Univerzális programozás

Így neveld a programozód!

Ed. BHAX, DEBRECEN,
2020. március 2, v. 0.0.5

COLLABORATORS

	TITLE : Univerzális programozás	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, Bátfai, Margaréta, Ã©s Tutor, Tünde	2020. április 21.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	A Brun tételes feladat kidolgozása.	nbatfai
0.0.5	2020-03-02	Az Chomsky/$a^nb^nc^n$ és Caesar/EXOR csokor feladatok kiírásának aktualizálása (a heti előadás és laborgyakorlatok támogatására).	nbatfai

Tartalomjegyzék

Ábrák jegyzéke

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]

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

Mi a programozás?

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

Milyen doksikat olvassak el?

- Kezd ezzel: http://esr.fsf.hu/hacker-howto.html!
- Olvasgasd aztán a kézikönyv lapjait, kezd a **man man** parancs kiadásával. A C programozásban a 3-as szintű lapokat fogod nézegetni, például az első feladat kapcsán ezt a **man 3 sleep** lapot
- C kapcsán a [KERNIGHANRITCHIE] könyv adott részei.
- C++ kapcsán a [BMECPP] könyv adott részei.
- Az igazi kockák persze csemegéznek a C nyelvi szabvány ISO/IEC 9899:2017 kódcsipeteiből is.
- Amiből viszont a legeslegjobban lehet tanulni, az a The GNU C Reference Manual, mert gcc specifikus és programozókra van hangolva: szinte csak 1-2 lényegi mondat és apró, lényegi kódcsipetek! Aki pdf-ben jobban szereti olvasni: `https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf`
- Az R kódok olvasása kis általános tapasztalat után automatikusan, erőfeszítés nélkül menni fog. A Python nincs ennyire a spektrum magától értetődő végén, ezért ahhoz olvasd el a [BMECPP] könyv - 20 oldalas gyorstalpaló részét.

Milyen filmeket nézzek meg?

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

- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a 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

Hello, Turing!

2.1 Végtelen ciklusok

Egy mag 100%-on: Az első végtelen ciklus egy processzor magot használ 100%-ban. Ez úgy lehetséges hogy itt egy while ciklust használunk mi folyamatosan vizsgálja a megadott feltételt, ehhez 1 cpu magot használ viszont annak a teljes teljesitményét muszáj kihasználnia hogy folyamatosan figyelje a ciklus feltételét.

Egy mag 0%-on: A második ciklus egy cpu magot közel 0%-ban használ. Ehhez használtunk egy for ciklust, a végtelen for ciklus assemly nyelven ugyanaz mint a végtelen while ciklus, alapvetőleg mindkettő 1 magot használ 100%-ban. Ahhoz hogy a cpu mag 0%-ot használjon használnunk kell a sleep(1) függvényt ami a számolást késlelteti ezért cpu rá van kényszerülve a késleltetésre, viszont így sem tudjuk elérni hogy pontosan 0%-on menjen, inkább közelit a 0-hoz, a gyakorlat azt mutatja hogy így kb 1%-on megy.

Az összes mag 100%-on: (WIP) A harmadik ciklusunk minden cpu magot 100%-osan használ. Ehhez szükségünk van egy for ciklusra és...

2.2 Lefagyott, nem fagyott...

A példában emlitett „Lefagy” függvénynek bármely programról el kell tudnia dönteni hogy tartalmaz-e végtelen ciklust. Ez nem feltétlen valósulhat meg, pl.: ha tartalmaz végtelen ciklust akkor true értékkel kell visszatérnie viszont ha nem tartalmaz akkor elindit egy végtelen ciklust. Ekkor ha a T1000 program „nem lefagyó” akkor mindenképp lefagy mert ha a „Lefagy” függvény nem true értékkel tér vissza elinditja a végtelen ciklust. Tehát ilyen esetben saját magáról nem tudja eldönteni hogy van-e benne végtelen ciklus avagy sem.

Megjegyzés: Néhány fejlesztői környezet jelzi ha a programunkban van végtelen ciklus de ezek is többnyire csak a legegyszerübbekre tudnak szürni szintaktika alapján pl.: ha a programunk tartalmaz while(true), while(1) vagy for(;;) ciklusokat. Amikor először próbáltam az első feladathpz irni ilyen ciklust nekem a fejlesztői környezetben való futtatáskor néhány másodperc után kiléptette a ciklusból.

2.3 Változócsere

Logikai utasitások nélkül a változók cseréjének legegyszerübb módja segédváltozóval megcserélni az értékeket de egyéb megoldások is vannak pl.: az összeadásos vagy kivonásos. Logikai utasitással is meg tudjuk ezt tenni, a legjobb példa erre a bitenkénti kizáró vagy (XOR). Az én programom 3 fajta változócserét tud végrehajtani, minden futtatáskor véletlenszerüen választott módszerrel de az eredmény nyilván mindig ugyanaz lesz.

2.4 Labdapattogtatás

A labdapattogtatásos feladat lényege hogy a képernyőt úgy tudjuk használni mint egy koordináta rendszert (mivel az is). A kiválasztott karakterünket, akár az O betüt megadott pályán tudjuk, elinditani, itt fontos hogy kicsit elcsúsztatva kezdjük el a pattogtatást hogy be tudja járni az egész képernyőt. Ha pl x egyenlo 2, y egyenlo 2 értékeken inditjuk el akkor csak a képernyő egy átlóját tudja bejárni mivel csak azon az utvonalon fog oda-vissza pattogni. A képernyőnket mátrixként is lehet értelmezni, nekem eszembe is jutott hogy a feladatot egy arduino és egy 8x8-as ledmátrix segitségével is megoldjam (WIP).

2.5 Szóhossz és Linus Tovald -féle BogoMIPS

Ennél a feladatnál a BogoMIPS program while ciklusának fejét kell felhasználnunk ami: while(loops_per_sec egyenlo 1) formájú. A ciklus fejében találunk egy „bitwise” operátort. Ez a „.. ” operátor alapvetőleg 2-vel szorozza a bal oldalán lévő változót, a jobb oldalán pedig egy egyes számot látunk, az operátor jobb oldalán azt a számot adhatjuk meg hogy a szorzást 2 hanyadik hatványával szeretnénk megtenni pl.: 9 .. 2 egyenlo 36. Ennek az operátornak van egy párja is „... ” itt a 2 hatványokkal való isztás valósul meg az előzőhöz hasonló módon. Pl: 36 .. 2 egyenlo 9. Ha tudjuk pl.: a string amiről szeretnénk megtudni hogy hány bit hosszú így viszonylag könnyen utána tudunk számolni a szóhosszúságát felhasználva. Általában egy karakter 8 bit de az UTF-8 esetén akár 32 bit is lehet egy karak

2.6 Hello Google! (WIP)

Ebben a feladatban egy 4 honlapból álló PageRank meghatározást kell készitenünk. Egy oldal PageRank értékét az befolyásolja hogy mekkora PageRank értékü oldalak hivatkoznak rá. Azaz minél jobb oldalak hivatkoznak az én honlapomra, az én honlapom annál jobb PageRank értékkel fog rendelkezni. Egy oldal PageRank értéke a rá mutató oldalak PageRnkank értékének és az adott oldalról kifele vezető linkek hányadosainak összege. Minél több linkből áll a PageRank hálózatunk annál bonyolultabbá válik. A példánkban 4 oldal PR értékeit kell kiszámolunk tehát egy 4x4-es mátrixot fogunk hasznélni ehhez.

2.7 Monty Hall probléma

A Monti Hall problémát én megpróbáltam sziulálni. Készitettem menüt, tudunk választani ajtót, azt pedig hogy melyik ajtó mögött van a nyeremény véletlenszerüen határozza meg a programunk. Én úgy fogtam hozzá hogy 3 fő eset van amikor a nyeremény az 1-es, a 2.es és a 3-mas ajtó mögött van. Ezeket az

eseteket alesetekre bontottam aszerint hogy a játékos melyik ajtót választja ki így 3*3 különböző eset lehetséges. Ha a program meghatározta melyik ajtó a nyertes és a játékos is választott magának ajtót az egyiket szükségképpen egy olyan ajtót „kinyitunk” amelyben biztosan nem a nyeremény van. Ekkor a játékosnak felajánlunk egy választási lehetőséget hogy az eredeti döntését megváltoztatja-e. Ennél a pintnál a játék lehetséges kimenetelei 3*3*2-re nőttek.

2.8 Brun Tétel

Ennél a fealadtál az első dolgunk olyan programot irni ami meghatározza mely primek ikerprimek. Én ehhez egy for ciklust szeretnék használni amelyben ha két szomszédos prim különbsége éppen kettő ezeket az elemeket kiválasztom és egymással párba rendezem. Ahhoz hogy megtalálhassuk a Brun konstans közelitő értékét a primek reciprokösszegét kell összeadunik, ehhez -1 hatványra emelem őket majd a kapott számokat összeaduk.

MAIMÖ csiga feladat

Youtube link:video here

3. fejezet

Helló, Chomsky!

3.1 Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Ebben a feladatban decimálisból unáris számrendszerbe váltó Turing gépről van szó, ahhoz hogy tudjuk mi ez tudnunk kell melyek a decimális illetve unáris számrendszerek. A decimális a jól ismert tizes számrendszer, melyben 10 számjegyből (0-9) tudjuk leképezni a számokat. Általános iskolából emlékezhetünk arra, hogy az ilyen tizes számrendszerbeli számokat oly módon irtuk fel hogy milyen számok vannak egyes helyiértékeken, pl.: egyes, tizes, százas, ezres. stb. Az adott helyiértéken állü számjegyet meg kell szoroznunk a helyiértékkel, majd ezekből összeget képzünk. Tehát pl.: a 321 = 3*100 + 2*10 + 1*1. Az egyes számrendszer olyan számrendszer melyben a számokat 1 számjeggyel az 1-essel tudjuk leirni mégpedig oly módon, hogy amennyi maga a szám egymás után annyi egyest irunk pl.: 3 = 111. A gép a két számrendszer között úgy vált át hogy a decimális számból addig von ki 1-eseket mig az 0 nem lesz.

Ha ezt az átváltást program segitségével szeretnénk lemodellezni megoldás lehet ha mondjuk egy while ciklust használnunk amelynek kilépési feltétele hogy a decimális számuk 0 legyen. A ciklus törzsében minden lefutáskor levonunk a decimális számból egyet és egy 1-es számjegyet hozzáfüzünk pl egy (a program idnitásakor még üres) txt állományhoz. Ekkor a ciklus végére a txt állományban keletkezett adat az eredeti decimális számunk unáris alakjának felel meg

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

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

Megoldás videó:

A generativ grammatika olyan nyeltani szabályrendszert jelent melyben véges számú szóból és véges szóból és véges számú szabályból végtelen számú mondatot lehet alkotni.

Sok olyan jelenség van am it nem lehet környezetfüggetlen nyelveken leirni, ilyen pl.:a természetes nyelvek, a primszámok halmaza. A környezetfüggő nyelvek zártak a halmazmüveletekre az unió müveletet leszámitva. S (S $\rightarrow$ aXbc) aXbc (Xb $\rightarrow$ bX) abXc (Xc $\rightarrow$ Ybcc) abYbcc (bY $\rightarrow$ Yb) aYbbcc (aY $\rightarrow$ aaX)

aaXbbcc (Xb → bX) aabXbcc (Xb → bX) aabbXcc (Xc → Ybcc) aabbYbccc (bY → Yb) aabYbbccc (bY → Yb) aaYbbbccc (aY → aa) aaabbbccc

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

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.

A BNF környezetfüggetlen szintaxisokat leiró szintaxis.

A C nyelvi utasitás fogalmának szintaktikai definiciója BNF sintaxisban:

```
<utasitás>::=
        <összetett_utasitás>
        <kifejezés>; (értékadás pl, num=10)
        if(<kifejezés>) <utasitás>
        else if(<kifejezés>) <utasitás>
        else <utasitás>
        switch (<kifejezés)
        <egész_konstans_kifejezés : <utasitás>
        goto <azonositó>;
        <azonositó> : <utasitás>
        break; continue; return<kifejezés>;
        or(<kifejezés1><kifejezés2><kifejezés3>) <utasitás>
        while(<kifejezés>) <utasitás>
        do <utasitás> while<kifejezés>
        ; (üres utasitás, pl FORTRAN continue-ja)
```

Példa olyan programra ami a C89-es szabvánnyal nem fordul de C99-cel igen. Ebben a programban ciklusson belül deklarálok változót ami a C99-es szabványtól lehetséges. Ha C89-cel próbáljuk meg forditani akkor a forditó javasolja is a C99 hazsnálatát.

```
int main()
{
    for(int i = 0; i < 20; i++)
    {
        printf("Cikluson beluli deklarálás.");
    }
}
```

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!

A lexer programok szöveg elemző programok amivel szövegeket lehet tetszőleges szempontból elemezni illetve átalakitani. A példaprogram a beirt szövegben valós számokat keres. A program 3 fő részre bontható (%% az elválasztójel), az első részben meghivjuk a szükséges header fájlokat és deklarálhatunk változókat. A második részben definiáljuk hogy mit is keresünk a szövegben, illetve mit csináljunk vele, azaz szabályokat alkotunk. Ebben a programban valós számokat keresünk, tehát olyan számokat amelyek tetszőleges számú számjegyből állhatnak vagy tetszőleges számú számjegyből és 1 db pontból majd újabb tetszőleges számú számjegyből állnak. A megtalált valós számokat a program a már átalakitott szövegben kiemeli. Minden megtalált valós szám után növeli a realnumbers nevü változót is, tehát számolja is a megtalált számokat. A program utolsó része a main függvény ahol meghivhatjuk a lexert és saját utasitásokkal egészithetjük ki a programot.

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/realnumber.l

```
%{
#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;
}
```

Tanulságok, tapasztalatok, magyarázat...

3.5 Leetspeak

Lexelj össze egy l33t ciphert!

Az előző programhoz hasonlóan ez is egy lexer program. Ez úgy müködik, hogy az adott szövegben ki tujda cserélni a karaktereket egy az eredetihez hasonló karakterre (9-est egy g betüre). Ezt olyan módon teszi meg hogy az angol abc betüihez illetve 0-9-ig a számokhoz társit 4 kinézetre hasonló karaktert (ha nincs 4 hasonló akkor pl 2 hasonlót kétszer társitunk hozzá). A program a beolvasást kisbetüsiti tehát mindegy hogy a bemenetre kis vagy nagy betük érkeznek felismeri az adott karaktert. A kicserélést random

módon teszi, 0 é 100 között sorsol egy random szémot, ha 91-nál kiseb akkor az első társitott karakterre cseréli a megtaláltat, tehát erre nagyobb esély van mint a többire. A második karakterre 4% esély van, a harmadikra 3% és a n egyedikre 2%.

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/l337d1c7.l

```
%{
  #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', {"0", "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;
}
```

Tanulságok, tapasztalatok, magyarázat...

3.6 A források olvasása

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?

