TARTU ÜLIKOOL
Arvutiteaduse instituut

Ain Isotamm

PROGRAMMEERIMINE *C*-KEELES

Algoritmide ja andmestruktuuride näidetel

TARTU 2009

Tartu Ülikool
Matemaatika-informaatikateaduskond
Arvutiteaduse instituut

Ain Isotamm

Programmeerimine *C*-keeles
Algoritmide ja andmestruktuuride näidetel

Tartu 2009

Käesoleva raamatu väljaandmist on toetanud Eesti Infotehnoloogia Sihtasutus projekti „Tiigriülikool“ raames.

Toimetaja: Jüri Kiho

Küljendus: Ain Isotamm
Keeletoimetus ja trükk: Tartu Ülikooli Kirjastus

ISBN 978-9949-19-302-8

Tartu Ülikooli Kirjastus
www.tyk.ee
Tellimus nr. 538

SAATEKS

Selle õppevahendi kaante vahel püüame anda abivahendeid kahe loengukursuse omandamiseks: need on „Programmeerimine *C*-keeles“ (MTAT.03.219) ja „Algoritmid ja andmestruktuurid“ (MTAT.03.133, edaspidi lühendatult ka *A&A*). Esmalähenduses võiks näida, et neid aineid seob ainult lektor, nende ridade autor. Ent meie arvates on need ained pisut orgaanilisemalt seotud; kuivõrd *C* on üldtunnustatud kui *süsteemiprogrammeerimise* keel, siis meie arvates on ta igati sobiv nii andmestruktuuride kui ka nende töötlemiseks mõeldud algoritmide esitamiseks. Nentigem etteruttavalt, et iga abstraktne andmestruktuur on vahetult seotud tema töötlemist võimaldavate algoritmidega. Näiteks, *vektor*i elemente saame kätte indeksitega ja loomulik algoritm on üles ehitatud tsüklile üle vektori elementide, *puu* läbime (loe: töötleme) tipphaaval rekursiivselt või magasini abil, *tabeli* kirjeid saame kätte võtme abil jne., ning ehkki algoritme võib esitada mitmeti, on programmeerijaile selleks arusaadavaim vorm tuntud-teatud programmeerimiskeelne – nagu *C* või *Forth* – programm. Meie kui programmeerijate jaoks on see üheselt arusaadav[1].

Siinkirjutaja jaoks on alati olnud kõige raskem otsustada, kuidas töö üles ehitada (kui kasutatav materjal on juba koos ja artikli või raamatu temaatika on läbi mõeldud). Millest *alustada*? Praegusel juhul, kas alustada programmeerija-vaatest andmestruktuuridele ja nendega manipuleerivatele algoritmidele, või siis ühe andmestruktuuride ja algoritmide esitamist võimaldava süsteemiprogrammeerimise keele *C* tutvustamisest. Otsus sai tehtud viimase variandi kasuks.

A&A valdkond on pea kõikehõlmav. Leidub hulk ülesandeid, mis opereerivad lihtandmetega kasutades nende töötlemiseks sobivaid algoritme (näiteks, kasvõi ruutjuure võtmine või siis paiskfunktsiooni väärtuse arvutamine). On ka ülesandeid, mis kasutavad suhteliselt keerulisi andmestruktuure (näiteks *puid* või *graafe*), kuid algoritm võib samas olla esitatav triviaalse rekursiooniga.

Ja muidugi leidub ülesandeid, kus tuleb kasutada nii keerulisi andmestruktuure kui ka keerulisi algoritme. Meil – ma mõtlen programmeerijaid, on vedanud: oluline osa *A&A* temaatikast lahendati möödunud sajandi 50–60-ndatel aastatel, ning leidus (tol ajal) noor mees, kes suutis anda selle valdkonna põhjaliku ja täpse kokkuvõtte ning ülevaate – *Don(ald) Knuth.* Tema elutöö, „*The Art of Computer Programming*“ (eesti keeles võiks see olla „Programmeerimiskunst“) hakkas ilmuma alates 1968. aastast[2]. Ent meie lugejatele võime mingigi kõhkluseta soovitada – kui te ei tea, kuidas midagi teha (andmestruktuurid? algoritm?), siis vaadake, kas *Knuth*i raamatutes pole lahendust. Tõenäoliselt on[3].

---

[1] Objektorienteeritud „stiil“ seadustab selle lähenemise üheselt: **andmestruktuurid+meetodidid**. Tegelik programmeerimiskeskkond on mõnevõrra vabam. Ja objektorienteeritud keeled (näit. *C++*) *pole MicroSofti* tarkvara tegemise vahendid (tehakse „tavalises *C*-s“). Mainigem veel, et kui programmi loogika on keeruline ning programm on „pikk“, siis on abiks (enne „kodeerima“ asumist) *lahendusgraafi* joonistamine ja läbi mängimine (vahet pole, kas joonistame plokk-skeeme või *Kiho skeeme* (vt. [Kiho 03, lk. 7 – 10] ja seal osundatud asjakohased materjalid) – mõistagi, osundatud materjalides viimaseid *Kiho skeemideks* ei nimetata).

[2] Kontsentreeritud ülevaate tänaseks ilmunud kolmest esimesest köitest, käsikirjana peaaegu valmis 4. ja 5. osast *Combinatorial Algorithms* ja *Syntactic Algorithms* ning kavandatud kahest viimasest, *The theory of context-free languages* ja *Compiler techniques* annab [Knuth TAoCP]. Samast allikast saame teada, et mainitud teos valiti 1999. aastal USAs möödunud sajandi 12 tähtsaima reaalteadusalase publikatsiooni hulka – ülejäänute autorid on näiteks *Dirac* (kvantmehhaanika), *Einstein* (relatiivsusteooria), *Pauling* (keemia), *Russell* ja *Whitehead* (matemaatika), *von Neumann* ja *Morgenstern* (mänguteooria, tõsi küll, meie jaoks on *von Neumann* eeskätt arvutiteaduse suurkuju), *Wiener* (küberneetika).

[3] Paraku, mitte *C*-keeles; *Knuth* kasutas oma abstraktse masina abstraktset keelt, ja *C* seda ei järginud.

Donald E. Knuth (高德纳), sünd. 10.01.1938 *Milwaukee*'s (*Wisconsin, USA*), *Stanford*i Ülikooli emeriitprofessor (hieroglüüfid edastavad tema nime hiina keeles) [Knuth].

A&A põhi- ja lisamaterjalidena soovitame meie (*TÜ*) üliõpilastele *Jüri Kiho* publikatsioone[1], ent lisaks ka *Tallinna Tehnikaülikooli* dotsendi *Viktor Leppikson*i [Leppikson, A&A] ja *Infotehnoloogia Kolledž*i dotsendi *Jaanus Pöial*i [Pöial] materjale.

Ja muidugi soovitame lugejal otsida *Internet*is leiduvaid asjakohaseid linke, samas rõhutades, et leitut tuleks alati juba olemasoleva infoga võrrelda ja vajadusel üle kontrollida.

C-keele tundmaõppimiseks on mõistagi parim *Dennis Ritchie* raamat, mida aitas kirjutada *Brian W. Kernighan* [K&R]. Teemakohaseid raamatuid on kirjutatud palju; huvilistel soovitame tutvuda õpperaamatukogude kataloogidega ning mõistagi otsida materjale *Internet*ist. Siinkohal võime soovitada eesti keeles ilmunud (ja kättesaadavaid) materjale:

- *Tõnis Kelder*i ja *Ülo Kaasik*u raamatud [KjaK], [KjaK-2];
- *Viktor Leppikson*i raamat [Leppikson C];
- *Veiko Sinivee* [Sinivee] materjalid.

Kuid meie raamat ei püüa anda kuigivõrd ammendavat ülevaadet *A&A* temaatikast. Näiteks, selle aine matemaatilist tausta, sh. *ajalist keerukust* refereerime vaid põgusalt. Ja täpselt samamoodi ei taotle me edastada *kogu* infot *C*-keele kohta (selleks võinuks me tõlkida [K&R]-raamatu mingitegi kommentaarideta); meie arvates kujuneb suvalises programmeerimiskeeles (sh. *C*) programmeerijal oma „alamhulk" kasutatavast keelest, mida ta valdab ilma manuaalideta ja mille kasutamisel ta on kindel oodatavates resultaatides. Niisiis: me käsitleme väljavõtteliselt *C*-keelt, *C*-näidetel *andmestruktuure* ja *C*-keeles mõningaid *algoritme*. Ning meie raamat ei pretendeeri olema ei *C*-keele ega *A&A* ammendav õpik, vaid olema *lihtsalt* õppevahendiks[2] nii *A&A* kui ka *C* ainekursuste omandajatele.

„Läbilöönud" süsteemprogrammeerimise keeltena on tänini tunnustatud eeskätt kaht keelt: kõikvõimalike riistvara-sensorite signaalidele interaktiivset reageerimist ja mõistagi riistvara juhtimist võimaldavat *Forth*i ning operatsioonisüsteemide programmeerimisele ja tarkvara

[1] Vt. [Kiho 97], [Kiho 03] ja [Kiho 05].
[2] Arvame, et *õpik* tohib esitada ja peab esitama ainult ajaproovi läbi teinud tõdesid, ja mitte hüpoteese, subjektiivseid arvamusi, kahtlasi mõttekäike jms. *Õppevahend* võiks olla neist kitsendustest vähem sõltuv.

portatiivsusele[1] orienteeritud *C*-keelt. Nende ridade autor vahetas *IBM*i makroassembleri pikemaks ajaks (1990 – 1993) *Forth*i vastu, ent läks *Forth*ilt *C*-le üle 1994. aastal, tehes tellimustööd kartograafiafirmale *Regio* (vt. [chcoord]). Edasine oli juba „asjade loomulik käik", lühikeste tähtaegadega töömahukaid projekte realiseerisime („meie" on *Mati Tombak* ja autor) edaspidi *C*-s. Kusjuures, subjektiivselt on *Forth* süsteemide programmeerimiseks parem kui *C*, ent viimane võimaldab kiiremini programmeerida[2].

Selle raamatu autor on siiralt tänulik kõigile, kes aitasid kaasa meie õppevahendi koostamisele, neiks on kõik loengukursuste *Programmeerimine C-keeles* ja *A&A* aktiivsed kuulajad; eraldi väärivad esiletõstmist *Margus Niitsoo*, *Lauri Rätsep* ja *Simon Vigonski*. Ja kõik viidatud autorid. Ning mõistagi on suurima tänu pälvinud toimetaja *Jüri Kiho*. Lisagem tsitaat:

„Loomulikult ei tähenda tänu avaldamine vastutuse jägamise katset – kõikide võimalike sisuliste vaieldavuste ja leidmata trüki- jm. vigade autorlus kuulub jägamatult nede ridade kirjutajale (programmeerijaina teame, et kirjutades keeles, kus puuduvad silumisvahendid, on vead vältimatud)[3]".

[1] *C* oli algselt samuti orienteeritud võimaldama masinatasemel programmeerimist. Ent lisagem, et teadaolevalt on konstrueeritud *Forth*- (nagu ka *Lisp*-)-protsessoreid (primitiivid on realiseeritud spetsiaalsete masinkoodikäskude abil), ent taoline *C*-protsessor pole mõeldav. *C* järgib traditsiooniliste protseduur-orienteeritud keelte liini. *C* portatiivsus seondub *UNIX*i võidukäigu ja selle implementeerimiskeele *C* endaga – aga need on omavahel orgaaniliselt seotud.

[2] *Windowsi-C* pole sellest aspektist üldse võrreldav *Forth*iga, kuivõrd mainitud orientatsiooniga *C*-kompilaatorid *ei võimalda* programmeerida riistvara juhtimist; *Window*si-põhisest *Forth*ist pole aga midagi kuulda.

[3] Vt. [Isotamm, PK].

Sisukord

1. UNIX

Joonis 1.a. *Ken Thompson, Dennis Ritchie* ja *Bill Clinton.*

Ülalolev foto on tehtud 27. aprillil 1999 Valges Majas, kus USA president andis *K. Thompson*ile ja *D. Ritchie*le üle *Rahvusliku tehnoloogia medalid UNIX*i ja *C* eest [wKT]. Kuidas need mehed, *Thompson* ja *Ritchie*, ning nende loodud programmeerimiskeskkonnad, so. operatsioonisüsteem *UNIX* ja süsteemiprogrammeerimise keel *C* üksteist mõjutasid ning kuidas ja miks need süsteemid tehti, sellest proovime anda lühikese ülevaate järgnevates alampunktides. Aastaid varem, 1983. aastal said mõlemad mehed *Turing*i auhinna (vt. lisa Turing) *UNIX*i eest.

1.1. Kenneth L. Thompson ja UNIX

Oleme harjunud seostama *UNIX*it eeskätt *Ken Thompson*iga ja *C*-d vaid *Dennis Ritchie*ga. Auhindade määramine näitab, et päris nii nende süsteemide tegemine ei käinud, mõlemi süsteemi tegemisel osalesid nähtavasti mõlemad mehed. Järgmised põgusad leheküljed peaksid seda muljet kinnitama.

C-keel ja (algselt firma *DEC PDP*-seeria jaoks loodud) operatsioonisüsteem *UNIX* on omavahel tihedalt seotud. *UNIX*i projekti[1] vedas *Kenneth* („*Ken*“) *Lane Thompson* (sündinud 4. veebruaril 1943).

Ken on pärit *New Orleans*ist (*Lousiana*). Ülikoolihariduse, so. bakalaureuse- ja magistrikraadi sai ta *California Berkeley* Ülikoolist, erialaks elektrotehnika ja arvutiteadus. 1960-ndatel aastatel olid nii *Ken* kui ka *Dennis Ritchie* tegusad *Multics*i-nimelise operatsioonisüsteemi[2] tegemise juures; *Ken* oli selle süsteemi üks peamistest vedajatest. Tolle operatsioonisüsteemi kirjutamise huvides evitas *Ken* muide *Bon*-keele. Suurarvutitele orienteeritud *Multics* osutus uutele masinatüüpidele ülekandmiseks siiski sobimatuks – mis viis *Ken*i ja ta sõbra *Dennis Ritchie* mõttele 1969. aastal teha lihtsam ja loomulikum operatsioonisüsteem *Unix*[3]. *Ken* lõi

[1] *UNIX*i autoritena mainitakse *Ken*i, *Dennis Ritchie*d, *Douglas McIlroy*d ja *Joe Ossanna*t

[2] *Multics*i eest sai 1990. aastal *Turing*i auhinna tolle projekti juht *Fernando J. Corbató*

[3] Täheldagem võimalikku sõnamängu: *multics*≈“mitmene“, ja *unics*≈“ühene“. *Unics* oligi uue süsteemi nimi seni, kuni *Brian Kernighan* pakkus välja tuntuks saanud nime *UNIX*.

uue operatsioonisüsteemi programmeerimiseks keele *B* (nimi viitab prototüübile *BCPL*, mida me tutvustame põgusalt hiljem) ning *B* sai hiljem omakorda prototüübiks *Ritchie C*-keelele.

Ken on hilisemal ajal tuntud kui autoriteet *regulaarsete keelte* ja *regulaaravaldiste* ning neid tuvastavate automaatide valdkonnas[1], ta on tegelenud muuseas ka maleprogrammidega (eriti lõppmängudega).

Kaasaegsete mikroprotsessorite eelkäijaks võib pidada *PDP*-arhitektuuri, ja *Ken* ning *Dennis* olid ühed esimesed, kes leidsid nonde „riistade“ jaoks adekvaatsed programmeerimisvahendid.

Joonis 1.b. Miniarvuti *PDP*-11.

PDP-11 evis olulist rolli kahe süsteemiprogrammeerimise keele (*Forth* ja *C*) ning ühe operatsioonisüsteemi (*UNIX*) saamisloos [PDP-11][2]. Selle masina arhitektuurist ja rollist on juttu ka raamatus [Isotamm PK, lk. 128 – 138].

Niisiis, järgnevas püüame anda lühiülevaate *UNIX*i[3] saamisloost. Süsteem tehti *Bell Laboratories* egiidi all, möödunud sajandi 70-ndate alguses. Eesmärk oli tol ajal uudsete serverite ja tööjaamade jaoks vastuvõetava keskkonna loomine. *UNIX* pidi olema mitmekasutaja ajajaotussüsteem, mis suhtleb kasutajatega interaktiivselt *plain-text* režiimis, so, *ASCII*-tasemel, toetama failide ja nendega võrdsustatud protsesside ja välisseadmete hierarhilist üles-ehitust (puustruktuuri) ning „konveierit“ – kus ühe protsessi väljundist saab teise sisend.

Juba ülalpool mainitud *Multics*[4] oli suurarvuti- (*mainframe*) operatsioonisüsteem, mida toetasid (arendasid) sellised suured keskused nagu *MIT* (*Massachusetts Institute of Technology*), *AT&T Bell Labs* ja *General Electric* (*GE*). Ja projekteeritud oli see *GE* suurarvuti jaoks. Ent erinevalt *IBM*i *OS*ist, mis oli orienteeritud assembleri makrodele, oli *Multics* orienteeritud käsureadirektiividele, just nii nagu hilisemad *UNIX* või *MS-DOS*. *Tom Van Vleck* [Vleck] kirjutab, et järjepidevus *Multics* $\rightarrow$ *UNIX* on paljudel juhtudel silmnähtav, näiteks (lisame *MS-DOS*i): käsk *ls* kuvab mõlemis süsteemis ekraanile teegi parameetrid (*MS-DOS*i analoogiline funktsioon on *dir*) või *UNIX*i *man*[5] on samaväärne *Multics*i ja *MS-DOS*i *help*-käsuga.

[1] Me tutvustame *Thompsoni algoritme* ettevalmistusjärgus olevas õppevahendis [IŠ].
[2] Samuti sobis *PDP*-11 väga hästi *Lisp*ile; selle autor *J. McCarthy* on maininud, et aparatuurne magasin realiseeriti *DEC*is just tema ettepanekul (ja palvel). Vt. näit. [Isotamm PK, lk. 176].
[3] Miks: esiteks, *UNIX* kirjutati lõppversioonis *C*-keeles, ja me käsitleme järgnevas *UNIX*-keskkonnas realiseeritud *C*-keskkondi (*gcc, djgpp, cygwin* jt.)
[4] *Multiplexed Information and Computing Service*.
[5] *man=manual*, käsiraamat.

Suurarvutite interaktiivsed operatsioonisüsteemid olid suhtlemisvõimelised operaatorikirjutusmasina või terminalide vahendusel, arvatavasti oli see nii ka *Multics*i puhul.

*AT&T Bell Labs*i meeskonnas jätkas *GE*-projektis osalemist *Ken Thompson,* kes kirjutas oma lõbuks selles keskkonnas töötava *mängu „Space Travel“* [wUnix][1]. Hiljem kirjutas ta koos *Dennis Ritchie*ga selle ümber *PDP*-7-le[2], ja sellega seoses hakkas disainima tollele masinale uut operatsioonisüsteemi. Eesmärgiks oli teha väike ajajaotussüsteem; selle ülekandmine sootuks paremale mudelile (*PDP-11/20*) muutus aktuaalseks 1970. aastal; mainitud mäng oli uue operatsioonipsüsteemi *UNIX* algversiooni esimene rakendus [Thompson]. Muide, *Thompson* kirjutas assembleris oma operatsioonisüsteemi tuuma[3] valmis umbes nädalaga.

Järgmine aasta kulus *Thompson*il *B*-keele tegemisele. Me käsitleme *B* kaugemat prototüüpi, *BCPL*i ja *B*-d ennast põgusalt hiljem. *B*-keelest *C* disainimise jättis *Thompson Dennis Ritchie* teha (valmis sai see 1972. aastal). Viimatimärgitud aastal kirjutas *Thompson UNIX*i tuuma ümber *C*-keeles, ning *C* tõttu muutus *UNIX* mobiilseks operatsioonisüsteemiks: *C* oli lihtsalt realiseeritav suvalisel platvormil ning *UNIX* oli realiseeritav igal pool, kus on realiseeritud *C*.

K. Thompson esitles *UNIX*it esmakordselt 1973. aastal.

Meie raamatu kontekstis on *UNIX* oluline kahel põhjusel: esiteks, *C*-keel loodi selle operatsioonisüsteemi programmeerimiseks, ja teiseks – mitmel põhjusel on meie ülikooli *C*-keele ja nende ridade autori juhendatavate *A&A* arvutipraktikumide keskkonnaks just *UNIX* [4].

Põhjused on lühidalt järgmised: töötada vabavaraga, vältida *Windows*i-kesksust, võimaldades ka *UNIX*i ja *Linux*i platvorme (ühel rakendusel – *Dev-C++* – on *Windows*i graafiline liides *UNIX*i keskkonnaga). Kasutatud materjalide loetelus on mitmeid viitu *UNIX*it tutvustavatele materjalidele; need on kokku võetud nimega [UNIX] ja on kõik viimati vaadatud 2007. a. 24. septembril.

1.2 UNIXi (Linuxi) lühikonspekt

Joonis 1.2.a. *Jaanus Pöial*.

Selles alapunktis reprodutseerime *Jaanus Pöial*i (ATI endise dotsendi, nüüd *IT kolledži* õppejõu) materjale [Pöial-U], kes andis selleks lahkelt loa. *Jaanus* tutvus *UNIX*iga, kui ta programmeeris *CM*-4 (loe: *PDP*-11) jaoks *Forth*i ja translaatorite tegemise süsteemi *Mati*

[1] See seik vihjab ilmselt asjaolule, et *Multics*i-poistel *polnud* piisavalt motivatsiooni ja *oli* piisavalt vaba aega. Pisut olulisem oli asjaolu, et ehkki *Multics* projekteeriti teenindama *sadu* kliente, siis tekkisid probleemid juba rohkem kui kolme kliendiga [Thompson].
[2] Üheks põhjuseks oli tolle mängu liigkõrge ($75) hind *GE*-masinale ligipääsu omajatele (muud kasutajad seda mängu ei saanud mängida).
[3] Osisteks olid käsurea-protsessor *shell*, tekstiredaktor ning assembler, kirjutamiskeeleks oli *PDP*-7 assembler.
[4] Siinkirjutaja jaoks on senikogetuist (mõnest aspektist) parim *Windows*i keskkonnale orienteeritud *Visual C++*, ent see on suhteliselt kallis, ka ülikoolidele.

Tombaku meeskonnas. Hiljem keskendus ta (*Forth*ist lähtuvalt) magasini kaudu edasta-tavate parameetrite semantikale ja saavutas oma valdkonnas laiema tunnustuse.

Niisiis, allpool *UNIX*i lühiülevaade *Jaanus Pöial*i käsitluses.

Seansi alustamine:
`login:` *`kasutajanimi`* (reavahetus lõppu)
`Password:` *`parool`* (ei näidata ekraanil)
Terminali tüübi valimine (tekstiterminali korral):
`dumb ansi vt100 vt220 xterm` ...
Seansi lõpetamine: `logout` või `exit`
Parooli muutmine: `passwd` (küsib vana parooli ja kui see oli õige, siis uut 2 korda)
Failid, kataloogid, seadmed:
Failisüsteem on organiseeritud **puu**na, füüsilised kettaseadmed pole lihtkasutajale olulised – failisüsteemi ehitamine (*mounting*) on üldjuhul administraatori töö. Andmete (failide) lugemiseks, kirjutamiseks või otsimiseks peab teadma **kataloogi**, milles andmed paiknevad. Kataloog (*directory*) on failisüsteemi loogiline osa, üks tipp kataloogide puus; igal kataloogil on omanik ja juurdepääsuõigused.

*UNIX*is eristatakse erinevat tüüpi katalooge:

- **Juurkataloog** (*root directory*, nimega `/`): kataloogide puu juur, failisüsteemi algus.
- **Alamkataloog** (*subdirectory*): iga kataloog võib failide kõrval sisaldada ka alamkataloge (ühe kataloogi failide ning alamkataloogide nimed ei saa korduda).
- **Kodukataloog** (*home directory, C-shell*is nimega `~`): igale kasutajale määratakse kindel "lähtepunkt" failisüsteemis.
- **Jooksev kataloog** (*working directory, current directory*, nimega "punkt": `.`): aktiivne kataloog, algselt reeglina kodukataloog. Jooksva kataloogi muutmiseks on käsk `cd` (*change directory*).
- **Ülemkataloog**i (*parent directory*) nimi on "kaks punkti": `..` .

Kataloogide puus navigeerimiseks kasutatakse **teed** (*path*), mis esitatakse liitnimena ja millega määratakse kataloogi asukoht puus (eraldajaks `/`). **Absoluutne** tee algab juurkataloogist (`/`...) ning **suhteline tee** algab kas kodukataloogist (`~`...), jooksvast kataloogist (`./` ...) või ülemkataloogist (`../` ...). *Jaanus Pöial* toob järgmised näited:

`/usr/local/lib`	(absoluutne);
`~/Mail`	(kodukataloogist lähtuv);
`./bin`	(jooksvast kataloogist alla);
`bin`	(sama, programmide käivitamise koha pealt siiski erinev);
`../../programs`	(jooksvast kataloogist üles).

Failid on kas tekstifailid (*text*) või kahendfailid (*binary*). Igal failil on omanik ja juurdepääsuõigused. **Nimed** on suhteliselt vabalt valitavad, suur- ja väiketähed on (erinevalt näiteks *MS-DOS*ist) erinevad asjad, so. *UNIX* (nagu ka *C*-keel) on *tõstutundlik.* Punktiga algavad nimed on "peidetud" (*hidden*): kui me väljastame kataloogi käsuga *ls* ekraanile, siis neid faile ei kuvata[1].

[1] Kuvatakse, kui kirjutame *ls –a*.

Nime „legaalsed“ komponendid on tähed, numbrid ja alakriips. Hoiduda tuleb järgmiste sümbolite kasutamisest nime moodustamisel:

**& ; | * ? ` " [ ]( ) $ < > { } % ! # @ **

Välisseadmeid käsitleb *UNIX* kui faile /dev-kataloogis, näiteks

```
/dev/audio
```

UNIX on orienteeritud teenindama eeskätt paljude kasutajate ja jagatavate ressurssidega keskkonda; tänapäeval on ta üldkasutatav serverite operatsioonisüsteemina. Kataloogide ja failide kasutajad on jaotatud kolme kategooriasse: omanik (u=*user*), rühm (g=*group*) ja ülejäänud (o=*others*) ning võimalikud tegevused on

- r – *read:* faili, kataloogi lugemine;
- w – *write:* faili muutmine, kataloogi muutmine, s.t. failide kirjutamine ja kustutamine kataloogist;
- x – *execute:* käsufaili või programmi käivitamine, nime otsimine kataloogist.

Pikemata on selge, et näiteks serveri teenuste kasutaja koduleheküljel olevate failidega manipuleerimise võimalused tuleb siduda erinevate õigustega. Modifitseerida faili tohib reeglina ainult omanik (või grupp, kui tegemist on foorumi logiga või kõik, kui tegemist on portaali kommentaariumiga), igale külastajale ei pruugi kõiki asju näidata jne. Juurdepääsuõiguste omistamine ja muutmine on omaniku (võrkude puhul ka administraatori) privileeg.

Õiguste näitamiseks on kasutusel 10-märgiline esitus, õiguste muutmiseks saab kasutada kas kaheksandkoodi või sümbolesitust. Õiguste omistamise ja muutmise viisid on kokku võetud järgmiselt: kasutaja kategooria (*u, g* või *o*) ja lubatud tegevus (*r, w* või *x*), vasakpoolseim märk näitab faili tüüpi (`d` on kataloog, `-` on tavaline fail, `l` on link, `b`, `c` on seadmed jne.) Niisiis, mainitud 10 märki on struktureeritud selliselt, et 0-s sümbol määrab tüübi, sümbolid 1..3 omaniku, 4..6 rühma ja 7..9 – ülejäänute õigused (lubatud tegevused):

- r-- --- --- 400 u+r (tavaline fail; ainult omanik võib ainult lugeda)

- -w- --- --- 200 u+w

- --x --- --- 100 u+x

- --- r-- --- 040 g+r

- --- -w- --- 020 g+w

- --- --x --- 010 g+x

- --- --- r-- 004 o+r

- --- --- -w- 002 o+w

- --- --- --x 001 o+x („külalised“ võivad faili nime teades ainult veenduda, et see fail tõepoolest kataloogis on).

Näiteks, `drwxr-x---` ütleb[1], et tegu on kataloogiga, millele omanikul on kõik õigused, rühmal on lugemise ja otsimise õigus, ja kõigil teistel pole mingeid õigusi. Kood on kaheksandarv 750 ehk kahendkujul 111 101 000. Õiguste komplekti näeb käsuga `ls -l` .

Õiguste kehtestamiseks on kasutatavad järgmised võimalused:

- `+` anda õigus;
- `-` võtta õigus ära;
- `=` anda täpselt määratletud õigused, võttes kõik ülejäänud ära;
- kordame juurdepääsuvõtmeid: `r` – *read*, `w` – *write*, `x` – *execute*, `s` – *set user ID* (ajutine omanikumuutus programmi täitmise ajaks), `u` – *user*, `g` – *group*, `o` – *others* ja `a` – *all* (`a` = `ugo`).

Näiteks, `chmod u=rwxg=rxo=lecture` on kodeeritult `chmod 750 lecture`

Käsurida (*command line*) võimaldab edastada tellimusi operatsioonisüsteemile, need tellimused on kas käsud (programmide väljakutsed või *UNIX*i direktiivid) või käsufailid (nn. *shell*-programmid, skriptid, stsenaariumid jne), neid täidab käsuinterpretaator *shell*.

Enne käsu täitmist asendatakse käsureas olevad **metamärgid `* ? [ ]`** vastavalt nende semantikale. Mainigem, et on olemas erinevaid käsuinterpretaatoreid – `sh, csh, ksh, zsh, tcsh` jm. Lihtsamal juhul koosneb käsurida käsu nimest (näit. `ls`), täpsustavatest lippudest (*flags*, näit. `-l`) ning muudest parameetritest (näit. failinimed).

Üldjuhul töötleb **käsk** mingeid sisendandmeid ning väljastab tulemuse ja võimalikud veateated väljunditesse.

Standardsisendiks on klaviatuur, standardväljundiks ja veaväljundiks terminal. Sisendit ja väljundeid saab ümber määrata.

Enne käsu täitmist asendab käsuinterpretaator kõik käsureal olevad erisümbolid vastavalt nende tähendusele. Metamärgid **`* ? [ ]`** on kasutusel failinimede genereerimiseks:

- **`*`** väljendab mistahes märgijada (sh. ka tühja);
- **`?`** väljendab mistahes (täpselt ühte) märki;
- sümbolid **`[`** ja **`]`** lubavad valida ühe nurksulgudes olevatest märkidest.

Näiteks, kui jooksvas kataloogis on failid

```
kiri1 kiri2 kiri10 Ptk4.txt ptk5.txt .login
```

siis genereeritakse nimesid järgmiselt:

```
*  kiri1 kiri2 kiri10 Ptk4.txt ptk5.txt .login

kiri?    kiri1 kiri2
```

[1] Grupeeritult: d | rwx | r-x | --- : seade, omanik, rühm ja muud.

kiri* kiri1 kiri2 kiri10

[Pp]tk* Ptk4.txt ptk5.txt

. Ptk4.txt ptk5.txt .login

.* .login

Ülakomade ' abil saab teksti "peita" laiendamise eest:

'siin sees ei ole * ? jne. metamärgid'

Käsu täitmise tulemus (standardväljund) saadakse käsureale "tagurpidi ülakomade" abil:

`käsk` annab tulemuseks selle käsu väljundi.

Muutujate väärtusi saab lugeda metamärgi "dollar" abil: $PATH annab tulemuseks muutuja PATH väärtuse.

Et testida metamärkide asendamist, võib kasutada käsku echo, mis peegeldab oma argumendi(d) standardväljundisse:

echo *käsurida*

toimel tehakse kõik asendused käsureas ning kuvatakse see siis standardväljundisse. Interaktiivset abiinformatsiooni saab *UNIX*i keskkonnas järgmiselt:

man – täisinfo käsu vm. kohta. Juhendid on organiseeritud peatükkide kaupa. vt. ka xman;

man <*käsk*> – tavaline kuju, võimalikud parameetrid on järgmised;

-k <võtmesõna>: võtmesõna järgi;

-f <failinimi>;

-s *ptk.nr käsk* – peatüki näitamine.

Muud käsud abi saamiseks on järgmised:

whatis <*käsk*> – lühiinfo käsu kohta;
apropos <*võtmesõna*> – info otsimine võtmesõna järgi;
whereis <*failinimi*> – käsuga seotud failide otsimine;
which <*käsk*> – käsu asukoha täpsustamine (kas *alias*, millises kataloogis).

J. Pöial juhib tähelepanu, et kõik käsukirjeldused on tema konspektis oluliselt lihtsustatud ja kogu informatsiooni saamiseks tuleks kasutada käsku *man*.

Käsud tööks kataloogide ja failidega:

- `ls` - kataloogi sisu näitamine: `ls` [`-alrtuR` ...] [*`failinimi`* ...][1] , kus näiteks võtmed `artu` toimivad järgmiselt:
    1. `r` – *reverse,* väljastatakse tagurpidi-järjekorras;
    2. `t` – väljastatakse viimase muutmise aja järgi;
    3. `a` – näidatakse peidetud infot ka;
    4. `u` – väljastatakse viimase kasutamise aja järgi.
- `pwd` (*print working directory*): jooksva kataloogi nime näitamine;
- `cd`: jooksva kataloogi muutmine – `cd` [*`kataloog`*] - *change directory*, vaikimisi – kodukataloog ;
- `cat` *`failinimi`*... - ühendada failid standardväljundisse (sageli tekstifailid);
- `more` (`less`) [*`failinimi`*...]: faili kuvamine peatustega;
- `ln`: viida (*link*) loomine;
- `chmod`: õiguste muutmine `chmod` *`õigused failinimi`*...
- `chown:` omaniku muutmine[2] `chown` *`uusomanik failinimi`* ...
- tühik - uus lehekülg;
- reavahetus - uus rida;
- `b` - lehekülje võrra tagasi (ei pruugi alati töötada, vt. ka `less`);
- `q` – lõpetada;
- */`regulaaravaldis`* – otsimine;
- `h` – *help*;
- `mkdir` *`kataloog`* ... – luua kataloog(id), eeldab otsimis- ja kirjutamisõigusi ülemkataloogis. Tekivad `.` ja `..`
- `rmdir` *`kataloog`* ... – eemaldada (kustutada) kataloog(id). Eemaldab ainult tühja kataloogi (vrd. `rm -r`);
- `cp` [`-i`] *lähtefail tulemusfail* : failide kopeerimine (nimemuutmisega);
- `mv` [`-if`] *`lähtefail tulemusfail`* : faili kopeerimine;
- `rm` [`-rfi`] *`failinimi`* ... – kustutamine, eeldab kirjutamisõigust vastavas kataloogis; lipp `-r` eemaldab kataloogid ning nende sisu rekursiivselt (see on ohtlik!)

Illustreerimaks tööd välisseadmetega (mis on *UNIX*i jaoks failid nagu kõik muud andmekogumid), toob *J. Pöial* mõned näited. Ta nendib, et multikasutusest tuleneb ressursside jaotamise vajadus – järjekorrad, lukud jms., ning toob näiteid:

- `lpr` – väljastamine printerile; lpr [-P*printer*] [-T*title*] [*filename* ...]. Printeritöö omanik võib numbri järgi töö eemaldada (numbrid vt. `lpq` abil);
- `lpq`, `lpstat` – printeritööde järjekorra vaatamine (vt. ka `lpc status`);
- `mtools` (*mcd, mdir, mcopy, mmove, mren, mdel, mdeltree, mmd, mrd, mtype, mformat, mattrib, mbadblocks, minfo, mlabel,* ...) – MS-DOS käskude analoogid *UNIX*ile (tavaliselt lihtkasutajale keelatud);

---

[1] Failinime puudumisel kasutatakse standardsisendit.

[2] Käske `chmod` ja `chown` täidab omanik (või `root`).

- `prm` – printeritöö eemaldamine järjekorrast.

Regulaaravaldisi kirjeldatakse *Chomsky* klassifikatsiooni (vt. näit. [Isotamm, PK lk. 234 jj.]) järgi 3. (viimase ehk madalaima) klassi grammatikatega. Mäletavasti on see valdkond üks neist, kus *Ken Thompson* on resultatiivne suutnud olla. Tsiteerigem *Pöial*t: „regulaaravaldis (*regular expression*) kirjeldab mustrit ("näidist", *pattern*) sõnetöötluses[1] (näit. otsimis- ja asendusoperatsioonides)".

Lihtsustatud samm-sammuline definitsioon algab ühesümbolilise avaldise defineerimisega ning annab seejärel vahendid keerukamate avaldiste moodustamiseks.

Ühesümboliline avaldis võib koosneda järgmistest osistest:

- tavaline sõmbol: **`a b c ... 0 1 2 ... + _ = : ,`**
- erisümbol: **`\. \* \[ \\ \^ \$ \( \) \|`**
- `.` (punkt) tähenduses "suvaline sümbol";
- `[`*märgid*`]` suvaline (üks) märk loetletutest;
- `[`*... algus*`-`*lõpp ...*`]` suvaline märk sellest vahemikust;
- `[^`*märgid*`]` suvaline märk, mis EI kuulu loetletute hulka.

Iteratsioon on defineeritid kui *ühesümboliline_avaldis*`*` , mis tähendab, et toda avaldist võib korrata null või rohkem korda.

Konkatenatsioon ühendab kaks avaldist (teine kirjutatakse esimese „sappa").

Avaldis võib olla rea alguses (rida algab sümboliga `^)` , keskel või lõpus (siis lõpetatakse rida sümboliga `$.` *Jaanus Pöial* toob näite ühesümbolilisest, täpselt ühest punktist koosnevast reast: `^\.$`

Kuivõrd meie raamatu ülesanne ei ole olla *UNIX*i õpik, siis jätame vahele mitmed *Jaanus Pöial*i konspekti teemad (nagu *grep* – otsi stringi etteantud failidest, *find* – otsi tingimusi rahuldavaid faile kataloogi(de)st või käsud tööks protsessidega), huvilistel on nende kohta lihtne teavet saada *UNIX*i keskkonnas olles käsuga *man* <käsk>. Lõpetagem *UNIX*i teema kahe viimase *Jaanus Pöial*i rubriigi refereerimisega. Need on valik informeerivaid käske ning valik tekstitöötluse käske (milledest mitmed on *C*-keeles hõlpsasti programmeeritavad). Nagu ka originaali autor, ei pööra me siin tähelepanu detailidele, kordame: need leiame käsu *man* abil.

Niisiis, **käske informatsiooni saamiseks:**

- `date` – kuupäev ja kellaaeg
- `cal` – kalender
- `du` – ketta täidetus (`du` *`[-k] kataloog`* ... hõivatud blokkide arv: `-k` kilobaitides, `s` – summaarne);
- `df` – vaba kettaruum, vt. ka `quota –v;`
- `who` – kes (ja kust) on masinas (loe: on serveri klient);
- `w` – kes on masinas ja mida teeb.

[1] Siinkirjutaja eelistab terminile *sõne* sünonüümi *string* (*sõne*le hakkab vastu tema subjektiivne keelevaist).

Valik **tekstitöötluskäske** on järgmine:

- `cat` – failide ühendamine;
- `grep`– ridade otsimine;
- `sed` – programmi järgi töötav redaktor;
- `sort` – ridade sorteerimine;
- `head` – faili algusosa väljastamine;
- `tail` – faili lõpuosa väljastamine;
- `diff` – erinevuste leidmine;
- `cmp` – failide võrdlemine;
- `join` – sorteeritud failide ühendamine;
- `split` – pika faili tükeldamine;
- `wc` – ridade, sõnade ja sümbolite arv;
- `tr` – sümbolikaupa teisendamine:
- `awk` – võimas tekstitöötlusvahend („me ei oska seda väidet kommenteerida“ – *J.P.* märkus).

2. *C*-keele arendamine

Reeglina pole programmeerimiskeeli disainitud ilma eeskujudeta: neist võetakse malli uue keele süntaksi kujundamisel, lisatakse neile uusi funktsionaalseid võimalusi (või vastupidi, loobutakse liigsetest võimalustest lihtsuse ja selguse huvides – tõsi, seda kohtab harvem) ning kohandatakse keelt vastavalt riistvara arengule.

Nii on see ka meie objektiks oleva *C*-keelega; järgitav on jada *Algol*-60 → *CPL* → *BCPL* → *B* → *C*. Ja *C* on omakorda olnud prototüübiks paljudele hilisematele keeltele.

2.1. CPL

See keel disainiti 1960-ndatel aastatel Inglismaal, Cambridge'i ja Londoni Ülikoolide koostööna; esialgu tõlgendati nime *CPL* kui *Cambridge Programming Language*, hiljem *Combined Programming Language.* Autoreid oli neli, eesotsas *Christopher Stracheyga*. Keel realiseeriti kahel tolleaegsel pesamasinal: *Titan*il Cambridge'is ja *Atlas*el[1] Londonis, 1963. aastal. Siinjuures toetume allikale [CPL], ja seal iseloomustatakse seda keelt kui „väikesest ja elegantsest" *Algol*ist arendatud samuti elegantset, ent suurt ja keerulist keelt. Me teame, et *Algol* töötati välja teadusarvutuste algoritmide publitseerimiseks[2]; *CPL* lisas (majandus)andmetöötluse vahendid. Seda keelt on võrreldud hilisemate *PL\I* ja *Ada*ga, neist eriti esimene oli liiga suur ja kohmakas, samamoodi arvasid kaasaegsed ka *CPL*ist. Kaasaegsete (sisuliselt väikeste) suurarvutite jaoks oli ta lihtsalt liiga suur – nõudis palju mälu ja töötas aeglaselt. Populaarne polnud ta kunagi, ja hääbus 1970-ndate alul.

Ehkki meil pole eriti materjali üldistamiseks tundub siiski, et mingi „süsteemi", olgu see siis programmeerimiskeel, operatsioonisüsteem vmt., oluline „laiendamine", nii süntaktilises kui ka funktsionaalses plaanis, kipub läbi kukkuma või paremal juhul vähemefektiivne olema. Esimeseks näiteks sobib arendus *Algol* → *CPL*: väike kompaktne keel laiendatakse suureks ja keeruliseks, aga see ei löö läbi.

Teine näide on seotav *MS-DOS*iga, mis toimis (vähemalt versioonini 3.x), ent see arendustöö lõppes 6-nda variandiga, edasi töötati juba *Windows*'iga. Programmeerija jaoks oli *MS-DOS* läbinähtav süsteem, mis ei piiranud juurdepääsu riistvarale[3]; üleminek *Windows*ile andis palju uusi (kasutaja)võimalusi, ent kitsendas oluliselt süsteemprogrammeerija omi. Lisaks: *MS-DOS*i ajal olid kasutaja võimalused stiilis „üks vajadus – üks lahendus". *Windows* pakub tavaliselt suvalise töö jaoks hulgaliselt võimalusi (nagu varajasem *IBM/OS*) – mis ainult *näib* hea. Usutavasti kõik kolleegid, kes on kirjutanud interaktiivseid programme *Windows*i keskkonna jaoks, on kogenud sama tunnet nagu nende ridade autor, kui ta otsis kümneid kordi juhtnööre *MSDN-Library*st: te saate suvalisele päringule hulgi vastuseid, ent infot nende otsatus hulgas orienteerumiseks te ei saa. Mainitud *library* on aga väga hea, kui te otsite sealt asja, mille te juba ära tegite: süda jääb rahule.

---

[1] Pöörake tähelepanu nende masinate *nimedele*, ent hoiduge liigsest üleolekutundest: me ei tea, millised masinad on meil 40 aasta pärast, ent võime üsna kindlad olla, et tarkvara pole ka siis olemuslikult (silmas pidades algoritme) oluliselt parem kui noil kaugetel 1960-ndatel. Põhjus on lihtne: algoritmiline mõtlemine (matemaatika, loogika, heuristika jne.) on lihtsalt tunduvalt vanem valdkond kui elektroonika. Esimesel on vähem arenguruumi.

[2] Neid programmeeriti aga *FORTRAN*is.

[3] *MS-DOS*i-põhised *C*-realisatsioonid võimaldavad kasutada *DOS*i ja *BIOS*i funktsioone, *UNIX*i- ja *Windows*i-põhised aga mitte, tõenäoliselt turvakaalutlustel on blokeeritud rakendusprogrammide juurdepääs arvuti seadmetele madalal tasemel.

Ent – olles kindel, et meie lugejatest on piisavalt palju neid, kes on teinud oma esimese „süsteemi“ – ja teavad „täpselt“, mis jäi realiseerimata ja mida võiks veel funktsionaalsuse mõttes lisada. Ja nad (nagu meiegi) võivad sattuda *teisese süsteemi sündroomi*[1] ohvriks: järgmisse projekti planeeritakse kõik see, mis esimesest välja jäi ja see järgmine, teine projekt kas ei saa valmis või on tulemus liiga kohmakas ja aeglane.

Niisis, tõdegem, et *CPL* ei löönud läbi, ent *Algol*iga võrreldes oli seal mitmeid uusi ideid. Inglismaa Cambridge'i ülikooli programmeerija *Martin Richards* redutseeris *CPL*ist enda jaoks olulise ning disainis keele *BCPL* (lühendi seletus on *Basic CPL*), keele suunitluseks sai *süsteemiprogrammeerimise* vajaduste[2] toetamine teadusarvutuste ja massilise (lihtsa) andmetöötluse – mida sisuldasa on majandusarvutused – arvelt.

2.2. BCPL

Joonis 2.2.a. *Martin Richards.*

Ülaloleval pildil on *Martin Richards*, kes disainis keele *BCPL* (1966, realiseeriti see *MIT*s[3] *M.R.* visiidi ajal mainitud mainekasse teaduskeskusse järgmise aasta kevadel), millest sai *C*-keele eelkjäija (vt. [Richards], [Feather], [wBCPL]). Allikas [wBCPL] iseloomustab seda keelt kui „puhast, võimsat ja portaablit“, mis mh. võimaldas kaasajal kirjutada tema jaoks nii väikest kui 16-kilobaidilist kompilaatorit[4]. Ja *BCPL* muutus seetõttu populaarseks *tarkvara mobiilsuse* probleemi (vt. näit. [Isotamm, PK lk. 196 jj.]) lahendamise vahendiks: kompilaatori esimene faas genereeris virtuaalmasina koodi ja teine transleeris selle objektmasina koodiks, ning kompilaatori kirjutamine osutus võimalikuks vähem kui poole inimaastaga[5]. Mainitakse, et selles suunas oli *BCPL* nii *Pascal*i kui ka *Java* realisatsioonide teerajajaks. Ja et *virtuaalmasina* mõiste tõi arvutiteaduse terminoloogiasse just see keel[6]. Lisagem veel, et 1979. aastal toetasid *BCPL*i 25 erineva arhitektuuriga masinat (tänapäeval me kasutame tavaliselt terminit *platvorm*). Ja aastaks 2001 oli *BCPL* jõudnud marginaalse tasemeni

[1] Seda mõistet kasutas esimesena Fred Brooks (OS/360 kirjutanud meeskonna juht) oma raamatus *The Mythical Man-Month* (vt. [wSec]).
[2] Süsteem(i)programmeerimise valdkonnale orienteeritud keelte hulgas oli *BCPL* esimeste seas; võib ka olla, et esimene; [wBCPL] sedastab, et see disainiti kirjutamaks kompilaatoreid muudest keeltest.
[3] *Massachusetts Institute of Technology. BCPL* jõudis ka meil Eestis üsna ruttu teadvusse; 1970. aastal kaitses selleteemalise diplomitöö TRÜ-s *Anne Villems*, juhendaja *Toomas Mikli* (vt [Isotamm, PK], märksõna *Mikli*).
[4] *Bootstrapping*ut kasutades, so. kood oli ise kirjutatud põhiliselt *BCPL*is; seda tehnoloogiat vt. lisa 3.
[5] [wBCPL]: kompilaatori koodist tuli vaid 1/5 ümber kirjutada uue masina koodi toetama, see võttis 2..5 inimkuud.
[6] Ja mitte ainult: [wBCPL] andmetel oli *BCPL* esimene keel, kus kirjutati klassikaline *hello-world*-programm, ning esimene *MUD – Telnet*i protokollil põhinev Internetis mängitav seiklus-, rolli- või matkimismäng (*Multi-User Dungeon* [Tavast&Hanson, lk. 141]) oli kirjutatud samuti *BCPL*is. Ning kommentaar „//...<reavahetus>“ pärineb samuti sellest keelest.

[wBCPL], so. teda märkimisväärselt enam ei kasutatud, *BCPL*i niši oli hõivanud eeskätt *C*. Allpool me püüame *BCPL*i tutvustada, ent anname endale aru, et vaevalt see tutvustus adekvaatne saab: me pole ise selles keeles ridagi kirjutanud ning me ei tee endale illusiooni, et oleksime kõikidest nüanssidest (kui üldse) aru saanud. Ent loodame, et lugejad ei ootagi meilt õpetust, kuidas hääbunud keeles koodi kirjutada, vaid – nagu nende ridade autorgi – üritavad nad ära tunda meie *C*-keele „juuri".

M. Richards iseloomustab BCPLi kui *lihtsat* ja *tüübivaba* (*simple typeless*) keelt [Richards]. *C. Feather* [Feather, lk. 1] ütleb selle kohta, et selles keeles määratakse operandide tüübid[1] operaatoritega (tüübideklaratsioonide asemel): iga objekt võib evida suvalist tüüpi ning konkreetse, jooksva tüübi määrab see, kuidas seda objekti manipuleerib järjekordne operaator. Seega: *BCPL* ei tegele tüübikontrollidega[2]. Samas, üks ja seesama objekt on alati *kas* operand *või* viit, aga mitte kunagi üks või teine vaheldumisi.

Ilmutatud kujul tüüpide puudumine oli potentsiaalne (näpu)vigade allikas, ning programmeerijate aitamiseks „leiutati" nn. *ungari notatsioon* (*Hungarian Notation*), mis on tänini tuntud tänu *Microsoft*ile (allpool tugineme allikale [Simonyi]): kokkuleppeliselt kasutatakse erinevate objektide (muutujad, protseduurid, funktsioonid) tüübile viitavate nimede valikul *prefikseid*[3], näiteks *sznimi* annab teada, et identifikaator *nimi* on seotud stringiga (***s***), mille lõputunnus on '\0' (***z***ero). Miks *ungari*: esiteks, selle „nipi" mõtles välja ungarlane, teiseks: ungari keeles on üsna tavaline, et sõnad algavadki taoliste prefiksitega (näiteks, perenimi *Szabo*).

BCPL-programmi *mälu* koosneb *väljadest*[4]. *Dennis Ritchie* [Ritchie, lk. 5] märgib, et *BCPL* on pigem ühe-tüübi-keel kui tüübitu: selleks tüübiks ongi väli – fikseeritud pikkusega bitijada. Mälu on sellistest väljadest koosnev vektor ning välja semantika sõltub operaatorist (näiteks, „**+"** liidab kaks bitivälja nagu *int*-arvud), ja „**!"** tähendab, et väljal pole mitte operand ise, vaid *viit* sellele operandile. Ja kui *p* on mingi välja aadress, siis *p*+1 on järgmise välja aadress.

Kui *BCPL*is on kirjutatud

```
LET V=vec 10
```

siis selle vektori *i*-ndat elementi (meile rohkem tuntud keeltes *V*[*i*]) osundatakse kui `V!i`.

Allpool – kui pole teisiti öeldud – tugineme *Clive D. W. Feather*i[5] [Feather] kontsentreeritud ülevaatele.

BCPLi elementaarkonstruktsioonid on järgmised:

---

[1] Võimalikud tüübid on *int*- ja *float*-tüüpi arvud, bitijada, aadress (so, viit), jmt. *C. Feather* [Feather, lk. 2] märgib, et tüübiteisendused paneb paika keele realiseerija.

[2] Teoreetilise arvutiteaduse populaarne diskussiooniteema *tüübiteisendused* näib mõneti „kunstlikuna"; õigluse huvides nentigem, et selle probleemi algallikaks võisid olla just *BCPL*i-taolised *tüübivabad* keeled.

[3] *Jüri Kiho* juhtis tähelepanu seigale, et muutuja nime esitäht määras vaikimisi tüübi juba *FORTRAN*is: *I, J,..,N*-algusega muutujate tüüp oli *INTEGER* ja kõigil ülejäänutel – *REAL*.

[4] Originaalis on kasutatud terminit *cell* (pesa), ent me oleme harjunud *pesa* samastama „pesamasinate" *sõnaga*. *BCPL*i *väli* **võib** olla sama pikk kui *masinasõna*, ent ei pruugi. Selline keele kirjelduse tasemel toimiv nääramatus tuli kasuks *BCPL*i realiseerimisele mistahes pesa pikkusega pesamasinatel või – hiljem – baitmasinatel. Pesa- ja baitmasinate kontseptuaalse vastuolu ületamiseks kasutati sõnade *pakkimist* (ja lahti*pakkimist*) bitistringideks.

[5] Ta oli *Cambridge*'i Ülikooli tudengina *M. Richards*i õpilane, ja kirjutas pärast lõpetamist *Z80-keskkonnas* süsteemiprogramme aastaid *BCPL*is, kuni tulid *UNIX* ja *C*. Siis õppis ta ümber.

