelentenek tárterület-foglalást az azonosító számára.

10.3. Programozás

[BMECPP]

A C++ általános célú programozási nyelv, amely lehetővé teszi az objektumorientált és generikus programozást, ugyanakkor alacsony szintű nyelvi konstrukciókat is támogat. A C++-t Bjarne Stroustrup fejlesztette ki. Az első verzió 1983-ban jelent meg. A C++ a C nyelvre épül: az első C++ fordítók C kódot generáltak.

A C-t sokan a C++ nyelv egy részhalmazának tekintik, hiszen a C++ a C szintaxisra épít, azt terjeszti ki. A legtöbb C program valóban C++ program is de van köztük néhány különbség.

Megjelent a bool típus a C++ nyelvben. Ez vagy 'true' vagy 'false' értéket vehet fel.

A 'C++'-ban a függvény neveket és az argumentumajikat együtt olvassa. Ez lehetőséget nyújt azonos nevű függvényekhez. Az argumentumaihoz már meg tudunk adni alapértelmezett értékeket is.

A 'C++'-ban megjelent a referenciatípus is, ami a pointerek szerepét felváltja.

Függvények túlterhelésére 'C++'-ban már van lehetőség, míg ez C-ben még nem.

Operátorok és túlterhelésük

Az operátorok az argumentumaik segítségével végeznek számításokat, amelyeket felhasználjuk a visszatérési értékek felhasználásával. Az operátorok sorrendjét befolyásolhatjuk zárójelekkel. A 'C++'-a C-hez képest bevezet néhány új operátort. Szerencsére nagyon rugalmas működéssel rendelkeznek.

A C-vel ellentétben, a 'C++'-ban lehetőség van referencia szerinti paraméterátadásra, míg C-ben csak pointerekkel van.

C++ osztálysablonok és a függvénysablonok definiálásakor bizonyos elemeket nem adunk meg, hanem paramétereként kezeljük. Ez lehet explicit vagy implicit módon is. Ezek generikus típusú sablonok.

A 'C++'-nyelvben a stream bájtok sorozatát jelenti. Ezeknek a stream-eknek lehet bemeneti vagy kimeneti adatfolyamuk. A rendszerhívások adatfolyamokat egy bufferrel látják el, amelyeket osztályok példányai valószínűleg meg. Ezek a bufferek össze gyűjtik a karaktersorozatokat, és több cout kiírást egy rendszerhívással írnak ki. Ezzel ellentétben az endl, illetve egy következő cin beolvasás kiírja a couthoz tartozó buffereket. A bufferek ürítéséhez használhatjuk a 'flush' parancsot.

A C++ állománykezeléshez adatfolyamokat használ. Ezek a stream-ek bemeneti és kimeneti állományokból állnak. Van lehetőség kétirányúra is (fstream). Objektumorientáltság lehetővé teszi a konstruktorok és destruktorok használatát.

Konstruktorok segítségével inicializálni tudjuk az osztályokat. Destruktorokkal pedig kiürítjük őket a memóriából.

A kivételkezelés olyan mechanizmus, amely biztosítja, hogy ha hibát fedezünk fel valahol, akkor a futás azonnal a hibakezelő ágon folytatódjon. A kivételkezelést nem véletlenül hívják kivétel-, és nem hibakezelésnek. A megoldás bármilyen kivételes helyzet esetén felhasználható, nem csak hiba esetén. A kivételkezelés mechanizmusa azt biztosítja, hogy kivétel esetén a futás azonnal a kivételkezelőre ugorjon. Például, ha van egy függvényünk ami egy számnak visszaadja a reciprokértékét. Azt az esetet figyelembe kell venni, hogy ha 0-t kapunk bemenetről, mivel 0-nak nincs reciproka. Ekkor lép életbe a kivételkezelés. A main függvénybe egy try-catch blokkot szoktak berakni ami a kivételkezeléssel foglalkozik. A try védett blokkba a { } közé írjuk be a normál működés kódját, illetve a catch ágba { } közé a hibakezelő kódot. Ha nincs hiba akkor a catch ág nem fut le, hanem csak a try részben lévő valamennyi paramcs fog lefutni. Ha mégis van hiba, akkor a throw kulcsszóval kivételt dobunk, majd a catchnál "elkapjuk".