1.
```
if(signal(SIGINT, SIG_IGN)!=SIG_IGN)
    signal(SIGINT, jelkezelo);
```
2.
```
for(i=0; i<5; ++i)
```
3.
```
for(i=0; i<5; i++)
```
4.
```
for(i=0; i<5; tomb[i] = i++)
```
5.
```
for(i=0; i<n && (*d++ = *s++); ++i)
```
6.
```
printf("%d %d", f(a, ++a), f(++a, a));
```
7.
```
printf("%d %d", f(a), a);
```
8.
```
printf("%d %d", f(&a), a);
```

1. Akkor és csakis akkor kezelje a 'jelkezelo' függvény a SIGINT jelet, ha az nincs ignorálva.
2. Egy for ciklus ami a ötször hajtja végre a hozzá rendelt utasitásokat. Preorder módon először az i-t növeli és csak aztán végzi el az utasitásokat.
3. Egy for ciklus ami a ötször hajtja végre a hozzá rendelt utasitásokat. Postorder módon először elvégzi az utasitásokat és csak azután növeli az i-t.
4. Egy for ciklus ami a ötször hajtja végre a hozzá rendelt utasitásokat. Viszont a tomb[] első öt értékét lecseréli az aktuális i értékre. Tehát 0,1,2,3,4.
5. WIP
6. A standard outputra kiiratjuk az f() függvény visszatérési értékét, decimális számban.
7. A standard outputra kiiratjuk az f() függvény visszatérési értékét 'a'-ra és magát az 'a'-t is, decimális számban.
8. A standard outputra kiiratjuk az f() függvény visszatérési értékét 'a'-ra (mivel annak a memória címére mutatunk) és magát az 'a'-t is, decimális számban

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat...

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)) $

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

- A primszámok száma végtelen.
- Az ikerprimek száma végtelen.
- A primszámok száma véges.
- A primszámok száma véges. (itt egy az előzővel ekvivalens formula volt)

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

Tanulságok, tapasztalatok, magyarázat...

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

```
int main()
            {

                //egész
                int num = 32;

                //egészre mutató mutató
                int* toNum = &num;

                //egészek tömbje
                int numArray[4] = {0,1,2,3};

                //egészek tömbjének referenciája
                int (&numArrayRef)[4] = numArray;

                //egészre mutató mutatók tömbje
                int* pointerArray[4];

                //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


                return 0;

            }
```

Mit vezetnek be a programba a következő nevek?

- `int a;`
- `int *b = &a;`
- `int &r = a;`
- `int c[5];`
- `int (&tr)[5] = c;`
- `int *d[5];`

- ```
  int *h ();
  ```
- ```
  int *(*l) ();
  ```
- ```
  int (*v (int c)) (int a, int b)
  ```
- ```
  int (*(*z) (int)) (int, int);
  ```

Megoldás videó:

- `a` mint egész tipusú változó.
- `a` memória cimére mutató mutató.
- Egy egészre mutató mutatót `r` névvel, ami az `a` értékét mint mutatócim tartalmazza.
- Öt elemü tömb, mely egészekből áll.
- Egy 5 elemü, egészeket tartalmazó tömbre mutató mutató, mely a `c` tömbre mutat.
- Öt elemü egészekre mutató mutatókból álló tömb.
- Egésszel visszatérő, paraméter nélküli függvényre mutató mutató.

Az utolsó két deklarációs példa demonstrálására két olyan kódot írtunk, amelyek összahasonlítása azt mutatja meg, hogy miért érdemes a **typedef** használata: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr.c, bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Chomsky/fptr2.c.

```
#include <stdio.h>

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

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

int (*sumormul (int c)) (int a, int b)
{
    if (c)
        return mul;
    else
```

```
        return sum;

}

int
main ()
{

    int (*f) (int, int);

    f = sum;

    printf ("%d\n", f (2, 3));

    int (*(*g) (int)) (int, int);

    g = sumormul;

    f = *g (42);

    printf ("%d\n", f (2, 3));

    return 0;
}
```

```
#include <stdio.h>

typedef int (*F) (int, int);
typedef int (*(*G) (int)) (int, int);

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

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

F sumormul (int c)
{
    if (c)
        return mul;
    else
        return sum;
}
```

```
int
main ()
{

    F f = sum;

    printf ("%d\n", f (2, 3));

    G g = sumormul;

    f = *g (42);

    printf ("%d\n", f (2, 3));

    return 0;
}
```

Tanulságok, tapasztalatok, magyarázat...

3.9 MALMÖ | Discrete csiga

A Turing feladatcsokorban hasonló feladattal találkozhattunk de ott diszkrét helyett folytonos mozgást alkalmaztunk. A két mozgás között jelentős különbségek vannak, mig folytonos mozgásnál a "move 1" parancs mint kapcsoló müködik ami miatt addig mozog amig a "time.sleep()" engedi, addig diszkrét mozgásnál a "move 1" utasitásként müködik tehát 1 blokknyit mozog előre a "time.sleep()"-ben megadott idő alatt. Azt hogy csigában haladjon mi úgy oldottuk meg, hogy először az legalsó szinten lévő virágot kibányássza majd egy szinttel ugorjon fentebb és ezután kezdj el a csigát. A csigához 2 db for ciklust használtunk, a belső for ciklusban mindig az oldalnyi mozgáshoz elegendő lépésmennyiséget teszi a külső for ciklus pedig azt teszi lehetővé hogy adott szinten az adott oldalhosszot 4x tegye meg, ennek elején mindig fentebb ugrik egy szintet.

Megoldás videó: vidi itt, katt

4. fejezet

Helló, Caesar!

4.1 Alsó háromszögmátrix

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

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

Először is mik azok a háromszögmátrixok? A háromszögmátrixok speciális kvadratikus mátrixok, álatlában 2 fajta háromszögmátrixot különböztetünk meg, van alsó pés felső háromszögmátrix. A felső háromszögmátrix jellemzője hogy a főátló alatt csupa 0 érték található, míg az alsó háromszögmátrix ennek fordítottja, annál a főátló fölött található csupa 0 érték.

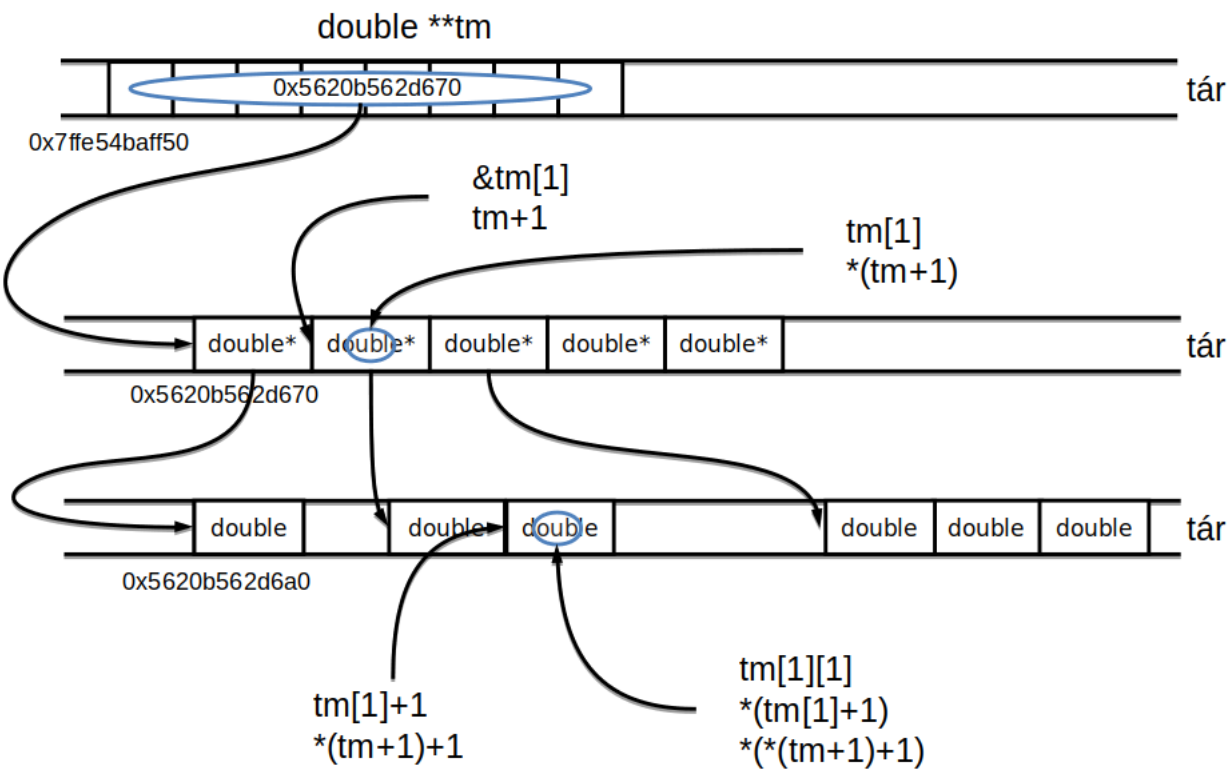
A példaprogram egy 5x5-ös alsóháromszögmátrix-ot ábrázol. A 8. sorban táluink egy if függvényt, ez egy hibakeresés és helyfoglalás egyben. A feltételében egy malloc-kal helyet foglalunk a mátrixunknak illetve megvizsgáljuk, hogy azok száma 0.e mert amennyiben a feltétel igaz nem beszélehtünk semmiképp háromszögmátrixról, és egy return -1, gibát kapunk vissza. Tehát ha pl a "nr" változó ami meghatározza hányszor hanyas lesz a mátrix, 0 akkor hibát ad vissza a program. Gyakori hogy a helyfoglalást egy if-en belül oldják meg, pint a hibakezelés miatt. Ezután egy preorder for cikluson beül helyet foglalunk a mátrixunk elemeinek ismét malloc-ot használva. A for cikluson belül pedig újabb hibakezeléssel találkozunk. A size ugye a mátrix beli elemek számát jelöli, ez sem lehet nulla, ha mégés az akkor return -1 és hibát ad vissza a program. Ha a program idáig helyesen lefutott és a helyfoglalással minden rendben volt, ezután 2 for cikluson belül feltölti a mátrixot. Az egyik for ciklus a mátrix sorain halad végig a másik pedig annak oszlopain. Alapból egy képlet segítségével az értékes elemek értéke az a szám ahanyadik értékes elemről van szó. Itt megjegyezném hogy az első értékes elem egy 1 mivel a C sok más programnyelvhez haonlóan 0-tól számol. Ezután ismét az előző 2 for ciklust felhasználva a program kíratja azok elemeit.

A program egyébként 2 db mátrixot ír ki a képernyőnkre. A második mátrixban bizonyos elemeknek konkrét értéket adva (42, 43, 44, 45). Ezt az értékadást minden elem esetében más módon teszi meg ezzel példákat adva a mutatók helyes használatára, ugyanis egy mátrix beli elemre több módon is hivatkozhatunk. Az első és egyben legegyszerűbb ha konkrétan megmondjuk mely elemről van szó "tm[s][o]". Ha a mutatókat is ki szeretnénk használni akkor több megoldás is helyes. Lehet mutatni a mátrixon blül csak sorra vagy oszlopra és a másik értéket konkrétan megadni vagy lejóhet a sorokhoz és az oszlopokoz is mutatót használni. Ezután ismét 2 for cilus segítségével kíratjuk magát a mátrixot.

A program utolsófontos része pedig a mátrixnak és annak elemeinek lefoglalt hely felszabadítása, ezt a free() függnyénnyel tehetjük meg.

```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;
}
```



4.1. ábra. A double ** háromszögmátrix a memóriában

Tanulságok, tapasztalatok, magyarázat...

4.2 C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás forrása: egy részletes feldolgozása az e.c és t.c forrásoknak.

Tanulságok, tapasztalatok, magyarázat... ezt kell az olvasónak kidolgoznia, mint labor- vagy otthoni mérési feladatot! Ha mi már megtettük, akkor használd azt, dolgozd fel, javítsd, adj hozzá értéket!

Az EXOR titkosító lényege hogy megadunk egy valahánybetűs kulcsot és egy szöveget. ezeket összeXOR-ozzuk. Ekkor az eredet szövegből egy furcsa bitsorozatot kapunk, a szövegünk titkosítva is van. Ha ezt a titkosított szöveget ismételten összeExorozzuk a kulccsal, akkor visszakapjuk az erdeti szövegünket. Hogyan lehetséges ez? Az XOR, azaz a kizáró vagy logikai művelet alkalmas pl karakterek bitenkénti cseréjére is,

itt gyakorlatilag vaz történik, hogy a szöveg bitjeit "összemossa" a kulcséval és egy felismerhetetlen bitsorozatot kapunk. Nyílván amikor ezt visszafele csináljuk ugyanaz történik és megkapjuk a tiszta szövegünket. Erről a javanál írok részletesebben.

4.3 Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

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

A java-ban megírt program rövid magyarázata:

Először is, a programnak szüksége van a kulcsra, amit egy stringben tárol el illetve kell egy 256 byte-os buffer a bemenetünknek. Ezek után egy while ciklus segítségével addig olvassa a bemenetet amíg az van, majd ezen belül egy for ciklust találunk. A for ciklus annyiszor hajtódik végre ahány beolvasott byte-unk, azaz karakterünk van, ezena for cikluson belül pedig a bifferbe beolvasott szövegből byte-onként összeEXOR-ozza a bufferben lévő dolgokat a kulcs egyik elemével.

```
        import java.io.InputStream;
import java.io.OutputStream;

public class main
{
    public static void encode (String key, InputStream in, OutputStream out ↩
       ) throws java.io.IOException
    {
        byte[] kulcs = key.getBytes();
        byte[] buffer = new byte[256];
        int kulcsIndex = 0;
        int readBytes = 0;

        while((readBytes = in.read(buffer)) != -1)
        {
            for(int i=0; i<readBytes; i++)
            {
                buffer[i] = (byte)(buffer[i] ^ kulcs[kulcsIndex]); //itt  ↩
                   csinálja az összeEXOR-ozást
                kulcsIndex = (kulcsIndex + 1) % key.length();
            }

            out.write(inputBuffer, 0, readBytes);
        }
    }

    public static main (String[] args)
    {
        if(args[0] != "")
```

```
        {
            try
            {
                encode(args[0], System.in, System.out); //
            }
            catch(java.io.IOException e)
            {
                e.printStackTrace();
            }
        }
        else
        {
            System.out.println("Please provide a key!");
            System.out.println("java main <key>");
        }

    }
}
```

4.4 C EXOR törő

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

Megoldás videó:

Megoldás forrása:

Adott volt az előző feladatban a t.c program ami arra való hogy ugye XOR titkosított szövegeket törjünk vele. Labor feladat volt ezt kivitelezni is, amit sikerült is megoldanom pár esetben. Nagyon felbosszantott hogy nem tudtam mindenki szövegét törni majd utána jártam a dolognak hogy mi volt a baj...

