Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

COLLABORATORS			
	TITLE : Univerzális programozás		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Kiss, Marcell	2019. november 10.	

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

Ajánlás

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

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

Tartalomjegyzék

Ábrák jegyzéke

Előszó

Amikor programozónak terveztem állni, ellenezték a környezetemben, mondván, hogy kell szövegszerkesztő meg táblázatkezelő, de az már van... nem lesz programozói munka.

Tévedtek. Hogy egy generáció múlva kell-e még tömegesen hús-vér programozó vagy olcsóbb lesz allokálni igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Mindenesetre a programozás a teoretikus kultúra csúcsa. A GNU mozgalomban látom annak garanciáját, hogy ebben a szellemi kalandban a gyerekeim is részt vehessenek majd. Ezért programozunk.

Hogyan forgasd

A könyv célja egy stabil programozási szemlélet kialakítása az olvasóban. Módszere, hogy hetekre bontva ad egy tematikus feladatcsokrot. Minden feladathoz megadja a megoldás forráskódját és forrásokat feldolgozó videókat. Az olvasó feladata, hogy ezek tanulmányozása után maga adja meg a feladat megoldásának lényegi magyarázatát, avagy írja meg a könyvet.

Miért univerzális? Mert az olvasótól (kvázi az írótól) függ, hogy kinek szól a könyv. Alapértelmezésben gyerekeknek, mert velük készítem az iniciális változatot. Ám tervezem felhasználását az egyetemi programozás oktatásban is. Ahogy szélesedni tudna a felhasználók köre, akkor lehetne kiadása különböző korosztályú gyerekeknek, családoknak, szakköröknek, programozás kurzusoknak, felnőtt és továbbképzési műhelyeknek és sorolhatnánk...

Milyen nyelven nyomjuk?

C (mutatók), C++ (másoló és mozgató szemantika) és Java (lebutított C++) nyelvekből kell egy jó alap, ezt kell kiegészíteni pár R (vektoros szemlélet), Python (gépi tanulás bevezető), Lisp és Prolog (hogy lássuk mást is) példával.

Hogyan nyomjuk?

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

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

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

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

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

II. rész

Tematikus feladatok

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás videó:

Megoldás forrása:itt

Tanulságok, tapasztalatok, magyarázat... A végtelen ciklus, olyan ciklus amely, soha nem ér véget, ezzel terhelve a processzort. A 100 százalékos terhelést egy magon a legkönnyebb végrehajtani, ehhez egy sima, egyszerű végtelen cikrusra van szükségünk, amely addig fut folyamatosan míg le nem állítjuk. A 0%-os terheléshez, a végtelen ciklusba szükséges egy "sleep" függvény, ami úgymond "elaltatja" azt a szálat amelyet a végtelen ciklus használna, így a processzor mag terhelése 0%-os lesz. Ahhoz hogy minden magot 100%-on dolgoztasson, szükségünk lesz az OpenMP-re és a "#pragma omp paralell" sorra. Ez röviden több szálon dolgoztatja a programot, így elérve, hogy minden processzormag 100%-osan legyen kihasználva.

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

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

Megoldás videó:

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

```
Program T100
{

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

```
      return false;
  }

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{

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

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

  main(Input Q)
  {
    Lefagy2(Q)
  }

}
```

Mit for kiírni erre a `T1000(T1000)` futtatásra?

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

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

Tanulságok, tapasztalatok, magyarázat... A T100 egy programot kap meg bemenetként, amiről neki el kell dönteni, hogy az legáll-e, vagy nem. Tehát a T100-as kap egy programot inputként, és megnézi, hogy az általa kapott program kódjában található-e végtelen ciklus. Ha ezt sikerült megállapítania kapunk egy igaz vagy hamis kimenetet. Ezt az outputot kapja meg majd a T1000-es, amely ha a igaz a bemenet, lefagy, ha hamis, akkor pedig bekerül egy végtelen ciklusba. Eddig minden oké, de mi történik akkor, ha a T1000-nek saját magát adjuk oda bemenetként? Ha ő azt látja, hogy van saját magában végtelen cilus, akkor le fog fagyni, ha nincs, akkor viszont bekerül egy végtelen ciklusba. Emiatt az ellentmondás miatt nem lehet ilyen programot írni. Vagy legalábbis eddig még senkinek nem sikerült

2.3. Változók értékének felcserélése

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

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

Megoldás forrása:itt

Tanulságok, tapasztalatok, magyarázat... Ennek a legegyszerűbb módja, ha matematikai művelettel cseréljük fel a két változó értékét. Én itt összeadás és kivonás segítségével hajtottam ezt végre, de ugyanúgy lehetséges szorzás/osztással is. Az én verzióm lényege, hogy megadom a programnak "a" és "b" értékét, jelen esetben 10 és 5. Majd "a" értékét átírom "a" és "b" összegére így "a" értéke jelenleg 15. Következő lépésben "b" értékének megadom, hogy "a" és "b" különbsége legyen, tehát 15-5, így a "b" értéke már 10, tehár féig megvagyunk. Az utolsó lépés, hogy "a" értékét ismét átiratom a programmal az a=a-b művelettel, azaz 15-10, így a értéke 5 lesz. Szóval a két változó értékét egyszerűen három matematikai művelettel felcseréltem.

2.4. Labdapattogás

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

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

Megoldás forrása:https://github.com/SylwerStone/Prog1/blob/master/labda.c

Tanulságok, tapasztalatok, magyarázat: Kezdésként include-oljuk szükséges headereket.

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

Ebben a programban, a megjelenítéshez a 'usleep' és a 'clear' a két legfontosabb függvény. Ahhoz hogy a labda pattogni tudjon, tudnunk kell hogy mekkora az ablak mérete, ezt az initscr() függvény használatával tudjuk meg. Ezután kell létrehoznunk pár változót, amikben az aktuális pozíciót, az x és y tengelyen lévő lépésközöket, valamint az ablak méretét tároljuk majd.

```
int x = 0; //x tengelyen lévő jelenlegi pozíció
int y = 0; //y tengelyen lévő jelenlegi pozíció

int xnov = 1; //x tengelyen lévő lépésköz
int ynov = 1; //y tengelyen lévő lépésköz

int mx; //ablak szélessége
int my; //ablak magassága
```

Ezután létrehozunk egy végtelen ciklust, amiben a labdánk 'pattogni' fog. Majd a getmaxyx függvénynek megadjuk az ablakban lévő értékeket, a mvprintw pedig a labdát fogja mozgatni a megadott értékekre.

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

A labdát mostmár az x és y értékének egyel növelésével tudjuk mozgásra bírni. Ebben játszik szerepet a usleep függvény, mivel ezzel tudjuk állítani, hogy ez milyen gyorsan történjen. (A usleep millisecundumokban számol) A clear függvény pedig törli a labda előző helyzetét és tényleg úgy látjuk mintha pattogna, nem pedig egy csíkot húz maga után. Az ifekkel tudjuk meghatározni azt, hogy az ablak szélénél a labda visszaforduljon, ezt úgy érjük el, hogy a lépésközt -1-el szorozzuk.

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

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/szohossz.c

Tanulságok, tapasztalatok, magyarázat...

```
#include <stdio.h>

int main()
{
  int hossz=0;
  int n=0x01;
  do
  {
    hossz++;
  }while(n<<=1);
  printf("Szohossz: %d bit\n",hossz);
  return 0;
}
```

Hasonló programot írtunk tavaly c++-ban. Ez a program úgynevezett shiftelés segítégével dönti el, hogy hány bitből áll a szó. Tehát addig lépeget míg az első szám 0-a nem lesz. Ez az ún. BogoMIPS-es shiftelős módszer. A BogoMIPS egy CPU sebességmérő, amit Linus Torvalds, (Linux kernel atyja) írt meg. Lényegében a program feladata az, hogy leméri mennyi idő alatt fut le, kapunk tőle egy értéket, amely alapján el lehet dönteni, hogy milyen gyors a processzor.

2.6. Helló, Google!

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

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/pagerank.c

Tanulságok, tapasztalatok, magyarázat... A PageRank algoritmust Larry Page és Sergey Brin fejlesztették ki 1998-ban. Ez az algoritmus a mai napig a Google keresőmotorjának a legfontosabb része. Ez egy olyan rendszer, amely arról szól, hogy melyik oldal milyen prioritást élvez, és hogy ezek az page-ek melyik másik page-re mutatnak. Így tudja az algoritmus meghatározni, hogy melyik az a legrelevánsabb oldal amely keresésünknek legjobban megfelel. Ez a kód ezt az algoritmus mutatja be, viszont csak egy 4 weblapos hálózaton. A weboldalak kapcsolatát egy mátrixban tároljuk el.

```
double L[4][4] = {
```

```
{0.0, 0.0, 1.0 / 3.0, 0.0},
{1.0, 1.0 / 2.0, 1.0 / 3.0, 1.0},
{0.0, 1.0 / 2.0, 0.0, 0.0},
{0.0, 0.0, 1.0 / 3.0, 0.0}
```

Ezt a mátrixot megkapja a pagerank függvény argumentumként.

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

  int i, j;

  for(;;){



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

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



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

  kiir (PR, 4);
```

A ’PRv’ blockban az oldalak első, eredeti értékét tároljuk, a PR blockban pedig a mátrixműveletet tároljuk. Ez a mátrixművelet egy szorzás, amely az ’L’ és a ’Prv’ block szorzását jelenti. Ennek a szorzásnak az eredményeként kapjuk meg az oldalak pagerank értékét. A ’kiir’ fügvénnyel íratjuk ki egyenként a weboldalak eredményét.

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

            int i;

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

```
                    }
                }
```

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/Primek_R A Brun tétel azt mondja, hogy az ikerprimek reciprokösszege egy bizonyos összeghez konvergál, szóval közel ér hozzájuk, de soha nem éri el magát a számot. A Tételt Viggo Brun, norvég matematikus dolgozta ki.

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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

Tanulságok, tapasztalatok, magyarázat... A Monty Hall probléma alapja egy amerikai tv show (Let's Make a Deal), amely arról szól, hogy a játék végén mutatnak a játékosnak 3 ajtót. A 3 ajtó közül kettő mögött nincs semmi, vagy csak egy értéktelen tárgy van, 1 mögött viszont értékes nyeremény. A játékos választ egy ajtót, de még nem nyitják ki, hanem a műsorvezető kinyitja az egyik olyan ajtót, ami mögött nem az értékes nyeremény van. Ekkora a játékos eldöntheti, hogy szeretne e változtatni döntésén és kinyitni a másik ajtót, vagy marad a választottjánál. Itt felmerül a kérdés, hogy egyáltalán megéri-e váltani. A válasz meglepő módon igen. Ez azért paradoxon, mert ellentmond a józan paraszti észnek. Mivel elvileg mikor rábökünk egy ajtóra akkor 1/3-ad az esélye, hogy jóra mutattunk, ezen nem változtat ha változtatunk. Vagy mégis? Mikor rámutattunk egy ajtóra még 2/3 valószínűséggel nem volt mögötte semmi, viszont kinyitották az egyik üres ajtót. Innentől biztos, hogy a nyeremény az egyik ajtó mögött van. Tehát így a nyeremény 2/3-ad eséllyes a másik ajtó mögött van. Először meg kell adnunk hogy hány kisérlet lesz. Ezt randommá a "sample" fügvénnyel tesszük, amelynek meg kell adni, hogy hánytól hányig generáljon számokat és hogy hányszor tegye ezt. A replace=T pedig megengedi az ismétlődést a számoknál. Az adatokat különböző blockokban tároljuk. A 'kiserlet' blockban azt tároljuk, hogy mikor, hová melyik ajtó mögött található a díj, a 'jatekos' blockban pedog a játékos döntését tároljuk. A 'musorvezeto' függ a nyeremény helyzetétől és hogy a player mit választ. Egy for ciklussal végignézzük az összes játékot, az if segítségével dönti el a műsörvezető, hogy melyik ajtót kell kinyitnia. Az if föggvényünk egyik érsze azt nézi meg, hogy a játékos, jól választott-e, tehát azt az ajtót, ahol a fődíj van. Ha jól választott a 'mibol' a másik két ajtó egyike lesz, ha viszont nem jól választott, akkor az else azt mondja, hogy azt az ajtót legyen ami mögött nem a nyeremény van és amit nem a játékos választott. Ha ez megvan, ezt az információt műsörvezető megkapja és egy üres ajtót fog kinyitni. Most jön a játékos, hogy akar-e változtatni a döntésén. A 'nemvaltoztatesnyer' akkor vesz csak fel értéket, ha a játékos jól választott és úgy dönt, hogy nem is választ másik ajtót. Ha viszont üres ajtóra mutatott, és úgy határoz hogy változtat, akkor kelleni fog a for ciklus ismét. A 'holvalt' annak

az ajtónak az értéke, se a műsörvezető nem nyitott ki, és a játékos sem választotta, majd ezt az értéket veszi át a 'valtoztat' tömb. Megnézi, hogy a játékos által kiválasztott ajtó és a nyertes ajtó megegegyezik-e, és ha igaz értéket kap, akkor a 'valtoztatesnyer' íródik ki. Végül a 'valtoztatesnyer' és a 'nemvaltoztatesnyer' hosszát meg kell néznünk, hogy el tudjuk dönteni, hogy melyik a melyik a nagyobb.

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

3.3. Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása:https://github.com/SylwerStone/Prog1/blob/master/C99.C

Tanulságok, tapasztalatok, magyarázat...

```
#include <stdio.h>

int main()
```

```
{
for(int a=5;a>10;a++);
return 0;
}
```

C szabvány fejlődésével egyre több funkciót kapott, ám ezek a funkciók nem kompatibilisek visszafelé. A fenti kód például c99-ben lefordul c89-ben viszont nem. Ennek oka, hogy C89-ben még nem lehetett a for ciklusban a ciklusfejben történő ciklusváltozót deklarálni.

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/lex.c

Tanulságok, tapasztalatok, magyarázat:

A lex forráskódunk 3 részből áll: Az 1. rész: definíciós rész, ahol headereket unclude-olhatunk, változókat deklarálhatunk A 2. rész: Itt a szabályok vannak megadva Ennek két része van, az egyik a reguláris kifejezések, a másik pedig az ezekhez a kifejezésekhez tartozó utasítások A 3. rész: Ez egy c kód Az első és a harmadik rész majd átkerül a generált forrásba is. Tehát az első részben include-oljuk a headereket, és változókat deklarálunk. A második rész, ahogy leírtam a szabályokat tartalmazza.

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

Itt található egy regex, amit előző félévben Operációs rendszerek órán tanultunk. Ez arra az inputokra illeszkedik amik számokkal kezdődnek, ez akárhányszor előfordulhat. (A * ezt jelzi regexben) Majd a zárójeles rész viszont csak 1-szer vagy 1-szer sem fordulhat elő (ezt a ? jelzi) A 3 rész:

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

Itt hívjuk meg a lexikális függvényt a yylex segítségével, majd egy printf-fel kiíratjuk az eredményt.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó:

Megoldás forrása:https://github.com/SylwerStone/Prog1/blob/master/l33t.c

Tanulságok, tapasztalatok, magyarázat:

A l33t nyelv lényege annyi, hogy a szavakban lévő betűket, valamilyen más karakterekre cseréljük, ezek lehetnek pl számok is akár. Ahogy fentebb láthattuk, itt is három részre oszlik a kódunk. Kezdésként include-oljuk a szükséges headereket, majd define-oljuk a L337SIZE-t, tehát ha valahol hivatkozva lesz rá, akkor a mellette lévő értékeket fogja használni

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

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

Ezután létrehozunk egy 'cipher' nevű struktúrát, amely egy char c-ből és egy char c* pointerből áll.

```
struct cipher {
char c;
char *leet[4];
```

Aztán létrehozzuk a l337d1c7 block-ot, amely azt tartalmazza, hogy melyik betűt milyen karakterekkel helyettesíthetünk.

```
} 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"}}
```

A következő részben, mivel pontot használunk, így minden kakarkert vizsgálnia kell a programnak. A program megvizsgálja a karaktereket, ha megtalálja a cipher tömbben, akkor generál egy random számot, ami alapján eldönti, hogy az 'l337d1c7' tömbben hányadik karakterrel helyettseítse. Ha nem találja meg akkor az eredti megakpott karaktert írja ki változatlanul. Mint látjuk ha a random szám kisebb mint 91 akkor az első karakterrel helyettesíti, ha kisebb mint 95, akkor a másodikkal és így tovább.

```
    . {

  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);

        }
```

Az utolsó rész itt is egy C forráskód, amiben a lexelést indítjuk el, ugyanúgy mint az előző programunknál.

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

3.6. A források olvasása

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

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

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

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

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

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

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

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

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

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

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

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

Megoldás forrása:

Megoldás videó:

Tanulságok, tapasztalatok, magyarázat:

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

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

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

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

Mit vezetnek be a programba a következő nevek?