- **identifikaatorid** on üsna harjumuspärased (algavad tähega, võivad sisaldada tähti, numbreid ja punkti, pikkus pole piiratud);
- **konstandid** on vaikimisi kümnendarvud; prefiks '#' määrab kaheksand- ja #x kuueteistkümnendarvu. Prefiks '**?**' tähendab, et konstandi tüüp selgub lahendamise ajal. Ühesümboliline konstant on apostroofide vahel ('c') ning string[1] – märgipaaride „ ja " vahel ning stringi pikkus on kuni 256 sümbolit. Tõeväärtused *TRUE* ja *FALSE* on reserveeritud sõnad.
- **paojadad** [Tavast&Hanson, lk. 80)], *escape sequence* [CD, lk. 133][2]: need on kujul *tärn-ja-sümbol*, näiteks `*N` on *C*-keeles `\n` või `*T` on `\t`;
- **operatsioonid** täidetakse (paari erandiga: -> ja *TABLE*) „vasakult paremale" ja avaldistes saab nende prioriteetidega määratud täitmise järjekorda muuta tavapäraste ümarsulgudega. Kõrgeim prioriteet on funktsiooni väljakutsel. Prioriteetide kahanemise järjekorras on operatsioonid[3] (kommentaarid on meie poolt):
  (binaarne) **! ,** %, // *a!b ≡ !(a+b)*; *„abc"%0=3, „abc"%1='a', „abc"%3='c'*
  @, (unaarne) ! // *!a on aadressiga a välja sisu; @!a ≡ a, @a!b ≡ (a+b)*
  * , /, REM #* #/ //*integer-arvude *, / ja jagatise jäägi leidmine; #*, #/ : ujupunkt*
  + , –, #+ #– //*integer- ja ujupunktarvude + ja – ;*
  =, ~=, <, <=, >, >= #= #~= #< #<= #> #>= //*int- ja ujupunktarvude võrdlemine*
  <<, >> // *bitiväljade nihutamine vasakule ja paremale*
  ~ //*loogiline eitus: ~a*
  & //*a&b loogiline korrutamine, konjunktsioon*
  | // *a|b loogiline liitmine, disjunktsioon*
  EQV, NEQV //*NEQV on loogiline mittesamaväärsus, a EQV b ≡ ~(a NEQV b)*
  -> *a->b,c // kui a=TRUE, siis b, muidu c (a, b ja c on operaatorid)*
  TABLE //*T=TABLE k_1,..k_n, kus iga k_i on konstantavaldis[4]; T on n üksteise järel initsialisieeritavate vektorite (pikkusega k_i) aadress*
  VALOF //*madalaim prioriteet, analoog ≈ C-keele return( )*
  SLCT //*selektor: (SLCT x:y)of b eraldab bitijada väljalt !b;kui x=0, siis on see samaväärne operatsiooniga (!b>>y).*
- **protseduuride väljakutsed** on kujul *e1( )* või *e1(e2, e3, ..., ex).* Parameetrid edastatakse *ainult* väärtuse järgi, mis välistab olukorra, kus alamprogramm võib ülemise taseme programmi mäluseisu rikkuda (so. välistab *kõrvalefekti*);

BCPL-programmi struktuuri ning juhtimisoperaatoreid iseloomustavad järgmised mõisted:

- **plokid** koosnevad nullist või rohkemast *deklaratsioonist*, millele järgneb üks või rohkem *käsku* (kõik on *sektsioonisulgude* vahel); käske eraldavad semikoolonid;
- **sektsioonid** on suvalised sektsioonisulgude vahel paiknevad konstruktsioonid. Mainitud sulud moodustavad paari *$(* ja *$)*[5], kusjuures neid *võib* varustada paarikaupa sama

---

[1] *BCPL*i *string* (*sõne*) on vektor, mille 0-nda elemendi väärtust interpreteeritakse stringi pikkusena (ja lõputunnust ei ole). Tekst paigutatakse algselt „üks sümbol ühele väljale", seejärel pakitakse kompaktsesse vektorisse (kaks või rohkem sümbolit väljale) [Ritchie, lk. 6]. Välja sümbolite arvu määrab globaalne muutuja *BYTESPERWORD* (baitide arv sõnas). Ja *BITESPERBYTE* määrab baidi bittide arvu.

[2] Vahekeele tasemel algavad need *Escape*-koodiga (27_{10}), millele järgneb üks või rohkem sümbolit; esitavad käsku kas riistvara või mingi madala taseme programmi juhtimiseks.

[3] Allikas [Feather] ei ava kõikide operatsioonide semantikat, ja seda ei püüa meiegi teha, kuivõrd käesolevas pole meie eesmärgiks olla *BCPL*i manuaal. Huvilised võivad vajalikud materjalid nagu ka kompilaatori alla laadida keele autori saidilt [Richards].

[4] Konstantavaldise kõik operandid on konstandid, näiteks 3*(2+7/3), võrdle *x*(2+*y*/3).

[5] [wBCPL] väidab, et *BCPL* oli esimene keel, mis kasutas operaatorsulupaari „{...}" – see uuendus tehti keele järgmises versioonis, kuivõrd sisend-väljundseadmete märgistik oli harjumuspärasest kirjutusmasina- ja teletai-

„lipikuga“ (*tag*), mille moodustamise reeglid on peaaegu samad nagu identifikaatoritel, ent nad võivad alata ka numbriga. Lipikud annavad mugava võimaluse sulgeda korraga kõik madalama taseme sektsioonid (vt. [Feather, lk. 7], lk. 13):

```
$(1
   mingi kood
   $(1,1
   $(1,2
      mingi kood
      $(
            mingi kood
$)1 //lõpetab selle sektsiooni kõik ilmutatud kujul lõpe-
      //tamata sektsioonid.
```

- **käsud** (*commands*)[1]. Käsk on kas protseduuri väljakutse (*call*), omistamine[2], plokk või mingi järgnevatest (e_i on avaldis ja c_i on käsk)[3] [Feather, lk. 7]:

```
c REPEAT     lõputu tsükkel välja saab BREAK- ja GOTO-käskudega
c REPEATWHILE e  see ja järgmine täidetakse vähemalt üks kord
c REPEATUNTIL e
IF e THEN c
UNLESS e DO c
TEST e THEN c1 ELSE c2
WHILE e DO c
UNTIL e DO c
FOR i = e1 TO e2 BY e3 DO c  e3 peab olema konstantavaldis[4]
RESULTIS e
SWITCHON e INTO c
      harud on blokid, mis ei tohi sisaldada deklaratsioone
GOTO e
FINISH  katkestab programmi täitmise
RETURN  lõpetab täidetava protseduuri
BREAK   katkestab „ja“
LOOP    lõpetab iteratsiooni
ENDCASE
```

- **deklaratsioonid** (*declarations*). *BCPL*-programm on semikoolonitega eraldatud deklaratsioonide jada, iga neist deklareerib ühe (või rohkem) identifikaatori(t), ning nad on kas *sektsiooni deklaratsioonid* (*section declaration*) või *simultaandeklaratsioonid* (*simultaneous declaration*).
    1. Sektsiooni deklaratsiooni identifikaatorite kehtivuspiirkond algab niipea, kui need identifikaatorid on deklareeritud; arvatavasti võime seda varianti pidada *Algol*i hierarhilise plokkstruktuuri analoogiks; piirkond suletakse plokist väljudes. Sektsiooni deklaratsioonidel on kolm varianti (identifikaator v_i on deklareeritud kõrgemal tasemel ja k_i on konstantavaldis):
        - MANIFEST \$(v_i=k_i; ... v_n=k_n \$) deklareerib <u>konstandid</u> v_i väärtustega k_i;
        - STATIC \$(v_i=k_i; ... v_n=k_n \$) teeb ja väärtsustab iga <u>muutuja</u> v_i jaoks välja;
        - GLOBAL \$(v_i : k_i; ... v_n : k_n \$) deklareerib globaalse vektori, mis on kättesaadav kõikides plokkides kogu programmi töötamise ajal[1].

---

bivõimalustest jõudsasti edasi arenenud. Meil ei õnnestunud tuvastada, kas *B* mitte ette ei jõudnud, ent see polegi tähtis.

[1] Me ei „kirjuta nende semantikat lahti“, lootes, et lugejal on juba programmeerimiskogemusi mitmes keeles ja oskus otsida internetist lisateavet.

[2] Üks variant on $a_1,a_2,..a_n := e_1,e_2,..e_n$ ($a_1:=e_1,...a_n:=e_n$).

[3] Kursiivis esitatud kommentaarid on meilt, käskude kirjeldused on kopeeritud *Feather*i artiklist.

[4] Kui fakultatiivne komponent *BY e3* puudub, siis vaikimisi *e3*=1

2. Simultaanne deklaratsioon defineerib *ühise* kehtivuspiirkonna (nimede või teisisõnu viitade keskkonna) kõigile plokki lülitatud deklaratsioonidele ning see piirkond kaotab kehtivuse ploki lõppedes. Meile tundub, et simultaansete deklaratsioonide kehtivuspiirkondade mehhanism on ühe teise *Algol*-põhise keele *Simula kaasprogrammide* mehhanismi analoog. Formaalselt näeb selline deklaratsioon nii välja (d_i on *i*-s deklaratsioon):

   LET d_1 AND d_2 AND...AND d_n .

   Siin on neli võimalust[2]:

   1. LET v_1, v_2, ..., v_n = e_1, e_2, ...,e_n Dünaamiline; iga identifikaator v_i saab algväärtuseks avldise e_i väärtuse (avaldis ei pea olema konstantavaldis);
   2. LET v=VEC k Identifikaatoriga *v* seotakse dünaamilise vektori esimene väli aadressiga *k*, järgnevad väljad on kättesaadavad kujul *k*+1,... Need kaks kuju on kasutatavad ainult ploki piires
   3. LET v(v_1, v_2, ..., v_n) BE *käsk* – nii „luuakse“ alamprogramm (*routine*) ilma ilmutatud kujul oleva resultaadita (iga v_i on formaalne parameeter);
   4. LET v(v_1, v_2, ..., v_n)=e kirjeldab funktsiooni (e on avaldis ja iga v_i on formaalne parameeter), näiteks: LET v(v_1, v_2, ..., v_n)=VALOF $(*käsk*; RESULTIS ? $).

Meenutagem, et *BCPL*i eelkäija *Algol*-60 ei tunnistanud ei eraldikomplileeritavaid alamprogramme (mis on *FORTRAN*i jätkuva „vitaalsuse“ peamiseks mootoriks) ega ka (standard)-funktsioonide teeke, kõik keelde sisse ehitatud funktsioonide algoritmid olid kättesaadavad keele kirjeldusest, ning „kasutaja-alamprogrammide“ algoritmid olid esitatud programmi terviktekstis; me teame, miks see nii oli – *Algol* oli mõeldud *algoritmide publitseerimise* keeleks.

*BCPL*i kogu funktsionaalsus pole seevastu keelde sisse kirjutatud: paljud asjad on jäetud keele suhtes välise, realisatsioonist sõltuva *süsteemiprogrammide teegi* (*system library*) lahendada. Eeskätt puudutab see sisendit ja väljundit (meenutagem, et *FORTRAN*is oli see haavatavaim koht: kirjutamine väsitav ning kompileerimise asemel lahendamise ajal teksti tasemel ma-sinast-sõltuvalt interpreteeritav, ja et *Algol*i standardis puudusid nood vahendid sootuks), samuti objektmasinale häälestamist jm (vt. [Isotamm, PK; lk. 108 – 126]).

Nii näiteks teegis *LIBHDR* (*LIBrary HeaDeR*) olid objektmasinast-sõltuvad konstandid *BITESPERBYTE* ja *BYTESPERWORD* või *I-O*-le orienteeritud teegis oli *C*-programmeerijatele üsna äratuntav trükifunktsioon *WRITEF*(*format*, *args*), kusjuures *format* on string, mis võib sisaldada „võtmeid“

- %% (väljastatakse ’%’);
- %S (string);
- %C (sümbol);
- %O1 (1-kohaline kaheksandarv; O7: seitsmekohaline – vihje formaaditud väljundile, sama kehtib ka järgmiste arv-formaatide puhul);
- %X1 (kuueteistkümnendarv);

---

[1] Globaalne väljade-vektor on tegelikult sootuks võimsam vahend, mis on võrreldav globaalse ühisväljaga eraldi transleeritud ja kokkukomplekteeritud programmide jaoks (vrd. *COMMON*i rolliga *FORTRAN*is [Isotamm, PK; lk. 108 jj.]) [wBCPL, lk. 2].

[2] Kasutame võtmesõna *LET* (nagu ongi esimeses deklaratsioonis), ent järgnevates kasutatakse võtit *AND*.

- %I1 (kümnendarv);
- %N1 (*integer*).

Lõpuks, paar omapärast seika:

- rea lõpuni kehtiv kommentaar võib alata sümbolitega //, || või \\:
- mitmerealise kommentaari sulud on kas paar /*..*/, |*..*| või * ..*\;
- kui rida peaks lõppema semikooloniga, ent seda pole, siis translaator lisab selle reavahetussümboli ette;
- võtmesõnad *THEN* ja *DO* on asendatavad sõnadega *OR* ja *ELSE*.

Lõpetame *BCPL*-keele lühitutvustuse *M. Richards*i koduleheküljelt [Richards] saadud[1] näiteprogrammiga, mis genereerib vektoreid ning järjestab igaühe neist *Shell*i meetodiga mittekahanevasse järjekorda. Me ei riski seda programmi kommenteerida, kuivõrd pole selles keeles midagi programmeerinud, ja suvalise keele *lühikirjeldus* ei anna kunagi kindlustunnet, et tollest keelest on adekvaatselt aru saadud. Seega: järgnev on illustratiivse iseloomuga.

```
SECTION "sort"
GET "libhdr"
LET shellsort(v, upb) BE
$( LET m = ?
   LET mtab = TABLE
      1,     2,     3,     4,     6,     8,     9,    12,    16,    18,
     24,    27,    32,    36,    48,    54,    64,    72,    81,    96,
    108,   128,   144,   162,   192,   216,   243,   256,   288,   324,
    384,   432,   486,   512,   576,   648,   729,   768,   864,   972,
   1024,  1152,  1296,  1458,  1536,  1728,  1944,  2048,  2187,  2304,
   2592,  2916,  3072,  3456,  3888,  4096,  4374,  4608,  5184,  5832,
   6144,  6561,  6912,  7776,  8192,  8748,  9216, 10368, 11664, 12288,
  13122, 13824, 15552, 16384, 17496, 18432, 19683, 20736, 23328, 24576,
  26244, 27648, 31104, 32768, 34992, 36864, 39366, 41472, 46656, 49152,
  52488, 55296, 59049, 62208, 65536, 69984, 73728, 78732, 82944, 93312

   UNTIL !mtab>upb DO mtab := mtab+1
   $( LET k = 0
      mtab := mtab-1
      m := !mtab
      FOR i = m+1 TO upb DO
      $( LET j = i-m
         LET vi, vj = v!i, v!j
         IF vj>vi DO v!j, v!i, k := vi, vj, k+1
      $)
      writef("m = %i4  swaps = %i4*n", m, k)
   $) REPEATUNTIL m=1
$)

MANIFEST $( upb = 5000  $)

LET start( ) BE
$( LET v = getvec(upb)

   try("shell23", shellsort, v, upb)

   writes("*nEnd of test*n")
   freevec(v)
$)
```

[1] Kasutades viita *bcpl.zip*-i allalaadimiseks ja järgnevat lahtipakkimist.

```
AND try(name, sortroutine, v, upb) BE
$( writef("*nSetting %n words of data for %s sort*n", upb, name)
   FOR i = 1 TO upb DO v!i := randno(10000)
   writef("Entering %s sort routine*n", name)
   sortroutine(v, upb)
   writes("Sorting complete*n")
   TEST sorted(v, upb)
   THEN writes("The data is now sorted*n")
   ELSE writef("### ERROR: %s sort does not work*n", name)
$)
AND sorted(v, n) = VALOF
$( FOR i = 1 TO n-1 UNLESS v!i<=v!(i+1) RESULTIS FALSE
   RESULTIS TRUE
$)
```

Ja päris lõpuks toome ära *Martin Richards*i programmi [wBCPL, lk. 3], mis „trükib“ arvu 5 faktoriaali arvutamise (nii vahe- kui ka lõpp)tulemused:

```
GET "libhdr"

LET start( ) = VALOF
$( FOR i = 1 TO 5 DO writef("fact(%n) = %i4*n", i, fact(i))
  RESULTIS 0
$)

AND fact(n) = n=0 -> 1, n*fact(n-1)
```

2.3. B

Allpool tugineme allikate [B-keel] materjalidele.

Eric Lebherz[1] [Lebherz] iseloomustab seda keelt lühidalt nii: „*B*-keel on disainitud süsteemprogrammeerimise rakenduste jaoks, pole kavandatud arvutustöödeks ja on rekursiivne. See on operatsioonisüsteemi *UNIX* arvuti *PDP*-11 jaoks kirjutatud keel. See keel on väga sarnane oma eelkäija *BCPL*iga – ent mitte süntaksi poolest. *B* oli kasutusel *Honeywell*i operatsioonisüsteemi *GCOS-3* jaoks, ning oli *C*-keele prototüübiks.“

Allikas [wB] tõdeb, et *B*-keele lõi (*designed by*) 1969. aastal *Ken Thompson* ja seda arendasid (*developed by*) *Ken Thompson ja Dennis Ritchie. Ken* lähtus *BCPL*ist ning – olles piiratud kaasaegsete miniarvutite ressurssidega[2] – jättis sellest keelest kõrvale kõik asjad, mida sai eemaldada keele jaoks olulist funktsionaalsust kaotamata.

Jätkamegi [wB] refereerimist. Märgitakse, et nagu *BCPL* ja *FORTH* (vt. näit. [Isotamm, PK], lk. 139 jj.), on ka *B* „ühe-tüübi-keel“, ja see tüüp on – kui jääme eelmise jaotise terminoloogia juurde – *väli* (ent silmnähtavalt masinasõna[3]-pikkune). Olenevalt kasutamise semantikast interpreteeritakse välju kas *integer*-tüüpi operandidena või *viitadena* (so, operandide või andmestruktuuride aadressidena). Lisaks võimaldab keel opereerida biti- ja sümbolistringidega, kuid mitte ujupunktarvudega.

[1] Tegemist on mehega, kes sai hakkama võimatuna näiva tööga: ta koostas enimtuntud (või -kasutatud) programmeerimiskeelte lühitutvustustest koosneva ülevaate.
[2] Peetakse silmas eeskätt operatiivmälu mahtu. *BCPL* oli mõeldud oluliselt võimsama(te) *meinfreim*-masina(te) jaoks; *Thompson*i kasutada oli 8K baiti *PDP*-7 mälu (vt. [Ritchie], lk. 3).
[3] Raamatus [Isotamm, PK] kasutasime *masinasõna* vastena terminit *pesa.*

Samuti nagu *BCPL*is on *B*-keeleski paljud (varasemates keeltes „sisse kirjutatud“) funktsionaalsused lahendatud keelevälise funktsioonide teegi abil – mis, peame nentima, on väga paindlik, võimalusterohke ja leidlik lahendusvariant. Tõsi, *B* piirdus peamiselt sisend-väljundfunktsioonidega[1]. Esialgse variandi tollest teegist kirjutas *S. C. Johnson.*

Ehkki *Eric Lebherz* seostas ülalpool *B*-keelt *PDP*-11-ga, realiseeriti ta esmalt arvutil *PDP*-7 (vt. [wThread]), kus objektkeeleks oli „punutud kood“[2] (originaalis *threaded code*).

Punutud kood on termin translaatorite programmeerimise valdkonnast: see tähistab tehnikat, kus kogu genereeritud kood koosneb eranditult alamprogrammide väljakutsetest; selle käsitlemine ei paku probleeme ei interpretaatorile ega ka kompilaatorile. Punutud kood on parimini tuntud kui Forthi realiseerimise vahend, ent B kõrval on seda tehnikat kasutatud ka mõnedes BASICu ja COBOLi realisatsioonides. Eriti populaarne oli see tehnika väikeste miniarvutite (tänapäeva pilguga vaadates mõõtmetelt väga suurte arvutite, selliste nagu PDP-mudelid või N. Liidu CM-seeria omad) ajastul [wThread].

D. Ritchie kirjutas *B*-kompilaatori ümber, otse *PDP*-masinkoodi; ikka veel mudeli „7“ jaoks. 1971. aastal tõi ta keelde *andmetüübi* mõiste (seni oli *B* „tüübivaba“: väli on bitijada, mida interpreteeritakse vastavalt kontekstile); lokaalse muutuja „vaba tüüp“ tuli deklareerida, võtmesõnaks oli *auto*. Globaalsed muutujad deklareeriti, kasutades võtmesõna *extrn*[3] (*external*) ning kolmandaks muutujate deklareerimise võimaluseks oli nende kasutamine funktsiooni argumentidena. Lokaalsete muutujate „elutsükkel“ on laenatud *Algol*ilt: nad luuakse alamprogrammi sisenedes ja „hävitatakse“ väljudes.

B-keelse programmi tekst koosneb *main*-moodulist (selle esimesest direktiivist algab programmi täitmine) ning – vajadusel – alamprogrammide kirjeldustest. Üldine struktuur on järgmine [Kernighan[4], lk.2]:

```
main( ) {
-- statements --
}

newfunc(arg1, arg2) {
-- statements --
}

fun3(arg) {
-- more statements --
}
```

Alamprogrammideta lihtnäite laename *B. Kernighan*ilt [Johnson&Kernighan]:

```
main( ){
      auto a,b,c,sum;
      a=1; b=2; c=3;
```

[1] Meenutagem, ka *FORTRAN* jättis nood operaatorid kui riistvaradraiveritest sõltuvad virtuaalmasina interpreteerida. Nii tagati keele *portatiivsus* (lihtne realiseeritavus suvalise arvutimudeli jaoks).
[2] Seda terminit kasutas Eesti *Forth*i seltskond (vt. [Isotamm, PK], lk. 139 jj.).
[3] [Kernighan, lk. 5] nendib, et *extrn*'i semantika on lähedane *FORTRAN*i ühisvälja (*COMMON*) omale.
[4] Kuivõrd *Brian W. Kernighan*ist on selles raamatus mitmes kohas juttu (sj. on ta [K&R]-raamatu kaasautor), siis tuleb teda põgusalt tutvustada. Ta on 1942. a. sündinud kanadalane, allika [wKernighan] järgi ei osalenud ta *C*-keele disainimisel ega realiseerimisel; tal on teeneid heuristilise programmeerimise ülesannete programmeerimisel, *UNIX*i ja *C* meeskondades ning väitluses *N. Wirth*iga (kes ründas aktiivselt *C*-keelt ja selle meeskonda). Vt. [wKernighan] http://en.wikipedia.org/wiki/Brian_Kernighan

```
        sum=a+b+c;
        putnumb(sum);
}
```

Märgitakse, et enamik *B* operaatoreid (*statement*) sisaldavad avaldisi (*expression*). Keel kasutab vaba formaati ning hea stiil eeldab treppimist. Nimede pikkuse ülempiir on 8 sümbolit; keel on tõstutundlik.

1970. aastal kasutati uue mudeli, *PDP*-11 jaoks punutud koodi üksnes välisseadmetega seonduva jaoks. Teatavasti[1] oli *PDP* mudel „11“ kvalitatiivselt erinev nii sama firma eelmistest mudelitest kui ka kõigist muudest tolleaegsetest masinatest; sisuliselt oli ta mikroprotsessorite loogilise arhitektuuri sissejuhataja. Ning *B*-keel ei suutnud uusi aparatuurseid võimalusi piisavalt ära kasutada. 1972. aastal kirjutas *Ritchie* keele *New B*, ja seejärel juba *C* (koos preprotsessoriga *B*-programmide transleerimiseks); aastatel 1972 – 73 kirjutas *Mike Lesk* „sisend-väljund-paketi“ (*portable I/O-package*), millest arendati välja *C*-keele *stdio*-teek.

2.4. C

Dennis Ritchie kirjutab [Ritchie, lk. 3], et nii *BCPL, B* kui ka *C* on protseduursed keeled nagu *Fortran* või *Algol*-60, nad võimaldavad süsteemprogrammeerimist[2], on väikesed ning kompaktselt kirjeldatavad ning hõlpsasti realiseeritavad väikeste lihtsate kompilaatoritega. Need keeled on „masinalähedased“, olles orienteeritud aparatuurselt interpreteeritavatele andmetüüpidele ja operaatoritele, ning jätavad sisend-väljundoperatsioonid ja suhtlemise operatsioonisüsteemiga keeleväliste alamprogrammide teekide hooleks. Nii tagatakse nende keelte mobiilsus (e. portatiivsus, so. lihtne realiseeritavus erinevatel platvormidel): keele *tuum* on hõlpsasti realiseeritav antud masina assembleris, kui see on tehtud, siis on lihtne realiseerida kogu keel toda tuuma kasutades.

Kui peame silmas *C*-keelt, siis uue platvormi jaoks oli hõlpus realiseerida *UNIX*, kasutades *C*-kompilaatorit. Standardfunktsioonide teegid tuli tavaliselt iga platvormi jaoks „kohendada“ – aga seda juba *C*-keeles, kuivõrd nad on üpris tuntavalt masinorienteeritud.

BCPL, B ja *C* on süntaktiliselt üsnagi erinevad, kuid sisuliselt sarnased; programmid koosnevad globaalsete deklaratsioonide ja alamprogrammide – funktsioonide või protseduuride – jadadest. *BCPL* lubas sarnaselt *Algol*ile alamprogrammide puid (*nested procedures*)[3], ent *B* ja *C* enam mitte. *Ritchie* nendib, et *BCPL* on süntaktiliselt ja leksikaliselt paljuski elegantsem ja regulaarsem kui *B* või *C*, ent sellega kaasnesid *BCPL*il mõningad komplikatsioonid (vt. täpsemalt [Ritchie, lk. 4]), mida püüti vältida. Näiteks loobus *B Algol*-stiilsest plokkstruktuurist, „globaalsest vektorist“ eraldikompileeritud alamprogrammide jaoks (kasutati „konventsionaalset komplekteerijat“ nagu assembleris või *Fortran*is); lisaks tehti mõningaid muudatusi leksikas, näiteks *Algol*i omistamine „**:=**“ asendati „algebralise“ „=“-ga ning rea lõpuni toimiv kommentaarimarker „//“ asendati pikemaid kommentaare võimaldava markeritepaariga „**/* ... */**“ (*C* võttis selle koha tagasi).

---

[1] Raamatu [Eckhouse&Morris] tõlke eessõnas kirjutas toimetaja *G. Vassiljev*, et „miniarvuti *PDP-11* kajastab arvutustehnika arengu väga tähtsaid tendentse“. Ja, et raamatus pakutava pragmaatilise teabe kõrval tuleks üle vaadata infotehnoloogia arengutendentsid üldisemas plaanis.

[2] *Fortran* ja *Algol* siiski teatavate reservatsioonidega – autori märkus.

[3] „Puu“ tähendab siin olukorda, kus alamprogrammi kirjelduses võis kirjeldada madalama taseme alamprogramme. *Algol*is tekitati nii plokkstruktuur ning rohkem kui kahetasemeline viitade keskkond.

Ritchie ([Ritchie, lk. 5]) nendib, et *BCPL*i esmaversioon (publitseeritud 1967. a.) oli *B* prototüüp, ent teine versioon (1979) oli juba mõneti mõjutatud ka *Thompson*i *B*-st. Samas rõhutatakse: hoolimata süntaktilistest erinevustest on *BCPL* ja *B* kontseptuaalselt lähedased keeled. Mõlemad on „tüübitud“ (*typeless* – kui mitte tüübiks pidada *välja* – fikseeritud pikkusega mälulõiku, millel salvestatud bitte saab mitmeti interpreteerida), mälu on nonde *väljade* vektor, *viit* on loomulik andmetüüp (ja viitadega saab sooritada aritmeetikatehteid) ning *string* pole mitte midagi „erilist“, vaid on baitvektor[1], mille elementide väärtusi võib mh. interpreteerida kui sümbolite (*ASCII*-) koode. *BCPL*i string oli pikkusatribuudiga (alguses), *B* kasutas lõpumarkerit („null-baiti“ '\0'), selle võttis üle *C*.

Ritchie rõhutab, et *B* minimalism polnud kontseptuaalne valik, vaid oli tingitud *PDP*-7 pisikesest operatiivmälu mahust[2]. *K. Thompson* kirjutas oma kompilaatori esimese versiooni *bootstrapping*ut (vt. lisa 3) kasutades ümber ja hoidis niimoodi pisut mälu kokku. Ning samal eesmärgil evitati ökonoomsemaid süntaktilisi vahendeid, näiteks $x=x+y$ asemel $x+=y$ (võttes eeskujuks *Algol*-68).

Pre- ja *postfiks-autoincrement* ja *-decrement*-režiimid (á la $i{+}{+}$ ja $-{-}j$) kuuluvad samasse ritta, kusjuures *Ritchie* [Ritchie, lk. 6] juhib tähelepanu tihti tehtavale eksitusele, mille kohaselt *B*- (ja *C*-)keele nood režiimid olevat pärit *PDP*-11 masinkoodist (kus nad on realiseeritud). Tegelikult, kui *B*-d tehti, polnud *PDP*-11 veel olemas, ja *Thompson*i objektmasinal *PDP*-7 neid režiime veel ei olnud, oli vaid mingi väike piirkond ++-adresseeritavat mälu – aga mis ikkagi inspireeris *Thompson*it[3].

Ning *B*-kompilaator ei genereerinud masinkoodi, vaid *punutud koodi*, mida on elementaarne interpreteerida lihtsa *LIFO*-magasini kasutava algoritmiga.

Ritchie (samas, lk. 7) kirjutab, et tema suurim saavutus noist aegadest on see, et ta kirjutas kompilaatori *B*-keelest *GE*-635 (36-bitise *meinfreim*-masina) masinkoodi (ja mitte punutud koodi) *B*-keeles, mis jooksis 18-bitisel masinal (*PDP*-7) 4-kilosõnase kasutajale kättesaadava operatiivmäluga.

Ritchie nendib, et tolleaegsed suured keeled (*Fortran*, *PL\I*, *Algol*-68) tundusid lootusetult ressursimahukatena, et nende abil realiseerida *UNIX*it, ent neist ammutati ideid. Aastal 1970 ootasid *Thompson* ja *Ritchie* oma laborisse uut *PDP*-11 masinat (16-bitine, ikka veel pesamasin, operatiivmälu 24 KB), ooteajal kirjutas *Thompson Unix*i (mahus 12 KB) uuele masinale ümber – testimiseks, ja mitte reaalseks tööks. 1971. aastal olid mainitud mehed kirjutanud mitmeid standardprogramme, mis leidsid oma firma kolleegide hulgas kasutajaid, ning *B* ja *UNIX* hakkasid populaarsust koguma. Ja ehkki ka see (mini)masin oli jätkuvalt *pesa*masin (ehkki muutuva pikkusega käsuformaatidega), hakkasid *Thompson* ja *Ritchie* üha rohkem tähelepanu pöörama *bait*masinatele[4].

Ritchie tõdeb, et *B*-keele elujõulisuse säilitamiseks oli hädavajalik lisada *tüübid* (eestkätt *uju-punktarvu*-tüüp), viit *baidile* (tüüp „sümbol“) ja midagi veel, ning sellega seoses tuli teha

[1] Põhimõtteliselt, mitte füüsilise realisatsiooni poolest, meenutagem, et kaasajal olid mõlemad keeled, *BCPL* ja *B*, orienteeritud mitte bait- vaid pesamasinatele. Viimastes pakiti sümbolid pesadesse kokku ning sümbolite adresseerimine oli „tülikas“ nii protsessorile kui ka programmeerijale (vt. näit. [Isotamm, PK] lk. 56).
[2] *Thompson* sai kasutada vaid 8*KB* mälu (kilo*baiti*, mitte sõna!), vt. [Ritchie, lk. 7].
[3] Ja siingi: $i{+}{+}$ „võtab vähem ruumi“ kui $i=i+1$.
[4] *Pesa*- ja *bait*masinatest loe näit. [Isotammmm PK] lk. 185 jj.

keele uus versioon – selle tegi *Ritchie* ise ning selle uue versiooni nimeks sai *C* (esialgu *New B* või *NB*). Algversioonis leidusid juba tüübid *int* ja *char* ning ilmutamata viidatüüp, mis seondus vektoriga/massiiviga. Kusjuures algusest peale on *C* olnud üheselt orienteeritud baitmasinale, so. igal baidil on oma unikaalne aadress – mis seondub *viidatüübiga.* Viimast ilmutatud kujul *C*-s pole, ent see-eest on suhteliselt keeruline mehhanism interpreteerimaks vektoreid ja massiive ning võimalused manipuleerimiseks tegelike (protsessori-)aadressidega, kasutades märke '*' ja '&'[1].

C-keele adresseerimisprobleem on pärit juba *FORTRAN*ist, kus muutuja nimi (mis transleeritakse aadressiks) on kahemõtteline: kord on ta tõepoolest mäluaadress, kord aga sellel aadressil olev „arv" (kuidas teda ka ei interpreteeritaks). *C*-keeles mõeldi välja mõisted *lvalue* (*left value*) ja *rvalue* (*right value*); vasak- ja parempoolne „väärtus". „Poolsuse" määras nime positsioon omistamisoperaatoris: nimi on kas selle operaatori vasakus või paremas pooles, näiteks omistamisoperaatoris `B=A+1`; on `B` *lvalue*, so. `B` interpreteeritakse aadressina ning `A` on *rvalue*, so. teda interpreteeritakse kui bitijada aadressilt `A`[2]. Aga *rvalue* rollis võib mõnedes konstruktsioonides esineda ka unaarne **nimi*-konstruktsioon, näiteks *for*- ja *while*-tsüklite päisetes või *switch*-operaatoris. Me käsitleme neid situatsioone tagapool.

Mainigem, et need interpreteerimisprobleemid pole pärit masinorienteeritud keeltest (aadresside semantika on seal üheselt paigas), vaid probleemid on tekitatud kõrgtaseme-keelte disainijate poolt. Kordame: *C*-keeles pole eraldi *viidatüüpi*, ent on võimalus seada viitu suvalist tüüpi andmetele, nii lihtmuutujatele, vektoritele, massiividele kui ka struktuuridele.

Tuleb tõdeda, et *C*-keele „viidamajandus" oli kõrgemal abstraktsioonitasemel kui senini oli tehtud, nimelt konstruktsioon „viit *A*+*i*" arvestas reaalse mäluviida tegemisel vektori *A* elemendi pikkusega; kui too *A* on baitvektor, siis *A*+2 ≡ vektori 2. baidi aadress (indekseerimine algab 0-st), ja kui *A* element on kuitahes keeruline andmestruktuur, siis *C* garanteerib, et *A*+2 väärtus on ikkagi viit tolle vektori teisele elemendile. Seega, *A*++ (kui *A* on vektor) ei tähenda, et aadressile *A* liidetakse konstant 1, vaid et *A* väärtuseks saab vektori järgmise elemendi aadress.

Niisiis, *C* esimene uuendus (võrreldes *B*-ga) oli tüüpide *char* ja *int*[3] (pikkustega 1 bait ja 2 baiti ≡ 1 *PDP*-11 pesa) evitamine ning neid tüüpe omavatele objektidele viitade deklareerimise ja kasutamise mehhanismi lisamine.

Teine uuendus oli struktuursete objektide ja nende viitadega seonduva edasiarendamine. *Ritchie* kirjutab [Ritchie, lk. 10]:

```
int i, *pi, **ppi;
```

deklareerivad *integer*i, viida mingile *int*-tüüpi objektile ja viida viidale, mis omakorda viitab *int*-tüüpi objektile. Ning et

[1] Variandid: *a* on lihtmuutuja nimi, **a* on nt. vektori *a* esimese elemendi väärtus (*rvalue*na) ning &*a* on muutuja *a* aadress – kuhu saab kirjutada, seda varianti kasutatakse võimaldamaks alamprogrammi parameetrile väärtuse omistamist alamprogrammi „seest".

[2] Sama konstruktsioon, *B*=*A*+1, on *Forth*-keeles *B A* @ 1 + ! (me käsitleme *C*- ja *Forth*i viidamajanduse iseärasusi tagapool veelgi). Märkigem etteruttavalt, et *C* annab alamprogrammide parameetrid edasi reeglina *väärtuse* (ja mitte *aadressi*) järgi, inglise k. vastavalt *by value* ja *by address.* Esimene variant ei lase parameetrit üle kirjutada, teine aga küll (kui ei kasutata *const*-kaitset vmt.), ja tundub, et *C* viidatüüpide mehhanism kaitseb kaht asja: alamprogrammide parameetreid (so, neiks olevate lihtmuutujate väärtusi) ning massiivide „sisu" – elementide väärtusi.

[3] Esialgu polnud ujupunkt-tüüpi, kuivõrd seda ei toetanud esimene *PDP*-11 versioon, ent oli teada, et see tugi tuleb, ning *Ritchie* oli vastavateks keele täiendusteks valmis.

```
int f( ), *f( ), (*f)( );
```

kirjeldavad funktsioone: esimene tagastab *int*-väärtuse, teine tagastab viida *int*-tüüpi objektile (lihtmuutujale või massiivile) ja viimane kirjeldab viita funktsioonile, mis tagastab *int*-arvu, ja

```
int *api[10], (*pai)[10];
```

deklareerivad vastavalt vektori, mille elementideks on viidad *int*-tüüpi objektidele ning viida *int*-tüüpi vektorile. Kõik kolm näidet on toodud selleks, et lugeja aduks, kuidas on seotud programmiobjektid ning neile viitamise võimalused. *Ritchie* tunnistab, et *C* viitademehhanismi disainimisel[1] oli tal suuresti eeskujuks *Algol*-68[2], kus agregeeritud andmetüübid „ehitatakse üles“ lihtsamatest.

Ritchie nendib [samas, lk. 10], et need kaks *uuendust* viis ta sisse juba keelde nimega *NB* (*a New B*). Koos keele edasiarendamisega sai muudetud ka keele nimi. Kuna *NB* tundus liiga pikana ja et ilusam oleks jätkata ühetähelise nimega, ja et uus keel lähtus keelest *BCPL* (esmalt *Thompson*i *B*), siis võeti mainit keele nimest *B*-le järgnev täht *C.* Uus *NB* versioon tehtigi juba nimega *C*[3].

Uue nime saanud keele uuendused algasid loogikatehete *and* ja *or* (konjunktsioon ja disjunktsioon) täpsustamisega: kas nii seotakse loogilisi osatingimusi (vastavalt && ja ||)või sooritatakse operandidega bitikaupa tehteid (vastavalt & ja |)[4]. Näiteks, kui me tahame kirjutada filtri

```
if((laulja==M2gi||laulja=Chalice)&&laulja!=Kuusik){...
```

siis see laseb läbi Mäe, Chalice’i ja nende koosesitatud laulud ja eirab Kuusiku laulduid. Ning kui kirjutame

```
a=a&3; //või a&=3;
```

siis muutuja *a* uueks väärtuseks saab ta väärtuse parempoolseimate kahe biti ($3=011_2$) väärtus. Asi on selles, et esimese variandi korral me „arvutame“ osiste (konjunktide ja disjunktide) tõeväärtused (*tõene* või *väär*) ning konjunktsiooni tõeväärtus on ainult siis tõene, kui kõik komponendid on tõesed, ning disjunktsioon on väär ainult siis, kui ükski komponentavaldis pole tõese väärtusega. Ent teise variandi korral on tehte resultaadiks bitijada, näiteks:

```
010 & 101 = 000
010 & 111 = 010
010 | 101 = 111
010 | 111 = 111
010 | 000 = 010
```

[1] *D. R.* nendib [Ritchie, lk. 16 jj.], et *C* „viidamajandus“ on raskesti jälgitav ja õpitav, ning põhjendab, miks see nii on. *C* arendati tüübivabast *B*-keelest (kõik objektid olid parajasti masinasõna pikkused ning viidaaritmeetika oli triviaalne) ja oluline oli *B*-programmide hõlpus konverteerimine *C*-sse, kus viidatavad objektid võivad olla erinevate pikkustega. *Ritchie* tunnistab, et vastuvõetud lahendused olid paratamatud, ja et kriitika on õigustatud.
[2] Innovaatiline, ent läbikukkunud projekt; mis ja miks vt. näit. [Isotamm, PK] lk. 199 jj.
[3] Seda mõttekäiku järgides oleks *Ritchie* (või ta õpilaste) sama suunda järgiva uue keele nimi *P*. Edasi tuleks *L* ja siis oleks see „ristimisrida“ ammendatud.
[4] Bitikaupa tehted on (vist samuti) *C*-keele panus uuemate kõrgtaseme keelte võimalustesse.

B-keeles polnud bitikaupa loogikat (see on tavaline masinorienteeritud keeltes: masinkoodis ja assembleris). *C* astus keele abstraktsioonitasemelt selle võimalusega sammu tagasi, masinorienteeritud keelte poole.

Edasi, *Ritchie* pani eelkäijatega võrreldes suuremat rõhku standardfunktsioonide teekidele (mis deklareeritakse #*include*-makrodega). Ning *C*-kompilaator peab taoliste teekide kasutamiseks tegelema eeltöötlusega (*preprocessing*), ent see pole meie raamatu teema.

Ja veel, uuenduslik oli idee interpreteerida stringe kui lõputunnusega baidivektoreid, so. kui tavalisi ühemõõtmelisi massiive, selmet käsitleda neid omaette andmetüübina.

Ritchie nendib [Ritchie, lk. 12], et 1973. a. alguses oli *C* n-ö. valmis. Keel oli valmis realiseerima *UNIX*i tuuma suvalisele platvormile, kuigi loodud oli ta *PDP*-11 jaoks. Varsti tehti seda masinate *Honeywell* 635 ja *IBM*/360/370 jaoks. Aastal 1978 üllitasid *Brian Kernighan* ja *Dennis Ritchie C*-keele „valge raamatu" [K&R][1], mille kohta ütleb *Ritchie*, et selle raamatu teksti kirjutas *Kernighan* ja ta ise kirjutas ainult paar selle raamatu lisa.

Edasi, aastatel 1973..80 lisati keelde tüübid *unsigned, long*[2] ja *union* ning arendati edasi tüüpi *struct.*

Ritchie vaeb põhjuseid, miks tema *C* osutus nii edukaks nagu me tänapäeval adume [Ritchie, lk. 18 – 19] ja leiab, et tähtsaim tegur oli *UNIX*i edu. Kuna viimase mobiilsuse eelduseks oli *C* kompilaator, siis nood kaks koos tekitasid olulise sünergia.

Aga *C* lõi läbi ka mujal kui pelgalt *UNIX*i portaabluse garandina. *Ritchie* kirjutab, et hoolimata esialgsetest õpiraskustest on tema keel ikkagi lihtne, väike ja hõlpsalt transleeritav; tänu masina riistvaraga arvestavatele standardprogrammide teekidele on ta realiseeritav mistahes platvormil, kusjuures programmeerija ei pea teadma antud masina iseärasusi. *Ritchie*: ta (*C*) tehti „...kui kasulikke asju tegevate programmide kirjutamise vahend, arvestati võimalustega koostööks suurte operatsioonisüsteemidega ja suurte süsteemprogrammide programmeerimiseks...*C* rahuldab paljude programmeerijate vajadusi, ent ei püüa pakkuda liiga paljut."

Lõpetame selle jaotise hoiatusega. *ANSI* (*American National Standards Institite*) standardiseeris *C* 1989. aastal: see hõlmab „keelt ennast" ja [K&R] lisas *Standard Library* (lk. 241 – 258) toodud viitteistkümmet standardfunktsioonide teeki. Viimastest on mitmed oma aktuaalsuse kaotanud, näiteks *setjmp.h*, mis püüdis leevendada *PC*-de 1*MB*-sest operatiivmälust tingitud seiku: sellest mälust olid probleemideta adresseeritavad vaid esimesed 640*KB*.

Tavaliselt kõik *C*-kompilaatorid suvalistel platvormidel püüavad noid standardeid järgida, ent paraku on erandeid: mingil platvormil, näiteks *Turbo-C*, kirjutatud programm ei pruugi kompileeruda teis(t)el platvormi(de)l, näiteks *djgpp*-keskkonnas. Põhjus pole mitte „keele enda" konstruktsioonide realiseerimises, vaid peitub standardfunktsioonide teekide erinevas tõlgendamises. Kui te vahetate platvormi ning kompileerite oma seni korrektset programmi uues keskkonnas ja saate arusaamatuid veateateid või kogete oma programmi tavatut käitumist, siis on asi tõenäoliselt standardfunktsioonide teekide erinevas sisus.

[1] Meie kasutasime selle raamatu teist, autorite poolt paljuski parandatud ja täiendatud trükki (1988).
[2] 16-bitisel masinal olid *char*, *short*, *int* ja *long* vastavalt 1, 1, 2 ja 4 baiti pikad. 32-bitistel on *char* endiselt ühebaidine, *short* on kahebaidine, kuid nii *int* kui ka *long* on tavaliselt neljabaidised.

2.5. „Suured“ ja „väikesed“ keeled

Niisiis, jälgitav on mõjutuste jada: (*Algol*) → *CPL* → *BCPL* → *B* → *C*. *Algol*-60 on sulgudes kahel põhjusel: esiteks, ta disainiti algoritmide publitseerimise keeleks ja mitte programmeerimiskeeleks, ja teiseks, tema orientatsioon oli teadusarvutustele ja mitte süsteemprogrammeerimisele. Ent paljus oli ta jada järgnevate liikmete jaoks teedrajav: plokkstruktuur ja sellest johtuv viitade keskkond, *FORTRAN*ist erinev alamprogrammide käsitlus, liitoperaatorid jmt.

Nagu me püüdsime varasemas õppevahendis [Isotamm, PK] näidata, toimus programmeerimiskeelte areng selle esimestes etappides suunaga „madalam tase → kõrgem tase“, kusjuures taseme kriteeriumiks oli keele „kaugus“ 0-taseme masinkoodist. Protseduurorienteeritud keelte (3. tase) arendamine järgis sama malli, iga uus keel pürgis tavaliselt kõrgemale abstraktsioonitasemele, kas siis sünteesides seniste keelte võimalusi (nagu *PL/I* või *CPL*) või siis pakkudes süntaktilisi vahendeid tervete protseduuride programmeerimiseks (nagu *Prolog* või *SETL*) – pisut sarnaselt probleemorienteeritud keeltega (nagu *RPG* või *SQL*) – ent viimased *pole* universaalsed.

Selle valguses on *C*-keele arendamine üsna unikaalne: *CPL* on suur (ja liiane) keel, *BCPL* on *CPL*i kontsentraat (keskendudes süsteemprogrammeerimise vajadustele), *B* kontsentreerub veel rohkem süsteemprogrammeerimisele kui *BCPL* (ja loobub paljudest vahenditest), ning *C* ongi ehe ja minimalistlik süsteemiprogrammeerimise keel[1].

Niisiis, *C* arendamisel tundub olevat toiminud „normaalsele“ vastupidine süsteem: seda pole tehtud valemi „madalam → kõrgem“ järgi, vaid *C* tehti vastupidi: alustati kõrgtaseme keelest ning iga järgmine realisatsioon esitas madalamat abstraktsioonitaset[2]. Oponentide arvates on *C* jõudnudki assemblerini välja[3].

Eespool nentisime, et *C* on *minimalistlik*, so. *võimalikult väike* keel. Peatugem siinkohal põgusalt mõistetel „suur“ ja „väike“ keel. Suureks peetakse keelt, mis katab oma süntaktiliste vahenditega *kõik võimalikud* semantilised funktsioonid, näiteks sisendi ja väljundi, matemaatilised funktsioonid, tüübiteisendused, mäluhalduse, andmestruktuurid jne. Näiteks võiksid sobida masinkood ja assembler(keel), *Algol-60*, *COBOL*, *Ada* jpt., usutavasti ka *CPL*.

Väike keel pakub süntaktilisel tasemel oluliselt vähem, ekstreemsel juhul minimaalse komplekti vahendeid ning kõikvõimalikud muud vajalikud funktsioonid tehakse kättesaadavaks *keeleväliste* vahenditega, osa neist „pannakse kaasa“ translaatoriga, ent kasutajale jäetakse vabad käed keele funktsionaalsust suvaliselt laiendada. Toogem siingi näiteid: makroassembler, (kavandatud) *Algol-68*, *Forth*[4] ja meie raamatu *C*.

---

[1] Tegelikult pole *C* „lihtsam“ kui *B*: on öeldud, et *C* on „andmetüüpidega *B*“, so. *B*-ga võrreldes esindab ta „rikkamat“ taset.

[2] *CPL*→ *BCPL*→ *B*→ *C* (siiski, vt. eelmist märkust).

[3] *Niklaus Wirth* [Wirth] kirjutas,: „Tegelikult esitab *C* assembleri koodi, mis on peidetud ilmetu süntaksi varju ning mis on täis tipitud igasugu salamärke“. *Eric Lebherz* [Lebherz] tsiteerib iroonilist kommentaari, et *C* „kombineerib assembler-keele kogu elegantsi ja võimsuse assembler-keele loetavuse ja hallatavusega“.

[4] Mingis mõttes on *Forth* läinud kõige kaugemale. Me võime suvalise *C* standardfunktsiooni üle kirjutada, kuid me ei saa mitte kuidagi muuta *C* operaatoreid. Ent *Forth* võimaldab üle kirjutada keele mistahes *sõna*, vahet pole, on see meie enda defineeritud või kuulub *Forth*i tuuma.

Kumb variant on parem? Meie kaldume eelistama „väikekeele“-varianti: „suur“ keel on lõplik. Keele autorid on kõik ette näinud ning sinna pole võimalik mistahes funktsionaalsusi lisada. Seevastu „väike“ on avatud ja laiendatav keel: suvaline kasutaja saab lisada omakirjutatud funktsioone või vajadusel standardfunktsioone „üle kirjutada“. Väikeste keelte – nagu *C* – jaoks on kirjutatud aukartustäratavate mõõtmetega keeleväliste funktsioonide teeke[1].

Asi on selles, et välisfunktsioone saame kasutada – kui nad on kirjeldatud ja kättesaadavad – just nii, nagu nad oleksid keelde „sisse kirjutatud“, ent programmis kirjeldatud funktsioonid ja protseduurid[2] ei *laienda keelt*; nad on kasutatavad ainult antud programmis. Selles mõttes pole *FORTRAN*i „*CALL*“-alamprogrammid *keele* laiendajad, vaid keelevälised, mittesüntaktilised seigad[3]. Samas on *C*-keele *fprintf*-funktsioon näiliselt selle keele loomulik süntaktiline komponent, seega *C*-keele orgaaniline osa. Vähetähtis pole tõik, et reeglina on *C* funktsioonide teegid programmeeritud *C*-keeles; *FORTRAN*i alamprogrammid võivad olla kirjutatud mistahes keeles ja peavad enne kasutamist olema kompileeritud masinkoodi. *C*-kompilaator genereerib kogu *.exe*-faili koodi, transleerides ka teegifunktsioonid, kuid *FORTRAN*-kompilaatoril *pole võimalust* „*CALL*“-alamprogrammide lähtetekstide kasutamiseks.

Niisiis, „väike keel“ võib olla dünaamiline ja laiendatav, ja „suur keel“ on lõplik ning ilma jäetud kasutaja-arenduste lisamise võimalustest. Kõrvalpõikena: suurprojekt *PL\I* oli oma kolme komponendi (*Algol-60, FORTRAN* ja *COBOL*) poolest nii suur kui ka lõplik, kuid *IBM/OS*i makrode kasutamise võimalus tegi ta pigem lähedaseks avatud ja väikestele keeltele. Aga! *PL\I oli liiga suur* ja *liiga vähe* muude kui arvutusülesannetele ja (majandus)andmetöötlemisele orienteeritud võimalustega arvestav.

2.6. *C* ja Eesti

Siin tugineme mälestustele ja *Tõnis Keldri* meenutustele TÜ Arvutuskeskuse juubeliväljaandes [Kelder]. Mälestused on pärit möödunud sajandi 80-ndate aastate keskpaigast, *Elbi* (spordibaas Pärnu jõe ääres, pisut ülesvett Sindi linnast) suvekoolist, kus tol ajal noor *NL TA Arvutuskeskuse* programmeerija *Volodja Serebrjakov* tutvustas meile igati huvitavat ja meie jaoks tundmatut keelt *C*. Muuhulgas ütles ta juba tsiteeritud lause [Isotamm, PK, lk. 169]: „Знаете, Си что-то вроде наркотика – сначала противно, а потом не отвыкнешь“[4].

Edasi hakkas (tolleaegses mõttes) suur- ja miniarvutite aeg lõppema, nii ka N. Liidus. Viimase jaoks oli aeg eriti nutune, sest hiljuti oli lõpetatud originaalarvutite-projektid (sh., *Minsk*iseeria) ning perspektiivi nähti „adapteerimises“ (loe: illegaalses kopeerimises)[5]. See protsess ei jätnud puutumata muidugi ka töörühma Minski Elektronarvutite Instituudis, kus *Mark*

[1] Näiteks on *GNU C* funktsioonide teegis [GNU C L] nende funktsioonide kirjeldusi **794 trüki**leheküljel.
[2] Meenutagem, nii protseduurid kui ka funktsioonid on keele jaoks *alamprogrammid*, funktsioone saame kasutada avaldistes operandidena (nad tagastavad oma väärtuse) ja protseduuride toime avaldub *kõrvalefektides.*
[3] *FORTRAN*i *CALL* ei laienda keelt, ent annab võimaluse mistahes keeles kirjutatud ja masinkoodi transleeritud alamprogrammi kasutamiseks *FORTRAN*-programmis. Infovahetus põhi- ja alamprogrammi vahel on alati üheselt fikseeritud (parameetritega või ühisvälja abil). *C*-programm saab ka niisuguseid „sõltumatuid“ programme välja kutsuda ja käivitada, ent sootuks „vähemavalikult“. Me käsitleme seda hiljem (vt. *system*). Aga need võimalused ei *laienda FORTRAN*it või *C-keelt.* Keele määrab ta süntaks, ja mitte keeleväliste funktsioonide või alamprogrammide kasutamise võimalused.
[4] „Teate, see on midagi narkootikumitaolist: alul on vastik, aga pärast ei saa lahti“. *V. Serebrjakov* on praegu Moskva Riikliku Ülikooli tarkvarasüsteemide professor. Sama mõtet väljendab *Viktor Leppikson* [Leppikson C, lk. 9] nii: “C...algul panevat.. õppureid vihast vanduma ja hiljem vaimustusest kiljuma“.
[5] Ausa (ametkondlikuks kasutamiseks mõeldud) märgukirja kirjutas selle strateegia kohta *Aleksandr Semjonovitš Narinjani* 2. augustil 1985 [Narinjani].