Egy kivétel dobásakor annak elkapásáig a függvények hívásai láncában felfelé haladva az egyes függvények lokális változói felszabadulnak. Ezt a folyamatot a hívási verem visszacsévézésének nevezzük.

III. rész

Második felvonás

**Bátf41 Haxor Stream**

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

11. fejezet

Helló, Arroway!

11.1. OO szemlélet

A módosított polártranszformációs normális generátor beprogramozása Java nyelven. Mutassunk rá, hogy a mi természetes saját megoldásunk (az algoritmus egyszerre két normálist állít elő, kell egy példánytag, amely a nem visszaadottat tárolja és egy logikai tag, hogy van-e tárolt vagy futtatni kell az algot.) és az OpenJDK, Oracle JDK-ban a Sun által adott OO szervezés ua.! <https://arato.inf.unideb.hu/batfai.norbert/UDPROC> (16-22 fólia) Ugyanezt írjuk meg C++ nyelven is! (lásd még UDPROG repó: [source/labor/polargen](#))

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Homokozó

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.3. „Gagyí”

Az ismert formális „while (x <= t && x >= t && t != x);” tesztkérdéstípusra adj a szokásosnál (miszerint x, t az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciája) „mélyebb”

választ, írd Java példaprogramot mely egyszer végtelen ciklus, más x, t értékekkel meg nem! A példát építsd a JDK Integer.java forrására³, hogy a 128-nál inkluzív objektum példányokat poolozza!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.4. Yoda

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.5. Kódolás from scratch

Induljunk ki ebből a tudományos közleményből: <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/bbpalg.pdf> és csak ezt tanulmányozva írjuk meg Java nyelven a BBP algoritmus megvalósítását! Ha megakadsz, de csak végső esetben: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok/javat/apbs02.html#pi_jegyei (mert ha csak lemásolod, akkor pont az a fejlesztői élmény marad ki, melyet szeretném, ha átélnél).

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12. fejezet

Helló, Liskov!

12.1. Liskov helyettesítés sértése

Írjunk olyan OO, leforduló Java és C++ kódcsipetet, amely megsérti a Liskov elvet! Mutassunk rá a megoldásra: jobb OO tervezés. https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf (93-99 fólia) (számos példa szerepel az elv megsértésére az UDPROG repóban, lásd pl. [source/binom/Batfai-Barki/madarak/](#))

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.2. Szülő-gyerek

Írjunk Szülő-gyerek Java és C++ osztálydefiníciót, amelyben demonstrálni tudjuk, hogy az ősön keresztül csak az ős üzenetei küldhetőek! https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_1.pdf (98. fólia)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.3. Anti OO

A BBP algoritmussal a Pi hexadecimális kifejtésének a 0. pozíciótól számított 10^6 , 10^7 , 10^8 darab jegyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket! <https://www.tankonyvtar.hu/hu/tartalom/tanitok-javat/apas03.html#id561066>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.4. deprecated - Hello, Android!

Élesszük fel a <https://github.com/nbatfai/SamuEntropy/tree/master/cs> projektjeit és vessünk össze néhány egymásra következőt, hogy hogyan változtak a források!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.5. Hello, Android!

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.6. Hello, SMNIST for Humans!

Fejleszd tovább az SMNIST for Humans projektet SMNIST for Anyone emberre szánt appá! Lásd az [smnist2_kutatasi_jegyzokonyv.pdf](#)-ben a részletesebb hátteret!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.7. Ciklomatikus komplexitás

Számoljuk ki valamelyik programunk függvényeinek ciklomatikus komplexitását! Lásd a fogalom tekintetében a https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_2.pdf (77-79 fóliát)!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13. fejezet

Helló, Mandelbrot!

13.1. Reverse engineering UML osztálydiagram

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.2. Forward engineering UML osztálydiagram

UML-ben tervezzünk osztályokat és generáljunk belőle forrást!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.3. Egy esettan

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.4. BPMN

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.5. BPEL Helló, Világ! - egy visszhang folyamat

Egy visszhang folyamat megvalósítása az alábbi teljes „videó tutorial” alapján: https://youtu.be/0OnlYWX2v_I

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.6. TeX UML

Valamilyen TeX-es csomag felhasználásával készíts szép diagramokat az OOCWC projektről (pl. use case és class diagramokat).