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

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/dek.c

Tanulságok, tapasztalatok, magyarázat: A programozási nyelvekben az egészeket ábrázoló adattípus az integer (röviden int). A következőket vezetjük be a programban:

- ```
  int a;
                  // Ez az egész változó (int típusú)
  ```
- ```
  int *b = &a;
                  //ez az egész változóra (a) mutató mutató (*b)
  ```

- ```
  int &r = a;
                  //Ez az egész típusú változó (a) a referenciája (&r)
  ```
- ```
  int c[5];
                  //Ez az egészek tömbje (5 elemű)
  ```
- ```
  int (&tr)[5] = c;
                  //Ez az előző (c) elem referenciája (&tr)
  ```
- ```
  int *d[5];
                  //A (d) egy tömb, amely 5 darab egészre mutató mutatót ↩
                     tartalmaz
  ```
- ```
  int *h ();
                  //A h függvény az a  egészre mutató mutatót visszaadó ↩
                     függvény
  ```
- ```
  int *(*l) ();
                  //Ez egészre mutató mutatót visszaadó függvényre mutató ↩
                     mutató
  ```
- ```
  int (*v (int c)) (int a, int b);
                  //Ez az egészet visszaadó (ami a c) és két egészet kapó ( ↩
                     Az a és b) függvényre mutató mutatót (*v) visszaadó, ↩
                     egészet kapó függvény
  ```
- ```
  int (*(*z) (int)) (int, int);
                  //Ez pedig a függvénymutató (az első int előtii (*(*z) ↩
                     egy egészet visszaadó (legelső 'int' a (*(*z) után ) ↩
                     és két egészet kapó (az utolsó 2 'int') függvényre ↩
                     mutató mutatót visszaadó (Ez a sima (*z)), egészet ↩
                     kapó függvényre
  ```

4. fejezet

Helló, Caesar!

4.1. int ** háromszögmátrix

Megoldás videó:

Megoldás forrása:https://gitlab.com/nbatfai/bhax/blob/master/thematic_tutorials/bhax_textbook_IgyNeveldaProg
Caesar/tm.c

Tanulságok, tapasztalatok, magyarázat:

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/e.c

Tanulságok, tapasztalatok, magyarázat: Kezdésnek szokás szerint include-oljuk a szükséges headereket, majd define-oljuk a buffer méretet (256) és a maximum kulcsot (100). A mainben fellelhető 'argc' és 'argv' argumentumokat és azok számát tárolja. Ezután deklarálunk két char típusú tömböt a 'buffer'-t és a 'kulcs'-ot, majd két integer típusú változót a 'kulcs_index'-et és az 'olvasott_bajtok'-at. Aztán deklarálnunk kell még egy integert a 'kulcs_meret'-et, aminek az értéke az argv 2. elemének nagysága lesz. Ezután ezt a 'strncpy' függvénnyel átmásoljuk a kulcs változóba. Ha ezzel megvagyunk a következő lépés, hogy beolvassuk a byte-okat, ameddig a bufferben van elég hely neki (ez 256 byte lesz). Amíg az 'i' kisebb mint az elemenként olvasott byte-ok, azokat a (kulcs[kulcs_index])-el kezeljük. Végül kiíratjuk a kapott byte-okat egy fájlba.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/exor.java

Tanulságok, tapasztalatok, magyarázat: Itt java nyelven írjuk meg ugyan azt az EXOR titkositót. A java egy objektumorientált nyelv, amely szintaxisát a C-től és a C++-tól örökölte, viszont egyszerűbb objektumodellekkel rendelkezik azoknál. A java alkalmazásokat általában bájtkód formájúvá alakitják, de lehet nativ kódot is késziteni belőle. A program ugyan azt csinálja mint az előző c nyelven íródott program. Megkapjuk az adatokat, amelyeket EXOR használatával byte-onként titkosítunk, majd kiíratjuk őket.

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/t.c

Tanulságok, tapasztalatok, magyarázat: Kezdésként szokás szerint include-oljuk a megfelelő headereket, majd definiálnunk kell pár függvényt. Az 'atlagos_szohossz' függvény kiszámolja a bemenet átl. szóhosszát.

```
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;
}
```

A tiszta_lehet azt nézi meg, hogy a szöveg amit megetetünk vele, feltört-e vagy nem. A szöveg általában, akkor tiszta, vagy tört ha benne vannak a 'hogy' 'nem' 'az' és a 'ha'szavak, és vizsgáljuk az átlagos szóhosszt is

```
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");

}
```

Ha ennek nem felel meg a szöveg, akkor nem fogjuk tudni feltörni. Következik az exoros eljárás, ahol megkapjuk a 'kulcs'-ot, a 'kulcs_meret'-et, a 'titkos'-t, a 'tikos_meret'-et és a 'buffer'-t. Ezután lefuttatunk egy for ciklust az összes karakteren.

```
void
exor (const char kulcs[], int kulcs_meret, char titkos[], int  ↩
   titkos_meret, char *buffer)
{

  int kulcs_index = 0;

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

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

Aztán következik az exor törés, ami ugyan azokkal az adatokkal dolgozik, mint az előző folyamat.

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

  char *buffer;

  if ((buffer = (char *)malloc(sizeof(char)*titkos_meret)) == NULL)
  {
    printf("Memoria (buffer) faliora\n");
    exit(-1);
  }

  exor (kulcs, kulcs_meret, titkos, titkos_meret, buffer);

  if(tiszta_lehet (buffer, titkos_meret))
  {
    printf("Kulcs: [%c%c%c%c%c%c%c%c]\nTiszta szoveg: [%s]\n",
       kulcs[0],kulcs[1],kulcs[2],kulcs[3],kulcs[4],kulcs[5],kulcs ↩
          [6],kulcs[7], buffer);
  }

  free(buffer);
```

A main függvényben deklarálnunk kell néhány 'char' és egy 'int' típusú változót.

```
int
main (void)
```

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

Ezután következik a titkos fájl 'berántása' egy while ciklussal, majd egy for ciklussal következik a maradék hely nullázása a titkos bufferben.

```
      // 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';
```

Ezután következik a 8 db for ciklus, amelyekkel előállítjuk a kulcsokat, majd minddel megpróbáljuk a törést külön külön. Ha valamelyik működik és megfelel a 'tiszta_lehet' függvénynek, akkor kiírja a helyes kulcsot és a feltört szöveget.

```
       // osszes kulcs eloallitasa
      //int ii, ji, ki, li, mi, ni, oi, pi; //-5.1-es példa: ezek cikluson  ↩
         kívül definiált változók
#pragma omp parallel for private(kulcs,ii, ji, ki, li, mi, ni, oi, pi)  ↩
   share(p, titkos)
     for (int ii = '0'; ii <= '9'; ++ii)
        for (int ji = '0'; ji <= '9'; ++ji)
            for (int ki = '0'; ki <= '9'; ++ki)
                for (int li = '0'; li <= '9'; ++li)
                    for (int mi = '0'; mi <= '9'; ++mi)
                        for (int ni = '0'; ni <= '9'; ++ni)
                            for (int oi = '0'; oi <= '9'; ++oi)
                                for (int pi = '0'; pi <= '9'; ++pi)
                                 {
            //printf("%p/n", kulcs); //-5.2-es példa:kulcámának tömbök  ↩
               számának nyomonkövetése

                                        kulcs[0] = ii;
                                        kulcs[1] = ji;
                                        kulcs[2] = ki;
                                        kulcs[3] = li;
                                        kulcs[4] = mi;
                                        kulcs[5] = ni;
                                        kulcs[6] = oi;
                                        kulcs[7] = pi;
```

```
                                    exor_tores (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: A neuronok feldata az, hogy valamilyen jeleket továbbítsanak. A neuronoknak 3 rétegük van: - bementi réteg - kimeneti réteg - rejtett réteg A bementi réteg feladata, hogy továbbadja a kapott adatokat a többi résznek. A kimeneti rétegben találhatóak meg a függvények és a kimeneti neuronok. Míg a rejtett részben zaljanak a lényeges folyamatok. A jel továbbítása egy küszöbértéktől függ, ha elérjük ezt a küszöbértéket akkor indul csak el a folyamat. A programban lévő a1 és a2 sorok, azok fix sorok, nem változnak. Az 'OR' 'AND' és 'EXOR' sorok valamilyen logikai művelettel jöttek létre. A különböző sorok utáni 'data.frame' paranccsal data frame-ket hozunk létre, ami lehetővé teszi, hogy a kapott adatokat táblázat formájában tároljuk.

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/SylwerStone-perceptron/main.cpp

Tanulságok, tapasztalatok, magyarázat: A perceptron a megserséges intelligenciában használt neuron egyik legelterjedtebb válzotata, ami úgymond egy alakfelismerő gép, amelynek az a feladata, hogy véges számú kísérlet alapján osztályozni tudja a bináris alakzatokat. Itt a mandelbrot halmaz alapján generált kép RGB kódját rakjuk be a percepton inputjába. A programhoz tehát szükségünk lesz a mandelbrot halmaz által generélt png-re, az ml.hpp-re és a main.cpp-re. A main.cpp forráskódjának elején include-oljuk például az ml.hpp, amely a Perceptron osztályt tartalmazza.

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

Ezután létrehozunk egy üres png-t

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

Egy változóban tároljuk el a kép méretét, majd létrehozzuk a Perceptron-t, amelyben azt adjuk meg, hogy hány rétegünk legyen (itt 3 lesz), és hogy azokon a rétegeken hány neuron legyen. Az utolsóba 1-et rakunk, mivel ez adja majd az eredményünket

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

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

A memóriába másoljuk a for ciklusok segítségével a kép piros pixeleit. Majd a Perceptron osztály operátora adja meg nekünk az eredményt, amit a cout-at kiíratunk

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

    double value = (*p) (image); //ez adja vissza az eredményt

    std::cout << value << std::endl; //ezzel íratjuk ki az eredményt
```

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/mandelbrot.cpp

Tanulságok, tapasztalatok, magyarázat:

A Mandelbrot halmaz felfedezője Benoît Mandelbrot volt, akiről a halmaz a nevét is kapta. A Mandelbrot halmaz elemei a komplex számok. Ha ezeket a komplex számokat ábrázoljuk, a komplex számsíkon, akkor különös formájú alakzatokat kapunk. A fenti c++ programmal tudjuk ezt megtenni, amely egy ábrát készít nekünk. Nézzük meg a programot: Először is include-oljuk a szükséges header fájlokat, aztán meg kell adnunk, hogy melyik fájlba szeretnénk menteni a képet, ha ezt nem adjuk meg kapunk egy hibaüzenetet.

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

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

Ezután megadunk egy értékkészletet a függvénynek, valamint megadjuk hogy milyen magas és széles legyen a képünk. Meg kell adnunk még az iterációs határt is.

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

Ezután létrehozzuk a png fájl-t amibe majd a mandelbrot halamaz kerül berajzolásra

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

Ezután a program végigmegy a koordináta rendszer pontjain

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

    int iteracio = 0;
    std::cout << "Szamitas";

    for (int j=0; j<magassag; ++j) {
        //sor = j;
        for (int k=0; k<szelesseg; ++k) {

            reC = a+k*dx;
            imC = d-j*dy;
            // z_0 = 0 = (reZ, imZ)
            reZ = 0;
            imZ = 0;
            iteracio = 0;

            while (reZ*reZ + imZ*imZ < 4 && iteracio < iteraciosHatar) {
                // z_{n+1} = z_n * z_n + c
                ujreZ = reZ*reZ - imZ*imZ + reC;
                ujimZ = 2*reZ*imZ + imC;
                reZ = ujreZ;
                imZ = ujimZ;

                ++iteracio;
```

Meg kell adnunk valamilyen színt a pixeleknek, aztán berajzoljuk a kapott Mandelbrot-halmazt abba az üres képfájlba, amit az elején létrehoztunk.

```
kep.set_pixel(k, j, png::rgb_pixel(255-iteracio%256,
                                                    255-iteracio%256, 255- ↩
                                                       iteracio%256));
        }
        std::cout << "." << std::flush;
    }

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat: Ez a program annyiben különbözik az előzőtől, hogy megadhatunk 8 paramétert parancssori argumentumként (Ha nem adjuk meg, akkor az alapértelmezettet fogja használni), illetve itt két változó helyett csupán egyet használunk a komplex számok tárolához. Valamint ennél színesebb ábrát fogunk kapni mint az előzőnél. Ehhez csupán a complex könyvtárra van szükségünk és máris spóroltunk magunknak 1 változót.

```
        // 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;
```

Amit még tud a program, hogy %-ban látjuk a folyamat állapotát.

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

5.3. Biomorfok

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

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

Tanulságok, tapasztalatok, magyarázat:

A Biomorfokat Clifford A. Pickover fedezte fel, a Julia halmaz kutatása alatt. Ugyanis megírt egy programot, az el˝obb említett halmaz megjelenítésére, ám a programkódban volt valami hiba. Ezáltal a hiba által fedezte fel ezeket az úgynevezett biomorfokat. A Julia halmaz egyébként részhalmaza a Mandelbrot halmaznak, annyi különbséggel, hogy míg a Juliában a "c" konstansként szerepel, addig a Mandelbrotban már változóként.

Nézzük a programot: Include-oljuk kezdésként a szükséges headereket. Itt látjuk, hogy 8 helyett már 10 parancssori argumentumunk van, amiket itt sem kötelező megadni, szimplán az alapértelmezett értékeket fogja használni a program.

```
    {
        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;
    }
```

Ezután hozzuk létre az üres png-t, valamint azt, hogy mekkora lépésközünk legyen.

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

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

Mint mondtam, a c konstansként szerepel, ezét a cc a cikluson kívül van.

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

Ez az a bug, ami miatt létrejött a program:

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

Végül ugyan azt csináljuk mint az előző programoknál: beállítjuk a pixelek színét és kiíratjuk 1 fájlba.

```
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

Megoldás videó:

Megoldás forrása:

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

Megoldás forrása:

Megoldás videó:

Megoldás forrása:

5.6. Mandelbrot nagyító és utazó Java nyelven

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás videó:

Megoldás forrása: c++ :https://github.com/SylwerStone/Prog1/blob/polargen/polargen.cpp java: https://github.com/SylwerStone/Prog1/blob/polargen/polargen.java

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

Először include-oljuk a szükséges headereket. Aztán létrehozunk egy 'Polargen'-nek elnevezett osztályt, ezen belül is egy public és egy private részt (a public osztályon kívül is, a private pedig csak osztályon belül elérhető) és megadjuk, hogy még nincs eltárolt szám.

```
class PolarGen
{
public:
  PolarGen ()
  {
    nincsTarolt = true;
    std::srand (std::time (NULL));
  }
```

```
private:
  bool nincsTarolt;
  double tarolt;
```

A generátor kap egy random seedet, majd a 'kovetkezo' függvény megnézi, hogy van e tárolt szám. Ha nincs akkor létrehoz kettőt, amelyek közül az egyiket elmenti a másikkal pedig return-öl. A másikat akkor adja vissza, ha már volt tárolt szám.

```
double kovetkezo ()
 {
   if (nincsTarolt)
     {
       double u1, u2, v1, v2, w;
       do
   {
     u1 = std::rand () / (RAND_MAX + 1.0);
     u2 = std::rand () / (RAND_MAX + 1.0);
     v1 = 2 * u1 - 1;
     v2 = 2 * u2 - 1;
     w = v1 * v1 + v2 * v2;
   }
       while (w > 1);

       double r = std::sqrt ((-2 * std::log (w)) / w);

       tarolt = r * v2;
       nincsTarolt = !nincsTarolt;

       return r * v1;
     }
   else
     {
       nincsTarolt = !nincsTarolt;
       return tarolt;
     }
 }
```

A program utolsó része pedig legenerál nekünk 10 véletlenszerű számot.

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

  for (int i = 0; i < 10; ++i)
    std::cout << pg.kovetkezo () << std::endl;

  return 0;
}
```

6.2. LZW

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

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/z.c

Tapasztalatok: Az LZW (Lempel-Ziv-Welch) algoritmust, amely egy veszteségmentes tőmőrtési algoritmus 1984-ben publikálta Terry Welch. (A Abraham Lempel és Jacob Ziv által fejlesztett LZ78 algoritmus továbbfejlesztéseként) Legfőbb felhasználása: Unix Compress programja, Gif, és a PDF tömöritő alforitmus között is szerepel. Az LZW a bemeneti adatokból egy úgynevezett binfát épít, olyan módon hogy végig, hogy megnézi van-e 1-es vagy 0-ás oldal, ha nincs, akkor létrehoz egyet és visszaugrik a gyökérre, ha van akkor vagy az 1-es vagy a 0-ás oldalra lép és addig megy lefelé míg létre tud hozni egy újat. A while ciklus hozza létre a fát, a bemenetet olvasva. Ha az első bit 0, akkor megnézzük hogy van-e 0-ás ág, ha nincs, akkor létre kell hozni egyet, majd visszamegyünk a gyökérre, ha van, akkor a bal oldalra a 0-ra ugrunk. Ha a bemenet 1, akkor ugyan ezt csináljuk ellenkezőleg.