Nemenman (pilti vt. ja teeneid loe [Isotamm PK, lk. 54 jj.]) juhtis projekti, mille ülesandeks oli ümberorienteerumine *Minsk*-seerialt mikroarvutile *EC-1840*, mille protsessoriks oli ajastu vaimule vastavalt „*Intel*-8086 analoog“ ja operatsioonisüsteemiks *Digital Research Inc* toote *CP/M* -86 „analoog“.

Mark osales koos meie *Merik Meriste*ga üleliidulises programmeerimiskeelte realiseerimise töögrupis, olid head tuttavad, ning 1985. aastal tegi *M. Nemenman Merik*ule ettepaneku kirjutada uue masina baastarkvara ühe osana *C*-kompilaator. Meie ülikool aktsepteeris lepingu sõlmimist ning varsti alustas tööd *Merik*u meeskonnas *Tõnis Kelder* ja pisut hiljem *Kersti Alev*, kes hakkas tegelema tüütu dokumentatsiooniga. Tegelikuks programmeerijaks hakkas *Tõnis. Merik* – olles ka oma kandidaadiväitekirja-teemaga seotud transleerimistehnikaga, – täitis projektijuhi rolli ja suhtles *Mark*iga.

Aega anti sellele meeskonnale umbes 1 aasta – õnneks, *C* alamhulga *Small-C* realiseerimiseks – aga see keel oli oluliselt vaesem [K&R]-versioonist: olema ei pidanud vahendeid toetamaks ujupunkttüüpi, struktuure, ühendeid (*union*), *n*-mõõtmelisi ($n>1$) massiive jne. Aga kuna *Mark* sai kasutada ainult 256 *KB* mälu, siis sai Tartu meeskond – teades kitsendusi – ülesandega hakkama.

Tõnis Kelder [Kelder, lk. 70]: „Leping sõlmiti 1986. aasta alguses tähtajaga umbes üks aasta (- - -). Järgnevalt tuli meil otsustada, kuidas siiski tööd teostada, sest arvutit, millele programm luua, meil ju tegelikult ei olnud. Tuli valida instrumentaalvahendid. Valikule aitas kaasa see, et ajakirjas *dr Dobbs's Journal* ilmusid keele *C* veelgi väiksema alamhulga, nn. *Tiny-C* kompilaatori lähtetekstid samas keeles. See kompilaator kompileeris keelest *Tiny-C Intel* 8080 (s.t. 8-bitise protsessori) assemblerisse. Meil oli juurdepääs Tartu Ülikooli programmeerimiskateedri personaalarvutile Apple II, millel oli lisaks 8080 protsessor, operatsioonisüsteem CP/M ning C-kompilaator. Seega otsustasime kasutada seda instrumentaalmasinana ning luua kompilaator nn. *bootstrapping*u (vt lisa 3) meetodil – täiendades kompilaatorit tema enda abil. Kirjutasime lähtetekstid ümber, täiendades neid nii, et nad rahuldaksid meie vajalikku lähtekeelt ning asendasime väljundis *Intel* 8080 assembleri *Intel* 8086 assembleriga. Kompilaatori põhiskeemis jätsime alles seal kasutatud nn. rekursiivse languse meetodi. Vastavalt kirjutasime samas keeles ka lahendusaja funktsioonide teegi (v.a. üsna minimaalne osa kõige madalamat taset, mis sai kirjutatud otseselt assembleris). Programmi tegelikku täitmist sai esialgselt kontrollida vaid Tallinnas TA Küberneetika-instituudis, kus meil oli aegajalt võimalik kasutada *Intel* 8086 protsessori baasil operatsioonisüsteemis *CP/M* -86 töötavat arvutit.(- - -) On selge, et selline töökorraldus ei saanud olla väga efektiivne. Seepärast tekkis mul idee kuidagi ära kasutada ka arvutuskeskuse suurarvutit *EC*-1060. Eelkõige otsustasin katsena realiseerida *C* kompilaatori ka sellele arvutile. Siin aga otsustasime kohe, et realiseerida tuleb täielik *C* ning instrumentaalkeelena kasutada assemblerit. Oli selge, et meie realiseeritav *C* alamhulk tegelikeks programmeerimisvajadusteks siiski vaevalt kõlbab ning me planeerisime juba ette järgmist lepingut – seekord juba täieliku keele *C* realiseerimise peale. Seepärast tahtsin juba varakult kontrollida kompileerimismeetodeid ning välja töötada andmestruktuure, mida edaspidises ära kasutada. Kompilaator valmis väga kiiresti ning oli 1986. aasta lõpuks juba kasutamiskõlblik. Kompileerimine toimus siin otse masinkoodi, loobutud oli vahepealsest assemblerist. Pärast lahendusaja teegi lisamist, mille juures aitasid mõningal määral *Tiit Eenma* ja *Hannes Näripä*, oli valmis saanud produkt, mis võimaldas küllaltki normaalselt programmeerida *EC*-arvutile keeles *C* ning mis ei jäänud oma tootlikkusnäitajate poolest põrmugi alla sel ajal kasutatud *FORTRAN* ning *PL\I* kompilaatoreile.“

Minskist tuli lõpuks valmis saanud arvuti *EC-1840* 1988. aasta lõpus; kompilaator sellele masinale saadi tähtajaks valmis ja anti tellijatele üle. Leping *C* täisversiooni realiseerimiseks oli sõlmitud juba varem (1987). Tsiteerigem taas *Tõnis Keldrit* [Kelder, lk. 72]: „Otsustasime kompilaatori realiseerimisel kasutada minu poolt arvutil *EC*-1060 saadud kogemusi, üldist programmi ülesehitust ning andmestruktuure. Kompilaatori realiseerimiskeeleks oli *Intel* 8086 assembler, kasutasime kompileerimist otse masinkoodi ning rekursiivse languse meetodit. Üle võtsime ka kogu projekti liigendamise väga paljudeks mooduliteks – kokku koosnes kompilaator lõpuks umbes 260 lähtetekstist, lahendusaja teegi koosseisus oli aga veelgi rohkem mooduleid. (- - -) Uue inimesena osales projektis *Jüri Helekivi*, kes oli sel ajal alles üliõpilane. Praktikatööna realiseeris ta standardfunktsioonide teegile ujupunktoperatsioonid. Sedapuhku oli meil juba kasutada arvuti ning olemas ka vastav instrumentaaltarkvara – assembler, linker ning *librarian*[1]. Seetõttu laabus töö ilma eriliste ekstravagantsusteta.

Joonis 2.6.a. *Tõnis Kelder* (ees) ja *Merik Meriste* (paremal), 1978. aastal Navesti jõel.

Ka see projekt lõppes meile edukalt ning sedapuhku oli tegemist juba üsna tõsiseltvõetava kompilaatoriga. Selle tugevaimaks küljeks jäi endiselt programmi väiksus ja kompaktsus – valmiskujul kompilaatori maht oli kokku vaid ligemale 62 kilobaiti. (- - -)

Kokkuvõtteks võib öelda, et meie esimene projekt (*Small-C* kompilaator) oli rohkem mänguasi, ehkki teda Venemaal mingil määral kasutati. Seevastu kompilaator keele *C* täisversioonile oli juba tööks täiesti kõlblik. Kasutasin seda ise, kuni tekkis juurdepääs parematele kompilaatoritele (eriti alates firma *Borland* kompilaatorist *Turbo-C* 1.5). Algul vaid katseliselt loodud kompilaator arvutile *EC*-1060 ehk arvutisüsteemile *IBM/*370 *OS* (edaspidi portisin selle ka süsteemi *VM*) pälvis aga teenimatult liiga vähe tähelepanu. (- - -) Minu arvates oleks see olnud kasutatav umbes viie aasta jooksul (1986 – 1991) mitte halvemini kui *FORTRAN* või *PL\I*. Keel *C* polnud aga kahjuks vastava süsteemi programmeerijate hulgas eriti levinud.

[1] *Tõnis Kelder* arvas, et *librarian* võiks eesti keeles olla „teegihaldur“, tol ajal oli *MS-DOS*is ja ka teistes analoogilistes programmeerimissüsteemides omaette utiliit teekide moodustamiseks, moodulite lisamiseks, asendamiseks ja kustutamiseks (*MS-DOS*is nimega *LIB.EXE*).

Ise olen ma kõigest sellest saanud poolehoiu keele *C* vastu, oma arvates küllaltki hea *C*-keeles programmeerimise oskuse, teadmise, et olen realiseerinud kolm keele *C* kompilaatorit ning lisaks veel 450 rubla[1] preemiat.“

Tõnis on nende ridade autorile sõbra ja kolleegina poetanud, et *C* realiseerimisele ajendas teda muuhulgas tahtmine seda keelt lähemalt tundma õppida, ja et parim võimalus seda saavutada on keele translaatori kirjutamine.

[1] Kuni *perestroika*-inflatsioonini võrdus paljude asjatundjate arvates üks rubla tänase 35 krooniga, seega *Tõnis* sai ligi 16 tuhat EEK-i. Ametniku kuupalk oli *ca* 100 rubla, *TRÜ* professorist kateedrijuhatajal 400.

3. C. Ülevaade

Oleme rõhutanud, et *C* on *väike keel*, so. enamik keele funktsionaalsusest on realiseeritud keeleväliste funktsioonide teekidega. Teine epiteet on olnud, et *C* on *staatiline keel*, so. keele tasemel me ei saa kirjeldada dünaamilisi massiive, vaid ainult staatilisi, st. selliseid, mille jaoks eraldab mälu *kompilaator*. Seega: *C* lubab kirjeldust *V*[17], ent mitte *V*[*n*], kui *n pole defineeritud konstant, vaid on sisestatud või arvutatud väärtus*. Programmi „normaalne" struktuur on järgmine:

1. kommentaariridа: täisnimi (näit. *first.c*), otstarve, autor, kirjutamise kuupäev;
2. makrokäsud (vähemalt *#include <stdio.h>*), muidu ei saa mitte midagi sisulist teha;
3. vajadusel globaalsete muutujate (sh. massiivide) kirjeldused;
4. vajadusel alamprogrammide e. moodulite (protseduurid ja funktsioonid) eelkirjeldused;
5. vajadusel alamprogrammide e. moodulite tekstid;
6. *main*-mooduli (st. põhiprogrammi) tekst; kui kõik alamprogrammid on eeldefineeritud, siis võib ta paikneda suvalises kohas. Vastasel juhul peab see moodul olema kogu *C*-programmi teksti viimane komponent.

Alljärgnevas käsitleme neid komponente pisut detailsemalt. Me hoidume selles peatükis nii „peensustest" – nendega saab tutvuda kas kirjanduse, Interneti või – *UNIX*-keskkonnas – *man*-direktiivi abil – kui ka näidetest, neid peaks leiduma piisavalt järgmistes peatükkides.

3.1. Teksti algus

Kommentaaride kirjutamise hea stiil on panna nad eraldi reale (või ridadele), või tekstirea lõppu. Viimasel juhul (nagu ka üherealise kommentaari puhul) algab kommentaar markeriga // ja lõpeb reavahetusega. Mitmerealine kommentaar tuleb panna sulupaari /*...*/ vahele, see ei saa sisaldada mistahes teist kommentaari.

Makrovahenditest on tavaliselt vaja oma rakendusprogrammis kasutada kindlasti ***#include***-makrot ja (tihti) ***#define***-ga algavat makromäärangut. Esimene neist annab juurdepääsu makroga määratud *funktsioonide teegi* moodulitele, need teegid on kas süsteemsed või siis erateegid, so. kasutaja enda loodud.

Esimesed kirjeldatakse kujul *#include <nimi.h>* ning kasutajaloodud kujul *#include „nimi.h"*, viimasel juhul peab erateek olema transleeritava C-programmiga samas teegis.

Kui *ANSI*[1] standardiseeris *C*, siis standardi alla kuulus 15 standardprogrammide teeki, mis tuli realiseerida uutel platvormidel *ANSI* nõudeid järgides, [K&R, lk. 241] järgi olid nendeks teegid[2]

assert.h, ctype.h, errno.h, float.h, limits.h, locale.h, math.h, setjmp.h, signal.h, stdarg.h, stddef.h, stdio.h, stdlib.h, string.h ja *time.h.*

[1] *American National Standard Institute*; *C* standardiseeriti 1983. a.
[2] Kõikide nende teekide kõikide funktsioonide kirjeldusi vt. [K&R, lk. 241 jj.]

Tänapäeval pole küll mõned siintooduist enam aktuaalsed (näit. *locale* või *setjmp*), kuid enamik neist on siiski vältimatult vajalikud. Viimastest teeme lühiülevaate jaotises 3.5[1].

Definitsioonide jaoks on makrovahend *#define.* See sarnaneb klassikalisele makrovahendite käsitlusele[2]: *makromäärang* kirjeldab *makrokäsu* ja *genereerimiseeskirjad* ning programmi tekstis leitud makrokäsud asendatakse eeskirjadega määratud *makrolaienditega.*

Käsu formaat on järgmine: *#define* <makrokäsu nimi><genereerimiseeskiri>

Toome mõned variandid, tuginedes eeskätt [K&R, lk. 89..91].

Konstandi defineerimine. Keele tasemel ei võimalda *C* dünaamiliste andmestruktuuride (vektor või massiiv) defineerimist – just nii nagu *FORTRAN*gi; massiivide indeksite ülemrajad tuleb konstantidena „sisse programmeerida". Tavaliselt me kasutame selliseid konstante oma programmis üsna mitmes kohas, näiteks tsüklioperaatorites, ja kui me peame mingil ajal oma programmi nii modifitseerima, et ta hakkaks töötama uute „pikkustega", siis tuleb programmis kõik need sissekirjutatud konstandid ära parandada.

Kindlam variant on defineerida konstandid ja kasutada neid massiivide kirjeldustes; muutmiseks peame lihtsalt konstandi ümber defineerima vaid ühes kohas ning kõik muud kasutused toimivad peale uut transleerimist korrektselt. Näiteks,

#define nelikilo 4096 //lõputunnust pole! Makrokäsu nimi on siin *nelikilo* ja see
//asendatakse kõikjal makrolaiendiga *4096.*
ja toimib kirjeldus `char[nelikilo];`

või operaatori päis

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

Teise näite toome raamatust [K&R, lk. 89], kus defineeritakse lõpmatu tsükli päis *forever*:

```
#define forever for(;;)
```

Lihtsates programmides on tihtipeale mitmeid tavatsüklipäiseid kujul `for(i=0;i<n;i++)` – me võiksime kirjutamisi kokku hoida, kui teeksime makromäärangu[3]

```
#define fin[4] for(i=0;i<n;i++)
```

Näiteks, kui programmis on lõik
`for(i=0;i<n;i++)sum+=a[i];`
võiksime selle asemel kirjutada
`fin sum+=a[i];`

[1] Erandina tutvume teegiga <stdarg.h> osas 7.5, tulles teema juurde „näite pealt".
[2] Makrokeelest loe nt. [Isotamm, PK], lk. 91 jj.
[3] Selle tsükli päises kasutatud *n* võib olla samuti defineeritud *#define* abil, so. makromäärangus võime kasutada teisi makrokäske.
[4] Mnemoonika: *fin=**f**or **i** to **n***

Globaalsed kirjeldused[1] kirjeldavad globaalsed muutujaid ja andmestruktuure. Alamprogramme on võimalik *eelkirjeldada*, näiteks

```
int ap(int a, char *b);
```

hiljem tuleb nood eelkirjeldatud alamprogrammid „lahti kirjutada“, andes alamprogrammi kirjeldusele lisaks tema algoritmi operaatorsulgude { } vahel. Seega, eelmist näidet järgides:

```
int ap(int a, char *b){ alamprogrammi „keha“ }
```

Kui aga alamprogramme ei eelkirjeldata, siis nende kirjeldused tuleb esitada kasutamise järjekorras: enne kirjeldus, siis kasutamine. Peamoodul (*main*) pole eelkirjeldatav, see on alati alamprogrammi tegelik tekst. Ja kui teised moodulid pole eelkirjeldatud, siis on *main*-moodul programmi teksti viimane (võib juhtuda, et ka ainus) lõik[2].

3.2. Moodulite tekstid

Need on lokaalsete, ainult antud programmis kasutatavate alamprogrammide tekstid. Kui nad pole eelkirjeldatud, siis nad tuleb kirjutada nii, et iga alamprogrammi tekst eelneks tema kasutamisele teise alamprogrammi või *main*-mooduli poolt. Alamprogramm algab kirjeldusega:

```
<tüüp> <nimi> (<formaalsed parameetrid>|void| ){...
```

näiteks, `void p1(int *a)` või
`int p2(void)` või
`char *p3( )`[3]

Alamprogrammi tekst ise asub operaatorsulgude {...} vahel ning tuleb vormistada järgmiselt: esmalt lokaalsete muutujate kirjeldused[4], ja siis operaatorid. Ja kui alamprogramm on vormistatud kui funktsioon, mis tagastab oma kirjelduses määratud tüüpi tulemuse[5], siis alamprogrammi viimane täidetav käsk peab olema *return <viit või väärtus>*. *Return*-käske võib tekstis olla rohkem kui üks. Alamprogramm *võib* globaalseid andmekirjeldusi üle kirjutada kuid loetavuse huvides tuleks seda programmi koostades vältida. Ülekirjutamine ei muuda globaalse samanimelise muutuja väärtust, küll aga lõikab ära juurdepääsu globaalsele muutujale alamprogrammist.

Peamooduli (***main***-mooduli) tekstile on seatud lisanõudeid, kuivõrd antud tekstifailist kompileeritud *.exe*-faili täitmine algab just *main*-moodulist ning see peab ka tagama tagasiside teda väljakutsunud programmile[6]. Seetõttu peab *UNIX*i-laadsetes keskkondades *main*-mooduli

[1] Globaalne objekt on kättesaadav igas antud *C*-teksti alamprogrammis; alamprogrammis kirjeldatud andmed on aga nähtavad ainult selle alamprogrammi piires, st. nad on *lokaalsed.*

[2] Vististi kõik siinkirjutaja näited järgivad seda stiili – põhjuseks on vastumeelsus sellise teksti kirjutamiseks, mida võib vältida – nagu alamprogrammide eelkirjelduste puhul ongi.

[3] Kirjeldus *char *p*3 tähendab, et funktsioon *p*3 tagastab viida *char*-tüüpi objektile.

[4] Erinevalt plokkstruktuuriga keeltest (nagu *Algol*-60) ei saa *C*-s kirjeldada lokaalseid alamprogramme.

[5] St. tüüp pole *void.*

[6] See on vähetähtis, kui juhtimine tagastatakse operatsioonisüsteemile, ent on oluline, kui eralditransleeritud ja käivitatud programm annab juhtimise tagasi teda väljakutsunud *C*-programmile. Tagastatud 0 tähendab normaalset lõppu, täpsemalt väite „lõppes tõrkega“ eitust.

tüüp olema *int* (*MS-DOS* lubab ka tüüpi *void*) ning sõltumatult platvormist saab *main*-moodulis kirjeldada programmi käsurea-parameetreid.

Parameetriteta programmi *main*-mooduli kirjeldus algab `int main( ){...` ja parameetritega variandi puhul: `int main(int argc,char *argv[ ]){...`

C täitmisaegne keskkond[1] (teisisõnu *C* virtuaalarvuti) omistab *argc* väärtuseks käsurea (*command line*) argumentide arvu ning nende *string*-väärtuste viidad kirjutatakse vasakult paremale vektorisse *argv*, seega `argv[0]` viitab alati *programmi laiendita nimele*. Näiteks, kui käsurealt käivitada programm *spuu* parameetritega *loomad* ja *logi.txt*:

```
>spuu loomad logi.txt
```

siis käivitatud programmis `argc`=3 ja `argv[0]="spuu"`, `argv[1]="loomad"` ja `argv[2]="logi.txt"`.

3.3. Avaldised

Kui programmeerimiskeele süntaks on formaliseeritav (näiteks, *Backus–Nauri* notatsiooni kasutades[2]), siis mõiste *programm* defineeritakse tavaliselt kui *operaatorite jada*[3]. Omakorda kasutatakse mõnes operaatoris (ingl. k. *statement*) avaldisi *(expression*); need on kas *aritmeetilised* või *loogilised.* Ning mõned operaatorid võivad sisaldada omistamis- ja tingimuslikke *avaldisi*, sj. eksisteerivad ka omistamis- ja tingimus*operaatorid.* Näiteks, `i=0`; on operaator, aga lõik (`b=a[i]`) tingimusoperaatoris `if((b=a[i])=='0') goto next;` on omistamis*avaldis.* Ja omistamisoperaatoris `z=(a>b)?a:b;` on `(a>b)` *tingimuslik avaldis.*

Sisuliselt on *operaator* programmi teksti iseseisev komponent, mille koostisse võivad kuuluda *avaldised*, mis *pole* sellised iseseisvad üksused; näiteks ei saa me avaldisi ei märgendada ega ka neile suunata. Kui programmi teksti võrrelda *jutuga*, siis iga operaator on semikooloniga lõppev iseseisev *lause* selles *jutus*.

3.3.1. Aritmeetiline avaldis

Aritmeetiline avaldis kujutab endast *infix*-kujul[4] (sulg)avaldist, kusjuures *esmasavaldisena* käsitletakse konstanti, muutujat või funktsiooni. Aritmeetilisteks avaldisteks on näiteks

```
17
a
a+17
a/sin(x)
(a+1)/(b*(c-log(x))
```

Üldiselt täidetakse tehteid vasakult paremale prioriteete järgides: esmalt korrutamine ja jagamine, siis liitmine ja lahutamine. Sulgudega saab tehete täitmise järjekorda muuta.

[1] Vt. [K&R], lk. 114 jj.
[2] Vt. näit. [Isotamm, PK] lk. 236.
[3] Ka kirjeldused on *operaatorid.*
[4] Vt. [Isotamm, PK] lk. 225.

Aritmeetilises avaldises on kasutatavad 5 binaarset tehet: +, –, *, / ja % (neli esimest on kasutatavad kõigi samatüüpi operandide vahel, sj. / tulemus on ujupunktarvude „täpne“ jagatis ja positiivsete *int*-arvude jagatise täisosa ning %-tehe annab *int*-arvude jagatise jäägi. Näiteks, 5/2 tulemus on 2 ning 5%2 tulemus on 1

3.3.2. Loogiline avaldis

Loogiline avaldis esitab algoritmi *tõeväärtuse* arvutamiseks, elementaarkonstruktsioonideks on üksikväärtused „tõene“ (*int*-arv>0) ja „väär“ (0), neid saab „välja töötada“ võrdlustehete või protsesside lõpu signaalidega ning neid saab omavahel siduda loogilise liitmise või korrutamise tehetega (vastavalt || ja &&); vajadusel kasutatakse sulgavaldisi. Näiteks,

```
((a>b)||(c<=2))&&(d!=0)
```

Avaldise operandiks võib olla seejuures bitikaupa tehte resultaat, nende arvutamiseks on järgmised (näite)võimalused:

```
|     0|1=1, 0|0=0
&     0&1=0, 1&1=1
^     0^0=0, 1^1=0, 0^1=1, 1^0=1
```

ja mitmebitiliste operandide puhul tehakse tehe bitikaupa, näit. `101&011=001`.

3.4. Operaatorid

3.4.1. Omistamine

Omistamisoperaatori üldkuju on <*muutuja*> = <*avaldis*>. Omistamisel on mitmeid erikujusid. Näiteks, `i=i+1`; saab kirjutada nii kujul `i++`, `++i` kui ka `i+=1`. Viimast varianti võime üldistatult tähistada kui `a⊕b`, harjumuspäraselt tähendab see tehet `a=a⊕b` ning ⊕ on kas +, –, *, /, %, ^, &, |, << või >> (viimased kaks on bitijada vasakule ja paremale nihutamised). Ja näiteks *a**=*y*+1 on sama, mis *a*=*a**(*y*+1).

Increment- ja *decrement*-omistamisrežiimid (`i++` või `++i` ning `i--` või `--i`) kätkevad teatud ohtu. Kui me kasutame neid omistamis*operaatoritena*, näiteks `i++`; või `++i`; on nad tõepoolest samaväärsed operaatoritega `i=i+1`; või `i+=1`; Aga kui me kasutame neid mingi operaatori koosseisus omistamis*avaldistena*, on lihtne eksida[1].

Omistamist võime kasutada ka kombineeritult. Näiteks, `a=(b=c/2);` Siin on tegemist omistamisoperaatoriga, mille paremas pooles kasutatakse omistamisavaldist.

3.4.2. Suunamine

Suunamine on kujul *goto märgend*, kus märgendiga saab varustada kõiki *operaatoreid* peale kirjelduste. Märgend tuleb kirjutada kui ***nimi:*** ja nime moodustamisel tuleb järgida samu

[1] *Kernighan* ja *Ritchie* toovad järgmise näite [K&R, lk. 46]: kui *n*=5, siis *x*=*n*++ on sama, mis *x*=*n*; *n*=*n*+1; aga *x*=++*n*; tähendab: *n*=*n*+1; *x*=*n*; Niisiis, esimesel juhul *x*=5 ja teisel *x*=6.

reegleid nagu muude idntifikaatorite puhulgi. Tehtav on ka suunamine tühjale märgendatud operaatorile, näiteks `goto ots;` kui tekstis leidub lõik `ots:;`

3.4.3. Tingimus

Tingimus (tingimuslik operaator) on kujul

```
if(loogiline avaldis) operaator1
else operaator2   //võib puududa
```

kui loogiline avaldis on tõene (väärtus > 0), siis täidetakse `operaator`$_1$ ja kui ei ole tõene (väärtus = 0), siis (kui on antud *else*-komponent) `operaator`$_2$. Mõlemad operaatorid võivad olla *liitoperaatorid*: sel juhul on operaatorsulgude paari { } vahel rohkem kui üks operaator.

Omaette käsitlust väärib *else-if*-konstruktsioon (vt. [K&R], lk. 57):

```
if(avaldis) operaator
else if(avaldis) operaator
else if(avaldis) operaator
else if(avaldis) operaator
else operaator
```

Viimane operaator täidetakse ainult siis, kui kõik talle eelnevad tingimused on rahuldamata; kogu konstruktsiooni täitmine lõpetatakse niipea kui saab täidetud üks operaator.

C-s on võimalus „tingimuslikuks omistamiseks“, kasutades kolmend- (*ternary*)-operaatorit „?“[1]. Näiteks, programmilõik

```
if(a>b) z=a;
else z=b;
```

on asendatav avaldisega

```
z=(a>b)? a : b;
```

Omistamise kasutamist tingimuses illustreerib ka järgmine näide:

```
if((b=a[i])=='0') goto next;
```

3.4.4. Lüliti

Lülitioperaator on kujul

```
switch (avaldis){
     case konstantavaldis: operaatorid [break;]
     case konstantavaldis: operaatorid [break;]
     ...
     [default: operaatorid]
     }
```

Tavaliselt lõpeb *case*-„rida“ *break*-käsuga, mis annab juhtimise lüliti tegevuspiirkonnast välja, ent on juhtumeid, kus on otstarbekas täita ka algvalikule järgnev operaator ─ sel juhul käsku *break* ei kirjutata. Märkusena meenutagem, et avaldiseks on ka muutuja nimi ning konstantavaldis ─ konstant. *Default* on sundvalik: tegevus, mis käivitub, kui avaldise väärtus ei

[1] Vt. [K&R], lk. 51.

võrdu ühegi konstantavaldise väärtusega, ent see on variant pole kohuslik Kui seda pole, siis lülitioperaator lülitioperaator ei teegi midagi juhul, kui ükski valikuvariant ei realiseeru.

3.4.5. Tsükkel

Tsüklioperaatoritest on kasutatavad nii *for*- kui ka *while*-variandid. Neist esimese kuju on *for(avaldis1;avaldis2;avaldis3) operaator,* mille kirjeldus on „klassikaline" tsükkel: tüüpiliselt esimene avaldis algväärtustab tsükliindeksi, teine määrab jätkamistingimuse ning kolmas tsükli sammu[1]. Teise variandi kuju on *while (avaldis) operaator;* – kuni *avaldise* väärtus on tõene, korratakse *operaatorit,* kusjuures *avaldise* argumentide väärtusi muudetakse *operaatoris.*

Tsükli katkestamiseks saab (nagu lüliti puhulgi) kasutada operaatorit *break*, mis aga ei pruugi aidata, kui tegemist on tsükliga tsüklis: *break* katkestab ainult kõige lähema hõlmava tsükli täitmise; vajadusel tuleb kasutada operaatoriga *goto* tsüklitest välja suunamist – või *return*i, mis lõpetab kogu alamprogrammi töö.

3.5. Standardfunktsioonid

Selles alapeatükis tutvustame minimaalset komplekti *ANSI* poolt standardiseeritud funktsioonidest. Tugineme *Kernighan*i ja *Ritchie* „valgele raamatule" [K&R, lk. 241 jj.] ning *Tõnis Kelderi* ja *Ülo Kaasik*u brošüürile [KjaK-2]. *UNIX*i-keskkonnas töötades saab interaktiivselt teavet enamiku funktsioonide kohta käsuga *man* (näiteks, *man stdio.h* või *man fopen*).

Allpool kohtame funktsioonide formaalsete parameetrite kirjeldustes vahel esitust *const char *nimi* : sellise parameetri väärtuseks on viit stringile (baidivektorile) ning atribuut *const* paneb kompilaatori ära hoidma selle stringi võimalikku ülekirjutamist funktsiooni poolt ning annab kasutajale kindlustunde, et tema string on kaitstud.

Juhime lugeja tähelepanu tõigale, et keelevälised vahendid (aga funktsioonide teegid seda sisuliselt on) *ei ole rangelt standardiseeritud*[2] ning erinevates keskkondades (*UNIX*-põhised *gcc*, *Cygwin*, *Dev-C++* või *DOS*ipõhised *Borland Turbo C*, *djgpp*) kasutatavate funktsioonide teegid võivad olla realiseeritud erinevalt, isegi kui funktsiooni kirjeldus on ühesugune. Seetõttu võib juhtuda, et ühes keskkonnas saame oma *C*-programmi probleemideta kompileerida, teises aga mitte või et kompileerida saame, aga programm ei tööta nii nagu me soovime.

3.5.1. Sisend ja väljund: <stdio.h>

3.5.1.1. Töö failidega

Failide alaline asukoht on välismälus (tänapäeval kettal) ning nendega manipuleerimine toimub „pidemete"[3] abil. Pide on *FILE*-tüüpi viit, mis tuleb kirjelduses kindlasti väärtustada tühjana, näiteks:

```
FILE *WrEf=NULL;
```

[1] Iga avaldis võib olla „tühi", kuid eraldaja „;" on kohustuslik. Näiteks, lõpmatu tsükli garanteerib *for*(;;)...
[2] Tõsi, *ANSI* standardiseeris 1983. a. 15 teeki, aga paraku ei pruugita mõnel platvormil standardeid järgida.
[3] *File handle*. [Tavast&Hanson]: faili ühene lühiidentifikaator, mille opsüsteem moodustab faili avamisel (lk. 86)

1. Avamine: `FILE *fopen(const char *filename, const char *mode)` avab faili (kui see õnnestus[1], siis tagastatakse *pide* (muidu jääb selle väärtuseks *NULL*)), sj. parameeter *mode* näitab, mis tüüpi see fail on ja mida temaga tohib teha. *Mode* eeldab, et vaikimisi avatakse *tekstifail* kas lugemiseks („r“), loomiseks („w“, kui samanimeline oli, siis kirjutatakse ta üle) või kirjutamiseks („a“: kui samanimeline oli, siis saab sinna kirjeid lõppu lisada). Kui faili tüüp on *binary*, siis tuleb kirjutada vastavalt „rb“, „wb“ ja „ab“. Näiteks, `WrEf=fopen(„minuoma.pin“,“wb“);`
2. Puhvri tühjendamine: `int fflush(FILE *pide)` kirjutab väljundpuhvri jäägi kohe kettale[2], tagastab lipu *EOF*, kui operatsioon nurjus.
3. Sulgemine: `int fclose(FILE *pide)` nullib pideme, kuid ei pruugi puhvrit tühjendada. Vea korral tagastab lipu *EOF*. Näiteks, `fclose(WrEf);`
4. Kustutamine: `int remove(const char *filename)` kustutab faili nime järgi; kui töö õnnestus, tagastatakse 0. Näiteks, `remove(„minuoma.pin“);`
5. Ümbernimetamine `int rename(const char *vananimi, const char *uusnimi)` tagastab väärtuse 0, kui nimevahetus õnnestus. Näiteks, `rename(„minuoma.pin“,“meieoma.pin“);`
6. Kirjutamine[3]: `size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *pide)` kirjutab viida *ptr* järgi faili pidemega *pide* *nobj* kirjet, igaüks pikkusega *size* baiti. Tagastab kirjutatud kirjete arvu, mis edu korral peab võrduma *nobj*ga (*nobj = number of objects*).
7. Lugemine: `size_t fread(void *ptr, size_t size, size_t nobj, FILE *pide)` loeb *ptr*-iga viidatud väljale failist pidemega *pide* ülimalt *nobj* kirjet, igaüks pikkusega *size* baiti. Tagastab loetud kirjete arvu, mis normaalsel juhul on väiksem kui *nobj* (kui see pole 1).
8. Faili tagasikerimine: `void rewind(FILE *pide)` viib „lugemispea“ faili algusse.
9. Sümboli lugemine: `int fgetc(FILE *pide)` edastab tekstifailist loetud sümboli *ASCII*-koodi (või signaali *EOF*, kui fail on otsas või juhtus tõrge).
10. Sümboli kirjutamine: `int fputc(int c, FILE *pide)` kirjutab sümboli (mis teisendatakse tüüpi *unsigned char*) faili. Tagastab kirjutatud sümboli koodi õnnestumise ja *EOF* ebaõnnestumise korral. Faili tüüp p.o. tekst.
11. Stringi lugemine: `char *fgets(char *s, int n, FILE *pide)` loeb tekstifailist kuni *n*–1 sümbolit vektorisse *s*, katkestab sisestamise rea lõputunnuse leidmisel ning lisab stringi lõpumarkeri ’\0’. Vea korral tagastab lipu *NULL*.
12. Stringi kirjutamine: `int fputs(const char *s, FILE *pide)` kirjutab stringi *s* (mis ei pea lõppema reavahetusega ’\n’) tekstifaili; vea korral tagastab signaali *EOF*.
13. Sümboli tagastamine: `int ungetc(int c, FILE *pide)` tõstab *c* tagasi sisendvoo järgmisena loetavaks sümboliks. Seda võimalust ei saa kasutada tsükliliselt. Vea korral tagastatakse signaal *EOF*. Fail p.o. tekstitüüpi.

[1] Meile tundub tänapäeval arusaamatuna, miks välisseadmetega tegelevad funktsioonid arvestavad võimalike riistvaratõrgetega, aga omal ajal olid need tavalised, võiks öelda, reeglipärased ning programmeerijad pidid selliste võimalustega alati arvestama. Samas, ebaõnnestub ka olematu faili avamine lugemiseks.
[2] *fflush* tühjendab muide suvalise *I/O*-puhvri, näit. klaviatuuri oma (*stdin*), mis võib olla kasulik *scanf*i puhul.
[3] Kirjutamise ja lugemise näiteid võib leida alapeatükist „puud“.

Sellelaadsete vahendite hulka kuuluvad ka funktsioonid *fscanf*( ) ja *fprintf*( ). Esimene neist loeb sisendvoost (failist) tekstiformaadis kirje ning teisendab (vajadusel) muutujate väärtused näidatud sisekujudele (üldjuhul on need *struktuuri* väljad), teine aga loeb failist „sisekujukirje“ ning konverteerib selle (vajadusel tüübiteisendusi kasutades) tekstiks. Me käsitleme neid funktsioone eraldi ja tagapool, kuivõrd nad on siinkäsitletuist oluliselt komplitseeritumad (vt osad „formaaditud sisend“ ja „formaaditud väljund“).

3.5.1.1.1. Faili pikkus: teek <sys/stat.h>

Seda teeki [K&R]-*ANSI C* ei kirjelda, kuid teek on kasutatav *gcc*-keskkonnas. Rakendusprogrammeerija jaoks on ta hea vahend eeskätt seetõttu, et võimaldab teada saada avatud faili pikkuse baitides – nii saame kogu faili mällu lugemiseks küsida funktsiooniga *malloc* vajaliku ruumi. Teegis defineeritakse struktuur *stat* (vt. [sys/stat]) järgmiste liikmetega (me ei tõlgi väljade kirjeldusi, lootes, et vajalik info on kas sõnaraamatuga või ilma arusaadav, liiatigi eluliselt vajalik väli tolles struktuuris on vaid *st_size*):

```
dev_t     st_dev     ID of device containing file
ino_t     st_ino     file serial number
mode_t    st_mode    mode of file (see below)
nlink_t   st_nlink   number of links to the file
uid_t     st_uid     user ID of file
gid_t     st_gid     group ID of file
dev_t     st_rdev    device ID (if file is character or block special)
off_t     st_size    file size in bytes (if file is a regular file)
time_t    st_atime   time of last access
time_t    st_mtime   time of last data modification
time_t    st_ctime   time of last status change
blksize_t st_blksize a filesystem-specific preferred I/O block size for
                     this object.  In some filesystem types, this may
                     vary from file to file
blkcnt_t  st_blocks  number of blocks allocated for this object
```

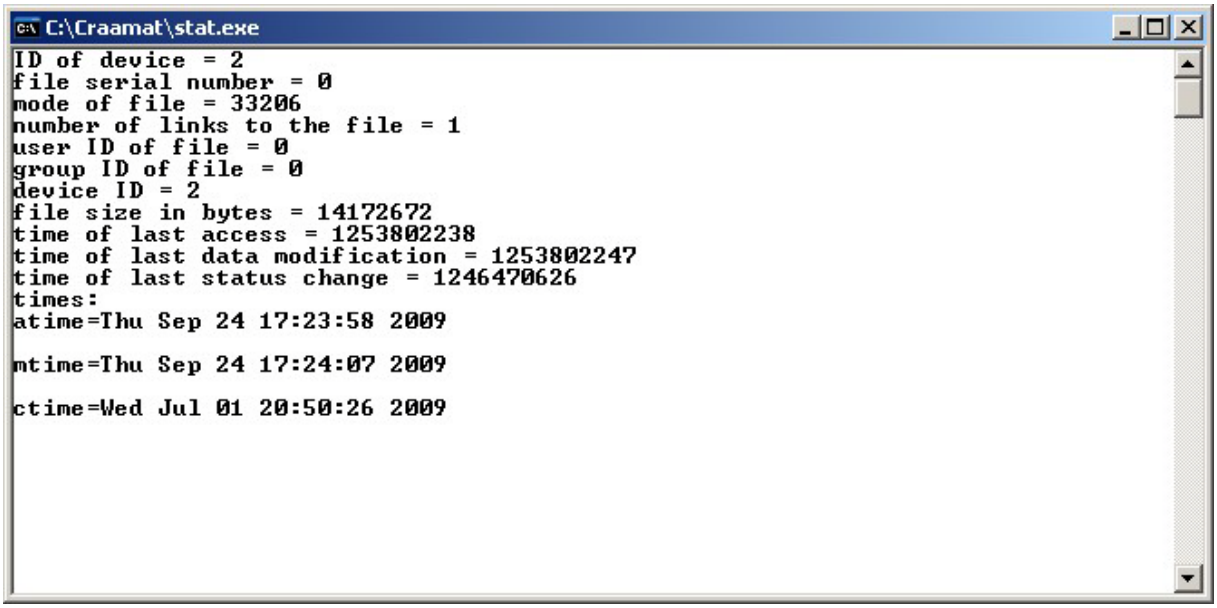
Selle struktuuri kasutamiseks peame tegema järgmist:

- `#include <sys/stat>` peab leiduma programmi alguses;
- kirjeldama muutujad `struct stat stbuf;` ja (näit.) `int pikkus;`
- kasutama funktsiooni `stat(*filename,&stbuf);`
- `pikkus= stbuf.st_size;`

Näiteprogramm *stat.c* trükib välja meie raamatu teksti faili-informatsiooni.

```
//stat.c :: struktuuri stbuf trykk. 24.09.09
#include <stdio.h>
#include <sys/stat.h>
int main( ){
    struct stat stbuf;
    stat("PC0207.doc",&stbuf);
    printf("ID of device = %d\n",stbuf.st_dev);
    printf("file serial number = %d\n",stbuf.st_ino);
    printf("mode of file = %d\n",stbuf.st_mode);
    printf("number of links to the file = %d\n",stbuf.st_nlink);
    printf("user ID of file = %d\n",stbuf.st_uid);
    printf("group ID of file = %d\n",stbuf.st_gid);
    printf("device ID = %d\n",stbuf.st_rdev);
```

```
    printf("file size in bytes = %d\n",stbuf.st_size);
    printf("time of last access = %d\n",stbuf.st_atime);
    printf("time of last data modification = %d\n",stbuf.st_mtime);
    printf("time of last status change = %d\n",stbuf.st_ctime);
//bloki-info pole Dev-C++ realisatsioonis kirjeldatud
//    printf("block size = %d\n",stbuf.st_blksize);
//    printf("number of blocks = %d\n",stbuf.st_blocks);
    printf("times:\n");  //vt. lk. 56
    printf("atime=%s\n",ctime(&stbuf.st_atime));
    printf("mtime=%s\n",ctime(&stbuf.st_mtime));
    printf("ctime=%s\n",ctime(&stbuf.st_ctime));
    getchar( );
}
```



Joonis 3.5.1.1.1.a. Programmi *stat.c* töö tulemused.

3.5.1.2. Sümbolite sisestamine ja väljastamine (konsool)

Meenutame, et *konsool* on *PC* standardse sisendseadme – klaviatuuri – ja väljundseadmeks oleva kuvari (displei) ühine nimetus[1]. Ning et klaviatuurilt sisestatu kopeeritakse ekraanile. Funktsioonid on järgmised:

1. Sümboli sisestamine: `int getchar(void): c=getchar( );` on samaväärne variandiga `c=fgetc(stdin);`
2. Sümboli väljastamine: `int putchar(int c)`≡ `fputc(c,stdout).`
3. Stringi sisestamine: `char *gets(char *s)` loeb stringi vektorisse *s* ning asendab *ASCII* reavahetussümbolid lõputunnusega '\0'. Vea korral (aga ka tühja stringi sisestamisel[2]) tagastab lipu *NULL*, muidu viida stringile (*s).
4. Stringi väljastamine: `int puts(const char *s)` kirjutab stringi väljundisse, asendades lõputunnuse reavahetusega. Vea korral tagastab lipu *EOF*, muidu mittenegatiivse täisarvu.

[1] Standardsisendi süsteemne nimi on *C*-s *stdin* ning –väljundil *stdout.*
[2] See tagastamine sõltub realisatsioonist. Kindlam on kontrollida, kas sisestati 0-pikkusega string („tühi *Enter*").

3.5.1.3. Formaaditud väljund

Nii formaaditud (ehk vormindatud) väljund kui ka sisend on kõikidest standardfunktsioonidest (peame silmas *ANSI*-versioone) kõige komplitseeritumad. Neil on palju võimalusi ja variante (näiteks, [KjaK-2] tutvustavad neid lk. 48 – 60) ning seetõttu soovitame lugejal küsimuste korral pöörduda *UNIX*-keskkonnas käsu *man* poole, ja suvalises keskkonnas kasutada *Interneti*-otsingut. Tegelik olukord pole nii dramaatiline, selle funktsiooni tavakasutamine saab üpris lihtsalt selgeks. Väljastamisfunktsiooni üldkuju on järgmine:

```
fprintf(seade,“formaat“,argumendid);
```

Näiteks, kui meil on kirjutamiseks avatud avatud faili pide *Logi* ning muutuja *fname* väärtus on *TRI.grm*, siis

```
fprintf(Logi,"CONSTRUCTOR for the Grammar %s",fname);
```
kirjutatab logifaili rea CONSTRUCTOR for the Grammar TRI.grm. Kui kirjutame
```
fprintf(stdout,"CONSTRUCTOR for the Grammar %s",fname);
```
väljastatakse see tekst ekraanile. Sama teeb ka
```
printf("CONSTRUCTOR for the Grammar %s",fname);
```
so. variant *printf* kasutab vaikimisi seadet *stdout*. Ja kui meil on kirjeldatud tekstivektor *rida*[80], siis variant *sprintf* kirjutab meie näiteteksti sinna:
```
sprintf(rida,"CONSTRUCTOR for the Grammar %s",fname);
```

Niisiis, komponent *formaat* kujutab endast teksti, kuhu saab ettenähtud kohtadesse sisse kirjutada argumentide (näites on ainult üks argument, *fname*) tekstilisi väärtusi. Ja kirjutamiskoha määrab meie näites *%s*, kus *s* tähendab, et argumendi väärtuseks on *string*; *%c* teatab argumendi väärtuseks ühe sümboli[1]. Muud variandid on teisendamist vajavate andmetüüpide jaoks[2] on esitatud järgmises tabelis:

sümbol	**argumendi tüüp ja trükiviis**
d, i	*int*, kümnendarv
o	*int*, kaheksandarv
x, X	*int*, kuuteistkümnendarv, vastavalt *abcdef* või *ABCDEF* (numbritele 10..15)
u	*int*, märgita kümnendarv
f	*double*, [-]*m.dddddd*, kus trükitakse vaikimisi 6 kohta peale koma (seda saab muuta)
e, E	*double*, [-]*m.ddddde±xx* (või ’e’ asemel ’E’), *d*-de arv on siingi vaikimisi 6.
p	*void **: viit; kuju sõltub „implementatsioonist“
%	formaadis on see kujul %% ja väljundritta läheb %

Tabel 3.5.1.3.a. Valik funktsiooni *fprintf* tüübiteisendusi.

Formaadi-osas oli meie näites väljastatav tekst koos sellesse lisatava muutuja väärtusega. Neid lisatavaid „asju“ võib olla 0 kuni kuitahes mitu; esimesel juhul argumentide-osa puudub, näiteks siin: `printf(„Tere maakera!\n“).` Kui argumente on rohkem kui üks, siis peavad *formaadi* %-kirjelduste arv ja tüübid täpselt klappima argumentide järjekorra, arvu ja

[1] Kui muutuja on kirjeldatud kui `char c;` ning tahtes teda „välja trükkida“ kirjutame eksikombel `%s`, saame lahendamiseaegse vea: funktsioon ei leia stringi lõputunnust ’\0’

[2] Toome siinkohal ära [K&R] tabeli lk. 154; sama raamatu lisas (lk. 244) on toodud detailsem tabel.

tüüpidega[1]. Ning argument ei pea tingimata olema muutuja, selleks võib olla ka *formaadis* deklareeritud tüüpi avaldis, erijuhul funktsioon. Näiteks:

```
printf(„pikkused: int=%d,long=%d\n“,sizeof(int), sizeof(long));
```

Lõpuks, *formaadi*-osas võib tavatekst sootuks puududa ning seal olla ainult %-väljad.

3.5.1.4. Formaaditud sisend

Formaaditud sisendi funktsioon *fscanf* on süntaktiliselt sarnane *fprintf*-le, sellegi üldkuju on

```
fscanf(seade,“formaat“,argumendid);
```

ja lisaks variandid `scanf(„formaat“,argumendid)`[2] ning `sscanf(väli, „formaat“,argumendid).`

Kui *fprintf* vormistab väljastamiseks tekstirea ja argumendid annavad sellesse teksti lisatavad argumentide väärtused, siis *fscanf* loeb sisse teksti, milles on %-ga märgistatud (võimalik, et teisendamist ja) salvestamist vajavad lõigud ning tulemuseks on väärtustatud muutujad. Seega saavad argumentideks[3] olla ainult kas ilmutatud kujul viidad lihtmuutujatele või vektorite nimed (*C*-s on massiivi nimi samaväärne viidaga).

Me esitame siingi – nagu *fprintf*i tutvustades – raamatust [K&R], lk. 157 jj. lihtsustatud käsitluse (täisversioon on seal lk. 245..246), vt. tabel 3.5.1.4.a.

Raamatus [K&R, lk. 159] on toodud näide, kus klaviatuurilt sisestatakse kuupäev ameerikapärases vormis *mm/dd/yy* , näiteks 09/14/09:

```
int day, month, year;
...
scanf(„%d/%d/%d“,&month,&day,&year);
```

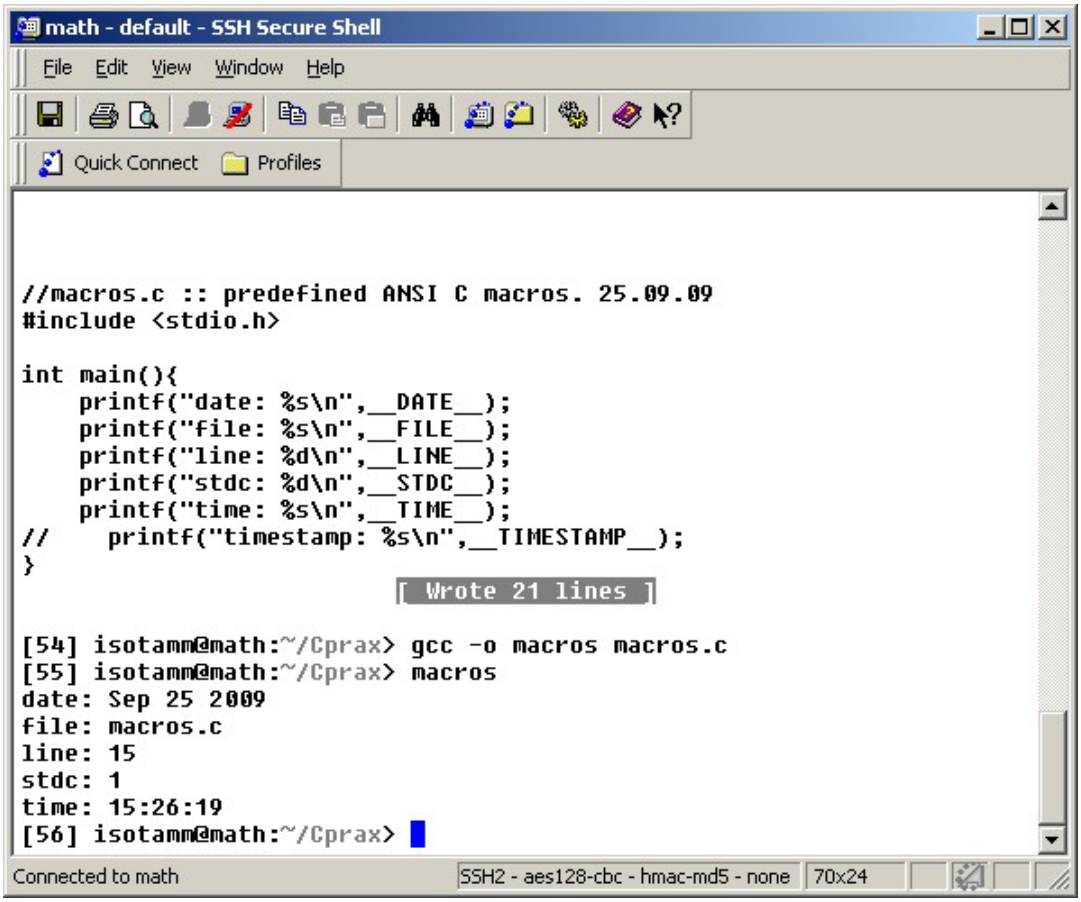
sümbol	**sisendandmed: argumendi tüüp**
d	kümnendtäisarv; int *
i	*integer*; *int* *, võib olla *octal* (algab 0-ga) või *hex* (algab 0x või 0X)
o	kaheksandarv (võib alata 0-ga); *int* *
u	märgita kümnendtäisarv; *unsigned int* *
x	16-ndtäisarv (võib, aga ei pea algama 0x või 0X-ga); *int* *
c	sümbol; *char* *, (on võimalus lugeda seeriaviisiliselt, vt. näit. [K&R])
s	string; *char* *, loetakse sõnahaaval (eraldajaks tühik)
e, f, g	ujupunktarvud; *float* *, märk, kümnendpunkt ja eksponent-osa on fakultatiivsed

Tabel 3.5.1.4.a. Valik funktsiooni *fscanf* tüübiteisendusi.

[1] Kui see nii pole, on tagajärjed ettearvamatud; *C* stiil on programmeerijat usaldav ja mittekontrolliv.
[2] See on samaväärne kujuga `fscanf(stdin,“formaat“,argumendid).`
[3] Erinevalt *fprintf*ist pole varianti, kus argumendid võivad puududa.

