Univerzális programozás

Így neveld a programozód!

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

COLLABORATORS			
	TITLE : Univerzális programozás		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, Bátfai, Margaréta, Ã©s Boda, Zsolt	2020. április 2.	

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

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?

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

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

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

II. rész

Tematikus feladatok

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

2. fejezet

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: https://youtu.be/lvmi6tyz-nI

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/infty-f.c, bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/infty-w.c.

Számos módon hozhatunk és hozunk létre végtelen ciklusokat. Vannak esetek, amikor ez a célunk, például egy szerverfolyamat fusson folyamatosan és van amikor egy bug, mert ott lesz végtelen ciklus, ahol nem akartunk. Saját példáinkban ilyen amikor a PageRank algoritmus rázza az 1 liter vizet az internetben, de az iteráció csak nem akar konvergálni...

Egy mag 100 százalékban:

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

  return 0;
}
```

vagy az olvashatóbb, de a programozók és fordítók (szabványok) között kevésbé hordozható

```
int
#include <stdbool.h>
main ()
{
  while(true);

  return 0;
}
```

Azért érdemes a `for(;;)` hagyományos formát használni, mert ez minden C szabvánnyal lefordul, másrészt a többi programozó azonnal látja, hogy az a végtelen ciklus szándékunk szerint végtelen és nem szoftverhiba. Mert ugye, ha a `while`-al trükközünk egy nem triviális `1` vagy `true` feltétellel, akkor ott egy másik, a forrást olvasó programozó nem látja azonnal a szándékunkat.

Egyébként a fordító a `for`-os és `while`-os ciklusból ugyanazt az assembly kódot fordítja:

```
$ gcc -S -o infty-f.S infty-f.c
$ gcc -S -o infty-w.S infty-w.c
$ diff infty-w.S infty-f.S
1c1
<    .file "infty-w.c"
---
>    .file "infty-f.c"
```

Egy mag 0 százalékban:

```
#include <unistd.h>
int
main ()
{
  for (;;)
    sleep(1);

  return 0;
}
```

Minden mag 100 százalékban:

```
#include <omp.h>
int
main ()
{
#pragma omp parallel
{
  for (;;);
}
  return 0;
}
```

A **gcc infty-f.c -o infty-f -fopenmp** parancssorral készítve a futtathatót, majd futtatva, közben egy másik terminálban a **top** parancsot kiadva tanulmányozzuk, mennyi CPU-t használunk:

```
top - 20:09:06 up  3:35,  1 user,  load average: 5,68, 2,91, 1,38
Tasks: 329 total,   2 running, 256 sleeping,   0 stopped,   1 zombie
%Cpu0 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu1 : 99,7 us, 0,3 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu2 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
```

```
%Cpu3 : 99,7 us, 0,3 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu4 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu5 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu6 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu7 :100,0 us, 0,0 sy, 0,0 ni, 0,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem :16373532 total,11701240 free, 2254256 used, 2418036 buff/cache
KiB Swap:16724988 total,16724988 free,       0 used. 13751608 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 5850 batfai    20   0   68360    932    836 R 798,3  0,0   8:14.23 infty-f
```

Werkfilm

- https://youtu.be/lvmi6tyz-nI

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.

Alan Turing bebizonyitotta,hogy ilyen programot képtelenség létrehozni,egyszerübb programoknál szemre láthatólag meg lehet állapitani,de komplexebb programoknál lehetetlen,erre bonyorult matematikai számításokkal tudott rájönni a 90-es években.

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: code/Turing/valtozo_erekenek_felcserelese.cpp

Változókat több féle képpen is fel lehet cserélni. Legkönyebb mikor létrehozunk egy változót, amiben belemásoljuk ideiglenesen az értéket, amit felülírunk. Majd amivel felülírtuk az felülírjuk az ideiglenes változó értékével.

Ha nem akarunk új változó bevezetni, akkor is több módon fel tudjuk cserélni az értékeket. Például szorzás-osztással, vagy XOR bit szintű változtatással.

```
#include <iostream>

int main()
{

    // Segédváltozóóval
    int f = 3;
    int g = 4;

    std::cout << "f=" << f << std::endl;
    std::cout << "g=" << g << std::endl;

    int tmp = f;
    f = g;
    g =  tmp;

    std::cout << "f=" << f << std::endl;
    std::cout << "g=" << g << std::endl;

    std::cout << std::endl;


    // Segéd változó nélkül
    int a = 10;
    int b = 20;

    std::cout << "a=" << a << std::endl;
    std::cout << "b=" << b << std::endl;

    a = a*b;
    b = a/b;
    a = a/b;

    std::cout << "a=" << a << std::endl;
    std::cout << "b=" << b << std::endl;
```

```
    std::cout << std::endl;


    // XOR
    int x = 5;
    int y = 7;

    std::cout << "x=" << x << std::endl;
    std::cout << "y=" << y << std::endl;

    x = x ^ y;
    y = x ^ y;
    x = x ^ y;

    std::cout << "x=" << x << std::endl;
    std::cout << "y=" << y << std::endl;
}
```

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: `code/Turing/labda.c`

Ezt a feladatot egy 2d-s játék motorjához tudom hasonlítáni.

Egy kordináta rendszert kell elképzeni, ahol a labda egy pont. Ez a pontot kell mozgatni egy "irány vektorral" (ez nem a matematikai irány vektor vektor). A vektornak 2d minenziós, van egy x és egy y offset. Ezeket at offseteket minden iterációban hozzáadjuk a labda kordinátáihoz. Az offsetek csak akkor változnak, ha elérik a megadott "screen" (terminál ablak) oldalait (szélső értékeit), ekkor az offsetek -1-szeresére változnak, vagyis megfordúlnak.

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


int main (void)
{
    WINDOW* window;
    window = initscr ();
```

```
    int xdesc=0,ydesc=0, xasc=0,yasc=0;
    int maxX;
    int maxY;

    for (;;) {

        getmaxyx ( window, maxY , maxX );
        maxY=maxY*2;
        maxX=maxX*2;
        refresh();
        clear();
        usleep (80000);

        xdesc = (xdesc-1) % maxX;
        xasc = (xasc+1) % maxX;
        ydesc = (ydesc-1) % maxY;
        yasc = (yasc+1) % maxY;

        mvprintw(abs((ydesc + (maxY-yasc))/2),abs((xdesc+(maxX-xasc))/2),"O ↩
           ");
    }
}
```

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: https://youtu.be/9KnMqrkj_kU, https://youtu.be/KRZlt1ZJ3qk, .

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/bogomips.c

A szóhossz megadja hogy az adott int-et a gép hány bit-en tárolja. A "word" változó értéke 1. vagyis 2^0-on (00000001) ez az egy bitet a shifteljük (mozgatjuk) balra. Így a word érteke 2^1 vagyis 2 (00000010). Ezzel párhuzamosan növelünk egy másik értéket is, hogy a végén megtudjuk hányszor shifteltünk. Addig tudjuk tólni ezt egy bitet, ahány bit-en tárolja a gép az int-et, vagyis a elérjük az utolsó 2 hatványát, akkor kifutunk a helyből és az az 1 bit már máshol lesz (nem lesz ott, eltűnik, minden bit 0 lesz) vagyis a word értéke 0 lesz, hamis.

```
#include<stdio.h>

int main(){
    unsigned long long int wordLength = 0, word = 1;
```

```
    while(word <<= 1){
        wordLength++;
    }

    wordLength++;
    printf("The word length on this PC is %llu bits\n", wordLength);
    return 0;
}
```

2.6. Helló, Google!

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

Megoldás videó: https://www.youtube.com/watch?v=5wRFa8l35QI

Megoldás forrása: code/Turing/page_rank.c

A page rank algoritmust a Google fejlesztette ki, hogy kategórizálni és értékelni tudja a weblapokat. Az alap gondolat az, hogy egy oldal annál értékesebb, minél több értékes oldal mutat rá.

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

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

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

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

    return sqrt(osszeg);
}

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

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

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

    int i,j;

    for(;;){
        for(i=0; i<4;++i){
            PR[i] = 0.0;
            for(j=0; j<4; ++j){
                PR[i] += L[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);
    return 0;
}
```

2.7. 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

A Monty Hall probléma egy régi tv műsor alapján létrejött teória. A műsorban a játékos 3 ajtó közül választhatott amelyek egyike mögött ott volt a nagy nyeremény. Eddig a középiskolás valószínűség számítás szerint mindenki úgy gondolja, hogy a nyerési esély 1/3 (33.33%) és ekkor még mindenkinek igaza lenne.

A játék úgy zajlott, hogy a játékos választott 1 ajtót a 3 közül. Utánna Monty (a műsorvezető, aki tudta, hogy minden ajtó mögött mi van) választott egy olyan ajtót ami mögött biztosan nincs semmi, aztán megkérdezte a játákost, hogy el akarja e cserélni a választott ajtót a másik ajtóval ami még zárva volt. A játékos megtarthatta a választott ajtót vagy átmehetett a másikhoz. Ebben az esetben, HOSSZÚTÁVON mindig az nyert többet, aki a másik ajtót választotta.

Miért? Azért mert az elején minden ajtó nyerési lehetősége 1/3 (33.33%). Amikor kiválasztunk egy ajtót akkor nálunk van egy ajtó 1/3 nyerési lehetőséggel, szemben velünk meg 2 1/3 ajtó vagyik az esélyek 1/3(33.33%) a 2/3(66.66%) ellen (2 ajtó formájában). Mikor Monty kinyit egy ajtót ami mögött nincs semmi, akkor annak az ajtónak a nyerési esélye átszáll az utolsó ajtóra, vagyik egy ajtóba sűrűsül 2 ajtó nyerési esélye, így az az ajtó értéke 2/3(66.66%) lesz. Így jogosan hangzik az, hogy jobb ha a másik ajtót választjuk aminek nagyobb az esélye. Fontos megjegyzés, hogy ez a technika nem garantálja a nyerést. Most tegyük fel, hogy van 100 (vagy több) ajtónk. Ha választunk egyet, akkor a nyerési esélye az ajtónak 1/100(1%). Utánna Monty kinyit nekünk 98 olyan ajtót ami mögött nics semmi, akkor az utolsó ajtónak a nyerési esélye 1/99(99%) (e teória alapján).

```
kiserletek_szama=100000
kiserlet = sample(1:3, kiserletek_szama, replace=T)
jatekos = sample(1:3, kiserletek_szama, replace=T)
musorvezeto=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama) {

    if(kiserlet[i]==jatekos[i]){

        mibol=setdiff(c(1,2,3), kiserlet[i])

    }else{

        mibol=setdiff(c(1,2,3), c(kiserlet[i], jatekos[i]))

    }

    musorvezeto[i] = mibol[sample(1:length(mibol),1)]

}

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

for (i in 1:kiserletek_szama) {

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

}

valtoztatesnyer = which(kiserlet==valtoztat)


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

2.8. 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 természetes számok építőelemei a prímszámok. Abban az értelemben, hogy minden természetes szám előállítható prímszámok szorzataként. Például 12=2*2*3, vagy például 33=3*11.

Prímszám az a természetes szám, amely csak önmagával és eggyel osztható. Eukleidész görög matematikus már Krisztus előtt tudta, hogy végtelen sok prímszám van, de ma sem tudja senki, hogy végtelen sok ikerprím van-e. Két prím ikerprím, ha különbségük 2.

Két egymást követő páratlan prím között a legkisebb távolság a 2, a legnagyobb távolság viszont bármilyen nagy lehet! Ez utóbbit könnyű bebizonyítani. Legyen n egy tetszőlegesen nagy szám. Akkor szorozzuk össze n+1-ig a számokat, azaz számoljuk ki az 1*2*3*... *(n-1)*n*(n+1) szorzatot, aminek a neve (n+1) faktoriális, jele (n+1)!.

Majd vizsgáljuk meg az a sorozatot:

(n+1)!+2, (n+1)!+3,... , (n+1)!+n, (n+1)!+ (n+1) ez n db egymást követő azám, ezekre (a jól ismert bizonyítás szerint) rendre igaz, hogy

- (n+1)!+2=1*2*3*... *(n-1)*n*(n+1)+2, azaz 2*valamennyi+2, 2 többszöröse, így ami osztható kettővel
- (n+1)!+3=1*2*3*... *(n-1)*n*(n+1)+3, azaz 3*valamennyi+3, ami osztható hárommal
- ...
- (n+1)!+(n-1)=1*2*3*... *(n-1)*n*(n+1)+(n-1), azaz (n-1)*valamennyi+(n-1), ami osztható (n-1)-el
- (n+1)!+n=1*2*3*... *(n-1)*n*(n+1)+n, azaz n*valamennyi+n-, ami osztható n-el
- (n+1)!+(n+1)=1*2*3*... *(n-1)*n*(n+1)+(n-1), azaz (n+1)*valamennyi+(n+1), ami osztható (n+1)-el

tehát ebben a sorozatban egy prim nincs, akkor a (n+1)!+2-nél kisebb első prim és a (n+1)!+ (n+1)-nél nagyobb első prim között a távolság legalább n.

Az ikerprímszám sejtés azzal foglalkozik, amikor a prímek közötti távolság 2. Azt mondja, hogy az egymástól 2 távolságra lévő prímek végtelen sokan vannak.

A Brun tétel azt mondja, hogy az ikerprímszámok reciprokaiból képzett sor összege, azaz a (1/3+1/5)+ (1/5+1/7)+ (1/11+1/13)+... véges vagy végtelen sor konvergens, ami azt jelenti, hogy ezek a törtek összeadva egy határt adnak ki pontosan vagy azt át nem lépve növekednek, ami határ számot B_2 Brun konstansnak neveznek. Tehát ez nem dönti el a több ezer éve nyitott kérdést, hogy az ikerprímszámok halmaza végtelen-e? Hiszen ha véges sok van és ezek reciprokait összeadjuk, akkor ugyanúgy nem lépjük át a B_2 Brun konstans értékét, mintha végtelen sok lenne, de ezek már csak olyan csökkenő mértékben járulnának hozzá a végtelen sor összegéhez, hogy így sem lépnék át a Brun konstans értékét.

Ebben a példában egy olyan programot készítettünk, amely közelíteni próbálja a Brun konstans értékét. A repó bhax/attention_raising/Primek_R/stp.r mevű állománya kiszámolja az ikerprímeket, összegzi a reciprokaikat és vizualizálja a kapott részeredményt.

```
#    Copyright (C) 2019  Dr. Norbert Bátfai, nbatfai@gmail.com
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see <http://www.gnu.org/licenses/>

library(matlab)

stp <- function(x){

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

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

Soronként értelemezzük ezt a programot:

```
 primes = primes(13)
```

Kiszámolja a megadott számig a prímeket.

```
> primes=primes(13)
> primes
[1]  2  3  5  7 11 13
```

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

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

```
> diff
[1] 1 2 2 4 2
```

Az egymást követő prímek különbségét képzi, tehát 3-2, 5-3, 7-5, 11-7, 13-11.

```
idx = which(diff==2)
```

```
> idx = which(diff==2)
> idx
[1] 2 3 5
```

Megnézi a `diff`-ben, hogy melyiknél lett kettő az eredmény, mert azok az ikerprím párok, ahol ez igaz. Ez a `diff`-ben lévő 3-2, 5-3, 7-5, 11-7, 13-11 különbségek közül ez a 2., 3. és 5. indexűre teljesül.

```
t1primes = primes[idx]
```

Kivette a primes-ból a párok első tagját.

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

A párok második tagját az első tagok kettő hozzáadásával képezzük.

```
rt1plust2 = 1/t1primes+1/t2primes
```

Az 1/t1primes a t1primes 3,5,11 értékéből az alábbi reciprokokat képzi:

```
> 1/t1primes
[1] 0.33333333 0.20000000 0.09090909
```

Az 1/t2primes a t2primes 5,7,13 értékéből az alábbi reciprokokat képzi:

```
> 1/t2primes
[1] 0.20000000 0.14285714 0.07692308
```

Az 1/t1primes + 1/t2primes pedig ezeket a törteket rendre összeadja.

```
> 1/t1primes+1/t2primes
[1] 0.5333333 0.3428571 0.1678322
```

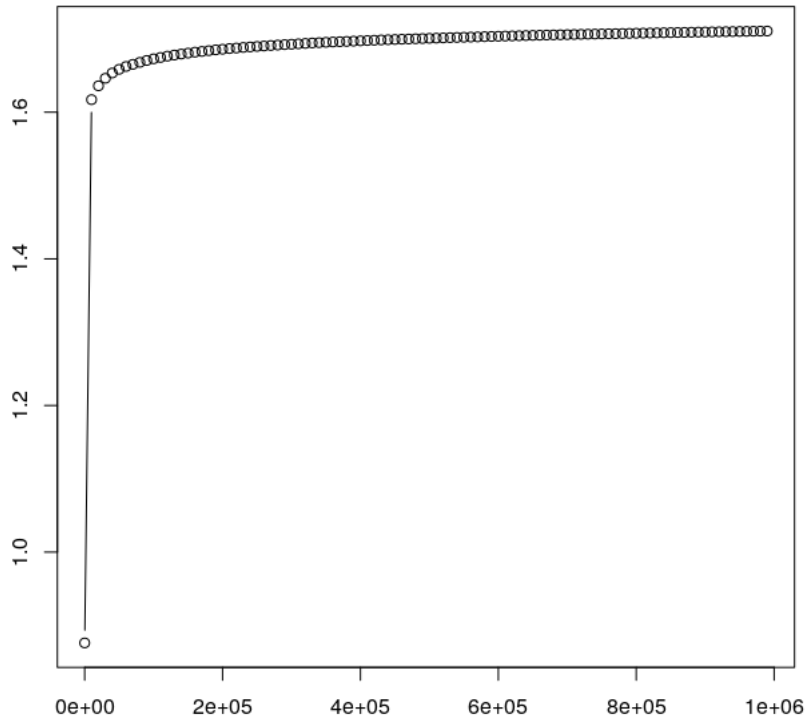
Nincs más dolgunk, mint ezeket a törteket összeadni a `sum` függvénnyel.