Kód: Először létrehozunk egy struktúrát, amely egy értékből, és annak gyerekeire mutató mutatókból áll.

```
        typedef struct binfa
        {
        int ertek;
        struct binfa *bal_nulla;
        struct binfa *jobb_egy;
        }  BINFA, *BINFA_PTR;
```

Ezután helyet kell foglalni, a változóknak, és visszakapunk egy pointert, ami a lefoglalt területre mutat. Ha nincs elég memória, akkor error-t kapunk és a program kilép.

```
BINFA_PTR
uj_elem ()
{
  BINFA_PTR p;

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

A main elején hozzuk létre a gyökeret, amit '/'-el jelölünk. Jeleneleg nincs gyereke, szóval a pointerek NULL értéket vesznek fel. A fa mutatót pedig a gyökérre állítjuk rá.

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

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

Ezután olvassuk be a bemenetet és itt jön létre a fa. Itt történik az amit fentebb írtam: Megnézzük, hogy a bemenet 1 vagy 0. Ha például 1 és nincs ilyen gyerek, akkor létrehoz egyet, a gyerekei pointerét NULL-RA állítjuk, a fa mutatót pedig visszaállítjuk a gyökérre. Ha viszont már van ilyen gyerek, akkor rálépünk arra és a következő bitet vizsgáljuk.

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

A main következő részében íratjuk ki a fát. A szabadit függvény felszabadítja a lefoglalt memóriát, a kiir függvény pedig inorder módon kiíratja a fát a standard outputra.

```
printf ("\n");
          kiir (gyoker);
          extern int max_melyseg;
          printf ("melyseg=%d", max_melyseg);
          szabadit (gyoker);
        }
        static int melyseg = 0;
        int max_melyseg = 0;
        void
        kiir (BINFA_PTR elem)
```

```
{
  if (elem != NULL)
    {
      ++melyseg;
      if (melyseg > max_melyseg)
  max_melyseg = melyseg;
      kiir (elem->jobb_egy);
      for (int i = 0; i < melyseg; ++i)
  printf ("---");
      printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek :  ↩
         elem->ertek,
        melyseg);
      kiir (elem->bal_nulla);
      --melyseg;
    }
}
void
szabadit (BINFA_PTR elem)
{
  if (elem != NULL)
    {
      szabadit (elem->jobb_egy);
      szabadit (elem->bal_nulla);
      free (elem);
    }
}
```

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: A preorder és az inorder bejárás közötti különbés annyi, hogy preorderben először a fa gyökerét dolgozzuk fel, majd bejárjuk a fa bal oldalát aztán a jobb oldalát. A postorder eljárásban pedig a preorderrel ellenkezőleg, előbb a fa bal oldalát járjuk be, aztán a jobb oldalát, és végül legutoljára járjuk be a fa gyökerét. Ehhez csak a kiir függvényt kell módosítanunk az előzőhöz képest. A postorder bejárásnál a for ciklust az utolsó helyre raktuk, a két gyerek feldolgozása utánra, így előbb a jobb és bal odali gyerek, aztán a gyökér kerül feldolgozásra.

```
void kiir(BINFA_PTR elem)
{
 if (elem !=NULL)
  {
   ++melyseg
   if (melyseg>max_melyseg)
```

```
            max melyseg = melyseg;
           kiir(elem->jobb_egy);
           kiir(elem->bal_nulla);
           for (int i=0; i<melyseg;i++)
            printf("---");
           printf("%c(%d)\n",elem->ertek<2 ? '0' + elem->ertek : elem-> ↩
              ertek, melyseg-1);
           --melyseg
          }
        }
```

Preordernél ellenkezőleg, a for ciklus kerül legelőlre, így előbb a gyökér kerül feldolgozásra, aztán a bal és jobb oldali gyerekei.

```
void kiir(BINFA_PTR elem)
        {
         if (elem !=NULL)
          {
           ++melyseg
           if (melyseg>max_melyseg)
            max melyseg = melyseg;
            for (int i=0; i<melyseg;i++)
            printf("---");
           printf("%c(%d)\n",elem->ertek<2 ? '0' + elem->ertek : elem-> ↩
              ertek, melyseg-1);
           kiir(elem->jobb_egy);
           kiir(elem->bal_nulla);

           --melyseg
          }
        }
```

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:

Megoldás forrása: https://progpater.blog.hu/2011/03/31/imadni_fogjatok_a_c_t_egy_emberkent_tiszta_szivbol

A c++-os kód, ugyan azt csinálja mint a C-s változata, csupán leegyszerűsödött maga a kód és ezáltal egyszerűbben olvashatóvá is vált. Először is a struktúrát átírjuk osztyállyá.

```
       class LZWBinFa
       {
```

```
        public:
        LZWBinFa (char b = '/'):betu (b), balNulla (NULL), jobbEgy (NULL)  ↩
           {};
        ~LZWBinFa () {};
```

Ezután jön a bemenet vizsgálata, ami annyiben különbözik a c-s verziótól, hogy van egy operátorunk, amellyel a bemnetet shifteljük a fába. Itt új csomópontnak a 'new'-val tudunk területet foglalni. Tehát ha nincs még 0/1-es csomópontunk a new-val foglalunk neki területet, majd az ujNullasGyermek/ujEgyesGyermek függvény segítségével adjuk a fához.

```
    void operator<<(char b)
    {
        if (b == '0')
        {
            // van '0'-s gyermeke az aktuális csomópontnak?
            if (!fa->nullasGyermek ()) // ha nincs, csinálunk
            {
                Csomopont *uj = new Csomopont ('0');
                fa->ujNullasGyermek (uj);
                fa = &gyoker;
            }
            else // ha van, arra lépünk
            {
                fa = fa->nullasGyermek ();
            }
        }
        else
        {
            if (!fa->egyesGyermek ())
            {
                Csomopont *uj = new Csomopont ('1');
                fa->ujEgyesGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->egyesGyermek ();
            }
        }
    }
```

Nézzük a private részt: A Csomopont értékét a konstruktorban '/'-re állítjuk, a gyermekei pedig 'NULL' értéket kapnak. A nullasGyermek és egyesGyermek a bal és jobb gyerekre mutató pointert adnak vissza. Az 'ujNullasGyermek' és az 'ujEgyesGyermek' a gyermekek pointerét állítja rá a paraméterként adott csomópontra.

```
private:
    class Csomopont
    {
    public:

        Csomopont (char b = '/'):betu (b), balNulla (0), jobbEgy (0)
        {
        };
        ~Csomopont ()
        {
        };

        Csomopont *nullasGyermek () const
        {
            return balNulla;
        }

        Csomopont *egyesGyermek () const
        {
            return jobbEgy;
        }

        void ujNullasGyermek (Csomopont * gy)
        {
            balNulla = gy;
        }

        void ujEgyesGyermek (Csomopont * gy)
        {
            jobbEgy = gy;
        }

        char getBetu () const
        {
            return betu;
        }

    private:

        char betu;
        Csomopont *balNulla;
        Csomopont *jobbEgy;
        Csomopont (const Csomopont &); //másoló konstruktor
        Csomopont & operator= (const Csomopont &);
    };
```

Végül nézzük a main-t: Itt a beolvasás történik egy while ciklus segítségével.

```
int
main ()
```

```
{
    char b;
    LZWBinFa gyoker, *fa = &gyoker;

    while (std::cin >> b)
    {
        if (b == '0')
        {
            // van '0'-s gyermeke az aktuális csomópontnak?
            if (!fa->nullasGyermek ()) // ha nincs, csinálunk
            {
                LZWBinFa *uj = new LZWBinFa ('0');
                fa->ujNullasGyermek (uj);
                fa = &gyoker;
            }
            else // ha van, arra lépünk
            {
                fa = fa->nullasGyermek ();
            }
        }
        else
        {
            if (!fa->egyesGyermek ())
            {
                LZWBinFa *uj = new LZWBinFa ('1');
                fa->ujEgyesGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->egyesGyermek ();
            }
        }
    }

    gyoker.kiir ();
    gyoker.szabadit ();

    return 0;
```

6.5. Mutató a gyökér

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

Megoldás videó:

Megoldás forrása: https://github.com/SylwerStone/Prog1/blob/master/z3a7agg.cpp

A gyökércsomópontot át kell írnunk mutatóvá.

```
                Csomopont *gyoker;
```

Majd a konstruktorban a fa pointert rá kell állítani a fa gyökerére.

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

Mivel a gyökér mostmár mutató típusú, így az összes helyen ahol pontokat használtunk, azokat nyilakkal kell felcserélnünk.

```
  }
    ~LZWBinFa ()
    {
        szabadit (gyoker->egyesGyermek ());
        szabadit (gyoker->nullasGyermek ());
        delete gyoker;
    }
```

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása: A másoló szemantika tömören annyi, hogy a kapott bináris fát értékül adja az eredeti fának, lemásolva annak összes értéket. A mozgató szemantika működése: az original fa gyökerét felcseréli annak a fának a gyökerével amelyet értékként megkapunk, és ezeknek a gyerekeit átállítja nullpointerre, hogy az ezután lefutó konstruktor miatt ne törlődjön az eredeti fa.

```
     LZWBinFa ( LZWBinFa && regi ) {
          std::cout << "LZWBinFa move ctor" << std::endl;

          gyoker.ujEgyesGyermek ( regi.gyoker.egyesGyermek() );
          gyoker.ujNullasGyermek ( regi.gyoker.nullasGyermek() );

          regi.gyoker.ujEgyesGyermek ( nullptr );
          regi.gyoker.ujNullasGyermek ( nullptr );

     }
     LZWBinFa& operator = (LZWBinFa && regi)
```

```
    {
        if (this == &regi)
            return *this;

        gyoker.ujEgyesGyermek ( regi.gyoker.egyesGyermek() );
        gyoker.ujNullasGyermek ( regi.gyoker.nullasGyermek() );

        regi.gyoker.ujEgyesGyermek ( nullptr );
        regi.gyoker.ujNullasGyermek ( nullptr );

        return *this;
    }
```

Az std::move függvény lényegében nem mozgat semmit, szimplán az átadott argumentumot tesszük vele jobbértékké és kikényszerítjük, hogy a mozgató értékadást használja. Aztán az új fát kiíratjuk

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

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

Az eredeti binfát már nem fogjuk tudni majd kiíratni, mivel annak gyökerét kinulláztuk.

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

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

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

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

Tanulságok, tapasztalatok, magyarázat. Ez a program egy ún. "hangyabojt" szimulál, vagyis a hangyákat és azok útjait. Az "Ant" class a hangya tulajdonságait tartalmazza: Kordináták (x, y), hova tart (dir)

```
class Ant
{

public:
    int x;
    int y;
    int dir;

    Ant(int x, int y): x(x), y(y) {

        dir = qrand() % 8;

    }

};

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

Az "Antwin" classben az ablak magasságát és szélességét, a hangyákat tartalamzó cellák magasságát szélességét tároljuk. A "keyPressEvent" a gomblenyomásokat kezeli, a "closeEVent" az ablak bezárásáért felelős, a "paintEvent" pedig a hangyák színeit alakítja.

```
class AntWin : public QMainWindow
{
    Q_OBJECT

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

    AntThread* antThread;

    void closeEvent ( QCloseEvent *event ) {

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

    void keyPressEvent ( QKeyEvent *event )
    {

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

    }

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

private:

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

```
public slots :
    void step ( const int &);

};
```

Az AntThread-ben tároljuk például a hangyák számát egy cellában, az evaporation mennyiségét, a pheromonok számát. Itt található meg a run és a finish funkció is. Itt taláható még az "isRunning" függvény, amely a nevéből következtethetően megnézi, hogy fut-e aztán visszaad egy igaz vagy hamis értéket. Pár privát függvény pl: a "newDir" (Új irány a hangyáknak), a "moveAnts" (Hangyák mozgatása) és a "setPheromone" (Pheromone mennyisége beállítása)

```
class AntThread : public QThread
{
    Q_OBJECT

public:
    AntThread(Ants * ants, int ***grids, int width, int height,
              int delay, int numAnts, int pheromone, int nbrPheromone,
              int evaporation, int min, int max, int cellAntMax);

    ~AntThread();

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

    void pause()
    {
        paused = !paused;
    }

    bool isRunnung()
    {
        return running;
    }

private:
    bool running {true};
    bool paused {false};
    Ants* ants;
    int** numAntsinCells;
    int min, max;
    int cellAntMax;
    int pheromone;
    int evaporation;
    int nbrPheromone;
```

```
    int ***grids;
    int width;
    int height;
    int gridIdx;
    int delay;

    void timeDevel();

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

signals:
    void step ( const int &);

};
```

7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html#id564903

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: https://gitlab.com/kissmarcell98/bhax/tree/master/attention_raising/QT

Tanulságok, tapasztalatok, magyarázat... Az életjáték John Conway nevéhez fűződik, aki a Cambridge egyetem egyik matematikusa volt. A játék egy ún. 'nullszemélyes' játék, tehát a játékos feladata szimplán annyiben kimerül, hogy egy kezdőalakzatot megad majd figyeli a történéseket. A "játék" lépéseinek eredményét a számítógép számolja, tehát a 'játékosnak' úgymond semmi teendője nincs a kezdőalakzat kiválasztásán kívül. A játék szabályai: A sejt ha 2 vagy 3 szomszédja van túléli a kört. A sejt, ha kettőnél kevesebb, vagy háromnál több szomszédja van akkor elpusztul. Új sejt akkor keletkezik, ha egy üres cellának pontosan 3 szomszédja van. A 'sejtablak.h'-ban és a 'sejtablak.cpp'-ben megtalálható a SejtAblak class, ami azért felelős, hogy a programunk kirajzolódjon.

```
#include <QtGui/QMainWindow>
#include <QPainter>
#include "sejtszal.h"

class SejtSzal;

class SejtAblak : public QMainWindow
{
    Q_OBJECT

public:
    SejtAblak(int szelesseg = 100, int magassag = 75, QWidget *parent = 0);
    ~SejtAblak();
    // Egy sejt lehet élõ
    static const bool ELO = true;
    // vagy halott
    static const bool HALOTT = false;
    void vissza(int racsIndex);

protected:
    // Két rácsot használunk majd, az egyik a sejttér állapotát
    // a t_n, a másik a t_n+1 idõpillanatban jellemzi.
    bool ***racsok;
    // Valamelyik rácsra mutat, technikai jellegû, hogy ne kelljen a
    // [2][][]-ból az elsõ dimenziót használni, mert vagy az egyikre
    // állítjuk, vagy a másikra.
    bool **racs;
    // Megmutatja melyik rács az aktuális: [rácsIndex][][]
    int racsIndex;
    // Pixelben egy cella adatai.
    int cellaSzelesseg;
    int cellaMagassag;
    // A sejttér nagysága, azaz hányszor hány cella van?
    int szelesseg;
    int magassag;
    void paintEvent(QPaintEvent*);
    void siklo(bool **racs, int x, int y);
    void sikloKilovo(bool **racs, int x, int y);

private:
    SejtSzal* eletjatek;

};
```

A ’sejtszal.cpp’ és ’sejtszal.h’-ban pedig a SejtSzal class található meg, ami a szabályokat tartalmazza. Tehát ez alapján jön létre új sejt, hal meg egy, vagy éppen marad életben

```
#ifndef SEJTSZAL_H
#define SEJTSZAL_H
```

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

class SejtAblak;

class SejtSzal : public QThread
{
    Q_OBJECT

public:
    SejtSzal(bool ***racsok, int szelesseg, int magassag,
             int varakozas, SejtAblak *sejtAblak);
    ~SejtSzal();
    void run();

protected:
    bool ***racsok;
    int szelesseg, magassag;
    // Megmutatja melyik rács az aktuális: [rácsIndex][][]
    int racsIndex;
    // A sejttér két egymást követõ t_n és t_n+1 diszkrét idõpillanata
    // közötti valós idõ.
    int varakozas;
    void idoFejlodes();
    int szomszedokSzama(bool **racs,
                        int sor, int oszlop, bool allapot);
    SejtAblak* sejtAblak;

};
```

A sikókilövő pedig szintén a sejtablak.cpp-ben található meg. Itt egyesével rajzolja ki a sejteket a megadott koordinátákra.

```
void SejtAblak::siklоKilovo(bool **racs, int x, int y) {

    racs[y+ 6][x+ 0] = ELO;
    racs[y+ 6][x+ 1] = ELO;
    racs[y+ 7][x+ 0] = ELO;
    racs[y+ 7][x+ 1] = ELO;

    racs[y+ 3][x+ 13] = ELO;

    racs[y+ 4][x+ 12] = ELO;
    racs[y+ 4][x+ 14] = ELO;

    racs[y+ 5][x+ 11] = ELO;
    racs[y+ 5][x+ 15] = ELO;
    racs[y+ 5][x+ 16] = ELO;
    racs[y+ 5][x+ 25] = ELO;
```

```
    racs[y+ 6][x+ 11] = ELO;
    racs[y+ 6][x+ 15] = ELO;
    racs[y+ 6][x+ 16] = ELO;
    racs[y+ 6][x+ 22] = ELO;
    racs[y+ 6][x+ 23] = ELO;
    racs[y+ 6][x+ 24] = ELO;
    racs[y+ 6][x+ 25] = ELO;

    racs[y+ 7][x+ 11] = ELO;
    racs[y+ 7][x+ 15] = ELO;
    racs[y+ 7][x+ 16] = ELO;
    racs[y+ 7][x+ 21] = ELO;
    racs[y+ 7][x+ 22] = ELO;
    racs[y+ 7][x+ 23] = ELO;
    racs[y+ 7][x+ 24] = ELO;

    racs[y+ 8][x+ 12] = ELO;
    racs[y+ 8][x+ 14] = ELO;
    racs[y+ 8][x+ 21] = ELO;
    racs[y+ 8][x+ 24] = ELO;
    racs[y+ 8][x+ 34] = ELO;
    racs[y+ 8][x+ 35] = ELO;

    racs[y+ 9][x+ 13] = ELO;
    racs[y+ 9][x+ 21] = ELO;
    racs[y+ 9][x+ 22] = ELO;
    racs[y+ 9][x+ 23] = ELO;
    racs[y+ 9][x+ 24] = ELO;
    racs[y+ 9][x+ 34] = ELO;
    racs[y+ 9][x+ 35] = ELO;

    racs[y+ 10][x+ 22] = ELO;
    racs[y+ 10][x+ 23] = ELO;
    racs[y+ 10][x+ 24] = ELO;
    racs[y+ 10][x+ 25] = ELO;

    racs[y+ 11][x+ 25] = ELO;

}
```

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: https://github.com/nbatfai/esport-talent-search

Tanulságok, tapasztalatok, magyarázat: A BrainB Benchmark azt teszteli, hogy mennyire tudunk figyelni a karakterünkre, mennyi idő alatt veszítjük el, illetve ha elveszítettük, mennyi idő alatt találjuk meg újból. A

feladat annyi, hogy a kurzort rajta kell tartanunk a saját "karakterünkön", ám eközben egyre több másik "new" karakter jelenik meg, így nehezedik egyre jobban a dolgunk. Ha elveszítjük a karaktert, akkor kevesebb új karakter jelenik meg, hogy könnyebb legyen megtalálni az eredetit. A folyamat 10 percig tart, majd a végén megkapod az eredményt. Ezeket az eredményeket összehasonlítva tudjuk összehasonlítani egyes egyének képességeit pl. Esport játékokan.

