

Magasabb szintű programozási nyelvek I.

Labor jegyzőkönyv

Ed. BHAX, DEBRECEN,
2020. február 22, v. 0.0.5

Copyright © 2019 Dr. Bátfai Norbert

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

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

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

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

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

COLLABORATORS

	<i>TITLE :</i> Magasabb szintű programozási nyelvek I.		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, Bátfai, Margaréta, Ács Human, Person	2020. április 18.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	A Brun tételes feladat kidolgozása.	nbatfai
0.0.5	2020-02-15	Forked and initied bhax derived bhax-derived	rkeeves

Ajánlás

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

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

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	4
2. Helló, Turing!	6
2.1. Végtelen ciklus	6
2.2. Lefagyott, nem fagyott, akkor most mi van?	10
2.3. Változók értékének felcserélése	13
2.4. Labdapattogás	16
2.5. Szóhossz és a Linus Torvalds féle BogoMIPS	20
2.6. Helló, Google!	24
2.7. A Monty Hall probléma	33
2.8. 100 éves a Brun tétel	34
3. Helló, Chomsky!	40
3.1. Decimálisból unárisba átváltó Turing gép	40
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	40
3.3. Hivatkozási nyelv	40
3.3.1. BNF	40
3.3.2. inline functions	41
3.3.3. új típusok	41

3.3.4. struct initialization	42
3.3.5. mixing decl and code	42
3.3.6. implicit fn	43
3.4. Saját lexikális elemző	43
3.5. Leetspeak	44
3.6. A források olvasása	47
3.7. Deklaráció	49
4. Helló, Caesar!	54
4.1. double ** hááromszögmátrix	54
4.2. C EXOR titkosító	61
4.3. Java EXOR titkosító	65
4.4. C EXOR törő	65
4.5. Neurális OR, AND és EXOR kapu	70
4.6. Hiba-visszaterjesztéses perceptron	70
5. Helló, Mandelbrot!	79
5.1. A Mandelbrot halmaz	79
5.2. A Mandelbrot halmaz a std::complex osztállyal	88
5.3. Biomorfok	93
5.4. Mandelbrot nagyító és utazó C++ nyelven	100
6. Helló, Welch!	119
6.1. Első osztályom	119
6.2. LZW	122
6.3. Fabejárás	130
6.4. Tag a gyökér	131
6.5. Mutató a gyökér	139
6.6. Mozgató és másoló szemantika	146
7. Helló, Conway!	163
7.1. Hangyaszimulációk	163
7.2. Java életjáték	164
7.3. Qt C++ életjáték	164
7.4. BrainB Benchmark	165
7.5. Vörös Pipacs Pokol/19 RF	165

8. Helló, Schwarzenegger!	166
8.1. Szoftmax Py MNIST	166
8.2. Mély MNIST	166
8.3. Minecraft-MALMÖ	166
9. Helló, Chaitin!	167
9.1. Iteratív és rekurzív faktoriális Lisp-ben	167
9.2. Gimp Scheme Script-fu: króm effekt	167
9.3. Gimp Scheme Script-fu: név mandala	167
10. Helló, Gutenberg!	168
10.1. Programozási Alapfogalmak	168
10.1.1. Gépi kód, assembler, magasabb szintű nyelvek	168
10.1.2. Adattípusok	177
10.1.3. A nevesített konstans	178
10.1.4. A változó	178
10.1.5. Alapelemek az egyes nyelvekben	178
10.1.6. Kifejezések	180
10.1.7. Utasítások	182
10.1.7.1. Deklarációs Utasítások	182
10.1.7.2. Végrehajtható Utasítások	182
10.1.7.2.1. Értékadó utasítás	183
10.1.7.2.2. Üres utasítás	183
10.1.7.2.3. Ugró utasítás	183
10.1.7.2.4. Elágaztató utasítások	183
10.1.7.2.4.1. Kétirányú elágaztató utasítás	183
10.1.7.2.4.2. Többirányú elágaztató utasítás	183
10.1.7.2.5. Ciklusszervező utasítások	184
10.1.7.2.5.1. Feltételes ciklus	184
10.1.7.2.5.2. Előírt lépésszámú ciklus	184
10.1.7.2.5.3. Felsorolásos ciklus	185
10.1.7.2.5.4. Végtelen ciklus	185
10.1.7.2.5.5. Összetett ciklus	185
10.1.7.2.5.6. C példák	185
10.1.7.2.6. Vezérlő utasítások C-ben	185

10.1.8. A programok szerkezete	186
10.1.8.1. Alprogramok	186
10.1.8.2. Hívási lánc, rekurzió	188
10.1.8.3. Másodlagos belépési pontok	188
10.1.8.4. Paraméterkiértékelés	188
10.1.8.5. Paraméterátadás	188
10.1.8.5.1. érték szerinti	189
10.1.8.5.2. cím szerinti	189
10.1.8.5.3. eredmény szerinti	189
10.1.8.5.4. érték-eredmény szerinti szerinti	189
10.1.8.5.5. név szerinti	189
10.1.8.5.6. szöveg szerinti	189
10.1.8.6. A blokk	190
10.1.8.6.1. Hatáskör	190
10.1.8.6.2. Fordítási egység	190
10.1.8.7. C	191
10.1.8.7.1. extern	191
10.1.8.7.2. auto	192
10.1.8.7.3. register	192
10.1.8.7.4. static	192
10.1.9. IO	192
10.1.9.1. Állományok kezelése	193
10.2. Programozás bevezetés	194
10.2.1. Alapismeretek	194
10.2.1.1. Adattípusok	194
10.2.1.2. Változók és aritmetikai kifejezések	195
10.2.1.3. Változók és aritmetikai kifejezések	196
10.2.1.4. Szimbolikus konstansok	196
10.2.1.5. Tömbök	196
10.2.1.6. Argumentumok érték szerinti átadása	197
10.2.1.7. Karakter tömbök	197
10.2.1.8. Érvényességi határ, külső változók	198
10.2.2. Típusok, Operátorok és Kifejezések	198
10.2.2.1. Változó nevek	198

10.2.2.2. Adattípus és méret	199
10.2.2.3. Konstansok	199
10.2.2.4. Deklarációk	200
10.2.2.5. Aritmetikai műveletek	200
10.2.2.6. Relációs és logikai operátorok	200
10.2.2.7. Típus konverzió	201
10.2.2.8. Bitenkénti műveletek	201
10.2.2.9. Értékadási operátorok és kifejezések	201
10.2.2.10. Feltételes kifejezések	202
10.2.2.11. Precedencia, kiértékelési sorrend	202
10.2.3. Vezérlési szerkezetek	202
10.2.3.1. Utasítások, blokkok	202
10.2.3.2. If-else	202
10.2.3.3. Else-if	203
10.2.3.4. Switch	203
10.2.3.5. Loops(for, while)	204
10.2.3.6. Loops(Do-while)	204
10.2.3.7. Break and continue	205
10.2.4. Függvények és programstruktúra	205
10.2.4.1. Alapok	205
10.2.4.2. Nem int visszatérési értékű függvények	205
10.2.4.3. Külső változók	206
10.2.4.4. Érvényességi Tartomány	206
10.2.4.5. Statikus változók	206
10.2.4.6. Regiszter változók	207
10.2.4.7. Blokk struktúra	207
10.2.4.8. Inicializáció	207
10.2.4.9. Rekurzió	207
10.2.5. Bevitel és kivitel	208
10.2.5.1. Szabványos kimenet és bemenet	208
10.2.5.2. Formátumozott kimenet- printf	208
10.2.5.3. Formattált bemenet - Scanf	209
10.2.5.4. Változó hosszú argumentum listák	210
10.2.5.5. Fájl hozzáférés	210

10.2.5.6. Hibakezelés és Exit	211
10.2.5.7. Soronkénti bemenet és kimenet	212
10.3. Programozás	212
10.3.1. C és C++	212
10.3.2. Function overloading	213
10.3.3. Alapértelmezett függvény argumentumok	213
10.3.4. Paraméterátadás referencia típussal	213
10.3.5. Objektum orientáltság	214
10.3.6. Dinamikus adattagot tartalmazó osztályok	215
10.3.7. Friend függvények és osztályok	215
10.3.8. Tagvátózők inicializálása	216
10.3.9. Statikus tagok	217
10.3.10. Beágyazott (nested) definíciók	218
10.3.11. Konstansok és inline függvények	218
10.3.12. C++ IO alapjai	219
10.3.12.1. Szabványos adatfolyamok	220
10.3.12.2. Manipulátorok és formázás	221
10.3.12.3. Állománykezelés	222
10.3.13. Operátorok és túlterhelésük	222
10.3.13.1. Függvény szintaxis és túlterhelés	223
10.3.13.2. Speciális operátorok túlterhelése	223
III. Második felvonás	225
11. Helló, Arroway!	227
11.1. A BPP algoritmus Java megvalósítása	227
11.2. Java osztályok a Pi-ben	227
IV. Irodalomjegyzék	228
11.3. Általános	229
11.4. C	229
11.5. C++	229
11.6. Lisp	229
11.7. My Little Ponys	229

Ábrák jegyzéke

2.1. For syntax	7
2.2. Alma no flag fail	9
2.3. Alma With flag	9
2.4. builtins	10
2.5. shiftl_uchar	21
2.6. shiftr_uchar	21
2.7. 2s complement	23
2.8. ullshift	23
2.9. bogomips kimeneten	24
2.10. pagerank kapcsolati irányított multigráf	25
2.11. pagerank link mátrix sanity check	26
2.12. pagerank vektor	27
2.13. pagerank trf	28
2.14. pagerank cpp konvergál	30
2.15. pagerank openoffice konvergál	30
2.16. A B_2 konstans közelítése	37
2.17. Az c++ B_2 konstans közelítés eredményei	39
3.1. leetspeak	45
4.1. mem mx	55
4.2. mem free	57
4.3. A double ** háromszögmátrix a memóriában	59
4.4. exorcism	62
4.5. nn input	71
4.6. nn layers	71
4.7. nnmxmult	72

4.8. sigmoid	72
5.1. $c = \{0,0\}$	80
5.2. $c = \{0.3,0\}$	81
5.3. $c = \{0.3,0.1\}$	82
5.4. $c = \{0.3,0.2\}$	83
5.5. $c = \{0.3,0.7\}$	84
5.6. A Mandelbrot halmaz win, csak iter számok	87
5.7. A Mandelbrot halmaz win, színes	88
5.8. A Mandelbrot halmaz win, színes, határ állítás	88
5.9. mandelcx comp	89
5.10. mandelcx	90
5.11. mandel step	94
5.12. biomorph iter	95
5.13. biomorph comp	96
5.14. biomorphnyuszi	96
5.15. biomorph	97
5.16. zoom rect	101
5.17. zoom	101
5.18. zoom quick succession overlap	102
5.19. tracer 0	103
5.20. tracer 1	103
6.1. Lzw C output	126
6.2. Traversals	131
6.3. ZLW out	142
7.1. qt alap hangya 0	163
7.2. qt alap hangya wraparound	164
10.1. Foo és Foo Main	172
10.2. FooABI és FooABI Main	174

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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.xsl
Build the book set list...
Build the listings...
XSLT stylesheets DocBook - LaTeX 2e (0.3.10)
=====
Stripping NS from DocBook 5/NG document.
Processing stripped document.
Image 'dblatex' not found
Build bhax-textbook-fdl.pdf
'bhax-textbook-fdl.pdf' successfully built
```

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



A DocBook XML 5.1 új neked?

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

I. rész

Bevezetés

DRAFT

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.

DRAFT

II. rész

Tematikus feladatok

DRAFT

**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.

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás videó: No link

Megoldás forrása:

- [Basic infinite for loop](#)
- [For with linux sleep](#)
- [While infinite](#)
- [Macros for OS call handling](#)

Végtelen ciklusokra gyakorlatban sok példa akad: szerverek, game loop-ok, interaktív interpreter terminálhoz. A végtelen ciklus azonban olykor nem várt. Például lehet bug, de akár lehet a természetes működés része. Erre jó példa a Tanár Úr által említett PageRank algoritmus. Én azonban, a hozzáadott munka miatt másik példát hoznék: mérnöki CAE végeselemes számítások (pl. Ansys) esetén egyes esetekben a megoldás nem konvergál n lépés után sem, és ez előre nem látható be. Erre a célra a felhasználó megadhat vészhelyzeti kilépési konvergencia kritériumokat. Ez nem matematikai kényszert jelent, hanem szimplán, ha n iteráció, vagy t idő elteltével a megoldáshoz nem jutunk közelebb, akkor a program ki break-el ciklusból.

Alább egy minimális végtelen ciklus látható for-ral. Amit vegyünk észre, az az hogy az egész "üres". Ezt a [for loop szintaktikai definíciója teszi lehetővé](#), hisz nyíltan kifejezi az optional, hogy nem kötelező megadni. Ez persze egyéb hatásokkal is jár, például, hogy a `for (; ;)`-ban használhatjuk az ehhez képest külső scope-ban deklarált változókat.

for loop

Executes *init-statement* once, then executes *statement* and *iteration_expression* repeatedly, until the value of *condition* becomes false. The test takes place before each iteration.

Syntax

formal syntax:

```
attr(optional) for ( init-statement condition(optional) ; iteration_expression(optional) ) statement
```

informal syntax:

```
attr(optional) for ( declaration-or-expression(optional) ; declaration-or-expression(optional) ; expression(optional) )  
statement
```

2.1. ábra. For syntax

```
1 main ()  
2 {  
3     for (;;) ;  
4  
5     return 0;  
6 }
```

Alább a fenti példa látható egy `while` loop-pal. Értелеmszerűen a `conditional`-ban mivel egy `true` bool literal-t írtuk, így ez mindig igazra fog kiértékelődni.

```
1 #include <stdbool.h>  
2 int  
3 main ()  
4 {  
5     while(true);  
6  
7     return 0;  
8 }
```

Alább egy `while`-t használtunk, de mostmár alszunk is. A `sleep` viszont sajnos OS dependens call (mi sem mondja el jobban mint az `unistd` include-olása). Mivel az OS feladata az ütemezés, plusz a megszakítások kezelése (hisz ugye az egész lelke az időzítés ezáltal a timer). `sleep` esetén jelezzük az OS-felé, hogy nem kérünk CPU időt (azaz váltsa a process status-t), viszont emellett viszont a megfelelő idő elmultával (megszakításos alapon) újból kerülünk számításra várakozóba. A probléma természetesen annyi, hogy scheduling-tól függően, lehet soha nem kerülünk vissza még az idő letelte után sem CPU-ra.

```
1 #include <unistd.h>  
2 int  
3 main ()  
4 {  
5     for (;;) ;
```

```
6     sleep(1);
7
8     return 0;
9 }
```

Mindenesetre az alvás egy OS specifikus dolog, pl.: [nanosleep](#). Ha több fajta OS-n kell futni, ahhoz sajnos trükközni kell. Egy barbár megoldás a preprocessor használata. Alábbi példában az látható, hogy preprocessor által értelmezett szöveget is elhelyeztem a fájlban. Technikailag ez annyit tesz, hogy compile előtt a preprocessor elődolgozza a forrásfájlt, és ez példánkban azzal jár

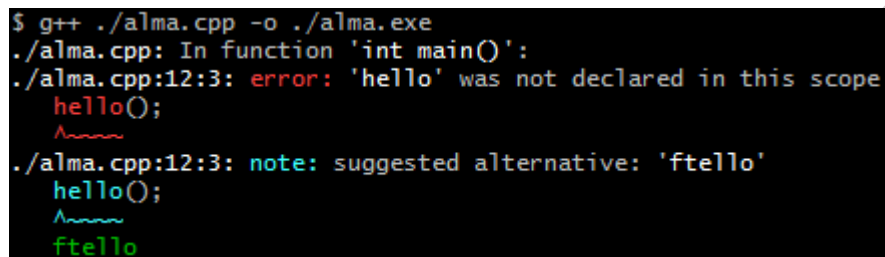
```
1  #ifdef __linux__
2      #include <unistd.h>
3  #elif _WIN32
4      #include <windows.h>
5  #else
6      #error Unsupported
7  #endif
8
9  void SleepProxy(int sleepMs)
10 {
11     #ifdef __linux__
12         usleep(sleepMs * 1000);
13     #elif _WIN32
14         Sleep(sleepMs);
15     #endif
16 }
17
18 int
19 main ()
20 {
21     for (;;)
22         SleepProxy(1000);
23
24     return 0;
25 }
```

Egyébként a gcc mellé betudunk passzolni flageket. Például `-DALMA`-val gyakorlatilag azt mondjuk a preprocessoroknak, hogy vegye úgy mintha `#define ALMA`. Ez pontosan annyira szép mint amennyire elegáns, de ez van. Nézzük csak meg ezt az almát!

```
1  #include <iostream>
2
3  #ifdef ALMA
4  void hello()
5  {
6      std::cout << "Hello world!" << std::endl;
7  };
8  #endif
9
10 int main ()
11 {
```

```
12  hello();  
13  return 0;  
14 }
```

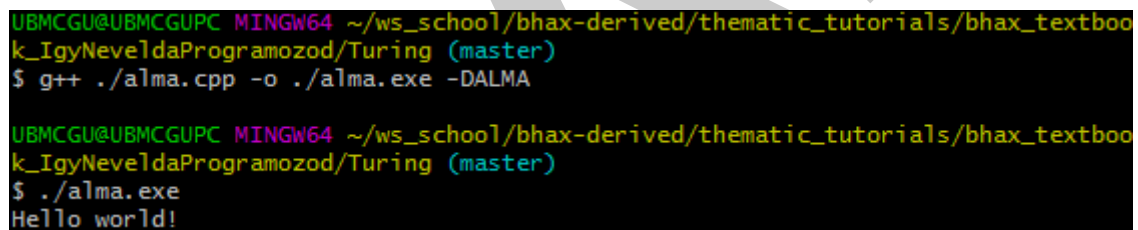
Láthatóan ez nem fog lefordulni ha nincs definiálva az ALMA, de azért tegyünk próbát!



```
$ g++ ./alma.cpp -o ./alma.exe  
./alma.cpp: In function 'int main()':  
./alma.cpp:12:3: error: 'hello' was not declared in this scope  
    hello();  
    ~~~~~  
./alma.cpp:12:3: note: suggested alternative: 'ftello'  
    hello();  
    ~~~~~  
    ftello
```

2.2. ábra. Alma no flag fail

Miért azt írja hogy hello was not declared in scope? Nos a preprocesszor végigment a forrásfájlon, és, mivel nem volt ALMA definiálva, ezért a compilation stage-ben az a kód részlet ami #ifdef-be volt zárva nem létezett. Próbáljuk ki azt hogy g++ alma.cpp -o alma -DALMA! Az alábbi screenshot-ról látszik, hogy mivel lusta vagyok, ezért a git bash-ből csináltam.



```
UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo  
k_IgyNeveldaProgramozod/Turing (master)  
$ g++ ./alma.cpp -o ./alma.exe -DALMA  
  
UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo  
k_IgyNeveldaProgramozod/Turing (master)  
$ ./alma.exe  
Hello world!
```

2.3. ábra. Alma With flag

Ha belenézel az STL-be, akkor az is tele lesz ilyenekkel, hisz különböző OS-ekkel kell tudni használni, és a rendszerhívások általában eltérőek. Egyébként ha valami IDE-vel dolgozol, ott gyakran még szépen be is szürkíti azokat a részeket a kódban amelyeket ki fog hagyni. Itt semmi mágia nem történik, hanem simán nem a forráskódod megy a compiler-nek, hanem előtte van egy preprocesszor, ami feldolgozza ezeket az utasításokat. Makrók ugyanilyenek. Tehát nem runtime kiértékelhető az #ifdef hanem egy utasítás a preprocesszornak. Leggyakrabban #ifdef-el header guard-ként fogsz találkozni (vele kb. ekvivalens a nem mindenhol támogatott #pragma once).

Na jó, de ez nem válaszolja meg a kérdést, hogy miért nem bírja a gépem OBS-el együtt a notepad++-t.

Mármint, akarom mondani, milyen ügyes kis dolog ez a preprocesszor! De csak ennyit tud? A válasz nem. Alábbi példában filename, linenumber-t íratunk ki.

```
1  #include <iostream>  
2  
3  int main ()  
4  {
```

```
5  std::cout << "Hello from " << __FILE__ << ", I'm from line " << __LINE__ << "\n";  
    << std::endl;  
6  return 0;  
7  }
```

```
$ ./builtins.exe  
Hello from ./builtins.cpp, I'm from line 5
```

2.4. ábra. builtins

Most pedig következzen egy összevagdossott code snippet egy régi projektéből. A lényeg annyi, hogy unit test-eléshez, egy makró alapú library-ról volt szó. (Csak poénból, tudom hogy van gtest).

```
1  #define EXPECT(case, arg0, arg1, pred, msg) \  
2  { bool res = pred(arg0, arg1); case.do_test(false, __FILE__, __LINE__, res, msg, \  
    arg0, arg1); }  
3  
4  #define EXPECT_EQ(case, arg0, arg1) EXPECT(case, arg0, arg1, pred_eq, "Expect \  
    failed ==")  
5  
6  void test_is_subs(TestCase& tc)  
7  {  
8      EXPECT_EQ(tc, true, is_ss Subs("ab", "ab"));  
9  }
```

A fenti snippetben látszik, hogy a `#define`-al meghatározott dolgok többek mint "változók". Ezeket preproceszor makróknak hívjuk, és az előfeldolgozáskor expandáljuk őket. Azaz először az `EXPECT_EQ` expandálódik `EXPECT(case, arg0, arg1, pred_eq, "Expect failed ==")`. Az `Expect`-ben meg egy pici scope-ot létrehozunk. A lényeg ebből annyi, hogy nem kell megijedni tőle, jóra is lehet használni. Annyi hogy az expanzióval óvatosan, mert elég nehéz debuggolni. (ugyanolyan szívás mint az STL-es template alapú hibákat, azaz nem egy one liner lesz.)

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

Lehet-e írni olyan programot amely minden programról megmondja, hogy lefut-e?

Megoldás videó: [No link](#)

Megoldás forrása:

- [Turing 100 \(pseudocode\)](#)
- [Turing 1000 \(pseudocode\)](#)

A probléma elég fontos. Miért is? Nos, a probléma az, hogyha ez nem lehetséges, akkor soha nem fogjuk tudni bizonyítani az összes program helyességét. Tegyük fel, hogy akkora haxorok vagyunk, hogy meg

tudjuk írni a `Lefagy` függvényt, amely tetszőleges programról el tudja dönteni, hogy van-e benne végtelen ciklus:

A borbély az aki azok haját vágja, akik nem teszik meg ezt saját maguknak. De a borbély vágja-e a saját haját? Nos, tegyük fel levágja: Ebben az esetben ő levágta a haját egy olyan embernek aki egyébként ezt megteszi saját magának. Ok, tegyük fel nem vágja: Viszont ebben az esetben nem vágja a saját haját, aka lehetne a saját ügyfele, viszont ha most levágja akkor kezdődik az egész előlről [Russel's paradox](#).

Ezzel a problémával sok helyen találkozhatunk. Például [\[DENOTATIONALSEMANTICS\]](#) 4.1.1. fejezet, vagy akár [\[SICP\]](#) Exercise 4.15.

```
1 Program T100
2 {
3
4   boolean Lefagy(Program P)
5   {
6     if(P has infinite loop)
7       return true;
8     else
9       return false;
10  }
11
12  main(Input Q)
13  {
14    Lefagy(Q)
15  }
16 }
```

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 építő `Lefagy2` már nem tartalmaz feltételezett, csak csak konkrét kódot:

```
1 Program T1000
2 {
3
4   boolean Lefagy(Program P)
5   {
6     if(P has infinite loop)
7       return true;
8     else
9       return false;
10  }
```

```

11
12 boolean Lefagy2(Program P)
13 {
14     if(Lefagy(P))
15         return true;
16     else
17         for(;;);
18 }
19
20 main(Input Q)
21 {
22     Lefagy2(Q)
23 }
24
25 }

```

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.

Persze ha már lefagyott, nem fagyott témánál vagyunk, akkor érdemes kicsit beszélni a kiértékelésről. Alább látható egy egyszerű kifejezés. Mi lesz vajon a vége?

$$(\lambda x.y)((\lambda z.zz)(\lambda z.zz)) \quad (2.1)$$

Redukáljuk!

$$(\lambda x.y)((\lambda z.zz)(\lambda z.zz)) \quad (2.2)$$

Humph, ugyanoda jutottunk, kezdjük előlről!

$$(\lambda x.y)((\lambda z.zz)(\lambda z.zz)) \quad (2.3)$$

Redukáljuk az x-t kötő baloldali lambdát a jobb oldallal a jobb oldal redukciója nélkül.

$$y \quad (2.4)$$

Azaz addig késleltettük a kiértékelést, amíg lehetett, és a végén kiderült, hogy abszolút nem is volt szükséges! Ez a módszer zseniális! Innentől mindent késleltetni fogunk!

Nézzünk egy példát a [SICP]-ből! Alábbi definíciók alapján ki fogjuk értékelni a (f 5) -t!

```

(define (square x) (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

```

Annyira jó ez a késleltetés, hogy alkalmazzuk a $(f\ 5)$ kiértékelésénél!

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)) )
(+ (* (+ 5 1) (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 (* 10 10))
(+ 36 100)
136
```

Mint ahogy azt látjuk $(+ 5 1)$ és $(* 5 2)$ kétszer kerül redukcióra. Hát így már nem is olyan jó...

Hogy fut le vajon egy másik út?

```
(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
(sum-of-squares 6 (* 5 2))
(sum-of-squares 6 10)
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 (* 10 10))
(+ 36 100)
136
```

Kevesebb számítást végzünk mint a késleltetős előző esetben. Azaz látjuk, hogy a redukciós stratégiának a teljesítményre is van hatása.

2.3. Változók értékének felcserélése

Írjunk egy olyan programot, amely felcseréli két változó értékét, bármiféle logikai utasítás vagy kifejezés használata nélkül, és ez ne bug hanem feature legyen!

Megoldás videó: No Link

Megoldás forrása:

- [Swap XOR](#)
- [Swap Subtract](#)
- [Swap Multiplication](#)
- [Swap Ptr](#)

Több megoldási lehetőség is van, ezeket fogjuk a következő bekezdésekben bemutatni.

Nézzünk valami szorzásos osztásosát!

```
1 #include <stdio.h>
2
3 int main(){
4     int a = 6;
5     int b = 7;
6     printf("Let a = %d, b = %d\n",a,b);
7     a = a * b;
8     printf("(1) a = a * b = %d\n",a);
9     b = a / b;
10    printf("(2) b = a / b = %d\n",b);
11    a = a / b;
12    printf("(3) a = a / b = %d\n",a);
13    printf("Result after swap by multiplication and division is a = %d, b = ↵
        %d\n",a,b);
14    return 0;
15 }
```

Kövessük nyomon mi történik!

(a, 6) (b, 7)
a = a * b => (a, 42) (b, 7)
b = a \ b => (a, 42) (b, 7)
a = a \ b => (a, 6) (b, 7)

Hát ez működik, de 0-val nem osztunk, plusz a szorzás osztás nehéz. Nézzünk mást!

Nézzünk valami összeadás kivonásosát!

```
1 #include <stdio.h>
2
3 int main(){
4     int a = 6;
5     int b = 7;
6     printf("Let a = %d, b = %d\n",a,b);
7     a = a + b;
8     printf("(1) a = a + b = %d\n",a);
9     b = a - b;
10    printf("(2) b = a - b = %d\n",b);
11    a = a - b;
12    printf("(3) a = a - b = %d\n",a);
13    printf("Result after swap by addition and subtract is a = %d, b = %d\n" ↵
        ,a,b);
14    return 0;
15 }
```

Kövessük nyomon mi történik!

(a, 6) (b, 7)

```
a = a + b => (a, 13) (b, 7)
b = a - b => (a, 13) (b, 6)
a = a - b => (a, 7) (b, 6)
```

Ez is jó, és ráadásul csak összeadással és kivonással terheljük a CPU-t!

De esetleg van valami más is? Nos nézzük a xor műveletet!

```
1 #include <stdio.h>
2
3 int main(){
4     int a = 6;
5     int b = 7;
6     printf("Let a = %d, b = %d\n", a, b);
7     a = a ^ b;
8     printf("(1) a = a ^ b = %d\n", a);
9     b = a ^ b;
10    printf("(2) b = a ^ b = %d\n", b);
11    a = a ^ b;
12    printf("(3) a = a ^ b = %d\n", a);
13    printf("Result after swap by xor is a = %d, b = %d\n", a, b);
14    return 0;
15 }
```

Kövessük végig!

```
(a, 110) (b, 111)
a = a XOR b => (a, 001) (b, 111)
b = a XOR b => (a, 001) (b, 110)
a = a XOR b => (a, 111) (b, 110)
```

Egy nagyon furcsa dolgot láthatunk. Az összes előző esetenél több operátort kellett használni (*) és (/) illetve (+) és (-). Most azonban egy olyan esetet látunk, amikor egyetlen operátorral megtudtuk oldani (kommutatív, asszociatív, identitás elem létezik, $f(a,a)=\text{Identitás elem}$).

Alábbi utolsó példa csak egy picit behozza a pointer-eket a képbe. Ezekkel ha még nem értjük mi van, ne aggódjunk. Minden pointert érdemes egy címként felfogni. Azaz ha létezik egy int változónk foo néven akkor foo-nak van ugye egy értéke, illetve egy címe. A cím az amit a ptr megtestesít (na jó nem, de most így elég...). Természetesen funckiókra is mutathatunk. Sőt, ha már valaha láttunk C kódot, akkor valószínűleg belefutottunk már callback-ekbe stb. Aka akár funckiókat és bepasszolhatunk funckiókba, stb.

```
1 #include <stdio.h>
2
3 void swap1(int *pa, int* pb ){
4     int t = *pa;
5     *pa = *pb;
6     *pb = t;
7 }
8
9 void swap2(int *pa, int* pb ){
10    *pa = *pa + *pb;
```

```
11     *pb = *pa - *pb;
12     *pa = *pa-*pb;
13 }
14
15 int main()
16 {
17     int a = 6;
18     int b = 7;
19     printf("Let a = %d, b = %d\n", a, b);
20     swap1(&a, &b);
21     printf("Swap1 %d, %d\n", a, b);
22     swap2(&a, &b);
23     printf("Swap2 %d, %d\n", a, b);
24     return 0;
25 }
```

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés használata nélkül írd egy olyan programot, ami egy labdát pattogtat a karakteres konzolon! Nem írhatasz hozzá DLC-t, és ne legyen benne lootbox, mert nem ez EA vagyunk.

Megoldás videó:

Megoldás forrása:

- [Helper header](#)
- [Ball](#)
- [Ball no if](#)

Ennek a feladatnak az volt a lényege, hogy elágazás nélkül kicsikarjunk a gépből valami féle döntés alapú útválasztást. Először nézzük az alapot.

```
1  #include "pubghypetrain.h"
2
3  void collide(int posX, int posY, int &velX, int &velY, int xmax, int ymax){
4      bool outX = (posX <= 0 || posX >= xmax-1);
5      velX += -2 * velX * outX;
6      bool outY = (posY <= 0 || posY >= ymax-1);
7      velY += -2 * velY * outY;
8  };
9
10 void moveBall(int &posX, int &posY, int velX, int velY){
11     posX += velX;
12     posY += velY;
13 };
14
```

```

15 void draw(int posX, int posY, int xmax, int ymax){
16     ClearScreen();
17     for(int y = 0; y < ymax; y++){
18         for(int x = 0; x < xmax; x++){
19             if( (posX == x) && (posY == y) ){ std::cout << 'O';}else{ std::cout << ' ';}
20         };
21         std::cout << std::endl;
22     };
23 };
24
25 int main(){
26     int T_SLEEP_MILLIS = 50;
27     int GLOBAL_H = 10;
28     int GLOBAL_W = 10;
29     int posX = 5;
30     int posY = 2;
31     int velX = 2;
32     int vely = -1;
33     for(int i = 0; i < 50; i++){
34         moveBall(posX,posY,velX,vely);
35         collide(posX,posY,velX,vely,GLOBAL_H,GLOBAL_W);
36         draw(posX,posY,GLOBAL_H,GLOBAL_W);
37         sleep(T_SLEEP_MILLIS);
38     };
39     return 0;
40 };

```

Nyilvánvalóan valahogyan trükközni kell, hisz az `if`-et ha kihagynánk, akkor nem tudnánk például dönteni, hogy mely karaktert küldjük `std::out`-ra. Nézzük egy lehetséges megoldást.

```
1 #include "pubghypetrain.h"
2 #include <iostream>
3
4 void collide(int posX, int posY, int &velX, int &velY, int xmax, int ymax);
5
6 void moveBall(int &posX, int &posY, int velX, int velY);
7
8 void draw(int posX, int posY, int xmax, int ymax, const char* syms);
9
10 int main()
11 {
12     const char SYMBOLS[2] = {'.','O'};
13     int T_SLEEP_MILLIS = 50;
14     int GLOBAL_H = 10;
15     int GLOBAL_W = 10;
16     int posX = 5;
17     int posY = 2;
18     int velX = 2;
19     int velY = -1;
20     for(int i = 0; i < 50; i++){
```

```

21     moveBall(posX, posY, velX, velY);
22     collide(posX, posY, velX, velY, GLOBAL_H, GLOBAL_W);
23     draw(posX, posY, GLOBAL_H, GLOBAL_W, SYMBOLS);
24     sleep(T_SLEEP_MILLIS);
25 };
26     return 0;
27 };
28
29
30 void moveBall(int &posX, int &posY, int velX, int velY){
31     posX += velX;
32     posY += velY;
33 };
34
35 void collide(int posX, int posY, int &velX, int &velY, int xmax, int ymax){
36     velX += -2 * velX * (posX <= 0 || posX >= xmax-1);
37     velY += -2 * velY * (posY <= 0 || posY >= ymax-1);
38 };
39
40 void draw(int posX, int posY, int xmax, int ymax, const char* syms){
41     ClearScreen();
42     for(int y = 0; y < ymax; y++){
43         for(int x = 0; x < xmax; x++){
44             std::cout << syms[(posX == x) && (posY == y)];
45         };
46         std::cout << std::endl;
47     };
48 };

```

Simán csak implicit konverzió történik, mind `collide`, mind `draw` esetén.

Ha már itt tartunk, akkor nézzünk más jellegű `if` encode-olást is! (használhatjuk például a [\[CHISOMORPH\]](#)-t)

```

true = λx.λy.x
false = λx.λy.y
if(B,P,Q) = B P Q

```

Az `if true then P else Q` redukciója

```

if true then P else Q
= true P Q
= λx.λy.x P Q
= λy.P Q
= P

```

Az `if false then P else Q` redukciója

```

if false then P else Q
= false P Q
= λx.λy.y P Q

```



```
= λy.y Q
= Q
```

Vezessünk be számokat!

```
if false then P else Q
0 = λf.λx.x
1 = λf.λx.f x
2 = λf.λx.f (f x)
...
```

Illetve egy isZero predikátumot!

```
IsZero = λn.n (λx.false) true
```

Nézzük meg pár esetre a predikátumunk működését!

```
IsZero 0
= (λn.n (λx.false) true) (λf.λx.x)
= ( (λf.λx.x) (λx.false) true)
= ( λx.x true)
= true
```

```
IsZero 1
= (λn.n (λx.false) true) (λf.λx.f x)
= ( (λf.λx.f x) (λx.false) true)
= ( (λx.(λx.false) x) true)
= (λx.false) true
= false
```

Mostmár mondhatunk olyat, hogy osztásnál ne legyen nulla a nevezőben, azaz `if IsZero b then 0 else (div a b)`. A lényeg az, hogy dobjon vissza egy `(div valami valami)-t` ha el lehet végezni, különben pedig egy 0-t. Például 4-el és 0-val hívjuk.

```
( λx.λy.(IsZero y) 0 (div x y)) 4 0
= ( λy.(IsZero y) 0 (div 4 y)) 0
= ((IsZero 0) 0 (div 4 0))
= (true 0 (div 4 0))
= ((λx.λy.x) 0 (div 4 0))
= ((λy.0) (div 4 0))
= 0
```

Most nézzük meg 4 és 2-vel.

```
( λx.λy.(IsZero y) 0 (div x y)) 4 2
( λy.IsZero y) 0 (div 4 y)) 2
((IsZero 2) 0 (div 4 2))
(false 2 (div 4 2))
((λx.λy.y) 0 (div 4 2))
((λy.y) (div 4 2))
(div 4 2)
```

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:

- [Wordsize](#)
- [Bogomips](#)

Először is nézzük a biteket! A számokat kettes számrendszerben tároljuk. A shift az értékek "tologatása". De pontosan mi az a shift?

Amikor left shiftelünk, akkor mi is történik? Nézzük unsigned char-ra! „For unsigned lhs, the value of LHS << RHS is the value of LHS * 2^{RHS}, reduced modulo maximum value of the return type plus 1 (that is, bitwise left shift is performed and the bits that get shifted out of the destination type are discarded).” Alább látható hogy miről van szó.

1	<< 1 =>	(1*2 ¹) % (255+1)	= 2	(00000010)
2	<< 1 =>	(2*2 ¹) % (255+1)	= 4	(00000100)
4	<< 1 =>	(4*2 ¹) % (255+1)	= 8	(00001000)
8	<< 1 =>	(8*2 ¹) % (255+1)	= 16	(00010000)
16	<< 1 =>	(16*2 ¹) % (255+1)	= 32	(00100000)
32	<< 1 =>	(32*2 ¹) % (255+1)	= 64	(01000000)
64	<< 1 =>	(64*2 ¹) % (255+1)	= 128	(10000000)
128	<< 1 =>	(128*2 ¹) % (255+1)	= 0	(00000000)

Alábbi kóddal nézzük meg:

```
1 #include <iostream>
2
3 int main()
4 {
5     unsigned char a = 1;
6     for(int i = 0; i<8; i++)
7     {
8         a=a<<1;
9         std::cout<< ((int)a) << std::endl;
10    }
11    return 0;
12 }
```

```
$ ./shiftr_uchar.exe
127
63
31
15
7
3
1
0
```

2.5. ábra. shiftr_uchar

Mi történik a right shift esetben? „For unsigned lhs and for signed lhs with nonnegative values, the value of $LHS \gg RHS$ is the integer part of $LHS / 2^{RHS}$.”

```
255 >> 1 => intp(255 div 2^1) = 127 (01111111)
127 >> 1 => intp(127 div 2^1) = 63  (00111111)
63  >> 1 => intp(63 div 2^1)  = 31  (00011111)
31  >> 1 => intp(31 div 2^1)  = 15  (00001111)
15  >> 1 => intp(15 div 2^1)  = 7   (00000111)
7   >> 1 => intp(7 div 2^1)   = 3   (00000011)
3   >> 1 => intp(3 div 2^1)   = 1   (00000001)
1   >> 1 => intp(1 div 2^1)   = 0   (00000000)
```

Alábbi kóddal nézzük meg:

```
1 #include <iostream>
2
3 int main()
4 {
5     unsigned char a = 255;
6     for(int i = 0; i<8; i++)
7     {
8         a=a>>1;
9         std::cout<< ((int)a) << std::endl;
10    }
11    return 0;
12 }
```

```
$ ./shiftr_uchar.exe
127
63
31
15
7
3
1
0
```

2.6. ábra. shiftr_uchar

Alábbi kód az előzőeket hajtja végre csak int-et használunk.

```
1  #include <iostream>
2
3  void shift_l()
4  {
5      unsigned int a = 1;
6      unsigned int i = 0;
7      while(a!=0) {
8          a = a<<1;
9          i++;
10     }
11     std::cout<<"Shift_L count was " << i << std::endl;
12     std::cout<<"Size " << ((i)/8) << std::endl;
13     std::cout<<"Sanity Check " << sizeof(int) << std::endl;
14 }
15
16 void shift_r()
17 {
18     unsigned int a = 0;
19     a = ~a;
20     int i = 0;
21     while(a!=0) {
22         a = a>>1;
23         i++;
24     }
25     std::cout<<"Shift_R count was " << i << std::endl;
26     std::cout<<"Size " << ((i)/8) << std::endl;
27     std::cout<<"Sanity Check " << sizeof(int) << std::endl;
28 }
29
30 int main()
31 {
32     shift_l();
33     shift_r();
34     return 0;
35 }
```

Előjeles esetben figyelembe kell venni, hogy például a felső bit előjelet is jelenthet akár. Alábbi ábra szép illusztrációt ad arra, mire is kell gondolni (2s complement).

Positive numbers		Negative numbers	
0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

2.7. ábra. 2s complement

Most áttérünk a bogomips-re. Itt egy új dologgal találkozunk `<<=` de ez igazából csak annyit tesz, hogy a shift rhs-t egyből az lhs oldalon lévő változóba vissza is tároljuk. A delay egy for loop ami elszámol loops-ig, azaz addig növeli (`++`) unáris operátorral `i` értékét amíg `a i < loops` conditional igaz. A while ciklus fejben pedig már a jól ismert left shift zajlik (`loops_per_sec` 1-ről indul). De hogy biztosak legyünk nézzük az alábbi programot (bár nagyon ismerős lesz...)

```

1  #include <iostream>
2
3  int main()
4  {
5      unsigned long long a = 1;
6      int i = 0;
7      while(a!=0)
8      {
9          a<<=1;
10         std::cout<< a << std::endl;
11         i++;
12     }
13     std::cout<< i << std::endl;
14     return 0;
15 }
```

```

1152921504606846976
2305843009213693952
4611686018427387904
9223372036854775808
0
64
```

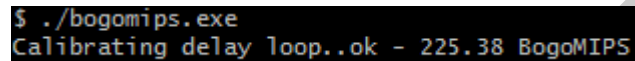
2.8. ábra. ullshift

A bogomips while body egyébként annyiról szól, hogy lekérjük hány tick telt el idáig, majd nyomunk

n mennyiség lépést a for ciklusban, visszajövünk és megint lekérdezzük az eltelt tick-ek számát. Ha a tickek száma nem kevesebb mint amennyit a CLOCKS_PER_SEC konstanssal állít magáról a gép, akkor elkezdjük a kiszállást. A loops_per_sec pedig a következő módon jön ki:

```
loop_per_masodperc = ( loopok_szama / tick_szam ) * CLOCKS_PER_SEC;  
[1/s] = ( [1] / [1] ) * [1/s]
```

Ezután következik zsonglőrködés a számokkal. Ez a rész a bogus a bogomips-ben. Ha sok időnk van akkor meg is mérhetjük a saját gépünkét. Alább egy random példa.



```
$ ./bogomips.exe  
Calibrating delay loop..ok - 225.38 BogoMIPS
```

2.9. ábra. bogomips kimeneten

2.6. Helló, Google!

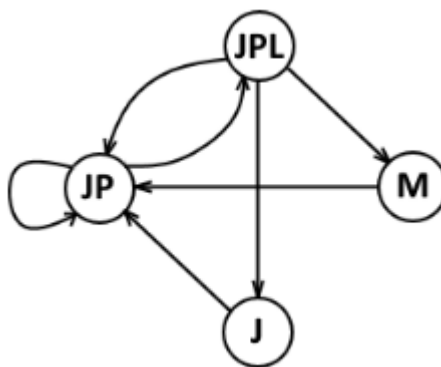
Írj olyan C programot, amely egy több milliárd(haha ne ne...elég lesz kevesebb srsly) honlapból álló hálózatra kiszámolja az N lap Page-Rank értékét, és [nem lesz belőle baj egy szenátusi kihallgatáson!](#)

Megoldás videó:

Megoldás forrása:

- [Pagerank Példa](#)
- [Pagerank C](#)
- [STL Iterator](#)
- [Purely STL Vector](#)

Pagerank a Google által használt adóelk...keresési algoritmus magja. A feladat arról szólt, hogy egy négy honlapból álló kapcsolati hálón kellene vele elemezni. A pageranktól azt várjuk, hogy az összes honlapnak mondjuk adjon egy 0-1-ig terjedő értéket (normalizált, azaz az összes lap rankjének összege 1 for sanity check). Van több implementáció is, de mi egy konkrétan kitenyészített példa progival fogunk dolgozni. Ez a pagerank vektor értékeit fő iterációs lépésenként kiírja egy fájlba. A pagerankkel a példában egy irányított multigráfon fogunk számolni. (irányított mert számít hogy honnan-hova megy az él, multigráf pedig a több él egy pontból plusz hurkok is lehetnek)



2.10. ábra. pagerank kapcsolati irányított multigráf

Ez ugyan emberként érthető, de próbáljuk valahogy szervezettebb formába hozni. Csináljunk egy kvadratus mátrixot, és töltsük ki a következő szabály szerint: Először mindenhova írjunk be nullát majd Ha A-ból B-be megy él, akkor A oszlop B sorába írjunk 1-et Először nézzük csak meg J-re(J-ből egy él megy ki JP felé...)

	J	JP	JPL	M
J	0	0	0	0
JP	1	0	0	0
JPL	0	0	0	0
M	0	0	0	0

Most pedig csináljuk meg JP-re(magába és JPLbe)

	J	JP	JPL	M
J	0	0	0	0
JP	1	1	0	0
JPL	0	1	0	0
M	0	0	0	0

Jöhet JPL (mindenkibe csak magába nem)

	J	JP	JPL	M
J	0	0	1	0
JP	1	1	0	0
JPL	0	1	1	0
M	0	0	1	0

Jöhet M (csak JP-be)

	J	JP	JPL	M
J	0	0	1	0
JP	1	1	0	1
JPL	0	1	1	0
M	0	0	1	0

Most normalizáljuk, azaz adjuk össze az egy oszlopban lévő számokat, nevezzük ezt *szummának*. Ha meg van, akkor utána ugyanezen az oszlopon menjünk végig és mindenkit osszunk el *szum*-mal, ha *szum* nem nulla.

	J	JP	JPL	M
J	0	0	1/3	0
JP	1	1/2	1/3	1
JPL	0	1/2	0	0
M	0	0	1/3	0

Vessük össze az előadás diával.

	J	JP	JPL	M
J	0	0	1/3	0
JP	1	1/2	1/3	1
JPL	0	1/2	0	0
M	0	0	1/3	0

2.11. ábra. pagerank link mátrix sanity check

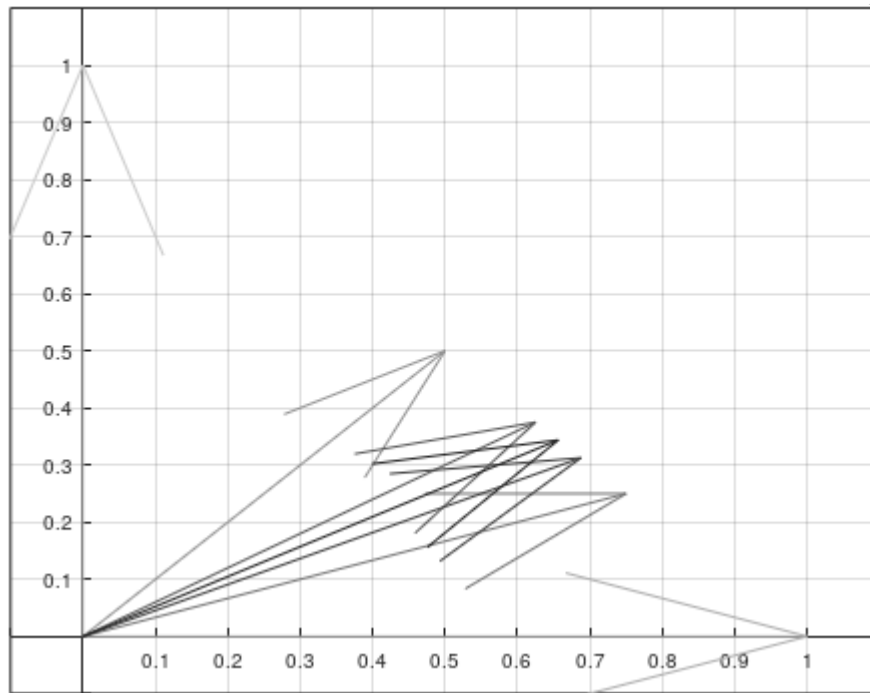
Most hogy végre van link mátrixunk elkezdhetünk számolni. De mi is a lényeg, mit fogunk számolni? Gondoljunk vissza a [sajátérték sajátvektorra](#), és [próbálgassuk](#) is ha tetszik! Magyarul amikor Tanár Úr azt mondja Tehát ha h jelöli a PR vektort, akkor $h = Lh$. Linalg kedvelőknek: a PageRank vektor az L linkmátrix 1 sajátértékhez tartozó sajátvektora. akkor gondoljunk a következőre

$$\begin{array}{c|cccc|c|c|c|c}
 & \text{PR}[1] & & & & & & & \\
 & \text{PR}[2] & & & & & & & \\
 & \text{PR}[3] & & & & & & & \\
 & \text{PR}[4] & & & & & & & \\
 \hline
 \begin{array}{c|cccc}
 0 & 0 & 1/3 & 0 \\
 1 & 1/2 & 1/3 & 1 \\
 0 & 1/2 & 0 & 0 \\
 0 & 0 & 1/3 & 0
 \end{array}
 & \begin{array}{c|c}
 ? & \\
 ? & \\
 ? & \\
 ? &
 \end{array}
 & = & \begin{array}{c|c}
 \text{PR}[1] & \\
 \text{PR}[2] & \\
 \text{PR}[3] & \\
 \text{PR}[4] &
 \end{array}
 \end{array}$$

De milyen vektorról is beszélünk? Mi a végcél ezzel az egész saját vektor dologgal? Vizualizáljuk!

Jelen példában 4 node van szóval PR 4 elemű, ezt nehéz lenne ábrázolnom. Viszont gondoljunk egy két node-ból álló esetre! Ezen esetben PR vektor 2 elemű lesz, szóval már plottolhatjuk 2dbe. Alábbi ábrán simán fogtuk magunkat és a PR vektor 1. számát x -nek 2. számát y -nak vettük csináltunk bele egy nyilat

az origóból. Színkódoltuk is: A legelső iter utáni PR vektor halvány szürke, az utolsó pedig fekete, közte pedig graduálisan változtattuk a feketeséget (hsv mert lusta voltam, de ne menjünk bele).

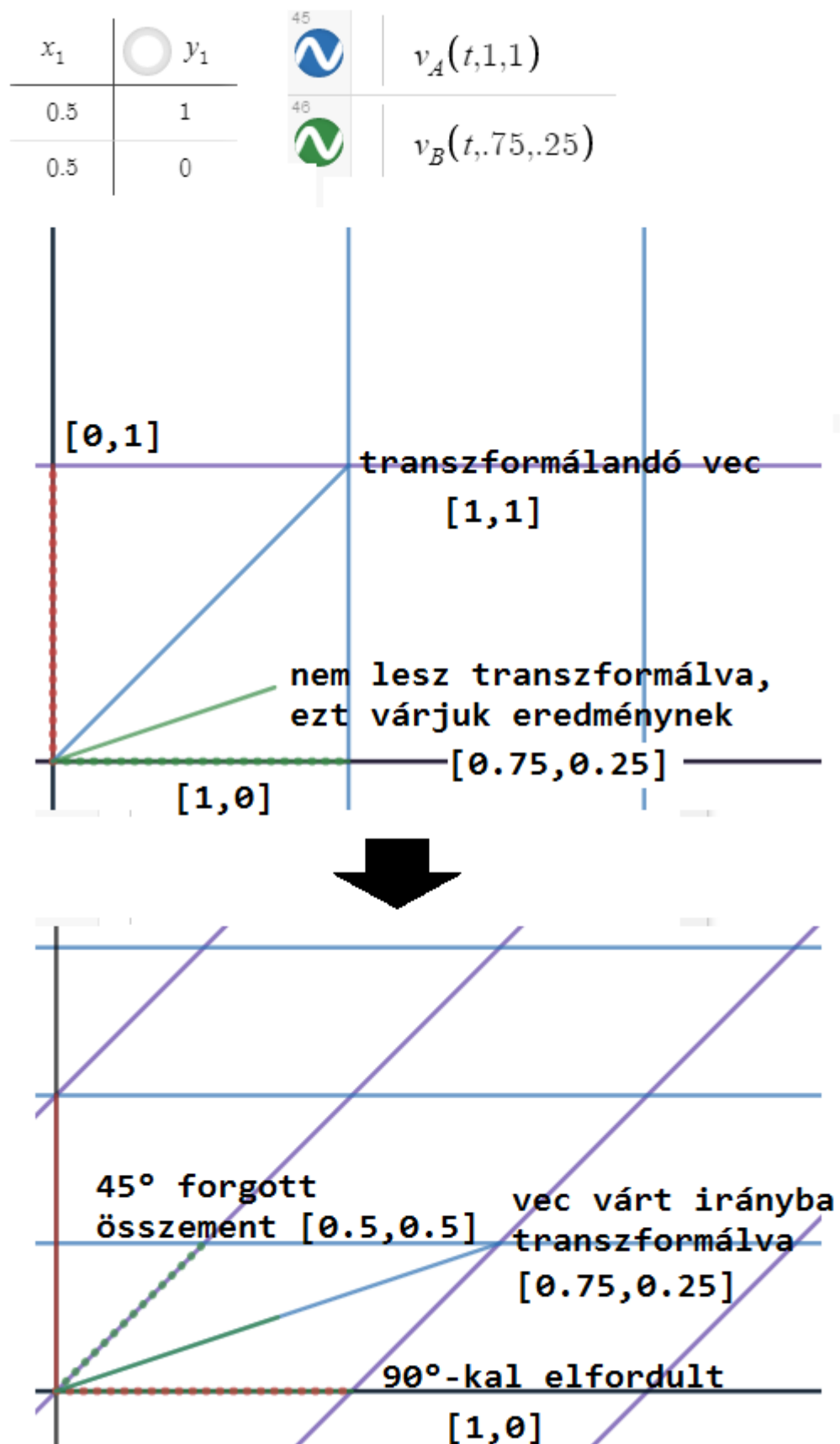
$$\begin{aligned}x &= [0 \quad 1 \quad 0.5 \quad 0.75 \quad 0.625 \quad 0.6875 \quad 0.65625 \quad] \\y &= [1 \quad 0 \quad 0.5 \quad 0.25 \quad 0.375 \quad 0.3125 \quad 0.34375 \quad]\end{aligned}$$


2.12. ábra. pagerank vektor

Fenti ábrán látható, hogy ahogy beindul az iteráció vadul $[0,1]$ -ből $[1,0]$ -ba vált, majd $[0.5,0.5]$ és szépen lassan kezd beállni $[0.66 \ 0.33]$ -ba!

Magyarul a nem pontos PR vektor közelítésünk egyre kisebb cikázással kezd beállni abba az irányba ami egyébként az "igazi" sajátvektor irány. Ez alapján világos, hogy az egész számításnak az a célja hogy megtaláljuk azt a vektort, amire ha alkalmazzuk a Link mátrixot transzformációként, akkor iránya már nem fog változni. (Ha valakit érdekel akkor a 2d-s eset az oppenoffice [fájl](#)-ban a twod munkalapon van, de semmi különös.)

Ha valaki mégsem értené, nézzünk egy konkrét példát: Hogyan lett pl $[0.5,0.5]$ irányú vektorból $[0.75,0.25]$? Alábbi ábrán felveszünk egy $[0.75,0.25]$ vektort(zöld folytonos vonal), és ezt változatlanul fogjuk hagyni. Ezután veszünk egy $[1,1]$ vektort(kék folytonos vonal). Azért $[1,1]$ mert csak az irány a fontos, és azt akartam hogy nyúljon túl a zöldön trafó után. Ezekután transzformáljuk a linkmátrix-szal a kéket. Azt várjuk hogy a transzformáció után a zöld folytonos és kék folytonos egyirányú legyen.