3.5.1.5. Standardmakrod

Gnu-keskkonnas (*gcc*) on teegis <*stdio.h*> kirjeldatud „viis identifikaatorit, mis pole makrod tavalises mõttes, kuna vastavaid makrodirektiive kusagil ei eksisteeri. Nad on n-ö. sisemised ehk standardmakrod, mida ei saa tühistada ega ümber defineerida.“ [KjaK-2, lk. 7]



Joonis 3.5.1.5.a. Standardmakrod.

Joonisele mahtusid nii näiteprogrammi tekst kui ka töö tulemused. Nende „identifikaatorite“ semantika on järgmine:

__DATE__ : programmi kompileerimise kuupäev;
__FILE__ : programmifaili nimi;
__LINE__ : sisendprogrammi (*C*-teksti) selle rea number, kus „__LINE__“ on kirjutatud;
__STDC__ : vastavus *ANSI C* standardile: 1 tähendab, et ei kompileerita *C++* - koodi;
__TIME__ : kompileerimise[1] kellaaeg.

Elektroonilises manuaalis [MSDNL][2] on toodud ka kuues standardmakro __TIMESTAMP__ – mis peab väljastama stringi: nädalapäev, kuupäev ja kellaaeg sekundi täpsusega, kuid *gcc*-keskkonnas seda ei ole (seetõttu on ta väljastamiskäsk muudetud näiteprogrammis kommentaariks).

[1] Mõnedes realisatsioonides (vt. [MSDNL]) edastatakse täitmisaegne (jooksev) kellaaeg.
[2] See teabekogu sisaldab informatsiooni *Window*si enda ja sellele platvormile orienteeritud keelte kohta.

3.5.2. Sümboliklassi testid: <ctype.h>

Kaks funktsiooni töötavad ainult kirjatähtedega (A..Z, *ASCII*-koodid 65..90 ja a..z, koodid 97 ..122):

`int tolower(int c)` tagastab *c* väiketähe-koodi, kui *c* oli suurtähe-kood, muidu tagastab *c* muutmata kujul;

`int toupper(int c)` käitub eelkirjeldatud moel väiketähe-koodidega: tagastab võimaluse korral vastava suurtähe-koodi.

Kõik selle teegi ülejäänud funktsioonid testivad etteantud sümbolit (esitatud *ASCII*-koodiga: *unsigned char*) ning tulemus on 0, kui sümbol ei kuulu testitavasse klassi, ja on nullist suurem, kui kuulub. Need on:

`isdigit(c)`	kümnendnumber (0..9);
`isupper(c)`	suurtäht (A..Z);
`islower(c)`	väiketäht (a..z);
`isalpha(c)`	isupper(c) või islower(c) on tõene;
`isalnum(c)`	isalpha(c) või isdigit(c) on tõene;
`isxdigit(c)`	kuueteistkümnendnumber (0..f);
`isgraph(c)`	trükikujuga sümbol, välja arvatud 'tühik' (*ASCII* 32);
`isprint(c)`	trükikujuga sümbol, kaasa arvatud 'tühik' (*ASCII* 32);
`ispunct(c)`	trükikujuga sümbol, välja arvatud 'tühik', täht või number;
`isspace(c)`	tühik, reavahetussümbolid (0a, 0d,0c), tabulatsioonid.

3.5.3. Stringifunktsioonid: <string.h>

Sellesse teeki on koondatud kahte tüüpi funktsioonid: need, mille nime prefiks on *str* ja need, millel selleks on *mem* (vastavalt nagu *str*ing ja *mem*ory). Tugineme [K&R] lehekülgedele 249 kuni 250. Kuivõrd meie raamatus on tagapool vektoritele (ja sh. stringidele) pööratud üsna palju tähelepanu, siis piirdume siin põgusa ja väljavõttelise ülevaatega selle teegi funktsioonidest.

Funktsioonide formaalseid parameetreid tähistatakse raamatus [K&R] kui muutujaid *s* ja *t* (*source* ja *target*, „lähtestring“ ja „resultaat“ tüüpidega *char* *) ning *cs* ja *ct* (kui samu asju modifitseerimiskaitsega, *const char* *).

Meenutagem, et *C*-string on indekseeritav, lõputunnusega ('\0') baidivektor (ühemõõtmeline *unsigned char* tüüpi massiiv), mille elemendid on ühebaidilised märgita *int*-tüüpi täisarvud. Kui me tahame seda interpreteerida trükitava ja loetava tekstina, siis on nendel baitidel paiknevate *ASCII*-koodide aktsepteeritavad väärtused vahemikust 32_{10} (tühik) kuni 126_{10} (~). Lisanduvad eriotstarbelised koodid, nagu lõputunnus '\0', reavahetus '\n', tabulatsioonid, helisignaal jmt.

3.5.3.1. *str*-funktsioonid

Allpool me ei tutvusta kõiki *ANSI*-funktsioone (sh. neid, kus operand(id) on pikkusepiiranguga, näiteks esimese funktsiooni erim on `strncpy`, mis kopeerib *ülimalt n* sümbolit[1]).

`char *strcpy(s, ct)` : kopeerimine (*ct* kirjutatakse *s*-i väljale koos lõputunnusega);

`char *strcat(s, ct)` : konkatenatsioon (teine string liidetakse esimesega, st. kirjutatakse selle lõppu);

`int strcmp(cs,ct)` : võrdlemine (kui cs < ct, siis resultaat on < 0, kui cs = ct, siis 0 ja kui cs > ct, siis resultaat on > 0);

`char *strchr(cs,c)` : tagastab viida `c` esimesele esinemisele `cs`-is (või `NULL`, kui ei ole);

`char *strrchr(cs,c)` : tagastab viida `c` viimasele esinemisele `cs`-is (või `NULL`, kui ei ole);

`size_t strspn(cs,ct)`: tagastab `cs`-prefiksi pikkuse, mis koosneb `ct` sümbolitest (või 0);

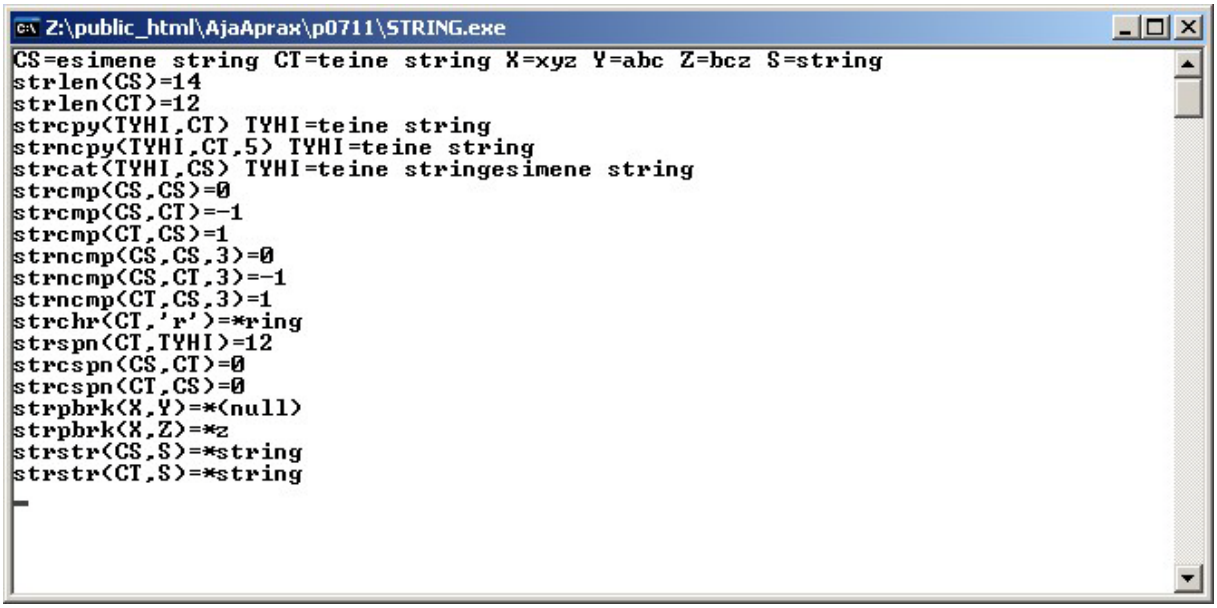
`size_t strcspn(cs,ct)`: tagastab `cs`- prefiksi pikkuse, mis ei koosne `ct` sümbolitest (või 0);

`char *strpbrk(cs,ct)` : tagastab viida esimesele `cs` sümbolile, mis sisaldub `ct`-s (või `NULL`);

`char *strstr(cs,ct)` : tagastab viida esimesele `cs` sümbolile, millest algab `ct` `cs`-i alamstringina (või `NULL`);

`size_t strlen(cs)` : tagastab stringi pikkuse (lõputunnust arvestamata);

Nende funktsioonide kasutamise näited on toodud joonisel 3.5.3.1.a.



Joonis 3.5.3.1.a. *str*-funktsioonide näiterakendusi.

[1] Lõputunnus '\0' kirjutatakse *s*-i ainult siis, kui string *s* on lühem kui *n* .

3.5.3.2. *mem*-funktsioonid

Kui *str*-funktsioonide parameetrid on *char*[ ]-tüüpi baidivektorid, siis *mem*-funktsioonid[1] on mõeldud mistahes tüüpi andmestruktuuride jaoks, so. nende funktsioonide ning nende parameetrite tüübiks on *void*. Stringifunktsioonide hulka kuuluvad nad seetõttu, et objektidega manipuleeritakse baithaaval just nagu stringidega.

Allpool tähistavad *s* ja *t* objekte tüübiga *void **, *cs* ja *ct – const void **, *n* tüüp on *size_t* (praktiliselt sama, mis *int*) ning *c* tüüp on *unsigned char*.

`void *memcpy(s,ct,n)` : kopeerib *n* esimest baiti *ct*-st *s*-i, tagastatakse *s*;
`void *memmove(s,ct,n)` : nagu *memcpy*, ent töötab ka siis, kui väljad on ühisosaga;
`int memcmp(cs,ct,n)` : *cs* ja *ct* baidikaupa võrdlemine, resultaat on nagu funktsioonil *strcmp*;
`void *memchr(cs,c,n)` : otsib baidi *c* esimest esinemist *cs*-is, tagastab kas viida või lipu *NULL* (so, tühiviida);
`void *memset(s,c,n)` : kirjutab väljale *s n* baiti *c* ja tagastab viida *s*-le.

Nende funktsioonide puhul võib keeruliseks osutuda välja pikkuse *n* kirjutamine (see peab olema baitides), kasvõi siis, kui me töötame erineva arhitektuuriga masinatega, näiteks 32- ja 64-bitistega ning tahame „nullida“ 512-elemendilise *int*-vektori *V*. Üks variant oleks muidugi kirjutada omistamistsükkel, ent lihtsam on kasutada *int*-tüüpi funktsiooni *sizeof*, nii, nagu alljärgnevas näites:

```
memset(V,'\0',512*sizeof(int));
```[2]

3.5.4. Matemaatilised funktsioonid: <math.h>

Funktsioonide kirjeldustes järgib [K&R, lk. 252 jj.] järgmisi tähistusi: *x* ja *y* on (tavaliselt) *double*-tüüpi ja kõik funktsioonid tagastavad *double*-tüüpi väärtuse. Kollektsioonis on trigonomeetrilised funktsioonid[3] *sin*(*x*), *cos*(*x*), *tan*(*x*), *asin*(*x*), *acos*(*x*), *atan*(*x*), *atan2*(*x*), *sinh*(*x*), *cosh*(*x*), ja *tan*(*h*) ning lisaks järgmised:

`exp(x)` : eksponentfunktsioon e^x;
`log(x)` : naturaallogaritm $ln(x)$, $x > 0$;
`log10(x)` : kümnendlogaritm, $x > 0$;
`pow(x,y)` : x^y viga tekib juhul, kui $x=0$ ja $y \le 0$ või kui $x<0$ ja *y* pole *int*-tüüpi;
`sqrt(x)` : $\sqrt{x}$, $x \ge 0$;
`cell(x)` : väikseim täisarv, mis pole väiksem kui *x*;
`floor(x)` : suurim täisarv, mis pole suurem kui *x*[4];
`fabs(x)` : *x* absoluutväärtus, $|x|$.

---

[1] Tugineme selles jaotises raamatu [K&R] leheküljele 250.
[2] Tähelepanu, kui kirjutate '\0' asemel '0', siis saate lahendamisaegse vea *segmentation fault*: esimese täitesümboli pikkus on 1 bait ja teise oma 4 baiti. Kirjutate „üle otsa“.
[3] Lisateabe saamiseks soovitame kasutada *UNIX*i-keskkonnas funktsiooni *man* abi.
[4] Puhtas *UNIX*-keskkonnas siin (*math.h*) seda funktsiooni pole, ent on näit. *Dev-C* omas.

3.5.5. Üldised (*utility*) funktsioonid: <stdlib.h>

See standardfunktsioonide komplekt[1] sisaldab tüübiteisendusi, dünaamilist mälujaotust jt. üldotstarbelisi vahendeid. Me ei tutvusta kõiki sellesse teeki kuuluvaid funktsioone, täisversiooniga tutvumiseks soovitame *UNIX*-käsku *man stdlib.h*. Lühitutvustus:

`double atof(const char *s)`teisendab (väljal *s*) tekstikujul esitatud ujupunktarvu masinkujule;

`int atoi(const char *s)` teisendab (väljal *s*) tekstikujul esitatud täisarvu masinkujule;

`int rand`[2]`(void)`genereerib *pseudojuhuslikke int*-tüüpi arve vahemikust 0..*RAND_MAX* (see konstant on fikseeritud teegis *limits.h*). Iseärasuseks on tõik, et ta genereerib *alati* sama jada (ükshaaval, igal järgmisel pöördumisel jada järgmise liikme); alustab *seed*=1 (vt. *srand*).

`void srand(unsigned int seed)` erineb *rand*ist selle poolest, et talle antakse ette võimalikult juhuslik lähteväärtus *seed*, ent iga tolle algväärtuse kordumisel genereeritakse ikka ja alati üks ja seesama jada (ükshaaval nagu funktsiooni *rand* puhulgi).

`void *malloc(size_t size)` küsib lahendamise ajal operatsioonisüsteemilt *size* baiti mälu; kui anti, siis resultaat on saadud mälu algusaadress, kui ei, siis lipp *NULL*. Tüüp *void* tähendab siin, et nii saab mälu küsida *suvalisele* andmestruktuurile, mille pikkust baitides saab edastada mh. funktsiooni *sizeof* abil. Funktsioon tagastab viida struktuuri algusbaidile.

`void free(void *p)` vabastab (tagastab operatsioonisüsteemile) viidaga *p* seotud dünaamilise mälu (mis on saadud funktsiooniga *malloc*).

`void abort(void)` katkestab programmi töö ja annab juhtimise operatsioonisüsteemile signaaliga „ebanormaalne lõpp“ (võimaluse korral on eelistatum funktsiooni *return* kasutamine);

`void exit(int status)` lõpetab samuti programmi täitmise, ent tagastab lõpukoodi *status* (*ANSI* sätestab, et *status*=0 on eduka lõpu tunnus);

`int system(const char *s)` edastab operatsioonisüsteemile *käsurea* teksti[3] *s*. Süsteem täidab käsu ning tagastab juhtimise *system*-käsule järgnevale direktiivile. Tavaline on, et *s* tõstetakse kokku funktsiooniga *sprintf*, nagu järgmises näites.

Oletame näiteks, et meil tekib tahtmine oma programmi täitmise ajal „võtta aeg maha“, tuues *UNIX*i tekstitoimeti *pico* abil[4] selle programmi teksti ekraanile, kas vaatamiseks või – miks ka mitte – modifitseerimiseks. Meie programmis on kirjeldatud vektor *KR* („käsurida“):

```
char KR[40];
...
```

---

[1] Vt. [K&R], lk. 251 jj.

[2] *rand* on lühend sõnast *random*, ingl. k. *juhuslik.*

[3] Vt. osa 3.2. Seal on kirjeldatud käsurea argumentide jaoks loodud tugi: argumentide arv *argc* ja viitade vektor argumentide nimedele *argv*.

[4] Näiteks, kui meie programmi nimi on *auto.c*, siis ekraanile saame ta käsurea `>pico auto.c` abil. Programm *auto* käivitatakse käsurealt kui `>auto` : nimelaiendita ning `argv[0]` väärtus on string `auto.`

```
sprintf(KR,“pico %s.c“, argv[0]);
system(KR);
```
Selles teegis on teisigi (huvitavaid) funktsioone; siinkohal tutvustame neist jagamise tulemuse täis- ja murdosa leidmise funktsiooni (resultaat on `div_t`-tüüpi struktuur väljadega `quot` (jagatis) ja `rem` (jääk):

```
div_t div(int num,int denom),
```

kus *num* on jagatav, *denom* on jagaja. Näiteks, kui on kirjeldatud `div_t J;` ning *a* väärtus on 23, siis kirjutades
```
J=div(a,7);
```
omistatakse *J.quot* = 3 ning *J.rem*=2.

3.5.6. Ajafunktsioonid: <time.h>

See funktsioonide komplekt[1] pakub võimalusi manipuleerida nii kuupäeva kui ka kellaajaga ja ajavahemikega. Kui kuupäev ja kellaaeg on üheselt määratletavad ja mõistetavad (vt. struktuuri *tm* pisut allpool), siis programmi töötamise aja mõõtmine seda ei ole. *GNU* manuaalis [GNU C L, lk. 442] juhitakse tähelepanu tõigale, et tuleb vahet teha riistvarakomponendi *timer* (mille aeg[2] kulgeb nagu kalendriaeg, pidevalt ja ühtlaselt, vt. ka [CC], lk. 65), ja mõiste *clock* vahel, mis on seotud protsessi (programmi) tegeliku *CPU* kasutamisega[3]. St., kui me protsessorit omas programmis ei koorma, siis meie *clock*-näit ei kasva.

ANSI-standard kirjeldab järgmisi funktsioone[4] (vt. [K&R], lk. 255 jj.): *clock*, *time*, *difftime*, *mktime*, *asctime*, *ctime*, *gmtime*, *localtime* ja *strftime*. Neist kolm esimest on mõeldud programmi lahendamiseks kulunud aja fikseerimiseks, ülejäänud aga struktuuri *tm* väljade kasutamiseks. Allpool püüame selgitada lahendusaja mõõtmise variante ning tuua üks-kaks näidet ülejäänute peale kokku.

3.5.6.1. Kalendriaeg

Andmestruktuur ajafunktsioonide jaoks on platvormidest sõltumatu *tm*, nn. *kalendriaeg* (mille väärtsustab *C* virtuaalarvuti):

```
struct tm{
     int tm_sec;  //sekundid pärast minutit(0..59)
     int tm_min;  //minutid pärast tundi(0..59)
     int tm_hour; //tunnid alates kesköösT (0..23)
     int tm_mday; //päeva jrk-nr. kuus (1..)
     int tm_mon;  //jaanuar: 0, detsember: 11
     int tm_year; //aastad alates 1900-st: -101=1799, 109=2009
     int tm_wday; //pühapäev=0,.. laupäev=6
     int tm_yday; //1. jaanuar on 0, viimane detsember 365|366
     int tm_isdst; //kui 0-st erinev, siis päevaaeg[5]
     };
```

[1] Tugineme [K&R] lk. 255 jj, [KjaK-2], lk. 77 jj. ning [wTimes], lk. 442 jj.
[2] See on *protsessori*- (*CPU*-)aeg.
[3] See on nn. *protsessi aeg*, so. antud programmi käskude täitmiseks kulutatud aeg; siia ei kuulu näit. pausikäsuga *sleep* tekitatud ooteaeg. Verbaalse sarnasuse tõttu on oht *protsessori*- ja *protsessiaega* segi ajada.
[4] Ei pruugi kõikides realisatsioonides ühtemoodi töötada.
[5] Vt. [KjaK-2] lk. 80.

Kalendriaja tutvustamisel tugineme allikale [GNU C L, lk. 442 jj.]. Ilmselt *tm* mõõdab absoluutset aega (kitsamalt ja meie jaoks protsessoriaega: kui paneme oma programmi käima 22. sept. 2009 kell 18.39.35 ja ta lõpetab 22. sept. 2009 kell 18.41.06, siis absoluutajas võttis ta täitmine aega 1'31''. Ehkki protsessoriajast kulutati meie programmi (kui *protsessi*) täitmiseks näiteks vaid 7,112 sekundit). Seda lahutamistehet teeb funktsioon *difftime.*

Funktsioonid *time* ja *difftime* võimaldavad fikseerida ajavahemiku, mille jooksul meie programm oli täitmisel (ükskõik, mida ta sel ajal tegi, kas koormas protsessorit või oli ooteasendis), so. fikseerida „kalendaarset" *protsessori-* (*CPU-*)aega.

Funktsioon `time_t time(time_t *cp)` tagastab jooksva *kalendriaja* (vea korral –1). `(double) difftime(time_t time2,time_t time1)` tagastab pöördumiste aegade vahe sekundites (resultaat on reaalarv).

GNU C keskkonnas (kompilaator gcc) on *kalendriajast* kolm varianti:

- *Lihtne aeg* (tüübiga *time_t*): (tavaliselt) sekundite arv alates mingist nullpunktist (näit. 1.01.1900, kell 00.00). Seda kasutades saame oma programmi täitmise kuupäeva ja kellaaja (minuti täpsusega), allpool toome näite.
- *„Peenaeg"* (*high-resolution time*), mis kasutab struktuuri *timeval* ning võimaldab opereerida „*lihtsa aja*" ühikutega kui reaalarvudega (sekundi murdosadega). Kuivõrd selle abil me ei saa mõõta *protsesside* täitmise aegu, vaid summaarset *protsessori*ae-ga, siis meiesuguste (rakendus)programmeerijate jaoks pole see variant huvitav.
- *Kohalik* (*local time*) aeg, mis arvestab ajavööndidega. See võib korrigeerida nii kuupäeva kui ka kellaaega (aga mitte minuteid, sekundeid jne). Meie (eesti programmeerijate) jaoks on see oluline funktsioon, kuivõrd selle abil saame varustada oma väljatrükke (tüüpiliselt logi-tekste) meie ajavööndi tegelike kuupäevade ja kellaaegadega.

Kalendriajast on *C virtuaalmasin* võimeline konverteerima jooksva kuupäeva ja kellaaja. Me esitame allpool nii vastava programmi *timm1.c* kui ka ta lahendamise „protokolli" ekraanipildi näol.

Selles näiteprogrammis kasutame funktsiooni

```
char *asctime(const struct tm *tp),
```

mis konventeerib *tm*-struktuurist infot tekstikujule. Tavaliselt kasutatakse argumendi rollis teist teisendusfunktsiooni, näiteks `localtime.`

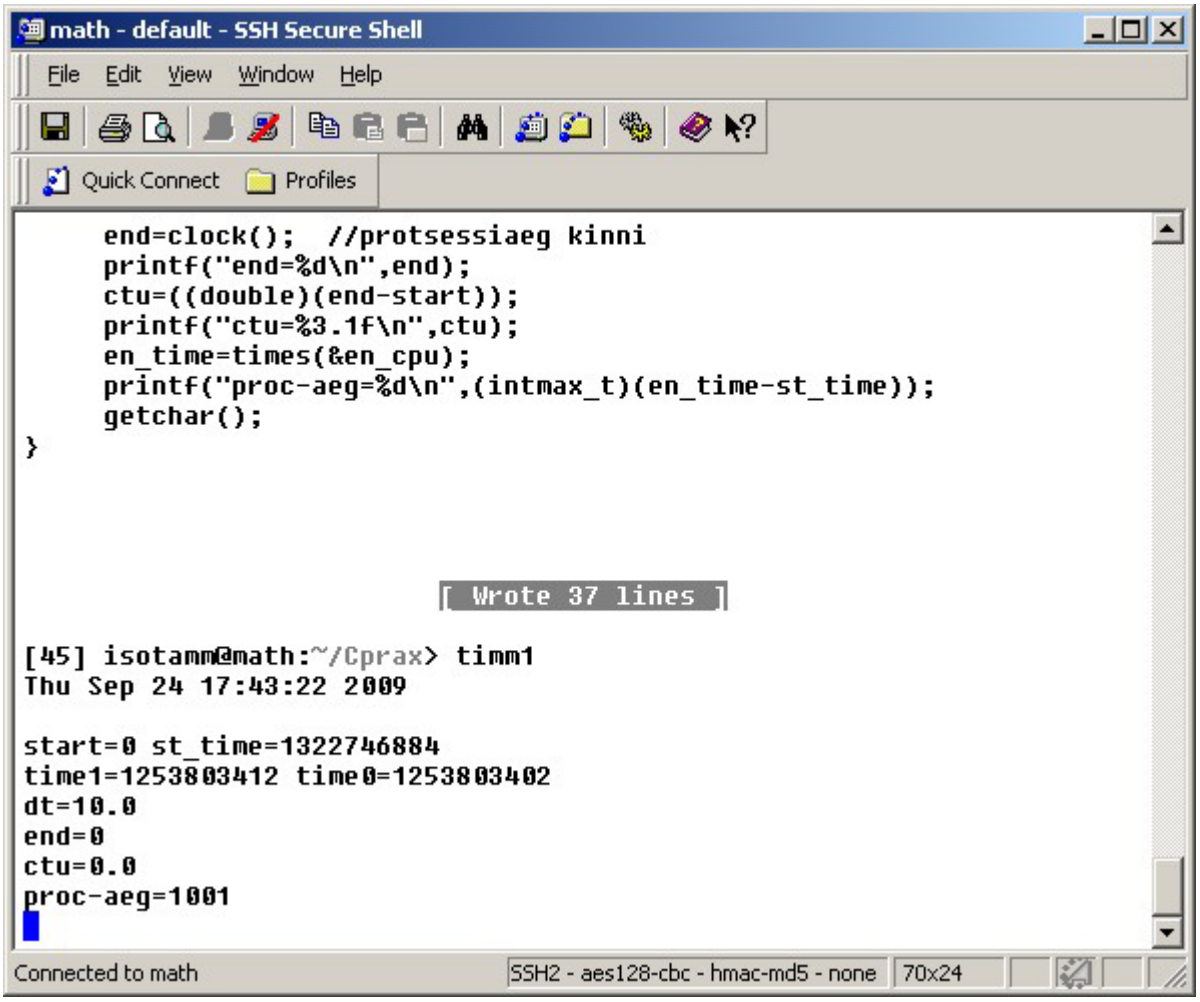
Funktsioon `char *ctime(const time_t *tp)` saab ette viida kalendriajale ning teisendab selle stringiks (nagu *asctime*, mis teeb ise jooksva kalendriaja). Näidet vt. jaotises 3.5.1.1.1, kus on toodud programmi *stat.c* tekst ja lahendamistulemuste ekraanipilt (joonis 3.5.1.1.1.a.).

Lisaks *ANSI*-vahenditele on näiteprogrammis *timm1* kasutusel struktuur *tms*, mis on defineeritud *gcc*-keskkonnas ja selle otstarve on protsessiaja info hoidmine. Selle programmi teksti esitame kohe ja lahendustulemuste slaidi järgmisena (joonis 3.5.6.1.a).

```
//timm1.c:  timer.h ANSI-standardfunktsioonid (K&R, lk. 256 jj.)
//20.09.09, lisaks gcc-vahendeid (GNU C L, lk. 442 jj.)

#include <sys/times.h>   //gcc
#include <time.h>
#include <stdio.h>
#include <unistd.h>      //gcc

     time_t t0,t1;    //cpu-aja start ja end
     clock_t start,end,pt,st_time,en_time;  //protsessiajad
     double dt,ctu;   //DiffTime, Cpu_Time_Used
     struct tms *buffer;   //tms:  gcc
     struct tms st_cpu;
     struct tms en_cpu;
```



Joonis 3.5.6.1.a. Kalendriaeg: programmi *timm1* käitlemise ekraanipilt.

```
int main( ){
     time(&t0);  //CPU-stopperi start
     printf("%s\n",asctime(localtime(&t0)));
     start=clock( );   //protsessiaja start
     st_time=times(&st_cpu);  //protsessoriaeg
     printf("start=%d st_time=%d\n",start,st_time);
     sleep(10);   //protsessor ei tegele protsessiga 10"
```

```
    time(&t1);   //CPU-stopper kinni
    printf("time1=%d time0=%d\n",t1,t0);
    dt=difftime(t1,t0);
    printf("dt=%3.1f\n",dt);
    end=clock( );  //protsessiaeg kinni
    printf("end=%d\n",end);
    ctu=((double)(end-start));
    printf("ctu=%3.1f\n",ctu);
    en_time=times(&en_cpu);
    printf("proc-aeg=%d\n",(intmax_t)(en_time-st_time));
    getchar( );
}
```

Ülaltoodud programmis kasutasime funktsiooni *sleep*[1] ning näitlikustasime protsessori (*CPU*) -aja (mõõtühikuks *CLK_TCK*) ja protsessi-aja erinevust.

Kalendriaja-info kasutamiseks on olemas funktsioon *strftime* (*str* nagu *string*, *f* nagu *format*), mis saab andmed struktuurist *tm* ning kasutab andmete väljastamiseks *printf*-funktsiooniga sarnast mehhanismi. Selle funktsiooni kirjeldus on [K&R] lk. 256 ja 257 ning tema kasutamist näitlikustab programm *strf.c* (allikas on kommentaarireas ja tõlge meilt) ning selle lahendamistulemustega joonis on 3.5.6.1.b.

```
// http://www.thinkage.ca/english/gcos/expl/c/lib/strfti.html strftime
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int main( ){
char s[240];
size_t i;
struct tm tim;
time_t now;
now = time(NULL);
tim = *(localtime(&now));

strftime(s,240,"lyhip=%a pikkp=%A, lyhikuu=%b; kuu=%B\n",&tim);
printf("%s",s);
strftime(s,240,"kuup=%c p2evanr=%d\n",&tim);
printf("%s",s);

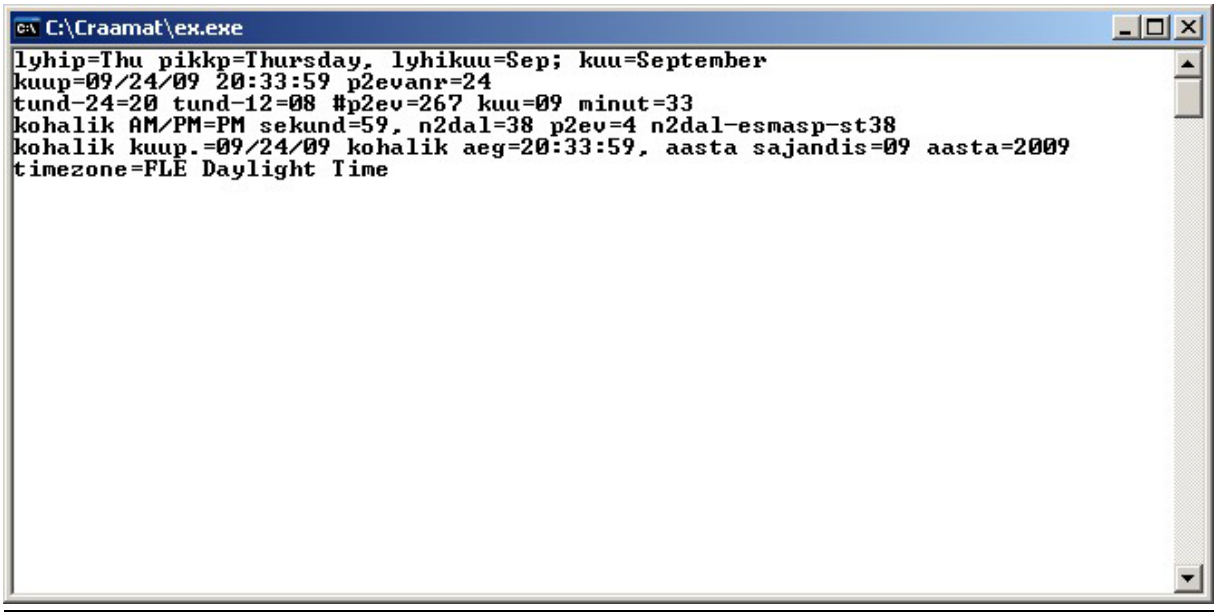
strftime(s,80,"tund-24=%H tund-12=%I #p2ev=%j kuu=%m minut=%M\n",&tim);
printf("%s",s);

strftime(s,80,"kohalik AM/PM=%p sekund=%S, n2dal=%U p2ev=%w n2dal-esmasp-
st%W\n",&tim);
printf("%s",s);

strftime(s,80,"kohalik kuup.=%x kohalik aeg=%X, aasta sajandis=%y
aasta=%Y\n",&tim);
printf("%s",s);

strftime(s,80,"timezone=%Z\n",&tim);
printf("%s",s);
getchar( );
}
```

[1] Selle kirjeldus on toodud [GNU C L, lk. 467]: `sleep(unsigned int seconds)`: näidatud arvu sekundite jooksul on protsessoril „vabad käed“ tegeleda muude töödega ning meie programmi (protsessi) jaoks kulutatud aja arvestamine on noiks *sekundeiks* peatatud. Aeg käib edasi *sys/times.h*-s defineeritud keskkonnas.



Joonis 3.5.6.1.b.Funktsiooni *strftime* kasutamine.

3.5.6.2. Protsessiaja mõõtmine

Ajavahemiku mõõtmiseks on niisiis kaks võimalust: me mõõdame kas protsessori (*CPU*-) aega või protsessiaega, so. aega, mis tegelikult kulus meie programmi (=protsessi) täitmiseks.

```
clock_t clock(void)
```

võimaldab fikseerida protsessi täitmise tegelikku aega (mitte *CPU* kalendaarset aega).

Tüüp `clock_t`[1] pole täpselt standardiseeritud. 16-bitise arhitektuuri ajal oli see tavaliselt *unsigned long int*, 32-bitise puhul *unsigned int.* Protsessiaja mõõtmiseks tuleb fikseerida „stardiaeg“ ning „vahe- või lõpptulemuste“ ajad (või aeg) – ükskõik, kas see fikseerida protsessori-taktides (*CLK_TCK*) või mikrosekundites (või reaalarvulistes sekundites).

`clock( )` omistab `clock_t`-tüüpi muutuja väärtuseks täitmise hetke jooksva väärtuse[2]. Kui see funktsioon pole millegipärast käivitatav, siis tagastatakse väärtus -1. Allpool toome näiteprogrammi stopperi kasutamise kohta; kaks kõige sisemist tsüklit on kirjutatud „protsessorit-koormavaks täitmiseks“[3].

```
//timm.c:  timer (protsessi-aeg). 21.09.09
#include <time.h>
#include <stdio.h>
     int i,j,k,a=11;
```

[1] „Kellatüüp“ on tavaliselt realiseeritud kui (*long* või *unsigned long*) *int*, seega 32-bitise arhitektuuri korral *uint. Kelder* ja *Kaasik* [KjaK, lk. 77] juhivad tähelepanu seigale, et aega mõõta saab ainult neis piires, mis kulub mõõtmiseks eraldatud välja täitmiseks, näiteks [KjaK] kirjutamise ajal oli 32-bitise „stopperivälja“ ületäitumiseni kuluv aeg *ca* 36 minutit (kaasaegsetel masinatel).

[2] Loe: stopperi näidu (fikseeritud ühikutes). Analoogiliselt stopperiga: algseis on 0.

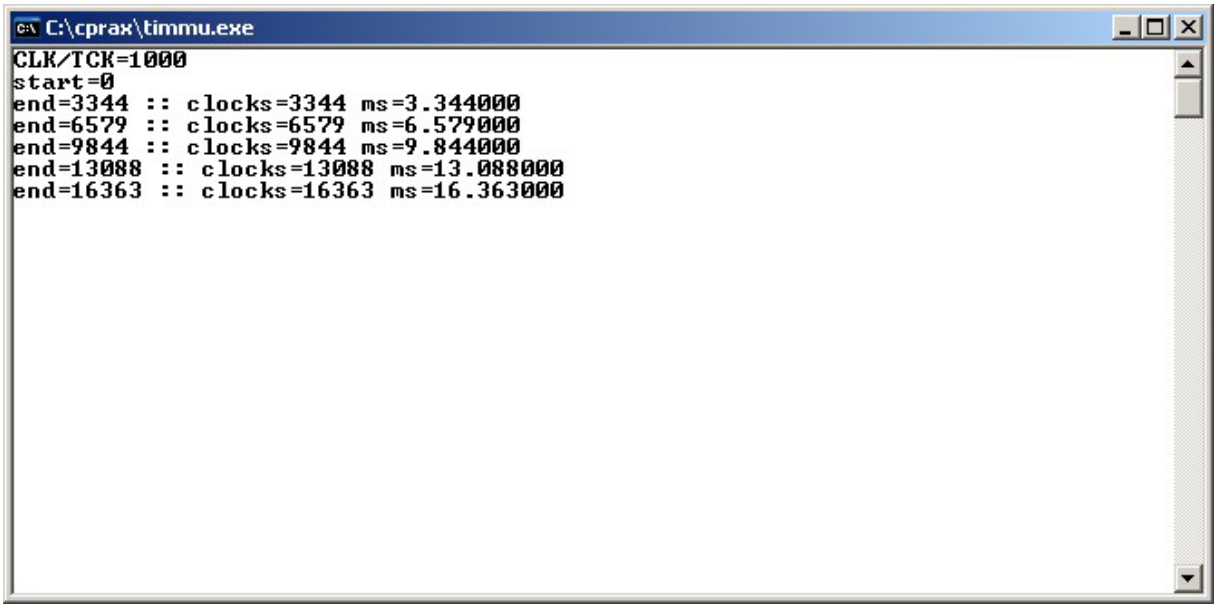
[3] See „venitamine“ koormab protsessorit, seega suurendab ka taimeri („stopperi“) näitu. Kui aga *CPU* meie prog-rammiga ei tegele, siis „stopper“ ei käi (vt. funktsiooni `sleep`).

```
     clock_t start, end;
     int cpu_time_used;

int main( ){
     printf("CLK/TCK=%d\n",CLK_TCK);
     start = clock( );
     printf("start=%d\n",start);
     for(i=0;1<1000;i++){
      for(j=0;j<1000;j++){
          for(k=0;k<100000;k++) a=a*a/a;
          }
      end=clock( );
      printf("end=%d :: ",end);
      printf("clocks=%d ms=%f\n",
             (end-start),((double)(end-start))/CLK_TCK );
      }
     return(1);
}
```

Programm *timm.c* ja tema lahendamisaegsete väljatrükkide ekraanipilt (vt. joonis 3.5.6.2.a) on mõeldud näitama, et funktsioon *clock* toimib tõepoolest stopperina: *start* saab alg-väärtuseks 0 ning *end* omandab protsessi kulgemise käigus uusi väärtusi, ja et stopperinäit „*end – start*“ on (vähemalt *gcc*-keskkonnas) asendatav muutuja *end* väärtusega. Ent, kui mõõdetav protsess koosneb alamprotsessidest, siis on taimerinäitude vahe kasutamine vältimatu, kui meid huvitavad ka nende alamprotsesside täitmise ajad.

[KjaK, lk. 78] märgivad, et viimases trükikäsus toimub *ilmutatud tüübiteisendus*: siin teostatakse „reaalarvuline jagamine“, tänu konstruktsioonile (*double*)(...).



Joomis 3.5.6.2.a. Protsessi täitmise aja mõõtmine (*timm.c* ekraanipilt).

Niisiis, *clock*-funktsiooni abil saame mõõta meie programmi kui *protsessi* poolt tegelikult kasutatud protsessoriaega.

4. Andmestruktuurid: sissejuhatus

4.1. Väljad

4.1.1. Lihttüübid

C-keeles on ainult kaks elementaarandmetüüpi (ehk lihttüüpi):

- märgiga või märgita[1] *täisarv*, mis kujutatakse homogeense bitijadana; kui protsessor „eeldab", et tegemist on märgiga arvuga, siis märgina interpreteeritakse arvu vasakpoolseima biti väärtust: kui see on 0, on arv positiivne, kui 1, siis negatiivne;
- *ujupunktarvu* (*float*-tüüpi) väli on struktuurne: seal paiknevad järgu märk, järk, mantissi märk ja mantiss.

Täisarvude tüüp jaguneb olenevalt välja pikkusest neljaks alamtüübiks:

- *char* pikkusega 1 bait. Vaikimisi interpreteeritakse seda tüüpi välja väärtust kui märgita täisarvu (0..255), mis mh. võivad esitada *ASCII* sümbolite koode. Näiteks, 41_{16}='A', 123_{16}='{' jne, ent mingilgi moel pole keelatud ühebaidilisi koode interpreteerida kui täisarve, märgita vahemikus $0..255_{10}$ ja märgiga vahemikus -127.. +127 (ikka kümnendarvud). Näiteks, *char a;* kirjeldab 8-bitilist mäluvälja, millel kujutatud bitijada interpreteeritakse kui sümbolite koode (märgita täisarvud) või täisarve vahemikust 0..255, ja *signed char b;* deklareerib, et muutuja *b* väärtusi interpreteeritakse kui täisarve vahemikust -127..127.
- Ülejäänud *integer*- (ehk *int*- ehk täisarvu-) tüübid on kahe- ja enamabaidilised, kusjuures tüübi tegelik pikkus oleneb objektmasina arhitektuurist, ent mõneti ka realiseerija suvast. *Tõnis Kelder* kirjutas [TK89, lk., 60 jj.]: „*Märgita täisarvutüüpe* on keeles *C* kolme pikkusega, mida pikkuste mittekahanevas järjekorras kirjeldatakse võtmesõnadega *short*, *int* ja *long*…Kirjeldaja *short int* on samaväärne kirjeldajaga *short* ja *long int* samaväärne kirjeldajaga *long*…Need kolm tüüpi ei pea aga igas realisatsioonis tingimata esitama erineva pikkusega täisarve – pikkused sõltuvad konkreetsest arvutist[2]. Üldnõue on siiski, et *short ei oleks kunagi pikem kui int* ja *int pikem kui long*".

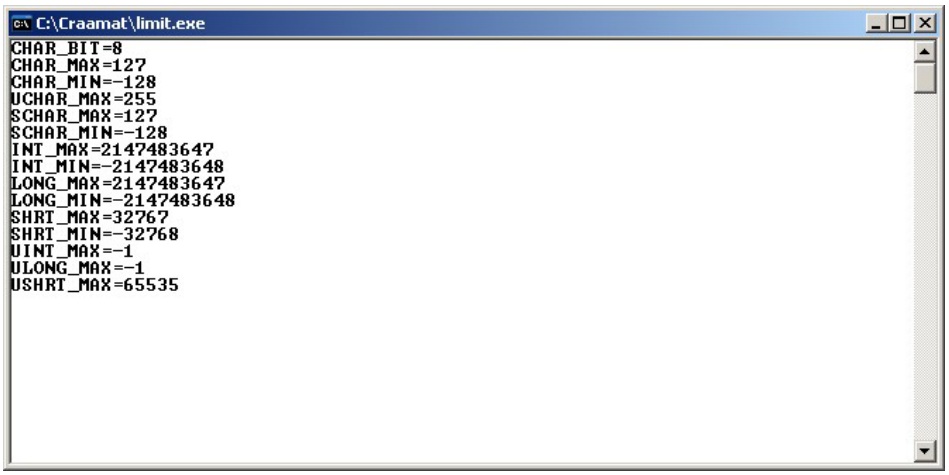
Masintsõltuvate konstantide väärtused on kasutajale kättesaadavad teegist *limits.h.* Näiteks, 32-bitise arvuti täisarve iseloomustavaid konstante (vt. [K&R], lk. 257) trükitakse välja programmiga *limit.c* ning selle lahendust illustreerib joonis 4.1.1.a.

```
//limit.c :: limiite failist limits.h trykk. 17.10.09
#include <stdio.h>
#include <limits.h>
int main( ){
    printf("CHAR_BIT=%d\n",CHAR_BIT);    printf("CHAR_MAX=%d\n",CHAR_MAX);
    printf("CHAR_MIN=%d\n",CHAR_MIN);    printf("UCHAR_MAX=%d\n",UCHAR_MAX);
    printf("SCHAR_MAX=%d\n",SCHAR_MAX);  printf("SCHAR_MIN=%d\n",SCHAR_MIN);
    printf("INT_MAX=%d\n",INT_MAX);      printf("INT_MIN=%d\n",INT_MIN);
```

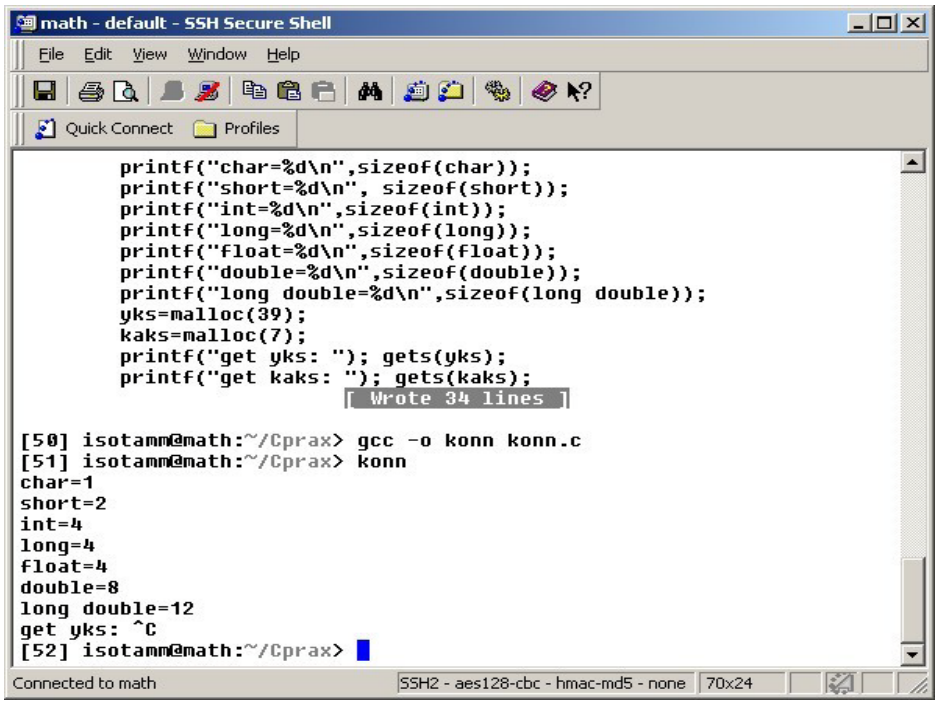
[1] Vaikimisi eeldatakse, et väljal on *märgiga* täisarv; kui tahetakse kirjeldada arve vahemikust 0..*max*, siis tuleb tüüpi määrava sõna ette kirjutada *unsigned* või prefiks *u.* Erand on ühebaidine täisarv (*char*): tavaliselt interpreteeritakse seda märgituna, ja kui seda ei soovita, tuleb lisada täiend *signed.*

[2] 16-bitisel masinal on *char* ja *short* 1- *int* 2- ja *long* 4-baidine; 32-bitisel vastavalt 1, 2, 4 ja 4 baiti.

```
printf("LONG_MAX=%d\n",LONG_MAX);     printf("LONG_MIN=%d\n",LONG_MIN);
printf("SHRT_MAX=%d\n",SHRT_MAX);     printf("SHRT_MIN=%d\n",SHRT_MIN);
printf("UINT_MAX=%d\n",UINT_MAX);     printf("ULONG_MAX=%d\n",ULONG_MAX);
printf("USHRT_MAX=%d\n",USHRT_MAX);   getchar( );}
```



Joonis 4.1.1.a. *Int*-tüübi modifikatsioonide karakteristikuid (*gcc*-keskkond).



Joonis 4.1.1.b. Pikkused 32-bitise arhitektuuri korral (*gcc*-keskkonnas)[1].

Reaalarvulised tüübid on *float* (üks masinasõna, meie arvutitel 4 baiti), *double* (8 baiti) ja *long double* (nagu nägime, on selle pikkus *UNIX*-keskkonnas 12 baiti.

Omapärast „tühitüüpi“ *void* käsitleme mõnevõrra hiljem.

[1] *DOS*-platvormi „ekraanipildid“ on vaikimisi valge tekstiga mustal põhjal; meie raamatu piltide saamiseks tuleb klikkida hiire paremklahviga *DOS*i akna ülaribal ning avanevate võimaluste hulgast valida *properties*, seal saab paika panna nii tausta kui ka tähtede värvid. Meil on need vastavalt valge ja must.

Konstandid on samuti „lihttüübid“, selles mõttes, et nad hõivavad tüübiga määratud pikkusega mäluvälja ning evivad „nime“[1]. *C*-keeles saab neid defineerida makroga *#define*, näiteks

```
#define pii 3.14
```

pii tüüp on *float*, kuivõrd asendustekstis esineb kümnendpunkt. Täpsemalt, makrokäsku *pii* saab kasutada seal, kus on lubatud ujupunktarv.

C-keeles on defineeritud mõned erisümbolid (paojadadena, *escape sequences*), mida saab kasutada väljastamisel ekraanile[2], sisuliselt on needki konstandid:

- \0 : bait väärtusega 0
- \a : lühike vile
- \b : tühik
- \f : lehevahetus
- \n : reavahetus
- \r : kursor rea algusse
- \t : horisontaalne tabulatsioon
- \v : vertikaalne tabulatsioon
- \\ : üks „\“
- \? : küsimärk
- \’ : apostroof
- \“ : jutumärk
- \o : kaheksandarvu tunnus, näiteks ’\o77’
- \x : kuueteistkümnendarvu tunnus, näiteks ’\xf9’

Kõik need (peale \0) toimivad, näiteks, funktsiooni *printf* jutumärkidega „..“ eraldatud osas. Tundub, et enamik neist on seotud omaaegsete (tänaseks igenenud) terminalide, teletaipide jms. välisseadmetega; nende ridade kirjutaja on kasutanud neist ainult (sageli) ’\n’ ning ’\0’ ja (vahel) ’\a’ ning (mõnikord) kaheksand-. ja kuueteistkümnendarvude trükkimiseks.

[1] Meenutades masinorienteeritud keeli (vt. näit. [Isotamm, PK], lk. 37 jj.) alates assemblerist, on konstant kas ilmutatud kujul mingile aadressile kirjutatud väärtus või siis esineb käsus *literaalina*, so. „käsku kirjutatud“ konstandina. Viimasel juhul „tekitab“ konstandivälja (literaali väärtuse aadressi) kompilaator. *C*-keeles on literaal näiteks „1“ avaldises `a=a+1;` Meil pole mõtet noid „literaale“ käsitleda konstantidena keele mõttes.

[2] [K&R], lk. 38.

4.1.2. Viidad

C-keeles pole „ilmutatud“ **viidatüüpi.** Meenutagem: viit[1] on mingi objekti tegelik ehk absoluutne mäluaadress, masinkoodis:

viit *A*=(*B*)+(*I*)+*D*,

kus (*B*) on baasregistri sisu (absoluutne mäluaadress), näiteks 14 000, (*I*) on indeksregistri sisu (suhteline aadress), näiteks 8, ja *D* on suhteline (baasregistri suhtes) aadress, näiteks 40, ehk „nihe“[2], ja kui nii määratud aadressil paikneb näiteks 16-ndarv 2A = 42_{10}, siis *A*=14048 ning (*A*)[3]=arv $2A_{16}$ ehk 42_{10} ehk sümbol ’*’.

Ja „viit“ on masinkoodi-operandi tegelik, absoluutne mäluaadress, nende aadresside väärtusvaru on vahemikus 0..operatiivmälu maht – 1. Reeglina reserveeritakse mingi arv „väiksemaid aadresse“ 0..? süsteemseks otstarbeks ning nii saab rakendusprogrammides interpreteerida viita *A* = 0 kui „tühiviita“[4].

C-keeles, nagu juba öeldud, pole *ilmutatud viidatüüpi*, ent on olemas kõik võimalused viitade moodustamiseks ja nendega opereerimiseks. Selleks otstarbeks kasutatakse kahte sümbolit: „*“ ja „&“. Alustagem näitest (selles jaotises tugineme kolmele publikatsioonile: [K&R], [KjaK] ning [Jensen]; eriti tahaks soovitada lugejaile tutvuda neist viimasega, kuivõrd see on mõel-dud olema just *õppevahend*).

Olgu programmis kirjeldatud kaks *int*-tüüpi muutujat:
```
int i,j=2;
```
Muutuja *i* väärtus on määramata, *j* on algväärtustatud: väljal nimega *j* on *int*-väärtus 2. Järgmisena kirjeldame välja, mille väärtuseks hakkab olema *viit int-tüüpi muutujale*:
```
int *ip;
```
Välja nimi on *ip* (väli on algväärtustamata) ning seal kavatsetakse hakata hoidma *viita int*-tüüpi muutujale. Meenutagem, et 32-bitises masinas on viida (kui absoluutse aadressi) säilitamiseks vaja nelja baiti.

Kui kirjeldame muutujad `char b`; ning `char *cp`; siis neljabaidisel väljal nimega *cp* hakatakse hoidma viita ühebaidisele objektile[5], ja muutujate *ip* ja *cp* väärtustamiseks võime kirjutada

`ip=&j`; ja `cp=&b`; – nüüd on *ip* väärtuseks muutuja *j* ja *cp* väärtuseks muutuja *b* absoluutne aadress. Harjutamaks viitadest aru saama, võib lugeja viidad *ip* ja *cp* välja trükkida ja tõenäosus, et kahel järjestikusel viidakatsetusprogrammi käivitamisel viitade väärtused on samad, on kaduvväike:
```
printf(„ip=%p cp=%p\n“,ip,cp);
```
Trükkida võidakse näiteks:
```
ip=8047a1c cp=8047a120
```

[1] Ingl. k. *pointer*, vene k. ссылка.
[2] Vt. näiteks [PK, lk. 34 – 35].
[3] (*A*) tähistab kokkulepitult „aadressil *A* olevat „väärtust“.
[4] Tühiviida tekitamiseks tuleb *C*-keeles kasutada makrot *NULL*, ent nagu kogemused näitavad, genereerib see just (meie masinates) 32-bitise sõna, kus kõik bitid on nullid.
[5] Miks viidavälja kirjeldamisel on oluline teatada viidatava objekti tüüpi, näeme jaotises, kus käsitleme vektoreid (ühemõõtmelisi massiive).

