Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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

COLLABORATORS			
	TITLE : Univerzális programozás		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Petrus, József Tamás	May 7, 2019	

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

Ajánlás

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

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

Contents

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 3

2 Helló, Turing! 5

2.1 Végtelen ciklus . . . 5
2.2 Lefagyott, nem fagyott, akkor most mi van? . . . 6
2.3 Változók értékének felcserélése . . . 8
2.4 Labdapattogás . . . 9
2.5 Szóhossz és a Linus Torvalds féle BogoMIPS . . . 9
2.6 Helló, Google! . . . 10
2.7 100 éves a Brun tétel . . . 10
2.8 A Monty Hall probléma . . . 10

3 Helló, Chomsky! 12

3.1 Decimálisból unárisba átváltó Turing gép . . . 12
3.2 Az $a^nb^nc^n$ nyelv nem környezetfüggetlen . . . 13
3.3 Hivatkozási nyelv . . . 13
3.4 Saját lexikális elemző . . . 14
3.5 l33t.l . . . 15
3.6 A források olvasása . . . 15
3.7 Logikus . . . 18
3.8 Deklaráció . . . 19

4 Helló, Caesar! 23
4.1 double ** háromszögmátrix . . . 23
4.2 C EXOR titkosító . . . 23
4.3 Java EXOR titkosító . . . 23
4.4 C EXOR törő . . . 24
4.5 Neurális OR, AND és EXOR kapu . . . 24
4.6 Hiba-visszaterjesztéses perceptron . . . 26

5 Helló, Mandelbrot! 27
5.1 A Mandelbrot halmaz . . . 27
5.2 A Mandelbrot halmaz a `std::complex` osztállyal . . . 27
5.3 Biomorfok . . . 28
5.4 A Mandelbrot halmaz CUDA megvalósítása . . . 28
5.5 Mandelbrot nagyító és utazó C++ nyelven . . . 28
5.6 Mandelbrot nagyító és utazó Java nyelven . . . 29

6 Helló, Welch! 30
6.1 Első osztályom . . . 30
6.2 LZW . . . 30
6.3 Fabejárás . . . 31
6.4 Tag a gyökér . . . 31
6.5 Mutató a gyökér . . . 32
6.6 Mozgató szemantika . . . 32

7 Helló, Conway! 33
7.1 Hangyaszimulációk . . . 33
7.2 Java életjáték . . . 33
7.3 Qt C++ életjáték . . . 34
7.4 BrainB Benchmark . . . 35

8 Helló, Schwarzenegger! 36
8.1 Szoftmax Py MNIST . . . 36
8.2 Mély MNIST . . . 36
8.3 Minecraft-MALMÖ . . . 36

9 Helló, Chaitin! 38
9.1 Iteratív és rekurzív faktoriális Lisp-ben . . . 38
9.2 Gimp Scheme Script-fu: króm effekt . . . 38
9.3 Gimp Scheme Script-fu: név mandala . . . 38

10 Helló, Gutenberg! 40
10.1 Programozási alapfogalmak . . . 40
10.2 Magas szintű programozási nyelvek 2 by Juhász István (Pici könyv 2) . . . 43
10.3 Programozás bevezetés . . . 43
10.4 Programozás . . . 44

III Második felvonás 50

11 Helló, Arroway! 52
11.1 A BPP algoritmus Java megvalósítása . . . 52
11.2 Java osztályok a Pi-ben . . . 52

IV Irodalomjegyzék 53
11.3 Általános . . . 54
11.4 C . . . 54
11.5 C++ . . . 54
11.6 Lisp . . . 54

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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

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

Part I

Bevezetés

Chapter 1

Vízió

1.1 Mi a programozás?

1.2 Milyen doksikat olvassak el?

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

1.3 Milyen filmeket nézzek meg?

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

Part II

Tematikus feladatok

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

Chapter 2

Helló, Turing!

2.1 Végtelen ciklus

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

Megoldás videó:

Megoldás forrása: Ez a program egy magon végtelen ciklust futtat, viszont a top parancs alapbeállításaival 0.0%-os processzorhasználatot mutat.

```
#include <stdio.h>

int main()
{
    for(;;)
    {
        sleep(1);
        printf("Ez egy végtelen ciklus!\n");
    }
    return 0;
}
```

Ez a program egy magot terhel 100%-osan.

```
int main()
{
    while(1);
    return 0;
}
```

Ez a program pedig négy magot (a számítógépem négy magos processzorral rendelkezik) terhel 100%-osan.

```
#include <unistd.h>

int main()
{
    int t1,t2,t3;
    if(!(t1=fork()))
    {
        for(;;);
    }
    if(!(t2=fork()))
    {
        for(;;);
    }
    if(!(t3=fork()))
    {
        for(;;);
    }
    for(;;);
}
```

Tanulságok, tapasztalatok, magyarázat... Az egy magon 0%-os processzorhasználathoz az volt az elgondolás, hogy ha a ciklusmagban "elaltatjuk" a programot, akkor nem lesz kimutatható processzorhasználat. Az egy mag 100%-os leterhelését egy egyszerű "üres" végtelen ciklussal próbáltam először, ami sikeresnek bizonyult. A processzor összes magjának terhelését viszont már egyértelműen nem lehet egy egyetlen szálon futó programmal terhelni, ezért forkoltam három gyerek szálat, mind a háromban "üres" végtelen ciklussal, ami sikeresen terhelte a processzorom összes magját.

2.2 Lefagyott, nem fagyott, akkor most mi van?

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

Megoldás videó:

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

```
Program T100
{

  boolean Lefagy(Program P)
  {
```

```
     if(P-ben van végtelen ciklus)
      return true;
     else
      return false;
  }

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

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

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

akár önmagára

```
T100(T100)
false
```

ezt a kimenetet adja.

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

```
Program T1000
{

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

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

  main(Input Q)
```

```
  {
    Lefagy2(Q)
  }

}
```

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

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

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

Ez a feladat a megállási probléma bemutatására szolgál. A megállási probléma abból áll, hogy el lehet-e dönteni egy porgramról adott bemenet esetén, hogy végtelen ciklusba kerül-e. Alan Turing 1936-ban bizonyította be, hogy nem lehetséges olyan általános algoritmust írni, amely minden program-bemenet párról megmondja, hogy végtelen ciklusba kerül-e.

2.3 Változók értékének felcserélése

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

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

Megoldás forrása:

```
#include <stdio.h>

int main()
{
    int a=5,b=4;

    printf("a=%d, b=%d\n",a,b);
    int tmp = a;
    a=b;
    b=tmp;
    printf("a=%d,b=%d\n",a,b);
    return 0;
}
```

A megoldásomban azt a módszert alkalmaztam, amit először ismertem erre a feladatra. Ezt egykori informatika tanárom "bögrés cserének" hívta. Ezt úgy kell elképzelni, mintha a két változó egy bögre tej és egy bögre tea lenne, és úgy kell kicserélni a tartalmukat, hogy azok ne keveredjenek. A megoldás, hogy venni kell egy harmadik (segéd) bögrét, esetünkben változót, és abba beletölteni az első bögre tartalmát. Az első bögrébe beletölteni a második bögre tartalmát, és a második bögrébe beletölteni a harmadik (segéd) bögre tartalmát.

2.4 Labdapattogás

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

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

Megoldás forrása:

https://github.com/Ignissen/pjt_bevprog/blob/master/pattogas_c.c

https://github.com/Ignissen/pjt_prog1/blob/master/pattogas_if.c

Ezt a feladatot először C++ nyelven készítettem el a Bevezetés a programozásba nevű kurzuson. Amikor atírtam a programot C-re, egyből szembetűntek a legalapvetőbb különbségek a nyelvek között. Például a standard kimenetre való írás módjának különbözősége. A program tartalmaz egy n*m-es karaktermátrixot, aminek a "szélső" elemei '#' karakterrel jelöltek, hogy láthatóak legyenek a "pálya" szélei. A labda '@' karakterrel van jelölve a képernyőn. Tartalmaz egy i int típusú számlálót, amely a program működéséhez szükséges. A program main függvényében található egy végtelen while ciklus, amelynek legelején a labda koordinátáinak kiszámítátása áll. Ezután a program lemásolja a program legelején létrehozott tömböt egy másik tömbbe. A másolás után meghívja a draw függvényt, amely egy kétdimenziós karaktertömböt, 2 int-et (labda x és y koordinátája), illetve egy karaktert, amely a labdát jelölő karakter. A karaktertömb másolására azért van szükség, mert a megoldásomban a `draw` függvényben a tömb azon elemét fölülírom a labda karakterével, amely koordinátája megegyezik a labdáéval. Ennek eredményeként, ha az eredeti tömböt adnám át a függvénynek, idővel az egész "pálya" betelne labdával. A draw függvény hívása után a számlálót növelem eggyel, illetve várakoztatom a program végrehajtását 50 ms-al, hogy a labda mozgása szemmel követhető legyen.

2.5 Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/szohossz.c

A legtöbb személyi számítógép esetében az int mérete 32 bit. Ez azt jelenti, hogy egy int típusú változó -2,147,483,648 és +2,147,483,647 közötti számokat képes tárolni. Az általam tesztelt számítógépen 32 bites az int mérete. Ez valószínűleg a legtöbb személyi számítógépre igaz.

2.6 Helló, Google!

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/pagerank.c

A PageRank egy olyan algoritmus, amellyel weboldalak relatív fontosságát lehet megállapítani. Azt adja meg, hogy véletlenszerű böngészés esetén mekkora az esélye annak, hogy az adott oldalra találunk. Alapja, hogy egy oldalon minden hivatkozás egy-egy "szavazat" a hivatkozott oldalra. Az alapján meg lehet állapítani egy oldal relatív fontosságát, hogy hány az oldalra mutató hivatkozás van a többi oldalon, illetve, hogy hány oldalra hivatkozik az adott oldal. Az algoritmusban egy jobb minőségű oldal "szavazata" erősebbnek számít, mint egy kis relatív fontosságúé. Mivel a felhasználó általában nem fogja végignézni az összes linket a weboldalon, ezért bevezettek a képletbe egy csillapító faktort. A megoldásomban Lovász Botond segített.

2.7 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

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

A Brun-tétel szerint az ikerprímek reciprokösszege egy meghatározható szám felé tart, amelyet a B_2 konstanssal jelölnek, amely értéke $B_2 \approx 1{,}902160583104$. Az ikerprím olyan két egymást követő prímszámot jelent, amelyek különbsége 2.

Az stp változó egy függvénypointer lesz, amely függvény a megadott intervallum prímszámait hivatott megkeresni. az x tömb értékeket tartalmaz 13-tól 1000000-ig. Az y tömb az stp általam mutatott függvény értékei lesznek az x tömböt kapva. Végül a `plot` függvény ábrázolja grafikonon az x és y tömböket koordináta-rendszerben.

2.8 A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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

A Monty Hall probléma egy az Amerikai Egyesült Államokban sugárzott televíziós vetélkedő játékszabályai alapján jött létre. A szabály az, hogy a versenyző kiválaszt 3 ajtó közül egyet, amelyek mögött vagy kecske van (2 ajtó esetében) vagy egy autó van az ajtó mögött (1 ajtó mögött van csak autó). A játékvezető, aki tudja, melyik ajtó mögött mi található, kinyit egy ajtót, amelyet a versenyző nem választott, majd megkérdezi, hogy szándékszik-e ajtót váltani. Ezután, ha a versenyző nem váltott ajtót, akkor

a játékvezető kinyitja a második nem választott ajtót. Ha a versenyző ajtót váltott, akkor az eredetileg választott ajtót nyitja ki a játékvezető.

Az a probléma azért paradoxon, mert a válasz arra, hogy megéri-e váltani az ajtók között a játékvezető kérdése után, az, hogy igen, érdemes változtatni az ajtón, viszont ez a józan észnek annyira ellentmond, hogy ezt a problémát paradoxonnak minősítik.

A probléma megoldása azon alapszik, hogy, maikor választunk a három ajtó közül, akkor 1/3 az esélyünk arra, hogy a választott ajtó mögött található az autó. A játékvezető először mindenképp kecskét rejtő ajtót nyit ki a játékosnak. Az eslő ajtó választásánál 2/3 az esélyünk arra, hogy kecskét választunk, ez alapján a játékvezető kénytelen a másik kecskét rejtő ajtót kinyitni. Ez alapján látható, hogy ha váltunk az első ajtó kinyitása előtt, javul az esélye annak, hogy a játékos autót nyer.

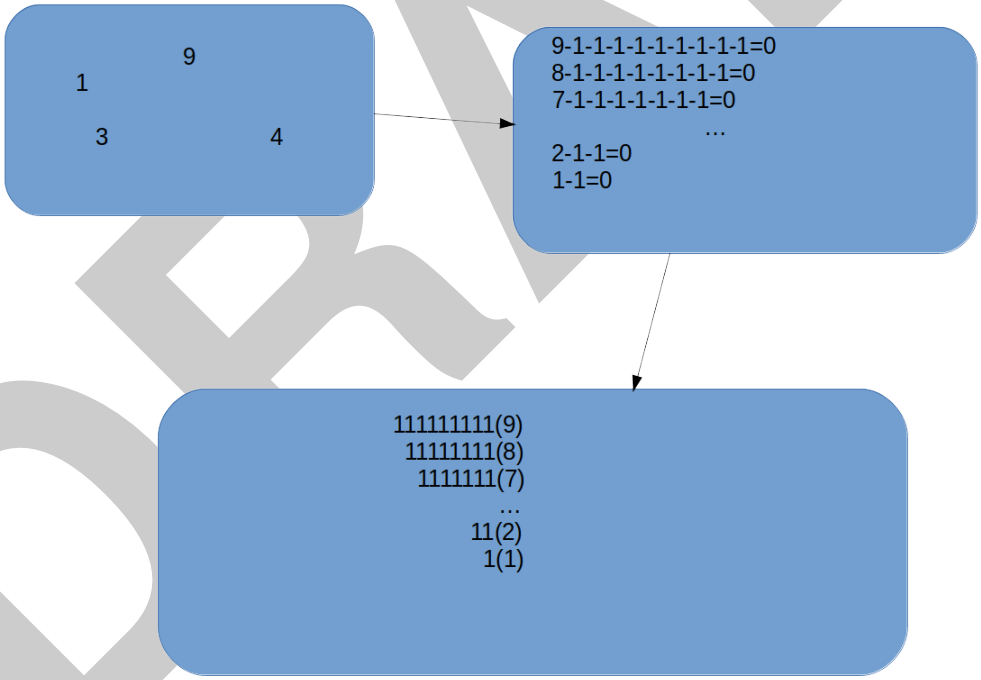
Chapter 3

Helló, Chomsky!

3.1 Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás forrása:



Az unáris számrendszerben való ábrázolás n darab (n a decimális szám) egyforma jel, karakter egymás utáni leírásával történik.

A decimálisból unárisba átváltás úgy történik, hogy folyamatosan 1-eket vonunk ki a számból, és tároljuk a levont egyeseket.

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:

Első környezetfüggő generatív grammatika:

```
Szabályok:
S -> aBSc
S -> abc
Ba -> aB
Bb -> bb

Példa levezetés:
S -> aBSc -> aBaBScc -> aBaBabccc -> aaBBabccc -> aaBaBbccc -> aaaBBbccc -> ↩
    aaaBbbccc -> aaabbbccc
```

Második környezetfüggő generatív grammatika:

```
Szabályok:
S -> abc
S -> aXbc
Xb -> bX
Xc -> Ybcc
bY -> Yb
aY -> aaX
aY -> aa

Példa levezetés:
S -> aXbc -> abXc -> abYbcc -> aYbbcc -> aaXbbcc -> aabXbcc -> aabbXcc ->  ↩
   aabbYbccc -> aabYbbccc -> aaYbbbccc -> aaabbbccc
```

A generatív nyelvtan elméletét Noam Chomsky alkotta meg, és ő dolgozta ki a Chomsky-hierarchiát. A formális grammatikáknak három típusa van, a környezetfüggetlen nyelvtan, a szabályos nyelvtan és a generatív nyelvtan. A környezetfüggetlen nyelvtanban a szabályok megadása esetén a sazbály bal oldalán csak nem terminális változó állhat, illetve a jobb oldalán csak terminális változók állhatnak.

3.3 Hivatkozási nyelv

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/utasitasok.txt

```
#include <stdio.h>

int main()
{
    for(int i=0;i<5;i++)
    {
        printf("%d\n",i);
    }
    return 0;
}
```

A Backus-Naur-forma a különböző nyelvek egyik lehetséges leírási módszere. Ezt a leírási módszert John Backus hozta létre, eredetileg az ALGOL programozási nyelvhez. Azóta már a legtöbb programozási nyelv szintaxisát BNF-ben adják meg, illetve természetes leírásához is használják alkalmanként. Peter Naur egyszerűsítette le a leírási módszert, ezért Donald Knuth javaslatára Naur neve is belekerült a leírási módszer megnevezésébe.

Feladat volt még olyan C programot írni, amely egyes nyelvi szabvánnyal, jelen esetben a C89-es szabvánnyal, nem fordul le, míg például a C99-es szabvánnyal már igen. A C89-es szabvány például még nem engedte a for ciklusban a ciklusfejben történő ciklusváltozó deklarálását. Ezt szemlélteti a fenti kód is, ugyanis a -std=c89 kapcsolót használva hibát jelez a fordító.

3.4 Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása:

```
%{
#include <stdio.h>
int szamok=0;
%}
%option noyywrap
%%
[[:digit:]]+  {
            szamok++;
        }
```

```
[[:alpha:]][[:print:]]*     {    /* Nem csinálunk semmit. */    }
%%
int main(void)
{
    yylex();
    printf("%d számot talált a lexer: \n",szamok);
    return 0;
}
```

Az lex fájl első részében, amit a %{...%} jelöl, jelzem a fordítónak, hogy az stdio.h függvénykönyvtárat szeretném haszálni az stdoutra való íráshoz, majd létrehozok egy számlálót, ami a lexer által észlelt számok darabszámát fogja tárolni. A második részben, amit a %%...%% jelöl, megadtam a lexernek, hogy bizonyos mintákra hogyan reagáljon. A [[:digit:]]+ azt jelenti, hogy legalább egy számjegy egymás után. Ha legalább egy számjegyet talál egymás után, abban az esetben eggyel növeli a számlálót. Ezután megadtam, hogy bármilyen más minta eseté ben, effektíve ne csináljon semmit. Majd az utolsó részben, amely már nincs külön jelölve, a `main` függvényben meghívjuk a `yylex` függvényt, amely meghívja magát a lexert, amely végigfutja bájtonként a bemenetet. Ha a lexer futása véget írt, kiíratom a számláló értékét. Majd a return 0-val jelzi az operációs rendszer felé, hogy a program futása véget ért.

3.5 l33t.l

Lexelj össze egy l33t ciphert!

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/leet.c

A lex első részében a program által látható függvénykönyvtárak include-jai láthatók. Ezután egy int típusú változó létrehozása áll, amely a random számok generálásának beállításához szükséges. A program működésének alapja a cipher típusú tömb létrehozása, ami a különböző betűkhöz és számokhoz tartozó lehetséges leet kódokat tartalmazza, többnyire három kódolt betűt és 4 "eredeti" betűt, hogy kisebb eséllyel legyen minden betű átalakítva.

A kód következő részében minden a lexer által beolvasott karakterre megnézi a program, hogy benne van-e a cipher típusú tömb kódolandó karakterei között. Ha megtalálja, akkor ahhoz a karakterhez tartozó egyik kódolást véletlenszerűen kiválasztja, majd a standard kimenetre kiírja. Ha nem találta meg, akkor az eredeti karaktert kiírja a standard kimenetre.

A `main` függvényben a program meghívja a `yylex` függvényt, azaz magát a lexert. Ha a lexer futása véget ér, akkor a program 0-val tér vissza, amely azt jelzi az operációs rendszernek, hogy a program futása sikeresen véget ért.

3.6 A források olvasása

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

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

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

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

i.

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

ii.

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

iii.

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

iv.

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

v.

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

vi.

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

vii.

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

viii.

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

Megoldás forrása:

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

void jelkez()
{
    printf("\nNem nyert.\n");
}

int main()
{
    while(1)
    {
        if(signal(SIGINT, jelkez)==SIG_IGN)
            signal(SIGINT, SIG_IGN);
    }
    return 0;
}
```

A feladatban leírt kódcsipetet természetes nyelven valahogy így lehet elképzelni: Ha a program elkap egy megszakítás jelet, akkor meghívja a `jelkezelo` függvényt/eljárást, amely valamit csinál a megszakításra, ezután pedig visszajelzi az operációs rendszernek, hogy a program kezelte a jelet.

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

Ez a csipet nem fog a célnak megfelelően működni, mert a manual signal lapja alapján a signal függvény a második argumentumával fog visszatérni, amelynek az if-ben saját magával kéne nem egyenlőnek lennie, ezért a jelkezelő függvény soha nem lesz meghívva.

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

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

Ez a két csipet szintaktikailag különbözik abba, hogy az elsőnél először növelnénk az i értéket, majd használnánk az értékét, de eppen a példában nem használjuk az értékét egyből, tehát nem számít. A két csipet szemantikailag azonos, az előbb említett ok miatt, minden esetben hibátlanul fog lefutni.

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

Ez a csipet egy léptetős ciklus, amely ötször fog lefutni, és minden iterációban a tomb nevű tömb i-edig elemét egyenlővé teszi i-vel. Ez a csipet nem minden esetben fog hibátlanul lefutni, mert ez a csipet feltételezi, hogy a tomb nevű tömb legalább 5 elemű, illetve a ciklus előtt deklarálva és inicializálva van

egy i nevű ciklusváltozó. Egyéb esetben a program olyan memóriaterületre fog hivatkozni, ami számára nem megengedett, ez pedig az (modern) operációs rendszer általi azonnali leállítást eredményez.

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

Ez a csipet minden estben le fog fordulni, azonban ha az n változó nagyobb, mint a d vagy s tömb hossza, abban az esetben az előző csipetnél leírt probléma fog előfordulni.

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

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

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

Ezek a csipetek uyganazt csinálják, csak kicsit másképp. A csipetek kiírnak standard inputra két egészet. Ezen csipeteknek közös lehetséges bugforrása, hogy az argumentumok sorrendjétől nem független a kiértékelés.