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.2. Szoftmax R MNIST

R

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.4. Deep dream

Keras

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.5. Robotpszichológia

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Megoldás forrása:

9.2. Weizenbaum Eliza programja

Éleszd fel Weizenbaum Eliza programját!

Megoldás videó:

Megoldás forrása:

9.3. 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...

9.4. 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...

9.5. Lambda

Hasonlítsd össze a következő programokat!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9.6. Omega

Megoldás videó:

Megoldás forrása:

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.

10. fejezet

Helló, Berners-Lee!

10.1. Java és C++ összehasonlítása

C++: Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven Java: Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II.

A Javát 1991-ben fejlesztették, amely a Sun Microsystems nevéhez fűződik, fő fejlesztője James Gosling, akit a java atyjaként is emlegetnek. Először a nyelvet otthoni elektromos eszközök (pl TV) programozására akarták használni, de ez túlmutatott az akkori technológián. A nyelv szintaxisát direkt a C és C++-hoz hasonlóra tervezték, hogy akik ismerték ezt a két nyelvet könnyeb legyen eligazodniuk benne. A nyelv elnevezése egy kávé nevéből eredeztethető.

Hasonlóságok:

- Nagyon hasonló a két nyelv szintaxisa
- Mindkét nyelvben 'main' function-ben kezdődik a program végrahájtása
- Az elágazások és loopok szintén hasonlítanak
- A kommentelés a két nyelvben azonos módon működik
- Több szálon tudunk futtatni dolgokat mindkét nyelvben

Természetesen eltérések is vannak:

Általánosságban elmondható, hogy a Java inkább a platformfüggetlenséget és implementációindependenciát részesíti előnyben, a C++ nál minden a programozóra van bízva, és a fő célja a maximális hatékonyság elérése.

Nagy különbség, hogy a C++-al ellentétben, a Javában nincsenek külön objektumok és a címükre mutató mutatók, hanem mindig referenciákon keresztül érjük el őket.

A hordozhatósága is eltérő a két nyelvnek. A C++ források hordozhatóak, viszont a már lefordított fájlok nem. Ezzel szemben a Java jobb hordozhatósággal rendelkezik, mivel nem csak a forrásokat, hanem a már lefordított programok is futtathatóak más gépen. Ezt egy Java virtuális gép alkalmazása teszi lehetővé.

A memóriakezelés is máshogy megy végbe. A C++-nál három memóriakezelés létezik : statikus, automatikus, és dinamikus. A Javaban viszont minden memóriaelérés hivatkozásokon keresztül történik.

C++-ban az osztálydeklarációk a végére pontosvesszőt kell raknunk, Java-ban viszont nem.

10.2.

Python: Forstner Bertalan, Ekler Péter, Kelényi Imre: Bevezetés a mobilprogramozásba. (35-51. oldalak)

A Python egy általános célú programozás nyelv, amit Guido von Rossum 1990-ben alkottott meg és 1991-ben hozta nyilvánosságra. A Python egy objektum orientált, platformfüggetlen nyelv. Leginkább prototípusok megírására és tesztelésére használják, leginkább mobilalkalmazásoknál, de egyszerűbb programok megírására is alkalmas. Python-ban, a C, és Java nyelvekkel ellentétben, nem kell külön lefordítanunk a programot, amit írunk.Elég a Python forrást megadni az interpreternek, és lefut a program.

A C alapú nyelvek után a Python szintaxisa elsőre fura lehet, ugyanis a Python nyelv behúzásalapú nyelv. Ez azt jelenti, hogy az állításokat a különböző szintű behúzásokkal tudjuk csoportosítani. Nem kell a kapcsos zárójeleket, vagy a begin-end kifejezést használni. Fontos, hogy a behúzásoknak egységesnek kell lenniük, mindenhol vagy TAB-ot vagy SPACE-t használjunk. A kommentelés is eltérő a C-alapú nyelvekné megszokottnál, // helyet a # jelet használunk a kommentek írására. Pontosvesszőt nem kell használnunk, mert minden utasítás csak az adott sor végéig tart. Ha ez mondjuk nem fér ki egy sorba, akkor ezt tudjuk a Pythonban egy '\' jellel jelezni, hogy a következő sorban folytatódik az utasítás. A Pythonban előforduló adattípusok lehetnek: számok, sztringek, ennesek (tuples, n-es), listák vagy szótárak (dictionaries). tt is létezik kivételkezelés, melynek a try és except a kulcsszavai. A Pythonban is megtalálható pl az if elágazás és a while vagy for ciklus is.

11. fejezet

Helló, Arroway!

11.1. OO szemlélet

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Az objektumelvű programozás (object-oriented programming vagy OOP) az objektumok fogalmán alapuló programozási paradigma. Mind a Java mind a C++ támogatja az objektum orientált programozást. A fenti feladat szerves része az objektumorientáltság.

A módosított polártranszformációs normális generátor véletlen számokat állít elő. Ezeket egy matematikai algoritmussal hozzuk létre (szóval nem teljesen véletlen). Nézzük a program Java kódját:

```
public class PolárGenerátor {

  boolean nincsTárolt = true;
  double tárolt;

  public PolárGenerátor() {
    nincsTárolt = true;
  }

  public double következő() {
    if (nincsTárolt) {
      double u1, u2, v1, v2, w;
      do {
        u1 = Math.random();
        u2 = Math.random();
        v1 = 2*u1-1;
```

```
        v2 = 2*u2-1;
        w = v1*v1+v2*v2;
      } while (w > 1);
      double r = Math.sqrt((-2*Math.log(w)) / w);
      tárolt = r*v2;
      nincsTárolt = !nincsTárolt;
      return r*v1;
    } else {
      nincsTárolt = !nincsTárolt;
      return tárolt;
    }
  }

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

Mivel a Java egy teljesen objektum-orientált nyelv, mindennek muszáj osztályban lennie. A 'nincsTárolt' változó mutatja, hogy van-e atm még felhasználatlan előállított véletlen szám. A táralandó értékek a 'tárolt' változóba kerülnek

Következzen program lényegi része, egy double típusú szám a visszatérési értéke. Ha nincs tárolt szám, akkor előállít egyet. A nincsTárolt értéke !nincstárolt lesz, vagyis hamis, tehát van tárolt, majd visszatéríti amit kell. A következő hívásnál tehát az else ággal megyünk tovább, mivel már van tárolt értékünk. Itt is megfordítjuk a nincsTárolt változó értékét , majd a tárolt számot adja a függvény.

A főprogramban szimplán az történik, hogy létrehozunk egy példányt a PolárGenerátor osztályból, majd a for ciklusban tízszer meghívjuk a következő függvényt és kiíratjuk a generált számot.

Nézzük a kódot C++ nyelven is. Tulajdonképpen csak a nyelvi különbségekben különbözik a kód.

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>

class PolarGen
{
public:
  PolarGen ()
  {
    nincsTarolt = true;
    std::srand (std::time (NULL));
  }
   ~PolarGen ()
  {
  }
  double kovetkezo ();
```

```
private:
  bool nincsTarolt;
  double tarolt;

};
double
PolarGen::kovetkezo ()
{
  if (nincsTarolt)
    {
      double u1, u2, v1, v2, w;
      do
  {
    u1 = std::rand () / (RAND_MAX + 1.0);
    u2 = std::rand () / (RAND_MAX + 1.0);
    v1 = 2 * u1 - 1;
    v2 = 2 * u2 - 1;
    w = v1 * v1 + v2 * v2;
  }
      while (w > 1);

      double r = std::sqrt ((-2 * std::log (w)) / w);

      tarolt = r * v2;
      nincsTarolt = !nincsTarolt;

      return r * v1;
    }
  else
    {
      nincsTarolt = !nincsTarolt;
      return tarolt;
    }
}
int
main (int argc, char **argv)
{
  PolarGen pg;

  for (int i = 0; i < 10; ++i)
    std::cout << pg.kovetkezo () << std::endl;

  return 0;
}
```

A következő és a main gyakorlatilag teljesen ugyan azok, ahogy látjuk, csak a C++ kódban már szerepel konstruktor és destruktor is.

11.2. Homokozó

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.3. Gagyi

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

Az ismert formális **„while (x <= t && x >= t && t != x);”** tesztkérdéstípusra adj a szokásosnál (miszerint x, t az egyik esetben az objektum által hordozott érték, a másikban meg az objektum referenciája) „mélyebb” választ, írj Java példaprogramot mely egyszer végtelen ciklus, más x, t értékekkel meg nem! A példát építsd a JDK Integer.java forrására, hogy a 128-nál inkluzív objektum példányokat poolozza!

```
class Gagyi
{
  public static void main(String[] args)
  {
    Integer t = 127;
    Integer x = 127;

    while(x <= t && x>=t && t != x)
      System.out.println("Oo");
  }
}
```

A fent látható kód nem végtelen ciklus, mivel a while ciklus hamis lesz. Az első két összehasonlításnál a két objektumértéke kerül összehasonlításra, ami egyértelmű, hogy meg fog egyezni, az éselés igaz lesz, de t!=x hamis lesz, mivel t=x referenciaként kell értelmezni, és ez igaz. A Java 128 alatti Integer objektumok esetén gyorsítótáraz, vagyis az objektumok referenciája ugyanaz lesz minden példánynál.

A következő kód már végtelen ciklus:

```
class Gagyi
```

```
{
  public static void main(String[] args)
  {
    Integer t = 128;
    Integer x = 128;

    while(x <= t && x>=t && t != x)
      System.out.println("Oo");
  }
}
```

11.4. Yoda

Írjunk olyan Java programot, ami java.lang.NullPointerEx-el leáll, ha nem követjük a Yoda conditions-t!

Forrás: https://en.wikipedia.org/wiki/Yoda_conditions

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

A Yoda conditions egy olyan programozási módszer, ahol egy feltétel két része a megszokottal ellentétes módon van. A kifejezés konstans része szerepel a egyenlőségjel bal oldalán. A neve Yoda mestertől származik, aki szintén a megszokottal ellentétben "fordítva" beszélt.

Egy hagyományos feltétel így néz ki:

```
    if (value==42) {/*...*/}
```

Ugyanez a Yoda conditions-szel:

```
    if (42==value) {/*...*/}
```

Ez egy olyan Java program, ami java.lang.NullPointerEx-el leáll, ha nem követjük a Yoda conditions-t:

```
    class yoda{
      public static void main(String[] args){
        String myString = null;
        if (myString.equals("valami")) { System.out.println("valami"); }
      }
    }
```

Ez a programrész java.lang.NullPointerEx-el leáll, Yoda conditions-t használva viszont már nem:

```
    class yoda{
      public static void main(String[] args){
        String myString = null;
        if ("valami".equals(myString)) { System.out.println("Ha ezt nem ↩
           látod, akkor jó"); }
```

```
        }
    }
```

11.5. Kódolás from scratch

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

12. fejezet

Helló, Liskov!

12.1. Liskov helyettesítés sértése

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

A Liskov elv, vagy teljes nevén Liskov helyettesítés elve, az objektum orientált programozás egy elve, mely szerint egy programban, ha S egy származtatott típus a T-ből (S és T itt objektumokat jelölnek), akkor a T típusú objektumok helyettesíthetőek S típusú objektumokkal anélkül, hogy módosítanánk a program kívánatos tulajdonságait(pl:helyesség, feladatok végrehalytása).

Példa a Liskov elv sértésére:

```
// ez a T az LSP-ben
class Madar {
public:
     virtual void repul() {};
};

// ez a két osztály alkotja a "P programot" az LPS-ben
class Program {
public:
     void fgv ( Madar &madar ) {
          madar.repul();
     }
};

// itt jönnek az LSP-s S osztályok
class Sas : public Madar
{};

class Pingvin : public Madar // ezt úgy is lehet/kell olvasni, hogy a ↩
   pingvin tud repülni
{};
```

```
int main ( int argc, char **argv )
{
     Program program;
     Madar madar;
     program.fgv ( madar );

     Sas sas;
     program.fgv ( sas );

     Pingvin pingvin;
     program.fgv ( pingvin ); // sérül az LSP, mert a P::fgv röptetné a  ↩
        Pingvint, ami ugye lehetetlen.

}
```

A fenti programban, hibásan úgy definiáltuk a Madar class-t, hogy tartalmazza a repul() metódust(ez alapvetően hibás, hiszen nem minden madár tud repülni). Ennek következménye hogy az összes Madar-ból származtatott class is tartalmazni fogja a repul() funkciót, hibába akkor ütközünk amikor a Pingvin class-t is a madárból származtatjuk: A pingvin madár viszont nem tud repülni. Megoldás a problémára: jobb tervezés, amikor a Madar class-t definiáltuk jobban át kellet volna gondolnunk a felépítését.

A program Jávába átírva:

```
class liskov{

  // ez a T az LSP-ben
   static class Madar {
  public void repul() {System.out.println("Repülök");}
  };

  // ez a két osztály alkotja a "P programot" az LPS-ben
   static class Program {
  public void fgv ( Madar madar ) {
            madar.repul();
       }
  };

  // itt jönnek az LSP-s S osztályok
   static class Sas extends Madar
  {};

   static class Pingvin extends Madar // ezt úgy is lehet/kell olvasni,  ↩
      hogy a pingvin tud repülni
  {};

  public static void main(String args[])
  {
     Program program = new Program();
     Madar madar= new Madar();
     program.fgv ( madar );
```

```
     Sas sas= new Sas();
     program.fgv ( sas );

     Pingvin pingvin= new Pingvin();
     program.fgv ( pingvin ); // sérül az LSP, mert a P::fgv röptetné a ↩
        Pingvint, ami ugye lehetetlen.

  }
}
```

12.2. Szülő-gyerek

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

C++ kóddal:

```
#include <iostream>
using namespace std;

// szulo osztály, ebből fogom származtatni a leszarmaztatott osztályt.
class szulo {
public:
       void uzenet() {cout<<"Az ős üzenete\n";}
};

//ez a függvény kapja meg paraméterként a szulo&
void fgv ( szulo& os ) {
          os.uzenet();
}


// gyerek osztály szulo-ből származtatva
class gyerek : public szulo
{
  public: void uzenet() {cout<<"Az gyerek üzenete\n";}
  };



int main ( int argc, char **argv )
{
     szulo* szulo1 = new gyerek();
     szulo1->uzenet(); //az ős üzenete

     gyerek gyerek1;
```

```
     fgv(gyerek1); //az ős üzenete
}
```

A szulo és gyerek osztályunk, a gyerek a szulo-ből származtatva, valamint fgv() nevű függvényünk, ami egy szulo referenciát vár paraméterűl. C++ ban csak akkor dinamikus a kötés ha a viselkedés virtuálisa van állítva(virtual kulcsszó), egyébbként referencián vagy pointeren keresztűl csak az ős üzeneteit tudjuk elérni.

A kód első részében gyereknek foglalunk helyet, majd erre egy szulo pointer állítunk, majd a meghívjuk a szulo1 által mutatott osztály üzenetét, ami a szulo üzenete lesz. A következő példában pedig a szulo referenciát váró függvénynek adjuk a gyerek típusú objektumot, így szintén a szulo üzenetét fogjuk elérni.