Előzmények: Az XOR törő program alapvetőleg a brute force technikát alkalmazza, azaz próbáljunk ki minden lehetséges kulcs kombinációt. Nyílván ez nem működhet mindenféle jelszótörésre meg egyéb haxorkodáshoz mivel ez az egyik leggyakoribb törési mechanizmus és gyakran pl a nemes egyszerűséggel ettől le vannak véve úgy hogy nem próbálkozhatunk mondjuk 5-nél több alkalommal. Na de itt nem eza helyzet szóval törjük meg azt a titkosítást... Alapból a program 8 karakteres kulcsokat tud törni, akiknek a szövegét én kinéztem nekik zömében 4 karakterből álló kulcsuk volt. Oké. A következő lépés az volt hogy a segítségként megadott karaktereket egy tömbbe vettem hogy ne kelljen a teljes ABC-n végészaladnia a programnak, illetve a for ciklusokból is kiszedtem 4 db-ot. Leteszteltem a saját szövegemen, a program készen állt a törésre. Ekkor meglepetésemre sorra nem törte a mások által titkoított szövegeket. Kérdezősködtem és a csoporttársaim nagyrésze ugyanígy járt. De ami igazán érdekes volt, hogy az XOR törőink ugyanazokat a szövegeket törték és ugyanazokat nem. Végül utána jártunk és az volt a baj hogy a törő a magyar szövegeket tudja hatékonyan törni így meg kellett elégedni azzal hogy azt a pár szöveget helyesen és viszonylag gyorsan törte a program.

Itt pedig az az XOR törő van mellékelve ami 4 karakteres kulcsot tör, amely a g, s, e és p karakterekből áll:

```
#define MAX_TITKOS 4096
            #define OLVASAS_BUFFER 256
```

```
#define KULCS_MERET 4
#define _GNU_SOURCE

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

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

  return (double) titkos_meret / sz;
}

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

  double szohossz = atlagos_szohossz (titkos, titkos_meret);

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

}

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

  int kulcs_index = 0;

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

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

    }
```

```
}

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

  exor (kulcs, kulcs_meret, titkos, titkos_meret);

  return tiszta_lehet (titkos, titkos_meret);

}

int
main (void)
{

  char kulcs[KULCS_MERET];
  char titkos[MAX_TITKOS];
  char betuk[5] = {'g','s','e','p', '\0'};
  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 <= 4; ++ii)
    for (int ji = 0; ji <= 4; ++ji)
      for (int ki = 0; ki <= 4; ++ki)
        for (int pi = 0; pi <= 4; ++pi)
          {
            kulcs[0] = betuk[ii];
            kulcs[1] = betuk[ji];
            kulcs[2] = betuk[ki];
            kulcs[3] = betuk[pi];

            if (exor_tores (kulcs, KULCS_MERET, titkos, p -  ↩
               titkos))
              printf
                  ("Kulcs: [%c%c%c%c]\nTiszta szoveg: [%s]\n",
```

```
                                ii, ji, ki, pi, titkos);

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

            return 0;
        }
```

4.5 Neurális OR, AND és EXOR kapu

R

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

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

Tanulságok, tapasztalatok, magyarázat...

Egy nagyon jó szemléltető videó: https://www.youtube.com/watch?v=qv6UVOQ0F44

A neurális háló egy nagyon érdekes de nem könnyű téma. Itt az első bekezdésben röviden bemutatnám miről is van szó. Egy neurális háló 2 főbb dologból áll: neuronokból és rétegekből. A neuronok lényege hogy a kapott adatokat továbbítani vagy módosítani legyenek képesek. A neuronoknak van egy bemeneti oldala, itt a bejövő adatokat szummázza illetve van egy kimeneti oldala. A valódi idegsjtekben a "bemeneti" adatok axonokon keresztül érkeznek, az axonok által összekötött idegsejtek pedig a neuron hálózatot alakítják ki, ehhez hasonlóan működnek a programozásbeli neurális halózatok is, ezeket gráffal tudjuk szemlélltetni. A neuronok tehát adatokat kapnak és azokat továbbítják. Ez a kapcsolat alakítja ki a rétegeket. 3 fő rétegből szoktak állni a neurális hálózatok, ezek a bemeneti, a rejtett és a kimeneti réteg. Rejtett rétegből lehet több, vagy akár egy sem, minél több rejtett rétegből áll egy hálózat annál komplexebb feladatokat képes végrehajtani. A rejtett réteget lehet műveleti rétegnek is nevezni, ahogy ezt sejteni lehet itt a kapott adatokkal a neuronok műveleteket végeznek, ezek leggyakrabban a legalapabb logikai műveletek, azaz OR, ANd, XOR stb. Léteznek úgy nevezett deep learning neural network-ök, ezeknek a lényege az hogy a neurális hálózatunkhoz, egy rewar systemes. jutalmazási rendszert adunk, ezek a hálózatok sok lefutás, generáción keresztül tanulnak, a folyamatosan érkező adatokat feldolgozzák. Minél pontosabb reward systemet alakítunk ki annál hatékonyabb lesz a tanulási folyamata.

Na de térjünk is rá a neurális OR, AND és XOR kapukra. Ezekhez 2 db input szükséges, az egyik lgyen A a másik pedig B. Az input módosítás nélül adja tovább az adatokat. Az OR és az And esetén nincs rejtett rétegünk, ezek elemi logikai műveletek. Ahhoz hogy a neurális hálónkat ezek elvégzésére megtanítsuk egfy data.frame táblára van szükségünk, iletve a neuralnet() függvényre. Ez azt jelenti, hogy a data-frame-ben megadjuk neki az inputokat és a művelet elvégzése ztáni eredményt, pédát mutatunk be.

Az AND és az OR műveleteket, rejtett réteg bélkül is el tudja végezni, akár a kettőt is egyszerre, de az XOR, azaz a kizáró vagy esetén más a helyzet. Ha ítéletlogikai formulaként felírjuk láthatjuk hogy egy sokkal összetettebb formuláról van szó, míg at AND egy elemi konjunkciónak felel meg az OR pedig ugye elemi diszjunkciónak. Az XOR-hoz 3 rejtett rétegre lesz szükségünk.

```
library(neuralnet)
```

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

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

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

plot(nn.or)

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


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

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

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

plot(nn.orand)

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



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

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

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

plot(nn.exor)

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



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

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

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

plot(nn.exor)

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

4.6 Hiba-visszaterjesztéses perceptron

C++

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

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

Tanulságok, tapasztalatok, magyarázat...

Ez a program egy elég érdekes program. Én elsősorban a megoldás videóban található programról szeretnék írni. A program a mandelbrot halmazról kirazoltatott png képpel dolgozik. Ebben a png képben fekte és fehér pixelek vannak, itt fontos megjegyezni hogy a fehér pixelek tartalmaznak piros színt míg a feketék nem. A program elején meg van hívva az mlp.hpp. Az mlp kibontva MUlti Layer Perceptron, ez adja a programhoz a 3 rétegű neurális hálót, ettől lesz képes tanulni. A 9. sorban van deklaráva a kép mérete azaz size, ami a szélesség*magasság, azaz a tartalmazott pixelek száma. A program további részészében a 13. sorig a png-nek való helyfoglalás történik, a képet mint egy mátrixot kell elképzelünk, ugye van szélessége (oszlopok) és magassága (sorok). Ezután létrehoz egy p, Perceptron típusú mutaót, ait az mlp beli osztály miatt tehet meg, illetve egy double típusút, ami az image, azaz maga a png kép mint tömb, elemeire, tehát a pixelekre mutat. Ezt követően a dupla forcilusban feltölti az image mátrixot a pixelekkel, hasonló módon mint a háromszögmátrix esetében és megvizsgálja a piros színű pixeleket. A program végén a foglalt helyeket szabaddá tesszük.

Ezt a programot még sokat kell tanulmányozno de nagyon rajta vagyok az ügyön, a neurális háló működéséről pedig az előző feladatban írtam tehát ne,m szeretném ismételni magam.

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

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

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

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

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

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

        double value = (*p)(image);

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

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

4.7 Steve látása

Megoldás videó: https://www.youtube.com/watch?v=DX8dI04rWtk

Steve 2 módon láthat. Az egyik a lineOfSight, azaz azt a blokkot látja amin a kereszt van. A másik mód pedig egy körülötte lévő cuboidot használ. Ezt az xml módosításával kiterjeszthetjük az alap 3x3x3-masról akár 7x7x7-esre is. Itt fontos arra ügyelni hogy mely általa érzékelt blokkokkal szeretnénk dolgozni, ugyanis ezek egy tömbben vannak eltárolva és meg ios vannak számozva. Ezeknek a cuboidoknak a közepén van mindig Steve. Ezekre a tömbbeli elemekre tudunk hivatkozni tehát ki tudjuk íratni a tartalmukat, vagy egy if függvénnyel külön tudunk kiíratást csinálni hogy mondja el Steve hol és mit lát.

5. fejezet

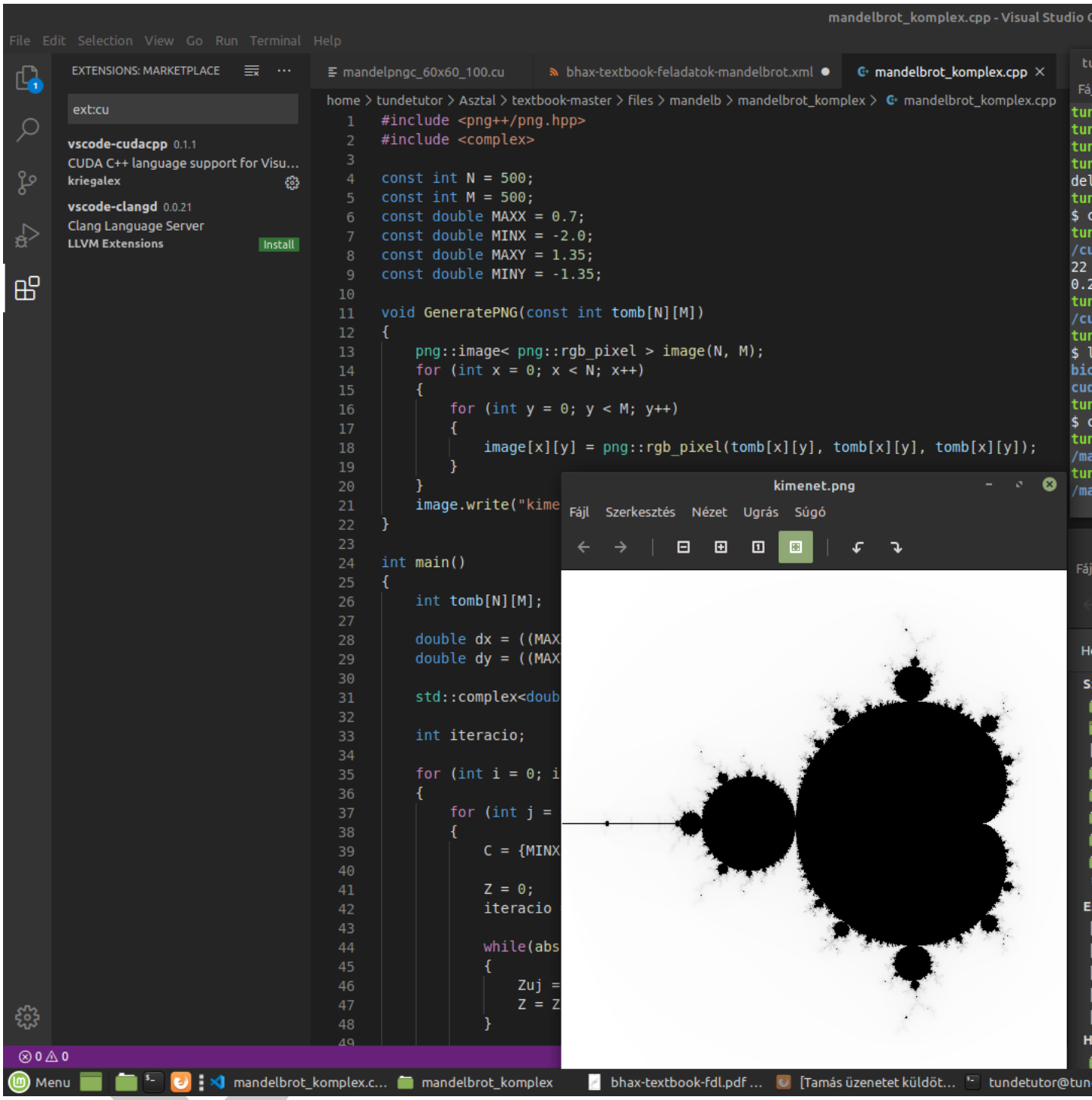
Helló, Mandelbrot!

5.1 A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

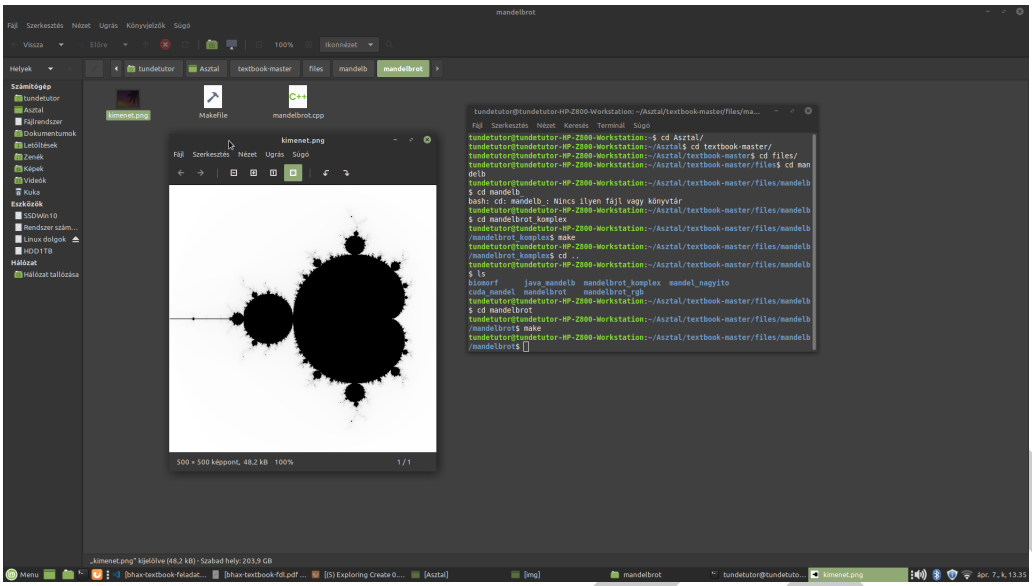
Megoldás videó: https://youtu.be/gvaqijHlRUs

Megoldás forrása: `bhax/attention_raising/CUDA/mandelpngt.c++` nevű állománya.



5.1. ábra. A Mandelbrot halmaz a komplex síkon

A Mandelbrot halmazt 1980-ban találta meg Benoit Mandelbrot a komplex számsíkon. Komplex számok azok a számok, amelyek körében válaszolni lehet az olyan egyébként értelmezhetetlen kérdésekre, hogy melyik az a két szám, amelyet összeszorozva -9-et kapunk, mert ez a szám például a 3i komplex szám.