3.7 Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

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

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

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

Az első formula természetes nyelven: Minden x-re létezik olyan y, hogy x kisebb, mint y, és y prím. Tehát: Minden számnál létezik nagyobb prímszám.

A második formula természetes nyelven: Minden x-re létezik olyan y, hogy x kisebb, mint y, y prím és y rákövetkezőjének rákövetkezője is prím. Tehát: Minden számnál léteznek nagyobb ikerprímek.

A harmadik formula természetes nyelven: Létezik olyan y, hogy minden x-re igaz, hogy ha x prím, akkor x kisebb, mint y. Tehát: Minden prímszámra igaz, hogy létezik tőle nagyobb szám.

A negyedik formula természetes nyelven: Létezik olyan y, hogy minden x-re igaz, hogy ha y kisebb, mint x, akkor x nem prím. Tehát: Létezik olyan szám, amelytől nem létezik kisebb prímszám.

3.8 Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató
- egész referenciája
- egészek tömbje
- egészek tömbjének referenciája (nem az első elemé)
- egészre mutató mutatók tömbje
- egészre mutató mutatót visszaadó függvény
- egészre mutató mutatót visszaadó függvényre mutató mutató
- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

Mit vezetnek be a programba a következő nevek?

-
```
int a;  //egész típusú változó létrehozása
```
-
```
int *b = &a;  //egészre mutató mutató
```
-
```
int &r = a;  //egész referenciája
```
-
```
int c[5];  //egészek tömbje
```
-
```
int (&tr)[5] = c;  //egészek tömbjének referenciája
```
-
```
int *d[5];  //egészre mutató mutatók tömbje
```

- ```
  int *h ();  //egészre mutató mutatót visszaadó függvény
  ```

- ```
  int *(*l) ();  //egészre mutató mutatót visszaadó függvényre mutató  ↩
     mutató
  ```

- ```
  int (*v (int c)) (int a, int b)  //egészet visszaadó és két egészet kapó  ↩
     függvényre mutató mutatót visszaadó, egészet kapó függvény
  ```

- ```
  int (*(*z) (int)) (int, int);  //függvénymutató egy egészet visszaadó és  ↩
     két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó  ↩
     függvényre
  ```

Megoldás videó:

Megoldás forrása:

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

int main()
{
    //Első csipet
    int a=5;
    //Második csipet
    int *b = &a;
    *b=2;
    printf("a=%d\n",a);
    //Harmadik csipet
    int &r=a;
    r=4;
    printf("a=%d\n\n",a);
    return 0;
}
```

Ez a program gcc-vel nem fordul, mert a c-ben nincs referencia típus. Az első csipetben létrehoztam egy int típusú változót. A második csipetben létrehoztam egy egészre mutató mutatót, amely az első egészre mutat. A harmadik csipetben létrehoztam egy egész referenciáját, amely értéke az a-val egyenlő. A referencia típus azt valahogy úgy kell elképzelni, mint linuxon a hardlinkeket.

```c
#include <stdio.h>

int main()
    //Negyedik csipet
    int c[5]={1,2,3,4,5};
    //Ötödik csipet
    int (&tr)[5] = c;
    for(int i=0;i<5;i++)
    {
        printf("%d\n",tr[i]);
    }
    //Hatodik csipet
    int *d[5];
    for(int i=0;i<5;i++)
    {
        printf("%p\n",d[i]);
    }
    return 0;
}
```

Ez a program gcc-vel nem fordul, mert a c-ben nincs referencia típus. A negyedik csipetben létrehoztam egy egész típusú tömböt, amely elemei rendre 1, 2, 3, 4, 5. Az ötödik csipetben létrehoztam egy egészek tömbjének referenciáját. A hatodik csipetben létrehoztam egy egészekre mutató mutatók tömbjét, amely 5 elemű.

```c
#include <stdio.h>
#include <stdlib.h>
//Hetedik csipet
int *h()
{
    return (int*) malloc(sizeof(int));
}

int main()
{
    //Nyolcadik csipet
    int *(*l)() = h;

    printf("%p\n",l());
    return 0;
}
```

A hetedik csipetben létrehoztam egy egészre mutató mutatót visszaadó függvényt. A nyolcadik csipetben létrehoztam egy, az előző csipetben létrehozott függvényre mutató mutatót. Ezután egy `printf` függvénnyel standard outpura kiírattam a függvény pointer segítségével a `h` függvény által lefoglalt egész memóriacímét.

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

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

int (*asd(int c))()
{
    if(c) return sum;
    else return mul;
}
///Kilencedik csipet
int (*v(int c))(int a,int b)
{
    return asd(c);
}



int main()
{
    //Tizedik csipet
    int *(*z(int))(int,int) = v;
    printf("%d\n",z(0)(5,5));
    return 0;
}
```

A kilencedi csipetben létrehoztam egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényt. A tizedik csipetben létrehoztam egy, az előző feladatben létrehozott függvényre mutató mutatót.

Chapter 4

Helló, Caesar!

4.1 double ** háromszögmátrix

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/haromszog_matrix.c

A programban helyet foglalok a memóriában egy double-öket tartalmazó alsó háromszögmátrixnak. Majd k=0-tól a mátrix minden elemének 1.1-es lépésközzel értékül adtam k-t. Ezután a mátrix standard uotputra való kiíratása történik. Következőnek felszabadítom a pointerek által lefoglalt memóriacímeket, majd a "return 0"-val jelzem az operációs rendszer felé, hogy a program futása hiba nélkül befejeződött.

4.2 C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/exor.c

Ez a program a legelején megnézi, hogy a kapott argumentumok száma több, mint kettő. Erre azért van szükség, mert a kódolandó szövegfájl nevét és a titkosításhoz használt kulcsot parancssori argumentumként kapja meg a program. Ha a feltétel teljesül, tovább fut a program, ha nem, akkor kiírja standard outputra a program helyes használatának módját. Ezután az igaz ágon belül megnyitjuk olvasásra a kódolandó szöveget tartalmazó fájlt. Ha a fájl megnyitása nem sikerült, hibaüzenettel kilép a program. Ha sikerült, akkor karakterenként beolvassuk, a kulcs megfelelő karakterével "össze-exorozzuk", majd kiíratjuk standard outputra. Ha végzett a beolvasással, akkor a tiszta szöveget tartalmazó fájlt bezárjuk, majd kilép és jelzi az operációs rendszer felé, hogy a program futása hiba nélkül véget ér.

4.3 Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/exor.java

Ez a program az előző feladat megoldásának java átirata, amely annyival különbözik az előzőtől, hogy itt a tiszta szöveget tartalmazó fájl neve tiszta.txt, amely egy mappában van a programmal, iletve a kulcsot a standard inputról kéri be, nem parancssori argumentumokként.

4.4 C EXOR törő

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/t.c

Ez a program több függvényből is áll. A `atlagos_szohossz` függvény a program által éppen generált tesztkulcscsal visszafejtett szöveg átlagos szóhosszát adja vissza.

A `tiszta_lehet` függvény azt adja meg, hogy a tesztkulcscsal visszafejtett szöveg lehet-e a tényleges visszafejtett szöveg. Ennek az a feltétele, hogy az átlagos szóhossz az hat és kilenc között legyen, illetve tartalmazza a következő négy szót: "hogy", "az", "nem", "ha".

Az `exor` függvény hajtja végre a titkos szöveg a program által generált kulcsokkal történő visszafejtését, amely vagy a tényleges visszafejtés, vagy nem.

Az `exor_tores` függvény meghívja az `exor` függvényt, majd a `tiszta_lehet` függvényt, amely, ha igaz értékkel tér vissza, akkor kiírja az adott kulcsot, illetve a kulcscsal visszafejtett szöveget.

Végül a `main` függvényben beolvassuk a titkosított szöveget, majd nyolc egymásbaágyazott for ciklussal végig vizsgáljuk az összes kulcsot. Ezt a lépést Bátfai Norbert Párhuzamos programozás GNU/Linux környezetben c. könyvének linkelt fejezete alapján OpenMP használatával párhuzamossá tettem.

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

Ebben a feladatban a cél egy olyan neurális háló létrehozása és tanítása, amely az egyszerű logikai műveletek elvégzésére képes.

Ez a kód igazából négy kis eltéréssel ismétlődő részből áll. Minden részben lényegében ugyanaz történik, egyedül a logikai művelet változik, amelyre feltanítjuk a neurális hálót. Ezalól kivétel az utolsó, amiről lentebb szó lesz.

Első rész:

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

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

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

plot(nn.or)

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

Az elején megadjuk, hogy milyen bementi adatokból milyen eredményt kell megközelítenie a threshold-dal jelölt hibahatáron belül. Ezután ezt megadjuk a neurális hálónak is, majd a neurális hálót feltanítjuk a feladatra. Itt meghívjuk a `neuralnet` függvényt, amely megkapja a bementi adatokat és az elvárt kimeneteket, 0 rejtett réteggel, 0.000001-es hibahatárral. Ezután a `plot` függvénnyel kirajzoljuk a neurális háló sematikus képét egy gráf segítségével.

Majd a `compute` függvénnyel meghívjuk a már feltanított neurális hálót az elején megadott adatokkal, hogy kiszámolja a logikai műveletek eredményét.

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

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

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

plot(nn.exor)

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

Itt annyi különbség van, hogy míg a harmadik részben feltanított neurális háló kb. 50%-os pontossággal dolgozott, ami annyit jelent, mintha véletlenszeűen találgatott volna, itt már több neuron van a hálóban, növelve a pontosságot. Ebben az esetben három rejtett neuronréteg van, amelyek rendre 6, 4, 6 neuronból állnak. Ezeket a "hidden=c(6,4,6)" argumentum jelöli. Ezzel már a hibahatáron belülre kerül többnyire az EXOR logikai művelet értékének kiszámítása.

4.6 Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/perc.cpp

A program a kód elején vizsgálja, hogy a parancssori argumentumok száma kettő-e. Erre azért van szükség, mert az bemenetként szolgáló png fájl nevét parancssori argumentumként kapja meg a program. Ezután a png++ függvénykönyvtár segítségével beolvassa a bementi fájlt. Ezután létrehoz egy perceptront 4 rétegű neurális hálóval, amelynek neuron száma sorra 3, a kép pixeleinek száma, 256, 1. Majd létrehoz egy dinamikusan foglalt a kép pixeleinek megfelelő számosságú double tömböt, amelybe belemásolja a kép minden pixelének vörös értékét. Ezután meghívja a perceptront, hogy dolgozza fel a képet. A progvam végén felszabadítja a pointerek által foglalt memóriát, majd kilép.

Chapter 5

Helló, Mandelbrot!

5.1 A Mandelbrot halmaz

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/mandelbrot.cpp

A program elején létrehozunk néhány változót a precompiler számára, amelyek a kimeneti kép méretét határozzák meg, illetve vizsgált tartományt a komplex számsíkon.

A `GeneratePNG` függvény paraméterként megkapja a program által generált kép adatait pixelenként a tomb nevű int típusú N*M-es mátrixban tárolva. A függvényen belül létrehozunk egy N*M pixeles PNG kiterjesztésű képfájlt, amelybe az egymásba ágyazott for ciklus pixelenként beletölti az adott pixelre vonatkozó adatokat, majd a for ciklus után kiírja a képfájlt lemezre "kimenet.png" néven.