Java kód:

```
class szarmaz
{

  public class szulo
  {
    public void uzenet ()
    {
      System.out.println ("Az ős üzenete");
    }
  };


  public void fgv (szulo szul)
  {
    szul.uzenet ();
  }


  public class gyerek extends szulo
  {
    public void uzenet ()
    {
      System.out.println ("A gyerek üzenete");
    }
  };


  public static void main (String args[])
  {
    szarmaz szarm = new szarmaz ();
    szulo szul = szarm.new gyerek ();
    szul.uzenet ();

    gyerek gyer = szarm.new gyerek ();
    szarm.fgv (gyer);
  }
}
```

Mivel Javában mindig dinamikus a kötés, ezért a fenti kóddal, itt már a gyerek üzeneteit érjük el:

```
class szarmaz
{

  public class szulo
  {
    public void uzenet ()
    {
      System.out.println ("Az ős üzenete");
    }
  };


  public void fgv (szulo szul)
  {
    szul.uzenet ();
  }


  public class gyerek extends szulo
  {
    public void uzenet ()
    {
      System.out.println ("A gyerek üzenete");
    }
  };


  public static void main (String args[])
  {
    szarmaz szarm = new szarmaz ();
    szulo szul = szarm.new gyerek ();
    szul.uzenet ();

    gyerek gyer = szarm.new gyerek ();
    szarm.fgv (gyer);
  }
}
```

Viszont a gyerek által hordozott új üzeneteket már nem hívhatjuk meg:

12.3. Anti OO (Passz)

A BBP algoritmussal 4 a Pi hexadecimális kifejtésének a 0. pozíciótól számított 10 6, 107, 108 darab jegyét határozzuk meg C, C++, Java és C# nyelveken és vessük össze a futási időket! https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apas03.html#id561066

12.4. Hello, Android! (Passz)

Élesszük fel az SMNIST for Humans projektet! https://gitlab.com/nbatfai/smnist/tree/master/forHumans/SMNISTforHumansExp3/app/src/main Apró módosításokat eszközölj benne, pl. színvilág.

12.5. Ciklomatikus komplexitás

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

A ciklomatikus komplexitás egy szoftvermetrika, amely egy adott szoftver forráskódja alapján határrozza meg annak komplexitását egy konkrét számértékben kifejezve. Az érték kiszámítása a gráfelméletre alapul. A forráskódban az elágazásokból felépülő gráf pontjai, és a köztük lévő élek alapján számítható az alábbi módon:

```
M = E - N + 2P
```

Ahol:

- E: A gráf éleinek száma
- N: A gráfban lévő csúcsok száma
- P: Az összefüggő komponensek száma

Átfogalmazva szoftverekre: a gráfot az adott függvényben lévő utasítások alkotják, él akkor van 2 utasítás között, ha az egyik után a másik azonnal végrehajtódhat, így a metrika közvetlenül számolja a lineárisan független útvonalakat a forráskódon keresztül.

Ciklomatikus komplexitást a Lizard nevű kód komplexitás analizálóval fogom kiszámolni, ami elérhető a következő honlapon keresztűl:

www.lizard.ws/

Használata egyszerű: válasszuk ki hogy milyen nyelvben írt kódot szeretnénk analizálni majd másoljuk be a kódot az ablakba, ezután kattintsunk az Analyze gombra, ezzel elindítjuk a számítást. A jobb oldali ablakban a Complexity nevű oszlop jelzi a ciklomatikus komplexitás értékeket.

Analizáljunk először egy egyszerűbb programot:

12.1. ábra.

Majd egy bonyolultabb analizálása:

12.2. ábra.

13. fejezet

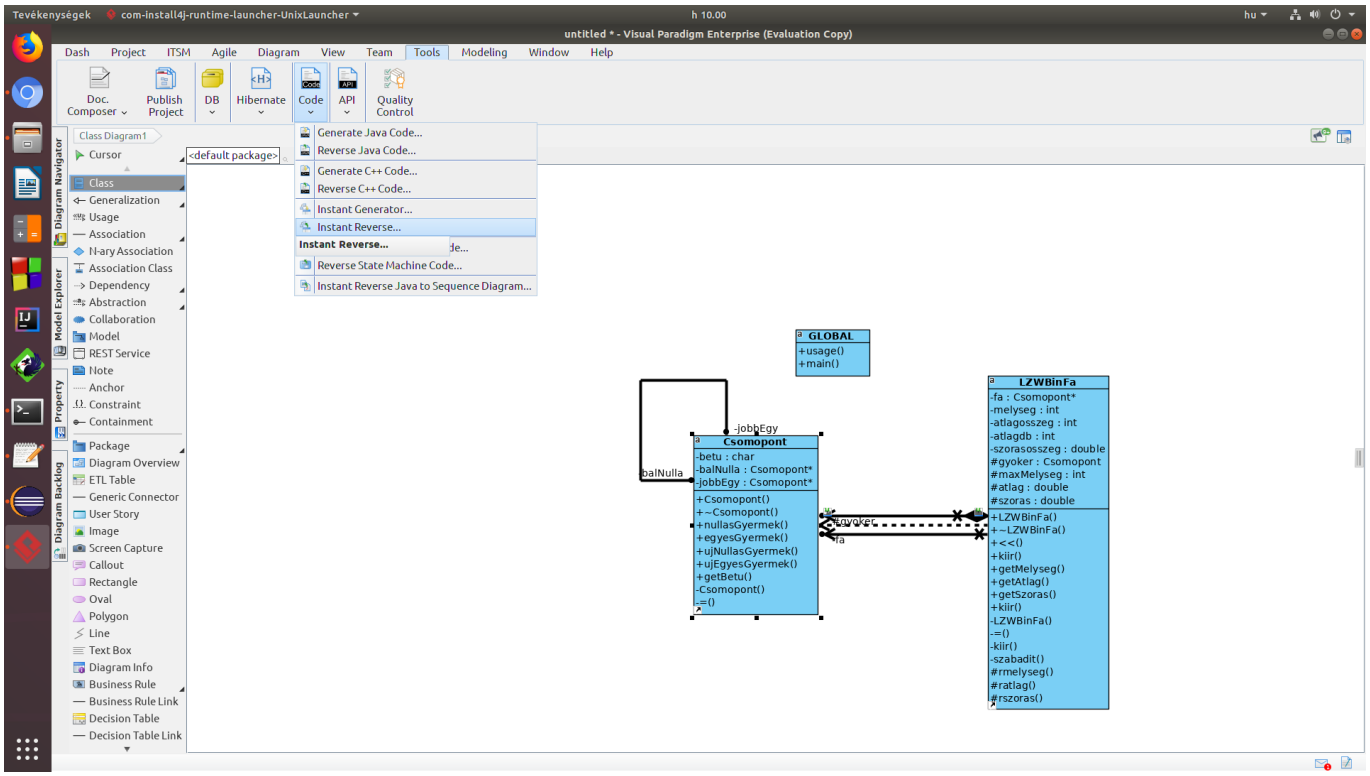
Helló, Mandelbrot

13.1. Reverse engineering UML osztálydiagram

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

UML diagrammokkal egy kicsit már ismerkedtünk a könyv első fejezetében is, de most még részletessebben megnézzük, mire is jó. Egy UML osztálydiagramot fogunk generálni. Ez egy olyan ábra, amin szemléltethetjük egy program(részlet) osztályainak szerkezetét, a felépítésüket (metódusok, attribútumok), illetve a közöttük fennálló viszonyt. A diagrammon láthatjuk, hogy egy adott class/metódus/attribútum láthatósága milyen (private/public...), erre általában a +/- jelek szolgálnak.

Én a feladat megoldásához a Visual Paradigm nevű szoftvert választottam, ami remek megoldást nyújt, rengeteg funkcióval, és a használata sem túl bonyolult (https://www.visual-paradigm.com/). A program installálása után mindössze annyi a dolgunk, hogy megkeressük a forrásfájlt, majd néhány kattintással kiválasszuk a nekünk megfelelő feladatot (instatnt reverse, ez a kódból készít egy UML osztálydiagramot):



13.1. ábra. UML diagram készítése

A diagramon láthatjuk, hogy az osztályok viszonya is fel van tüntetve, például ha a kódban a gyöker tagként szerepel, ez a diagramon is látszik, megfelelő szintaktikájú nyilakat láthatunk. A kész diagram:

13.2. ábra. UML diagram

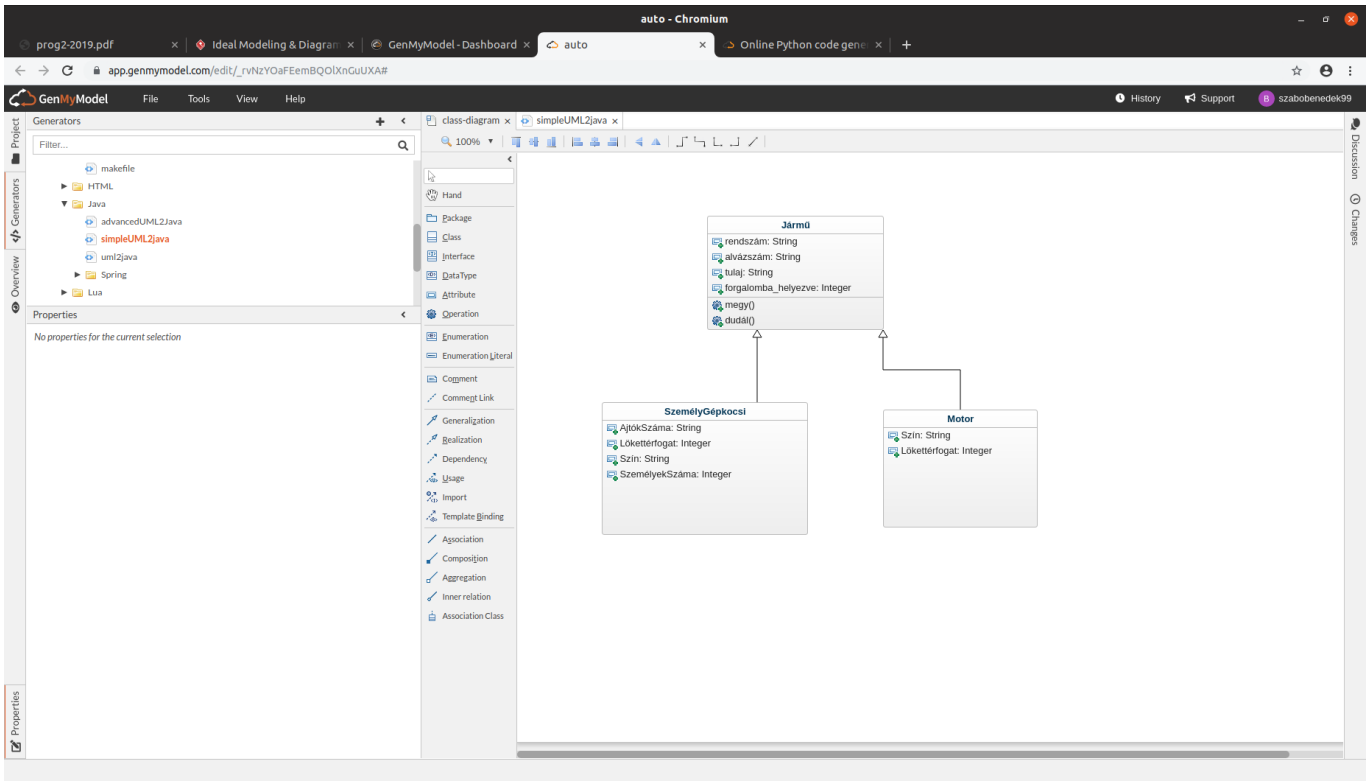
Mi a haszna egy ilyen diagramnak? A programunk tervezésénél hasznos lehet, ha jól átlátjuk a szerkezetét, főleg, ha az olyan bonyolult, hogy nem tudunk mindent egyszerre fejben tartani, illetve később, miután egy darabig nem foglalkoztunk a kóddal, egy ilyen diagramra ránézve egyszerűbb lehet megérteni a programot, mint magát a kódot nézve. Vannak olyan programozók, akik inkább csak a forráskóddal szeretnek dolgozni, de másoknak jól jöhet egy ilyen diagram.

13.2. Forward engineering UML osztálydiagram

UML-ben tervezzünk osztályokat és generáljunk belőle forrást!

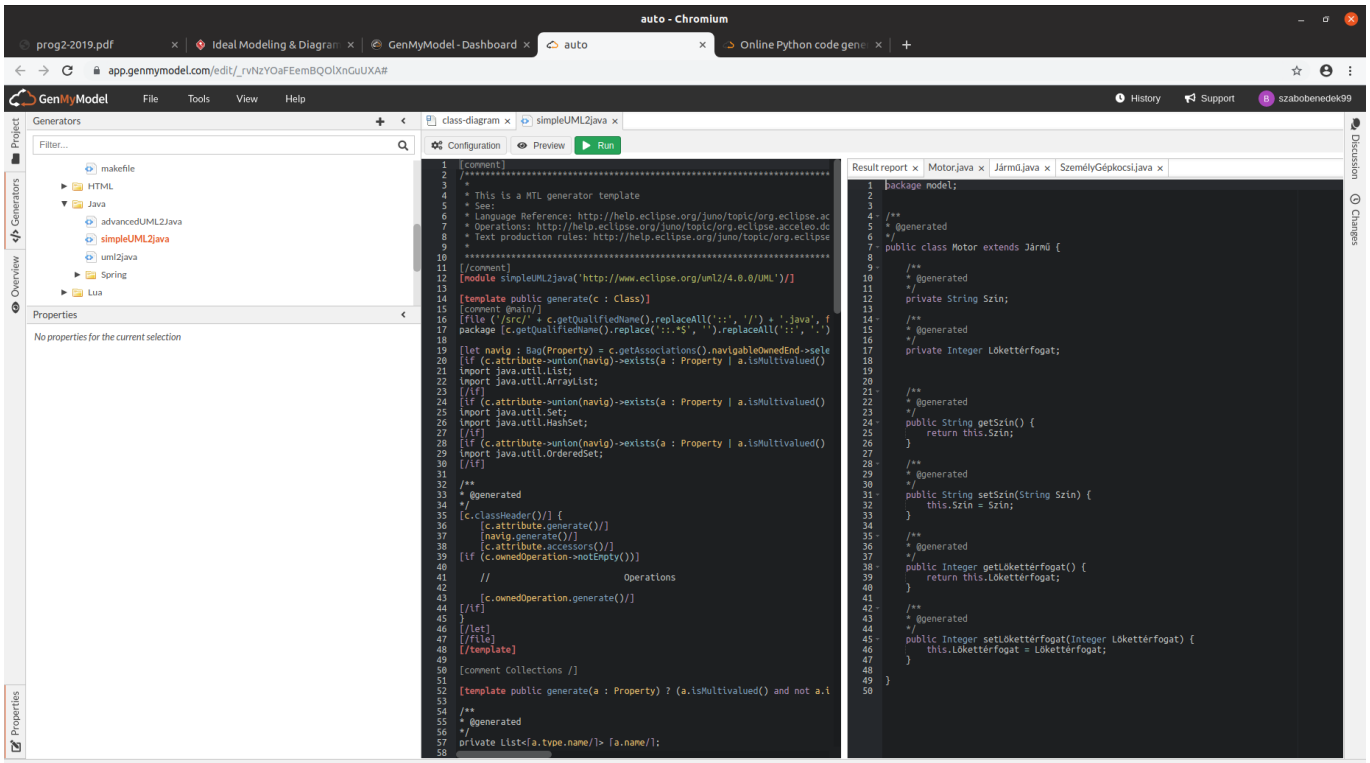
Ez a feladat gyakorlatilag az előző feladat megfordítása, most készítünk egy UML diagramot, és ebből generálunk kódot. Ezúttal egy másik szoftvert választottam: GenMyModel (https://genmymodel.com/), mivel ezzel online is lehet dolgozni. Rengeteg dolgot tud, és a felülete is felhasználóbarát.