```
sum(rt1plust2)
```

```
>   sum(rt1plust2)
[1] 1.044023
```

A következő ábra azt mutatja, hogy a szumma értéke, hogyan nő, egy határértékhez tart, a B_2 Brun konstanshoz. Ezt ezzel a csipettel rajzoltuk ki, ahol először a fenti számítást 13-ig végezzük, majd 10013, majd 20013-ig, egészen 990013-ig, azaz közel 1 millióig. Vegyük észre, hogy az ábra első köre, a 13 értékhez tartozó 1.044023.

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



2.1. ábra. A B_2 konstans közelítése

Werkfilm

- https://youtu.be/VkMFrgBhN1g
- https://youtu.be/aF4YK6mBwf4

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó: Készül

Megoldás forrása:https://turingmachine.io/ Fájl forrása: `code/Chomsky/unaris.c`

Az unáris számrendszer a természetes számok leírására alkalmas. Általában egyesekkel vagy pálcikákkal jelöljük a számokat, de bármilyen szimbólumot is bevezethetünk. A szimbólumot annyiszor írjuk le, amennyi az ábrázolandó számunk értéke.

```
        #include <stdio.h>

int main(){
    int x;
    printf("Adjon meg egy értéket: ");
    scanf("%d", &x);
    for(int i=0; i<x; ++i)
        printf("1");
    printf("\n");
    return 0;

}
```

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:

Chomsky 4 különböző osztályba csoportosította a grammatikákat (általános, környezetfüggő, környezetfüggetlen, reguláris). Ebben a feladatban környezetfüggő grammatikákkal foglalkozunk, ezeknek a helyettesítési szabályai xYz → xyz alakúak.

Ebben az esetben a levezetési szabályok mindkét oldalán szerepelhetnek terminális szimbólumok, melyeket konstansoknak nevezünk és kisbetűkkel jelöljük. A nem terminális szimbólumokat változóknak nevezzük és nagybetűkkel jelöljük. Mindkét esetben a levezetést a kezdő szimbólummal (S) kezdjük. Ez egy kitüntetett, nem terminális elem.

```
                S, X, Y változók
        a, b, c konstansok
Szabályok:
        S  -> abc
        S  -> aXbc
        Xb -> bX
        Xc -> Ybcc
        bY -> Yb
        aY -> aaX
        aY -> aa
Levezetés:
    S        (S → aXbc)
    aXbc     (Xb → bX)
    abXc     (Xc → Ybcc)
    abYbcc   (bY → Yb)
    aYbbcc   (aY -> aaX)
    aaXbbcc  (Xb → bX)
    aabXbcc  (Xb → bX)
    aabbXcc  (Xc → Ybcc)
    aabbYbccc (bY → Yb)
    aabYbbccc (bY → Yb)
    aaYbbbccc (aY → aa)
    aaabbbccc
```

3.3. Hivatkozási nyelv

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

Megoldás videó: Jön

Megoldás forrása:

Az alábbi programok C89 szabvánnyal nem fognak lefordulni. Az első esetben a for ciklusfejben történő deklaráció miatt, a második esetben pedig az egysoros komment miatt.

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

```
int main()
{
    for(int i=0; i<4; i++)
    {
        printf("Hello world!");
    }
}
```

```
#include <stdio.h>

int main()
{
    //int a = 89;
}
```

Az új szabvány létrejöttével számos újdonság jelent meg, például: változó méretű tömbök, új függvények, új adattípusok, új header állományok. Néhány dolgot a C++ -ból emeltek át.

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: https://youtu.be/9KnMqrkj_kU (15:01-től).

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

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

A lexer gyakorlatilag egy szövegelemző program. Beolvas egy forráskódot és felismeri benne a tokeneket, melyek a program építőelemei.

A fenti program három fő részből áll, melyeket **%%** jelek választják el egymástól. Az első részben történnek a deklarációk. Megadjuk, hogy mit értünk számjegynek(0 és 9 közötti számokat). A második részben a valós számokat definiáljuk: bármennyi számjegy(lehet nulla is, ezt a * jelzi) + pont + bármennyi számjegy, de legalább egy(+ jelzi). Ha a program talál egy számot, akkor növeli a realnumbers változó értékét eggyel, illetve kiírja a találatot. A program utolsó része már teljesen C nyelvben látható, ahol függvénymeghívást és a találatok számának kiírását láthatjuk. A program futásához szükséges a flex telepítése, majd fordításkor a -lfl kapcsoló használata.

3.5. Leetspeak

Lexelj össze egy l33t ciphert!

Megoldás videó: https://youtu.be/06C_PqDpD_k

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

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

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

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

  {'a', {"4", "4", "@", "/-\\"}},
  {'b', {"b", "8", "|3", "|}"}},
  {'c', {"c", "(", "<", "{"}},
  {'d', {"d", "|)", "|]", "|}"}},
  {'e', {"3", "3", "3", "3"}},
  {'f', {"f", "|=", "ph", "|#"}},
  {'g', {"g", "6", "[", "[+"}},
  {'h', {"h", "4", "|-|", "[-]"}},
  {'i', {"1", "1", "|", "!"}},
  {'j', {"j", "7", "_|", "_/"}},
  {'k', {"k", "|<", "1<", "|{"}},
  {'l', {"l", "1", "|", "|_"}},
  {'m', {"m", "44", "(V)", "|\\/|"}},
  {'n', {"n", "|\\|", "/\\/", "/V"}},
  {'o', {"0", "0", "()", "[]"}},
  {'p', {"p", "/o", "|D", "|o"}},
  {'q', {"q", "9", "O_", "(,)"}},
  {'r', {"r", "12", "12", "|2"}},
  {'s', {"s", "5", "$", "$"}},
```

```
  {'t', {"t", "7", "7", "'|'"}},
  {'u', {"u", "|_|", "(_)", "[_]"}},
  {'v', {"v", "\\/", "\\/", "\\/"}},
  {'w', {"w", "VV", "\\/\\/", "(/\\)"}},
  {'x', {"x", "%", ")(", ")("}},
  {'y', {"y", "", "", ""}},
  {'z', {"z", "2", "7_", ">_"}},

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

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

%}
%%
. {

    int found = 0;
    for(int i=0; i<L337SIZE; ++i)
    {

      if(l337d1c7[i].c == tolower(*yytext))
      {

        int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0));

          if(r<91)
          printf("%s", l337d1c7[i].leet[0]);
          else if(r<95)
          printf("%s", l337d1c7[i].leet[1]);
        else if(r<98)
          printf("%s", l337d1c7[i].leet[2]);
        else
          printf("%s", l337d1c7[i].leet[3]);

        found = 1;
        break;
      }

    }
```

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

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

Ez a program egy beolvasott szöveget karakterenként alakít át a l337d1c7 tömbben definiált módon. A struct-ban adjuk meg a karakterek lehetséges "titkosított" változatát. A program elején definiáljuk a tömb méretét, így nem szükséges azt előre megadni. Ezáltal könnyen hozzá tudunk írni a tömbhöz, kiegészíthetjük bármennyi tetszőleges elemmel anélkül, hogy a sorok megszámolásával és a méret átírásával foglalkoznánk. A program tehát karakterenként vizsgálja a szöveget. Ha az aktuális karaktert megtalálja a tömbben, akkor egy véletlenszerűen generált szám alapján(sorsolással) kiválaszt a megadott négy lehetséges karakter közül egyet. Ha az aktuálisan vizsgált karakter nem találja meg a tömbben, akkor pedig ugyanúgy, változatlanul kiírja. Az utolsó részben az srand() függvény a random szám generátort (rand()) inicializálja. A fordítás során a -lfl kapcsoló szükséges, ez a flex library-ra hivatkozik

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);
```

i. Ha a SIGINT jel nincs figyelmen kívül hagyva(ignorálva), akkor ezentúl a jelkezelő függvény kezelje azt.

ii. For ciklus legyen ötször végrehajtva. Nullától indítjuk, iterációnként eggyel inkrementáljuk i értékét. Az inkrementáció alakja preorder, a következő iterációnál i értéke i+1 lesz. A feltétel alapján i utolsó értéke 4 lesz, mivel ez az utolsó érték, amire teljesül a feltétel(ötnél kisebb).

iii. Az előző példától annyiban különbözik, hogy az inkrementáció alakja postorder, de ez nem változtat az iterációk számán, ugyanúgy ötször fog lefutni a ciklus.

iv. For ciklus, mely ötször fog lefutni. Ránézésre azt várjuk, hogy a tömb i-edik elemének helyére i+1-et ír, tehát az elemek sorra 1,2,3,4,5 lesznek. Azonban nem teljesen ez fog történni, az első(nulladik) elem változatlan marad, utána pedig az elemek sorra 1,2,3,4 lesznek. Ha a tömb első eleme eredetileg nem volt feltöltve értékkel, akkor random szám lesz a helyén. Ez hibának tekinthető, splint-tel futtatva "Parse Error" jelzést kapunk.

v. For ciklus, mely addig fut, amíg teljesül két feltétel: az egyik, hogy i értéke kisebb, mint n értéke. A másik feltételt vizsgáljuk meg részletesen. Először visszakapjuk d és s értékét, majd azokat a következő iterációban növeljük. S és d mutatók, tehát az inkrementálás itt azt jelzi, hogy a következő elemre mutatnak. A * dereferencia jelzés visszaadja azt az értéket, amire mutatnak. Ebben a zárójelben tehát ha tömbről beszélünk, akkor d értéknek helyébe azt az értéket írja, amire s mutat. Lényegében egy tömb(d) értékeit lecseréli a másik tömb(s) értékeire. Azonban ez csak akkor működne tökéletesen, ha n értéke pontosan megegyezne a két tömb méretével. Splint-tel futtatva szintén "Parse Error" jelzést kapunk.

vi. Standart kimenetre írunk. Kétszer hívjuk meg ugyanazt a függvényt, egyszer a-val és a+1-gyel, majd fordítva. Nem ajánlott egy zárójelen belül ilyet tenni, mivel nem azt fogjuk kapni amit első ránézésre gondolnánk. Ez a kódcsipet hibásnak tekinthető, annak ellenre, hogy le fog futni. Splint-tel futtatva ezt kapjuk(részlet):

vii. Szintén a standard kimenetre írunk, először f-et hívjuk meg a-val és a visszatért értéket írjuk ki, majd a eredeti értékét íratjuk ki. A értékén nem változtat a függvényhívás akkor se, ha a függvényben módosítjuk azt.

viii. A változó címét argumentumként átatjuk az f függvénynek és kiiratjuk amivel visszatér, majd kiírjuk a (eredeti) értékét is a standard kimenetre.

Megoldás forrása:

Megoldás videó: Jön

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

1. Minden számhoz(x) létezik egy nála nagyobb szám(y), ami prím. Azaz, a prímszámok száma végtelen.

2. Minden x-hez létezik egy nála nagyobb y, hogy: y és y+2 is prímszám. Vagy: minden számnál(x) tudunk egy annál nagyobbat mondani(y), amire igaz hogy ő(y) és a kettővel nagyobb(y+2) értékű szám is prím. Azaz, az ikerprímek száma végtelen.

3. Létezik egy szám(y), ami minden másik prímszámnál(x) nagyobb. Tehát minden prímszámnál(x) létezik egy nagyobb szám(y), ami nem prím.

4. Ekvivalens átalakításokat végezve ugyanazt kapjuk, mint az első példában: minden számhoz(x) létezik egy nála nagyobb szám(y), ami prím. Azaz, a prímszámok száma végtelen.

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:

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

```
#include <stdio.h>

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

int
mul (int a, int b)
{
```

```
    return a * b;
}

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

}

int
main ()
{

    int (*f) (int, int);

    f = sum;

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

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

    g = sumormul;

    f = *g (42);

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

    return 0;
}
```

```
#include <stdio.h>

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

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

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

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

int
main ()
{

    F f = sum;

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

    G g = sumormul;

    f = *g (42);

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

    return 0;
}
```

A mutatók memóriacímet tárolnak. Ha egy változó elé * jelet írunk, akkor viszont a változó értékét kérdezzük le. Dereferenciával tudjuk egy változó memóriacímét lekérdezni.

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

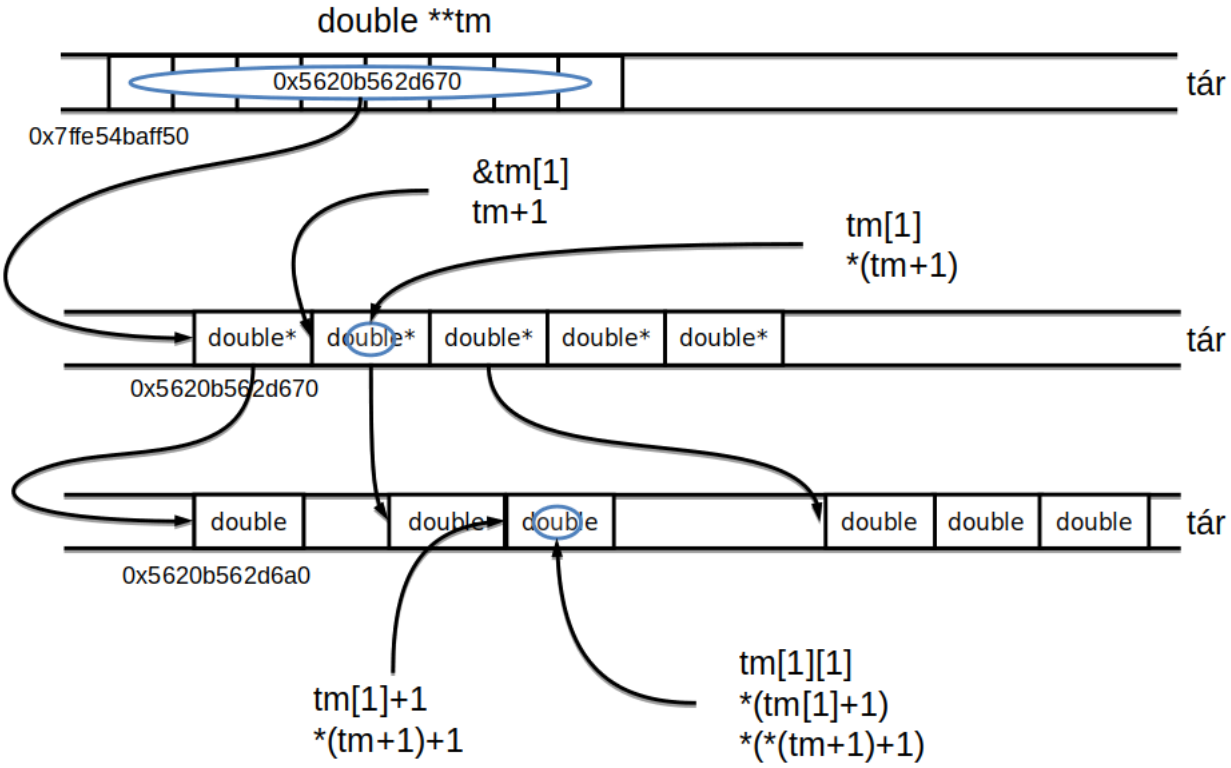
Írj egy olyan `malloc` és `free` párost használó C programot, amely helyet foglal egy alsó háromszög mátrixnak a szabad tárban!

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

Megoldás forrása: bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Caesar/tm.c

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

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

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

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

    }

    for (int i = 0; i < nr; ++i)
        for (int j = 0; j < i + 1; ++j)
```

```
            tm[i][j] = i * (i + 1) / 2 + j;

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

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

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

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

    free (tm);

    return 0;
}
```



4.1. ábra. A double ** háromszögmátrix a memóriában

A feladat során egy háromszögmátrixnak foglalunk helyet. Emiatt soronként végignézzük, hogy van-e megfelelő hely az adott sornak és ha igen lefoglajuk. Majd az adott soron belül helyet foglalunk a megadott elemszámnak, ami soronként mindig eggyel több az első sorban egyről indulva. Ha ez megtörtént, akkor feltöltjük elemekkel a mátrixot. Erre több fajta lehetőségünk is van, melyre több lehetőséget is látunk a programkódban.