Nägime, et „*“ mõte oli teatada kompilaatorile, aga ka programmi inimesest lugejale, et mingi muutuja (*ip*, *cp*) väärtusteks saavad viidad vastavalt *int* ja *char*-tüüpi objektidele. Ent „*“ on kasutatav ka teisiti; meie näidet kasutades avaldise „*ip“ väärtuseks on *ip*-ga viidatud objekti väärtus. Näiteks, kui kirjutame

```
printf(„*ip=%d\n“;*ip);
```

trükitakse `*ip=2` – kuivõrd `ip` viitab muutujale *j* ning selle väärtuseks omistasime äsja 2. Ja kui kirjutame `i=*ip`, siis omistatakse muutujale *i* avaldise **ip* väärtus, so. 2[1].

Viitu lihtmuutujatele kasutatakse tavaliselt infovahetusel alamprogrammidega (eeskätt siis, kui need on teegifunktsioonid). Asi on selles, et alamprogrammile (funktsioon, protseduur, üldistatult – moodul) edastatakse lihtmuutujatest parameetrid[2] eranditult „väärtuste järgi“ (ingl. k, *by value*). Kui parameeter on kirjeldatud kui mingit tüüpi (*char*, *int*, *double* vms) muutuja, siis alamprogrammi jaoks teeb *C*-virtuaalarvuti (vt. [Isotamm, PK]) selle *koopia* (nimi ja väärtus, aga mitte viit) ning alamprogrammil pole mingitki võimalust üle kirjutada sisendparameetri väärtust väljakutsuvas programmis. Kui aga me just seda tahame, siis peame kirjeldama si-sendparameetri kui viida mingit tüüpi muutujale, selle parameetri *väärtus* lahendamise ajal on *viit* ning alamprogrammil on „vabad käed“ viidatud väljale kirjutamiseks. Näiteks sobib formatiseeritud sisendfunktsioon (kirjeldatud teegis <*stdio.h*>) *scanf*, mis sisestab *ASCII*-stringi klaviatuurilt ning (vajadusel) teisendab selle mingiks ettemääratud masinakujul väärtuseks etteantud aadressile. Funktsiooni kirjeldus [K&R, lk. 245] on:

```
int scanf(const char *format,...)
```

näiteks, kui tahame omistada „oma“ muutujale `int i` uue väärtuse klaviatuurilt, peame kirjutama

```
scanf(„%d“,&i);
```

see on: *scanf* nõuab, et aadress, kuhu teisendatakse sisestatud arv, peab olema *viit* ja mitte pelk *C*-programmi muutuja nimi (vastasel korral edastataks *scanf*-le selle muutuja väärtus, ent mitte aadress).

Niisiis, v*iida* tekitamine viitamaks „tavalisele“ mäluväljale on vältimatu, kui alamprogramm peab parameetrina antud (liht)muutuja väärtust muutma. Näiteks, kui kirjutasime mooduli *ap* vahetamaks *a* ja *b* väärtusi:

```
void ap(int a, int b){
     int temp;
     temp=a;
     a=b;
     b=temp;
}
```

siis kutsudes selle välja `ap(x,y)`; ei tee see moodul midagi sisulist, *x* ja *y* väärtused on nagu nad olid. Näide on pärit ([K&R], lk.95 – 96) ning korrektne algoritm on selline (samast allikast):

```
void swap(int *px, int *py){
     int temp;  //abimuutuja
```

[1] Avaldis **identifikaator* annab viidatud objekti väärtuse siis, kui ta on *rvalue* rollis.

[2] Kui parameeter on massiiv, string või struktuur, siis ta edastatakse viidana (*by pointer*); vältimaks võimalikku ülekirjutamist tuleb formaalne parameeter varustada reservsõnaga *const* (kompilaator saab nii käsu kontrollida võimalikku viida järgi kirjutamist ja see „ära keelata“).

```
temp=*px;  //temp=väärtus aadressilt px
*px=*py;  //aadressile px kirjutatakse py-l olev väärtus
*py=temp;      }
}
```

Võtkem kokku: *C*-keeles me saame kirjeldada (ja hiljem väärtustada) muutujaid, mille väärtuseks on viit kirjeldatud tüüpi (*char*, *short*, *int* jne) objektile; viit *ise* on 32-bitise arhitektuuri puhul alati 4-baidine. Seik, et viit on seotud viidatava objekti tüübiga, kitsendab viitadega manipuleerimise võimalusi (võrdlemine, omistamine, aritmeetika), samas on paljudel juhtudel vajalik, et too kitsendus ei kehtiks. Seetõttu on keelde lisatud andmetüüp *void* (vt. [KjaK], lk. 79): sel pole ei objekti ega seega ka operatsioone ning *void* * defineerib viida suvalisele objektile; sedatüüpi muutujale võib omistada viida suvalisele tegelikule objektile[1]. Tavaliselt kasutatakse universaalse viidana viita lühimat tüüpi (*char*) objektile, so. *char **; näidet vt. vektorite (täpsemalt, stringide) käsitlemise juures – *void*-tüüpi kasutab mh. funktsioon *memset*, võimaldamaks „nullida“ (täita „masinanullidega“, so. baitidega, mille kõikide bittide väärtusteks on nullid) suvalist kõrgemat-kui-elementaartüüpi mäluvälja 1-baidiliste 0-väärtustega; vt. ka teisi *mem*-funktsioone.

Sootuks suurem roll on viitadel *agregeeritud andmestruktuuride* puhul ning me naaseme nendega seotud seikade juurde tagapool korduvalt.

Subjektiivse märkusena mainigem, et viidad on *C*-keele haavatav koht. Keele eeskujudeks olid *Algol* (ja mõneti ka *FORTRAN*), milles polnud viidatüüpi, ehkki *Algol* kasutas täiel määral dünaamilist mälujaotust. *C* süsteemiprogrammeerimise keelena pidi tahes-tahtmata seda tüüpi toetama – nagu seda tegid loomulikult masinkoodid ja assemblerid. Ent eeskuju-keeled ei andnud mingeidki vihjeid, kuidas seda teha. Ja nii on klassikalises [K&R] raamatus viidaasjandusele pühendatud terve 5. peatükk (lk. 93 – 126), kusjuures praktikas kipub ka neist seletustest vahel väheks jääma. Võrdluseks meenutame, et vaid pisut vanema sama suunitlusega (so. süsteemiprogrammeerimise) keele *Forth* jaoks polnud see üldse probleem: muutuja *nimi* **A** on *alati* aadress (so, viit) ning **A @** on muutuja *A* väärtus; ükskõik, kas operand ise või jälle aadress, viimasel juhul tuleb tollelt aadressilt (vajadusel) taas lugeda, kasutades uuesti *sõna* @[2]. Ja see ongi „kõik viitadest“ *Forth*is: 33 lehekülje asemel pool lauset.

Viitadest tuleb veel juttu osas 5.1, kus käsitleme vektoreid (ühemõõtmelisi massiive).

4.2. Kasutaja-andmetüübid

Kasutaja-andmetüüpe, so. programmeerija defineeritud tüüpe võime pidada sillaks lihttüüpide ja programmeeritavate agregeeritud andmetüüpide vahel. *C*-keeles on sellisteks *struktuuritüüp* (*structure*) ja *ühenditüüp*[3] (*union*)

Struktuuritüüp. Tsiteerigem ([KjaK], lk. 71): „Struktuuritüüp on keeles *C* suurel määral sarnane tüüpidega, mida mõningates teistes programmeerimiskeeltes nimetatakse kirjeteks (*record*): struktuur on kogum nimedega varustatud komponentidest, mis võivad üldiselt olla eri-

[1] Samas, kui alamprogramm on kirjeldatud kui *void*-tüüpi funktsioon, siis see tähendab, et kirjeldatud on protseduuri, mis ei tagasta oma väärtust, ja teda ei saa kasutada avaldistes operandina.
[2] Näiteks, kui *Forth*-programmis on tekst *A* @ @ @, siis kantakse magasini viit aadressile *A*, seejärel viit magasini tipus olevalt aadressilt ja siis magasini tipust viidatud „objekt“: „arv“ või – miks ka mitte – taas viit.
[3] Järgime ([KjaK], lk. 71..77) terminoloogiat.

nevat tüüpi. Struktuurid annavad programmeerijale teatava võimaluse abstraktsete andmetüüpide realiseerimiseks. Tüüpiline näide on kompleksarvu defineerimine“[1]:

```
struct complex{
     double real; //reaalosa
     double imag; //imaginaarosa
     };   //NB! Struktuuri kirjeldamisel on lõpetav „;“ kohustuslik. Translaator viga ei tuvasta, ent
          //struktuuri täitmise ajal kasutada pole võimalik.
```

Märgime (mõneti etteruttavalt), et *struktuuri* komponendid võivad olla mistahes tüüpi, ka rekursiivselt omaenda, so., defineeritava struktuuri tüüpi.

struct complex on *tüübi* nimi, viita sedatüüpi objektile võime kirjeldada kui

```
struct complex *c;
```

ning moodulis, millele see struktuur parameetrina edastatakse, saab struktuurikomponente adresseerida kui *c->real* ja *c->imag* (metasümbol „->“ kirjutatakse kui „miinus“ ja „suurem“). Juhime tähelepanu tõigale, et *struktuuriga* kirjeldatud mäluväli edastatakse viida abil.

Struktuuridega on seotud mitmeid formaalseid kitsendusi, mida me siinkohal kommenteerima ei hakka; probleemide puhul soovitame lugeda ([KjaK], lk. 74). Märkigem, et struktuurid tavakasutajale probleeme ei tekita.

Ühenditüüp[2] on sellise andmevälja tüüp, kus võib hoida lahendamise ajal erinevat tüüpi andmeid (lihtandmeid või nende jadasid); juhul kui need erinevad andmed pole sama pikkusega, siis välja pikkuseks reserveeritakse pikima variandiga määratu. Näiteks:

```
union u{
     int ival;
     float fval;
     char *sval;
     };
```

kirjeldab *ühte* 4-baidise mäluvälja, millele saab vastavalt olukorrale viidata kas kui `u.ival`, `u.fval` või `u.sval`.

Üks näide, kus ühenditüüp on silmnähtavalt kasulik, on toodud raamatus [Isotamm PK, lk. 167], viidates *Robert Laforele*[3] – alltoodud programm toob ekraanile operatiivmälu mahu[4]:

```
#include <dos.h>
#include <bios.h>
#define MEM 0x12  //BIOS-katkestuse number
int main( ){
```

[1] Tsitaadi lõpp, ent järgnev näide pärineb samast allikast; kommentaarid on meilt.
[2] vt. [K&R], lk. 147 – 148.
[3] vt. [Lafore, lk. 327]
[4] meenutagem, et *DOS*- ja *BIOS*-tase oli kättesaadav esimestes *C*-„laiatarbe“-versioonides, näiteks *Borlandi Turbo-C*-s. Saamaks teada, kas Teie *C* seda taset võimaldab, kirjutage teksti *#include <dos.h>* ja *<bios.h>*. Kui kompilaator veateadet ei anna, siis on „madala taseme“ operatsioonid kättesaadavad.

```
     struct WORDREGS{   //16-bitised registrid
          unsigned int ax;
          unsigned int bx;
          unsigned int cx;
          unsigned int dx;
          unsigned int si;
          unsigned int di;
          unsigned int flags;
          };

     struct BYTEREGS{   //8-bitised registrid
          unsigned char al,ah;
          unsigned char bl,bh;
          unsigned char cl,ch;
          unsigned char dl,dh;
          };
     union REGS{   //nii baidid kui ka sõnad samal väljal
          struct WORDREGS x;
          struct BYTEREGS h;
          };
union REGS regs;
unsigned int size;
int86(MEM,&regs,&regs);  //regs on nii sisend- kui ka
                         //väljundparameeter
size=regs.x.ax;  //mälu maht edastatakse (ax) väärtusena
printf(„Memory size is %d Kbytes\n“,size);
}
```

On lihtne näha, et ühenditüübi *REGS* pikkuseks on struktuuri *WORDREGS* pikkus. Ja et selle iga 4 esimest kahebaidilist *int*-väärtust interpreteeritakse *BYTEREGS*-variandis kui kahte ühebaidilist väärtust.

4.3. Agregaadid

Suvaline arvutiprogramm esitab mingit konkreetset algoritmi, mille realiseerimiseks tuleb paljudel juhtudel kasutada talle sobival viisil korrastatud andmekogumeid, viimaseid nii lähteandmete kui ka resultaatide jaoks, aga tihti ka lahendamist toetavate andmestruktuuridena, sj. võivad sisend ja/või väljund olla struktureerimata lihtandmete kujul.

Programmeerimises on tavaline, et vähegi komplitseeritumate ülesannete jaoks pole teada ühte ja parimat lahenduskäiku: kas siis veel ei ole või siis ei saa teooria abil tõestada, et see üks oleks parim; ekstreemsetel juhtudel on aga tõestatud, et üldist algoritmi ei eksisteerigi. Ent seni me ei tea, et mingi programm oleks kirjutamata ja vastuvõetavatel tingimustel tööle panemata jäetud noil põhjustel[1].

Programmeerija jaoks on algoritmi realiseerimisel alati oluline fikseerida enda jaoks andmestruktuurid: sisend, vahekuju(d) ja väljund. Seejuures saame pea alati tugineda eelkäijate pikaajalisele kogemusele – millised algoritmid on analoogiliste ülesannete jaoks teada ja millised andmestruktuurid on selliste ülesannete jaoks sobivaks osutunud. Nii näiteks on suvalise

[1] Ekstreemsete juhtumite puhul võib abi olla *heuristika*st (vt. lisa 8).

taseme programmeerimiskeele translaatorile töötlemiseks sobivaim andmestruktuur puu või kiireimale paisksalvestusmeetodile viidad ahelatele.

Tuletagem meelde, et *elementaar*andmed olid *char*, *short*...*double*-tüüpi, ja et need tüübid on *C*-keelde nö. sisse kirjutatud. Ja et deklareeritud tüüpi muutujale saame sättida viida. Ja et kasutada saame tüüpe *struct* ning *union.*

Andmestruktuurid on elementaartüüpide *agregaadid*, pealisehitused neile tüüpidele:

- vektorid;
- kahe- ja enamamõõtmelised massiivid;
- ahelad;
- magasinid;
- puud;
- graafid;
- tabelid.

Keele tasemel toetab *C* staatilisi vektoreid – ühemõõtmelisi massiive, ning staatilisi kahe- ja enamamõõtmelisi massiive. Dünaamilised massiivid nagu ka ülejäänud agregaadid tuleb kasutajal endal luua, tavaliselt standardfunktsiooni *malloc*[1] abil.

Järgnevates peatükkides käsitlemegi ülalloetletud pealisehitusi ehk *abstraktseid andmestruktuure*. „Abstraktseteks“ nimetame neid seetõttu, et abstraheerime konkreetsete struktuuride *füüsilisest esitusest* – nii nimetatakse abstraktsete anmestruktuuride programset esitust. Allpool näeme, et peaaegu kõiki selliseid andmestruktuure saab programmeerida mitmeti. Näiteks, mitmemõõtmelist massiivi võime kujutada *vektor*ina või viidastruktuuri, näiteks *Iliffe*'i vektorite abil.

Allpool püüame tutvustada nii agregaate, neid kasutavaid ülesandeid kui ka mõningaid algoritme.

[1] Sellega saame süsteemilt lahendamise ajal mälu küsida.

5. Vektorid

5.1. Üldist. *Int*-vektorid

Vektor[1] on sisuliselt ühemõõtmeline massiiv, mis koosneb *n* samatüüpi andmeelemendist ja iga neist on adresseeritav *indeksi i* ($0 \le i \le n-1$)[2] abil. Kuivõrd vektor on indekseeritav andmestruktuur, siis füüsiliselt kujutatakse teda *n*-elemendilise sidusa mäluväljana.

Keele tasemel võimaldab *C* manipuleerida ainult staatiliste objektidega, ja vektoreid saab kirjeldada ainult „sissekirjutatud“ pikkusega, näiteks

```
int a[20];  või  char b[20];
```

Näiteks, järgmine moodul omistab etteantud *int*-vektori elementidele väärtusteks nende *indeksid* ja väljastab need ekraanile:

```
void priv(int n,int A[ ]){
     int i;
     for(i=0; i<n; i++){
          A[i]=i;
          printf(„%d “,A[i]);
          }
     }
```

ja seda võiks meie vektori *a* jaoks rakendada nii:

```
     priv(20,a);
```

Mõistagi, staatilisest vektorist parem variant on kirjeldada ja kasutada *dünaamilisi* vektoreid, näiteks:

```
     int *da;  char *db;
```

ja kui nad mõlemad peavad olema *n*-elemendilised, võtta neile lahendamise ajal mälu nii:

```
     da=(int *)malloc(n*sizeof(int));
     db=(char *)malloc(n);
```

Peatugem kahel andmekirjeldusel: `int a[20];` ja `int *da;` Esimesega antakse teada, et 20-elemendilise *int*-vektori nimi on *a*, teisega aga, et muutuja *da* väärtuseks saab omistada *viida int*-tüüpi muutujale (nagu muide tehtigi mõni rida ülevalpool). Meie jaoks on oluline teada, et *C*-kompilaatorid seovad *vektori nimega* viida vektori esimesele elemendile (indeksiga 0), ja kui kirjutame

```
     da=a;
```

siis nimi *da* dubleerib nime *a*: mõlemad viitavad vektori *a* esimesele elemendile (indeksiga 0). Meenutades viitu tutvustatud jaotist ja erisümbolit “&“, võime ilmselt muutujale *da* omistada sama väärtuse, kirjutades

```
     da=&a[0];
```

seega, et

$a \equiv \&a[0]$

---

[1] ingl. k. *vector*, vene k. вектор.

[2] Füüsilisel tasemel tähistab indeks *i* avaldist *indeks* × *välja_pikkus*, näiteks, kui väli on *int*-tüüpi ja *i*=7, siis aadressile liidetava *i* väärtus on *indeks* 7 × 4 =28. Juhime tähelepanu seigale, et indeksi määramispiirkond 0..*n*–1 on *loomulik* (so, vastab täpselt masinkoodi nõuetele). *C* järgib seda stiili, ent mitu temast vanemat keelt mitte. Asjakohast teavet vt. näiteks [wArray, lõik „*Indexing*“].

Soovitame lugejal kirjutada väikseid testprogramme, kontrollimaks meie väiteid.

Edasi. Kui oletada, et meie vektor *a* oli mooduli *priv* „klient“, siis `a[5]=5` ja `a[0]=0`: vektori *a* elementide väärtusteks on nende elementide indeksid, mis omakorda olid kirjeldatud kui *int i*. Oletagem, et *a* aadress mälus on 200 000; see on üksiti ka *a*[0] aadress. Ja et iga *a* elemendi pikkus on 4 baiti, siis *a*[5] aadress on 200 020 (=200 000+5×4) ning (200 020)=5 (so, arv aadressilt 200 020)[1].

Meenutagem, et kui meil on *int c=11*; *int *cp*; *cp*=&*c*; ning **cp* väärtus (*rvalue*na) on 11. Just samuti, jätkates meie manipuleerimist muutujatega *int a*[20] ja *int *da*, on **da* väärtuseks 0 ning **da*[5] väärtuseks 5 (meenutagem, me omistasime *a*[*i*]=*i*, ja *da*=*a* ja et **da* on *rvalue* rollis).

Oluline on mõista, et indekseerimissamm ei olene mitte *indeks*i, vaid *vektori elementide* tüübist. Vektori *int*-tüübi puhul on samm 4, *char*-tüübi puhul 1 ja *double*-tüübi puhul 8. See seletab ka, miks *viit* tuleb alati deklareerida koos viidatava objekti tüübiga[2]. Andmekirjeldus määrab vektori elementide indekseerimise sammu.

Edasi: vektori elemendi adresseerimiseks saame kasutade nii *int*-tüüpi muutujat kui ka *int*-tüüpi konstanti. Seejuures aadressile liidetav indekskonstant ei modifitseeri aadressi mitte vahetult, vaid seda interpreteeritakse kui „indeks × elemendi_pikkus“; ükskõik, kas *indeks*i väärtus on salvestatud kui *int*-tüüpi muutuja *või konstant.*

Tuletagem meelde sümboli „*“ rolli viitadega manipuleerimisel. Esiteks, andmekirjelduses andis ta kompilaatorile märku, et nii kirjeldet muutuja väärtuseks omistatakse viit deklareeritud tüüpi väärtusele (või – nagu nüüd teame – vektorile), näiteks *int *v*; teiseks, seal, kus operaatori semantika määrab, et *-konstruktsioon esitab operandi *väärtust* viidatud väljalt, tähendabki **v* sel aadressil paikneva lihtmuutuja või vektori elemendi väärtust, ja kolmandaks, indeksitele mõeldes, et *[*a*+*i*] ≡ *a*[*i*] (ehk *[*a*+5] ≡ *a*[*5*]), *-ga tähistatud objektid on siingi *rvalue* rollis (so, omistamisoperaatori „paremas pooles“, *lvalue* on alati absoluutne aadress.).

Mõtelgem viivuks realisatsioonile. Näiteks, kuidas vektorit realiseerida *IBM*/360-süsteemis (vt. näiteks [PK, lk. 65 jj]). Vektori *A* absoluutse aadressi võime kirjutada näiteks registrisse *R*2, indeksi *i* väärtuse registrisse *R*3 ning vektori *A* elemendile indeksiga *i* saame viidata kui väärtusele *x* aadressilt (*R*2)+(*R*3); vahet pole: aadressi võime kirjutada registrisse *R*3 ja indeksi – *R*2. Samamoodi on *C*-keeles ekvivalentsed avaldised (meie *int*-vektori *a* jaoks):

a[3] ja 3[*a*] (*indeks* on 3 ja *aadress* on *a*).

Ülalpool väitsime, et kui on kirjeldused *int a*[ ]; *int *da*; siis pärast *da=a*; viitavad nii *a* kui ka *da* ühele ja samale objektile. Ja, et omistamisoperaatori „paremas pooles“ esinev **da* väärtuseks saab lahendamise ajal „arv“ tollelt aadressilt (kuhu viitab **da*).

Niisiis, *da* on kompilaatori jaoks pärast omistamist **da=a* sama, mis vektor *a*. Viimast saame keele vahenditega indekseerida, kusjuures, nagu nägime, võrdub indeksi *int*-samm 4-baidise

[1] Masinorienteeritud keeltes *a* või 123000 tähistasid mäluaadresse, so. viitu, ning (*a*) või (123000) bitijadasid neilt (viidatud) aadressidelt.

[2] Mõningatel juhtudel on kasulik deklareerida viit *tühiobjektile* tüübiga *void* (seda tüüpi kasutatakse tavaliselt kirjeldamaks mooduleid, mis pole funktsioonid, so. selliseid, mis tekitavad kõrvalefekte, selmet tagastada funktsiooni väärtust. Vt. näiteks [K&R, indeksi järgi]. *void*-viit on 32-bitises masinas 4 baiti pikk.

sammuga (näiteks, kirjutades *a*[2], *a*[7] . või *a*[*i*], 0≤i≤19). Aga edasi: avaldise operandina (*rvalue*) tähistab **da* operandi jooksvat väärust aadressilt (**da*), näiteks pärast *a*[ ] elementidele nende indeksitega võrdsete vääruste omistamist on **da* vääruseks 0, ja aadress *da*[1] on just sama, nagu *a*[1] (=1); seega saame dünaamilist vektorit *da* indekseerida just samuti nagu tema staatilist ekvivalenti *a*: *a*[*i*]≡*da*[*i*].[1]

Ent naaskem *-sümboolika juurde. *rvalue*-mõttes tähistab **da* viidaga *da* osutatud väärtust, so., kui *lvalue* (vt. ülaltpoolt näiteid) on aadress 200 020, siis *rvalue* on tollel aadressil olev bitijada „0..0101“. Jätkem see meelde.

Ülalpool esines meil kirjeldus *int *da*; ning omistasime *da=a*; (*a* on kirjeldatud kui *int a*[20]). Mis tähendab, et *da* väärtuseks on *viit* vektori *a* esimesele elemendile (*a*[0]). Selle viida väärtuse suurendamine 1 võrra, operaatoriga **da*++ tähendab viida seadmist vektori järgmisele elemendile, so. kuivõrd tegemist on *int*-tüüpi vektoriga, siis füüsilisele viidale liidetakse 4 ja mitte 1. Kui me aga tahame elemendi *a*[0] – teisiti tähistatult **da* – väärtust suurendada 1 võrra, siis peame kirjutama (**da*)++.

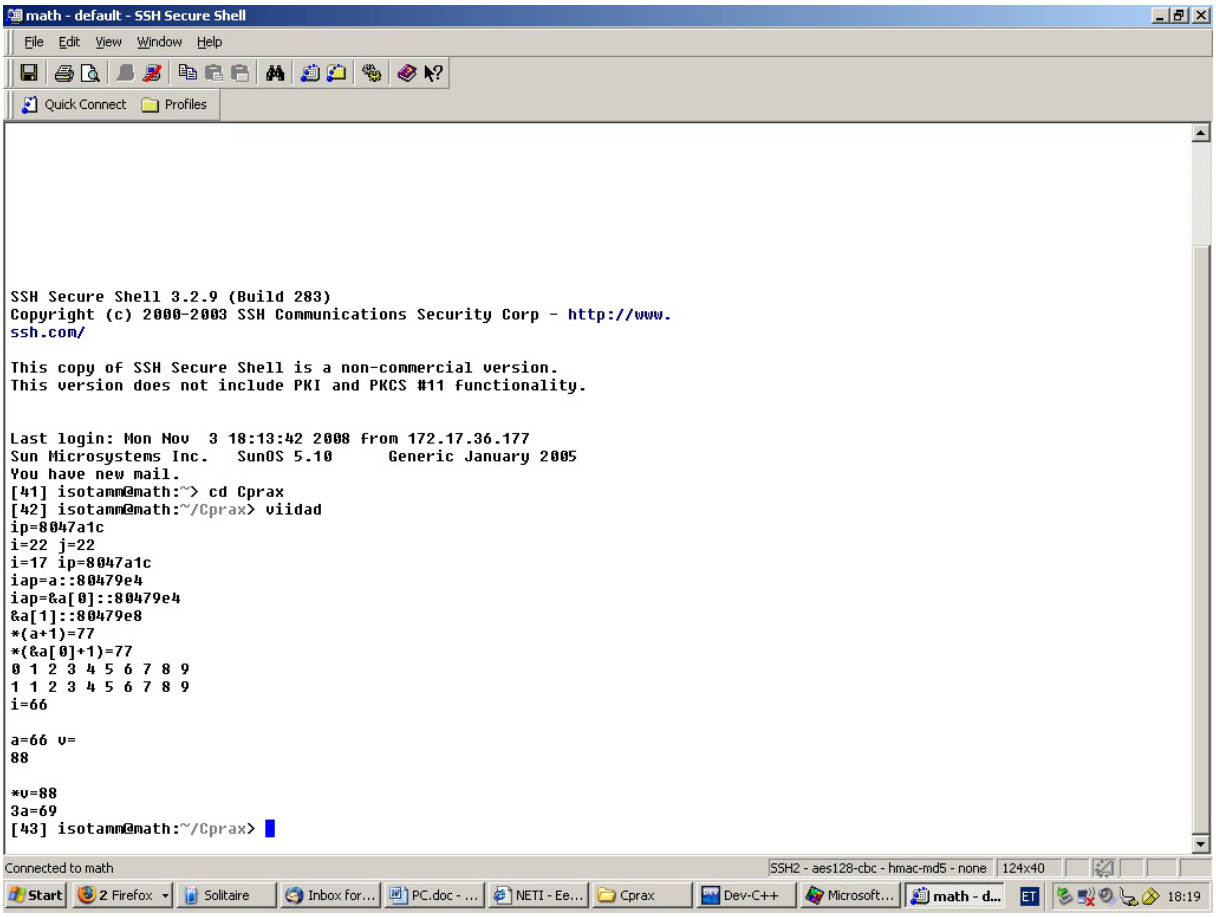
Andmaks lugejale võimaluse hakata katsetama tööd viitadega, toome allpool ühe testprogrammi teksti, kuhu soovitame selsamal lugejal lisada suvalisi operaatoreid eesmärgiga tunnetada *C* viidamajandust ja kontrollida käesoleva raamatu väiteid (ja näiteid).

```
// viidad.c T. Kelder, Programmeerimiskeel C, Tartu 1989, lk
//65 ja 67, K&R jt.
#include <stdio.h>
#include <stdlib.h>

int main( ){
     int *v;
     int i,j,*ip;
     int a[10],*iap;
     ip=&i;  // ip viitab muutujale i
     printf("ip=%p\n",ip);
     i=22; j=*ip;  // j value=22
     printf("i=%d j=%d\n",i,j);
     *ip=17;  // i value=17
     printf("i=%d ip=%p\n",i,ip);
     iap=a;  // iap viitab a esimesele elemendile
     printf("iap=a::%p\n",iap);
     iap=&a[0];  // sama, mis iap=a
     printf("iap=&a[0]::%p\n",iap);
     iap=&a[1];  // iap=a teise elemendi aadress
     printf("&a[1]::%p\n",iap);
     a[1]=77;
     i=*(a+1);  //a[1] on sama mis *(a+1)
     printf("*(a+1)=%d\n",i);
     i=*(&a[0]+1);  // seegi on sama, mis a[1] ja *(a+1)
     printf("*(&a[0]+1)=%d\n",i);
//   *iap--; printf("%d %p\n",*iap,iap);
```

[1] Seega, kui *da*=(*int* *)*malloc*(..), siis *da* on *int*-tüüpi väärtuste vektor, kusjuures indeks muutub sammuga 4 (meie tänapäevaste protsessorite *int*-tüüpi muutuja pikkusest tingitult).

```
    v=(int *)malloc(10*sizeof(int));
    for(i=0;i<10;i++) v[i]=i;
    for(i=0;i<10;i++) printf("%d ",v[i]);
    printf("\n");
    (*v)++;  //v[0]=v[0]+1
    for(i=0;i<10;*v++,i++) printf("%d ",*v);
    printf("\ni=");
    scanf("%d",a);  //scan aadressile a[0]
    printf("\na=%d v=\n",a[0]);
    scanf("%d",v);  // scan aadressile v
    printf("\n*v=%d\n",*v);
    3[a]=69;  //a[3] on sama, mis 3[a]
    printf("3a=%d\n",a[3]);
}
```



Joonis 5.1.a. Programmi *viidad.c* väljatrükid.

5.2. Algoritme tööks vektoriga

Reaalsetes programmeerimisülesannetes pole sugugi harv juhtum, kus töödelda tuleb vektori abil esitatud andmeid, toogem mõned näited:

- Võrreldakse kaht erinevat sama tööd tegevat andmetöötlusmeetodit selgitamaks välja efektiivsem (näiteks, kiiremini töötav), kumbagi testitakse näit. 100 korda ning lahendusajad salvestatakse kahte vektorisse (mõlemad pikkusega 100). Kiirema meetodi empiiriliseks leidmiseks tundub olevat loomulik, et leitakse kummagi meetodi keskmine lahendusaeg, selleks aga peame esmalt leidma mõõtmistulemuste aritmeetilised keskmised. Seega: on vaja algoritme jada liikmete väärtuste summeerimiseks.
- Teatavasti tehakse turuhindade statistikat *moodkeskmist*[1] kasutades; selle leidmiseks tundub parima variandina vaatlusandmete eelnev järjestamine, näiteks hindade mittekahanevasse järjekorda.

Mõnikord on vaja teada, kas vektori kõik elemendid on *unikaalsed*, st. ei leidu kaht sama väärtusega elementi. Kui selgub, et jada on kordumisteta, võib see olla positiivne algus näiteks *tabeli* moodustamiseks (kui testiti võtmete jada, teatavasti tabeli võtmed *peavad* olema unikaalsed[2]), aga vahel on positiivne resultaat hoopis korduvate elementide leidmine: *D. Knuth* kirjutab oma raamatu 5. peatüki [Knuth III, lk. 13] motos[3]: „„Aga me ju ei jõua kõiki autonumbreid üle vaadata“, vaidles Drake vastu. „Aga me ei peagi seda tegema, Paul. Me lihtsalt paneme nad järjekorda ja otsime ühesuguseid“, vastas Perry“.

- Vektorite (elementide väärtuste järgi) järjestamise mõttekuse pooltargumente leiame *D. Knuth*i raamatu III köite [Knuth III, lk. 14] algusest: vaatlusandmete grupeerimiseks on hea, kui samased väärtused on jadas kõrvuti; kui on vaja paralleelselt töödelda kaht sama tüüpi ja semantikaga (ent erinevaid kogumeid kajastavat jada) on see töö järjestatud jadade puhul lineaarse keerukusega, ja järjestatud vektorist elemendi otsimine (kas kuulub või ei kuulu jadasse) on kiire.

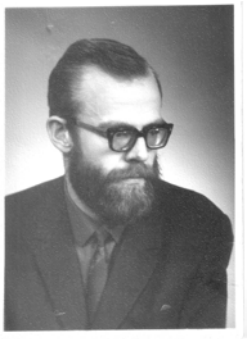
Üldjuhul on järjestamine, otsimine jms. seostatav rohkem *tabelite* kui vektoritega, ent vastavad algoritmid on lihtsamini hoomatavad vektorinäidetel. Ning kursuse „Algoritmid ja andmestruktuurid“ materjal abstraheeribki vektorkäsitluse kasuks. Kusjuures, selle kursuse eesmärk *pole* parimate algoritmide tutvustamine (ons see võimalikki?), vaid *ajalise keerukuse* tunnetamise õpetamine.

5.3. Ajaline keerukus ja kiirushinnang

Joonis 5.3.a.*Jüri Kiho*, aastal 1975, Elbis (Pärnu lähistel) lauatennist mängimas.

[1] Meenutagem, see on mingi kauba (mõistlikult ümardatud) kõige sagedamini registreeritud hind.
[2] Selles mõttes sarnaneb *tabel relatsiooniga* vastavast andmebaasimudelist.
[3] *Knuth* tsiteerib *Erle Stanley Gardner*i „kriminulli“ *The Case of Angry Mourner*; kõikide *Gardner*i „juhtumite“ peategelane on väikeste aferistikalduvustega advokaat *Parry Mason* ja temaga teeb tihtipeale koostööd ettevaatlik eradetektiiv *Paul Drake.*

Tavaliselt saab suvalist ülesannet programmeerida mitmeti, ent suurte andmemahtude puhul ilmneb, et mõned programsed lahendused ei rahulda, sest lahendusaeg on ülemäära pikk. Rahuldava algoritmi valikuks on kasulik teada algoritmi *ajalise keerukuse* hinnangut, aga ka *kiirushinnangut.* Kiirushinnang on seotud algoritmi igal sammul sooritatavate tehete arvuga. Kui vektor on suhteliselt lühike, siis võib juhtuda, et halvema ajalise keerukuse hinnanguga algoritm töötab kiiremini kui mõne muu, parema ajalise keerukuse hinnanguga algoritm – seda juhul, kui viimane teeb igal tsüklisammul rohkem ja keerulisemat tööd kui tema halvema keerukushinnanguga konkurent.



Joonis 5.3.b. *Mati Tombak* 1970-ndate alguses.

Nende ridade autor on programmeerijana ajalise keerukuse problemaatikaga üsna kaua seotud olnud, kuuludes *Mati Tombaku* (tema keerukustemaatilistest publikatsioonidest vt. märksõna [*Tombak*]) töörühma. Süvenemata valdkonna matemaatilistesse nüanssidesse, võib väita, et *ajaline keerukus* väljendab algoritmi lahendi leidmise kiiruse hinnangut töödeldavate andmete hulgast (meie peatüki kontekstis vektori pikkusest) sõltuvalt. Näiteks, kui jadas on 100 liiget (n=100) ja töötlus (mingi algoritmiga) võtab aega 1 sekundi, ning kui jada on 1000-liikmeline (n=1000) ja töötlusaeg on 10 sekundit, siis „ilmselt“ lahendusaeg kasvab jada pikkusega samas proportsioonis. Ütleme, et selle algoritmi ajalise keerukuse hinnang on $O(n)$ – ajaline keerukushinnang on *lineaarne*.

Joonis 5.3.c. *Konstantin Tretjakov, TÜ* kraadiõppur (2008), siin koolipoisina.

Niisiis, me võime korrutada jada pikkuse mingi (suvalise) konstandiga c (see konstant võiks olla igal sammul tehtavate elementaaroperatsioonide summaarne ajakulu), ent algoritmi täitmise aja suhet töödeldavate andmete mahtu see konstant (c) ei muuda; ajalise keerukuse hinnang on c väärtusest olenemata ikka $O(n)$. Naaskem *kiirushinnangu* mõiste juurde. *Konstantin Tretjakov* [Tretjakov][1] toob sellise näite: „Oletame, et algoritm A kulutab oma töö tege-

[1] Kuivõrd meie raamat *pole* mõeldud „Algoritmide ja andmestruktuuride“ otseseks õppevahendiks, siis loodetavasti andestab lugeja (iseenesest olulise) ajalise keerukuse temaatikast pikema peatuseta ülelibisemise. Seda

miseks n sammu ning algoritm B $3n+4$ sammu. Kui me võrdleme algoritmide kiirust ühel ja samal masinal, siis ilmselt on A alati kiirem kui B. Kui me aga paneme A jooksma arvutil, mis teeb sammu sekundiga, aga B arvutil, mis teeb neli sammu sekundiga, siis kui $n > 0$ töötab B kiiremini kui A. Seega mingis mõttes on algoritmid A ja B ekvivalentsed" ehk nende mõlema ajaline keerukus on $O(n)$. Ent *A kiirushinnang* on rohkem kui kolm korda parem kui algoritmil *B*. Jätkame *K. Tretjakov*i tsiteerides: „Tehku algoritm A kokku n sammu ning B – n^2 sammu. Siis sõltumata sellest, kui kiire arvuti peal me paneme B jooksma, mingist n-ist alates töötab A kiiremini. Täpsemalt: iga c jaoks leidub selline n_0, et iga $n > n_0$ jaoks $n^2 > cn$."

K. Tretjakov: " Kasulik on osata ülesande püstitusest arvata, mis keerukusega algoritmi nõutakse. Kui on öeldud, et andmete maht on kuni

- 1 000 000, siis peate leidma lineaarse algoritmi (st. $O(n)$, võib sobida ka $O(n \times \log n)$).
- 1 000 – nõutakse[1] ruutkeerukusega asja (st. $O(n^2)$).
- 100 – kuupkeerukus.
- 20 – ilmselt on lahendus seotud kõigi variantide läbivaatusega ning on eksponentsiaalse $O(2^n)$ või faktoriaalse ($O(n!)$) keerukusega.

Seega, kui ülesandes on öeldud, et andmete maht on kuni 1000 ning teie algoritm on $O(n^2)$, siis läheb tal maksimaalse testi jaoks *ca* 1 sekund. Kui aga teie algoritm on $O(n^3)$, siis kukub ta maksimaalses testis läbi."

Näitamaks, *kui* drastiliselt võib meie programmeerimispingutuse mõttekus oleneda algoritmi ajalisest keerukusest, laename tabeli *Jüri Kiho* [Kiho 03, lk. 13] raamatust (joonis 5.3.d).

Tabel 1.2: Erineva keerukusega programmide ajalised piirid (eeldusel, et ühele operatsioonile kulub 1 mikrosekund).

Keerukus	Suurim ülesanne, mille lahendamise aeg < 1 sek.	Suurim ülesanne, mille lahendamise aeg < 1 päev	Suurim ülesanne, mille lahendamise aeg < 1 aasta
n	$n = 1\,000\,000$	$n = 86\,400\,000\,000$	$n = 31\,530\,000\,000\,000$
$n \log_2 n$	$n = 62\,746$	$n = 2\,755\,147\,514$	$n = 798\,160\,978\,500$
n^2	$n = 1000$	$n = 293\,938$	$n = 5\,615\,692$
n^3	$n = 100$	$n = 4\,421$	$n = 31\,593$
2^n	$n = 19$	$n = 36$	$n = 44$
$n!$	$n = 9$	$n = 14$	$n = 16$

Joonis 5.3.d. Tabel *Jüri Kiho* raamatust.

Alfred A. Aho, John E. Hopcroft ja *Jeffrey D. Ullman* [A,H&U, lk. 14] vaatavad probleemi ka teisest küljest. Olgu meil sama ülesande lahendamisel (n>1) valida viie algoritmi – A_1 on keerukusega $O(n)$, A_2 – $O(n \times \log n)$, A_3 – $O(n^2)$, A_4 – $O(n^3)$ ja A_5 ajaline keerukus on $O(2^n)$ – va-

valdkonda käsitletakse nii „A&A" loengutes (vt. [Kiho 03, lk. 11 jj.]) kui ka (eeskätt kraadiõppuritele mõeldud) kursuses „Keerukusteooria" (vt. [Tombak 07]). Siinkirjutajale meeldib „*Kostja" Tretjakov*i teemakäsitlus ning ta loodab, et lugejale ka. Mõistagi, *K. Tretjakov*i (kooliõpilastele ette kandmiseks mõeldud) tööst on siin tsiteeritud vaid vähesed lõigud.

[1] „Nõutakse" tähendab ilmselt seda, et sellise andmemahu korral *ei tohi* kasutada *halvema* hinnanguga algoritmi, mis ei välista parema(te) hinnangu(te)ga algoritmide kasutamist.

hel. Nad osutavad, et eeldusel, et peame hakkama saama suvalise n-ga, siis algoritm A_5 on neist parim, kui $2 \le n \le 9$, A_3, kui $10 \le n \le 58$, A_2, kui $59 \le n \le 1024$ ja A_1, kui $n > 1024$.

Nentigem, et ajaline keerukus on *asümptootiline* hinnang: ta avaldub seda ilmsemalt, mida rutem $n \to \infty$.

Selles jaotises – kus me tegeleme *arv*vektoritega – vaatleme ülesandeid, mille maksimaalne (kiirus)hinnang on ruutkeerukus (keerulisemaid tutvustame tagapool, *heuristikat* käsitleda püüdes, ent seal andmed *pole vektorid*). Ning *stringid* (sümbolvektorid) pakuvad algoritmiliselt rohkemat kui arv-vektorid. Neidki asju püüame käsitleda hiljem, aga – liikugem edasi sammhaaval.

5.4. Jada elementide järjestamine

Me võime jagada järjestamismeetodid kahte klassi *universaalsed* ja *kitsendustega* meetodid[1]; esimesed on kasutatavad mistahes (arvuti või programmeerimiskeele poolt toetatava) vektorielemendi *tüübi* (*int, real, char* jne) puhul, teised aga tavaliselt ei sobi reaalarvude (*C* tüübid *float* ja *double*) ning mõned neist „teistest“ ei sobi ka märgiga täisarvude järjestamiseks.

Universaalsete järjestamismeetodite parim keerukushinnang on $O(n \times \log n)$ ning aeglasemad (ent lihtsamini programmeeritavad) on ruuthinnanguga ($O(n \times n)$) algoritmid.

Kitsendustega meetodid on üldjuhul lineaarse keerukusega, kuid võivad olla „halva“ kiirushinnanguga (näiteks siis, kui järjestame bitikaupa hõreda väärtusvaruga ja suure varieeruvusamplituudiga *int*-arvusid ning vektor on suhteliselt lühike).

Me ei pea ei võimalikuks ega ka vajalikuks sorteerimisalgoritmide „massilist“ tutvustamist: võimalik pole see ülesanne erinevate meetodite rohkuse tõttu ja vajalik tuntumate algoritmide kättesaadavuse tõttu[2]. Ja nii loodame, et *A&A*-õppur leiab oma tarbeks piisavalt materjale kirjandusest ja internetist, ning *C*-õppur saab noist materjalidest piisavat infot algoritmi programmeerimiseks. Sestap piirdume siin konkurentsitult *lihtsaima, „mullimeetodi*“ ning suhteliselt rafineeritud *Shell*i meetodi tutvustamisega, lisaks vaatleme paari universaalset „kiiret“ meetodit, ning lõpetame mõnede viimastest kiiremate, ent kitsendustega algoritmide käsitlemisega.

---

[1] Teine võimalik klassifitseerimistunnus võiks olla *dünaamilisus*: ära seletatult, kas me peame järjestama (teisisõnu, *sorteerima*) mälus oleva staatilise vektori või peame garanteerima, et lahendamise käigus võib lisada (algselt tühja vektorisse) elemente ja neid töö käigus kustutada, ent meie programm peab suvalisel hetkel esitatud päringu rahuldamiseks edastama nõutud moel (mittekasvav või mittekahanev järjestus) sorteeritud jada. Vektoritega manipuleerivad algoritmid dünaamikaga üldjuhul toime ei tule (erandiks on näiteks *pistemeetod*); me käsitleme selle probleemi võimalikku lahendust *ahelate* ja *puude* temaatikaga tutvudes.

[2] Ainult mõned näited: *J. Kiho* raamatus [Kiho 03] tutvustatakse universaalsetest piste-, kiir- ja ühildusmeetodit, *V. Leppikson* [Leppikson A&A] lisaks neile valik-, mulli-, *Shell*i, kuhja- jm. meetodiga järjestamist, juhuslikult leitud internetiallikas [Sorting Algorithms] kirjeldab 16 meetodit ning loodab oma nimekirja lisada veel 10, rääkimata *D. Knuth*ist, kes pühendab järjestamisele oma raamatu III köites üle 400 leheküle [Knuth III] – tõsi, *ca* 150 lk. *välissorteerimisele* (välisseadmetel paiknevate andmekogumite järjestamisele), mis oli tolleaegsete masinate piiratud operatiivmälu mahtu arvestades märksa olulisem teema kui see tänapäeva rakendusprogrammeerijate jaoks on.

5.4.1. Mullimeetod

Mullimeetod on lihtsaim ning hõlpsaimini programmeeritav järjestamismeetod: *bubble*[1] *sort* töötab ajalise keerukusega $O(n^2)$ ja kiirushinnanguga $n(n\text{-}1)/2$. Omapärast nimetust võiksime seletada näitega: olgu meil kõrge ja kitsas pokaal, kuhu valame gaseeritud mehu (viljalihaga mahla). Mingi aja jooksul[2] näeme, kuidas gaasimullid kerkivad pinnale (need kujutavad endast meie jada väiksemate väärtustega elemente), viljaliha sadestub põhja (olles suuremate väärtuste rollis) ning sademe kohale jääb viljalihast ilma jäänud mahl – keskmiste väärtustega elemendid. *Owen Astrachan* [Astrachan] kirjutas 2003. aastal mullimeetodist ülevaateartikli, millele järgnevas toetume.

Sissejuhatuses tsiteerib autor *Knuth*i 3. köidet: „Lühidalt, mullimeetod ei näi evivat midagi, miks teda soovitada võiks – välja arvatud tabav nimi ja tõik, et ta juhib mõningate huvitavate teoreetiliste probleemide juurde“[3] ja jätkab huvitava mõttekäiguga – mida mäletab tänane üliõpilane 10 aasta pärast järjestamist käsitlevatest loengutest-praktikumidest: tõenäoliselt oma õppejõudusid ja lisaks mullimeetodit. Ning nendib, et see viimane seik ei pruugi olla juhus. Tuginedes kasutatud 25 allikmaterjalile toob ta välja mullimeetodi raskestivaidlustatavad plussid:

- Suhteliselt lühikeste jadade puhul ajalise keerukuse ruuthinnang ei ole hoomatav, enamgi veel[4], tänu väga lihtsatele iga sorteerimissammu tegevustele (üks võrdlustehe ja – vajadusel – kahe elemendi vahetamine) töötab ta mingi piirini kiiremini kui parema keerukushinnanguga, ent keerulisemad meetodid.
- Mullimeetod jääb kergesti meelde ja vajadusel saab „käigult“ vektori järjestamise koodi kirjutada (mõistagi siis, kui jada pikkus meid mõtlema ei pane).
- Enamik professionaale kasutab vektori sobiva pikkuse *n* korral käigult programmeeritavat mullimeetodit (iseenesest pole see tõsiseltvõetav argument[5], ent midagi siiski näitab).
- Mullimeetod töötab lineaarselähedase kiirushinnanguga „peaaegu järjestatud“ vektoril.

Owen Astrachan uuris *Google*'i abil nii 2000. kui ka 2002.aastal enimkasutatavate järjestusmeetodite populaarsust programmeerijate hulgas, laename sealt vastava tabeli:

sort	# hits 2000	# hits 2002
Quick	26,780	80,200
Merge	13,330	33,500
Heap	9,830	22,960
Bubble	12,400	33,800
Insertion	8,450	21,870
Selection	6,720	20,600
Shell	4,540	8,620

Table 1: Web-based popularity of sorts

kiirmeetod
ühildusmeetod
kuhjameetod
mullimeetod
pistemeetod
valikmeetod
*Shell*i meetod

Arvud ridades-veergudes näitavad päringute arvu vastavate meetodite käsitlustele. Ja mullimeetod on neil andmetel populaarseim mittelineaarse keerukushinnanguga järjestusmeetod.

[1] Inglisekeelne *bubble* on *Melanie Rauk*'i sõnastiku järgi [Rauk, lk. 48] „1.*n* mull, vull; 2. *v.* mulle ajama, kihisema, vulisema“. Vene keeles on ta пузырек ja *mullimeetod* – метод пузырька.

[2] Katset peaks olema üsna lihtne teha, kuivõrd gaseeritud mehu ei meelita arvatavasti kedagi seda ära jooma.

[3] Venekeelses tõlkes (mida meie kasutame) vt [Knuth III, lk. 137]

[4] See siin on meie märkus.

[5] Võrdluseks: maailma parim programmeerimiskeel on *COBOL* (suurarvutite-ajastul oli lõviosa *USA*s kirjutatud programmidest selles keeles) või et maailma lihtsaim keel on hiina keel (iga neljas inimene saab sellest aru).

Mullimeetodi autorit me paraku ei tea, ent *Owen Astrachan*i otsingud viivad – küll autorit tuvastamata – tagasi 1955. aastasse. Aastaarvu vaadates tundub tõenäolisena, et esmarealisatsioon oli kas mingi tolleaegse arvuti masinkoodis või assembleris ning hilisemad restauratsioonid pole ilmselt ainuvõimalikud (sh. kaks *O. Astrachan*i toodut), ent see polegi oluline. Meie *C*-variant moodulist sai selline, nagu näeme prorgrammi *shell.c* tekstis allpool.

Iga välimise tsükli (*i* järgi) sammu lõppedes on garanteeritult esimesed *i+1* elementi sorteerituse mõttes oma õigetel kohtadel.

5.4.2. Shell'i meetod

*Shell*i meetod (*Shell sort*) sai tuntuks 1959. aastal, kui *CACM*is ilmus *Donald L. Shell*i artikkel *„A high-speed sorting procedure*" (vt. [Shell]). Mõneti on see meetod võrreldav *mullimeetodiga*, ent *Shell*i algoritm kasutab sellist taktikat, et esimesel ringil võrreldakse üksteisest ca *n*/2 (*n* on järjestatavate elementide arv) kaugusel olevaid elemente, teisel ringil on kauguseks *n*/4 jne, kuni jõutakse naabrite võrdlemiseni (ja vajadusel vahetamiseni).