Először is elkészítettem a diagramot:



13.3. ábra. UML diagram készítése

Ezután pedig a bal oldalon látható "Generators" fülre kattintva találjuk a felületet, ahova egyrészt tehetünk saját generátor kódot, másrészt pedig számos előre elkészített generátor közül választhatunk, rengeteg nyelven. Én a Java kódot választottam, a "Preview" gomra kattintva láthatjuk is a generált kódot:



13.4. ábra. Kód generálása

Láthatjuk, hogy getter és setter metódusokat is kaptunk egyből. Ha sok függvényt kell írnunk, aminek a tartalma viszonylag egyszerű, egy ilyen generálással sokat tudunk gyorsítani a munkánkon. Néhány részlet az elkészült kódokból (a teljes kódok a repóban találhatók):

```
package model;

/**
* @generated
*/
public class Jármű {

    /**
    * @generated
    */
    private String rendszám;

    /**
    * @generated
    */
    private String alvázszám;
```

```
    public String getRendszám() {
        return this.rendszám;
    }
```

```
    /**
    * @generated
    */
    public String setRendszám(String rendszám) {
        this.rendszám = rendszám;
    }
```

```
public class Motor extends Jármű {

    /**
    * @generated
    */
    private String Szín;
```

```
    public String getSzín() {
        return this.Szín;
    }

    /**
    * @generated
    */
    public String setSzín(String Szín) {
        this.Szín = Szín;
    }
```

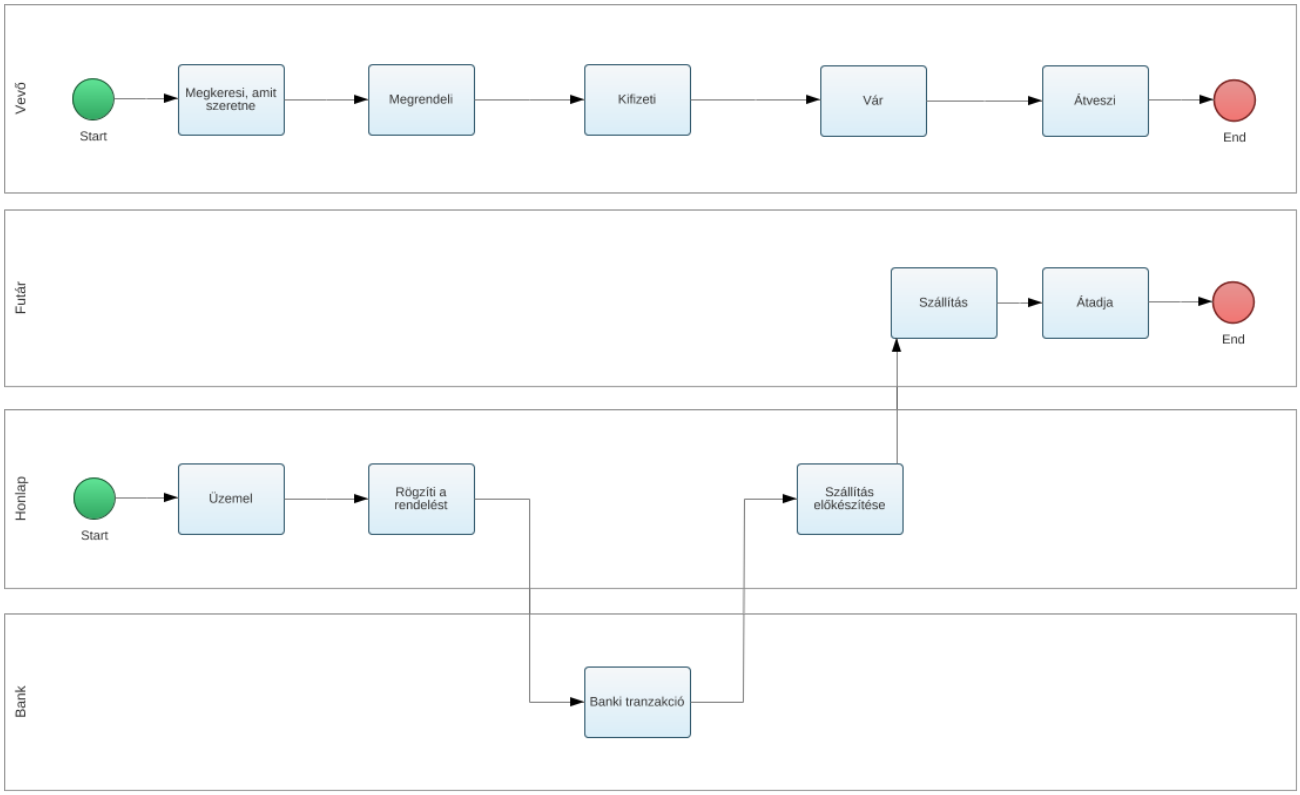
Természetesen tartalmilag nem tudjuk ilyen módon elkészíteni a teljes kódot, de nagy segítség lehet egy ilyen eszköz, ha sok a "mechanikus" része a kódolásnak, illetve azért a kifejezőereje se gyenge, az UML készítésekor nagyon sok lehetőségünk van (generalizáció, kompozició, aggregáció, láthatóság...).

13.3. BPMN

Rajzoljunk le egy tevékenységet BPMN-ben!

https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog2_7.pdf (34-47 fólia)

BPMN: Business Process Model and Notation, az UML aktivitás diagramjához hasonló grafikus modellező nyelv, mely segítségével folyamatokat, tevékenységeket modellezhetünk le, ezzel segítve az átláthatóságot, megértést. Mint a neve is sugallja, üzleti célú modellek készítésére használatos leginkább, de persze ettől el lehet rugaszkodni, bármilyen más témába. Én is egy héptkönapi példát választottam:



13.5. ábra. Kód generálása

A képen egy tetszőleges termék online rendelésének folyamatáról készítettem egy általános modellt. A vevő először keresgél az interneten, majd ha egy termék mellett dönt, megrendeli azt, majd kifizeti. Eközben a honlap rögzíti a rendelést, illetve átirányít egy bank felületére, ahol elvégezzük a tranzakciót. Ezután a vevő vár, míg a honlapot üzemeltető cég előkészíti a csomagját, illetve a futár kiszállítja azt. Végül a vevő átveszi a csomagot.

Ha esetleg fontos lenne számunkra, hogy pontosan mit és hogyan is kell használni, a következő linken megnézhetjük a szabványban: https://www.omg.org/spec/BPMN/2.0/PDF/

13.4. Reverse engineering UML osztálydiagram

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

14. fejezet

Helló, Chomsky!

14.1. Encoding

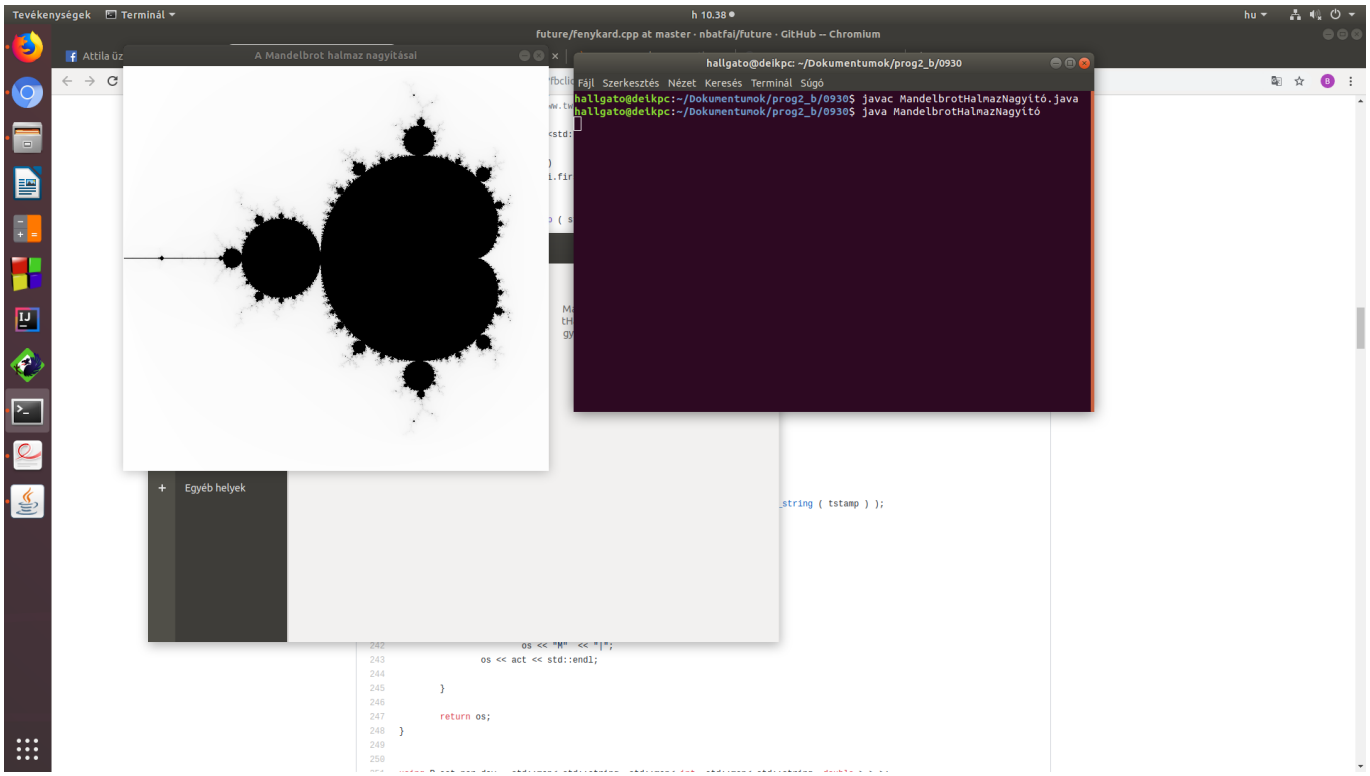
Fordítsuk le és futtassuk a Javat tanítok könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezetes betűket! https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/adatok.html

A feladathoz szükséges források a repómban megtalálható, még az első felvonás feladatai között, vagy akár a feladatleírásban található linken. A feladat nem túl nehéz: Egy program fordítása és futtatása. Az az egy dolog tűnhet fel, hogy a fájl nevében magyar, ékezetes betűk is szerepelnek: MandelbrotHalmazNagyító.

A többi programnyelv használatakor ez azért jelentős problémát okozott volna, de Javanal ez nem így van. Ha a forráskódot megnyitjuk, láthatjuk, hogy nem csak a kommentekben találhatók ékezetes betűk, hanem a forráskód többi részében is, leginkább változó-, vagy függvénynevekben. A Java programok forráskódjában teszőleges Unicode karakterek szerepelhetnek. A nyelv, a fordító, és a futtató környezet mind ezt a karakterkészletet használja.

Amennyiben nem magyar kiosztás alatt dolgozunk, hamar hibákba ütközünk. Ezeket egyszerűen ki lehet küszöbölni, ha windows alatt lett kódolva, a fordításnál használjuk:**-encoding windows-1252**

Magyar kiosztás alatt simán ha beírjuk a szokásos parancsokat, működik minden:



14.1. ábra. Futtatás

14.2. OOCWC lexer

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

Ebben a feladatban a nyelv lexikális egységeivel ismerkedhetünk.

Mi is az a lexer? Leginkább egy olyan szoftver, ami valamilyen input sztringeket dolgoz fel. Most is ez lesz a feladat. Előre meghatározott nevesített reguláris kifejezések vannak:

```
INIT "<init"
INITG "<init guided"
WS [ \t]*
WORD [^-:\n \t()]{2,}
INT [0123456789]+
FLOAT [-.0123456789]+
ROUTE "<route"
CAR "<car"
POS "<pos"
GANGSTERS "<gangsters"
STAT "<stat"
DISP "<disp>"
```

Ezután a "szabályok": az adott kifejezésnek megfelelő blokkhoz ugrik a vezérlés, az a kód fog lefutni:

```
{DISP} {
  m_cmd = 0;
  }

{POS}{WS}{INT}{WS}{INT}{WS}{INT} {

  std::sscanf(yytext, "<pos %d %u %u", &m_id, &from, &to);
  m_cmd = 10001;
  }

{CAR}{WS}{INT} {

  std::sscanf(yytext, "<car %d", &m_id);
  m_cmd = 1001;
  }

{STAT}{WS}{INT} {

  std::sscanf(yytext, "<stat %d", &m_id);
  m_cmd = 1003;
  }

{GANGSTERS}{WS}{INT} {

  std::sscanf(yytext, "<gangsters %d", &m_id);
  m_cmd = 1002;
  }

{ROUTE}{WS}{INT}{WS}{INT}({WS}{INT})* {

  int size{0};
  int ss{0};
  int sn{0};
  std::sscanf(yytext, "<route %d %d%n", &size, &m_id, &sn);
  ss += sn;
  for(int i{0}; i<size; ++i)
  {
  unsigned int u{0u};
  std::sscanf(yytext+ss, "%u%n", &u, &sn);
  route.push_back(u);
  ss += sn;
  }
  m_cmd = 101;
  }
  {INIT}{WS}{WORD}{WS}("c"|"g") {

  std::sscanf(yytext, "<init %s %c>", name, &role);
  num = 1;
  m_cmd = 0;
```

```
  }
{INIT}{WS}{WORD}{WS}{INT}{WS}("c"|"g") {

  std::sscanf(yytext, "<init %s %d %c>", name, &num, &role);
  if(num >200)
  {
  m_errnumber = 1;
  num = 200;
  }
  m_cmd = 1;
  }
{INITG}{WS}{WORD}{WS}("c"|"g") {

  std::sscanf(yytext, "<init guided %s %c>", name, &role);
  num = 1;
  m_guided = true;
  m_cmd = 3;
  }
{INITG}{WS}{WORD}{WS}{INT}{WS}("c"|"g") {

  std::sscanf(yytext, "<init guided %s %d %c>", name, &num, &role);
  if(num >200)
  {
  m_errnumber = 1;
  num = 200;
  }
  m_guided = true;
  m_cmd = 2;
  }
. {;}
```

Az történik a fenti kódban, hogy bizonyos input sztringek hatására a sscanf függvény más és más értékeket olvas be, mindig annyit, amennyi meg van adva, a formátumnak megfelelően, így könnyebb lesz a beolvasás, mert pontosan tudjuk, mit akarunk beolvasni, hogy is néz ki a sztring. A Car Lexer headerjére vessünk még egy pillantást, hogy a program struktúráját egy kicsit megértsük:

```
private:
  int m_cmd {0};
  char name[128];
  int num {0};
  char role;
  int m_errnumber {0};
  bool m_guided {false};
  std::vector<unsigned int> route;
  int m_id {0};
  unsigned int from {0u};
  unsigned int to {0u};
```

Ez az osztály a FlexLexer alosztálya, és getter függvények vannak benne, az előbbi változókhoz, illetve egy virtuális függvény, a yylex(). Ezzel az osztállyal egy helyes bemenetről sikeresen fogunk beolvasni, ezt a

projectben használják.

Az információk egy tcp porton érkeznek, és a lexer feladata az ő feldolgozásuk, a beérkező sztringek az input sztringek, ezeket dolgozza fel a lexer, ahogy fentebb láthattuk. A beérkező adatoknak megfelelően történnek a továbbiak a programban.

14.3. l334d1c4

Írj olyan OO Java vagy C++ osztályt, amely leet cipherként működik, azaz megvalósítja ezt a betű helyettesítést: https://simple.wikipedia.org/wiki/Leet

Mi is az a leet nyelv? Röviden: a szavakban lévő betűket kicseréljük más karakterekre. Ezek nagyjából logikus, érthető cserék, ki lehet találni, hogy mi mit szeretne jelölni.

Hol találkozhattál már vele? Elterjedt szokás, hogy számítógépes játékokban a felhasználó olyan nevet ad, melyben ilyen mód vannak karakterek kicserélve, egyrészt az egyezés elkerülése végett, másrészt olyan hackeres kinézetet is ad, például LEVI helyett L3V1.

Nézzük meg a forrást, a szükséges könyvtárak, és az osztály definíciója:

```
#include<iostream>
#include<fstream>
#include<vector>
#include<string>
#include <algorithm>
#include <cctype>
using namespace std;

class LeetCipher {
public:
  LeetCipher();
  string decipher(string& text);
private:
  vector<vector <string>> leetabc;
  vector<string> abc;
};
```

Az osztályban van két abc, az egyik az átalakítandó angol abc lesz, a másik pedig a feladatban megadott leet nyelv ábécéje. A forrás ábécé egy sima vektor, a leetabc pedig egy sztringeket tartalmazó vektorokból álló vektor, így egyszerűbb, mivel nem minden betűre lesz ugyanannyi helyettesítési lehetőség. Van egy konstruktor, plusz egy tagfüggvény, ami elvégzi az átalakítást. A konstruktorban töltjük fel az ábécéket:

```
LeetCipher::LeetCipher() {

srand (time(NULL));
leetabc.resize(26);

leetabc[0]= {"4", "/-||", "/_||", "@", "/||"};
leetabc[1]={"8","|3", "13", "|}", "|:", "|8", "18", "6", "|B", "|8", "lo",  ↩
   "|o"};
```

```
leetabc[2]= { "<", "{", "[", "("};
leetabc[3] = {"|)", "|}", "|]"};
leetabc[4] = {"3"};
leetabc[5] = { "|=", "ph", "|#", "||"};
leetabc[6] = {"[", "-", "[+", "6"};
leetabc[7] = {"4", "|-|", "[-]", "{-}", "|=|", "[=]", "{=}"};
leetabc[8] = {"1", "|", "!", "9"};
leetabc[9] = {"_|", "_/", "_7", "_)", "_]", "_}"};
leetabc[10] = {"|<", "1<", "l<", "|{", "l{"};
leetabc[11] = {"|_", "|", "1", "]["};
leetabc[12] = {"44", "|||/|", "^^", "/||/||", "/X||", "[]||/][", "[]V[]"," ↩
   ][|||//][", "(V)","//., .||||", "N||"};
leetabc[13] = {"||||", "/||/", "/V", "][|||||]["};
leetabc[14] ={"0", "()", "[]", "{}", "<>"};
leetabc[15] = { "|o", "|O", "|>", "|*", "||textdegree{}", "|D", "/o"};
leetabc[16] = {"O_", "9", "(,)", "0,"};
leetabc[17] = {"|2"," 12", ".-", "|^", "l2"};
leetabc[18] = {"5", "$", "§"};
leetabc[19] = {"7", "+", "7`", "|'||'" , "`|`" , "~|~" , "-|-"};
leetabc[20] = {"|_|", "||_||", "/_/", "||_/", "(_)", "[_]", "{_}"};
leetabc[21] = {"||/"};
leetabc[22] = {"||/||/", "(/||)", "||^/", "|/|||", "||X/", "|||||'", "|'//" ↩
   , "VV"};
leetabc[23] = {"%", "*", "><", "}{", ")("};
leetabc[24] = {"`/", "$|yen$"};
leetabc[25] = { "2", "7_", ">_"};

abc =
{
  "A" ,
  "B",
  "C",
  "D",
  "E",
  "F",
  "G",
  "H",
  "I",
  "J",
  "K",
  "L",
  "M",
  "N",
  "O",
  "P",
  "Q",
  "R",
  "S",
  "T",
  "U",
```

```
  "V",
  "W",
  "X",
  "Y",
  "Z"
};
}
```

A leetabc-t ugyanakkorára méretezzük, mint a forrás ábécét (jelenleg 26), majd a lehetséges helyettesítéseket egymás után megadjuk, ezek közül lesz egy véletlenszerű helyettesítés majd kicsit később. Ezután a forrás ábécét is megadjuk, az egyszerűség kedvéért csak a nagybetűket, és majd később nagybetűvé alakítjuk a forrásszöveget.

