Univerzális programozás

Programozzunk együtt!

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

KÖZREMÜKÖDTEK

	CÍM : Univerzális programozás	

HOZZÁJÁRULÁS	*NÉV*	*DÁTUM*	*ALÁÍRÁS*
ÍRTA	Bátfai Norbert és Füleky Ladislav	2019. szeptember 14.	

VERZIÓTÖRTÉNET

VERZIÓ	DÁTUM	LEÍRÁS	NÉV
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	Aktualizálás, javítások.	nbatfai
0.0.5	2019-02-26	Turing csokor elkészítése és feltöltése GitHub-ra.	fulekylaszlo
0.0.6	2019-03-06	Chomsky csokor elkészítése és feltöltése GitHub-ra.	fulekylaszlo

VERZIÓTÖRTÉNET

VERZIÓ	DÁTUM	LEÍRÁS	NÉV
0.0.7	2019-03-13	Turing csokorban kissebb változások és a Caesar csokor elkészítése és feltöltése GitHub-ra.	fulekylaszlo
0.0.8	2019-03-20	Mandelbrot csokor befejezve valamint feltöltése GitHub-ra.	fulekylaszlo
0.0.9	2019-03-27	Welch csokor elkészítése és feltöltése GitHub-ra.	fulekylaszlo
0.0.10	2019-04-02	Conway csokor elkészítése és feltöltése GitHub-ra.	fulekylaszlo
0.0.11	2019-04-10	Schwarzenegger csokor elkészítése és feltöltése GitHub-ra.	fulekylaszlo
0.0.12	2019-04-11	Gutenberg csokor feltöltése GitHub-ra.	fulekylaszlo
0.0.13	2019-04-17	Chaitin csokor elkészítése és feltöltése GitHub-ra.	fulekylaszlo
0.1.0	2019-05-9	A könyv teljes elkészülte.	fulekylaszlo

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

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 (hogy lássuk mást is) példával.

Hogyan nyomjuk?

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

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

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

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

A DocBook XML 5.1 új neked?
Ez esetben forgasd a https://tdg.docbook.org/tdg/5.1/ könyvet, a végén találod az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

Bátf41 Haxor Stream
A feladatokkal kapcsolatos élő adásokat sugároz a https://www.twitch.tv/-nbatfai csatorna, melynek permanens archívuma a https://www.youtube.com/-c/nbatfai csatornán található.

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

Tutoriált
Ebben a feladatban tutoriált Molnár Antal Albert.

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

Megoldás forrása:

- Egy mag 0%-on
- Egy mag 100%-on
- Összes mag 100%-on

Itt található egy végtelen ciklus mely 1 magot használ ki 0%-ban.

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

int main ()
 {
 for ( ; ; )
  {
  sleep (1);
  printf("Végtelen ciklus.\n");
  }
 return 0;
 }
```

Feljebb látható egy `for ( ;;)` végtelen ciklus melyben be van ágyazva egy `sleep(1)` ami annyit takar, hogy a következő 1 milliszekundumban nem szeretnénk kihasználni az adott szálat semmilyen mértékben. Így kapunk 1 szálon 0%-ot.

Itt található egy végtelen ciklus mely 1 magot használ ki 100%-ban.

```
#include <stdio.h>

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

Hasonló, mint az előző példa azzal az ellentéttel, hogy itt nem 0%-on szeretnénk kihasználni a szálunkat, ezért a `sleep(1)`-t elhagyjuk, és hagyjuk futni a végtelen ciklust.

Itt található egy végtelen ciklus mely az összes magot használja ki 100%-ban.

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

int main ()
 {
 #pragma omp parallel for
  {
   for ( ; ; )
    {

    }
  }
 return 0;
 }
```

Ebben az esetben amikor minden magot szeretnénk 100%-ban kihasználni, segítségül vesszük az `omp parallel` utasítást mely a folyamatokat szétosztja a szálak között és végtelen ciklust használva, a szétosztott szálakon (mindegyiken) 100%-ot kapunk. Ügyeljünk arra, hogy az `omp.h` könyvtárat ne felejtsük el beágyazni.

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

```
Program T100
{

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

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{

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

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

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

}
```

Mit fog 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.

Amikor azt kell eldöntenünk egy programól, hogy le fog-e fagyni, azt manuálisan csak rövidebb forráskódokról tudjuk megállapítani, hiszen pár sort meg tudunk nézni és logikusan el tudjuk dönteni, hogy működőképes-e a programunk, de több ezer sornál ez szinte lehetetlen. Bővebben ezzel a problémával Alan Turing nevezetű brit matematikus foglalkozott.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása: Változók cseréje logikai utasítás vagy kifejezés használata nélkül

```
#include <stdio.h>

int main()
{

int a = 7;
int b = 5;

printf("a=%d b=%d\n", a, b);

b = b - a;
a = a + b;
b = a - b;

printf("a=%d b=%d\n", a, b);
}
```

A következő feladatban 2 változó értékének felcserélését kell megoldanunk, ami nem egy nehéz feladat. A szokásos módon kezdjük, beágyazzuk a szükséges könyvtárakat majd a ”main”-ben definiálunk két változót, kiiratjuk, hogy alap esetben milyen értékkel bírnak, majd a ”kivonás és összeadás” művelettel felcseréljük a kettőt, mivel minden logikai utasítás és kifejezés nélkül kell megoldanunk a feladatot. Majd pedig újra kiiratjuk a két változó értékét, csak azért, hogy lássuk tényleg sikerült-e felcserélnünk a változók értékeit.

2.4. Labdapattogás

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

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

Megoldás forrása:

- Labdapattogtatás ”IF” fügvénnyel
- Labdapattogtatás ”IF” fügvény nélkül

Ebben a két feladatban azt fontos észrevennünk, hogy a lényeg azon van, hogy a labda kiszámított matematikai koordinátákon mozog, amelyeket mi határozunk meg, valamint fontos észbentartatunk, hogy a labda ne pattogjon ki a ”konzol”-on kívülre. Továbbá nem szabad elfelejtenünk, hogy mindkét esetben használtuk a `curses.h` könyvtárat, így futtatnunk is másképpen kellesz, a megszokott `gcc -o kívánt_név program_neve.c` helyett használjuk a `gcc -o kívánt_név program_neve.c -lncurses`

A forrás megírása, megint egyszerű lesz, ugyan úgy ahogyan eddig, most is beágyazzuk a szükséges könyvtárakat majd a ”main”-ben deklaráljuk a változáinkat és az első esetben egy ”for” és ”if” ciklus segítségével meg is tudjuk írni a programunkat. Ne felejtsük el beállítani az ablak méretét valamint a koordináták értékeit.

A második esetben (IF nélkül) pedig a logika szabályait követve elhagyjuk az ”if” függvényt és az `mvprintw` függvényben határozzuk meg a koordináták értékeinek a kiszámítását.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás forrása: Szóhossz ”while” ciklussal

```
#include <stdio.h>

int main()
 {
 int a=1;
 int i=1;

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

 printf("A szóhossz:%d\n",i);
 return 0;
 }
```

A következő feladatban a szokásos módon kezdünk...a könyvtárakkal, ezután pedig a "main"-ben megadjuk a változóinknak az értékeit és létrehozunk egy `while` ciklust ami egészen addig fog futni amíg az "a>0" és a `while` ciklusban pedig az "a" változónkat addig "shifteljük" jobbról a bal oldalra egyesével amíg el nem éri a feljebb említett nullát, így egymás után beolvasva a karaktereket. Majd növeljük az "i" változónkat amit a "karakterek számának" hoztunk létre, és a végén pedig már csak kiiratjuk.

2.6. Helló, Google!

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

Megoldás forrása: Pagerank(4 honlap)

A következő programunk már egy picivel összetetteb lesz. A PageRank-ról van szó, ezt az algoritmust amelyet a Google is használ Sergey Brin és Larry Page page fejlesztettek ki 1998-ban. Az algoritmus pedig abból a gondolatmenetből született, hogy annál jobb minőségű lesz egy weblap minél jobb minőségű weblap/weblapok mutatnak rá.

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

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

Ebben az esetben fontos tudnunk, hogy mik a prímek és az ikerprímek. A prímek azok olyan természetes számok, melyeknek pontosan 2 osztójuk van(1 és önmaguk), az ikerpímek pedig olyan prímpárok melyek különbsége 2. A Brun-tétel pedig azt mondja ki, hogy az ikerprímek reciprokának összege egy véges értékhez konvergál (megközelítőleg 1,9-hez), később 1919-ben ez bizonyítva lett Viggo Brun által.

```
library(matlab)

                stp <- function(x){

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

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

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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

A Monty Hall-ként megfogalmazódó probléma először egy televíziós sorozatban alakult ki. A lényege az volt, hogy van 3 ajtónk, ezek közül kettő mögött semmi sem található, a harmadik mögött pedig egy nyeremény van. A szimulációban pedig azt vizsgáljuk, hogy ha kiválasztunk egy ajtót, akkor érdemes a végéig annál az ajtónál maradni, vagy mindig változtassunk a kiválasztott ajtónkon.

A szimulációt futtatva, azt kapjuk eredményül, hogy megéri változtatni a kiválasztott ajtóinkon, mert rendkívülien nőnek az esélyeink a nyerésre.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Állapotmenetes kép:Állapotmenet gráf

Megoldás forrása:Decimálisból unárisba átváltó Turing-gép

Ugyebár többféle számrendszert ismerünk, most pedig megbarátkozunk az unáris számrendszerrel. Nagy egyszerű és könnyen érthető. Annyi a lényege, hogy a számokat nem számmal írjuk le, hanem vonalakkal ábrázoljuk, és természetesen 1 vonal (|) az az 1-el egyenlő, így például ha össze akarjuk adni a következő számokat pl. (1 + 1) akkor azt így fogjuk megtenni: (||) és ez lesz = 2-vel. Az általunk ismert Turing gép, pedig pont így végzi el az átváltásokat.

