Univerzális programozás

Így neveld a programozód!

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

COLLABORATORS

	TITLE : Univerzális programozás	

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

REVISION HISTORY

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

Ajánlás

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

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

Tartalomjegyzék

Előszó

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

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

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

Hogyan forgasd

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

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

Milyen nyelven nyomjuk?

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

Hogyan nyomjuk?

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

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

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

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

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

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

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

1.2. Milyen doksikat olvassak el?

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

1.3. Milyen filmeket nézzek meg?

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

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

II. rész

Tematikus feladatok

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

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

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

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI Turing/ciklus.c

Egy magot 100%-on dolgoztatni egész egyszerű feladat, hiszen ha egy szimpla while ciklust megírunk, az alapvetően így működik. Egy magot 0%-on dolgoztatni sem egy egetrengető kihívás, viszont itt már kell minimálisan gondolkodni. De hamar rájövünk, hogy a sleep(x) parancs kiadásával x másodpercig nem használja a processzort a program. Viszont ha nincs parancs a cikluson belül, a processzor ugyanúgy 100 %-on dolgozik, ami abból adódik, hogy az operációs rendszer elvégzendő feladatnak tekinti és neki adja az összes processzoridőt. Viszont az összes magot 100 %-on dolgoztatni nehezebb feladatnak bizonyult. Az egyik ismerősöm tanácsára az OpenMP-t kezdtem el nézegetni a feladat megoldásához. Pár fórumbejegyzés és egy kis utánajárás után pedig sikerült a feladatot megoldanom.

A programot roppant egyszerű használni. Ha egy magot szeretnénk 100%-ban dolgoztatni, akkor semmit nem kell módosítani, szimplán csak le kell fordítani és futtatni.

Ha egy magot szeretnénk 100%-ban dolgoztatni, akkor vegyük ki a //-t a

```
//sleep(1)
```

függvényhívásból.

Ha pedig az összes magot szeretnénk 100%-ban dolgoztatni, akkor ugyanúgy a //-t kell kitörölni a következő helyről:

```
#pragma omp parallel while
```

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:

Nem tudunk olyan programot írni, ami minden más programról eldönti, hogy van e benne végtelen ciklus. Mivel, ha tudnánk, akkor már valószínűleg lett volna olyan ember, aki ezt a programot megírja.

De tegyük fel, hogy megírjuk ezt a programot, aminek a neve legyen eldöntő. Annak a programnak a neve, amelyről el kell dönteni, hogy van e benne végtelen ciklus, legyen eldöntendő. Nyílván az eldöntő bemeneti argumentuma lesz az eldöntendő. Ahhoz, hogy eldöntő megállapítsa, hogy van e eldöntendőben végtelen ciklus, futtatnia kell az eldöntendő kérdéses részleteit. Ekkor ha az eldöntendő programban nincs végtelen ciklus, eldöntő hamissal tér vissza, ami azt jelenti, hogy nincs eldöntendőben végtelen ciklus.

Azonban ha az eldöntendő programban tényleg van egy végtelen ciklus, és azt eldöntő futtatja, hogy megbizonyosodjon róla, akkor eldöntő maga is egy végtelen ciklussá válik. Éppen ezért eldöntő sose fog igazzal visszatérni, mert minden ilyen esetben ő is le fog fagyni.

2.3. Változók értékének felcserélése

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

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

Megoldás forrása:https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyN Turing/felcserelo.c

Annak ellenére, hogy nem használhatunk logikai utasításokat/kifejezéseket, ez egy egyszerű feladatnak bizonyul egy kezdő programozó számára is. Itt most négy módját is bemutatom.

A program bekér két számot és eredményül a kettő értékének a felcseréltjét adja vissza.

```
  int main()

{
    int a;
    int b;
    int c;
    printf("Kerem a felcserelni kivant szamokat! \n");
    scanf("%d",a);
    scanf("%d",b);
    printf("A ket szam: \n");
    printf("A =" "%d",a);
    printf("B = " "%d",b);

    //Elso valtozat:
    /*
        c = a;
        a = b;
        b = c;
    */
    //Masodik valtozat:
    /*
        a = a-b;
```

```
        b = a+b;
        a = b-a;
    */
    //Harmadik valtozat:
    /*
        a = a*b;
        b = a/b;
        a = a/b;
    */
    //Negyedik valtozat:
    /*
        a=a^b;
        b=a^b;
        a=a^b;
    */
    printf("A ket szam felcserelve: \n");
    printf("A =" "%d",a);
    printf("B = " "%d",b);

    return 0;
}
```

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 Bátfai Norbert tulajdona.

Ez egy egyszerű program, ami a grafikus megjelenítést imitálja. Egy ladbádnak álcázott o betűt mozgat egy while ciklus segítségével. Ez a program egy nagyon jó kezdet a grafikus felületű programokkal való megismerkedéshez.

A program két fő részből áll. Az egyik egy függvény, ami a labdát rajzolja ki a konzolra, A másik pedig maga a main.

A main-ben először létrehozunk egy maxX és egy maxY változót, amiket át is adunk a tx és a ty tömbök méretének.

Ezután két for ciklus végigmegy a két tömbön, a második, és az utolsó elemek értéke -1 lesz, a többi elemé pedig 1

végül pedig egy while ciklus és a függvény segítségével kiírja a konzolra a labdát.

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

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

Megoldás videó: https://www.youtube.com/watch?v=xLu3I6lBH-E

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/bogo.cpp

A BogoMips a processzorunk sebességének meghatározásához használatos mértékegység. Azt mondja meg, hogy a számítógép processzora mekkora szóhosszal dolgozik.

Ezt a XOR ^ művelet segítségével számolja ki a program, ami a kizáró vagy művelete. Az int értékének 1-et adunk, és addig shifteljük balra, ameddig lehet, vagyis amíg az int értéke egyenlő nem lesz nullával.

Közben egy másik változóval számoljuk, hogy hányszor shiftelt balra az int, ezzel meghatározva a szóhosszt. Az én esetemben az eredmény 32 lett, ami a virtuális gép miatt van. Valójában 64 bit-es a processzorom ami 8 bájtnak felel meg.

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/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyNeveldaProgramozod/Turing/pagerank.c

Én a Bevezetés a Programozásba nevű tárgyon már átnézett PageRank programot vettem alapnak, és azt alakítottam át C++-ból C-re.

2.7. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

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

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

A megoldás Bátfai Norbert tulajdona.

A Monty Hall problémát még középiskolában ismertem meg, sok különböző változata van. Az általam ismert történetben Monty Hall egy műsorvezető volt, akinél a nyertes játékosok választhattak három darab ajtó közül. A háromból két ajtó mögött 1-1 darab kecske, míg a harmadik ajtó mögött egy sportautó volt. A játékos választott egy ajtót, majd Monty Hall, aki tudta, hogy melyik ajtó mögött van az autó, kinyitott egy másik ajtót, ami mögött egy kecske lapult. Ezek után a játékosnak lehetősége volt változtatni a döntésén, vagy maradhatott az eredetileg kiválasztott ajtónál. A kérdés az, hogy mely esetben van több esélye megnyerni az autót? A legtöbb ember azt mondaná, hogy 50-50% esélye van megnyerni az autót, hiszen vagy

az egyik ajtó mögött van az autó, vagy a másik mögött. Ekkor persze hiába magyarázzuk, hogy 1/3 esélye van megnyerni az autót, ha nem vált, és 2/3 ha vált, a legtöbb embert elég nehéz meggyőzni erről. Ekkor kell kicsit átalakitani a kérdést. Ha van 1 millió ajtó, ebből kiválaszt a játékos 1-et, majd kinyitnak 999,998 ajtót, amik mögött kecske van, akkor melyik esetben van több esélye a játékosnak megnyerni az autót? Ilyenkor már a legtöbb ember eggyértelműnek tartja, hogy vált, de van olyan ismerősöm, aki még ekkor is azt mondta, hogy 50-50% esélye van megnyerni az autót, ha vált ha nem. Ez a program ennek a játéknak a nyerési eseteit szimulálja. Tízmillió esetből hányszor nyer az, aki mindig vált, és az aki egyáltalán nem vált.

2.8. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

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

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

A megoldás Bátfai Norbert tulajdona.

Mint tudjuk, léteznek a prímszámok. Ezek olyan számok, amik csak 1-el és önmagukkal oszthatóak. Valamint léteznek az ikerprímek. Ezek pedig olyan prímszámpárok, amiknek a különbsége pontosan 2. Ha minden ikerprím reciprokának az összegének vesszük a sorozatát, akkor ez a sorozat egy számhoz konvergál. Ez a szám a Brun-konstans. Nem tudjuk azt, hogy az ikerprímek száma véges vagy végtelen e, de ez nem okoz gondot, hiszen elvileg ha végtelen se lépi túl az összegük a Brun-konstanst. Na most be kell vallanom, hogy számtalan olyan ember létezik a földön, aki nálam jobban ért a matematikához. Viszont nekem erről egy elég érdekes dolog jutott eszembe, ami nem más, mint Zeno paradoxona. E szerint x útat teszünk meg, hogy elérjük a célunkat. Ezek alapján megteszünk 1/2x utat + 1/4x utat + 1/8x utat + 1/16x utat... Ha ezekből képzünk egy sorozatot, az a sorozat 1-hez fog konvergálni, Éppen ezért soha nem érünk el oda, ahova megyünk. Maga a tétel matematikailag helyes, azonban a való életben tudjuk, hogy ez nem így működik.

2.9. Vörös Pipacs Pokol/csiga folytonos mozgási parancsokkal

Megoldás videó: https://youtu.be/uA6RHzXH840

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

Ebben a feladatban a karaktert folytonos mozgásra bírtuk, különböző módokon. Ez egy nagyon kezdetleges lépés volt, de mint már azt tudjuk, kis lépés egy karakternek, óriási a hackernek :).