A felada elején az `nr` egész típusú változóban eltároljuk a sorok számát. Majd egy tm double típusú értékre mutatóra mutatót létrehozunk és ennek segítségével próbálunk helyet foglalni neki. Először megnézzük, hogy van-e elég szabad hely a sorok számának megfelelő doblra mutató mutatókak, azaz a `tm*`-nak, mivel ebben tároljuk a sorok kezdetének memóriacímét. Ezután megnézzük soronként, hogy van-e a háromszögmátrixban tárolt adatoknak elég hely, ami soronként mindig eggyel több double értéknek való helyfoglalást jelent. Amennyiben ezek a helyfoglalások valahol sikertelenek, akkor `NULL` értéket kapunk vissza. Ezután a háromszögmátrixot sorfolytonosan feltöltjük az 1, 2... értékekkel. Majd kiiratjuk a kapott mátrixot. Utána megnézünk többfajta feltöltési módot, aszerint, hogy hogy hivatkozhatjuk meg egy i-edk sor j-edik elemét, amennyiben mutatóra mutató mutatót használtunk. Utána ezt a mátrixot is töröljü végül pedig felszabadítjuk a helyet először soronként majd a teljes tárat.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása:

```
        #include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <locale.h>
#include <wchar.h>
#include <stdlib.h>

#define MAX_KULCS 100
#define BUFFER_MERET 256


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

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

  int kulcs_index = 0;
  int olvasott_bajtok = 0;

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

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

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

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

  }

      write (1, buffer, olvasott_bajtok);

    }
}
```

A titkosítás során megadunk egy szót, ami alapján titkosítunk. Ezt a szót és a titkos szöveget bájtonként átírjuk. Az átírott szövegekre a kizáró vagy műveletet alkalmazzuk és visszaírjuk bitenként szöveggé. Ami miatt az így titkosított szöveg már csak egy bináris szemétnek tűnik. Ahányszor csak egymás után tudjuk írni a "jelszót" a szöveg hosszában annyiszor tesszük meg. Emellet fontos megemlíteni, hogy a beolvasás során egy úgynevezett buffert használunk, hogy ne terheljük túl a memóriát.

Ha mégegyszer alkalmazzuk a már kódolt szövegre a titkosítást, akkor visszakapjuk az eredeti szöveget.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

```
        public class ExorTitkosító {

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

        byte [] kulcs = kulcsSzöveg.getBytes();
        byte [] buffer = new byte[256];
        int kulcsIndex = 0;
        int olvasottBájtok = 0;

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

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

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

            }

            kimenőCsatorna.write(buffer, 0, olvasottBájtok);

        }

    }

    public static void main(String[] args) {

        try {

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

        } catch(java.io.IOException e) {

            e.printStackTrace();

        }

    }

}
```

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

Tanulságok, tapasztalatok, magyarázat...

4.4. C EXOR törő

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

Megoldás videó:

Megoldás forrása:

```
                  #define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 5
#define _GNU_SOURCE

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


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

  return (double) titkos_meret / sz;
}

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

  double szohossz = atlagos_szohossz (titkos, titkos_meret);

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

}
```

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

  int kulcs_index = 0;

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

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

    }

}

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

  exor (kulcs, kulcs_meret, titkos, titkos_meret);

  return tiszta_lehet (titkos, titkos_meret);

}

int main (void)
{

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

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

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

  // osszes kulcs eloallitasa
char a[5]={'k','u','t','y','a'};
  for (int ii = 0; ii <= 4; ++ii)
    for (int ji = 0; ji <= 4; ++ji)
      for (int ki = 0; ki <= 4; ++ki)
```

```
  for (int li = 0; li <= 4; ++li)
    for(int mi=0; mi<=4; ++mi)
      {
        kulcs[0] = a[ii];
        kulcs[1] = a[ji];
        kulcs[2] = a[ki];
        kulcs[3] = a[li];
        kulcs[4] = a[mi];


        //printf("%c", a[li]);

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

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

  return 0;
}
```

A feladatban egy az előző felatokban bemutatott exor segítségével titkosított szövegett probálunk meg feltörni. Amit fel tudunk használni, hogy ismerjük a kulcs karaktereit, illetve a hosszát. Ekkor több egymásba ágyazott `for` ciklus segítségével előállítjuk az összes lehetséges jelszót. Annyi `for`-t használunk, ahány hosszú a kulcs. Emellet van egy függvényünk, ami azt vizsgálja, hogy lehetséges, hogy a kulcs segítségével volt szöveg volt az eredeti szöveg. Ezt a magyar nyelv néhány statisztikai jellemzője segítségével teszteli. Megnézi, hogy megtalálható-e a szövegben a magyar nyelv leggyakrabban használt szavai közül valamelyik. Illetve a magyar nyelv átlagos szóhossza érvényes-e a szövegben. Ha ezeknek a feltételeknek megfelel egy szöveg, akkor visszakapjuk a lehetséges eredeti szöveget és a hozzá tartozó kulcsot a kimeneten.

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

A program megírása során egy neurális hálónak fogjuk megtanítani a logikai alapműveleteket. Először az or, azaz vagy műveletet adjuk meg. Két lista segítségével előállítjuk a lehetséges állapotokat, úgy hogy a listák azonos elemeit használva egy OR nevű lista azonos számú elemét a megfelelő értéknek mentjük el. Ezután a neuralnet függvénynek betápláljuk a megkapott adatokat. És ennek segítségével a bemenetekhez olyan súlyt próbál számolni, amivel őket megszorozva majd a szorzatokat összeadva általánosságban

megkapjuk a megfelelő kimeneteket. Ehhez használhatunk rejtett neuronokat is melyek segítik a pontosabb eredmény megkapását. Ha megtalálja a megfelelő súlyokat, akkor megtanulta a neuronháló az adott műveletet.

```
        library(neuralnet)

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

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

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

plot(nn.or)

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

A második példa az and azaz az és. Itt is az előző példának megfelelő műveleteket végezzük el, csak eredetileg az és műveletnek mmegfelelő értékeket reprezentálja.

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

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

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

plot(nn.and)

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

A harmadik művelet, amelyet vizsgálunk, az a xor vagyis a kizáró vagy. Ebben az esetben, ha rejtett neuronok nélkül, azaz az előző példák mintájára szeretnénk végrehajtani a feladatot, akkor nem ad vissza megfelelő eredményt. Viszont, ha rejtett neuronok számát legalább kettőre állítjuk, akkor már megfelelő eredményt kapunk.

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

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

nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=2, linear.output=FALSE,  ↩
   stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

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

4.6. Hiba-visszaterjesztéses perceptron

C++

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

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

Ebben a feladatban az előző feladatban már megismert neurális hálót fogjuk alkalmazni egy kép valamely adott RGB komponensére. Ehhez meg kell hívnunk egy mlp.hpp nevű könyvtárat és a kép miatt a png++/png.hpp könyvtárat. A fordítást emiatt a következő parancs segítségével végezhetjük: **g++ mlp.hpp perceptron.cpp -o perceptron -lpng**. Az mlp.hpp könyvtár itt található meg.

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

int main(int argc, char **argv)
{
    png::image<png::rgb_pixel>png_image(argv[1]);
    int size = png_image.get_width()*png_image.get_height();
    Perceptron* p = new Perceptron(3, size ,256, 1);
    double* kep = new double[size];

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

    double value = (*p)(image);
    std::cout<<value<<std::endl;

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

A megoldás során a kép egyes képpontjainak vörös árnyalatának értékét mentjük el egy egydimenziós tömbbe sorfolytonosan, ezek az értékek lesznek a perceptron bemeneti értékei. Emmellett beillesztünk 256 rejtett neuront is és végül egy neuronon kapjuk vissza a kívánt értéket.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

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

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

Az ebben a fejezetben szereplő programok, mind szervesen kötődnek egymáshoz. Ha sikerül egy képet megcsinálni akkor az összes programot át tudjuk rá ültetni, tehát egy másik programváltozattal tudunk benne majd nagyítani, vagy úgy is át tudjuk majd írni, hogy a GPU-t használja, ezáltal nagyon meggyorsítva a számítási folyamatokat.

A forráskód megtalálható a következő linken is: ../Forraskodok/Mandelbrot/5.1/mandelpngt.cpp

```
// mandelpngt.c++
// Copyright (C) 2019
// Norbert Bátfai, batfai.norbert@inf.unideb.hu
//
//  This program is free software: you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation, either version 3 of the License, or
//  (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program.  If not, see <https://www.gnu.org/licenses/>.
//
// Version history
//
//  Mandelbrot png
//  Programozó Páternoszter/PARP
```

```
//   https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063 ↩
   _01_parhuzamos_prog_linux
//
//  https://youtu.be/gvaqijHlRUs
//
#include <iostream>
#include "png++/png.hpp"
#include <sys/times.h>

#define MERET 600
#define ITER_HAT 32000
```

A program elején, mint mindig, megadjuk a header fájlokat. Ez esetben ami ismeretlen lehet az a png++/png.hpp és a sys/times.h fejlécek. Az elsőre azért van szükségünk, hogy létre tudjuk hozni a képet, hogy viziálisan is láthassuk a Mandelbrot halmaz eredményét, ami kép. A második header fájl pedig az idővel kapcsolatos számításokat, az idővel összefüggő függvények meghívását biztosítja számunkra. Idlletve efiníció megadaások is szerepelnek, melyek majd könnyítik a dolgunkat a program írása alatt.

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

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

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

    // a számítás
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, reZ, imZ, ujreZ, ujimZ;
    // Hány iterációt csináltunk?
    int iteracio = 0;
    // Végigzongorázzuk a szélesség x magasság rácsot:
    for (int j = 0; j < magassag; ++j)
        {
        //sor = j;
        for (int k = 0; k < szelesseg; ++k)
            {
            // c = (reC, imC) a rács csomópontjainak
            // megfelelő komplex szám
            reC = a + k * dx;
            imC = d - j * dy;
            // z_0 = 0 = (reZ, imZ)
            reZ = 0;
            imZ = 0;
```

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

            kepadat[j][k] = iteracio;
            }
        }

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

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

        }
```

A `mandel` függvény az ami majd kiszámolja nekünk a Mandelbrot halmazt. Egy időszámítás jön, ami segítségével majd a program lefutás után kiírja azt az időt amit igénybe vett a számolás (az én gépen és egy i3-3227U-s processzor esetén ez körülbelül egy 15 másodpercbe telt). Ezek és a változódeklarálások után jöhetnek a nagyobb számítások. Létrehozunk egy dx szélességű és dy magasságú rácsot a két egymásba-ágyazódó for ciklus segítségével, majd megvizsgáljuk, hogy a `c` komplex szám (mely megkapja a `reC` és a `imC` értékek által meghatározott pontot, pixelt) benne van-e a Mandelbrot halmazban, ha igen akkor azt a pontot beszinezi.

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

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

    int kepadat[MERET][MERET];
```

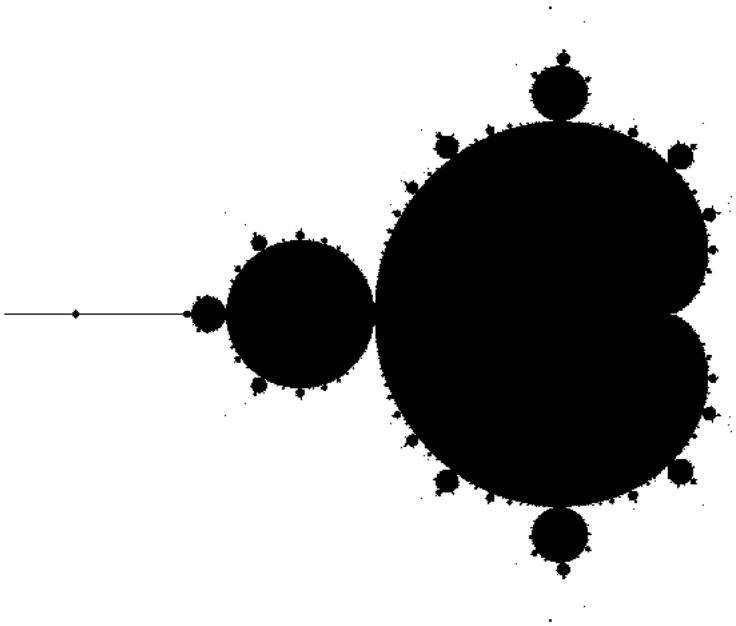
```
    mandel(kepadat);

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

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

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

A főprogram elején megvizsgáljuk, hogy a felhasználó jól hsználta-e a futtatási parancsot, tehát hogyha a parancs nem két argumentumból áll, akkor kiírunk egy hibaüzenetet a kimenetre. A `kepadat` változóba bekérjük a méreteket, tehát a kép szélességét és a magasságát majd ezeket átadjuk a `mandel` függvénynek, ami kiszámolja nekünk a mandelbrot halmazt, és ez alapján megalkotjuk a png kiterjesztésű képet. Miutána program befejezte a számításokat és sikeresen megcsinálta a képet, kiírja a kimenetre a "mentve" üzenetet a felhasználó számára, tudatva a sikert.

Fordítás: **g++ mandelpngt.cpp -lpng -O3 -omandelpngt**

Futtatás: **./mandelpngt mt.png**

Kép megnyitása terminálból: **eog mt.png**



5.1. ábra. A kiadott kép

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

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

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

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

A feladatban tutora voltam Schachinger Zsolt-nak.

Felvetődik egy kérdés: melyikaz a szám amelyiket önmagával megszorozva 9-et kapunk (természetesen ez a 3). Ebből kiindulva a következő kérdés pedig az, hogy melyik az a szám amit ha megszorozunk önmagával -9-et kapunk? Ez pedig már nem lehetséges a valós számhalmazon, így jönnek képbe a komplex számok, melyek úgymond a valós számhalmaz továbbővítése. A komplex számok alapja az "i" szám, melynek értéke a sqrt(-1), ennek a segítségével el lehet végezni a negatív számból való négyzetgyökvonást. Így már meg tudjuk válaszolni az előbb feltett kérdést, az "i" szám segítségével már ki tudjuk hozni a -9-et (tehát 3i*3i).

Ez a program és az előző közötti legnagyobb kölönbség az hogy a `c` amit vizsgálunk, hogy benne van-e a Mandelbrot halmazban, az előző programban egy változó, ebben pedig egy állandó. Így itt a c a rács minden vizsgálandó pontját befutja.

A forráskód megtalálható a következő linken is: ../Forraskodok/Mandelbrot/5.1/mandelbrotcomplex.cpp

```
// Verzio: 3.1.2.cpp
```

```
// Forditas:
// g++ 3.1.2.cpp -lpng -O3 -o 3.1.2
// Futtatas:
// ./3.1.2 mandel.png 1920 1080 2040  ↩
   -0.019473810573093663922605855987058021128 18  ↩
   -0.019473810572541341845642648422654019668 7  ↩
   0.798505756933826860155534177465597167611 1  ↩
   0.798505756934379196110285192844457924366
// ./3.1.2 mandel.png 1920 1080 1020  ↩
   0.412765541820958925534057470940751954913 1  ↩
   0.412765541824581805308014281763462349772 5  ↩
   0.213538705176874649138696327097512154281  ↩
   0.213538705180497528912653137922461610287 4
// Nyomtatas:
// a2ps 3.1.2.cpp -o 3.1.2.cpp.pdf -1 --line-numbers=1  --left-footer=" ↩
   BATF41 HAXOR STR34M" --right-footer="https://bhaxor.blog.hu/" --pro= ↩
   color
// ps2pdf 3.1.2.cpp.pdf 3.1.2.cpp.pdf.pdf
//
//
// Copyright (C) 2019
// Norbert Bátfai, batfai.norbert@inf.unideb.hu
//
//  This program is free software: you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation, either version 3 of the License, or
//  (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program.  If not, see <https://www.gnu.org/licenses/>.


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

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

  int szelesseg = 1920;
  int magassag = 1080;
  int iteraciosHatar = 255;
  double a = -1.9;
  double b = 0.7;
```

```
double c = -1.3;
double d = 1.3;

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

A program elején a header fájlok deklarálása után (iostream, a cin, cout miatt főleg; png++/png.hpp, a kép létrehozása miatt; complex, a komplex iterációk és ahogy a címben is említve van a komplex osztályos megoldás miatt) jönnek a változók deklarálása, ezek a változók (`szelesseg`, `magassag`, `iteraciosHatar`, `a`, `b`, `c` és `d`) a futtaási parancsban is fontos szerepet játszanak hiszen az első két argumentum után (ami a ./futtató fájl neve illetve egy png féjlnév, ami a mentett kép neve lesz) ahogy deklarálva vannak, úgynilyen sorrenbe vannak megadva szóközzel elválasztva egymástól. Miután a változók deklarálva lettek, megvizsgéljuk hogy a felhasználó jól futtat-e a programot, tehét hogyha a parans 9 argumentumból áll akkor megadjuk a programban, hogy az egyes változók hányadik értéket kapják meg a parancsból, tehát hogy a program tudja hogy az egyes értékek mit jelentenek számára. Ha vizsont a parancs nem 9 argumentumból áll, akkor kiíratunk az outputra egy üzenetet a felhasználónak, hogy tudja a helyes futtatási használatot, **return -1**-el pedig az operációs rendszernek is jelezzük a hibát.

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

double dx = ( b - a ) / szelesseg;
double dy = ( d - c ) / magassag;
double reC, imC, reZ, imZ;
int iteracio = 0;

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

for ( int j = 0; j < magassag; ++j )
  {
    for ( int k = 0; k < szelesseg; ++k )
      {
        reC = a + k * dx;
        imC = d - j * dy;
        std::complex<double> c ( reC, imC );

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

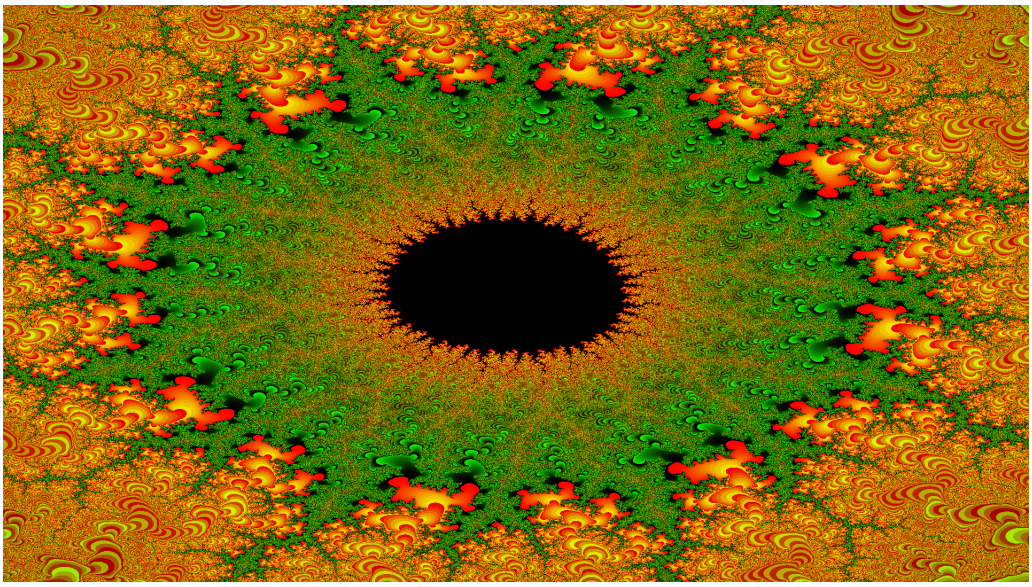
```
          iteracio = 0;