A `main` függvényen belül létrehozzuk az N*M-es mátrixot, amelyben tároljuk a kép pixelenkénti adatait, beállítjuk a komplex számsíkon való lépegetés lépésközét a "dx" és "dy" változóban, majd létrehozunk három Komplex típusú változót, amely a komplex számokat fogja tárolni a Mandelbrot-halmaz kiszámításához. Ezután belép a porgram a for ciklusba, ahol lépked a komplex számokkal a megadott tartományban a megadott lépésközzel, majd a benne lévő while ciklus meghatározza a kép megfelelő pixelének színét annak függvényében, hogy a while ciklus fejében megadott formula hány iteráció alatt lesz nagyobb vagy egyenlő, mint négy. Miután a for ciklus bejárta a komplex számsík megadott tartományát, meghívja a `GeneratePNG` függvényt.

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/mandelbrot_komplex.cpp

Az előző programhoz képest itt annyi az eltérés, hogy a

```
struct Komplex{
    double re,im;
```

```
};
```

helyett a c++ beépített complex osztályát használva lépegetünk a komplex számsíkon.

5.3 Biomorfok

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

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

A biomorfok olyan alakzatok, amelyek ránézésre akár élő organizmusok is lehetnének, viszont nem muszáj természetes eredetűnek lennie az alakzatnak (magyarán, akár lehetnek számítógép által generáltak is).

Ez a program nagyon hasonlít az előzőhöz, ugyanis ennek az alapja a Mandelbrot-halmaz. A legjelentősebb eltérés az előző programhoz képest, hogy itt más a megadott formula a pixelek színeinek számításánál.

5.4 A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/mandelpngc_60x60_100.cu

Ez a feladat az 5.1-es feladatra épül, uyganis CUDA használata esetén device kódból nem hívhatunk olyan függvényeket, osztályokat, amelyek csak a hoston érhetők el.

A `mandel` függvény előtt álló "__device__" jelzi a fordítónak, hogy az a függvény csak a videókártyáról lesz elérhető, a hostról nem. Ez a függvény tartalmazza a pixelenkénti Mandelbrot-halmaz számolását.

A `mandelkernel` előtt álló __global__ jelzi a fordítónak, hogy a függvény egy kernelfüggvény, amelyet meg lehet hívni a host kódból, illetve meg tud hívni device függvényeket. Ebben a függvényben az aktuális szál meghatározása áll, amely azért kell, hogy tudjuk, éppen melyik komplex számot kell vizsgálnia a `mandel` függvénynek, majd meghívja a `mandel` függvényt.

A `cudamandel` függvény végzi a kernel meghívásához szükséges műveleteket. Például a memóriafoglalást a videókártyán, illetve a kernel hívását, majd a videókártyán tárolt adatokat a RAM-ba másolását.

A `main` függvényben a számunkra lényeges sorok a kepadat nevű mátrix lefoglalása, majd a `cudamandel` meghívása, illetve a PNG képfájl létrehozása, majd lemezre írása. A többi sor a `main`-ben a futási idő mérésére szolgál.

5.5 Mandelbrot nagyító és utazó C++ nyelven

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

Megoldás forrása:

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/mandel_zoom.cpp

A feladat megoldásához SFML-t használtam. A feladatban a mandelbrot halmaz meghatározására az 5.2-es feladat megoldását vettem alapul.

A compute függvény határozza meg minden pixel színét, ugyanazon módszerrel, mint az 5.2-es feladatban.

A következő néhány függvény segédfüggvény a nagyításhoz illetve a halmaz feltérképezéséhez.

A `main` függvényben létrehozom a grafikus megjelenítéshez szükséges objektumokat, változókat. A while cikluson belül először eseménykezelés található, amely azért felelős, hogy mi történjen, ha a felhasználó a képernyőre kattint, illetve ha a egér görgőjével görget. Majd meghívom a `compute` függvényt, ezután pixelenként kirajzoltatom a kiszámolt mandelbrot-halmazt.

5.6 Mandelbrot nagyító és utazó Java nyelven

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/Mandelbrot.java

Az osztály konstruktorában létrehozzuk a GUI-t, illetve megadjuk a paramétereit (méret, méretezhetőség), létrehozzuk a kontroll objektumokat, a gombokat, amelyekkel változtatható a vizsgált komplex szám tartomány. A `plotPoints` eljárás felelős a vizsgált tartomány bejárásáért és az alapján a halmaz elemeinek kiszámításáért. A `actionPerformed` eljárás felelős azért, hogy a felhasználói interakciót lehetővé tegye az által, hogy "megmondja", mi történjen, ha a felhasználó rákattint egy gombra.

A megoldásomat Lovász Botond segítette.

Chapter 6

Helló, Welch!

6.1 Első osztályom

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

Megoldás videó:

Megoldás forrása:

C++: https://github.com/Ignissen/pjt_prog1/blob/master/polargen.cpp

Java: https://github.com/Ignissen/pjt_prog1/blob/master/PolarGen.java

A megoldásban létrehoztunk egy Polargen nevű osztályt, amely konstruktorában megadjuk, hogy még nincs eltárolt szám, illetve a véletlenszám-generátornak random seedet adunk.

A `kovetkezo` függvény megnézi, hogy van-e tárolt szám, ha nincs, akkor generál két számot, amelyből az egyiket eltárolja, majd a logikai változót hamisra állítja, majd a másik számmal visszatér.

A feladat célja az, hogy lássuk, hogy az objektum orientált programozás nem nehéz, de még természetes is. A Java SDK-ban is hasonlóan megírt programrészleteket találhatunk. Természetesnek azért mondható, mert mi emberek a világon mindent objektumokként képzelünk el. Például minden tárgy egy objektum, aminek vannak különböző adatai, például szín, méret, hely, illetve lehetnek "eljárásai/függvényei", például egy telefonnak, ha telefonálunk, stb.

6.2 LZW

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/binfa.c

A progra melején létrehoztam egy Node nevű struktúrát, amely aZ LZW bináris fában lévő csomópontokat reprezentálja. Ennek van egy char típusú változója, illetve két Node-ra mutató mutató. Az egyik a nullás gyerekre mutat, a másik az egyes gyerekre.

A `create_empty` függvény inicializálja a bináris fát egy kitüntetett gyökérelemmel, amely a '/' karakterrel van megjelenítve a bejárásoknál.

A `create_node` függvény létrehoz egy csomópontot az argumentumként kapott karakterrel, a gyermekekre mutató mutatókat NULL-ra állítja.

A `insert_tree` függvény valósítja meg az LZW bináris fa építését. A függvény először megnézi, hogy a kapott érték '0'-e. Ha igen, akkor megnézi, hogy a fa mutató által címzett csomópontak van-e bal oldali gyermeke (0-s gyermek). Ha van, akkor a fa mutató az aktuális csomópont bal gyermekére lép. Ha nincs, akkor létrehozza az aktuális csomópont bal oldali gyermekét, majd a fa mutatót a gyökérre állítja. Ha a kapott érték nem '0', akkor a függvény végrehajtja a fent leírt utasításokat, csak az aktuális csomópont jobb gyermekére.

Az `inorder` eljárás inorder módon rekurzívan bejárja a bináris fát. Az inorder bejárásnál először a bináris részfa bal oldalát járjuk be, majd feldolgozzuk a részfa gyökérelemét, aztán feldolgozzuk a részfa jobb oldalát.

A `destroy_tree` eljárás rekurzívan postorder módon bejárja a fát és minden rekurzió végén felszabadítja a részfa gyökérelemét. A felszabadítás előtt meg kell vizsgálni, hogy a részfa gyökere egyenlő-e a teljes fa gyökerével, mert ebben a megoldásban a teljes fa gyökéreleme nem dinamikusan foglalt.

A `main` függvényben feltöltöm a fát 10000 elemmel, ezután inorder módon bejárom a bináris fát, majd felszabadítom a fa pointereit.

6.3 Fabejárás

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

Megoldás videó:

Megoldás forrása: Az előző (6.2) feladat megoldásában megtalálható ennek a feladatnak a megoldása is.

A `preorder` eljárás annyiban különbözik az `inorder` eljárástól, hogy ebben először feldolgozzuk a részfa gyökerét, majd bejárjuk a részfa bal oldalát, aztán pedig a jobb oldalát.

A `postorder` eljárás annyiban különbözik az `inorder` eljárástól, hogy ebben először bejárjuk a részfa bal oldalát, aztán pedig a jobb oldalát, majd feldolgozzuk a részfa gyökerét.

A `usage` eljárás kirírja standard kimenetre, hogy hogyan kell/lehet futtatni a programot. Ebben az esetben három kapcsoló közül kell egyet megadni, annak megfelelően, hogy preorder, inorder vagy postorder módon szeretnénk bajárni a bináris fát.

6.4 Tag a gyökér

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/binfa.cpp

Ez a megoldás nagyban épül az előző feladat (6.3) megoldására. A különbség, hogy a bináris fát kezelő függvényeket és eljárásokat a Binfa osztályba rendeztem, illetve a Binfa osztály privát részéve tettem a Node struktúrát. A binfa osztályon belül túlterheltem a balra bitshift operátort, amely mostmár a bináris fa építését látja el, ugyanazon elven, mint az előző feladatokban az `insert_tree` eljárás.

6.5 Mutató a gyökér

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/binfa_6.5.cpp

Ez a megoldás az előző feladat megoldósának egy módosítása. A különbség, hogy ebben a gyökérelemre is már egy mutató mutat, azért a Binfa konstruktorában létre kell hozni a gyökérobjektumot. Ahol a program eddig a gyökérelem referenciáját adta át függvénynek, vagy a faépítő eljárásban, ott mostmár a gyökérelemet kell átadni és nem a referenciáját.

6.6 Mozgató szemantika

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/binfa_move.cpp

Ez a megoldás a Tag a gyökér (6.4) feladat kiegészítése másoló és mozgató szemantikával. A másoló szemantika lényege, hogy az értékül kapott (esetünkben) bináris fát értékül adja az eredeti fának, minden érték lemásolásával. A mozgató szemantika pedig úgy működik, hogy az eredeti bináris fa gyökerét "kicseréli" az értékül kapott fa gyökerével, és az értékül kapott fa gyökerének gyermekeit nullpointerre állítja, hogy a scope elhagyása után lefutó konstruktor ne törölje le az eredetileg létező fát.

Chapter 7

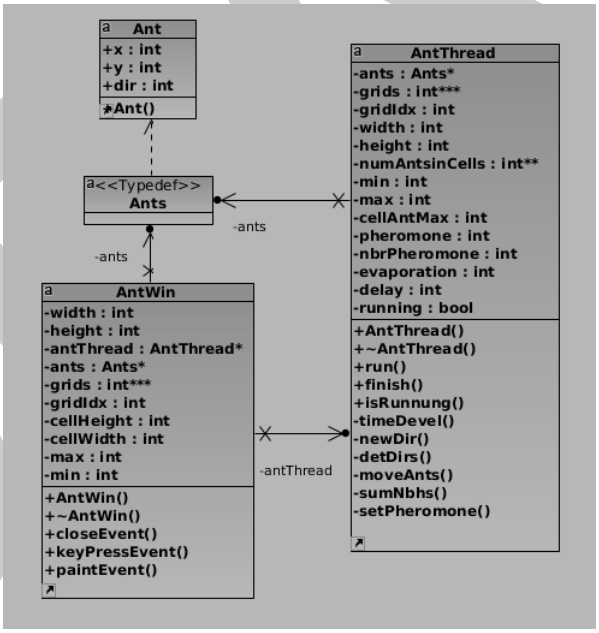
Helló, Conway!