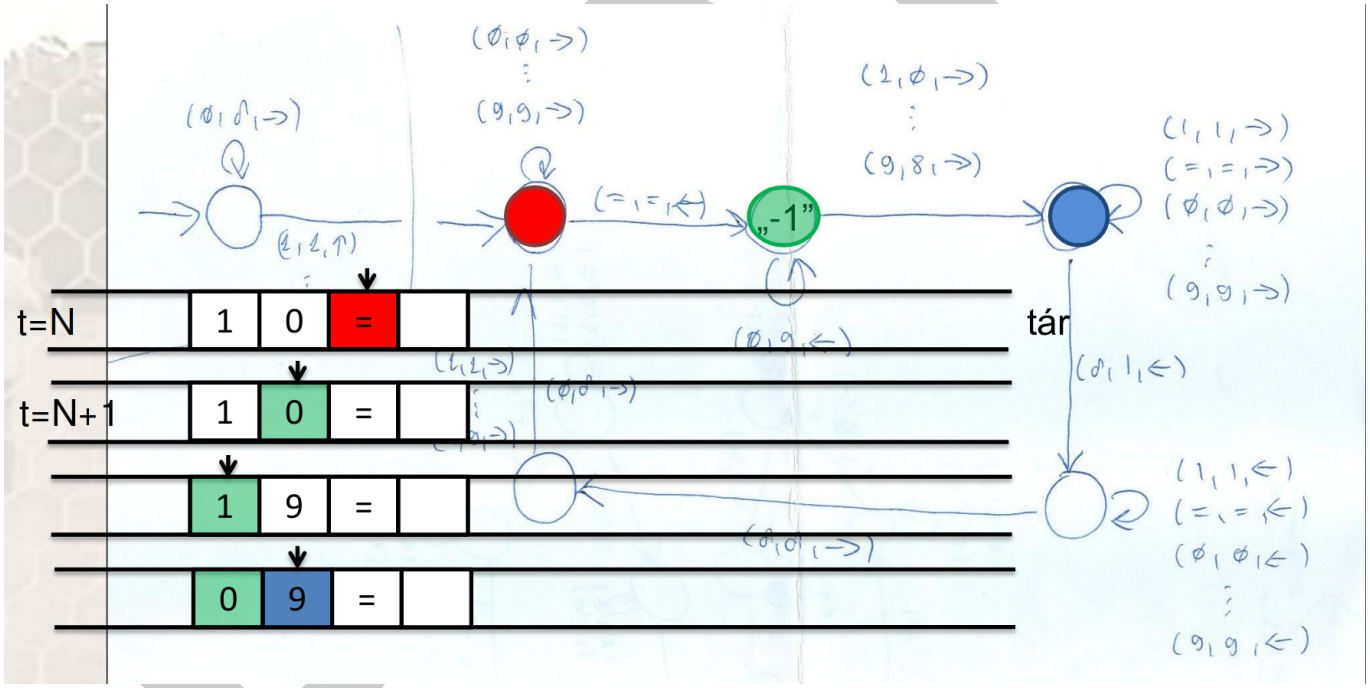
3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfjával megadva írd meg ezt a gépet!

Megoldás videó:



Megoldás forrása: A kép nem az én tulajdonom, hanem a Magas szintű programozási nyelvek 1 nevű tárgyon kivetített előadás fóliáiról másoltam.

Az unáris számrendszer csupa 1-esekből vagy vonalakból áll. Ilyen például, amikor a kezünkön számolunk, vagy amikor az iskolában a gyerekek a számolópálcikákkal rakják ki a dolgokat. Pontosan annyi 1-es vagy pálcika, vagy akármilyen jel kell, amennyi maga a szám értéke. Például ha a 25-ös számot szeretnénk felírni unárisban, akkor 25 darab 1-est kellene leírni egymás után, ezért ebben a számrendszerben csak a természetes számokat tudjuk ábrázolni.

Egy ilyen decinálisból unárisba átváltó Turing gépet mutat a fenti ábra is. Ez a gép a kapott szám utolsó számjegyéből von le egyeseket. Ha a számjegy 0 akkor kilenc darabot von le, ha öt akkor négyet. Ezzel

együtt a levont egyeseket a tárba helyezi. Ezt minden számjeggyel megismétli.

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

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

Megoldás videó:

Megoldás forrása:

Az $a^nb^nc^n$ nyelv nem környezetfüggetlen. Na de először értsük meg, hogy mit is jelent ez a nyelv.

Az $a^nb^nc^n$ annyit jelent, hogy n darab a, majd n darab b, majd végül n darab c áll egymás után. Ezek a terminális szimbólumok. A szabály alapján a környezetfüggő nyelveknél bal oldalon csak egy önmagában álló nem-terminális szimbólum állhat. Azonban nem létezik olyan képzési szabály ami alapján ez a szabály teljesíthető, ezért ez a nyelv nem környezetfüggetlen.

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: https://youtu.be/Dkp4gI9JZNs

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI
Chomsky/nyelvek.c

A BNF (Backus-Naur-forma) használatával környezetfüggetlen nyelveket lehet leirni. Nagyon sok programozási nyelv szintaxisa is BNF-ben vannak leírva.

A programozási nyelveknek is van nyelvtana, illetve nyelvtani szabályaik. Az egyik ilyen szabály C89-ben az, hogy a for ciklus fejrészében nem lehetett változót deklarálni, éppen ezért ha a a következőképpen szeretnénk lefordítani a fenti programot:

gcc -o nyelv nyelv.c -std=c89

Akkor a következő hibaüzenetet kapjuk:

nyelv.c:3:2: error: ‘for’ loop initial declarations are only allowed in C99 or C11 mode

Ez pontosan leírja nekünk, hogy a for ciklusban deklarálni csak c99 vagy c11 módban lehet.

Éppen ezért ha -std=c89-et kihagyjuk és nem írunk oda semmit, vagy -std=c99-et írunk, akkor a program gond nélkül lefordul.

3.4. Saját lexikális elemző

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

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_Igyl Chomsky/lex.l

A megoldás forrása Bátfai Norbert tulajdona.

A lexer-rel szövegelemző programokat lehet generálni az általunk megadott szabályok alapján. A program különböző részeit % jelekkel kell elválasztani egymástól. Itt a numbers változóban fogjuk számolni a valós számok darabszámát. Majd megmondjuk, hogy a digit egy 0 és 9 között lévő számot jelöl. Ezek után jön az a kódrészlet, ami megmondja a lexernek, hogy a valós számokat számolja meg. Végül pedig kiíratjuk a valós számok darabszámát. A futtatáshoz először is telepítenünk kell a lex-et majd a forrásban található programot kell megirni.

Majd azt a következőképp kell lefordítanunk:

lex -o lex.c lex.l

gcc -o lex lex.c -lfl

Ezzel megkapjuk a https://github.com/ Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_Igy Chomsky/lex.c oldalon található programot, ami a feladat megoldása.

3.5. Leetspeak

Lexelj össze egy l33t ciphert!

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

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_Igyl Chomsky/lex.l

Megoldás forrása: A megoldás forrása Bátfai Norbert tulajdona.

A leet (1337, saját formában írva) egy, az internettel együtt elterjedt szleng nyelv, amiben a betűket különböző számokként, és egyéb ASCII karakterekként, a számokat pedig különböző betűkként ábrázoljuk.

Itt is érvényes az a szabály, hogy a program egyes részeit % jelekkel kell elválasztani egymástól. Itt a legelső részben a cipher struktúrában meg vannak adva a karakterek leet formái

A második részben történik az igazi munka, először a szöveget kisbetűssé alakítja a program, majd pedig végigmegy a szövegen és minden karaktert a neki megfelelő leet formájú karakterré alakítja át.

A harmadik és egyben utolsó részben található a main() amiben a lex meghívása történik.

3.6. A források olvasása

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

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

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

Bugok

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

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

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

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

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

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

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

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

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

Megoldás forrása:

Megoldás videó:

1: Ha kapunk egy INTERACT szignált, akkor a jelkezelő fügvénnyel eldöntjük, hogy mihez kezdjünk azzal a szignállal, mit reagáljon rá a program.

2: Egy for ciklus ami nullától négyig megy és a ciklus törzsében lévő művelet elvégzése előtt nő az értéke eggyel.

3: Szintén egy for ciklus, ami szintén nullától négyig megy, viszont itt már a ciklus törzsében lévő műveletek elvégzése után növekszik az értéke.

4: Egy for ciklus, ami berakja a tomb[i]-edik helyére az i értékénél eggyel nagyobb értéket, és közben i értékét is növeli.

5: Egy for ciklus, ami addig megy, amíg i kisseb mint n, illetve amíg a d és s pointerek értékei megyegyeznek.

6: Kiírunk két, az f nevű függvény által generált számot. Az egyik szám az a majd a eggyel megnövelt értékének a feldolgozásából jön létre, míg a másik szám a+1 és a feldolgozásából. Fontos a sorrend.

7: Szintén két számot írunk ki, az egyik szám az f nevű függvény által feldolgozott a nevű számból előállt érték, a másik pedig a értéke.

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

Első értelmezése: Minden számra igaz, hogy létezik tőle nagyobb y prímszám.

Második értelmezése: Minden számra igaz, hogy létezik egy olyan tőle nagyobb y prímszám, hogy y+2 is prím.

Harmadik értelmezése: Létezik olyan szám, amitől minden prímszám kissebb.

Negyedik értelmezése: Létezik olyan szám, amitől egyik kissebb szám se prím.