Decimálisból unárisba való átváltás:Átváltás

```
./a.out
||||| ||||| ||||| |||||
```

A programunkban láthatjuk, hogy az ”a” változónknak, a 20-as értéket adtuk meg, így 20 vonalat is rajzolt ki a programunk. Ha azt a változót kedvünk szerint változtatjuk, akkor annyi vonalat fog kirajzolni a Turing gépünk.

3.2. Az $a^nb^nc^n$ nyelv nem környezetfüggetlen

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

Megoldás forrása:Környezetfüggő generatív grammatika

Noam Chomsky 1928-ban született és ő volt a generatív nyelvtan megalkotója, valamint filozófus is volt. A fenti 2 példa bizonyíték arra, hogy az $a^nb^nc^n$ nyelv nem környezetfüggetlen, hiszen sikerült megadnunk ezt a 2 nyelvet.

```
1.Példa:
Megadatt változók= S, X, Y.
Konstansok: a, b, c.
Szabályok melyeket alkalmazunk: S->abc, S->aXScc, X->aYa, Ya-> ↩
   aX, Xa->aabb.
Így azt kapjuk, hogy: S->aXScc->aXabccc->aaabbbccc.
```

```
2.Példa:
Megadott változók= S, X, Y.
Konstansok: a, b, c.
Szabályok melyeket alkalmazunk: S->abc, S->aXaYS, Yab->bcc, Xa ↩
   ->aabb.
Így azt kapjuk, hogy: S->aXaYS->aXaYabc->aXabccc->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.

Megoldás forrása:C89-C99 hiba

```
#include <stdio.h>

int main ()
 {
 // Emiatt a komment miatt nem fog működni.
 printf (”Hello World\n”);
 return 0;
}
```

Ebben a példában azt vehetjük észre, hogy az újkori C99-es szabvány támogatja az egysoros kommenteket viszon, ha a C89-es szabvánnyal fordítjuk akkor nem fog működni, mert nem támogatja azt. Ha különböző szabványokkal szeretnénk futtatni a programunkat, azt úgy tudjuk elérni, hogy a következők alapján fordítjuk.

Ha C89-el akarjuk fordítani:`c89 forras_neve.c -o kivant_nev`.

Ha pedig C99-el: `c99 forras_neve.c -o kivant_nev`.

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!

Megoldás forrása: Lexikális elemző

Először fontos észben tartanunk, hogy szükséges a ”LEX” telepítése a számítógépünkre, és most már a futtatás is picit másképpen fog kinézni, először `lex programunk_neve.c` majd ezután ha nem kapunk hibát, akkor kapunk egy olyan nevezetű fájlt, hogy ”lex.yy.c” és ezt pedig már csak fordítanunk kell mégpedig úgy hogy `gcc -lfl lex.yy.c`, fontos, hogy ne hagyjuk ki a `-lfl` összekötőt.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás forrása:l33t cipher

Mivel, hogy ”gcc”-vel szeretnénk fordítani a forráskódunkat, ezért először ki kell adnunk egy `lex forrasunk_neve.c` parancsot. A forráskódra nézve a szokásos dolgokat láthatjuk, beágyazzuk a függvénykönyvtárakat, majd egy random számgenerátort hozunk létre.

A lényeg az úgynevezett ”cipher típusú tömb” létrehozása ami, ahogyan a forráskódban is láthatjuk, különböző betűkhöz és számokhoz tartozó lehetséges ”leet” kódokat tartalmazza. Következő lépésben a program megnézi a beolvasott karaktereket és, hogy illeszkednek-e rá a ”cipher típusú tömb” karakterei.

3.6. A források olvasása

Tutoriált
Ebben a feladatban tutoriált Molnár Antal Albert.

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

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

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

Forráskód és magyarázat:Jelkezelő

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

i.
```
if(signal(SIGINT, SIG_IGN)!=SIG_IGN)
    signal(SIGINT, jelkezelo);
```

Forrás és magyarázat: Kódcsipet(i)

Ha a `SIGINT` nincs ignorálva akkor a `jelkezelo` végzi a jelkezelést.

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

Forrás és magyarázat: Kódcsipet(ii)

Itt ötször elvégezzük, azt amit megadunk a programnak, a ++i (prefix increment).

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

Forrás és magyarázat: Kódcsipet(iii)

Itt is ötször elvégezzük, azt amit megadunk a programnak, a i++ (postfix increment).

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

Forrás és magyarázat: Kódcsipet(iv)

Itt nem lehet pontosan megmondani, hogy mi is fog történni.

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

Forrás és magyarázat:Kódcsipet(v)

Minden lefuttatáskor más eredményt fogunk kapni, a kód miatt.

vi.
```
printf(”%d %d”, f(a, ++a), f(++a, a));
```

Magyarázat: Itt a `printf` függvény argumentuma miatt nem lehetünk abban biztos, hogy milyen sorrendben fog végrehajtódni a kiértékelés.

vii.
```
printf(”%d %d”, f(a), a);
```

Magyarázat: Egyszerűen kiiratjuk az ”f” függvény ”a” kimenetét, és még az ”a”-t is.

viii.
```
printf(”%d %d”, f(&a), a);
```

Magyarázat: A hiba megint a `printf` miatt van, nem lesz megint egyértelmű a kiértékelési sorrend.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

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

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

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

1: Azt írja le, hogy minden ”x” esetén létezik legalább egy olyan ”y” ami nagyobb, mint ”x” és az ”y” prím. Ebből azt vehetjük észre, hogy végtelen sok prímszám van.

2: Azt írja le, hogy minden ”x” esetén létezik legalább egy olyan ”y” ami nagyobb, mint az ”x” és az ”y” prímszám valamint ikerprím(A könyv ezelőtti részeiben megbeszéltük mit jelent az ikerprím fogalma) is, ebből adódóan végtelen sok ikerprím van.

3: Ez az eset azt írja le, hogy létezik legalább 1 olyan ”y” ami minden ”x” esetén nagyobb ha az ”x” az prímszám.Itt láthatunk egy új fogalmat a `supset` ez annyit jelent, mint a matematikában az `implikáció`, tehát ebből következik, hogy véges sok prímszámunk van.

4: Ebben az esetben azt láthatjuk, hogy ugyanúgy létezik legalább egy olyan ”y” amely minden ”x” esetén igaz, ha ”y” kisebb, mint ”x” akkor az ”x” nem prím. Láthatunk egy következő fogalmat ami a `neg` ami válójában a `negációnak` tehát tagadásnak felel meg, így, tagadom, hogy az ”x” az prím, ezáltal azt kapjuk, hogy véges sok prímszámunk van.

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

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

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

Mit vezetnek be a programba a következő nevek?

Megoldás forrása:Deklaráció

- ```
  int a;
  ```

  Egy egészet vezet be.
- ```
  int *b = &a;
  ```

  Egy egészre mutató mutatót vezet be.
- ```
  int &r = a;
  ```

  Egy egésznek a referenciáját vezeti be.
- ```
  int c[5];
  ```

  Ez egy egésznek a tömbjét vezeti be.
- ```
  int (&tr)[5] = c;
  ```

  Ez pedig egy egészek tömbjének a referenciáját.
- ```
  int *d[5];
  ```

  Egy egésznek a tömbjére mutató mutatót vezet be.
- ```
  int *h ();
  ```

  Ez egy egészre mutató mutatót visszadó függvényt vezet be.
- ```
  int *(*l) ();
  ```

  Ez egy egésznek a muatatójára mutató mutatót visszadó függvényt vezet be.
- ```
  int (*v (int c)) (int a, int b)
  ```

  Ez két egészet kapva, majd egy egészre mutató mutatót visszaadó függvényt vezet be.
  ```