2.13. ábra. pagerank trf

Vissza matekra! Mátrixok vektorok stb. szorzást már tanultunk szóval, oldjuk is meg!

$$\begin{array}{cccc|cccc}
 & & & & & & \text{PR}[1] & \\
 & & & & & & \text{PR}[2] & \\
 & & & & & & \text{PR}[3] & \\
 & & & & & & \text{PR}[4] & \\
 \hline
 0 & 0 & 1/3 & 0 & & & \text{PR}[3]/3 & \\
 1 & 1/2 & 1/3 & 1 & & & \text{PR}[1] + \text{PR}[2]/2 + \text{PR}[3]/3 + \text{PR}[4] & \\
 0 & 1/2 & 0 & 0 & & & \text{PR}[2]/2 & \\
 0 & 0 & 1/3 & 0 & & & \text{PR}[3]/3 &
 \end{array} = \begin{array}{cccc}
 \text{PR}[1] \\
 \text{PR}[2] \\
 \text{PR}[3] \\
 \text{PR}[4]
 \end{array}$$

Szét is robbanthatjuk akár egyenletekre...

$$\begin{array}{lcl}
 \text{PR}[3]/3 & = & \text{PR}[1] \\
 \text{PR}[1] + \text{PR}[2]/2 + \text{PR}[4] & = & \text{PR}[2] \\
 \text{PR}[2]/2 + \text{PR}[3]/3 & = & \text{PR}[3] \\
 \text{PR}[3]/3 & = & \text{PR}[4]
 \end{array}$$

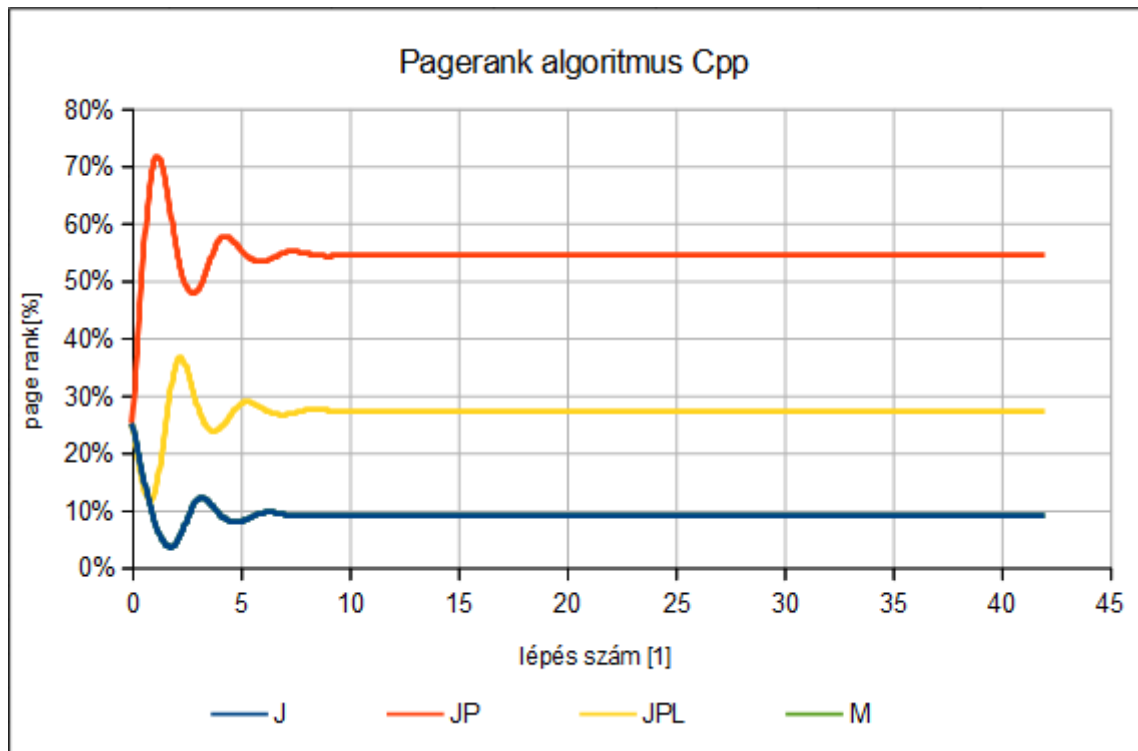
Most pedig jön a kérdés, hogy hogyan oldjuk meg? PR[2] önmagából számolja önmagát!

A válasz nem túlzottan bonyolult...tároljuk el az előző lépés értékeit minden ciklusban és használjuk azokat. (Ez lesz a megoldásban PRv)

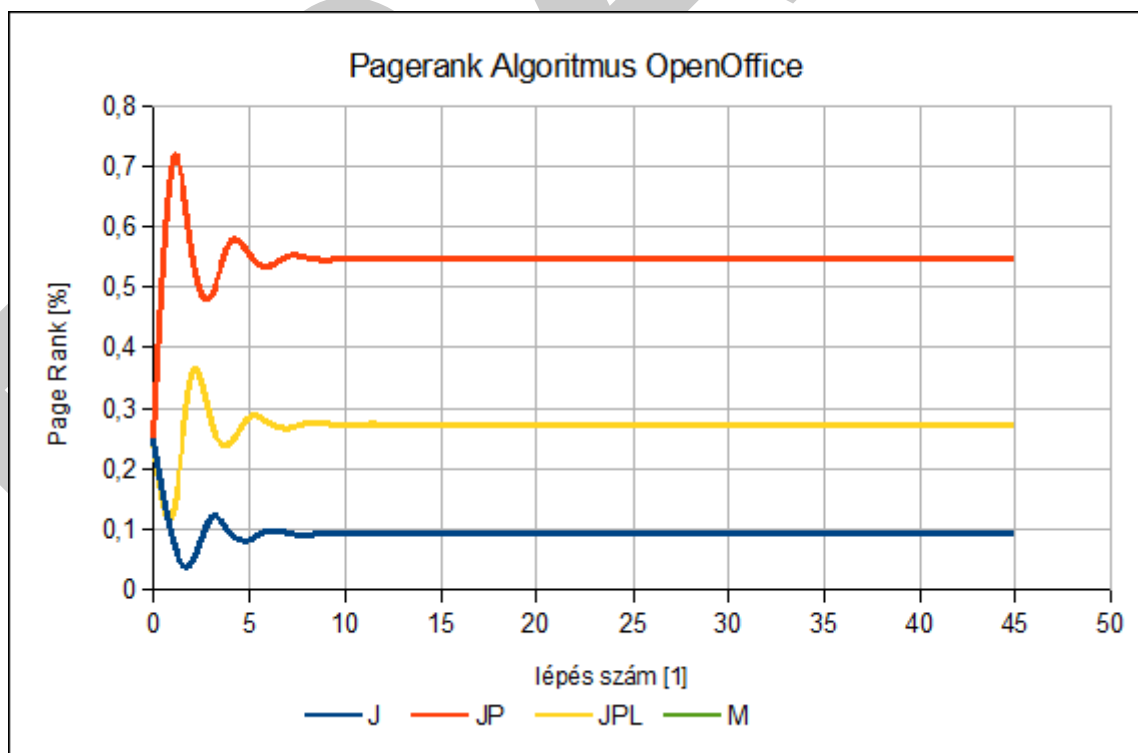
$$\begin{array}{lcl}
 \text{PRv}[3]/3 & = & \text{PR}[1] \\
 \text{PRv}[1] + \text{PR}[2]/2 + \text{PR}[4] & = & \text{PR}[2] \\
 \text{PRv}[2]/2 + \text{PR}[3]/3 & = & \text{PR}[3] \\
 \text{PRv}[3]/3 & = & \text{PR}[4]
 \end{array}$$

Egy kérdés még maradt: Mennyi legyen PRv iniciális értéke? A válasz annyi, hogy mindegyik érték 1 aztán benormáljuk, hogy összegük 1 legyen (Mat viz).

Ha midezt megértettük, akkor már könnyű lesz megérteni az alábbi kódomat! A refekkel, constokkal, inicializer listekkel meg a többi szeméttel ne törődjünk még nem kell őket érteni, csak a pagerank-re fókuszáljunk. Ha esetleg nem lenne érthető a kód, akkor beraktam a repóba egy openoffice calc (olyan mint az excel csak kevesebb benne a fluff) doksit, ami ugyanezt a pagerank számítást csinálja... A fejezet végén beillesztettem a C++ kódot, viszont most következzen két diagram. Az első a C++ algoritmus PR vektorának értékeit mutatja iterációnként, a másik pedig az [open office](#)-ost, ha valaki valami miatt szereti az irodai munkát.



2.14. ábra. pagerank cpp konvergál



2.15. ábra. pagerank openoffice konvergál

Kicsit azért vigyázzunk...ugyanis van két kis probléma. Tanár Úr azért mondta hogy rázzuk a vizet az internetben, mert arra gondolt, hogy ez az iteratív megoldás olyan mintha vödörök között öntögetnénk vizet, vagy mondjuk egy fémlemez hőmérsékletét akarnánk úgy kiszámolni, hogy az egyik szélén tudjuk a hőmérsékletet a többit pedig úgy számoljuk, hogy a négy (felső-alsó-bal-jobb egy kockás lapon mondjuk) szomszédját átlagoljuk. Nos első esetben az a probléma, hogy az egyik vödörbe folyik víz be, vissza is folyik magába, de kifele nem. A példa fájlban egy `get_dangling` funkció csinál olyan mátrix-ot amiben van egy nyelő jellegű node.

```
{ 1, 0, 1, 0, 1, 0 },
{ 1, 1, 0, 1, 0, 0 },
{ 0, 1, 1, 0, 1, 0 },
{ 1, 0, 1, 1, 0, 0 },
{ 0, 1, 0, 1, 1, 0 },
{ 1, 0, 1, 0, 1, 0 }
```

Most pedig következzen az igazi vizes locspocs. Vegyük a következő kapcsolati gráfot:

	A	A
A	0	0
B	1	1

A locspocs-t indítsuk a következő PRv-vel:

```
PRv  0
PRv  1
```

Érezhető hogy ennek mi lesz a vége, ha valaki elképesztően kíváncsi az open office fájlban-ban látható egy Infinite nevű munkalap.

```
1  #include <iostream>
2  #include <vector>
3  #include <cmath>
4  #include <fstream>
5  #include <iomanip>
6
7
8  void kiir(std::ofstream& os, const std::vector<double> &v) {
9      int sz = v.size();
10     if(sz<1) return;
11     os << v[0];
12     for(int i=1; i<sz; i++)
13         os << ";" << v[i];
14     os << std::endl;
15 }
16
17 double tavolsag(const std::vector<double> &PR, const std::vector<double> & ←
18     PRv) {
19     int i;
20     double osszeg = 0;
21     for(i=0; i<PR.size(); ++i) osszeg += (PRv[i]-PR[i])*(PRv[i]-PR[i]);
22     return sqrt(osszeg);
```

```
22 }
23
24 std::vector<std::vector<double>> get_batfai_graph()
25 {
26     return std::move(std::vector<std::vector<double>>{
27         /*J*/ { 0, 0, 1, 0},
28         /*JP*/ { 1, 1, 1, 1},
29         /*JPL*/ { 0, 1, 0, 0},
30         /*M*/ { 0, 0, 1, 0}
31     });
32 }
33
34 std::vector<std::vector<double>> get_dangling()
35 {
36     return std::move(std::vector<std::vector<double>>{
37         { 1,0,1,0,1,0},
38         { 1,1,0,1,0,0},
39         { 0,1,1,0,1,0},
40         { 1,0,1,1,0,0},
41         { 0,1,0,1,1,0},
42         { 1,0,1,0,1,1}
43     });
44 }
45
46 std::vector<std::vector<double>> get_symm()
47 {
48     return std::move(std::vector<std::vector<double>>{
49         { 0,1 },
50         { 1,0 }
51     });
52 }
53
54 void normalize(std::vector<std::vector<double>>& quadmx)
55 {
56     int size = quadmx.size();
57     for(int j=0; j<size; j++){
58         double sum = 0;
59         for(int i=0; i<size; i++){
60             sum+=quadmx[i][j];
61         }
62         if(sum!=0){
63             for(int i=0; i<size; i++){
64                 quadmx[i][j]/=sum;
65             }
66         }
67     }
68 }
69
70 int main(void){
71     // ha probalgatod es ne adj isten valami miatt ne konvergalna
```

```
72 // akkor ennyi iter után auto kilep a fo loopbol
73 int SAFE_LIMIT = 100;
74 std::ofstream outf;
75 outf.open ("pagerank.csv");
76 // at this point L is not normalized
77 std::vector<std::vector<double>>L{std::move(get_dangling())};
78 normalize(L);
79 int size = L.size();
80 std::vector<double> PR;
81 for(int i = 0; i<size; i++)PR.push_back(0.0);
82 std::vector<double> PRv;
83 for(int i = 0; i<size; i++)PRv.push_back(1.0/size);
84 for(int c;c<SAFE_LIMIT;c++){
85     for(int i = 0; i < size; ++i){
86         PR[i] = 0.0;
87         for(int j = 0; j < size; ++j) PR[i] += L[i][j] * PRv[j];
88     }
89     kiir(outf,PR);
90     if(tavolsag(PR,PRv)<0.0000000001) break;
91     PRv = PR;
92 }
93 outf.close();
94 return 0;
95 }
```

2.7. A Monty Hall probléma

Írj szimulációt a Monty Hall problémára!

Megoldás videó: [No link](#)

Megoldás forrása: [No link](#)

Gondolkodjunk úgy hogy két stratégia van: "nem választunk újra" és "újra választunk". Alábbi kódban "nem választunk újra" esetben N lehetőség közül $1/N$ valószínűségünk van nyerni. Viszont ha tudjuk hogy "nem választunk újra" veszített, és a fazon már kinyitott egy ajtót, akkor $N-2$ -ből kell csak választanunk. Utolsó sorban kiírja az nbatfai és ezen kód is sanity check jelleggel a két startégia nyertes kimeneteleinek összegét, ami $N=3$ esetben egyenlő N -nel. Magasabb N -nél ez nem így van, hisz ha "nem választunk újra" veszített és a fazon kinyitottegy ajtót és $N>3$ akkor nekünk kisebb mint 100% esélyünk van "újra választunk"-kal nyerni.

```
1 #include <iostream>
2 #include <stdlib.h>
3
4 int main()
5 {
6     const int trial_count = 100000;
7     const int door_count = 3;
8     int count_keptbet_won = 0;
```

```
9  int count_newbet_won = 0;
10 for (int i = 0; i < trial_count; i++)
11 {
12     if(rand() % door_count == 0){
13         count_keptbet_won++;
14     }else{
15         if(rand() % (door_count-2) == 0) count_newbet_won++;
16     }
17 }
18 std::cout<< "Monty Hall"<< std::endl;
19 std::cout<< "Trial count " << trial_count << std::endl;
20 std::cout << "Wins(first bet) " << count_keptbet_won << std::endl;
21 std::cout << "Wins(next bet) " << count_newbet_won << std::endl;
22 std::cout << "Win Ratio " << ((count_newbet_won==0) ? 0.0 : (static_cast<←
    double>(count_keptbet_won)/static_cast<double>(count_newbet_won)))<< ←
    std::endl;
23 std::cout << "Sum (sanity check)" << count_keptbet_won+count_newbet_won ←
    << std::endl;
24 return 0;
25 }
```

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 $1*2*3*\dots*(n-1)*n*(n+1)$ 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 szám, ezekre (a jól ismert bizonyítás szerint) rendre igaz, hogy

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

- $(n+1)!+(n-1)=1*2*3*\dots*(n-1)*n*(n+1)+(n-1)$, azaz $(n-1)*\text{valamennyi}+(n-1)$, ami osztható $(n-1)$ -el
- $(n+1)!+n=1*2*3*\dots*(n-1)*n*(n+1)+n$, azaz $n*\text{valamennyi}+n$, ami osztható n -el
- $(n+1)!+(n+1)=1*2*3*\dots*(n-1)*n*(n+1)+(n+1)$, azaz $(n+1)*\text{valamennyi}+(n+1)$, 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 $(1/3+1/5)+(1/5+1/7)+(1/11+1/13)+\dots$ 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](#) nevű állománya kiszámolja az ikerprímeket, összegzi a reciprokaikat és vizualizálja a kapott részeredményt.

```
# Copyright (C) 2019 Dr. Norbert Bاتفai, nbاتفai@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 értelmezzük 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 $1/t_1\text{primes} + 1/t_2\text{primes}$ pedig ezeket a törteket rendre összeadja.

```
> 1/t1primes+1/t2primes  
[1] 0.53333333 0.3428571 0.1678322
```

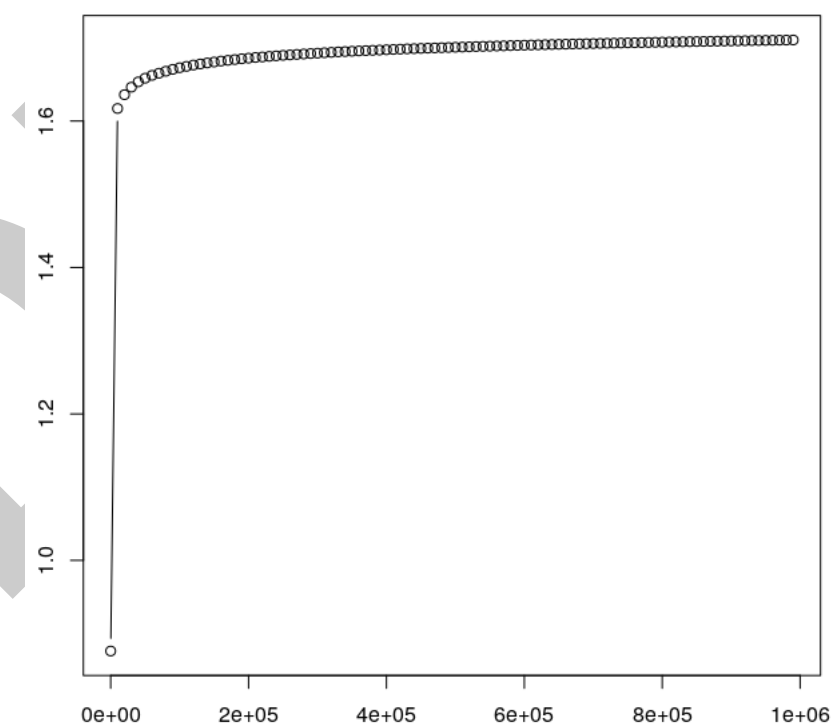
Nincs más dolgunk, mint ezeket a törteket összeadni a `sum` függvénnyel.

```
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.

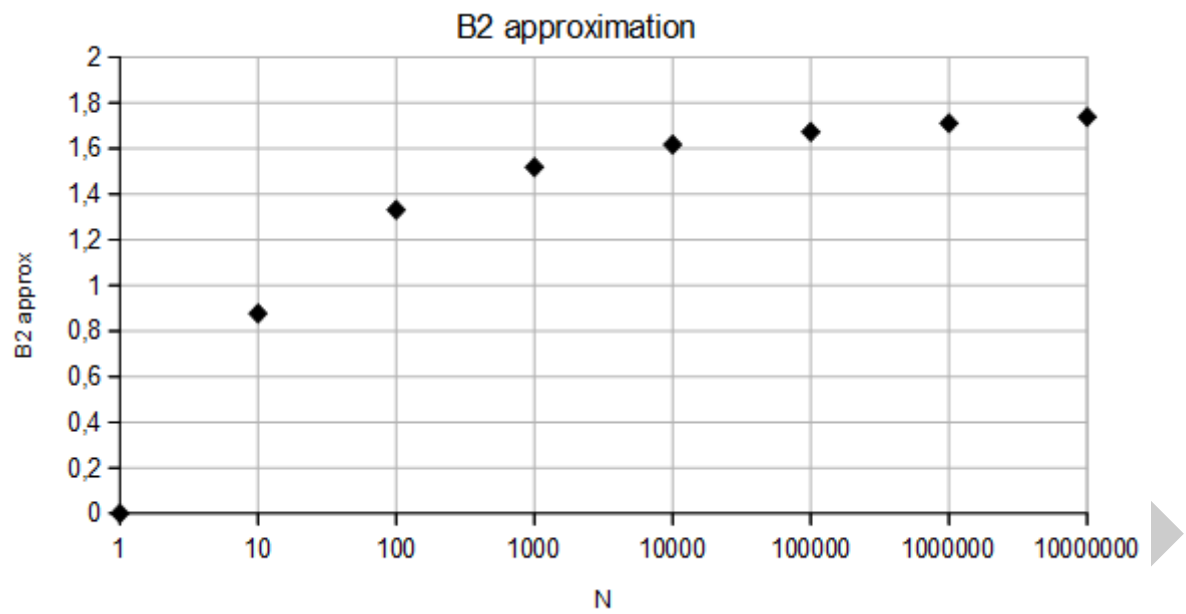
```
x=seq(13, 1000000, by=10000)  
y=sapply(x, FUN = stp)  
plot(x,y,type="b")
```



2.16. ábra. A B_2 konstans közelítése

Az nbatfai program C++-ban bitset-tel. (Bitset miatt compile time tudnunk kell N-et.)

```
1  #include <iostream>
2  #include <bitset>
3  #include <math.h>    // sqrt
4
5  const int N = 100; // The catch is we have to know N compile time
6
7  int main()
8  {
9      auto bs = std::move(std::bitset<N+1>{}.set());
10     bs.set(0, false);
11     bs.set(1, false);
12     int m = sqrt(N);
13     for (int p=2; p<=m; p++)
14         if (bs.test(p))
15             for (int i=p*2; i<=N; i += p)
16                 bs.set(i, false);
17     double b2approx = 0.0;
18     for(int i = 2; i <bs.size(); i++)
19         if(bs.test(i) && bs.test(i-2))
20             b2approx+=1.0/(i-2)+1.0/i;
21     std::cout<<"B2 approximation for "<<N<<" is "<< b2approx <<std::endl;
22     return 0;
23 }
24 /* N      B2approx
25 * 10      0.87619
26 * 100     1.33099
27 * 1000    1.51803
28 * 10000   1.61689
29 * 100000  1.6728
30 * 1000000 1.71078
31 * 10000000 1.73836
32 */
33 /*
34 * Didn't use accu, functional or lambdas
35 * But you could do it based on nbatfais R code...
36 * std::accumulate(t1.begin(), t1.end(), 0.0, [](const double &a, const ←
37 * double &b){return a+(1.0/b)+(1/(b+2.0));});
38 */
```



2.17. ábra. Az c++ B_2 konstans közelítés eredményei



Werkfilm

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

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet grájával megadva írd meg ezt a gépet!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

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

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

3.3. Hivatkozási nyelv

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

3.3.1. BNF

statement-ek BNF formában

```
<statement> ::= <labeled-statement>
```

```
4         | <expression-statement>
5         | <compound-statement>
6         | <selection-statement>
7         | <iteration-statement>
8         | <jump-statement>
9
10 <labeled-statement> ::= <identifier> : <statement>
11                       | case <constant-expression> : <statement>
12                       | default : <statement>
13
14 <expression-statement> ::= {<expression>}? ;
15
16 <compound-statement> ::= { {<declaration>}* {<statement>}* }
17
18 <selection-statement> ::= if ( <expression> ) <statement>
19                       | if ( <expression> ) <statement> else <statement>
20                       | switch ( <expression> ) <statement>
21
22 <iteration-statement> ::= while ( <expression> ) <statement>
23                       | do <statement> while ( <expression> ) ;
24                       | for ( {<expression>}? ; {<expression>}? ; {< ←
25                           expression>}? ) <statement>
26
27 <jump-statement> ::= goto <identifier> ;
28                   | continue ;
29                   | break ;
30                   | return {<expression>}? ;
```

3.3.2. inline functions

Inline régebben nem volt.

```
1 #include <stdio.h>
2
3 extern inline int getval(void);
4
5 int main()
6 {
7     int a = getval();
8     printf("%i", a);
9     return 0;
10 }
11
12 inline int getval(void){return 1;}
```

3.3.3. új típusok

Például long long int. Érdemes -Wall -Wpedantic-al compile-olni, hogy dobja a hibát.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     long long int a = 1;
6     printf("%I64d", a);
7     return 0;
8 }
```

Complex például nem volt része a szabványnak

```
1 #include <complex.h>
2 #include <tgmath.h>
3 #include <stdio.h>
4 int main(void)
5 {
6     double complex z1 = I * I;
7     printf("I * I = %.1f%+.1fi\n", creal(z1), cimag(z1));
8     return 0;
9 }
```

3.3.4. struct initialization

Például designated initialization

```
1 #include <stdio.h>
2
3 struct my_t{
4     int x;
5     int y;
6 };
7
8 int main()
9 {
10     struct my_t foo = {.x=1, .y=2};
11     printf("%i", foo.x);
12     return 0;
13 }
```

3.3.5. mixing decl and code

Deklaráció lehet akár kód után is

```
1 #include <stdio.h>
2
3 int main(){
4     int b;
```



```
5  b = 2;
6  int a = 1;
7  printf("%i %i", a, b);
8  return 0;
9  }
```

3.3.6. implicit fn

Régebben, ha például include math lemaradt, attól még compile működött, mert implicit egy int sin() deklaráció generált. Mivel ez többnyire nem egyezik a valóságossal, ezért ez általában hibához vezet.

```
1  #include <stdio.h>
2  int main(int argc, char **argv)
3  {
4      double d = sin(1.0);
5      printf("%f", d);
6      return 0;
7  }
```

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!

Első és legfontosabb, hogy math.h atoi atof miatt explicit beraktam. (Persze jogos kérdés, hogy miért kell egyszerre öv és hózentróger a nadrághoz...)

Első és legfontosabb, hogy math.h atoi atof miatt explicit beraktam. (Persze jogos kérdés, hogy miért kell egyszerre öv és hózentróger a nadrághoz...) A Tanár Úr verziója csak valós számokat parse-ol. Az enyémben int és float parse megy külön counterrel. Ahhoz hogy ezt elérjük definiáltuk a DIGIT-et, és ha 1 vagy több digit van egymás mellett és a végén terminál a karakterlánc, vagy whitespace van, akkor int, ha pedig pont van (és utána 0 vagy több DIGIT) akkor pedig float.

A másik dolog, hogy Tanár Úr verziója nem lép ki. Nos, a mi esetünkben a brexit karakterlánc hatására a program leáll.

Lehetne hosszabban taglalni a feladatot (pl. 1.1e3 parse?), de direkt rövidre vettük.

```
1  %option noyywrap
2  %{
3  #include <stdio.h>
4  #include <math.h>
5  int intnumbers = 0;
6  int floatnumbers = 0;
7  %}
8  DIGIT [0-9]
9  %%
```

```
10 {DIGIT}+      {++intnumbers; printf("[int=%s %i]", yytext, atoi(yytext));}  
11 {DIGIT}+"."{DIGIT}* {++floatnumbers; printf("[float=%s %f]", yytext, atof(↵  
    yytext));}  
12 "brexit"      { printf("Boris Johnson was called! Exiting EU!\n"); exit(0); ↵  
    }  
13 %%  
14 int  
15 main ()  
16 {  
17     yylex ();  
18     printf("The number of ints is %d\n", intnumbers);  
19     printf("The number of floats is %d\n", floatnumbers);  
20     return 0;  
21 }  
22
```

A lényeg annyi a flex az alap lex fájl alapján generál egy C forrásfájlt amit aztán le kell fordítani gcc-vel. Annyi hogy gcc compile esetén mellé kell linkelni valamely library-t dependencia miatt. (A bepasszolt -ll vagy -lml stb. flag). Ahhoz hogy ez menjen olykor a -lml stb. flag ELŐTT -L"Myath"-al meg kell adni a lib-et tartalmazó dir abszolút elérését.

A kigenerált forrásba nézzünk bele mert érdekes. Láthatóan label-eket használ és goto-t. De mit is ugrál ez a kis szörnyeteg? Nos, a Tanár Úr által említett Turing-os dologról van szó! Emlékezzünk a bevprog-os Bjarne Stroustrup-os példára. Ott például emlékezhetünk, hogy nem lehetett simán azt mondani, hogy 1 vagy több digit az int lesz, hiszen ha a következő char . akkor float-ot parse-olunk. Nos ez startégia van lekódolva label-ekkel. Ha state machine-ként képzeljük el, akkor gyakorlatilag ezek a label-ek a állapotok.

3.5. Leetspeak

Lexelj össze egy l33t ciphert!

Annyi történik, hogy a lexernél nem bonyolítottuk túl a szabályokat. Egy szabály van ami bármely char esetén érvényes.

Alábbi sor annyit jelent, hogy l337d1c7 egy array melynek elemei user defined cipher struct típusúak. Az hogy hány elemű l337d1c7 azt úgy tudjuk meg, hogy l337d1c7 méretét (memóriabeli foglalási méretét) elosztjuk cipher foglalási méretével. Eredményü azt kapjuk hány cipher van az arrayben. Magyarul ez csak egy handy dolog, hogy ne manuálisan kelljen megadni.(it would be prone to errors)

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

A lényege az egésznek, hogy stdin-re gyűlik az input. Mihelyst EOF van "aktivizálódunk", és minden karaktert egyenként beolvasunk (magyarul nincs sok értelme a lexernek, hisz nincs igazi szabály)

l337d1c7-t legegyszerűbb egy szótárként felfogni: Minden charhoz megadja a leet beli karakterlánc megfelelőit

Minden char-nál csinálunk egy lineáris keresést (nem a char-ral, hanem a lowercase verziójával), azaz addig megyünk l337d1c7 amíg meg nem találjuk azt a ciphert amelynek az adott charról szól. Mivel nem biztos

hogyan lesz ilyen a végén found-dal van egy guard ami annyit jelent, ha nem találtunk a szótárban, akkor írjuk vissza stdout-ra az eredeti chart.

Ha viszont benne volt a szótárban, akkor generálunk egy random számot(main-ben már megseedetük sys time-al és linuxon a saját pid-nkkal)

Ha a random szám

kisebb mint 91 akkor a szótárból az első karakterláncot írjuk ki stdout-ra

kisebb mint 95 akkor a szótárból a második karakterláncot írjuk ki stdout-ra

kisebb mint 98 akkor a szótárból a harmadik karakterláncot írjuk ki stdout-ra

egyébként a szótárból a negyedik karakterláncot írjuk ki stdout-ra

```
abba was a paganíc band carrying the bloodline of vicious viking warriors
4bb4 w4s 4 p4g4n1c b4nd c4rry1n1 th3 bl00dl1n3 0f v1c1<us v1king \^/4rr10rs
```

3.1. ábra. leetspeak

Alább a source kód

```
1  /*
2  Copyright (C) 2019
3  Norbert Bã;tfai, batfai.norbert@inf.unideb.hu
4
5  This program is free software: you can redistribute it and/or modify
6  it under the terms of the GNU General Public License as published by
7  the Free Software Foundation, either version 3 of the License, or
8  (at your option) any later version.
9
10 This program is distributed in the hope that it will be useful,
11 but WITHOUT ANY WARRANTY; without even the implied warranty of
12 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13 GNU General Public License for more details.
14
15 You should have received a copy of the GNU General Public License
16 along with this program. If not, see <https://www.gnu.org/licenses/>.
17 */
18 %option noyywrap
19 %{
20     #include <stdio.h>
21     #include <stdlib.h>
22     #include <time.h>
23     #include <ctype.h>
24
25     #define L337SIZE (sizeof l337d1c7 / sizeof (struct cipher))
26
27     struct cipher {
28         char c;
29         char *leet[4];
```

```
30 } l337d1c7 [] = {
31
32 { 'a', { "4", "4", "@", "/-\\\" }},
33 { 'b', { "b", "8", "|3", "|"} }},
34 { 'c', { "c", "(", "<", "{" }},
35 { 'd', { "d", "|)", "|]", "|"} }},
36 { 'e', { "3", "3", "3", "3"} }},
37 { 'f', { "f", "|=", "ph", "|#" }},
38 { 'g', { "g", "6", "[", "[+" }},
39 { 'h', { "h", "4", "|-|", "[-"] }},
40 { 'i', { "1", "1", "|", "!" }},
41 { 'j', { "j", "7", "_|", "_/" }},
42 { 'k', { "k", "|<", "1<", "|{" }},
43 { 'l', { "l", "1", "|", "|_" }},
44 { 'm', { "m", "44", "(V)", "\\|/" }},
45 { 'n', { "n", "\\|", "/\\/", "/V"} }},
46 { 'o', { "0", "0", "()", "[]" }},
47 { 'p', { "p", "/o", "|D", "|o"} }},
48 { 'q', { "q", "9", "O_", "(,)" }},
49 { 'r', { "r", "12", "12", "|2"} }},
50 { 's', { "s", "5", "$", "$"} }},
51 { 't', { "t", "7", "7", "'|'" }},
52 { 'u', { "u", "|_|", "(_)", "[_]" }},
53 { 'v', { "v", "\\|/", "\\|/", "\\|/" }},
54 { 'w', { "w", "VV", "\\|/\\|/", "(/\\|)" }},
55 { 'x', { "x", "%", ")(", ")("} }},
56 { 'y', { "y", "", "", "" }},
57 { 'z', { "z", "2", "7_", ">_" }},
58
59 { '0', { "D", "0", "D", "0"} }},
60 { '1', { "I", "I", "L", "L"} }},
61 { '2', { "Z", "Z", "Z", "e"} }},
62 { '3', { "E", "E", "E", "E"} }},
63 { '4', { "h", "h", "A", "A"} }},
64 { '5', { "S", "S", "S", "S"} }},
65 { '6', { "b", "b", "G", "G"} }},
66 { '7', { "T", "T", "j", "j"} }},
67 { '8', { "X", "X", "X", "X"} }},
68 { '9', { "g", "g", "j", "j"} }},
69
70 // https://simple.wikipedia.org/wiki/Leet
71 };
72
73 %}
74 %%
75 . {
76
77     int found = 0;
78     for(int i=0; i<L337SIZE; ++i)
79     {
```

```
80
81     if(l337d1c7[i].c == tolower(*yytext))
82     {
83
84         int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0));
85
86         if(r<91)
87             printf("%s", l337d1c7[i].leet[0]);
88         else if(r<95)
89             printf("%s", l337d1c7[i].leet[1]);
90         else if(r<98)
91             printf("%s", l337d1c7[i].leet[2]);
92         else
93             printf("%s", l337d1c7[i].leet[3]);
94
95         found = 1;
96         break;
97     }
98
99 }
100
101 if(!found)
102     printf("%c", *yytext);
103
104 }
105 %%
106 int
107 main()
108 {
109     srand(time(NULL));
110     yylex();
111     return 0;
112 }
```

Tanulságok, tapasztalatok, magyarázat...

3.6. A források olvasása

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

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

Nem kell túlbonyolítani... signal function-nek két argja van: signum, és handler.

Minden signal csak egy szám, hogy "hanyas csatornán jön az üzenet". Ha jön egy jel mondjuk a 15-ös csatornán, akkor az fogja kezelni, aki a 15-ösre be lett állítva.

Mikor signal-hoz új handlert rendelünk akkor return-ben visszakapjuk az eddigi handler fn ptr-ét.

Annyit még tisztázzunk hogy SIG_IGN egy makró ami egy default handler címévé expandál. Annyi a trükk hogy ennek a címe SOHA nem egyezik meg egy normális funkció címével se. Azaz ez az address unique.

Hogy tiszta legyen a kép, alább egy példa program a signal működéséről.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4
5 typedef void (*t_handler_fn) (int);
6
7 void do_nothing(int signum){
8     printf("Ez lesz az első sor! %i\n", signum);
9 }
10
11 void do_exit(int signum)
12 {
13     printf("Ez lesz a harmadik sor! %i\n", signum);
14     exit(0);
15 }
16
17 int main(void)
18 {
19     t_handler_fn last;
20     last = signal(SIGTERM, do_nothing);
21     raise(SIGTERM);
22     printf("Ez lesz a második sor!\n");
23     last = signal(SIGTERM, do_exit);
24     raise(SIGTERM);
25     printf("De ide nem jut el\n");
26     return 0;
27 }
```



Bugok

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

i.

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

Átállítom ignoredra. Ha eddig is ignored volt minden ok, de ha eddig nem ignored volt, akkor átrakom jelkezelőre.

ii.

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

Valid, prefix de mivel az expression return value semmire se kell, ezért lehetne postfix is

iii.

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

Valid, postfix

iv.

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

belép loop-ba, felveszi a nullát i, szóval `tomb[0]=0`

következő menetben a postfix miatt, először lemásolja a jelenlegi értéket, inkrementálja i-t, majd a másolt érték lesz a kiértékelés vége.

Magyarul `tomb[1]=0`

v.

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

Maybe unintended assignment versus equality comparison. (nagyon rossz debuggolni az ilyet, volt vele szerencsém)

vi.

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

EZ IGAZI BUG!

A kiértékelési sorrend nem függ sztenderdtől, szóval bármit kaphatunk (implementáció dönt).

vii.

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

Ha a egy ptr akkor okozhat meglepetéseket, de egyébként mivel az értékek alapvetően másolódva adódnak át, ezért ok lehet.

viii.

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

Ez sem eldönthető ennyi alapján, annyi hozzáfűzve, hogy f láthatóan okozhat mellékhatást, hisz címet kap, de akár ok is lehet.

3.7. 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

- egészret 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

Alább mellékeltem egy példát, próbáltam értelmessé tenni, szóval nem csak compile-ol, hanem eredményeket is ad...

```
1  #include <stdio.h>
2  int val = 8;
3  int val2 = 16;
4  // egészre mutató mutatót visszaadó függvény
5  int* get_val_ptr();
6
7  int* get_int_ptr(int, int);
8
9  int* (*get_fn_ptr(int))(int, int);
10
11 int add(int a, int b);
12
13 int (*get_addfn_ptr(int a))(int, int);
14
15 int main()
16 {
17     // egész
18     int a = 1;
19     printf("%i\n", a);
20     // egészre mutató mutató
21     int *ptr_a = &a;
22     printf("%i\n", *ptr_a);
23     // egész referenciája
24     int& ref_a = a;
25     printf("%i\n", ref_a);
26     // egészek tömbje
27     int arr1[2] = {2, 4};
28     printf("%i %i\n", arr1[0], arr1[1]);
29     // egészek tömbjének referenciája
30     int (&ref_arr1)[2] = arr1;
31     printf("%i\n", ref_arr1[0]);
32     // egészre mutató mutatók tömbje
33     int* arr2[2];
34     arr2[0] = &a;
35     arr2[1] = &arr1[1];
36     printf("%i %i\n", *arr2[0], *arr2[1]);
37     // egészre mutató mutatót visszaadó függvényt mutató mutató
38     int* (*fun0)(void) = get_val_ptr;
39     printf("%i\n", *fun0());
40     // egészet visszaadó és két egészet kapó függvényre mutató mutatót ←
41     // visszaadó, egészet kapó függvény
42     printf("%i\n", *(get_fn_ptr(1)(2, 4)));
```



```
42 // függvenymutató egy egészet visszaadó és két egészet kapó függvényre ↔  
    mutató mutatott visszaadó, egészet kapó függvényre  
43 printf("%i\n", get_addfn_ptr(1)(16, 16));  
44 return 0;  
45 }  
46  
47 int* get_val_ptr(){return &val;}  
48  
49 int* get_val2_ptr(int a, int b){return &val2;}  
50  
51 int* (*get_fn_ptr(int a))(int, int){ return get_val2_ptr;};  
52  
53 int (*get_addfn_ptr(int a))(int, int){ return add;};  
54  
55 int add(int a, int b){return a+b;}
```

Mit vezetnek be a programba a következő nevek?

- `int a;`
`int` típusú `a` nevű változó adott scope-ban.
- `int *b = &a;`
`ptr` típusú `b` nevű változó, mely `int` típusra mutat. értéke legyen az `a` változó címe.
- `int &r = a;`
referencia `r` néven `int` típusra. referencia mutasson a változóra.
- `int c[5];`
`tömb` típusú `c` nevű 5 elemű `int` típust tartalmazó, inicializálatlan (azaz ram szeméttől függnek értékei)
- `int (&tr)[5] = c;`
referencia `tr` néven tömbre `int` elemekkel 5-ös mérettel
- `int *d[5];`
`d` néven tömb(bár igazából minden tömb `ptr`) 5 elemű elemei `int ptr`-ek.
- `int *h ();`
`h` néven `int ptr` return type-ú void arg-ú függvény.
- `int *(*l) ();`
`l` néven `fn ptr`. A mutatott függvény `int ptr` return type-ú és void arg-ú

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

v néven function. v egy intet fogad és egy fn ptr-et ad vissza. A visszaadott fn ptr int-et ad vissza és két argja van int, int.

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

z néven egy fn ptr. Fn ptr egy int-et kaphat, és fn ptr-t ad vissza. A visszaadott fn ptr két intet fogad és int-et ad vissza.

A typedef használata következik most. DIREKT TÉRTEM EL Tanár ÚR példájától. Okom erre az volt, hogy az alábbi pattern-t nagyon gyakran használják C API-k esetén.

A lényeg az, hogy van valami adatstruktúra, vagy adat, vagy akármi amit zárni akarnak api használó elől, viszont indirekt hozzáférés kell hozzá.

Alábbi példában annyi történik, hogy (ismerős lehet Java, Cpp lambdákból) egy funkciót, és egy user defined void ptr-t adunk be, és a library ezt fogja hívogatni.

Annyival megspékeltük a dolgot, hogyha a visit function nem 0-t ad vissza akkor a lib megáll és nem iterál tovább.

Long story short, ez a program az 5 indexű elemig eliterál, de azt már nem írja ki és befejezi a visit-et.

```
1 #include <stdio.h>
2
3 typedef int ErrorCode;
4
5 typedef void* FooUserData;
6
7 typedef ErrorCode (*FooVisitFunction) (int, FooUserData);
8
9 struct CustomData{int a;};
10
11 int arr[] = {0,1,2,3,4,5};
12
13 int visit(int data, FooUserData ud)
14 {
15     CustomData* custom = (CustomData*) ud;
16     if(data>custom->a){return 1;}
17     printf("Visiting %i\n",data);
18     return 0;
19 }
20
21 int do_visit(FooVisitFunction fn, FooUserData ud)
22 {
23     int status = 0;
24     for(int i = 0; i<6;i++){
25         status = fn(arr[i],ud);
26         if(status !=0){ break; }
27     }
```

```
28     return status;
29 }
30
31 int main() {
32     CustomData cd = {.a=4};
33     do_visit(visit, (void*)&cd);
34     return 0;
35 }
```

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!

Mi az az alsó háromszög mx? Nos, vegyünk egy nxn-es mátrixot. A főátló és az alatti elemek az alsó háromszög mx.

```
X O O O O
X X O O O
X X X O O
X X X X O
X X X X X
```

Gyakorlati haszna az, hogy ha pl. szimmetrikus a mátrixunk, akkor elég az alsó háromszög mx-ot tárolni.

Nézzük a kódot, de előtte tisztázzuk malloc-ot és free-t!

malloc egy értéket vár, hogy hány bájt memória területet(konzisztens, tehát egybefüggően szabad) adjon. Amit vissza ad, az egy ptr a mem a blokk kezdőcímére.

free egy ptr-t vár. Adott ptr-t kezdeti címnek veszi és ezen címtől kezdve annyit szabadít fel, amennyit a ptr által mutatott típus foglal.

Alább egy malloc call:

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

Azaz kérünk memória nr darab double ptr-nek. Ha malloc nem tud adni, akkor null ptr jön vissza, ezáltal igaz lesz a kondíció és kilépünk -1-el.

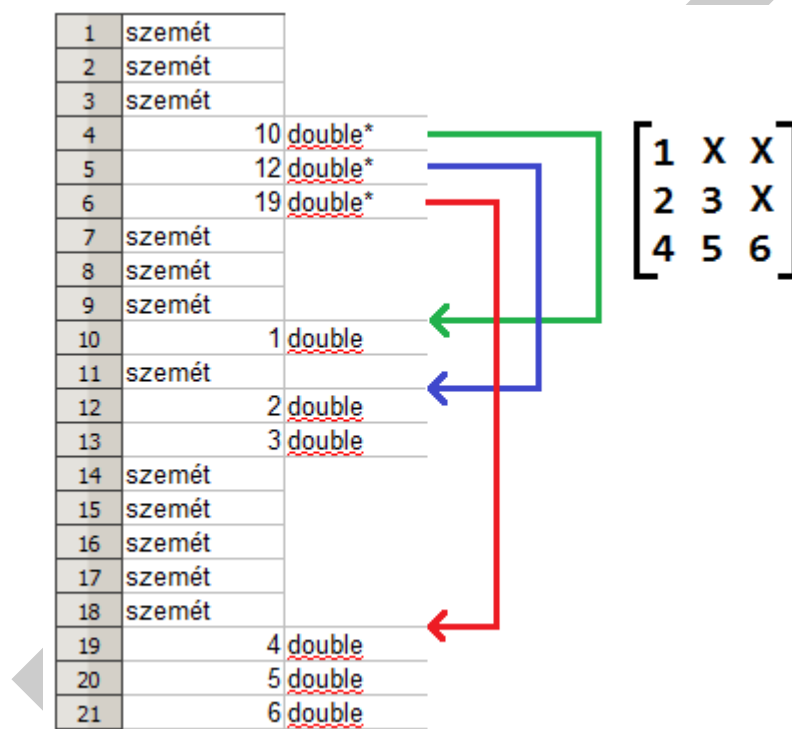
Persze, ez csak a madártávlati nézet, hiszen technikailag, tm-be assignoljuk a malloc által adott értéket és utána checkeljük, hogy tm==NULL -e.

A lényeg hogyha ezen túl vagyunk, akkor van egy tömbünk a memóriában pointerekkel (melyek double-re mutatnak). Ezek a pointerek jelenleg mem szemetet tárolnak, úgyhogy kezdjük el őket valódi címekkel feltölteni.

Mivel nr darab-szor kell elvégezni ugyanazt a műveletet majd, ezért logikusan jön egy for, de nekünk a for teste a lényeg.

```
if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL)
{
    return -1;
}
```

Mivel előbb a malloc-ot tisztáztuk, most inkább a méretbe és az elméletbe akarok belemenni. Mármost miért ez a (i+1) szorzó?



4.1. ábra. mem mx

Fenti ábrán látható, hogy minden pointer "eggyel hosszab" blokkra mutat mint a másik, az egész pedig egyről indul (hisz az első a főatlóbeli elemet tartalmazza csak)

Most ne menjünk bele az alignment-be, meg hogy valójában hogyan is tárolódik, inkább örüljünk az ábrának!

Azaz kérünk blokkokat, megkapjuk a címeket, és ezeket tm[i]-be assignoljuk...

Sajnos a double-ök még mindig memória szemetet tartalmaznak, ezért felülírjuk őket 0, 1, 2...stb. vel

```
for (int i = 0; i < nr; ++i)
    for (int j = 0; j < i + 1; ++j)
        tm[i][j] = i * (i + 1) / 2 + j;
Szamoljuk t[i][j] értékét i=2 j=2 esetben
tm[i][j] = i * (i + 1) / 2 + j;
```

```

tm[2][2] = 2 * (2 + 1) / 2 + 2;
tm[2][2] = 2 * 3 / 2 + 2;
tm[2][2] = 3 + 2;
tm[2][2] = 5;
  j 0 1 2
i
0 0
1 1 2
2 3 4 5

```

Ezekután stdoutra kiprinteljük a mx elemeit soronként

Ezekután jön egy kis trükközés

```

tm[3][0] = 42.0;
egyértelmű

```

```

*(tm + 3)[1] = 43.0;
(tm + 3) = tm + 3*sizeof(double*)
(tm+3) at derefeljük *-gal
azaz megmondjuk, hogy amit ott talál azt egy double ptr-ként ←
értelmezze (hisz ugye alaphól double** volt tm...)
utána pedig this[1]=*(this+1*sizeof(double))
Azaz a mátrixba assignolunk tm[3][1]=43

```

```

*(tm[3] + 2) = 44.0;
tm[3] = *(tm+3*sizeof(double*))
*(tm[3] + 2) pedig simán annyit hogy tm[3] + 2= tm[3] + 2*sizeof( ←
double)
deref miatt doubleként dolgozzuk fel a cym által mutatott helytől ←
kezdve a memóriát (és ugye annyit amennyi a sizeof(double))
Azaz a mátrixba assignolunk tm[3][2]=44

```

```

*(*(tm + 3) + 3) = 45.0;
(tm + 3) = tm + 3*sizeof(double*)
derefeljük úgyhogy double* kapunk.
hozzáadunk 3-at azaz *(tm + 3) + 3 = *(tm + 3) + 3*sizeof(double)
Azaz a mátrixba assignolunk tm[3][3]=45

```

Tanár Úr még kérdésként feltette mi van ha

```

*(tm + 3)[1] = 43.0;

```

Szóval **baj**. Indirection ugye RL asszociatív és 2-es precedence, míg [] LR asszociatív és 1-es precedence. Hogy jól eltudjam magyarázni csináltam egy labortenyésztett frankenstein példát:

```

*(tm + 3)[1] = 43.0;
*((tm + 3)[1])
*(*(tm + 3)+(1))

```

```

*(*(tm + 4))
**(tm + 4)
tm[4][0] Azaz a tm[4][0] értéket fogjuk átírni

```

```

0.000000,
1.000000, 2.000000,
3.000000, 4.000000, 5.000000,
42.000000, 7.000000, 44.000000, 45.000000,
43.000000, 11.000000, 12.000000, 13.000000, 14.000000,

```

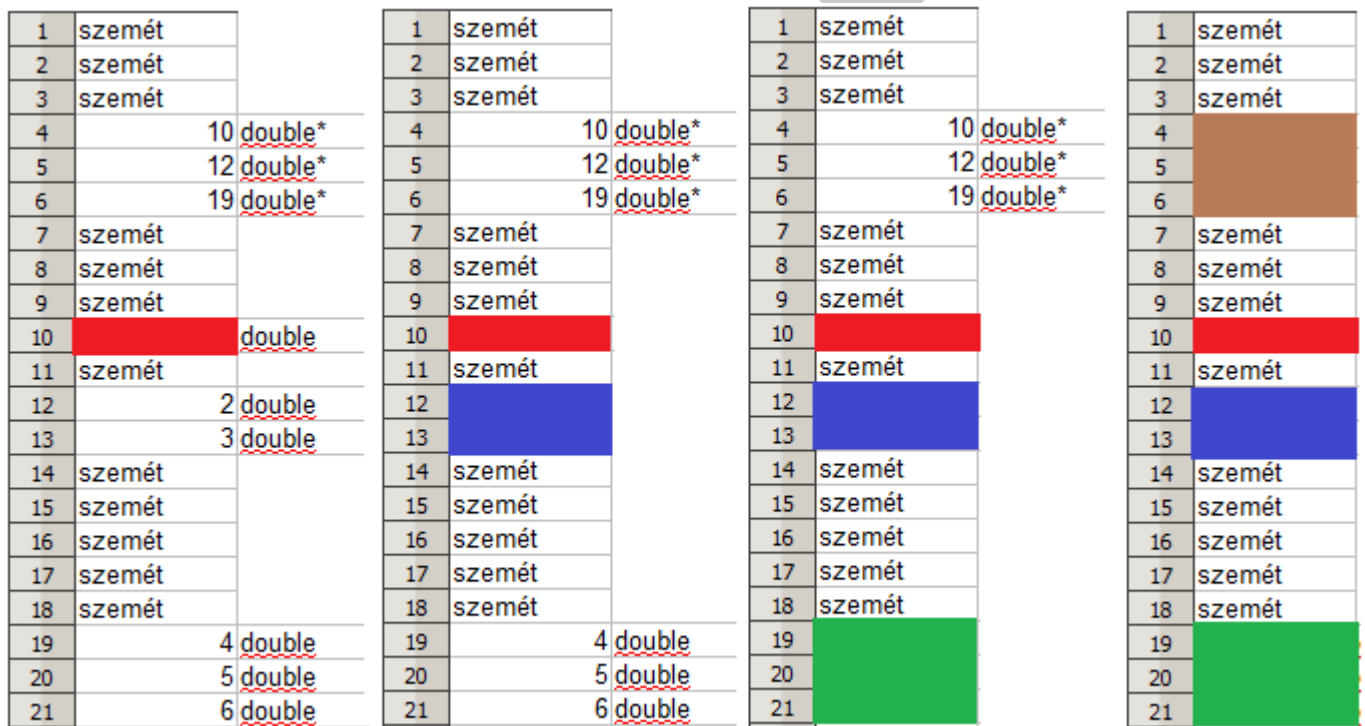
Újra kinyomtatjuk a mátrixot.