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; //létrehoz egy egész típusú változót
  ```
- ```
  int *b = &a; //egy pointer, ami a memóriacímére hivatkozik
  ```
- ```
  int &r = a; //egy referencia a-ra
  ```
- ```
  int c[5]; //5 elemű tömb aminek c a neve
  ```
- ```
  int (&tr)[5] = c; //egy tr nevű referencia c-re
  ```
- ```
  int *d[5]; //egy 5 elemű pointerekből álló tömb
  ```
- ```
  int *h (); //Egy egészre mutató mutatót visszaadó függvény
  ```
- ```
  int *(*l) (); //Egy egészre mutató mutatóra mutató mutatót visszaadó ↩
     függvény
  ```
- ```
  int (*v (int c)) (int a, int b) //Függvénymutató, ami egy egészet ↩
     visszaadó függvényre mutató mutatóval visszatérő függvény
  ```
- ```
  int (*(*z) (int)) (int, int); //Függvénymutató, ami egy egészet visszaadó ↩
      függvényre mutató mutatót visszaadó függvényre mutat
  ```

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI Chomsky/deklaracio.cpp

Egész:

```
int a;
```

Egészre mutató mutató:

```
int *b = &a;
```

Egész referenciája:

```
int &r = a;
```

Itt fontos megjegyezni, hogy c-ben nincs referencia, ezért ezt a kódcsipetet érdemes g++-al forditani gcc helyett.

Egészek tömbje:

```
int c[5];
```

Egészek tömbjének referenciája (nem az első elemé):

```
int (&tr)[5] = c;
```

Egészre mutató mutatók tömbje:

```
int *d[5];
```

Egészre mutató mutatót visszaadó függvény:

```
int *h ();
```

Egészre mutató mutatót visszaadó függvényre mutató mutató:

```
int *(*l) ();
```

Egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény:

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

Függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre:

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

3.9. Vörös Pipacs Pokol/csiga diszkrét mozgási parancsokkal

Megoldás videó: https://youtu.be/Fc33ByQ6mh8

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

Ebben a feladatban ugyanazt csináltuk mint az előző hetiben, viszont itt diszkrét mozgási parancsokat kell megadnunk a karakternek.

4. fejezet

Helló, Caesar!

4.1. double** háromszögmátrix

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

Megoldás videó:https://youtu.be/1_EcjAmCTHA

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI Caesar/tm.c

A megoldás forrása Bátfai Norbert tulajdona.

Először is egy alapfogalom: Az alsó háromszög mátrixnak ugyanannyi sora van, mint oszlopa. Ezen kívül még egy nagyon fontos tényezője az is, hogy a főátlója felett csak 0 szerepel.

Az ilyen mátrixokat, ha tömbökben tároljuk, akkor nincs értelme a nullákat is tárolni a többi, számunkra fontos elemmel együtt, ezért ezeket nem is tároljuk. Amikor egy ilyen tömböt vissza szeretnénk alakítani az eredeti alakjára, akkor sorfolytonosan írjuk fel az elemeit. Ez annyit jelent, hogy a mátrix első sorába az első elemet írjuk fel, a második sorába a második és harmadik elemet és így tovább minden sorban eggyel több elemet írunk fel mint az előző sorban.

Ebben a programban egy ilyen alsó háromszög mátrixot hozunk létre egy

```
double **
```

segítségével. Ez egy pointerre mutató pointer, ami tökéletes a többdimenziós tömbök használatához.

Ezek után a következő kis programrészlet:

```
if ((tm = (double **) malloc (nr * sizeof (double *))) == NULL)
    {
        return -1;
    }
    printf("%p\n", tm);

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

```
        {
            return -1;
        }
}
```

Ellenőrzi, hogy történt e valamilyen memóriahiba, (pl. nincs tele a memória?) és ha történt, akkor -1-el tér vissza.

Ellenkező esetben a program a

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

képletet használva feltölti a tömböt. Ezután két egymásba ágyazott for ciklus segítségével kiírja azt.

Ezek után módosítunk a tömb egyes elemein, majd megint kiírjuk őket.

Legvégül, pedig a

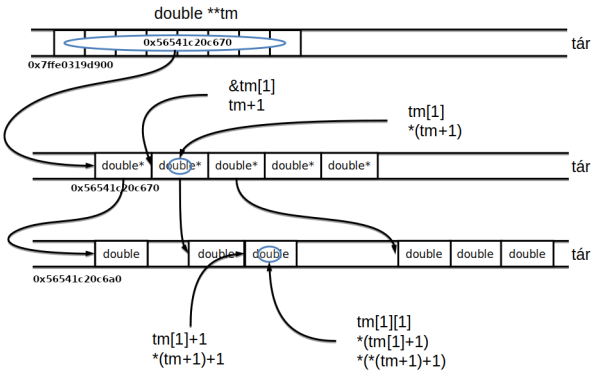
```
for (int i = 0; i < nr; ++i)
        free (tm[i]);
free (tm);
```

függvény használatával felszabadítjuk a tömbnek lefoglalt helyet.

A program futtatásnál a következő memóriacímeket írta ki:

```
  ./tm
0x7ffe0319d900
0x56541c20c670
0x56541c20c6a0
```

Aminek a jelentése:



A képnek az alapját Bátfai Norbert biztosította, azt módosítottam.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó:https://youtu.be/osQ0iZu4Swg

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI Caesar/exor.c

A megoldás forrása Bátfai Norbert tulajdona.

Ez a fajta titkosítás a kizáró vagy műveleten alapul. A megadott kulcs, és a forrásfájl karaktereit kizáró vaggyal titkosítva egy szöveget úgy tudunk titkosítani, hogy egy olvashatatlan karakterhalmazt kapunk végeredményül. Viszont az a személy, aki ismeri a kulcsot ugyanolyan egyszerűen vissza is tudja alakítani a szöveget az eredeti alakjára úgy, hogy mégegyszer lefuttatja a programot, de a titkosított forrást adja meg titkosítandóként, ezzel visszakapva az eredeti szöveget. Így más nem tudja elolvasni a titkainkat, csak az, aki ismeri hozzá a kulcsot.

Először is a

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

használatával megadjuk a maximális kulcs és buffer méretet. a main osztály első argumentuma a kulcs lesz, mig a második az maga a szöveg, amit titkosítani szeretnénk.

A következő ciklusok használatával:

```
while ((olvasott_bajtok = read (0, (void *) buffer, BUFFER_MERET)))
    {
      for (int i = 0; i < olvasott_bajtok; ++i)
  {
    buffer[i] = buffer[i] ^ kulcs[kulcs_index];
    kulcs_index = (kulcs_index + 1) % kulcs_meret;
  }
      write (1, buffer, olvasott_bajtok);
}
```

program végigmegy a bemeneti adatok (titkosítandó fájl) karakterein, mindegyiket titkosítja a kulcs használatával és kiírja a végeredményt.

A program használata: ./exor kulcs <titkosítandó fájl> titkosított fájl

Erre egy példa: **./exor 12345678 <lista> titkoslista**

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:https://youtu.be/P7PbIIRZiSA

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI Caesar/exor.java

A megoldás forrása Bátfai Norbert tulajdona.

Itt az előző feladatban megírt EXOR titkosítót írjuk át java nyelvre. Ehhez importálnunk kell az input/output streamet. Ez ahhoz kell, hogy olvasni tudjuk a bemeneti fájlt, illetve, hogy írni tudjuk a kimeneti fájlt.

A main-ben megpróbáljuk a try-al beolvasni az args tömbbe azt a fájlt, amit titkosítani szeretnénk és ha ez nem sikerült, akkor "elkapjuk" a hibát a catch szerkezettel, és kiíratjuk, hogy mi a hiba:

```
[
  public static void main(String[] args) {

        try {

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

        } catch(java.io.IOException e) {

            e.printStackTrace();
        }
}
```

Ha viszont sikerült beolvasni a fájlt, akkor az ExorTitkosító nevű függvényt meghívva előállítjuk a titkosított szöveget. A System.in illetve System.out a bemenő és a kimenő fájlra utalnak.

Először a függvény átadja a program a kulcs nevű tömbjének a bemenő szöveget, és létrehoz egy buffer nevű tömböt is 256-os mérettel. Erre az EXOR művelethez lesz szükség.

Végül a program egy while-ba épített for ciklus segítségével végigzongorázza a szöveget, minden egyes karakternek meghatározza a titkosított verzióját, és kiírja azt a kimeneti fájlba.

4.4. C EXOR törő

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

Megoldás videó:https://youtu.be/oii26G1ubGk

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI Caesar/tores.c

A megoldás forrása Bátfai Norbert tulajdona.

Elfelejtetted egy EXOR-ral titkosított fájlod kulcsát? Szeretnél kutakodni mások fájljai között, de azok titkosítva vannak?

Ne is kínlódj tovább. Az EXOR törőt neked találták ki! A program sikerességéhez mindössze annyit kell tudnod , hogy hány karakterből áll a kulcs, és máris használhatod ezt a fantasztikus programot!

A működése nagyon egyszerű. Mivel nem ismerjük a kulcsot, ezért a program az összes lehetséges kombinációt végigpróbálja. A következőkben bemutatott példában a kulcs 8 darab karakterből áll.

Legelőször a program a következő while ciklus

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

használatával beolvassa a feltörni kívánt fájlt, majd a maradék helyet a bufferben, egy for ciklust használva,

```
 for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)
titkos[p - titkos + i] = '\0';
```

feltölti 0 értékekkel.

Ezek után egy csomó (ami jelen esetben 8) for ciklussal

```
 #pragma omp parallel for private(kulcs)
    for (int ii = '0'; ii <= '9'; ++ii)
        for (int ji = '0'; ji <= '9'; ++ji)
            for (int ki = '0'; ki <= '9'; ++ki)
                for (int li = '0'; li <= '9'; ++li)
                    for (int mi = '0'; mi <= '9'; ++mi)
                        for (int ni = '0'; ni <= '9'; ++ni)
                            for (int oi = '0'; oi <= '9'; ++oi)
                                for (int pi = '0'; pi <= '9'; ++pi)
                                {
                                    kulcs[0] = ii;
                                    kulcs[1] = ji;
                                    kulcs[2] = ki;
                                    kulcs[3] = li;
                                    kulcs[4] = mi;
                                    kulcs[5] = ni;
                                    kulcs[6] = oi;
                                    kulcs[7] = pi;
                                    exor_tores (kulcs, KULCS_MERET, titkos, ↩
                                        p - titkos);
}
```

megpróbálja a program előállítani az eredeti szöveget. Azonban több kombináció is ad eredményt, ezért nekünk kell kitalálni, hogy a kapott eredmények közül melyik a helyes. Ezek a for ciklusok az összes magot dolgoztatni fogják, ezzel jelentősen lecsökkentve a töréshez szükséges időt.

Ha a kulcs nem 8 karakterből áll, akkor sem kell aggódnunk! Csupán néhány (pontosan 3) szekcióban kell módositani a program kódját. Ezek a következők:

Először is a program fejében a

```
[#define KULCS_MERET 8
```

sorban a 8-at át kell írni arra a számra, amennyi karakterből áll a kulcs.

Majd a 70. és 71. sorokban lévő

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

utasításokban annyi **%c** és **kulcs[n]** legyen, amennyi karakterből áll a kulcs.