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

              ++iteracio;
            }

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

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

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

}
```

A számítások részben deklaráljuk a `reC`, `imC`, `reZ` és `imZ` (valós és imaginárius részű) változókat, melyekből már látszik is hogy a feladatban sokszerepet fognak játszani a komplex számok. A c=(reC, imC) a haló rácspontjainak megfelelő aktualis pont Úgyanúgy mint az előző programban a két for ciklus segítségével megcsináljuk a rácsot, elindulunk az origóban majd egy `c` pontra ugrunk a c^2+c majd ez az egesz a négyzeten +c és így tovább. Ha sikeres volt minden művelet, akkor kiíratunk a standard kimenetre egy "mentve" üzenetet, melyből a felhasználó tudja is hogy a kép sikeresen el lett készítve, lehet megtekinteni.

Fordítás: **g++ mandelbrotcomplex.cpp -lpng -O3 -o mandelbrotcomplex**

Futtatás: **./mandelbrotcomplex mandelcomplex.png 1920 1080 1020 0.412765541820958925534057470940750.412765541824581805308014281763462349772 5 0.213538705176874649138696327099751215428 1 0.21353**

Kép megnyitása terminálból: **eog mandelcomplex.png**



5.2. ábra. A program áltak megcsinált kép

5.3. Biomorfok

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

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

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

A cikkben arról olvashatunk, hogy Pickover amikor felfedezte a biomorfokat, teljesen meg volt győződve arról hogy felfedezte a természet törvényeit, tehát hogy hogyan néznek ki és alakulnak ki az élő orgazmusok. A cikkben látjuk azt is, hogy a különböző képeket különböző függvények segítségével hozunk létre, hasonlóan a Mandelbrot halmazhoz, elindul a rácson és azon végzi el a képhez tartozó függvényt. A program a komplex számsíkon dolgozik, tehát van "i" számunk, és a c egy állandó .Próbáljuk ki és nézzük meg hogy hogy is mülködik a program.

A forráskód megtalálható a következő linken is: ../Forraskodok/Mandelbrot/5.1/biomorfok.cpp

```
// Verzio: 3.1.3.cpp
// Forditas:
// g++ 3.1.3.cpp -lpng -O3 -o 3.1.3
// Futtatas:
// ./3.1.3 bmorf.png 800 800 10 -2 2 -2 2 .285 0 10
// Nyomtatas:
// a2ps 3.1.3.cpp -o 3.1.3.cpp.pdf -1 --line-numbers=1  --left-footer=" ↩
   BATF41 HAXOR STR34M" --right-footer="https://bhaxor.blog.hu/" --pro= ↩
   color
```

```
//
// BHAX Biomorphs
// Copyright (C) 2019
// Norbert Batfai, batfai.norbert@inf.unideb.hu
//
//  This program is free software: you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation, either version 3 of the License, or
//  (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program.  If not, see <https://www.gnu.org/licenses/>.
//
// Version history
//
// https://youtu.be/IJMbgRzY76E
// See also https://www.emis.de/journals/TJNSA/includes/files/articles/ ↩
   Vol9_Iss5_2305--2315_Biomorphs_via_modified_iterations.pdf
//

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

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

    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double xmin = -1.9;
    double xmax = 0.7;
    double ymin = -1.3;
    double ymax = 1.3;
    double reC = .285, imC = 0;
    double R = 10.0;

    if ( argc == 12 )
    {
        szelesseg = atoi ( argv[2] );
        magassag =  atoi ( argv[3] );
        iteraciosHatar =  atoi ( argv[4] );
        xmin = atof ( argv[5] );
        xmax = atof ( argv[6] );
```

```
        ymin = atof ( argv[7] );
        ymax = atof ( argv[8] );
        reC = atof ( argv[9] );
        imC = atof ( argv[10] );
        R = atof ( argv[11] );

    }
    else
    {
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c ↩
            d reC imC R" << std::endl;
        return -1;
    }
```

Header fájlként megadjuk a képkészítéshez a png++/png.hpp fejlécet, majd ami új az pedig a complex header fájl ami természetesen a komplex számokkal való számolás miatt kell. Ezek után már kezdjük is a főprogramot, melyben legelőször deklaráljuk a változókat, majd meghatározzuk a magasságot, szélességet és a rácsban való mozgáshoz szükségez koordnátákat. Ez a kép kirajzoltatásához a c állandó értékében most nincs "i", ezért a c-nek a valós részéhez (reC) adunk meg 0-tól különböző értéket, az imaginárius, képzett részéhez (imC) pedig 0-t.

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

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

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

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

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

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

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

            int iteracio = 0;
            for (int i=0; i < iteraciosHatar; ++i)
            {

                z_n = std::pow(z_n, 3) + cc;
                //z_n = std::pow(z_n, 2) + std::sin(z_n) + cc;
                if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
                {
```

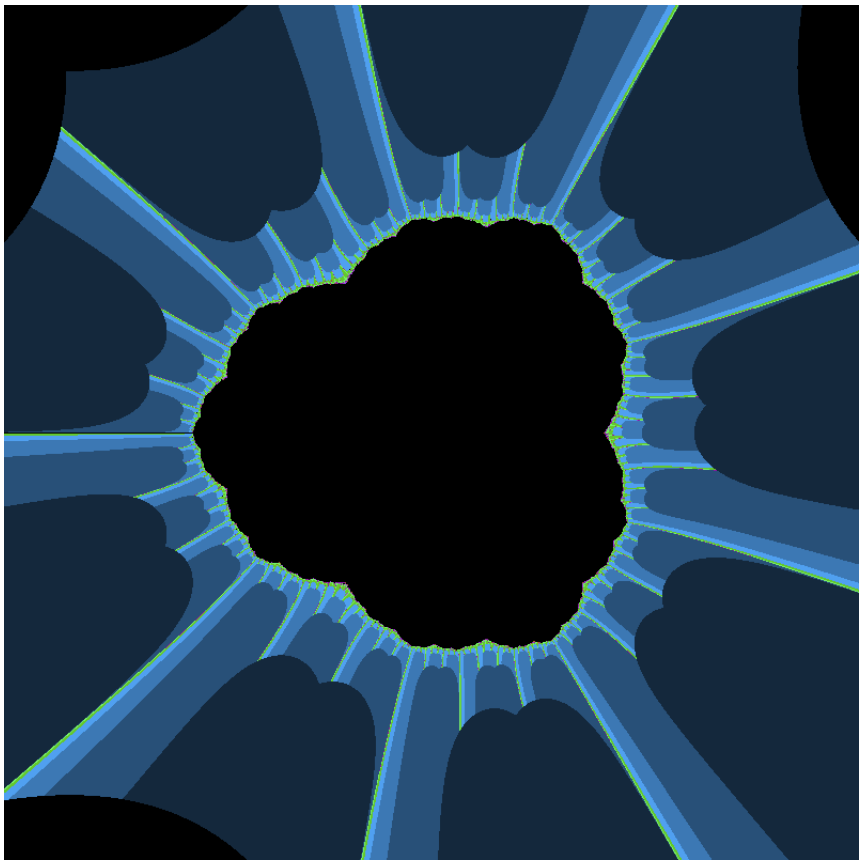
```
                    iteracio = i;
                    break;
                }
            }

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

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

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

}
```

A következő programrészben, az első pixelre állunk, majd elkezdjük a számításokat. A két egymásbaágyzó for ciklussal rácsot teszünk a komplex számsíkra (ahol a `j`-vel a soron, a `k`-al pedig az oszlopon megyünk végig). Az előző complex osztállyal megvalósított Mandelbrot halmazzal szemben, itt a `cc` nem változik, hanem minden vizsgált z rácspontra ugyanaz, állandó lesz (a program eme változata a Júlia halmaz része). A mi programunkban a függvény a kövztkezőképpen néz ki: z_n^3+c. A függűvény változtatásával a készített kép is fog változni.

Fordítás: **g++ biomorfok.cpp -lpng -O3 -o biomorfok**

Futtatás: **./biomorfok biomorf.png 800 800 10 -2 2 -2 2 .285 0 10**

Kép megnyitása terminálból: **eog biomorf.png**



5.3. ábra. Biomorf kép

5.4. A Mandelbrot halmaz CUDA megvalósítása

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

Megoldás forrása: `bhax/attention_raising/CUDA/mandelpngc_60x60_100.cu` nevű állománya.

A feladatban tutorom volt Racs Tamás.

A forráskód megtalálható a következő linken is: ../Forraskodok/Mandelbrot/5.4/mandelpngc_60x60_100.cu

```
// mandelpngc_60x60_100.cu
// Copyright (C) 2019
// Norbert Bátfai, batfai.norbert@inf.unideb.hu
//
//  This program is free software: you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation, either version 3 of the License, or
//  (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program.  If not, see <https://www.gnu.org/licenses/>.
//
// Version history
//
//  Mandelbrot png
//  Programozó Páternoszter/PARP
//   https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063 ↩
   _01_parhuzamos_prog_linux
//
//  https://youtu.be/gvaqijHlRUs
//

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

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


#define MERET 600
#define ITER_HAT 32000

__device__ int
mandel (int k, int j)
{
  // számítás adatai
  float a = -2.0, b = .7, c = -1.35, d = 1.35;
  int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;

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

  // c = (reC, imC) a rács csomópontjainak
  // megfelelő komplex szám
  reC = a + k * dx;
  imC = d - j * dy;
  // z_0 = 0 = (reZ, imZ)
  reZ = 0.0;
  imZ = 0.0;
  iteracio = 0;

  while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
    {
```

```
      // z_{n+1} = z_n * z_n + c
      ujreZ = reZ * reZ - imZ * imZ + reC;
      ujimZ = 2 * reZ * imZ + imC;
      reZ = ujreZ;
      imZ = ujimZ;

      ++iteracio;

    }
  return iteracio;
}

__global__ void
mandelkernel (int *kepadat)
{

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

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

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

void
cudamandel (int kepadat[MERET][MERET])
{
  int *device_kepadat;
  cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));

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

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

int
main (int argc, char *argv[])
{
  // Mérünk időt (PP 64)
  clock_t delta = clock ();
  // Mérünk időt (PP 66)
  struct tms tmsbuf1, tmsbuf2;
  times (&tmsbuf1);

  if (argc != 2)
```

```
    {
      std::cout << "Hasznalat: ./mandelpngc fajlnev";
      return -1;
    }

  int kepadat[MERET][MERET];
  cudamandel (kepadat);
  png::image < png::rgb_pixel > kep (MERET, MERET);

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

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

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

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

}
```

Ugynúgy mint a legelső Mandelbrotos feladat, a mandelbrot halmazt számolja ki, hasonló műveletekkel, viszont van egy nagy különbség a kettő között: míg az első a CPU-t használja a számítások közben, addig ez a program a GPU-t használja. A program futtatása után a számok melyek az időt mutatják magukért beszélnek. Míg a CPU-t használó program 15 másodpercet vett igénybe, addig CUDA-s program mintegy 0.15 másodperc alatt kiszámolja nekünk ugyanazt az eredményt. A két program leírásban nagyon hasonlít az előző feladathoz, mint mondtam itt a számítási idők mások, ezt az időt a program során számoljuk és iratjuk ki, hogy legyen ű vizsonyítási alapunk. Mivel nekem nincs CUDA kártyám ezért nem tudom élvezni ezt a gyors programfutás által gerjeszett élményeket, de a fent belinkelt videón mindez megtapasztalható.

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

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

Megoldás forrása:

A forráskód megtalálható a következő linken is: ../Forraskodok/Mandelbrot/5.5/mandelbrotnagyito.cpp

A Mandelbrot halmaz kiszámítása mellett ezzel a programmal a kiadott képen nagyítást is tudunk végrehajtani. A kép megnyitása után az egerünk görgőjének segítségével tudunk benne nagyítani, illetve touchpaden a kétújjas mozdulattal.

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

int
main ( int argc, char *argv[] )
{
  int szelesseg = 1920;
  int magassag = 1080;
  int iteraciosHatar = 255;
  double a = -1.9;
  double b = 0.7;
  double c = -1.3;
  double d = 1.3;

  if ( argc == 9 )
  {
    szelesseg = atoi ( argv[2] );
    magassag = atoi ( argv[3] );
    iteraciosHatar = atoi ( argv[4] );
    a = atof ( argv[5] );
    b = atof ( argv[6] );
    c = atof ( argv[7] );
    d = atof ( argv[8] );
  }
  else
  {
    std::cout << "Hasznalat: ./3.1.1 fajlnev szelesseg magassag n a b c d"  ↩
       << std::endl;
    std::cout << "Most az alapbeallitasokkal futtatjuk " << szelesseg << "  ↩
       "
    << magassag << " "
    << iteraciosHatar << " "
    << a << " "
    << b << " "
    << c << " "
    << d << " " << std::endl;
```