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

  Két egészet kapó és egy egészet visszadó függvényre való mutató mutatót vezet be.
  ```

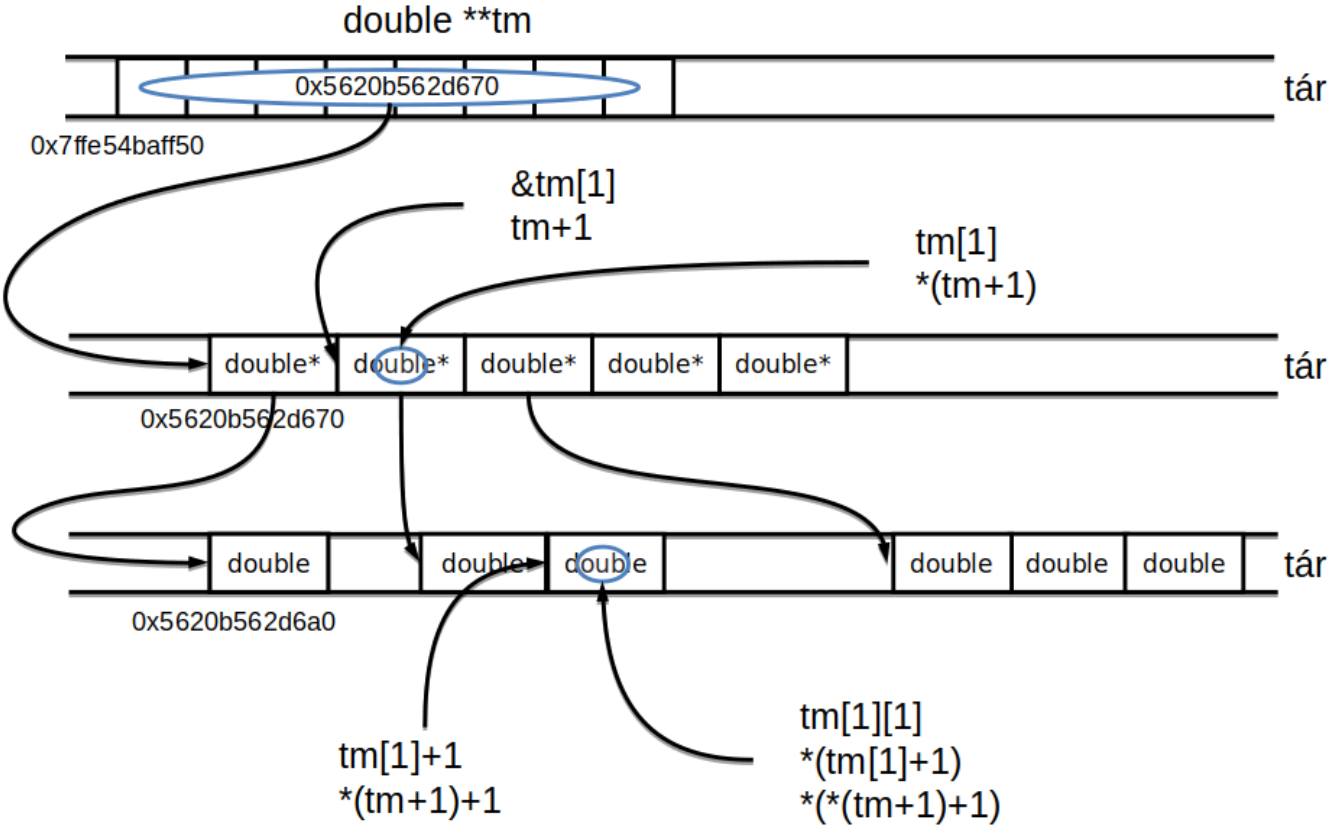
4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

Megoldás forrása:double ** háromszögmátrix

Ebben a példában egy ”double ** háromszögmátrixot” fogunk létrehozni. Dinamikus tömbök segítségével, először megadjuk milyen(a ”milyen tömböt” úgy kell érteni, hogy hányszor hányas tömbre gondolunk) tömböt szeretnék létrehozni és tároljuk is azt, utána pedig lefoglaljuk a háromszögmátrixot, majd pedig kiíratjuk a memória címét. Ezek után ismét lefoglalunk tömböket, a program pedig elkészíti a háromszögmátrixunkat.



4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás forrása:Exor titkosító

Itt egy titkosító programot fogunk létrehozni, és egy ”x” hosszúságú kulcs segítségével titkosíthatjuk majd a szövegünket. Különböző függvényeket/ ”function”-ket használunk mint például a `MAX_TITKOS` vagy a `OLVASAS_BUFFER` illetve `KULCS_MERET`. Ne felejtsük el, hogy ezeket a forráskódunkban először ”definiálni” kell, amit így tehetünk meg pl.: `#define MAX_TITKOS 4096` .

Ezt követően a szokásos módon beágyazzuk az általunk kívánt függvénykönyvtárakat. Majd beállítjuk a kölünböző specifikációkat, pl. a kulcs méretét, magát a titkosítót, stb.

4.3. Java EXOR titkosító

Tutoriált
Ebben a feladatban tutoriált Molnár Antal Albert.

Írj egy EXOR titkosítót Java-ban!

Megoldás forrása:Exor titkosító Java-ban

Egy EXOR titkosító, olyan alapon mint az előző volt, csak ez most Java környezetben. Viszont amire a Java-ban figyelni kell az az, hogy ez egy objektumorientált nyelv, valamint nincs benne semmi fajta memóriakezelés, mint ahogyan a C-ben megszokhattuk.

4.4. C EXOR törő

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

Megoldás forrása:Exor törő

Az előző feladatokban megírt titkos szövegünket kellene feltörnie ennek a forráskódnak ami úgy működik, hogy megadjuk a forráskódban pl.az átlagos szóhosszt(5-9 betű közé esik), majd megadjuk a leggyakoribb magyar szavakat(az,nem,ha,igen...) és ezeket a forráskód megvizsgálja és rögzíti.

A lényege ennek a forráskódnak, hogy a ”main”-ben az input fájlból kiolvastattjuk a titkos szövegünket és és ezt pedig a ”buffer”-ben tároljuk. Majd ezeket összeexorozzuk és kiiratjuk a kulcsot és a kapott szövegünket is.

4.5. Neurális OR, AND és EXOR kapu

R

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

Megoldás forrása: Neurális OR, AND és EXOR kapu

Ebben a feladatban felépítünk egy neurális hálót, amit úgy tudunk megtenni, hogy megadjuk, hogy egy adott bemenetre milyen kimenetet adjon a program és ezt pedig mesterségesen megpróbálja utánozni.

Majd pedig láthatjuk, hogy a különböző példákhoz amit meg szeretnénk tanítani a programnak, hány lépés volt szükséges és ezt megtanulnia mennyi időbe tartott. Valamit észrevehetünk egy olyan funkciót is, hogy ”Error” az a hibaszámot jelenti, logikusan, minél kisebb ez a szám (legjobb eset a 0) annál jobb eredményt kapunk.

4.6. Hiba-visszaterjesztéses perceptron

C++

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

Megoldás forrása:Hiba-visszaterjesztéses perceptron

A perceptron a mesterséges intelligenciában a neuron egyik legelterjedtebb változata. Tekinthetjük őt egy ”arcfelismerő” gépnek, mivel egy idő után(véges számú kísérlet után) megtanulja csoportosítani és osztályozni a 0 és 1-ekből álló bemeneti mintákat.

A program elég könnyen olvasható és érhető is, a lényege az az, hogy a program a bemenetként megadott képnek végigmegy az összes pixelén és megvizsgálja a színkomponenseit. Majd pedig láthatjuk, hogy az adott szín amit vizsgáltunk, az hány százalékát rakja ki az egész képnek.

5. fejezet

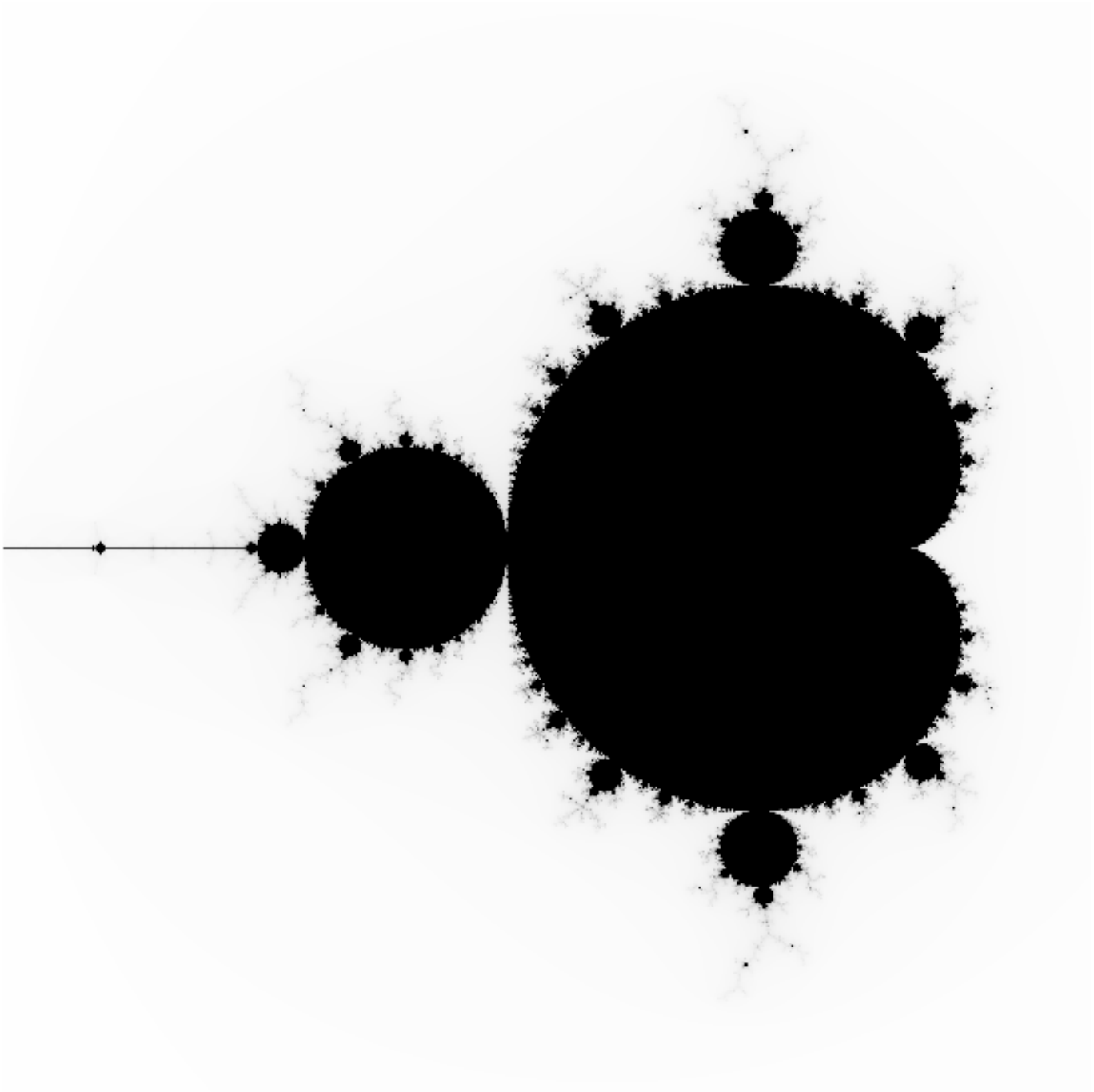
Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás forrása:Mandelbrot halmaz

Ez a halmaz Benoît Mandelbrot-ról lett elnevezve, röviden a Mandelbrot-halmaz definíciója az az, hogy a Mandelbrot-halmaznak csak azok a C komplex számok az elemei amelyekben ez a sorozat nullához tart: Z0:=C Zi+1:=Zi*Zi+C. Minél magasabb az iterációk száma annál nagyobb lesz a részletgazdagság. Akinek van egy kevés ideje, az mindenképpen tegye meg, és nagyítson bele az így kapott képbe, nagyon sok időt el lehet ezzel ”pazarolni”. Látni fogjuk mire gondolok a következő feladatokban, amikor is egy nagyítót fogunk majd létrehozni.

Mandelbrot halmaz a komplex síkon:



A programunk működése nagyon egyszerű mivel megvan a fentebb említett egyenletünk, erre megadunk egy halmazt és a program végigmegy az általunk megadott halmazon és az erre illeszkedő pixeleket pedig egy színnel kiszinezi.

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

Megoldás forrása:Mandelbrot halmaz std::complex osztállyal

Ez a forráskód tulajdonképpen ugyan azt csinálja mint az előző, azzal a különbséggel, hogy az előző példában a ”header”-ben definiálnunk kellett a függvényeket, itt

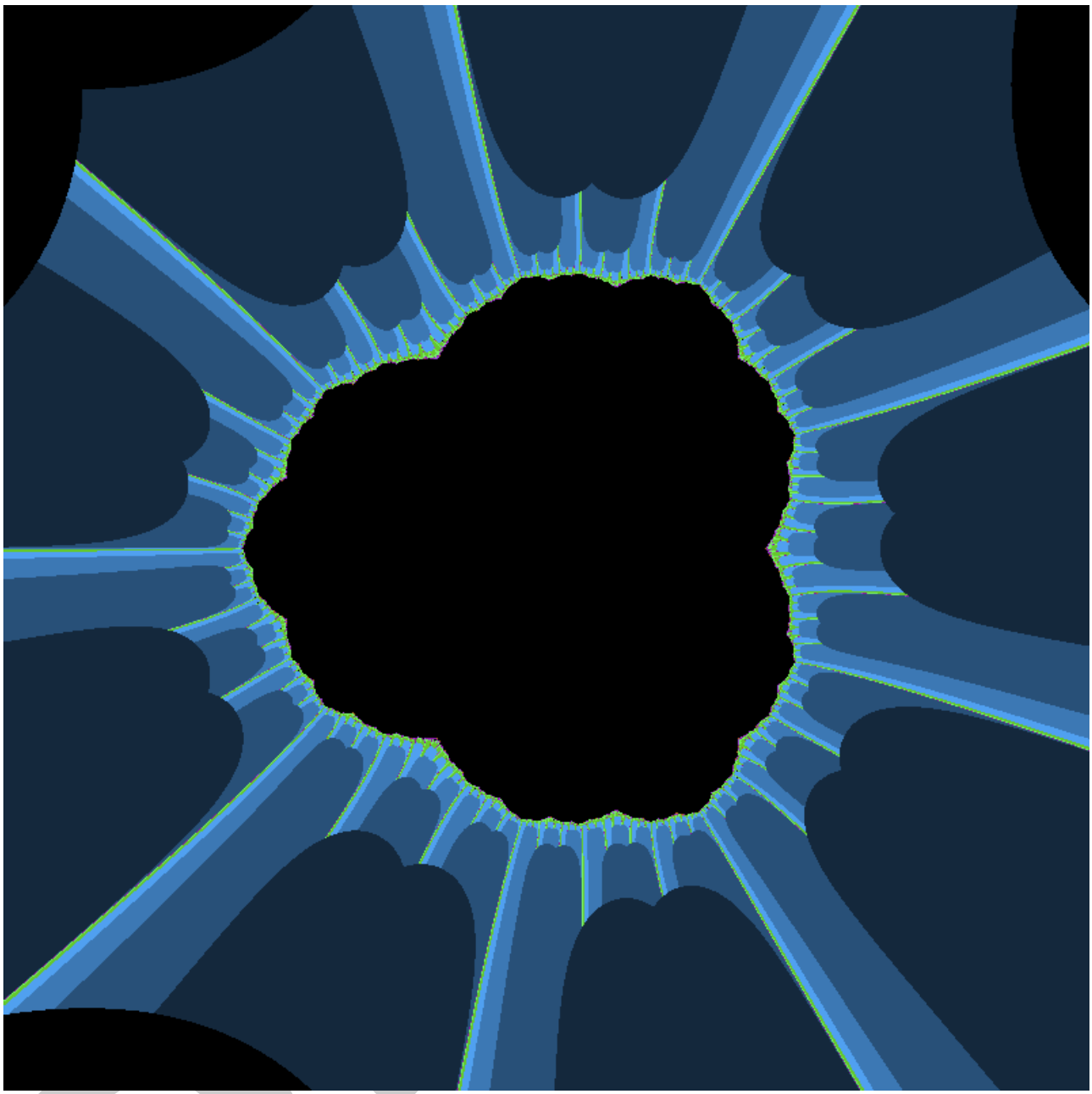
pedig egy egyszerű `#include complex` parancsorral, meg is tudjuk tenni ugyan azt. Ugyan úgy mint az előző feladatban, a Mandelbrot-halmazt egy komplex számsíkon ha ábrázoljuk akkor egy nevezetes fraktálalakzatot kapunk.

5.3. Biomorfok

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

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

A Biomorfok nagyon szoros kapcsolatban állnak a Mandelbrot-halmazzal. A forráskódok lefuttatása után észrevehető, hogy míg a Mandelbrot-halmazban fekete-fehér képet kapunk, addig itt szineset, amit a forráskódban a RGB színkódokkal tudunk elérni. Egyből az első képen észrevehető, hogy nem teljesen olyan mint az előző porgramunk, mivel a most kapott ”.png” képünk az szines, ez pedig azért lehetséges mivel teljesen más egyenlettel és képlettel dolgozunk. Valamint a pixeleket RGB színkód segítségével szinezzük.



5.4. A Mandelbrot halmaz CUDA megvalósítása

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

Megoldás forrása: CUDA

A CUDA az NVidia kártyáknak a programozói interfésze, amit valójában párhuzamos számításokhoz lehet felhasználni. Ha NVidia kártyával rendelkezünk, akkor nagyon sok terhet le tudunk venni a CPU válláról.