Végül pedig az előzőekben már látott for ciklus halmon kell módosítanunk úgy, hogy pontosan annyi **for** ciklus, és pontosan annyi **kulcs[n] = xi;** legyen a programban, amennyi karakterből áll a kulcs.

Most ezeknek az ismeretében, a programot a következőképpen kell fordítani: **gcc tores.c -fopenmp -o tores -std=c99**

És futtatni: **./tores <titkosfájl**

4.5. Neurális OR, AND és EXOR kapu

R

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

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

A megoldás forrása Bátfai Norbert tulajdona.

A neurális hálózatokban például a machine learning esetében, a neuronok egy gráfban elhelyezkedve egymással kommunikálnak ún. "Activation function", magyarul Aktivációs függvény segítségével.

Léteznek bemeneti, kimeneti, és rejtett neuronok is.

A bemeneti neuronok kapják meg a bemenetet. Itt több különböző fajta neuront is meg lehet különböztetni. Vannak egybemenetű és több bemenetű neuronok is. Ezeknek a neuronoknak nincs különösebb feldolgozó feladatuk, továbbítják a bemenetet a többi neuronnak.

A kimeneti neuronok amik a környezetnek adják tovább a kapott információt.

A rejtett neuronoknak pedig a bemenete és a kimenete is csakis más neuronokhoz kapcsolódik.

Ezek alapján egy neurális hálónak legalább két rétegből kell állnia. Egy bemenetiből, és egy kimenetiből. Felső határ, azaz hogy a bemeneti és a kimeneti neuronok között hány darab további réteg helyezkedik el, elviekben nincs.

Először minden neuron megkapja a saját bemeneteit, és minden neuron ebből a bemenetből előállít egy úgynevezett súlyozott összeget, és ezt az értéket vezeti végig az aktivációs függvényen. Egy példa lehet az, hogy ha a súlyozott összeg pozitív lesz, akkor az érték 1, míg ha a súlyozott összeg negatív, akkor az érték -1 lesz.

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó:

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

A megoldás forrása Bátfai Norbert tulajdona.

Perceptronról a Mesterséges Intelligenciák, és a neurális hálók témakörében lehet szó. Ellenőrzi a bemenetet, és egy feltétel alapján eldönti, hogy mi legyen a kimenet, Egy példa:

Van három bemeneti adatunk amikbe pozitív egész számokat várunk. Ha a három bemeneti számból kettő kisebb mint 0, akkor a kimeneti adat -1 lesz, ha viszont a háromból legalább kettő pozitív szám, akkor a számok összege lesz a kimeneti adat.

Ekkor kimondhatjuk, hogy 1 a hibahatár, mert ekkor még megkapjuk az általunk kért dolgot, viszont ha már kettőt hibázunk akkor már -1 lesz a válasz.

Ezt a hibahatárt szokták finomítani. Nagyon magas hibahatárnál kezdenek, és egyre kisebbé teszik egészen addig, amíg elfogadható a hibák mennyisége.

Persze a mi három bemeneti adatos példánknál nem sokat lehet finomítani, de ha több millió bemeneti adatról beszélünk, ott ez egy elég fontos dolog.

4.7. Vörös Pipacs Pokol/írd ki, mit lát Steve

Megoldás videó: https://youtu.be/-GX8dzGqTdM

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

Ebben a feladatban a kódhoz hozzáadunk egy olyan részt, ami kiírja, hogy mi található a karakter 3x3-as környezetében, merre néz a karakter, és azt is, hogy éppen mire néz. A cél az, hogy a karakter kiírja a képernyőre, hogy éppen egy vörös virágot lát.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Megoldás videó:https: //www.youtube.com/watch?v=hHKhGdCN2Rolink

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI
Mandelbrot/mandelbrot.cpp

A megoldás forrása Bátfai Norbert tulajdona.

Mielőtt bármihez hozzákezdenénk egy nagyon fontos információ. Ahhoz, hogy lefordúljon a programunk, szükséges a png++. Ezt a legegyszerűbben a **sudo apt install png++** paranccsal lehet megtenni. Most hogy ezt letudtuk, jöhet pár alapvető információ.

A Mandelbrot halmaz lényege az, hogy komplex számokkal, és egy egyenlettel dolgozik. Azok a számok amelyek kielégítik ezt az egyenletet egy nagyon szép képet alkotnak, ha levetítjük őket egy kétdimenziós síkra.

A program legelején includeoljuk a png++-t, hiszen nagyrészt ezt fogja használni a program.

```
#include <png++-0.2.9/png.hpp>
```

Ezek után létrehozunk végleges értékeket N-nek és M-nek, valamint megadjuk X és Y lehetséges minimum és maximum értékét is.

```
#define N 500
#define M 500
#define MAXX 0.7
#define MINX -2.0
#define MAXY 1.35
#define MINY -1.35
```

két egymásba ágyazott for ciklus használatával megadjuk a C, a Z, és a Zuj nevű Komplex számok valós és imaginárius értékét. Ezek korábban lettek létrehozva a mainen belül, és a Komplex nevű struktúrához tartoznak.

```
  struct Komplex
{
```

```
    double re, im;
};
```

```
  struct Komplex C, Z, Zuj;
```

Végül pedig a `GeneratePNG(tomb)` nevű függvény használatával a program legenerálja a PNG fájlt pixelről pixelre.

A programot a következőképpen tudjuk fordítani: **g++ mandelbrot.cpp -lpng16 -o mandelbrot** Futtatni pedig a szokásos módon a **./mandelbrot** parancsal tudjuk.

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

Megoldás videó:

Megoldás forrása:https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyN Mandelbrot/mandelbrotkomplex.cpp

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az UDPROG repóban.

Ebben a feladatban a végeredmény ugyan az kellene hogy legyen, mint az előző feladatban. Illetve azóta még a mandelbrot halmaz lényege sem változott.

A png++ ebben az esetben is kelleni fog, így ha nincs leszedve, akkor pillants az előző feladat magyarázatára, ahol megtalálod a szükséges dolgokat ahhoz, hogy le tudjon fordulni a program.

Ebben az esetben az `std::complex` osztályt fogjuk használni a program megvalósításához. Ez az osztály, ahogy a neve is utal rá, a komplex számok kezelése miatt jött létre.

A program által használt függvényei a következők:

A `real(C)` a komplex szám valós részét határozza meg.

A `imag(C)` a komplex szám képzetes részét határozza meg.

Legelőször a program

```
#include <png++-0.2.9/png.hpp>
#include <complex>
```

beincludeolja a png++-t és a komplex osztályt

Ezek után az előző feladathoz hasonlóan itt is megadjuk a végleges értékeket az N, M valamit X és Y maximum és minimum értékeinek.

Legnagyobb részben ennek a feladatnak a megoldása megegyezik az előző feladat megoldásával, ezért azt nem írnám le újra, inkább arra koncentrálnék, hogy miben más ez a forrás mint az előző.

Az érdemi különbség a két forrás között az az, hogy itt az std::complex osztályt használva, már nem kell létrehoznunk egy saját struktúrát a komplex számoknak.

Ehelyett szimplán létrehozzuk a double típusú komplex számokat a következőképpen:

```
std::complex<double> C, Z, Zuj;
```

Illetve a for cikluson belül sem a struktúrán belüli elemek imaginárius és valós részére hivatkozunk, hanem a `real()` és `imag()` nevű függvényeket meghívva mondjuk meg a komplex szám részeinek értékét.

```
  real(C) = MINX + j * dx;
imag(C) = MAXY - i * dy;
```

A programot fordítani és futtatni ugyan úgy kell, mint az előző feladatot.

5.3. Biomorfok

Megoldás videó: https://www.youtube.com/watch?v=nUAG7xP54AQ

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

A megoldás forrása Bátfai Norbert tulajdona.

A két előző feladathoz hasonlóan itt is szükségünk van a png++ ra, ezért ha még nem szedted le akkor pillants rá a az első feladat magyarázatára, ahol részletesen le vannak írva az ehez szükséges parancsok.

Ez egy olyan mandelbrot program, ahol maga a user adja meg a határokat. Előnye hogy az eredetihez képest teljesen más képeket kapunk, hátránya viszon hogy ha a user nem tudja, hogy mit csinál akkor az egész kép egy nagy fekete semmi lesz.

Először is

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

includeoljuk az iostreamet a png++t és a komplex osztályt.

A main argumentumai a bemeneti adatok, amikből előállítjuk magát a képet.

Ellenőrzi a program, hogy megfelelő mennyiségű bemeneti értéket adott e meg a felhasználó, és ha nem, akkor felvilágosítja, hogy hogyan kell használni a programot.

```
  if ( argc == 12 )
    {
        szelesseg = atoi ( argv[2] );
        magassag =  atoi ( argv[3] );
        iteraciosHatar =  atoi ( argv[4] );
        xmin = atof ( argv[5] );
        xmax = atof ( argv[6] );
        ymin = atof ( argv[7] );
        ymax = atof ( argv[8] );
        reC = atof ( argv[9] );
        imC = atof ( argv[10] );
        R = atof ( argv[11] );
    }
    else
    {
        std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c ↩
            d reC imC R" << std::endl;
```

```
        return -1;
    }
```

Ha viszont megfelelő mennyiségű argumentumot adott meg a felhasználó, akkor létrehozza a képet aminek a szélessége és a magassága a felhasználó által megadot szélesség és magasság lesz.

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

Ezek után a program két egymásba ágyazott for ciklus segítségével kiszámolja, létrehozza a képet és el is menti a felhasználó által megadott néven.

A fordítása az előző két programhoz hasonlóan működik, a futtatása azonban már igy néz ki:

./3.1.3 fajlnev szelesseg magassag n a b c d reC imC R Erre egy példa:

./3.1.3 biomorf.png 800 800 10 -2 2 -2 2 .285 0 10

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/CUDA/mandelpngc_60x60_100.

A megoldás forrása Bátfai Norbert tulajdona.