```
    //return -1;
  }
```

A program eleje ismerős hiszen úgyanaz mint az előző programok esetében: header fájlok dekalrálása, majd a változók beadagolása (`szelesseg`, `magassag`, `iteraciosHatar`, `a`, `b`, `c` és `d`). Most is megvizsgáljuk hogy a parancs amivel a felhasználó futtatta a programot, az tényleg 9 argumentumból áll-e. Ha igen akkor az az egyes argumentumokat, értékeket átadjuk az egyes változóknak, ha viszont az argumentumok száma nem felel meg, akkor használati utasítást adunk a programot futtatónak hogy mi lenne a helyes mód, illetve még segítségképp megmutatjuk azt is hogy mik voltak az általunk adott alap beállítások.

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

  double dx = ( b - a ) / szelesseg;
  double dy = ( d - c ) / magassag;
  double reC, imC, reZ, imZ;
  int iteracio = 0;

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

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

      iteracio %= 256;

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

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

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

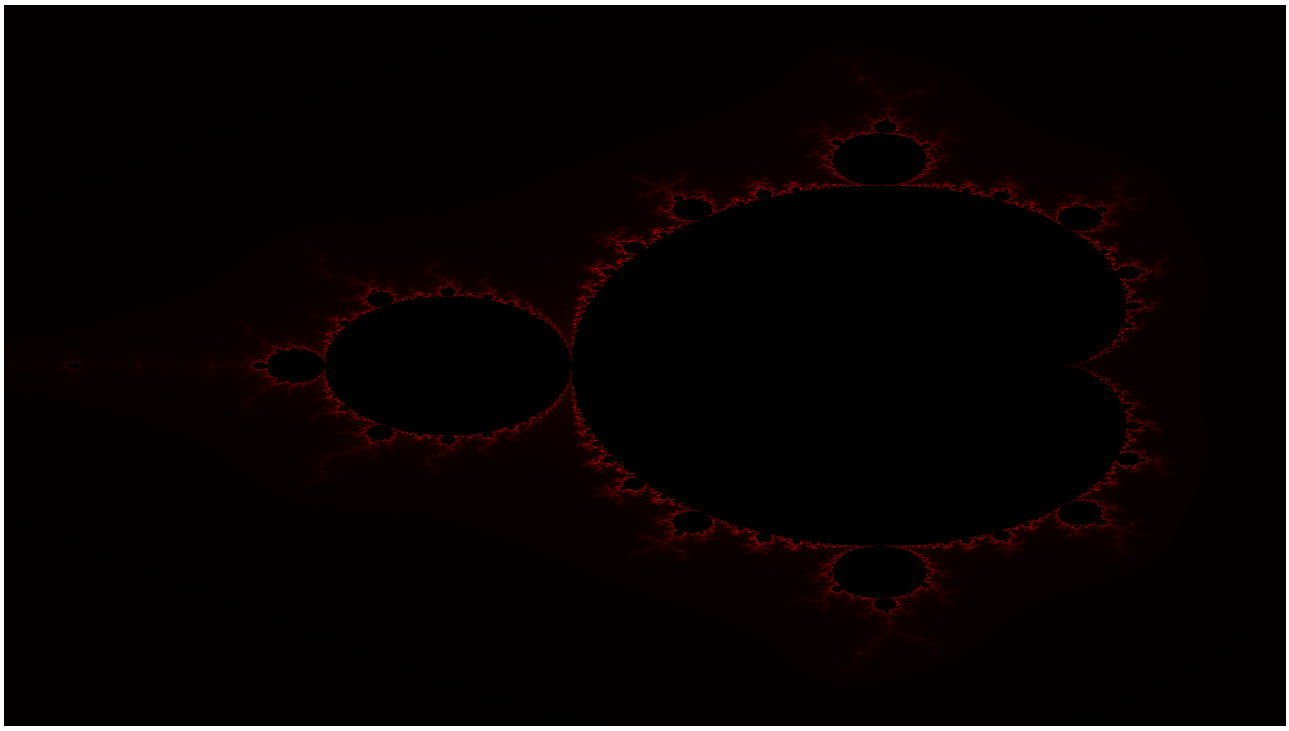
```
}
```

A program utolsó része ugynőgy komplexszámos megoldásssal van csinálva. A két for cuklussal végigmegyünk a rácson `j`-vel a soron, `k`-val az oszlopon. A a `reC`, `imC`, `reZ` és `imZ` (valós és imaginárius részű) változókat használva végigmegyünk minden rácsponton úgy, hogy a c=(reC, imC) a haló rácspontjainak megfelelő komplex szám. A megadott képletet használva megalkotjuk a várt képet, megkapjuk a "mentve" üzenetet, megnyitjuk a képet és tetszésünk szerint nagyítgathatunk a kapott képben aszerint ahogy a feladat legelején bemutattam.

Fordítás: **g++ mandelbrotnagyito.cpp -lpng16 -O3 -o mandelbrotnagyito**

Futtatás: **./mandelbrotnagyito mandelnagyitocpp.png 1920 1080 1020 0.412765541820958925534057470940**
0.213538705176874649138696327099751215428 1 0.213538705180497528912653137922461610287 4

Kép megnyitása terminálból: **eog mandelnagyitocpp.png**



5.4. ábra. C++ mandelbrot nagyító és utazó

5.6. Mandelbrot nagyító és utazó Java nyelven

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

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

A program célja ugyanaz mint az előző programé: nagyítsunk a kapott képben. A java-s program úgy van megírva, hogy itt a bal egérgomb segítségével kell kijelöljük azt a részt amt pontosabban, "közelebbről"

megszeretnénk tekinteni. Ezt a műveletet nagyon sokszor tudjuk megcsinálni egymás után. A programban commentként rengeteg magyarázatot találunk, melyek a különböző sorok, parancsok mülködését írják le.

A forráskód megtalálható a következő linken is: ../Forraskodok/Mandelbrot/5.6/MandelbrotHalmazNagyito.java

```
/*
 * MandelbrotHalmazNagyító.java
 *
 * DIGIT 2005, Javat tanítok
 * Bátfai Norbert, nbatfai@inf.unideb.hu
 *
 */
/**
 * A Mandelbrot halmazt nagyító és kirajzoló osztály.
 *
 * @author Bátfai Norbert, nbatfai@inf.unideb.hu
 * @version 0.0.1
 */
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {
    /** A nagyítandó kijelölt területet bal felső sarka. */
    private int x, y;
    /** A nagyítandó kijelölt terület szélessége és magassága. */
    private int mx, my;
    /**
     * Létrehoz egy a Mandelbrot halmazt a komplex sík
     * [a,b]x[c,d] tartománya felett kiszámoló és nygítani tudó
     * <code>MandelbrotHalmazNagyító</code> objektumot.
     *
     * @param      a              a [a,b]x[c,d] tartomány a koordinátája.
     * @param      b              a [a,b]x[c,d] tartomány b koordinátája.
     * @param      c              a [a,b]x[c,d] tartomány c koordinátája.
     * @param      d              a [a,b]x[c,d] tartomány d koordinátája.
     * @param      szélesség      a halmazt tartalmazó tömb szélessége.
     * @param      iterációsHatár a számítás pontossága.
     */
    public MandelbrotHalmazNagyító(double a, double b, double c, double d,
            int szélesség, int iterációsHatár) {
        super(a, b, c, d, szélesség, iterációsHatár);
        setTitle("A Mandelbrot halmaz nagyításai");
        addMouseListener(new java.awt.event.MouseAdapter() {
            public void mousePressed(java.awt.event.MouseEvent m) {
                x = m.getX();
                y = m.getY();
                mx = 0;
                my = 0;
                repaint();
            }
            public void mouseReleased(java.awt.event.MouseEvent m) {
                double dx = (MandelbrotHalmazNagyító.this.b
                        - MandelbrotHalmazNagyító.this.a)
                        /MandelbrotHalmazNagyító.this.szélesség;
```

```
                double dy = (MandelbrotHalmazNagyító.this.d
                        - MandelbrotHalmazNagyító.this.c)
                        /MandelbrotHalmazNagyító.this.magasság;
                new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+ ↩
                    x*dx,
                        MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,
                        MandelbrotHalmazNagyító.this.d-y*dy-my*dy,
                        MandelbrotHalmazNagyító.this.d-y*dy,
                        600,
                        MandelbrotHalmazNagyító.this.iterációsHatár);
            }
        });
```

A program elején úgyanúgy mint ahogy megszoktuk jönnek a változók deklarálásai, ahol az `x` és az `y` változókkal határoljuk be a nagyítandó terület bal felső sarkát, illetve az `mx` és `my` pedig a kijelölt nagyítandó terület magassága és szélessége. vjuk az ős osztály konstruktorát, beállítjuk az ablak címét (amiben majd nagyítunk) és deklaráljuk az egér kattintás müveletét. Az azonnal következő `mousePressed` függvény segítségével deklaráljuk a nagyítandó terület bal felső sarkát, szélességé és magasságát. A következő `mouseReleased` függvénnyel az a művelet aktivizálódik, amikor kattintjuk az egeret, vonszolással kijelöljük a nagyítani kívánt területet és amikor elengedjük az egérgombot akkor a kijelölt terület újraszámítása kezdődik el, illetve megalkotódik a kinagyított terület, az lesz az aktuális kép amit az ablakban látunk és továbbnagyítunk majd.

```
        addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
            public void mouseDragged(java.awt.event.MouseEvent m) {
                mx = m.getX() - x;
                my = m.getY() - y;
                repaint();
            }
        });
    }
```

Itt deklaráljuk a fentebb már használt függvényt, ami arra szolgál hogy követni lehessen a eglr mozgását, kattintásának feldolgozását, a kijelölés menetét mindazáltal hogy közben megadjuk a `mouseDragged` föggvényt (ami nem más mint a vonszolás, a kijelölés menete, az új szélesség és a magasság bellítása és az újraméretezés által).

```
    public void pillanatfelvétel() {
        java.awt.image.BufferedImage mentKép =
                new java.awt.image.BufferedImage(szélesség, magasság,
                java.awt.image.BufferedImage.TYPE_INT_RGB);
        java.awt.Graphics g = mentKép.getGraphics();
        g.drawImage(kép, 0, 0, this);
        g.setColor(java.awt.Color.BLUE);
        g.drawString("a=" + a, 10, 15);
        g.drawString("b=" + b, 10, 30);
        g.drawString("c=" + c, 10, 45);
        g.drawString("d=" + d, 10, 60);
        g.drawString("n=" + iterációsHatár, 10, 75);
        if(számításFut) {
```

```
            g.setColor(java.awt.Color.RED);
            g.drawLine(0, sor, getWidth(), sor);
        }
        g.setColor(java.awt.Color.GREEN);
        g.drawRect(x, y, mx, my);
        g.dispose();
        StringBuffer sb = new StringBuffer();
        sb = sb.delete(0, sb.length());
        sb.append("MandelbrotHalmazNagyitas_");
        sb.append(++pillanatfelvételSzámláló);
        sb.append("_");
        sb.append(a);
        sb.append("_");
        sb.append(b);
        sb.append("_");
        sb.append(c);
        sb.append("_");
        sb.append(d);
        sb.append(".png");
        // png formátumú képet mentünk
        try {
            javax.imageio.ImageIO.write(mentKép, "png",
                    new java.io.File(sb.toString()));
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
    }
```

Mint a sima nagyítás nélküli programban, itt is lehetővé tesszük a felhasználó számára a pillanatfelvétel készítésének a lehetőségét. Ebben az esetben figyelme vesszük azt is, hogy az adott képernyőfotó hányadik volt, mivel hogy könnyebben tudjunk majd eligazodni a képek között, a kép nevében szerepelni fog hogy mikor csináltuk azt, hányadikként. A fájl nevébe belevesszük, hogy melyik tartományban találtuk a halmazt:

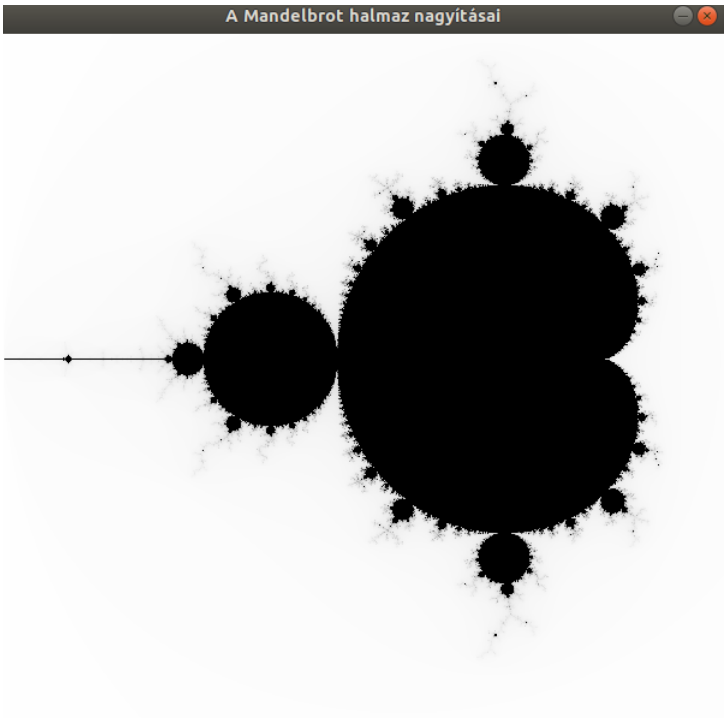
```
    public void paint(java.awt.Graphics g) {
         g.drawImage(kép, 0, 0, this);
        if(számításFut) {
            g.setColor(java.awt.Color.RED);
            g.drawLine(0, sor, getWidth(), sor);
        }
        g.setColor(java.awt.Color.GREEN);
        g.drawRect(x, y, mx, my);
    }

    public static void main(String[] args) {
        new MandelbrotHalmazNagyító(-2.0, .7, -1.35, 1.35, 600, 255);
    }
}
```

A `paint` segítségével legelőször is kirajzoljuk a Mandelbrot halmazt. Ha a számítások éppen futnak, akkor azt hogy melyik sorban tart egy vörös csíkkal jelöljük. Végül pedig kirajzoltatjuk a nagyítandó

terület körvonalát zöld színnel.

Fordítás: **javac MandelbrotHalmazNagyító.java**

Futtatás: **java MandelbrotHalmazNagyító**



5.5. ábra. Java mandelbrot nagyító és utazó

5.6. ábra. Többszöri nagyítás után

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:

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!

6.2. LZW

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

Megoldás videó:

Megoldás forrása:

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása:

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:

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:

6.6. Mozgató szemantika

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

Megoldás videó:

Megoldás forrása:

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:

Tanulságok, tapasztalatok, magyarázat...

7.2. Java életjáték

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

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

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

Tanulságok, tapasztalatok, magyarázat...

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Minecraft-MALMÖ

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

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

Tanulságok, tapasztalatok, magyarázat...

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Tanulságok, tapasztalatok, magyarázat...

10. fejezet

Helló, Gutenberg!

10.1. Magas szintű programozási nyelvek 1

A programozási nyelveket sok tulajdonságuk alapján tudjuk besorolni őket különböző osztályokba, de ténylegesen 3 különböző szintet különítünk el amikor beakarunk sorolni egy nyelvet. Ezek a gépi, assembly szintű és magas szintű nyelvek, utóbbival foglalkozunk jelenleg (2018/2019/2 félév). A magas szintű nyelvvel elkészített programokat forrásprogramnak, illetve forrásszövegnek(source code) nevezzük. Ezek megírásának szabályait szintaktikai szabályoknak nevezzük, értelmezési szabályokból épülnek fel a szemantikai szabályok.

Tudjuk számítástechnikai ismereteinkből, hogy minden egyes feldolgozó egység csak az ő saját nyelvén, saját dialektusán megírt programokat tudja végrehajtani, ebből adódik az, hogy a magas prog. nyelvek forráskódját át kell alakítani valahogyan, például interpreteres(értelmező) technikával vagy fordítóprogramos technikával. A fordító(program) tárgykódot állít elő, ami gépi kód.

A fordító működése során a következő lépéseket hajtja végre, jó esetben:

- lexikálás elemzés
- szintaktikai elemzés
- szemantikai elemzés
- kódgenerálás

Lexikális egységre bontás után ellenőrizzük, hogy szintaktikailag helyes-e a fkód mivel csak azokat tudjuk lefordítani. A futtatható programot a szerkesztő vagy kapcsolatszerkesztő (linker) állít elő. A betöltő viszi a tárba, válik processzussá a program, és a felügyelést a futtattó rendszer felügyeli (pl Out Of Bound kivétel dobása).

Nyelvről nyelvre fordítás történik, léteznek olyan nyelvek (pl C) ami egy előfordító segítségével először a forrásprogramból egy adott nyelvű programot készít ami majd feldolgoztatja magával a nyelv fordítója.

Minden nyelvnek van szabványa, ami az idő során fejlődhet/módosulhat. Ezek leírják, hogy hogyan lehet használni az adott nyelvet (C89 C99). Sokféle fordítót használhatunk pl GNU Compiler Collection fordítás során, és sokféle forráskód szerkesztőt, szövegszerkesztőt is például Kate, vagy integrált fejlesztési környezetet pl Visual Studio, Code::Blocks.

Az interpreteres tech. nem készít tárgykódot, hanem értelmezi és végrehajtja a kódost sorrol sorra.

Imperatív nyelvek A hacker algoritmusokat ír és ez működteti majd a processzort. A programja utasítások sorozata, változókat használhat(tárt érheti el vele) Alcsoportjai eljárásorientált/objektumoriantált nyelvek.

Deklaratív nyelvek Ezek nem algoritmikus p. nyelvek, a problémát írjuk le velük aminek megoldási lépései a nyelvi implementációban van beépítve. Nincsenek memóraműveletek elérhetőek a programozók számára, vagy csak nagyon korlátozottan. Funkcionális és logikai nyelvek alcsoportjai. Ezenkívól lehet még többbe is sorolni őket, a nyelveket.

Forrásszövegek legkisebb részei a karakterek, tehát mindegyiknek van karakterkészlete. Eljárásorientált nyelvek nyelvei elemei: lexikális/szintaktikai/program/fordítási egységek, utasítások, a program. Általában az angol ABC betűit használják a programnyelvek(például van orosz is), és az arab számokat használják.

A lexikális egységeket ismeri fel a fordító. Az operátorok(például ++ C/C++/.. stb) ilyen egységek, de ilyenek például a változók azonosítója (`me`) ezek nyílván korlátozottak az ABC-re és lehet hosszhatáruk is. A nyelv fenntart kulcsszavak is, például `if` elágazás C-ben, ezek nem definiálhatók újra a user által. Megjegyzéseket is támogathat egy nyelv, ezeket a fordító ignorálja, segítik a user áltla írt kód megértését. A konstansok nem módosíthatóak(értékük).

A sorokban az utasítások nyelvtől függően tartalmazhat csak 1 vagy több utasítást, utasításokat ált. a ';' választja el de lehet akár '\n' is..

Egy adattípust 3 dolog ad meg(tartomány, műveletek, reprezentáció). A tartományuk azokat az elemeket tartalmazzák, melyet a megadott típusú konkrét programozási eszköz fölvehet értékként, akár ezek literálok is lehetnek. A műveletek ezeken az elemek használjuk, belső ábrázolási módot az implementáció határozza meg(például szértszórt, folytonos ábrázolás.) Saját típust is lehet definiálni egyes nyelvekben például C.

A nevesített kontansoknak 3 komponense van: név, típus, érték, mindig deklarálni kell! Konstans, tehát értéke nem változtatható meg, sem helye az operatív tárban. Például az úgynevezett magic numbers helyett használjuk. Létrehozásuk például C-ben a #define preprocessor direktívával vagy const.

Változók(komponensek: név, attribútumok, cím, érték). Név azonosító, attribútumok a futás közben viselkedésüket befolyásolja(típus). Explicit/implicit vagy automatikus módon deklarálhatjuk őket(változókat). Automatikus esetnél a fordítóprogram rendel attribútumot azokhoz a változókhoz, amelyknél nincsen explicit módon ez deklarálva, másik két esetben a programozó feladata. A címkomponens a tárbeli helyét tartalmazza az értéknek, a futási idő azon részét, amikor rendelkezik ezzel a változó élettartamának nevezzük. A tárkiosztás lehet statikus/dinamikus, vagy eldől futás előtt a címe vagy csak futás alatt, de a programozó is rendelhet futási idő alatt(abszolút/relatív címzés, vagy megadja mikor legyen cím hozzárendelve a renszer által)

A C-nek vannak aritmetikai(egyszerű) és származtott(összetett) típusai, illetve void. Aritmetikai lehet integrális(egész,karakter,felsorolásos) és valós(float, double). A származtatott: tömb, függvény, mutató, struktúra, union. A C csak egydimenziós tömböket kezel(darabszám megadás szükséges), és azt mindig mutató típusként, van auto dek. alapértelmezés az int.

Két részük van a kifejezéseknek: érték és típus. Operandusok, operátorok, kerek zárójelek az összetevői. Legegyszerű kif. csak egyetlen operandusból áll. Léteznek unáris, bináris, ternáris operátorok, tehát, hogy hány operanduson dolgoznak. Alakjuk lehet prefix(+ a b), infix(a + b), postfix(a b +). A végrehajtási sorrend lehet balról-jobbra, fordítva, vagy precedencia tábla szerint ballról-jobbra. Az operandusok értékének meghatározásának sorrendje vagy szabályozza a nyelv vagy nem pl a C nem. Zárójelezés felülírhatja a precendia táblázat szabályainak alkalmazását, ugye mindig azt kell kiértékelni először. Teljesen zárójelezett infix egyértelmű. Logikai kifejezéseket kilehet értekelni rövidzár módszerrel(konjunkció első tagja

hamis, akkor már nem kell a másik részt vizsgálni) teljes kiértékeléssel. A kif. típusának meghatározásánál kétféle elv: típusegyenértékűség vagy típuskényszerítés. Első esetben egy kétoperandusú operátornak csak azonos típusú operandusai lehetnek, nincs castolás ekkor, vagy eldöntheti az operátor is(==). Második esetnél lehetnek különbözők, de mivel csak azonos ábrázolású operanduszok között végezhetőek műveletek konverzió van. A nyelv definiálja a típust ekkor. C enged például valóst egésszé vagy fordítva kasztolni. A C eleve a típuskényszerítés elvét vallja. A muatókon értelmezett a kivonás és összeadás, egésznek tekinthetők(unsigned int).

Utasítások alkotják az algoritmus lépéseit illetve ez alapján készül a tárgykód. Két csoportja van a deklarációs és a végrehajtható utasítások. Utóbbi csakis a fordítóprogramnak kotyog, mindenféle igényt kérnek, mint például üzemmód beállítása. A hacker csak a névvel rendelkező ő általa birtokolt prog. eszközeit tudja deklarálni.

Megkülönböztetünk: értékadó, üres, ugró, elágaztató, ciklusszervező, hívó, vezérlésátadó, I/O és egyéb utasításokat.

Értékadó utasítás feladata beállítani , frissíteni egy változó értékkomponensét akármikor futási idő alatt. AZ üres utasítás hatására a processzor üres gépi utasítást hajt végre. A ugró utasítás egy megcímkézett utasításra adhatjuk a vezérlést. NEM BIZTONSÁGOS, ÁTLÁTHATATLAN KÓDOT VONHAT MAGA UTÁN!!

Az elágaztató utasítás révén egy adott ponton két aktivitás közül választhatunk végrehajtani. Ált. felépítése if feltétel then tevékenység else tevékenység. A feltétel logikai kifejezés, a tevékenység utasításainak száma függ a nyelvtől, lehet egyetlen egy vagy egy blokknyi utasítás. Az IF-utasítások tetszőlegesen egymásba vihetők. Felmerülhet a csellengő ELSE probléma ekkor, azaz 'if then if then else' esetén kihez tartozik az ELSE? Válasz soféle lehet: Elkerülhető a probléma, ha mindig hosszú IF-utasítást írunk, vagy implementáció függő, vagy a szintaktika alapján egyértelmű.

Többirányú elágaztató utasítás: amikor olyan ponton vagyunk ahol diszjunkt tevékenységek közül egyet kell végrehajtanunk kifejezés alapján. C-en ez a switch statement