```
// mandelpngc_60x60_100.cu
// Copyright (C) 2019
```

```
// Norbert Bátfai, batfai.norbert@inf.unideb.hu
//
//  This program is free software: you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation, either version 3 of the License, or
//  (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program.  If not, see <https://www.gnu.org/licenses/>.
//
// Version history
//
//  Mandelbrot png
//  Programozó Páternoszter/PARP
//   https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux
//
//  https://youtu.be/gvaqijHlRUs
//

#include <png++/image.hpp>
#include <png++/rgb_pixel.hpp>

#include <sys/times.h>
#include <iostream>


#define MERET 600
#define ITER_HAT 32000

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

// számítás adatai
```

```
                float a = -2.0, b = .7, c = -1.35, d = 1.35;
                int szelesseg = MERET, magassag = MERET, iteraciosHatar =  ↩
                   ITER_HAT;

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

                // c = (reC, imC) a rács csomópontjainak
                // megfelelő komplex szám
                reC = a + k * dx;
                imC = d - j * dy;
                // z_0 = 0 = (reZ, imZ)
                reZ = 0.0;
                imZ = 0.0;
                iteracio = 0;
                // z_{n+1} = z_n * z_n + c iterációk
                // számítása, amíg |z_n| < 2 vagy még
                // nem értük el a 255 iterációt, ha
                // viszont elértük, akkor úgy vesszük,
                // hogy a kiinduláci c komplex számra
                // az iteráció konvergens, azaz a c a
                // Mandelbrot halmaz eleme
                while (reZ * reZ + imZ * imZ < 4 && iteracio <  ↩
                   iteraciosHatar)
                {
                // z_{n+1} = z_n * z_n + c
                ujreZ = reZ * reZ - imZ * imZ + reC;
                ujimZ = 2 * reZ * imZ + imC;
                reZ = ujreZ;
                imZ = ujimZ;

                ++iteracio;

                }
                return iteracio;
                }


                /*
                __global__ void
                mandelkernel (int *kepadat)
                {

                int j = blockIdx.x;
                int k = blockIdx.y;
```

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

                }
                */

                __global__ void
                mandelkernel (int *kepadat)
                {

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

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

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

                }

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

                int *device_kepadat;
                cudaMalloc ((void **) &device_kepadat, MERET * MERET *  ↩
                   sizeof (int));

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

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

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

                }

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

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

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

int kepadat[MERET][MERET];

cudamandel (kepadat);

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

for (int j = 0; j < MERET; ++j)
{
//sor = j;
for (int k = 0; k < MERET; ++k)
{
kep.set_pixel (k, j,
png::rgb_pixel (255 -
(255 * kepadat[j][k]) / ITER_HAT,
255 -
(255 * kepadat[j][k]) / ITER_HAT,
255 -
(255 * kepadat[j][k]) / ITER_HAT));
}
}
kep.write (argv[1]);

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

times (&tmsbuf2);
std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
+ tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;

delta = clock () - delta;
std::cout << (float) delta / CLOCKS_PER_SEC << ” sec” <<  ↩
   std::endl;

}
```

Az alábbi forráskódot megtekintve észrevehetjük, hogy most egy CUDA féle API-n belül dolgozunk. Észrevehetjük, hogy itt azért jelennek meg új dolgok mint pl. __global__ stb. Ezzel például a Kernel rendelkezik. Az ilyen ”minősítők” ugyan olyan minősítők mint példáual amikor egy változót úgy adunk meg, hogy ”unsigned short int a”.

Ahhoz, hogy egy ilyen forráskódot fordítani és futtatni tudjunk, szükségünk lesz ér-

telemszerűen egy CUDA-t támogató kártyára és magára a CUDA-ra aminek telepítve kell lennie a számítógépünkön. Viszont amikor ezekkel az előfeltételekkel fordítjuk és futtatjuk a programunkat akkor megkapjuk a Mandelbrot halmaz képet, ahogyan ezt az előző feladatokban már megszokhattuk.

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 iteréció bejárta z_n komplex számokat!

Megoldás forrása:Mandelbrot nagyító és utazó C++ nyelven

Fontos megjegyeznünk, hogy a szükséges csomagot, ha még nem telepítettük, akkor tegyük meg, majd csak azután kezdhetünk neki a forráskódunk megírásához. Ahogy a Java kódban is láthatjuk majd nem szabad elfelejtünk az ablakunk beállítását. Továbbá létrehozunk egy Mandelbrot-halmazt, ahogyan azt az előző forráskódokban is megszokhattuk, ezután pedig már csak pár finomhangolásra van szükség :)

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása:Mandelbrot nagyító és utazó Java nyelven

Picit más megközelítést fogunk látni, amikor a ”.java” forráskódot olvassuk. Amint egyőbl láthatjuk is, importáljuk a szükséges csomagokat, majd pedig létrehozunk egy ablakot amiben dolgozni fogunk, a további sorokban pedig annak testreszabását visszük véghez, mint például a keret, méret, stb.. Majd pedig a lényeges dolog amit láthatunk a forráskódban az a `switch` függvény lesz, mely keretein megadjuk, hogy egyes billentyűkhöz köthetően mi történjen.