Először is közérdekű közlemény, hogy ennek a programnak a sikeres fordításához szükségünk lesz egy CUDA magokat használó NVIDIA kártyára, illetve az nvidia-cuda-toolkit re amit a következő parancsal tudunk feltenni:

sudo apt install nvidia-cuda-toolkit

Ez a program ugyanúgy a mandelbrot halmazt rajzolja ki, mint az előzőek, azonban itt egy nagyon fontos különbség az, hogy míg az előző feladatoknál a képet a CPU számolta ki és készítette el, addig itt, az NVIDIA kártyák CUDA magjait használjuk a kép kiszámításához.

Ez azért fontos, mert az előző feladatoknál egyetlen egy mag dolgozott és számolt ki mindent, addig itt az én GTX 1660 SUPER videókártyám esetében 1408 darab cuda mag számolná és rajzolná ki a képet. Azért írom feltételes módban, mert a virtuális gép egyik hátránya, hogy nem tudja a gép minden részét kihasználni.

Ez nyílvánvalóan egy sokszor gyorsabb futási időt eredményez. Az én esetemben például amikor CPU-val futtattam volna a programot akkor egy kb 20 másodperces futási idő jött volna ki.

Azonban ha már a fentebb említett GTX 1660 SUPER kártyát használva futtatom a programot, akkor már egy kicsit hamarabb lefut a program.

A programot a gcc helyett az nvcc nevű paranccsal kell forditani. Futtatni pedig a szokásos módon.

5.5. Mandelbrot nagyító és utazó C++ nyelven

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

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

Megoldás forrása: https://sourceforge.net/p/udprog/code/ci/master/tree/source/binom/Batfai-Barki/frak/

Visszatérünk a Mandebrot halmazokhoz, de most nem csak egy képet szeretnénk végeredményként kapni, hanem egy olyan ablakot, ahol képesek vagyunk ránagyítani a kapott halmazunkra. Ez lehetséges egy képfájl esetén is, de ott egy idő után a képet nem lehet lesz értelmezni, olyan szinten pixeles lesz.

Amit mi itt használni fogunk ennél egy kicsit bonyolultabb, emiatt talán ez a fejezet legnehezebb feladata. A megoldás erre a problémára az, hogy az egérrel kijelölt részre az ablak nem ránagyít, hanem újragenerálja azt a részt. Így elkerülhetőhető az a probléma, hogy nagyon pixelessé válik a kép, ugyanis a pixelszám azonos lesz.

Az első probléma még a kódolás előtt megjelenik, hogyan készitsük el ezt az ablakot. Egy nagyszerű megoldás a Qt eszköztár használata, amelyben elkészithető egy gui(grafikus interfész). Ennek az interfésznek a futtatásához 5 kódrészre van szükségünk, ezek szerepei a következők:

frakablak.h

Ebben a kódrészben egy osztályt hozunk létre, aminek van védett, privát és publikus tagja is. A publikus részben a tartományokat/határokat szabjuk meg, amik között mozoghatunk. A védett részben egy függvényt deklarálunk, amely figyelemmel követi, mit csinál a felhasználó. Ennek fontos szerepe lesz, hogy a program tudja azt, hogy mikor a halmazra ránagyítania. Végül a privát tagban a nagyitandó terület van meghatározva.

frakszal.h

Ismét egy osztályt láthatunk és még pár változót. Ezek segitenek majd a programnak a számolásban és rajzolásban.

frakszal.cpp

Ebben a kódrészben történik a legfontosabb dolog: a Madelbrot halmaz elemeinek kiszámolása, ehhez hasonló programot írtunk az első feladatokban is.

main.cpp

Itt kerül meghívásra a Qt konstruktora, és itt tesszük a kódhoz a Qt könyvtárat.

frak.pro

A frak.pro lesz a "főnök", alatta fognak futni az előbb bemutatott kódok, az ő feladata lesz a teljes folyamat kezelése.

5.6. Mandelbrot nagyító és utazó Java nyelven

Megoldás Forrása: https://github.com/Viktorpalffy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyN
Mandelbrot/MandelJava

A megoldás forrása Bátfai Norbert tulajdona.

A feladat az ezt megelőző feladat átírása Java nyelvre. A GUI megírásához szükség van egy keretrendszerre, ami jelen esetben az Abstract Window Toolkit lesz.

Először nézzük a *Mandelbrothalmaz.java* fájlt.

A main-ben a MandelbrotHalmaz() meghívásával létrehozunk egy új halmazt a megadott paraméterekkel. Ezek a paraméterek a tartományok koordinátái, a halmazt tartalmazó tömb szélessége, és a számítás pontossága.

Utána a felhasználó tevékenységeit figyeli a program, és megfelelően reagál rájuk, valamint a GUI ablak tulajdonságait adja meg, illetve kirajzolja magának a halmaznak a képét.

A következő fájlunk a *MandelbrotHalmazNagyító.java*

A nevéből adódóan ez végzi a halmazon a nagyítást, illetve magának a halmaznak a kirajzolását is. Maga a MandelborHalmazNagyító osztály figyeli a felhasználó egér tevékenységeit, azzal kapcsolatban, hogy hol szeretné nagyítani a képet, illetve kirajzolja az új, nagyított képet. Ezen kívül ez végzi az elmentendő képek készítését, és elmentését is.

Végül pedig a *MandelbrotIterációk.java* fájl szerepe.

Ez a programrészlet a nagyított mandelbrot halmazok pontjait tartja nyílván. Ez egy számításra létrehozott osztály, ami a kiválasztott ponthoz tartó utat mutatja meg.

5.7. Vörös Pipacs Pokol/fel a láváig és vissza

Megoldás videó: https://youtu.be/I6n8acZoyoo

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

Ebben a feladatban az a lényeg, hogy a karakter fusson fel az aréna aljától a tetejéig, majd ha a 3x3-as látóterében lávát érzékel, akkor a lávával együtt szépen sétáljon vissza le az aréna aljáig.

6. fejezet

Helló, Welch!

6.1. Első osztályom

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

Megoldás videó:

C++ forrás: https://sourceforge.net/p/udprog/code/ci/master/tree/source/labor/polargen/

Java forrás: https://sourceforge.net/p/udprog/code/ci/master/tree/source/kezdo/elsojava/PolarGen.java#l10

Ehhez a programhoz java-ban szükségünk lesz az util.random, az io.* illetve a lang.math java könyvtárakra. Először is a bExist változót hamisra állitjuk a konstruktoron belül, majd pedig inicializálunk egy randomot, és gyakorlatilag ennyi a konstruktor.

Ezek után a PolarGet függvény az, ami az érdemi munkát végzi. Először is ellenőrzi, hogy volt-e már generálás. Ha volt akkor azt adja vissza, ha nem, akkor a matekos algoritmus segítségével legenerálja a két random normált és bExists-et átállítja az ellentetjére.

6.2. LZW

Valósítsd meg C++-ban az LZW algoritmus fa-építését!

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI
Welch/lzw.cpp

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az UDPROG repóban.

Mindkét esetben a bináris fa felépítésének a lépései a következők:

Ha 1-est szeretnénk betenni a fába, akkor először megnézzük, hogy az aktuális csomópontnak van-e már ilyen eleme. Ha még nincs, akkor egyszerűen betesszük neki az 1-es gyermekének az 1-et. Azonban ha már van ilyen gyermeke, akkor létre kell hozni egy új csomópontot és az ő gyermekének adjuk át az 1-et.

Ez hasonlóan működik akkor is, ha nullást szeretnénk betenni, annyi különbséggel, hogy nem az egyeseket vizsgáljuk, hanem a nullásokat. Ezt a lépést a programban a következő rész oldja meg:

```
void operator<<(char b) {
      if (b == '0') {
        if (!fa->nullasGyermek()) {
          Csomopont *uj = new Csomopont('0');
          fa->ujNullasGyermek(uj);
          fa = &gyoker;
        } else {
          fa = fa->nullasGyermek();
        }
      }
      else {
        if (!fa->egyesGyermek()) {
          Csomopont *uj = new Csomopont('1');
          fa->ujEgyesGyermek(uj);
          fa = &gyoker;
        } else {
          fa = fa->egyesGyermek();
        }
      }
}
```

A megadott fájl tartalma alapján felépíti az LZWBinfa csomópontjait. Jelen esetben ezt a Bináris Fát in order bejárással dolgozzuk fel, ami annyit jelent, hogy először a fa bal oldalát dolgozzuk fel, majd a fának a gyökerét, és legvégül pedig a jobb oldalt. A következő feladatban ezen viszont már változtatunk.

Fordítása a szokásos módon történik a futtatása pedig a következőképpen:

./lzw bemenet -o kimenet

6.3. Fabejárás

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

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI
Welch/fabe.cpp

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az UDPROG repóban.

Az előző feladatban tárgyalt fát In Order módszerrel járta be a program. Ez azt jelenti, hogy először a részfa bal oldalát dolgozzuk fel, majd a részfa gyökerét, és utoljára pedig a részfa jobb oldalát.

Erre ugyanúgy megmaradt a lehetőségünk, csupán a következőképp kell futtatni a programot:

./lzw bemenet -o kimenet i

Ezzel szemben itt két másik fajta bejárási módszerrel dolgozzuk fel a fát. Az egyik a Pre Order bejárási mód, a másik pedig a Post Order.

A Pre Order bejárási módnál először a részfa gyökerét dolgozzuk fel, másodjára a részfa bal oldalát, és utoljára pedig a részfa jobb oldalát. A Pre Order bejárási mód használatához a következőképpen kell futtatni a programot:

./lzw bemenet -o kimenet r

A Post Order bejárási módnál pedig legelőször a részfa bal oldalát dolgozza fel a program, majd a jobb oldalát, és utoljára pedig a részfa gyökerét. A Post Order bejáráshoz a következő parancs használatával kell futtatni a programot:

./lzw bemenet -o kimenet r

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/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI
Welch/tag.cpp

A megoldás forrása nem az én tulajdonom. Az eredeti forrás megtalálható az UDPROG repóban.

Ez a program az eredeti Bevezetés a Programozásba tárgyon már tanult *z3a7.cpp* nevű program szerint működik, hiszem itt a csomópont már kompozícióban van a fával. Az egész az LZWBinfa osztállyal kezdődik, aminek van privát, és publikus része is. A publikus részen belül található a konstruktor, és a destruktor deklarációja. Itt kerülnek vizsgálatra a bemenő elemek, és jönnek létre a nullás illetve egyes elemek is. Túlterhelődik az operátor, és megvizsgálja a program, hogy létezik-e már nullás gyermek. Ha nincs, akkor létrejön. Egyes gyermeknél ugyanez a helyzet.