```
switch(kifejezés)
{
    case egész_konstans_kif : [tevékenység]
    ...
    default: tevékenység
}
```

A kifejezésnek egészre konvertálhatónak kell lennie! Case ágak kifejezései DISZJUNKTAKNAK KELL LENNIÜK!! Default bárhol szerepelhet. Kiértékelődik a kifejezés majd sorrendre összehasonlításra kerül a case ágak értékeivel, van egyezés végrehajtódik, ha nem akkor a default fog. Ha nincs default akkor egy üres utasítás fog. Akárhogy is, a case és default ágakban szereplenie kell a break utasításnak, azzal lépünk ki a switchből.

A ciklusszervező utasítások lehetővé teszik, hogy a program egy adott pontján egy bizonyos aktivitást többször is megismételjünk. Egy ciklus ált. felépítése: fej mag vég. Fejben vannak az ismétlésre vonatkozó információk vagy a végben. Mag tartalmazza az ismételni kívánt utasítások halmazát. Két radikális típusa van a ciklusoknak végtelen és üres, előbbi soha nem áll le, másik egyszer sem fog végrehajtódni. Ciklusfajták: feltételes, előírt lépésszámú, felsorolásos, végtelen és összetett.

A feltételesnél az ismétlődést nyílván egy igaz/hamis érték fogja meghatározni. Van kezdőfeltétel és végfeltételes feltételes ciklusok. Rendre: a feltétel a fejben, a feltétel a végben(ált.). Első verzió kiértéklődik először, ha igaz végrehajtódik, és ahányszor/ameddig igaz lesz addig végre is fog hajtódni a ciklusmag. Kell lennie valahol emiatt a magban olyan utasítás ami változtat a feltétel értékét. Végtelen vagy üres lehet.

A végfeltételes ciklusok először hajtódnak végre, majd kiértékelődik a feltétel és ha az igaz lesz akkor megint végrehajtódik és ezt folytatjuk amíg hamissá nem értékelődik ki. DE vannak olyan nyelvek, ahol a fordítottja megy végbe, addig hajtódik végre a mag amíg igazzá nem válik a feltétel. Soha nem lehet üres, mert egyszer lefut mindig. Végtelenség engedélyezett.

AZ előírt lépésszámú ciklus azz ismétlésre vonatkozó információk a fejben vannak. Minden esetben tartozik hozzá van ciklusváltozója, és erre felvett értékekre fut le a mag. Egy előírt tartományból vehet fel értékeket, ezt a fejben adjuk meg, mint kezdő és végértéket. Lépésköz ált 1, tehát minden elemet felvesz, de ez módosítható. A tartományt befuthatja növekvőleg/csökkenően.

Lehet előltesztelős vagy hátultesztelős, a működés implementáció függő. Előbbi esetében működés kezdetében definiálva lesznek a ciklusparaméterek, majd a futattó rendszer megnézi, hogy a megadott tartomány üres-e. Ha az, üres ciklus, különben lefut a mag miután a kezdőértéket felvette az ciklusváltozó. Ezután megnézi van-e még olyan érték amit felvehet, ha van újra lefut a mag, és ezek a lépések ismétlődnek. Hátultesztelős ugyanúgy meghatározódnak a paraméterek, de előbb lefut a mag, majd értékelődik ki a feltétel..

A felsorolásos ciklusnak van ciklusváltozója és annak értéke amit a fejben adunk meg, minden felvett érték mellett végrehajtódik a mag. Nem lehet üres, végtelen.

Végtelen ciklusnak sem fejben sem végben nincs információ az ismétlődésre vonatkozóan. Üres ciklus nem lehet. Használatánál a magban kell olyan utasítást alkalmazni, ami megállítja.

Az összetett ciklus 4 ciklusfajta kombinációiból tevődik össze. A fejben bármennyi ismétlésre értetendő információk sorolhatóak fel, jelentések pedig szuperponálódik.

3 vezérlő utasíása van a C-nek a fentebb említettek mellet. Ez a continue; break; és return[kifli]. A continue; ciklusmagban használatos, kihagyja az őt követő utasítások végrehajtását és megvizsgálja a ciklusfeltételt és az alapján kilép vagy újabb ciklusléps végrehajtásba kezd. A break is a magban lehet, befejezteti a cklust, kilép az elágaztató utasításokból. A return; szabványosan bejezteti a függvényt és a hívónak adja vissza a vezérlést.

A programok több egymástól valamennyire független részekből áll az eljárásorientált típusú nyelveknél, ezeket nevezzük programegységeknek. Ebből adóóan kérdések fogalmazhatóak meg velük kapcsolatban.

Ha több egység áll egy program akkor egyben kell az egészet fordítanunk vagy lehetőség van-e arra, hogy egyenként, mivel függetlenek, fordítsuk le őket?

Erre a válasz a nyelv típusától, a nyelv adta lehetőségektől függ. Egyes nyelvekben ténylegesen van lehetőség arra, hogy külön fordítsuk le őket, hiszen fizikailag önálló programokból áll maga a "főprogram". Ezek nem strukturáltak.

Nem ilyen nyelvek esetében csak arra van mód, és ez kötelező, máshogy nem lehet, hogy egyetlen egy egységként legyen fordítva a program, strukturálható ekkor a program kód, viszont a programegységek fizikailag nem függetlenek.

A fenti kettő együttese is valós opció lehet, ekkor függetlenek az egységek, de akármilyen struktúrával bíró egységek vannak.

Emellett feltehetünk olyan kérdésket is, minthogy ha külön fordulnak az egységek, mi adja az önálló fordítási egységet, vagy milyen és milyen viszony van a programegységek között, hogyan kommunikálnak..

Az eljárásorientált nyelvekben a következő programegységekről beszélünk: alprogram, blokk, csomag, taszk.

Eljárásorientált nyelveknél az alprogram egy absztrakciós réteg. Azért mondjuk rá, hogy absztrakciós réteg vagy absztrakciós eszköz, mert formális paraméterekkel látjuk el. Ez egy általánosítást jelent, tehát absztrakció ment végbe.

A feladata az alprogramnak az, hogy a bemeneti adatoknak egy halamzát képezi le kimeneti halmazba, úgy hogy csak a leírás(specifikáció) az adatoknak csak leírását adja meg, az implementálásból de a hátérben zajló leképezésről nincs információnk.

Az alprogramot, absztrakciós volta miatt, olyan helyeken használjuk a programunkban, ahol egy állandóan ismétlődő műveletet szeretnénk kiváltani vele. Azaz nem akarjuk újra meg újra leírni a műveletet, hanem egyszer megírjuk és amikor épp az a művelet szükséges hivatkozunk rá.

Az alprogram áll fejből, törzsből és végből. Első kettőt rendre szoktuk hívni specifikácíiónak és implementációnak is. Ez a formális felépítése.

Ha mint eszköz gondolunk rá, akkor név, formális paraméter lista, törzs és környezet komponensek építi fel.

A név, azonosító, mindig a fejben lesz megtalálható.

A paraméter lista azonosító lista, egy absztrakt/általán műveltet írnak le, a hivás helyén konkrét értéket kell megadnunk nekik.

Kezdetben a formális paraméter listán csak paraméterek nevei lehetettek, később bővült azzal, hogy olyan információkat is megadhatunk amik változtatásokat visznek végbe a paraméterek viselkedésében, futási időben. A lista lehet üres is.

A törzsben utasításokból deklarációs és végrehajtható utasítások lehetnek, néhány nyelv esetében előfordulhat az, hogy kötelező elkülöníteni a kétfajta, előbb említett, utasítást, így 2 része lesz a törzsnek. Illetve vannak nyelvek amikben szabadon keverhetőek.

A globális változók együttese az alprogram környezete. Alprogram lehet eljárás a függvény. A kettő között az a különbség, hogy az eljárás nem tér vissza visszatérési értékkel, hanem paramétereit és/vagy környezetét frissíti, a függvény pedig olyan alprogram aminek van visszatérési értéke, típusa tetszőleges de ez hozzátartozik a függvény leírásához. A függvény is befolyásolhatja paramétereit, ha ez megtörténik mellékhatásról beszélünk.

Az eljrást bárhonnan ahol utasítás állhat meghívható. A meghívás is nyelv függő, valamelyik nyelv esetében explicit meg kell mondani, hogy meghívni akarjuk az eljárást, máshol automatikus az eljárás első sorára kerül a vezérlés. Szabályosan vagy szabálytalanul fejeződhet be egy eljárás, szabályos az amikor az eljárás végére értünk vagy egy kulcsszó használatával kilépünk. Szabálytalan az, nyelv függő ez is, amikor gotoval kiugrunk egy nem benne lévő címkéhez vagy valamilyen eszköz hatására az egész program leáll és az operációs rendszeré lesz a vezérlés.

Csakis kifejezésben lehet függvényt meghívni, ha szabályosan fejeződött be akkor a kifejezésbe tér vissza a vezérlés folytatva annak kiértékelését.

A visszatérési érték meghatározása több módszerrel történhet:

A függvénynév változóként használató a törzsében, a visszatérési érték az utoljára kiszámított érték.

A függvény nevéhez kell az értéket rendelni, itt is az utoljára kapott érték a visszatérési érték.

Külön utasítással adjuk vissza a visszatérési értéket a törzsben, ez befejezteti a függvényt.

Minden megírt programnek kell lennie egy olyan különleges egységének amit főprogramnak nevezünk. A betöltő ennek adja a vezérlést és a többi egységet is ő manipulálja, szabályosan befejeződik a program ha ez szabályosan ér végét.

Szabályosan fejezül be, ha elértük a végét és van visszatérési érték, vagy olyan utasítást használunk ami befejeztei és van visszatérési érték, vagy olyan utasítást használunk ami be is fejezeteti és meg is határozza a visszatérési értéket.

Azokban az esetekben amikor nem goto-t használunk visszatérünk a kifejezés kiértékeléséhez.

Nem szabálosan fejezül be: ha goto utasítást használtunk, nincs visszatérési érték.

A programegység képességei tartozik az, hogy újabb programegységet hívhatnak meg, az így kialakulú láncot nevezzük hívási láncnak. A lánc feje mindig a főprogram. A lánc felfogható stacknek is(verem), hiszen mindig az utoljára "betett" program végzi először be működését(szabályos esetben). A lánc dinamikusan alakul.

Rekurzióról beszélünk ha egy program önmagát hívja meg vagy amikor már egy hívási láncban lévő egységet hív meg. Mindne rekurziós megoldás átírható iteratív algoritmussá, ez a kevesebb memóriafoglalás miatt gyorsabb programot eredményez. Nem minden nyelvben értelmezett a rekurzió.

Előfordulhat olyan funkció egyes nyelvek esetében, amikor megengedett egy másodlagos belépési pont használata, ekkor a törzsnek csupán egy tartománya hajtódik végre.

Olyan programegységet ami egy programegységen belül van blokknak nevezünk.

A blokkoknak van kezdetük, törzsük és végük. Törzsben ugyanúgy lehetnek deklarációs és végrehajtható utasítások, mint előző esetekben, és keverhetőek vagy nem keverhetőek. Nincs paraméterük, nevük létezése nyelv függű. Ott helyezhető el, ahol végrehajtható utasítás állhat. Vezérlést átadni gotoval van lehetőség vagy automatikusan oda kerül a vezérlés a program működése során. Ugyanígy befejezhetődhet. Hatásköröket elhatárolására használatos.

Paraméterkiértékelésről beszélünk, amikor az aktuális paraméterek a formális paraméterekhez csatolódnak az alprogram meghívásakor. Ekkor határozódnak meg, olyan információk, amelyek kommunikácót adják paraméterek megadásánál.

Mindig a formális lista elsődleges, egy darab van belől de aktuális listából(paraméter) annyi van ahányszor meghívjuk a programot, a paraméterkiértékelésnél.

Az dönti el, hogy melyik formális p-hoz melyik akruális paraméter, hogy milyen kötésről beszélünk.

Sorrendi kötés sorrenbeli hozzárendelést jelent. Ez az alapértelmezett a legtöbb nyelv esetében.

Név szereinti kötés mi adhatjuk meg a hozzárendelést az aktuális paraméterlistában. Ezt a technikát néhány nyelv ismeri. E két kötés kombinációjának használata is megtörténhet olyan módon, hogy aktuális plista elején sorrendi, majd név szerinti kötés van.

Ha fix számosságú a formális paraméterek listája akkor vagy meg kell egyeznie az aktuális paraméterek számának megadáskor, vagy kevesebb (csak értéki szerinti pmegadás során) és ekkor alapértelmezett érték rendelődik a meg nem adott helyekre.

Dinamikus számosság esetén a aktuális paraméterek száma is dinamikus, azaz tetszőleges mennyiségű.

A két paraméterlista elemeire további megszorítások is értelmezve vannak, például meg kell egyezniük a paraméterek típusainak vagy legalábbis az aktuális paraméterlista elemeinek konvertálhatóak kellenek leniiük a formális paraméter típusára.

A paraméteradás kommunikációt valósít meg programegységek között, mindig van hívó és hívott, rendre ki ad és kap.

Létezik érték, cím, eredmény, érték-eredmény, név és szöveg szerinti átadás.

Érték szerinti esetben a formális paraméter címkomponenssel rendelkeznek , az aktuális paraméterek értékkomponenssel kell rendelkezniük. E érték meghatározódik a pkiértékelés folyamán és majd végén a címkomponensre kerül. Egyirányban áramlik az információ, mivel értékmásolásról beszélünk, ez hosszadalmas folyamat, minél bonyolultabb struktúrát szeretnénk átadni.

Címes esetben nincsen címkomponensük lefoglalva a hívott alprogram helyén a formális paramétereknek, de címkomponenssel rendelkezniük kell az aktuális paramétereknek a hívó félen. A formális paraméter címkopense az akt paraméter címe, amit átadásra kerül, lesz. Kétirányú információ áramlás, alprogram hívót eléri, ez veszélyes!

Kötelezően rendelkezniük kell címkomponenssel az aktuális paramétereknek eredményes alapú paraméterátadásnál. Az alprogram futási idő alatt nem használja a neki átadott kiértékeléskor megkapott aktuális paraméter címét, de működése végén átmosolja a formális paramétert a címkomponensre. Egyirányú kommunikáció.

Az érték-eredmény verziónál az előző megkötést kiegésziti, hogy az alprogramnak értékkomponenssel is rendelkezniük kell. Kiértékeléskor a hívotthoz adódik az akutális paraméter címe és értéke, de a címét nem használja az alprogram. Végén a formális paraméter átmásolódik az aktuális paraméter címére. Kétirányú, értékmásolás kétszer.

Név szerintinél rögzül az alprogram szövegének környezete, az információ áramlás iránya a szövegkörnyezettől függ.

Szöveg szerintinél amikor a formális paraméter neve legelőször fordul elő a szövegben(alprogramében) akkor kerül rögzítésre a szövegkörnyezet és felülírása a formális paraméternek. A hívás utáni azonnal munkához lát az alprogram. C-ben csak egyetlen (érték szerinti) paraméterátadási mód van.

Az input paraméterek az alprogramnak adnak át információt a hívótól. Output paraméter ennek fordítottja, és van a kettő kombinációja amikor oda-vissza megy az információ.

Blokkok csak programegységen belül lehetnek, vannak nekik formálisan kezdetük, törzsük és végük. Kezdetet alapszó jelöli, törzsben utasítások sorozata lehet.

Nevük csak egyes nyelvekben lehet, no paraméter. Ott helyezhető el, ahol a végrehajtható utasítás is.

Vagy rákerül szekvenciálisan a vezérlés, vagy használhatunk GOTO utasítást egy felette lévő címkére. Nevek hatáskörének elkülönítésére használatos.

Hatáskör alatt azt értjük, hogy a szöveg egy részében egy név ugyanarra a eszközre hivatkozik, tehát láthatóságról beszélünk.

Programegységen belül beszélünk lokális, amiket nem ebben deklarálunk pedig szabad névnek nevezzük.

Ha megállíptjuk egy név hatáskörét, akkor hatáskörkezelésről beszélünk. Hatáskörből kétféle van, statikus és dinamikus hatáskörkezelés.

Fordítási időben történik a statikus hatáskörkezelés, fordító végzi, addig rekurzívan halad a programegységből kifelé ameddig meg nem találja lokális névként. Ha nem volt deklarálva kint akkor hibát dob, ha olyan nyelvről beszélünk ami szerint mindig deklarálni kell, ellenben automatikus deklaráció fog végrehajtódni.

A hatáskör csak befelé terjedhet! Globális név az, ami az adott programegységben nincs deklarálva, de kivül igen.

A dinamikus hatáskörkezelés futási időben zajlik le, hívási láncot felhasználva megy vissza a láncon ameddig lokális névként nem találja a nevet. Ugyanazok az esetek zajlanak le, mint előbb, vagy hiba, vagy automatikus deklaráció.

Név hatásköre amiben deklaráltuk programegység dinamiku shatáskörkezelésnél, és azok amik ebből induló hívási láncokban helyezkednek el.

Statikusnál a forrásszöveg alapján egyértelműen megállapítható a hatáskör. Dinamikus esetben változhat futási időben. Eljárásorientált nyelvek statikus hatáskörkezelést implementálnak legtöbbször.

A C ismeri a blokkotkat és a függvényt, lambdákon kívül nem ágyazhatóak be függvények más programegységekbe, blokkokat tehtjük akárhogyan.

Megjegyezehtő, hogy a függvényeknek Cben az alapértelmezett visszatérési értéke jelölt egész. A függvényekből a return kulcsszóval térünk vissza.

Az extern kulcsszóval rendelkező nevek hatásköre az egész program, illetve a futási idő végéig létezni fognak.

Az auto attribútum miatt hatáskörkezelésük statikus, de láthatóak deklarációjuktól kezdve, dinamikus élettartam, nincs kezdőérték automatikusan.

A register hatására értéke regiszterben tárolódik, ha van szabad.

A static hatására élettartamuk a futási idő, fordítási egység a hatáskörük, van automatikus kezdőérték.

Az absztrakt adattípus enkapszulációt valósít meg, amely következtében a logikailag összeillő adatokat egy struktúrában tudjuk kezelni. Nem ismerjük az implementációt, interfészeken kersztül tudjuk felhasználni programozás során.

A procedurális asbztrakció eszköze a generikus programozási paradigma. Bármely nyelvbe beépíthető, lényege, hogy paraméterezhető forrásszöveg-mintát adunk meg. A mintaszövegből az aktuális paraméterek segítségével állítjuk ki a mintaszövegből a konktrét szöveget, amit majd lefordítunk. Típus paraméterrel paraméterezzük.

Input/Output

Az I/O területének kezelése nagyon eltérő programozási nyelvenként, hiszen az I/O kezelése az operációs rendszer feladata amikből rengeteg féle létezik(Haiku, Linux disztrók, BSD disztók...) és mindegyik másképp kezeli az I/O kérdést. Ahhoz, hogy egy programozási nyelv hordozható legyen leválasztották a nyelvektől(némelyiknél) az I/O, így az I/O implementációját rábízzák az operációs rendszerre, ők csak egy interfészt bocsátanak ki a programozó számára.

Az I/O a perifériákkal való kommunikációért felelős, amely az operatív tárba vár és onnan küld adatokat. Ez mind állományokkal történik meg amiből van logikai és fizikai. A logikai állomány olyan egy reláció aminek van azonosítója(neve) és attribútumai. A fizikai állomány meg a OS szintű perifériákon lévő tényleges adatállomány.

Az input állományból olvasni lehet csak, tehát léteznie kell a műveletek végrehajtása előtt.

A output állomány az előbbi ellentettje, csak írni lehet bele, feldolgozás során jön létre a lemezen.

A kettő kombinációja az input-output állomány e kettő ötvözete.

Folyamatos adatátvitelnél eltér az ábrázolási mód tár és periféria között, a nyelvek ekkor folyamatos karakterláncnak nézik a periféria adatait, de megfelelő belső reprezentálás valósul meg a tárban. Konverzió történik, azaz olvasáskor meg kell mondani, hogyan osszuk fel az adatokat, íráskor pedig melyik helyen mennyi karakterrel jelenjen meg az adat a karakterláncban.

Megadásra pár alapeszköz:

Formátumos módú adatátvitel: meg kell adni az adathoz rendelt formátum segítsgével az érintett karakterek darabszámát és a típust

Szerkesztett módú adatátvitel: maszkot adunk meg amely szerkesztő és átvivendő karakterekből áll. Szerkesztő karakterek adják meg, hogy a pozíción milyen kategóriájú karakternek kell lennie. A masz elemszáma adja meg az érintett karakterek mennyiségét.

Listázott módú adatátvitel: Karaktreslánccba vannak beletéve a tördelő speciális karakterek.

Nincsen konverzó a bináris adatátvitel esetén, azaz a tárbeli és periférián lévő reprezentáció megegyezik.

Ahhoz, hogy állományokkal tudjunk dolgozni deklarálás, összerendelés, állomány megnyitás, feldolgozás és lezárás műveleteket kell végrehajtanunk.

A nyelv szintaxisának megfelelően deklaráljuk a logikai állományt, adunk neki nevet és attribútumokat(nyelv függű).

Összerendelésnél megfeletetjük a logikai állományt a fizikaival. EZ nyelvi eszközzel történik, vagy operációs rendszer szintjén végezzük el.

Műveletek elvégzése előtt meg kell nyitnunk a fájlt, erre operációs rendszer speciális hívásai zajlanak le, az operációs rendszer nyílván tartja, hogy mely fájlok vannak megnyitva. Egy nyitott fájlból akármennyi processzus olvashat, de írni csak egy írhat egy időben.

Feldolgozáskor írunk, olvasunk a fájlból az előbb felsorolt módok alapján a megfelelőnek megfelelően megadjuk a nevet, formátumot, változólistát, maszkot. Kiíró eszközrendszerben a név mellé kifejezéslitát is szerepeltetni kell, ezek kerülnek kiírásra kiértékelés után, szükséges emelett megadni a maszkot/formátumot. Binárisnál rekordot kell adnia a kifejezésnek.

Lezárásnál megint rendszerhívások fognak lefutni, le kell zárni az output-intput, output állományokat és az inputot is. Megszűnik az összerendelés logikai és fizikai állomány között.

Amikor például billentyűzetett használunk vagy a képernyőn megjelenik adat akkor implicit állománnyal dolgoztunk, hiszen maga a billentyűzet és a monitor is fájlként van számon tartva a rendszeren(UNIX alatt minden fájl!), de ez el van "rejtve" a felhasználók elől. Létezik tehát logikai és fizikai állomány is ennél az esetnél. Szabványos perifériákról beszélünk itt(0 STDIN, 1 STDOUT, 2 STDERR).

Ahogy említve volt a C I/O rendszere nem a nyelvhez tartozik, hiszen akkor nem lenne hordozható operációs rendszerek között. Létezik neki bináris és folyamatos módú átvitele, utóbbi keveréke egy szerkeszett átvitel és formátumos átvitelnek. Az I/O függvények minimálisan egy karakter(bájt) vagy karaktercsoport(bájtcsoport) olvasását/írását biztosítják.

Kivételkezelésről általában

Kivételkezeléssel minden program rendelkezik, egy szinten. A kivételkezelés kivételkezelőrendszerrel történik, ez lényegéb olyan, mint operációs rendszer szintjén a megszakításkezelés és annak kezelője. A kivételek olyan történések amik pontosan az előbb említett dolgot váltják ki, megszakítást.

Minden kivétel besorolható valamilyen csoportba, például vannak olyan kivételek amik végtelen ciklust okoznának, ha erre nem lenne eleve egy alapértelmezett kezelés írva, tehát mondjuk a 0-val történő osztás. Vagy annak memórakezelésre vonatkozó kivételek, például kilépünk a processzusunkhoz tartozó címtartományból megsértve ezzel más területét.

Mivel megszabhatjuk/megfigyelhetjük a kivételeket, figyelmen is kívül hagyhatunk néhányat(amit engednek) nyelvi szinten is. A letiltás is kivételkezelésnek minősül. Nem vesz ez esetben tudomást a programunk a kivétel által kiváltott megszakítás legenerált jelre.

A kivételeknek vannak neveik és azonosítóik.

Kivételkezelésről számos kérdést feltehetnénk, nézzünk meg néhányat:

Milyen beépített kivételkezelések lehetnek egy nyelvben? Például 0-val való osztás, vagy out-of-bounds memóriaelérés,..stb

Lehetnek saját kivételeink? Természetesen, ellenben elég gyakran kéne új nyelveket megalkotnunk.

Van-e a hatóköre a kivételkezelésnek? Van, de akár az egész programra is bevezethetnénk kivételkezelést. Javaban pl a kivételkezelő egy blokk.

Kivételkezelésután hogyan fut tovább a program? Futhat tovább, de hiábótól függően le fog, vagy le kell állnia, utóbbi esetben kilövi a kernel. Ignorálhatjuk is a kivételt, ha olyan típusú.

A nyelveknek lehetnek beépített kivételkezelői, például a Javanak a virtuális gépe is képes erre.

Ha Java kód során valamilyen speciális kivétel bekövetkezik akkor létrejön a megfelelő kivétel-osztálynak egy példánya/objektuma. Ekkor a kivétel a Java virtuális géphez kerül, aminek feladata, hogy a megfelelő típusú kezelőnek átadja a kivétel kezelését. Javabána kivételkezelő egy blokk, tehát van láthatósága és azt meg is szablya.

10.2. KERNIGHANRITCHIE

A C nyelv kevés adattípust használ, de járulhatnak minősítők is hozzájuk. (char, int, float, double) Minősítők pl (short(16 bit), long(32 bit) int). Bevezetésük oka az volt, hogy más hosszúságú egészekkel is dolgozhasson a programozó. A signed/unsigned minősítők az előjel meglétére/hiányára utal, ha előjeles az egész vagy char, akkor -határ,+határ intervallumban vehet fel értéket az adott típus, ha előjelmentes, akkor 0,2*határ intervallumban. Valós típusoknál a pontosságot lehet növelni a long-gal.

Állandók is rendelkeznek a fenti típusokkal, minősítőkkel, pl: 2018 signed int is lehet. Léteznek karakterállandók pl 'a', értéke a gépi karakterkészletbeni kódszáma. '0'=48. Karaktersorozatok is lehetnek állandók "alma", bizonyos karakterek pedig escape sorozattal adhatóak meg pl '\a' csipog, '\n' új sor. A '\0' null-karakter karaktersorozat-állandót zár le. Az állandó kifejezés csak állandókat tartalmaz, fordítás során értékelődnek ki. #define MAX AZ enum felsorolt állandó, egész értékek listája. pl enum boolean{no, yes}; Explicit mód megadhatjuk a legelső egészt, pl enum wat {e=5, g,h,z};, avgy összesnek. Állandók neveinek különbözőeknek kell lennie.

A C-ben változók neveinek első karaktere betűnek kell lennie(_ még jó, de nem ajánlott ezzel kezdeni a konyvtári eljárások gyakran ezt használják). A nyelv kulcsszavai nem lehetnek változónevek. Egész típusok(int, char) lebegőpontosak(float,double), lebegőpontosak pontosság eltérő, double pontosabb, de nagyobb hrlyet igényel. A nyelv megengedi, hogy ezek között konvertáljunk, de ekkor például ha floatot konvertálunk egésszé a "pont utáni rész" leválik. Ha egy kifejezésben egy lebegőpontos utasítás van, tehát egyik operandus lebegőpontos akkor az eredmény is az lesz, az egész típus lebegőpontossá alakul.

A vezérléstátadó utasítások a végrehajtás sorrendiségét adják meg. Egy olyan kifejezés C-ben, mint x=666 utasítássá válik, ha ';' rakunk utána pl `x=666;` ';' utasításlezáró jel tehát. A {} zárójelekkel deklarációk és utasítások halmazát fogjuk be egyetlen egy blokkba vagy összetett utasításba, ez nyelvtanliag egy utasítás lesz. Ilyen pl a függvények utasításait behatároló {} vagy if,else,while,for...

Az if-else utasítást döntéshelyzet kifejezésére alkalmazzuk.