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás forrása:Első osztály

A forráskódunkban először készítünk egy osztályt, mely a random számok generálására fog szolgálni. Ennek lesz egy private és egy public része is.

Lesz egy konstruktorunk, ami valójában a `polargen()`, viszont ezt csak egyszer fogjuk visszahívni. A másik ilyen függvényünk a destruktor lesz amit akkor hívunk meg amikor a tárterületen akarunk egy picit felszabadítani.

A mainben fogjuk deklarálni a ”polargen rnd” váltózót és generálunk 10 random számot a `for` ciklus segítségével.

Javában annyi lesz a változás, hogy rövidebb, egyszerűbb a forráskódunk, és valójában egy nagy osztály az egész. Itt is lesz egy konstruktorunk, és a függvények ugyan azt hajtják végre mint a ”c++”-os változatban, viszont, itt sokkal egyszerűbben is meg tudjuk írni a 10 random szám generálását.

6.2. LZW

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

Megoldás forrása:LZW Binfa

Az LZW algoritmus-t különböző GNU eszközök használják, mint például a ”compress” vagy a ”gunzip”, mivel az LZW algoritmus az egy tömörítő algoritmus.

A működésének a lényege az az, hogy a bemeneti nullákból és egyesekből épít fel egy bináris fát, majd ezt követően ellenőrzéseken megy végig és megvizsgálja, hogy a

szülőnek van-e nullás/egyes gyermeke, ha nincs akkor létrehoz egye és ezután pedig visszatér a gyökérhez. Abban az esetben amikor talál a szülőnek egy nullás/egyes gyermeket akkor ”rálép” és egészen addig követi ezt a ciklust amíg nem talál egy olyan részfát ahol nincs gyermek és újból vissztérhetne a gyökérhez.

A futtatás a következőképpen fog történni:

```
./binfa bemeneti_fájl -o kimeneti_fájl
```

6.3. Fabejárás

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

Megoldás forrása:Fabejárás

A ”preorder” és a ”postorder” fabejárás között csak annyi a különbség, hogy míg a ”preorder”-ben feldolgozzuk az elemet és bejárjuk preorder a bal oldali részfát majd preorder a jobb oldali részfát, a másodikban pedig a feldolgozzuk az elemet, bejárjuk postorder a jobb oldali részfát, majd ezután pedig postorder a bal oldali részfát. Ezek után már csak futtatnunk kell a programotkódunkat amit majd úgy tudunk megtenni, hogy `.binfa befile -o/r kifile` itt viszont fontos megjegyezni, hogy az `-o/r` közül választanunk kell mégpedig aszerint, hogy mit szerenténk. Az o jelzi a ”postorder”-t és az r pedig a ”preorder”-t.

6.4. Tag a gyökér

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

Megoldás forrása:Tag a gyökér

Az LZW Binfa forráskódja alapból úgy van, hogy a gyökér együttesen van a fával.

6.5. Mutató a gyökér

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

Megoldás forrása: Mutató a gyökér

Itt annyit kell tennünk, hogy módosítjuk a forráskódunkat, mégpedig úgy, hogy a bináris fa gyökere mutató legyen. Ezt úgy tudjuk megtenni, hogy a gyökér csomópontjának a definíciójánál, helyet foglalunk a gyökérnek, ezt követően pedig az így kapott mutatót egyszerűen behelyettesítjük azokra a helyekre, ahol a gyökér változóra hivatkoztunk előzőleg.

6.6. Mozgató szemantika

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

Megoldás forrása:Mozgató szemantika

Mielőtt elkezdenénk bármit is tenni, meg kell jegyeznünk, hogy mit is értünk mozgató szemantika alatt. 3 különböző dolgot értünk alatta:

- Másoló konstruktor

- Mozgató konstruktor

- Destruktor

Megjegyezném még, hogy gondolhatjuk, hogy elég lesz egy `std::move`, de sajnos ez nem igaz, ugyanis a bemenetekből jobbérték referenciát csinál sé utána meghívja annak a mozgató konstruktorát. Tehát konkrétan itt nem válik hasznunkra semmit.