Most jön a lényeges rész: mi legyen a memóriával ha már nem kell?

```

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

```



1	szemét		
2	szemét		
3	szemét		
4		10	double*
5		12	double*
6		19	double*
7	szemét		
8	szemét		
9	szemét		
10			double
11	szemét		
12		2	double
13		3	double
14	szemét		
15	szemét		
16	szemét		
17	szemét		
18	szemét		
19		4	double
20		5	double
21		6	double

4.2. ábra. mem free

Megoldás forrása: [bmax/thematic-tutorials/bmax-textbook_IgyNeveldaProgramozod/Caesar/tm.c](https://bmax.thematic-tutorials.com/bmax-textbook/IgyNeveldaProgramozod/Caesar/tm.c)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3

```

```
4  int
5  main ()
6  {
7      int nr = 5;
8      double **tm;
9
10     printf("%p\n", &tm);
11
12     if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
13     {
14         return -1;
15     }
16
17     printf("%p\n", tm);
18
19     for (int i = 0; i < nr; ++i)
20     {
21         if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL ↵
22             )
23         {
24             return -1;
25         }
26     }
27
28     printf("%p\n", tm[0]);
29
30     for (int i = 0; i < nr; ++i)
31         for (int j = 0; j < i + 1; ++j)
32             tm[i][j] = i * (i + 1) / 2 + j;
33
34     for (int i = 0; i < nr; ++i)
35     {
36         for (int j = 0; j < i + 1; ++j)
37             printf ("%f, ", tm[i][j]);
38         printf ("\n");
39     }
40
41     tm[3][0] = 42.0;
42     (*(tm + 3))[1] = 43.0;
43     *(tm[3] + 2) = 44.0;
44     (*(tm + 3) + 3) = 45.0;
45
46     for (int i = 0; i < nr; ++i)
47     {
48         for (int j = 0; j < i + 1; ++j)
49             printf ("%f, ", tm[i][j]);
50         printf ("\n");
51     }
52
```

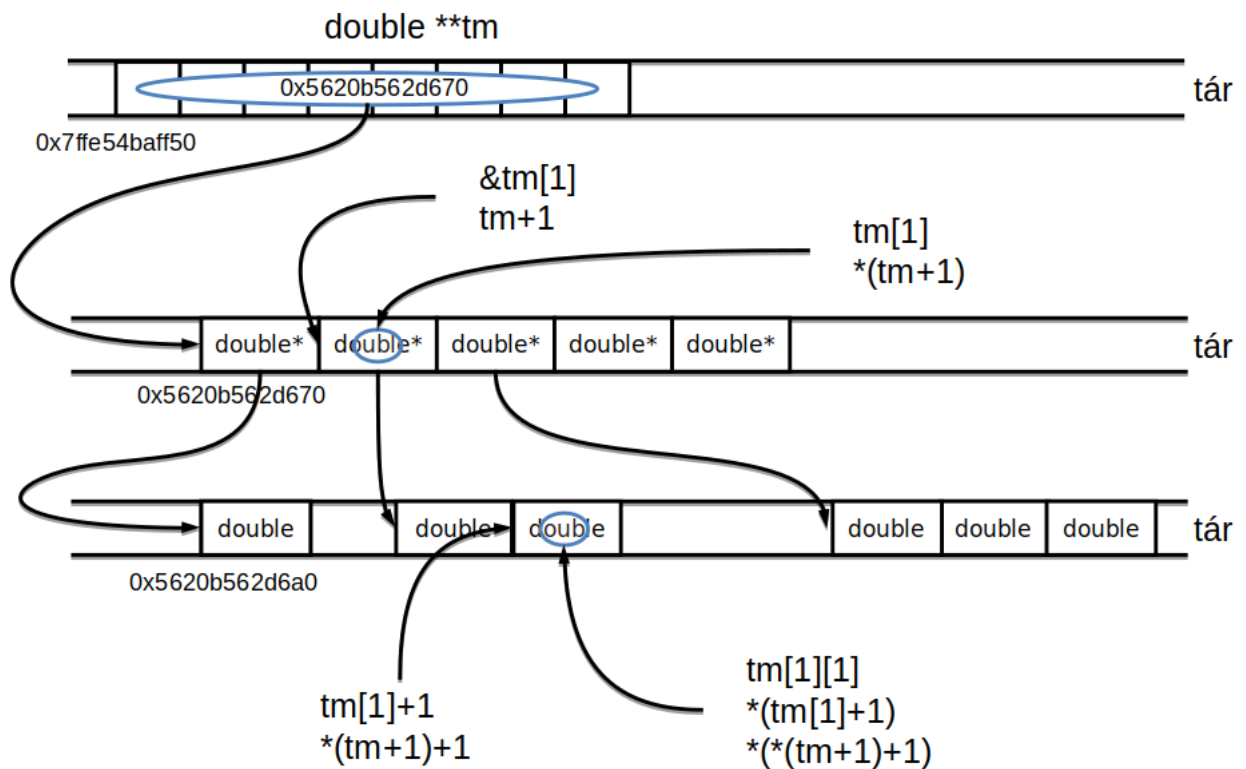


```

53     for (int i = 0; i < nr; ++i)
54         free (tm[i]);
55
56     free (tm);
57
58     return 0;
59 }

```

Bennt hagytam Tanár Úr képét mert jobb mint az enyém.



4.3. ábra. A `double **` háromszögmátrix a memóriában

A lényeg az egészből az, ha más nem is, hogy azért szertjük a `cpp`-t, mert nem kézzel kell takarítani, hanem vannak smart pointerek. Sajnos azonban `shared_ptr` esetén overhead-je van a dolognak. Illetve azt is érdemes fejben tartani, hogy "kivétel kezelés" esetén a memória kezelést nem szabad elfelejteni.

Ha valaki kíváncsi a címekre

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int
5  main ()
6  {
7      int nr = 5;
8      double **tm;
9

```

```
10     printf("%p\n", &tm);
11
12     if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
13     {
14         return -1;
15     }
16
17     printf("%p\n", tm);
18
19     for (int i = 0; i < nr; ++i)
20     {
21         if ((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL ←
22             )
23         {
24             return -1;
25         }
26     }
27
28     printf("%p\n", tm[0]);
29
30     for (int i = 0; i < nr; ++i)
31         for (int j = 0; j < i + 1; ++j)
32             tm[i][j] = i * (i + 1) / 2 + j;
33
34     for (int i = 0; i < nr; ++i)
35     {
36         for (int j = 0; j < i + 1; ++j)
37             printf ("%f, ", tm[i][j]);
38         printf ("\n");
39     }
40
41     tm[3][0] = 42.0;
42     //(* (tm + 3))[1] = 43.0;
43     printf("tm addr %p\n", &tm);
44     printf("tm[0] addr %p\n", &tm[0]);
45     printf("tm[1] addr %p\n", &tm[1]);
46     long long unsigned int pa = (long long unsigned int) &tm[1];
47     long long unsigned int pb = (long long unsigned int) &tm[0];
48     printf("dist tm[1] tm[0] %lu\n", ( pa - pb ));
49     printf("tm[4] addr %p\n", &tm[4]);
50     printf("(tm + 3)[1] addr %p\n", &((tm + 3)[1]));
51     double* p = (tm + 3)[1];
52     printf("p, tm[4][0] %p, %p\n", p, &tm[4][0]);
53     printf("*(tm + 3)[1], tm[4][0] %lf , %lf \n", *(tm + 3)[1], tm[4][0]);
54     *(tm + 3)[1] = 43.0;
55     *(tm[3] + 2) = 44.0;
56     *(* (tm + 3) + 3) = 45.0;
57
58     for (int i = 0; i < nr; ++i)
```

```
59 {  
60     for (int j = 0; j < i + 1; ++j)  
61         printf ("%f, ", tm[i][j]);  
62     printf ("\n");  
63 }  
64  
65 for (int i = 0; i < nr; ++i)  
66     free (tm[i]);  
67  
68 free (tm);  
69  
70 return 0;  
71 }
```

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

XOR-ral már találkoztunk a swap feladatnál. Ott már tárgyaltuk, hogy a XOR nagyon különleges művelet tulajdonságai miatt: kommutatív, asszociatív, létezik identitás elem, azaz létezik I, hogy bármely A esetén $A \text{ XOR } I = A$. Minden bitminta önmagával XOR-ozva Identitást ad, és azt már tisztáztuk hogy $A \text{ XOR } I = A$.

Ezek alapján az exor program a következő módon működik: Vesszük a titkosítandó szöveget és egy kulcsot. Képzeljük úgy, hogy a kulcsot többször a titkosítandó alá terítjük, és az egymás alatt álló karakterek össze XOR-ozásával kapjuk a titkosított szöveget.

ez egy uzenet
kulcskulcskul

Az első lépés "e" XOR "k"
(e) 01100101
(k) 01101011
00001110
...

Természetesen, ha titkosított szöveget ugyanazzal a kulccsal még egyszer le xor-ozzuk, akkor az eredeti szöveget kapjuk vissza:

00001110
(k) 01101011
(e) 01100101

Természetesen, ha titkosított szöveget ugyanazzal a kulccsal még egyszer le xor-ozzuk, akkor az eredeti szöveget kapjuk vissza:

Most viszont nézzük meg az implementáció specifikus részleteket.

A kódban használunk preprocesszornak szánt utasításokat:

```
#define MAX_KULCS 100
#define BUFFER_MERET 256
```

Ez egyszerűen annyit tesz, hogy a preprocesszor, ha belefut bármikor MAX_KULCS vagy BUFFER_MERET karakterláncokba, akkor ezeket a megadott int literalokra cseréli. Ezek egyébként technikailag azért kellene, mert tömbökkel fogunk dolgozni és ezek méretét compile time közölni kell a rendszerrel. Azért, hogy ne csak számokat adjunk be, ezért olyan nevekkel láttuk el ezeket a preprocesszoros módszerrel, mely a gépi kódot nem változtatja meg, de nekünk segít a kód olvasásban.

Nézzük, most a bementről olvasást! Mivel karakteres módban meglepetések érhetnek minket, ehelyett binárisan akarjuk olvasni a dolgokat. Tanár Úr megoldásától eltérően én DIREKT NEM stdin és stdout-tal dolgoztam. Ugyanis Mingw esetén csak és kizárólag binary mode-ban lehet a titkosított szöveget normálisan beolvasni. Szóval annyit csináltam, hogy fopen-nel az első explicit futtatási argumentum alapján próbálok READ és BINARY módban nyitni egy fájlt (ugyanazt kimenthez is). A lényeg, hogy ha sikeres akkor egy a fájlra mutató ptr-et kapok vissza.

```
FILE *fdi = 0;
fdi = fopen(argv[2], "rb");
```

Ezekután kezdődhet a beolvasás. fread-nek először átadunk egy ptr-t(buffer) ami az általa használható memória területre mutat (ide fogja írni a fájlból olvasott értékeket). Ezután az 1 konstans annyit mond, hogy minden egység 1 byte (ez sajnos elég veszélyes, lásd encoding). Ezután írjuk a max egyhuzamban olvasandó elem számát, ez a mi esetünkben a buffer mérete. Végül pedig a ptr az olvasandó fájlra. Vissza adott érték az olvasott bájtok száma. Ha 0 az olvasott bájtok száma, a while ciklus fejbeli conditional hamissá válik, azaz abbahagyjuk a ciklus futtatását.

```
while ((olvasott_bajtok = fread (buffer, 1, BUFFER_MERET, fdi)))
```

Ha sikeres az olvasás, akkor amíg az olvasott bájtok számát el nem érjük karakterenként össze xor-ozzuk a kiindulási szöveg és a kulcs egy karakterének bitjeit.

Mivel a kulcsot wraparoundolni kell, ezért egész osztást alkalmazunk. (Azaz, ha pl kulcsméret 8, és kulcs_index+1=8 lenne akkor az új kulcsindex $8\%8=0$ ezáltal elkerülve az index out of bounds esetet)

```
for (int i = 0; i < olvasott_bajtok; ++i)
{
    buffer[i] = buffer[i] ^ kulcs[kulcs_index];
    kulcs_index = (kulcs_index + 1) % kulcs_meret;
}
```

Ha ezzel meg vagyunk, akkor a mostmár "titkosított" buffer-t kiírjuk a kimenetre

```
fwrite (buffer, 1, olvasott_bajtok, fdo);
```

```
C:\Users\UBMCGU\ws_school\bhax-derived\thematic_tutorials\bhax_textbook_IgyNevel
daProgramozod\labor>exor.exe "20171014" "secret.txt" "tort.txt"
```

4.4. ábra. exorcnd

A példának a facebook csoportba 2017-ben feladott secret.txt-t használtam.

Bár az RSA jól adja át a nyilvános kulcsú titkosítás tulajdonságait, ←
egy dolgot még nem tárgyaltunk, mely a titkosítás egyik ←
alapkövetelménye, miszerint hogyan tehetjük biztossá, hogy ←
tényleg a feladótól kaptuk az üzenetet, és nem valaki más küldött ←
az ő nevében? Az alábbiakban ezt tárgyaljuk.

Tegyük fel, hogy Alíz (A) Bob (B) nyilvános kulcsát használja, hogy ←
egy titkosított üzenetet küldjön neki. Az üzenetében ←
bizonygathatja, hogy ő valóban A, de B-nek mégsem lesz semmi ←
konkrét bizonyítéka, hogy ténylegesen A írt neki, hiszen a ←
nyilvános kulcsát mindenki használhatja arra hogy titkos üzenetet ←
írjon neki. Ilyen bizonytalanságok elkerülése végett is ←
használható az RSA, hogy RSA szintű biztonsággal tanúsíthassuk ←
szerzői kilétünket. Ezzel a lépéssel pedig az RSA valódi ←
nyilvános kulcsú titkosító eljárássá növi ki magát.

Tehát A szeretne küldeni egy üzenetet B-nek. Kívág az üzenetéből egy ←
kis töredéket – ebből lesz a megjelölt üzenet – veszi ennek ←
mondjuk az ASCII értékét, ezt az értéket felemeli a d-edik ←
hatványára, majd veszi a kapott számot modulo N (pont így ←
csinálná, amikor dekódolna egy üzenetet), s a kapott végeredményt ←
aláírásként hozzátcsolja az egyszerű módon titkosított ←
üzenethez. (Mint látjuk ez is ugyanolyan hatásos mint ha az egész ←
üzenetével az előbb vázolt műveleteket hajtotta volna végre ←
csupán így sokkal gyorsabbá vált, hogy csak egy kis töredék ←
értékre mutatta meg, hogy tényleg A az üzenet szerzője).

Hogyan dekódolja az aláírást B?

Mikor megkapja a megjelölt üzenetet, az aláírást felemeli A ←
nyilvános e kitevőjére, s veszi modulo N az értéket (pont, mint ←
amikor A-nak kódolna egy üzenetet), mivel a két művelet egymás ←
inverzei, ezért összehasonlítja az eredményként kapott kivágott ←
üzenetet az üzenetben szereplő egyszerű módon kódolt ←
szövegrészlettel, s ha a kettő megegyezik, B biztosan tudhatja, ←
hogy az üzenet szerzője A titkos kulcsának birtokában volt.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <io.h>
5 #include <unistd.h>
6 #include <string.h>
7
8 #define MAX_KULCS 100
9 #define BUFFER_MERET 256
```

```
10
11 void usage(){
12     printf("exor [KEY] [INF] [OUTF]");
13 }
14
15 int
16 main (int argc, char **argv)
17 {
18     int status = 0;
19     char kulcs[MAX_KULCS];
20     char buffer[BUFFER_MERET];
21     int kulcs_index = 0;
22     int olvasott_bajtok = 0;
23     FILE *fdi = 0;
24     FILE *fdo = 0;
25     int kulcs_meret = 0;
26
27     if (argc != 4){
28         printf("Bad args \n");
29         usage();
30         return 1;
31     }
32
33     kulcs_meret = strlen (argv[1]);
34     if (kulcs_meret < 1){
35         printf("Key was empty \n");
36         usage();
37         return 1;
38     }
39     strncpy (kulcs, argv[1], MAX_KULCS);
40
41     fdi = fopen(argv[2], "rb");
42     if (fdi == 0){
43         printf("Failed to open in file \n");
44         usage();
45         return 1;
46     }
47
48     fdo = fopen(argv[3], "wb");
49     if (fdo == 0){
50         printf("Failed to open out file \n");
51         usage();
52         return 1;
53     }
54
55     while ((olvasott_bajtok = fread (buffer, 1, BUFFER_MERET, fdi))
56     {
57         for (int i = 0; i < olvasott_bajtok; ++i)
58         {
59             buffer[i] = buffer[i] ^ kulcs[kulcs_index];
```

```
60     kulcs_index = (kulcs_index + 1) % kulcs_meret;  
61 }  
62 fwrite (buffer,1, olvasott_bajtok,fdo);  
63 }  
64 }
```

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

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

Tanulságok, tapasztalatok, magyarázat...

4.4. C EXOR törő

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

A sima exor-hoz képest annyi a csavarás a feladaton, hogy nem tudjuk a kulcsot, illetve nem tudjuk hogy minek kell lennie a végeredménynek.

Az első problémát viszonylag egyszerű módon oldottuk meg, egyszerűen csak keresztül tekerünk egymásba ágyazott for loopokkal az összes lehetséges kulcson. Feltételezve, hogy a kulcs csak olyan 0,1,2,3,4,5,6,7,8,9 karakterekből állhat, és a kulcsméret BIZTOSAN 8.

A második probléma nehezebb! Hogyan tudjuk, hogy amit törtünk az, nos hogy jó-e? A program ezt heurisztikus módon oldja meg...vagy nem oldja meg. A lényeg annyi, hogy az algoritmus addig fog futni, amíg az összes kulcsot ki nem próbálta, viszont ha felmerül a gyanúja, hogy jó lehet a kulcs amit próbál, akkor stdoutra kiírja a siker gyanús kulcsot és a vele előállított szöveget.

A sikeresség eldöntéséhez két dolgot használunk: átlagos szóhossz és gyakori szavak

Az átlagos szóhossz...nos elég ember nyelv specifikus, mert , nos inkább nézzük: Mivel a space-eket össze-számolja és leosztja a szöveghosszat vele, ezért "aaa aaa aaa " = 4 "aaaaaaaaaaaaa "=4

A gyakori szavakat csak simán megpróbálja megtalálni a szövegben (ignore case módban). Sajnos az élet nem olyan egyszerű, ezért mingw esetén strcasestr-t meg kell írni. Nem, itt tényleg nem segít a #define _GNU_SOURCE string header előtt...

Na de lássuk a while fejből a read-et!(kicsit megtördeltem)

```
olvasott_bajtok  
=  
read ( 0,  
      (void *) p,  
      (p - titkos + OLVASAS_BUFFER < MAX_TITKOS) ? OLVASAS_BUFFER : ←  
      titkos + MAX_TITKOS - p))
```

Első arg 0, tehát stdin, eddig OK. következő egy p ptr, ami a buffer a read szempontjából. Most nézzük a ternary-t önmagában!

Már a fájl elején észre vehetjük MAX_TITKOS és OLVASAS_BUFFER-ből, hogy gyakorlatilag annyiról van szó, hogy max 4096 bájtot fogunk összvissz beolvasni, de ezt 256-os falatokban. Na de nézzük inkább a kódot, a ternary-t átírtam magyarázat miatt inkább egy if-re alább! (NE FELDJÜK: A read 3. argját számítjuk, ami azt adja meg hány bájtot kell olvasni a read-nek. Azaz az egésznek egy számot kell visszaadnia ami megmondja mennyit olvasson a read...)

```
int num;
if(p - titkos + OLVASAS_BUFFER < MAX_TITKOS)
{
    num = OLVASAS_BUFFER;
}else{
    num = titkos + MAX_TITKOS - p
}
```

Magyarázattal együtt:

```
bool a = (p - titkos + OLVASAS_BUFFER < MAX_TITKOS);
```

Próbáljuk ki pár értéken:

$(p - \text{titkos} + 256 < 4096) = ?$

Let $p=1024$, $\text{titkos}=1024$

$(1024 - 1024 + 256 < 4096) = (256 < 4096) = \text{true}$

Ekkor ugye azt mondjuk read-nek hogy OLVASAS_BUFFER-nyit tudunk ←
olvasson, szóval p-t növeljük 256-tal!

Let $p=4864$, $\text{titkos}=1024$

$(4864 - 1024 + 256 < 4096) = ((3840+256) < 4096) = ((4096) < 4096) = \text{false}$

Ekkor ugye azt mondjuk read-nek hogy $(\text{titkos} + \text{MAX_TITKOS} - p)$ -nyit ←
olvasson.

Ez jelen esetben $(1024 + 4096 - 4864) = 256$

Most pedig nézzük mi lesz ezután:

ugye p-t 256-tal növeltük, szóval most $p=4864+256=5120$

$(5120 - 1024 + 256 < 4096) = ((4096+256) < 4096) = ((4352) < 4096) = \text{false}$

Ekkor ugye azt mondjuk read-nek hogy $(\text{titkos} + \text{MAX_TITKOS} - p)$ -nyit ←
olvasson.

Ez jelen esetben $(1024 + 4096 - 5120) = 0$

Ez 0 bájt beolvasásával fog járni, ezáltal kiesünk a while-ból.

Ezekután annyi történik, hogyha a buffer nem telt meg teljesen akkor kinullázzuk, nehogy a mem szemét véletlenül infonak hasson...illetve string terminálás miatt (általában érdemes null terminálni a c stringeket)

Ezekután pedig csak simán a fentebb leírt xor-ozást és tiszta szöveg heurisztikus keresgetés zajlik.

Egyetlen fontos dolog van még: Ha titkost lexorozom, akkor a következő körben friss kulccsal baj lesz, hisz nem az eredetit hanem egy törtet kezdenék újratörni.

A probléma megoldása simán annyi (a xor műveleti tulajdonságai miatt), hogy nem plusz memóriát használok egy tiszta verzió tartására, hanem vissza xoroandom ugyanazon array-t. ÉS ugye emlékszünk hogy (A XOR B) XOR B = A

```
1  #define MAX_TITKOS 4096
2  #define OLVASAS_BUFFER 256
3  #define KULCS_MERET 8
4
5  #define _GNU_SOURCE
6  #include <ctype.h>
7  #ifdef _WIN32
8      char *strcasestr(const char *str, const char *pattern) {
9          int i;
10
11         if (!*pattern){
12             return (char*)str;
13         }
14         for (; *str; str++) {
15             if (toupper((unsigned char)*str) == toupper((unsigned char)*pattern ↔
16                 )) {
17                 for (i = 1;; i++) {
18                     if (!pattern[i]){
19                         return (char*)str;
20                     }
21                     if (toupper((unsigned char)str[i]) != toupper((unsigned ↔
22                         char)pattern[i])){
23                         break;
24                     }
25                 }
26             }
27             return 0;
28         }
29 #endif
30
31 #include <stdlib.h>
32 #include <stdio.h>
33 #include <unistd.h>
34 #include <fcntl.h>
35 #include <io.h>
36 #include <string.h>
37
38
39 double
40 atlagos_szohossz (const char *titkos, int titkos_meret)
41 {
42     int sz = 0;
43     for (int i = 0; i < titkos_meret; ++i)
44     {
```

```
45     if (titkos[i] == ' '){
46         ++sz;
47     }
48 }
49 if(sz == 0){return 0;}
50 else{ return ((double) (titkos_meret)) / sz; }
51
52 }
53
54 int
55 tiszta_lehet (const char *titkos, int titkos_meret)
56 {
57     double szohossz = atlagos_szohossz (titkos, titkos_meret);
58     return szohossz > 6.0 && szohossz < 9.0
59         && strcasestr (titkos, "hogy") && strcasestr (titkos, "nem")
60         && strcasestr (titkos, "az") && strcasestr (titkos, "ha");
61 }
62
63 void
64 exor (const char kulcs[], int kulcs_meret, char titkos[], int titkos_meret)
65 {
66
67     int kulcs_index = 0;
68
69     for (int i = 0; i < titkos_meret; ++i)
70     {
71
72         titkos[i] = titkos[i] ^ kulcs[kulcs_index];
73         kulcs_index = (kulcs_index + 1) % kulcs_meret;
74
75     }
76
77 }
78
79 int
80 exor_tores (const char kulcs[], int kulcs_meret, char titkos[],
81             int titkos_meret)
82 {
83
84     exor (kulcs, kulcs_meret, titkos, titkos_meret);
85
86     return tiszta_lehet (titkos, titkos_meret);
87
88 }
89
90 int
91 main (void)
92 {
93
94     char kulcs[KULCS_MERET];
```

```
95 char titkos[MAX_TITKOS];
96 char *p = titkos;
97 int olvasott_bajtok;
98 int status = 0;
99 /* stdin binary mingw... :)
100 status = _setmode( _fileno( stdin ), _O_BINARY );
101 if( status == -1 ){
102     printf( "Cannot set binary mode of stdin" );
103     return 1;
104 }
105 */
106 while ((olvasott_bajtok =
107     read (0, (void *) p,
108         (p - titkos + OLVASAS_BUFFER <
109             MAX_TITKOS) ? OLVASAS_BUFFER : titkos + MAX_TITKOS - p)))
110     {
111         p += olvasott_bajtok;
112     }
113 for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
114 {
115     titkos[p - titkos + i] = '\\0';
116 }
117
118
119 int loop = 0;
120
121 for (int ii = '0'; ii <= '9'; ++ii)
122     for (int ji = '0'; ji <= '9'; ++ji)
123         for (int ki = '0'; ki <= '9'; ++ki)
124             for (int li = '0'; li <= '9'; ++li)
125                 for (int mi = '0'; mi <= '9'; ++mi)
126                     for (int ni = '0'; ni <= '9'; ++ni)
127                         for (int oi = '0'; oi <= '9'; ++oi)
128                             for (int pi = '0'; pi <= '9'; ++pi)
129                                 {
130                                     kulcs[0] = ii;
131                                     kulcs[1] = ji;
132                                     kulcs[2] = ki;
133                                     kulcs[3] = li;
134                                     kulcs[4] = mi;
135                                     kulcs[5] = ni;
136                                     kulcs[6] = oi;
137                                     kulcs[7] = pi;
138
139                                     if (exor_tores (kulcs, KULCS_MERET, titkos, p - titkos))
140                                     {
141                                         printf
142                                             ("Kulcs: [%c%c%c%c%c%c%c%c]\nTiszta szoveg: [%s]\n",
143                                              ii, ji, ki, li, mi, ni, oi, pi, titkos);
144                                     }
```

```
145         // újra EXOR-ozunk, így nem kell egy második buffer
146         exor (kulcs, KULCS_MERET, titkos, p - titkos);
147     }
148     return 0;
149 }
```

4.5. Neurális OR, AND és EXOR kapu

R

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

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

Tanulságok, tapasztalatok, magyarázat...

4.6. Hiba-visszaterjesztéses perceptron

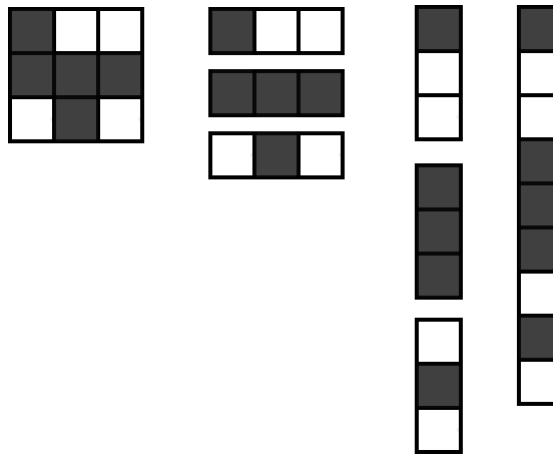
Ebben a feladatban a double*** handling a lényeg, de mindez cpp-ben, azaz malloc és free helyett new és delete. Tanár Úr kódját fogom használni, de a nem szükséges részeket kivettem (hisz egyébként 900 sor és például backpropagation nem kell...).

Mielőtt nekiugrunk a feladatnak nézzük végig matematikailag mit akart elérni Tanár Úr, és csak ezután nézzük a kódot. Ez így talán megfogja könnyíteni a megértést, hisz például a var arg handling nem a feedforward része, úgyhogy csak azután fogunk kitérni rá, hogy az alap matek megvan. Backproppal most nem fogunk foglalkozni, csak feedforward-al. Tegyük fel egy 3x3-as "képből" indulunk ki (nem rgb, csak fekete és fehér). A 3x3-asból előállítunk egy vektort. Ez lesz az input adata a hálónak.

Feladat legyen a következő: Kapunk egy 3x3-as fekete képet. Ezt beküldjük egy neurális háló kezdeménybe, és a végén például egy 1 elemű vektort kapunk. A vektor 0. eleme akkor és csak akkor 1 értékű, ha a képen mondjuk 4-es van. Cseresznye a torta tetején: Nincs bias.

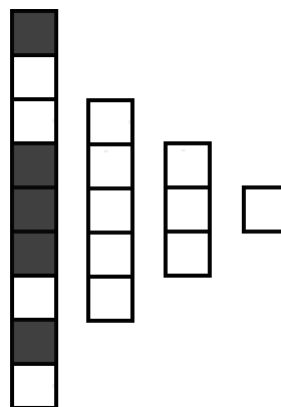
Első hallásra bonyolult, de valójában egyszerű:

A kép egy 3x3-as mátrix, lapítsuk egy vektorba. Ez lesz az input.



4.5. ábra. nn input

A háló hasraütésszerűen (totál mindegy, csak kellett a konkrét számok, hogy le tudjam rajzolni) 9-5-3-1



4.6. ábra. nn layers

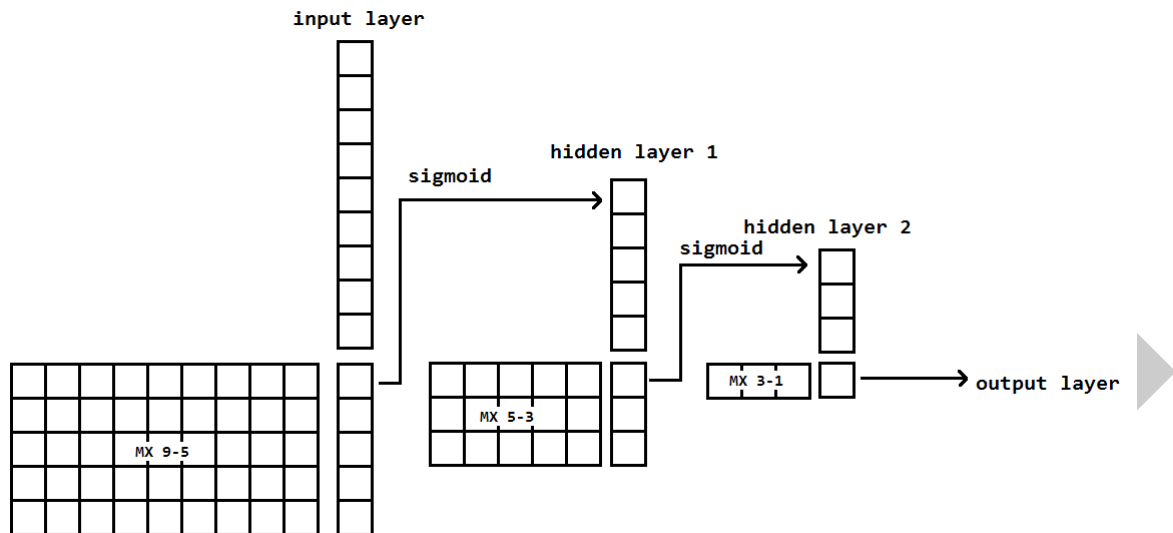
A háló a következő módon működik: nincs benne feedback, azaz i -edik layeren lévő node csak és kizárólag $i-1$ -edik node-ot használhat inputnak.

Ahelyett, hogy hardcodeoljuk ki kihez kapcsolódik a pagerank-nél látott módon mátrixos formában írjuk fel. Vegyük például a 9 elemű input layer és az 5 elemű layer (ő az első hidden layer) kapcsolatát leíró mátrixot. A layerek gyakorlatilag vektorok. A lényeg, hogy kell egy M kapcsolati mátrix. A célja annyi, hogyha megszorozom egy 9 elemű-vel, akkor egy 5 eleműt kapjak. A kapcsolati mátrix tehát pl. 5×9 -es, hisz $M \cdot A = B$:

$$\begin{array}{|c|c|c|} \hline ? & ? & ? \\ \hline ? & ? & ? \\ \hline ? & ? & ? \\ \hline \end{array}$$

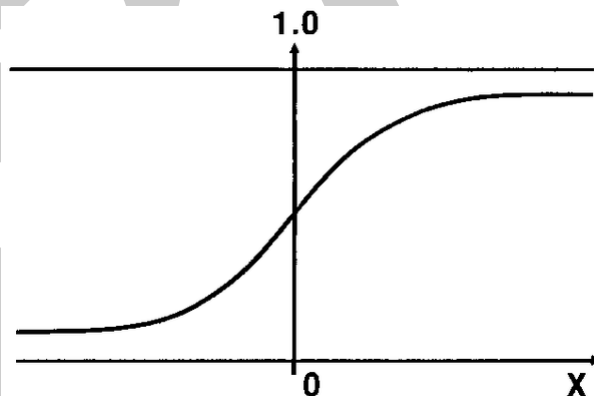
$$\begin{array}{|c|c|c|c|} \hline ? & ? & ? & ? \\ \hline ? & ? & ? & ? \\ \hline \end{array} = \begin{array}{|c|c|} \hline ? & ? \\ \hline ? & ? \\ \hline \end{array}$$

Azaz, ha kapok egy inputot, akkor össze szorzom a 9-5 közti trafó mátrixsal, és megkapom az 5 elemű layer értékeit. Utána ezt megszorozom az 5-3 trafóval, majd ezt megszorozom a 3-1 trafóval és kijön a végeredmény:



4.7. ábra. nmxmult

Azaz, az egész egy mátrix vektor szorzás lánc (sőt, mivel nincs bias ezért még elméletben sem kell trükközni). Egyetlen egy komoly trükk van. Mikor kijön egy mátrix-vektor szorzás segítségével egy új vektor, akkor minden elemén alkalmazzuk a sigmoid funkciót. Ez egy folytonos jellegű vágó függvény.



4.8. ábra. sigmoid

Ez alapján próbáljuk meg meghatározni milyen adatokra lesz szüksége az n layer számú Perceptron-nak. El kell tárolni n darab vektort. El kell tárolni $n-1$ darab transzformációs mátrixot. Mivel kicsit C-sen fogjuk írni a kódot, ezért nem csak az értékeket, ha nem a tömb méreteket is. Szóval el kell tárolni hogy hány layer van. El kell tárolni, hogy MINDEN vektor hosszát. A trafó mátrixok hosszait nem kell eltárolni, mert ahogy az ábrák is mutatják a vektor hosszakból számolhatóak.

Mielőtt belemegyünk a kódba, egy dolgot még tisztáznunk kell: Milyen értékei legyenek a weight mátrixoknak? Nos, teljesen random $[-1,1]$ tartománybeli double.

Most nézzük a kódot. Létrehozzuk a Perceptron osztályt. Ctor-ban inicializáljuk

A Perceptron (int nof, ...) kicsit furcsának tűnhet mert var arg-os, de nem kell megijedni. Simán annit jelent, hogy int nof után véges sok argumentum jöhet.

```
va_list vap;  
va_start ( vap, nof );
```

va_list-ből pedig va_arg-gal pedig megpróbálunk egy általunk megadott explicit típusú értéket "kiszedni" a listából. (Ezek makrók nem function call-ok, azaz ezeket a preprozessor expandálni fogja.)

```
n_units[i] = va_arg ( vap, int );
```

Egyből jön a kérdés, hogy itt akkor elméletileg bug-ok keletkezhetnek [default arg promotion](#) miatt. Igen ez így van de mi ezzel most nem foglalkozunk. Másik kérdés, hogy mi történik akkor, ha nof nem egyezik a megadott argumentumok száma -1-el.

Mikor megvagyunk a va_list traversalt illető makró használatával va_end-et használunk.

```
va_end ( vap );
```

Nézzük a ctor-t pici részekben:

```
n_layers = nof;  
units = new double*[n_layers];  
n_units = new int[n_layers];  
va_list vap;  
va_start ( vap, nof );  
for ( int i {0}; i < n_layers; ++i ){  
    n_units[i] = va_arg ( vap, int );  
    if ( i ){units[i] = new double [n_units[i]];}  
}  
va_end ( vap );
```

A Perceptron osztályt ctor-ban egy int-et kap, azaz a layer számot, illetve var arg-ként int-eket, amik az egyes layer-ekben lévő neuron számokat adják meg (azaz hogy melyik layer hány elemű vektor). Ezt simán átmásoljuk n_units-ba, ami egy int-ekből álló tömb. A units tárolja a vektorainkat, szóval érthető hogy ez miért is double** (n darab vektor matematikailag). Természetesen egyrészt magát units-ot initelni kell, és az általa mutatott double array-eket is. Ugyan értékeket még nem tudunk, de a szükséges hosszakat igen, hisz ezt a user megadta var arg-ban.

Annyit vegyünk észre, hogy ha $i == 0$, akkor nem kérünk helyet egy double array-nek. Ez amiatt van így, mert ez az inputunk, inputot meg nem fogjuk átmásolni, hanem majd units[0]-t megadjuk az input címének (hisz ugye units[0] egy double ptr!)

A másik nagyon fontos dolog, amit észre kell vennünk az az hogy nem malloc-ot használunk hanem new-t. Még hozzá úgy hogy new-nak nem adunk címet ahova hívja a ctor-t, azaz new MEMÓRIÁT IS FOGLAL és még ráadásul MEG IS HÍVJA a ctor-t. Persze kérhettün volna malloc-cal címet és a ptr-et megadva new-nak nem kért volna memóriát csak a megadott címen meghívta volna a ctor-t.

Na de most nézzük a weight mátrixot! Gyors kérdés kód előtt: Mit várunk?

Ugye `weights` egy csomó mátrix, tehát `weights = mátrix*`? Viszont mátrix egy csomó sor, vagy szám `n-es`, stb., azaz `mátrix=sor*`. Sor viszont egy csomó `double`, `vektor=double*`. Azaz `weights double***` lesz.

```
weights = new double**[n_layers-1];
std::default_random_engine gen;
std::uniform_real_distribution<double> dist ( -1.0, 1.0 );
for ( int i {1}; i < n_layers; ++i ){
    weights[i-1] = new double *[n_units[i]];
    for ( int j {0}; j < n_units[i]; ++j ){
        weights[i-1][j] = new double [n_units[i-1]];
        for ( int k {0}; k < n_units[i-1]; ++k )
        {
            weights[i-1][j][k] = dist ( gen );
        }
    }
}
```

Ahogy fent láthatjuk, kérünk helyet `weights`-nek (aki egy `double***`), aztán `weights` egyes mátrixainak (`double**`), mátrixok sorainak (`double*`) majd a sorok értékeinél egy `[-1,1]` tartománybeli random számot adunk.

A sigmoid-ot valamiért a class-ba raktuk(inkább rakjuk majd egy header file-ba inline-olva saját implementációban, hátha kelleni fog máshol is), de nem igényel túl sok magyarázatot.

```
double sigmoid ( double x ) {
    return 1.0/ ( 1.0 + exp ( -x ) );
}
```

Most pedig jöhet az `operator()` overload. Ő fogja csinálni azt a mátrix vektor szorzási láncot amit fentebb rajzoltunk.

Kicsit álljunk meg, tippeljünk meg előre mit fog csinálni!

Valószínűleg lesz egy argumentuma, ami egy `double*` lesz. Ez lesz az input layer bemeneti adata. Ezt KELL hogy assignolja `units[0]`-ra, hisz `units[0]`-nak nem kértünk egy új `double*`-t(és jelenleg ráadásul szuper veszélyes, mert nem állítottuk nullptr-re, hanem mem szemét van rajta, szóval ha most meghívánk akkor jó esetben segfaultolnánk.)

Ezekután pedig végig fogunk iterálni az összes trafón és elvégezzük őket. DE EMLÉKEZZÜNK `n` darab vektorunk van, amihez `n-1` trafó társul, azaz biztosan `n-1`-ig fogunk iterálni.

Egy adott `i`-edik iterációban pedig `i`-edik mátrixot `i`-edik vektorral szorozva `i+1`-edik vektort fogjuk kapni.

Technikailag mielőtt elvégezzük `i+1` elem kiszámítását, valószínűleg 0-ra fogjuk állítani, mert adja magát, hogy `+=` használjunk.

Miután kijönnek az értékek vagy egyben a végén, vagy még benn a loopban, az mátrix-vektor szorzatként kapott `i+1` vektor értékét átírjuk a sigmoid által hozzárendelt értékre

Nézzük a kódot


```
double operator() ( double image [] )
{
    units[0] = image;
    for ( int i {1}; i < n_layers; ++i ){
        for ( int j = 0; j < n_units[i]; ++j ){
            units[i][j] = 0.0;
            for ( int k = 0; k < n_units[i-1]; ++k ){
                units[i][j] += weights[i-1][j][k] * units[i-1][k];
            }
            units[i][j] = sigmoid ( units[i][j] );
        }
    }
    return sigmoid ( units[n_layers - 1][0] );
}
```

Az egyetlen váratlan dolog, hogy az output vektor 0. elemének értékéhez a sigmoid által hozzárendelt értéket visszaadjuk. Biztos a nagyobb progi így működött, azaz hogy valami miatt az output vektor 0. eleme fontos volt neki. Persze ez ne tévesszen meg senkit. Egy klasszifikációs probléma lehet olyan lesz, hogy mondjuk 10 output node lesz, és akkor kell felvillania mondjuk az 5-ös indexűnek mikor 5-öst lát, vagy mondjuk valami játékról van szó, és akkor kell felvillania egy 4 elemű vektor 2 elemének, ha keyboard S lenyomást akarunk szimbolizálni (FPS-eknél ez a hátra)

Alább az egész kód félig Cpp félig C-ben:

```
1 #include <iostream> // std out
2 #include <cstdlib> // var arg handling
3 #include <random> // init weight matrices
4
5 class Perceptron
6 {
7 public:
8     Perceptron ( int nof, ... )
9     {
10         n_layers = nof;
11         units = new double*[n_layers];
12         n_units = new int[n_layers];
13         va_list vap;
14         va_start ( vap, nof );
15         for ( int i {0}; i < n_layers; ++i ){
16             n_units[i] = va_arg ( vap, int );
17             if ( i ){units[i] = new double [n_units[i]];}
18         }
19         va_end ( vap );
20         weights = new double**[n_layers-1];
21         std::default_random_engine gen;
22         std::uniform_real_distribution<double> dist ( -1.0, 1.0 );
23         for ( int i {1}; i < n_layers; ++i ){
24             weights[i-1] = new double *[n_units[i]];
25             for ( int j {0}; j < n_units[i]; ++j ){
```

```
26         weights[i-1][j] = new double [n_units[i-1]];
27         for ( int k {0}; k < n_units[i-1]; ++k )
28         {
29             weights[i-1][j][k] = dist ( gen );
30         }
31     }
32 }
33 }
34
35 double sigmoid ( double x )
36 {
37     return 1.0/ ( 1.0 + exp ( -x ) );
38 }
39
40
41 double operator() ( double image [] )
42 {
43     units[0] = image;
44     for ( int i {1}; i < n_layers; ++i ){
45         for ( int j = 0; j < n_units[i]; ++j ){
46             units[i][j] = 0.0;
47             for ( int k = 0; k < n_units[i-1]; ++k ){
48                 units[i][j] += weights[i-1][j][k] * units[i-1][k];
49             }
50             units[i][j] = sigmoid ( units[i][j] );
51         }
52     }
53     return sigmoid ( units[n_layers - 1][0] );
54 }
55
56 ~Perceptron(){
57     for ( int i {1}; i < n_layers; ++i )
58     {
59         for ( int j {0}; j < n_units[i]; ++j ){
60             delete [] weights[i-1][j];
61         }
62         delete [] weights[i-1];
63     }
64     delete [] weights;
65     for ( int i {0}; i < n_layers; ++i )
66     {
67         if ( i ){delete [] units[i];}
68     }
69     delete [] units;
70     delete [] n_units;
71 }
72
73 private:
74     Perceptron ( const Perceptron & );
75     Perceptron & operator= ( const Perceptron & );
```

```
76
77     int n_layers;
78     int* n_units;
79     double **units;
80     double ***weights;
81
82 };
83
84 int main(){
85     Perceptron p(3,3,2,1);
86     double img[] = {0.0,0.9,0.7};
87     double res = p(img);
88     std::cout << res << std::endl;
89 }
```

Alább az egész kód Java-ban kilapított mátrixokkal. Itt az API-ban annyi változás van, hogy a full output vektort visszaadjuk.

```
1 package org.rkeeves.backprop;
2
3 import java.util.Arrays;
4 import java.util.Random;
5
6 public class NNet {
7
8     public class Matrix {
9
10         public Matrix(int rows, int cols)
11         {
12             this.cols = cols;
13             this.values = new Random().doubles(rows*cols, -1.0, 1.0).toArray();
14         }
15
16         public void transform(double[] from, double[] to) {
17             Arrays.fill(to, 0.0);
18             int row;
19             int col;
20             for (int i = 0; i < values.length; i++) {
21                 row = i / cols;
22                 col = i % cols;
23                 to[row] += values[row*cols+col]*from[col];
24             }
25             for (int i = 0; i < to.length; i++) {
26                 to[i] = sigmoid(to[i]);
27             }
28         }
29
30         private final int cols;
31         private final double values[];
32     }
33 }
```

```
34 public static final double sigmoid(double z) {
35     return 1.0/(1.0+Math.exp(-z));
36 }
37
38 public NNet(int[] vectors_sizes) {
39     vectors = new double[vectors_sizes.length][];
40     for (int i = 0; i < vectors_sizes.length; i++) {
41         vectors[i] = new double[vectors_sizes[i]];
42     }
43     transforms = new Matrix[vectors_sizes.length-1];
44     for (int i = 1; i < vectors_sizes.length; i++) {
45         transforms[i-1] = new Matrix(vectors_sizes[i], vectors_sizes[i-1]);
46     }
47 }
48
49 public double[] compute(double[] input) {
50     vectors[0] = input;
51     for (int i = 0; i < transforms.length; i++) {
52         transforms[i].transform(vectors[i], vectors[i+1]);
53     }
54     return vectors[vectors.length-1];
55 }
56
57 private final double[][] vectors;
58 private final Matrix[] transforms;
59
60 public static void main(String[] args) {
61     System.out.println(
62         Arrays.toString(
63             new NNet(new int[] {3,2,1})
64                 .compute(new double[] {0.1,-0.5,0.1})));
65 }
66 }
```

Többszálazással stb. nem foglalkoztam, mert a feladat az array hendlingről szólt. (Backproppal sem, mert explicit ki lett mondva hogy elég a feedforward)

5. fejezet

Helló, Mandelbrot!

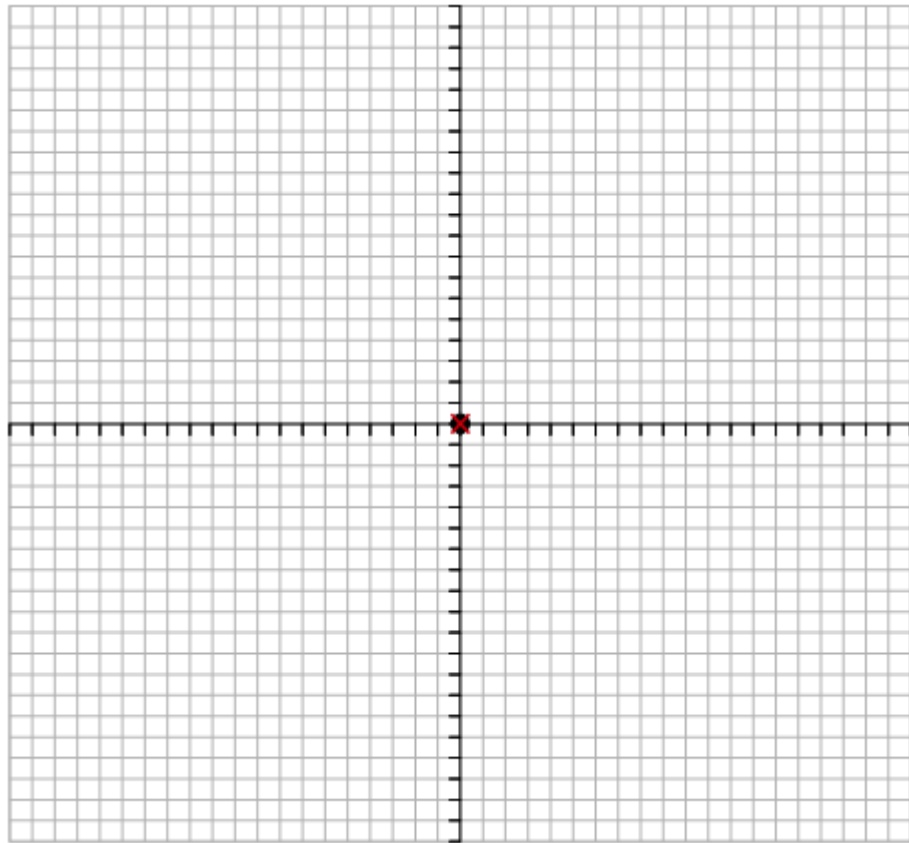
5.1. A Mandelbrot halmaz

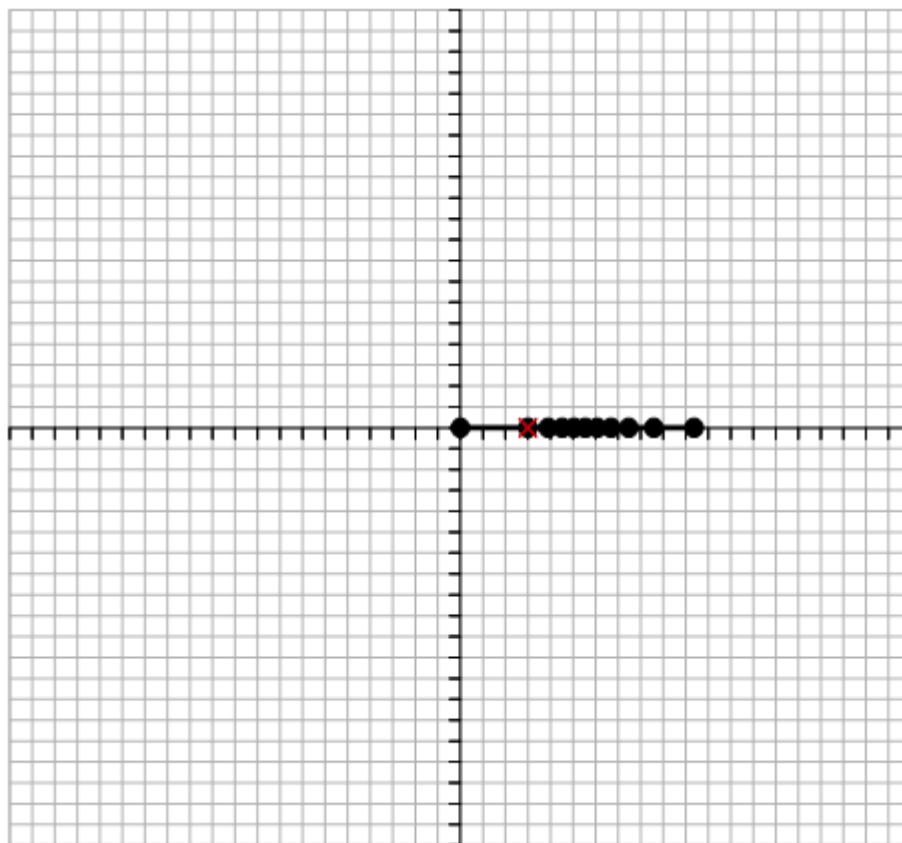
Írj olyan C programot, amely kiszámolja a Mandelbrot halmazt!

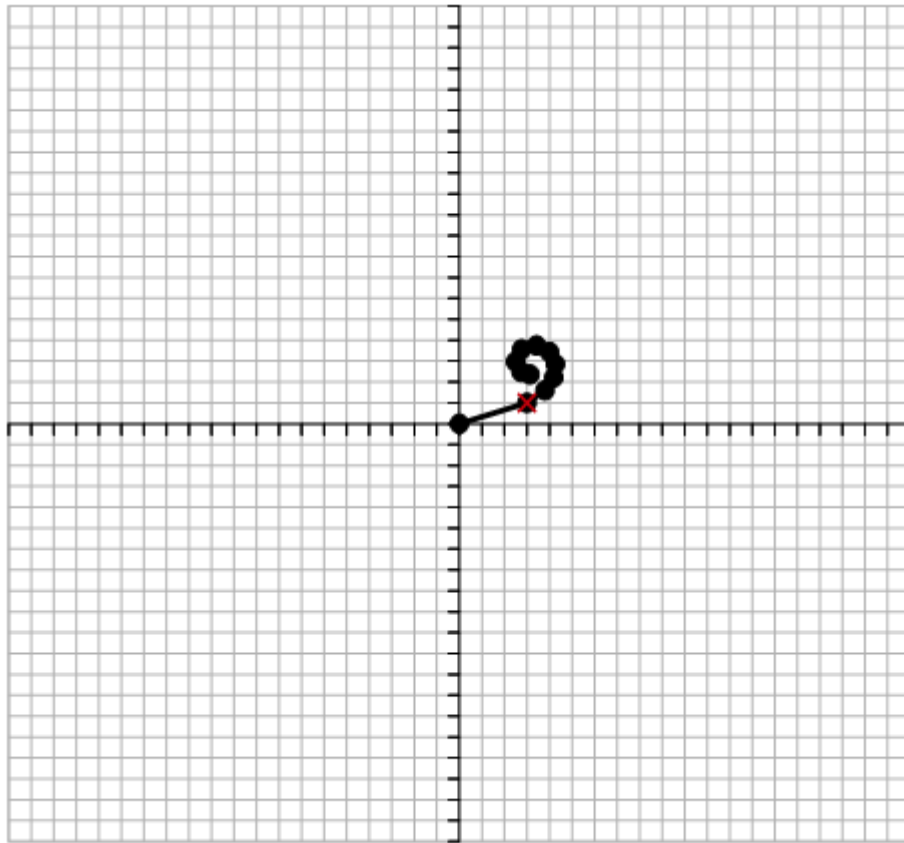
A Mandelbrot halmaz számításához egy kis matekra lesz szükség. Nagyon vizualizációba most nem fogunk kód oldalról belemenni, mert egy másik alfejezet pont erről fog szólni...

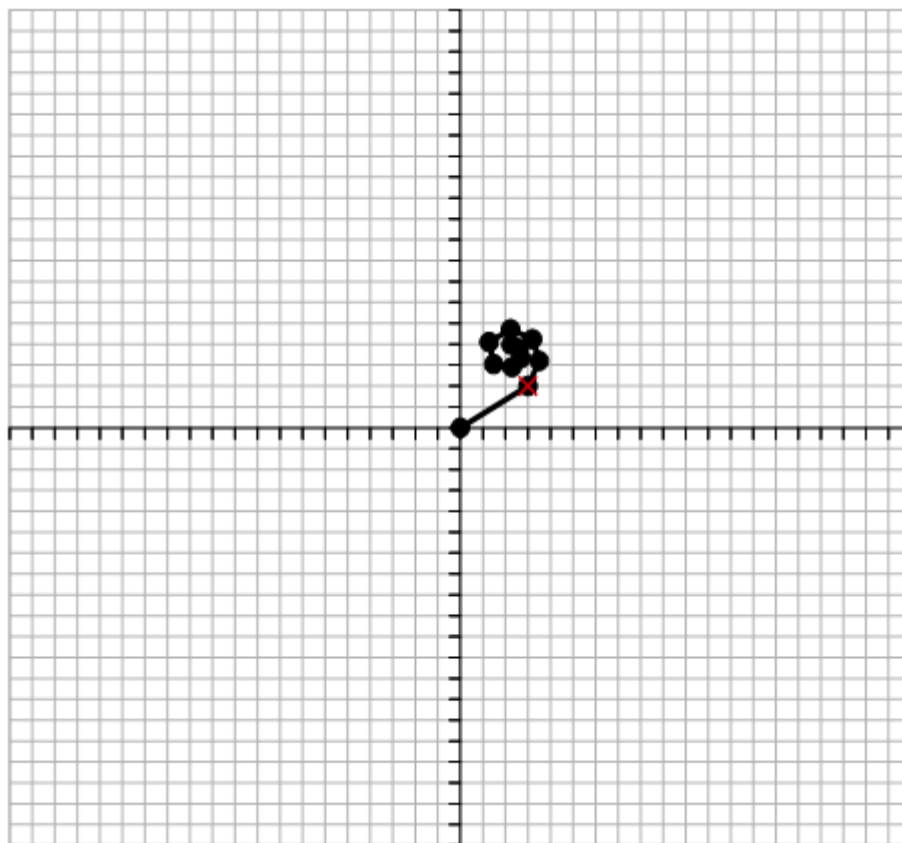
```
z[0] = 0
z[1] = z[0]^2+c
z[2] = z[1]^2+c
...
z[n] = z[n-1]^2+c
```

Azaz mindig 0, 0 pontból indulunk, majd négyzetét vesszük és egy konstans számot hozzáadunk. Nézzünk pár esetet! Az ábrákon valós tengely vízszintes, míg képzetes függőleges. Az ábrán piros X-el jelöltem a konstanst és az step-eket fekete pontokkal (össze is kötöttem őket).

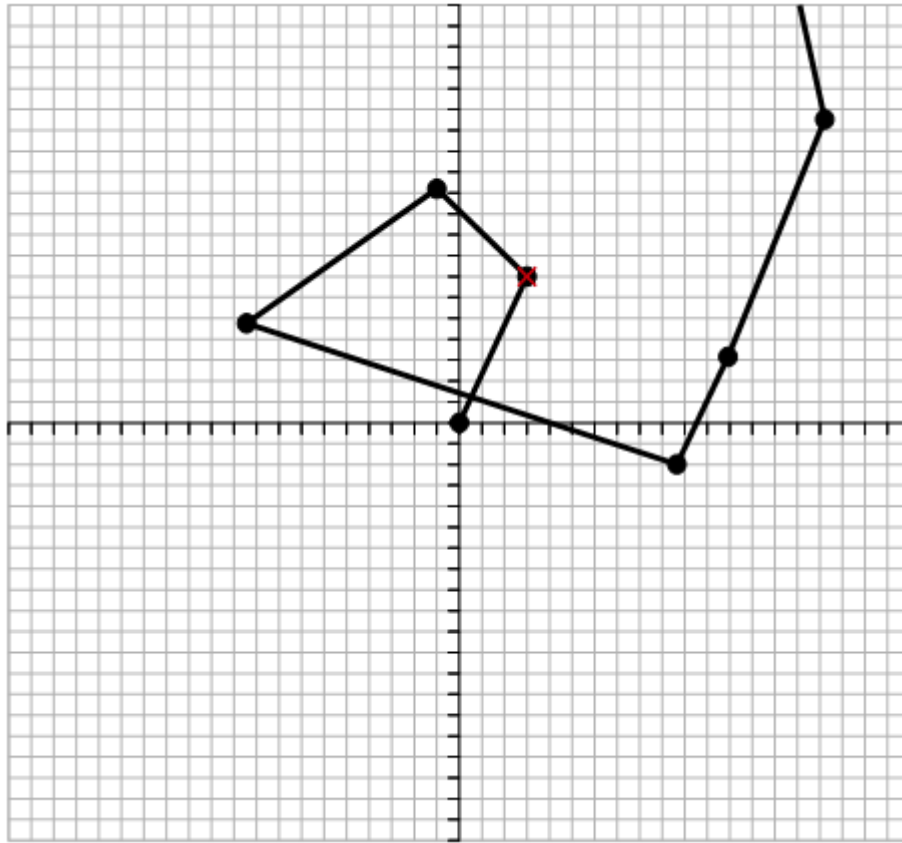
5.1. ábra. $c = \{0,0\}$

5.2. ábra. $c = \{0.3, 0\}$

5.3. ábra. $c = \{0.3, 0.1\}$

5.4. ábra. $c = \{0.3, 0.2\}$

Ha a konstant például 0.3, 0.7 akkor nagyon gyorsan "elszáll" az érték.

5.5. ábra. $c = \{0.3, 0.7\}$

Mit is csináltunk tulajdonképpen? Nos, rögzítettük 0,3 re értéket, és elkezdünk "pásztázni" im értékeket 0,1-es lépés közzel. A kódban is ezt fogjuk csinálni, csak minden pásztázott ponthoz (re im pár) rögzíteni fogjuk mennyi iteráció után kerültek ki az origó 2 sugarú köréből. Azért hogy ne fusson a végetelenségig, ezért az iterációknak lesz egy teteje. Azaz ha a max iter számot elérjük akkor meg fogunk állni.

A user megad egy re alsó és felső határt (megtől meddig pásztázunk Re tengelyen), illetve ugyanezt Im tengelyen. A lépésszám állítható is lenne, de mivel a legvégén terminálra akarunk írni, ezért ettől most eltekintünk.

Tanár Úrtól eltérően én feldaraboltam a kódot, de csak azért hogy egyszerűbb legyen bemutatni.

Az első kód részlet egy olyan funkció, aminek adunk egy komplex konstanst és egy max iter számot, és ez a fent bemutatott algoritmust végigcsinálja. A visszaadott érték az iter szám. Ha ez alacsony akkor nagyon gyorsan elszállt a számítás, ha magas akkor sokáig stabil volt.