C-algoritmi laename autoriteetidelt, *Kernighani* ja *Ritchie* raamatust [K&R, lk.62].

```
//shell.c: shellsort: sort v[0]..v[n-1] into increasing order
//lk. 62 ja mullimeetod

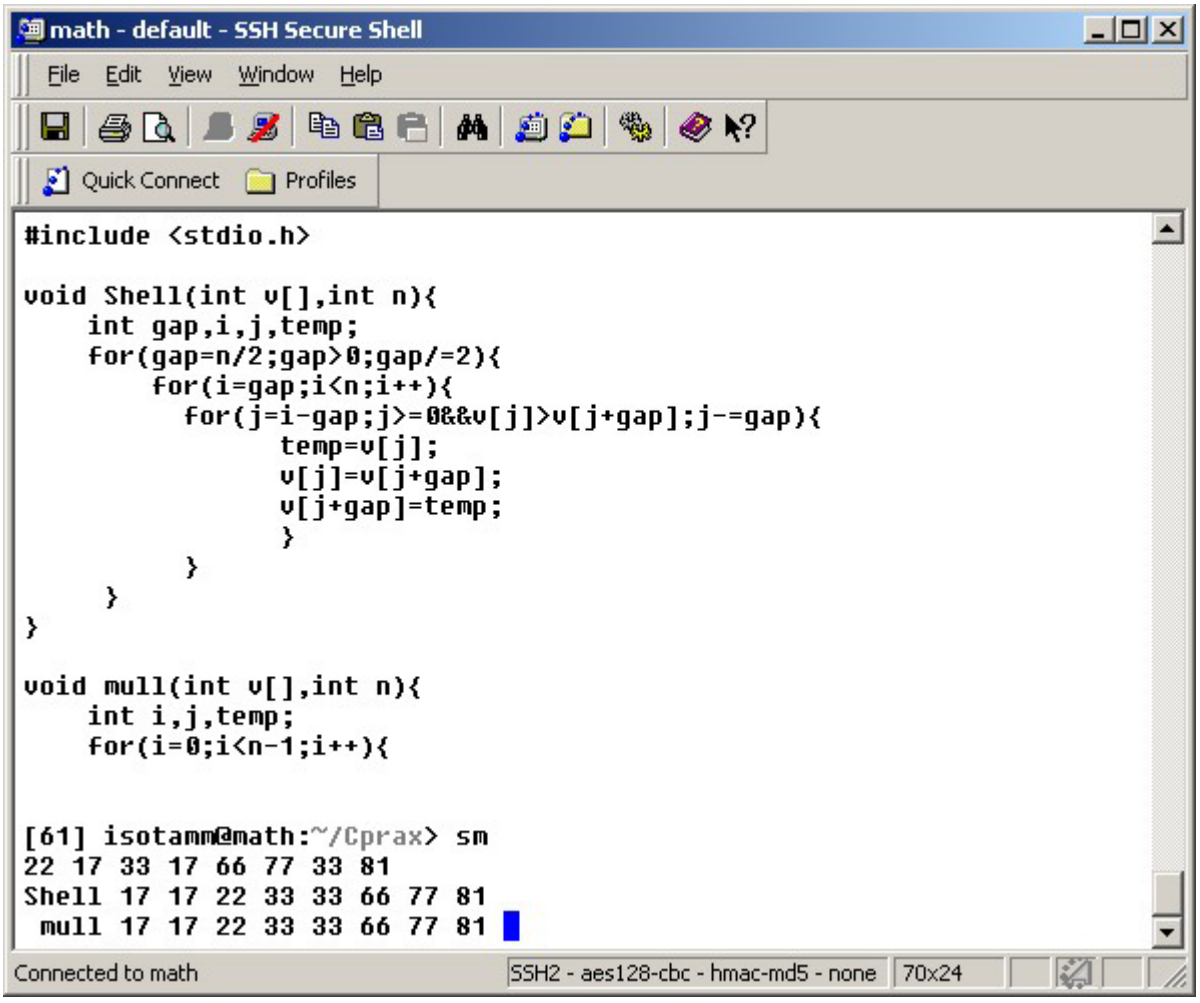
#include <stdio.h>

//see on meie tekst
void mull(int v[ ],int n){
    int i,j,temp;
    for(i=0;i<n-1;i++){
     for(j=i+1;j<n;j++){
         if(v[i]>v[j]){
          temp=v[i];
          v[i]=v[j];
          v[j]=temp;
          }
         }
     }
}

//siit algav tekst on K&R-raamatu oma
void shellsort(int v[ ],int n){
     int gap,i,j,temp;

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

```
//see pole enam K&R-raamatu tekst.
int main( ){
     int i;
     int a[8]={22,17,33,17,66,77,33,81};  //Shell'i jaoks
     int b[8]={22,17,33,17,66,77,33,81};  //mullimeetodi jaoks:
                                          //Shell rikub nad ära
     for(i=0; i<8;i++) printf("%d ",a[i]);
     printf("\nShell ");
     Shell(a,8);
     for(i=0;i<8;i++) printf("%d ",a[i]);
     printf("\n mull ");
     mull(b,8);
     for(i=0;i<8;i++) printf("%d ",b[i]);
     getchar( );
}
```



Joonis 5.4.2.a. *Shell*i meetod ja mullimeetod.

*Shell*i meetodi ajaliseks keerukuseks on pakutud halvimal juhul $O(n^2)$ ja keskmiselt $O(n^{3/2})$ (vt. [Knuth III], lk. 113), viimast hinnangut võime kirjutada ka kui $O(\sqrt{n}^3)$.

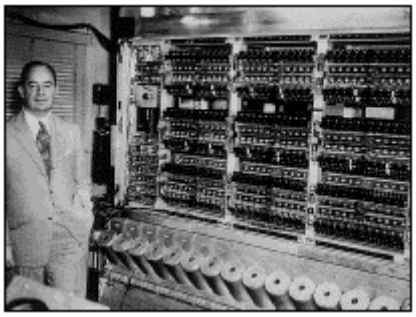
Järjestamise üldjuhul – milleks on *kirjete* ja mitte pelgalt *võtmete*[1] järjestamine, so. üldjuht on selline, kus järjestatavate väärtustega on seotud mingi semantiliselt kaalukas info – üheks oluliseks omaduseks on *järjestusmeetodi stabiilsus*: stabiilsed meetodid on sellised, kus võrdsete väärtustega jadaelementide omavahelist algset järjestust ei muudeta. Vektori puhul pole see seik oluline, ja kui olekski, poleks *stabiilsus* tuvastatav: kui vektoris on koolipoiste kasvud, siis näiteks 18 registreeritud väärtusest „176“ pole ei enne ega pärast järjestamist tuvastatav, mitmendana see pikkus fikseeriti, ja kes nii pikk oli.

Meie aeglastest näitemeetoditest on *mullimeetod stabiilne* ja *Shell*i meetod *ei ole stabiilne*.

5.4.3. Kiired universaalsed meetodid

Kiiretest universaalsetest järjestusmeetoditest on üsna levinud arvamuse kohaselt (vt. näit. [Kiho 03], lk. 62 – 71) populaarseimad *John v. Neumann*i *ühildusmeetod* (*merge sort*, 1945) ja *C. A. R. Hoare*'i *kiirmeetod* (*quick sort*, 1962). Mõlema meetodi ajaline keerukus on tänaste teadmiste järgi parim võimalikest (kitsendusteta meetoditele kehtivatest) hinnangutest, so. $O(n \times \log(n))$. Tuletagem meelde, et ajaline keerukus on „jämedalt võttes“ funktsioon, mis näitab, kuidas suureneb lahendamisaeg (meie peatüki kontekstis) vektori pikkuse n kasvades.

Nii *kiirmeetod* kui ka *ühildusmeetod* kasutavad poliitikateaduse klassikast inspireeritud „jaga-ja-valitse“-printsiipi (ladina keeles *devide et impera*[2]): „esialgne ülesanne jagatakse kaheks väiksema mahuga ülesandeks, mis siis omakorda lahendatakse samal meetodil[3] rekursiivselt; alamülesannete lahendustulemuste põhjal leitakse esialgse ülesande lahend.“ [Kiho 03, lk. 62]



John v. Neumann (algupäraselt *Johann* ja pereringis *Jancsi*), 28.12.1903 Budapest – 8.02.1957 Washington, DC. [v. Neumann]). Ta on üks olulisimatest tegijatest arvutiteaduse valdkonnas (vt. näit. [Isotamm, PK] lk. 20), peame silmas *von Neumann*i masinat. Peale selle on *J. v. N*-l teeneid mänguteooria, vesinikupommi-projekti, ent ka esimese $O(n \times \log(n))$ -keerukusega järjestusmeetodi (ja seda 1945. aastal) temaatikas. Pildil on *J. v. Neumann* ja tolleaegne arvuti.

*Von Neumann*i meetod on leidnud kirjeldamist vististi kõigis järjestusmeetodeid käsitlevates ülevaatematerjalides, sh. [Knuth III, lk. 193 jj.] ja õppevahendites, sh. [Kiho 03, lk. 66 jj.]. Tsiteerimegi viimast allikat: „...igal sammul jaotatakse sorteeritav (osa)järjend lihtsalt kaheks võrdse (või siis ühe võrra erineva) pikkusega pooleks. Mõlemale poolele rakendatakse rekursiivselt sedasama protseduuri. Töömahukamaks osutub aga just „valitsemine“: sorteeritud poolte ühendamine ehk *ühildamine, põimimine* üheks sorteeritud järjendiks. Kuna poolte määramine toimub täiesti sõltumatult n-elemendiliste lähtejärjendi elementide väärtustest, siis on

[1] Meie kontekstis vektori elementide järjestamine.

[2] See printsiip on meieni jõudnud *Niccolo Machiavelli traktaadi* "*Dell'arte della guerra*" vahendusel: „Hea juht peab kasutama ära iga võimalust, et oma vastaliste jõude hajutada: kas siis tehes neid umbuslikuks oma liitlaste vastu või andes neile hea põhjuse jagada oma väed väiksemateks gruppideks ning läbi selle muutes neid nõrgemaks.“ [pronto]

[3] *Alfred Aho* koos kaasautoritega osutab, et *jaga-ja-valitse*-printsiibi järgimine ei eelda tingimata rekursiooni; ka *mullimeetod* **jagab** igal välimise tsükli sammul vektori kaheks osaks – järjestatuks ja veel järjestamatuks – ning kasutab „teises pooles“ sama algoritmi (vt. [A,H&U, lk. 81]). See printsiip on efektiivne, kui algoritm järgib *jagamisel* balansseerimisprintsiipi (osad peavad olema võimalikult võrdsete pikkustega). *Mullimeetod* ilmselgelt nii ei käitu.

selge, et algoritmi rekursiivse[1] põhiosa rakenduste arv on igal juhul $O(\log n)$ ning mingit eriti halba algandmete juhtu ei saagi olla. Ühe n-elemendilise sorteeritud järjendi kokkuseadmiseks kahe lühema sorteeritud järjendi elementidest...kulub aega $O(n)$. Järelikult ühildusmeetodi ajaline keerukus on $O(n \log n)$. ...ühildusmeetod realiseeritakse alati stabiilsena."

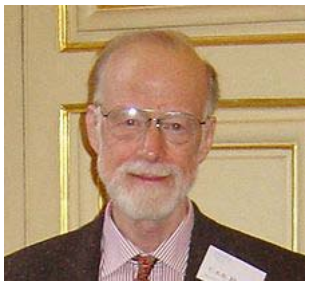
Ühildamise (*merge*, слияние) näite laename *Knuth*ilt [Knuth III, lk. 193]: olgu meil kaks järjestatud vektorit, esimene (a[ ] väärtustega 503, 703, 765 ja teine b[ ] – 087, 512 ja 677. Lihtsaim ühildamisvariant on võrrelda omavahel kaht vasakpoolseimat elementi ning neist väiksema viimine (sorteeritud) väljundritta c[ ] (vaba tõlge ja tõlgendus *Knuth*i raamatust).

Algseis: a={503, 703, 765}; b={087, 512, 677} ja c={ }.

Pärast esimest sammu: a={503, 703, 765}; b={512, 677} ja c={087}.
Pärast teist sammu: a={703, 765}; b={512, 677} ja c={087, 503}.

ja nii edasi, nii lõpetas näite ka *Knuth. Kiho*: „Ühildusmeetodi praktilise kasutamise teeb ebamugavaks asjaolu, et massiivina esitatud järjendi sorteerimisel tuleb (ühildamise alamprotseduuris) kasutada lisamälu"[2].

Kiirmeetodi pakkus 17 aastat *J. v. Neumanni* kiirest meetodist hiljem välja britt *C. A. R. Hoare* [Hoare]; tema meetod on üldjuhul ühildusmeetodist kiirem, ent halvimal juhul on ruuthinnanguga – mis pole ühildusmeetodil mõeldav, viimasel taoliseid „halvemaid juhte" pole.



Sir Charles Antony Richard Hoare (*Tony Hoare*, ka *C.A.R. Hoare*, sünd. 11 jaanuaril 1934 Tseiloni saarel), valdab vene keelt (pärast *Oxford*i *Ülikooli* õppis *Moskva Riiklikus Ülikoolis* loomulike keelte tõlkimise teooriat). Meie jaoks on ta oluline eeskätt kui *kiirmeetodi autor*, ent ka kui üks esimesi abstraktse andmestruktuuri *tabel* (kui võtmetega kirjete hulk) evitajaid [PK].

Kiirmeetodit kirjeldame taas *Jüri Kiho* [Kiho 03, lk. 62 jj.] tsiteerides: „antud juhul osutub keerukamaks just *jagamise* etapp, kus antud väärtused sorteeritakse selliselt, et järjendi esimesse ossa jäävad väiksemad, teise ossa aga suuremad elemendid. Oluline on asjaolu, et ükski element esimeses osas ei ole suurem ühestki teise osa elemendist. Tänu sellele saabki järjendi kumbagi osa nüüd omaette sorteerida. Pärast osade sorteerimist osutub sorteerituks ka kogu järjend. Seega „valitsemise" etapp siin tegelikult puudub: alamülesannete lahendamistulemustest esialgse ülesande lahenduse saamiseks pole tarvis midagi teha.

Jaotamise põhitsüklis funktsioonis *jaotada* võrreldakse järjendi elemente arvuga b, mida võib vaadelda kui „veelahet" kahe otsitava osa vahel. Nimelt need väärtused, mis ei ületa suurust b, tuuakse algusossa ja need, mis pole väiksemad b-st, viiakse järjendi lõpuossa...Kõige soodsam on muidugi selline veelahe, millest mõlemale poole jääb enam-vähem ühepalju väärtusi, halvim aga selline, mille korral ühte poolde satub vaid üks element (päris tühjaks ei saa kumbki osa jääda). Väga ebavõrdsete poolte puhul suureneb oluliselt oluliselt rekursiivsete rakenduste arv ja seega ka sorteerimiseks kuluv aeg. Viimasest asjaolust tulenebki õigupoolest kiirmeetodi $O(n^2)$ hinnang halvimal juhul".

[1] *Algoritmide ja andmestruktuuride*-temaatika ei saa mööda minna *rekursioonist*, seda võtet toetab ka *C.*
[2] Meenutagem, et omal ajal oli operatiivmälu kallis ja defitsiitne ressurss.

Knuth käsitleb *kiirsorti* põhjalikult (vt [Knuth III], lk. 140..151, 161, 165..167, 181 ja tagapoolgi).

Me laename nii *ühildus*- kui ka *kiir*meetodi *C*-tekstid kursuse „Programmeerimine C-keeles (MTAT.03.219)“ arvestuse saamiseks 2007. aasta sügissemestril *Lauri Rätsep*a (pildil) kirjutatud ja silutud *C*-failist[1]:

```
// Lauri :: ühildus- ja kiirmeetod, sügis 2007.
#include <stdio.h>
#include <time.h>
#define SIZE 100000

// Peameetod

  int main( ) {
      int massiiv1[SIZE];
      int massiiv2[SIZE];
      int i;
      clock_t start, end;
      double cpu_time_used = 0.0;
  /* Testimiseks loome kaks identset massiivi, mis sisaldavad suvalisi
      elemente */
     fillRand(massiiv1, massiiv2);
     printf("\nAlgne massiiv: ");
     printArray(massiiv1);

  // Käivitame kella
   start = clock( );
   quickSort(massiiv1, 0, SIZE - 1);
   end = clock( );
 // Sulgeme kella
   cpu_time_used = ((double)(1000.0*(end - start)))/CLOCKS_PER_SEC;
   printf("\nKiirmeetodil läks aega: %4.0f ms.\n", cpu_time_used);
   printf("\nSorteeritud massiiv: ");
   printArray(massiiv1);

  start = clock( );
  mergeSort(massiiv2, 0, SIZE - 1);
  end = clock( );
  cpu_time_used = ((double)(1000.0*(end - start)))/CLOCKS_PER_SEC;
  printf("\nÜhildusmeetodil läks aega: %4.0f ms.\n", cpu_time_used);
  printf("\nSorteeritud massiiv: ");
  printArray(massiiv1);
  return 0;
}

//väljastab vektori alguse... ja ...lõpu ekraanile, edasi: "any key"
printArray(int *massiiv1) {
   int i;
   for(i = 0; i < 10; i++)
     printf("%d, ", massiiv1[i]);
   printf("..., ");
   for(i = SIZE - 10; i < SIZE - 1; i++)
     printf("%d, ", massiiv1[i]);
   printf("%d. ", massiiv1[SIZE - 1]);
   getchar( );
}
```

[1] NB! Kellel on vaja lahendusaega fikseerida, on *Lauri* programmist hea eeskuju võtta.

```
// Täidab massiivid SIZE juhuslike arvudega.
fillRand(int *massiiv1, int *massiiv2) {
  int i;
  double r;
  for(i = 0; i < SIZE; i++) {
    r = rand( );
    massiiv1[i] = r;
    massiiv2[i] = r;
  }
}

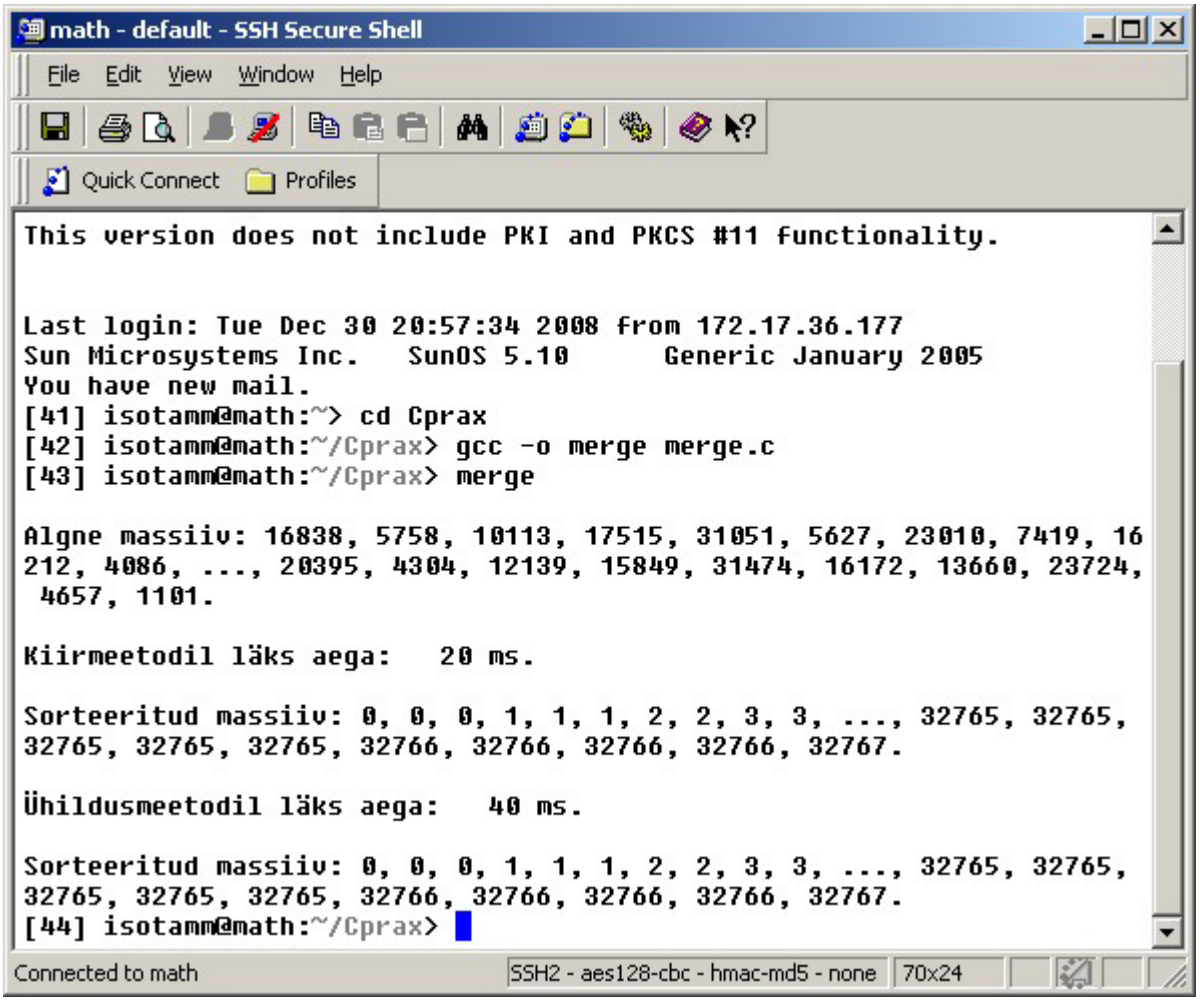
// Kiirmeetod
quickSort(int massiiv[], int l, int r) {
  // Poolituspunkt
  int c;
  // Kui vasak indeks on paremast indeksist möödunud, siis rekursioon
lõppeb
  if(l < r) {
    // jaga
    c = divide(massiiv, l, r);
    // ja valitse
    quickSort(massiiv, l, c - 1);
    quickSort(massiiv, c + 1, r);
  }
}

int divide(int massiiv[ ], int l, int r) {
  int d, i, j, temp;
  d = massiiv[l];
  i = l;
  j = r + 1;
  while (1 == 1) {
      do
        i++;
      while (massiiv[i] <= d && i <= r);
      do
        j--;
      while (massiiv[j] > d);
      if (i >= j) break;
      temp = massiiv[i];
      massiiv[i] = massiiv[j];
      massiiv[j] = temp;
      }
     temp = massiiv[l];
     massiiv[l] = massiiv[j];
     massiiv[j] = temp;
     return j;
}

// Ühildusmeetod
mergeSort(int massiiv[ ], int l, int r) {
  int c;
  if(l < r) {
      // jaga
    c = (l + r)/2;
    // ja valitse
    // sorteerimine toimub mõlemas alamosas eraldi
    mergeSort(massiiv, l, c);
    mergeSort(massiiv, c + 1, r);
    // alamosad ühendatakse
    merge(massiiv, c, l, r);
```

```
  }
}

merge(int massiiv[ ], int c, int l, int r) {
  int i, j, k, temp[SIZE];
  i=l;
  j=c+1;
  k=l;
  while ((i <= c) && (j <= r)) {
    if (massiiv[i] < massiiv[j]) {
      temp[k] = massiiv[i];
      k++;
      i++;
    }
    else {
      temp[k] = massiiv[j];
      k++;
      j++;
    }
  }
  while (i <= c) {
    temp[k] = massiiv[i];
    k++;
    i++;
  }
```



Joonis 5.4.3.a. Ühildus- ja kiirmeetod. *Lauri Rätsep*a „merge.c“ protokoll.

```
 while (j <= r) {
    temp[k] = massiiv[j];
    k++;
    j++;
  }
  for (i = l; i < k; i++) {
      massiiv[i] = temp[i];
      }
}
```

Võrdlemaks *kiir-* ja *ühildusmeetodit*, tegi *Simon Vigonski* (pildil, *A&A*-kursuse omaaegne kuulaja) seeria katseid *int*-tüüpi vektoritega: nende pikkus muutus vahemikus 10 000 kuni 1 miljon ning meetoditele anti ette neli varianti: vektor juhuslikus järjestuses, kasvavalt ja kahanevalt sorteeritud ning nullidega täidetult. Kiirmeetodit katsetati kaheti, esiteks suvaliselt, teiseks kasutati *mediaan*-lahet. Testimistulemused on lisas 4**;** mõistagi soovitame neid resultaate süvenenult uurida, ent jäme hinnang on: kiirmeetod on ühildusmeetodist alati pisut (vähem kui kaks korda) kiirem[1] ning kiirmeetodi töökiirust *mediaanlahkme* kasutamine märkimisväärselt ei suurenda [Vigonski].

5.4.4. Kitsendustega meetodid

Kitsendustega järjestusmeetoditest vaatleme kolme: kaht *positsioonilist* meetodit (*biti-* ja *baidikaupa* sorteerimine ning *loendamismeetodi* modifikatsiooni „*luuresort*“. Positsioonilistest meetoditest vt. [Kiho 03, lk.72..73] ja [Knuth III, lk. 151 jj., 462 ja 587]. *Knuth*i andmeil pärinevad esimesed teated positsioonilistest meetoditest 1959. aastast[2]; tema raamatus refereeritud algoritm on üsnagi keeruline ning selle kiirushinnang on $\approx n \times \log n$. Tänaseks teame, et „positsioonimeetodi ajaline keerukus on $O(n)$, täpsemalt $O(dn + nk)$ [Kiho 03, lk. 73], kus

- n on vektori pikkus,
- d on „numbrite arv“ võtmes (bitikaupa sorteerimisel võtme maksimaalne bittide arv, baidikaupa sorteerimisel – maksimaalne baitide arv,
- k on positsioonilise arvusüsteemi *alus* (igas „positsioonis“ paikneb nunber hulgast $K=\{0, 1, .., k\}$

„Seega sorteeritakse d-numbrilisi arve, mis on kirjutatud positsioonilises arvusüsteemis alusel $k + 1$“ [Kiho 03, lk. 72].

[1] Meenutagem, mõlema meetodi ajaline keerukus on $O(n \times log(n))$. Vahe on *kiirushinnangus.*

[2] Nende ridade kirjutaja ja *Anne Villems*i mäletamist mööda leidis 1970. aasta paiku selleteemalise ungarikeelse artikli Eesti Raadio Arvutuskeskuse programmeerija *Naima Villo*, meetodit kasutas ta *SODI*-süsteemis (ankeettüüpi andmete statistilise töötlemise süsteem, vastutavad täitjad olid TPI Arvutuskeskusest *Leo Võhandu* juhendamisel). *TPI*s stažeerinud *Anne* realiseeris selle algoritmi paar aastat hiljem, projekti *VILLIS* jaoks (aruannete generaator, masin oli *Minsk*-32 ja programmeerimiskeeleks viimase assembler). Siintoodud algoritm on tollest originaalist (mille kohta saab teavet *Annelt* või allakirjutanult, olemas on ka *C*-programm) mõnevõrra lihtsam.

5.4.4.1. Bitikaupa järjestamine

Bitikaupa järjestamise kiirushinnang on niisiis $n \times k$, kus n on järjestatavate arv ning k on järjestustunnuse (vektori elemendi või kirje võtme) pikkus bittides, seega ajaline keerukus on $O(n)$. Meie programmiversioon on järgmine:

```
//bitp.c  12.11.07  bitikaupa järjestamine paremalt vasakule ilma muutuva
//maskita[1]
#include <stdio.h>
#include <stdlib.h>

    char tc[10]={'b','x','0','m','a','8','h','v','y','*'};    //sümbolid
    char td[10]={'9','8','7','5','1','4','6','0','2','3'};   //numbrid

//parameetrid:  võtme pikkus bittides, vektori pikkus, viit vektorile,
//vahetrükk (1: trüki)

void bitsort(int bitarv,int n,char *t,int p){
     int i,j,k,b1,sb,yx=1;   //yx: aktiivse biti mask
     char x;
     for(i=0; i<bitarv; i++){
       b1=-1;              //1-bitti (veel) pole
       for(j=0; j<n; j++){
          sb=t[j]&yx;
          if((sb!=0)&&(b1<0)) b1=j;//kui 1-bitt oli fikseerimata:fikseerin
          if(sb==0){
             if(b1>-1){
                x=t[j];  //leitud 0, selle asemele viimane 1
                for(k=j; k>b1; k--) t[k]=t[k-1]; //yhtede jada nihutamine 1
                        //koht alla
                t[b1]=x;  //vana esimese 1 asemele 0 (eespool ainult 0-d)
                b1++;  //uus esimene 1
                }
              }
          if(p==1){
             printf("yx=%d i=%d j=%d b1=%d tj=%c\n",yx,i,j,b1,t[j]);
             for(k=0; k<10; k++) printf("%c ",t[k]);
             getchar( );
             }
          }
       yx<<=1;   //maski nihutamine 1 biti võrra vasakule.
       }
    for(i=0; i<n; i++) printf("%c ",t[i]);
    getchar( );
}

int main( ){
    bitsort(8,10,tc,0);    //80 tsüklisammu
    bitsort(4,10,td,0);    //40 tsüklisammu
}
```

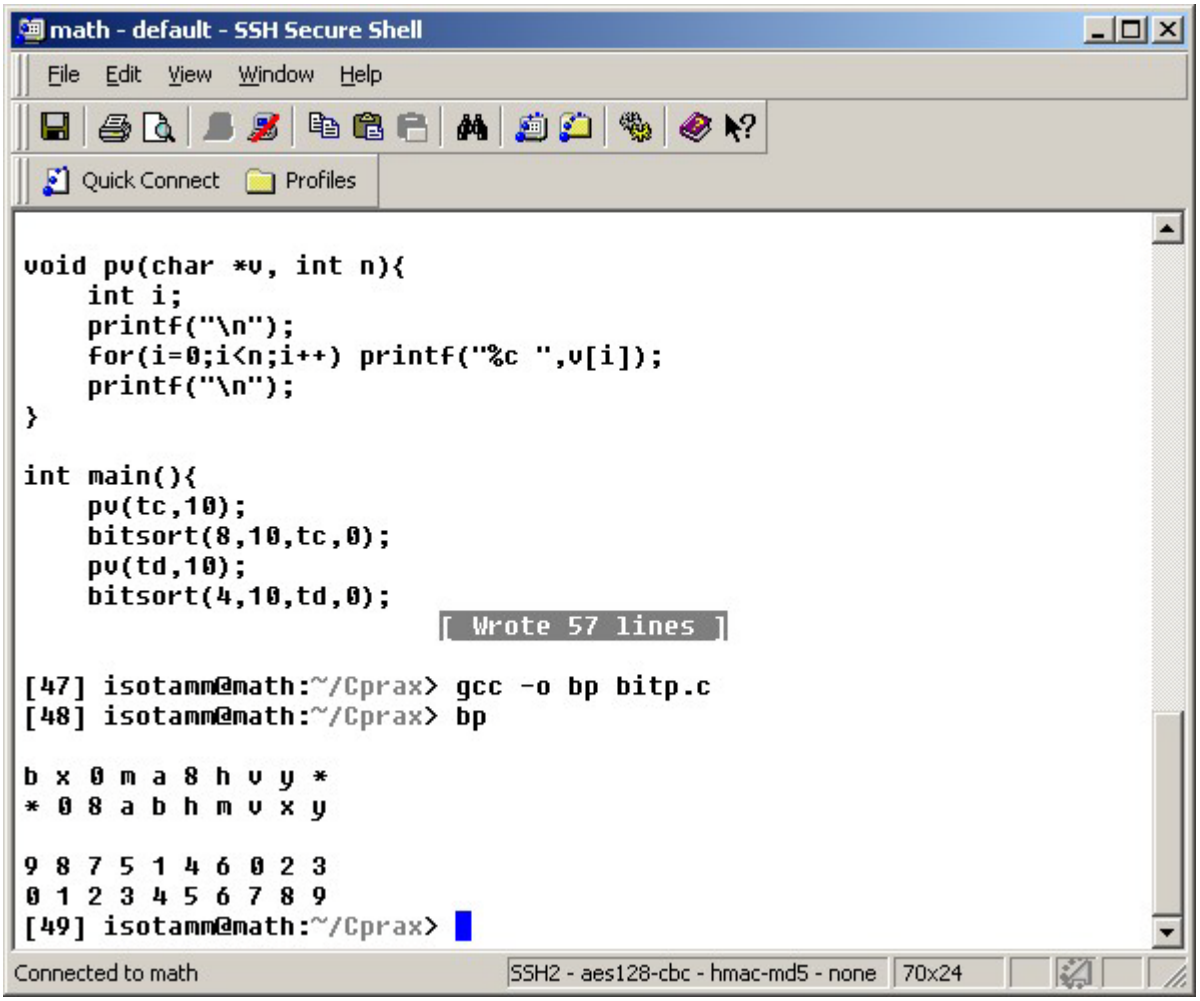
Bitikaupa järjestamise meetodil on olulised kitsendused:

- Ujupunktarvude jaoks ta ei sobi, kuivõrd nende bitikaupa-kujutamine on struktuurne;
- *Märgiga int*-arvude jaoks ei sobi ta samuti: tänastel arvutitel kujutatakse „miinus-märki“ arvuvälja vasakpoolseima biti väärtuse „1“ abil; bitikaupa-järjestamine peab aga

[1] Originaalalgoritm kasutas „pesa-mask-nihutaja“-võtet.

kõiki negatiivseid arve suurematekskõikidest mittenegatiivsetest, ja seda suuremaks, mida väiksem ta on (–2 > –1).

Niisiis, *bitikaupa järjestamine* on kasutatav, kui võtmed on kas mittenegatiivsed täisarvud või tekst, ja *pole kasutatav*, kui võtmed on märgiga täisarvud või reaalarvud. Meetodi ajalise keerukuse hinnang on lineaarne: O(*n*).



Joonis 5.4.4.1.a. Bitikaupa järjestamine.

5.4.4.2. Baidikaupa järjestamine

Baidikaupa sorteerimine toimib sama loogikaga nagu bitikaupa järjestamiselgi, ent ehkki *bait* on bitist kaheksa korda pikem, pole baidikaupa sorteerimine 8 korda kiirem, põhjuse leiavad meie lugejad [Kiho 03, lk. 72] valemist. Kaheksabitine kood on 256 korda mahukam kahendkoodist, aga see „maksab“. Bitikaupa sorteerimiseks piisas meile *ühest int*-tüüpi „järjehoidjast“ (*b*1), mille abil pidasime meeles piiri viimase 0-väärtuse ja esimese 1-väärtuse vahel. Allpool toodud näiteprogramm järjestab sümbolkujul esitatud mittenegatiivseid täisarve ning „järje hoidmiseks“ on vaja 10-elemendilist vektorit (*B*) numbrite 0..9 vaheliste analoogiliste piiride jaoks. Kui kirjutame programmi suvalistest *ASCII*-koodidest koosnevate tekstide järjestamiseks, siis peame kasutama 256-elemendilist „järjehoiu“-vektorit.

Baidikaupa järjestamise kitsendused on samasugused nagu *bitikaupa* järjestamisel: sobib nii suvalistest sümbolitest koosnev tekst kui ka mittenegatiivsed täisarvud, ja ajalise keerukuse hinnang on nagu bitikaupa järjestamiselgi lineaarne: O(*n*).

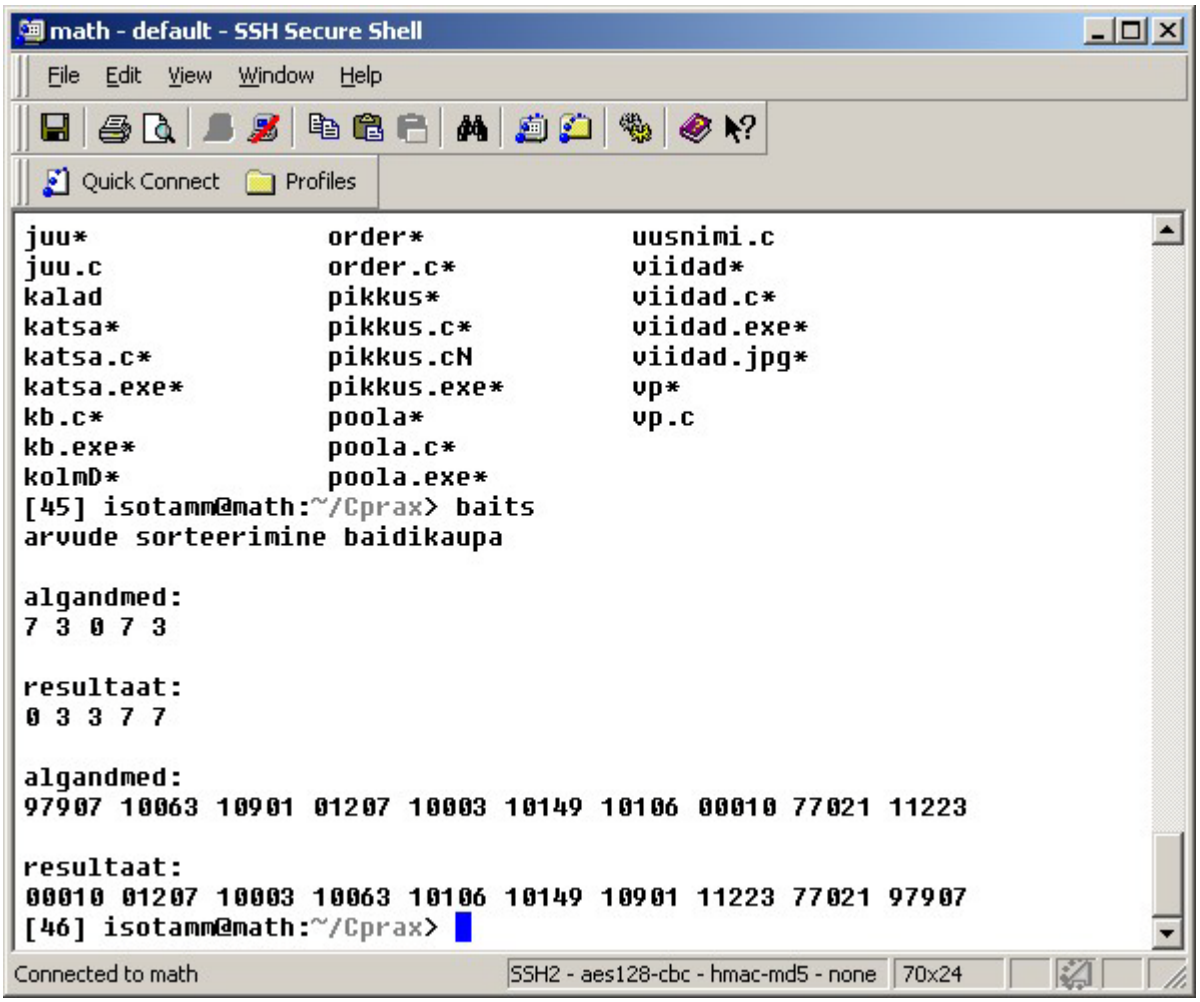
```
//baits.c  12.11.07  baidikaupa järjestamine paremalt vasakule ilma muutuva maskita.
// Arvud on tekstikujul: iga numbrikoht on 1-baidine sümbol.
#include <stdio.h>
#include <stdlib.h>

   char *tc[5]={"7","3","0","7","3"};
   char *td[10]=
   {"97907","10063","10901","01207","10003","10149","10106","00010","77021","11223"};
   int B[10];  //B[i]:: -1 või nr. i-ga algava vektori algus   //lisamälu, pikkusega 10.

void baitsort(int baitarv,int n,char **t,int p){
   int i,j,k,b1,sb,b;
   int bc,bf;  //current & first & index
   char *x;   //current t[i]

  printf("\nalgandmed:\n");
   for(k=0; k<n; k++) printf("%s ",t[k]); printf("\n"); getchar( );  //lähteandmete trykk
   b=baitarv-1;  //aktiivse järjestusbaidi indeks, alustame parempoolseimast.
   for(i=0; i<baitarv; i++){
     for(k=0; k<10; k++) B[k]=-1;  //init: pole veel ühtegi 0..9-väärtust
     for(j=0; j<n; j++){
       x=t[j];           //x on järjekorne „baitarvu“-pikkune arv sümbolkujul
       bc=x[b]-48;  //b-nda numbri kood indeksiks (ASCII 48 = '0'): current
       bf=-1;  //esimene nihutatav: next..t[j]
       for(k=bc+1; k<10; k++){
          if(B[k]>-1){
            bf=k;  //leidsin
            break;
            }
          }
       if(bf==-1){
           if(B[bc]==-1) B[bc]=j;  //ylejäänud vektor initsialiseerimata
           goto nextj;
           }
       b1=B[bf]; //sinna tuleb x vahele
       for(k=j; k>b1; k--) t[k]=t[k-1];  //nihutamine alla, alates t[j]-kohast
       t[b1]=x;
       if(B[bc]==-1) B[bc]=b1;  //selle numbriga algava jada esimene
       for(k=bf; k<10; k++){
         if(B[k]>-1) B[k]++;       //korrigeerin algusaadresse, allpool
         }
        if(p==1){       //vahetulemuste trükk (kui p = 0, siis ei tee
          printf("j=%d bc=%d bf=%d x=%s\n",j,bc,bf,x);
          for(k=0; k<10; k++) printf("%d ",B[k]); printf("\n");
```

```
            for(k=0; k<n; k++) printf("%s ",t[k]); getchar( );
            }
          nextj:;       //tühioperaator: tsükli märgendatud lõpp
          }
        b--;      //võta vaatluse alla võtme eelmine numbrikoht
        }
        printf("resultaat:\n");
        for(k=0; k<n; k++) printf("%s ",t[k]); getchar( );    //getchar( ): ekraani peetamiseks.
}
```



Joonis 5.4.4.2.a. Baidikaupa järjestamine.

```
int main( ){
   printf("arvude sorteerimine baidikaupa\n");
   baitsort(1,5,tc,0);
   baitsort(5,10,td,0);
   }
```

5.4.5. Järjestamise strateegiad

Donald Knuth ([Knuth III], lk. 93 – 94) tutvustab erinevaid järjestusstrateegiaid järgmiselt:

- *Pistmine* (*insertion sort*, сортировка вставками): elemente vaadeldakse ükshaaval ning iga uus element pannakse oma kohale juba sorteeritud jadas. Just nii, nagu kaardimängija järjestab oma kätte talle jaotatud kaarte laualt. Kui sel meetodil järjestada staatilist vektorit, siis on ajaline keerukus ruuthinnanguga ($O(n^2)$), ent dünaamiliseks järjestamiseks[1] (eriti siis, kui me ei kasuta andmestruktuurina *vektorit*, vaid *ahelat*) on *pistemeetod* efektiivne. Algoritmi vt.[Kiho 03, lk. 60 – 61].
- *Vahetamine* (*exhange sort*, обменная сортировка): kui kaks elementi pole „õiges järjekorras“, siis nad vahetatakse omavahel. See protsess toimub kuni järjestatuse saavutamiseni. Niisiis, igal juhul *võrreldakse* ja vajadusel *vahetatakse*. Valdav enamus tuntud/tunnustatud järjestusmeetodeid kasutab just seda strateegiat, sh. ka kõik meie raamatus seni käsitletud meetodid (vahet pole, kas me võrdleme „tervikväärtusi“ nagu mulli- või ühildusmeetodis või väärtuste alamhulki nagu biti- ja baidikaupa sorteerimisel).
- *Valimine* (*selection sort*, сортировка посредством выбора): mittekahaneval järjestamisel leitakse väikseim element ja saadetakse ta väljundisse, seejärel „uus väikseim“ jne, kuni lähtevektor on tühi. Kiirushinnang on $O(n^2)$.
- *Loendamine* (*counting sort*, сортировка подсчетом): *Knuth*i ja *Kiho* järgi „järjendi iga elemendi a_i jaoks leitakse temast väiksemate elementide arv l_i. Juhul, kui järjendis ei ole korduvaid väärtusi, näitabki suurus $l_i + 1$ kohta, kus tulemusjärjendis peab paiknema a_i esialgne väärtus. Mõnevõrra komplitseerib loendamismeetodi algoritmi vajadus arvestada ka korduvate väärtustega.“ [Kiho 03, lk. 72 – 73]. *Knuth*i 1973. aastal ilmunud raamatu koostamise ajal oli selle meetodi kiirushinnanguks *ruuthinnang* [Knuth III, lk. 99], kuivõrd algoritm tegeles tõepoolest erinevate väärtuste *loendamisega*, ükshaaval. Me käsitleme pisut paremat lahendust üsna pea.
- *Erisorteerimine* (специальная сортировка), millega tutvumiseks võiks lugeda [Knuth III, lk. 93] märkusi, ent milleks pole vist vajadust, kuivõrd *D. Knuth* märgib, et see meetod toimib, kuni elementide arv ei ületa viit, ja et seda pole õnnestunud üldistada.
- *Laisk lahendus* (ленивое решение): *Knuth* kirjutab, et „Te ei reageerinud meie ettepanekule ja ei hakanudki ülesandeid lahendama. Kahju! aga nüüd, peale nii pikka lugemist on teie šanss kadunud.“
- *Uus supermeetod*, mis „teeb ära“ kõigile senitutvustatud meetoditele; *Knuth* palub: „palun, teavitega sellest kohe mind!“.

Ilmselgelt me *ei pea* kolme viimast võimalust käsitlema. Kõik me teame, mida mõeldakse, kui räägitakse „jalgratta leiutamisest“: mingi asi või meetod on *valmis* ja intellektuaalsed pingutused selles valdkonnas peaksid olema mõttetud. Ent: *loendamismeetodit* on võimalik realiseerida ka teisiti. Näiteks nii, et teame järjestatava vektori elementide miinimum- ja maksimumväärtusi (ja kui ei tea, siis *luurame*) ning et nendevaheline intervall pole „liiga“ suur ja siis saame vektori elemente järjestada kasutamata ei *võrdlemist* ega ka *vahetamist.*

[1] Alustame tühjast järjendist, elemendid laekuvad töö käigus ning iga uus element tuleb „pista“ oma kohale.

5.4.6. Luure

Luure on üsnagi universaalne *eeltöötluse* meetod. See termin (*spying*? разведка?) pole andmetöötluses kuigivõrd tuntud; aruannete generaatori *VILLIS*[1] väljatöötamise käigus mõtlesime selle välja. Põhjus peitus seigas, et *VILLIS* genereeris aruandeid *andmete poolt juhitavana* ning too „juhitavus“ baseerus võtmete väärtusvarudele so, võtme fikseeritud võimalike väärtuste järjestatud hulgale või nende „luuratavatele“ hulkadele – viimased tuvastati lähteandmete töötluse-eelse lineaarse läbivaatusega.

Meid huvitas ainult iga võtmeväärtuse esinemine (ja mitte selle esinemise sagedus), seetõttu saime *luuramiseks* kasutada bitivektorit. Võtme võimalik minimaalne ja maksimaalne väärtus olid üldjuhul teada ülesande püstitusest ning *Minsk*-32 assembler andis hea võimaluse tuvastada iga võtmeväärtuse *indeksi* – tänu käsule „CE: сложить единицы“, mille resultaat oli pesa 1-bittide arv. Üldjuhul oli meie bitivektor mitme-pesa-pikkune. Kõrvalefektina saime teada võtmeväärtuste järjestuse ja võimaluse seda kasutada.

Toome mõned juhunäited ülesannetest, kus töötluseelsest *luurest* võiks olla kasu.

Küllaltki pika[2] vektori järjestamise eel võib teha luuretsükli uurimaks, mil määral on vektor juba järjestatud; moodul võiks olla selline:

```
int luure(int n, int *a){
      int i, rike=0;
      for(i=0; i<n; i++){
            if(a[i] > a[i+1]) rike++;
            }
      }
      return(rike);
}
```

Moodul loendab vektori elementide paaride arvu, mis „rikuvad“ elementide mittekahanevat järjestatust. Kui selgub, et *rike* = 0, siis me ei peagi enam järjestusprogrammi käivitama, kui *rike* on vektori pikkusega (*n*) võrreldes väike, võime kasutada *pistemeetodit*, kui aga *rike* ≈ *n*, siis on vektor peaaegu kahanevate väärtustega elementide jada ning mõeldav oleks ta ühekordse tsükliga „ümber pöörata“ ja seejärel kasutada taas näit. *pistemeetodit.*

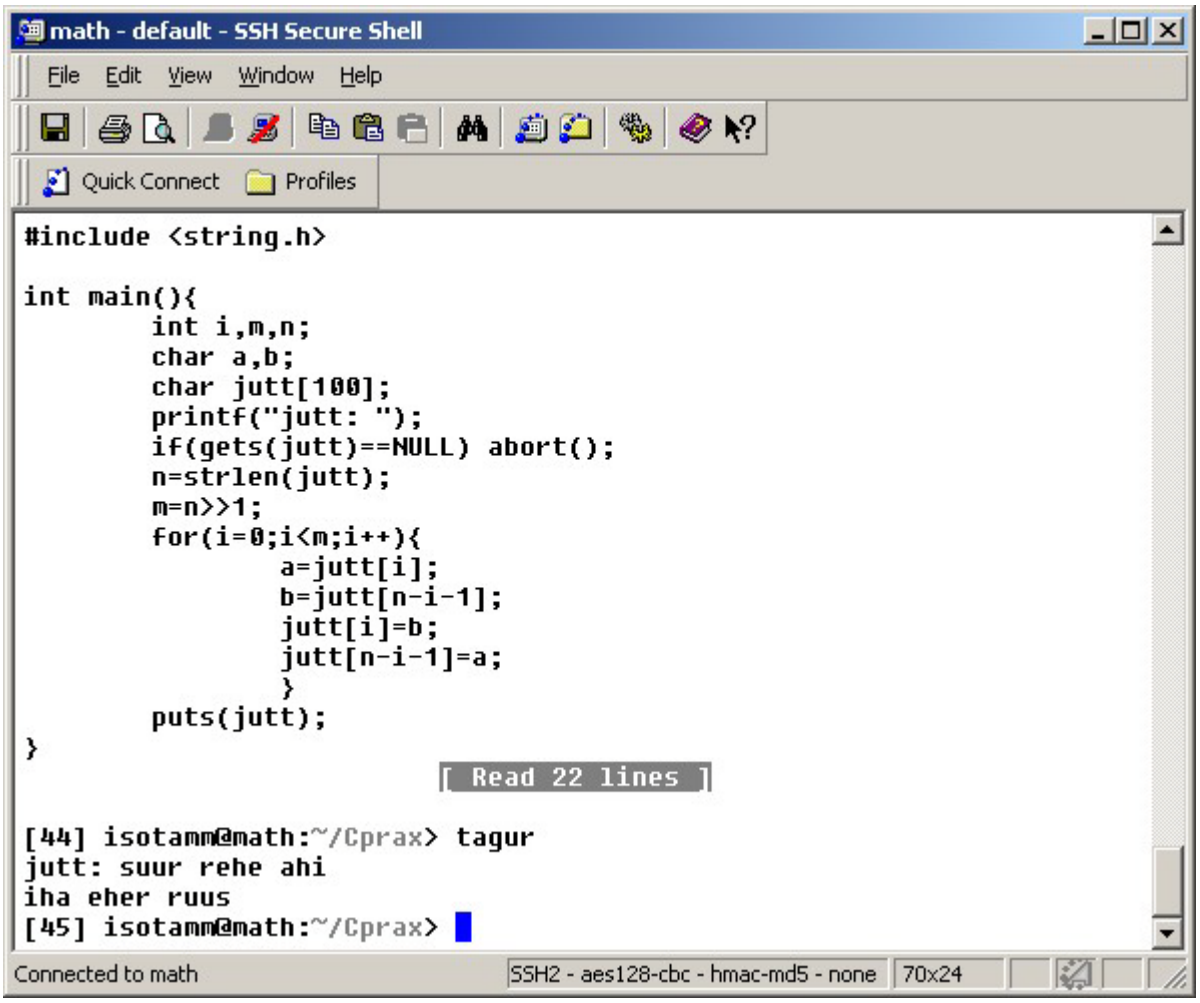
Lihtne ümberpööramise-moodul võib olla järgmine:

```
//tagur.c :: sisestab klaviatuurilt stringi ja väljastab selle anagrammi[3]
#include <stdio.h>
#include <string.h>

int main( ){
      int i,m,n;
      char a,b;
      char jutt[100];
      printf("jutt: ");
      if(gets(jutt)==NULL) abort( );
      n=strlen(jutt);
```

[1] Vt. [VILLIS].
[2] Mõtleme sellist vektori pikkust, kus ruuthinnanguga järjestusmeetodid võivad oluliselt vähendada järjestamist kasutava programmi efektiivsust.
[3] Ilmselt saame seda algoritmi kasutada ka siis, kui vektor on järjestatud elementide väärtuste mittekahanevas või mittekasvavas järjekorras, muutmaks järjestatust vastupidiseks, keerukushinnanguga $O(n)$.