A kiír függvény pedig kiírja a csomópontokat.

Majd jön az LZWBinfa privát része. Itt található meg a Csomópont osztály amin belül a konstruktor megkapja a gyökeret. Még a Csomópont osztályon belül találhatóak azok a függvények, amivel le tudjuk kérdezni, hogy ki az aktuális csomópont nullás illetve egyes gyermeke, valamint az `ujNullasGyermek()` illetve `ujEgyesGyermek()` függvények, amik létrehozzák az új nullás és egyes gyermekeket

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/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI
Welch/gyoker.cpp

Ehhez a feladathoz az UDPROG repóban megtalálható BinFa programot vettem alapul.

Ebben a megoldásban az előző feladathoz képest kicsit másképp a megoldás. A következő dolgokat kell átírni a már meglévő programban:

Először is a 315. sorban a csomopont után tegyünk egy *-ot ezzel mutatóvá téve a gyökeret. Ha így megpróbáljuk lefordítani a programot akkor nagyon sok szintaktikai hibát fogunk kapni a fordítótól válaszként. Nem kell megijedni. Az a dolgunk, hogy ezeket a hibákat egyesével kijavítsuk. Az első két hiba kijavításához a következő részletet kell átírni.

A 92. és a 93. sorban a

```
 szabadit (gyoker.egyesGyermek ());
 szabadit (gyoker.nullasGyermek ());
```

utasítások helyett a

```
szabadit (gyoker->egyesGyermek ());
 szabadit (gyoker->nullasGyermek ());
```

utasításokat kell használni.

Ez után már kettővel kevesebb hibát kapunk. Az összes többi hibát a referenciák okozzák. Ahoz hogy ezeket a hibákat megoldjuk a következő sorokban kell tevékenykednünk: 92, 132, 147, 170, 210, 336, 344, és 356. Azonban a hibát minden sorban ugyan azzal a módszerrel kell javítani, ami nem más mint hogy a

```
&gyoker
```

helyett azt kell írni hogy

```
gyoker
```

Vagyis kiszedjük a referenciákat, mivel alapból a memóriacímek lesznek átadva.

Ezek után a programunk ugyan lefordul, de amikor megprobáljuk futtatni, akkor szegmentálási hibát kapunk. Ennek a javításához a konstruktort kell átírni a következőképpen:

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

Ezzel foglalunk helyet a memóriában a gyökérnek. Viszont amit lefoglalunk, azt fel is kell szabadítani, éppen ezért a destruktort is módosítani kell a következőképpen:

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

Mostmár fel is szabadul, amit lefoglaltunk.

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/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI
Welch/mozgato.cpp

A megoldás forrásának az alapja megtalálható az UDPROG repóban. Én ezt módosítottam.

Maga az LZWBinFa osztály felépítése úgy néz ki, hogy az osztályon belül léteznek a beágyazott csomópont objektummok amik a fát alkotják. Ezek alapján a fa másolása nem más, mint ezeknek a csomópontoknak a másolása. Ehhez létre kell hoznunk a mozgató illetve mozgató értékadás konstruktorokat.

```
    LZWBinFa (LZWBinFa&& masik){
    gyoker=nullptr;
    *this= std::move(masik);
    }

    LZWBinFa& operator= (LZWBinFa&& masik){
    std::swap(gyoker,masik.gyoker);
    return *this;

}
```

Először a mozgató értékadásról (alsó) szólnék pár szót, ami csupán annyit jelent, hogy ha egyenlőségjel operátort használunk, akkor az `std::swap()` függvénnyel megcserélődik a két gyökér mutatója.

Másodszor pedig a mozgató konstruktor. Itt először is nullptr (nullpointer) értéket adunk abban a binfában lévő gyökérnek, amelyik fába akarjuk mozgatni a ("masik") fát. Majd a "masik" nevű fát átmozgatjuk az `std::move()` függvénnyel, ami annyit jelent, hogy a gyökér mutató mostmár a paraméterként kapott "masik" fára mutat, ami azért történhetett meg, mert az std::move() függvény tulajdonképpen nem is mozgat semmit, hanem a paraméterül kapott értéket jobbérték referenciává alakítja.

6.7. Vörös Pipacs Pokol/5x5x5 ObservationFromGrid

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

Ebben a feladatban a karakter látóterét 3x3x3-asról 5x5x5-ösre bővítettük, ezáltal az eddigi kettő szint helyett már hármat is tudunk egyszerre vizsgálni. Az 5x5x5-ös blokkban lévő elemeket pedig kiírathatjuk vele a konzolra, vagy csak szimplán felhasználhatjuk az adatokat.

7. fejezet

Helló, Conway!

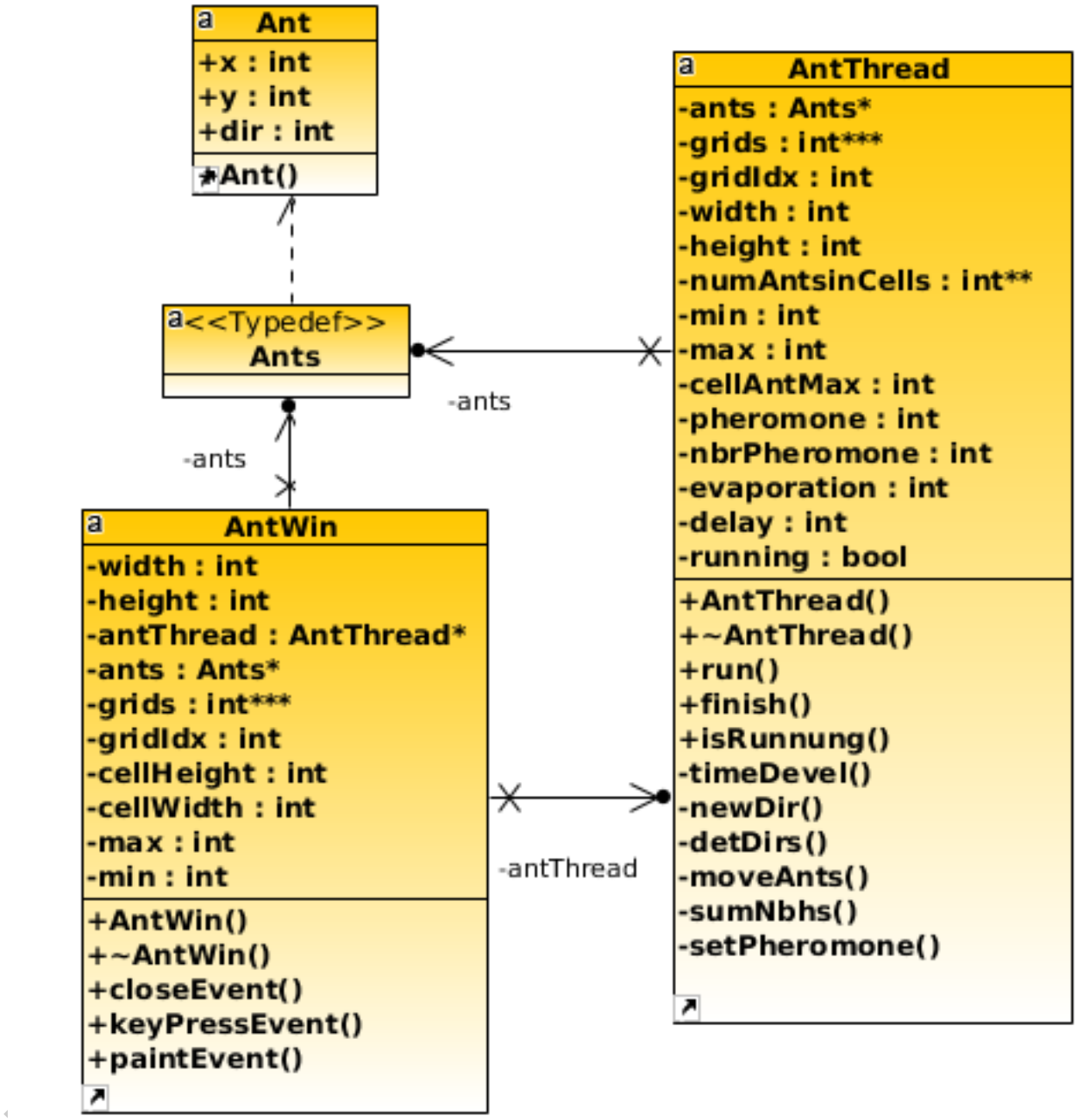
7.1. Hangyaszimulációk

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

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

Megoldás forrása: https://github.com/Viktorpalffy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyN Conway/Ant

Az osztálydiagram:



A megoldás forrása, illetve az UML osztálydiagram Bátfai Norbert tulajdona.

Ebben a feladatban hangyákat kell szimulálnunk. A megoldás azt a biológiából már ismert tényt használja, hogy a hangyák a való életben szagokkal (feromonokkal) kommunikálnak egymással. Ha például egy hangya valamilyen érdekes dolgot talált, akkor ott hagyja a nyomát, illetve megjelöli az útvonalat. Az éppen arra járó többi hangya ezt megérzi, a legfrissebb feromon nyomát követve ők is el fognak jutni a célba. Ezeket felhasználva készítették el ezt a hangyaszimulációt.

Az osztálydiagrammon belül négy egységet találhatunk, ezek a következők: Ant; Ants; AntWin; és AntThread.

Ezek a programunk osztályai, ezeken az egységeken belül vannak megadva az adott osztály változói és függvényei is. Ilyen például az AntWin egységen belül található *width* és *height* változók, amik a képernyő

hosszúságát és szélességét adják meg. Vagy például a `closeEvent()` és a `keyPressEvent()` függvények, amik szintén az AntWin osztály részei. Ezek alapján meghatározhatjuk, hogy az AntWin osztály a szimuláción belül a világot kezeli.

Az AntThread osztály kezeli a hangyákat, mozgásukat, illetve a virtuális feromonok terjedéséért is ez az osztály felelős.