```
int mandel(  
    float c_re,  
    float c_im,  
    int max_iter)  
{  
    int i = 0;  
    float temp=0.0, z_re=0.0, z_im=0.0;  
    while ( ( ((z_re*z_re)+(z_im*z_im)) < 4.0 ) && ( i < max_iter ) )
```

```
{
    temp = (z_re*z_re) - (z_im*z_im) + c_re;
    z_im = 2.0*(z_re*z_im) + c_im;
    z_re=temp;
    i++;
}
return i;
}
```

A második kód részlet már csak boilerplate kód. Erősen átírtam Tanár Úréhoz képest, direkt hogy `int**`-t használjunk. A másik dolog, hogy Tanár Úré valami miatt a második `for` bodyban számolja a reál és imaginárius részét is érthetőség miatt, én ezt egy picit optimáltam. (elég a külsőben számolni az egyiket) Annyi történik, hogy végig járjuk a kapott rácsot és kitöltögetjük, hogy melyik komplex konstans-hoz mennyi iter tartozik.

```
void apply_mandel(
    int** quad_mx,
    int re_size,
    int im_size,
    float re_lo,
    float re_hi,
    float im_lo,
    float im_hi,
    int max_iter)
{
    int re_step, im_step, itercount;
    float d_re, d_im, c_re, c_im;
    if(quad_mx == NULL || re_size < 1 || im_size < 1) return;
    d_re = (re_hi-re_lo)/re_size;
    d_im = (im_hi-im_lo)/im_size;
    for(re_step = 0; re_step < re_size; ++re_step){
        c_re = re_lo + d_re * re_step;
        for(im_step = 0; im_step < im_size; ++im_step){
            c_im = im_lo + d_im * im_step;
            itercount = mandel( c_re, c_im, max_iter);
            quad_mx[re_step][im_step] = itercount;
        }
    }
}
```

A harmadik rész maga a `main`. Csak beolvassuk a `user`-től mely határok között számoljunk, illetve deklaráljuk `init` elnevezéssel az `int**` `ptr`-t, plusz a végén feltakarítunk.

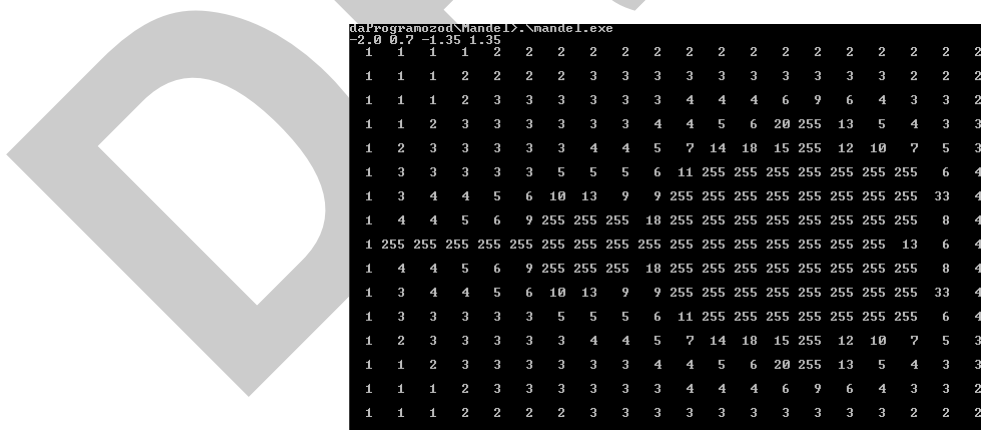
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
```

```
5 int mandel(
6     float c_re,
7     float c_im,
8     int max_iter)
9 {
10     int i = 0;
11     float temp=0.0, z_re=0.0, z_im=0.0;
12     while ( ( ((z_re*z_re)+(z_im*z_im)) < 4.0 ) && ( i < max_iter ) )
13     {
14         temp =(z_re*z_re)-(z_im*z_im)+c_re;
15         z_im = 2.0*(z_re*z_im)+c_im;
16         z_re=temp;
17         i++;
18     }
19     return i;
20 }
21
22 void apply_mandel(
23     int** quad_mx,
24     int re_size,
25     int im_size,
26     float re_lo,
27     float re_hi,
28     float im_lo,
29     float im_hi,
30     int max_iter)
31 {
32
33     int re_step,im_step,itercount;
34     float d_re,d_im,c_re,c_im;
35     if(quad_mx == NULL || re_size < 1 || im_size < 1) return;
36     d_re = (re_hi-re_lo)/re_size;
37     d_im = (im_hi-im_lo)/im_size;
38     for(re_step = 0; re_step<re_size;++re_step){
39         c_re = re_lo + d_re * re_step;
40         for(im_step = 0; im_step<im_size;++im_step){
41             c_im = im_hi - d_im * im_step;
42             itercount = mandel( c_re,c_im,max_iter);
43             quad_mx[re_step][im_step]=mandel( c_re,c_im,max_iter);
44         }
45     }
46 }
47
48 #define MANDEL_RE_SIZE 20
49 #define MANDEL_IM_SIZE 16
50 #define MANDEL_ITER_MAX 255
51
52 #define MANDEL_RE_LO -2.0
53 #define MANDEL_RE_HI 0.7
54 #define MANDEL_IM_LO -1.35
```

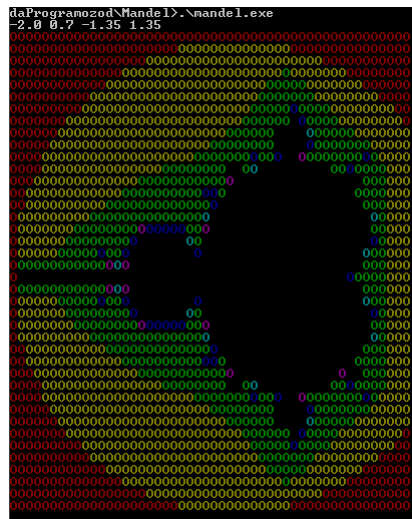
```

55 #define MANDEL_IM_HI 1.35
56
57 int main()
58 {
59     float re_lo, re_hi, im_lo, im_hi;
60     scanf("%f %f %f %f", &re_lo, &re_hi, &im_lo, &im_hi);
61     if(re_lo >= re_hi || im_lo >= im_hi){
62         printf("bad limits");
63         return 1;
64     }
65     int n_re, n_im, x, y;
66     int** mx;
67     n_re = MANDEL_RE_SIZE;
68     n_im = MANDEL_IM_SIZE;
69     if ((mx = (int **) malloc (n_re * sizeof (int *))) == NULL){ return -1;}
70     for (x = 0; x < n_re; ++x){
71         if ((mx[x] = (int *) malloc (n_im * sizeof (int))) == NULL){return -1;}
72     }
73     apply_mandel(mx, n_re, n_im, re_lo, re_hi, im_lo, im_hi, MANDEL_ITER_MAX);
74     for(y = 0; y < n_im; ++y){
75         for(x = 0; x < n_re; ++x){
76             printf("%3d ", mx[x][y]);
77         }
78         printf("\n");
79     }
80     for (x = 0; x < n_re; ++x){
81         free (mx[x]);
82     }
83     free (mx);
84     return 0;
85 }

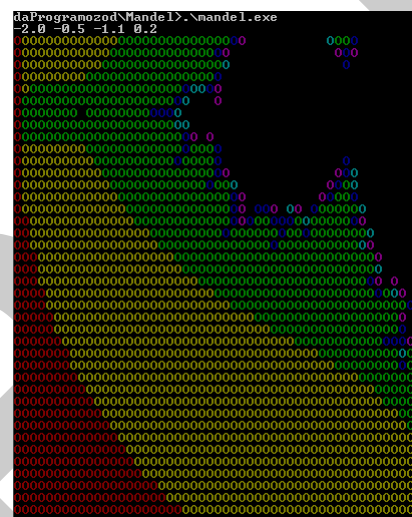
```



5.6. ábra. A Mandelbrot halmaz win, csak iter számok



5.7. ábra. A Mandelbrot halmaz win, színes



5.8. ábra. A Mandelbrot halmaz win, színes, határ állítás

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

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

Png++ helyett SDL2-t fogunk használni, és egyenesen a képernyőre renderelünk. A feladat szempontjából érdekes részek a MandelDraw class-ba kerültek.

Az osztály az alábbi állapotváltozókkal rendelkezik. Egyedül a `b` `c` és `d` szorulnak külön magyarázatra. Ugye egy valós és képzetes tengelyünk van, `a` és `b` azt mutatja meg hogy a valós tengelyen mettől meddig számtunk. `c` és `d` azt mutatja meg hogy a képzetes tengelyen mettől meddig számtunk.

```
int w; // screen width
```

```
int h; // screen height
int iter_lim; // iteration limit for mandel
double a; //
double b;
double c;
double d;
```

Az egyetlen funkció a render. Ez egy SDL renderer ptr-t kap és elvégzi az értékek számítását és a rendernek megmondja mely pixel milyen rgba számokat kapjon.

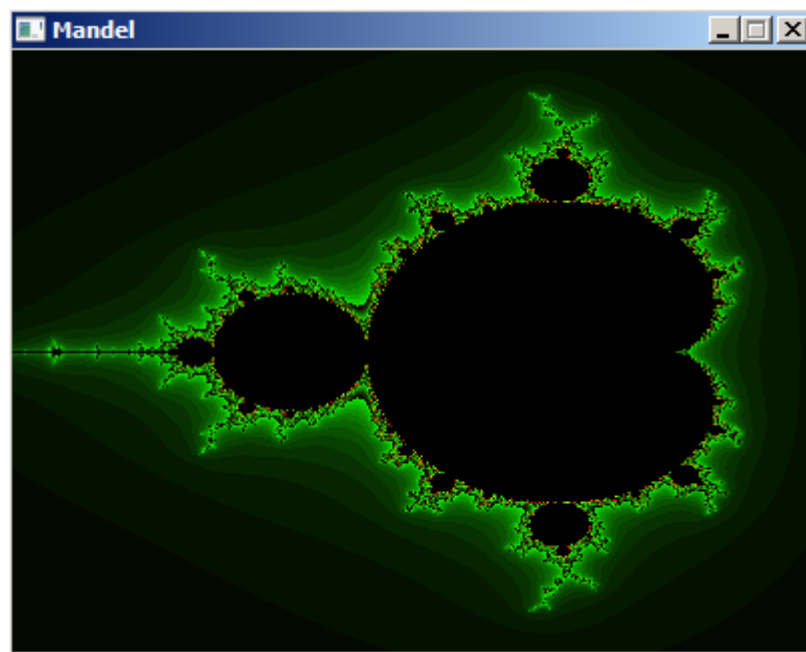
Maga a számítás ugyanaz mint az előbbieken. j a képzetes része a koordinátáknak, k a valós része. j és k -t úgy mappeljük fel a képernyőre, hogy j jelentse a képernyő esetén az y -t (magasság), k pedig x -et azaz hosszát.

```
void render(SDL_Renderer *renderer)
{
    int szazalek = 0; // unused
    double dx = ( b - a ) / w;
    double dy = ( d - c ) / h;
    double reC, imC, reZ, imZ;
    int iteracio = 0;
    for ( int j = 0; j < h; ++j ){
        imC = d - j * dy;
        for ( int k = 0; k < w; ++k ){
            reC = a + k * dx;
            std::complex<double> c ( reC, imC );
            std::complex<double> z_n ( 0, 0 );
            iteracio = 0;
            while ( std::abs ( z_n ) < 4 && iteracio < iter_lim ){
                z_n = z_n * z_n + c;
                ++iteracio;
            }
            SDL_SetRenderDrawColor(renderer, iteracio%255, (iteracio* ←
                iteracio)%255, 0, 0xff);
            SDL_RenderDrawPoint(renderer, k, j);
        }
        szazalek = ( double ) j / ( double ) h * 100.0;
    }
}
```

Ahhoz hogy forduljon például a következő flageket kell használni `g++ mandelcx.cpp -Wl,-subsystem,windows -lmingw32 -lSDL2main -lSDL2`

```
daProgramozod\Mandel>g++ mandelcx.cpp -o mandelcx.exe -Wl,-subsystem,windows -lmingw32 -lSDL2main -lSDL2
```

arg-ok nélkül futtatva default értékekkel a következő renderelődik. A programból úgy lehet kilépni, hogy a megnyílt ablakot default bezáró gombot nyomjuk például.



5.10. ábra. mandelcx

A fájl többi része SDL specifikus. Az egyetlen érdekes dolog, hogy a program egy while-ban fut és egy bool változóval kontrolláljuk a kilépést ezen ciklusból. A változó alapvetően true értékű, azonban, ha ablakbezárást érzékelünk, akkor ezt átállítjuk.

Event handling-gel nem játszottunk. A lehető legegyszerűbben valósítottuk meg, ugyanis csak a kilépésre kell figyelni. Az utazóban persze ezt majd kicsit át fogjuk alakítani.

```
1 // Uses SDL2, because of fun and the fact that png++ uses non standard ↩
   strerror_r
2 // For compilation use
3 // -Wl,-subsystem,windows -lmingw32 -lSDL2main -lSDL2
4 // KAWOOOSH
5
6 #include <iostream>
7 #include <complex>
8 #define SDL_MAIN_HANDLED
9 #include <SDL2/SDL.h>
10
11 class MandelDraw
12 {
13 public:
14     MandelDraw(int w,int h, int iter_lim,double a,double b,double c,double d)
15         : w(w),h(h),iter_lim(iter_lim),a(a),b(b),c(c),d(d)
16     { }
17
18     void render(SDL_Renderer *renderer)
```



```
19 {
20     int szazalek = 0; // unused
21     double dx = ( b - a ) / w;
22     double dy = ( d - c ) / h;
23     double reC, imC, reZ, imZ;
24     int iteracio = 0;
25     for ( int j = 0; j < h; ++j ){
26         imC = d - j * dy;
27         for ( int k = 0; k < w; ++k ){
28             reC = a + k * dx;
29             std::complex<double> c ( reC, imC );
30             std::complex<double> z_n ( 0, 0 );
31             iteracio = 0;
32             while ( std::abs ( z_n ) < 4 && iteracio < iter_lim ){
33                 z_n = z_n * z_n + c;
34                 ++iteracio;
35             }
36             SDL_SetRenderDrawColor(renderer, iteracio%255, (iteracio*iteracio) <=
                 %255, 0, 0xff);
37             SDL_RenderDrawPoint(renderer, k, j);
38         }
39         szazalek = ( double ) j / ( double ) h * 100.0;
40     }
41 }
42 private:
43     int w;
44     int h;
45     int iter_lim;
46     double a;
47     double b;
48     double c;
49     double d;
50 };
51
52 int main(int argc, char** argv)
53 {
54     SDL_SetMainReady(); // just for check
55     int szelesseg = 400;
56     int magassag = 300;
57     int iteraciosHatar = 255;
58     double a = -1.9;
59     double b = 0.7;
60     double c = -1.3;
61     double d = 1.3;
62     if ( argc == 1 ){
63         // Use defaults
64     }else if ( argc == 8 ){
65         szelesseg = atoi ( argv[1] );
66         magassag = atoi ( argv[2] );
67         iteraciosHatar = atoi ( argv[3] );
```

```
68     a = atof ( argv[4] );
69     b = atof ( argv[5] );
70     c = atof ( argv[6] );
71     d = atof ( argv[7] );
72 }else{
73     std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d" <-
        << std::endl;
74     return -1;
75 }
76 if(SDL_Init(SDL_INIT_VIDEO) != 0){
77     std::cout << "SDL_Init failed" << std::endl;
78     return -1;
79 }
80 SDL_Window *window = SDL_CreateWindow( "Mandel", SDL_WINDOWPOS_CENTERED, <-
    SDL_WINDOWPOS_CENTERED, szelesseg, magassag, SDL_WINDOW_SHOWN );
81 if( window == NULL )
82 {
83     std::cout << "Window could not be created! SDL_Error: " << SDL_GetError <-
        () << std::endl;
84     SDL_Quit();
85     return -1;
86 }
87 SDL_Renderer* renderer = SDL_CreateRenderer(window,1,0);
88 if( renderer == NULL )
89 {
90     std::stringstream ss;
91     std::cout << "Renderer could not be created! SDL_Error: " << <-
        SDL_GetError() << std::endl;
92     SDL_DestroyWindow(window);
93     SDL_Quit();
94     return -1;
95 }
96 SDL_SetWindowPosition(window, SDL_WINDOWPOS_CENTERED, <-
    SDL_WINDOWPOS_CENTERED);
97
98 MandelDraw mdrw(szelesseg,magassag,iteraciosHatar,a,b,c,d);
99 mdrw.render(renderer);
100 SDL_RenderPresent(renderer);
101
102 const int FPS = 60;
103 const int frame_delay = 1000/FPS;
104 Uint32 frame_start;
105 int frame_time;
106 SDL_Event evt;
107 bool should_run = true;
108 while(should_run){
109     frame_start = SDL_GetTicks();
110     SDL_PollEvent(&evt);
111     if(evt.type==SDL_QUIT)should_run = false;
112     frame_time = SDL_GetTicks() - frame_start;
```

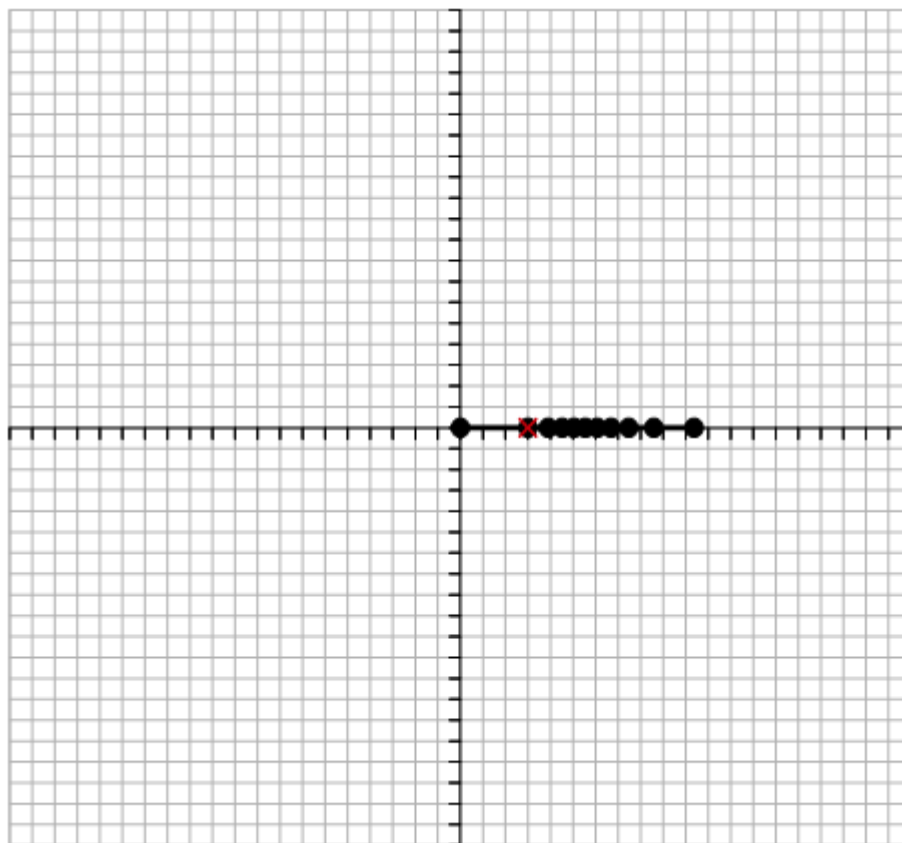
```
113     if (frame_delay > frame_time) {  
114         SDL_Delay(frame_delay-frame_time);  
115     }  
116 }  
117 SDL_DestroyRenderer(renderer);  
118 SDL_DestroyWindow(window);  
119 SDL_Quit();  
120 return 0;  
121 }
```

5.3. Biomorfok

Az előző mandelbrot feladat mintájára, csak a számítási algoritmust illetve a formális paramétereket, és azok technikai kezelését kell módosítani.

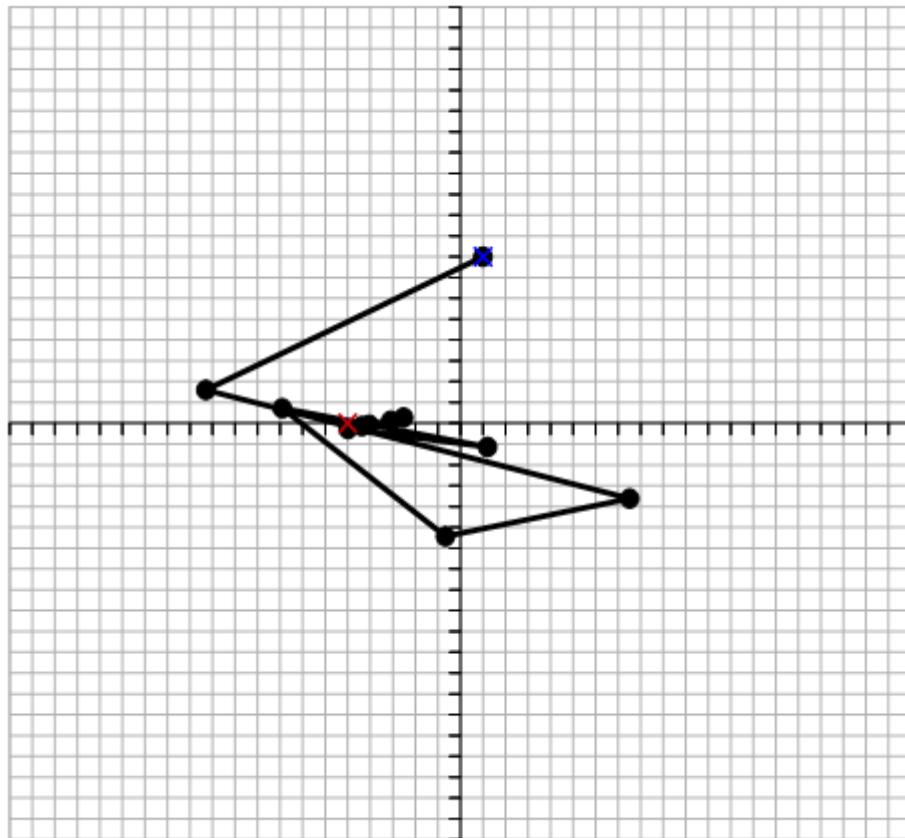
De mi is az a biomorph? Nos, emlékezzünk, hogy mandelbrot esetben kézzel 0,0-ból lépegettünk egy az alábbi diagramon meghatározott piros pont (re és im koordináták miatt ugye ez egy komplex szám)-tot adogattunk hozzá miután négyzetre emeltük.

Azaz piros ponttal körbejártuk a képernyő által jelentett komplex számsíkot, és minden pontba beírtuk mennyi lépés után szállt el a számítás (vagy kerültünk egy adott origó középpontú adott R sugarú körön kívülre.)



5.11. ábra. mandel step

Biomoprh-nál egyetlen dolog történik: nem 0,0-ból indulunk hanem egy magadott pontból. Alábbi diagramon kékkel jelöltem egy példát.



5.12. ábra. biomorph iter

Emiatt ugye szükség lesz két új paraméterre: a kezdő pont valós és képzetes koordináta párosára.

A másik paraméter egy R szám. Ez azért kell, mert az iterációs terminálásának kondícióját megváltoztattuk: Akkor lépünk ki az iterációból, ha vagy valós, vagy képzetes tengelyen R -nél távolabb kerülünk az origótól.

Az eredeti kódot kicsit tehát módosítani kell!

`std::pow` az eredetiben köbös, viszont mi nyuszit akarunk rajzolni, ezért négyzetes kell.

A másik változtatás, hogy akkor lépünk ki a ciklusból, ha vagy elértük `iter_limit`et vagy a számított komplex szám távolabb van origótól, mint R , azaz abszolút értékét vesszük. Ez gyakorlatilag a vektor hossza lenne, ha pl. valami 2d-s grafikával dolgoznánk.

```
std::complex<double> cc ( reC, imC );
double dx = ( xmax - xmin ) / w;
double dy = ( ymax - ymin ) / h;
for ( int y = 0; y < h; ++y ){
    for ( int x = 0; x < w; ++x ){
        double reZ = xmin + x * dx;
        double imZ = ymax - y * dy;
        std::complex<double> z_n ( reZ, imZ );
        int iteracio = 0;
```

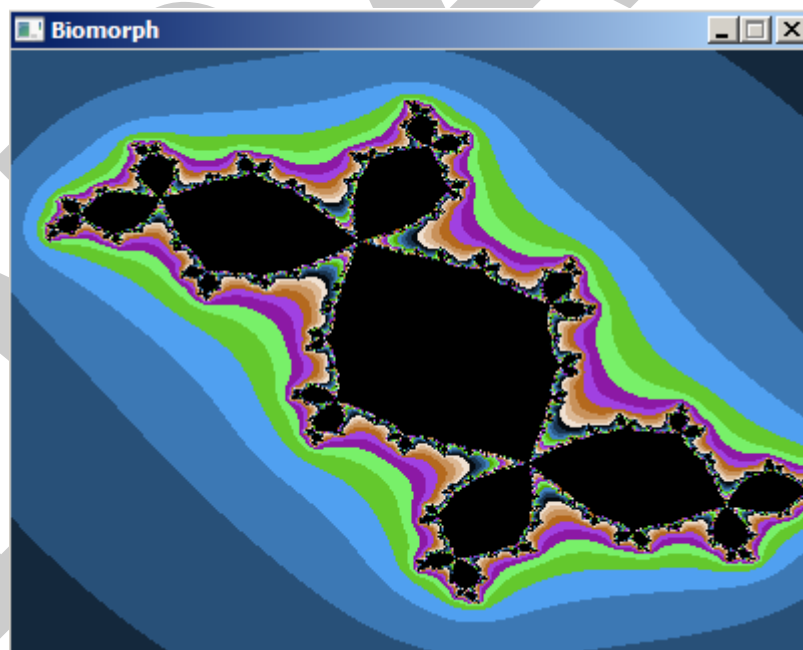
```
for (int i=0; i < iter_lim; ++i){
    z_n = std::pow(z_n, 2) + cc;
    if( std::abs(z_n) > R){
        iteracio = i;
        break;
    }
}
SDL_SetRenderDrawColor(renderer, (iteracio * 20)%255, ( ←
    iteracio* 40)%255, (iteracio* 60)%255 , 0xff);
SDL_RenderDrawPoint(renderer, x,y);
}
```

Compile esetén g++-nak a szokásos sdl által megkövetelt flag-eket küldjük, azaz g++ biomorph.cpp -Wl,-subsystem,windows -lmingw32 -lSDL2main -lSDL2

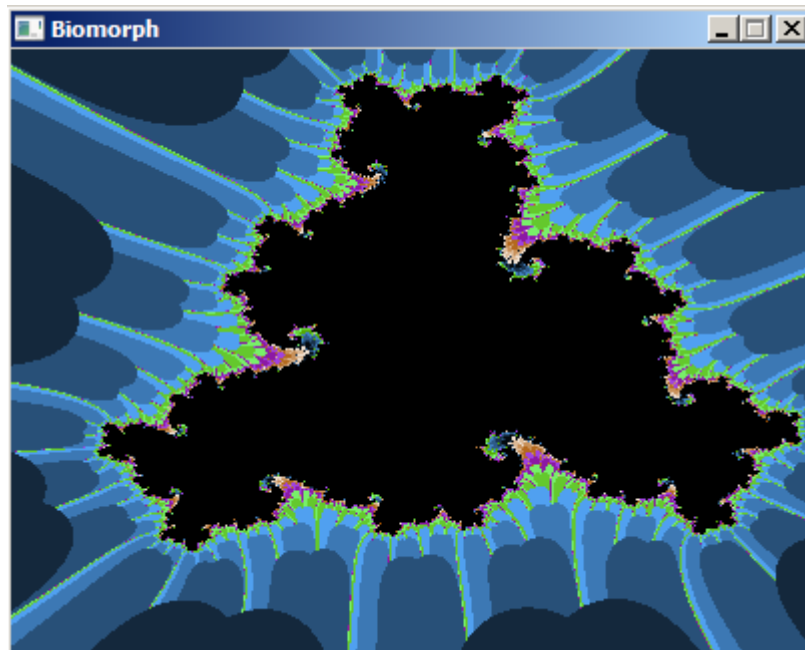
```
daProgramozod\Mandel>g++ biomorph.cpp -o biomorph.exe -Wl,-subsystem,windows -lmingw32 -lSDL2main -lSDL2
```

5.13. ábra. biomorph comp

Ha mindent jól csináltunk akkor, egy biomorph [nyuszival](#) kell találkozunk.



5.14. ábra. biomorphnyuszi



5.15. ábra. biomorph

```
1 // Uses SDL2, because of fun and the fact that png++ uses non standard ↵
   strerror_r
2 // For compilation use
3 // -Wl,-subsystem,windows -lmingw32 -lSDL2main -lSDL2
4 // KAWOOOSH
5
6 #include <iostream>
7 #include <complex>
8 #define SDL_MAIN_HANDLED
9 #include <SDL2/SDL.h>
10
11 class BiomorphDraw
12 {
13 public:
14     BiomorphDraw(int w,
15                 int h,
16                 int iter_lim,
17                 double xmin,
18                 double xmax,
19                 double ymin,
20                 double ymax,
21                 double reC,
22                 double imC,
23                 double R)
24
25     : w(w), h(h), iter_lim(iter_lim), xmin(xmin), xmax(xmax), ymin(ymin), ymax(ymax) ↵
       , reC(reC), imC(imC), R(R)
26     { }
```

```
27
28 void render(SDL_Renderer *renderer)
29 {
30     std::complex<double> cc ( reC, imC );
31     int szazalek = 0; // unused
32     double dx = ( xmax - xmin ) / w;
33     double dy = ( ymax - ymin ) / h;
34     for ( int y = 0; y < h; ++y ){
35         for ( int x = 0; x < w; ++x ){
36             double reZ = xmin + x * dx;
37             double imZ = ymax - y * dy;
38             std::complex<double> z_n ( reZ, imZ );
39             int iteracio = 0;
40             for (int i=0; i < iter_lim; ++i){
41                 z_n = std::pow(z_n, 2) + cc;
42                 if( std::abs(z_n) > R){
43                     iteracio = i;
44                     break;
45                 }
46             }
47             SDL_SetRenderDrawColor(renderer, (iteracio * 20)%255, (iteracio * 40)%255, (iteracio * 60)%255 , 0xff);
48             SDL_RenderDrawPoint(renderer, x,y);
49         }
50     }
51 }
52 private:
53     int w;
54     int h;
55     int iter_lim;
56     double xmin;
57     double xmax;
58     double ymin;
59     double ymax;
60     double reC;
61     double imC;
62     double R;
63 };
64
65 int main(int argc, char** argv)
66 {
67     SDL_SetMainReady(); // just for check
68     int szelesseg = 400;
69     int magassag = 300;
70     int iteraciosHatar = 255;
71     double xmin = -1.4;
72     double xmax = 1.2;
73     double ymin = -1.3;
74     double ymax = 1.3;
75     //double reC = .285, imC = 0;
```



```
76 double reC = -0.123, imC = 0.745;
77 double R = 10.0;
78 if ( argc == 1 ){
79     // Use defaults
80 }else if ( argc == 11 ){
81     szelesseg = atoi ( argv[1] );
82     magassag =  atoi ( argv[2] );
83     iteraciosHatar =  atoi ( argv[3] );
84     xmin = atof ( argv[4] );
85     xmax = atof ( argv[5] );
86     ymin = atof ( argv[6] );
87     ymax = atof ( argv[7] );
88     reC = atof ( argv[8] );
89     imC = atof ( argv[9] );
90     R = atof ( argv[10] );
91 }else{
92     std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag ↵
93         iteraciosHatar xmin xmax ymin ymax reC imC R" << std::endl;
94     return -1;
95 }
96 if(SDL_Init(SDL_INIT_VIDEO) != 0){
97     std::cout << "SDL_Init failed" << std::endl;
98     return -1;
99 }
100 SDL_Window *window = SDL_CreateWindow( "Biomorph", SDL_WINDOWPOS_CENTERED ↵
101     , SDL_WINDOWPOS_CENTERED, szelesseg, magassag, SDL_WINDOW_SHOWN );
102 if( window == NULL )
103 {
104     std::cout << "Window could not be created! SDL_Error: " << SDL_GetError ↵
105         () << std::endl;
106     SDL_Quit();
107     return -1;
108 }
109 SDL_Renderer* renderer = SDL_CreateRenderer(window,1,0);
110 if( renderer == NULL )
111 {
112     std::stringstream ss;
113     std::cout << "Renderer could not be created! SDL_Error: " << ↵
114         SDL_GetError() << std::endl;
115     SDL_DestroyWindow(window);
116     SDL_Quit();
117     return -1;
118 }
119 SDL_SetWindowPosition(window, SDL_WINDOWPOS_CENTERED, ↵
120     SDL_WINDOWPOS_CENTERED);
121
122 BiomorphDraw bdrw(szelesseg,magassag,iteraciosHatar,xmin,xmax,ymin,ymax, ↵
123     reC,imC,R);
124 bdrw.render(renderer);
125 SDL_RenderPresent(renderer);
```

```
120
121  const int FPS = 60;
122  const int frame_delay = 1000/FPS;
123  Uint32 frame_start;
124  int frame_time;
125  SDL_Event evt;
126  bool should_run = true;
127  while(should_run){
128      frame_start = SDL_GetTicks();
129      SDL_PollEvent(&evt);
130      if(evt.type==SDL_QUIT) should_run = false;
131      frame_time = SDL_GetTicks() - frame_start;
132      if(frame_delay > frame_time){
133          SDL_Delay(frame_delay-frame_time);
134      }
135  }
136  SDL_DestroyRenderer(renderer);
137  SDL_DestroyWindow(window);
138  SDL_Quit();
139  return 0;
140 }
141
```

A biomorfos 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:

5.4. Mandelbrot nagyító és utazó C++ nyelven

SDL2. C++ wrapperrel SDL C API körül. A signal slot barebones verziója [Simple Signal](#)

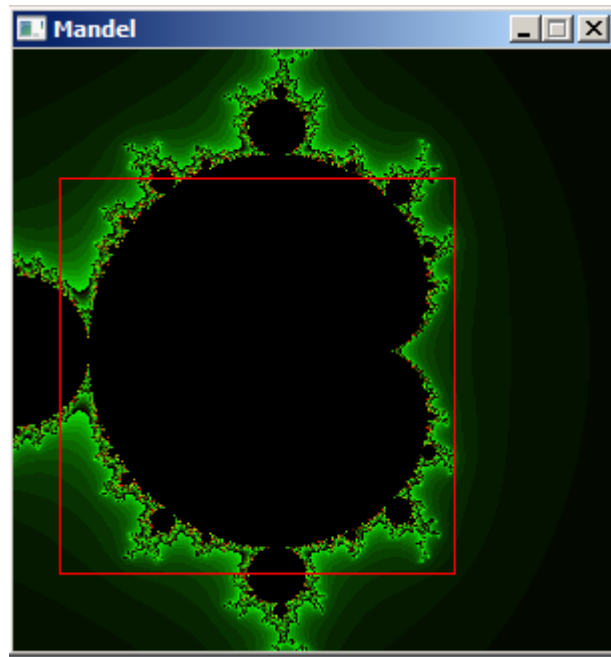
[mandelbrot utazó](#)

[wrappers](#)

[old bad c](#)

Argumentumok nélkül futtatható. Debug üzenetekért kimenetet irányítsuk log file-ba stb. Debug log-hoz megfelelő preproc #define-t uncommentelni...

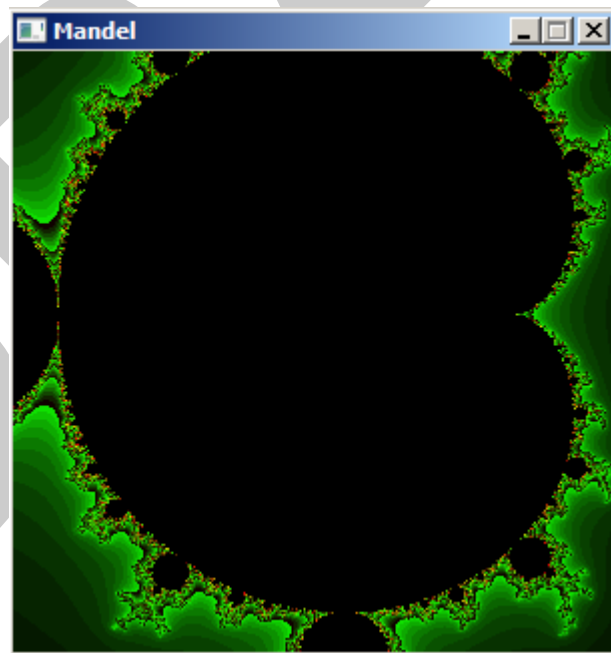
Zoom-olást bal klikkel indítja az ember. Nyomva tartva a bal gombot téglalapot rajzol a program.



5.16. ábra. zoom rect

A téglalap húzgálásra DIREKT TARTJA a képernyő arányt.

Bal egérgomb felegedéskor a kijelölt téglalap alapján történik a nagyítás.



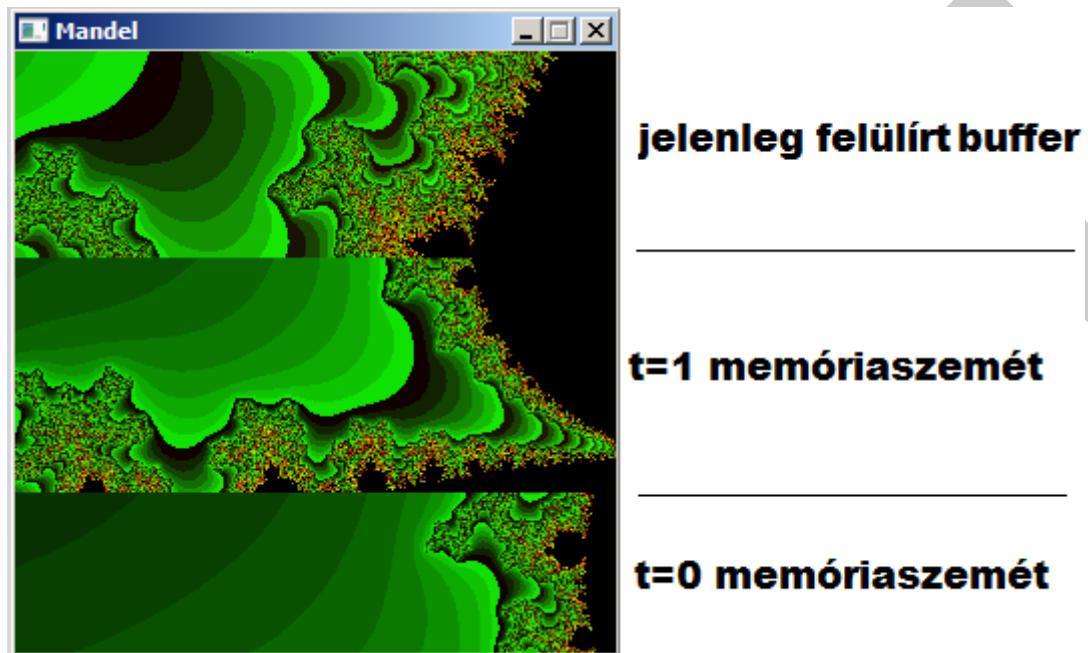
5.17. ábra. zoom

A nagyítás konkrét frame-jében nem a teljes képet számoljuk. Ehelyett dinamikus számú sornyi pixelt számolunk ki mindig. A lényeg hogy 60 FPS a cél, ha túl sok időt vesz igénybe adott frame-en a dolog,

akkor abbahagyjuk a számítást, és majd következő frame-ben folytatjuk.

Ha épp számolódik a background, és újra nagyítunk DIREKT nem tisztítjuk a buffert.

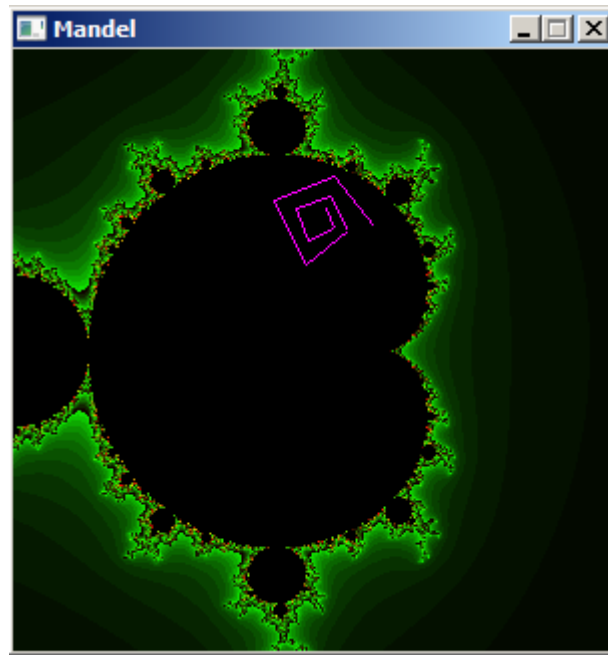
User szinten emiatt egy érdekes egymásra lapolódás lesz megfigyelhető. Alábbi ábrán egy futó számítás közben újra nagyítottunk. Az előző számítási értékek természetesen egyből eltűnnek, ha a jelenleg futó számítás odaér a buffer írásban.



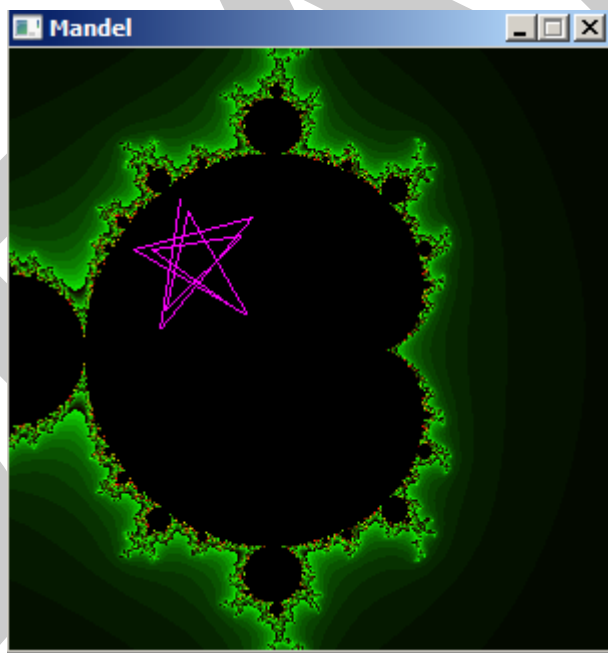
5.18. ábra. zoom quick succession overlap

A fentiekből következik, hogy a nagyítás DIREKT "lassul" és "gyorsul" olykor, nem hiba. Ha az épp most számolandó sorok gyorsan elszállnak, akkor gyorsan végez velük azaz super gyorsan végezni fog velük. Ellenkező esetben ha a neki kiosztott sorok nagyon sokáig benn maradnak az R sugarú Q-beli körben, akkor kevés ilyen sort tud kiszámolni egy frame alatt, ezért úgy fog tűnni. mintha lassulna.

Jobb gombbal kapcsoljuk be a tracert. Ez egy predefined iter számig végig számítja a komplex számokat. Ehhez c komplex számot(amit mindig hozzáadunk) a user egér pozíciója alapján veszi.