A Mandelbrot halmazt úgy láthatjuk meg, hogy a sík origója középpontú 4 oldalhosszúságú négyzetbe lefektetünk egy, mondjuk 800x800-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<=n) képlet alapján úgy, hogy a c az éppen vizsgált rácspont. A z0 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. Nyilván nem tudunk végtelen sok z-t megvizsgálni, ezért csak véges sok z elemet nézünk meg minden rácsponthoz. Ha eközben nem lép ki a körből, akkor feketére színezzük, hogy az a c rácspont a halmaz része. (Színes meg úgy lesz a kép, hogy változatosan színezzük, például minél későbbi z-nél lép ki a körből, annál sötétebbre).

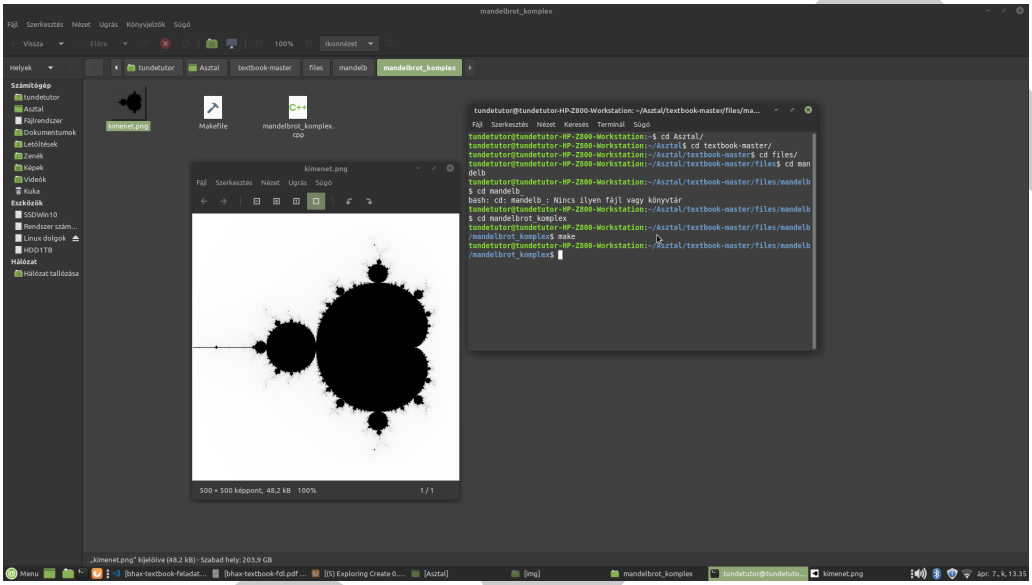
Ez végül is egy C alapú C++ program lett, mivel ahhoz hogy ténylegesen elkészüljön a png, ahhoz szükség van gy png könyvtárra viszont ez C-hez nem igazán található. A program úgy működik, hogy egy struktúrát használunk ahhoz, hogy komplex számokkal dolgozhassunk. A mandelbrot halmazt úgy tudjuk megjeleníteni, hogy a png-t mint egy mátrixot képzeljük el, és minden fog tartozni egy 2-256 közötti szám, ez határozza meg hogy az adott pixel nmilyen színű legyen. Ezt a számot úgy fogjuk megkapni hogy a ckiluson belüli iteráció száma lesz az, azaz hogy az adott értékkel ki tudunk-e kerülni a ciklusból vagy sem. Ha nem sikerül, az iteráció száma 256 lesz, tehát fekete pixelt kapunk, ebben az esetben arról van szó, hogy az adott érték nem tudott kikerülni a while ciklusból, tehát nem tudta elhagyni magát a mandelbrot halmazt sem, azaz annak eleme. Ha ez a szám 0, akkor fehér pixelt kapunk. A png-ben megjelennek sszürke pixelek is, ezek a 2 érték között valahol vannak. Összességében ezért van az, hogy a kapott png fekete és fehér

színekből áll. A png-t még lehetne cifrázni aszerint, hogy milyen "gyorsan" lépnek ki egy-egy értékek a while ciklusból, ilyen módon generálják a sok színes mandelbrot halmazt ábrázoló képet is.

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

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

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



Az előző feladathpz képest itt a legnagyobb különbség az az, hogy itt kihasználjuk a C++ előnyeit a C-vel szemben. Mint ahogy azt már említettem az előző egy C alapú program volt, viszont ehhez már használtunk C++ könyvtárakat ami nagyban segíti a programozók dolgát. Ilyen pl a komplex számokhoz tartozó könyvtár, itt már nem kellett struktúráz használnunk a komplex számok leírásához, a C++ ezt tudja magától is, illetve itt van külön függvénye a négyzetre emelésnek is. Ezeken kívül ugyanazt az algoritmust használjuk a mandelbrot halmaz megvalósításához mint az előző feladatban, tehát lényegi változás nincs, csak a részletekről lenne szó.

A Mandelbrot halmaz pontban vázolt ismert algoritmust valósítja meg a repó `bhax/attention_raising/Mandelbrot/3.1.2.cpp` nevű állománya.

```
// Verzio: 3.1.2.cpp
// Forditas:
// g++ 3.1.2.cpp -lpng -O3 -o 3.1.2
// Futtatas:
// ./3.1.2 mandel.png 1920 1080 2040  ↩
   -0.019473810573093663922605855987058021128l8  ↩
   -0.019473810572541341845642648422654019668 7  ↩
   0.798505756933826860155534177465597167611 1  ↩
   0.798505756934379196110285192844457924366
// ./3.1.2 mandel.png 1920 1080 1020  ↩
   0.412765541820958925534057470940751954913 1  ↩
   0.412765541824581805308014281763462349772 5  ↩
   0.213538705176874649138696327099751215428 1  ↩
   0.213538705180497528912653137922461610287 4
```

```
// Nyomtatas:
// a2ps 3.1.2.cpp -o 3.1.2.cpp.pdf -1 --line-numbers=1  --left-footer=" ↩
   BATF41 HAXOR STR34M" --right-footer="https://bhaxor.blog.hu/" --pro= ↩
   color
// ps2pdf 3.1.2.cpp.pdf 3.1.2.cpp.pdf.pdf
//
//
// 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/>.


#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;

}
```

5.3 Biomorfok

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

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

A biomorfokra (a Julia halmazokat rajzoló bug-os programjával) rátaláló Clifford Pickover azt hitte természeti törvényre bukkant: https://www.emis.de/journals/TJNSA/includes/files/articles/Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf (lásd a 2307. oldal aljától).

A különbség a Mandelbrot halmaz és 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 cc 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)
                {
                    iteracio = i;
                    break;
                }
            }
```

A biomorfos feladat szerintem nagyon érdekes volt. A biomorfokról érdemes tudni, hogy a Julia halamzok, Ezek a Mandelbrot halmaz részei, és számuk végtelen, ugye egy biomorfban a konstansunk nem változik, emiatt van hogy a Mandelbrot halmaz részei és számuk is végtelen mivel végtelen kontsans létezik. A program makefile-jában tudunk a kirajzolt képen módosítani, ezt több módon is próbálgattam. Az r-t lecseréltem 10-ről 5-re, ez a halmazra való ráközelítésen módosított, a minimum és maximum x és y-ok módosításával pedig a halmaz fekete része körüli "ráközelítést" lehetett módosítani, tehát ha ide nagyon pici értékeket írunk, gyakorlatilag semmi sem fog látszódni a halmazunkból. Ezen kívűl a felbontást nagyon megemeltem, hogy a halmaz szélén a színes részek jobban kirajzolódjanak szerintem ugyanus az benne a legösszetettebb textúra és ez a legérdekesebb is egyben. Csatolok néhány képet a a kirajzoltatott png-kről, úgy hogy látszódjanak a makefile-ban megadott értékek is.

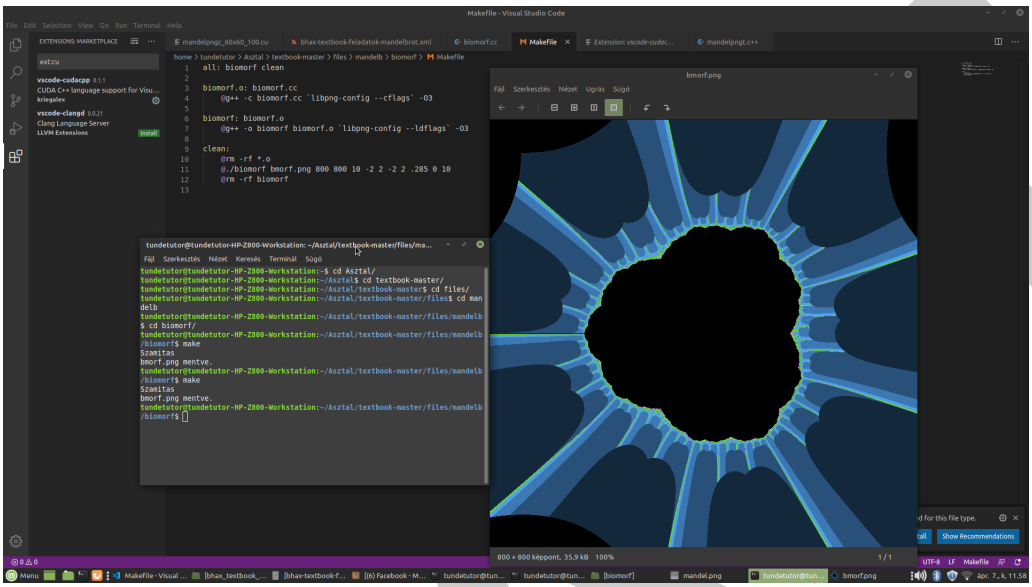
A bimorfos algoritmus pontos megismeréséhez ezt a cikket javasoljuk: https://www.emis.de/journals/-TJNSA/includes/files/articles/Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf. Az is jó gyakorlat, ha magából ebből a cikkből from scratch kódoljuk be a sajátunkat, de mi a királyi úton járva a korábbi Mandelbrot halmazt kiszámoló forrásunkat módosítjuk. Viszont a program változóinak elnevezését összhangba hozzuk a közlemény jelöléseivel:

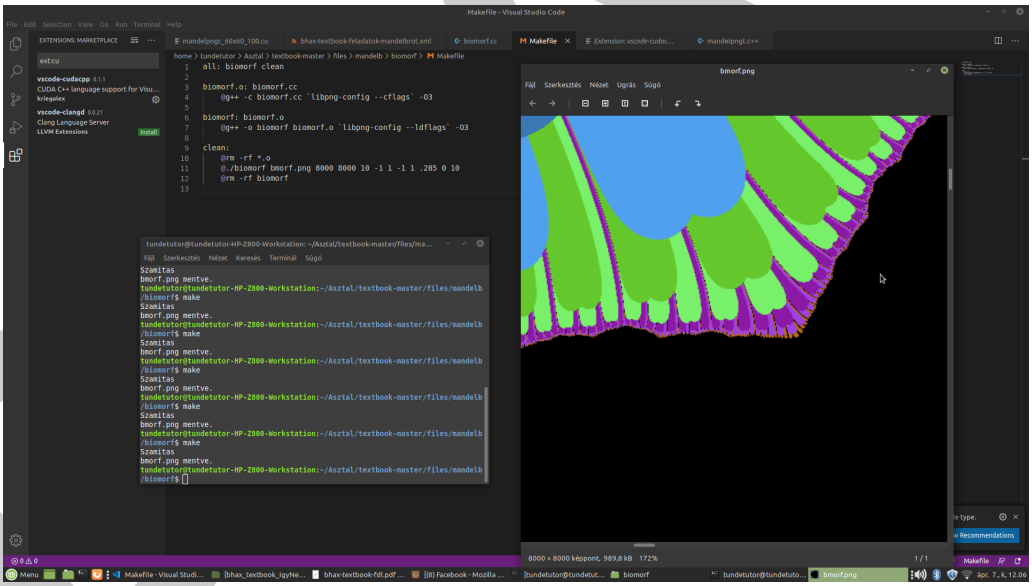
```
// Verzio: 3.1.3.cpp
// Forditas:
// g++ 3.1.3.cpp -lpng -O3 -o 3.1.3
// Futtatas:
// ./3.1.3 bmorf.png 800 800 10 -2 2 -2 2 .285 0 10
// Nyomtatas:
// a2ps 3.1.3.cpp -o 3.1.3.cpp.pdf -1 --line-numbers=1  --left-footer=" ↩
   BATF41 HAXOR STR34M" --right-footer="https://bhaxor.blog.hu/" --pro= ↩
   color
//
// BHAX Biomorphs
// Copyright (C) 2019
// Norbert Batfai, batfai.norbert@inf.unideb.hu
//
//  This program is free software: you can redistribute it and/or modify
```

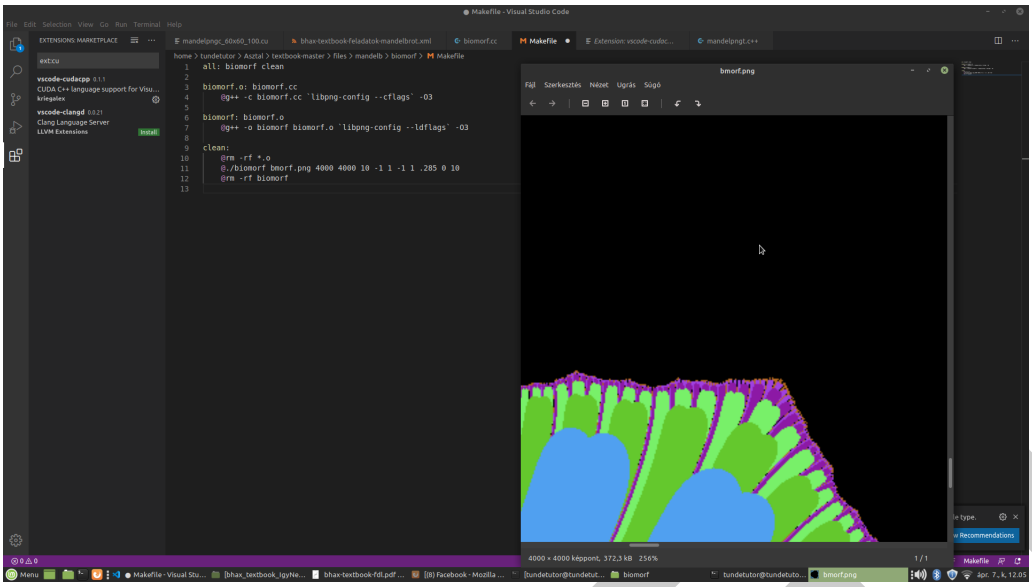
```
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation, either version 3 of the License, or
//  (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even 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
//
// https://youtu.be/IJMbgRzY76E
// See also https://www.emis.de/journals/TJNSA/includes/files/articles/ ↩
   Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf
//

#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;