7.1 Hangyaszimulációk

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

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

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



Ennek a programnak a célja hangyák szimulálása, amelyek "utakat" alakítanak ki, illetve azokon az utakon közlekednek. A program futása után kb. egy perccel már látható, hogy a hangyák, ha találnak egy utat, akkor próbálnak azon közlekedni. Az utatakat a halvány kékeszöld négyzetek jelentik.

7.2 Java életjáték

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

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/game_of_life.java

A program elején importálom a program működéséhez szükséges package-eket, például az ablakkezeléshez, illetve az egér-, és billentyűzetkezeléshez. A program egyetlen publikus osztálya a game_of_life. Ebben található a program belépési pontja (`main` függvény). Létrehozok egy RenderArea típusú változót, amelyet majd a game_of_life konstruktorában inicializálok. A konstruktor emellett még beállítja az ablak címét és egyéb tulajdonságait. Az `update` függvény felelős a Conway-féle életjáték szabályainak ellenőrzésére és alkalmazására. A RenderArea private/rejtett osztály felelős a rajzolásért, egér-, és billentyűzetkezelésért, ezeket implementálja. Végül a `main` függvény következik, a program belépési pontja. Létrehoz egy példányt a game_of_life osztályból, majd amíg az életjáték fut, addig ismétli a ciklusmagot. Ha az életjáték futása véget ért, a program kilép.

7.3 Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/game_of_life.cpp

A feladat megoldásához SFML-t használtam. A program két osztályból és három függvényből áll. Az első osztály feladata, hogy kirajzolja a négyzetrácsos hálót, amely négyzetei jelképezik a sejteket. A következő osztály a Square osztály, amely feladata a hálóban lévő sejtek helyének, állapotának tárolása, illetve azok hálón belülre rajzolása. Az első osztályon kívüli függvény az `update` függvény, amely célja, hogy mindeg egyes sejtet megvizsgál, az életjáték szabályai alapján. Ezek a szabályok pedig, ha egy sejt él és 2 vagy 3 szomszédja van, akkor életben marad, ellenkező esetben elpusztul, ha halott és pontosan 3 szomszédja van, élő sejt lesz belőle, ellenkező esetben halott marad. A következő függvény a `killall`, amely célja, hogyha a "játékos" szerkesztő módban van, akkor megöli az összes sejtet. Ez nem szükséges bele. Tesztelés céljából került bele, de hasznos lehet, ha nem akarjuk újraindítani a programot egy másik kezdeti alakzat szimulálásáért. Az utolsó függvény a `main`, amelyben létrehozok egy 1000x1000 pixeles ablakot, majd beállítom, hogy 10 FPS legyen a felső határ megjelenítésnél. Ezután létrehozok néhány változót ,amelyek a játék működéséhez kellenek. Ezek után elérkezünk a fő ciklushoz, amely addig fut, amíg a létrehozott ablak be nem záródik valamilyen módon (erről később). A ciklusban először letörlöm a képernyőt, hogy ne az előzőnek kirajzolt képre rajzoljon rá a program. Majd a belső while ciklus megvizsgálja, hogy történt-e valamilyen esemény (jelen esetben egér és billentyű eseménynek lehetnek ezek, illetve az ablak bezárása). Ha az ablak bezáródik, illetve, ha a felhasználó megnyomja a q betűt, a program leáll. Ha szerkesztő módban bal egérgombbal kattintunk, akkor a kurzor alatt lévő halott sejt élő lesz, és fordítva. A c betű lenyomása esetén, ha szerkesztő módban vagyunk, akkor megöli az összes sejtet. Ezután következik a háló és a sejtek kirajzolása, majd az `update` függvény hívása, ha nem szerkesztő módban vagyunk, illetve minden sejtet frissíti, hogy megkapják az új állapotukat (élő/halott) kirajzoláshoz.

Az életjátékot a cambridge-i egyetem matematikusa, John Conway találta ki. A játékként való megnevezés megtévesztő lehet, ugyanis a játékos egyetlen feladata, hogy megadja a kiindulási alakzatot, majd figyeli, hogy mi jön ki a program futása során a kiindulásialakzatból.

7.4 BrainB Benchmark

Megoldás videó:

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

A program célja azt mérni, hogy mennyire vagyunk képesek figyelemmel kísérni karakterünket, illetve, ha "elveszítjük" a képernyőn, akkor milyen gyorsan találjuk meg újból. Ez főleg online játékokban fontos. Ugyanis online játékokban, például a League of Legends nevű MOBA játékban ez egy elég gyakori probléma, hogy nagyobb összecsapás esetén egyszerűen nem találjuk a képernyőn a karakterünket.

A megoldásomat Fürjes-Benke Péter segítette.

Chapter 8

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

A program elején importájuk a tensorflow modult, amely segítségével töltjük be az MNIST adatbázisban tárolt adatokat. Az x_train, y_train, x_test, y_test változókban eltároljuk az MNIST adatbázisban található képeket, és hozzá tartozó címkéket, illetve a tesztképeket és címkéket. Ezután konvertáljuk az x_train és x_test tömbök értékeit float típusúra, majd normalizáljuk az értékeket, hogy 0 és 1 közé essenek. Importálunk még hat modult, amelyek a neurális háló felépítéséhez szükségesek. Ezután létrehozzuk a neurális háló modelljét, majd feltanítjuk a hálót. Ezután pedig kiértékeljük a neurális háló "tudását" a teszt képekkel és címkékkel.

8.2 Mély MNIST

Python

Megoldás videó:

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/deep_mnist.py

A program elején importájuk a szükséges modulokat. Az importálások után létrehozunk négy függvényt, amelyek a programot hivatottak rövidíteni. A függvények után létrehozzuk a neurélis háló rétegeit. Ebben a programban 5 rétegű a neurális háló. A rétegek megadása után megadjuk, hogy milyen módon szeretnénk kiértékeltetni a bemenetet, majd megadjuk, hogy milyen algoritmussal tanítjuk fel a hálót. Ezután következik a tanítás, illetve minden századik iterációban leteszteljük a háló tudását.

8.3 Minecraft-MALMÖ

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

Megoldás forrása: https://github.com/Ignissen/pjt_prog1/blob/master/malmo.py

A Minecraft-MALMÖ projektet azért hozta létre a Microsoft, hogy a Minecraft játékon belül lehessen mesterséges intelligenciákat létrehozni, tanítani és tesztelni.

Az én programomban alapvetően az a célja Steve-nek, a karakternek, hogy ne akadjon el 5 percen keresztül, miközben folyamatosan halad. Ezt úgy éri el többnyire, hogy megvizsgálja az előtte lévő 12 blokkot, és bizonyos feltételek teljesülése esetén ugrik, vagy elfordul és halad más irányban. Steve még közel sem tökéletes, ugyanis a lávatavakat és a szakadékokat még nem tudja kikerülni.

A programom megírásához sokat felhasználtam a példaprogramokból. A programban először importálom a szükséges modulokat a program futásához, illetve a MALMÖ API-t. A `restart_minecraft` függvény arra szolgál, hogy a mission lefutása után a Minecraft tudja fogadni a következő missiont. A `run` függvényben található a mission pontos megadása, illetve a mission alatti vezérlés. a missionXML string-ben van megadva a világ generálásának szabályai, Steve kiindulási helyzete, illetve itt van megadva, hogy figyelje a körülötte lévő blokkokat. Ezután megadom, hogy 300 másodpercig fusson a mission, illetve, hogy 640x480-as felbontással fusson a Minecraft. Megadom, hogy Minecraft kliens milyen címen és milyen porton éri el, majd megpróbál a program csatlakozni. Az első while ciklus vár addig, amíg a kliens készen áll a program általi irányítást fogadni. A második while ciklusban van Steve vezérlése. A ciklus elején az első két elágazás arra szolgál, hogy Steve át tudja vagy fel tudjon ugrani az akadályokra és ne folyamatosan ugráljon utána. A harmadik elágazás arra szolgál, hogy váltakozva jobbra és balra is forduljon Steve. A következő elégazás arra szolgál, hogy eldöntse, melyik égtáj felé néz Steve. Erre azért van szükség, hogy a megfelelő környező blokkokat lehessen vizsgálni. a blocks nevű kétdimenziós tömbbe töltöm bele a vizsgált blokkok nevét. Ezután a program az alapján, hogy Steve merre néz, megvizsgálja az előtte lévő blokkokat, hogy mit csináljon (ugorjon vagy forduljon).

Chapter 9

Helló, Chaitin!

9.1 Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:

https://github.com/Ignissen/pjt_prog1/blob/master/fact.lisp

https://github.com/Ignissen/pjt_prog1/blob/master/fact_recursive.lisp

Iteratív: A program elején létrehozom a `fact` függvényt, amely kiszámolja a faktoriális értékét. A függvényben létrehozom a sum nevű változót 1 értékkel, majd a ciklusban a sum változót megszorzom x-szel, majd x-et csökkentem 1-gyel. A függvény definiálása után bekérünk standard inputról egy számot, majd a bekért számra meghívom a `fact` függvényt.

Rekurzív: A program elején létrehozom a `factorial` függvényt, amely rekurzív módon kiszámolja a faktoriális értékét. A függvényben megnézem, hogy x egyenlő-e 1-gyel, ha igen, akkor a visszatérési érték 1 legyen, ha pedig nagyobb, akkor a sum értéke x-szer a factorial(x-1) értéke lesz. Az előbb nem a lisp szintaxisa szerint írtam. A függvény létrehozása után meghívom a `factorial` függvényt az 5 számra.

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...

Chapter 10

Helló, Gutenberg!

10.1 Programozási alapfogalmak

[?]

II. heti előadás (11. oldal, az "1.2 Alapfogalmak" című rész):

A programozási nyelvek három szintje van, a gépi nyelv ,ezek azok, amelyek már a processzor nyelvére vannak lefordítva, az assembly szintű nyelv, másnéven gépközeli nyelvek, illetve a magas szintű nyelvek, mint például a java és C++. Minde processzor saját gépi nyelvvel rendelkezik, ezért a forrás szöveget a porcesszor gépi kódjának megfelelő kóddá kell alakítani. Erre két megoldás létezik: a fordítóprogramos és az interpreteres. A fordítóprogramos megoldás a forrásszöveget lefordítja gépi kódra, majd ezután válik futtathatóvá, míg az interpreteres megoldás esetében az interpretes soronként halad végig a forráskódon és olvasási sorrendben hajtja végig az utasításokat, tehát itt nincs szükség futtatás előtti fordításra. Minden programozási nyelvnek van saját hivatkozási nyelve, azaz szabványa. Ebben vannak definiálva a szintaktikai és szemantikai szabályok, legtöbb esetben angolul. Léteznek implementációk, melyek operációs rendszereken való fordítóprogram-, vagy interpreter megvalósítást jelent. Ezek nem kompatibilisek egymással. Létezhet egy operációs rendszeren több implementáció is, ezek sem feltétlen kompatibilisek egymással. Manapság a programozáshoz IDE-ket használunk (Integrated Development Environment), amelyek grafikus programok, amelyekben általában van beépített szövegszerkesztő, fordító, futtatórendszer.