5.19. ábra. tracer 0



5.20. ábra. tracer 1

Egér középső gombot vagy görgőt megnyomva visszaugrik default zoomba. Az átláthatóság és rövidség rovására ment, de próbáltam amennyire lehet elszeparálni a különböző alrendszereket.

```
1 // [COMPILE]
2 // -Wl,-subsystem,windows -lmingw32 -lSDL2main -lSDL2
3 #include "sdl_wrapper.hpp"
```

```
4  #include <complex>
5  #include <cstdlib>
6
7  constexpr double DEFAULT_RE0 = -1.0;
8  constexpr double DEFAULT_RE1 = 1.0;
9  constexpr double DEFAULT_IM0 = -1.0;
10 constexpr double DEFAULT_IM1 = 1.0;
11 constexpr int ITER_LIMIT = 255;
12 constexpr int FPS_GOAL = 60;
13 constexpr int BG_CALC_MAX_ROWS_PER_FRAME = 300;
14 constexpr int SMALLEST_VALID_ZOOM_BOX_SIZE = 10;
15 constexpr int TRACER_COUNT = 10;
16
17 const Color white(255,255,255);
18 const Color black(0,0,0);
19 const Color red(255,0,0);
20 const Color green(0,255,0);
21 const Color blue(0,0,255);
22
23 inline double trf_coords(double cur, double a_lo, double a_hi, double b_lo, ←
    double b_hi)
24 {
25     return static_cast<double>(cur-a_lo)/(a_hi-a_lo)*(b_hi-b_lo)+b_lo;
26 }
27
28 class ComplexIter
29 {
30 public:
31     ComplexIter(double z_re, double z_im, double c_re, double c_im):z_n(z_re, ←
        z_im), c(c_re,c_im){};
32
33     void set_c( double c_re, double c_im)
34     {
35         z_n = std::complex<double>(0,0);
36         c=std::complex<double>(c_re,c_im);
37     }
38
39     ComplexIter& next()
40     {
41         z_n = z_n * z_n + c;
42         return *this;
43     }
44
45     double re(){return std::real(z_n);}
46
47     double im(){ return std::imag(z_n);}
48
49     double abs() { return std::abs(z_n);};
50 private:
51     std::complex<double> c;
```

```
52     std::complex<double> z_n;
53 };
54
55 struct ScreenDimensions
56 {
57     ScreenDimensions(int width, int height) : width(width), height(height){}
58     int width;
59     int height;
60 };
61
62 struct ComplexDimensions
63 {
64     ComplexDimensions(
65         int re0 = DEFAULT_RE0,
66         int im0=DEFAULT_IM0,
67         int re1=DEFAULT_RE1,
68         int im1=DEFAULT_IM1,
69         int iter_lim=ITER_LIMIT) :
70         re0(re0), im0(im0), re1(re1), im1(im1), iter_lim(iter_lim){}
71     double re0;
72     double im0;
73     double re1;
74     double im1;
75     int iter_lim;
76 };
77
78 struct Selbox
79 {
80 public:
81     Selbox() :active(false){}
82
83     bool active;
84
85     Rect rect;
86 };
87
88 struct Tracer
89 {
90 public:
91     Tracer(int tracer_count=TRACER_COUNT) : active(false),tracer_count( ←
92         tracer_count){}
93
94     int tracer_count;
95
96     bool active;
97
98     double c_re;
99
100     double c_im;
```

```
101     std::vector<Vec2D> points;
102 };
103
104 struct StaticTex
105 {
106     StaticTex(Renderer& renderer, Uint32 pixel_format, int w, int h) :
107         w(w), h(h), texture(renderer, pixel_format, SDL_TEXTUREACCESS_STATIC, w, h) {}
108     int w;
109     int h;
110     Texture texture;
111 };
112
113 class PassiveDataRepo
114 {
115
116 public:
117     PassiveDataRepo(Renderer& renderer, int w, int h, Uint32 bg_format):
118         screen_dims(w,h), cx_dims(), tracer(), bg_tex(renderer, bg_format, w, h)
119     {}
120
121     ~PassiveDataRepo() {}
122
123     PassiveDataRepo(const PassiveDataRepo&) = delete;
124
125     PassiveDataRepo& operator=(const PassiveDataRepo&) = delete;
126
127     PassiveDataRepo(PassiveDataRepo&&) =delete;
128
129     PassiveDataRepo& operator=(PassiveDataRepo&&) = delete;
130
131 public:
132
133     ScreenDimensions& get_screen_dims(){ return screen_dims;}
134
135     ComplexDimensions& get_cx_dims(){ return cx_dims; }
136
137     Tracer& get_tracer(){ return tracer;}
138
139     Selbox& get_selbox() { return selbox;}
140
141     StaticTex& get_bg_tex(){ return bg_tex;}
142 private:
143     ScreenDimensions screen_dims;
144     ComplexDimensions cx_dims;
145     Tracer tracer;
146     Selbox selbox;
147     StaticTex bg_tex;
148 };
149
```



```
150 struct BoxSelectionEvent
151 {
152     Rect rect;
153 };
154
155 struct RecomputeBackgroundEvent
156 {
157     RecomputeBackgroundEvent() : compute_whole_in_one_frame(false) {}
158     bool compute_whole_in_one_frame; /* currently unused */
159 };
160
161 class EventHandlerImpl : public EventHandler
162 {
163 public:
164     EventHandlerImpl() : EventHandler() {}
165
166     virtual ~EventHandlerImpl() {}
167
168     EventHandlerImpl(const EventHandlerImpl&) = delete;
169
170     EventHandlerImpl& operator=(const EventHandlerImpl&) = delete;
171
172     EventHandlerImpl(EventHandlerImpl&&) =delete;
173
174     EventHandlerImpl& operator=(EventHandlerImpl&&) = delete;
175
176 public:
177     Signal<BoxSelectionEvent> sig_boxselection;
178
179     Signal<RecomputeBackgroundEvent> sig_recompute_bg;
180 };
181
182
183 class AppService{
184 public:
185
186     AppService() {}
187
188     virtual ~AppService() {}
189
190     AppService(const AppService&) = delete;
191
192     AppService& operator=(const AppService&) = delete;
193
194     AppService(AppService&&) =delete;
195
196     AppService& operator=(AppService&&) = delete;
197
198 public:
199
```

```
200     virtual void configure(EventHandlerImpl& evts) = 0;
201
202     virtual void update(const FPSClock& clock, PassiveDataRepo& repo, ↵
        EventHandlerImpl& evts) = 0;
203 };
204
205
206 class SelectionService : public AppService
207 {
208 public:
209
210     SelectionService() : AppService(), active(false){}
211
212     ~SelectionService() = default;
213
214     SelectionService(const SelectionService&) = delete;
215
216     SelectionService& operator=(const SelectionService&) = delete;
217
218     SelectionService(SelectionService&&) =delete;
219
220     SelectionService& operator=(SelectionService&&) = delete;
221
222 public:
223
224     virtual void configure(EventHandlerImpl& evts)
225     {
226         evts.sig_mousebutton_up.connect(Simple::slot (*this, &SelectionService ↵
            ::on_mb_up));
227         evts.sig_mousebutton_down.connect(Simple::slot (*this, & ↵
            SelectionService::on_mb_down));
228         evts.sig_mousemotion.connect(Simple::slot (*this, &SelectionService:: ↵
            on_mouse_moved));
229     }
230
231     virtual void update(const FPSClock& clock, PassiveDataRepo& repo, ↵
        EventHandlerImpl& evts)
232     {
233         Selbox& selbox = repo.get_selbox();
234         if(active){
235             ScreenDimensions& sd = repo.get_screen_dims();
236             double w_mult = static_cast<double>(std::abs(x1-x0))/sd.width;
237             double h_mult = static_cast<double>(std::abs(y1-y0))/sd.height;
238             double mult = std::max(w_mult,h_mult);
239             x1=(x1<x0) ? x0-mult*sd.width : x0+mult*sd.width;
240             y1=(y1<y0) ? y0-mult*sd.height : y0+mult*sd.height;
241             if((x1<0||x1>sd.width) || (y1<0||y1>sd.height)){
242                 mult = std::min(w_mult,h_mult);
243                 x1=(x1<x0) ? x0-mult*sd.width : x0+mult*sd.width;
244                 y1=(y1<y0) ? y0-mult*sd.height : y0+mult*sd.height;
```

```
245     }
246     Rect newr = Rect::From_Corner_Points(x0,y0,x1,y1);
247     if(selbox.active){
248         if(selbox.rect!=newr){
249             selbox.rect = std::move(newr);
250         }
251     }else{
252         selbox.rect = std::move(newr);
253         selbox.active = true;
254     }
255 }else{
256     if(selbox.active){
257         selbox.active = false;
258         if(
259             selbox.rect.w>SMALLEST_VALID_ZOOM_BOX_SIZE &&
260             selbox.rect.h>SMALLEST_VALID_ZOOM_BOX_SIZE)
261         {
262             BoxSelectionEvent e{.rect=selbox.rect};
263             evts.sig_boxselection.emit(e);
264         }
265     }
266 }
267 }
268 }
269
270 void on_mouse_moved(const SDL_MouseMotionEvent& evt)
271 {
272     if(active){
273         x1=evt.x;
274         y1=evt.y;
275     }
276 }
277
278 void on_mb_up(const SDL_MouseButtonEvent& evt)
279 {
280     if(evt.button==SDL_BUTTON_LEFT){
281         if(active){
282             active=false;
283         }
284         return;
285     }
286 }
287
288 void on_mb_down(const SDL_MouseButtonEvent& evt)
289 {
290     if(evt.button==SDL_BUTTON_LEFT){
291         if(!active){
292             x0 = x1 = evt.x;
293             y0 = y1 = evt.y;
294             active=true;
```

```
295     }
296     return;
297 }
298 }
299 private:
300
301 private:
302     bool active;
303     int x0;
304     int y0;
305     int x1;
306     int y1;
307 };
308
309 class TracerService : public AppService
310 {
311 public:
312     TracerService() : AppService(), active(false){}
313
314     ~TracerService() = default;
315
316     TracerService(const TracerService&) = delete;
317
318     TracerService& operator=(const TracerService&) = delete;
319
320     TracerService(TracerService&&) =delete;
321
322     TracerService& operator=(TracerService&&) = delete;
323
324 public:
325
326     virtual void configure(EventHandlerImpl& evts)
327     {
328         evts.sig_mousebutton_up.connect(Simple::slot (*this, &TracerService:: ←
329             on_mb_up));
330     }
331
332     virtual void update(const FPSClock& clock, PassiveDataRepo& repo, ←
333         EventHandlerImpl& evts)
334     {
335         Tracer& tracer = repo.get_tracer();
336         if(active){
337             if(!tracer.active){
338                 tracer.active=true;
339                 recalculate(tracer, repo.get_screen_dims(),repo.get_cx_dims());
340             }
341         }else{
342             tracer.active=false;
343         }
344     }
345 }
```

```
343
344 void on_mb_up(const SDL_MouseButtonEvent& evt)
345 {
346     if(evt.button==SDL_BUTTON_RIGHT){
347         if(active)return;
348         screen_pick_x = evt.x;
349         screen_pick_y = evt.y;
350         active=true;
351         return;
352     }else if(evt.button==SDL_BUTTON_LEFT){
353         if(active)active=false;
354     }
355 }
356
357 private:
358 void recalculate(Tracer& tracer,ScreenDimensions& screen_dims, ↵
    ComplexDimensions& cx_dims)
359 {
360     tracer.points.clear();
361     const double c_re = trf_coords(screen_pick_x,0,screen_dims.width, ↵
        cx_dims.re0,cx_dims.re1);
362     const double c_im = trf_coords(screen_pick_y,0,screen_dims.height, ↵
        cx_dims.im0,cx_dims.im1);
363     ComplexIter it(0,0,c_re,c_im);
364     int x,y;
365     for(int i = 0; i<tracer.tracer_count; ++i)
366     {
367         it.next();
368         x = static_cast<int>(trf_coords(it.re(),cx_dims.re0,cx_dims.re1,0, ↵
            screen_dims.width));
369         y = static_cast<int>(trf_coords(it.im(),cx_dims.im0,cx_dims.im1,0, ↵
            screen_dims.height));
370         tracer.points.push_back(std::move(Vec2D(x,y)));
371     }
372 }
373 private:
374
375 bool active;
376
377 int screen_pick_x;
378
379 int screen_pick_y;
380 };
381
382
383 class ComplexBoundsService : public AppService
384 {
385 public:
386     ComplexBoundsService() : AppService(), recalc_limits(false)
387     {
```

```
388     }
389
390     virtual ~ComplexBoundsService() {}
391
392     ComplexBoundsService(const ComplexBoundsService&) = delete;
393
394     ComplexBoundsService& operator=(const ComplexBoundsService&) = delete;
395
396     ComplexBoundsService(ComplexBoundsService&&) = delete;
397
398     ComplexBoundsService& operator=(ComplexBoundsService&&) = delete;
399
400 public:
401
402     virtual void configure(EventHandlerImpl& evts) override{
403         evts.sig_boxselection.connect(Simple::slot (*this, & ↵
404             ComplexBoundsService::on_boxsel_ended));
405         evts.sig_mousebutton_down.connect(Simple::slot (*this, & ↵
406             ComplexBoundsService::on_mb_down));
407     }
408
409     virtual void update(const FPSClock& clock, PassiveDataRepo& repo, ↵
410         EventHandlerImpl& evts) override
411     {
412         if(recalc_limits){
413             recalc_cx_dims(repo.get_screen_dims(), repo.get_cx_dims());
414             evts.sig_recompute_bg.emit(RecomputeBackgroundEvent());
415             recalc_limits = false;
416         }
417     }
418
419     void on_mb_down(const SDL_MouseButtonEvent& e)
420     {
421         if(e.button!=SDL_BUTTON_MIDDLE) return;
422         x0=-1;
423         y0=-1;
424         x1=-1;
425         y1=-1;
426         recalc_limits = true;
427     }
428
429     void on_boxsel_ended(const BoxSelectionEvent& e)
430     {
431         x0=e.rect.x;
432         y0=e.rect.y;
433         x1=e.rect.x+e.rect.w;
434         y1=e.rect.y+e.rect.h;
435         recalc_limits = true;
436     }
437
438 private:
```

```
435
436 void recalc_cx_dims(const ScreenDimensions& scr, ComplexDimensions& ↵
    cx_dims)
437 {
438     if(x0<0||y0<0||x1<0||y1<0){
439         cx_dims.re0=DEFAULT_RE0;
440         cx_dims.re1=DEFAULT_RE1;
441         cx_dims.im0=DEFAULT_IM0;
442         cx_dims.im1=DEFAULT_IM1;
443     }else{
444         double re0 = trf_coords(std::min(x0,x1),0,scr.width,cx_dims.re0, ↵
            cx_dims.re1);
445         double re1 = trf_coords(std::max(x0,x1),0,scr.width,cx_dims.re0, ↵
            cx_dims.re1);
446         double im0 = trf_coords(std::min(y0,y1),0,scr.height,cx_dims.im0, ↵
            cx_dims.im1);
447         double im1 = trf_coords(std::max(y0,y1),0,scr.height,cx_dims.im0, ↵
            cx_dims.im1);
448         cx_dims.re0=re0;
449         cx_dims.re1=re1;
450         cx_dims.im0=im0;
451         cx_dims.im1=im1;
452     }
453 }
454 private:
455     bool recalc_limits;
456     int x0;
457     int y0;
458     int x1;
459     int y1;
460 };
461
462 class MandelUpdaterService : public AppService
463 {
464 public:
465     MandelUpdaterService(const StaticTex& bg_tex) :
466     AppService(), recompute_requested(false), bg_calc_running(false), ↵
        bg_calc_y(0)
467     {
468         argb_buffer = (uint32_t*) malloc(bg_tex.w*bg_tex.h*sizeof(uint32_t));
469         if( argb_buffer == nullptr ){throw std::runtime_error("Failed to ↵
            allocate bg buffer");}
470     }
471
472     virtual ~MandelUpdaterService(){if(argb_buffer!=nullptr)free(argb_buffer) ↵
        ;}
473
474     MandelUpdaterService(const MandelUpdaterService&) = delete;
475
476     MandelUpdaterService& operator=(const MandelUpdaterService&) = delete;
```

```
477 MandelUpdaterService(MandelUpdaterService&&) = delete;
478
479 MandelUpdaterService& operator=(MandelUpdaterService&&) = delete;
480
481
482 public:
483
484 virtual void configure(EventHandlerImpl& evts) override{
485     evts.sig_recompute_bg.connect(Simple::slot (*this, & ↵
486         MandelUpdaterService::on_recompute_request));
487 }
488
489 void on_recompute_request(const RecomputeBackgroundEvent& e)
490 {
491     recompute_requested = true;
492 }
493
494 virtual void update(const FPSClock& clock, PassiveDataRepo& repo, ↵
495     EventHandlerImpl& evts)override
496 {
497     if(recompute_requested){
498         bg_calc_running = true;
499         bg_calc_y = 0;
500         recompute_requested=false;
501     }
502     if(!bg_calc_running){return;};
503     compute_bg(clock, repo.get_screen_dims(), repo.get_cx_dims(), repo. ↵
504         get_bg_tex());
505 }
506
507 private:
508
509 void compute_bg(const FPSClock& clock, const ScreenDimensions& scr, const ↵
510     ComplexDimensions& cx_dims, StaticTex& bg_tex)
511 {
512     double d_re = ( cx_dims.re1 - cx_dims.re0 ) / bg_tex.w;
513     double d_im = ( cx_dims.im1 - cx_dims.im0 ) / bg_tex.h;
514     double c_re,c_im;
515     int iter;
516     const int limit = std::min(bg_tex.h,bg_calc_y+ ↵
517         BG_CALC_MAX_ROWS_PER_FRAME);
518     ComplexIter cit(0,0,0,0);
519     int y = bg_calc_y;
520     PixelFormat format(bg_tex.texture);
521     int head = y*bg_tex.w;
522     while(y<limit && clock.remaining()>0){
523         c_im = cx_dims.im0 + y * d_im;
524         for ( int x = 0; x < bg_tex.w; ++x ){
525             c_re = cx_dims.re0 + x * d_re;
526             cit.set_c(c_re,c_im);
```



```
522         iter = 0;
523         while ( cit.abs() < 4 && iter < cx_dims.iter_lim ){
524             cit.next();
525             ++iter;
526         }
527         argb_buffer[ y*bg_tex.w+x] = format.map_rgba(iter%255,(iter*iter) ←
            %255,0,0xff);
528     }
529     y++;
530 }
531 if(head!=y*bg_tex.w){
532     Rect subarea(0,bg_calc_y,bg_tex.w,y-bg_calc_y);
533     bg_tex.texture.update_subarea(subarea,(argb_buffer+head),bg_tex.w*4);
534 }
535 bg_calc_y = y;
536 if(bg_calc_y>=bg_tex.h)bg_calc_running=false;
537 }
538
539 private:
540     uint32_t* argb_buffer;
541     bool recompute_requested;
542     bool bg_calc_running;
543     int bg_calc_y;
544     int x0;
545     int y0;
546     int x1;
547     int y1;
548 };
549
550
551 class ScreenRendererService : public AppService
552 {
553 public:
554     ScreenRendererService(Renderer& renderer) : AppService(), renderer( ←
        renderer)
555     { }
556
557     virtual ~ScreenRendererService(){}
558
559     ScreenRendererService(const ScreenRendererService&) = delete;
560
561     ScreenRendererService& operator=(const ScreenRendererService&) = delete;
562
563     ScreenRendererService(ScreenRendererService&&) = delete;
564
565     ScreenRendererService& operator=(ScreenRendererService&&) = delete;
566
567 public:
568
569     virtual void configure(EventHandlerImpl& evts) override{}
```

```
570
571 virtual void update(const FPSClock& clock, PassiveDataRepo& repo, ←
    EventHandlerImpl& evts)override
572 {
573     renderer.copy_fulltex_to_fullscreen(repo.get_bg_tex().texture);
574     Tracer& tracer = repo.get_tracer();
575     if(tracer.active){
576         renderer.set_color(red).draw_lines(tracer.points);
577     }
578     Selbox& selbox = repo.get_selbox();
579     if(selbox.active){
580         renderer.set_color(green).draw_rect(selbox.rect);
581     }
582     renderer.present();
583 }
584 private:
585
586     Renderer& renderer;
587 };
588
589
590 class App
591 {
592
593 public:
594     App(const std::string& title, int width, int height) :
595         window(title,width,height),
596         renderer(window),
597         clock(FPS_GOAL),
598         repo(renderer,width,height,SDL_PIXELFORMAT_ARGB8888),
599         srv_sel(),
600         srv_tracer(),
601         srv_mandel(repo.get_bg_tex()),
602         srv_screen_renderer(renderer),
603         running(false)
604     {
605
606     }
607
608     virtual ~App()
609     {
610
611     }
612
613     App(const App&) = delete;
614
615     App& operator=(const App&) = delete;
616
617     App( App&&) = delete;
618
```

```
619 App& operator=( App&&) = delete;
620
621 public:
622
623 void configure()
624 {
625     evts.sig_quit.connect(Simple::slot (*this, &App::stop));
626     srv_sel.configure(evts);
627     srv_tracer.configure(evts);
628     srv_cx_bounds.configure(evts);
629     srv_mandel.configure(evts);
630     srv_screen_renderer.configure(evts);
631 }
632
633 void run()
634 {
635     if(running) return;
636     running=true;
637     const Uint32 FPS = 60;
638     const Uint32 frame_delay = 1000/FPS;
639     Uint32 frame_time;
640     clock.restart();
641     evts.sig_recompute_bg.emit(RecomputeBackgroundEvent());
642     while(running){
643         clock.restart();
644         while(evts.poll()){ }
645         update();
646         frame_time = clock.restart();
647         if(frame_delay>frame_time)SDL_Delay(frame_delay-frame_time);
648     }
649 }
650
651 void stop(const SDL_QuitEvent& e){running = false;}
652
653 private:
654 void update()
655 {
656     srv_sel.update(clock, repo, evts);
657     srv_tracer.update(clock, repo, evts);
658     srv_cx_bounds.update(clock, repo, evts);
659     srv_mandel.update(clock, repo, evts);
660     srv_screen_renderer.update(clock, repo, evts);
661 }
662
663 private:
664 Window window;
665
666 Renderer renderer;
667
668 FPSClock clock;
```

```
669
670     EventHandlerImpl evts;
671
672     PassiveDataRepo repo;
673
674     SelectionService srv_sel;
675
676     TracerService srv_tracer;
677
678     ComplexBoundsService srv_cx_bounds;
679
680     MandelUpdaterService srv_mandel;
681
682     ScreenRendererService srv_screen_renderer;
683     bool running;
684
685 };
686
687 int main()
688 {
689     try{
690         SDL_Guard guard(SDL_INIT_EVERYTHING);
691         App app("Mandel", 300, 300);
692         app.configure();
693         app.run();
694     }catch(const std::runtime_error& ex){
695         std::cout << ex.what() << std::endl;
696     }catch(...){
697         std::cout << "Unknown error! Exiting " << std::endl;
698     }
699     return 0;
700 }
```

6. fejezet

Helló, Welch!

6.1. Első osztályom

A lényeg, hogy adott egy számítási algoritmus melyet meghívva nem csak a következő értéket, hanem a rákövetkezőt is megkapjuk. Azaz az 1. hívásnál kell számítani, de a 2. hívásnál nem kell, hiszen az 1. számítás a 2. értékét is kiszámította előre. Azért hogy ezt kezeljük egy egyszerű osztályt fogunk írni. Ahhoz hogy ezt a "bufferelést" megoldjuk két változót vezetünk be: magát a tárolt értéket, illetve egy boolean-t, hogy jelenleg úgy szólnán szabad-e aktívnak tekinteni a tárolt értéket.

```
boolean store_empty;  
double stored;
```

Maga az algoritmus csak simán annyiról szól, hogy ha store_empty true, akkor kell számítást végezni. A számítás első eredményét visszaadjuk, a másikat betároljuk, és a store_empty-t false-re állítjuk. Ha store_empty false, akkor store_empty-t true-ra állítjuk, hisz úgymond ürítjük a buffert, és visszaadjuk a tárolt értéket.

A feladat könnyebb megértéséhez egy hackelt verzió az eredeti java class-ból.

```
1 package test;  
2  
3 public class PolarGen2 {  
4  
5     class Pair<T1,T2>{Pair(T1 a, T2 b){this.a=a;this.b=b;} public T1 a; ←  
6         public T2 b;}  
7  
8     public PolarGen2() { store_empty = true; }  
9  
10    public double next() {  
11        if(store_empty) {  
12            Pair<Double, Double> p = calculate();  
13            stored = p.b;  
14            return p.a;  
15        }else {  
16            store_empty = true;  
17            return stored;  
18        }  
19    }  
20 }
```

```
17     }
18 }
19
20 private Pair<Double, Double> calculate() {
21     double u1, u2, v1, v2, w;
22     do {
23         u1 = Math.random();
24         u2 = Math.random();
25         v1 = 2*u1 - 1;
26         v2 = 2*u2 - 1;
27         w = v1*v1 + v2*v2;
28     } while(w > 1);
29     double r = Math.sqrt((-2*Math.log(w))/w);
30     return new Pair<Double, Double>(r*v1, r*v2);
31 }
32
33 boolean store_empty;
34 double stored;
35
36 public static void main(String[] args) {
37     PolarGen2 g = new PolarGen2();
38     for(int i=0; i<10; ++i)
39         System.out.println(g.next());
40
41 }
42 }
43
```

Alább az eredeti feladat:

```
1 package test;
2
3 public class PolarGen {
4
5     public PolarGen() { store_empty = true; }
6
7     public double next() {
8         if(store_empty) {
9             double u1, u2, v1, v2, w;
10            do {
11                u1 = Math.random();
12                u2 = Math.random();
13                v1 = 2*u1 - 1;
14                v2 = 2*u2 - 1;
15                w = v1*v1 + v2*v2;
16            } while(w > 1);
17            double r = Math.sqrt((-2*Math.log(w))/w);
18            stored = r*v2;
19            store_empty = false;
20            return r*v1;
21        }
22    }
23 }
```

```
22         }else {
23             store_empty = true;
24             return stored;
25         }
26     }
27
28     boolean store_empty;
29     double stored;
30
31     public static void main(String[] args) {
32         PolarGen g = new PolarGen();
33         for(int i=0; i<10; ++i)
34             System.out.println(g.next());
35     }
36 }
37
38
```

C++ esetben három különbség lesz: random, sqrt és log.

[0,1) tartományon random double generálását végezhetjük az alábbi módon. Ehhez szükség lesz stdio-ra rnd() illetve RAND_MAX miatt.

```
double rnd01(){return rand() / (RAND_MAX + 1.);}
```

sqrt és log a már ismert math lib-ben van, azaz ne felejtjük include-olni.

```
1  #include <iostream>
2  #include <stdlib.h> // random
3  #include <math.h> // sqrt, log and rick n morty
4
5  class PolarGen
6  {
7  public:
8
9      PolarGen() : store_empty(true) { }
10
11     double next()
12     {
13         if(store_empty) {
14             double u1, u2, v1, v2, w;
15             do {
16                 u1 = rnd01();
17                 u2 = rnd01();
18                 v1 = 2*u1 - 1;
19                 v2 = 2*u2 - 1;
20                 w = v1*v1 + v2*v2;
21             } while(w > 1);
22             double r = sqrt((-2*log(w))/w);
23             stored = r*v2;
24             store_empty = false;
```

```
25     return r*v1;
26 }else {
27     store_empty = true;
28     return stored;
29 }
30 }
31 private:
32
33 double rnd01(){return rand() / (RAND_MAX + 1.);}
34
35 double rnd(double min_inclusive, double max_exclusive)
36 {
37     return min_inclusive + (rand() / ( RAND_MAX / (max_exclusive-
38         min_inclusive) ) ) ;
39 }
40 private:
41 bool store_empty;
42 double stored;
43 };
44 int main()
45 {
46     PolarGen pg;
47     for(int i=0; i<10; ++i){ std::cout<< pg.next() << std::endl;}
48     return 0;
49 }
```

6.2. LZW

Mielőtt neki ugrunk egy kis áttekintés a [progpater](#) alapján.

A programunk kapni fog egy {0,1} karakterekből (bár lehetnének bitek is) álló mintát. Ez alapján memóriában

01111001001001000111

A kapott minta alapján memóriában létre kell hoznunk egy fát. A fa csomópontokból áll. Minden csomópontoz tartozik egy érték illetve lehet két gyermeke, továbbiakban jobb és bal. A fa továbbá tartalmaz egy segéd ptr-t arra a csomópontra, ahol éppen állunk. A következő szabályok szerint építjük a fát:

- Alapból a gyökér létezik értéke /, és ezen gyökér csomóponton állunk(erre mutat a segéd ptr).
- Ha 0-t kapunk, és létezik bal gyermek, akkor a segéd ptr-rel inntől erre fogunk mutatni.
- Ha 0-t kapunk, és nem létezik bal gyermek, akkor segéd ptr által mutatott node bal gyermekeként létrehozuk. Segéd ptr-rel pedig gyökérre mutatunk inntől.
- Ha 1-t kapunk, akkor teljesen szimmetrikusan a fenti két szabályt jobb gyerekekre alkalmazzuk.

Pici példácskán, mondjuk 0110-re, nézzük meg hogy működik! Minden lépésnél ábrázolni fogom a fát is! Csillaggal ábrázoljuk hol áll éppen a segéd ptr!

```
Feldolgozva=[]
---/*
Feldolgozva=[0]
---/*
-----0
Feldolgozva=[01]
-----1
---/*
-----0
Feldolgozva=[011]
-----1*
---/
-----0
Feldolgozva=[0110]
-----1
-----0
---/*
-----0
```

Most nézzük meg mi ez az egész max, átlag, szórás!

Két fogalmat azonban be kell hozzá vezetni. Az egyik a levél, a másik a mélység.

Egy csomópontot levélnek nevezünk, ha nincsenek gyermekei.

Egy csomópont mélységén azt értjük (ezen esetben) hogy a hány csomópont érintésével tudunk eljutni gyökér nodeból az adott node-ba.

Vegyünk propateres példát és írjuk fel rá kézzel az értékeket!

```
-----1 (3)
-----1 (2)
-----1 (1)
-----0 (2)
-----0 (3)
-----0 (4)
---/ (0)
-----1 (2)
-----0 (1)
-----0 (2)
melyseg=4
altag=2.750000
szoras=0.957427
```

Max mélység alatt a legnagyobb mélységet értjük, azaz 4-et.

Átlagmélység a leaf mélységek összege osztva a leaf számmal, azaz:

```
hosszak={3,4,2,2}
leafs=4

atlag=4/3+4/4+2/4+2/4
atlag=0.75+1+0.5+0.5
atlag=2.75
```

A teljes fa szórása alatt a szórásösszeg és a levél szám hányadosának négyzetgyökét értjük. Meg kell jegyezni, hogy valójában leaf számnál eggyel kisebb számot használunk 1-nél nagyobb leaf számú esetekben.

Szórásösszeg, az összes levél szórásának összege.

Egy node szórása alatt az adott node mélységének átlagmélységtől vett különbségének négyzetét értjük.

```
hosszak={3,4,2,2}
leafs=4
atlag=2.75

szorasosszeg=(3-2.75)^2+(4-2.75)^2+(2-2.75)^2+(2-2.75)^2
szorasosszeg=0.0625+1.5625+0.5625+0.5625
szorasosszeg=2.75
szoras=(2.75/4-1)^0.5
szoras=(0.9166667)^0.5
szoras=0.957427
```

Most hogy kézzel kiszámoltuk, írjuk át C-re.

A C kódban akét komoly eltérés a Tanár Úr féle kódtól:

- allokáció/felaszabadításnál a ptr-t kiírjuk standard output-ra
- Általános traversalokat használunk funtion ptr-ekkel

A ptr-ek kilogolása csak amiatt fontos hogy grafikus visszajelzést is kapjunk, hiszen most ugye dinamikusan allokált adatokkal dolgozunk, amelyeket kézíleg kell életciklus menedzselni.

```
void free_node(int depth, Node* node, UserData*ud)
{
    if(node == NULL) return;
#ifdef CUSTOM_DEBUG_OUT
    printf("Free node %p\n",node);
#endif
    free(node);
}
```

Az általános traversal amiatt kellett, hogy egy kicsit az stl-es algorithm-re hajazva, legyen lehetősége a user-nek megadnia, hogy mi történjen, a fa pedig csak a helyes traversalért legyen felelős.

Nézzünk egy példát!

```
void foreach_inorder(int depth, Node* root, traversal_fn fptr, ↵
    UserData* ud )
{
    if(root == NULL) return;
    foreach_inorder(depth+1, root->right, fptr, ud );
    fptr(depth, root, ud );
    foreach_inorder(depth+1, root->left, fptr, ud );
}
```

Láthatóan a fenti funckió simán rekurzívan hívja magát a két child-ra. A fn ptr csak akkor jön be a képbe mikor magát az adott subtree root-ját dolgozzuk fel. UserData csak egy typedef-elt void ptr, hogy a user állapotot is betudjon adni, ne csak viselkedést. Egy példa fn amit fn ptr-ként beküldhetünk legyen a print

```
void print_node(int depth, Node* node, UserData*ud)
{
    int i;
    if(node == NULL) return;
    for(i=0; i<depth; ++i){printf("---"); }
    printf("%c (%d) %p\n", node->value, depth, node);
}
```

A számolásokhoz simán végigszaladunk a leafeken és mivel nem tudjuk előre a leaf számot, ezért egy dinamikusan allokált helyre gyűjtjük be a leaf-ek mélységeit. De csinálhattuk volna úgyis hogy többször megyünk végig a leaf-eken és akkor kevesebb memóriát használtunk volna, azaz CPU vs. MEM tradeoff.

Igen... LeafCountData-t totál kikerülhettük volna: bedobhattuk volna simán a nyers ptr-t aztán inkrementáltuk volna...

Alább a progpater-es példa inputként kimeneten.

```
Alloc node 000000000042E5E0
Alloc tree 00000000003B77E0
Alloc node 00000000003B7800
Alloc node 00000000003B7820
Alloc node 00000000003B7840
Alloc node 00000000003B7860
Alloc node 00000000003B7880
Alloc node 00000000003B78A0
Alloc node 00000000003B78C0
Alloc node 00000000003B78E0
Alloc node 00000000003B7900

-----1 <3> 00000000003B7900
-----1 <2> 00000000003B7840
----1 <1> 00000000003B7820
-----0 <2> 00000000003B7860
-----0 <3> 00000000003B78C0
-----0 <4> 00000000003B78E0
/ <0> 000000000042E5E0
-----1 <2> 00000000003B7880
----0 <1> 00000000003B7800
-----0 <2> 00000000003B78A0

Leaf depths: 3 4 2 2
deviation=2.750000
melyseg=4
atlag=2.750000
szoras=0.957427
Free node 00000000003B7900
Free node 00000000003B7840
Free node 00000000003B78E0
Free node 00000000003B78C0
Free node 00000000003B7860
Free node 00000000003B7820
Free node 00000000003B7880
Free node 00000000003B78A0
Free node 00000000003B7800
Free node 000000000042E5E0
Free tree 00000000003B77E0
```

6.1. ábra. Lzw C output

Alább a C forrás

```
1 #include <stdio.h> // print
2 #include <stdlib.h> // mem
3 #include <math.h> // sqrt
4
5 #define CUSTOM_DEBUG_OUT
6
7 typedef struct node_t{
8     struct node_t* left;
9     struct node_t* right;
10    char value;
11 } Node;
12
13
14 typedef struct bintree_t{
15     Node* root;
16     Node* cur;
17     int leaf_count;
18 } BinTree;
```

```
19
20 typedef void* UserData;
21
22 typedef void (*traversal_fn)(int depth, Node* node, UserData*ud);
23
24 void foreach_preorder(int depth,Node* root, traversal_fn fptr, UserData* ud ↔
    )
25 {
26     if(root == NULL) return;
27     fptr(depth,root,ud);
28     foreach_preorder(depth+1,root->right, fptr, ud );
29     foreach_preorder(depth+1,root->left, fptr, ud );
30 }
31
32 void foreach_inorder(int depth,Node* root, traversal_fn fptr, UserData* ud ↔
    )
33 {
34     if(root == NULL) return;
35     foreach_inorder(depth+1,root->right, fptr, ud );
36     fptr(depth,root,ud);
37     foreach_inorder(depth+1,root->left, fptr, ud );
38 }
39
40 void foreach_postorder(int depth,Node* root, traversal_fn fptr, UserData* ↔
    ud )
41 {
42     if(root == NULL) return;
43     foreach_postorder(depth+1,root->right, fptr, ud );
44     foreach_postorder(depth+1,root->left, fptr, ud );
45     fptr(depth,root,ud);
46 }
47
48 void foreach_leaf(int depth,Node* root, traversal_fn fptr, UserData* ud )
49 {
50     if(root == NULL) return;
51     foreach_leaf(depth+1,root->right, fptr, ud );
52     if(root->right==NULL&&root->left==NULL) fptr(depth,root,ud);
53     foreach_leaf(depth+1,root->left, fptr, ud );
54 }
55
56 Node* make_node(char value)
57 {
58     Node* node;
59     node = (Node*) malloc(sizeof(Node));
60     if(node == NULL){return NULL;}
61     node->value=value;
62     node->left= NULL;
63     node->right=NULL;
64     #ifdef CUSTOM_DEBUG_OUT
65     printf("Alloc node %p\n",node);
```

```
66     #endif
67     return node;
68 }
69
70 BinTree* make_bintree()
71 {
72     BinTree* tree;
73     Node* root;
74     root = make_node('/');
75     if(root == NULL){return NULL;}
76     tree = (BinTree*) malloc(sizeof(BinTree));
77     if(tree == NULL){free(root);return NULL;}
78     tree->root = tree->cur = root;
79     tree->leaf_count=1;
80     #ifdef CUSTOM_DEBUG_OUT
81         printf("Alloc tree %p\n",tree);
82     #endif
83     return tree;
84 }
85
86 void push_value(BinTree* tree, char value)
87 {
88     if(tree==NULL) return;
89     if(tree->root==NULL || tree->cur==NULL) return;
90     if(value=='0'){
91         if(tree->cur->left == NULL){
92             if(tree->cur->right!=NULL) tree->leaf_count=tree->leaf_count+1;
93             tree->cur->left = make_node(value);
94             tree->cur = tree->root;
95         }else{
96             tree->cur = tree->cur->left;
97         }
98     }else{
99         if(tree->cur->right == NULL){
100             if(tree->cur->left!=NULL) tree->leaf_count=tree->leaf_count+1;
101             tree->cur->right = make_node(value);
102             tree->cur = tree->root;
103         }else{
104             tree->cur = tree->cur->right;
105         }
106     }
107 }
108
109 void print_node(int depth, Node* node, UserData*ud)
110 {
111     int i;
112     if(node == NULL) return;
113     for(i=0;i<depth; ++i){printf("---"); }
114     printf("%c (%d) %p\n",node->value, depth,node);
115 }
```

```
116
117 void free_node(int depth, Node* node, UserData*ud)
118 {
119     if(node == NULL) return;
120     #ifdef CUSTOM_DEBUG_OUT
121         printf("Free node %p\n",node);
122     #endif
123     free(node);
124 }
125
126 void free_tree(BinTree* tree)
127 {
128     if(tree==NULL) return;
129     foreach_postorder(0,tree->root,free_node,NULL);
130     #ifdef CUSTOM_DEBUG_OUT
131         printf("Free tree %p\n",tree);
132     #endif
133     free(tree);
134 }
135
136 typedef struct leafcountdata_t{
137     int cur;
138     int* heights;
139 } LeafCountData;
140
141 void collect_leaf_heights(int depth, Node* node, UserData*ud)
142 {
143     if(node == NULL) return;
144     LeafCountData* lcd = (LeafCountData*)ud;
145     lcd->heights[lcd->cur] = depth;
146     lcd->cur=lcd->cur+1;
147 }
148
149 int main(int argc, char** argv)
150 {
151     int i;
152     int leaf_count;
153     BinTree* bt;
154     int *leaf_heights;
155     int max_melyseg;
156     double deviation, avg;
157     LeafCountData lc;
158     char arr[20] = {'0','1','1','1','1','0','0','1','0','0','1','0','0','1','0','0','1','0', ←
159                    '0','0','0','1','1','1'};
160     bt = make_bintree();
161     if(bt == NULL){ return 1;}
162     for(i = 0; i < 20; ++i){push_value(bt,arr[i]);}
163     printf("\nINORDER\n");
164     foreach_inorder(0,bt->root,print_node,NULL);
165     printf("\n");
```

```
165 leaf_count = bt->leaf_count;
166 leaf_heights = (int*) malloc(sizeof(int)*leaf_count);
167 if(leaf_heights==NULL){free_tree(bt); return 1;}
168 lc.cur=0;
169 lc.heights=leaf_heights;
170 foreach_leaf(0,bt->root,collect_leaf_heights,(UserData)&lc);
171 printf("Leaf depths: ");
172 max_melyseg = 0;
173 deviation = 0.0;
174 for(i = 0; i < leaf_count; ++i){
175     if(max_melyseg<leaf_heights[i]){max_melyseg=leaf_heights[i];}
176     avg = avg+leaf_heights[i];
177     printf(" %d",leaf_heights[i]);
178 }
179 printf("\n");
180 avg = avg / leaf_count;
181 deviation = 0.0;
182 for(i = 0; i < leaf_count; ++i){
183     deviation = deviation + ((leaf_heights[i]-avg)*(leaf_heights[i]-avg));
184 }
185 if (leaf_count - 1 > 0)
186     deviation = sqrt( deviation / (leaf_count - 1));
187 else
188     deviation = sqrt ( deviation );
189 printf("melyseg=%d\n",max_melyseg);
190 printf("atlag=%f\n",avg);
191 printf("szoras=%f\n",deviation);
192 printf("\nPREORDER\n");
193 foreach_preorder(0,bt->root,print_node,NULL);
194 printf("\nPOSTORDER\n");
195 foreach_postorder(0,bt->root,print_node,NULL);
196 printf("\n");
197 free_tree(bt);
198 free(leaf_heights);
199 return 0;
200 }
```

6.3. Fabejárás

Az előző feladatban még a free-t is a function ptr-es traversallal csináltuk, azaz az előző feladatban ez részletezésre került. Alábbi kód részleten látszik, hogy az előző részben bevezetett módon elég egyszerű a dolog.

```
printf("\nPREORDER\n");
foreach_preorder(0,bt->root,print_node,NULL);
printf("\nPOSTORDER\n");
foreach_postorder(0,bt->root,print_node,NULL);
```


A hangsúly a fn ptr alkalmazásán van. Ha azt értjük, akkor az alábbi kép nem lesz meglepő.

```
INORDER
-----1 (3) 0000000000367900
-----1 (2) 0000000000367840
---1 (1) 0000000000367820
-----0 (2) 0000000000367860
-----0 (3) 00000000003678C0
-----0 (4) 00000000003678E0
/ (0) 000000000060E610
-----1 (2) 0000000000367880
---0 (1) 0000000000367800
-----0 (2) 00000000003678A0

Leaf depths: 3 4 2 2
melyseg=4
atlag=2.750000
szoras=0.957427

PREORDER
/ (0) 000000000060E610
---1 (1) 0000000000367820
-----1 (2) 0000000000367840
-----1 (3) 0000000000367900
-----0 (2) 0000000000367860
-----0 (3) 00000000003678C0
-----0 (4) 00000000003678E0
---0 (1) 0000000000367800
-----1 (2) 0000000000367880
-----0 (2) 00000000003678A0

POSTORDER
-----1 (3) 0000000000367900
-----1 (2) 0000000000367840
-----0 (4) 00000000003678E0
-----0 (3) 00000000003678C0
-----0 (2) 0000000000367860
---1 (1) 0000000000367820
-----1 (2) 0000000000367880
-----0 (2) 00000000003678A0
---0 (1) 0000000000367800
/ (0) 000000000060E610
```

6.2. ábra. Traversals

6.4. Tag a gyökér

A Twitch-es nbatfai megoldást fogom átszerkeszteni. A builderezést és lambdázást a mozgató szemantika részben fogom csinálni, ahol tényleg nullából újra írom.

Mivel az nbatfai root ptr-esre alapszik, és azt a következő fejezetben dolgoztam ki, itt effektíve csak egy diff-et írok doksiba.

Mielőtt belemegyünk, annyit érdemes tisztázni, hogy ValueType-től nem várhatjuk el, hogy lesz default ctor-a, így sajnos a tree ctor-ához hozzá kell nyúlni.

Magyarul, ha biztosra tudnánk venni, hogy ValueType halmaznak van egy speciális eleme ami a halmaz alját, vagy null-t vagy stb.-t jelentené, akkor nem kéne mindenképpen értéket kérni a tree ctorban. Azonban ez elég erős megkötés lenne.

Kilenc helyen értem hozzá a kódhoz, ezek tételes felsorolása és magyarázata alább következik.

Az első módosítás a tree ctor. Én úgy gondolom érdemes mindenképpen explicit értéket kérni, mert különben arra kéne hagyatkoznunk, hogy ValueType-nak van default ctor-a. Természetesen treep-t root címére kell állítanunk.

```
BinTree(ValueType rootval) : root(rootval), treep(&root), depth(0)
```

Az második módosítás a dtor-t illeti. Mivel root kompozíciós tag, ezért csak a child node-okra kell hívunk a rekurzív delete-t.

```
deltree(root.left_child() );  
deltree(root.right_child() );
```

A harmadik módosítás a bintree leftshiftet érinti. Root ptr-es esetben vizsgálni kell, hogy root létezik-e, viszont ezen esetben ha tree létezik akkor root-is (Java-ban swing-ben van erre sok példa, hogy van egy default úgymond implicit root mindig.).

A negyedik módosítás hasonlóan a bintree leftshiftet érinti. Treep-nek magát root-ot nem assignolhatjuk, hanem root címe fog kellene.

```
treep = &root;
```

Az ötödik módosítás ZLWTree template mentesítése. Enélkül az eredeti kód implicit konverziókat fog végezni és a leftshift is rosszul fog működni char-tól eltérő datatype esetén.

A hatodik módosítás valójában csak annyi, hogy delegáltuk a node példányosítást BinTree ctor-nak.

```
ZLWTree() : BinTree<char>(' /')
```

A hetedik, nyolcadik és kilencedik módosítás valójában csak annyi, hogy root címe kell treep-nek hisz az pointer.

```
1  #include <iostream>  
2  template<typename ValueType>  
3  class BinTree  
4  {  
5  protected:  
6      class Node  
7      {  
8      public:  
9          Node(ValueType value) : value(value), left(nullptr), right(nullptr), ←  
10             count(0) {}  
11      private:  
12          Node(const Node&);  
13          Node& operator=(const Node&);  
14          Node(const Node&&);  
15          Node& operator=(const Node&&);  
16      public:  
17          ValueType get_value() { return value; }  
18          Node* left_child() { return left; }  
19          Node* right_child() { return right; }  
20          void left_child(Node* node) { this->left=node; }  
21          void right_child(Node* node) { this->right=node; }
```

```
21     int get_count(){ return count;};
22     void incr_count(){ this->count++;}
23 private:
24     ValueType value;
25     Node* left;
26     Node* right;
27     int count;
28 }; /* class Node */
29 public:
30     // CHANGE 1
31     BinTree(ValueType rootval) : root(rootval), treep(&root), depth(0)
32     {
33
34     }
35     ~BinTree()
36     {
37         // CHANGE 2
38         deltree(root.left_child() );
39         deltree(root.right_child() );
40     }
41 private:
42     BinTree(const BinTree&);
43     BinTree& operator=(const BinTree&);
44     BinTree(const BinTree&&);
45     BinTree& operator=(const BinTree&&);
46
47
48 public:
49     BinTree& operator<<(ValueType);
50     void print(){ print(&root, std::cout);}
51     void print(Node* node, std::ostream& os);
52     void deltree(Node* node);
53 protected:
54     Node root;
55     Node* treep;
56     int depth;
57 }; /* class Tree */
58
59 template<typename ValueType>
60 BinTree<ValueType>& BinTree<ValueType>::operator<<(ValueType value)
61 {
62     // CHANGE 3
63     if(treep->get_value() == value){
64         treep->incr_count();
65     }else if(treep->get_value() > value){
66         if(!treep->left_child()){
67             treep->left_child(new Node(value));
68         }else{
69             treep = treep->left_child();
70             *this<<value;
```

```
71     }
72 }else{
73     if(!treep->right_child()){
74         treep->right_child(new Node(value));
75     }else{
76         treep = treep->right_child();
77         *this<<value;
78     }
79 }
80 // CHANGE 4
81 treep = &root;
82 return *this;
83 }
84
85 template<typename ValueType>
86 void BinTree<ValueType>::print(Node* node, std::ostream& os)
87 {
88     if(node){
89         ++depth;
90         print(node->right_child(),os );
91         for(int i = 1; i < depth; ++i){os << "---";}
92         os << node->get_value() << " " << depth << " " << node->get_count() << " ←"
93         std::endl;
94         print(node->left_child(),os );
95         --depth;
96     }
97 }
98
99 template<typename ValueType>
100 void BinTree<ValueType>::deltree(typename BinTree<ValueType>::Node* node)
101 {
102     if(node){
103         deltree(node->left_child() );
104         deltree(node->right_child() );
105         delete node;
106     }
107 }
108 // CHANGE 5
109 class ZLWTree : public BinTree<char>
110 {
111 public:
112     // CHANGE 6
113     ZLWTree() : BinTree<char>('/')
114     {
115         // CHANGE 7
116         this->treep = &(this->root);
117     }
118 private:
119 public:
```

```

120     ZLWTree& operator<<(char);
121 private:
122
123 };
124
125 ZLWTree& ZLWTree::operator<<(char value)
126 {
127     if(value=='0'){
128         if(!this->treep->left_child()){
129             typename BinTree<char>::Node* node = new typename BinTree<char>::Node ←
130                 (value);
131             this->treep->left_child(node);
132             // CHANGE 8
133             this->treep = &(this->root);
134         }else{
135             this->treep = this->treep->left_child();
136         }
137     }else{
138         if(!this->treep->right_child()){
139             typename BinTree<char>::Node* node = new typename BinTree<char>::Node ←
140                 (value);
141             this->treep->right_child(node);
142             // CHANGE 9
143             this->treep = &(this->root);
144         }else{
145             this->treep = this->treep->right_child();
146         }
147     }
148     return *this;
149 }
150
151 int main(int argc, char** argv, char** env)
152 {
153     BinTree<int> bt(8);
154     bt <<9<<5<<2<<7;
155     bt.print();
156     std::cout << std::endl;
157     ZLWTree zt;
158     zt<<'0'<<'1'<<'1'<<'1'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1' ←
159         <<'0'<<'0'<<'0'<<'1'<<'1'<<'1';
160     zt.print();
161     return 0;
162 }

```

Persze hogy ne legyen baj, akár a ctorba is bepassezolhatjuk argként root és leftshift comparatorban használt value-t

```

1 #include <iostream>
2 #include <string>
3 template<typename ValueType>

```

```
4 class BinTree
5 {
6 protected:
7     class Node
8     {
9     public:
10         Node(ValueType value) : value(value), left(nullptr), right(nullptr), ←
            count(0){}
11     private:
12         Node(const Node&);
13         Node& operator=(const Node&);
14         Node(const Node&&);
15         Node& operator=(const Node&&);
16     public:
17         ValueType get_value(){ return value;}
18         Node* left_child(){ return left;}
19         Node* right_child(){ return right;}
20         void left_child(Node* node){ this->left=node;}
21         void right_child(Node* node){ this->right=node;}
22         int get_count(){ return count;};
23         void incr_count(){ this->count++;}
24     private:
25         ValueType value;
26         Node* left;
27         Node* right;
28         int count;
29     }; /* class Node */
30 public:
31     // CHANGE 1
32     BinTree(ValueType rootval) : root(rootval), treep(&root), depth(0)
33     {
34
35     }
36     ~BinTree()
37     {
38         // CHANGE 2
39         deltree(root.left_child() );
40         deltree(root.right_child() );
41     }
42 private:
43     BinTree(const BinTree&);
44     BinTree& operator=(const BinTree&);
45     BinTree(const BinTree&&);
46     BinTree& operator=(const BinTree&&);
47
48
49 public:
50     BinTree& operator<<(ValueType);
51     void print(){ print(&root, std::cout);}
52     void print(Node* node, std::ostream& os);
```

```
53 void deltree(Node* node);
54 protected:
55     Node root;
56     Node* treep;
57     int depth;
58 }; /* class Tree */
59
60 template<typename ValueType>
61 BinTree<ValueType>& BinTree<ValueType>::operator<<(ValueType value)
62 {
63     // CHANGE 3
64     if(treep->get_value() == value){
65         treep->incr_count();
66     }else if(treep->get_value() > value){
67         if(!treep->left_child()){
68             treep->left_child(new Node(value));
69         }else{
70             treep = treep->left_child();
71             *this<<value;
72         }
73     }else{
74         if(!treep->right_child()){
75             treep->right_child(new Node(value));
76         }else{
77             treep = treep->right_child();
78             *this<<value;
79         }
80     }
81     // CHANGE 4
82     treep = &root;
83     return *this;
84 }
85
86 template<typename ValueType>
87 void BinTree<ValueType>::print(Node* node, std::ostream& os)
88 {
89     if(node){
90         ++depth;
91         print(node->right_child(), os);
92         for(int i = 1; i < depth; ++i){os << "---";}
93         os << node->get_value() << " " << depth << " " << node->get_count() << ←
94             std::endl;
95         print(node->left_child(), os);
96         --depth;
97     }
98 }
99
100 template<typename ValueType>
101 void BinTree<ValueType>::deltree(typename BinTree<ValueType>::Node* node)
```

```
102 {
103     if(node){
104         deltree(node->left_child() );
105         deltree(node->right_child() );
106         delete node;
107     }
108 }
109 // CHANGE 5
110 template<typename ValueType>
111 class ZLWTree : public BinTree<ValueType>
112 {
113 public:
114     // CHANGE 6
115     ZLWTree(const ValueType&& rootval, const ValueType&& leftval) : BinTree< ←
        ValueType>(std::move(rootval)), leftval(std::move(leftval))
116     {
117         // CHANGE 7
118         this->treep = &(this->root);
119     }
120 private:
121 public:
122     ZLWTree& operator<<(ValueType);
123 private:
124     ValueType leftval;
125 };
126
127 template<typename ValueType>
128 ZLWTree<ValueType>& ZLWTree<ValueType>::operator<<(ValueType value)
129 {
130     if(value==this->leftval){
131         if(!this->treep->left_child()){
132             typename BinTree<ValueType>::Node* node = new typename BinTree< ←
                ValueType>::Node(value);
133             this->treep->left_child(node);
134             // CHANGE 8
135             this->treep = &(this->root);
136         }else{
137             this->treep = this->treep->left_child();
138         }
139     }else{
140         if(!this->treep->right_child()){
141             typename BinTree<ValueType>::Node* node = new typename BinTree< ←
                ValueType>::Node(value);
142             this->treep->right_child(node);
143             // CHANGE 9
144             this->treep = &(this->root);
145         }else{
146             this->treep = this->treep->right_child();
147         }
148     }
```



```

149     return *this;
150 }
151
152
153 int main(int argc, char** argv, char** env)
154 {
155     BinTree<int> bt(8);
156     bt <<9<<5<<2<<7;
157     std::cout << "BinTree" << std::endl;
158     bt.print();
159     ZLWTree<char> zt('/', '0');
160     zt<<'0'<<'1'<<'1'<<'1'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1' <-
        <<'0'<<'0'<<'0'<<'1'<<'1'<<'1';
161     std::cout << "ZLW char" << std::endl;
162     zt.print();
163     ZLWTree<std::string> zst("/", "0");
164     zst<<"0"<<"1"<<"1"<<"1"<<"1"<<"0"<<"0"<<"1"<<"0"<<"0"<<"1"<<"0"<<"0"<<"1" <-
        <<"0"<<"0"<<"0"<<"1"<<"1"<<"1";
165     std::cout << "ZLW string" << std::endl;
166     zst.print();
167     return 0;
168 }

```

6.5. Mutató a gyökér

A kísérleti nbatfai twitch adásban látható megoldás lehet egy példa erre.

Áttekintve annyiról van szó, hogy a egyszerre egy bináris fát és egy lzw fát fogunk csinálni. Kód duplikáció elkerülése végett pedig inheritance-t használunk. Apró kellemetlenség hogy egyenesen cpp-be megy, de mozgató szemantikás feladatban a saját megoldásomban úgyis nulláról újraírom az egészet, és ott már header is lesz.

Úgy döntöttem ZLWTree-t template mentesítem és explicit BinTree<char>-ról származtatom, mert ctor és left shift esetében alul definiált lenne.

Úgy döntöttünk hogy nesteljük bele a node osztályt. Talán ennek a bemutatásával érdemes kezdeni a dolgot:

```

class Node
{
    ...
    ValueType value;
    Node* left;
    Node* right;
    int count;
};

```

ValueType BinTree template paramétere, ezt (mivel inner class), használhatjuk Node esetén is.

Left right a gyermekekre mutató, ctor-ban nullptr-re initelt pointerok.

Count a bináris kereső fa miatt fontos állapotváltozó. Egyszerűen annyiról van szó, hogyha például egy Bintree<int> be kétszer left shiftelek mondjuk 9-et, akkor ne két külön node jöjjön létre 9-es value-val. Ehelyett ilyen esetben a már létező 9-es értékű node count állapotváltozóját fogjuk a második 9-es miatt inkrementálni.

A funkciók csak egyszerű setter-ek getter-ek. Incr count pedig csak egy shortcut, mert a count setter-t get_count()+1 kéne hívni.

Most nézzük tree állapotváltozóit!

Root a gyökerre mutató ptr. Treep a fenti manuális példákban a segéd ptr, míg depth egy internal változó, amit a traversalnál fogunk használni annak adminisztrálására, hogy milyen mélyen vagyunk.

Persze, treep és depth nem kell törvényyszerűleg BinTree-be rakni. A mozgató szemantikás delegáljuk is majd ezt a felelősséget egy iterátornak ami csak és kizárólag a faépítéséért felelős.

```
template<typename ValueType>
class BinTree
{
    ...
    Node* root;
    Node* treep;
    int depth;
};
```

Viselkedések közül a left shift a legfontosabb, hiszen ez a node példányosítást végzi.

```
template<typename ValueType>
BinTree<ValueType>& BinTree<ValueType>::operator<<(ValueType value)
{
    if(!treep){
        root = treep = new Node(value);
    }else if(treep->get_value() == value){
        treep->incr_count();
    }else if(treep->get_value() > value){
        if(!treep->left_child()){
            treep->left_child(new Node(value));
        }else{
            treep = treep->left_child();
            *this<<value;
        }
    }else{
        if(!treep->right_child()){
            treep->right_child(new Node(value));
        }else{
            treep = treep->right_child();
            *this<<value;
        }
    }
    treep = root;
    return *this;
}
```

```
}
```

Class-on kívül definiáltuk, de természetesen továbbra is template-es.

A funkció minden esetben azzal zár hogy visszaugrunk root-ra.

Ha a value egyenlő azzal a value-va amit a jelenlegi segéd ptr mutat, akkor csak inkrementálni fogjuk.

Ha a value kisebb akkor bal irányba fogunk "mozdulni".

Ha balra nincs még node, akkor létrehozunk egy bal gyermek node-ot a jelenleg treep által mutatott node-on.

Ha balra van már node, akkor treep-t áthelyezzük és rekurzívan újra meghívjuk a funkciót (de ugye nem ugyanaz fog történni, hisz treep már máshol áll!).

Ha az érték nagyobb akkor teljesen szimmetrikusan jobb irányba végezzük el ugyanezt

print egy egyszerű inorder bejárást végez. Ezt a C-s példában fn ptr-el, a mozgatóban később pedig funccal fogjuk kiváltani.

Itt annyit érdemes megjegyezni, hogy depth az BinTree osztály tulajdonsága.

```
template<typename ValueType>
void BinTree<ValueType>::print(Node* node, std::ostream& os)
{
    if(node){
        ++depth;
        print(node->right_child(), os );
        for(int i = 1; i < depth; ++i){os << "---";}
        os << node->get_value() << " " << depth << " " << node-> ←
            get_count() << std::endl;
        print(node->left_child(), os );
        --depth;
    }
}
```

deltree csak postorder-ben végig járja a node gyökerű subtree-t és felszabadítja a foglalt memóriát. A C-s kódban ez az a rész amit direkt címre pontosan ki is irattam.

```
template<typename ValueType>
void BinTree<ValueType>::deltree(typename BinTree<ValueType>::Node* ←
    node)
{
    if(node){
        deltree(node->left_child() );
        deltree(node->right_child() );
        delete node;
    }
}
```

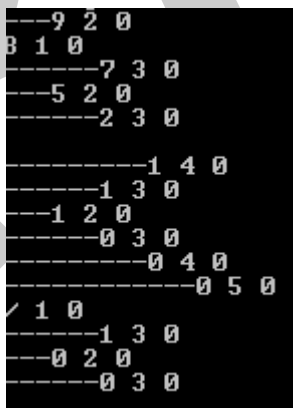
ZLWTree esetén kicsit más a left shift. Ezt manuálisan levezettük feljebb. Az egyetlen külön említést érdemlő dolog az az hogy, mivel class-on kívül írjuk a definíciót ezért globál namespace-ben nincs csak úgy benne Node varázssütésre. Ez látható a node példányosításos sorokban.

```

ZLWTree& ZLWTree::operator<<(char value)
{
    if(value=='0'){
        if(!this->treep->left_child()){
            typename BinTree<char>::Node* node = new typename BinTree<char> <-
                >::Node(value);
            this->treep->left_child(node);
            this->treep = this->root;
        }else{
            this->treep = this->treep->left_child();
        }
    }else{
        if(!this->treep->right_child()){
            typename BinTree<char>::Node* node = new typename BinTree<char> <-
                >::Node(value);
            this->treep->right_child(node);
            this->treep = this->root;
        }else{
            this->treep = this->treep->right_child();
        }
    }
    return *this;
}

```

Alább látható a zlw progí kimenete



```

--9 2 0
3 1 0
--7 3 0
--5 2 0
---2 3 0

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

```

6.3. ábra. ZLW out

```

1 #include <iostream>
2 template<typename ValueType>
3 class BinTree
4 {
5 protected:
6     class Node

```

```
7 {
8 public:
9     Node(ValueType value) : value(value), left(nullptr), right(nullptr), ←
        count(0) {}
10 private:
11     // TODO impl rule of five
12     Node(const Node&);
13     Node& operator=(const Node&);
14     Node(const Node&&);
15     Node& operator=(const Node&&);
16 public:
17     ValueType get_value(){ return value;}
18     Node* left_child(){ return left;}
19     Node* right_child(){ return right;}
20     void left_child(Node* node){ this->left=node;}
21     void right_child(Node* node){ this->right=node;}
22     int get_count(){ return count;};
23     void incr_count(){ this->count++;}
24 private:
25     ValueType value;
26     Node* left;
27     Node* right;
28     int count;
29 }; /* class Node */
30 public:
31     BinTree(Node* root=nullptr, Node* treep=nullptr) : root(root), treep( ←
        treep), depth(0)
32 {
33
34 }
35 ~BinTree()
36 {
37     deltree(root);
38 }
39 private:
40     // TODO impl rule of five
41     BinTree(const BinTree&);
42     BinTree& operator=(const BinTree&);
43     BinTree(const BinTree&&);
44     BinTree& operator=(const BinTree&&);
45
46
47 public:
48     BinTree& operator<<(ValueType);
49     void print(){ print(root, std::cout);}
50     void print(Node* node, std::ostream& os);
51     void deltree(Node* node);
52 protected:
53     Node* root;
54     Node* treep;
```

```
55     int depth;
56 }; /* class Tree */
57
58 template<typename ValueType>
59 BinTree<ValueType>& BinTree<ValueType>::operator<<(ValueType value)
60 {
61     if(!treep) {
62         root = treep = new Node(value);
63     }else if(treep->get_value() == value){
64         treep->incr_count();
65     }else if(treep->get_value() > value){
66         if(!treep->left_child()){
67             treep->left_child(new Node(value));
68         }else{
69             treep = treep->left_child();
70             *this<<value;
71         }
72     }else{
73         if(!treep->right_child()){
74             treep->right_child(new Node(value));
75         }else{
76             treep = treep->right_child();
77             *this<<value;
78         }
79     }
80     treep = root;
81     return *this;
82 }
83
84 template<typename ValueType>
85 void BinTree<ValueType>::print(Node* node, std::ostream& os)
86 {
87     if(node) {
88         ++depth;
89         print(node->right_child(), os );
90         for(int i = 1; i < depth; ++i){os << "---";}
91         os << node->get_value() << " " << depth << " " << node->get_count() << ←
92         std::endl;
93         print(node->left_child(), os );
94         --depth;
95     }
96 }
97
98 template<typename ValueType>
99 void BinTree<ValueType>::deltree(typename BinTree<ValueType>::Node* node)
100 {
101     if(node) {
102         deltree(node->left_child() );
103         deltree(node->right_child() );
```

```
104     delete node;
105 }
106 }
107
108 class ZLWTree : public BinTree<char>
109 {
110 public:
111     ZLWTree() : BinTree<char>(new typename BinTree<char>::Node('/'))
112     {
113         this->treep = this->root;
114     }
115 private:
116 public:
117     ZLWTree& operator<<(char);
118 private:
119
120 };
121
122 ZLWTree& ZLWTree::operator<<(char value)
123 {
124     if(value=='0'){
125         if(!this->treep->left_child()){
126             typename BinTree<char>::Node* node = new typename BinTree<char>::Node ←
127                 (value);
128             this->treep->left_child(node);
129             this->treep = this->root;
130         }else{
131             this->treep = this->treep->left_child();
132         }
133     }else{
134         if(!this->treep->right_child()){
135             typename BinTree<char>::Node* node = new typename BinTree<char>::Node ←
136                 (value);
137             this->treep->right_child(node);
138             this->treep = this->root;
139         }else{
140             this->treep = this->treep->right_child();
141         }
142     }
143     return *this;
144 }
145
146 int main(int argc, char** argv, char** env)
147 {
148     BinTree<int> bt;
149     bt <<8<<9<<5<<2<<7;
150     bt.print();
151     std::cout << std::endl;
152     ZLWTree zt;
```

```

152     zt<<'0'<<'1'<<'1'<<'1'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1'  ↵
        <<'0'<<'0'<<'0'<<'1'<<'1'<<'1';
153     zt.print();
154     return 0;
155 }

```

6.6. Mozzató és másoló szemantika

Az előbbiekben megírt osztály(oka)t fogjuk felkészíteni mozzatásra és másolásra. Ez a következő viselkedések implementációját vonja maga után:

- Copy ctor
- Copy assignment
- Move ctor
- Move assignment

Először a copy-val és utána a move-al fogunk foglalkozni.