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14. fejezet

Helló, Chomsky!

14.1. Encoding

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.2. OOCWC lexer

Izzítsuk be az OOCWC-t és vázoljuk a <https://github.com/nbatfai/robocaremulator/blob/master/justine/rcemu/src/lexert> és kapcsolását a programunk OO struktúrájába!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.3. l334d1c45

Írj olyan OO Java vagy C++ osztályt, amely leet cipherként működik, azaz megvalósítja ezt a betű helyettesítést: <https://simple.wikipedia.org/wiki/Leet> (Ha ez első részben nem tette meg, akkor írasd ki és magyarázd meg a használt struktúratömb memóiafoglalását!)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.4. Full screen

Készítsünk egy teljes képernyős Java programot! Tipp: https://www.tankonyvtar.hu/en/tartalom/tkt/javat-tanitok-javat/ch03.html#labirintus_jatek

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.5. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.6. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Lásd vis_prel_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, textúrázás, a szintek jobb elkülönítése, kézreállóbb irányítás.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.7. Paszigráfia Rapszódia LuaLaTeX vizualizáció

Lásd vis_prel_para.pdf! Apró módosításokat eszközölj benne, pl. színvilág, még erősebb 3D-s hatás.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.8. Perceptron osztály

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15. fejezet

Helló, Stroustrup!

15.1. JDK osztályok

Írjunk olyan Boost C++ programot (indulj ki például a fénykardból) amely kilistázza a JDK összes osztályát (miután kicsomagoltuk az src.zip állományt, arra ráengedve)!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.2. Másoló-mozgató szemantika

Kódcsipeteken (copy és move ctor és assign) keresztül vedd össze a C++11 másoló és a mozgató szemantikáját, a mozgató konstruktort alapozd a mozgató értékadásra!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.3. Hibásan implementált RSA törése

Készítsünk betű gyakoriság alapú törést egy hibásan implementált RSA kódoló: <https://arato.inf.unideb.hu/batfai.r> (71-73 fólia) által készített titkos szövegen.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.4. Változó argumentumszámú ctor

Készítsünk olyan példát, amely egy képet tesz az alábbi projekt Perceptron osztályának bemenetére és a Perceptron ne egy értéket, hanem egy ugyanakkora méretű „képet” adjon vissza. (Lásd még a 4 hét/Perceptron osztály feladatát is.)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.5. Összefoglaló

Az előző 4 feladat egyikéről írd egy 1 oldalas bemutató „”esszé szöveget!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16. fejezet

Helló, Gödel!

16.1. Gengszterek

Gengszterek rendezése lambdával a Robotautó Világbajnokságban <https://youtu.be/DL6iQwPx1Yw> (8:05-től)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.2. C++11 Custom Allocator

<https://prezi.com/jvvbytkwgsxj/high-level-programming-languages-2-c11-allocators/> a CustomAlloc-os példa, lásd C forrást az UDPROG repóban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.3. STL map érték szerinti rendezése

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.4. Alternatív Tabella rendezése

Mutassuk be a https://progpater.blog.hu/2011/03/11/alternativ_tabella a programban a java.lang Interface Comparable T szerepét!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.5. Prolog családfa

Ágyazd be a Prolog családfa programot C++ vagy Java programba! Lásd [para_prog_guide.pdf](#)!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.6. GIMP Scheme hack

Ha az előző félévben nem dolgoztad fel a témát (például a mandalás vagy a króm szöveges dobozosat) akkor itt az alkalom!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17. fejezet

Helló, !

17.1. FUTURE tevékenység editor

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.2. OOCWC Boost ASIO hálózatkezelése

Mutassunk rá a scanf szerepére és használatára! <https://github.com/nbatfai/robocaremulator/blob/master/justine/ro>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.3. SamuCam

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.4. BrainB

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.5. OSM térképre rajzolása

Debrecen térképre dobjunk rá cuccokat, ennek mintájára, ahol én az országba helyeztem el a DEAC hekkereket: <https://www.twitch.tv/videos/182262537> (de az OOCWC Java Swinges megjelenítőjéből: <https://github.com/emulator/tree/master/justine/rcwin> is kiindulhatsz, mondjuk az komplexebb, mert ott időfejlődés is van...)