}
```







5.4 A Mandelbrot halmaz CUDA megvalósítása

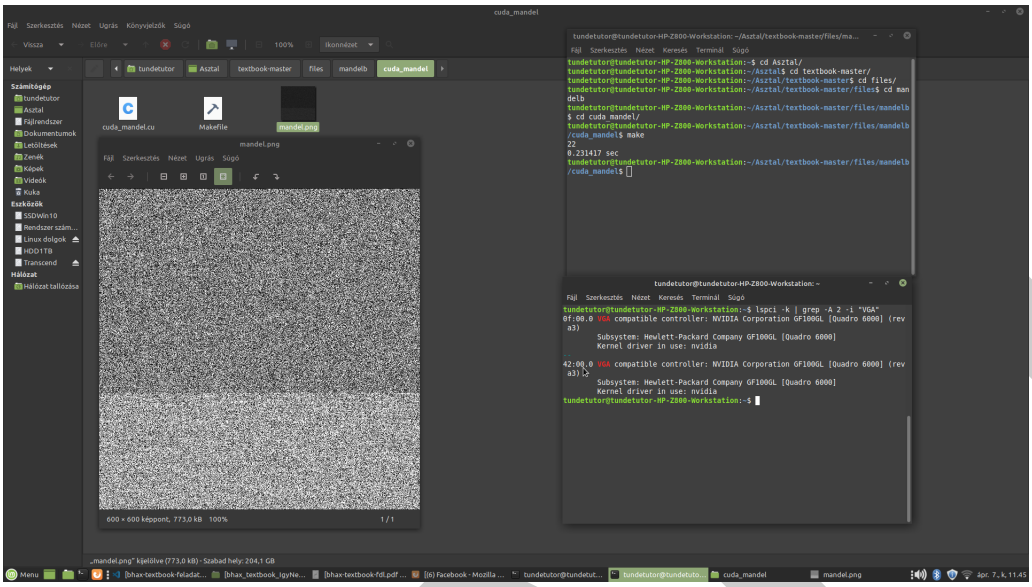
Ebben a feladatban ugyanazzal a mandelbrot halmaz algoritmussal dolgozunk mint az eddigi feladatokban is viszont itt van egy nagy különbség az eddigiekhez képest. Itt a feladat lényege hogy a CPU helyett a videókártya szmáítási kapacitását használjuk ki. Ahhoz hogy ezt meg tudjuk csinálni mindenképp szükségünk van egy NVIDIA kártyára mivel a CUDA az NVIDIA saját fejlesztése.

A CUDA lényege, hogy az NVIDIA kártyák rendelkeznek bizonyos mennyiségű CUDA magokkal, ezekkel párhuzamosan tudunk egyszerre több számítást is végezni. A CPU is rendelkezik magokkal amellyel párhuzamos számtásokat végezhetünk viszont nem annyival mint egy NVIDIA kártya és máshogyan is működnek, egy ilyen program esetében a CPU caósak egy magot használ. Ahhoz hogy rávegyük a videókártyánkat a számolásra mindenképp telepítenünk kell az NVIDIA Cuda Toolkitet. CUDA illesztőprogramokból egyébként találhatunk többet is, ugyanis mostanában kezd egyre jobban elterjedni hogy a deep learning miatt, ami nagy számítási kapacitás igényű folyamat. Véleményem szerint a CUDA a következő évkben eléggé elterjedté válhat ennek köszönhetően. Viszont most nekünk az alap NVIDIA Cuda Toolkit tökéletesen mgfelel.

Az elképzelés az lenne hogy a png minden pixelét más szál, más CUDA mag tudja kiszámolni. Ha belegondolunk ez hatalmas újítás ahhoz képest mintha ugyanezt a CPU-val szeretnénk megvaósítani. Nyílván ez a gyakorlatban nem teljesen így van hogy minden pixelre jut egy szál, de a CUDA-val ehhez már közelítünk, gondoljunk bele hogy ha egy erősebb GPU-val rendelkezünk amiben közel 1200 cuda mag van valószínűleg a számítás ideje jóval lecsökken a CPU-hoz képest. Azt is érdemes megemlítenünk, hogy hiába van sokmagos, sokszálas erős processzorunk ha egy folyamat csak 1 szálat fog használni, nem tudjuk kihasználni az egész teljesítményét míg a GPU esetén igen.

Én a CUDA-t a Blenderben való Cycles rendermotorral történő rendereléshez szoktam is használni. A Blender Cycles-e is ha nem rendelkezünk GPU-val processzorból fog számolni. A renderelés során egy ilyen technológia komoly napokat is megspórolhat nekünk, itt azért már érzi az ember higy valóban jelentősége van ennek.Egyébként 3D renderelés esetén hasonló dolog történik, mint a mandelbrot halmaznál, csak ott nem ezt az algoritmust használjuk, hanem a 3D felületet alkotó háromszögek normálvektoraiból lehet meghatározni egy-egy pixel milyen színű legyen.

Nálam megcsinál egy png-t viszont az közel sem a mandelbrot hamlmaz amit kiad. Nekem az a sejtésem hogy a hrsware konfigom a baj, én 2 db nvidia quadro kártyát használok viszont még nem sikerült SLI hidba kötnöm őket és szerintem ez okozza itt a gondot, viszont hamarosan tesztelem másik gépen is, hogy kiderüljön mi is itt a gond.



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

Megoldás forrása: `bhax/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 iteréció bejárta z_n komplex számokat!

Megoldás videó: Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a_mandelbrot_halmazzal.

Megoldás forrása:

Ebben a feladatban olyan programot kellett készíteni amivel már valódi GUI-t (Graphical User Interface) is tudunk használni. ahhoz, hogy egy ilyet lehessen készíteni szükségünk van QT library-ra. Ezzel egyszerű GUI-kat lehet létrehozni, a nagyító utauó programot az egérrel lehet haszbálni, illetve kap saját program-ablakot. Ez nagyon nagy újítás az eddigi csak terminálban futó programjainkhoz képest, és úgy érezhetjük, hogy ez egy mérföldkő. Viszont ehhez telepítenünk kell a QT-t. És igen... itt jöttek elő a problémák. Én Linux Mint rendszert használok, ami egyébként egy Ubuntu alapú, ahhoz nagyon közel álló rendszer, általában nem szokott baj lenni vele, amit Ubuntuval ,meg lehet csinálni azt Linux Minttel is de! Ezt a QT egyelőre nem tudtam telepíteni, még keresgetek, hogy ezt a problémát hogyan oldhatnám meg. Amikor rákerestem erre a gondra akkor szinte az első találatok között egy hasnoló kérdés volt, a kérdező pedig hozzátette hogy Linux Mintet használ. Viszont az ott adott válasz egyelőre nem oldotta meg az én problémámat sem szóval még folyamatban van a dolog.

5.6 Mandelbrot nagyító és utazó Java nyelven

Itt az előző programunk java átíratával van dolgunk. Ez szerencsére nem okozott az előzőhöz hasonló gondot, csak a jdk 8-at kellett telepítenünk,

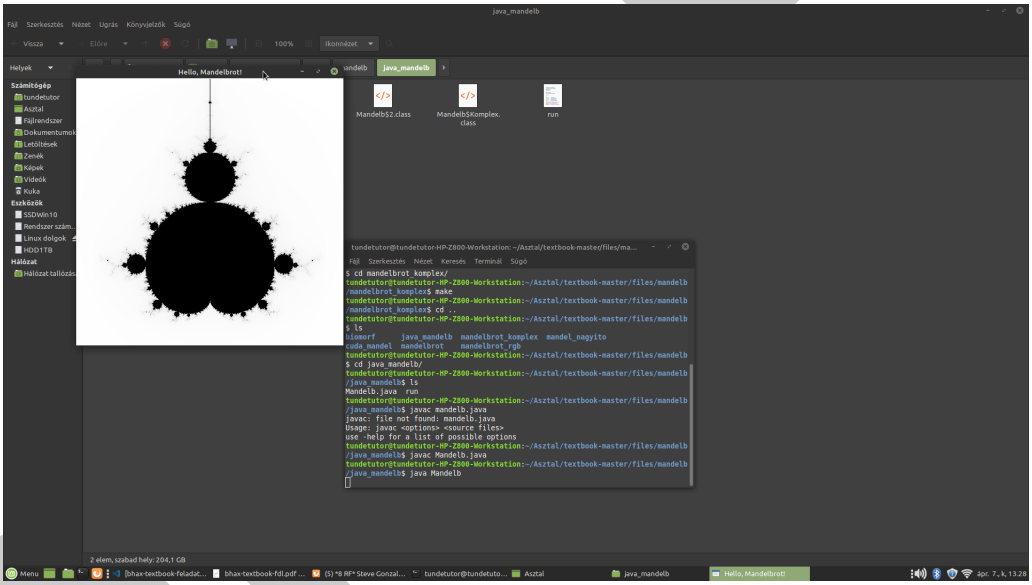
```
sudo apt-get install opnjdk-8-jdk
```

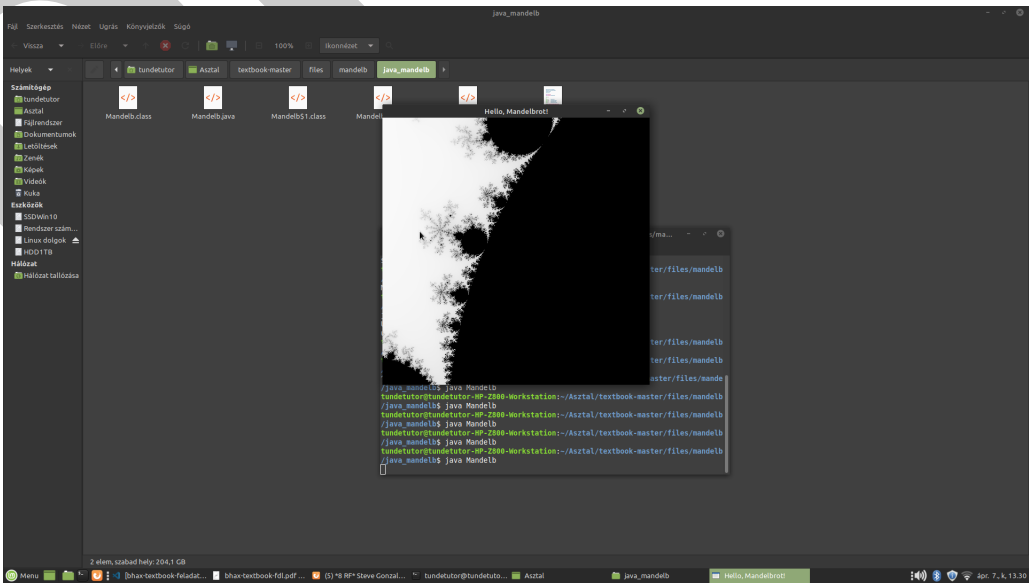
ez volt a csomagtárolóban is, tehát egy paranccsal ezt a gondot le is tudtuk. A futtatás sem okozott különösebb problémát. A futtatás így nézett ki:

```
javac mandelb.java
```

```
java mandelb
```

A C++ program fordítása és futtatása is elég macerás volt egyébként, ahhoz képest a a java-val nagyon könnyű dolgunk volt. Az eredményt pedig néhány screenshot formájában csatalom ide. Vicces volt kipróbálni hogy a nagyító meddig nagyít, egy idő után sajnos elértem azt hogy elpixeleződött a kép.





Megoldás videó: https://youtu.be/Ui3B6IJnssY, 4:27-től. Illetve https://bhaxor.blog.hu/2018/09/02/ismerkedes_a

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#id570518

5.7 Steve felszalad a láváig

Ebben a programban Steve első dolga, hogy elszadjon a láváig, ezt move, jumpmove párosokkal teszi egy while cikluson belül amineka kilépési feltételében szerepel, az általa érzékelt kockákból ha az előtte és fölötte előtte lévő kockákban "flowing_lava" vgay "lava szerepel". Ha ez megtörténik Steve leszalad és elkezdi csigában gyűjteni a virágokat. A program lefutása az alábbi videóban látható.

Megoldás videó: https://www.youtube.com/watch?v=zO6cNp8L4-Q

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 videó:

A következő java kód Bátfai Norbert Tanár Úr kódja, a polártranszformációs algoritmus segítségével véletlenszám generálást tudunk végrehajtani. Ez is, a fejezetben objektum orientált programokkal foglalkozunk. A java alapvetőleg objektum orientált nyelv, ahhoz hogy tudjuk a programot futtatni, ugyanúgy kell elneveznünk a .java fájlt is mint ami a benne létrehozott osztály neve. Nézzük is meg ez a java program hogyan fut.

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

        nincsTarolt = true;

    }
    public double kovetkezo() {
        if(nincsTarolt) {
            double u1, u2, v1, v2, w;
            do {
                u1 = Math.random();
                u2 = Math.random();

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

                w = v1*v1 + v2*v2;
```

```
                } while(w > 1);

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

                tarolt = r*v2;
                nincsTarolt = !nincsTarolt;

                return r*v1;

            } else {
                nincsTarolt = !nincsTarolt;
                return tarolt;
            }
        }

        public static void main(String[] args) {
            PolarGenerator g = new PolarGenerator();
            for(int i=0; i<10; ++i)
                System.out.println(g.kovetkezo());
        }
    }
```

Fordítás és futtatás után a következő gyönyörű eredményeket kaptuk:

```
0.500010678804642
0.7466265624656746
1.0803216647807399
-0.32064099460970763
-2.5477451034196923
0.6811730798994344
-0.44975361547907916
-0.9422083605110528
0.490151771519700 96
2.058535110772562
```

6.2 LZW

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

A feladatcsokorban ez volt szerintem a legfontosabb feladat és a többi is főleg erre épített. Ezt a programot, Bátfau Norbert Tanár Úr videói alapján írtam meg. Most ezt a binárs fát a nulláról keztük el írni, ez a "from scratch" fánk. Viszont felmerül a kérdés, mi is maga a bináris fa?

A bináris fák adatstruktrák. A félév során az Adatszerkezetek és algoritmusok nevű tantárgy is sokat foglalkozik ezzel a fajta adatstruktúrával. A bináris fákban különböző adattípusokat tudunk tárolni. Az egyik legegyszerűbb talán amiben csak egész számokat tárolunk, ezen keresztül bemutatom a működését. Ha

adott egy egész számokat tartalmazó rendezett tömbünk, annak elemeit bináris fában is tudjuk tárolni. Ezzel lényegesen lerövidíthetjük az adathalmazunkban történő keresés idejét. A bináris fáknak van 1 db gyökere, és 2 db gyermeke a gyökérnek, egy jobb és egy baloldali. Ha az egész számos példánál maradunk, akkor a gyökér a tömbként ábrázolt adathalmazban a középső indexű elem, a bal oldali gyermeke a gyökérnél kisebb értékek között a középső elem, a jobboldalinál pedig az annál nagyobbak közti középső elem és így tovább minden gyermekkel és azok gyermekeivel. Tehát a bináris fánk szintenként duplázódik elemszámot tekintve, ez azt jelenti, hogy egy levélemet (olyan csomópont amely nem rendelkezik gyermekekkel) könnyedén elérhetővé válik. Ha van plkb 2500 adatunk, látható, hogy egy keresett elem maximum 11 lépésből elérhető, ez adja az algoritmus gyorsaságát.