```
	m=n>>1;		//m = täisosa(n/2)
	for(i=0;i<m;i++){
		a=jutt[i];
		b=jutt[n-i-1];
		jutt[i]=b;
		jutt[n-i-1]=a;
		}
	puts(jutt);
}
```



Joonis 5.4.6.a. Anagramm.

Teine luuramisnäide on seotud „pika“ sulgavaldisega, kus enne töötlusalgorimi käivitamist on mõttekas kontrollida, kas *sulgude paarsus* klapib. Kui me peaksime kirjutama *Lisp*i interpretaatorit, oleks selline eeltöötlus suisa vältimatu.

Järgmises alapunktis näitame, kuidas luuramist saab kasutada *int*-tüüpi arvude vektori järjestamiseks.

5.4.7. Luuresort

Luuresort on järjestusmeetod, kus üldjuhul tuleb enne vektori elementide järjestamist teha *luuretsükkel* teada saamaks jada minimaalse ja maksimaalse elemendi väärtused; kui me neid töödeldavate andmete iseloomu tõttu aprioori teame, jääb luuresamm mõistagi ära. Näiteprogramm on selline:

```
//luuresort.c 18.11.07 kiirushinnang::2n+m. n: luure, n: sagedusvektori tegemine, m: taga-
//si, ajaline keerukus on O(n).
#include <stdio.h>
#include <stdlib.h>

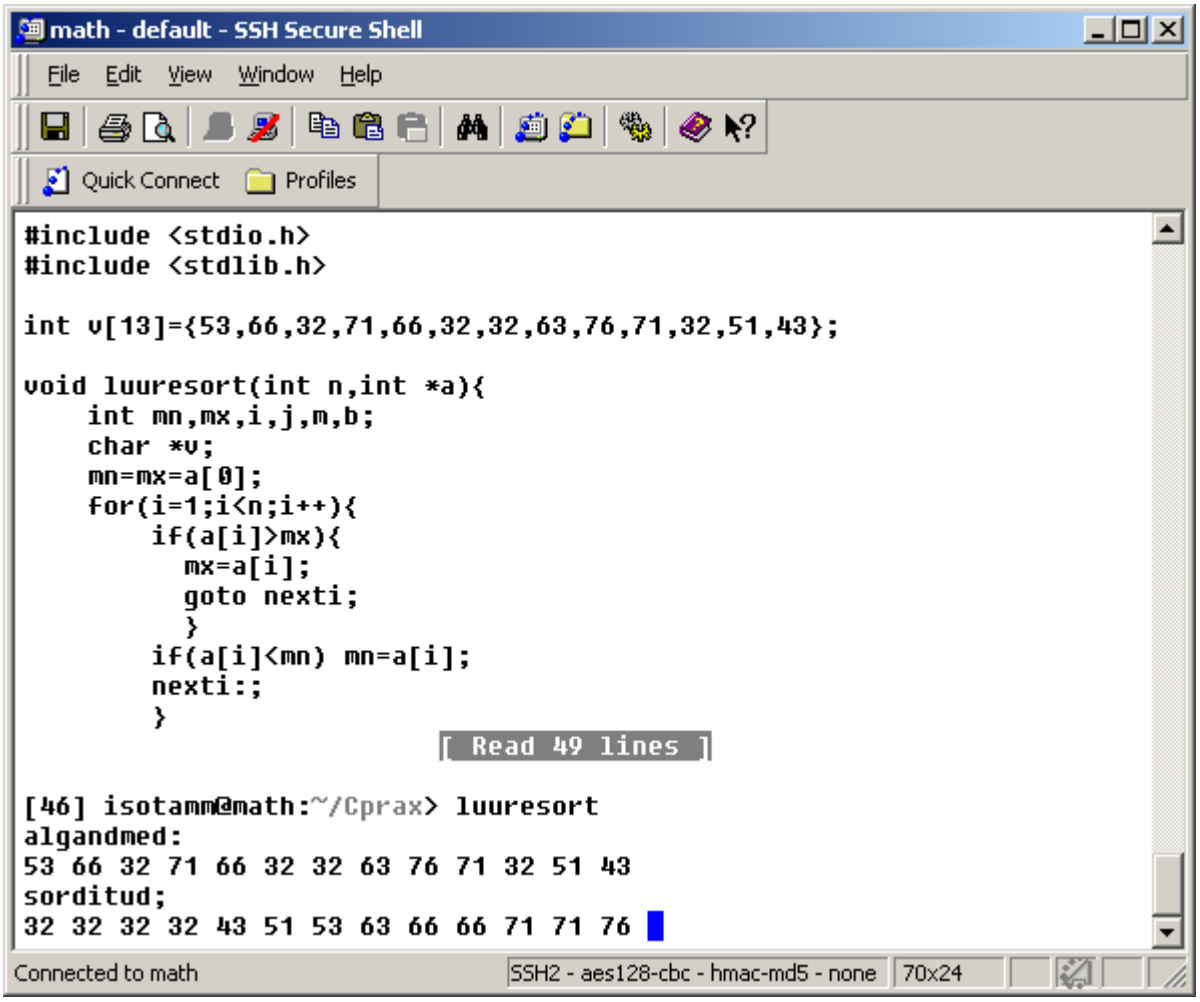
int v[13]={53,66,32,71,66,32,32,63,76,71,32,51,43};

void luuresort(int n,int *a){
   int mn,mx,i,j,m,b;
   char *v;   //kui on karta, et mõne väärtuse sagedus > 255, tuleb teha nt. short int-vektor
   mn=mx=a[0];
   //luuran maksimumi ja miinimumi
   for(i=1;i<n;i++){
      if(a[i]>mx){
        mx=a[i];
        goto nexti;
        }
      if(a[i]<mn) mn=a[i];
      nexti:;
      }
   m=mx-mn+1;  //sageduste vektori pikkus
   v=malloc(m);
   memset(v,'\0',m);  //sageduste nullimine, järgnev tsükkel leiab sagedused
   for(i=0;i<n;i++){
     j=a[i]-mn;
     v[j]++;
     }
    j=0;    //kirjutan lähtevektori üle järjestatud väärtustega
    for(i=0;i<m;i++){
      b=v[i];
      for(;b>0;b--){
        a[j]=i+mn;
        j++;
        }
      }
    free(v);   //vabastan sageduste vektori mälu  (selle võib tagastada, kui on lisakasutusi[1])
```

[1] so, *free*(*m*) asemel kirjutades *return*(*m*); moodul tuleb sel juhul kirjeldada kui *int *luuresort*(...Ühte sellist „lisakasutust“ käsitleme tagapool, *tabelite* jaotises – nii saame realiseerida ***otseadresseerimise***.

```
}

int main( ){
   int i;
   printf("algandmed:\n");
   for(i=0;i<13;i++) printf("%d ",v[i]);
   luuresort(13,v);
   printf("\nsorditud;\n");
   for(i=0;i<13;i++) printf("%d ",v[i]);
   getchar( );
}
```



Joonis 5.4.7.a. Luuresort.

See meetod sobib *int*-tüüpi arvude (sh. väärtused võivad olla ka negatiivsed) järjestamiseks; ujupunkt-arvude jaoks ilmselt mitte, samuti mitte ka vähegi pikemate stringide jaoks. Üks kitsendus, mis veel suhteliselt hiljuti oli *oluline*, on seotud vektori elementide väärtuste varieerumise ulatusega (*max – min*): see määrab sageduste vektori pikkuse ning see vektor *peab* (kui tahame mõistliku ajaga järjestada) operatiivmällu mahtuma. *Knuth*i III köite kirjutamise ajal oli operatiivmälu maht 1 miljon masinasõna suhteliselt haruldane luksus (mida evis näiteks N. Liidu tolleaegne superarvuti БЭСМ-6) ning pikkade sõreda väärtusvaruga vektorite

järjes-tamine loendusmeetodil oligi mõeldav *tegelikku loendamist* tehes (nagu mõni lehekülg varem refereerisime). Veel paar-kolm aastat tagasi oli heal kohverarvutil[1] 1 gigabait mälu, tänapäe-val on taskukohasel masinal 4 giga ning mõõdukat optimismi säilitades naerame varsti sellegi mahu üle samamoodi, nagu me nüüd esimeste *XT*-masinate ühe*mega*baidistele mäludele mõeldes teeme. Niisiis, sageduste vektori vastuvõetav pikkus kasvab üsna ruttu, ja *loendamis-meetod* kui termin on saanud uue sisu. Nende ridade autor oli rõõmsalt üllatunud, leides 2009. a. jaanuaris materjali [C_S], mis oleks kui meie „luuresordi" pealt maha kirjutatud (mida ta muidugi pole, see lähenemine on ju nii loomulik kui ka triviaalne).

Samas oli ka *Counting sort*i erijuhu – *Tally*[2] *sort* – mis on kasutatav kas siis, kui on teada, et järjestatava vektori elemendid on unikaalsed, või siis, kui (kõrval)eesmärgiks on duplikaatide eemaldamine, ning siis saame kasutada *sageduste vektori* asemel „märkimisvektorit" ning see võib olla *bitivektor*. Kui sageduste vektori tegemisel kasutasime aritmeetilist liitmist, siis bitivektori tegemisel tuleb kasutada bitiviisilist loogilist liitmist (just samuti, nagu pisut ülalpool rääkisime *VILLIS*e väärtusvarude vektorite moodustamisest).

Meie *luuresordi* programm tegeles pelgalt etteantud jada liikmete järjestamisega, ent selle töö käigus moodustatud *sageduste vektoril* võib olla ka muid kasutusi. Vihjasime sellele programmi teksti kommenteerides, et toda vektorit ei pruugi me mooduli lõpus kustutada, vaid võime ta tagastada väljakutsuvale moodulile – kui moodul programmeerida funktsioonina ning nii sageduste vektori pikkus kui ka elemendi minimaalne väärtus on globaalsed parameetrid. Vaatleme paari neist võimalustest.

- Kui vektoris on suhteliselt palju korduvaid elemente suhteliselt suurte sagedustega[3], siis võime jada liikmete summat S (näiteks eesmärgiga leida nonde elementide väärtuste aritmeetilist keskmist) arvutada valemiga $S=\Sigma a_i \times f_i$, kus a_i on elemendi väärtus ja f_i on selle sagedus ($i = 0..n-1$, n on vektori elementide arv).
- Sagedusvektori abil on triviaalne jada moodkeskmis(t)e leidmine
- Kui meid huvitab (kõrvaleesmärgina[4]) jada elementide unikaalsus (so, kõik elemendid on omavahel erinevate väärtustega), siis sagedusvektori lineaarse läbivaatuse võime katkestada niipea, kui sagedusvektori mingi elemendi väärtus on suurem kui 1.
- Kui tahame pärast sageduste leidmist muuta vektori väärtuste mõttes unikaalseks, siis piisab, kui kirjutame tagasi igast (suurema kui 1-sagedusega) väärtusest lähtevektorisse ainult ühe; sel juhul on järjestatud vektori pikkuseks sageduste vektorite nende elementide arv, mille väärtus on suurem kui 0 (see tuleb tagastada meetodi väljakutsujale).
- Me saame suvalist väärtust vahemikust *min...maks* sageduste vektorist otsida *ühe sammuga* (vastused on „on" ja „pole"), ajalise keerukusega $O(1)$.
- *Kui* vektorielementide *min* ja *max* on fikseeritud, siis saame sagedusvektorit kasutada *dünaamiliseks* järjestamiseks.

[1] Siinkirjutaja arvates võiksime nii kutsuda *laptop*-masinaid; *sülearvuti* tundub talle suisa pentsiku nimetusena. Ons keegi näinud kedagi, kes masinat *süles* hoiab ja seejuures sellega *midagi mõistlikku* teeb?
[2] *Tally* üks vaste on „duplikaat", see haakub pisut meetodi olemusega: duplikaatide elimineerimine.
[3] Liitmine on oluliselt „odavam" tehe korrutamisest, ent mingist piirist alates (liidetavate arv *vs* korrutaja) on ökonoomsem korrutada.
[4] Kui see on põhieesmärk, siis saame unikaalsuskontrolli negatiivse tulemusega lõpetada niipea, kui mingi väärtuse sagedus on suurem kui 1

6. Stringid

C-string[1] on *baidivektor* kirjeldusega kas *char string*[<konstant>]; (staatiline, konstantne) või *char *s*; (dünaamiline või juba adresseeritud stringi aadressi duplikaat). Varasemates tuntud keeltes oli – kui üldse – *string* spetsiifiline andmetüüp, millel polnud muude andmetüüpide ega -struktuuridega kuigivõrd ühist. Seevastu *C* jaoks on *string* tavaline vektor (indekseerimissammuga 1: baidi pikkus) ühe tähtsa iseärasusega: stringile tuleb reserveerida 1 võrra pikem väli vajalikust sümbolite arvust, põhjuseks on seik, et stringil peab olema (erinevalt muud tüüpi vektoritest) *lõputunnus*, milleks on stringi väljal paiknev, ent mitte stringi pikkusse lülitatav bait väärtusega '\0', so. bait, mille kõikide bittide väärtuseks on 0. Seega, kui tahame kirjeldada sümbolvektorit maksimaalselt 4-täheliste sõnade hoidmiseks, peame selle kirjeldama kui *char s6na*[5]. Näiteks, kui „sõna" on „karu", siis baithaaval kujutatakse ta aadressil *s6na* kui „karu'\0'" (16-ndesituses 6b61727500).

6.1. Stringifunktsioonid

Pärast *ANSI* (*American National Standard Institute*) standardi fikseerimist (1983[2]) pidi iga uus *C*-kompilaator realiseerima kokkulepitud moel 15 funktsioonide standardteeki, sh. teegi <*string.h*>, kus on kirjeldatud realiseerimisele kohustuslikult kuuluvad stringifunktsioonid: *strcpy*, *strncpy*, *strcat*, *strncat*, *strcmp*, *strchr*, *strrchr*, *strstr*, *strlen* ja mõned veel[3]. Allpool esitame neist mõnede algoritmid (laenates neid võimalusel [K&R]-raamatust) ja vahel improviseerides originaali vaimus. Mõnedele [K&R]-näidetele oleme lisanud kompileeritava keskkonna (*include*-makrod, *main*-moodul jmt).

- Stringi pikkuse leidmine (kõik kommentaarid on meilt):

```
//slen.c : 4.11.08
#include <stdio.h>

//K&R, lk. 99
int strlen1(char *s){
   int n;
   for(n=0; *s!='\0'; s++) n++;  //vt. joonealust märkust „4"
   return n;
}

//näitame, et kui viit on baitvektorile (stringile), siis s++ on sama, mis *s++[4]
int strlen2(char *s){
   int n;
   for(n=0; *s!='\0'; *s++) n++;
   return n;
}
```

[1] Vene k. строка, eesti keeles eelistavad mõned autorid terminit *sõne*.
[2] [K&R], lk. ix.
[3] Vt [K&R], lk. 249..250. Mainitud „mõnedest veel" moodustavad omaette grupi funktsioonid prefiksiga *mem*: need toimivad suvaliste andmeagregaatidega (vektor, kirje jne), kirjeldades andmeid kui *void** (ja käsitledes viitu kui *char **). Pisut tagapool kasutame funktsiooni *memset.*
[4] Tuletame meelde, et „**v*++" interpreteeritakse kui „(viit vektorile *v*) + (elemendi pikkus)". Kuivõrd stringi elemendi pikkus on 1, siis võime rahumeeles suurendada stringi aadressi „avalikult" ühe võrra, kirjutades *v*++.

```
//K&R, lk.103
int strlen3(char *s){
   char *p=s;
   while(*p!='\0') p++;
   return p-s;
  //viitade lahutamine; vahe ongi stringi pikkus (ilma lõpumarkerita)
}


//kasutame signaali[1] '0', mis saabub, kui *p väärtus on '\0'
int strlen4(char *s){
   char *p=s;
   while(*p) p++;
   return p-s;
}


int main( ){
   char t[100];
   printf("anna string: ");
   gets(t);
   printf("\npikkus1 on %d\n",strlen1(t));
   printf("\npikkus2 on %d\n",strlen2(t));
   printf("\npikkus3 on %d\n",strlen3(t));
   printf("\npikkus4 on %d\n",strlen4(t));
   getchar( );
}
```

- Stringide kopeerimine *strcpy*. Kommentaarid on meilt.

```
//strcpy: erinevad variandid. K&R: lk. 105 jj
#include <stdio.h>

char orig[128]={"originaalstring"};
char manu[32];

//stringcpy: copy t to s, array subscript version:
void stringcpy(char *s,char*t){
    int i=0;
    while ((s[i]=t[i])!='\0') i++;  //omistamine, kuni kopeeriti t[i] = '\0'
    }
```

[1] Signaalidest vt. näit. [Isotamm, PK, lk. 19, 25, 68, 84]

```
//sama töö viitade abil:
void strcpy1(char *s,char *t){
    while((*s=*t)!='\0'){
      s++;  //kui viimane kopeeritud sümbol oli lõputunnus, siis tsükkel lõppeb
      t++;
      }
    }

//kogenud C-programmeerijad teevad nii:
void strcpy2(char *s,char *t){
    while((*s++=*t++)!='\0');  //tsükli lõpetab viimasena üle kantud lõputunnus
}


/* siin kasutatakse tõika, et väärtuse '\0' omistamine tekitab signaali „0“, mida while
interpreteerib kui tõeväärtust false – see aga lõpetab tsükli.  */
void strcpy3(char *s,char *t){
    while(*s++=*t++);
}


int main( ){
   printf("orig=");
   gets(orig);
   printf("\n uus: ");
   gets(manu);
   stringcpy(orig,manu);
   puts(orig);
  strcpy1(orig,manu);
   puts(orig);
   strcpy2(orig,manu);
   puts(orig);
   strcpy3(orig,manu);
   puts(orig);
   getchar( );
}
```

- Stringide võrdlemine *strcmp*. Stringe s[ ] ja t[ ] võrreldakse baithaaval vasakult paremale, kusjuures „baithaaval“ tuleb lugeda nii, et võrreldakse sümbolite *ASCII*-koode (mis on konstrueeritud nii, et oleks säilitatud nende „loomulik“ leksikograafiline järjestus); näiteks[1]:

[1] vt. [CD, lk. 375 jj.]

Dec	Hex	Char
0	**00**	**NUL (Null)**
10	**0A**	**LF (Linefeed)**
27	**1B**	**ESC (Escape)**
32	**20**	**<space>**
34	**22**	**„**
42	**2A**	*****
48	**30**	**0**
57	**39**	**9**
65	**41**	**A**
90	**5A**	**Z**
97	**61**	**a**
122	**7A**	**z**

Tabel 6.1.a. Väljavõte *ASCII*-tabelist [CD, lk. 375].

Võrdlemisel omistatakse *int r=s*[*i*] – *t*[*i*] ja võrdlemine kestab, kuni üks võrreldavatest stringidest on otsas või kuni $r = 0$: kui $r < 0$, siis $s < t$ ja kui $r > 0$, siis $s > t$. Kui üks võrreldavatest stringidest osutus teise *prefiksiks*, loetakse ta *väiksemaks* (mis on ka formaalselt õige, kuivõrd lõputunnuse kood *NUL* on väiksem mistahes muust koodist). Kui võrreldavad stringid on sama pikkusega ja koosnevad samadest sümbolitest, siis on nad võrdsed (formaalselt, $NUL - NUL = 0$).

```
//strcmp:  stringide võrdlemine  K&R, lk. 106, 23.12.07
#include <stdio.h>

char s[128];
char t [128];

//indekseerimisega versioon
int strcmp(char *s, char *t){
   int i;
   //kui ükskõik kumb string on lühem, pole tingimus täidetud:
   for(i=0;s[i]==t[i];i++)  if(s[i]=='\0') return 0;
   return s[i]-t[i];
}

//viitadega versioon
int strcmp1(char *s,char *t){
   for(;*s==*t;s++,t++) if(*s=='\0') return 0;
//NB! Päise „esimene komponent“ on tühi ja „kolmandas“ on kaks operaatorit
   return *s-*t;
}

int main( ){
   int x;
```

```
    printf("anna s: "); gets(s);
    printf("anna t: "); gets(t);
    x=strcmp(s,t);
    printf("%d\n",x);
    x=strcmp1(s,t);
    printf("%d\n",x);
    getchar( );
}
```

- Stringide konkatenatsioon („liitmine“): parameetrid on stringide *s* ja *t* aadressid ning resultaadiks peab olema string *s*, mille „sappa“ on kirjutatud string *t* (kusjuures kasutaja peab ise hoolitsema, et *s* on defineeritud paraja varuga, vältimaks „üle otsa“ kirjutamist (millel on reeglina fataalsed tagajärjed – tavaliselt saame veateate *segmentation fault*).

```
//s ja t konkatenatsioon
// scon.c  :  s ja t konkatenatsioon
#include <stdio.h>
char s[128];
char t [32];

void strcon(char *s,char *t){
    while(*s) *s++;          //lõpetab, kui *s väärtus on '\0'
    while(*s++=*t++);    //kirjutab „sappa“, kuni kirjutab ümber '\0'
}

int main( ){
    int x;
    printf("anna s: "); gets(s);
    printf("anna t: "); gets(t);
    strcon(s,t);
    printf("%s\n",s);
    getchar( );
}
```

- Funktsioon *memset* . Ülalpool oli juttu sellest, et *void*-tüüpi viit võib viidata suvalist te-gelikku tüüpi objektile, ning et võimalusel interpreteerib *C* sedatüüpi viita kui viita ühebaidisele (*char*) objektile, kuivõrd just see on kõigi „kõrgemate“ struktuuride „aatom“. *UNIX*i keskkonnas (küsige >*man memset*) on meie funktsioon defineeritud nii

```
void *memset(void *s, char c, size_t n);
```

Funktsiooni *void*-tüüp vihjab kas sellele, et tegemist on protseduuriga, mis väärtust ei tagasta, või sellele, et tagastatakse viit *suvalist tüüpi* objektile **s* – sisendparameetrile. Selle *suvaline tüüp* tähendab, et *memset* töötab mistahes tüüpi vektoritega või kõrgemate andmestruktuuridega. Viidatud väli täidetakse *n* 1-baidilise *int*-väärtusega *c*.

Tavaliselt kasutavad programmeerijad seda funktsiooni täitmaks mäluvälja „masina-nullidega“, sel puhul on *c* väärtus ’\0’ ning *n* on välja pikkus baitides – mille välja-arvutamiseks on hõlpus kasutada funktsiooni *sizeof*( ).

Paar näidet:

```
char a[40];
...
memset(a,'\0',40),

int I[100];
...
memset(I,'\0',100*sizeof(int)),

struct b{
        int a[20];
        char c;
        struct b *next;
        };
struct b bb;
...
memset(bb,'\0' ,sizeof(struct b));
```

Loodetavasti näitasime, et *memset* on tõepoolest kasutatav mistahes tüüpi andmevälja puhul, ja seda tänu *void*-tüübile. Kui seda *C*-s poleks, tulnuks kirjutada iga tüübi jaoks oma „memset“. Päris-*memset* on võimeline kõiki *s-viitu interpreteerima kui **char*-viitu.

Lõpetame *stringide* käsitluse ühe üsna olulise seiga tutvustamisega. Nimelt on enamus *str*-funktsioonide argumente (kui nende väärtusteks on viidad stringidele) varustatud atribuudiga *const*, näiteks *gcc*-funktsioon *strcmp* kirjeldatud nii (vt. *UNIX*i keskkonnas *man strcmp* abil):

```
int strcmp(const char *s1, const char *s2);
```

Mõte on selles, et kaitsta argumente (mis on esitatud viitadena stringidele) funktsiooni eest, nimelt on viimasel võimalik viita kasutades kirjutada viidatud stringis midagi üle. Atribuudiga *const* teatatakse kompilaatorile, et viidatud string on konstant, mille mistahes modifitseerimist ei tohi lubada; tehkem vahet: konstant *pole* viit (antud juhul võib see viit viidata suvalisele stringile), vaid konstandina käsitletakse viidatud *stringi* (vt. ka [Jensen], osa „*const Pointers*“). Kompilaator peaks konstantideks kuulutatud objektide muutmise blokeerima.

Üsna tihti osutub võimalikuks stringi töötlemisel *lõpliku olekute hulgaga automaadi* kasutamine – neil juhtudel, kus algoritmi tööd juhivad stringi vasakult-paremale -järjekorras läbivaadatavad sümbolid. Lisas 5 on asjakohane näide: programmid teisendusteks erinevate arvusüsteemide vahel (*rooma* $\rightarrow$ *araabia* ja *araabia* $\rightarrow$ *rooma*).

Teistlaadi näide on Lisas 6 toodud *pseudo-tehisintellekti* programm *narinjani*, mis üritab luua illusiooni loomulikku keelt mõistvast rakendusest, ent kasutab ainult primitiivset stringitöötlust.

6.2. Tekstitöötlus

Nagu nägime, on *C* stringifunktsioonid pigem lihtsad, ent üldisemas plaanis on nad sootuks üldisema teema, *tekstitöötluse* elementaarsed töövahendid. Tekstitöötlus (laiemas mõttes) on aga palju vanem valdkond kui arvutiteadus, ulatudes üpris kaugesse aega. *Tekst* on sisuliselt vahend sõnumi edastamiseks ja sümbolid, millest too tekst koosneb, ei pruugi sugugi olla häälikutele vastavad märgid[1], on kasutatud ka kiil- või piltkirja, runomärkisid või tänapäevani hieroglüüfe. Kui kirjakeel pole foneetiline, siis üks sümbol võib väljendada nii tervet lauset, sõna või silpi (või midagi veel).

Meie oleme harjunud (eeskätt) 8-bitise *ASCII*-koodiga, kus „otseselt loetavad" sümbolid on suur- ja väiketähed, arusaadavad on numbrid ning kirjavahemärgid (sh. tühik) ning ülejäänud „nähtavat kuju" evivate märkide semantika on kokkuleppeline (so, õpitav instruktsioonide abil). Pole juhus, et esimeste personaalarvutite „laiatarbe"-tarkvarapakettide hulgas olid ühed edukamad just tekstitoimetid[2].

Ja *C*-vinklist vaadates võiksime nentida, et mistahes loetav tekst on pikk string, mida liigendavad lõigueraldajad (*ASCII 0d0ah*), lauseeraldajana *punkt* (*ASCII 2eh*), sõnaeraldajana *tühik* (*ASCII 20h*), ja lisaks mitmed semantilisi nüansse lisavad eraldajad nagu „;", „:" või „,".

6.3. Salakiri[3]

6.3.1. Sissejuhatus

Tekstitöötlusega on aga seotud *delikaatne valdkond* millega tegeleti aktiivselt ja resultatiivselt kaua enne[4] meie mõistes „masinate" leiutamist – tekstide šifreerimine[5] salastamist väärivate sõnumite sisu varjamiseks nii diplomaatias, äritegevuses kui ka sõjanduses. Rööbiti šifreerimisvõtete väljatöötamisega on salakirjanduse algusest alates tegeletud konkurentide või vaenlaste šifrite „murdmisega". Šifrite alal on seega kaks lahutamatult seotud töösuunda: esiteks, püütakse oma sõnumite salastamiseks välja töötada võimalikult murdmiskindlaid koode ja teiseks, püütakse teada saada võtmeid, mille abil saaks lugeda välisriikidelt kätte saadud diplomaatilist posti, eeskätt nende saatkondadesse laekuvate telegrammide kopeerimise teel[6] või raadioluurejaamade poolt kinni püütud võõraid šifreeritud sõnumeid[7].

---

[1] Hea näide on kaasaegne inglise või prantsuse keel, kus häälikute ja sümbolite vastavus on üpris keeruliselt kodeeritud. Näiteks, tahtes kirjutada prantsuse keeles nime *bordoo* peame kirjutama *Bordeaux*.

[2] Esialgu primitiivsed, sest neid kitsendasid olulisel määral riistvara-võimalused (*alpha-numeric display*, nõeltrükkalid piiratud fontidega (sh. eduka lahendusena vahetatava kuulpeaga printer, iga fondi jaoks oma pea)).

[3] Selles jaotises kasutame läbisegi (sümmeetrilisi) termneid „krüptimine"/"dekrüptimine", „kodeerimine"/"dekodeerimine" ja „šifreerimine"/"dešifreerimine", suuresti sellepärast, et me ei oska nende vahel sisulist vahet teha.

[4] Salakirja tundsid juba vanad egiptlased (me ei mõtle siin meile sellena tunduvat piltkirja); oma süsteemi mõtlesid välja ja kasutasid näiteks nii *Julius Caesar* kui ka *Napoleon*, ning mõtles välja ja kirjeldas oma novellis „*The Gold Bug*" *Edgar Allan Poe* [wHistory]

[5] Selle valdkonnaga tegeleb omaette teadusharu – *krüptograafia* (või *krüptoloogia*). Meie teaduskonnas on sellele pühendatud mitmed loengukursused, mida õpetavad aktiivse teadustöö kõrvalt doktorid *Ahto Buldas*, *Peeter Laud*, *Sven Laur* ja mitmed nende nooremad kolleegid. Seetõttu me ei üritagi krüpteerimise teooriat, probleeme ega algoritme nende kaante vahel tutvustada ning piirdume paari lihtsa näitega.

[6] Näiteks kehtis brittidel seadus, „mis käskis kõigil Suurbritannias esindatud telegraafifirmadel anda kümne päeva jooksul pärast saatmist või vastuvõtmist üle kõigi sõnumite koopiad. See andis küllaldase materjalivoo, ilma milleta on krüptoanalüüs võimatu". [Denniston, lk. 72]

[7] Huvitava ülevaate signaalluurest annab *Robin Dennisto*n raamat [Denniston] „30 sala-aastat. A. G. Dennistoni töö signaalluures 1914 – 1944", kust saame teavet luuretöö organisatsioonist ning töömeetoditest, ent mitte krüpteerimistehnikatest. *Alastair Dennisto*n meeskonnas töötas teiste hulgas ka *Alan Turing*. Mainitud raamatule tuginesime sageli käesolevat alajaotust kirjutades.

Maailmasõdade ajal ja nende vahel kasutas enamik riike lihtsat ning kiiret šifreerimist ja dešifreerimist võimaldavat koodi, mis tugines *koodiraamatutele.* Nende koostamise metoodilisi variante oli mitmeid, ent alati kasutati teksti kodeerimiseks *võtit*, mis kehtestas vastavuse salastatava (lähte)teksti ja krüptitud teksti sümbolite vahel.

Konstrueerime ühe primitiivse näite, millel pole tõenäoliselt mingit seost tegelikult kasutatud võtetega. Niisiis, paneme ennast agente kasutavas ametkonnas piisavalt kõrgele kohale, anname oma agentidele kaasa „juhuslikult" lugemiseks *R. Dennistoni* raamatu ning paneme nad meeles pidama, et tulevaste salakirjade võtmeks on tolle raamatu 23. lehekülje esimese rea algus: „*Suurbritannia andis oma Enigma saladusi ameeriklastele*".

Kui agent on tegutsemispiirkonnas (II Maailmasõda pole kas alanud, aga kindlasti mitte veel lõppenud) ning me saadame talle raadiotelegraafiga (koodiks *Morse tähestik*) radiogrammi

− − − − − − − − ·· − − − − − ·· − − − ·· − − − − − ··· − − − − − − − ··· · − − − − − − − − −

jne (vt. lisa 9) ning kogenenud raadiotelegrafistina kirjutab ta reaalajas üles kodeeritud sõnumi:

0802270710 2101145520 2017120609 1321150803 5924565100[1]

Lugejale: ärge kohe edasi lugege, katke lehekülje lugemata osa ning püüdke see sõnum dešifreerida. Kui see poole tunni jooksul õnnestub, siis oleksite Te tõenäoliselt olnud teretulnud arvutieelse ajajärgu signaalluure ametkondadesse (aga kindlasti siis, kui Te võtit ei vaata, vaid püüate selle ise rekonstrueerida).

Meie konstrueeritud näite triviaalne krüptimis-/dekrüptimiseeskiri on selline: kahekohaline arv on kirjatähe järjekorranumber (*stringi indeks*) võtmes: 08 = *T*, 02 = *U*, 27 = *G* jne, ning dešifreeritud sõnum on „*tugines koodiraamatutele*".

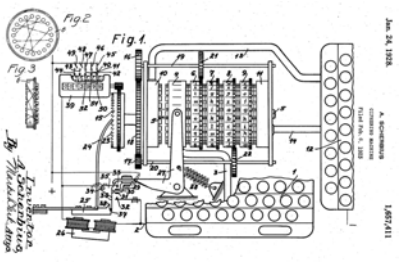
Koodiraamatuid võib vahel harva agentuurluure abil kätte saada, ent kindlam variant on koodi murdmine, so. võtme analüütiline tuletamine. See on võimalik ainult siis, kui ühe ja sama võtmega šifreeritud tekste on palju ning krüptoloog valdab teate saatja ning selle teate adressaadi keelt. Mõistagi võtab see ilma masinate abita küllaltki palju aega. Koodiraamatuid kasutavad riigid vahetasid võtmeid regulaarselt, tavaliselt iga 3 kuu järel, seega mingi aeg ei suudetud nende sõnumivahetust lugeda – kuni uue šifr murdmiseni[2].

Ülalpool märkisime, et enamus riike kasutas maailmasõdade-vahelisel ajal lihtsat koodiraamatute süsteemi ning selliselt kodeeritud sõnumid olid varem või hiljem kuni võtmevahetuseni mõneks ajaks loetavad. Erandi moodustasid kaks riiki: Saksamaa, kus I Maailmasõja lõpus leiutas insener *Arthur Scherbius* elektromehhaanilise kodeerimise ja dekodeerimise masina, mille nimeks sai *Enigma* (vt joonis 6.3.2.a) ning N. Liit, mis kasutas *Vernami koodi*[3].

[1] Morset edastatakse reeglina 5-sümboliliste *gruppidena*; viimases grupis on siin „täitesümbol „0"".
[2] Brittide jaoks oli meeldiv erand Itaalia, itaallastel „oli komme šifreerida päevalehtedes avaldatud pikki poliitilisi juhtkirju"" [Denniston, lk. 75].
[3] Seda käsitleme pisut hiljem.

6.3.2. Enigma





Joonis 6.3.2.a. *Wehrmacht*i *Enigma* (1943. a.) [wEnigma] ja tema autor, Arthur Scherbius ning joonis patenditaotluse kaaskirjast (http://upload.wikimedia.org/wikipedia/commons/b/b3/Scherbius-1928-patent.png).

Sakslaste masina tööpõhimõte seisnes lihtsustatult sõnumi mitmekordses kindlaksmääratud algoritmiga antud ümberkodeerimises, mida sai kiiresti dešifreerida sama algoritmiga varustatud masin, ent inimese jaoks käis see praktiliselt üle jõu (isegi kui ta kõiki teisendamissamme teab). Sõdadevahelisel ajal avalikustati ja turustati *Enigma* lihtsustatud kommertsvarianti, ent hiljem tehti sõjaväe jaoks selle salajane ning võimsam mudel. Mõistagi hoolitsesid sakslased, et ükski *Wehrmacht Enigma*-masin ei satuks ei oma liitlaste ega vaenlaste kätte ning luureagente nendega loomulikult ei varustatud, viimased kasutasid koodiraamatuid nagu muu maailma luurajadki. *Enigma* ei vajanud koodiraamatuid, masin kodeeriti ise iga päev ümber, „kruttides“ esimese versiooni puhul kolme ning viimase puhul viit „ratast“ märkidega *A, B,..,Z* (iga ratta aknas oli nähtav ainult üks täht). Uue päeva seanss algas selle päeva seadistuse (algversioonis 3 tähemärki, 1938. aastast 5 tähemärki) teatamisega eelmise päeva šifrit kasutades (eksituste vältimiseks saadeti seadistusešiffer kahekordselt, märgid olid mõistagi erinevad – ent just see seik, topeltedastamine, viis 3-märgise koodi murdmisele Poolas. Noor Poola matemaatik *Marian Rejewski* (23-aastane) oli õnneliku juhuse[1] ja oma leidlikkuse ja andekuse varal murdnud sakslaste koodi (3-tähelise seadistuse) ja dešifreerimiseks töötas 6st *Enigma* koopiast koosnev (3 esimese ja 3 teistkordse päevakoodi jaoks) plokk *bomba*, mis aga uue versiooniga hakkama ei saanud: vaja olnuks plokki 60 masinast. Poola krüptograafiaalane tegevus omal maal katkes sõja puhkedes. *Rejewski* jätkas tööd Suurbritannias[2].

Enigma töötas[3] sümmeetriliselt: kirjutusmasina-klaviatuurilt (märgid *A..Z*) sisestati kodeerimist vajav tekst ja fikseeriti töö käigus masina näidatavad sümbolite vasted käsitsi (krüptitud

[1] Üks asjaga seotud sakslane (*Hans-Thilo Schmidt*, 43-ne) müüs prantslastele asjaomast informatsiooni, mida viimased jagasid poolakatega.

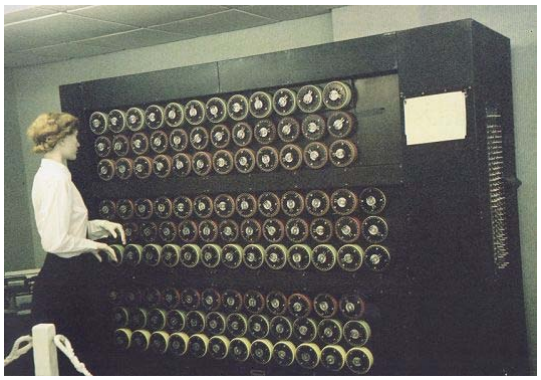
[2] Viitame siin nii *Enigma* kui ka Poolaga seonduvale allikat http://www.vectorsite.net/ttcode_08.html (vaadatud 12.08.09).

[3] Lihtsustatult töötas see masin nii: kirjutusmasina 26 tähemärki olid „ümber lülitatud“: esimese „rootori“ 26 märgile (näiteks, „F“ oli ühendatud „rootori“ „U“-ga, esimese rootori väljund teisele rootorile oli taas „müra“, näiteks, „U“ kodeeris „A“, ja juba mainitud „rattakeste“ abil sai iga rootori lülitusi ümber seadistada, iga seadistuse muutmine muutis kõik kahe rootori vahelised signaalid. Viierootorilise variandi põhimõte oli sama. Asja tegi „murdmiseks“ keeruliseks seik, et kolme „rootori“ puhul oli võimalike ümberlülituste arv 26×26×26 = 17 576, viie-rootori-variantide arv oli veel 26×26 korda suurem. „Rootor“ viib mõtte millelegi kogu aeg pöörlevale, ent siin kasutati seda terminit viitamaks ümberlülituste võimalikkusele, lihtsusele ja rotatsioonile.

tekst) ning kirjutati jooksvalt üles ja saadeti morsekoodis eetrisse, vastuvõtja(d) tippis(id) selle oma masina(te)sse ja kirjutas(id) üles näidatavad sümbolid, taastades nii esialgse sõnumi.

Teise Maailmasõja eelõhtul oli brittide käsutuses kommerts-*Enigma*, ent nad ei suutnud lugeda selle militaarvariantide koode. 1939. aasta suvel konsulteerisid Suurbritannia, Prantsusmaa ja Poola luureohvitserid selles asjas (Poola oli ju otseses ründeohus ning neil oli brittide ees teatud edumaa[1]) [Denniston, lk. 142 jj.]).

Märkimisväärse juhusena kutsus *A. Denniston Alan Turing*i oma meeskonda alates 1. septembrist 1939, kes pakkus välja idee ehitada *Enigma* murdmiseks tema nägemusele vastav masin; see sai valmis 18. märtsil 1940 ning sai koodnimeks „pomm“ (*Bombe*)[2]. On märgitud, et põhimõtteliselt järgis ta poolakate konstrueeritud masinat, ent komponentideks olevaid *Enigma*-masinate põhimõttelisi „kloone“ oli rohkem. „Pomm“ osutus edukaks, saades hakkama ka viiesümbolilise seadistamis-koodiga.



Joonis 6.3.2.b. „TuringBombeBletchleyPark.jpg“ [wBombe]

R. Denniston kirjutab, et „*Enigma*t poleks saanud 1940. aastal murda ilma *Knox*i 1930. aastate teoreetiliste arutlusteta ja *Turing*i matemaatilise geniaalsuseta ning *Welchman*i tehnoloogiliste lahendusteta“ [Denniston, lk. 86]. Lõpetuseks märgime, et lahti murti tolle masina *loogika*, seda tegi lihtsama versiooni jaoks *Rejewski* ning lõppversiooni jaoks *Turing.*

6.3.3. Vernami šifr

Joonis 6.3.3.a. *Gilbert S. Vernam* (1890 – 1960) [GSV]

Ühekordselt kasutatava võtme meetodi (ingl. k. tavaliselt *One-Time Pad*) mõtles 1917. aastal välja USA kompanii *AT&T* noor kaastöötaja *Gilbert Vernam* šifreerimaks teletaibiga edastata-

[1] Kohtumise ajal nad värskeid sõnumeid *Enigma* modifitseerimise tõttu enam lugeda ei saanud, mis viis nii britid kui ka prantslased mõttele, et neid lollitatakse.
[2] Kindlasti järgides poola projekti nimetust. Me ei tea, kas *Turing* oli tuttav *Rejewski* lahendusega või tema endaga.

vat infot. Teletaip oli elektromehhaaniline (perfo)seade, mida kasutati telegraafsideks posti-, raudtee-, valitsusasutuste jmt. vahel[1]; see tagas telegraafiliinide kaudu reaalajaside, mida sai toetada perfolintsisendi ja/või väljundiga. *Vernam*i leiutis kombineeris aparatuurselt sõnumilinti võtit sisaldava sama pika perfolindi omaga[2] [wOTP], kasutades *Boole'*i operaatorit *välistav või* (*exclusive or*, *C*-keeles „^“)[3].

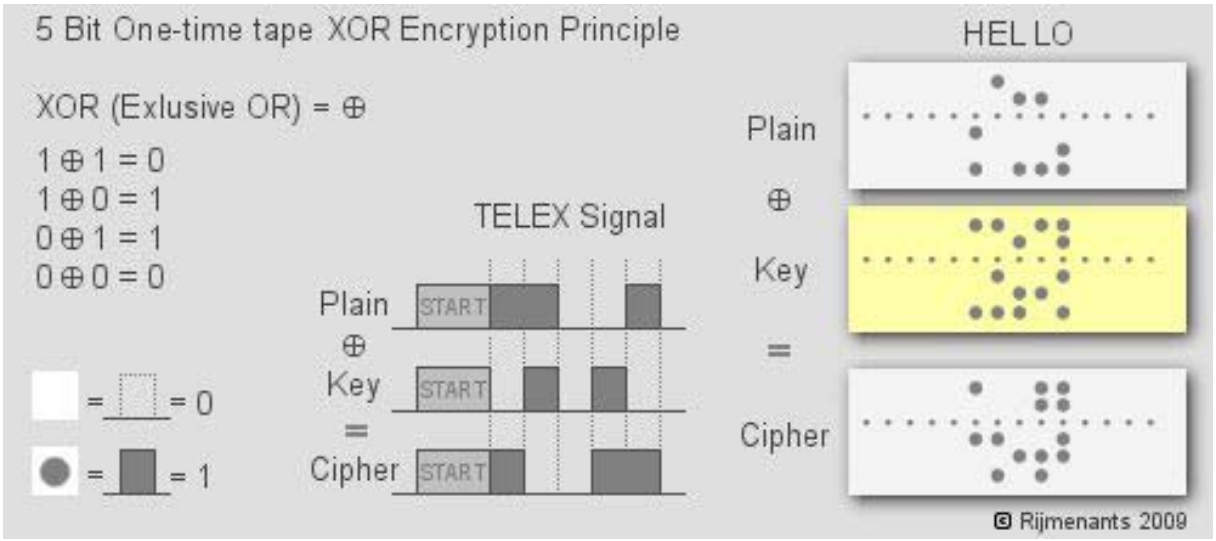
Joonis 6.3.3.b. Inglise matemaatik *George Boole* (1815 – 1864).

Mainitud võtet illustreerib alltoodud joonis (6.3.3.c). Loodame, et eestikeelset tõlget pole vaja. Enne Teist Maailmasõda täiustasid *Vernam*i süsteemi sakslased, sõja ajal inglased ning teineteisest sõltumatult *Vladimir Kotelnikov* (N. Liit) 1941. aastal ja *Claude Shannon* (USA) sõja lõpul. [wOTP]. Sõjaväelise agentuurside jaoks teletaip mõistetavatel põhjustel ei sobinud. Sestap töötati välja *Vernam*i süsteemi „pliiatsi-ja-paberi-variant“, agentidele anti pisiformaadis võtmeraamatud, mis tuli hävitada kohe pärast dešifreerimist[4]. Allpool reprodutseerime nii teletaibi (joon. 6.3.3.d) kui ka pisikese võtmemärkmiku (*pad*, joon. 6.3.3.e) pildid.

Ühekordsus on seejuures vägagi oluline, kuivõrd šifrid on üldjuhul murtavad vaid suure hulga sõnumite statistilist analüüsi kasutades. Kui võtit kasutada tõepoolest ühekordselt, siis ei teki vähimatki statistilise analüüsi võimalust. Soodsal juhul saab šifrit murda ka siis, kui käepärast on vaid paar-kolm kodeeritud teksti – juhtudel, kui koodi murdjal on õnne ja kodeerijal (ettevaatamatusest) pole.

Britid nentisid, et „Alustasime 1919. aastal kiviaegsete meetodite ehk alfabeetiliste raamatutega, jälgisime seejärel iga riigi turvameetmete arengut, jõudsime 1939. aastal täieliku teadmistepagasini kõigi kasutatavate meetodite kohta ja oskasime lugeda kõigi suurriikide diplomaatilist suhtlust, välja arvatud nende riikide (nagu Saksamaa ja Venemaa) puhul, kes olid sunnitud[5] kasutama *Vernam*i šifrit“ [Denniston, lk. 74]. Osundatud raamatu joonealune märkus samal leheküljel seletab viimase mõiste lahti: „*Vernam*i šiffer on süsteem, kus sõnumi krüptimiseks kasutatakse juhuslikult loodavat ja ainult ühekordselt kasutatavat võtit, mida saab dekrüptida ainult sama ühekordset šifrit ja võtit omav kasutaja“.

[1] Muuhulgas oli teletaip *Minsk*-22 sisendperforaatoriks paberlindile ja võis täita ka väljundperforaatori rolli; lint oli 5-realine. Arvutuskeskuste ajal olid paljud Eesti sedalaadi asutused omavahel seotud vastavate kanalitega.
[2] *Vernam* patenteeris oma leiutise 1919. aastal ja seega oli ta üldkättesaadav. Perfolindi pilti vt. näit. [Isotamm, PK, lk. 31].
[3] Krüptijate jaoks on see ideaalne tehe: kodeerimiseks tuleb rakendada sõnumile võtit (kasutada *XOR*-tehet), saata sõnum ära, ja vastuvõtja kasutab **täpselt sama võtit** (ja *XOR*i) šifreeritud sõnumi dekodeerimiseks. Seega: neil mõlemil on sama šifr ja täpselt sama „programm“ (sümmeetriline algoritm).
[4] Siit on ka pärit termin *one time pad* (*OTP*); „ühekordne märkmik“, *pad*'i paljudest tähendusest üks ongi just „märkmik“.
[5] Siinkirjutaja ei oska kommenteerida originaali epiteeti „sunnitud“.



Joonis 6.3.3.c. Teletaibi-šifreering [*One-time Pad*].

Joonis 6.3.3.d. *Siemens*i teletaip (all paremal olev seade manipuleerib perfolindiga) [wSiemens]

Joonis 6.3.3.e. *OTP*-märkmik [One-time Pad].

Briti vastuluurajad nendivad, et see šiffer on väga suure tõenäosusega murdmatu[1]. Oluline on, et kõik sõnumivahetajad *hävitaksid viivitamatult* šifri pärast sõnumi saatmist/lugemist; selle hõlbustamiseks anti agentidele spetsiaalsed tselluloidlindid, millele nad kirjutasid järgmise sideseansi šifri; neid linte oli väga lihtne süüdata ning nad põlesid tuhka jätmata. Miks nii: krüptoanalüüsiks on vaja statistika mõttes piisavat andmekogumit; kui šifrit kasutatakse tõepoolest ühekordselt, siis pole juttugi statistiliselt usaldusväärsest andmekogumist.

Tekib küsimus, miks käsitsišifreerijad/dešifreerijad ei kasutanud kõik turvalist *Vernam*i šifrit, ja vastus võiks olla, et lihtne koodiraamatu-süsteem oli oluliselt kiiremini krüptitav ja dekrüptitav, *Vernam*i koodi puhul tuli sõnumi iga sümbol käsitsi „välja *arvutada*" ja operatiivolukorras ei pruukinud selleks alati aega olla. Piisavalt sage koodiraamatuvahetus võis agendi kindlamini hoida kauem efektiivse ja elusana.

Märgitakse (näit. [wOTP]), et *Vernam*i šifr on murdmiskindel siis, kui on täidetud samaaegselt *kõik* neli järgmist tingimust:

- võti peab olema (vähemalt) sama pikk kui krüptitav tekst;
- võti tuleb *genereerida* võimalikult juhuslikuna (selleks ei sobi näiteks tänapäevased pseudojuhuarvude *random*-tüüpi generaatorid);
- võtmest tohib olla *ainult kaks* koopiat: üks saatjal, ja teine vastuvõtjal;
- peale ühekordset kasutamist tuleb võti *hävitada.*

Äsjaosundatud allikas on lk. 9 toodud neli juhtumit, kus vähemalt ühe tingimuse rikkumine põhjustas koodi murdmise[2]. N. Liidu kodeeritud tekste hakkas USA lugema 1942. aastal ja tegi seda 1947. aastani tänu sellele, et venelased hakkasid oma vanu võtmeid taaskasutama [Goebel].

*Vernam*i šifri kasutamine oli käsitsi märkimisväärselt töömahukas, ent arvuti jaoks on see lihtsalt programmeeritav ning kiire meetod. Allpool toome näite mõlemi variandi jaoks; alustagem käsitsišifreerimisest (ehkki automaatvariant teletaibiga on vanem ja parem).

Esimene näide (käsitsikodeerimine) pärineb allikast [wOTP, lk. 4]. Teksti kodeerimiseks on mugav, kui sümbolid saavad väärtusteks „koodid", näit. A=0, B=1 jne, Z=25 (inglaste versioonis, nende tähestikus on 26 tähemärki).

Näites edastatakse tekst *THISISSECRET* (*This is secret*, „see on salajane", sõnavahedeta) ning šifreerimisel kasutatakse võtit *XVHEUWNOPGDZ.* Sümbolite asemel kasutatakse nende koode (0..25), töötlemine käib sümbolhaaval vasakult paremale; illustreerime seda lihtsa programmiga:

```
//VBH.c :: Vernam By Hand  30.07.09 Vernami šiffer
#include <stdio.h>
#include <string.h>
```

[1] Meie märkus: kui vastupidi pea 100%-sele tõenäosusele õnnestukski üks sõnum lugeda, ei annaks see midagi (koodi murdmise aspektist), kuivõrd kõik muud sõnumid on kodeeritud muude, samuti ühekordselt kasutatud võtmete abil.

[2] Üks mõtlemapanevamaid oli järgmine: vene pool genereeris „täiesti juhuslikud" võtmed elukutseliste masinkirjutajate abil: nad pidid kahe käega vajutama klahve „nagu-pähe-tuleb", ent vajutasid ikka neid klahve, mida harjunud olid, so. klaviatuuri harjumuspärases valikus, kusjuures vene klaviatuur on konstrueeritud just venekeelse teksti tähtede esinemissagedusi silmas pidades. Kood murti, kuivõrd võti polnud piisavalt *juhuslik.*

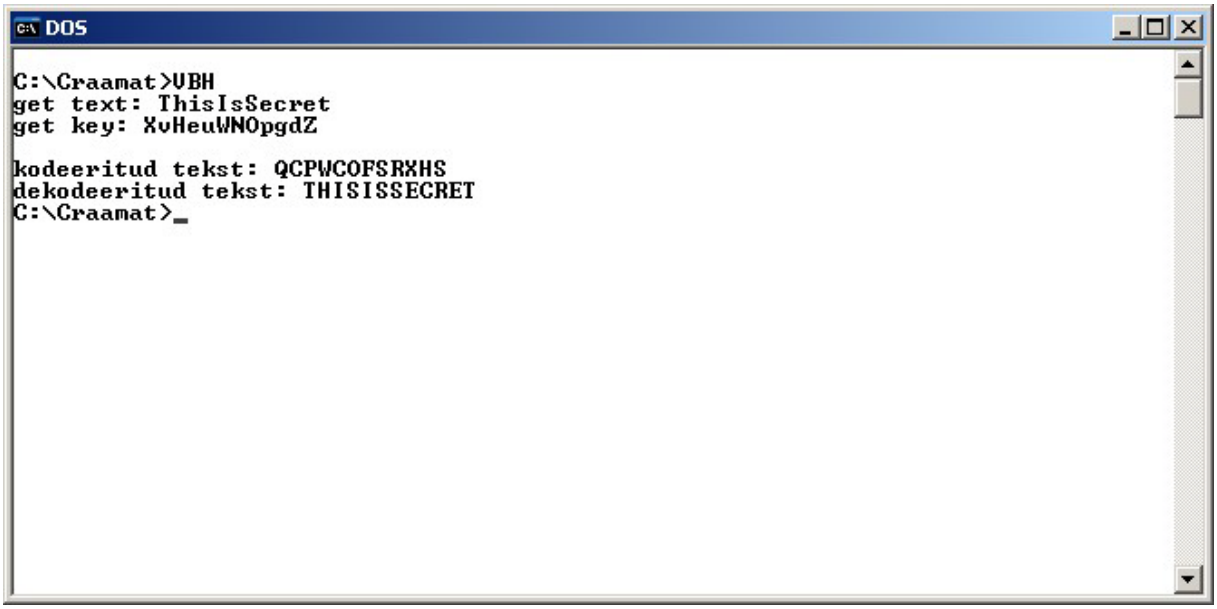
```
int main( ){
   char text[128];   //VBH on pelgalt demo-programm, sõnumid on lühikesed
   char key[128];
   int i,n;
   char c;

//kysin salastatava teksti
   printf("get text: ");
   gets(text);
   n=strlen(text);
   for(i=0;i<n;i++) text[i]=toupper(text[i])-65; //teisendab suurtähtedeks ja kodeerib 0..25
   //'A' ASCII-kood on 65, 'B'=66,...,'Z'=90.
//kysin v6tme
k:  printf("get key: ");
   gets(key);
   if(strlen(key)<n){
      printf("key is too short");
      goto k;
      }
   for(i=0;i<n;i++) key[i]=toupper(key[i])-65;  //võti peab olema suurtähtede jada: teen.
//kodeerin teksti lähteteksti väljale
   for(i=0;i<n;i++){
      c=text[i]+key[i];
      if(c>25) c-=26;
      text[i]=c;
      }
   printf("\nkodeeritud tekst: ");
   for(i=0;i<n;i++) printf("%c",text[i]+65);
//dekodeerin teksti lähteteksti väljale
   for(i=0;i<n;i++){
      c=(text[i]< key[i]) ? text[i]+26 : text[i];  //tingimuslik omistamine[1]
      text[i]=(c-key[i])+65;
      }
   printf("\ndekodeeritud tekst: %s",text);
   return(0);
}
```

Selle programmi testimist „raamatunäitega“ illustreerib joonisel 6.3.3.f kujutatud ekraanipilt.

Käsitsi sõnumite kodeerimine ja (eriti) dekodeerimine olid ilmselt üsnagi aeganõudvad tööd. Äsjaosundatud allikas [wOTP] on toodud esiteks kodeerimisnäide (vt. joonis 6.3.3.g).

[1] Loe: kui text[i]< key[i], siis c=text[i]+26, muidu c=text[i];



Joonis 6.3.3.f. *Vernam*i „käsitsi-šifreerimise“ imitatsioon

```
 Text:          T  H  I  S  I  S  S  E  C  R  E
T
               19 07 08 18 08 18 18 04 02 17 04
19
 OTP-Key:       X  V  H  E  U  W  N  O  P  G  D
Z
              +23 21 07 04 20 22 13 14 15 06 03
25
              ----------------------------------
-
 Result:       42 28 15 22 28 40 31 18 17 23 07
44
 Mod 26 =      16 02 15 22 02 14 05 18 17 23 07
18
              ----------------------------------
-
 Ciphertext:    Q  C  P  W  C  O  F  S  R  X  H
S

 Ciphertext:    QCPWC OFSRX HS
```

Joonis 6.3.3.g. Kodeerimisnäide.