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18. fejezet

Helló, Schwarzenegger!

18.1. Port scan

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.2. AOP

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.3. Android Játék

Írjunk egy egyszerű Androidos „játékot”! Építkezzünk például a 2. hét „Helló, Android!” feladatára!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.4. Junit teszt

A https://progater.blog.hu/2011/03/05/labormeres_otthon_avagy_hogyan_dolgozok_fel_egy_pedat poszt kézzel számított mélységét és szórását dolgozd be egy Junit tesztbe (sztenderd védési feladat volt korábban).

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.5. OSCI

Készíts egyszerű C++/OpenGL-es megjelenítőt, amiben egy kocsit irányítasz az úton. A koci állapotát minden pillanatban mentsd le. Ezeket add át egy Prolog programnak, ami egyszerű reflex ágensként adjon vezérlést a kocsinak, hasonlítsd össze a kézi és a Prolog-os vezérlést. Módosítsd úgy a programodat, hogy ne csak kézzel lehessen vezérelni a kocsit, hanem a Prolog reflex ágens vezérelje!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19. fejezet

Helló, Calvin!

19.1. MNIST

Az alap feladat megoldása, +saját kézzel rajzolt képet is ismerjen fel, https://progpater.blog.hu/2016/11/13/hello_samu_a_mnist bol Hátterként ezt vetítsük le: <https://prezi.com/0u8ncvvoabcr/no-programming-programming/>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.2. Deep MNIST

Mint az előző, de a mély változattal. Segítő ábra, vedd össze a forráskóddal a <https://arato.inf.unideb.hu/batfai.norbert> 8. fóliáját!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.3. CIFAR-10

Az alap feladat megoldása, +saját fotót is ismerjen fel, https://progpater.blog.hu/2016/12/10/hello_samu_a_cifar-10_tf_tutorial_peldabol

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.4. Android telefonra a TF objektum detektálója

Telepítsük fel, próbáljuk ki!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.5. SMNIST for Machines

Készíts saját modellt, vagy használj meglévőt, lásd: <https://arxiv.org/abs/1906.12213>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.6. Minecraft MALMO-s példa

A <https://github.com/Microsoft/malmo> felhasználásával egy ágens példa, lásd pl.: <https://youtu.be/bAPSu3Rndi8>, https://bhaxor.blog.hu/2018/11/29/eddig_csaltunk_de_innentol_mi, <https://bhaxor.blog.hu/2018/10/28/minecraft>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

20. fejezet

Helló, Arroway!

20.1. C++ és Java összehasonlítás

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

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

Számos C++ kifejezés, utasítás szintaktikailag helyes Java-ban is, és sokszor hasonló jelentéssel bír. A hasonlóság azonban nem azonosság. A Java mint nyelv szűkebb a C++-nál, ugyanakkor szabványos osztálykönyvtárai révén szélesebb alkalmazási területet fed le: nyelvi szinten támogatja a thread-eket, a grafikus felhasználói felület programozását, a hálózati programozást, a perzisztenciát, különböző processzek közötti osztott objektumelérést, adatbázisok tartalmának elérését és így tovább. Ezeket a fogalmakat persze C++-ban is lehet kezelni, de külső könyvtár segítségével kell igénybe vennünk. A C++ lehetővé teszi a forrásszinten hordozható programok írását, viszont az összeszerkesztett, futtatható bináris kód azonban nem hordozható. A Java célkitűzései között szerepel a platformok közötti bináris hordozhatóság. A bájtkóddá lefordított programot átvihetjük más gépekre, ahol szabványos Java virtuális gép környezetet találunk. Javában minden függvény virtuális, extenzív futási idejű mechanizmusok léteznek az egyes osztályok, objektumok, metódusok azonosítására, akár létrehozására is. Javában a Statikus változók inicializálása is futási időben történik. A Java kevesebb dolgot bíz az implementációra, mint a C és a C++. Így például a kiértékelési sorrend, mellékhatások részletei pontosabban definiáltak a Java-ban, mint a C-ben és a C++-ban. Például a kifejezések esetében a részkifejezések kiértékelési sorrendje, az argumentumok kiértékelési sorrendje definiált a Java-ban, míg a C-ben és C++-ban nem. A Java fordító ellenőrzi, hogy használatuk előtt a lokális változók kapnak-e értéket. Alapvető különbségek adódnak a két nyelv között az objektummodellek különbözőségéből. A C++ nyelv az objektumokat, mint a memória egy összefüggő területén elhelyezkedő bájtsorozatot fogja fel, ami ismert memóriakiosztással rendelkezik, és amit ennek megfelelően manipulál a lefordított program. A C++-ban a memóriát alapvetően háromféle módon használjuk: statikus memória, automatikus memória és dinamikus memória. A Java nyelv virtuális gépben futnak: a memóriát közvetlenül nem tudjuk elérni, ha nem csak szimbolikusan, hivatkozásokon keresztül. A Java esetén nincs linker, amelyik címekké oldaná fel a hivatkozásokat. Egy osztály egy önálló class-fájllá fordul le. A class-fájl formátuma szabványos, platform-független. A Java nyelv megkülönböztet primitív típusokat és objektumokat; ez utóbbiakat hivatkozásokon keresztül érjük el. Primitív adattípusok és objektumhivatkozások lehetnek automatikus vagy statikus adatterületen, de maguk az objektumok nem. A c++ nyelvben az objektumok élettartama pontosan ismert. Az automatikus változók a deklaráló blokkba való belépéskor keletkeznek és pontosan a blokk végrehajtásának végén felszámolódnak. A statikus élettartamú objektumok a program végrehajtása végéig élnek. A szabad tárban allokált objektumok élettartamát mi határozzuk meg a new