A programunkban tárgyalt bináris fa első sorban 0 és 1 értékekkel dolgozik, ezeket úgy rendezzük el a fában, hogy ha a vizsgált elem 0 akkor balra kerül, ha 1 akkor pedig jobbra. A bemenő adatokat vizsgáljuk és ha nem létezik még olyan csomópont mint a bemenő adat akkor létrehozunk egyet és a mutatót visszaállítjuk a gyökérre, ha már létezik akkor pedig ráállítjuk a mutatót. Ilyen módon tudjuk magát a fát felépíteni.

A videóban ami alapján én is írtam a programot, Bátfai Norbert z alábbi bitsorral tesztelte a programját, így én is úgy döntöttem azzal tesztelem, hogy biztos legyek annak működésében. Szerencsére jól lefutott a program és ezt a kimenetet kaptam:

```
---------0 3 0
------0 2 0
---------1 3 0
---/ 1 0
---------------0 5 0
------------0 4 0
---------0 3 0
------1 2 0
---------1 3 0
------------1 4 0
```

6.3 Fabejárás

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

Megoldás videó:

Ehhez a feladathoz is a "from sracth" programot használtam. Eza kód alapvetőleg inorder bejárást használ, azaz először a baloldalt vizsgálja, aztán az adott csomópont gyökerét majd annak a jobb oldali gyermekét, tehát ha inorder kiíratást végzünk magának a fának a gyökere középen fog megjelenni. Nézzünk is meg egy kódcsipetet és az ilyen módonbejárt fa kimetetét is.

```
            template <typename ValueType>
void BinTree<ValueType>::print(Node *node, std::ostream & os)
{
    if(node)
    {
        ++depth;
```

```
        print(node -> leftChild(), os);

        for(int i = 0; i < depth; ++i)
        {
            os << "---";
        }

        os << node -> getValue() << ' ' << depth << ' ' << node -> getCount ↩
           () << ' ' << std::endl;

        print(node -> rightChild(), os);
        --depth;
    }
}
            }
```

```
            ---------0 3 0
            ------0 2 0
            ---------1 3 0
            ---/ 1 0
            ---------------0 5 0
            ------------0 4 0
            ---------0 3 0
            ------1 2 0
            ---------1 3 0
            ------------1 4 0
```

A következő bejárás a preorder bejárás lesz, ez azt jelenti, hogy először mindig a gyökeret vizsgáljuk, azt követően a előzör a baloldalt, majd a jobb oldalt. Itt is megmutatom a kódcsipetet és a kimenetet is, ebben az esetben a teljes fa gyökere lesz az első kiíratott elem.

```
            template <typename ValueType>
void BinTree<ValueType>::print(Node *node, std::ostream & os)
{
    if(node)
    {
        ++depth;

        for(int i = 0; i < depth; ++i)
        {
            os << "---";
        }

        os << node -> getValue() << ' ' << depth << ' ' << node -> getCount ↩
           () << ' ' << std::endl;

        print(node -> leftChild(), os);
```

```
        print(node -> rightChild(), os);
        --depth;
    }
}
            }
```

```
            ---/ 1 0
            ------0 2 0
            ---------0 3 0
            ---------1 3 0
            ------1 2 0
            ---------0 3 0
            ------------0 4 0
            ---------------0 5 0
            ---------1 3 0
            ------------1 4 0
```

Végül nézzzük meg a postorder bejárást is. Ennél a bejárásnál az előzőhöz képest annyi különbség van, hogy a yökeret vizsgáljuk utolára, de a balról-jobbra bejárás megmarad. Ekkor az utolsó kiíratott elem lesz magának a fának a gyökere.

```
            template <typename ValueType>
void BinTree<ValueType>::print(Node *node, std::ostream & os)
{
    if(node)
    {
        ++depth;

        print(node -> leftChild(), os);

        print(node -> rightChild(), os);

        for(int i = 0; i < depth; ++i)
        {
            os << "---";
        }

        os << node -> getValue() << ' ' << depth << ' ' << node -> getCount ↩
           () << ' ' << std::endl;


        --depth;
    }
}
            }
```

```
---------0 3 0
---------1 3 0
------0 2 0
---------------0 5 0
------------0 4 0
---------0 3 0
------------1 4 0
---------1 3 0
------1 2 0
---/ 1 0
```

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 videó:

Ehhez a feladathoz egy másik bináris fa programot használtam, a z3a7.cpp névvel elátottat, ebben a programban a gyökér alapból tagja a fának, tehát kompozícióban van azzal. Ezt az alábbi apró kódcsipetben megtekinthetjük. És a következő feladatban pedig megmuatom, hogyan lehet ugyanezt a programot úgy módosítani, hogy a gyökérből mutató váljon.

```
        LZWBinFa ():fa (&gyoker)
{
}
 ~LZWBinFa ()
{
  szabadit (gyoker.egyesGyermek ());
  szabadit (gyoker.nullasGyermek ());
}
```

Egyébként ezt és a következő programot is lefutattam a COVID19 vírus genomjával, mint bemenet és ugyanazt kaptam eredményül. Ezt csak mint érdekesség gondoltam megjegyzem, persze ennek a kimenetét nem csatolnám mivel annál egy 41 mélységű fáról beszélünk.

6.5 Mutató a gyökér

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

Megoldás videó:

Az általunk használt "from scratch" fás programban a gyökér eleve mutató ezért kerestem egy násik forrást ahol nem az, ez a z3a7.cpp, és úgy gondoltam érdekesebb ha ebben a feladatban ebből a forrásból készítünk olyat ahol a gyökér mutató, mintha megmutatnám azt amelyiket alapból úgy írtuk.

```
            LZWBinFa ()
{
gyoker = new Csomopont('/');
fa = gyoker;
}
~LZWBinFa ()
{
    szabadit (gyoker->egyesGyermek ());
    szabadit (gyoker->nullasGyermek ());
}
```

Itt ebben a kódcsipetben látszik, hogy a gyökérből az előzőhöz képest mutatót készítettünk, a gyökér is már új csomópont. Persze a kód többi részét is keleltt módosítanunk, de szerintem itt az LZWBinFa osztályban látszik a legjobba a különbség.

6.6 Mozgató szemantika

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

Megoldás videó:

Ehhez a feladathoz azt a programot fejlesztük tovább amit a 2. feladathoz használtunk. Itt mindenképp szóba kell hoznunk, hogy ha a fát szeretnénk mozgatni, akkor ezt csak úgy tehetjük meg a `BinTree` ősosztályba beágyazott `Node` osztállyal együtt rekurzívan hajtjuk végre. Egyébként készíttetünk másoló és másoló értékadó konstruktort is de most azok kódcsipetét nem teszem bele a könyvbe, lehet a későbbiek során még belekerül, de most a mozgató szemantikán volt a hangsúly. Nézzünk egy kimenetet is, mivel látható a kódcsipetben hogy a mozgató konstruktorba tesztelés cljából tettünk kiíratást is.

```
    BinTree & operator=(const BinTree & old)
{
    std::cout << "BT copy assign ctor, masolo ertekadas" << std::endl;

    BinTree tmp{old};
    std::swap(*this, tmp);
    return *this;
}
BinTree(BinTree && old){
    std::cout << "BT move ctor, mozgato " << std::endl;

    root = nullptr;
```

```
        *this = std::move(old);


    }
    BinTree & operator=(BinTree && old){
        std::cout << "BT move assign ctor, mozgato ertekadas" << std::endl;

        std::swap(old.root, root);
        std::swap(old.treep, treep);

        return *this;
    }
```

```
            BT ctor
            ---------2 3 0
            ------5 2 0
            ---------7 3 0
            ---8 1 0
            ------9 2 0

            BT ctor
            ---------0 3 0
            ------0 2 0
            ---/ 1 0
            BT  copy ctor, masolo konstruktor
            BT ctor
            ***
            BT copy assign ctor, masolo ertekadas
            BT  copy ctor, masolo konstruktor
            BT move ctor, mozgato
            BT move assign ctor, mozgato ertekadas
            BT move assign ctor, mozgato ertekadas
            BT move assign ctor, mozgato ertekadas
            BT dtor
            BT dtor
            ***
            BT move ctor, mozgato
            BT move assign ctor, mozgato ertekadas
            BT dtor
            BT dtor
            BT dtor
            BT dtor
            BT dtor
```

6.7 Steve szemüvege

Megoldás videó: https://www.youtube.com/watch?v=DX8dI04rWtk

Steve 2 módon láthat. Az egyik a lineOfSight, azaz azt a blokkot látja amin a kereszt van. A másik mód pedig egy körülötte lévő cuboidot használ. Ezt az xml módosításával kiterjeszthetjük az alap 3x3x3-masról akár 7x7x7-esre is. Itt fontos arra ügyelni hogy mely általa érzékelt blokkokkal szeretnénk dolgozni, ugyanis ezek egy tömbben vannak eltárolva és meg ios vannak számozva. Ezeknek a cuboidoknak a közepén van mindig Steve. Ezekre a tömbbeli elemekre tudunk hivatkozni tehát ki tudjuk íratni a tartalmukat, vagy egy if függvénnyel külön tudunk kiíratást csinálni hogy mondja el Steve hol és mit lát.

7. fejezet

Helló, Conway!

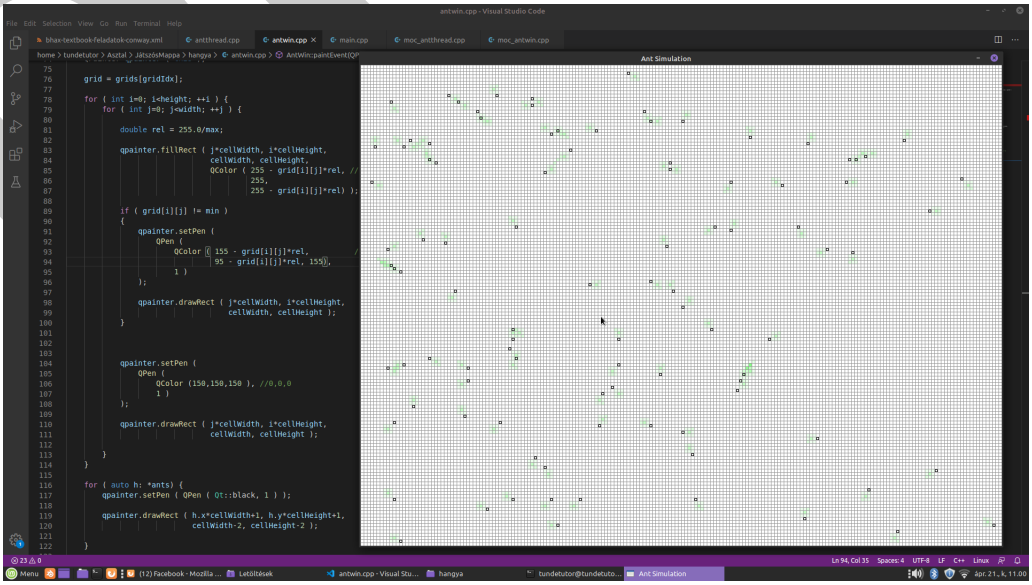
8.1 Hangyaszimulációk

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:

Ebben a feladatban egy hangyaszimulációs programot vizsgálunk. Ahhoz, hogy megértsük működésének alapjait, ismernünk kell a valódi hangyák viselkedését is. Nekem egy rövid ideig, volt egy kb 30 hangyából álló apró hangyakolóniám, sajnos a felelőtlenségem miatt elpusztultak (nem megfelő ételt adtam nekik) de addig is picit beleláttam, hogy hogyan élnek együtt. A hangyák úgynevezett kolónitudattal élnek, egy-egy példánynak esélye sincs egyedül túlélni, nagyon szorosan egymásra vannak utalva, gyakorlatilag a kolóniára úgy kell tekintenünk mint egy állatra, mikor ettettem a hangyáimat akkor is ezt a szempontot kellett szem előtt tartani, hogy úgy adjak nekik élelmet, hogy az a teljes bolynak elég legyen. A valóságtól való első számú különbség a szimulációs programban, hogy itt a hangyabolyban nincs királynő, pedig egyébként ténylegesen ő a hangyaboly legfontosabb eleme, tehát ha ő elpusztul a boly nem tud túlélni, elsősorban amiatt hogy csak a királynő tud petéket lerakni, de egyébként sem tudnak királynő nélkül élni.



Maga program egyébként 5 db C++ kódból és 4 db headerből áll. Nem fogom mindet részletesen bemutatni, mert nekem is feleslegesen sok munka lenne és az olvasót sem szeretném untatni, szóval a számomra érdekesebb részekről fogok írni. Mivel GUI-cal szeretnénk megjeleníteni a programot, ezrt ismét a Qt-t hívtuk segítségül, végül sikerült ezt a Linux Mint-emen is megoldani és a qmake make parancsok majd a "myrmecologist" futtatása után szépe kirajzolódott a hangyabolyom. Az ablakban a következőket láthatjuk: a pici mozgó négyzetek nyílván a hanygáink, ezek zöld szálakat húznak maguk után, ezek a maguk mögött hagyott feromon nyomok, ezek alapján tudnak megfelően mozogni, és láthatóan egy nagy rácshálót, ami az egész programablakot beteríti. Ez a rács szerintem rendkívűl zavarja a szemet és az ember nem szívesen nézi így perceken át a hangyákat szóval én azt világosszürkére állítottam, látható az alábbi, antwin.cpp-ből kódcsipetben, a QPaintEvent felel ugyanis a megjelenítésért.

```
            void AntWin::paintEvent ( QPaintEvent* )
{
    QPainter qpainter ( this );

    grid = grids[gridIdx];

    for ( int i=0; i<height; ++i ) {
        for ( int j=0; j<width; ++j ) {

            double rel = 255.0/max;

            qpainter.fillRect ( j*cellWidth, i*cellHeight,
                                cellWidth, cellHeight,
                                QColor ( 255 - grid[i][j]*rel, //255 volt  ↩
                                   mindhárom
                                         255,
                                         255 - grid[i][j]*rel) );

            if ( grid[i][j] != min )
            {
                qpainter.setPen (
                    QPen (
                        QColor ( 255 - grid[i][j]*rel,          //255
                                 255 - grid[i][j]*rel, 255),
                        1 )
                );

                qpainter.drawRect ( j*cellWidth, i*cellHeight,
                                    cellWidth, cellHeight );
            }



            qpainter.setPen (
                QPen (
                    QColor (150,150,150 ), //0,0,0
                    1 )
            );
```

```
            qpainter.drawRect ( j*cellWidth, i*cellHeight,
                                cellWidth, cellHeight );

        }
    }
```

Az antwin.cpp-ben ebben a kódrészletben van egyébként nagyon sok dolog ami a GUI megjelenítésért felel. Mos más dolgunk nem igazán volt a Qt-vel, nincs mouseMove Event stb. Mivel ez egy szimuláció, nyílván nem interaktív.