Nézzük a copy ctor-t! Sajnos nem egyszerű a dolgunk. Gondoljunk bele, hogyha másolni akarjuk a fát, akkor a root node-ot át kell másolnunk az újba. Viszont ha itt megállunk, akkor az új root node-unk a régi root node gyerekeire fog muattni, és amikor az egyik fa törlődik, akkor szépen ki fogja törölni a másik fa alól is a gyerekeket. Emiatt nem elég root-ot másolni, hanem deltreet-hez hasonlóan rekurzívan másolni kell minden node-ot, effektíve klónozva a fát. Ezt nevezzük deep copy-nak (shallow copy lenne, ha csak pl. root-ot másolnánk.) Ilyet automata módon a compiler nem csinál, ezért nekünk kell megírni.

Sajnos még mindig van egy baj. Ha a fentiek alapján megírnánk a kódot, akkor lenne két tökéletesen jó fánk A és B. Ahogy A-ba shifteljük az értékeket, azok szépen jelennének meg A-ban. Mi lenne B esetben?

Nos, azzal a buggal szembesülnénk, hogy ahogy shifteljük B-be az értékeket, azok A-n jelennek meg egy darabig olykor, hiszen B treep-je a régi A egy node-jára mutatna, amíg treep vissza nem ugrana root-ra. Hogy ezt elkerüljük a deep copy során treep-t is menedzselni kell, azaz amikor épp azon node-ot másoljuk amin a régi treep van, akkor a másik B fa treep-jét az új node-ra állítjuk.

```

BinTree(const BinTree& old)
{
    #ifdef LOG_TREE_LIFECYCLE
        std::cout<<"btree copy ctor "<< static_cast<void*>(this) << std::endl;
    #endif
    root = deep_copy(old.root, old.treep);
}

```

Deep copy esetén első arg-ként a régi A tree egy node-ját, másodikként a régi treep-t kapjuk. Ha a régi node nullptr akkor csak visszaadunk egy nullptr-t. Ha azonban létezik, akkor csinálunk egy újat átmásoljuk az értéket (pontosabban a ctor-ba arg-ként küldjük), és átmásoljuk a count int típusú data membert. A rekurzív

rész most jön, ugyanis a gyerek node-okat nem a régi fa nodejaira akarjuk állítani, hanem újonnan készített node-okra. Azaz ezekre újból rekurzívan hívjuk `deep_copy`-t. Ezt persze mind bal mind jobb gyereknél meg kell csinálni. A végén ellenőriznünk kell, hogy az a régi node amit jelenleg dolgoztunk fel éppen véletlenül az-e amire a régi `treep` mutatott. Ha ez így van akkor az új fa `treep`-jét átállítjuk az új node-ra.

```
Node* deep_copy(Node* old_n, Node* old_treep)
{
    Node* newn = nullptr;
    if(old_n) {
        newn = new Node(old_n->get_value());
        newn->left_child(deep_copy(old_n->left_child(), old_treep));
        newn->right_child(deep_copy(old_n->right_child(), old_treep));
        if(old_n==old_treep){this->treep=newn;}
    }
    return newn;
}
```

A következő a `move assignment`. Itt annyi történik, hogy a régi `BinTree` instance és az új `BinTree` instance `root` illetve `treep` pointereit megcseréljük. Nem kell ki `nullptr`-ezni a saját `root` és `treep`-ket `swap` előtt, hiszen ha már `move assignment` hívódik, akkor ezek vagy eleve `nullptr`-ek, vagy valid ptr-ek.

```
BinTree& operator=(BinTree&& old)
{
    #ifdef LOG_TREE_LIFECYCLE
        std::cout<<"btree move assign "<< static_cast<void*>(this) << " ←
        std::endl;
    #endif
    std::swap(old.root, root);
    std::swap(old.treep, treep);
    return *this;
}
```

A következő a `move ctor`. Itt kihasználjuk `move assignment`-et. Vegyük észre, hogy előtte a saját `root` és `treep`-t `nullptr`-re vesszük fel. Ha nem így tennénk akkor ezek uninitialized-ok lennének, magyarul memória szemét lenne rajtuk, és invalid helyekre mutatnának.

```
BinTree(BinTree&& old)
{
    #ifdef LOG_TREE_LIFECYCLE
        std::cout<<"btree move ctor "<< static_cast<void*>(this) << " ←
        std::endl;
    #endif
    root = nullptr;
    treep = nullptr;
    *this=std::move(old);
}
```

`Copy assignment` a `copy ctor`, `move ctor` és `move assignment` ismeretében egy trükkel könnyen megvalósítható. Egyszerűen létrehozunk egy `temp` nevű `ctor` body scope-an létező lokális `BinTree` instance-t. Ezen

BinTree instance copy ctor-ral lemásolja a régit. Ezután `std::swap`-el `tmp` és a jelenleg `this` által mutatott BinTree példányok közt cserét hajt végre. Mikor elhagyjuk a ctor scope-ot a `this` által mutatott instance életben marad (ő az old másolat), a régi pedig "elpusztul", azaz dtor hívódik az általa mutatott címre majd fel lesz szabadítva az általa foglalt memória terület.

```
BinTree& operator=(const BinTree& old)
{
    #ifndef LOG_TREE_LIFECYCLE
        std::cout<<"btree copy assign "<< static_cast<void*>(this) << " ←
        std::endl;
    #endif
    BinTree tmp(old);
    std::swap(*this,tmp);
    return *this;
}
```

Alább a teljes kód.

```
1  #include <iostream>
2  #include <algorithm>
3
4  // #define LOG_NODE_LIFECYCLE
5
6  #define LOG_TREE_LIFECYCLE
7
8  template<typename ValueType>
9  class BinTree
10 {
11 protected:
12     class Node
13     {
14     public:
15         Node(ValueType value) : value(value), left(nullptr), right(nullptr), ←
            count(0)
16         {
17             #ifndef LOG_NODE_LIFECYCLE
18                 std::cout<<"node ctor"<< static_cast<void*>(this) << std::endl;
19             #endif
20         }
21     private:
22         // TODO impl rule of five
23         Node(const Node&);
24         Node& operator=(const Node&);
25         Node(const Node&&);
26         Node& operator=(const Node&&);
27     public:
28         ValueType get_value(){ return value;}
29         Node* left_child(){ return left;}
30         Node* right_child(){ return right;}
31         void left_child(Node* node){ this->left=node;}
```

```

32     void right_child(Node* node){ this->right=node;}
33     int get_count(){ return count;};
34     void set_count(int c){ count=c; };
35     void incr_count(){ this->count++;}
36 private:
37     ValueType value;
38     Node* left;
39     Node* right;
40     int count;
41 }; /* class Node */
42 public:
43     BinTree(Node* root=nullptr, Node* treep=nullptr) : root(root), treep( ←
         treep), depth(0)
44     {
45         #ifdef LOG_TREE_LIFECYCLE
46             std::cout<<"btree ctor "<< static_cast<void*>(this) << std::endl;
47         #endif
48     }
49
50     ~BinTree()
51     {
52         #ifdef LOG_TREE_LIFECYCLE
53             std::cout<<"btree dtor "<< static_cast<void*>(this) << std::endl;
54         #endif
55         deltree(root);
56     }
57
58     BinTree(const BinTree& old)
59     {
60         #ifdef LOG_TREE_LIFECYCLE
61             std::cout<<"btree copy ctor "<< static_cast<void*>(this) << std::endl ←
                ;
62         #endif
63         root = deep_copy(old.root,old.treep);
64     }
65
66     BinTree& operator=(const BinTree& old)
67     {
68         #ifdef LOG_TREE_LIFECYCLE
69             std::cout<<"btree copy assign "<< static_cast<void*>(this) << std:: ←
                endl;
70         #endif
71         BinTree tmp(old);
72         std::swap(*this,tmp);
73         return *this;
74     }
75
76     BinTree(BinTree&& old)
77     {
78         #ifdef LOG_TREE_LIFECYCLE

```

```
79     std::cout<<"btree move ctor "<< static_cast<void*>(this) << std::endl ←
80     ;
81     #endif
82     root = nullptr;
83     treep = nullptr;
84     *this=std::move(old);
85 }
86
87 BinTree& operator=(BinTree&& old)
88 {
89     #ifdef LOG_TREE_LIFECYCLE
90     std::cout<<"btree move assign "<< static_cast<void*>(this) << std:: ←
91     endl;
92     #endif
93     std::swap(old.root,root);
94     std::swap(old.treep,treep);
95     return *this;
96 }
97
98 public:
99     BinTree& operator<<(ValueType);
100     void print(){ print(root, std::cout);}
101     void print(Node* node, std::ostream& os);
102     void deltree(Node* node);
103
104 private:
105     Node* deep_copy(Node* old_n, Node* old_treep)
106     {
107         Node* newn = nullptr;
108         if(old_n){
109             newn = new Node(old_n->get_value());
110             newn->set_count(old_n->get_count());
111             newn->left_child(deep_copy(old_n->left_child(),old_treep));
112             newn->right_child( deep_copy(old_n->right_child(),old_treep));
113             if(old_n==old_treep){this->treep=newn;}
114         }
115         return newn;
116     }
117
118 protected:
119     Node* root;
120     Node* treep;
121     int depth;
122 }; /* class Tree */
123
124 template<typename ValueType>
125 BinTree<ValueType>& BinTree<ValueType>::operator<<(ValueType value)
126 {
```

```
127     if(!treep){
128         root = treep = new Node(value);
129     }else if(treep->get_value() == value){
130         treep->incr_count();
131     }else if(treep->get_value() > value){
132         if(!treep->left_child()){
133             treep->left_child(new Node(value));
134         }else{
135             treep = treep->left_child();
136             *this<<value;
137         }
138     }else{
139         if(!treep->right_child()){
140             treep->right_child(new Node(value));
141         }else{
142             treep = treep->right_child();
143             *this<<value;
144         }
145     }
146     treep = root;
147     return *this;
148 }
149
150 template<typename ValueType>
151 void BinTree<ValueType>::print(Node* node, std::ostream& os)
152 {
153     if(node){
154         ++depth;
155         print(node->right_child(),os );
156         for(int i = 1; i < depth; ++i){os << "---";}
157         os << node->get_value() << " " << depth << " " << node->get_count() << " ←"
158             std::endl;
159         print(node->left_child(),os );
160         --depth;
161     }
162 }
163
164 template<typename ValueType>
165 void BinTree<ValueType>::deltree(typename BinTree<ValueType>::Node* node)
166 {
167     if(node){
168         deltree(node->left_child() );
169         deltree(node->right_child() );
170         #ifdef LOG_NODE_LIFECYCLE
171             std::cout<<"node delete "<< static_cast<void*>(node) << std::endl;
172         #endif
173         delete node;
174     }
175 }
```

```
176 template<typename ValueType, ValueType vr, ValueType v0>
177 class ZLWTree : public BinTree<char>
178 {
179 public:
180     ZLWTree() : BinTree<char>(new typename BinTree<char>::Node(vr))
181     {
182         this->treep = this->root;
183     }
184 private:
185 public:
186     ZLWTree& operator<<(char);
187 private:
188
189 };
190 template<typename ValueType, ValueType vr, ValueType v0>
191 ZLWTree<ValueType, vr, v0>& ZLWTree<ValueType, vr, v0>::operator<<(char ←
    value)
192 {
193     if(value==v0){
194         if(!this->treep->left_child()){
195             typename BinTree<char>::Node* node = new typename BinTree<char>::Node ←
                (value);
196             this->treep->left_child(node);
197             this->treep = this->root;
198         }else{
199             this->treep = this->treep->left_child();
200         }
201     }else{
202         if(!this->treep->right_child()){
203             typename BinTree<char>::Node* node = new typename BinTree<char>::Node ←
                (value);
204             this->treep->right_child(node);
205             this->treep = this->root;
206         }else{
207             this->treep = this->treep->right_child();
208         }
209     }
210     return *this;
211 }
212
213 void test_cp_mv();
214
215 void test_bintree();
216
217 void test_zlw();
218
219 int main(int argc, char** argv, char** env)
220 {
221     test_cp_mv();
222 }
```

```

223 }
224
225 void test_cp_mv()
226 {
227     BinTree<int> bt;
228     bt <<8<<9<<5<<2<<7;
229     bt.print();
230     std::cout << std::endl;
231     ZLWTree<char, '/', '0'> zt;
232     zt <<'0'<<'0'<<'0'<<'0'<<'0';
233     zt.print();
234     ZLWTree<char, '/', '0'> zt2(zt);
235     ZLWTree<char, '/', '0'> zt3;
236     zt3 <<'1'<<'1'<<'1'<<'1'<<'1';
237     std::cout<<"***"<<std::endl;
238     zt = zt3;
239     std::cout<<"***"<<std::endl;
240     ZLWTree<char, '/', '0'> zt4 = std::move(zt2);
241 }
242
243 void test_bintree()
244 {
245     BinTree<int> bt;
246     bt <<8<<9<<5<<2<<7;
247     bt.print();
248     std::cout << std::endl;
249 }
250
251 void test_zlw()
252 {
253     ZLWTree<char, '/', '0'> zt;
254     zt<<'0'<<'1'<<'1'<<'1'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1' ←
        <<'0'<<'0'<<'0'<<'1'<<'1'<<'1';
255     zt.print();
256     std::cout << std::endl;
257 }

```

Ha a traversált esetleg egy user supplied lambdára akarjuk cserélni akkor alábbi kódban találhatunk rá megoldást: Az előző kódhoz képest itt a számítást is megcsináltuk. Mármost node depth max, átlag, szórás.

```

1  #include <iostream>
2  #include <string>
3  #include <vector>
4  #include <utility>
5  #include <functional>
6  #include <algorithm>
7
8  // #define LOG_NODE_LIFECYCLE
9
10 #define LOG_TREE_LIFECYCLE
11

```

```
12 template<typename V>
13 class BTree;
14
15 template<typename V>
16 class SearchTree;
17
18 template<typename V>
19 class LZWTree;
20
21 template<typename V>
22 class BNode
23 {
24     // [LIFECYCLE]
25 public:
26     BNode(V v) : v(v), l(nullptr), r(nullptr), dup(0)
27     {
28         #ifdef LOG_NODE_LIFECYCLE
29             std::cout<<"node ctor"<< static_cast<void*>(this) << std::endl;
30         #endif
31     }
32
33 private:
34     BNode(const BNode&);
35     BNode& operator=(const BNode&);
36     BNode(BNode&&);
37     BNode& operator=(BNode&&);
38     // [API]
39 public:
40     unsigned int get_duplication_count() const { return dup; }
41     bool has_left() const { return (l!=nullptr); }
42     bool has_right() const { return (r!=nullptr); }
43     bool is_leaf() const { return ((r==nullptr)&&(l==nullptr)); }
44     V get_value() const { return v; }
45     // [STATE]
46 private:
47     V v;
48     BNode* l;
49     BNode* r;
50     unsigned int dup;
51
52 friend class BTree<V>;
53 friend class SearchTree<V>;
54 friend class LZWTree<V>;
55 };
56
57 template<typename V>
58 class BTree
59 {
60 protected:
61
```



```
62 // [LIFECYCLE]
63 public:
64     BTree() : root(nullptr), cur(root)
65     {
66         #ifdef LOG_TREE_LIFECYCLE
67             std::cout<<"btree ctor "<< static_cast<void*>(this) << std::endl;
68         #endif
69     }
70
71     virtual ~BTree()
72     {
73         del_subtree(root);
74         #ifdef LOG_TREE_LIFECYCLE
75             std::cout<<"btree dtor "<< static_cast<void*>(this) << std::endl;
76         #endif
77     }
78
79     BTree(const BTree& old)
80     {
81         #ifdef LOG_TREE_LIFECYCLE
82             std::cout<<"btree copy ctor "<< static_cast<void*>(this) << std::endl ↵
83             ;
84         #endif
85         root = deep_copy(old.root ,old.cur);
86     }
87
88     BTree& operator=(const BTree& old)
89     {
90         #ifdef LOG_TREE_LIFECYCLE
91             std::cout<<"btree copy assign "<< static_cast<void*>(this) << std:: ↵
92             endl;
93         #endif
94         BTree tmp(old);
95         std::swap(*this,tmp);
96         return *this;
97     }
98
99     BTree(BTree&& old)
100     {
101         #ifdef LOG_TREE_LIFECYCLE
102             std::cout<<"btree move ctor "<< static_cast<void*>(this) << std::endl ↵
103             ;
104         #endif
105         root = nullptr;
106         *this = std::move(old);
107     }
108
109     BTree& operator=(BTree&& old)
110     {
111         #ifdef LOG_TREE_LIFECYCLE
```

```
109     std::cout<<"btree move assign "<< static_cast<void*>(this) << std:: ←
        endl;
110     #endif
111     std::swap(old.root, root);
112     std::swap(old.cur, cur);
113     return *this;
114 }
115 // [API]
116 public:
117
118     virtual BTree& operator<<(V val){ return *this;}
119
120     template<typename F>
121     void foreach_inorder(F f){ foreach_inorder(0,root, f); }
122
123     template<typename F>
124     void foreach_postorder(F f){ foreach_postorder(0,root, f); }
125
126     template<typename F>
127     void foreach_preorder(F f){ foreach_preorder(0,root, f); }
128
129 private:
130     template<typename F>
131     void foreach_inorder(int depth, BNode<V>* n, F f)
132     {
133         if(n){
134             foreach_inorder(depth+1,n->r,f);
135             f(depth,n->v,n->dup, n->is_leaf());
136             foreach_inorder(depth+1,n->l,f);
137         }
138     }
139
140     template<typename F>
141     void foreach_postorder(int depth, BNode<V>* n, F f)
142     {
143         if(n){
144             foreach_postorder(depth+1,n->r,f);
145             foreach_postorder(depth+1,n->l,f);
146             f(depth,n->v,n->dup, n->is_leaf());
147         }
148     }
149
150     template<typename F>
151     void foreach_preorder(int depth, BNode<V>* n, F f)
152     {
153         if(n){
154             f(depth,n->v,n->dup, n->is_leaf());
155             foreach_preorder(depth+1,n->r,f);
156             foreach_preorder(depth+1,n->l,f);
157         }
158     }
```

```
158     }
159
160     void del_subtree(BNode<V>* n)
161     {
162         if(n) {
163             del_subtree(n->l);
164             del_subtree(n->r);
165             #ifdef LOG_NODE_LIFECYCLE
166                 std::cout<<"node delete "<< static_cast<void*>(n) << std::endl;
167             #endif
168             delete n;
169         }
170     }
171
172     BNode<V>* deep_copy(BNode<V>* old_n, BNode<V>* old_cur)
173     {
174         BNode<V>* newn = nullptr;
175         if(old_n) {
176             newn = new BNode<V>(old_n->v);
177             newn->l = deep_copy(old_n->l, old_cur);
178             newn->r = deep_copy(old_n->r, old_cur);
179             if(old_n==old_cur) {this->cur=newn;}
180         }
181         return newn;
182     }
183
184     // [STATE]
185     protected:
186         BNode<V>* root;
187         BNode<V>* cur;
188     };
189
190     template<typename V>
191     class SearchTree : public BTree<V>
192     {
193     // [LIFECYCLE]
194     public:
195         SearchTree() : BTree<V>(){};
196
197     // [API]
198         virtual SearchTree& operator<<(V val) override
199         {
200             // invalid states
201             if( this->root == nullptr && this->cur != nullptr
202                 || this->root != nullptr && this->cur == nullptr)
203                 {return *this;}
204             // valid, but tree uninited, so we inject root
205             if(this->root == nullptr){ this->cur = this->root = new BNode<V>(val); ←
206                 return *this; }
207             // valid, tree inited
```

```
207     if(val == this->cur->v) {
208         this->cur->dup++;
209     }else{
210         if(val<this->cur->v) {
211             // left
212             if(this->cur->l == nullptr){
213                 this->cur->l = new BNode<V>(val);
214             }else{
215                 this->cur = this->cur->l;
216                 *this<<val;
217             }
218         }else{
219             // right
220             if(this->cur->r == nullptr){
221                 this->cur->r = new BNode<V>(val);
222             }else{
223                 this->cur = this->cur->r;
224                 *this<<val;
225             }
226         }
227     }
228     this->cur = this->root;
229     return *this;
230 }
231 private:
232
233 // [STATE]
234 private:
235
236 };
237
238 template<typename V>
239 class LZWTree : public BTree<V>
240 {
241     // [LIFECYCLE]
242 public:
243     LZWTree(V vr, std::function<bool(V)> predicate_is_left)
244     : BTree<V>(), vr(vr),predicate_is_left(predicate_is_left)
245     {
246         #ifdef LOG_TREE_LIFECYCLE
247             std::cout<<"lzwree ctor "<< static_cast<void*>(this) << std::endl;
248         #endif
249     };
250
251     virtual ~LZWTree()
252     {
253         #ifdef LOG_TREE_LIFECYCLE
254             std::cout<<"lzwree dtor "<< static_cast<void*>(this) << std::endl;
255         #endif
256     }
```

```
257
258
259 // [API]
260 virtual LZWTree& operator<<(V val) override
261 {
262     // invalid states
263     if( this->root == nullptr && this->cur != nullptr
264         || this->root != nullptr && this->cur == nullptr)
265     {return *this;}
266     // valid, but tree uninited
267     // we just inject first the root value
268     if(this->root == nullptr){ this->cur = this->root = new BNode<V>(vr); }
269     // valid, tree initied
270     if(predicate_is_left(val)){
271         // left
272         if(this->cur->l == nullptr){
273             this->cur->l = new BNode<V>(val);
274             this->cur = this->root;
275         }else{
276             this->cur = this->cur->l;
277         }
278     }else{
279         // right
280         if(this->cur->r == nullptr){
281             this->cur->r = new BNode<V>(val);
282             this->cur = this->root;
283         }else{
284             this->cur = this->cur->r;
285         }
286     }
287     return *this;
288 }
289 // [STATE]
290 private:
291     V vr;
292     std::function<bool(V)> predicate_is_left;
293 };
294
295 template<typename V>
296 void printer(int depth, V node_value, int duplicate_count, bool leaf)
297 {
298     for(int i = 0; i < depth+1; ++i)std::cout<<"---";
299     std::cout<<node_value<<" ("<<depth<<")"<<std::endl;
300 };
301
302 typedef struct depthdata_t{
303     int leaf_count;
304     int max;
305     double avg;
306     double dev;
```

```
307 }DepthData;
308
309 template<typename V>
310 std::vector<int> collect_depths(BTree<V>& tree){
311     std::vector<int> depths;
312     auto leaf_depth_counter = [&depths](int depth,V node_value, int ←
        duplicate_count, bool leaf)
313     {
314         if(leaf) depths.push_back(depth);
315     };
316     tree.foreach_inorder(leaf_depth_counter);
317     return std::move(depths);
318 }
319
320 DepthData compute_depth_data(const std::vector<int>& depths)
321 {
322     const int leaf_count = depths.size();
323     if(leaf_count < 1){return {.leaf_count=0, .max=0, .avg=0, .dev=0};}
324     int max = 0;
325     double avg = 0.0;
326     auto calc = [&max, &avg](const int& n) {if(max<n){max=n;} avg+=n;};
327     std::for_each(depths.begin(), depths.end(), calc );
328     avg=avg/leaf_count;
329     auto calc_dev_sq=[&leaf_count](double sum, double depth){ return sum+ ←
        depth/leaf_count;};
330     double dev = std::accumulate(depths.begin(), depths.end(), 0.0, ←
        calc_dev_sq);
331     if(leaf_count > 1){dev = sqrt( dev/(leaf_count-1));}
332     else{dev = sqrt( dev );}
333     return {.leaf_count=leaf_count, .max=max, .avg=avg, .dev=dev};
334 }
335
336 void print_depth_data(const DepthData& dd)
337 {
338     std::cout
339     << "Leaf " << dd.leaf_count << std::endl
340     << "Max " << dd.max << std::endl
341     << "Avg " << dd.avg << std::endl
342     << "Dev " << dd.dev << std::endl;
343 }
344
345 void test_binary();
346
347 void test_binary_mv_cp();
348
349 void test_lzw_char();
350
351 void test_lzw_str();
352
353 void test_lzw_cp_mv();
```

```
354
355 int main(int argc, char** argv, char** env)
356 {
357
358     test_lzw_cp_mv();
359 }
360
361 void test_binary()
362 {
363     SearchTree<int> bt;
364     bt<<8<<9<<5<<2<<7<<9;
365     std::cout << "BIN INORDER" << std::endl;
366     bt.foreach_inorder(printer<int>);
367     print_depth_data(compute_depth_data(collect_depths<int>(bt)));
368     std::cout << "BIN POSTORDER" << std::endl;
369     bt.foreach_postorder(printer<int>);
370     std::cout << "BIN PREORDER" << std::endl;
371     bt.foreach_preorder(printer<int>);
372 }
373
374 void test_binary_mv_cp()
375 {
376     SearchTree<int> zt;
377     zt <<8<<7<<10<<9<<11;
378     zt.foreach_inorder(printer<int>);
379     SearchTree<int> zt2(zt);
380     SearchTree<int> zt3;
381     zt3 <<11<<8<<7<<9<<10;
382     std::cout<<"zt = zt3"<<std::endl;
383     zt = zt3;
384     std::cout<<"std::move(zt2)"<<std::endl;
385     SearchTree<int> zt4 = std::move(zt2);
386 }
387
388 void test_lzw_char()
389 {
390     auto pred_left = [](std::string s){ return (s.compare("0")==0);};
391     LZWTre<std::string> bt("/",pred_left);
392     bt <<"0"<<"1"<<"1"<<"1"<<"1"<<"0"<<"0"<<"1"<<"0"<<"0"<<"1"<<"0"<<"0"<<"1" ←
        <<"0"<<"0"<<"0"<<"1"<<"1"<<"1";
393     std::cout << "LZW STR INORDER" << std::endl;
394     bt.foreach_inorder(printer<std::string>);
395     print_depth_data(compute_depth_data(collect_depths<std::string>(bt)));
396     std::cout << "LZW STR POSTORDER" << std::endl;
397     bt.foreach_postorder(printer<std::string>);
398     std::cout << "LZW STR PREORDER" << std::endl;
399     bt.foreach_preorder(printer<std::string>);
400 }
401
402 void test_lzw_str()
```

```
403 {
404     auto pred_left = [](char c){ return (c=='0');};
405     LZWTree<char> bt('/',pred_left);
406     bt <<'0'<<'1'<<'1'<<'1'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1'<<'0'<<'0'<<'1' ←
        <<'0'<<'0'<<'0'<<'1'<<'1'<<'1';
407     std::cout << "LZW CHAR INORDER" << std::endl;
408     bt.foreach_inorder(printer<char>);
409     print_depth_data(compute_depth_data(collect_depths<char>(bt)));
410     std::cout << "LZW CHAR POSTORDER" << std::endl;
411     bt.foreach_postorder(printer<char>);
412     std::cout << "LZW CHAR PREORDER" << std::endl;
413     bt.foreach_preorder(printer<char>);
414 }
415
416 void test_lzw_cp_mv()
417 {
418     auto pred_left = [](char c){ return (c=='0');};
419     LZWTree<char> zt('/',pred_left);
420     zt <<'0'<<'0'<<'0'<<'0'<<'0';
421     std::cout << "zt print"<<std::endl;
422     zt.foreach_inorder(printer<char>);
423     LZWTree<char> zt2(zt);
424     std::cout << "zt2 print"<<std::endl;
425     zt2.foreach_inorder(printer<char>);
426     LZWTree<char> zt3('/',pred_left);
427     zt3 <<'1'<<'1'<<'1'<<'1'<<'1';
428     std::cout << "zt3 print"<<std::endl;
429     zt3.foreach_inorder(printer<char>);
430     std::cout<<"zt = zt3"<<std::endl;
431     zt = zt3;
432     std::cout << "zt print"<<std::endl;
433     zt.foreach_inorder(printer<char>);
434     std::cout<<"std::move(zt2)"<<std::endl;
435     LZWTree<char> zt4 = std::move(zt2);
436     std::cout << "zt4 print"<<std::endl;
437     zt4.foreach_inorder(printer<char>);
438 }
```


7. fejezet

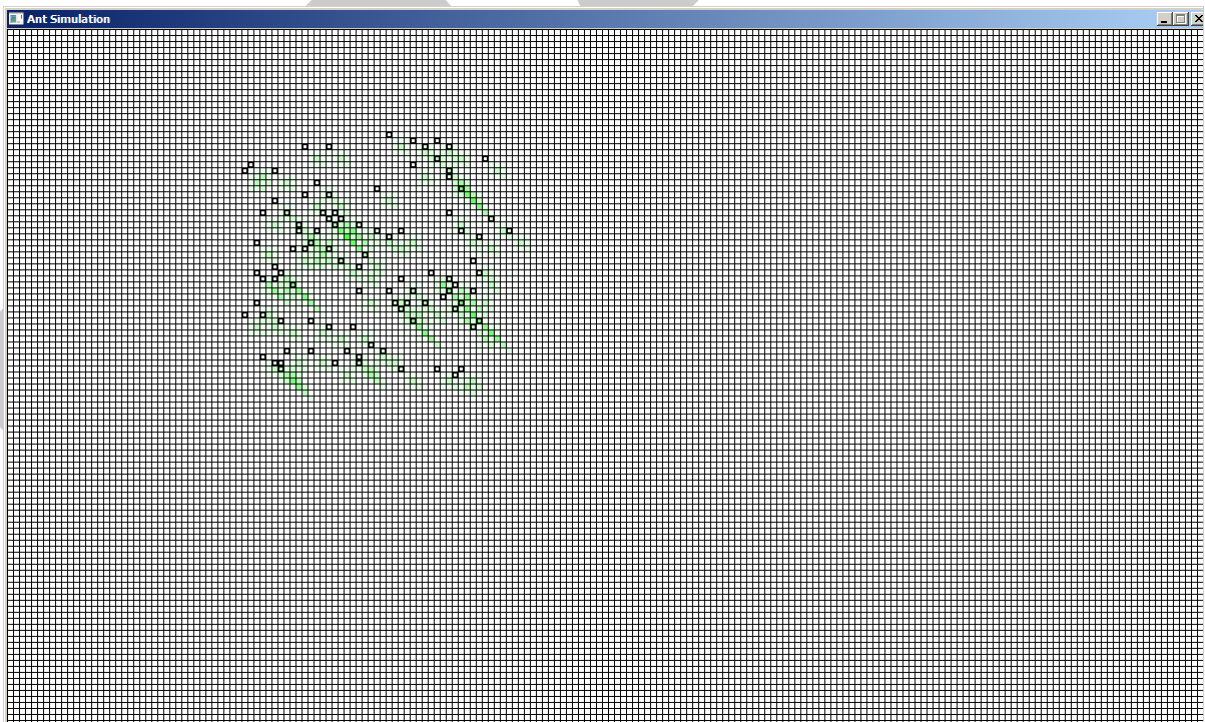
Helló, Conway!

7.1. Hangyaszimulációk

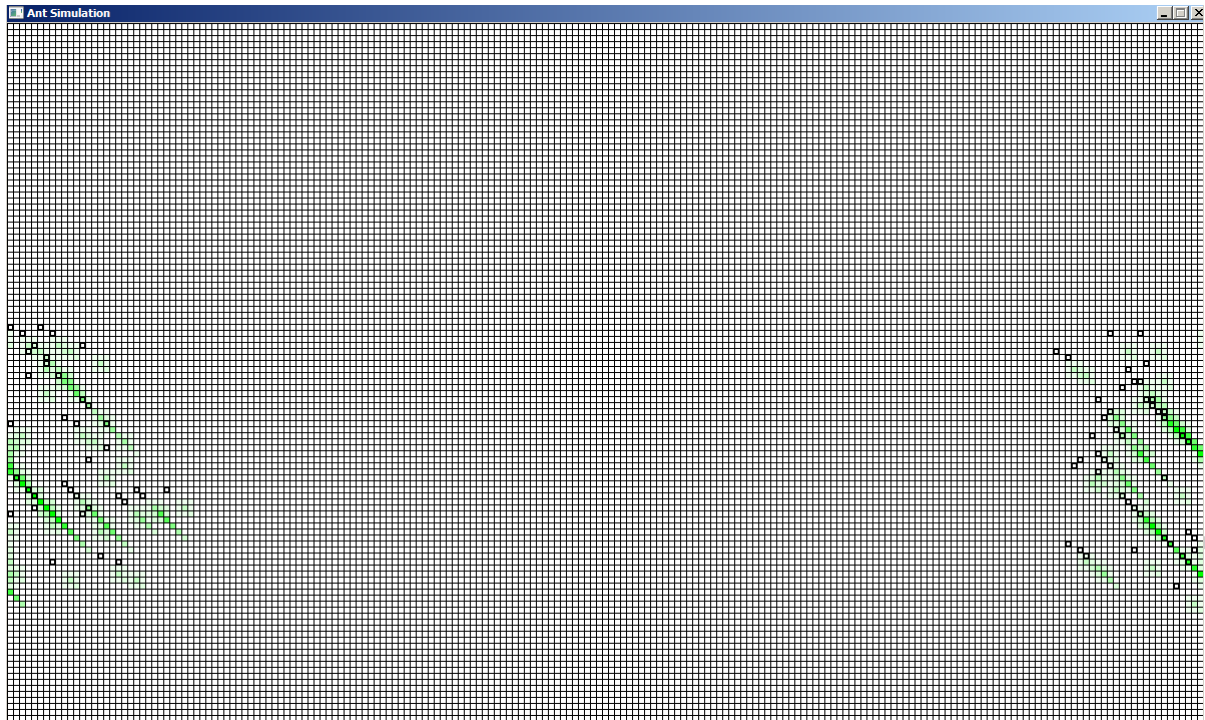
Írj Qt C++-ban egy hangyaszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Az nbatfais alap project fájlt pl módosítani kell. Project file-hoz például `QMAKE_CXXFLAGS += -std=c++11` lehet definiálni milyen flag-eket adunk hozzá. Mivel a project file alapján generálja a makefile-t, ezért érdemes nem kihagyni a flag-eket különben compilertől függő mértékben lesz sírás.

Miután meg van a makefile compile. Az applikációról alább két kép.



7.1. ábra. qt alap hangya 0



7.2. ábra. qt alap hangya wraparound

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

Megoldás forrása: https://gitlab.com/nbatfai/bhax/-/tree/master/attention_raising%2FMyrmecologist

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

7.2. Java életjáték

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

Megoldás forrása: <https://regi.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html#conway>

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

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás forrása: https://bhaxor.blog.hu/2018/09/09/ismerkedes_az_eletjatekkal

Megoldás videó: a hivatkozott blogba ágyazva.

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

7.4. BrainB Benchmark

Megoldás videó: initial hack: <https://www.twitch.tv/videos/139186614>

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

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

7.5. Vörös Pipacs Pokol/19 RF

Megoldás videó: <https://youtu.be/VP0kfvRYD1Y>

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

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

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. Programozási Alapfogalmak

10.1.1. Gépi kód, assembler, magasabb szintű nyelvek

Mielőtt beszélünk a típusokról nézzünk egy nem típusos nyelvet! A [\[NANDTOTETRIS\]](#)-hez írtunk egy c++ interpreter jellegű programot, mely úgy viselkedik mintha egy vlós CPU lenne(csak jóval egyszerűbbek az opcode-ok). A lényeg, hogy 2 regiszter van. A és D. A "Adress" ugyanis a jump-ok mindig az A-ban lévő értékű címre ugranak. D "Data" register pedig egy "sima" regiszter. "A" regiszterrel a trükközés azért kell, mert így borzasztóan egyszerűvé válnak az opcode-ok. 0-kat és 1-eseket nem akarunk írni, ezért írtunk rá egy assemblert. Alább látható egy szuper egyszerű assembly kód erre a teljesen minimalista kis gépre.

```
@2
D=A
@3
D=D+A
@0
M=D
```

Direkt a fenti egyszerűbbet mert látható a példán egy gcc -S -el készült sima main-ből ez lesz a körítés miatt.

```
1  .file "one.c"
2  .def __main; .scl 2; .type 32; .endef
3  .text
4  .globl main
5  .def main; .scl 2; .type 32; .endef
6  .seh_proc main
7  main:
8      pushq %rbp
9      .seh_pushreg %rbp
10     movq %rsp, %rbp
11     .seh_setframe %rbp, 0
12     subq $32, %rsp
13     .seh_stackalloc 32
```

```
14 .seh_endprologue
15 call __main
16 movl $1, %eax
17 addq $32, %rsp
18 popq %rbp
19 ret
20 .seh_endproc
21 .ident "GCC: (Rev1, Built by MSYS2 project) 7.2.0"
```

```
1 CHIP ALU {
2     IN
3         x[16], y[16], // 16-bit inputs
4         zx, // zero the x input?
5         nx, // negate the x input?
6         zy, // zero the y input?
7         ny, // negate the y input?
8         f, // compute out = x + y (if 1) or (x and y) (if 0)
9         no; // negate the out output?
10
11     OUT
12         out[16], // 16-bit output
13         zr, // 1 if (out == 0), 0 otherwise
14         ng; // 1 if (out less than 0), 0 otherwise
15
16     PARTS:
17
18     // X INP
19     Mux16(a=x,b[0..15]=false,sel=zx,out=xt);
20     Not16(in=xt,out=xtn);
21     Mux16(a=xt,b=xtn,sel=nx,out=xarg);
22     // Y INP
23     Mux16(a=y,b[0..15]=false,sel=zy,out=yt);
24     Not16(in=yt,out=ytn);
25     Mux16(a=yt,b=ytn,sel=ny,out=yarg);
26     // F
27     And16(a=xarg,b=yarg,out=oand);
28     Add16(a=xarg,b=yarg,out=oadd);
29     Mux16(a=oand,b=oadd,sel=f,out=o);
30     // OUT POST
31     Not16(in=o,out=onot);
32     Mux16(a=o,b=onot,sel=no,out=oo);
33     // ZR
34     Or16Way(in=oo,out=tzr);
35     Not(in=tzr,out=zr);
36     // NG
37     And16(a[0..15]=true,b=oo,out[15]=ng,out[1..14]=drop);
38     // OUT
39     Or16(a=oo,b[0..15]=false,out=out);
40 }
```

A lényeg, hogy az assembler (label és egyéb dolgok mellett) elsősorban azt a célt szolgálja, hogy a fenti szöveg átforduljon bytecode-ra. Alább látható a fordított gépkód.

```
00000000000000010
1110110000010000
0000000000000011
1110000010010000
0000000000000000
1110001100001000
```

Sajnos, el kellett engednem a teljes leírást, mert egyszerűen nincs rá idő, de komolyan ajánlom mindenkinek a [\[NANDTOTETRIS\]](#)-t. Alább például látszik egy szuper bugyuta kis ALU. Akármennyire bugyuta és tele van csalással a lényeg, hogy közelebb visz a szoftver és hardware találkozásához, ahol az igazi varázslat történik. (Hisz papíron ugyan Gödel megcsinálta, de sok idő kellett mire mindekinek lett macskáskép nézegetője.) Másrésztől nincs jobb érzés, mint amikor megcsinálja az ember a kapukat, majd ráküldi a kódot és megtudja vele csinálni a "for"-t! Tényleg fáj a szívem hogy nincs módomban berakni a doksiba. De őszintén ajánlom a könyvet, mert valójában az NEM EGY KÖNYV. Minden fejezet egy minimális elméleti alapozó és utána szuper egyértelmű task-ok vannak, TESZTEKKEL és platformmal együtt. Annyi, hogy én nem szeretem a Java-t mert az Oracle gonosz, ezért csak az assembler-es részt rossz minőségű c++-ban reprodukáltam a [ide](#).

Ha valakit abszolút nem érdekel a dolog, akkor is egyszer javaslom, csak amiatt, hogy átérezzük, hogy mennyire komoly segítséget adnak a mérnökinfósok és villamos mérnökök nekem illetve nekünk.

Ha pedig valakit a mérnökinfósok sem érdekelnek és nem szeret olvasni, legalább vessen egy pillantást [erre](#).

Nincsenek típusok, minden "szó" N mennyiségű bitből álló rendezett 16-os. Műveleteket nem definiálhatunk magunk, hisz azt a CPU csinálja.

Innentől kezdve bármit tanulunk emlékezzünk arra, hogy hasonló lesz a vége. (Persze a valóságban jóval összetettebb, de 1-esek és 0-k lesznek a legadvancedebb cpp kódból a nap végén.)

A típus megadja a gépnek hogy mikor írtunk egy programot és ráengedjük a lexert, parsert, compilert vagy interpretert akkor mit fogadjon el egyáltalán. Azaz hogy milyen elemei lehetnek. Azaz a típus egy halmazként is felfogható, melynek elemei a lehetséges értékek. Halmazoknál ugye felsorolhatjuk, de akár ha pro-k vagyunk szabályokkal is megadhatjuk (emlékezzünk a természetes számok halmazán successor-ra, vagy akár a modulo kongruencia osztályokra egészeknél)

A típus megadja a gépnek hogy milyen műveleteket és hogyan kell végezni. Például egy bool-t ha negálunk más történik, mintha egy int-et. Sőt, sokszor nem is lehet bizonyos dolgokat értelmezni, például Várterész Tanárnő nem nagyon szorozgatott igaz-t hamis-sal (majd később belemegyünk a szorzásba, most simán csak gondoljunk gyerekkorunkban tanultakra).

A típus megadja a gépnek hogy hogyan kell interpretálni az adatot. Például gondoljunk egy egyszerű C struct-ra, van két char fieldje "foo" és "bar". Elrakjuk valahova a memóriába (és tároljuk a címét), majd kis idő múlva kellene az "b" field. Honnan fogjuk tudni, hogy a sok bit közül hol kezdődnek a "b" field bitjei illetve, hogy hány bitből is áll? Például erre (is) ad választ a primitív char típus.

Amit még nagyon fontos lefektetni, az az hogy inheritance, primitív típus, template mind csak fluff és eyecandy a CPU szempontjából. Előbb utóbb mindenből 0101 0011 1111 0000 lesz. Igen igen 32 64, plusz valójában nem egy szó kerül be stb. de a lényeg hogy mindent számokra képezünk le. Az összes többi dolog csak és kizárólag azért kell, mert az ember biológiailag nem 0 és 1 olvasásra és nagy sebességű aritmetikai műveletek elvégzésére fajlódott hanem az ágakon tekergő kígyók elől való elugrálásra.

A tankönyv említi hogy forrásszöveget írunk, amelyből aztán két mágiával lehet gépi kód. Compiler-es és interpreter-es. Ez a valóságban sajnos nem ilyen egyértelmű. Nézzünk például egy Java-s példát. Igen compiled, de...mégis a VM stack machine-en fut. A stack machine csak egy absztrakció, nem a tiszta vas. Ez is a középpontja az Java azon ígéretének mi szerint "write once, run before Oracle sues you for using VM without paying your subscription for server side usage".

Másik Java példa: Project Lombok. Fel annotáljuk meta nyelven a forrás szöveget, és a class file-ba belegeneráljuk a boilerplate code-ot, anélkül, hogy telenyomnánk vele a source-t.

Másik Java példa: Spring, xml vagy reflection (annotation) based meta adatok. Igen a forrás fájl része, de egy framework használja az adatokat...

És a akkor a kedvencem: Írok egy progit C-ben. Mondjuk egy macskakép játék. A business logic-ot direkt C helyett Lua-ban írom, magyarul a C programom tartja számon a Lua state-t. A programom compiled, viszont ha a lua szkriptet változtatom alatta akkor gond nélkül hot swappelhetem mondjuk szerver oldalon. Most akkor része a business logic a programomnak? Vagy a programom egy hyper program ami önmaga nem a macskás játék? De hát a grafikus funkciók C-ben vannak írva? A lényeg, hogy nem ilyen egyértelmű a dolog.

Arról pedig már ne is beszéljünk, amit egy JIT compiler egy átlagos hétfő délután csinál.

A tankönyv ezután belemegy a fordító programok világába. Ez ahogy láttuk nem egy merev dolog, de ennél még rosszabb is történhet. Egyes "compiler"-ek azért vannak hogy C-kódot generáljanak valami deklaratív jellegű nyelvből. De ennél még rosszabb, hogy van aminek az a célja hogy C kódot fordítson Javascriptre. A fordítás általános feladatai a tankönyv szerint a következők:

- lexikális elemzés
- szintaktikai elemzés
- szemantikai elemzés
- kódgenerálás

A könyv kiemeli, hogy lehet szó előfordítókról. Most egy tanulságos történet: Java spring-based web server. Hibernate előtti időszak, szóval perzisztenciát from scratch. Amerikaiak úgy döntöttek, hogy egy perzisztens class-t annotációkkal fognak "dekorálni" (ez akkor még nagy szó volt, mert ez még az xml config-os spring era), és technikai okokból, ha ez megtörtént, az annotációkban megjelölt információk alapján a SUPER class auto generálni fogjuk. Igen `Derived extends Base` és `Base` még nem volt kész, hanem `Derived` alapján jött létre "automatán" a `Derived` annotált source kód alapján. Például olyan célt szolgált, hogy a null check-ek validálások, `propertyeventchanged` küldések stb. ne kézzel íródjanak. **Ő az.** Az már egy másik cseresznye a tortán, hogy nem a hétköznapi módon csináltak `Product` táblát, `Employee` táblát stb., hanem például `TypeIdentifier`, `AttributeTypeIdentifier` és hasonló táblák voltak, azaz runtime lehetett új "típusokat" létrehozni, úgy hogy ezek nem csak a field-eket örökölték, hanem viselkedést is (igen a munka nagyrésze az application layerben ment).

A könyv kitér rá, és tényleg nagyon fontos a linker. Egyébként így elsőre prog 1-ből az lesz az előnye, hogy nem kell mindig az egészet újra fordítani. Persze ha nem kell mindig újra fordítani. Persze a compile-olgatásnak és linkelgetésnek is meg van a maga ára. Például ha ritkán változó dologról van szó, akkor lehet precompiled header-ekkel dolgozni.

Na jó...de mi ez az egész linkelés? Pl. C++ esetén a compilation unit `Foo`, illetve van egy `FooMain`-ünk ahol használjuk. Külön külön fordítjuk, és ha minden rendben akkor végül együtt kell működniük majd,

tehát linkelésnél valamilyen módon a FooMain beli használathoz társítani kell a Foo beli implementációt. A szerződés az együttműködésre Foo.hpp.

```
1 #ifndef FOO_H
2 #define FOO_H
3 class Foo{
4 public:
5     Foo(int v);
6     int getA() const;
7 private:
8     int a;
9 };
10 #endif
```

```
1 #include "Foo.hpp"
2
3 Foo::Foo(int v){
4     this->a = v;
5 }
6
7 int Foo::getA() const
8 {
9     return this->a;
10 }
```

```
1 #include "Foo.hpp"
2 #include <iostream>
3 int main(){
4     Foo foo(1);
5     std::cout << foo.getA() << std::endl;
6     return 0;
7 }
```

```
UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo
k_IgyNeveldaProgramozod/cbook (master)
$ g++ -c Foo.cpp

UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo
k_IgyNeveldaProgramozod/cbook (master)
$ g++ -c FooMain.cpp

UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo
k_IgyNeveldaProgramozod/cbook (master)
$ g++ -o FooMain.exe Foo.o FooMain.o

UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo
k_IgyNeveldaProgramozod/cbook (master)
$ ./FooMain.exe
1
```

10.1. ábra. Foo és Foo Main

Na jó... de mi történik, ha megváltozik az a field? Mi van ha kifele int-et mutatok, de valójában másképp akarom tárolni?

Amíg a hpp változatlan addig azt csinálom implementációban amit akarok!

De...szóval mi van ha esetleg az a implementációjához akarok hozzányúlni. Bad luck! Hpp-t módosítani kell és akkor már nem tudnak ellened linke...VÁRJUNK CSAK!

Egy kis trükközéssel encapsulating kivitelezhető ezen kívánságra is, csak kompozíciót kell alkalmazni és egy struct-ba wrappelni amit rejtetni kívánunk.

```
1 #ifndef FOOABI_H
2 #define FOOABI_H
3
4 #include <memory>
5
6 class FooABI{
7 public:
8     ~FooABI();
9
10    FooABI(const FooABI&) = delete;
11
12    FooABI& operator=(const FooABI&) = delete;
13
14    FooABI(FooABI&&) = delete;
15
16    FooABI& operator=(FooABI&&) = delete;
17 public:
18     FooABI(int v);
19
20     int getA() const;
21 private:
22     struct Impl;
23     std::unique_ptr<struct Impl> impl_;
24 };
25 #endif
```

```
1 #include "FooABI.hpp"
2
3 struct FooABI::Impl
4 {
5     Impl(int v) : a(v) {};
6     int a;
7 };
8
9 FooABI::FooABI(int v) : impl_(new Impl(v)) {
10
11 }
12
13 FooABI::~~FooABI() = default;
14
15 int FooABI::getA() const
```

```
16 {
17     return impl_>a;
18 }

1 #include "FooABI.hpp"
2 #include <iostream>
3 int main() {
4     FooABI fooabi(1);
5     std::cout << fooabi.getA() << std::endl;
6     return 0;
7 }
```

```
UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo
k_IgyNeveldaProgramozod/cbook (master)
$ g++ -c FooABI.cpp

UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo
k_IgyNeveldaProgramozod/cbook (master)
$ g++ -c FooABIMain.cpp

UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo
k_IgyNeveldaProgramozod/cbook (master)
$ g++ -o FooABIMain.exe FooABI.o FooABIMain.o

UBMCGU@UBMCGUPC MINGW64 ~/ws_school/bhax-derived/thematic_tutorials/bhax_textboo
k_IgyNeveldaProgramozod/cbook (master)
$ ./FooABIMain.exe
1
```

10.2. ábra. FooABI és FooABI Main

A nevekben felfedezhető ABI az Application Binary Interface szóra utal. Erről most nem írok részletesen, de mint ahogy az Application Programming Interface azaz API jó ha konzisztens tud maradni például egy library különböző verziói között, addig a gép számára hasonlóan jó dolog ha az ABI nem változik.

A tankönyvet régen írták, de igen, továbbra is nagyon fontos hogy ki hogyan és mihez kapcsolódik, viszont mivel telt múlt azóta az idő, vannak új trükkök!

A betöltés egy nagyon fontos dolog. Miért is? A gépekben az adat, az eljárás és a macskás képek nem különülnek el. Minden adat. Vicces túlzással élve a számítógép valójában egy ipari lyukasztó gép amit, nos lényegében lyukasztott kártyákkal programozunk, innentől kezdve ő kilyukasztja az utasítás kártyát ha kell, és ha kell lefuttatja a kilyukasztott végterméket egyaránt ha beadjuk neki. Számára lyuk-lyuk egyre megy.

Betöltés

C++, C, Java oldalról nehéz megérteni a betöltést. Nézzük assemblyvel egy egyszerű példán:

```
@0
D=M
@INFINITE_LOOP
D;JLE
@counter
```

```
M=D
@SCREEN
D=A
@address
M=D
(LOOP)
@address
A=M
M=-1
@address
D=M
@32
D=D+A
@address
M=D
@counter
MD=M-1
@LOOP
D;JGT
(INFINITE_LOOP)
@INFINITE_LOOP
0;JMP
```

A lényeg, hogy van egy LOOP label-em. Ez egy hely a kódban ahova ugorhatok. Hogy ugrom oda? Long story short @LOOP-al betárolom LOOP helyét majd JGT-vel ugrom.

Ok. Pszeudokódban megy, de...Mennyi is a LOOP label címe? Mármint konkrétan nekem kellene, hogy az most akkor 0111 0111 0111 1111? Honnan tudom?

Naív válasz: Oké, 0 memória címre lesz betöltve a programom, szóval simán kiszámolom hogy az @address(hisz oda fogok ugrani, mert a LOOP az csak egy sajtos papír "tag", tag alatt az angol tag-et értem)

Ez egy tökéletes megoldás lehet Nintendo-n, vagy nem tudom... valami ROM-on!

De akkor mi van, ha én nem oda kerülök, hanem mondjuk már előttem vannak dolgok, mondjuk egy macskakép sokszorosító?

Egyszerű megoldás: Akkor derüljön ki LOOP értéke, mikor én elhelyezésre kerülök! Zseniális!