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/Viktorpalffy/bhax/blob/master/bhax/thematic_tutorials/bhax_textbook_IgyI Conway/eletjatek.java

A Megoldás forrása Bátfai Norbert tulajdona.

Arról, hogy mi az az életjáték, illetve, hogy mik a szabályai, a következő feladat leírásában részletesebben kifejtem. Röviden az életjáték szabályai:

Ha egy négyzetnek pontosan három darab élő szomszédja van, akkor abban a négyzetben egy új sejt jön létre.

Ha egy már élő sejtnek pontosan kettő vagy három darab szomszédja van, akkor az a sejt továbbra is életben marad.

Ha viszont egy már meglévő élő sejtnek háromnál több élő szomszédja van (túlnépesedés), vagy kettőnél kevesebb, akkor az a sejt meghal. Ezt szimulálja az életjáték.

Ezek a szabályok az `időFejlődés()` függvényben vannak lefektetve.

```
if(rácsElőtte[i][j] == ÉLŐ) {
                /* Élő élő marad, ha kettő vagy három élő
                 szomszedja van, különben halott lesz. */
                    if(élők==2 || élők==3)
                        rácsUtána[i][j] = ÉLŐ;
                    else
                        rácsUtána[i][j] = HALOTT;
                }  else {
                /* Halott halott marad, ha három élő
                 szomszedja van, különben élő lesz. */
                    if(élők==3)
                        rácsUtána[i][j] = ÉLŐ;
                    else
                        rácsUtána[i][j] = HALOTT;
```

Igaz ugyan, hogy az életjáték egy úgynevezett nullszemélyes játék, de ebben a példában a játékos mégis tudja irányitani kicsit a dolgokat. Ugyanis a program figyeli a billentyűzet bizonyos gombjait (k, n ,l ,g ,s), illetve az egér mozgását, és kattintásait is. Ezt három fügvénnyel teszi. Az `addKeyListener(new java.awt.event.KeyAdapter()` fügvénnyel figyeli a billentyűzetet. Ezen a függvényen belül egy if-else szerkezet állapitja meg, hogy éppen melyik gombot nyomta le a felhasználó a

```
if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K){}
```

feltétel a K betű lenyomását azonosítja, és csökkenti a sejtek méretét.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N){}
```

feltétel az N betű lenyomását azonosítja, és növeli a sejtek méretét.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S){}
```

feltétel az S betű lenyomását azonosítja, és készít egy képet a sejttérről.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
```

feltétel a G betű lenyomását azonosítja, gyorsítja a szimuláció sebességét.