A programunkban tehát jelenleg csak dolgozókkal találkozunk és az ő mozgásukat tudjuk megfigyelni. Ha elindítottuk a programot, akkor láthatjuk, hogy különböző pontokon jelennek meg a hangyáink, ahhoz hogy így induljon el a program és ne egy pontban jelenjen meg az összes hangya, a main.cpp-ben a qsrand függvény segítségére volt szükség, erről teszek is ide egy kódcsipetet. Fontos ugye, hogy ne egy helyen jöjjenek létre.

```
            qsrand ( QDateTime::currentMSecsSinceEpoch() );
```

8.2 Java életjáték

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

Megoldás videó:

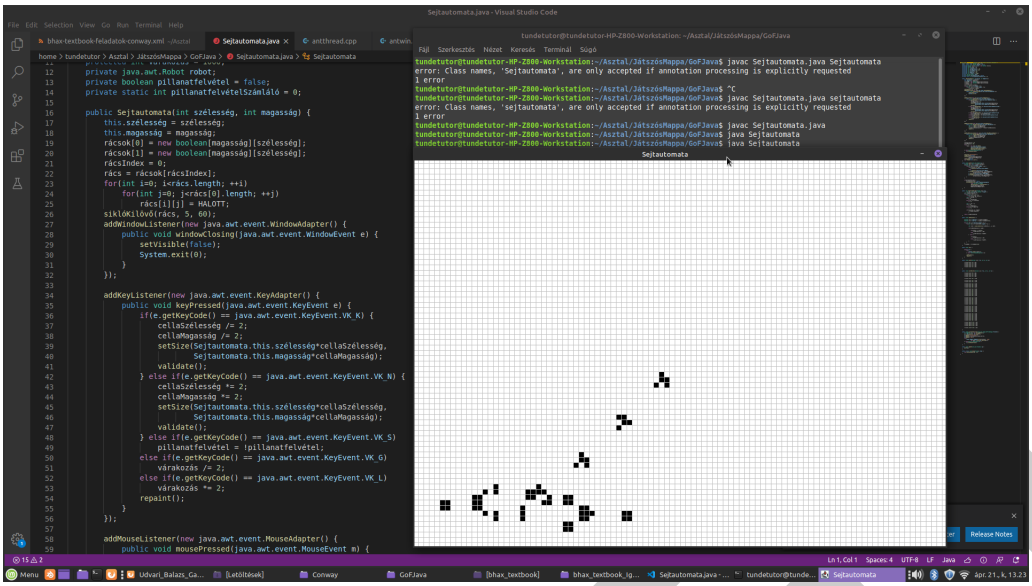
Megoldás forrása:

A játékot John Horton Conway. Cambridge-i matematikus alkotta meg. A játék egy sejtautomata. A lényege az hogy, szimulálja a sejtek életét, egyszerű szabályok meghatározásával. Maga a "játékmenet" egyébként eléggé passzív, a játkosnak annyi dolga van, hogy az első generációni sejtet ő helyezi el a rácshálón majd figyeli mi történik velük.

Mivel a következő feladat is ugyanerről az életjátékról szól csak C++ megvalósításban arra gondoltam, hogy a játékról magáról itt írok bővebben, a következő feladatban pedig inkább a kódot vizsgáljuk. Szóval ismerjük is meg a szabályokat!

A sejtek a rácshálóban egy 8 rácspontnyi távolságban tudnak "érzékeéni". Ez azért fontos mert egy sejt önmagában elpusztul, szüksége van más sejtekre is a környezetében, ezeknek a száma 2 és 3 lehet, a közvetlen szomszédságában. Tehát 1 sejt 1 szomszéddal elpusztul de ha 3-nál több szomszédja van, akkor már "túlszaporodtak" és megint elpusztul az a sejt amelynek több szomszédja van. Új sejt születik akkor ha egy sejtnek pontosan 3 szomszédja van. a játék körökre, generációkra bontható, és minden generáció változásait vizsgálhatjuk. A játék szabályai ilyen egyszerűek mégis nagyon összetett dolgokkal találkozhatunk.

A játék során kialakulhatnak úgynevezett stabil alakzatok, ilyen például egy egyszerű négyzet amely 4 db sejtből áll, ekkor minden sejtnek 3 szomszédja van tehát túlélik a kört. Az egyik leghíresebb azakzat az a "sikló" amely átlósan tud mozogni is, iletve a vízszintesen mozgó űrhajó. Ezek olyan alakzatok amelyek idővel "mozgásuk során" önmagukba alakulnak vissza. Csatolok is róluk ided egy képet, arról mikor futattam ezt a programt:



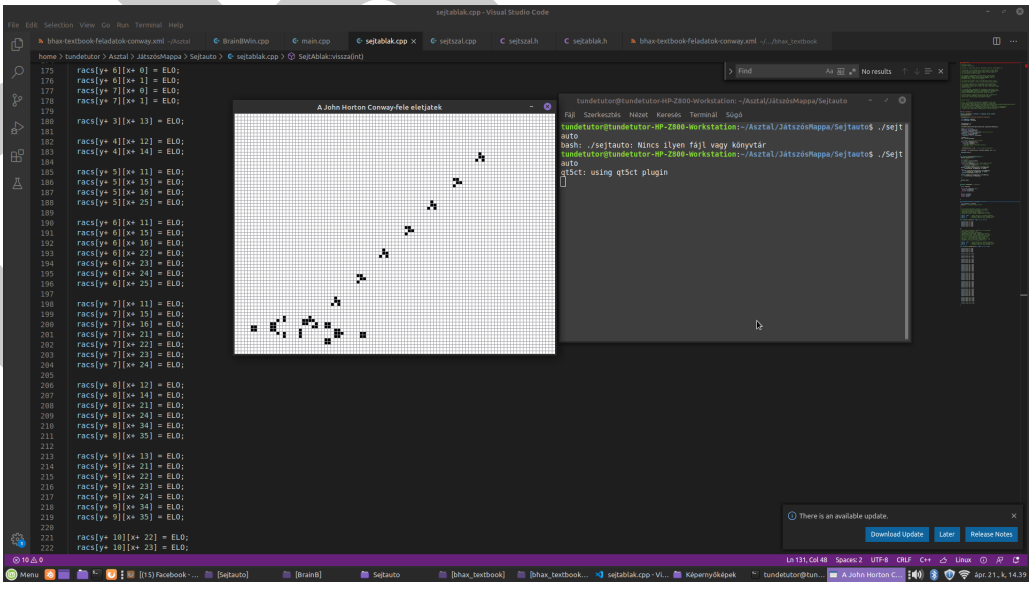
Fontosnak tartom mgejegyezni, hogy olyan lakzatok is kirajzolódhatnak a játék futása során amelyek természetes jelenségekre hasonlítanak. Ilyen pl.: egy furcsa tintapaca szerű nagyjából összefüggő alak, az ilyen jellegű mintákat sok helyen tudják hasznosítani, ilyen lehet mondjuk a barlagok generálása is, aminél ezt egyébként nagyon gyakran alkalmazzák is.

Na és ha már java, nem hagyhatom ki hogy megmutassam miket találtam, a youtube-on. Több minecraftos megvalósítása is létezik ennek a játéknak, mégpedig úgy, hogy a 3D-s térben zajlanak ezek az események.

Minecraft 3D https://www.youtube.com/watch?v=wNypW-aSCmE

8.3 Qt C++ életjáték

Most Qt C++-ban!



Megoldás forrása:

Ez a Qt C++ megvalósítás egy egészen összetett, objetumorientált progam. Először a sejtablak.h-val kezdeném. Ebben a programban, rögzítjük a változókat, szélességet, magasságot, illetve meghatározzuk, hogy

eg ysejt lehet élő vagy halott sejt.

```
        class SejtAblak : public QMainWindow
{
  Q_OBJECT

public:
  SejtAblak(int szelesseg = 100, int magassag = 75, QWidget *parent = 0);

  ~SejtAblak();
  // Egy sejt lehet elo
  static const bool ELO = true;
  // vagy halott
  static const bool HALOTT = false;
  void vissza(int racsIndex);
```

Eztán találunk egy protected részt, itt van egy ***rácsok mutató, ez azért, kell mert egyszerre 2 db mátrixra szeretnénk rámutatni, ez amiatt kell, mert amikor lépünk generációt, nem kerülhet oda élő sejt ahol halott van. Ezen kívűl ami még fontos, hogy itt találjuk a pintEventet, és a void siklo illetve a void sikloKilovo-t is. a sikloKilovo azért fontos mert ebben a verzióban nem a felhasználóra van bízva a játék kezdete, hanem előre defeniált.

```
protected:

  bool ***racsok;

  bool **racs;

  int racsIndex;

  int cellaSzelesseg;
  int cellaMagassag;

  int szelesseg;
  int magassag;
  void paintEvent(QPaintEvent*);
  void siklo(bool **racs, int x, int y);
  void sikloKilovo(bool **racs, int x, int y);
```

Tekintsük meg a sejtablak.cpp--t is! Itt több függvényt is találunk, az első a sejtAblak, itt hozzuk létre magát a programablakot, szélességgel, magassággal, a cellák szélességével és magasságával együtt, illetve ebben létrehozzuk a mátrixainkat is. Illetve leszögezzük, hogy a kiinduló programban minden cella halott. Ezután pedig start függvény tudja indítani a programot.

Az alábbi kódcsipetben pedig megtaláljuk a paintEvent-et. Itt is mátrix-szal dolgozunk.

```
void SejtAblak::paintEvent(QPaintEvent*) {
  QPainter qpainter(this);

  // Az aktualis
  bool **racs = racsok[racsIndex];
  // racsot rajzoljuk ki:
  for(int i=0; i<magassag; ++i) { // vegig lepked a sorokon
    for(int j=0; j<szelesseg; ++j) { // s az oszlopok
      // Sejt cella kirajzolasa
      if(racs[i][j] == ELO) //feltölti az adott színnel
  qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
        cellaSzelesseg, cellaMagassag, Qt::black);
  else
    qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
          cellaSzelesseg, cellaMagassag, Qt::white);
    qpainter.setPen(QPen(Qt::gray, 1));

  qpainter.drawRect(j*cellaSzelesseg, i*cellaMagassag,
        cellaSzelesseg, cellaMagassag);
    }
  }

  qpainter.end();
}
```

Itt ebben a kódcsipetben pedig a folyamatos frissülésért felelős függvényt találjuk:

```
        void SejtAblak::vissza(int racsIndex) //frissít
{
  this->racsIndex = racsIndex;
  update(); // ő meghívja a paint eventet
}
```

Ebben a részben pedig az egész sejtszal.cpp-t találjuk. Ennek érdekessége, hogy az előző feladatban említett szabályok vannak leírva C++ nyelven.

```
            #include "sejtszal.h"

SejtSzal::SejtSzal(bool ***racsok, int szelesseg, int magassag, int  ↩
   varakozas, SejtAblak *sejtAblak)
{
    this->racsok = racsok;
    this->szelesseg = szelesseg;
```

```
    this->magassag = magassag;
    this->varakozas = varakozas;
    this->sejtAblak = sejtAblak;

    racsIndex = 0;
}

/**
   * Az kerdezett allapotban levo nyolcszomszedok szama.
   *
   * @param   racs    a sejtter racs
   * @param   sor     a racs vizsgalt sora
   * @param   oszlop  a racs vizsgalt oszlopa
   * @param   allapor a nyolcszomszedok vizsgalt allapota
   * @return int a kerdezett allapotbeli nyolcszomszedok szama.
   */
int SejtSzal::szomszedokSzama(bool **racs,
                              int sor, int oszlop, bool allapot) {
    int allapotuSzomszed = 0;
    // A nyolcszomszedok vegigzongorazasa:
    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)
            // A vizsgalt sejtet magat kihagyva:
            if(!((i==0) && (j==0))) {
        // A sejtterbol szelenek szomszedai
        // a szembe oldalakon ("periodikus hatarfeltetel")
        int o = oszlop + j;
        if(o < 0)
            o = szelesseg-1;
        else if(o >= szelesseg)
            o = 0;

        int s = sor + i;
        if(s < 0)
            s = magassag-1;
        else if(s >= magassag)
            s = 0;

        if(racs[s][o] == allapot)
            ++allapotuSzomszed;
    }

    return allapotuSzomszed;
}

/**
 * A sejtter idobeli fejlodese a John H. Conway fele
 * eletjatek sejtautomata szabalyai alapjan tortenik.
 * A szabalyok reszletes ismerteteset lasd peldaul a
 * [MATEK JATEK] hivatkozasban (Csakany Bela: Diszkret
```

```
 * matematikai jatekok. Polygon, Szeged 1998. 171. oldal.)
 */
void SejtSzal::idoFejlodes() {

    bool **racsElotte = racsok[racsIndex];
    bool **racsUtana = racsok[(racsIndex+1)%2];

    for(int i=0; i<magassag; ++i) { // sorok
        for(int j=0; j<szelesseg; ++j) { // oszlopok

            int elok = szomszedokSzama(racsElotte, i, j, SejtAblak::ELO);

            if(racsElotte[i][j] == SejtAblak::ELO) {
                /* Elo elo marad, ha ketto vagy harom elo
             szomszedja van, kulonben halott lesz. */
                if(elok==2 || elok==3)
                    racsUtana[i][j] = SejtAblak::ELO;
                else
                    racsUtana[i][j] = SejtAblak::HALOTT;
            }  else {
                /* Halott halott marad, ha harom elo
             szomszedja van, kulonben elo lesz. */
                if(elok==3)
                    racsUtana[i][j] = SejtAblak::ELO;
                else
                    racsUtana[i][j] = SejtAblak::HALOTT;
            }
        }
    }
    racsIndex = (racsIndex+1)%2;
}


/** A sejtter idobeli fejlodese. */
void SejtSzal::run()
{
    while(true) {
        QThread::msleep(varakozas);
        idoFejlodes();
        sejtAblak->vissza(racsIndex);
    }

}

SejtSzal::~SejtSzal()
{
}
```

8.4 BrainB Benchmark

Megoldás videó:

Megoldás forrása:

Aki, a félév elején bekapcsolódott a TP vadászatba, annak ismerős lesz ez a program. Ez a BrainB program Bátfai Norbert Tanár Úr illetve más fejlesztők által megírt program. Maga a program egy játék, ami gyakorlatilag egy pszichológiai teszt. A lényeg a következő: Van egy felületünk, amin félig átlátszó négyzetek jelennek meg, ezek közül van egy kibálasztott különleges négyzet, amit Samu entrophy-nak nevezünk. A játék 10 perces, 10 per után leáll magától. A 10 perc leforgása alatt a játékosnak 1 feladata van, mégpedig, hogy a Samu entrophy-n tartsa az egerét lenyomott egérgombbal. Elsőre ez nem is tűnik olyan nehéz feladatnak viszont találkozunk nehezítő tényezőkkel. Ugyanolyan kinézetű négyzetek jelennek meg sorra mint a Samu entrophy, és mindegyikőjük apró mozgásokat végez. Említettem, hogy félig átlátszóak ezek a négyzetek tehát valameddig lehet is követni a dolgot, de egy idő után szinte teljesen elveszítjük Samut. Pont emiatt érdekes teszt lehet, hogy a játék során a sajá karakterünket és annak körülményeit nem mi irányítjuk. A játék 10 perc, se kevesebb sem több, tehát nem ér véget a játék ha eetleg levszetettük Samut, ilyenkor meg kell keresnünk! A program ezeket az elhaygyásokat is rögzíti és a végső kimenetében ezeket is számolj a teljesítményünk kiértékelésekor.