Alap esetben az LZW Binfában beágyazott ”csomópont” osztályú objektumok vannak, amik alkotják a fát. Így a fát úgy tudjuk majd másolni, hogy ezeket a csomópontok majd rekurzívan másoljuk. A rekurzív másoló függvénynek át kell adnunk a gyökeret és a fát is. A mozgatókonstruktor elkészítése annyit takar, hogy az ”=” operátort túlterheljük.

```
Csomopont *masol(Csomopont * elem, Csomopont * regifa) {
Csomopont *ujelem = NULL;
if (elem != NULL) {
switch (elem->getBetu()) {
case ’/’:
ujelem = new Csomopont(’/’);
break;
case ’0’:
ujelem = new Csomopont(’1’);
break;
case ’1’:
ujelem = new Csomopont(’0’);
break;
default:
std::cerr << ”HIBA!” << std::endl;
break;
}
ujelem->ujEgyesGyermek(masol
(elem->egyesGyermek(), regifa));
ujelem->ujNullasGyermek(masol
(elem->nullasGyermek(),
regifa));
if (regifa == elem)
fa = ujelem;

}
```

```
            return ujelem;
            }

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

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

A fenti kódcsipetben láthatjuk, hogy ez egy másoló függvény, melyet a másoló konstruktorból tudtunk meghívni, és amint említettem korábban, át is adtuk neki a fát és a gyökeret is.

7. fejezet

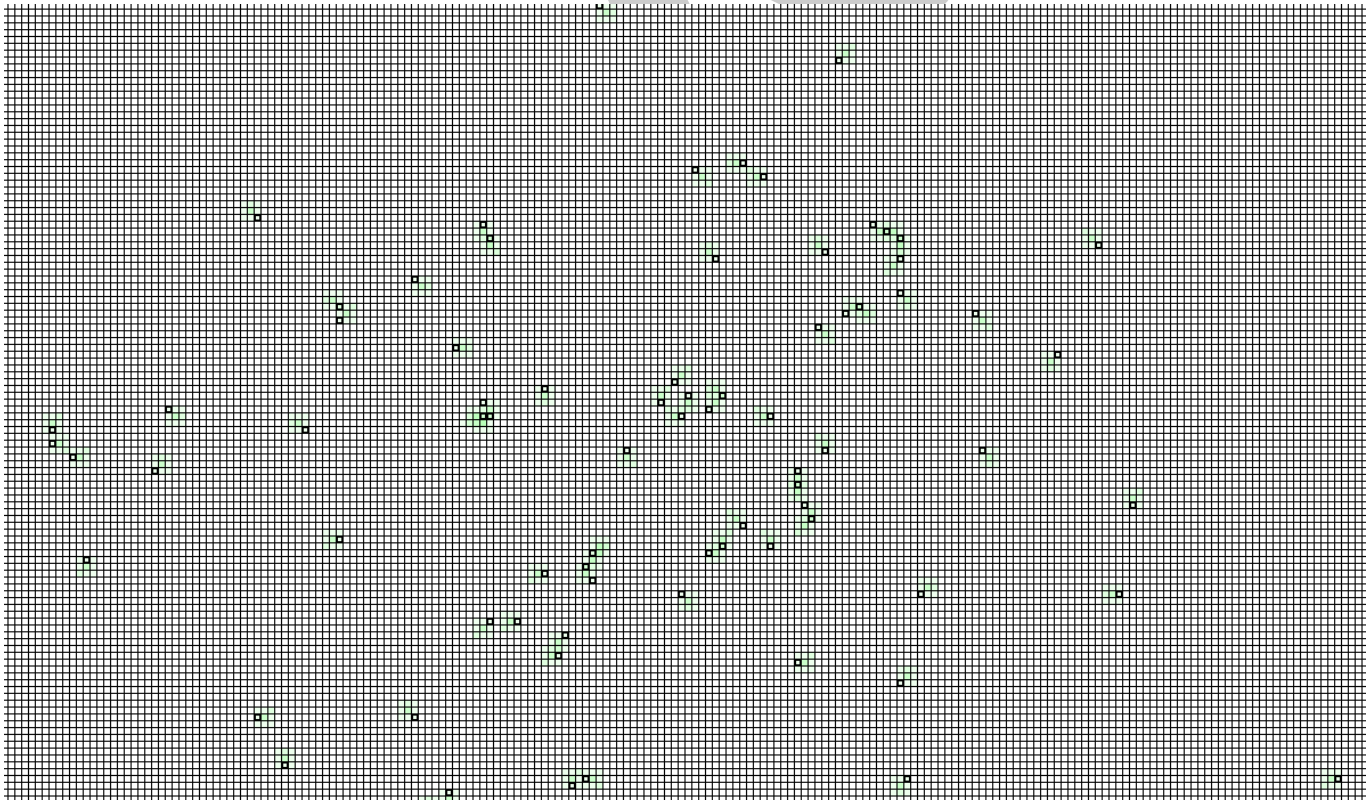
Helló, Conway!

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://gitlab.com/nbatfai/bhax/tree/master/attention_raising/Myrmeco



Amikor ránézünk a fájljainkra, láthatjuk, hogy 4 nagy különböző osztály van: ant, antwin, typdeg és az antthread. Fontos itt elemeznünk, hogy maga a 4 osztály mire is való.

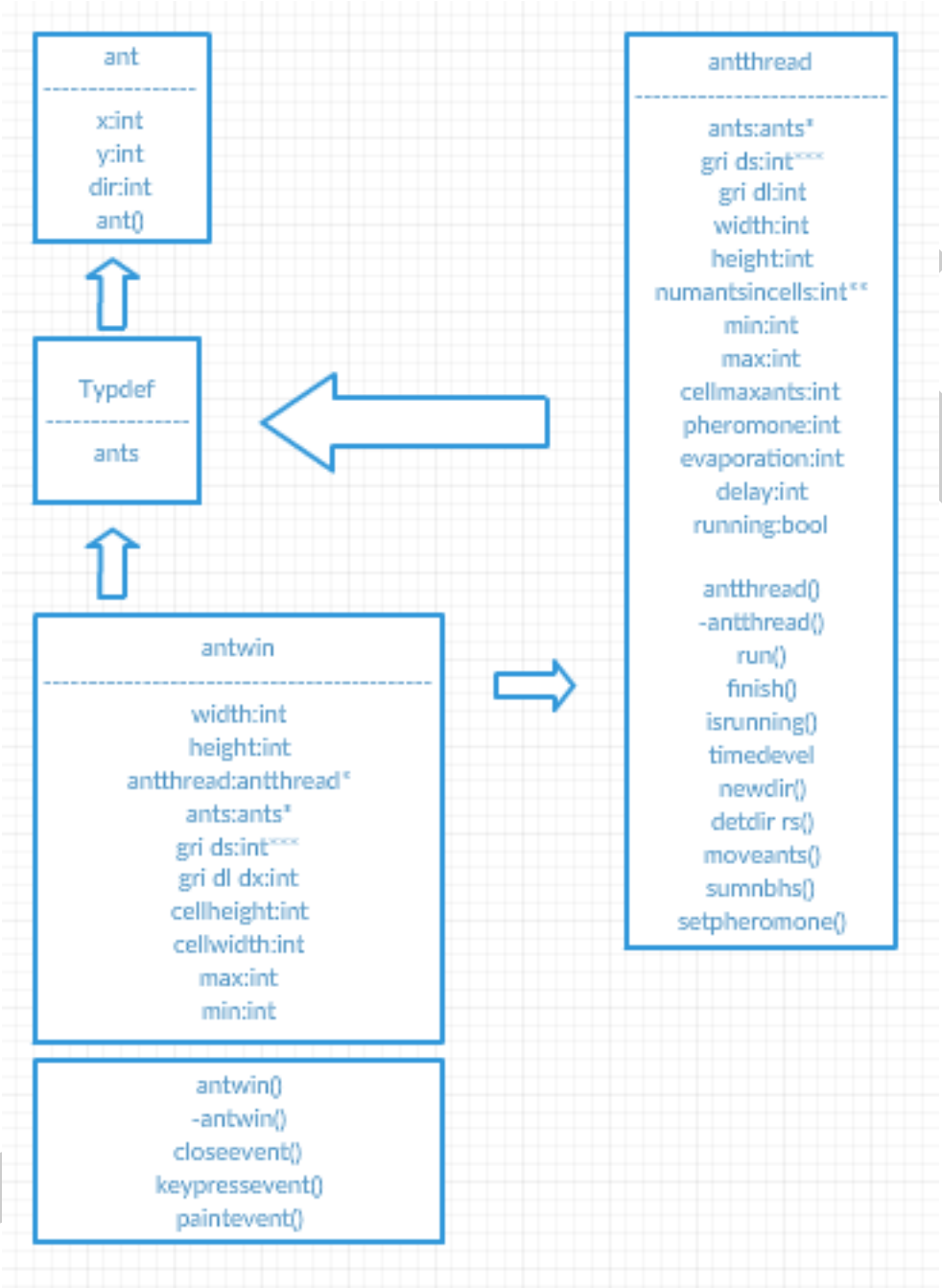
Kezdjük az elsővel, az ”ant”- ban találhatóak a hangyák tulajdonságai, itt található az x,y koordináta amelyből a pontos helyüket határozzuk meg, valamint ezeknek az elmozdításával, az irányukat, ”útjukat” is meg tudjuk határozni.

A második az ”antwin” osztály található, melyben az ablak méretét állíthatjuk be, amikben a hangyák mozogni fognak.Mindent ami az ablakkal kapcsolatos azt itt tudjuk állítani.

A harmadik osztály a ”typedef” mint a címkéből is gondolhatjuk, ez az osztály definiálja az ”ants” típust.

Az utolsó osztályunk pedig az ”antthread”. Ez az osztály az ”antwin” osztályban meg van hívva. Minden ami a hangyák tulajdonságát illeti itt tudjuk beállítani, pl. útvonal, amit követnek a hangyák, stb.

Diagrammal:



7.2. Java életjáték

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

Megoldás forrása: Java Sejtautó

A programot most egy picit másképpen kellesz futtatnunk, mint ahogyan a következő feladatban. A következő parancsokkal tudjuk lefuttatni, először írjuk be `javac Sejtautomata.java` majd ezután pedig `java Sejtautomata`.

Ezután pedig már láthatjuk is a sikló-kilövőnket.

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás forrása: Sejtautó

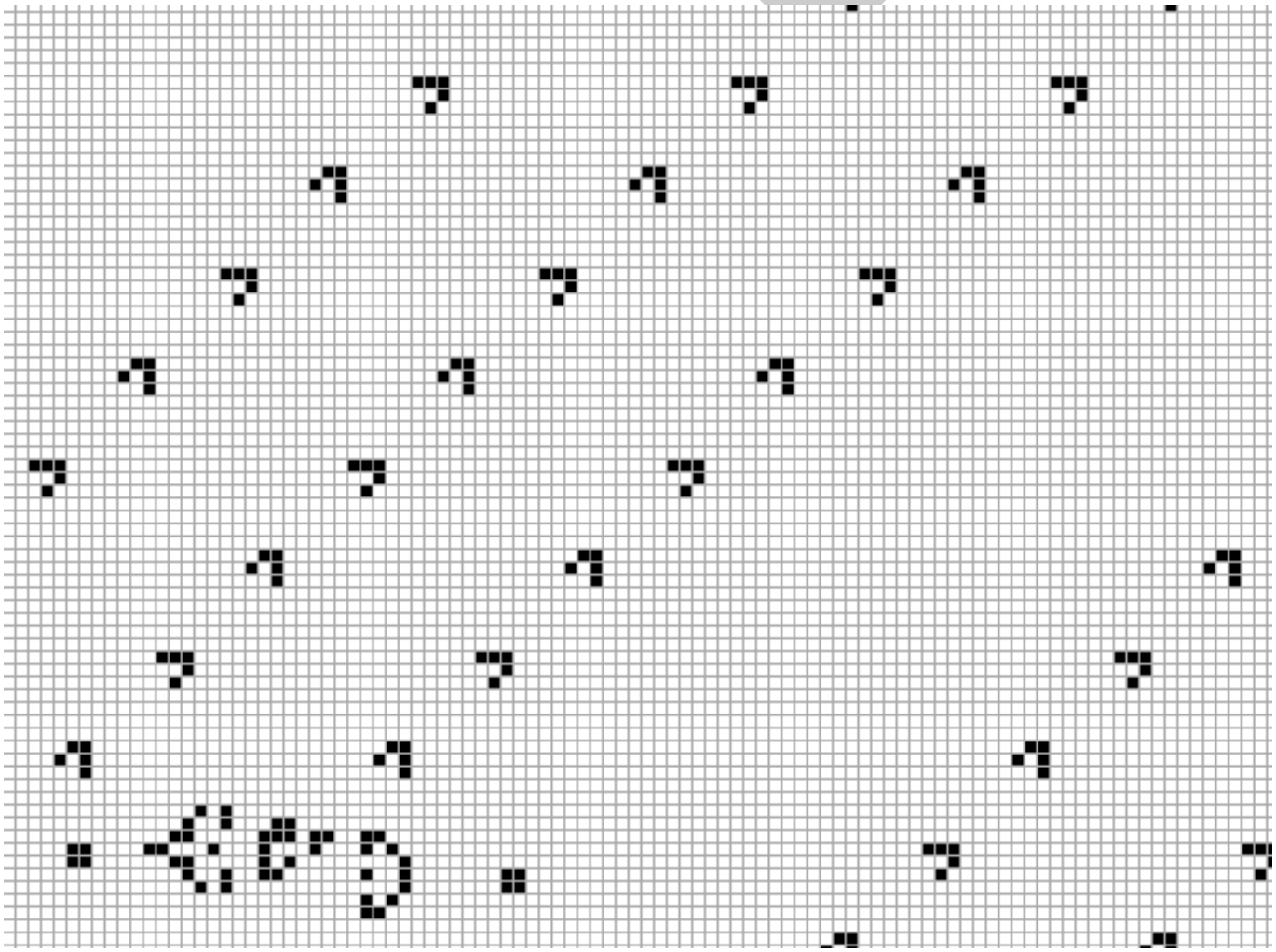
John Conway életjátékát fogjuk kipróbálni aki egy matematikus volt a Cambridge Egyetem-en. A játék címke picit megtévesztő lehet, ugyanis, nem egy tipikus játékról van szó, mivel a felhasználónak nem kell mást tennie, csak megadni egy kezdőalakzatot és nézni az eredményt. A játék egyes lépéseit a számítógép számolja ki és végzi el, lehet emberi erővel is, de ez nagyon hosszadalmas.

Van pár szabály amit tudnunk kell a kezdés előtt:

1: A sejt csak akkor éli túl a kört, ha 2 vagy 3 szomszédja van.

2: A sejt elpusztul ha 2-nél kevesebb vagy 3-nál több szomszédja van.

3: Új sejt születhet minden olyan cellában, melynek környezetében pontosan 3 sejt található.

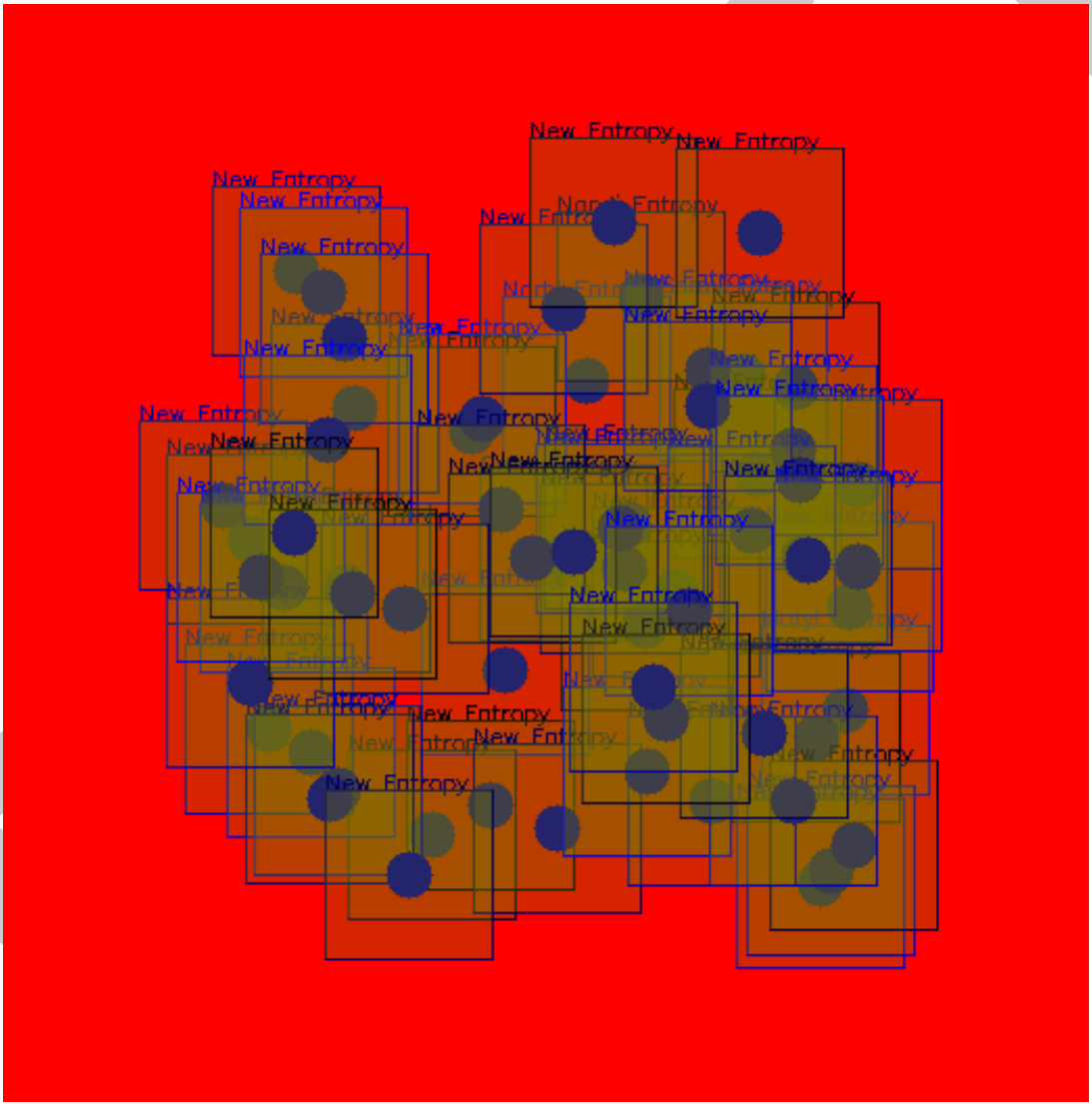


7.4. BrainB Benchmark

Megoldás forrása: BrainB Benchmark

Ennek a játéknak egyszerű a célja, a program segítségével megnézhetjük, hogy egy ”átlagos játékos”-t mi választ el attól, hogy profi E-sportoló lehessen, illetve, hogy megnézhessük a különbségeket. Elég csak pár percet eltöltenünk a játékban és máris fogjuk látni, mik a nehézségek.

A játék lényege, miután futtatjuk az az, hogy a képernyőn látni fogunk, bizonyos alakzatokat és egy fekete pöttyöt is. A mi feladatunk az, hogy az egeret a fekete pöttyön tartsuk. A dolgunkat viszont megnehezíti az, hogy a fekete pötty folyamatosan mozog a képernyőn, rázkódik és a különböző alakzatok alá, felé, mellé mozog.



8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

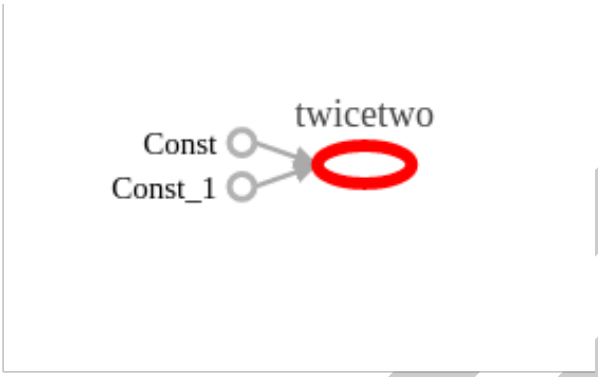
Megoldás videó: https://youtu.be/j7f9SkJR3oc

Megoldás forrása: https://github.com/tensorflow/tensorflow/releases/tag/v0.9.0 (/tensorfl
0.9.0/tensorflow/examples/tutorials/mnist/), https://progpater.blog.hu/2016/11/13/hello_s
bol