Viszont...nos, innentől kezdve én elmozdíthatatlan vagyok! Pontosabban elmozdíthatnak, de mivel a LOOP egy konkrét érték, ezért ha arébb raknak, akkor rossz területre fog hivatkozni.

És akkor például itt jöhetnek trükkös megoldások a cím újra számításra, vagy esetleg arra, hogy én ne direktbe hivatkozzak egyenesen a fizikai címre, hanem magamhoz képest relatív.

Java esetben ez másképp van hiszen egy stack machine-be pakolunk dolgokat, ami az Oracle szerint write once run...

Persze a VM egyébként egy [nagyon jó dolog](#). Vagy például az eve online [Stackless Python](#)-t használ ami egyébként ugyanúgy a unmutabilityt választotta, hasonlóan az Erlang-hoz

Interpreternél ugye nincs szükség ekkora hercehurcára, kivéve ha van szükség. Mármint például egyes interpretált nyelveknél direkt egy előfordított formába rkhatjuk a szkriptet és akkor kicsit gyorsítani tudunk a dolgokon.

Interpreteres esetekben persze mindig ott a lehetőség, hogy a CPU intenzív dolgokat natívba rakjuk. Pl. a [dont starve](#) esetén Lua intézi az üzleti logikát, ami az állatok párzási időszakban erősödő agressziójáért felelős, de a grafika, fizika, collision C/C++ oldalon van tartva. Azért nem mondok tiszta Cpp-t, mert Lua raw c ptr-eket fogad, illetve C-s callbackek szolgálnak hook-ként a lifecycle eventekre (magyarul ha a Lua gc elakar takarítani valamit, és az egy küldő kódból származó raw ptr, akkor egy user defined c callback-et hív ezen ptr-el. Mi például itt tudjuk az átküldött címre hívni a destruktort explicit, utána visszakerül luanak az irányítás. Azért nincs free vagy delete, mert az is customizable, azaz lehet például, hogy mi írunk alá memory managementet, mert folyamatosan az OS-től kérni apró chunkokat elég lassú.).

Ezzel az egész résszel az volt a célom, hogy kifejezzem, hogy a könyv nagyon jó, de már régóta eltűntek azok az éles határok, illetve mivel nem láttam pontos definíciót ezért nehéz egyáltalán megtámadni is.

Természetesen a lexikális elemzés során megtörténik a forrás szöveg lexikális egységekre történő bontása. Ez ma is így van.

Egyébként viszont az is egy érdekes kérdés, hogy a Cpp type system az imperatív nyelvbe hogy kerül bele. Mármint arra a vicces dologra akarom felhívni a figyelmet, hogy mondjuk én egy extends-el egy abszolút nem imperatív dolgot csinállok, a type inference pedig...nos ennél kevésbé imperatív dolog nincs. Persze, igen, C-style cast.

Most bele lehetne menni szárazon a BNF-be, de ennél arányosabb a $Q = \{ "1", "3", "t", a++b \mid a, b \text{ eleme } Q \}$ Szóval [133t](#), de Várterész Tanárnő egyébként szó szerint ilyen "elemzést végzett", amikor felírta az ábécét és a szabályokat.

A szintaktikai szabályok kicsit hajlékonyak, például ha Tanárnő hiányzik, akkor emlékezzünk arra, hogy a `-Wpedantic`(pl: field initialization sorrend csak a deklarációs sorrendben megengedett)

Imperatív nyelveknél a programozó mondja meg hogy hogyan, ezért tele van bugokkal. Mellette szól viszont, hogy [gyorsabb kódot lehet így írni az elméletileg lehetségesnél](#)

Deklaratív nyelveknél a programozó nem mondja meg hogyan. Emiatt nincs hiba. Technikailag. De természetesen abszolút nem az fog történni amire az ember gondol és sok szerencsét a Prolog debuggolással.

Imperatív nyelvekhez még talán annyit, hogy...nos az OOP nagyon jó dolog. Bizonyos feladatokra. Folyamatos vessző paripám az [ECS](#). Nem ez nem egy nyelv, hanem egy megközelítési forma. Az egész arról szól hogy passzív adatstruktúráim vannak és a viselkedést megvalósító részeket megpróbálom (bár általában nehéz) állapot mentesíteni. Szerintem az OOP az emberi intuíciót és a problémákról történő gondolkodást elősegíti, de semmilyen bizonyíték nincs arra, hogy karbantartható codebase-hez vezet. (Például a [Tony Hawk](#)) Vannak olyan nyelvek melyek az interface-t (vagy teljesen absztrakt class-t) preferálják, és kigyomlázták a hétköznapi java-ban burjánzó inheritance fákat, erdőket.

A forrás fájlunkban emberileg értelmezhető szöveget írunk karakterek felhasználásával. A nyelv amin írjuk egy megfelelő abécéből (Logika emlékszünk?) és ezek alkalmazási szabályaiból áll

Lexikális egységek a következők

- többkarakteres szimbólum

Pl.: ++, --, azaz ahelyett hogy valami furcsa új szimbólumot használnánk technikailag több szimbólum együttese alkot együtt egy szimbólumot. Például gondoljunk arra, hogy egyszerű abécé betűkkel mondjuk csinálunk egy olyan nyelvet amibe kéne egzisztenciális kvantor, és azt találnánk ki, hogy az `ee` jelentse azt.

- szimbolikus név

Azonosító, adott nyelv szintaktikai szabályai szerint olykor például kötelezően betűvel kezdődő karakter sorozat. A lényeg hogy ezt használjuk a dolgot elnevezésére. Azaz például egy változóra hog hivatkozunk az azonosítóját írosgatjuk be a forrásszövege.

Kulcs szó, a nyelv által védett, különös jelentősséggel bíró név. Például a `for` mely érezhetően ahhoz kell, hogy jelezzük, hogy ami utána jön abból valami loop jellegű dolog legyen. Turing fejezetben konkrétan a nyelvi spec-cel is foglalkoztunk optional-östől mindenesetül, ezért erre most nem térek ki.

Standard azonosító, melynek a nyelv tulajdonít jelentést. Pl.: Null

- címke

Ugráláshoz kell, hogy hova ugorjunk! Hogy mondjuk meg hova kéne ugorni, mindezt menedzselhetően? Adjunk egy címkét annak a sornak ahova ugorni akarunk! Fentebbi assembly kódban a loop-olás miatt láttunk labelt.

- megjegyzés

- literál

Nyelv által megengedett szabályok szerinti karakterlánc mellyel konkrét értéket tudunk bevinni. Egzotikus példa a `c++` szabványos bit megadásos `0b01101010`.

Sorok fontossága szerint

- Kötétt formátumú nyelvek

Egy sor, egy utasítás. Legjobb példa az előbb már említett assembly (kis trükkal, hisz a label maga ugyan egy sor ott, de úgy írtam meg a parsert, hogy azt kikapja véglegesből hisz az csak meta adat :)

- Szabad formátumú nyelvek

Ok, ha az új sor nem a vége a dolognak, akkor viszont meg kell valahol állni, ugye?! Nos ezt vagy explicit jelezzük pl. `C`, vagy khm... `js!`.

10.1.2. Adattípusok

(28) tartomány, műveletek, reprezentáció, egyszerű, összetett, mutató

Típusok, röviden felfoghatóak: lehetséges értékek halmaza + halmaz elemeken értelmezhető műveletek + reprezentáció együttese ként.

- numerikus - pl.: `(C)int`, `(C)float`

- karakteres - pl.: `(C)char`

- karakterlánc - pl.: ezt általában nem igazi primitív, hanem a nyelv mellé adott alap library része, például Erlang-ban egyébként egy lista

- logikai - pl.: `(C) bool`

- felsorolásos - pl.: `(C) enum`

- sorszámozott - pl.: (Pascal) `byte`, `word`, `int`, ...

A nyelv által definiált egyszerű típusokból van lehetőség új strukturált típusok összerakására ezek az összetett típusok.

A típusnak pedig végül kell hogy legyen valami azonosítója, hogy tudjunk rá a későbbiekben hivatkozni.

A mutató típusról kicsit külön érdemes beszélni. A mutató egy tárbeli címre mutat vagy **NULL**-ra. Érdemes tudni, hogy a mutatott cím egyáltalán milyen típus. Azaz a `char* foo` ptr típusú, viszont amire mutat azt char-ként fogja "interpretálni". Persze egy int-et tároló mem területre rámehetünk egy char ptr-rel gond nélkül.

10.1.3. A nevesített konstans

(34) név, típus, érték

Szerintem a könyvbeli preprocesszor-os példa technikailag nem igaz. Az hogy a preproceszor mit csinál már a Turing-os fejezetben bemutattam, plusz mutattam olyat is amikor makró alkalmaz makrót ami kódot injektál. Na egy ilyen esetben látszik hogy a preprocesszor csak egy "szövegszerkesztő" eszköz, nem a nyelv része, plusz úgy ütöm felül a a define-okat ahogy akarom.

Ellenben a `const int =6;` egy konstans. típus, const qualifier, és értékadás, just like God intended. Ha már itt tartunk akkor itt a világ legmegbízhatóbb Java kódja, ami viszont olykor mégis hibát okozhat (assuming that it wont get optimized away): `private static void foo(){}; . He. He.`

10.1.4. A változó

(35) név, attribútumok, cím, érték

A változóknak négy komponense van:

- név - pl.: user defined, scope-on belül egyedi
- attribútumok - pl.: típus, vagy qualifiers
- cím - stack, heap vagy manual
- érték - értékadás, itt annyit érdemes megjegyezni, hogy attól hogy létrehozom és kap címet, azt nem lehet várni hogy a tár tiszta legyen, úgyhogy érdemes lehet initelni.
- felsorolásos - pl.: (C) enum
- sorszámozott - pl.: (Pascal) byte, word, int, ...

10.1.5. Alapelemek az egyes nyelvekben

(39) innen csak a C nyelvész rész persze

Aritmetikai típusok

- integrális - egész (int, short[int], long[int]) : signed unsigned-ról már turingban írtam példával együtt

- integrális - karakter (char)
- integrális - felsorolásos (enum)
- valós - (float, double, long double)

Származtatott típusok

- tömb

Memóriában garantáltan egymás mellé kerülő hasonló típusú értékek. Azért ilyen furcsán írtam le mert ez a lényeg. Miért ez a lényeg? Nos, ha array-be tárolok valami akkor a cache miatt array-t nagyon gyorsan tudok "végig iterálni". Ez BORZASZTÓAN jól tud jönni. Persze sajnos hátulütője, hogy előre tudni kell a hosszt amennyit foglalni akarunk...

- függvény
- mutató

Architektúrától függő. Hossza pl. 64 bit, 32 bit. Ez egy sima egyszerű szám. Csak azáltal lesz különleges hogy általában a számot RAM hozzáférésre használjuk, de tényleg csak egy szám (főként void ptr esetén, hisz ott már nem cipel maga mellé meta adatként a mutatott adat típusát illető komoly információt). Egyébként cpp-ben szoktuk nyers ptr-nek is nevezni, mert ma már csak akkor használjuk ha nagyon fontos.

- struktúra

A user a meglévő típusokból (akár újonnan létrehozott structokból) rakhat össze újakat. Az ABI esetén említett struct-os kód példa bemutatja hogyan lehet abuzálni.

- union

Ahelyett hogy beszélgetünk róla itt van valami production code-ből(random CAD program). Embrace the C!

```
1 typedef struct le_line
2 {
3     int type;
4     Le3dPnt end1;
5     Le3dPnt end2;
6 } LeLinedata;
7
8 typedef struct le_b_spline
9 {
10     int type;
11     int degree;
12     double *params;
13     double *weights;
14     Le3dPnt *c_pnts;
15     int num_knots;
16     int num_c_points;
17 } LeBsplinedata;
18
```

```
19 typedef struct le_circle
20 {
21     int         type;
22     Le3dPnt center;
23     Le3dPnt norm_axis_unit_vect;
24     double      radius;
25 } LeCircledata;
26
27 typedef union le_curve
28 {
29     LeLinedata      leline;
30     LeBsplinedata   le_b_spline;
31     LeCircledata     le_circle;
32 } LeCurvedata;
```

- void

10.1.6. Kifejezések

Kifejezésekkel a program egy adott pontján ismert értékekből újakat határozunk meg. Értékük és típusuk van.

Formálisan operátorból, operandusokból és csoportosító jelekből állnak (pl. zárójel).

Attól függően hány operanduson történik, beszélhetünk unáry, binary, ternary stb. kifejezésekről. (Hehe funkcionális nyelveknél currying...)

A kifejezések több fajta alakban leírhatóak. Suliban az egyiket szoktuk meg, a gépek meg egy másikat. :)

- prefix

(* 3 5)

- infix

(3 * 5)

- postfix

(3 5 *)

Amikor a kifejezés értéke meghatározódik, azt kiértékelésnek is lehet nevezni, de Turing-ban adtam példát lambda kalkulusban normál formára hozásra. (Hisz a kiértékelés az, hogy normálformára hozzuk, ami lambdában nem mindig jelenti azt hogy pl. egy "számot" kapunk, lehet egy lambda absztrakció lesz a vége.)

A műveletek végrehajtási sorrendje a következő lehet

- felírási sorrend - balról jobbra
- anti felírási sorrend - jobbról balra

- precedencia alapján

Igen van olyan nyelv ahol magát a precedenciát is meg lehet adni...sőt ha két argunk van akkor infixben is lehet használni... De ez persze nem a C :)

Ha infixnél nem vagyunk biztosak abban hogy mi is lesz a sorrend, zárójelezzünk...

Ahhoz hogy ki lehessen értékelni egy operátor alkalmazását operandusokon érdemes tudni az operandusokat. A C erről nem köt meg semmit szabványban.

Ahol logikai kifejezések szerepelnek, ott rövidzárral nem feltétlenül fut le az egész, hisz az eredmény tudható anélkül is. Például egy vicces nyelv független példa:

```
TömbNemÜres ÉS TömbNulladikElemeÖt  
HAMIS ÉS ? = HAMIS
```

Fontos hogy a két operanduson lehet-e alkalmazni az operátort. Ahhoz hogy ezt eldöntsük kell a típus.

Két programozási eszköz típusa azonos ha (...valóságban megint nem ennyire egyszerű de hadd menjen)

- deklaráció egyenértékőség
egyszerre ugyanazzal a típussal deklarálódtak
- név egyenértékőség
ugyanaz a típus név
- struktúra egyenértékőség
összetett típusúak és szerkezetük megegyezik

típuskényszerítéses nyelvnél ha különböző típusúak az operandusok, akkor type conversion lesz. Ilyen esetben a nyelv leszögezi hogy milyen konverziók történnek és hogyan.

típus egyenértékűségnél megtörténhet de... most ugorjunk el a könyvtől. A lényeg a következő: Ezeket a nyelveket régen találták ki, de a következőről van szó: Van mondjuk A típus és B típus, ezek mind egy meta C típusba tartoznak és a művelet definiálva van a C-n, szóval B-n és A-n is menni fog (nem sima inheritance-re gondolok C superrel, inkább olyan jellegű mint pl. a Bácsó Tanár Úr által említett Abel csoport). Amiatt van ez a kernel panic a fogalom tárban mert amikor ezek a nyelvek születtek akkor nagyon gyorsan jöttek ki az új architektúrák int, aztán long int is kellett stb. Szóval inkább arra koncentráltak hogy polcra kerüljön a termék.

Felfele cast-olni gond nélkül lehet(int->float), de lefele castolni(float->int) nem egyértelmű. Mármint nem csak annyi a kérdés, hogy befogok-e férni, hanem például ha A teljesen C "típusú" lenne akkor A "részhalmaza" lenne C-nek C pedig "A"-nak ez pedig nagyon nincs így. A tény hogy ez nincs normálisan formalizálva, nem customizable csak builtin az arra enged következtetni, hogy még mindig van hova fejlődni.

Konstans kifejezés (cpp-ben constexpr) compile time dől el. Ebből következik, hogy konstansokat vagy beégetett literálokat lehet benne használni csak.

A könyv most ad egy rövid leírást a lehetséges operátorokról és precedencia táblázatról. Én ebbe most nem megyek bele inkább itt egy példa egy másik nyelvből hogy lehet megadni a fent említett dolgokat:

```
infixr precedence nm1 nm2 ...
```

Annyit jelent hogy infix esetben right associative, utána egy szám amivel megadjuk a precedencia számát után pedig. Long story short, alább egy left associative infix dolog amire a "+" szimbólummal lehet hivatkozni. Huzzah!

```
infixl 5 _+_
```

10.1.7. Utasítások

Az utasítások az algoritmusok egyes lépéseit megadó egységek. A fordító ezek segítségével készíti a forrás kódunkból a tárgykódot.

- deklarációs
- végrehajtható

10.1.7.1. Deklarációs Utasítások

A deklarációs utasítások hatására nem generálódik tárgykód. Ezek magának a fordítónak kiadott utasítások. Pl.:szolgáltatás kérés, üzemmódot beállítás stb. Természetesen befolyásolják a generált tárgykódot, de önnön maguk nem kerülnek fordításra.

10.1.7.2. Végrehajtható Utasítások

Ezek tárgykóddá fordulnak deklarációsakkal ellentétben(...optimalizálás miatt ez nem teljesen 100%-ban igaz). Következő csoportokra bonthatjuk ezen nagy csoportot:

- Értékadó utasítás
- Üres utasítás
- Ugró utasítás
- Elágaztató utasítások
- Ciklusszervező utasítások
- Hívó utasítás
- Vezérlésátadó utasítások
- I/O utasítások
- Egyéb utasítások

Az ugró, elágaztató, ciklusszervező, hívó és vezérlésátadó utasítások az ún. vezérlési szerkezetet megvalósító utasítások. Az egyéb utasítás csoportba tartozóbból nincs ilyen C-ben, viszont egyes helyeken van belülük (pl. PL/I).

10.1.7.2.1. Értékadó utasítás

Feladata változókhoz érték rendelése, vagy ezen érték módosítása.

10.1.7.2.2. Üres utasítás

Az üres utasítás hatására a processzor egy üres gépi utasítást hajt végre. Inkább szemantikai okokból létezik egyes nyelvekben

10.1.7.2.3. Ugró utasítás

Ezen utasítással feltétel nélkül ugorhatunk egy label-el definiált pontjára a kódnak.

10.1.7.2.4. Elágaztató utasítások

10.1.7.2.4.1. Kétirányú elágaztató utasítás

Általában a `IF feltétel THEN tevékenység [ELSE tevékenység]` alakot követi.

A feltétel egy igaz-hamis eldönthető predikátum (vagy durva esetekben implicit type conversion-nel meg is lehet trükközni a dolgot)

A tevékenység nyelv függő. Van ahol csak és kizárólag egyetlen utasítás állhat itt. Máshol lehet több utasítás csoportosítására utasítás zárójeleket használni pl. `BEGIN END`, ezt így utasítás csoportnak nevezzük. Végül pedig C-ben például tevékenység helyén vagy egy utasítás vagy egy blokk állhat.

Ha nincs else ág akkor rövid alakról, else ág megléte esetén hosszú alakról beszélünk.

Természetesen egymásba ágyazhatóak az elágaztatások, és ilyenkor merül fel a csellengő else problémája. Ezt persze mindig ki lehet kerülni ha pedantikusan a hosszú alakot használjuk. Más esetben implementáció és nyelvtan függő módon kell eljárunk.

10.1.7.2.4.2. Többirányú elágaztató utasítás

A többirányú elágaztató utasítás arra szolgál, hogy a program egy adott pontján egymást kölcsönösen kizáró `n` tevékenység közül egyet végrehajtsunk. A végrehajtandó tevékenység kiválasztását egy konkrét kifejezés értékei szerint tesszük meg.

Nyelv függő. C Példa:

```
SWITCH (kifejezes) {  
CASE egesz_konstans_kifejezes : [ tevekenyseg ]  
[ CASE egesz_konstans_kifejezes : [tevekenyseg ] ]...  
[ DEFAULT: tevekenyseg ]  
};
```

A kifejezés típusának numerikus egészre konvertálhatónak kell lennie. Az ágak értékei nem tartalmazhatnak duplikációkat. A tevékenység végrehajtható utasítás, vagy blokk lehet. A `DEFAULT`-ág bárhol szerepelhet.

Kiértékelődik a kifejezés, majd szépen a forráskódi felírási sorrendben megpróbálunk végigmenni az összes ágon. Ha egy ág-hoz megadott egész konstans kifejezéssel megegyezik az érték, akkor azon ág tevékenysége végrehajthatódik. Ha nem volt egyezés, akkor a default ág végrehajthatódik. Ha nincs default ág, akkor üres utasítás hajthatódik végre. Mindez egyben azt is jelenti, hogy pl. BREAK-et használva egy ág tevékenységében el kell hagyni a switch-et.

10.1.7.2.5. Ciklusszervező utasítások

A ciklusszervező utasítások lehetővé teszik, hogy a program egy adott pontján egy bizonyos tevékenységet többször is megismételjünk. Ciklus fejből, magból és végből áll. Az ismétlést meghatározó információk vagy a ciklus fejben vagy a ciklus végben találhatóak. Maga az ismétlendő tevékenység a magban található. Két szélsőséges esetről külön beszélünk. Egyik az üres ciklus, mikor egyszer sem fut le a ciklus. A másik a végtelen ciklus, mikor a ciklus soha sem áll le.

A következő ciklusfajtákat különböztetjük meg:

- feltételes
- előírt lépésszámú
- felsorolásos
- végtelen
- összetett

10.1.7.2.5.1. Feltételes ciklus

Ennél a ciklusnál az ismétlődést egy feltétel határozza meg. A feltétel vagy a fejben vagy a végben van. Kezdő- és végfeltételes ciklusról beszélhetünk.

Kezdőfeltételes ciklus esetén először kiértékelődik a feltétel. Ha igaz(hamis, nyelv függő lehet) belépünk a magba és végrehajtjuk az ott írtakat, majd újra a feltétel kiértékelésre ugrunk és indulunk újra. Ha hamis(igaz, nyelv függő lehet) akkor nem lépünk be a magba plusz kilépünk a ciklusból.

Végfeltételes ciklus esetén először végrehajtjuk a magot, majd kiértékelődik a feltétel. Ha igaz(hamis, nyelv függő lehet) vissza ugrunk és újrazuk a mag végrehajtását. Ha hamis(igaz, nyelv függő lehet) akkor kilépünk a ciklusból. Azaz ezen esetben egyszer mindenképp lefut a mag.

10.1.7.2.5.2. Előírt lépésszámú ciklus

Ezen esetben a fejben találhatóak a végrehajtásra vonatkozó információk. Minden esetben tartozik hozzá egy ciklusváltozó. A változó által felvett értékekre fut le a ciklusmag. A változó az értékeit egy általunk megadott tartományból veheti föl. A változó bejárhatja a tartományt csökkenőleg vagy növekvőleg. Ha a változó nem akarjuk hogy felvegye az összes értéket a tartományból, akkor érdemes megadnunk lépésközt.

A ciklusváltozó típusa nyelvenként eltérő lehet. A lépésköz és ciklusváltozó típusa vagy megegyezőnek kell lennie, vagy konvertálhatónak kell lennie.

A ciklusváltozó értékének megadásához minden nyelv esetén megengedett a literál, változó és nevesített konstans. Egyes nyelveknél kifejezéssel is megadható.

A lépésköz előjele dönti el – ha pozitív, akkor növekvő, ha negatív, akkor csökkenő. Általában azok a nyelvek vallják ezt, melyekben a ciklusváltozó csak numerikus típusú lehet. Egyes nyelveknél külön alapszót kell használni.

A ciklusparaméterek egyes nyelvek esetén csak egyszer értékelődnek ki, míg más nyelveknél minden mag végrehajtás után.

Általában a ciklus végrehajtás vagy a feltétel nem teljesülése miatt ér véget, vagy akár a magban kiadott speciális utasítás miatt. A GOTO-val történő ciklusból történő kilépést nem tekintjük szabályszerűnek.

A ciklusváltozó értéke nyelv függő lehet a ciklus elhagyása után. Három eset van: értéke az utolsó amire lefutott a mag, értéke az utolsó amit felvett, undefined.

10.1.7.2.5.3. Felsorolós ciklus

A felsorolós ciklus az előírt lépésszámú ciklus egyfajta általánosításának tekinthető. Van ciklusváltozója, amely explicit módon megadott értékeket vesz fel, és minden felvett érték mellett lefut a mag. A ciklusváltozót és az értékeket a fejben adjuk meg, ez utóbbiakat kifejezéssel. A ciklusváltozó típusa általában tetszőleges. Nem lehet sem üres, sem végtelen ciklus.

10.1.7.2.5.4. Végtelen ciklus

Sem a fejben sem a végben nincs információ ciklus elhagyással kapcsolatban, tehát a magban kell lennie olyan utasításnak amely miatt eltudjuk hagyni a ciklust.

10.1.7.2.5.5. Összetett ciklus

Az előző négy ciklusfajta kombinációiból áll össze.

10.1.7.2.5.6. C példák

- `WHILE(feltétel) végrehajtható_utasítás`
- `DO végrehajtható_utasítás WHILE(feltétel);`
- `FOR([kifejezés1]; [kifejezés2]; [kifejezés3]) végrehajtható_utasítás`

10.1.7.2.6. Vezérlő utasítások C-ben

`CONTINUE` ciklus hátralévő utasításait nem hajtja végre és újra indul a feltétel kiértékelés stb.

`BREAK` szabályszerű kilépés a ciklusból magon belül kiadhatóan.

`RETURN` befejezteti a függvényt és visszaadja a kontrollt a hívónak.

10.1.8. A programok szerkezete

Az eljárásorientált programnyelvekben a program szövege többé-kevésbé független, önálló részekre, ún. programegységekre tagolható. Ezen kis egységekből nyelvtől és implementációtól függően három módon állhat össze a teljes program: Fizikailag önálló részek, Nem önálló részek (strukturáltan egymásba ágyazott) Fenti kettő kombinációja. Az eljárásorientált nyelvekben az alábbi programegységek léteznek:

- alprogram
- blokk
- csomag
- taszk

10.1.8.1. Alprogramok

Túl szép mondat hogy kihagyjam „Az alprogram az eljárásorientált nyelvekben a procedurális absztrakció első megjelenési formája, alapvető szerepet játszik ebben a paradigmában, sőt meghatározója annak.”

Ezen alprogramokat próbáljuk elszeparálni a lehető legjobban, például minden szükséges információt formális paraméterként átadni. A céunk ezzel, hogy működése legkevésbé függjön az őt körbevevő kontextustól, főként csak az általunk megadott (szűkebb) formális paraméterektől. Sok pozitív tulajdonsága lesz így az alprogramnak, de a legfőbb a mi szempontunkból a code reuse.

Formálisan fej (specifikáció), törzs (implementáció) és végből állnak.

- név
- formális paraméter lista
- törzs
- környezet

A név egy azonosító, a fejben szerepel.

A formális paraméter lista is a specifikáció része. A formális paraméter listában azonosítók szerepelnek(...ez szerintem nyelv függő...mármint pl. Haskell Agda totál másképp közelíti meg pattern matching miatt a dolgokat), ezek a törzsben saját programozási eszközök nevei lehetnek, és egy általános szerepkört írnak le, amelyet a hívás helyén konkretizálni kell az aktuális paraméterek segítségével.

A korai nyelvekben a formális paraméter listán csak a paraméterek nevei szerepelhettek. A mai modern nyelveknél azonban olyan egyéb információk mellyel a paraméterek viselkedését szabályozhatjuk. A formális paraméter lista kerek zárójelek között áll (általában). A nyelvek egy része szerint a zárójelek a formális paraméter listához, mások szerint a névhez tartoznak. A formális paraméter lista lehet üres is, ekkor paraméter nélküli alprogramról beszélünk.

A törzsben deklarációs és végrehajtható utasítások szerepelnek. A nyelvek egy része azt mondja, hogy ezeket el kell különíteni egymástól, tehát a törzsnek van egy deklarációs és egy végrehajtható része. Erre a fenti mondatra már effektíve ki is tértünk a C standard-ek összehasonlításánál. Más nyelvek szerint viszont

a kétféle utasítás tetszőlegesen keverhető. Az alprogramban deklarált programozási eszközök kívülről nem láthatóak azaz ezek az alprogram scope-jában élnek (aka variables declared in the block are considered to be local to that block and they are visible only in that scope or from another which has visibility on that scope). Persze global scope-ba is rakhatjuk a dolgokat, de ez általában nagyon erősen kerülendő.

Az alprogramoknak két fajtája van: eljárás és függvény.

Az eljárás a hatását a paramétereinek vagy a környezetének megváltoztatásával illetve a törzsben elhelyezett végrehajtható utasítások által meghatározott tevékenység elvégzésével fejezi ki.

A függvény elsődlegesen egy értéket szolgáltat vissza. Mellékhatás ha a függvény környezetét vagy paramétereit módosítja. Tiszta esetben ennyi a lényege és nincs mellékhatása. Sajnos a való életben főként a mellékhatásos verzióval futhatunk össze. Ez pontosan ugyanolyan veszélyes és megkerülhetetlen, mint az eljárás.

A könyv mellékhatás definíciója...mármint biztos jó...de szerintem eljárás maga és a függvény is, ha bármit módosít a környezetből paraméterből az már mellékhatás. Ezt arra alapozom, hogy pl. ha optimalizálni akarok egy kódot, akkor csak akkor hagyhatok ki dolgokat, ha tudom, hogy kihagyásukkal semmilyen módon nem változik a program eredménye és az állapot.

Az eljárást aktivizálásához utasításszerűen hívunk kell azt (gyakran még valami alapszó-t is elé kell írni). Például a lenti (absztrakt) módon:

```
[alapszo] eljárásnev(aktualis_parameter_lista)
```

Egy eljárás szabályosan befejeződik ha elérjük a végét. Másik út, hogy külön utasítással befejeztetjük, ez bárhol kiadható az eljárás törzsében. Szabályos befejezés esetén a program a hívást követő utasításon folytatódik. Általában nem szabályos befejezésnek tekintjük a következőket: A nyelvek általában megengedik, hogy GOTO-val kiugorjunk eljárásból.

Függvényt kifejezés alakban lehet hívni.

```
fuggvenynev(aktualis_parameter_lista)
```

A függvényhívás után normális befejeződést feltételezve a vezérlés a kifejezésbe tér vissza, és továbbfolytatódik annak a kiértékelése.

Egy függvény a következő módokon határozhatja meg a visszatérési értékét:

- A függvényneve változóként szabadon változtatható. Visszatérésnél utolsó felvett értéke lesz használva.
- A függvénynevének értéket kell adni, de nem használható szabadon. Visszatérésnél utolsó felvett értéke lesz használva.
- Külön befejeztető utasítással megadjuk az értéket és befejezzük a függvényt

A függvény szabályosan befejeződik, ha:

- elérjük a végét, és már van visszatérési érték
- Külön befejeztető utasítás, és már van visszatérési érték,
- Külön befejeztető utasítás ami meghatározza az értéket

Ha ezek nem állnak fenn, akkor nem szabályos a kilépés. Tehát pl.: GOTO szabálytalan.

Az eljárásorientált programozási nyelvekben megírt minden programban kötelezően lennie egy főprogramnak. Egy program szabályos befejeződése a főprogram befejeződésével történik meg, ekkor a vezérlés visszakérül az operációs rendszerhez.

10.1.8.2. Hívási lánc, rekurzió

Ha egy programegység meghív egy másikat majd az egy másikat stb. akkor kialakul egy call chain, hívási lánc. A hívási lánc első tagja mindig a főprogram. A hívási lánc minden tagja aktív de csak a legvégső fut éppen (mármint igazából a schedulertől függ hogy ez megfog-e történni valaha :). Mikor a hívási lánc egyik tagja befejeződik az előző programegység végrehajtási folytatódik.

Mikor egy aktív alprogramot hívunk meg, azt rekurciónak nevezzük. Rekurzió lehet közvetlen, mikor önmagát hívja az alprogram. Lehet közvetett, mikor a hívási lánc más tagja hívja az alprogramot.

Rekurzió általában átírható iteratív megoldássá. Iteratív jobb, hiszen kevesebb erőforrást igényel (rövidebb call chain, nem kell annyi mindent állapotot kimenteni és cipelni). Vannak olyan nyelvek amelyek főként a rekurziót preferálják, pl. Haskell.

10.1.8.3. Másodlagos belépési pontok

Egyes nyelvek megengedik, hogy egy alprogramot meghívni ne csak a fejen keresztül lehessen, hanem a törzsben ki lehessen alakítani ún. másodlagos belépési pontokat, így vagy a fejben megadott névvel vagy a másodlagos belépési pont nevével lehet hivatkozni az alprogramra.

10.1.8.4. Paraméterkiértékelés

Egy alprogram hívásakor a hívásban megadott aktuális paraméterek a formálisakhoz rendelődnek. Ezt paraméterkiértékelésnek hívjuk.

Hogy melyik formálishoz, melyik aktuális param rendelődik, több módon kezelhetik a nyelvek: Sorrendi, vagy név szerinti kötés (binding) A sorrendi esetében a lista beli helyük alapján történik az összerendelés. Név szerinti esetben a nevesített aktuális paramétereket explicit rendeljük össze a nevesített formálisakkal.

Nyelvtől eltérő hogy mennyi aktuális paramétert kell megadni. Alapesetben persze annyit, amennyi a formális elemszáma, de egyes nyelvekben lehet kevesebbet, és ekkor default értékek rendelhetőek a fennmaradó formálisakhoz. Van változó számú (nem rögzített) eset is. Var args.

Egyes nyelvek esetén az aktuális és formális paraméter típusának meg kell egyeznie. Más nyelvek esetén elegendő ha típuskényszerítéssel elérhető legyen a konverzió.

10.1.8.5. Paraméterátadás

Mikor egy alprogram(hívó) meghív egy másik alprogramot, olyankor a kommunikációt paraméterátadásnak nevezzük.

Paraméterátadás fajtái:

- érték szerinti

- cím szerinti
- eredmény szerinti
- érték-eredmény szerinti szerinti
- név szerinti
- szöveg szerinti

10.1.8.5.1. érték szerinti

A hívott területén a paramétereknek van címkomponensük. A hívó oldalon kell hogy legyen érték komponensük. Az hívó oldali érték átmásolódik a hívott oldali területre. A hívó nem fogja látni a változtatásokat saját oldalon.

10.1.8.5.2. cím szerinti

Hívott oldalon nincs címkomponens, viszont hívó oldalon van. Ezen esetben hívó oldalon hívás után látni fogjuk a hívó általi változtatásokat.

10.1.8.5.3. eredmény szerinti

Hívó oldalon és hívott oldalon egyaránt címkomponens. Hívott nem használja a hívott oldali címet, simán local formálisba dolgozik. Szabályos visszatéréskor, local formális értéke átmásolódik a hívott oldali címre.

10.1.8.5.4. érték-eredmény szerinti szerinti

Hívó oldalról átmásolódik az érték a hívott oldalra ez lesz a helyi kezdőérték. Szabályos visszatéréskor a hívott oldali végleges érték átmásolódik hívó oldalra.

10.1.8.5.5. név szerinti

A formális paraméter nevének összes előfordulása hívott oldalon átíródik adott szimbólummá.

10.1.8.5.6. szöveg szerinti

A név szerinti, egyetlen eltérés, hogy hívottban a név felülírás csak akkor fut le, amikor formális param első előfordulását elérjük.

10.1.8.6. A blokk

Programegységben helyezkedik el, azon kívül sohasem álhat. Van kezdete, törzse, vége. A kezdetet és a véget egy-egy speciális karaktersorozat vagy alapszó jelzi. A törzsben lehetnek deklarációs és végrehajtható utasítások.

Blokk általában úgy kezdődik hogy vagy kezdetére ér a végrehajtás, vagy GOTO-t követő ugrás.

Blokk általában úgy végződik hogy vagy végére ér a végrehajtás, vagy GOTO-t követő ugrással kiugrunk belőle.

Fő szerepe a nevek hatáskörének tárolása. ÉLETCIKLUS!!!! PROPER CLEANUP ORDER!!!! Gyakorlatilag enélkül nem lenne smart ptr :)

10.1.8.6.1. Hatáskör

A hatáskör a nevekhez kapcsolódó fogalom. Egy név hatásköre alatt értjük a program szövegének azon részét, ahol az adott név ugyanazt a programozási eszközt hivatkozza, tehát jelentése, felhasználási módja, jellemzői azonosak. A hatáskör szinonimája a láthatóság. Ez egyébként legegyszerűbben logikából lett bemutatva először a képzés során. Ott is például elég komoly átnevezéseket kellett olykor elvégezni, és ezek csak akkor történhettek meg, ha nem volt szabad és tudtuk mi köti pontosan.

A név hatásköre az eljárásorientált programnyelvekben a programegységekhez, illetve a fordítási egységekhez kapcsolódik.

Egy programegységben deklarált nevet a programegység lokális nevének nevezzük. Azt a nevet, amelyet nem a programegységben deklaráltunk, de ott hivatkozunk rá, szabad névnek hívjuk. (Emlékezzünk logikából is, hogy átnevezést nem hajhattunk pl ilyen esetben végre stb.)

Azt a tevékenységet, mikor egy név hatáskörét megállapítjuk, hatáskörkezelésnek hívjuk. Kétféle hatáskörkezelést ismerünk, a statikus és a dinamikus hatáskörkezelést.

Statikus esetben fordítás időben current scope-ban próbáljuk megtalálni az adott nevet. Ha ez nem található akkor addig keresünk a felsőbb scope-okban amíg meg nem találjuk, vagy ki nem érünk. (Példa, ctor arg "a" de class-nak van a data member-e. Mivel legszűkebb scope-omban keresek először ezért ha simán a-t írok be, akkor a ctor argra fogok hivatkozni huzzah.) Ha nem található a név a legkülsőben, akkor egyes nyelvek esetén hiba áll elő. Más nyelvek ilyenkor a legkülső scope-ban auto deklarálják.

Hatáskör befelé terjed kifelé soha.

A dinamikus hatáskörkezelés futás idejű. Ha nem talál egy nevet a futtató rendszer akkor a hívási láncban kezd visszafele keresni. Ilyen esetben a hatásköre egy névnek a programegység ahol deklaráltuk, és minden belőle induló hívási láncban elhelyezkedő programegység.

Az eljárásorientált nyelvek a statikus hatáskörkezelést valósítják meg. Az alprogramok formális paramétereire az alprogramra lokálisak. A kulcsszavak, mint nevek a program bármely pontjáról láthatók. A program egységek nevei azonban globális láthatóak. (azért ez sem így egy az egyben c++-ban)

10.1.8.6.2. Fordítási egység

Az eljárásorientált nyelvekben a program közvetlenül fordítási egységekből épül föl. Ezek olyan forrásszövegrészek, melyek önállóan, a program többi részétől fizikailag különválasztva fordíthatók le. Az egyes nyelvekben a fordítási egységek felépítése igen eltérő lehet. A fordítási egységek általában hatásköri és gyakran élettartam definiáló egységek is.

10.1.8.7. C

A C nyelv a function-t és a block-ot ismeri. Egy function a másikba nem ágyazható, de function-be block igen és block-ba block igen.

Block

```
{  
deklaraciok  
vegrehajthato_utasitasok  
}
```

Function

```
[típus] név([formális_paraméter_lista])  
block
```

Ha nem szerepel a típus, akkor az alapértelmezés int. Ha void a típus, akkor lényegében egy eljárásról van szó.

A fő program main, mely maga egy function.

Függvény befejeződik RETURN-el. Ezen esetben void típusú funckióval szokásos visszatérés, míg egyéb T típus esetén nem definiált visszatérési érték. Vigyázzunk ez C++-ban is maradt...

RETURN kifejezés; esetén a visszatérési érték a kifejezés kiértékelt értéke.

A formális paramétereket vesszővel elválasztva, explicite típust megadva tudjuk definiálni. A C-ben a programozó tud nem fix paraméterszámú függvényt deklarálni úgy, hogy megad legalább egy formális paramétert, és a formális paraméter listát ... zárja. Az üres formális paraméter listát explicit módon jelölhetjük a void alapszó megadásával.

Sorrendi kötés, típuskényszerítés (elég vad) és fix paraméterszám esetén számbeli egyeztetés van. A paraméterátadás érték szerinti.

A C-ben a fordítási egység a forrásállomány. Ez ún. külső deklarációkat (nevesített konstans, változó, típus, függvény) tartalmaz. A fordítási egység elején más olyan fordítási egységekre, amelyek eszközeit használni akarjuk, a `#include <forrásállománynév>` preprocesszor utasítással hivatkozhatunk.

A C a hatáskör és élettartam szabályozására bevezeti a tárolási osztály attribútumokat:

- extern
- auto
- register
- static

10.1.8.7.1. extern

A fordítási egység szintjén deklarált nevek alapértelmezett tárolási osztálya. Lokális neveknel explicit módon meg kell adni. Az ilyen nevek hatásköre a teljes program, élettartamuk a program futási ideje. Van automatikus kezdőértékük.

10.1.8.7.2. auto

lokális nevek alapértelmezett tárolási osztálya. Hatáskörkezelésük statikus, de csak a deklarációtól kezdve láthatók. Élettartamuk dinamikus. Nincs automatikus kezdőértékük.

10.1.8.7.3. register

Speciális auto, amelynek értéke regiszterben tárolódik, ha van szabad regiszter, egyébként nincs különbség.

10.1.8.7.4. static

Bármely névnél explicit módon meg kell adni. Hatáskörük a fordítási egység, élettartamuk a program futási ideje. Van automatikus kezdőértékük.

10.1.9. IO

Az I/O az a területe a programnyelveknek, ahol azok leginkább eltérnek egymástól. Az I/O platform-, operációs rendszer-, implementációfüggő. Egyes nyelvek nem is tartalmazznak eszközt ennek megvalósítására, eleve az implementációra bízzák a megoldást.

A perifériákkal való kommunikációról szól összességében az IO. Az IO hogy egységesen tudja tárgyalni a témát, azért az absztrakt állomány fogalmat vezeti be. A programnyelvi fogalom fedésben van ezen absztrakt állomány fogalommal. A programban a logikai állomány egy olyan programozási eszköz, amely nevesített, és amelynél az absztrakt állományjellemzők (rekordfelépítés, rekordformátum, elérés, szerkezet, blokkolás, rekordazonosító stb.) attribútumként jelennek meg. A fizikai állomány pedig az op rendszer szintű impl. azaz perifériákon megjelenő adatokat tartalmazó állomány.

Egy állomány funkció szerint lehet:

- input állomány: feldolgozás előtt létezik, nem változik
- output állomány: feldolgozás előtt NEM létezik, változik, írni lehet
- input-output állomány: általában létezik a feldolgozás előtt és után is, változik, olvasni és írni is lehet.

Az IO során tár és a periféria között adatokat mozgatunk. A tárban és periféria ábrázolási módja eltérhet, ez felveti egy fajta konverzió szükségességének kérdését, illetve specifikációjának mikéntjét. Az alapján hogy történik-e konverzió megkülönböztetünk: folyamatos módú (van konverzió), és bináris, rekordalapút (nincs konverzió).

Folyamatos módú esetben a periférián tárolt adatokat egy folyamatos karaktersorként értelmezzük. A tárban azonban a típusnak megfelelő ábrázolási mód szerint definiált bitsorozatok vannak.

Olvasáskor meg kell mondanunk, hogy a folytonos karaktersorozatot hogyan tördeljük fel olyan karaktercsoportokra, amelyek az egyedi adatokat jelentik, és hogy az adott karaktercsoport milyen típusú adatot jelent. Íráskor pedig azt kell meghatároznunk, hogy a tárban tárolt bitsorozatokot hogyan konvertáljuk át folytonos karaktersorozattá és milyen szabályokkal.

A fenti szabály megadása három módon történhet:

- formátumos módú adatátvitel
- szerkesztett módú adatátvitel
- listázott módú adatátvitel

Formátumos módú adatátvitelnél minden egyes egyedi adathoz a formátumok segítségével explicit módon meg kell adni a kezelendő karakterek darabszámát és a típust.

Szerkesztett módú adatátvitelnél minden egyes egyedi adathoz meg kell adni egy maszkot, amely szerkesztő és átvendő karakterekből áll. A maszk elemeinek száma határozza meg a kezelendő karakterek darabszámát. A szerkesztő karakterek megadják, hogy az adott pozíción milyen kategóriájú karakternek kell megjelennie. A többi karakter változtatás nélkül átvitelre kerül.

Listázott módú adatátvitelnél a típusra nincs explicit módon megadott információ. Viszont hogy hol van az egyedi adatok közt a határ azt magában a folyamatos karaktorsorban elhelyezett speciális karakterek jelentik.

Bináris módban a periféria és tár közötti kommunikáció során nem történik konverzió.

10.1.9.1. Állományok kezelése

Az állomány kezelés a következő lépésekre bontható:

- Deklaráció
- Összerendelés
- Állomány megnyitása
- Feldolgozás
- Lezárás

Deklarációnál a nyelv által megszabott módon be kell vezetnünk a logikai állományt, megfelelő attribútumokkal.

Összerendelés során a logikai állományt egy OS által kezelt fizikai állománnyal rendeljük össze. Ezekután a logikai állománnyal végzett tevékenységek az alatta lévő fizikaira fognak kihatni.

Állomány megnyitása során OS check-ek futnak le, hogy a logikai és fizikai állomány kompatibilis-e. Ezenkívül például regisztrálódik, hogy általunk használatban van az erőforrás. Másrészt olykor a funkció is itt tisztázódik (pl. read only célból nyitjuk).

A már nyitott állományból írhatunk, vagy olvashatunk. Az írást/olvasást végző eszköznél a fenti fejezetek alapján folyamatos módban meg kell adni az információkat a konverzióhoz.

A lezárás ismét operációs rendszer rutinokat aktivizál. Például itt veszi ki az OS az eddig általunk fogottnak jelzett állományokról, hogy elengedjük őket. A könyvtárak információinak aktualizálása ilyenkor történik meg. A lezárás során a logikai és fizikai állomány közti kapcsolat megszűnik. Általában a főprogram szabályos befejeződésekor az összes nyitott állomány bezáródik.

A programozási nyelvek a programozó számára megengedik azt, hogy input-output esetén ne állományokban gondolkozzon, hanem az írás-olvasást úgy képzelje el, hogy az közvetlenül valamelyik perifériával

történik. Ezt hívjuk implicit állománynak. A megfelelő logikai és fizikai állomány most is létezik standard nevekkkel és jellemzőkkel, de ezt a futtató rendszer automatikusan kezeli. T ehát az implicit állományt nem kell deklarálni, összerendelni, megnyitni és lezárni. Az implicit input állomány a szabvány rendszerbemeneti periféria (általában a billentyűzet), az implicit output állomány a szabvány rendszerkimeneti periféria (általában a képernyő). A programozó bármely állományokkal kapcsolatos tevékenységet elvégezhet explicit módon (pl. az implicit output állományhoz hozzárendelheti a nyomtatót). Ha az író és olvasó eszközben nem adjuk meg a logikai állomány nevét, akkor a művelet az implicit állománnyal történik. Implicit helyett olykor érdemes lehet direkt explicitben nyitni, ugyanis ilyenkor nagyobb kontrollunk a módról.

10.2. Programozás bevezetés

[KERNIGHANRITCHIE] (2nd edition)

Előző alfejezetben a [JUHASZ] könyv kapcsán már lementünk assemblyig és vissza, szóval itt nem fogunk újra arról beszélni hogy mi az a bool, csak az új dolgok.

10.2.1. Alapismeretek

A legegyszerűbb C program, gcc hw.c-vel compile-olunk.

```
main ()  
{printf ("Hello World\n");}
```

Ez a fejezet elég alap dolgokkal foglalkozik, inkább csak a lényeges részeket emeletem ki csak. (Pl. for ciklusról volt egy pár mondatos rész, de erre úgylis egy egész afejezet lesz stb...)

10.2.1.1. Adattípusok

C-beli alaptípusok

- int - egész szám
- float - lebegőpontos szám
- char - karakter, egyetlen byte
- short - rövid egész szám
- long - hosszú egész szám
- double - dupla pontnosságú lebegőpontos szám

Ezenkívül lehetőség van user defined struct-ok kialakítására, ahol a fenti adattípusokból és egyéb struktókból rakhatunk össze új típusú struktókat, union-nal több struct típusból képezhetünk egy közös uniót stb.

10.2.1.2. Változók és aritmetikai kifejezések

A fordító minden, a `/*` és `*/` között előforduló karaktert és mid figyelmen kívül hagy, ezek kommentek. A C nyelvben használat előtt minden változót deklarálni kell. Mindezt (szabványtól függően) az első végrehajtható utasítás előtt, ezáltal a függvény test egy deklarációs és végrehajtási részre tagolható. A deklarációban egy típus megadását követően az ezen típusú változók neveinek felsorolását kell elvégeznünk. Példa:

```
int lower, upper, step; float fahr, celsius;
```

Értékadást a `=` operátorral érhetünk el, azaz pl.:

```
int lower, upper, step; float fahr, celsius;
lower = 0;
```

Persze az értékadás kiértékelődése miatt láncolhatjuk is a változókat:

```
int lower, upper, step; float fahr, celsius;
upper = lower = 0;
```

```
main ()
{
    int lower, upper, step;
    float fahr, celsius;
    lower = 0; /* A hőmérséklet-táblázat alsó határa */
    upper = 300; /* felső határ */
    step = 20; /* lépésköz */
    fahr = lower;
    while (fahr <= upper)
    {
        celsius = (5.0 / 9.0) * (fahr - 32.0);
        printf ("%4.0f %6.1f \n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Az előzőekben említett típusok fontosak, hiszen ezek szabályák meg a foglalt hely méretét. Ez gép (architektúra) függő.

A `printf` első argumentumában lévő format karakterláncban minden egyes `%` konstrukcióhoz hozzárendelődik a neki megfelelő második, harmadik stb. argumentum. Egyébként a `printf` nem része a C nyelvnek: a C nyelven belül a be- és kivetel nincs definiálva. Formatálásához segítség:

- `%d` decimális egész
- `%6d` decimális egész, de legalább 6 char hosszan
- `%f` lebegőpontos
- `%6f` lebegőpontos, de legalább 6 char hosszan
- `%.2f` decimális egész, kettő a tizedes vessző után
- `%6.2d` lebegőpontos, de legalább 6 char hosszan, kettő a tizedes vessző után

10.2.1.3. Változók és aritmetikai kifejezések

```
#include <stdio.h>
main () / * Fahrenheit-Celsius táblázat */
{
    int fahr;
    for (fahr = 0; fahr <= 300; fahr = fahr + 20){
        printf ("%4d %6.1f \n", fahr, (5.0 / 9.0) * (fahr - 32));
    }
}
```

A `for` egy ciklusutasítás, tekinthetünk rá a `while` általánosításaként. Három részt tartalmaz, amelyeket pontosvesszők választanak el. Az első rész, `fahr = 0` értékadása egyszer hajtódik végre a ciklusba való belépés előtt. A második rész a ciklust vezérlő ellenőrzés vagy feltétel. Minden alkalommal megvizsgáljuk hogy a feltétel teljesül-e, ha igen végrehajtjuk a ciklus magot, amit az újrainicializáló lépés azaz `fahr =fahr + 20` követ. A ciklus akkor ér véget, amikor a feltétel hamissá válik. Csakúgy, mint a `while` esetében, a törzs vagy egyetlen utasítás, vagy pedig kapcsos zárójelek közé zárt utasítások csoportja. Az inicializáló és újrainicializáló tetszőleges kifejezés lehet. A ciklusnak nincs saját scope-ja C++-tól eltérően.

10.2.1.4. Szimbolikus konstansok

Az előfeldolgozó preprocesszor segítségével úgymond szimbolikus konstansokat használhatunk. A preprocessor compile előtt végig megy a forrás szövegen és ha olyan szimbólumba fut, mely `define`-al értelmezve lett számára, akkor azon előfordulásokat cseréli a `define`-ban megadott karakterláncra. Ezek használatával az előző `fahrenheit` program.

```
#include <stdio.h>
#define LO 0
#define UP 300
#define STEP 20
main () / * Fahrenheit-Celsius táblázat */
{
    int fahr;
    for (fahr = LO; fahr <= UP; fahr = fahr + STEP){
        printf ("%4d %6.1f \n", fahr, (5.0 / 9.0) * (fahr - 32));
    }
}
```

10.2.1.5. Tömbök

C-ben a tömb egy adott adattípusból n darabot tartalmazó adattároló. Legfontosabb tulajdonsága, hogy memóriában az array egy egybefüggő helyen tárolódik. Emiatt allokalásakor nem csak hogy elegendő szabad memóriával kell a gépnek rendelkeznie, de kell lennie legalább egy olyan memória résznek, mely nem kisebb mint az igényelt terület az array által. Ezen helyfoglalás miatt az array mérete utólag nem növelhető. Az egyetlen módszer új elem hozzáadására, ha kérünk egy nagyobb array-t és átmásoljuk az elemeket a régitől az újba, majd a régit felszabadítjuk stb.

10.2.1.6. Argumentumok érték szerinti átadása

C-ben az argumentumok érték szerint adódnak át, azaz a hívó nem fogja látni a hívott az argumentumokon végzett változtatásokat. Példa:

```
#include <stdio.h>

int add(int x, int y)
{
    x = 5;
    return x + y;
}

main () / * Fahrenheit-Celsius táblázat */
{
    int a = 1;
    int b = 2;
    add(a,b);
}
```

Fenti példában az add által okozott változtatás `x = 5;` nem lesz kihatással `main`-beli `a`-ra.

Ez amiatt van így, mert `add` hívásakor `main`-beli `a` értéke átmásolódik `add` területén lévő `x` változóba. `x`-en végzett bármilyen változtatás lokális `add`-ra, és kívülről nem is megfigyelhető.

A későbbiekben kifogunk térni egy megoldásra abban az esetben, ha olyan változást akarunk amit a hívónak is látnia kellene. Ezt pointer-ekkel fogjuk megoldani (azaz a változtatandó változó értéke helyett a tárbeli címét küldjük a hívottnak.)

10.2.1.7. Karakter tömbök

A C nyelvben leggyakoribb tömbtípus valószínűleg a karaktertömb. Érdemes felhívni a figyelmet, hogy egy 0 értékű zárókarakterrel a c karaktertömbök általában le vannak zárva. Azaz például:

```
"hello\n"
['h','e','l','l','o','\n','\0']
```

Sok C funkció ha karakter tömböt fogad el, akkor elvárja hogy ezen szabványnak megfelelő módon előállított tömböt kapjon. Más esetben nem definiált működésük.