A program folyamatosan vizsgálja és frissíti azokat az adatokat amelyek a végeredmény meghatározásához kellenek. Ilynek pl.: Samu pozíciója illetve az egér pozíciója és az ezek közti távolság. Ezen kívűl említettem már, hogy ha elhagytuk Samut akkor sincs minden veszve, a forráskódban található egy függény, az "updateHeroes" itt tartja számon a program, hogy hányszor vesztettük el samu és a megtalálásokat is feljegyzi.

```
                void BrainBWin::updateHeroes ( const QImage &image, const  ↩
                   int &x, const int &y )
{

        if ( start && !brainBThread->get_paused() ) {

                int dist = ( this->mouse_x - x ) * ( this->mouse_x - x ) +  ↩
                   ( this->mouse_y - y ) * ( this->mouse_y - y );

                if ( dist > 121 ) {
                        ++nofLost;
                        nofFound = 0;
                        if ( nofLost > 12 ) {

                                if ( state == found && firstLost ) {
                                        found2lost.push_back ( brainBThread ↩
                                           ->get_bps() );
                                }

                                firstLost = true;

                                state = lost;
                                nofLost = 0;
                                //qDebug() << "LOST";
```

```
                                //double mean = brainBThread->meanLost();
                                //qDebug() << mean;

                                brainBThread->decComp();
                        }
                } else {
                        ++nofFound;
                        nofLost = 0;
                        if ( nofFound > 12 ) {

                                if ( state == lost && firstLost ) {
                                        lost2found.push_back ( brainBThread ↩
                                           ->get_bps() );
                                }

                                state = found;
                                nofFound = 0;
                                //qDebug() << "FOUND";
                                //double mean = brainBThread->meanFound();
                                //qDebug() << mean;

                                brainBThread->incComp();
                        }

                }

        }
        pixmap = QPixmap::fromImage ( image );
        update();
}
```

8.5 Malmö 19 RF

A félév során volt egy feladat amiben, Bátfai Norbert Tanár Úr adott nekünk egy kódot amivel Steve 19 pipacs felszdeésére volt képes és ezt kellett úgy továbbfejlesztenünk, hogy több virágot tudjon szedni, erről fogom csatolni a videónkat.

8. fejezet

Helló, Schwarzenegger!

Szoftmax Py MNIST

Python

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...

Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Minecraft-MALMÖ

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:

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

Tanulságok, tapasztalatok, magyarázat...

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

Tanulságok, tapasztalatok, magyarázat...

10. fejezet

Hello, Gutenberg!

10.1. Programozási alapfogalmak

A programozási nyelveknek alapvetőleg 3 szintjét tudjuk megkülönböztetni: gépi nyelv, assemly szintü nyelv és a magas programozási nyelvek. A gépi nyelv jelentősége az hogy a CPU-k saját gépi nyelvvel rendelkeznek és csak az azon a nyelven irt programokat tudják futtatni. Kezdő programozók tapasztalhatnak olyat ha pl .c kiterjesztésü fájlt próbénak futattni hogy szintaktikai hibát dob vissza mondjuk zárójel használatra, pontosan ezért van szükség fordítóprogramokra. Én linuxot használok igy mondjuk gcc prog.c -o prog paranccsl tudom előállitani a forráskódomból a futtatható tárgyprogramot. Amikor ezt tesszük a fordítóprogram először egy lexikális elemzést hajt végre majd ezt követően szintaktikai és szemantikai elemzést és az utolsó lépésben kódot generál. Ezzel szemben az interpretens technika az utolsó lépést kihagyja és soronként elemzi és hajtja végre a megadott lépéseket, ilyen technikával tudunk futtatni Python nyelven irt programokat. Assemly szintü nyelv: ezekről a köny csak emlités szintjén ir, ezek a nyelvek a gépi kód és a magas szintü programozási nyelvek közti átmenetet képviselik. Magas szintü programozási nyelvek:

Keringhan.Ritchie: A C programozási nyelv

Az első Alapismeretek c. fejezetet olvastam el. Mivel én már korábban is olvastam a könyvből, illetve főleg példaprogramokon keresztül mutatja be a programozás alapjait, ezért számomra viszonylag kevés új dolog volt benne főleg emlités szintjén számolok be az olvasott témákról.

Először egy klasszikus Hello Word! (A könyvben "Figyelem emberek!") program bemutatásával foglalkozik a könyv és vázolja is mire van szükségünk ahhoz hogy ezt futattni tudjuk, itt megemliti, hogy valamilyen környezetben irjuk meg a forráskódot, az elengedhetetlen fordität is emliti iiletve hogy ki kell találnunk, hogy a szöveget milyen képernyőn szeretnénk kiirva látni. Megjegyzem Linuxban mindezt terminálból végrehajthatjuk, pl a nano szövegszerkesztő környezetben illetve a beépitett gcc-ven tudunk is forditani. Bemutatásra kerül a jól ismert "main()" függvény, amibe a fő programunk kerül.

A következő pontban a váltotók kerülnek teritékre és azok deklarálása, ugye C-ben meg kell adnunk milyen tipusú változót szeretnénk létrehozni viszont ekkor még nem szükséges neki kezdőértéket adni mint Pythonban, ilyenkor változó tartalma gyakorlatilag memóriaszemét. A különböző tipusú változók (a példában int és float) használatát és a while ciklus müködését a köny egy Clesius-Farenheight váltóprogramon keresztül mutatja be, illetve a változó tipusokra való hivatkozás módját is szemlélteti (printf() függvény

használata).Ugyanezena példán keresztül bemutatja a for ciklus müködését és szintaxisát is, hogy az olvasó láthassa ugyanúgy célra vezető lehet midnkét ciklus használata, viszont itt fontos megjegyezni, hogy probléma specifikus mikor melyiket érdemes használni. Ezen kivül megemliti a könyv a szmbolikus állandókat melyekkel bizonyos objektumokra hivatkozhatunk a "#define" kifejezés segitségével.

A következő pont a karakterek ki és bevitelével foglalkozik a "getchar()" és "putchar()" függvények bemutatásán keresztül. A könyv is kiemeli, hogy ezeket a müveleteket a "printf()" és "scanf()" függvényekkel is megtehetjük, az hogy melyeket használjuk megint probéla specifikus. Ebben a bekezdésben szó esik az ASCII karakterkészletről is. a megoldandó példaprogram itta a különböző karakterek megszámlálásáról szól az előző témakörben tárgyalt ciklusok segitségével. A következő példaprogram bemenetre érkező szavak számlálását várja el, a szavakat szóköz választja el és az adott szavak első karakterét kell felismernie a kész programozóknak ez alapján annyi szót számol ahány szókezdő karaktert talál. Ekkor felmerül az elágazás kérdése is. A könyv bemutatja az if/else szintaxisát és az "andandhelye" azaz "és", illetve a "||" azaz "vagy" operátor müködését, amik elengedhetetlenek a különböző feltételek pontos megfogalmazásához.

A karakterek témaköre utána könyv a tömböket taglalja, ezeket C-ben szerintem egyszerü kezelni, szerencsére a számozásuk is 0-ról indul mint általában a magasabb szintü programozási nyelveknél megszokott. Fontosnak tartom megjegyezni hogy C-ben a stringek leirására gyakran a karaktertömbök használata a megoldás ami furcsa lehet annak aki korábban mondjuk C++-ban programozott.

Az utolsó témakör a függvények meghivásával, paraméterinek megadásáról és általános használatukról szól. Le van irva hogyan tudunk visszatérésé értékekkel pontosan dolgozni, egyik függvényét átadni másik függvénynek, általános hazsnálatukat a könyv jól szemlélteti.

Összefogalalásva: Az Alapismeretek c. fejezet valóban a legelemibb szinten mutatja be a C nyelv használatát Az olvasás során felmerülő témák majdnem mindegyékével találkoztam már korábban. Kezdő programozóknak ajánlanám, én is mikor elkezdtem C nyelven programozni ezt a könyvet kezdtem el olvasgatni és valóban hasznos is volt.

10.4. Python bevezetés

A Python programozási nyelv, amely 2018-ra a legnépszerübb programozási nyelvvé vált, Guido van Rossum nevéhez füződik aki azt 1990-ben alkotta meg. A kezdő programozóknak a python ideális nyelv számos előnye maitt, viszonylag könnyen elsajátitható, objektum orientált és platformfüggetlen is. A python olyan magas szintü prograozási nyelv amely jobban hasonlit a természetes angol nyelvhez mint a többi magas szintü programozási nyelv, tömör, jól olvasható kódokat irhatunk és nincs szükség zárójelezésre sem. Fontos megjegyezni hogy alkalmazásfejlesztéshez is ideális, gyorsabban lehet fele dolgozni mint pl a C/C++ esetében mivel itt kimarad a forditási fázis. A forráskódot az interpreternek köszönhetően azonnal futatthatjuk és lényegesen gyorsabbá válik a programozási folyamat.

SZINTAXIS: Ha már a python nyelvről irok, fontos megemliteni a szintaxist. A nyelv szintaktikájának egyik legfontosabb tulajdonsága hogy behúzás alapú. Pl.: ha irunk egy for ciklust a ciklusban lévő állitások egy behúzással (tab = 4 szóköz) benteb vannak mint a for ciklus feje. Erre emiatt a nyelv nagyon érzékeny is, ha nem megfelelően vannak a behúzásaink elrendezve egyből kapunk hibeüzenetet. A C/C++ nyelvekhez képest, amelyeket én is használtam korábban, talán a legszembetünőbb külonbség a ";" hiánya, illetve a ciklusok és függvények után kapcsos zárójelek helyett használatos ":". Ha kommentelni szeretnénk akkor tudunk soronként a "#"-el vagy több sort 3 db aposztróffal melyeket a komment elejére és végére kell illesztenünk.

A könyvben olvashattam a tipusokról és változókról. Ebben a programozási nyelvben is a szokásos tipusokat tudjuk használni (számok, azon belül eglszek tizedes/lebegőpontos törtek, stringek stb.) viszont jelentős különbség, hogy a váltózóinkat nem kell deklarálnunk, futás közben az interpreter a egadott kezdőérték alapján felismeri milyen tipusu változóval van dolga. Véltozók alatt az egyes objektumokra mutató refernciákat értjük. Vannak lokális és globális változók. Ha az adott egy függvényben vesszük fel akkor lokális változóról beszélünk, ha globálissá szeretnén tenni a függvény elején a változó felvételekor használnunk kell a "global" kulcsszót. Különböző változótipusok között szabad a konverzió, tehát tudunk stringől számot képezni stb. Szerencsére a nyelv sok beépitett függvényt tartalmaz igy a változóinkat könnyen tudjuk kezelni. A tömbök számozása 0-tól kezdődik.

Fontos megemliteni a nyelv eszközeit. Az első az elágazás, itt is a jól ismert if/elif/else kulccsszavakat használjuk, müködését tekintve olyan mint bármely más programozási nyelvben, a szitaxisában pedig a kettőspont használata az ami eltérést mutat más nyelvekhez képest. A ciklusoknál hasonló a helyzet, a while ciklusnál is a feltltel mögötti kettőspontot használjuk a kapcsos zárójelek helyett. Eltérés a for ciklusban van leginkébb ugyanis annak 2 fajtájával találkozhatunk. Az első a megszokott for ciklus, szintaxist illetően eltér de ezen kivül nincs különbség. A második fajta for cilus a "range()" függvényt használja, mely futtatása alattt egy listát ad vissza. A függvényeinkhez a "def" kulcsszót használhatjuk de itt is lányegbeli eltérést nem igazán tapasztalhatunk. Amikor az osztélyokról és objektumokról olvastam láttam magam előtt a "Class Steve:"-et és annak attribútumait. Mivel én korábban nem találkoztam objektum orientált programozással a gyakorlatban ezt még mindig tanulom és próbálom a gyakorlatba beépiteni de mindenképp hasznos volt olvasnom erről is a könyvben.

10.2. Szoftverfejlesztés C++ nyelven

A C++ egy magas szintü programozási nyelv általános célú frlhasználásra. Az első verziója 1983-ban jelent meg. jelentős különbség elődjéhez, a Chez klpest, hogy a C++ objektum orientált. C-hez nagyon közel áll, egy C++ fordító jó eséllyel tudja a C-t is forditani, hiába vannak benne pl C kulccsszavak vagy éppen C header fájlok. A 2-6. oldalon a C nyelvhez képesti továbbfejlesztésről olvashatunk, meg is emliti a könyv hogy az itt bemutatott kódokat már csak C++ forditóval fordulnak.

Először a függvények ekrülnek megemlitésre. A C nyevben egy üres paraméterlistájú függvényt tetszőleges számú paraméterrel hivhatunk meg, mig C++-ban az üres paraméterlistára a "void" kifejezés utal. Ezen kivül jelentős különbség, hogy ha nem diniálun k visszatérési értéket C-ben alapértelmezettként int tipusú visszatérési értéket kapunk vissza mig C++-ban forditási hibát. A "main()" függvény estén nem kötelező a return utasitást használni mivel ha a kód sikeresen a fordul annak visszatérési értéke mindenképp 0 lesz.

A bool tipust szükségszerü kiemelnem. Fontos különbség a két nyelv között, hogy C++-ban van bool tipusú változó amely "true" azaz igaz vagy "false" azaz hamis logikai értékkel rendelkezhet. C nyelvben ezt a logikai értéket int tipussal tudjuk kifejezni ami lehet 0 vagy 1.

A C-ben és a C++-ban is lehet használni több bájtos stringeket, A C++-szal ellentétben ami rednelkezik ezek kezeléséhez szükséges függvényekkel, C-ben ahhoz hogy a wchar_t tpiust tudjuk használni meg kell hivnunk vagy az stdlib.h, a z stddef.h vagy a wchar.h header fájlokat.

C++-ban mindenhol lehet változókat deklarálni ahol utasitást tudunk megadni. Ez azért hasznos mert ott tudjuk deklarálni ahol használni szeretnénk igy átláthatóbb lesz a kódunk és kisebb eséllyel feledkezünk meg a változóinkról. Pl.: ha egy for ciklust irunk annak fejében tudjuk deklarálni a ciklusváltozót, ekkor annak csak a ciklus törzsére lesz hatása.

A fordítás, futattás, nyomonkövetésről: Ez a bekezdés a könyvben számomra nem volt túl hasznos mivel itt a fejlesztői környezet fordításra és futtatásra való felhasználása volt kifejve, ráadásul mint exe fájl Windows alatt. Én általában Linux rendszert használok és az elmentett .c vagy .cpp kiterjesztésü fájljaimat a rendszerbe beépietett gcc és g++ fordítókkal tudom forditani terminálon keresztül, tehát a hibaüzenetek is a terminál képernyőjén jelennek meg. Egy helyen tudom a sikeres fordítás után futtatni a kész futattható állományaimat. Szerintem ezzel a módszerrel forditani mint ahogy az a könyvben van bemutatva de persze érdemes ismerni mindkét módszert.

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!

A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

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

Általános

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

C

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

C++

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

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.