A következő rész végzi magát a fordítást:

```
string LeetCipher::decipher(string &text) {

  string leet = "";
  string be = text;

  transform(be.begin(), be.end(),be.begin(), ::toupper);

  while(!be.empty())
  {
  bool find=false;
  for (int i=0; i<abc.size(); i++)
    {
    size_t found = be.find(abc[i]);
    if (found==0)
      {
      find =true;
      leet+=leetabc[i][rand() % leetabc[i].size()];
      if(be.length() > 1)
      be=be.substr(1);
    else
      be.clear();
    }
  }

  if(!find){
    leet+=be[0];
    if(be.length()>1)
      be=be.substr(1);
    else
      be.clear();
    }
  }

return leet;
}
```

Ez a függvény a LeetCipher osztály decipher függvénye, egy sztringet kap bemenetként, és egy sztringet ad vissza, a leetelt változatot. A bemenetet először is átalakítjuk nagybetűssé az egyszerűség kedvéért. Ezután karakterenként olvassuk a szöveget (addig megyünk, míg nem üres). Ha a következő karakter angol abc-beli karakter, akkor a megfelelő leet betűk közül egy véletlenszerűt teszünk a kimenet sztringbe, ha nem, akkor meghagyjuk az eredetit.

A main függvény:

```
int main(int argc, char * argv[])
{
  ifstream infile (argv[1]);
  fstream outfile (argv[2], ios::out);

  string text = "";
  string temp = "";

  while(getline(infile, temp)) {
    text+=temp;
  }

  LeetCipher* cipher = new LeetCipher();
  string cipheredText = cipher->decipher(text);
  outfile<<cipheredText<<endl;
}
```

Két argumentum van, a be-, és kimeneti fájlok, ezeket a parancssorban tudjuk megadni. Példányosítunk egy új LeetCipher()-t, majd a bemenet szövegére meghívjuk a decipher() függvényt, és kész is vagyunk.

15. fejezet

Helló, Stroustrup!

15.1. JDK osztályok

Tanulságok, tapasztalatok, magyarázat...:

A JDK összes osztályát kilistázó program elkészítéséhez mintául szolgálhat a FUTURE projekt `fenykard.cpp` fájlja, melynek `read_acts` függvénye a .props fájlok kereséséért és listázásáért volt felelős.

```
void read_acts ( boost::filesystem::path path, std::map <std::string, int>  ↩
   &acts )
{

        if ( is_regular_file ( path ) ) {

                std::string ext ( ".props" );
                if ( !ext.compare ( boost::filesystem::extension ( path ) ) ↩
                    ) {

                        std::string actpropspath = path.string();
                        std::size_t end = actpropspath.find_last_of ( "/" ) ↩
                           ;
                        std::string act = actpropspath.substr ( 0, end );

                        acts[act] = get_points ( path );

                }

        } else if ( is_directory ( path ) )
                for ( boost::filesystem::directory_entry & entry : boost:: ↩
                   filesystem::directory_iterator ( path ) )
                        read_acts ( entry.path(), acts );

}
```

A `boost.cpp`-vel tehát a JDK-ban található `src.zip`-ben lévő .java állományokat fogjuk kilistázni (a könyvtárszerkezet kicsomagolása után), majd ezek alapján kiiratjuk a JDK osztályok számát. A Boost

könyvtár segítségével rekurzívan megyünk végig a könyvtárszerkezeten, így a .java fájlokban lévő osztályok kigyűjtése felgyorsul és leegyszerűsödik.

```
class Feldolgoz {

private:
    std::string _path;
    int _count = 0;

public:
    Feldolgoz(std::string filePath):_path(filePath) {}

    void travel(boost::filesystem::path path) {
        boost::filesystem::directory_iterator it{path}, eod;
        BOOST_FOREACH(boost::filesystem::path const& p, std::make_pair(it,  ↩
           eod)) {
            if (boost::filesystem::is_regular_file(p) && boost::filesystem ↩
               ::extension(p.string()) == ".java") {
                std::cout << p << std::endl;
                _count++;
            }
            else if(boost::filesystem::is_directory(p)) travel(p);
        }
    }

    std::string getPath() {
        return _path;
    }

    int getCount() {
        return _count;
    }

};

void usage(){
    std::cout << "./boost <mappa>\n";
}

int main(int argc, char** argv){
    if (argc < 2) {
        usage();
        return -1;
    }

    Feldolgoz* obj = new Feldolgoz (argv[1]);
    obj->travel(obj->getPath());
    std::cout << obj->getCount() << std::endl;
}
```

Ha az `src.zip` tartalmát kicsomagoltuk, akkor következhet a .cpp fordítása. Ezután ha argumentumként

megadjuk az src mappát futtatáskor, megkapjuk a JDK osztályok számát:

```
$ g++ boost.cpp -o bejaro -lboost_system -lboost_filesystem - ↩
   lboost_program_options -std=c++14
$ ./boost src
```

15.2. Hibásan implementált RSA törése

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

Jelenlegi feladatban az RSA kódolást kell elrontani, szóval betűnként kell titkosítani az egész szöveg helyett.
A megadott publikus kulccsal kell lekódolnunk az összes betűt.

```
public class rsa_chiper {
  public static void main(String[] args) {
    int bitlength = 2100;

    SecureRandom random = new SecureRandom();

    BigInteger p = BigInteger.probablePrime(bitlength/2, random);
    BigInteger q = BigInteger.probablePrime(bitlength/2, random);

    BigInteger publicKey = new BigInteger("65537");
    BigInteger modulus = p.multiply(q);

    String str = "this is a perfect string".toUpperCase();
    System.out.println("Eredeti: " + str);

    byte[] out = new byte[str.length()];
    for (int i = 0; i < str.length(); i++) {
      char c = str.charAt(i);
      if (c == ' ')
        out[i] = (byte)c;
      else
        out[i] = new BigInteger(new byte[] {(byte)c}).modPow(publicKey,  ↩
           modulus).byteValue();
    }
    String encoded = new String(out);
    System.out.println("Kódolt:" + encoded);

    Decode de = new Decode(encoded);
    System.out.println("Visszafejtett: " + de.getDecoded());
  }
}
```

A fő osztály az rsa_chiper lesz, itt található a main függvény, ami elvégzi a neki adott szöveg kódolását, majd megpróbálja dekódolni is azt. Itt a szöveg a `this is a perfect string` lesz, a program ezt próbálja majd vissza fejteni. A szöveg kódolása, a ciklus miatt betűről betűre történik.

A visszafejtéshez szükségünk van egy listára, ami tartalmazza a betűk gyakoriságát. Itt fontos, hogy minden nyelvnek más betűgyakorisági listája van, sőt ez szövegkörnyezettől is függ. Ha nagyon eltérő a két nyelv gyakorisága, akkor teljesen értelmetlen szöveget fog a dekóder kiadni.

```
private void loadFreqList() {
  BufferedReader reader;
  try {
    reader = new BufferedReader(new FileReader("freq.txt"));
    String line;
    while((line = reader.readLine()) != null) {
      String[] args = line.split("\t");
      char c = args[0].charAt(0);
      int num = Integer.parseInt(args[1]);
      this.charRank.put(c, num);
    }
  } catch (Exception e) {
    System.out.println("Error when loading list -> " + e.getMessage());
  }
}
```

Ha a program sikeresen beolvasta a betűgyakoriság listát, megnézi, hogy melyik karakterből mennyi van a szövegben. Ez azért kell, mivel ez alapján fogja később eldönteni, hogy melyik karaktert rendelje hozzá. A ciklus betűként megy végig a szövegen és megnézi a karaktereket, ha space van csak simán tovább keres, ha viszont egy másik karakter, akkor megnézi, hogy benne van-e már a listában. Ha benne van akkor növeli az értékét eggyel, ha nincs akkor bele rakja 1-es értékkel. Így kapunk majd egy listát, ami a karakterek előfordulásának számát fogja tartalmazni.

```
public Decode(String str) {
  this.charRank = new HashMap<Character, Integer>();
  this.decoded = str;

  this.loadFreqList();

  HashMap<Character, Integer> frequency = new HashMap<Character, Integer ↩
     >();
  for (int i = 0; i < str.length(); i++) {
    char c = str.charAt(i);
    if (c != ' ')
      if(frequency.containsKey(c))
        frequency.put(c, frequency.get(c) + 1);
      else
        frequency.put(c, 1);
  }

  while (frequency.size() > 0) {
    int mi = 0;
    char c = 0;
```

```
      for (Entry<Character, Integer> e : frequency.entrySet()) {
        if (mi < e.getValue()) {
          mi = e.getValue();
          c = e.getKey();
        }
      }
      this.decoded = this.decoded.replace(c, this.nextFreq());
      frequency.remove(c);
    }
  }
```

A listánk elkészülte után, ráengedünk egy ciklust, ami addig fut, amíg üres nem lesz. Egy maximum kiválasztással megnézzük mi a leggyakoribb elem, majd ezt az elemet kiveszem a listából, és átírom a jelenlegi leggyakoribb karakterre. Ezt a karaktert a `nextFreq()` függvénnyel kapjuk meg.

```
  private char nextFreq() {
    char c = 0;
    int nowFreq = 0;
    for(Entry<Character, Integer> e : this.charRank.entrySet()) {
      if (e.getValue() > nowFreq) {
        nowFreq = e.getValue();
        c = e.getKey();
      }
    }
    if (this.charRank.containsKey(c))
      this.charRank.remove(c);
    return c;
  }
```

Amint lefutott a ciklus és az összes karakter behelyettesítésre került a `getDecoded` függvénnyel tudjuk lekérni a visszafejtett szövegünket.

15.3. Változó argumentumszámú ctor

Megoldás forrása: https://gitlab.com/whoisZORZ/bhax/tree/master/attention_raising/Stroustrup/Ctor

Tanulságok, tapasztalatok, magyarázat...:

A Benoît Mandelbrot nevéhez köthető Mandelbrot-halmaz a komplex számsíkon ábrázolva egy fraktálalakzatot jelent. A kétdimenziós halmaz png ábráját fogjuk először létrehozni a `mandel.cpp` file segítségével.

```
$ more mandel.cpp
#include <iostream>
#include "png++/png.hpp"

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

```
    }

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

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

    double dx = (b-a)/szelesseg;
    double dy = (d-c)/magassag;
    double reC, imC, reZ, imZ, ujreZ, ujimZ;
    int iteracio = 0;
    std::cout << "Szamitas";
    for (int j=0; j<magassag; ++j) {

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

            reC = a+k*dx;
            imC = d-j*dy;

            reZ = 0;
            imZ = 0;
            iteracio = 0;
            while (reZ*reZ + imZ*imZ < 4 && iteracio < iteraciosHatar) {

                ujreZ = reZ*reZ - imZ*imZ + reC;
                ujimZ = 2*reZ*imZ + imC;
                reZ = ujreZ;
                imZ = ujimZ;

                ++iteracio;

            }

            kep.set_pixel(k, j, png::rgb_pixel(255-iteracio%256,
                                               255-iteracio%256, 255- ↩
                                                  iteracio%256));
        }
        std::cout << "." << std::flush;
    }

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

Ahogy látjuk, a `mandel.cpp` sikeres fordításához és futtatásához szükségünk lesz a libpng, libpng++ könyvtárakra és a `png++/png.hpp` header filera. A könyvtárakat a sudo parancs segítségével tudjuk telepíteni, viszont a headerhez le kell töltenünk a png++ hivatalos oldaláról a becsomagolt telepítő állományokat: png++-0.2.9.tar.gz

```
//A terminálba leadott parancsok:
```

```
$ sudo apt-get install libpng-dev
$ sudo apt-get install libpng++-dev
$ tar -zxf png++-0.2.9.tar.gz -C ~/Letöltések
$ cd ~/Letöltések/png++-0.2.9
$ make
```

Ha minden jól megy, mostmár tudjuk fordítani és futtatni a png elkészítéséhez szükséges programot.

```
$ g++ mandel.cpp `libpng-config --ldflags` -o mandel
$ ./mandel mandel.png
Szamitas.....................................mandel.png mentve
```

A perceptron az egyik legegyszerűbb előrecsatolt neurális hálózat. A `main.cpp` segítségével fogjuk szimulálni a hiba-visszaterjesztéses módszert, mely a többrétegű perceptronok (MLP) egyik legfőbb tanítási módszere. Ahhoz, hogy ezt fordítani és futtatni tudjuk később, szükségünk lesz az `mlp.hpp` header filera, mely már tartalmazza a Perceptron osztályt.

Az előző program futtatásával létrejött Mandelbrot png ábrát fogjuk beimportálni. A header filenak köszönhetően megadhatjuk a rétegek számát, illetve a neuronok darabszámát. A beolvasásra kerülő kép piros részeit a lefoglalt tárba másoljuk bele. A `mandel.png` alapján új képet állítunk elő, mely megkapja az eredeti kép magasságát és szélességét. A visszakapott értékeket megfeleltetjük a blue értékeknek.

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

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

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

  double* image = new double[size];

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

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

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

    png_image.write("output.png");

  delete p;
  delete [] image;
```

```
}
```

Az első felvonás Perceptron feladatához képest módosításokat kell végeznünk a header file-on is, ugyanis új képet akarunk előállítani. A double pointer () operátor már egy tömböt térít vissza, melynek segítségével bele tudunk nyúlni a képbe. Az utolsó units tömb értékei átkerülnek a paraméterként kapott tömbbe, az eredeti és az új kép egyforma méretű lesz.

```
double* operator() ( double image [] )
  {

    units[0] = image;

    for ( int i {1}; i < n_layers; ++i )
      {

#ifdef CUDA_PRCPS

        cuda_layer ( i, n_units, units, weights );

#else

        #pragma omp parallel for
        for ( int j = 0; j < n_units[i]; ++j )
          {
            units[i][j] = 0.0;

            for ( int k = 0; k < n_units[i-1]; ++k )
              {
                units[i][j] += weights[i-1][j][k] * units[i-1][k];
              }

            units[i][j] = sigmoid ( units[i][j] );

          }

#endif

      }

    for (int i = 0; i < n_units[n_layers - 1]; i++)
        image[i] = units[n_layers - 1][i];

    return image;

  }
```

A fordításhoz a C++11 szabványt használjuk, futtatáskor pedig megadjuk annak a képfájlnak a nevét és kiterjesztését, amelyet be kívánunk olvasni (a `mandel.cpp`-ből kapott `mandel.png` ábra).

16. fejezet

Helló, Gödel!

16.1. Gengszterek

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

Először is azt nézzük meg, mi is az a lambda kifejezés. Ez egy olyan funkcionális elem, mely az imperatív nyelvekben is megtalálható. A lényege, hogy egy olyan névtelen függvényt írunk, amit egyből fel is használunk, a későbbiekben pedig már nem lesz lehetőség a használatukra, csak oda van megírva, ahol tényleg használjuk. A szintaktukájuk egyszerűen a következő:

```
[ captures ] ( params ) -> ret { body }
```

A visszatérítési típust nem is kell megadni, mert a fordító azt tudni fogja.

A feladara térve: a gengsztereket rendezni lambdával a Robotautó Világbajnokságban. Lássuk a kódot:

```
std::sort (gangsters.begin(), gangsters.end(), [this, cop] (Gangster x,  ↩
   Gangster y)
{
    return dst (cop, x.to) < dst (cop, y.to);
});
```

```
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare  ↩
   comp);
```

Magához a rendezéshez a std::sort() függvényt használjuk, ami a lambda kifejezés alapján fog rendezni, az összehasonlítás abban van. A gangsters elejétől a végéig megyünk, és a gengsztereket a következő alapján rendezi: (két Gangster obejektum a bemenet) ha x közelebb van a cop-hoz, mint y, akkor igaz a függvény visszatérítési értéke, tehát a rendezés után a vektor elején a legközelebbi bűnözők lesznek a rendőrhöz képest. A második kódcsipet pedig a sort jelenlegi használatát mutatja.

16.2. C++11 Custom Allocator

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

A feladat egy saját allokátor készítése volt, a fenti prezi alapján, C++ nyelven. Én a következő megoldást készítettem:

```
#include <iostream>
#include <exception>
#include <iostream>
#include <cxxabi.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <vector>
#include <unistd.h>
using namespace std;

template<typename T>
struct CustomAlloc{

  using size_type = size_t;
  using value_type = T;
  using pointer = T*;
  using const_pointer = const T*;
  using reference = T&;
  using const_reference = const T&;
  using difference_type = ptrdiff_t;

  CustomAlloc(){}
  CustomAlloc(const CustomAlloc &){}
  ~CustomAlloc(){}
  pointer allocate (size_type n) {
    int s;
    char * p = abi::__cxa_demangle (typeid(T).name(),0,0,&s);

    std::cout << "Allocating "
          << n << " object of "
          << n*sizeof(T)
          << "  bytes. "
          << typeid(T).name() << " = " << p
          << std::endl;
    free (p);

    return reinterpret_cast<T*> (
            new char[n*sizeof(T)]
    );
  }

  void  deallocate (pointer p, size_type n) {
    delete[] reinterpret_cast<char *>(p);
    std::cout<<" deallocate ";
    }
};
```

```
int main()
{
    vector<int, CustomAlloc<int> > v1;
    for (int i=1;i<11;i++) v1.push_back(i);

    return 0;
}
```

A témához is kapcsolódik ez a feladat, ugyanis egy gyűjteményt (kollekciót) használunk, ez gyakorlatilag egy vektor lesz itt. Egytől tízig tíz darab int lesz a vektorban, és közben a nyomkövető üzenetekkel áthatjuk, mi is történik a háttérben, a lenti kép mutatja. Ugye a vektor eredetileg üres, ezért amikor először fűzünk a végére egy intet, lefoglal neki 4 bájtot. Ezután, a következő inthez már kétszer négy bájt kell, és a harmadiknál 4x4 bájtot foglal le az egész vektornak, de fel is szabadítja a már nem kellő 1x4 bájtot, és így tovább, mindig kettőhatvány számú objektumnak foglal helyet, és felszabadítja, amire már nincs szükség.

Ha tíz helyett tizenhat intet teszünk a vektorba, akkor annak is pont ugyannnyi memóriát fog allokálni.

A forráskódban még megfigyelhetjük, hogy noha nem objektum, hanem struktúra az allokátor, megfigyelhető benne a hármas szabály: van konstruktor, másoló konstruktor és destruktor.

16.3. STL map érték szerinti rendezése

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

Először pár gondolat arról, hogy mi is az a map adatszerkezet. A Map egy olyan asszociatív konténer, melyben az adatok kulcs (key value), és map (mapped value) érték szerinti kombináció alapján vannak tárolva. STL: standard template library. Formálisan a következőképpen adjuk meg:

```
std::map <key_type, data_type>
```

Ismét a fénykard projecthez fogunk nyúlni, a megadott kódrészletet fogjuk kicsit megnézni:

```
std::vector<std::pair<std::string, int>> sort_map ( std::map <std::string,  ↩
   int> &rank )
{
        std::vector<std::pair<std::string, int>> ordered;

        for ( auto & i : rank ) {
                if ( i.second ) {
                        std::pair<std::string, int> p {i.first, i.second};
                        ordered.push_back ( p );
                }
        }

        std::sort (
                std::begin ( ordered ), std::end ( ordered ),
        [ = ] ( auto && p1, auto && p2 ) {
                return p1.second > p2.second;
        }
```

```
        );

        return ordered;
}
```

A map rendezése általában kulcs alapján történik, de most érték szerint akarjuk, ez a lényeg. Már az első sor is nehéznek tűnik ránézésre, de azért ki lehet hámozni: egy sort_map nevű, vektor visszatérési típusú függvényt látunk. Olyan vektort ad vissza, mely párokat tartalmaz, ezek a párok egy stringből és egy intből állnak. Bemenetként pedig kap egy mapot érték szerint, ez a map string kulcsú, és integer adatokat tartalmaz, ezek szerint fogunk rendezni.

Létrehozunk egy, a visszatérési értéknek megfelelő ordered nevű vektort, ebben lesz az eredmény. A for ciklusban gyakorlatilag átpakoljuk a map tartalmát az ordered vektorba. Végigmegyünk a rank-on, és ha van érték, akkor létrehozunk egy új párt, feltöltjük, és a vektor végére rakjuk.

Ezután következik a rendezés. A csokor első feladatához hasonlóan, az std::sort függvényt használjuk, és ismét lambda függvényt. Csökkenő sorrendbe rendezzük az elemeket, mivel akkor lesz igaz a sort, ha "p1.second > p2.second". Végeredményül a már rendezett vektort adja a függvény.

16.4. Alternatív Tabella rendezése

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

Az Alternatív Tabella egy olyan alternatív rangsor, mely nem csupán a szerzett pontok alapján állít rangsort a csapatok között, hanem valami egyéb szempontot is figyelembe vesz, például, hogy ki ellen ért el győzelmet a csapat, mert ugye erősebb ellenfél legyőzése nagyobb teljesítményt igényelhet. Ezáltal árnyaltabb képet kaphatunk a csapatokról. A példában a magyar labdarúgó bajnokság első osztályának csapataival foglalkozunk.

Ami a témáinkat érinti, ebben a feladatban megismerkedhetünk az interfészek felhasználásával, implenetálásukkal. A Csapat osztály egy olyan osztály, mely implementálja a Comparable interfészt:

```
class Csapat implements Comparable<Csapat> {

  protected String nev;
  protected double ertek;

  public Csapat(String nev, double ertek) {
    this.nev = nev;
    this.ertek = ertek;
  }

  public int compareTo(Csapat csapat) {
    if (this.ertek < csapat.ertek) {
      return -1;
    } else if (this.ertek > csapat.ertek) {
      return 1;
    } else {
      return 0;
```

```
    }
  }
}
```

Tehát a Csapat típusú objektumnak lesz egy neve és értéke (string és double). Ezután egy sima konstruktort látunk, majd egy függvény definícióját, mivel az osztály a Comparable interfészt implementálja, ki kell fejtenünk, hogy a compareTo függvény hogyan működjön, ez szolgál majd az összehasonlítás alapjaként. Maga az összehasonlítás nem túl bonyolult, kap egy csapatot, és az adott objektumhoz képest, ha ez a kapott csapat értéke nagyobb, mínusz egyet, ha kisebb, egyet, különben nullát ad vissza a metódus.

De erre miért is van szükség? Miért nem működne a program, ha kihagynánk az egész Comparable dolgot?

Az előfordulásokat megnézve, a válasz a következő: ezeket a Csapat objektumokat listában tároljuk:

```
java.util.List<Csapat> rendezettCsapatok = java.util.Arrays.asList(csapatok ↩
   );
```

És rendezzük is őket:

```
java.util.Collections.sort(rendezettCsapatok);
```

Emiatt van szükség az implementációra, ugyanis, ha rákeresünk a "java.util.Collections.sort"-ra, a hivatalos dokumentációban a következőt találjuk:

Az utolsó mondat szerint, a lista minden elemének implementálnia kell a Comparable interfészt. Ez egyébként elég logikus, mert a rendezésnek valami alapján össze kell hasonlítania az elemeket, tehát összehasonlíthatónak kell lenniük, és persze ezt magától honnan is tudná a Java, hogy két objektumot mi alapján hasonlítson össze.

Úgy kerülhetnénk ki ezt a dolgot, ha nem a java.util.Collections.sort()-ot használnánk a rendezésre, hanem kézzel írnánk egy olyan kódrészletet, amely megcsinálja a rendezést, de ez vélhetően bonyolultabb, és kevésbé hatékony lenne. Itt láthatjuk egy magas szintű nyelv előnyét is.

17. fejezet

Helló, !

17.1. OOCWC Boost ASIO hálózatkezelése

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

Ez a feladat az input stream (adatfolyam) feldolgozásával kapcsolatos. Állománykezelés szempontjából a C++ streamekben, vagyis adatfolyamokban gondolkozik. Ez gyakorlatilag bájtok egymás utáni sorozata, és lehet be-, vagy kimeneti adatfolyam (istream/ostream, fstream: kétirányú). Az OOCWC projectre nézzünk rá:

```
while ( std::sscanf ( data+nn, "<OK %d %u %u %u>%n", &idd, &f, &t, &s, &n ) ↩
    == 4 )
    {
        nn += n;
        gangsters.push_back ( Gangster {idd, f, t, s} );
    }
```

Egy while ciklust láthatunk, ami ugye addig ismétli a törzset, amíg a feltétel igaz. Tehát a sscanf() függvény egy számot ad visszatérési értékként, ez pedig a beolvasott paraméterek száma. Tehát a ciklus addig megy, míg sikerül négy paraméter beolvasása. Maga a függvény bemenetül egy sztringet vár, ez az első paraméter, ebből olvassa a bemenetet. Ezután a kiolvasási sablon: olyan sztring, mely <OK-val kezdődik, ezután egy egész, majd három előjel nélküli egész következik. Ezeket, mint például egy printf-nél, vesszővel követik a változók, hogy hova olvassuk be. Az utolsó paraméter a beolvasott bájtok számát jelöli.

A ciklus törzsében pedig megnöveljük az nn változót a beolvasott bájtok számáva, vagyis az eddig összesen beolvasott bájtok számát tartjuk benne számon. Majd a gangsters tömbhöz hozzátesszük az új beolvasott gengsztert, a paraméterekkel együtt.

17.2. BrainB

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

Ha belenézünk a projectbe, a számunkra a feladathoz releváns kódcsipetet a BrainBWin nevű cpp forrásban találjuk. De mi is az a Qt slot-signal mechanizmus? Objektumok közötti kommunikációra használatos

mechanizmus a Qt-ban. Előnye, hogy nem kell az egyik osztályban a másik függvényeit használni, ezáltal elkerülhető a cross reference. Az elve a következő: az esemény hatására egy signal lesz kibocsájtva, és a slot pedig egy olyan függvény, ami a megfelelő signal esetén hívódik meg.

A connect függvénnyel lehet a slotokat és signalokat összekapcsolni. Két ilyen függvény van a forrásban:

```
connect ( brainBThread, SIGNAL ( heroesChanged ( QImage, int, int ) ),
      this, SLOT ( updateHeroes ( QImage, int, int ) ) );

connect ( brainBThread, SIGNAL ( endAndStats ( int ) ),
      this, SLOT ( endAndStats ( int ) ) );
```

A signal és slot szignatúrája (paraméterek száma, típusa) mindig megegyezik. A fenti kód értelmezése: amikor a brainBThread-szálon a heroesChanged signal következik be, akkor a slot hatására lejátszódik az updateHeroes függvény. A másik connectnél hasonlóképp: amikor a brainBThread-szálon az endAndStats signal következik be, vagyis ha vége a futásnak, akkor vége a játéknak, és a következő függvény kiírja fájlba az eredményt, és vége a futásnak:

```
void BrainBWin::endAndStats ( const int &t )
{

        qDebug()  << "\n\n\n";
        qDebug()  << "Thank you for using " + appName;
        qDebug()  << "The result can be found in the directory " + statDir;
        qDebug()  << "\n\n\n";

        save ( t );
        close();
}
```

A BrainBThread.cpp-ben az a kódcsipet, ami kiváltja ezt a signalt:

```
void BrainBThread::run()
{
        while ( time < endTime ) {

                /*
    ...
    */
        }

        emit endAndStats ( endTime );

}
```

17.3. SamuCam

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

Ebben a feladatban nem csak egyszerű szöveges streamekkel, hanem a kameránk képével fogunk dolgozni, az openCv segítségével. A gépeden a következő telepítésre lesz szükséged:

```
sudo apt install libopencv-dev
```

A feladat értelmében a SamuCam.cpp és a SamuCam.h fájlok tartalmával fogunk dolgozni. A következő pár sor még a main-ből van, ebben a parancssori argumentumként kapott ip címet stringként kezelve átadjuk a samulife-nak.

```
std::string videoStream = parser.value ( webcamipOption ).toStdString();
SamuLife samulife ( videoStream, 176, 144 );
```

Nézzünk bele a headerbe: itt gyakorlatilag a SamuCam osztály deklarációja található, konstruktor, destruktor, attribútumok, függvények. A legtöbb kifejtése a másik fájlban van.

```
public:
    SamuCam ( std::string videoStream, int width, int height );
    ~SamuCam();

    void openVideoStream();
    void run();

private:
    std::string videoStream;
    cv::VideoCapture videoCapture;
```

Áttérünk a SamuCam.cpp-re, a példányosításhoz szükséges konstruktor következik. A webkamera szervere által biztosított felvétel eléréséhez a `SamuCam::openVideoStream()` függvényt használjuk. A VideoCapture osztály segít hozzáférni a kamera által szolgáltatott adatokhoz.

```
SamuCam::SamuCam ( std::string videoStream, int width = 176, int height =  ↩
   144 )
  : videoStream ( videoStream ), width ( width ), height ( height )
{
  openVideoStream();
}

SamuCam::~SamuCam ()
{
}

void SamuCam::openVideoStream()
{
  videoCapture.open ( videoStream );

  videoCapture.set ( CV_CAP_PROP_FRAME_WIDTH, width );
  videoCapture.set ( CV_CAP_PROP_FRAME_HEIGHT, height );
  videoCapture.set ( CV_CAP_PROP_FPS, 10 );
}
```

A video streamet a videoCapture open() függvényével nyitjuk meg. Paraméterként megadhatunk ip címet is, de ha például a nulla paramétert adjuk meg, ez esetben a saját webkameráról szerver nélkül is menni fog a dolog, ugyanis egy deviceID-t kap így, a kamera indexét. Ezután beállítjuk a szélességet, magasságot,

és akár az FPS értékét is. Most következik a SamuCam::run() függvény, a legtöbb lényegi dolog ebben található:

```
void SamuCam::run()
{

  cv::CascadeClassifier faceClassifier;

  std::string faceXML = "lbpcascade_frontalface.xml"; // https://github.com ↩
     /Itseez/opencv/tree/master/data/lbpcascades

  if ( !faceClassifier.load ( faceXML ) )
    {
      qDebug() << "error: cannot found" << faceXML.c_str();
      return;
    }

  cv::Mat frame;

  while ( videoCapture.isOpened() )
    {

      QThread::msleep ( 50 );
      while ( videoCapture.read ( frame ) )
        {

          if ( !frame.empty() )
            {

              cv::resize ( frame, frame, cv::Size ( 176, 144 ), 0, 0, cv:: ↩
                 INTER_CUBIC );

              std::vector<cv::Rect> faces;
              cv::Mat grayFrame;

              cv::cvtColor ( frame, grayFrame, cv::COLOR_BGR2GRAY );
              cv::equalizeHist ( grayFrame, grayFrame );

              faceClassifier.detectMultiScale ( grayFrame, faces, 1.1, 4,  ↩
                 0, cv::Size ( 60, 60 ) );

              if ( faces.size() > 0 )
                {

                  cv::Mat onlyFace = frame ( faces[0] ).clone();

                  QImage* face = new QImage ( onlyFace.data,
                                              onlyFace.cols,
                                              onlyFace.rows,
                                              onlyFace.step,
```

```
                                                QImage::Format_RGB888 );

                    cv::Point x ( faces[0].x-1, faces[0].y-1 );
                    cv::Point y ( faces[0].x + faces[0].width+2, faces[0].y + ↩
                        faces[0].height+2 );
                    cv::rectangle ( frame, x, y, cv::Scalar ( 240, 230, 200 ) ↩
                        );


                    emit  faceChanged ( face );
                  }

                QImage*  webcam = new QImage ( frame.data,
                                               frame.cols,
                                               frame.rows,
                                               frame.step,
                                               QImage::Format_RGB888 );

                emit  webcamChanged ( webcam );

              }

            QThread::msleep ( 80 );

          }

        if ( ! videoCapture.isOpened() )
          {
            openVideoStream();
          }

      }

}
```

A képek elemzésére betöltünk egy CascadeClassifier-t. ha a betöltés a load függvény segítségével nem sikerül, hibaüzenet keletkezik. Ez a CascadeClassifier teszi lehetővé a program számára, hogy arcokat ismerjen fel. AZ arcokhoz pontosabban a faceClassifier objektum kell, ez egy emberi arcot ismer fel a kamera képén.

A while ciklus tartalmára pillantsunk még rá. A ciklus addig fut, míg a kamera meg van nyitva. 50 millisze-kundumonként olasunk be egy képet a read függvénnyel. A `cv::Mat` osztályú objektum arra kell, hogy az egyes képkockákat kétdimenziós tömbben tudjuk tárolni, ez persze a kezeléshez fontos. Egy ilyen tömbbe olvassuk az adott képkockát. Ezután átméretezzük a cv:: resize() segítségével, és még egy pár műveletet elvégez a program. A detectMultiScale függvénnyel keressük az objektumokat, vagyis most arcokat, és ezeket egy vekorban tároljuk. Ha találtunk arcot, akkor az elsőből csinálunk egy QImage-et, majd keretet rajzolunk köré, ezeket továbbadjuk a program többi részének.

A futtatáshoz erre szükségünk lesz: **wget https://github.com/Itseez/opencv/raw/master/data/lbpcascades/lbpc**

IV. rész

Irodalomjegyzék

17.4. Általános

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

17.5. C

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

17.6. C++

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

17.7. 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.