III. heti előadás (28. oldal, a "2.4. Adattípusok" című rész):

Az adatabsztrakció első formája az adattípus. Az adattípus rendelkezik névvel, amely azonosítja a típust, például int, double. Léteznek típusos és nem típusos programozási nyelvek. A típususosok engedik, hogy a programozó adja meg a változók típusát. Ilyenek például a C++ és a Java. A nem típusosok automatikusan állapítják meg a változó típusát. Ilyenek például a R és a Python. Adattípusoknak két csoportja van, az egyszerű és az összetett. Az egyszerű adattípusok azok, amelyeket nem lehet tovább bontani, például int. Az összetett típusok például a struktúrák vagy a felhasználó által definiált típusok.

III. heti előadás (34. oldal, a "2.5. A nevesített konstans" című rész):

A nevesített konstansok azt a célt szolgálják a programokban, hogy a konstansoknak olyan nevet adjunk, amely jelképezi annak típusát és értékét. Illetve másik célja, hogy sokszori használat esetén csak a definiálásnál kelljen változtatni az értékét, ha szükséges. Ezeket a konstansokat mindig definiálni kell.

III. heti előadás (35. oldal a "2.6. A változó" című rész):

A változónak négy komponense van: a név, az attribútomok, a cím és az érték. A név az egy azonosító, a másik három komponenst egy névhez rendeljük hozzá. A legfőbb atribútom, a típus, amely a változó által felvett értéket határolja be. A változóhoz az attribútumok deklarációk segítségével rendelődnek. A deklarációnak különböző fajtáit simerjük: Explicit deklaráció, Implicit deklaráció, Automatikus deklaráció. A változó címe meghatározza a változó értékének a helyét. A címrendelésnek három fajtáját ismerjük: a Statikus tárkiosztás, a Dinamikus tárkiosztás, és a programozó által vezérelt kiosztás. A változó értékének a meghatározására több opció is van: értékadó utasítás, kezdőérték adás.

III. heti előadás (39. oldal, az "2.7. Alapelemek az egyes nyelvekben" című rész):

C-ben az aritmetikai típusok az egyszerű típusok, a származtatottak az összetett típusok. A karakter típus elemeit belső kódok alkotják. Logikai típus nincs, a hamis az int 0 az igaz pedig az int 1. A struktúra egy fix szerkeztű rekord. A void tartománya üres. A felsorolásos típusok nem fedhetik egymást. Különböző elemekhez ugyanazt az értéket hozzárendelhetjük.

IV. heti előadás (46. oldal, az "3. Kifejezések" című rész):

A kifejezések szintaktikai eszközök. A kifejezések formálisan három dologból állnak: operandusokból, operátorokból, kerek zárójelekből. Létezik egyoperandusú(unáris), kétoperandusú(bináris) és háromoperandusú(ternáris) operátor, ezek attól függnek, hogy egy operátor hány operandussal végzi a műveletet. A kifejezéseknek három alakja lehet: a prefix, az infix, a postfix. A folyamat, amikor a kifejezés értéke és típusa meghatározódik, az a kifejezés kiértékelése. A kifejezéseknek van két típusa: a típusegyenértékűség, és a típuskényszerítés. Azt a kifejezést, amelynek értéke fordítási időben eldől, és a kiértékelését a fordító végzi, azt konstans kifejezésnek hívjuk.

V. heti előadás (56. oldal, az "4. Utasítások" című rész):

Az utasítások megalkotják a programok egységeit: az algoritmusok egyes lépései, a fordítóprogram ezzel generálja a tárgyporgramot. Két csoportjuk van: a deklarációs utasítások, és a végrehajtó utasítások. A deklarációs utasítások mögött nem áll tárgykód, a fordítóprogramnak szólnak. A végrehajtó utasításokból pedig a fordító generálja a kódot. A végrehajtó utasításokat csoportosíthatjuk: értékadó utasítás, üres utasítás, ugró utasítás, elágaztató utasítás, ciklusszervező utasítás, hívó utasítás, vezérlésátadó utasítás, I/O utasítás, egyéb utassítás. A vezérlési szerkezetet megvalósító utasítások: ugró utasítás, elágaztató utasítás, ciklusszervező utasítás, hívó utasítás, vezérlésátadó utasítás.

VII. heti előadás (78-84. oldal):

A paraméterátadásnak többféle módja is lehet, ezek nyelvfüggőek, hogy melyik nyelv melyiket alkalmazza.

Történhet érték szerint, mint a C-ben például. Ekkor a formális paraméter értékül kapja az aktuális paraméter értékét. Ennél a módszernél a függvényben nem lehet megváltoztatni a aktuális paraméter értékét. Lehet címszerinti a paraméterátadás. Ekkor a formális paraméter címe értékül kapja az aktuális paraméter címét. Ilyenkor a függvényben meg lehet változtatni az aktuális paraméter értékét. Lehet eredmény szerinti átadás is, ekkor a formális paraméter szintén megkapja az aktuális paraméter címét, de nem használja, csak a végén beletölti az adatokat. Létezik még érték-eredmény szerinti, ekkor másolódik a cím szintén, és használja is az adatokat, majd a függvény végén belemásolja a formális paraméterbe az adatokat.

VIII. heti előadás (82. oldal, a "A blokk" című rész):

A VI. heti előadáson már volt róla szó. Ott található A blokk című rész lényege.

VIII. heti előadás (83. oldal, a "Hatáskör" című rész):

A hatáskör szinonímája a láthatóság. Egy név hatásköre a program szövegének azon részét jelenti, ahol az adott név ugyanazt a programozási eszközt hivatkozza. A név hatásköre a programegység. A programegységben a deklarált nevet, a programegység lokális nevénk nevezzük. Azt a nevet, amelyre csak

a programegységben hivatkozunk(nem deklaráljuk) szabad névnek hívjuk. Hatáskörkezelésnek hívjuk azt a folyamatot, amikor megállapítjuk egy név hatáskörét. Két fajtája van, a hatáskörkezelésnek az egyik a statikus, a másik pedig a dinamikus. A hatáskör mindig befelé terjed, kifelé soha. Ha egy név nem lokális egy programegységben, de onnan látható, azt globális névnek hívjuk. A globális név és a lokális név relatív fogalmak. Statikus hatáskörkezelésnél a programban szereplő összes név hatásköre a forrásszöveg alapján egyértelműen megállapítható. Dinamikus hatáskörkezelésnél, viszont a hatáskör futási időben változhat és minden futásnál lehet. Az eljárásorientált nyelvek a statikus hatáskörkezelést valósítják meg.

VIII. heti előadás (98. oldal, a "Absztrakt adattípus" című rész):

Olyan adattípus, amely megvalósítja a bezárást vagy információ rejtést. Az ilyen típusú programozási eszeközök műveleteihez a specifikációi által meghatározott interfészen keresztül férhetük hozzá. Így az értékeket véletlenül vagy szándékosan nem ronthatjuk el(biztonságos programozás). Az elmúlt évtizedekben nagyon fontos fogalommá vált és befolyásolta a nyelvek fejlődését.

VIII. heti előadás (121. oldal, a "Generikus programozás" című rész):

A generikus programozás az újrafelhasználhatóság, és így a procedurális absztrakció eszköze. Bármely programozási nyelvbe beépíthető. A generikus programozás lényege: Megadunk egy paraméterezhető forrásszöveg-mintát, ami majd fordítási időben lesz kezelve. A mintaszövegből paraméterek segítségével előállítható egy lefordítható konkrét szöveg. Az újrafelhasználás alatt azt értjük, hogy egy mintaszövegből tetszőleges számú konkrét szöveg generálható, a mintaszöveg típussal is paramétertezhető. A generikus formális paramétereinek száma mindig fix. A paraméterkiértékelésnél a kötés az alapértelmezett, de alkalmazható a név szerinti kötés is. A paraméterátadás változónál értékszerinti, típusnévnél pedig névszerinti.

IX. heti előadás (134. oldal, az "Input/Output" című rész):

Az I/O platform-, operációs rendszer-, implemetációfüggő. Léteznek nyelvek, amelyek nem tartalmaznak eszközt, így az implementációra bízzák a megoldást. Az I/O a programnyelvekben egy eszközrendszer, amely a perifériákkal való kommunikációért felel. Az I/O közzéppontjában az állomány áll. Logikai állomány egy olyan programozási eszköz, amelynek van neve, illetve amelynél az absztrakt állományjellemzők attribútumként jelennek meg. A fizikai állomány operációs rendszer szintű, konkrét, a perifériákon megjelenő, az adatokat tartalmazó állomány. Egy állomány funkció szerint lehet: Input állomány, Output állomány, Input-Output állomány. Az I/O során adatok mozognak a tár és a periféria között. Kérdés, hogy az adatmozgatás közben történik-e konverzió. Ennek megfelelően kétféle adatátviteli mód létezik: folyamatos(van konverzió) vagy a bináris másnéven rekord módú(nincs konverzió). A folyamatos módú átvitelnél a tárban és a periférián eltér a reprezentáció. A nyelvekben három alapvető eszközrendszer alakult ki: formátumos módú adatátvitel, szerkesztett módú adatátvitel, listázott módú adatátvitel. A bináris adatátvitelnél az adatok a tárban és a periférián ugyanúgy jelennek meg, ez csak a háttértáraknál való kommunikációnál jöhet szóba. Az átvitel alapja itt a rekord. Ha állományokkal akarunk dolgozni, akkor a következőket kell végrehajtanunk: Deklaráció: A logikai állományt mindig deklarálni kell, el kell látni a megfelelő névvel és attribútumokkal. Összerendelés: A logikai állománynak megfeleltetünk egy fizikai állományt a háttértáron. Állomány megnyitása: Egy álománnyal csak akkor tudunk dolgozni, ha megnyitottuk. Feldolgozás: Ha az állományt megnyitottuk, akkor abba írhatunk, vagy olvashatunk belőle. Lezárás: A lezárás operációs rendszer rutinokat aktivizál, meszünteti a kapcsolatot a logikai és a fizikai állomány között. Az implicit input állomány a szabvány rendszerbemeneti periféria, az implicit output állomány a szabvány rendszerkimeneti periféria. C-ben az I/O eszközrendszer nem része a nyelvnek. Ezek az utasítások/függvények a standard könyvtárak részei.

XI. heti előadás (112. oldal, a "Kivételkezelés" című rész):

A kivételkezelés lehetővé teszi, hogy az operációs rendszertől átvegyük a megszakítások kezelését. A kivételek olyan események, amelyek megszakítást okoznak a program futása közben. A kivételkezelés az a

tevékenység, amit a program végez, ha egy kivétel következik be. A kivételkezelő egy olyan programrész, amely egy adott kivétel bekövetkezése után lép működésbe. A kivételkezeléssel az eseményvezérlést teszi lehetővé a programozásban. Lehetőségünk van, akár nyelvi szinten is maszkolni a megszakításokat. Egyes kivételek figyelése letiltható vagy engedélyezhető. A kivételeknek általában van neve, és kódja is.