és delete operátorokkal. Javában nincsen érvénytelen hivatkozás; amíg referenciánk van egy objektumra, addig az az objektum garantáltan létezik. A C++ programozási nyelv többparadigmás. Írhatunk procedurális programokat, egymást hívó függvényekkel, változókkal. Alkalmazhatjuk az objektum-orientált elveket; akár objektumokat létrehozva, akár bonyolult osztály-hierarchiák, öröklődés, virtuális függvények és egyéb technikák használatával. A Java nyelv csak az objektumorientált programozást támogatja. Nincsenek globális változók és függvények, csak osztályokhoz tartozó attribútumok, változók, konstansok és metódusok; még a main is egy osztály publikus metódusa. A main metódus paramétere: egyetlen String tömb értékű argumentumról van szó. Mivel a Java tömbök „tudják” a méretüket, szükségtelen a C++ argc paraméter átadása. Megjegyezzük, hogy a C-vel, C++-szal ellentétben a Java-ban a main első argumentuma nem tartalmazza a program nevét.

Tanulságok, tapasztalatok, magyarázat...

20.2. Python

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

A Python egy általános célú programozási nyelv, amely magas szintű, dinamikus, objektumorientált platform független. Ez a nyelv, tulajdonképpen egy szkriptnyelv, azonban rendkívül sok csomagot és beépített eljárást tartalmaz, így komolyabb problémák megoldására és összetett alkalmazások elkészítésére így használható. A C, C++ és a Javával szemben a Pythonnak nincs szüksége a fordítás fázisára, az értelmezőnek elegendő a Python-forrást megadni, és automatikusan futtatja is az adott alkalmazást. A Python interpreterek az alábbi népszerű platformokon érhető el: Windows, MacOS X, Unix, S60, UIQ, Windows CE, iPhone. Amennyiben egy prototípus-alkalmazást szeretnénk elkészíteni valamilyen algoritmus, alkalmazási lehetőség, felhasználói interfész vagy elmélet tesztelésére, akkor a Python az ideális választás. A Pythont egyszerű használni, megbízható és jelentős támogatást biztosít a hibák javítására. A Python nyelv tulajdonképpen egy magas szintű programozási nyelv, amely segítségével nagyobb problémák is leírhatóak. A Python nyelvhez szorosan kapcsolódó standard Python kódkönyvtár rengeteg újrahasznosítható, teljes mértékben Python nyelven megírt modult tartalmaz, amelyek így meggyorsítják az alkalmazásfejlesztést. A világhálón egyre több példakód található Python nyelven, illetve számos leírás is, amely segít a nyelv elsajátításában. A Python segítségével tömör és mégis könnyen olvasható programokat készíthetünk. A magas szintű adattípusok lehetővé teszik, hogy összetett kifejezéseket írjunk le egy rövid állításban. A kódcsopontosítás egyszerű tagolással (új sor, tabulátor) történik, nincs szükség nyitó- és záró jelekre. Nincs szükség változó vagy argumentumdefiniálására. A legfőbb jellemzője a nyelvnek, hogy behúzásalapú a szintaxisa. A programban szereplő állításokat az azonos szintű behúzásokkal tudjuk csoportokba szervezni. A nyelv sajátossága, hogy a sor végéig tart egy utasítás, nincs szükség a megszokott ';' használatára. Ha egy sor több sorban fér csak el, akkor ezt a sor végére írt '\n' jellel lehet jelezni. Az értelmező minden egyes sort úgynevezett tokenekre bont, amelyek közt tetszőleges üres karakter lehet. A token különböző fajtái a következők: azonosító, kulcsszó, operátor, delimiter, literál. Az azonosító lehet egy változó, osztály, függvénymodul vagy egyéb objektum neve. Betűvel vagy alulvonással kezdődik, és ezeken kívül még számokat is tartalmazhat. A kis- és nagybetűket megkülönböztetjük. Minden adatot objektumok reprezentálnak. Az adatokon végezhető műveleteket az objektum határozza meg. Az adattípusok a következők lehetnek: számok, sztringek, ennesek, listák, szótárak. A számok lehetnek egészek, lebegőpontosak és komplex számok. Az egész számok lehetnek decimálisak, oktálisak (ez esetben egy bevezető 0-t írunk elé) vagy hexadecimális számok 0x-szel bevezetve. Az egészek [-sys.maxint-1 ... sys.maxint] közötti értéket vehetnek fel. A lebegőpontos számok típusa leginkább a C nyelv double típusának felel meg, ábrázolása