```
int getline(char s[], int lim)
{
    int c,i;
    for(i=0;i<lim-1&&(c=getchar())!=EOF&&c!='\n';++i){
        s[i]=c;
    }
    if(c=='\n'){
        s[i]=c;
        ++i;
    }
}
```

```
    }
    s[i]='\0';
    return i;
}

void copy(char to[], char from[])
{
    int i;
    i=0;
    while((to[i]=from[i])!='\0'){
        ++i;
    }
}
```

Fenti példában látható, hogy `copy` arra támaszkodik hogy null-terminated C char array-t kapjon. `getline` esetében pedig láthatjuk hogy a végén garantáljuk, hogy az utolsó karakter a terminátor legyen. Itt érdemes megjegyezni, hogy a fordító is hasonló technikával fordításközben átalakítja a forrásban található egymásután álló karaktereket egy nullterminált karaktertömbbé.

10.2.1.8. Érvényességi határ, külső változók

A main függvényen belüli változók (line, save stb.) a main-re nézve lokálisak. Mivel ezeket a main-en belül deklaráltuk, egyetlen más függvény sem tud közvetlenül hozzájuk férni. Ez fennáll fordítva is, azaz egy másik funkciónak lokális változó nem látható main számára (és ezáltal nem is okoz összeférhetlenséget névegyezés miatt). A függvények lokális változói csak meghívásukkor jönnek létre, és megsemmisülnek a scope elhagyáskor. Emiatt az ilyen változókat automatikus változóknak nevezzük. Ezek értéküket nem őrzik meg egyik hívástól a másikig, így minden belépéskor explicit módon értéket kell adni nekik. Ha azonban azt akarjuk hogy ne az előbb ismertetett "tranziens" módon viselkedjenek, akkor lehetőségünk van globális változókat használni. Ezeket a globális változókat bármelyik függvény név szerint elérheti. Az összes függvényen kívül kell definiálni, hogy ezzel tárolóhelyet foglaljunk le a számukra. A változókat minden olyan függvényben, ahol használni akarjuk, vagy explicit módon az extern alapszóval, vagy implicit módon értelemszerűen, de deklarálnunk is kell.

10.2.2. Típusok, Operátorok és Kifejezések

A programokban változókat és állandókat használunk. A deklarációk effektíve kijelentik a fordító számára az általunk felhasználni kívánt változókat. A fordítónak szüksége van emiatt a típusra, és esetleges kezdeti értékre is. A kifejezésekkel és operátorokkal ezen változókat módosíthatjuk plusz számításokat végezhetünk rajtuk. (Mármint a legvégén a legbonyolultabb dolog is egyszerű ALU által végrehajtható elemi műveletekké fog egyszerűsödni.)

10.2.2.1. Változó nevek

A nevek betűkből és számjegyekből állnak: az első karakter betű kell, hogy legyen. Az aláhúzás karakter (`_`) betűnek számít: ezzel javíthatjuk a hosszú változónevek olvashatóságát. Lehetőleg aláhúzással ne kezdjünk nevet (`_`) A nagy- és a kisbetű különbözőnek számít. A belső nevekben és külső nevekben eltérés van.

Ugyan akármilyen hosszú nevet megadhatunk mindkét esetben, de szabványtól függ, hogy ezen karakterekből mennyi lesz szignifikáns. Ezenkívül az olyan kulcsszavak, mint `if`, `else`, `int`, `float` stb. fenntartott szavak, változónévként nem használhatók.

10.2.2.2. Adattípus és méret

A C-beli alaptípusok

- `int` - egész szám
- `float` - lebegőpontos szám
- `char` - karakter, egyetlen byte
- `short` - rövid egész szám
- `long` - hosszú egész szám
- `double` - dupla pontnosságú lebegőpontos szám

`int`, `short` és `long` architektúrától eltérő, de a lényeges egymáshoz képest relatív reprezentált nagyságuk. (pl. `int` és `short` 16 bit, `long` 32 bit) `char` és `int` esetén tovább módosíthatjuk `unsigned` és `signed` hozzáadásával. Ezek annyit jelentenek, hogy előjeles vagy nem előjeles megjelenítést és aritmetikát terveznünk használni. Példa `char` esetén, hogy `unsigned char` `[0,255]` intervallumból, míg `signed` társa `[-128,128]` intervallumból vehet fel értékeket. `limits.h` és `float.h` tartalmazza ezen típusokhoz tartozó utility konstansokat(pl. `max`).

10.2.2.3. Konstansok

`int` konstans és `long`

```
1234
1234L
```

`long int` `signed` `unsigned`

```
1234L
1234UL
```

`float`

```
123.4
1e-2
```

Egy vezető 0 `int` előtt oktális, míg `0x` vagy `0X` hexadecimálist jelent

```
012
0x1f
```

Karakter konstanst aposztrófok közé írjuk. Newline és egyéb speciális karakterekhez escape char-t használhatunk ami backslash.

```
'a'  
'\n'
```

Stringeket idézőjelek közé írjuk

```
"aba"  
"ab" "a"
```

felsorolások (enum) a következő módon:

```
enum foo{AAR, BAR, CAR}
```

Enum érték 0-ról indul, de megadhatunk mást is

```
enum foo{AAR=1, BAR, CAR, UNDEF=0}
```

10.2.2.4. Deklarációk

Minden változót deklarálni kell használat előtt. A deklaráció meghatároz egy típust, amelyet az illető típusú változó(ka)t megadó lista követ, mint például: Külső vagy statikus változó esetén az inicializálás csak egyszer értelemszerűen a program végrehajtásának megkezdése előtt - történik meg. Az explicit módon inicializált automatikus változók minden alkalommal inicializálódnak, amikor az őket tartalmazó függvényt egy program meghívja. Az explicit inicializálás nélküli automatikus változók értéke határozatlan. A külső és statikus változók kezdeti értéke alapértelmezés szerint nulla, de stílusosan helyesebb, ha minden esetben megadjuk a kezdeti értéket.

```
int lower, upper, step; char c, line [1000];
```

A változók saját deklarációikban inicializálhatók is, bár ezzel kapcsolatban vannak megkötések. Ha a nevet egy egyenlőségjel és egy állandó követi, akkor az az illető változó kezdeti értékének megadását(inicializálását) jelenti.

10.2.2.5. Aritmetikai műveletek

Az aritmetikai operátorok a +, -, *, / és a % (moduló) operátor. Van egyoperandusú -, de nincs egyoperandusú +. Az egész típusú (integer) osztás levágja a tört részt. Az $x \% y$ kifejezés az x -nek y -nal történő osztásakor keletkező maradékot jelenti, tehát értéke nulla, ha x pontosan osztható y -nal.

10.2.2.6. Relációs és logikai operátorok

Relációs operátorok > >= < <=

Egyenlőséget vizsgáló operátorok == !=

Logikai operátorok && ||

10.2.2.7. Típus konverzió

Ha egy kifejezésben különböző típusú operandusok fordulnak elő, a kifejezés kiértékeléséhez az operandusokat azonos típusúakká kell alakítani. Belátható, hogy bizonyos esetekben a konverzió nem lehetséges, például a tört számot nem tudunk egész számként leképezni. Azonban ha például egy float-ként tárolt szám valójában éppen egész értékű, akkor nem okoz adatvesztést a konverzió. Bizonyos esetekben a konverzió garantálhatóan sikeres lesz. Ez például tegyük fel char-t akarunk int-é alakítani. Ez biztosan mindig megtörténhet, hiszen egy szűkebb intervallumon lévő elem biztos leképezhető egy olyan intervallum által aminek része ezen intervallum. Általában csak az értelmes konverziók történnek meg automatikusan, például egész típusú mennyiségek átalakítása lebegőpontossá olyan kifejezésekben, mint $f + i$, ahol f float, i pedig int típusú. Az értelmetlen kifejezések, mint például a float indexként való használata, nem megengedettek. A char és int típusú mennyiségek aritmetikai kifejezésekben szabadon keveredhetnek. Konverziót kényszeríthetünk is, ezt az alábbi módon tehetjük meg:

```
int a;  
char b;  
a = 5;  
b = (char)a;
```

10.2.2.8. Bitenkénti műveletek

A következő bitenkénti műveleteket használhatjuk.

- $\&$ - bitenkénti ÉS
- $|$ - bitenkénti megengedő (inkluzív) VAGY
- \wedge - bitenkénti kizáró (exkluzív) VAGY,
- \ll - bitléptetés (shift) balra,
- \gg - bitléptetés (shift) jobbra,
- \sim - egyes komplement (egyoperandusú).

10.2.2.9. Értékadási operátorok és kifejezések

Az olyan kifejezések, mint $i = i + 2$ amelyekben a bal oldal a jobb oldalon megismétlődik, a $+=$ értékadó operátor segítségével az $i += 2$ tömörített alakban is írhatók. A C-ben legtöbb aritmetikai művelet esetén van értékadó megfelelő. Ne feledekezzünk meg az erőssorrendről.

```
x *= y + 1 // 1.  
x = x * (y + 1) // 2. ekvivalens 1.-vel  
x = x * y + 1 // Nem ekvivalens sem 1.-vel sem 2.-kal
```

10.2.2.10. Feltételes kifejezések

```
if (a > b)
z = a;
else
z = b;
```

A fenti feltételes utasítás eredményeként z a és b közül a nagyobbik értékét veszi fel. A C-ben a háromoperandusú $?:$ operátor segítségével az ilyen szerkezeteket sokkal rövidebben leírhatjuk.

```
z = (a > b) ? a : b;
```

A feltételes kifejezésben az első kifejezést nem kötelező zárójelbe tenni, mivel $?:$ precedenciája igen alacsony (pontosan az értékadás fölötti). Zárójelezéssel azonban érthetőbbé tehetjük a kifejezés feltételrészét.

10.2.2.11. Precedencia, kiértékelési sorrend

Ha nem teljesen zárójelezett alakot adunk meg, akkor érdemes lehet tanuémányozni a [precedencia táblát](#).

10.2.3. Vezérlési szerkezetek

A nyelv vezérlésátadó utasításai a számítások végrehajtásának sorrendjét határozzák meg.

10.2.3.1. Utasítások, blokkok

A C-ben a pontosvesszővel zárjuk az utasításokat. A $\{$ és $\}$ kapcsos zárójelek felhasználásával deklarációkat és utasításokat egyetlen összetett utasításba vagy blokkba foghatunk össze. Ez szintaktikailag egyetlen utasítással egyenértékű. A blokkot lezáró jobb oldali kapcsos zárójel után soha nincs pontosvessző. A blokkon belül deklarálási sorrendtől fordított sorrendben szűnik meg a változók élete. (Ez később C++ esetén lehet hasznos amikor gura-okat akarunk csinálni, és garantálni akarjuk hogy egy bizonyos dtor mindig a végén hívódjon meg.)

10.2.3.2. If-else

Elágaztatás esetén az if-else-t használjuk

```
if(feltétel)
egy utasítás vagy egy blokk
else
egy utasítás vagy egy blokk
```

Azaz például

```
if(a>5)
    a=5
else
```



```
{  
a=b;  
c=2*d;  
b=c;  
}
```

Az else el is hagyható

```
if (g>5)  
    g=5
```

Emiatt azonban olykor nem teljesen egyértelmű hogy mely if hez tartozik egy else. Ennek feloldására szabvány szerint az else a hozzá legközelebbi else nélküli if-hez kapcsolódik.

```
if (n > 0)  
    if (a > b)  
        z = a;  
    else  
        z = b;
```

Ha ettől eltérőt akarunk, akkor azt a blokk jelöléssel érhetjük el

```
if (n > 0) {  
    if (a > b)  
        z = a;  
} else  
    z = b;
```

10.2.3.3. Else-if

Else if az else és if közé akárhányszor beépíthető

```
if (feltetel0)  
    utasitas0  
else if (feltetel1)  
    utasitas1  
else if (feltetel2)  
    utasitas2  
else  
    utasitas3
```

10.2.3.4. Switch

A switch utasítással a többirányban ágaztathatjuk a programot.

```
switch (kifejezes)  
{
```

```
case konstans-kifejezes: utasitasok
case konstans-kifejezes: utasitasok
default: utasitasok
}
```

A switch először kiértékeli a zárójelek közötti kifejezést utána pedig egyenként forráskódbeli sorrendben összehasonlítja a case-eknél megadott konstans kifejezések értékeivel. Minden case-t egész állandó kifejezéssel kell ellátni. Ha egy case azonos a kifejezés értékével, akkor a casehez rendelt tevékenységek elfognak végződni. A default címkéjű case-re akkor kerül a vezérlés, ha a többi case egyike sem teljesül. A default elhagyható : ha nem szerepel és a case-ek egyike sem teljesül, semmi nem történik. A case-ek és a default tetszőleges sorrendben követhetik egymást. A case utasítások címkéinek különbözniük kell egymástól. A break utasítás hatására a vezérlés azonnal kilép a switch-ből. Mivel a case-ek címkeként működnek, miután valamelyik case-hez tartozó programrész végrehajtása befejeződött, a vezérlés a következő case-re kerül, hacsak explicit módon nem intézkedünk a kilépésről. A switch-ből való kilépés legközönségesebb módja a break és a return. Az egymást követő case-ekbe való belépés nem egyértelműen előnyös. A case-ken történő lépkedés azért is veszélyes, mert a vezérlés széteshet, ha a programot módosítjuk. Azokat az eseteket kivéve, amikor ugyanahhoz a számításhoz több címke tartozik, a case-ek közötti átmenetek használatával célszerű takarékoskodni.

10.2.3.5. Loops(for, while)

a while esetben kiértékeljük a fejben lévő feltételt. Ha igaz, akkor belépünk a magba és végrehajtjuk az utasításokat, majd visszaugrunk fejbe és újakezdjük a folyamatot. Ha hamis, akkor elhagyjuk a ciklust:

```
while (kifejezes)
    statement
```

A for ciklus egy speciális while, ahogy azt az alábbi példa is mutatja:

```
for (expr0;expr1;expr2)
    statement
//ami ekvivalens
expr0
while (expr1){
    statement
    expr2;
}
```

10.2.3.6. Loops(Do-while)

A do-while esetben a mag egyszer garantáltan végrehajtódik. Alább látható alakú:

```
do
    utasitas
while (kifejezes)
```

Azaz utasitas mindig legalább egyszer végre fog hajtódni.

10.2.3.7. Break and continue

A ciklusból való kilépést vezérlhetjük nem csak a ciklus elején vagy végén való feltételvizsgálattal, hanem a magban kiadott utasítással is. A `break` utasítással a vizsgálat előtt is ki lehet ugrani a `for`, `while` és `do` ciklusokból, csakúgy, mint a `switch`-ből. A `break` utasítás hatására a vezérlés azonnal kilép a legbelső zárt ciklusból.

A `continue` utasítás a `break`-hez kapcsolódik, de a `break`-nél ritkábban használjuk. A `continue` esetén a ciklus (`for`, `while`, `do`) a következő iterációjának megkezdését idézi elő. A `while` és a `do` esetében ez azt jelenti, hogy azonnal végrehajtódik a feltételvizsgálat, a `for` esetében pedig a vezérlés azonnal az újrainicializálási lépésre kerül. `Switch`-re ez nem alkalmazható.

10.2.4. Függvények és programstruktúra

A függvények segítségével a nagy feladatokat kisebbekre tudjuk bontani, illetve kisebb problémákra adott megoldásokat újból felhasználni. A jól megírt függvények gyakran elrejtik az implementációs részleteket így felsőbb szinten a program áttekinthetőbbé válik.

10.2.4.1. Alapok

A függvényeket más helyekről a következő módon hívjuk:

```
fuggvenyneve(aktualis_parameterek_listaja)
```

Azaz a függvény neve után zárójelek között felsoroljuk azon paramétereket melyeket aktuálisnak tekintünk, és tovább akarunk adni. Amikor a program ide ér végrehajtásban érték szerint átadja az aktuális paramétereket, ezek a hívottban a formális paraméterek konkrét értékeként lesznek használva. Miután az irányítás átkerült a hívottnak addig amíg ki el nem éri utolsó sorát vagy egy `return`-t végrehajtódik. Ha végrehajtódásának végére ér, az irányítás visszaadódik a hívónak, és ha volt visszatérési érték, azt hívó látni fogja. Nem szükséges hogy a `return` után kifejezés álljon, ez esetben a hívó nem kap vissza semmit. A vezérlés visszatér (visszatérési érték nélkül) azon esetben is, ha a blokk végét úgy érjük el, hogy nem futottunk bele `return`-be. Mikor a visszatérési érték nem adódik meg a hívott által a hívó által látott visszatérési érték nem `NULL` vagy egyéb, hanem nem definiált, azaz ne alapozzuk erre a program későbbi működését, logikáját. Az ilyen hibákat `lint` helyességvizsgálóval analizálhatjuk.

10.2.4.2. Nem int visszatérési értékű függvények

A függvények implicit módon deklaráltak azáltal, hogy megjelennek valamely utasításban vagy kifejezésben. Ha valamely kifejezésben korábban még nem deklarált név fordul elő, amelyet bal oldali kerek zárójel követ, akkor ezt a gép a szöveggörnyezet alapján függvénynévként deklarálja. Default ezen függvény visszatérési értékének `int`-et veszi. Ha a fenti default működéstől eltérő értelmezést akarunk, akkor explicit deklarálnunk kell a függvényt. Ilyen esetben deklarálnunk a függvény visszatérési érték típusát a függvény neve előtt:

```
double foo(int a);
```

10.2.4.3. Külső változók

A C program külső változókból és függvényekből áll. Külsővel szemben belső az amit függvényen belül deklaráltunk vagy automatikusan deklarálódtak. A külső változókat függvényeken kívül definiáljuk, így sok függvény számára elérhetők. Maguk a függvények mindig külsők, hiszen függvényt függvényen belül nem szabad definiálni. Megállapodás szerint a külső változók egyben globális változók is tehát minden, az ilyen változóra ugyanazzal a névvel történő hivatkozás (még a teljesen külön fordított függvényekből is) ugyanarra a programozási objektumra történő hivatkozást jelent. Bármelyik függvény hozzáférhet külső változóhoz az illető változó nevére történő hivatkozással, ha a nevet korábban deklarálták.

10.2.4.4. Érvényességi Tartomány

Nem szükséges egyszerre lefordítani a C programot alkotó összes függvényt és külső változót: a program forrásszövege több állományban tárolható, és könyvtárakból már előzőleg lefordított rutinok is betölthetők.

Egy név érvényességi tartománya a programnak az a része, amelyre vonatkozóan a nevet definiáltuk. A függvény elején definiált automatikus változó érvényességi tartománya az a függvény, amelyben a nevet deklaráltuk, és a más függvényekben ugyanilyen néven létező változókat ez nem érinti. Ugyanez igaz a függvény argumentumaira. A külső változó érvényességi tartománya ott kezdődik, ahol a forrásállományban a változót deklaráltuk és az illető állomány végéig tart.

Ha viszont egy külső változóra még annak definiálása előtt kell hivatkozni, vagy ha egy külső változót más forrásállományban definiálunk, mint ahol használunk, akkor kötelezően extern deklarációt kell alkalmazni.

A forrásprogramot alkotó állományok között csupán egyben kell a külső változó definíciójának szerepelnie; a többi állományban extern deklarációval biztosítjuk a változó elérését. (A definíciót tartalmazó állományban is lehet extern deklaráció.) Külső változót csak definiáláskor lehet inicializálni. A tömbméreteket a definícióban kell megadni, de opcionálisan extern deklarációban is szerepelhetnek.

10.2.4.5. Statikus változók

A már korábban megismert extern és automatikus változók mellett a statikus (static) változók jelentik a harmadik tárolási osztályt. A static változók akár belsők, akár külsők lehetnek. Belső esetben ugyanúgy csak a függvényen belülről elérhetőek, de az automatikusoktól eltérően, nem inicializálódik újra értékük hívásonként és az egész program élettartama alatt megmaradnak. A belső static változók a függvényen belül állandó tárban tárolódnak le. A függvényeken belül megjelenő karakterláncok, mint pl. a printf argumentumai, belső static változók.

A külső static változó azon forrásállományban ahol deklaráltuk látható lesz, és a program egész élettartama alatt létezni fog. Miel csak az adott forrásállományban használhatjuk azután hogy deklarálva lett, ezért ennek két következménye lesz: Egy forrásállományban deklarációja után több függvénnyel is használhatjuk, módosíthatjuk ezt. Más forrásállományból nem látjuk, nem használhatjuk, módosíthatjuk.

A statikus tárolást úgy jelöljük hogy static szót írjuk a típus elé. A változó külső, ha nem függvényen belül deklaráltuk, belső ha függvényen belül.

```
static int a;  
foo(int b)  
{  
static int c;
```

```
}
```

10.2.4.6. Regiszter változók

Amennyiben lehetséges, a register típusú változók a gép regisztereibe kerülnek, miáltal rövidebb és gyorsabb programok jönnek létre. A register deklaráció alakja:

```
register int c, n;  
f(register unsigned int a, register long b)
```

Gyakorlatban persze elég gép függő kényszerek között dolgozunk. Hisz már ha csak a használható regiszterek számára gondolunk, akkor is láthatjuk hogy ez elég gép függő lehet.

10.2.4.7. Blokk struktúra

C-ben nem deklarálhatóak függvények függvények testében, viszont bizonyos részek elküönítésére alkalmasak a blokkok. Az adott blokk kezdete után deklarált változók a blokk végéig érvényben maradnak. Ha a blokkon kívül egy adott névvel már létezik változó de mi a blokkban deklarálunk ezen a néven újból egyet, akkor ezen blokkon belül, ez az új felül fogja bírálni a régit, tehát ezen blokkban mindig a blokkra lokálisra fogunk hivatkozni.

10.2.4.8. Inicializáció

Az explicit inicializálatlan külső és statikus változók értéke mindig garantáltan null lesz, míg az auto és regiszter változók értéke ilyen esetben definiálatlan.

Külső és statikus változók esetében az inicializálás egyszer fordítási időben történik meg. Az auto és regiszter változók minden függvénybe vagy blokkba lépéskor inicializálódnak, ha oda ért a vezérlés. Automatikus és regiszterváltozók esetében az inicializálás jobb oldalán állhat állandó, előzőleg definiált érték, függvény hívás stb.

Tömbök inicializálása során kapcsolósárjak között az elemeket vesszővel elválasztva kell felsorolnunk.

```
int arr [10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, } ;
```

Karakter tömbök esetén literállal is megadhatjuk.

```
char ar0 [] = "abc";  
char ar1 [] = {'a', 'b', 'c', '\0'};
```

10.2.4.9. Rekurzio

C megengedi hogy a függvények önnomagukat hívják. Ilyenkor az automatikus változól az új hívás esetén újra inicializálódnak. Mivel a hívó és a hívott adatait el kell szeparálni, ezért ez plusz terhelés a veremnek. Emiatt általában a rekurzivitás lassabb és nagyobb tárigányú kódhoz vezet.

10.2.5. Bevitel és kivitel

A bevitel és kivitel nem részei a C nyelvnek. Azonban a C nyelv mellé használatos szabványos könyvtár orvosolja ezt a problémát. Ez a könyvtár függvények olyan készletét tartalmazza, melyek a szabványos be- és kivitelről gondoskodnak. A könyvtárbeli rutinok gépfüggetlenek azaz kompatibilis módon működnek minde olyan rendszeren melyen a C létezik. A fenti könyvtárat `stdio.h` fájl includejával be kell emelnünk az előfordító segítségével használat előtt.

10.2.5.1. Szabványos kimenet és bemenet

A legegyszerűbb beviteli mód a `getchar`

```
int getchar(void);
```

Minden híváskor a soronkövetkező karaktert vagy EOF konstanst ad vissza. EOF jelentése end of file, azaz az input végére ért. Ezt az `stdio.h` definiálja.

Sok rendszeren lehetőségünk van szabványos bemenetre billentyűzet helyett fájl állományt irányítani.

```
foo < file.txt
```

Emellett lehetőségünk van egyik program kimenetét átirányítani egy másik program bementére:

```
foo | bar
```

A kimenetre legegyszerűbb lehetőségünk a `putchar`:

```
int putchar(int);
```

Ez egyszerűen a megadott értéket megpróbálja a szabványos kimenetre írni. Visszatérési értéke a kiírt érték, vagy hiba esetén EOF.

10.2.5.2. Formátumozott kimenet- printf

A formátumozott kimenetet megavlósító `printf`

```
int printf(char* format, arg1, arg2);
```

A függvény láthatóan vár egy `format` string-et, és ez illetve a megadott változó számú argumentumok alapján elvégzi a kiírást az `stdout`-ra.

A `format` string két fajta objektumot tartalmazhat: közönséges karaktereket, amelyeket egyszerűen a kimeneti folyamra másol és konverzió-specifikációkat, amelyek mindegyike a `printf` soron következő argumentumának konvertálását és ki-nyomtatását írja elő.

A `format` stringben `%` után szerepelnie kell a következő felsorolásbeli szimbólumoknak, majd egy konverziós karakterrel kell ezt a karakter felsorolást zárni.

- `minusz` - balra igazítás
- `szám` - mely megadja a minimális hosszat megjelenített karakterszámban

- pont - ami szeparálja a precíziót
- szám - megadja a precízió pontosságát
- h, vagy l - h ha short, l ha longként kell a számot kezelni

A konverziós karakterek:

- d,i - int decimális
- o - int unsigned oktális
- x,X - int unsigned hexadecimális
- u - int unsigned decimális
- c - egy karakter
- s - char* mely null terminátorig a char array-t kiírja
- f - double lebegőpontos szám [-]m.dddddd alakban
- e,E - double lebegőpontos szám [-]m.ddddde[+/-]xx alakban
- g,G - ha a kitevő kisebb mint -4, vagy nagyobb mint a precízió akkor úgy viselkedik mint e, egyébként f
- p - pointer
- % - nincs konverzió, %-t nyomtat

10.2.5.3. Formattált bemenet - Scanf

A formátumozott bemenetet megavlósító scanf

```
int scanf(char* format, arg1, arg2);
```

scanf akkor áll meg, ha a format string-ben megadott össze olvasási kérést elvégezte, vagy az egyik input format alapján történő beolvasása és értelmezése közben hiba lép fel. A scanf számára meg kell adni hogy beolvasás esetén hova tárolja a értékeket. Példa:

```
int a,b,c;  
double v;  
char arr[20];  
scanf("%lf",&v);  
scanf("%s",arr);  
scanf("%d/%d/%d",&a,&b,&c);
```

Fenti példán látszik hogy az érték szerinti átadás miatt nem egyenesen a változókat adtuk át hanem azok címét. Ettől csak a char array esetén térünk el, hiszen az array, és eredendően a a karaktertömbnek foglalt hely kezdő memória címét adja.

Scanf átugorja a whitespace karaktereket.

10.2.5.4. Változó hosszú argumentum listák

Az eddigiekben látott `printf` függvény esetében láthattuk, hogy nem volt megszabott az argumentumok száma. Hogyan lehet jelölni, hogy `n` darab előre nem meghatározott argumentumot akarunk elfogadni?

```
int myprintf(char*fmt, ...);
```

Ahhoz hogy végigtudjuk járni az argumentumokat használnunk kell az `stdarg.h`-t.

```
int myprintf(char*fmt, ...)
{
    va_list ap;char *p, *sval;
    int ival;
    double dval;

    va_start(ap,fmt);
    for(p=fmt; *p; p++){
        if(*p!='%'){
            putchar(*p);
            continue;
        }
        switch(++p){
            case 'd':
                ival = va_arg(ap,int);
                printf("%d",ival);
                break;
            case 'f':
                dval = va_arg(ap,double);
                printf("%f",dval);
                break;
            case 's':
                for(sval=va_arg(ap,char*);*sval;sval++){
                    putchar(*sval);
                }
                break;
            default:
                putchar(*p);
                break;
        }
        va_end(ap);
    }
}
```

10.2.5.5. Fájl hozzáférés

Az eddigiekben szabványos bemenettel, kimenettel foglalkoztunk. Azonban olykor szükség lehet például egyéb állományok használatára. Gyakran fájlok tartalmával kell foglalkoznunk. Ezek fizikailag egy adott

módon vannak leképezve, viszont a C szempontjából logikailag kezelésük egységes. Azaz a logikai állomány egy absztrakció a periféria, fájlrendszer és OS által megvalósított fizikai szint felett. Minden alapja a FILE ami az `stdio.h`-ban deklarált struktúra.

Az állományt bármilyen más művelet előtt meg kell nyitni. Ez az a pont amikor összerendelődik a fizikai és logikai állomány és emellett a hozzáférés módja is tisztázódik. Másrészt ez jelzi az OS számára hogy a folyamat igényt tart az adott állományra.

```
FILE* fp;  
fp = fopen(name, mode);
```

Az első arg az állomány neve (az OS kezeli egyébként, hiszen a valóságban nem név alapján azonosítja egyedileg a fájlokat.) A mode egy karakterlánc ami specifikálja hogy milyen módon akarjuk megnyitni. `r` olvasás, `w` írás, `a` hozzáfűzés. `b` bináris, azaz nem történik karakter konverzió, hanem a nyers bájtokat kapjuk meg. Ha egy nem létező fájlt nyitunk meg írási célból akkor az ilyenkor létrejön. Ha létező fájlt nyitunk írásra, akkor az összes előzetes eddigi tartalmat felülírjuk. Ha nem létező fájlt akarunk olvasni akkor az hiba. Másik hiba lehetőség, ha nincs megfelelő jogosultságunk, de ezt az OS kezeli vagy DOS esetén nem kezeli. Hiba esetén `fopen` garantáltan NULL értékkel tér vissza.

Bemenet kimenet legegyszerűbb tevékenységei a:

```
int getc(FILE* fp);  
int putc(int c, FILE* fp);
```

Format vezérelt bemenet kimenet legegyszerűbb tevékenységei a:

```
int fscanf(FILE* fp, char* fmt, ...);  
int fprintf(FILE* fp, char* fmt, ...);
```

Ha végeztünk a tevékenységekkel, akkor be kell zárni a fájlt. Ez valóságban azt jelenti, hogy bontani kell a megfeleltetési kapcsolatot a logikai és a fizikai állomány között. Másrészt ez jelzi az OS számára hogy a folyamat továbbá nem tart igényt az adott állományra.

```
int fclose(FILE* fp);
```

10.2.5.6. Hibakezelés és Exit

A `stdout` mellett létezik még egy szabványos kimenet. Ez az `stderr`. Ez általában akkor is megjelenik a képernyőn ha `stdout` át lett irányítva. (De külön ezt is lehet átírányítani.) Példa:

```
if(is_error){  
    fprintf(stderr, "%s: error\n", msg);  
    exit(2);  
}
```

A másik dolog amit a fenti kód használ az az `exit`. Ennek egyetlen argumentuma egy `int`. A program ennek hatására elkezd terminálni a futását. Visszatérési értéke pedig az ezen függvény argumentumában adott `int` lesz.

10.2.5.7. Soronkénti bemenet és kimenet

Lehetőségünk van stdlib-el sorok beolvasására.

```
char* fgets(char*line, int maxline, FILE* fp);
```

`fgets` beolvassa a következő karakterláncot `fp` által mutatott állományból. Maximum `maxline-1` karaktert olvas. Az olvasás eredményéből egy karakter tömböt hoz létre mely null terminált. Ha fájl végére ért, vagy egyéb hiba jelentkezett akkor `NULL` ptr-t ad vissza.

Lehetőségünk van stdlib-el sorok kiírására.

```
int fputs(char*line, FILE* fp);
```

`fputs` kiírja egy sorban a kimentre a megadott karakterláncot az `fp` által mutatott állományból. Alapvetően 0 visszatérési értéket ad vissza. Hiba esetén EOF-t.

10.3. Programozás

[BMECPP]

10.3.1. C és C++

C-ben üres paramlista azt jelenti hogy tetszőleges számú parammal hívható. C++-ban azonban ez konkrétan az alábbi jelenti:

```
void foo(void){}
```

Ha azonban előre nem definiált számú paraméteres hívást akarunk, akkor használjuk a köv alakot:

```
void foo(...){}
```

Másik eltérés a visszatérési típus explicit specifikálása:

```
foo(...){}
```

Ez C-ben int visszatérési típusú lesz (ugyanis implicitly assigned by compiler under the hood :)) C++ban ugyanez hiba compile time.

C++ban a főprogram két formája szabványos

```
int main(){}
```

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

C++-ban bevezetésre került kód olvashatóság miatt `bool` típus és ehhez kapcsolódóan `bool true false` kulcsszavak.

C++-ban több bájtos karakterek reprezentációjára beépített típus a `wchar_t`. String literal ezen esetben `L"bar"`.

C++-ban ahol utasítás állhat, ott állhat változó deklaráció is. Azaz C-hez képest nem különíthető el a body deklarációs és utasításos részre.

10.3.2. Function overloading

C-ben egy függvényt neve azonosítja egyértelműen. C++-ban azonban neve ÉS formális param listája együttesen. Az overload során létrehozott változatoknak meg KELL egyezniük visszatérési típusukban.

C linker aláhúzás+function name-et használ, azonban C++ esetében az overload megvalósítása miatt mást alkalmaznak.

C++ esetében a formális paraméterek alapján képzett prefixumot vagy posztfixumot adnak a függvény névhez. Ez a name mangling. Ennek pontos kivitelezése egyrészt nem sztenderdizált, hanem implementáció függő.

Ez persze elég problémássá teszi a C és C++ közti hívást. A non-mangling "bekapcsolható" ha `extern-el` specifikáljuk hogy mangling nélkül fordítsunk.

```
extern "C" int foo(int flags)
{
}
```

10.3.3. Alapértelmezett függvény argumentumok

Lehetséges default értékek definiálására formális paraméterek számára. Vegyünk egy példát.

```
extern "C" int bar(int a, int b, int c){}
```

Default-olás hátulról indul és folyamatos kell hogy legyen.

```
int bar(int a, int b = 10, int c = 10){} //Jó
int bar(int a, int b=10, int c){}       //Rossz
```

Hívásnál hátulról kezdve hagyhatóak el.

```
bar(0,0){} //Jó
bar(0,,0){} //Rossz
```

Default-olni egyszer definícióban és deklarációban nem lehet. (Érdemes decl-nél megadni, mert user úgyis a header-t fogja nézni.)

```
int bar(int a, int b = 10, int c = 10); //Idáig még oké
int bar(int a, int b = 10, int c = 10){} //Ez pedig hiba
```

10.3.4. Paraméterátadás referencia típussal

C-ben by value pass érvényes, ezért amikor azt akarjuk hogy a hívott függvény hatása látszódjon hívónál cím szerinti átadást alkalmazunk.

C++-ban egy újabb lehetőséget vezet be. Ehhez először bevezeti a referencia típust melyet egy & jellel jelezhetünk pl:

```
void foo(int& a)
{
    a = 5;
}
```

Fent vázolt esetben a hívó látni fogja az általa megadott aktuális paraméteren a változást. Referenciát **MINDIG** inicializálni KELL.

Apró kis probléma a ref visszaadás. Ennek során egy ideiglenes ref változót hoz létre a fordító.

```
int& bar()
{
    int x = 5;
    return x;
}

int calc(int a, int& b)
{
    return 2*a*b;
}

int main()
{
    std::cout<<calc(a,bar());
}
```

Itt annyi történik, hogy invalid területre hivatkozunk, hogy x bar végrehajtásakor élt. bar után nem garantált léte, és ha le is fut a dolog továbbra is a referencia a verem egy olyan helyére hivatkozik, ami nekünk nem állt szándékunkban.

10.3.5. Objektum orientáltság

A C++-ban lehetőség van összetartozó adatok és rajtuk végzendő műveletek egy egységbe való foglalására. Az elv encapsulation(egységbe zárás), és a megfelelő adatok és műveletek összefogását a class-ok(osztályok) hivatottak elérni. A class-oknak instance-ei (példányai) léteznek, és ezek konkrét adat értékei eltérőek, de típusaik és nevük megegyezik, illetve a műveletek egyeznek. Inheritance-ről (öröklés) is beszélhetünk, ahol mind a műveletek, mind az állapotváltozók örökölhettek. Más nyelvek ezt direkt tiltják, mert kora 2000-es évekbeli inheritance rémálommá válhatnak a projektek, ezért csak interface-ek implementációját engedik. Data hidingről (Adatrejtés) akkor beszélhetünk, ha például nem akarjuk engedni, hogy valamilyen implementációs detailhez osztályon kívülről hozzáférhessenek. A substitutability (behelyettesíthetőség) azon mélyebb elvet takarja, hogy ha egy általánosabb osztályon értelmezhető egy művelet például, akkor a specifikáltabb osztályon is értelmezhető az.

Az osztálynak tagváltozói, tag függvényei vannak. Maga az osztály egyébként egyben egy hatáskör is. Ez azért van így, mert különben name clash alakulhatna ki különböző osztályok tag változói és tag függvényeinek nevei között.

A tagfüggvényeknek vagy egy láthatatlan első paraméterük, amiben megkapják a struktúrára mutató ptr-t. Erre a `this`-el hivatkozhatunk.

Data hiding-ra private, public, protected használható. Private esetben csak az osztályon belül elérhető. Public esetben belül kívül. Protected esetben az osztályon belülről és minden leszármazottban.

Az objektum inicializálását egy konstruktor végzi. Ha a konstruktornak nincsenek paraméterei akkor azt default constructor-nak hívjuk. Ha egy paramétere van akkor azt gyakran konverziós konstruktornak nevezzük. Az objektum életciklusa végén a destruktorkat hívódik meg.

C++ osztályok a C struct-okhoz képest nem feltétlenül az attribútumok által meghatározott méretben és sorrendben foglalnak helyet a memóriában.

10.3.6. Dinamikus adattagot tartalmazó osztályok

C-ben a `malloc` és `free` volt használatos a dinamikus foglalásra. `Malloc` nem kezelte a típusokat, hanem egyenesen a lefoglalandó méretet várta

C++-ban a `new` és `delete` használatos amik már típus alapján képesek a méretet számítani. Továbbá `new` egyben a `ctor` hívását is elvégzi a megadott argumentumok alapján. Mivel az új `new`-ba nem adhatjuk be explicit a méretet, ezért a C-style array `malloc` nem fog működni. Emiatt `new[]` és `delete[]` került bevezetésre tömböknek történő helyfoglaláshoz és felszabadításhoz. `new[]` esetén nem adhatunk meg argumentumokat `ctor` híváshoz ezért a default `ctor` hívódik meg. Természetesen ha nem létezik default `ctor`, akkor `compile time error`-t fogunk kapni.

Amennyiben osztályunk dinamikus adattagot tartalmaz aminek életciklusáért mi vagyunk felelősek, akkor az osztály destruktoraiban el kell végeznünk a felszabadítást.

Másoló konstruktor esetén egy új objektumot hozunk létre egy előző alapján, az előző módosítása nélkül. Ez beépített típusoknál viszonylag egyszerű bitenkénti átmásolást jelent, viszont user defined esetekben erről nekünk kell gondoskodnunk.

```
T(const T& o){}
```

Ha dinamikus tagot tartalmaz osztályunk aminek életciklusáért mi vagyunk felelősek, akkor előáll egy probléma: Ha csak átmásoljuk a `ptr`-et A objektumból B `ctor`-ában, akkor ha A élete hamarabb véget ér (és lefut a `dtor` ezáltal felszabadítva a dinamikus tagot), akkor mellékhatásként B egy mostmár nem valid memóriaterületre fog mutatni és nem értesül a változásról.

Ilyenkor lehetséges megoldás a `ptr` által mutatott adat(ok) teljes rekonstrukciója egy újonnan igényelt helyen a memóriában. Ezt hívjuk `deep copy`-nak. Amikor ilyenről nincs szó, akkor `shallow copy`-ról beszélünk. A compiler által auto generált `copy ctor` `shallow copy`-t valósít meg.

10.3.7. Friend függvények és osztályok

C++-ban lehetséges van, hogy egy osztály globális függvényeket, vagy más osztályok tagfüggvényeit feljogosítsa, hogy tagváltozóikhoz és tagfüggvényeihez hozzáférjenek.

Friend függvényekre példa:

```
class Bar;
```

```
void howdareyou(Bar& b);
```

```
class Stranger{
void danger (Bar&);
};

class Bar{
int y;
int x;
friend void howdareyou (Bar&);
friend void Stranger::danger (Bar&);
};

void Stranger::danger (Bar& b) {
    b.x=b.y=0;
}

void howdareyou (Bar& b) {
    b.x=b.y=0;
}
```

Friend osztályra példa:

```
class Stranger;

class Bar{
int y;
int x;
friend class Stranger;
};

class Stranger{
void danger (Bar& b) {
b.x=b.y=0;
}
};
```

A class friend "jogosítványa" nem öröklődik subclass-okra. Illetve friend "jogosítvány" nem tranzitív.

10.3.8. Tagvátozók inicializálása

Először is tisztázzuk a az inicializálást és az értékadást!

Inicializálás

```
int i = 0;
Foo foo1 (1, 2);
int k;
Foo foo1;
```

Értékadás

```
int i;  
i = 6;
```

Menjünk kicsit mélyebbre, nézzük meg Foo-t!

```
class Foo  
{  
public:  
    Foo(int av, int bv)  
    { a=av; b=bv; }  
int a;  
int b;  
};
```

ctor testben `a=av` és `b=bv` már értékadásnak számítanak. Ha mégis inicializálni akarjuk a dolgokat, akkor a C++ a ctor initializer listet nyújtja segítségül.

```
class Foo  
{  
public:  
    Foo(int av, int bv) : a(av), b(bv) {}  
int a;  
int b;  
};
```

Ez akkor lehet hasznos, amikor valami miatt tartózkodni akarunk az assignment-től. Bár gyakoribb hogy olyan taggal kell dologoznunk akinek simán nincs default ctor-a. Ezen esetben technikailag nincs más lehetőségünk mint ctor initializer list-et használni.

10.3.9. Statikus tagok

Osztályok esetén lehetőségünk van olyan tagok létrehozására, melyek nem példányok, hanem az osztálynak részei.

Deklarálni `static` keywordddel tudjuk, viszont ez nem elég, ugyanis ez nem biztosít hely foglalást.

Azaz például a `cpp` fájlban külön fel kell tüntetni.

```
-- foo.hpp  
class Foo  
{  
public:  
    Foo() {}  
    static int a;  
};  
-- foo.cpp  
int Foo::a = 1;
```

Kívülről az osztály namespace-én keresztül kell elérni, azaz

```
fn_int_consumer(Foo::a);
```

Statikus tagfüggvények is definiálhatóak, viszont ezekből (logikus módon) nem érhetőek el a példányokra jellemző tagok. Sőt...logikus módon implicit egyedre mutató ptr-t sem kapnak (azaz `this` ne értelmezett, hisz nem példányra jellemző tagfüggvény).

```
-- foo.hpp
class Foo
{
public:
    Foo(){}
    static void naughty()
    {
        a = 0; HIBA!!!
        this->a = 0; HIBA!!!
    }
    int a;
};
-- foo.cpp
int Foo::a = 1;
```

A statikus tagváltozók a `main` előtt inicializálódnak! Azaz nem garantálható egy C++ programban, hogy a `main` első sora lesz a kezdő sor. (plusz még ott vannak a globális változók initjei)

10.3.10. Beágyazott (nested) definíciók

C++ esetében lehetőség van `enum`, `struct`, `class` és `typedef` osztály definíción belüli megadására. Ezek kívülről a teljesen mnősített nevükkel érhetőek el pl.

```
class Foo
{
class Bar{};
};
-- Foo::Bar
```

Nestelt esetben sem a nestet megvalósító(tartalmazó) osztály sem a nestelt (tartalmazott) osztály nem kap a másik felé külön jogokat. A kívüllág számára a `private` után deklarált nestelt `class`-ok nem láthatóak. STL-ben gyakran találkozhatunk ilyen nested `class`-okkal container-ek esetén.

10.3.11. Konstansok és inline függvények

C-ben konstansokra gyakran használják a preprocesszor adta lehetőségeket `#define`-al. Ez azonban valójában csak nyers behelyettesítést jelent. Szószertint a preprocesszor csak felcseréli a `define`-al definiált szimbólumsor összes előfordulását a megadott nyers értékkel. Ezáltal `type` információt sem képes rögzíteni "magáról".

C++ esetében a `const` type modifier-t használhatjuk. Ezzel jelezzük a compiler számára, hogy az érték init után garantáltan nem fog változni, és ha ilyet íránk, kezelje hibaként.

Const ptr-ek esetén két különböző dolgot is megadhatunk

A mutatott érték ne legyen változtatható

```
char t[10];
const char* p = t;
*p = 0;    HIBA!
p++;      OK!
```

A mutató érték ne legyen változtatható

```
char t[10];
char* const p = t;
*p = 0;    OK!
p++;      HIBA!
```

Const függvény paraméterek esetén is használható. Ekkor azt a jelentést teszi fel, hogy a paraméter olvasható, de nem változtatható. Vegyük például referenciákat:

```
void foo(Foo& o)
{
    o.x=5;    OK!
}
```

```
void foo(const Foo& o)
{
    o.x=5;    HIBA!
}
```

Függvények visszatérési értéke is lehet konstans. Például `std::string c_str` tagfüggvénye. A `const` overaload szempontjából megkülönböztető jelentőségű.

Osztályok tagváltozói is lehetnek konstansok. Ezeket a `ctor` inicializer list-ben KELL inicializálni.

Const függvények fejlécének végén is használható annak kifejezésére, hogy a függvénynek garantáltan nincs olyan mellékhatása mely módosítani a példány állapotát. Ha azonban egy `const` függvényben mégis módosítani akarunk egy tagváltozót, akkor az nem lehet `static` vagy `const` és a `mutable`-t kell használni.

```
class Foo
{
public:
    const int a;
    mutable int b;
    void safe() const{b = 7;}
};
```

10.3.12. C++ IO alapjai

Ezen alfejezetben a C++ IO-val fogunk foglalkozni, kicsit a C-ből indulva.

10.3.12.1. Szabványos adatfolyamok

C nyelvben három előre megnyitott szabványos állomány leíró áll rendelkezésre. `stdin`, `stdout` és `stderr`. A C++ adatfolyamokban gondolkodik, ami felfogható byte-ok sorának. Egy folyam lehet bemeneti (istream), kimeneti (ostream), vagy ezek kombinációja. A könyv azt írja, hogy a kimenetnek alapértelmezetten a képernyő a kimenete. Ez szerintem nem teljesen van így (mármint ha jól értem akkor arra gondolhatott a szerző, hogy ha terminál ablakból nyitottuk, akkor annak az OS által kialakított OS dependens terminálnak valamilyen módon ír egy bufferére, ami a végén azt eredményezi hogy megjelenik a szöveg a terminál ablakban. De nem direkt a képernyőn jelenik meg még ekkor sem...).

`std` namespace `cout`, `cin` és `cerr` megfeleltethető a C-s szabvány állományoknak céljuk és rendeltetésük szerint. A left shift `<<` alkalmazható `cout-on` (ostream) a kiírásra, míg a `>>` right shift operátorral `cin`-ből végezhetünk beolvasást. `cin` esetén fontos tudni, hogy whitespace jellegű karaktereket elhagyja. Alább egy példa a folyam állapotáról:

```
#include <iostream>
int main()
{
    std::cout<<"Enter an int: "<<std::endl;
    int i;
    std::cin >> i;
    std::cout<<"Enter a dbl: "<<std::endl;
    double d;
    std::cin >> d;
    if(cin)
        std::cout<<"The number was "<<i<<" , "<<d<<std::endl;
    else
        std::cout<<"It was not a number"<<std::endl;
}
```

A fent megadott kód a következő user input-ra `12.3` várakozásunk ellenére `12`, `0.3`-at fog kiírni anélkül hogy bekérné a `double`-t, vagy hibát dobna `12.3`-nál. C-ben `fflush(stdin);` -el értük el az eddigi adatok ejtését (azaz a flush-t). C++-ban más módon ugyan de azt megtudjuk adni, hogy a sorvége karakterig hagyja figyelmen kívül a karaktereket.

```
#include <limits>
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
```

A `numeric_limits` első paramétere megadja hány karaktert ignoráljunk legfeljebb, míg a másik argumentummal azt adjuk meg melyik karakterig.

`flush`-ra egyébként van lehetőség a bufferelt adatfolyamokon:

```
std::cout<<std::flush;

std::cout.flush();
```

A `cerr` nem bufferelt, ami segítség, ha pl. memória szűkében állunk és pont ezt akarjuk kiírni, hiszen ha bufferelni kéne, nem lenne elég memóriánk, hisz pontosan amiatt kezdtük el a hiba írást.

A `clog`, hasonló rendeltetésű mint a `cerr`, azonban ez bufferelt.

Az adatfolyam állapotát egy `iostate` nevű tagváltozó reprezentálja. Ez a következő értékeket veheti fel:

- `eofbit`: adatfolyam elérte végét
- `failbit`: pl. formátummal hiba
- `badbit`: fatális hiba, pl. adatvesztés
- `goodbit`: egyik másik állapot sem áll fenn

Ezek együtt maszkokként használandóak, alább példa:

```
std::cout.clear(std::ios::eofbit | std::ios::failbit);
```

A bit-ek lekérdezése tagfüggvényekkel történik, pl.:

```
bool g = std::cout.good();
```

Ha az adatfolyam bármely hibabitje beállítódik, akkor az össze további írási és olvasási tevékenység hatástalan marad.

Az adatfolyam igaz-hamis értékke történő kiértékelése hamisat ad, ha bármely hiba bit be van állítva. Ennek logikusan tagadás is működik

```
while(std::cin>>v)
{ siker(v); }
```

```
if(!cin)
{ std::cout<<"Err"<<std::endl; }
```

Az `istream` osztálynak egyéb hasznos tagfüggvényei is vannak.

`get()` utolsó karakter vagy eof visszaadása.

`getline()` egy sor olvasása a sorvégig vagy más hatroló karkterig

`fread()` bináris adat olvasása.

10.3.12.2. Manipulátorok és formázás

Az előre definiált manipulátorokat az `omanip` header-ben találjuk

Például a `noskipws` segítségével beállíthatjuk hogy a sztenderd input ne ignorálja a whitespace karaktereket. `skipws`-el pedig visszatudjuk ezt állítani.

```
std::cin>>std::noskipws;
std::cin>>std::skipws;
```

A `setprecision`-el a megjelenített tizedestört kiírási pontosságát állíthatjuk be például `setprecision(4)`. A formázásra különböző jelző bitek szolgálnak. Alább példa lebegőpontos szám tizedestört alakban történő kiírására:

```
cout<<setiosflags(ios::fixed)<<3.14<<endl;
cout<<resetiosflags(ios::fixed);
```

10.3.12.3. Állománykezelés

C-ben az állomány kezelés `FILE*` típusú leíróval kapcsolatos függvényekkel történik. C++-ban ez is adatfolyamokkal történik. Ezek `ifstream` `ofstream` az ezekhez kapcsolódó header `fstream` fájlban találhatóak. Az állományok megnyitását ezen objektumok konstruktorai, míg lezárást ezek destruktoraik végzik.

Egyes OS-k eltérően ábrázolhatják és kezelhetik egyes speciális karakterek csoportját. Legegyszerűbb a WIN-re jellemző carriage return + new line. Emiatt olykor érdemes lehet binárisban nyitni a fájlt(begyűk észre a flag-ek össze vagy-olását):

```
ofstream ofs("sample.txt", ios::out | ios::binary)
```

Ha valami miatt nem `ctor`-ban akarjuk nyitni a fájlt, vagy nem `dtor`-ban zárni, akkor `open()`, `close()` és `is_open()` állnak rendelkezésünkre.

Bizonyos állományok esetén pozícionálásra is van szükség.

`tellg()` visszaadja a jelenlegi pozíciót.

`seekg(position)` a megadott pozíciót állítja be.

`seekg(offset, position)` Relatív a pozícióhoz képest állítja be az `offset` által megadott pozíciót. `position` lehet: `ios::beg` állomány kezdet, `ios::end` állomány vég, `ios::cur` jelen pozíció.

Az ezekkel beállított pozíciók validitását a hívónak (szóval a programozónak) a felelőssége garantálni.

Átírányításra alább egy példa

```
// Állományt nyitjuk
ofstream log_file("log.txt");
// clog formázási beállításainak másolás
log_file.copyfmt(clog);
// clog eredeti bufferének eltárolása
streambuf* clog_buf=clog.rdbuf();
// clog buffere az állomány buffere lesz
clog.rdbuf(log_file.rdbuf());
// Ezen a ponton clog már át van irányítva
clog<<"Redirect happend "<<endl;
// Régi buffer vissza
clog.rdbuf(clog_buf);
```

10.3.13. Operátorok és túlterhelésük

Az operátorok argumentumaikon végzett műveletekkel visszatérési értéket állítanak elő. Általában emiatt használjuk őket, de egyes operátorok ezenkívül argumentumaikon végzett változtatásuk miatt (mellékhatás) is hasznosak.

Operátorok kiértékelési sorrendjét a nyelv által lefeketetett minden implementáció által betartandó precedencia táblázat rögzíti.

10.3.13.1. Függvényszintaxis és túlterhelés

C nyelv esetében a függvényeknek nem lehet mellékhatásuk a pass by value miatt. Az egyetlen mód a kikényszerítésére ha ptr-t adunk át. C++ esetében azonban referenciák használatával ez már nem így van.

Példának vegyünk egy postfix operátort

```
int postfix_incr(int& arg)
{
    int a = arg;
    arg = arg + 1;
    return a;
}

int main()
{
    int i = 1;
    int j;
    j = i++;
    j = postfix_incr(i);
}
```

A fenti példa is mutatja, hogy az operátorok és a függvények között csak nagyon kis különbség van. A különbség csak a kiértékelés szabályaiban van. Sőt, szószerint függvény hívási szintaxissal is hívhatjuk!

```
++i;
operator++ (i);
```

Mintahogy a függvények, úgy az operátorok is túlterhelhetők. Alapvetően a global namespace-ben kell dolgozni:

```
Foo operator(double d, Foo z);
```

Ha azonban az első paraméter olyan user defined type (class) amit mi írtunk, akkor azon adott osztály tagfüggvényeként fogalmazzuk meg!

```
Foo Foo::operator(Foo a, Foo b);
```

10.3.13.2. Speciális operátorok túlterhelése

Az életciklus szempontjából nem triviális az assignment, értékadás operátor túlterhelése. Dinamikus adat-tagokat tartalmazó osztályban ezt érdemes végig gondolni. A ctor-okat is behozva a témába a következő életciklust befolyásoló esetek vannak:

- default ctor
- other ctors
- copy ctor

- copy assignment
- move ctor
- move assignment

Ezeket érdemes mindig elkészíteni. Ha egy ctor-t definiáltunk, akkor a hozzátartozó assignment-et is definiálni kell szabály szerint. Ha mondjuk valamit nagyon nem akarunk (pl.: non-copyable) akkor érdemes azokat direkt törölni, megakadályozva a compiler generálta shallow copy implementációt.

Default ctor-ról már volt szó. Hiánya esetén az inicializálást kezelünk kell majd ha beépítjük más osztályba.

Más ctor-ok, ezeknél talán az egy paraméterűek speciálisak, ugyanis ezek konverzióknál használhatóak. (És a compiler is megpróbálja őket használni, ha csak nem tiltjuk meg)

Copy ctor és assignment esetén egy létező példány alapján egy teljesen újat akarunk kialakítani. Mindezt úgy, hogy a másolandó állapotát nem változtathatjuk (azaz max mutable változókhoz nyúlhatunk hozzá)

III. rész

Második felvonás

DRAFT

**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.

DRAFT

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

DRAFT

11.3. Általános

[JUHASZ] Juhász, István, *Magas szintű programozási nyelvek*, mobiDiák , 2008.

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

[SICP] Abelson, Harold, *Structure and Interpretation of Computer Programs*, MIT , 1996.

11.7. My Little Ponys

[CHISOMORPH] Sorensen, Morten Heine B, *Lectures on the Curry-Howard Isomorphism*, Pdf , 1998.

[DENOTATIONALSEMANTICS] Allison, Lloyd, *A practical introduction to denotational semantics*, Cambridge University Press , 1986.

[NANDTOTETRIS] Nisan, Noam, *The Elements of Computing Systems*, Homepage of the book , 2005.

[PROGLANGS] Harper, Robert, *Practical Foundations for Programming Language*, Carnegie Mellon University , 2016.

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.