Milyen beépített kivételek vannak a nyelvben? Például a memóriaelérés vagy a nullával való osztás stb.

Definiálhat-e a programozó saját kivételt? Igen.

Milyenek a kivételkezelő hatáskör szabályai? Van, de ez akár lehet az egész program is.

Hogyan folytatódik a program a kivételkezelés után? Futhat tovább, de hibától függ, hogy le kell állítai(kernel megállítja) vagy le áll magáól.

Van-e a nyelvben beépített kivételkezelő? Igen van.

10.2 Magas szintű programozási nyelvek 2 by Juhász István (Pici könyv 2)

XI. heti előadás (38. oldal, a "Kivételkezelés a Javaban" című rész):

Alapvető eszköze a Java-nak. Használatakor létrejön egy kivétel objektum: vannak kivétel-osztályok és azoknak kivétel-példányai. A "JVM" feladata, hogy megkeressen egy adott objektumnak megfelelő típusú, az adott pontban látható kivételkezelőt, amely kivételkezelő az adott kivételt elkapja. Egy kivételkezelő megfelelő típusú, ha a kivételkezelő típusa megegyezik a kivétel típusával, vagy ha a kivételkezelő típusa őse a kivétel típusának. A láthatóságot, maga a kivételkezelő definiálja, ami egy blokk. A kivételkezelő a tetszőleges kódrészlethez köthető a JAVA-ban, és egymásba ágyazhatóak. JAVA-ban két fajta kivételt különböztetünk meg: az ellenőrzött(egy metódus láthatósági körében léphet fel), és a nem ellenőrzött(ellenőrzése vagy nagyon kényelmetlen vagy lehetetlen) kivételt. Egy módszer fejében tehát meg kell adni, azokat az ellenőrzött kivételeket, melyeket a módszer nem kezel, de futás közben bekövetkezhetnek, ez a "THROW kivételnév_lista" utsaítás segítségével történik. A kivételek kezeléséhez a "java.lang" csomagban definiált ősosztálya "Throwable" objektumai dobhatók el. Két standard alosztály van: az Error(rendszerhibák, ezek nem ellenörzöttek) és a Exception(ellenőrzött kivételek osztálya, innen származtathatunk saját ellnőrzött kivételeket). A "catch" ág teljesen hiányozhat. A típusegyeztetés miatt a felírás sorrendje nagyon lényeges, ugyanis a "catch" ág az ellenőrzött kivételt kapja el az első megfelelő típusú kivételkezelőnél.

10.3 Programozás bevezetés

[KERNIGHANRITCHIE]

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

V.heti előadás (Vezérlési szerkezetek című fejezet):

Egy nyelv vezérlésátadó utasításai az egyes műveletek végrehajtási sorrendjét határozzák meg. A C nyelvben a pontosvesző az utasításlezáró jel. A kapcsos zárójelekkel deklarációk és utasítások csoportját fogjuk össze egyetlen összetett blokba. Az "if-else" utasítás döntés kifejezésére használjuk, az utasítás először kiértékeli a kifejezést, és ha ennek az értéke igaz, akkor az első utasítást hajtja végre, ha a kifejezés értéke viszont nem igaz, és van "else" rész, akkor a második utasítás hajtódik végre. Általános szabály, hogy az "else"

mindig a hozzá legközelebb eső if utasításhoz tartozik. A switch utasítás is a többirányú programelágazás egyik eszköze. Összehasonlítja egy kifejezés értékét több egész értékű állandó kifejezés értékével, és az ennek megfelelő utasítást hajtja végre. A "switch" -ben sok "case" és egy "default" található. A "default" akkor hajtódik végre, ha egyik "case" ághoz tartozó feltétel sem teljesül. A while - for szerkezet először kiértékeli a kifejezést, ha ennek az értéke nem nulla, azaz hamis, akkor az utasítás végrehajtódik, ez addig ismétlődik, amíg nulla, azaz hamis, nem lesz a kifejezés értéke. A "do - while" szerkezet először végrehajtja az utasítást,és csak utána értékeli ki a kifejezést. Ha a kifejezés értéke igaz, akkor az utasítást újból végrehajtják. Ez addig ismétlődik, amíg a kifejezés értéke hamis nem lesz. A "break" lehető teszi, hogy elhagyjuk a ckliusokat, még idő előtt(for,while, do, switch). A "continue" utasítás a "break" utasításhoz kapcsolódik. hatására azonnal megkezdődik a következő iteráció lépés. A "goto" utasítás, akkor előnyös, ha ki akarunk lépni egy több szinten egymásba ágyazott ciklusból(a "break" egyszerre csak egy ciklusból tud kilépni). A címke ugyanolyan szabályok szerint alakítható ki, mint a változók neve és mindig kettőspont zárja.

V.heti előadás (Függelékből az Utasítások című fejezet):

Az utasítások a leírásuk sorrendjében hajtódnak végre, általános a szintaktikai leírásuk, és számos csoportba sorolhatók: Címkézett utasítások, például a case és default címkék, amelyek a switch utasítással használhatók,illetve a goto utasítás (amely használatát kerülni kell), amelyhez mi adhatjuk meg a címkéket. A címke egy azonosító nélküli deklarált azonosítóból áll. Kifejezésutasítsok, az utasítások(kifejezésutasítás, értékadás, függvényhívás) többsége ilyen. Összetett utasítás, több utasítást egyetlen utasításként kezeli, ez a fordításhoz szükséges, mivel sok fordítóprogram csak egyetlen utasítást fogad el. Kiválasztott utasítások, minden esetben a ehetséges végrehajtási sorrendek egyikét választják ki(if, if-else, switch). Iterációs utasítások, egy ciklust határoznak meg(while, do-while, for). Vezérlésátadó utasítások, vezérlés feltétel nélküli átadására alkalmasak(goto, continue, break, return).

10.4 Programozás

[BMECPP]

V.heti előadás (1.-16.):

A C++ a C-nek a továbbfejlesztése. A C++ sok problémára biztonságosabb, és kényelmesebb megoldást kínál, mint a C. C-ben üres paraméterlistával definiálunnk, akkor az tetszőleges számú paramáéterrel hívható. A C++-ban azonban az üres paraméterlista egy "void" paraméter megadásával megegyező. C nyelvben is létezik több bájtos sztring. C++-ban miinden olyan helyen állhat változódeklaráció, ahol utasítás állhat. A C nyelvben a neve azonosít egy függvényt, C++-ban viszont a függvényeket a nevük, és az argumentumlistájuk azonosítja. Míg a C nyelv úgy hivatkozik egy függvényre a linker szintjén, hogy egy aláhúzást tesz a függvénynevek elé, addig a C++ az egyes fordítókra bízza a névferdítés implementálását. Cím szerinti paraméterátadás, ha a változó címét adjuk át, ebben az esetben nem tudjuk megváltoztatni úgy a változót, hogy az értéke megmaradjon. Az érték szerinti paraméteradásnál viszont, készül másolat a változóról, így végezhetünk műveleteket úgy, hogy a változó értékét nem befolyásoljuk. A C++ referenciatípus bevezetése feleselgessé teszi a pointerek cím szerinti pareméterátadását.

VI. heti előadás (17-58.):

Ez a fejezet a C++ osztályairól szól. Az objektum orientált programozás alapelve, hogy a probléma megoldását segítse azzal, hogy az emberi gondolkodáshoz közelebb hozza a programozást az osztályok és objektumok bevezetésével. Az egységbe zárás jelenti azt, hogy az összetartozó változók és függvények egy egységben legyen, ezek lesznek az adattagok és a tagfüggvények. Adatrejtés a private és protected

adattagok és tagfüggvények bevezetésével jött létre. Az adatrejtés célja, hogy az osztály egyes tagjait ne lehessen kívülről elérni. A konstruktor szerepe, hogy lefusson, amikor létrejön az objektum, ezáltal akár inicializálva az adattagokat. A destruktor célja, hogy lefusson, amikor az objektum megsemmisül, ezáltal akár felszabadítva a diamikus adattagokat. A dinamikus adattagok osztályon belüli pointerek, amelyeket futásidőben hozzuk létre dinamikus memóriafoglalással, ezért ezeket, ha már nincs rájuk szükség, de legkésőbb a destruktorban fel kell szabadítani. A friend osztályok, illetve függvények olyan osztályok, illetve függvénynek, amelyek ugyan nem tagjai az osztálynak, viszont hozzáférnek azok private tagjaihoz. A tagváltozók inicializálása történhet a konstruktoron belül, illetve tagfüggvénnyel, vagy külső függvénnyel is. A statikus tagok azzal a tulajdonsággal rendelkeznek, hogy nem kell az osztályt példányosítani, hogy használni tudjuk. Az osztályok tartalmazhatnak beágyazott definíciókat, amelyek lehetnek enumerációk, struktúrák vagy akár osztályok is.

VII. heti előadás (93-96.):

Ez a fejezet az operátokról és azok túlterheléséről szól C++ nyelvben. Az operátorok alapértelmézés szerint az argumentumain hadjtanak végre műveletet, amelyek visszatérési értékével tudunk dolgozni. C++-ban nem lehet új operátorokat létrehozni, azonban majdnem mindet túl lehet terhelni. A túlterhelés célja, hogy bizonyos operátorokat más célra használhassunk, mint az erdeti célja, a programozás megkönnyítése céljából, mint például a bitshift operátor túlterhelése az alapértelmezett IO objektumoknál(cin,cout,cerr).