```
# TensorFlow Hello World 1!
# twicetwo.py
#
import tensorflow

node1 = tensorflow.constant(2)
node2 = tensorflow.constant(2)

node_twicetwo = tensorflow.mul(node1, node2, name="twicetwo")

sess = tensorflow.Session()
print sess.run(node_twicetwo)

writer = tensorflow.train.SummaryWriter("/tmp/twicetwo", sess. ↩
   graph)
# nbatfai@robopsy:~/Robopsychology/repos/tensorflow/tensorflow/ ↩
   tensorboard$ python tensorboard.py --logdir=/tmp/twicetwo

tensorflow.train.write_graph(sess.graph_def, "models/", " ↩
   twicetwo.pb", as_text=False)
# nbatfai@robopsy:~/Robopsychology/repos/tensorflow/tensorflow/ ↩
   twicetwo$ bazel build :twicetwo
# nbatfai@robopsy:~/Robopsychology/repos/tensorflow/bazel-bin/ ↩
   tensorflow/twicetwo$ cp -r ~/Robopsychology/r
```

Alapismertetőnek, hogy tudjuk mivel is dolgozunk: A TensorFlow egy nagyon ismert program melyet még a híres ”Google” cég is használ. Nyílt forráskódú.

Az első program amit megírunk nem nehéz, mivel csak annyit fogunk csinálni, hogy összeszorzunk két számot egy neurális háló segítségével Python-ban. Fentebb láthatjuk is a forráskódunkat.



Feljebb látható egy számítási gráf. A lényege, hogy van 2 darab konstans csomópont , ezek a twicetwo műveleti csomópontnak a bementeire vannak kötve, ami majd összeszorozza őket.

8.2. Mély MNIST

Python

Megoldás forrása:

PASSZ

8.3. Minecraft-MALMÖ

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

Megoldás forrása:Minecraft tematikájú programozások

Mielőtt elkezdenénk Minecraftban programozni, először is tisztázzuk mi is az a Minecraft. Ez egy nyílt világú játék, melyben barkácsolhatunk az összegyűjtött anyagákból, építhetünk belőlük, harcolhatunk szörnyek ellen, felfedezhetünk, és tulajdonképpen kiélhetjük a gyerekkori LEGO vágyainkat egy számítógépes játékon belül.

Ebben a feladatban egy mesterséges inteligenciát próbálunk meg megírni, mely folyamatosan előre fog menni a játékban. Ez nem is hangzik túl bonyolultnak, de a Minecraft világa ezt nagyon megnehezíti különböző dolgokkal, mint pl. hegyek, tavak, gödrök, szakadékok, láva és hasonló dolgok.

A lényeg az lenne, hogy a karakterünk (Steve) először is kikerülje az akadályok a koordinátákat figyelve. Ha viszont ezt nem sikerül és tegyük fel, beleesik egy gödörbe, akkor ne álljon meg helyben, hanem ”nézzen körül” vizsgálja meg a koordinátákat és a legoptimálisabb módszerrel kijusson onnan.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:Faktoriális

Ahhoz, hogy el tudjunk kezdeni Lispben programozni, az én esetemben(MACOSX), de Linux disztribúciók alatt is szükségünk van arra, hogy telepítsünk egy Lisp-et futtató programot, ami ebben a két esetben a `clisp`.

Első fontos dolog, amit muszáj megjegyeznünk a Lispben való programozás elkezdése előtt, hogy nem a megszokott módon végezzük el az adott számítási műveleteket, így az általunk ismert (1+1)-ből (+ 1 1) lesz.

Továbbá amit fontos megjegyeznünk, hogy a programokat `clisp fájlnevünk.lisp` módon fogjuk futtatni, valamint a fent leírt szabály alapján fontos, hogy szóközzökkel választjuk el a ”paramétereket”.

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_Li
Chrome

Ahhoz, hogy használni tudjuk a króm effektet azt kell tennünk, hogy letöltjük a ”bhax_chron
nevű fájlt, amit be kell helyeznünk a GIMP ”scripts” nevű mappájába, amit alapértelmezett esetben itt tudunk elérni: `~/.config/GIMP/2.10/scripts`

Ezután már személyre is tudjuk szabni az effektünket valamint használni is tudjuk már.

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_pro

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

A következő feladatban is GIMP-et fogunk használni, ráadásul, kipróbáljuk azt, hogy hogyan is hozhatunk létre egy mandalát a programban.

A forráskódot bemásolhatjuk akár a Gimps Script-fu konzoljába, és akkor kapunk egy mandalát, de ha személyre szeretnénk szabni, akkor a fájlt a `~/.config/GIMP/2.10/scripts` helyre kell bemásolnunk, és akkor már a programon belül tudjuk személyre szabni(szín, szöveg, felület, font, stb...).

10. fejezet

Helló, Gutenberg!

10.1. Programozási alapfogalmak

[**?**]

A programozási nyelveknek 3 szintjét különböztetjük meg:

- Gépi nyelv

- Assembly szintű nyelv

- Magas szintű nyelv

A magas szintű programozási nyelven megírt programokat forrásprogramoknak illetve forrásszövegeknek nevezzük, de inkább az első verzió a használatosabb. A forrásprogramokból pedig el kell jutnunk egy gépi nyelvű programokig, amit a számítógépünk is értelmezni tud. Ezt fordítóprogramok segítségével tudjuk megtenni. A fordítóprogram különböző lépéseken megy keresztül:

- Lexikális elemzés

- Szintaktikai elemzés

- Szemantikai elemzés

- Kódgenerálás

Továbbá minden forrásprogramnak vannak kissebb alkotórészei, a legkisebbek a karakterek. Az eljáráorientált nyelvek esetén a karakterkészlet így néz ki:

- Lexikális egységek

- Szintaktikai egységek

- Utasítások

- Programegységek

- Fordítási egységek

- Program

A C programozási nyelvben az alapelemek így néznek ki:

Aritmetikai típusok

- Egész
- Karakter
- Valós

Származtatott típusok

- Tömb
- Függvény
- Mutató
- Struktúra
- Union

Void típus

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

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

Az első fejezetben, egyből láthatunk egy egyszerű kis C programot, ami így néz ki:

```
#include <stdio.h>

main()
{
printf(”Halló mindenki!\n”);
}
```

ez egy egyszerű szöveg kiiratása, talán amit mindenki ismer még aki nem is programozó a: ”Hello World”. Ezt követve megtudjuk, hogyan kell fordítanunk és futtatnunk a forráskódunkat, majd a végeredményt is láthatjuk.

Ezt követően megtanuljuk az alapvető utasításokat, hogy hogyan tudjuk a kiirandó szövet változtatni (új sorba rakni, stb..), megtanuljuk, hogyan kommentelhetünk a forráskódunkba ami egyébként egy rendkívül hasznos dolog, hiszen nagyon száraz látni egy forráskódot, midnenféle magyarázat nélkül. Megtanuljuk a változók használatát, különböző utasításokat mint pl. (while, for, stb..), továbbá még a változók típusaival is megismerkedünk, valamint, hogy hogyan ágyazzuk be a különböző függvénykönyvtárakat és, hogyan használjuk a `define` név helyettesítő szöveget.

Apróbb programokat írunk meg közben, mint pl. (karakterek számlálása, sorok számlálása). Majd ezt követően a tömbökkel foglalkozunk és ezáltal megtanulunk létrehozni különböző tömböket valamint ezzel párhuzamosan az `IF` függvény többszöri egybeágyazását (Nested-IF) is gyakoroljuk.

A második fejezet elején a változó típusokkal foglalkozunk és definíció szerint meg is tanuljuk őket a fejezet során.

```
char
int
float
double
```

Valamint ezekhez hozzájöhetnek specifikációk mint pl.

```
short
long
signed
unsigned
```

A fejezetet tovább olvasva a deklarációkról olvashatunk, megtanuljuk a különböző logikai operátorokat és használatukat is:

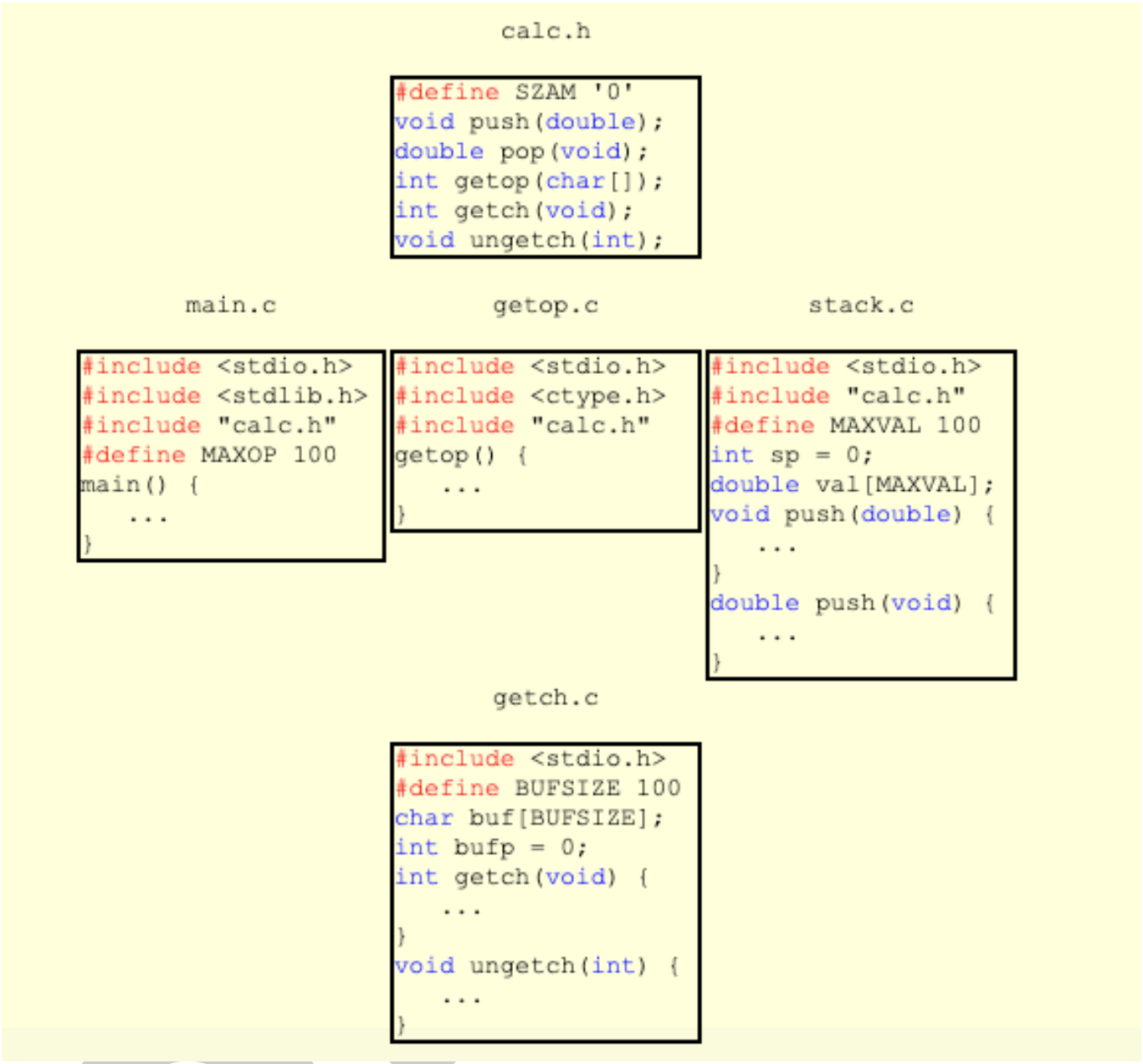
```
>
>=
<=
<
```

Majd különböző értékadó operátorokat és kifejezésket nézünk meg.

A harmadik fejezetben legelőször az utasításokkal foglalkozunk részletesebben (IF, IF- else, switch, case), hogy hogyan is hazsnáljuk őket helyesen és mikor érdemes őket használni. Majd különböző ciklusokat írunk a (while és for) utasításokkal. Ezt követően megnézzük ezeket a ciklusokat (do-while) utasításokkal is.

Miután már tisztában vagyunk az utasításokkal ezután a (break, continue) utasításokat nézzük át kis forráskódokban.

A negyedik fejezet a függvényekről szól és a valük kapcsolatos alapfogalmakkal. Tovább olvasva a header állományokkal foglalkozunk:



Majd a különböző változótípusokkal folytatjuk. Továbbá kiemelném a ”Rekurzió”-s részt amit fontosnak tartok és mehetünk is tovább a következő fejezetre.

A következő fejezetben a mutatók és tömbök kapnak főszerepet. Megtanuljuk kezelni a mutatókat és függvényargumentumokat, valamint belemerülünk a tömbökbe illetve többdimenziós tömbökbe is. Mint eddig is itt is nagy segítségünkre lesznek a kis kódcsipetek melyek szemléletesen bemutatják egy egy kis szakasz illetve lényeges rész működését.

A hatodik fejezetben a struktútákkal fogunk foglalkozni, először is az alapfogalmakkal:

```
struct pont
{
int x;
int y;
```

```
};
```

Majd különböző olyan dolgokkal foglalkozunk mint például a ”Struktúrák és függvények” , ”Struktúratömbök”. Majd ezt követően rátérünk a `typedef` utasításra

A következő hetedik fejezetben az adatbevitel és adatkivitel a főbb témánk. Itt kitárgyalunk olyan utasításokat mint a (printf, scanf, stb..). Megtanuljuk, hogy a programunkból, hogyan féhetünk hozzá különböző adatállományokhoz. Majd egy fontos rész követi ezt a ”Hibakezelés”

Majd az utolsó nyolcadik fejezetben az UNIX alapú operációs rendszerekről beszélünk, különböző rendszerhívásokkal, különböző fügvényekkel és kiiratásokkal zárjuk az utolsó fejezetünket.

10.3. Programozás

[BMECPP]

A legfontosabb rész amivel ebben a könyvben találkozunk az a hibakezelés. Először tisztában leszünk azzal, hogy hogyan néz ki egy hibakezelés folyamata, valamint pár példa programkódokat is kapunk.

Majd ezt követően a hibakezelés alapjaival fogunk foglalkozni. A legismertebb kivételkezelési módszer a try/catch módszer mely úgy fut le, hogy a forráskódunkba beépítünk egy try/catch blokkot, majd amikor hiba keletkezik a forráskódunkban ez a blokk ezt elfogja és a forráskódunk csak akkor fog minden gond nélkül lefutni miután már a try/catch blokk lefutott.

Nézzünk egy példát:

Kivételkezelés

A forráskódban szemléletesen láthatjuk is a try/catch blokkot.

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

Valamint a catch része:

```
catch (std::invalid_argument& e)
{
std::cout << e.what() << std::endl;
}
catch (std::ios::failure& e)
{
```

```
                std::cout << e.what() << std::endl;
                }
```

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. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

Az előző feladat kódját fejleszd tovább: vizsgáld, hogy Vannak-e Java osztályok a Pi hexadecimális kifejtésében!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

Köszönet illeti a NEMESPOR, https://groups.google.com/forum/#!forum/nemespor, az 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.