```
if(kif)
    utasítás
else
    utasítás
```

A else rész elhagyható, utasítás a kifejezés kiértékelődse szerint az első utasítást hajtja végre, ha az igaz volt, ellenben 2.-at. Az if-else else ága egymásba ágyazott if-else szerkezetnél opcionális, de előfordulhat olyan, hogy nem világos, hogy a jelenlévő else ég melyik if utasításhoz tartozik. Például:

```
if(n>0)
    if(x>y)
        z = x;
    else
        z = y;
```

A fenti forráskódban az else a benső if tulajdonában áll, tagolás is szemlélteti, tehát az else mindig a hozzá legközelebb eső nem else ágat tartalamzó ifhez tartozik. Zárójelezéssél megoldhatjuk, hogy más ághoz tartozzon.

Az else-if utasítás

```
if(kif)
    utasítás
else if(kif)
    utasítás
else
    utasítás
```

Többszörös elágazások programozásának egyik legáltalánosabb módszere, a gép sorra értékeli ki a kifejezéseket és ha valamelyik igaz lesz akkor a hozzátartozó utasítást hajtja végre, majd befejezi a további vizsgálatokat. Például ilyet használhatunk a bináris keresés implementálásának:

```
int binsearch(int x, int v[ ], int n)
{
   int a, f, k;

   a = 0;
   f = n - 1;
   while (a <= f) {
      k = (a + f) / 2;
      if (x < v[k])
         f = k - 1;
      else if (x > v[k])
         a = k + 1;
```

```
      else
         return k;
   }
   return -1;
}
```

Másik eszköz erre(többirányú elágazás) a switch utasítás. Ez összehasonlítja egy kifejezés értékét több egész értékű állandó kifejezés értékével, végül végrehajtva a hozzátartozó utasítást.