IX. heti előadás (73.-90.):

A C nyelvben három állomány leíró áll a rendelkezésünkre: "stdin" a standard(alapértelmezett) bemenet, "stdout" a standard kimenet, és az "stderr" pedig a hibakimenet. Ezek mindegyike "File" típusú. Az I/O használatához be kell építenünk az "iostream" állományt. A programban a beolvasást a "cin" objektummal, a kiírtaást pedig a "cout" objektummal végezzük. Érdemes figyelni a "nyilakra", mert az irányükk függ attól, hogy éppen be olvasunk, vagy kiíratunk. A cin állpotát beolvasás után mindig ellenőriznünk kell. Az "ignore" függvény segítségével megadhatjuk, hogy a beolvasás a sor végéig történjen, mert alapból egy írásjel(, . : stb.), illetve a whitespace karakterek megtörik a beolvasást. Megadhatjuk a maximális adatfolyam-méretét a "limits" segítségével. Így megelőzhetjük a beolvasások egymásra futását. A rendszerhívások költsége igen nagy, ezért az adatfolyamokat egy bufferrel látják el. Ezek a bufferek összegyűjtik a karaktersorozatokat, és több "cout" kiírást egy rendszerhívással írnak ki a képernyőre. A buffereket a "flush" segítségével lehet üríteni. Az adatfolyam állapotát az "iostat" típusú tagváltozó jelzi, ennek állapotát az alábbi konstansokkal lehet beállítani: eofbit(adatfolyam elérte az állomány végét), failbit(formátum hibát jelez), badbit(fatális hibát jelez), goodbit(jelzi, hogy minden rendben van). A jelzők beállítását a "clear" tagfüggvény végzi. Nagyon fontos, hogy ha egy adatfolyam bármelyik hibabitje beállítódik, akkor az összes utána következő írási, és olvasási művelet, azon az adatfolyamon hatástalan marad(lefut, de nem tesz semmit). A C++ tartalmaz egy "string" osztályt, amely szükség szerint változtatja a méretét. Ha egy ilyen "stringet" szeretnénk beolvasni, akkor azt a "std::getline" fügvénnyel tehetjük meg, mert a szólözöknél megakadna a beolvasás. Az adatfolyam-objektumoknak vannak tagfügvényeik, amelyekkel beéllíthatjuk az állapotát(olvashatunk, írhatunk, műveleteket végezhetünk). Használhatunk még manipulátorokat is. Az I/O manipulátor egy adatfolyam-módosító speciális objektum. Vannak jelzőbitek is, olyan bitek, amelyeket bállíthatunk, vagy törölhetünk. Minden bithez tartozik egy bináris szám: ebben a bináris számban csak az a bit egyes, ahányadik biten az adott tulajdonságot beállítjuk. A C++ I/O-hoz kapcsolódó jelzőbitek az "ios" nevű osztályban vannak definiálva. Az alábbi dolgokat formázhatjuk: mezőszélesség, a kitöltő karakter, igazítás, az egész számok számrendszere, a lebegőpontos számok formátuma, mutassa-e a "+" jelet, a helykitöltő nullákat, ill. az egész számok számrendszerének alapját, kis- vagy nagybetűk. ha egyik jelzőbit sincs beállítva, akkor az adatfolyam a legjobb formázást próbálja kiválasztani egy adott számra, annak nagyságától függően. A C állománykezelése egy "FILE" típusú leíró köré csoportosul, amelyet az "fopen" függvény ad vissza. A C++ az állománykezeléshez is adatfolyamatokat használ, amelyeket ezúttal az "ifstream" (input

file stream), illetve az "ofstream" (output file stream) osztályok reprezentálnak. A kétirányú adatfolyamot az "fstream" osztály valósítja meg. Az állományok megnyitását a konstruktorok végzik, a lezárását pedig a destruktorok. Ha a konstruktor, vagy a destruktor nem felel meg nekünk, akkor létezik "open", illetve "close" függvény is erre a célra. Az állomány-adatfolyamosztályok ddefiníciói az "fstream" fejlécben találhatók az "std" névtérben. A jelzőbiteket az "ios::" előtaggal kell ellátni. A bemeneti adatfolyamok esetén az olvasási pozícionálást a "get", míg a kimneneti adatfolyamatok esetén az írási pozícionálást a "put" végzi. A "tell" függvények visszatérési típusa a "pos_type", amely nem egész jellegű. A pozíciókat el is tárolhatjuk, ezzeket a "seek" függvényeknek adható. A "cin", "cout", "cerr" és "clog" esetén nem használhatunk pozícionáló függvényeket. Fájlrendszerbeli állomány esetén nem válthatunk akárhogyan az olvasás és az írás között. Általában egy pozícionáló műveletet kell végeznünk.

IX. heti előadás (165.-178.):

C nyelvben az "enum" és az "int" típus között oda-vissza létezik implicit konverzió. A C++-ban viszont, ha "enum" típusra konvertálunk, akkor jelölnünk kell ezt a típuskonverziót. A C automatikus konverziót biztosít a "void*" típusú pointer és tetszőleges típusú pointer között oda-vissza, a C++-ban ezt a konverziót is ki kell írnunk. Nem konstans referenciára nincs automatikus konverzió inkompatibilis típusok referenciáiról. Referenciát akkor is használunk, ha meg akarjuk takaírtani függvényhívásból eredő másolást. A konstans referencia alkalmazásával nagyobb objektumok átadása esetén jelentős másolási költséget takaríthaunk meg. A típuskonverziós lehetőségek két fontos esetét különböztetjük meg: az öröklés szempontjából két független típus közti típuskonverzió, és az öröklés hierarchia mentén típuskonverzió. C++-ban ha egy másik típusról szeretnénk konvertálni a mi osztályunk típusára, akkor a konverziós konstruktor jelent megoldást. Ha az osztályunkról szertnénk egy másik típusra konvertálni, akkor a konverziós operátort érdemes használnunk. A konstruktor képes konverziót végrehajtani. A konverziók leggyakoribb hibája, hogy bizonyos kifejezések esetén több megoldás is létezik, és a fordító nem tud választani. Ilyenkor a C++ hibaüzeneteket ad ilyen esetekben. ha a típuskonverziós útvonal nem egyértelmű, akkor fordítási hibát jelent. Ha az adottt pointer, illetve referencia mögött nem olyan leszármazott van, amire konvertálunk, a viselkedés durva futási idejű hiba lesz. Az explicit típuskonverziót C nyelven a kifejezés elé () zárjólelek közé írt új típus megadásával definiálhatjuk. A C++ saját konverziós operátrokat definiál, amelyek jobban kifejezik a típuskonverzió jelentését. Az alábbi operátorok segítenek, hogy pontosabban meg tudjuk adni a konverzió célját: static_cast(statikus típuskonverzió), const_cast(konstans típuskonverzió), dynamic_cast(dinamikus típuskonverzió) és a reinterprent_cast(újraértelmező típuskonverzió). A C stílusú típuskonverzió helyett leggyakrabban a statikus típuskonverziót használjuk. A statikus típuskonverziónak megmaradtak azok a megkötései, amely a C stílusú elődjének pl.:nem konvertálhat struktúrát egész típussá vagy konstans típust nem konstans típussá. Erre van külön típuskonverzió operátor: a konstans típuskonverzió. A konstans típuskonverzió képes egyedül konstans típust nem konstanssá tenni, illetve "volatile" típust nem azzá tenni. A dinamikus típuskonverzió szintén speciális típuskonverziót valósít meg: az öröklési hierarchián lefelé történő konverziókhoz szükséges. Az újraértelmezhető típuskonverzió az implementációfüggő konverziók esetén használható. Általában pointerekre alkalmazzuk.

XI. heti előadás (187.-197.):

A kivételkezelés olyan mechanizmus, amely biztosítja, hogy ha megszakítást detektálunk valahol, akkor a futás a hibakezelő ágon folytatódjon. A megoldás nem csak hiba, hanem bármilyen "kivételes" helyzet esetén használható, ezért hívják kivételkezelésnek.

Egy példa a kivételkezelésre:

```
    #include <iostream>
```

```
    usning namespace std;

    int main()
    {
      try
      {
        double d;
        cout << "Enter a nonzero number: ";
        cin >> d;
        if(d == 0)
        {
         throw "The number can no be zero.";
        }
        cout << "The reciprocal is: " << 1/d << endl;
      }
      catch (const char* exc)
      {
        cout << "Error! The error text is: " << exc << endl;
      }
      cout << "Done." << endl;
    }
```

Bekérünk a felhasználótól egy nem nulla számot, majd ezt eltároljuk a "d" nevű változóban. Utána a "d" változóban eltárolt számot ellenőrizzük, hogy nem nulla e. Erre kell az "if", ha ez nulla akkor a "throw" segítségével kidobjuk, mint lehetséges hibát. Ha nem nulla, akkor meghatározzuk a reciprokát, és kiíratjuk. A "catch" részben elkapjuk a "throw" által eldobott hibát, és kiírjuk, az "if"-ben megadott mondatot. Mindkét megoldás végén a program kiírja, hogy "Done.".

```
A kimenet, ha a felhasználó nem nullát ad meg:

    Enter a nonzero number: 2
    The reciprocal is: 0,5
    Done
```

```
A kimenet, ha a felhasználó nullát ad meg:

    Enter a nonzero number: 0
    Error! The error text is: The number can not be zero.
    Done.
```

A "try-catch" blokkok egymásba ágyazhatóak. Így lehetőségünk van arra, hogy bizonyos kivételeket a dobott kivételhez közel, alacsonyabb szinten kapjuk el és kezeljük. Az elkapott kivétel a "throw" kulcsszó paraméter nélküli alkalmazásával újradobható. Egy kivétel dobásakkor annak elkapásáig a függvények hívási láncában felfelé haladva az egyes függvények lokális változói felszabadulnak. Ez a folyamat a hívási verem visszacsévélése.

```
A verem visszacsévélése:

    int main()
    {
     try
     {
       f1();
     }
     catch(const char* errorText)
     {
       cerr << errotext << endl;
     }
    }

    void f1()
    {
      Fifo fifo; //a fifo egy általunk megírt osztály
      f2();
      ...
    }

    void f2()
    {
      int i = 1;
      throw "error1";
    }
```

A lépések a példában:

Először az "f2" kivételt dob, ezután az "f2"-ben definiált "i" lokális változó felszabadul. Majd az "f1"-ben lefoglalt "Fifo fifo" objektum felszabadul, meghívódik a destruktorra. Végül pedig lefut a "main" függvényben lévő "catch" blokk.

A kivétel elkapása, és dobása között futhat le kód, mivel meghívódnak a verem visszacsévélése során felszabadított objektumok destruktorai. Fontos, hogy a kivétel dobása, és elkapása között ne dobjunk újabb kivételt, mert az a kivétel kezeletlen marad.

XI. heti előadás (211.):

```
Erőforrás kezelés:

    class MessageHandler
    {
     public:
        void ProcessMessage(istream& is)
        {
          Message *pMessage;
          //Következő üzenet beolvasása.
          while((pMessage = readNextMessage(is)) != NULL)
          {
```

```
          try
          {
            //Kivételt dobhat!
            pMessage->Process();
            // ...
            // Ha végeztünk, felszabadítjuk a Message objektumot.
            delete pMessage;
          }
          catch(...)
          {
            delete pMessage;
            throw;
          }
        }
      }
    private:
      Message* readNextMessage(istream& is)
      {   ... }
  };
```

Példa magyarázata:

A "MessageHandler" egy bemeneti folyamból üzeneteket kiolvasó, és feldolgozó osztály. A "ProcessMessage" tagfüggvénye mindaddig "Message" objektumokat olvas a bemeneti folyamból a "readNextMessage" függvény meghívásával, amíg az "NULL"-al nem tér vissza. "readNextMessage" működése: a new operátorral létrehoz egy "Message"-t, a hozzá tartozó adatokat kiolvassa a folyamból, és visszatér a "Message" objektumra mutató pointerrel. A "ProcessMessage" a "readNextMessage" hívását követően olyan függvényeket hív, melyek feldolgozzák a beolvasott üzenetet, ezt követően pedig a "delete" operátorral felszabadítjuk a "Message" objektumot. Probléma akkor van, ha az üzenet a feldolgozást végző fügvények kivételt dobnak, mert így akkor nem tudjuk felszabadítani az utoljára beolvasott "Message" objektumot. Erre a problémára a megoldás a "try-catch", mivel az üzenet feldolgozása során bármilyen kivétel keletkezik, a "catch"-el elkapjuk, felszabadítjuk a helyileg lefoglalt memóriát, majd újradobjuk a kivételt. A kivétel újradobása nagyon fontos, hiszen ha ezt kihagyjuk, akkor a hiba rejtve marad.

Part III

Második felvonás

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

Chapter 11

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...

Part IV

Irodalomjegyzék

11.3 Általános

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

11.4 C

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

11.5 C++

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

11.6 Lisp

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

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

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