```
else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L
```

feltétel az L betű lenyomását azonosítja, és lassítja a szimuláció sebességét.

Az egér mozgását, illetve kattintásait pedig a `addMouseListener(new java.awt.event.MouseAdapt` illetve a `addMouseMotionListener(new java.awt.event.MouseMotionAdapter(){}` függvények figyelik. Az egér kattintásaival egy sejt állapotát tudjuk megváltoztatni. Az egér mozgatásával pedig az összes érintett sejt élő állapotba kerül.

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalffy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyN Conway/Qt

A megoldás forrása Bátfai Norbert tulajdona.

Az életjátékot John Conway találta ki, és nem teljesen hiteles rá a játék kifejezés, mert ez egy úgynevezett nullszemélyes játék. Magának a játékosnak annyi a dolga, hogy megadja a kezdőalakzatot, majd pedig megfigyelheti, hogy mi lesz az eredmény.

Alapja egy négyzetrácsos tér, amikben élhetnek sejtek, de minden egyes négyzetben csak egy darab sejt élhet. A játék szabályai a következők:

Ha egy sejtnek kettő vagy három élő szomszédos sejtje van, akkor a sejt meg fogja élni a következő generációt. Az összes többi esetben viszont kihal a sejt, akár azért mert túl sok, akár azért mert túl kevés szomszédja van.

Ahol azonban egy üres négyzetrácsnak pontosan három élő sejt a szomszédja, akkor ott új sejt jön létre.

Érdekes az, hogy milyen hatást váltott ki az emberekből. Voltak akik napi rutinjukká tették azt, hogy az életjátékkal "játszanak", egyfajta függők lettek, és voltak azok, akik nem értették hogy mi a jó benne.

Maga a program ugyanúgy működik, mint a java verzió. Mind a két program két darab mátrixxal dolgozik, viszont itt a teljes kód megirása helyett a Q-t is segítségül hívjuk.

A programot a következőképpen tudjuk fordítani és futtatni: **qmake Sejtauto.pro make ./Sejtauto**

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása: https://github.com/Viktorpalffy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyN Conway/BrainB

A megoldás forrása Bátfai Norbert tulajdona.

A programhoz szükségünk lesz az OpenCV-re, aminek a feltelepitéséhez a lépéseket ezen a linken elérhető weblapon találjuk.

Ez egy miniatűr játék, ami a felhasználó szem-kéz koordinációjáról, illetve a megfigyelőképességéről gyűjt össze információkat.

Amikor elindítjuk a játékot akkor egy ablak fogad minket, és a lényege az, hogy a **Samu enthropy** nevű négyzeten belül lévő fekete pöttyön belül tartsuk az egerünk mutatóját.

A játék a teljesitményünk alapján lesz könnyebb, vagy nehezebb. Minél jöbban játszunk, annál több Enthropy lesz a képernyőn, ezáltal megnehezítve a Samu enthropy követését. Viszont ha már nem tudjuk nyomon követni a Samu enthropy-t akkor folyamatosan eltünteti a hozzáadott enthropy-kat, ezáltal megkönnyítve a játékot.

A programot futtatni az előző feladathoz hasonlóan szintén a **qmake** és **make** parancsokkal lehet forditani.

7.5. Vörös Pipacs Pokol/19 RF

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

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

A Red Flower Hell II. körében a célunk ugyanaz mint eddig volt: a lehető legtöbb pipcsaot gyűjteni, amíg a láva leér az aréna aljába. Azonban az első körhöz képest néhány változtatást hozott Tanár Úr a szabályokban: tilos az XML fájl adatait felhasználni, tilos abszolút mozgási formát használni, tilos játékmódot változtatni, tilos az XML fájlt változtatni és tilos átvenni az egér fölött az irányítást.

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

Megoldás videó: https://youtu.be/rH63aO_CVDg

Megoldás forrása: https://github.com/Viktorpalffy/bhax/tree/master/bhax/thematic_tutorials/bhax_textbook_IgyN Schwarzenegger

Ebben a feladatban egy neurális hálót építünk Pythonban, a Tensorflow használatával. A Tensorflow a Google gépi tanulási platformja.

Ez a példa 60000 mintából épít fel egy neurális hálót. A mintákon kézzel írt számjegyek láthatóak. A háló célja az, hogy a megtanult minták alapján meg tudja mondani egy képről, hogy azon milyen szám található.

Amint ez a videóban is látható, 5 futtatás után a pontosság egy kicsivel több mint 97 százalék.

8.2. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

8.3. Minecraft-MALMÖ

Most, hogy már van némi ágensprogramozási gyakorlatod, adj egy rövid általános áttekintést a MALMÖ projektről!

Megoldás videó: initial hack: https://youtu.be/bAPSu3Rndi8. Red Flower Hell: https://github.com/nbatfai/-RedFlowerHell.

Megoldás forrása: a Red Flower Hell repójában.

A Minecraft Malmö projekt egy mesterséges intelligencia fejlesztésére szolgáló platform. A projekt a Minecraft nevű játékra épül, mert ahogy annak mottója is hirdeti: "Csak a kreativitásod szab határt!". Ezt a játékot sokan csak virtuális LEGO-nak hívják a játékos szabadsága miatt, hogy tényleg bármit el lehet benne készíteni. A projekt nagyon sikeres a fiatalabb tanulók körében, hiszen mégse csak száraz kódot kell írniuk, hanem látják is a munkájuk eredményét egy pillanat alatt.

A BHAX Malmö projektben egy fordított piramis pályán kell minél több vörös pipacsot összegyűjtenie a karakterünknek 300 másodperc alatt, aminek a mozgását vagy mi, vagy pedig az általunk megírt program szabályozza. A célunk az, hogy előbb utóbb egy teljesen "önműködő" karaktert tudjunk létrehozni, aki a saját belátása szerint, az általa legoptimálisabbnak ítélt módon szedje fel az lehető legtöbb virágot, mielőtt a láva a pálya aljára ér, ezzel őt megölve. Hétről hétre egyre több elemmel bővítjük a tudását. Eleinte csak kezdetleges mozgási parancsokat adtunk ki neki, aztán látóteret is kapott (először 3x3x3-ast, azóta 7x7x7-nél járunk), utasításokat, hogy mit tegyen ha a látóterében virág található. A következő kihívás pedig az, hogy mihez kezdjen egy olyan pályával, ami minden futtatás alkalmával más és más.

8.4. Vörös Pipacs Pokol/javíts a 19 RF-en

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

Ez a feladat megegyezik az előző fejezetben lévő Malmö-s feldattal, csupán annyi benne a különbség, hogy itt a virágokat nem lentről felfelé haladva kezdjük el gyújteni, hanem a karakter az induláskor azonnal a pálya teteje felé veszi az irányt és onnan halad lefelé a megszokott csigavonalban.

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

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

Megoldás forrása:

9.2. Gimp Scheme Script-fu: króm effekt

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

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

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

Tanulságok, tapasztalatok, magyarázat...

9.3. Gimp Scheme Script-fu: név mandala

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

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

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

Tanulságok, tapasztalatok, magyarázat...

10. fejezet

Helló, Gutenberg!

10.1. Juhász István - Magas szintű programozási nyelvek 1 olvasónaplója

A számítógépet programozó nyelveknek három szintje van. Ezek a gépi nyelv, az assembly nyelv és a magas szintű nyelv. Mi a magas szintű programozási nyelvekkel foglalkozunk, ami az emberek által a legjobban érthető. Az ilyen nyelven megírt programot nevezzük forrásprogramnak. Azonban a processzorok csak az adott gépi nyelven írt programokat tudják végrehajtani. Ezért forrásprogramot át kell írni gépi kódra. Ezt a munkát végzik el a fordítók.

Minden programnyelvnek van saját szabványa, ez a hivatkozási nyelv. Pontosan meg vannak adva a nyelvtani szabályok, amiket be kell tartani, különben vagy szintaktikai, vagy szemantikai hibát fogunk kapni. A szintaktikai hiba az, amit a fordító észrevesz, és jelez nekünk, hogy gond van. Míg a szemantikai hiba esetén a fordító nem kapja el a hibát, de a program nem megfelelően fog működni.

Ezen kívül minden nyelvnek vannak Adattípusai is. Ezek lehetnek beépített, vagy a programozó által kreáltak is. Ilyen típusok például az egész számok, a karakterek, a karakterláncok, a tömbök, a listák, a mutatók.

Léteznek nevesített konstansok is. Ezek is lehetnek beépítettek, vagy létrehozottak is. Ilyen konstans például a pí. Létrehozni pedig c++ nyelvben a #define-al míg java-ban a final utasítással tudunk.

A legalapvetőbb dolgok azonban a változók. Ezekben tároljuk a számunkra szükséges dolgokat. Egy változónak van típusa és értéke. A típusa lehet szám, karakter, karakterlánc, illetve logikai. Az értéke pedig a lehetséges típusok alapján lehet szám, karakter, karakterek sorozata, illetve igaz/hamis.

A programozási nyelvekben használunk még Kifejezéseket is. Ezek egyfajta műveletek. Egy kifejezésnek három része van. A művelet bal szélén valamilyen változó áll, aminek szeretnénk egy értéket adni, középen egy műveleti jel, és jobb oldalt pedig vagy egy másik változó, vagy egy konkrét érték, ami a bal oldalt álló változónak az új értéke lesz.

A gépi kódot a fordító az utasítások alapján generálja. Ezek az utasítások a következők lehetnek : Értékadó utasítás; Üres utasítás; Ugró utasítás; Elágaztató utasítások; Ciklusszervező utasítások; Hívó utasítás; Vezérlésátadó utasítások; I/O utasítások; Egyéb utasítások.

A ciklusokat is rengetegszer használják a programozók. Ezek segítségével a megadott parancsokat egymás után többször is elvégzi a program. A ciklusokhoz tartoznak a Vezérlő utasítások, amik a következők: A

Continue parancs esetén a ciklus jelenlegi lépésében a hátra lévő utasításokat nem hajtja végre, hanem a következő ciklusléptésre ugrik. A Break parancs esetén a ciklus megáll, és nem fut tovább A Return parancs esetén leáll a ciklus és visszaadja az eredményt.

Az alprogramok, vagy másnéven függvények olyan programrészletek, amiket megírva később meg lehet hívni őket, és a megadott értékekből előállítanak egy eredményt. Az alprogramoknak van neve és vannak argumentumai. A nevével hívjuk meg őket, az argumentumok pedig azok az értékek amikből a végeredmény áll elő.

A programokban a blokkok olyan programrészletek amik programrészletekben helyezkednek el. Ilyen például az if elágazás után a potenciálisan végrehajtandó utasítások.

10.2. Kerninghan és Richie olvasónaplója

Először is A vezérlési Szerkezetek a ciklusok, az elágazások

Az elágazásokba beletartozik az if, if-else, else-if, else, és a switch feltételvizsgálatok. Ezekkel értékeket tudunk vizsgálni, és ezek végrehajtatni a megfelelő utasítást, utasításokat végrehajtani. Az if, if-else, else-if, else kifejezéseknél az if után vizsgáljuk meg az értéket, majd jönnek az utasítások, és végül opcionálisan else-if vagy else. Bármennyi else-if lehet egymás után, azonban else csak egy vagy nulla. Viszont célszerű else-t is használni, mert általában kevés esély van arra, hogy minden esetet lefedünk szimplán else-if használatával.

Ezzel szemben a switch esetében megadjuk az értéket, majd tetszőleges darabszámú case használatával megnézzük, hogy az e az érték, ami nekünk kell, és ha igen akkor az aktuális case utasításait hajtja végre. Célszerű megjegyezni, switch-case használatánál a program minden esetben végigellenőrzi az összes case-t, ezért ha nem szeretnénk, hogy az összeset ellenőrizze, ha már talált egy egyezést, akkor használjuk a break utasítást.

A Ciklusok esetében beszélhetünk for, while, és do while ciklusokról.

A for esetében a programozó adja meg, hogy hányszor fusson le a ciklus. A while és a do while esetében pedig addig fut a ciklus, amíg egy feltétel nem teljesül. Éppen ezért vigyázni kell, nehogy véletlenül egy végtelen ciklus alakuljon ki. Fontos különbség még a while és a do while ciklusok között, hogy míg a while ciklus először ellenőrzi, hogy teljesült e a feltétel, majd pedig lefuttatja az utasításokat, addig a do while ciklus először lefuttatja az utasításokat, majd pedig ellenőrzi, hogy teljesült e már a feltétel.

A C nyelv alapvető adattípusai az int, a float, a double, a char, és a bool.

Az intek (integerek) egész számok amik lehetnek pozitívak és negatívak is. Az int mérete 4bájt, azaz 32bit

A float és a double típusú változókban valós, úgynevezett lebegőpontos számokat lehet tárolni. Ilyen például a 0.5. A különbség a két változó között azonban az, hogy, hogy míg a float mérete csak 4bájt, addig a double méréte 8bájt.

A char (character) típusú változóban meglepő módon egy karaktert lehet eltárolni. A char mérete 1bájt.

Az alapvető adattípusokon túl a C nyelvnek vannak Állandói is. Ilyen például a #define, amivel meg tudunk adni meg nem változtatható értékeket. Ezekre később hivatkozni tudunk. De ilyen állandók még az escape sorozatok, amiket az adatok kiíratásánál tudunk alkalmazni. ilyen például a \n amivel egy új sort kezdünk.

10.3. Benedek Zoltán, Levendovszky Tihamér - Szoftverfejlesztés C++ nyelven olvasónaplója

A C++ egy objektum orientált programozási nyelv, ami egyben alacsonyabb szintű elemeket is támogat.

A C++-ban ha egy függvényt paraméterek nélkül hívunk meg, akkor az egyenértékű egy void paraméterrel. Aminek pont az a jelentése, hogy a függvénynek nincs paramétere. További különbség, hogy míg a C nyelvben egy függvényt csak a neve alapján azonosítunk, addig C++ ban egy függvényt a neve és az argumentumai határoznak meg. Ezáltal C++ ban előfordulhat két ugyan olyan nevű függvény különböző argumentumokkal. További változás, hogy C++ ba be lettek vezetve a referenciák, valamint egy új típus is bevezetésre került, ami nem más mint a bool. A bool egy logikai változó ami lehet igaz vagy hamis értékű

A C++ bevezette az osztályokat, amik az adatok, és metódusok együttese. Innen ered az objektum orientáltság, mivel az objektum a egy darab osztály egy darab előfordulása. A metódus pedig az osztálynak egy olyan eleme, egy olyan függvény, ami az osztályba tartozó adatokat manipulálja.

A konstruktorok és destruktorok előredefiniált függvénymezők, amelyek kulcsszerepet játszanak a C++ nyelvben. Alepvető probléma a programozásban az inicializálás. Mielőtt egy adatstruktúrát elkezdenénk használni, meg kell bizonyosodnunk arról, hogy megfelelő méretű tárterületet biztosítsunk a számára, és legyen kezdeti értéke. Ezt a problémát orvosolják a konstruktorok.

A destruktorok pedig egy konstruktor által már létrehozott objektum törlésében segítenek. Törlik a tartalmát, és felszabadítják az objektum által elfoglalt helyet. Ha mi nem hozunk létre destruktort, akkor a C++ a saját alapértelmezett változatát fogja használni.

Létezik még másoló konstruktor is, ami egy már meglévő objektumból hoz létre egy újat. Lefoglal a memóriában egy részletet, és annak az értékét felülírja a már létező objektum értékeivel.

A C++ nyelven az osztályok adattagjai előtt szerepelhet a static szó. Ez azt jelenti, hogy ezeket a tagokat az osztály objektumai megosztva használják.

Gyakran kerülünk olyan helyzetbe, hogy egy adott típusnak úgy kellene viselkednie, mint egy másiknak. Ekkor kell típuskonverziót alkalmazni. Ezt meg lehet tenni implicit és explicit módon is.

Implicit konverziót általában haonló típusokon lehet elvégezni. Ilyen például ha egy integer változó értékét szeretnénk átadni egy long típusú változónak.

```
int x = 5;
long y = x;
```

Mind a ketten egész szám típusok, viszont a long nagyobb méretű, ezért a konverzió gond nélkül megtörténik.

Ez a módszer explicit konverzió esetén nem biztos, hogy működni fog, és még adatvesztéssel is járhat. Ilyen például ha egy integer változó értékét szeretnénk átadni egy byte értékű változónak. A byte mérete kissebb mint az it, ezért a változó előtt kell lennie egy zárójelnek benne a típussal.

```
int x = 300;
byte y = (byte)x;
```

Itt például az y értéke 44 lesz, mert a 300-at kilenc biten kell felírni, azonban a byte csak 8 bitet tárol, ezért az x-nek csak az első 8 bitjét fogja eltárolni.

C++ ban lehetőségünk van függvénysablonok és osztálysablonok létrehozására is, ezek a temlatek. A template argumentumai eltérnek a hagyományos argumentumoktól. Egyrészt már a fordítás közben kiértékelődnek, ezért a futás közben már konstansok. Éppen e miatt az argumentumok típusok is lehetnek, nem csak értékek.

10.4. Python nyelvi bevezetés

Rövid olvasónapló a [**?**] könyvről.

A Python nyelv egy nagyon magas szintű programozási nyelv. Ezt a nyelvet a programozás megkönnyítésére hozták létre, ennek ellenére nagyon sokrétű a felhasználhatósága. Használják a gépi tanuláshoz, adatbázisok kezeléséhez. Híres programok amik rá épülnek: Blender, The Sims 4, World Of Tanks.

III. rész

Második felvonás

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

11. fejezet

Helló, Arroway!

11.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.2. Java osztályok a Pi-ben

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

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

11.3. Általános

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

11.4. C

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

11.5. C++

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

11.6. Lisp

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

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

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