$x.yEz$ vagy $x.yez$ formátumban van, ez az $x.y \cdot 10^z$ értékének felel meg. További típusok találhatóak a Python szekvenciái között. A szekvencia nemnegatív egész számokkal indexelt rendezett gyűjtőt jelent. Ilyen szekvencia a sztring, az ennes, illetve a lista, de lehetőség van saját szekvenciatípusok definiálására is. Sztringeket idézőjelek és aposztrófok közé írva is megadhatunk, illetve a 'u' betű használatával Unicode szöveget is felvehetünk. Az ennesek objektumok gyűjteményei vesszővel elválasztva. Az objektumok típusa lehet akár eltérő is. Az enneseket általában zárójelek közé írjuk. A lista akár különböző típusú elemek rendezett szekvenciája. A lista elemeit szögletes zárójelek közé zárjuk. A lista jellemzője, hogy dinamikusan nyújtózkodik, ha új elemet fűzünk hozzá. A szótár kulcsokkal azonosított elemek rendezetlen halmaza. A kulcs gyakorlatilag bármilyen egyszerű típus lehet (szám, sztring, stb.) Az elemek különböző típusok is lehetnek. A szótárban az elemek kapcsos zárójelek között, vesszővel elválasztva írandóak. Pythonban a változók alatt az egyes objektumokra mutató referenciákat értünk. A változók értékadása egyszerűen, az '=' jel segítségével történik. Változónak értékül adhatunk egy objektumot, egy típust, függvényt stb. A Python nyelvben egyaránt léteznek lokális és globális változók. A Python támogatja az if elágazást, valamint a for és while ciklusokat is. Pythonban függvényeket a def kulcsszóval definiálhatunk. A függvényekre mint értékekre is tekinthetünk, hiszen azok továbbadhatók más függvényeknek, illetve objektumkonstruktoroknak. A függvények rendelkeznek paraméterekkel, amelyeknek, a szokásos megkötésekkel és szintaxissal alapértelmezett értéket is adhatunk. A P. nyelv támogatja a klasszikus, objektumorientált fejlesztési eljárásokat. A nyelv a fejlesztés megkönnyítése érdekében sok szabványos modult tartalmaz. Ezekhez a modulokhoz adódnak hozzá a mobilkészülékeken való fejlesztés megkönnyítésére létrehozott modulok, például felhasználói felület kezelése, hálózatkezelés, kamera kezelése stb.

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

20.3. Általános

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

20.4. C

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

20.5. C++

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

20.6. Lisp

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

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPROG 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 érdeklődésükért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPROG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségben született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.