```
switch(kif)
{
    case állandó_kif: utasítás
    case állandó_kif: utasítás
    .
    .
    default: utasítások

}
```

Mindegyik case ágban egész konstans(konstans értékú állandó), ha ez egyezik a switches fejében lévő értékkel akkor az utasításai végrehajtásra kerülnek. Default ha egyikével sem egyezik meg, default elhagyható. Az utolsó case-ben lennie kell egy break; utasításnak, ha a felette lévőkben nem volt. Nincs switch break nélkül. A case-k mint címkék működnek, utasításaik végrehajtása után a köv. címkére akar ugrani a switch, de ezt break; utasítással kizárjuk.

Ciklusszervezés while és for utasításokkal: mindkettő fejében van a ciklusfeltétel, ami ha igazzá értékelődik ki végrehajtuk a magot és ezeket a lépéseket addig ismételjük amíg a feltétel igaz. A for fejében a ciklusváltozónak kezdőértéket adunk, megadjuk a feltételt, és egy ciklus utáni értékadó kifejezést is.

```
for(int i=0; i<6; i++); //c99-től
int j=-1;
while(j < 0)
    j++;
```

A for lehet végtelen `for(;;);` és a while is `while(1);`..

A programozó dönthet arról, hogy while vagy fort használ nincs előírva.

```
            void shellsort(int v[ ], int n)
{
   int t, i, j, atm;

   for t = n/2; t > 0; t /= 2)
      for (i = t i < n; i++)
         for (j = i-t;
               j >= 0 && v[j] > v[j + t
```

```
                j -= t
            atm = v[j];
            v[j] = v[j+t];
            v[j+t] = atm;
         }
}
```

A for-ban használhatunk ’,’ operátort is, ez lehetőséget ad arra, hogy egyes helyeken több kif. kerüljön elhelyezésre, pl indexek párhuzamos feldolgozásásánál használhatunk ilyet. pl: megfordítjuk a karakterláncot

```
            #include <string.h>


void reverse(char s[ ])
{
   int c, i, j;

   for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
      c = s[i];
      s[i] = s[j];
      s[j] = c;
   }
}
```

A fv-k argumentumait, deklarációkban változókat elválasztó vessző nem azonos a vesszőoperátorral.

A do-while utasítás hátultesztelős utasítás, azaz egyszer mindenképp végre fog hajtódni a ciklusmag és csak utána nézzük meg, hogy fennáll-e a ciklusfeltétel igaz volta.

```
            do
                utasítás
            while(kif);
```

A break; continue; utasítások a ciklus működését befolyásolják. A break; hatására kilpünk a ciklusból, a continue; hatására kihagyjuk a continue; utasítás utáni utasításokat és új ciklusba kezdünk.

A C nyelvben is van goto utasítás amivel a programkódban létrehozott címkékhez ugorhatunk és ott adjuk át a vezerlést. Nem használják.

```
 for(...)
   for(...) {
      ...
      if (err)
         goto error;
   }
   ...
```

```
error:
   a hiba kezelése
```

A címkéket mindig kettőspont zárja, a goto és a címke bárhol lehet, de a gotoval csak azonos függvényben lévő címke érhető el.

Összefoglalva: utasításuk leírásuk szerint hajtódnak végre, egymás után, kivéve ha nem, jelezzük. Vannak címkézett, kifejezés, összetett, kiválasztó, iterációs és vezérlésátadó utasítások.

A címkézett utasítások nyelvtana:

```
címkézett_utasítás
    azonosító: utasítás
    case állandó_kifejezés: utasítás
    default: utasítás
```

Azonosító címkét csak a goto utasítás céljaként használhatunk, címke csak a blokkjában érvényes, nem deklarálhatók újból. switch használja a case és default címkéket.

A kifejezésutasítások értékadások vagy függvényhívások legtöbbször. Ha nincs benne a kifejezés rész, akkor null-utasítás. Az összetett utasítás(blokk) több utasítást csoportosít így alkotba egyetlen egy utasítást, például a fvdefiníció magja. Egy blokkban csakis 1 azonosítót lehet deklarálni, és nem szereplhet blokkon kívül.

A kiválasztó utasítások a lehetséges végrehajtási utak egyikét választja.

```
kiválasztó_utasítás:
      if ( kifejezés ) utasítás
      if ( kifejezés ) utasítás else utasítás
      switch ( kifejezés ) utasítás
```

A if-s kifejezésnek mutató vagy aritmetikai kifejezésnek szükséges lennie, nyílván ha igazzá értékelődik ki végrehajtódik az if-s utasítások ellenben ha van else akkor azok. Az else ág nem egyértelműségét több if esetében már tárgyaltuk fentebb. switch utasítás csakis egész értékekkel működik és a case részeknél konstans értékeket kell használnunk. Ha végrehajtódott egy case utasításai break; utasítást illik végrehajtani amivel kilépünk a switchből, ellenben "átfolyunk" az alsó casekre. Default címkéből cska 1 lehet. Végrehajtáskor a switches kif. összehasonlításra kerül a case-k konstansaival, ha egyezés van akkor végrehajtódnak a megfelelő utasítások, ellenben default vagy üres utasítás. Iterációs utasítások és vezérlésátadó utasításokat fentebb már tárgyaltuk. ELőltesztelős ciklusok: for, while. Hátultesztelős: do .. while. goto címkéhez ugrik, a címke a gotoval egy függvényben kell lennie, continue a ciklusfeltétel kiértékelését idézi elő ezzel kihagyva utasításokat utána, break ciklusból lép ki, return pedig függvényből térít vissza.

10.3. Programozás

[BMECPP]

Bevezetés: A C++ lehetóvé teszi az objekutmoriemtált és generikus programozást. A C++-t Bjarne Stroustrup fejleztette ki. A C++ a C nyelvre alapszik, A C a C++ részhalmazának tekinthető.

2.1.1. C-ben a föggvényhívások esetén a oaraméterlistát üresen is hagyhatjuk. A C++ esetén a üres paramlterlista egynértékű azzal, hogy megadunk egy void paramétert (a zárójelek közé).

2.1.2. C++-ban kétfajta main függvénymegadás van: int main(), illetve int main (int argc, char* argv[]), ahol az argc a parancssorban megadott parancs argumentumainak szána, az argv pedig maga a paracs argumentumait kapja meg. A return hsználata nem kötelező

2.1.3. A bool típus a logikai igaz vagy hamis (true vagy false) értékeket képes visszaadni. A C-ben ez a bool még nincs, helyette az int vagy az enum típust használjuk erre a célra, úgy lehet megkülönbözettni int típus esetén hogy a 0-ás érték

2.1.4. A C++ már header fájlok terén is bóvült, illetve a C-ben lévő wchar típus a C++-ban beépített pípus lett

2.1.5. A C++-ban megjelent a másik nagy és hasznos újítás, ami az hogy cilusok feltételei megadása közben illetve a fuggvények paramétereinek magadásakor is definiálhatunk változókat.

2.2. C++-ban lehetőségünk van úgyanolyan nevű függvényeket létrehozni, abban az esetben ha az argumentumaik különböznek.

2.4. Ha van egy olyan függvénydeklarálásunk hogy void f(int i) és a mainben f(a)-val hívjuk meg akkor az i szimbólikusan az a lemásolt értékére fog hivathozni. A következő példában az atadott érték az a változó címe lesz, egy pointer, melyre a függvényben a pi szimbólummal hivatkozunk, a *operátorral tudunk hozzáférni az a változóhoz és változtatni is azon. Létezik cím- és értékszerinti paraméterátadás

3. Az osztály alapelve az egységbezárás, mivel egy adott dolog struktúrájának leírásást foglalja magába, úgy kell elpépzelni mint egyfajta kategóriát. Az osztályoknak lehetnek példányai, önálló egyedi amiket objektumoknak nevezünk. Az osztályok esetén tudjuk biztosítani, hogy az objektumok tulajdonságaihoz a program többi rész ene férjen hozzá, ne tudja változtani azokat, ezt a védekezést adatrejtésnek hívjuk. A példákon keresztül lathatjuk hogy astruktúrák esetében nemcsak tagváltozói hanem tagfüggvényei is lehetnek. A this poniterrel tagváltozókra tudunk hivatkozni.

Adatrejtés esetén a private kulcsszót hívjuk segítségül, melyet a struktúra vagy osztály definíciójába írunk. A private: után felsorolt tagváltozók és függvények csak az osztályon belül lesznek láthatóak majd. A C++ esetében érdemesebb az ostályz használni a struktúra helyett, struktúrákat a régebbi C nyelvben használtunk.

A konstruktor olyan specilis tagfüggvény melynek neve megegyezik az osztály nevével és automatikusan meghívódik amikor példányosítjuk az osztályt. Ha nem írunk mimagunk konstruktort az osztályba, attól még van ott egy alapértelmezett konstruktor ami nem csinál semmit. Tehát a konstruktor végzi el az inicializálást, ezzel ellentltben pedig van a destruktor ami pedig a felszabadításra szolgál. A destruktort a hullám jellel való kezdésről könnyű felismerni, melyet az osztály neve követ. A destruktor akkor hívódik meg amikor megszűnik az objektum.

A dinamikus memoriakezelésre C-ben a malloc és a free függvények állnak rendelkezésünkre. C++-ban viszont már nem is függvényeket hanem operátorokat használunk ezek helyett, ez pedig a new és a delete. A new a lefoglalt típusra mutató pointerrel tér vissza, használat után ezt a delete-el szebadítjuk fel. Tömbök lefoglalása és felszabadítsa esetén úgyanígy kell eljárni hozzátéve a tömb "jelet": []. Dinamikus adattagok térolása esetén megvalósítható a FIFO, ahol ha őj elemet adunk hozzá akkor meg kell növelnünk a dinamikus területet, amikor meg kiveszünk, akkor egyel csökkentjük. A másolókonstruktor esetln az új

ojbektumot egy már meglévő alapján inicializáljuk, másolatot akrunk létrehozni. Ha nem írunmk másoló konstruktort akkor bitenként másolunk, ellenkező esetben amásolót hívjuk meg.

A sablonok alatt olyan osztálysablnokat és függvénysablonokat értünk, amelyek deklarásása esetén bizonyos elemeket paraméterként kezelünk. A paramétereket explicit vagy implici módon adhatjuk meg. A függvénysablonok dekalrálását a tamplate kulcsszóval kezdjük, majd kacsacsőrök között felsoroljuk a sablonparamétereket (pl class T vagy name T vayg type T) vesszővel elválasztva.

A hagyományos hibakezelés tovébbfejlesztlse a kivételkezelések, melyekkle sokkal átláthatóbbá tudjuk tenni a programunkat. C++-ban a kivételkezelést try-catch blokkal oldjuk meg (a 190-es oldalon tökéletes szemléltetést létunk erre) ami mar a Pici könyv olvasónaplője esetében bemutattam.

10.4. Python bevezetés

Forrás http://users.atw.hu/progmat/letoltesek/Bevezetes%20a%20molbiprogramozasba.pdf

Általános információk

A Python egy magas szintű, általános célú programozási nyelv. A szkriptnyelvek családjába szokták sorolni. Guido vn Rossum 1990-ben alkotta meg. Egyik legnagyobb erőssége a funckiógazdag standard könyvtára, jól bővíthető.

Python-futtatókörnyezet számtalan különféle rendszerre létezik, így több mobilplatformra is(Apple IPhone, Palm, Windows Phone).

A nyelv népszerűségét nagymértékben az egyszerűségének köszönheti. Elsősorban kliensszoftverek, prototípusok készítésére alkalmas.

A Python nyelv jellemzői

Amikor alkalmazásokat készítünk és szükség van egy olyan program rész megírására ami a probléma szempontjából irrelváns, elkészítése mégis sok időt venne igénybe akkor a Python egy nagyon hasznos opció. Nem kell fordítani, elég az értelmezőnek a Python forrást megadni és az automatikus futtatja is az adott alkalmazást. A Pythonra tekinthetünk valódi programozási nyelvként is, mivel sokkal többet kínál, mint az általános szkript nyelvek vagy batch file-ok.

A Python tulajdonságai

A Python nyelvhez szorosan kapcsolódó standard Python kódkönyvtár rengeteg újrahasznosítható modult tartalmaz ,amelyek meggyorsítják az alkalmazásfejlesztést. Ilyen modulok találhatóak például fájlkezelésrem hálózatkezelésre, különféle rendszerhívásokra és akár felhasználói felület kialakítására is. Illetve az internete is egyre több Python példakód és leírás található amely segít a nyelvelsajátításában.

A Python egy köztes nyelv , nincs szükség fordításra se linkelésre, az értelmező interkaktían is használható. A Python segítségével tömörebb mégis olvashatóbb programokat készíthetünk amelyek rövidebbek(általában), mint a velük ekvivalens C,C++ vagy Java programok. Ennek okai a következtkezők:

A magas szintű adattípusok lehetővé teszik ,hogy összetett kifejezéseket írjunk le egy rövid állításban.

A kódcsoportosítás egyszerű tagolással (új sor, tabulátor) történik, nincs szükség nyitó és zárójelezésre.

Nincs szükség változó vagy argumentumdefiniálására.

A Python nyelv bemutása

Alap vető szintaxis

A kód szerkesztése

A Python nyelv legfőbb jellemzője , hogy behúzásalapu a szintaxisa. A programban szereplő állításokat az azonos szintű behúzásokkal tudjuk csoportba szervezni, nincs szükség kapcsos zárójelrevagy expilict kulcsszavakra. A sor végéig tart egy utasítás. Ha egy utasítás csak több sorban fér el ezt sor végére írt '\'-el kell jelezni.

AZ értelmező minden sort tokenekre bont. A token különböző fajtái a következők: azonosító, kulcsszó, operátor, delimiter, literál. A kis és nagybetűket Pythonban megkülönböztetjük. A Pythonban vannak lefoglaltak kulcssszavak, azok megtekinkhetőek a forrásban.

Típusok és változók

Típusok

Pythonban minden adatot objektumok reprezentálnak. Az adatokon végezhető műveleteket az objektum típusa határozza meg. Nincs szükség változók típusának megadására, azt a rendszer automatikus "kitalálja". Adattípusok a következők lehetnek: számok, sztringek, ennesek, listák, szótárak. A Pythonban is van a NULL értéknek megfelelő típus, itt None a neve.

Változók és alkalmazásuk

Pythonban a változók alatt az egyes objektumokra mutató referenciákat értünk. Maguknak a változóknak nincsenek típusai. A változó értékadása egyszerűen '=' jel segítségével történik. A Python nyelvben egyaránt létezenek globális és lokális változók.

A nyelve eszközei

Például print metódus, amivel sztringet vagy más változót írhatunk ki a konzolra.

A nyelv támogatja a más nyelvekben megszokott if elágazást if/elif/else kulcsszavakkal.

Természetesen támogatja a különféle ciklusok kezelését is, mint a for ciklus, while ciklus. Illetve támogatja a C-ben megismert break és continue kulcsszavakat is.

Címkéket a label paranccsal helyezhetünk e a kód egyes részeiben , és ezekhez a goto paranccsal ugorhatunk.

Függvények

Pythonban függvényeket a def kulcsszóval definiálhatunk. A függvények rendelkeznek paraméterekkel, amelyeknek a szokásos megkötésekkel és szintaxissal alapértelmezett értékeket is adhatunk. A függvényekenek egy visszatérési értékük van azonban visszatérhetnek például ennesekkel is.

Osztályok és objektumok

A Python nyelv támogatja a klasszikus, objektum orientált eljárásokat. Definiálhatunk osztályokat, amelyek pédényai objektumok. Az osztályoknak lehetnek attribútumaik: objektumok, illetve függvények. Ez utóbbiakat metódusnak vagy tagfüggvénynek is hívjuk. Ezenkívül az osztályok örökölhetnek más osztályokvól is. Az osztályoknak lehet egy speciális, konstruktor tulajdonságú metódusa, az __init __.

Modulok

A Python a fejlesztés megkönnyítése érdekében sok szabványis modult tartalmaz. Például: appuifw, messaging,syinfo,camera,audio.

Kivételkezelés

A Python nyelv támogatja a váratlan helyzetek kezelésére az úgynevetett kivételeket. Ebben,egyszerű esetben a try kulcsszó után írva szerepel az a kódblokk, amelyben a kivételes helyzet előállhat, majd ezt az expect blokk követi , amelyre a hiba esetén kerül a vezérlés, illetve opcionálisan egy else ág. Az utóbbi kettőt kiválthatja egyetlen finally blokk.

III. rész

Második felvonás

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

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

Köszönet illeti a NEMESPOR, https://groups.google.com/forum/#!forum/nemespor, az UDPROG tanulószoba, https://www.facebook.com/groups/udprog, a DEAC-Hackers előszoba, https://www.facebook.com/-groups/DEACHackers (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésükért és hasznos észrevételeikért.

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