```
   A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X
Y  Z
  ------------------------------------------------------------------------
----
  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25
  26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
50 --
```

Joonis 6.3.3.h. Teisendustabel.

Rõhutame, et šifreeritud teated edastati morsekoodis (tavapäraselt 5-sümboliliste gruppidena) ja *tähtedest* koosneva tekstina. Seega pidi *Vernam*i šifri kasutaja kodeerima teksti (ja šifri) numbriliseks ja arvutama iga sümboli koodi (ning dekodeerimise ajal asendama selle sümboliga). Käsitöö hõlbustamiseks (ja peaasi, kiirendamiseks) kasutati teravmeelseid abivahendeid. Esimene neist oli "teisendustabel" (*conversion table*), vt. joonis 6.3.3.h (vt. [wOTP]). Kuidas selle kasutamine kodeerimist lihtsustab, peaks lugejale selgeks saama, kui ta vaatab üle meie programmi *vbh.c* tekstist asjakohased operaatorid.

Juhime lugeja tähelepanu tõigale, et teisendustabelis on igal tähel ('A'...'Z') kaks "koodi": järjekorranumber tähestikus (0..25) k_a ja lisakood k_l: $k_l += k_a+26$.

Teine abivahend on pärit kaugest minevikust: see on nn. *Vigenère'i ruut.* Allpool laename taas allikalt [wOTP] selle "ruudu" näite (joonis 6.3.3.i). Kasutamisõpetus on selline: "tähe *H* krüptimiseks võtmega *E* leidke lahter *E*-rea *H*-veerust (näites on seal täht *L*). Seega, $H \rightarrow L$.

Tähe *L* dekrüptimiseks võtmetähe *E* abil võtke *E*-rida ning otsige sealt tähte *L*. Selle veeru "nimisümbol" *H* on otsitav vaste".

```
    A B C D E F G H I J K L M N O P Q R S T U V W X Y
Z
  +-----------------------------------------------
-
 A | A B C D E F G H I J K L M N O P Q R S T U V W X Y
Z
 B | B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A
 C | C D E F G H I J K L M N O P Q R S T U V W X Y Z A
B
 D | D E F G H I J K L M N O P Q R S T U V W X Y Z A B
C
 E | E F G H I J K L M N O P Q R S T U V W X Y Z A B C
D
 F | F G H I J K L M N O P Q R S T U V W X Y Z A B C D
E
 G | G H I J K L M N O P Q R S T U V W X Y Z A B C D E
F
 H | H I J K L M N O P Q R S T U V W X Y Z A B C D E F
G
 I | I J K L M N O P Q R S T U V W X Y Z A B C D E F G
H
 J | J K L M N O P Q R S T U V W X Y Z A B C D E F G H
I
 K | K L M N O P Q R S T U V W X Y Z A B C D E F G H I
J
 L | L M N O P Q R S T U V W X Y Z A B C D E F G H I J
K
 M | M N O P Q R S T U V W X Y Z A B C D E F G H I J K
L
 N | N O P Q R S T U V W X Y Z A B C D E F G H I J K L
M
 O | O P Q R S T U V W X Y Z A B C D E F G H I J K L M
N
 P | P Q R S T U V W X Y Z A B C D E F G H I J K L M N
O
 Q | Q R S T U V W X Y Z A B C D E F G H I J K L M N O
P
```

```
 R | R S T U V W X Y Z A B C D E F G H I J K L M N O P
Q
 S | S T U V W X Y Z A B C D E F G H I J K L M N O P Q
R
 T | T U V W X Y Z A B C D E F G H I J K L M N O P Q R
S
 U | U V W X Y Z A B C D E F G H I J K L M N O P Q R S
T
 V | V W X Y Z A B C D E F G H I J K L M N O P Q R S T
U
 W | W X Y Z A B C D E F G H I J K L M N O P Q R S T U
V
 X | X Y Z A B C D E F G H I J K L M N O P Q R S T U V
W
 Y | Y Z A B C D E F G H I J K L M N O P Q R S T U V W
X
 Z | Z A B C D E F G H I J K L M N O P Q R S T U V W X
Y
```

Joonis 6.3.3.i. *Vigenère*'i ruut.

6.3.4. Vernami šifri „masinanäide“

*Vernam*i meetod on vägagi lihtsalt ja lühidalt programmeeritav, ja kui selle kasutajad (saatja ja vastuvõtja) kasutavad ainult ühekordseid võtmeid (vältimaks koodi murdja jaoks statistilist analüüsi võimaldava infovoo tekkimist) on ta ka tänapäeval – kui uskuda *Interneti* – praktiliselt dešifreerimatu. Põhiprobleem on selles, kuidas genereerida (pseudo)juhuslik võti[1].

Meie raamatu programmeerimiskeel on *C*. Loomulikult me loodame, et saame võtme genereerida nii, et leiame šifreeritava „teksti“ (tegelikult baidijada) pikkuse *n* ning paneme *stdlib*-teegis leiduva funktsiooni *rand n*-kordsesse tsüklisse, kasutades oodatavate juhuarvudena selle funktsiooni väljundi viimase baidi väärtusi. Mainitud funktsioon väljastab väärtusi lõigust 0 kuni *RAND_MAX* (mille väärtus on fikseeritud failis *stdlib.h*) küllaltki ühtlase jaotumusega, ent krüptimiseks on ta täiesti kõlbmatu, kuivõrd ta genereerib *alati* sama arvuderea – genereerimist alustatakse alati „abiarvuarvu“ *seed*=1 kasutades.

Asja parandab pisut funktsiooni *srand* kasutamine, mis asendab funktsiooni *rand* lähteargumendi *seed* vaikeväärtuse „1“ uuega – see tuleb *srand*-funktsioonile parameetrina ette anda. Seega, kui meie eesmärk on koodi murdja tööd raskemaks muuta, siis ei tule muutuja *seed* väärtust mitte „sisse programmeerida“ (kui nii teeksime, siis saaksime taas programmi korduval lahendamisel sama „juhuarvude“ jada), vaid leida kuidagi teisiti.

Allpool esitame improvisatsiooni etteantud teemal *Vernam* ning juhuarvude generaatori *rand* stardiargumendi *seed* väärtusena kasutame päisfailis *time.h* kirjeldatud funktsiooni *time* väljundit; see on *time_t*-tüüpi ning kujutab endast jooksvat sekundite arvu, mis on möödunud alates 1970. aasta 1. jaanuari kella 00:00:00-st. *Seed*ina kasutame selle *int*-arvu viimast baiti[2].

Niisiis, näiteprogramm võtme genereerimiseks:

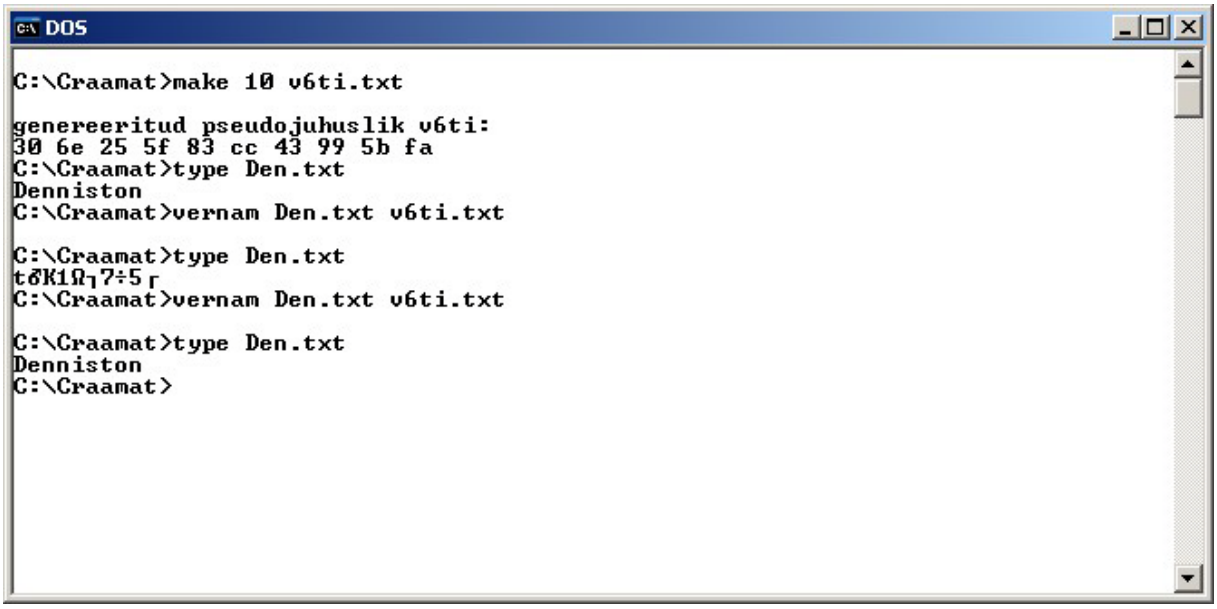
[1] Algoritmiliselt genereeritud „juhu“-arvude jada ei saa põhimõtteliselt olla *juhuslik*. Ent genereerimisalgoritm võib olla piisavalt keeruline, et selle jälile saamine mõistliku ajaga oleks vähetõenäoline.
[2] Mõistagi on ka see šiffer naiivne ja hõlpsasti murtav, ent siinkirjutaja arvates sobib küll algoritmi olemust illustreerima.

```
//make.c: Vernami xor-shifrile võtme genereerimine
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
    int i,n;
    char mask='\xff';
    char *key;

void makekey(int len){
    time_t timm;
    unsigned char m;
    key=malloc(len);
    clock( );
    timm=time(NULL);
    m=timm&mask;
    printf("\ngenereeritud pseudojuhuslik v6ti:\n",m);
    srand(m);
    rand( );
    memset(key,'\0',len);
    for(i=0;i<len;i++){
        m= rand( )& mask;
        key[i]=m;
        printf("%x ",m);
        }
}

//parameetrid: võtme pikkus, võtmefaili nimi
int main(int argc, char **argv){
    FILE *K=NULL;
    if(argc!=3){
       printf("parameetrite arv ei klapi");
       return(1);
       }
    K=fopen(argv[2],"wb");
    if(K==NULL){
       printf("%s: create failed",argv[2]);
       return(1);
       }
    n=atoi(argv[1]);
    makekey(n);
    fwrite(key,n,1,K);
    fclose(K);
    return(0);
}
```

Järgmisel ekraanipildil on võtme genereerimine (mitte pikema kui 10-sümbolilise teksti šifreerimiseks) ja salasõnumi *Denniston* šifreerimine ning dešifreerimine.



Joonis 6.3.4.a. Võtme genereerimine ning teksti šifreerimine ja dešifreerimine.

Šifreerimis- ja dešifreerimisprogramm võib olla näiteks selline, nagu allpool esitame (nimeks on meetodi autori auks *vernam.c*), mille kohustuslikud parameetrid on salastatava teksti täisnimi (so., koos nimelaiendiga) ja võtmena kasutatava faili täisnimi. Võti ei tohi olla lühem tekstist (meie programm seda ei kontrolli), pikem võib ta küll olla, sel juhul võib kasutada fakultatiivset kolmandat parameetrit, mis määrab *nihke*[1]: võtmefaili hakatakse kasutama alates baidist aadressiga „võtmefaili algus+nihe“. Konspiratsiooni huvides *kirjutatakse töödeldav tekst alati üle:* šifreerides – kodeeritud tekstiga ning dešifreerides – lähtetekstiga. Niisiis, programm *vernam.c*:

```
//vernam.c: Vernami xor-shiffer 27.07.09
#include <stdio.h>
#include <stdlib.h>

FILE *T=NULL;   //shifreeritav/deshifreeritav tekst
FILE *V=NULL;   //v6tmefail
FILE *S=NULL;   //shifreeritud/deshifreeritud tekst

//parameetrid: dokument, võti, [nihe]
int main(int argc,char *argv[ ]){

  int i, nihe=0;
   unsigned char c,v,t;
```

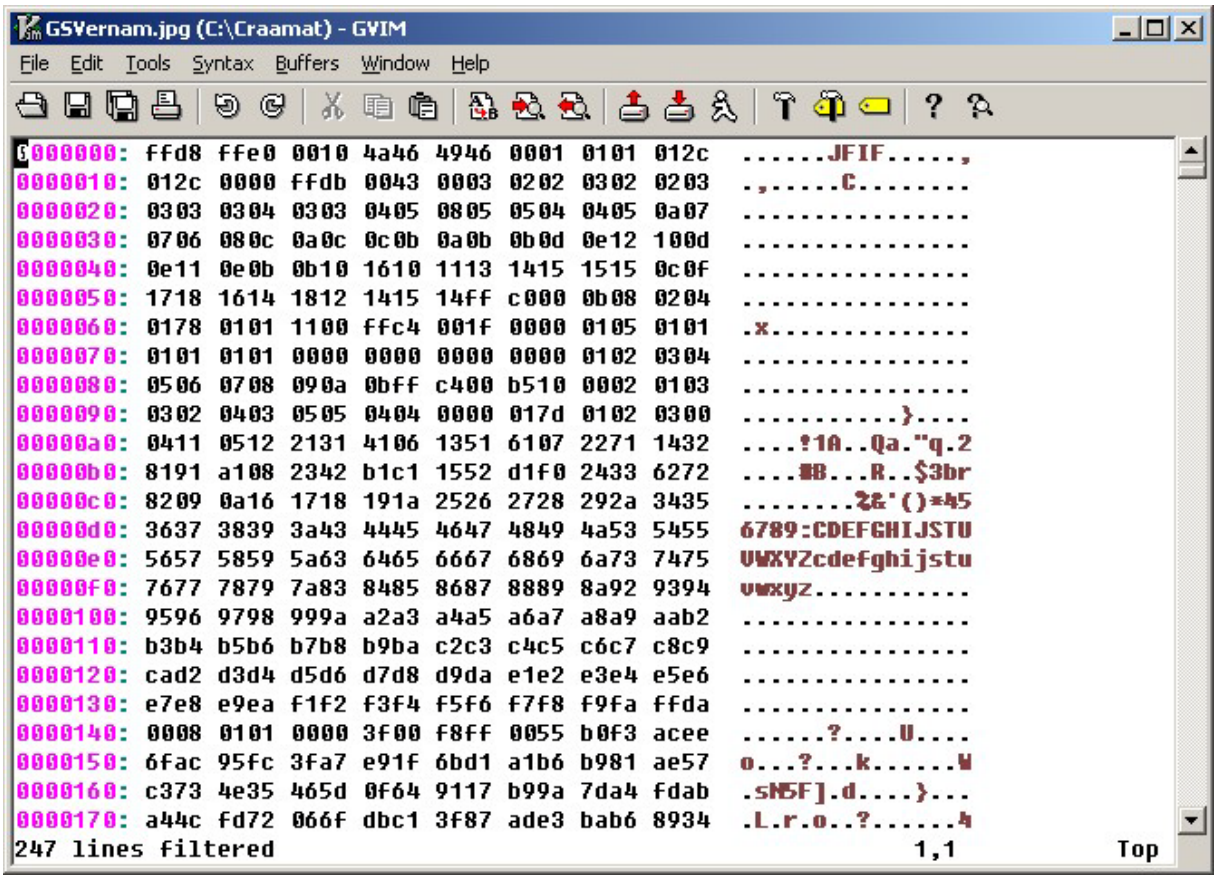
[1] Vaikimisiväärtus on 0.

```
    if(argc<3){
        printf("fatal error");  //see teade on spioonide maailma võtmes
        return(1);
        }
  if(argc==4)nihe=atoi(argv[3]);   //argv[0] on programmi nimi, edasi parameetrid
  T=fopen(argv[1],"rw");    //shifreerimist vajav tekst
  if(T==NULL){
    printf("%s opening error",argv[1]);
    return(1);
    }
  V=fopen(argv[2],"r");   //võtmena kasutatav fail
  if(V==NULL){
    printf("%s opening error",argv[2]);
    return(0);
    }
  S=fopen("t___p.txt","w");   //ajutiselt säilitatav shifreeritud sõnum
  if(S==NULL){
    printf("cannot create temporary file");
    return(1);
    }
  for(i=0; i<nihe; i++) t=fgetc(V);   //võtme häälestamine nihke suhtes
  while(!feof(T)){
    c=fgetc(T);    //originaalsümbol
    v=fgetc(V);    //võtmesümbol
    c^=v;          //
    t=fputc(c,S);
    }
  fclose(T);
  fclose(V);
  fclose(S);
  remove(argv[1]);   //kustutan lähtefaili
 //ajutine → lähtefail
 if(rename("t___p.txt",argv[1])!=0){
       printf("rename failed");
       return(1);
       }
 return(0);
}
```

Nagu püüdsime näidata, on triviaalse algoritmi abil genereeritud võti tõenäoliselt hõlpsasti murtav. Üha keerulisemate algoritmidega genereeritud võtmete kõrval võiksime otsida muid, mitteüldiseid ja -üldistatavaid variante. Üheks võimaluseks võiks olla variant, kus saatja ja vastuvõtja lepivad märke maha jätmata kokku, millist *faili* nad kasutavad šifreerimiseks, ainus tingimus on, et see on mõlemas arvutis (olemas või kättesaadav) ning nad (sõnumivahetajad)

teavad, mis fail see on[1]. Seejuures pole üldse oluline „võtmefaili“ tüüp: see võib olla nii *.txt*, *.exe*, *.doc*, *.bmp*- kui ka *.mov*-, *mp*3- või *.rar*-fail. Ja kui sõnumivahetajate arvutites on tuhandeid faile ning pärast valgustkartvat infovahetust kustutatakse mõlemast masinast „võtmefail“, siis on vähe võimalusi saladuse paljastamiseks – tingimusel, et seda võtit rohkem ei kasutata.

Illustreerigem sedagi varianti, seejuures *võtmefailina* kasutame asjakohaselt *Gilbert S. Vernam*i fotoportreed, mille reprodutseerisime osa 6.3.3 alguses, failinimega *GSVernam.jpg*. See pilt on arvutis tallel järgmise digitaalse failina (joonisel 6.3.4.b on faili algus):

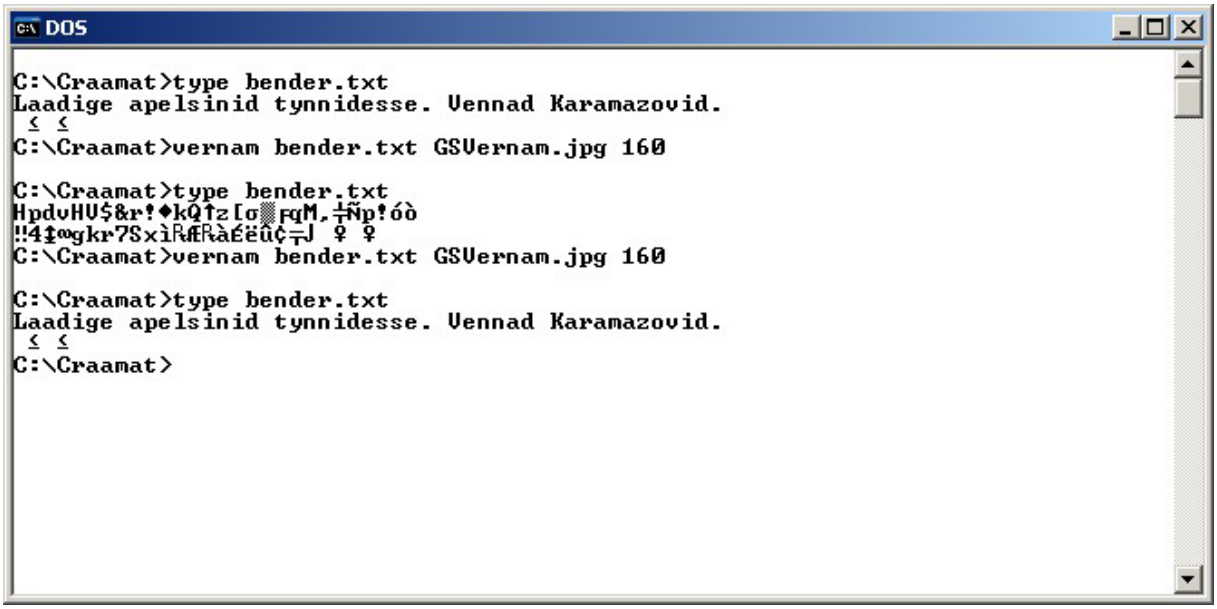


Joonis 6.3.4.b. *Vernam*i foto 16-ndkood kui salakirja võti.

Faili kasutamist *võtmena* võib mõistagi rohkem hägustada, näiteks nii, et võti on alguse suhtes nihkes, seda võimalust pakub ka meie programm. Järgmisel ekraanipildil on šifreeritud ja dešifreeritud *Ostap Bender*i kuulsat telegrammi salamiljonär *Koreiko*le.

Märkigem veel, et kui ajalooliselt vahetati delikaatset informatsiooni tekstisõnumitena (sj. tihtipeale *Morse* tähestikku kasutades), siis tänapäeval on võimalusi sootuks rohkem ning valgustkartvat informatsiooni saab vahetada (näiteks meie programmi kasutades) heli-, pildi-, video- jne. failidena

[1] See võib olla näiteks sõnumi saatmise päeval ilmunud „Eesti Päevalehe“ *H. Metsa* joonistatud karikatuur.



Joonis 6.3.4.c. Teksti *bender.txt* šifreerimine ja dešifreerimine.

Nagu nägime, on võtmeomanike krüpteeritud sõnumite vahetamise algoritmid vägagi lihtsad, ent koodi murdmise meetodid on üpris keerulised ning raskesti programmeeritavad (rääkimata käsitsitöö mahust enne arvutite appivõtmist). Seetõttu ei too me näiteprogramme[1], mille abil saaks koode murda (nad on pikad ja keerulised), murdmisalgoritmidest (ja kodeerimismeetoditest) soovitame lugeda formalismivaba ja arusaadavate näidetega *Greg Goebel*i materjali [Goebel].

[1] Siiski, järgmise peatüki jaotises „*F*- ja *C*-stiilid: realisatsioon“ toome näite naiivsest kodeerimisest ning selle koodi murdmisest.

7. Massiivid

7.1. Üldist

Kordame, et *vektor* on *ühemõõtmeline massiiv*[1] kirjeldusega *tüüp nimi*[*pikkus*], mille alternatiivne (võimalik, et sj. dünaamiline) kirjeldus on *tüüp *nimi*. Näiteks, staatiliselt *int a*[32]; või dünaamilisuse-võimalusega *int *b* (lahtiseks jääb, kas *b* on lihtmuutuja või teadmata pikkusega vektor).

Massiivid võivad (teoreetiliselt[2]) olla *n*-mõõtmelised. *C*-keel lubab näiteks kirjeldada massiive *int maatriks*[10] [20] – 10 rea ja 20 veeruga tabel, mille lahtrites on *int*-tüüpi arvud[3], kolme-mõõtmelisi massiive, näiteks *float*[3][4][5], mille elemendid on *float*-tüüpi jne.

Tsiteerigem [KjaK, lk. 68]: „Massiivid paigutatakse arvuti mällu nii, et kõige kiiremini muutub viimane indeks. Näiteks kirjeldus

```
int t[2][3]; //indeksite määramispiirkonnad on vastavalt 0..1 ja 0..2
```

määrab kaheelemendilise massiivi kolmeelemendilistest täisarvulistest massiividest, mille elemendid paiknevad arvuti mälus (aadresside suurenemise järjekorras)[4]

```
t[0][0], t[0][1], t[0][2], t[1][0], t[1][1], t[1][2]“.
```

7.2. Yngve hüpotees

Tehkem üks kõrvalepõige (meenutades, et meie teema pole ainult *C*-keel, vaid ka programmeerijaid huvitada võivad andmestruktuurid), küsides, kus ja miks on kasulikud rohkem kui kahemõõtmelised massiivid. Meenutagem [PK, lk. 110], et *FORTRAN* võimaldas kasutada *kuni* kolmemõõtmelisi massiive, ja sedagi, et meil on raske välja mõelda ülesandeid, kus rohkemaid mõõtmeid vaja oleks. Siinkohal tundub kohasena meenutada *Victor Yngve*t [Yngve_1, Yngve_2] ja tema hüpoteesi: meie (so. inimesed) pole võimelised opereerima süsteemidega, mille hierarhiatasemete arv on üle kuue, võimaliku hälbega ± 1. Ja seega, ka massiivid mõõtmete arvuga üle kuue on semantiliselt kahtlased.

Joonis 7.2.a. *Victor Yngve*

[1] Ingl. k. *array*, vene k. массив.
[2] Tsiteerides [KjaK, lk. 68]: „mitmemõõtmelised massiivid kirjeldatakse kui massiivid massiividest. Massiivi mõõtmete arvul ei ole seejuures piirangut.“
[3] Elemendile viidatakse kui *maatriks*[*i*] [*j*].
[4] Sellist paigutust võiksime nimetada „reakaupa“ paigutuseks; *FORTRAN* kasutas seevastu „veerukaupa“ paigutust; meie näiteprogramm *arr.c* demonstreerib mõlemat tehnikat; neid nimetatakse vahel *C* ja *F*[*ortran*i] stiiliks (või vastavalt *RC*- (*Row*, *Column*) ja *CR* (*Column, Row*)-järjekorraks, vt.[wArray])

V. Yngve, sünd. 5.07.20 on Chicago Ülikooli emeriitprofessor, eriala on kompuuterlingvistika. Ta on keele *COMIT* autor (võrrelda võib seda keeltega *SNOBOL* või *Perl*) ja ta oli 1957 – 1965 *MIT* (*Massachusetts Institute of Technology*) kaastööline.

7.3. Mitmemõõtmelise massiivi kujutamine vektorina

Naaskem [KjaK]-näite juurde: ilmselt on massiiv *t*[2][3] kujutatud mälus järgmise vektorina *v*:

0,0	0,1	0,2	1,0	1,1	1,2
0	1	2	3	4	5

„lahtrites“ on indeksid [*i*][*j*] ning (kui kujutame seda kahemõõtmelist massiivi tõepoolest vektorina *v*), siis „pildi“ all on massiivi elementide vektori-indeksid *i*
so. *t*[0][2] = *v*[2] ja *t*[1][1] = *v*[4]
ning elemendi tegelik mäluaadress = *v*+(*i* × 4), kuivõrd massiivi tüüp on *int* (4-baidise elemendiga).

Vektorina kujutatud mitmemõõtmelise massiivi indeksite jadade teisendamiseks vektori indeksiks on vana probleem, millel on vähemalt kaks dokumenteeritud ja teadmata arv dokumenteerimata lahendusi; *Donald Knuth* [Knuth I, lk. 561] kirjutab: „Arvutite algusaegadest peale on nutikad programmeerijad välja mõelnud palju erinevaid mitmemõõtmeliste massiivide adresseerimise ja läbimise meetodeid, ja nii sündis publitseerimata arvutifolkloori[1] veel üks osa“ ning mainib, et esimesed selleteemalised ülevaateartiklid ilmusid erialaajakirjades 1962. aastal.

Interneti andmetel [wArray] on massiivi *t* indeksite loetelu (*i, j,* ...) teisendamiseks vektori indeksiks *v*[*I*] kaks aktsepteeritud versiooni: see, mida kasutati *FORTRAN*i kompilaatori kirjutamisel (veerupõhine *F*-stiil) ja see, mida järgib *C* (reapõhine *C*-stiil)[2].

Selgitame nende (muide, võrdväärsete, kui pöördume massiivi elementide poole *eranditult* indeksite abil) variantide erinevust: olgu meil 3×3-maatriks *m*[3][3] ja elementide väärtused 0..8. Pilt on järgmine:

	0	1	2
0	**0**	**1**	**2**
1	**3**	**4**	**5**
2	**6**	**7**	**8**

Nagu näeme, on *m*[0][0] väärtus 0 ja *m*[2][2] väärtus 8. *F*-stiilis kujundatud vektor *v* on järgmine:

[1] Kõrvalepõikena, selline *folkloor* on programmeerijate vennaskonnas alati olnud ja usutavasti ka jääb, teemadeks on suvaliste süsteemide dokumenteerimata või segaselt või vigaselt dokumenteeritud osad, mille kohta levib asjatundjate hulgas teave, kuidas koodi nii kirjutada, et asi toimiks. Üks lihtsamaid valdkondi on *C*-kompilaatori veateated (milles asi? mida teha?).
[2] Samas, vaevalt, et *C* siin teerajaja oli, *C* tehti aastatel 1969 – 73 [Ritchie, lk.1], ent reapõhine teisendus on kirjeldatud juba *Knuth*i raamatus [Knuth I, lk. 371 – 372], mille originaal trükiti 1968. aastal.

0 3 6 1 4 7 2 5 8

I = 0 1 2 3 4 5 6 7 8

Näeme, et *v*[0]..*v*[2] on *m* esimene, *v*[3]..*v*[5] teine ja *v*[6]..*v*[8] on *m* kolmas veerg. *C*-stiili järgides saame sellise vektori:

0 1 2 3 4 5 6 7 8

I = 0 1 2 3 4 5 6 7 8

Vektori kolmel esimesel väljal on maatriksi esimene, järgmisel kolmel teine ning viimasel kolmel – kolmas rida[1].

Meenutagem mõistet „keele virtuaalarvuti“ (vt. näit. [PK, lk. 76 jm]), mis tähistab üldiselt translaatori poolt loodud *keskkonda*, mida kasutab kasutajaprogrammi „jooksutav“ *.exe*-fail (kui tegu on kompileeritava keelega, ja *C* seda on). See keskkond kujutab endast programmiobjektide kirjelduste tabeleid (andmekirjelduse info + muu vajalik) ning seda võib kasutada kompilaator ise (*C*-keele puhul näiteks lihtmuutujate ja vektorite töötlemiseks), ent vajadusel kirjutatakse need tabelid (*deskriptorid*) täidetavasse *.exe*-faili, kasutamaks neid täitmise ajal (*runtime*). Tänu sellisele tehnikale saab *C*-virtuaalarvuti kiiresti hakkama näiteks avaldisega *t*[2][7].

Korrakem, *C* on *väike* ja *staatiline* keel. „Väike“ selles mõttes, et keel pakub ainult minimaalse keeleliste vahendite komplekti[2], ja „staatiline“ selles mõttes, et keele tasemel saame opereerida ainult staatiliste (transleerimise ajal fikseeritud) objektidega, so. (näiteks) vektoriga *V*[17] ja mitte vektoriga *W*[*n*], kus *n* pole makroga *#define* määratud konstant, vaid lahendamise ajal arvutatud (või sisestatud) muutuja väärtus.

Pisut hiljem esitatame programmi *array.c*, mis on mõeldud olema näiteks

- dünaamiliste andmestruktuuride kasutamisele
- deskriptorile
- deskriptori kasutamisele
- *C*-mooduli muutuva pikkusega tegelike parameetrite loetelu käsitlemisele.

7.4. „Nähtamatud“ parameetrid. *Minprint*

Alustame viimasest teemast: „muutuva pikkusega tegelike parameetrite loetelu“, mooduli kirjelduses tähistatakse sellist *parameetrite loetelu* (näiteks) nii:

```
int printf(const char *format, ...);
```

[1] Maatriksite *A* ja *B* korrutamine (mis eeldab, et *A* veergude arv on võrdne *B* ridade arvuga) on näide sellest, et programmeerimise hõlbustamiseks on kasulik kujutada maatriksit *A C*- ja maatriksit *B FORTRAN*-stiilis, kuivõrd $C=A\times B \equiv c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + ...+ a_{in}b_{nj}$ (vt. näit. [matrix_product], sellele võimalusele vihjab [wArray]).

[2] Korrakem juba öeldut: *C* tuum on tõepoolest väike, ent keel on laiendatav suvalise võimsuseni funktsioonide teekide abil, millede vahenditega kaetakse sellised olulised valdkonnad nagu sisend/väljund, matemaatilised funktsioonid, töö stringidega, dünaamiline mälujaotus jne. Nood teegid *ei kuulu C*-keelde, ent nende loomisel tuleb tahestahtmata järgida *C*-stiili.

kus teise parameetri väärtuste võimalikke esitusmeetodeid ning nende arvugi märgitakse kõikelubavalt: „ ...“. Näiteks *printf* kasutamine, kus väljastatakse erinevat tüüpi väärtusi:
`printf(„asjad %s %d %p\n“,s,a,v);`
– siin vihjab eeldefineerimata väärtuste (mida *printf* väljastatab) arvule „%“-komponentide arv formaadiesituses. Ent on ka näiteid (vt. näit. *array.c*), kus selliseid ilmutatud kujul antud vihjeid pole.

Lahtise parameetrite loeteluga mooduli hea näide on [K&R, lk. 156] Toome selle järgnevas, lisades testimiseks vajaliku *main*-mooduli ja eestikeelseid kommentaare:

```
//minprint.c :: K&R lk. 155..156: minprintf 9.11.08
#include <stdarg.h>
#include <stdio.h>

//minprintf: minimal printf with variable argument list
void minprintf(char *fmt, ...){
    va_list ap;  //points to each unnamed arg in turn
    char *p,*sval;
    int ival;
    double dval;
    va_start(ap,fmt); //make ap point to 1st unnamed arg
   for(p=fmt; *p; p++){  //otsib stringist p sümbolit  '%', millele peab vastama parameetrite
//listi järjekordne element

       if(*p != '%'){
            putchar(*p);  //kuni ei leia, saadab sümbolid ekraanile
            continue;     // tagasi if-lausesse
            }
       switch(*++p){  //näiteks, kui oli %s, siis lüliti võti on 's'
           case 'd':
               ival=va_arg(ap,int);
               printf("%d",ival);  //siin ja tagapool: trükib printf( ) abil argumendi
               break;
           case 'f':
              dval=va_arg(ap,double);
               printf("%2.2f",dval);  // 2 kohta enne ja 2 pärast koma
               break;
           case 's':
               for(sval=va_arg(ap,char *); *sval; sval++)
                   putchar(*sval);  //string sümbolhaaval ekraanile
               break;
           default:
               putchar(*p);  //sümbol ekraanile
               break;
           }
       }
```

```
    va_end(ap);
}

int main( ){
    int a=99;
    double b=99.99;
    char *c="string";
    minprintf("start: %d %f %s\n",a,b,c);
    getchar( );
}
```

ning selle lahendamisel väljastatakse järgmine tekst:

```
start: 99 99.99 string
```

7.5. Argumentide määramata pikkusega loend: teek <stdarg.h>[1]

Määramata pikkusega ilma nimedeta ja deklareerimata tüüpi argumentide loendi töötlemiseks on *C*-keeles vahendid[2] olemas, ent paraku on nende realisatsioon erinevat tüüpi masinate ja platvormide jaoks erinevate kompilaatorikirjutajate jaoks standardiseerimata. Need vahendid (funktsioonid ja makrod) on tavaliselt kirjeldatud teegis *stdarg.h* (nagu *ANSI* standardit järgivate *gcc* või *Dev-C++* puhul, ent selleks võib olla ka näiteks teek *sys/varargs.h*) ning sõltumata realisatsiooni eripäradest on nendeks neli funktsiooni[3]:

```
void va_start(va_list pvar, name);
(type *)va_arg(va_list pvar, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list pvar);
```

Kommenteerime eelmise jaotise lõpus esitatud näidet *minprint.c*.

va_list ap; defineerib muutuja *ap* (*argument pointer*, viit argumendile), mis viitab järjekorras iga (järjekordse) „nähtamatu“ argumendi *väärtusele.*

va_start(*ap*,*fmt*); algatab argumentide loendi töötluse: *ap* väärtuseks saab viit esimesele argumendile ning seeläbi saabki tolle argumentide jada kasutamine võimalikuks. Ning teine parameeter (meil *fmt*) on *viimase* (parempoolseima) parameetri *nimi*. See ei kuulu argumentide loetellu[4]. Meie *minprint*-näites on **fmt* väärtuseks string "*start: %d %f %s\n*".

Ritchie näites on oodatav argumentide arv „sisse kirjutatud“ parameetrit **fmt* esindavasse stringi: ootame neid nii palju, kui tolles stringis on sümboleid ’%’[5].

---

[1] Osas 3.5.6 nentisime selle teegi olemasolu ja lükkasime selle kirjeldamise käesolevasse jaotisse.
[2] Tugineme [K&R, lk. 155–156 ning 254], [Sinivee-fp] ja *UNIX*-keskkonnas päringuga *man va_arg* saadule.
[3] T. Kelder osutab, et „mitte-*ANSI-C*“ kasutab lisaks vahendeid *va_alist* ja *va_del*, ja seal on teegi nimeks <*varargs.h*> [KjaK-2, lk.8].
[4] Juhul, kui sisuliselt pole moodulil muid parameetreid kui „määramatu pikkusega nähtamatu loetelu“, siis tuleb ainsa parameetrina anda selle loetelu sisuliselt esimese elemendi „nimi“ ning *va_arg*i abil loetakse loendi liikmeid alates teisest.
[5] Kui loendit kasutav moodul „teab“ argumentide arvu, siis initsialiseeritakse *ap* suvalise *parempoolseima* argumendi nime abil. Me kasutame seda programmis *arr.c*, jaotises 7.7.

Kolmas võimalus (mida nende ridade autor küll näinud pole), on argumentide loendi lõpetamine erimarkeriga, umbes samuti nagu '\0' toimib stringi puhul; näiteks neljabaidise markeriga *NULL*.

Funktsioon *va_arg* on ilmselt midagi rohkemat kui lihtsalt stringiviida nihutaja – kui ette kujutada, et argumentide loend on sümbolkujul. Ta teeb seda *ka*, ent teisendab (vajadusel) viidatud argumendi sisekujule – selleks annab võimaluse teine parameeter *tüüp*. Ja *va_arg* nihutab ikkagi ka *ap*-viida järgmisele argumendile (tuginedes *tüübile*).

Kuivõrd *va_copy*t *Ritchie* näide ei kasuta, ja me ei näe läbi ka ta otstarvet, siis hoidugem siinkohal kommentaaridest.

Funktsioon *va_end* kustutab kõik *va_start*iga ja argumentide loendi töötlemisega seotud andmed ja -struktuurid (kui neid loodi) ning kogu aparatuur on valmis järgmiseks kasutamiseks mõnes järgnevas pöördumises. *Veiko Sinivee* [Sinivee] hoiatab: „Kui te seda (*va_end*-makro[1] väljakutset, meie märkus) ei tee, siis võib programmi täitmise käigus tekkida hulgaliselt seletamatuid vigu. Translaator selle makro puudumist ei märka[2].“

7.6. F- ja C-stiilid: realisatsioon

Naaskem osas 7.3 mainitud *deskriptor*i mõiste juurde. Olgu massiivi mõõtmete arvu *n*. Triviaalsel juhul, kui $n=1$, on tegemist vektoriga ja indeksid on üheselt mõistetavad ja täitmise ajal pole deskriptorit vaja. Aga kui $n>1$, siis on otstarbekas toda deskriptorit kasutada, leidmaks võimalikult „odavalt“ massiivi indeksite väärtustega osundatud elemendi vektori-indeksi. Juba viidatud osas näitasime kaht head võimalust, kuidas kahemõõtmelise massiivi elemente kirjutada arvutatud vektoriindeksitega määratud väljadele, kasutades massiivi *m*:

```
int m[3][3];
```

Selle massiivi elementide arv on $3\times3=9$ ning indeksid i_0 ja i_1 muutuvad mõlemad piirides 0..2. ja deskriptor võiks olla järgmise kirjeldusega (ning näitekohaste väärtustega):

```
struct descr{
        int *a;     // viit vektorile v[9]
        int dima;   // dimensioonide arv=2
        int *dim;   // rajade vektor pikkusega dima: dim[0]=3, dim[1]=3
        int *DM;    // abivektor pikkusega dima – selle täitmist vaatleme allpool
        };
```

Niisiis, *abivektor DM*. Selle elementide arvutamise algoritm sõltub *stiilist* (*FORTRAN* või *C*), mida tuleb järgida *n*-mõõtmelise massiivi projektsioonil vektoriks[3].

*FORTRAN*i stiil: $DM[0]=1$ ning $DM[i]=dim[i-1]\times DM[i-1]$ $(1\le i<n)$; meie näite jaoks $DM[0]=1$, $DM[1]=3$.

[1] Mõnede *C*-st-kirjutajate arvates on *va_*-vahendid pigem makrod kui tavamõttes funktsioonid.
[2] Märkigem, et selline ongi *C*-stiil. Keel tehti professionaalidele, kes teavad, mida teevad, ja keda pole vaja kontrollida.
[3] Vt. näit. [Lebedev], lk.40..42.

C-stiil: *DM*[*n–1*]=1 ning *DM*[*i*]=*dim*[*i*+1]×*DM*[*i*+1] (*n*–2 > *i* ≥0); meie näite jaoks *DM*[0]=3, *DM*[1]=1

Massiivi indeksite loetelust (meil i_0 ja i_1) vektori indeksi *I* arvutamine ei sõltu enam stiilist:

$I = i_0 \times DM[0] + i_1 \times DM[1]$. Üldiselt,
$I = \sum_{j=0}^{n-1} i_j \times DM_j$, kus massiivi mõõdete arv on *n* ning i_j on *j*-ndale mõõtmele vastava indeksi *i* väärtus. Toome selle jaotise lõpetuseks *Knuth*i näite [Knuth I, lk. 371..372]. Olgu meil nelja-mõõtmeline massiiv[1] *Q* [*I, J, K, L*], iga element on 1 sõna pikkune ja $0 \le I \le 2$, $0 \le J \le 4$, $0 \le K \le 10$, $0 \le L \le 2$. Ilmselt on see massiiv *C* jaoks *int Q*[3][5][11][3]. Selle massiivi *DM*-vektori (analoogi) arvutas *Knuth* välja: *LOC*(*Q*[*I,J,K,L*])=*LOC*(*Q*[0,0,0,0])+165*I*+33*J*+3*K*+*L*. Meie *DM* elemendid *C*-versioonis on 165, 33, 3, 1. Ning *FORTRAN*i versioonis on nad 1, 3, 15, 165.

7.7. Programm arr.c

```
//arr.c : n-mõõtmeline dünaamiline int-massiiv. 12.11.08
//'F' : Fortran-stiil ("veerupõhine") ja 'C' : C-stiil ("reapõhine")
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

/* massivi kirjeldus */
struct descr{
        int *a;
        int dima;
        int *dim;
        int *DM;
        };

// massiivi kirjeldamine, mälueraldus. n=dimensioonide arv, edasi rajad.

struct descr *array(char style, int n, ...){
        va_list ap;
        struct descr *D;
        int b,i;
        D=(struct descr *)malloc(sizeof (struct descr));  //mälueraldus deskriptorile
        memset(D,'\0',sizeof(struct descr));  //deskriptori nullimine
        va_start(ap,n);  /* initsialiseerib argumentide listi */
        D->dima=n;
        D->dim=(int *)malloc(n*sizeof(int));  //mälueraldused vektoritele
        D->DM=(int *)malloc(n*sizeof(int));
```

[1] *Knuth* järgib süntaktiliselt ilmselt *FORTRAN*i-stiili ning *C*-stiili reakaupa-paigutust.

```
        for(i=0;i<n;i++) D->dim[i]=va_arg(ap,int);  //rajade salvestamine
        va_end(ap);  //ap desaktiveerimine
        printf("rajad: ");
        for(i=0;i<n;i++) printf("%d ",D->dim[i]); printf("\n");  //demo!
        switch(style){
                case 'F':
                        D->DM[0]=1;
                         for(i=1;i<n;i++) D->DM[i]=D->DM[i-1]*D->dim[i-1];
                        break;
                case 'C':
                        D->DM[n-1]=1;
                         for(i=n-2;i>=0;i--) D->DM[i]=D->DM[i+1]*D->dim[i+1];
                        break;
                }
        printf("DM: ");
        for(i=0;i<n;i++) printf("%d ",D->DM[i]);  //demo!
        //vektori pikkuse arvutamine ja mälu reserveerimine
        b=1;  //b:=vektori elementide arv
        for(i=0;i<n;i++) b=b*D->dim[i];
        D->a=(int *)malloc(b*sizeof(int));
        memset(D->a,'\0',b*sizeof(int));   //vektori nullimine
        printf("\n");   //reavahetus
        return(D);
}

/* massiivi  elemendi lugemine: nähtamatud argumendid on indeksite loetelu
   NB! indeks[i]=0..dim[i]-1. Indeksite oodatava arvu[1] n saame deskriptorist D.  */

int val(struct descr *D, ...){
        va_list ap;
        int n,i,x;
        va_start(ap,D);
        x=0;
        n=D->dima;
        for(i=0;i<n;i++) x+=D->DM[i]*(va_arg(ap,int));  //vektoriindeksi arvutamine
        va_end(ap);
         return(D->a[x]);
}

/* massiivi D kirjutamine: v on väärtus */
void wri(int v,struct descr *D, ...){
```

[1] Kui indekseid antakse oodatust vähem, on resultaat ilmselgelt etteaimamatu. Aga *C* stiil ongi selline – seda-tüüpi vigu ta ei püüagi avastada ega vältida. Kogu vastutus lasub programmeerijal, kes eeldatavasti on profes-sionaal. Meenutagem *Tomba*ku paradoksi: „mida targem programmeerija, seda lollemaid vigu tohib ta teha“ (*Mati Tombak*, suusõnaline teade).

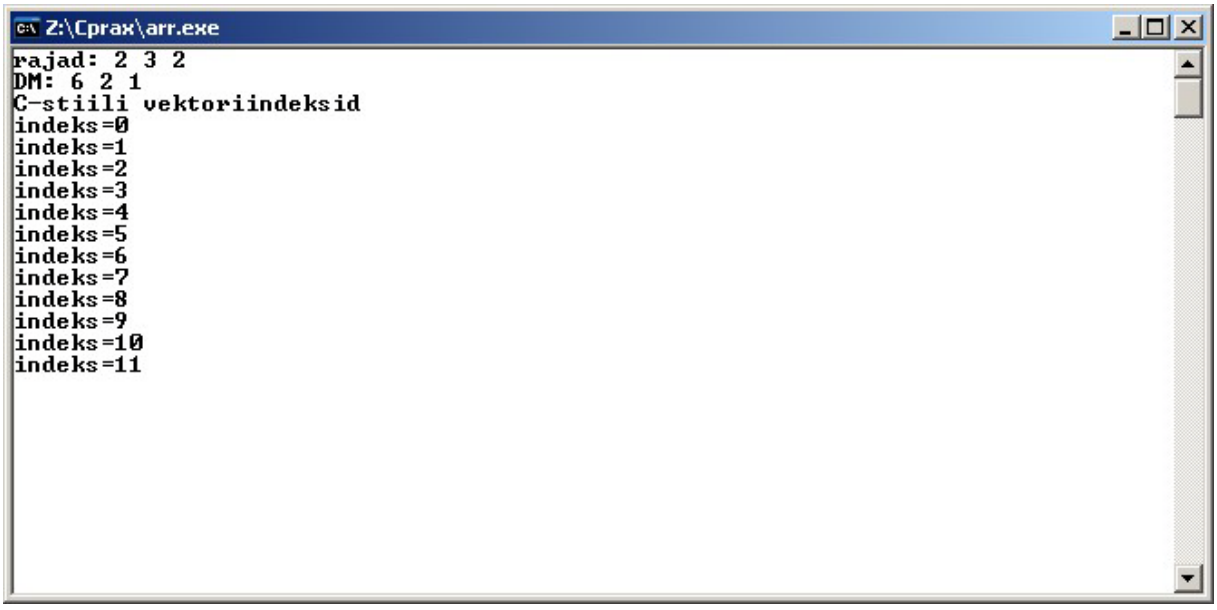
```
	va_list ap;
	int i,n,x=0;
	va_start(ap,D);
	n=D->dima;
	for(i=0;i<n;i++) x+=D->DM[i]*(va_arg(ap,int));  //vektoriindeksi arvutamine
	va_end(ap);
	printf("indeks=%d\n",x);  //demo
	D->a[x]=v;
}

//trükib massivi-indeksite loetelust arvutatud vektoriindeksi: demo
void inx(struct descr *D, ...){
	va_list ap;
	int i,n,x=0;
	va_start(ap,D);
	n=D->dima;
	for(i=0;i<n;i++) x+=D->DM[i]*(va_arg(ap,int));  //vektoriindeksi arvutamine
	va_end(ap);
	printf("indeks=%d\n",x);
}

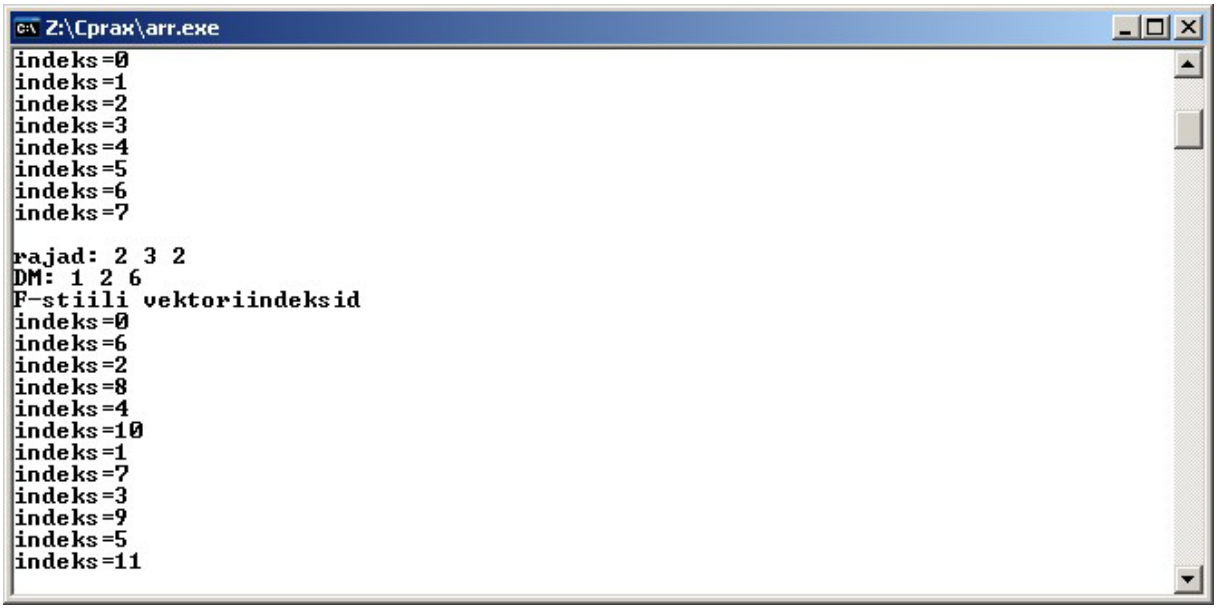
//main-moodul tegeleb peamiselt testimistega („demo“)
int main( ){
	struct descr *A;
	int i,j,k;
	A=array('C',3,2,3,2);
	printf("C-stiili vektoriindeksid\n");
	for(i=0;i<A->dim[0];i++){
		for(j=0;j<A->dim[1];j++){
			for(k=0;k<A->dim[2];k++){
				inx(A,i,j,k);
				}
			}
		}
	getchar( );   //ekraanipilt läheb edasi pärast mõne klahvi vajutamist
	A=array('C',2,2,4);
	printf("C-stiili vektoriindeksid\n");
	for(i=0;i<A->dim[0];i++){
		for(j=0;j<A->dim[1];j++){
				inx(A,i,j);
				}
			}
	getchar( );
	A=array('F',3,2,3,2);
	printf("F-stiili vektoriindeksid\n");
```

```
        for(i=0;i<A->dim[0];i++){
                for(j=0;j<A->dim[1];j++){
                        for(k=0;k<A->dim[2];k++){
                                inx(A,i,j,k);
                                }
                        }
                }
        getchar( );
        A=array('F',2,2,4);
        printf("F-stiili vektoriindeksid\n");
        for(i=0;i<A->dim[0];i++){
                for(j=0;j<A->dim[1];j++){
                                inx(A,i,j);
                                }
                        }
        getchar( );
        //Knuth, I, lk. 371..372 Siin trükib ainult DM-vektoreid.
        A=array('C',4,3,5,11,3);
        A=array('F',4,3,5,11,3);
         getchar( );
}
```

Allpool toome kaks arr.exe (valikulist) lahendamispilti.



Joonis 7.7.a. *C*-stiil.



Joonis 7.7.b. *F*-stiil.

7.8. Veel salakirjast

Nagu me kõik teame, leidub täiesti originaalseid asju vähe, iga uus on kas hästi äraunustatud vana või siis miski, mille analooge on varemgi teatud. Nii on ka massiivide esitusviisidega, nende analoogi on kasutatud ammusest ajast salakirjanduses [Goebel]: salastamist vajav tekst kirjutatakse suurtähtedega, ilma kirjavahemärkide ning sõnavahedeta ning paigutatakse „vasakult paremale, ülalt alla“ risttabelisse (see kirjutamisviis järgib *C*-stiili). Näiteks võime juba ülalpool kasutatud *Ostap Bender*i telegrammi kirjutada järgmise tabeli kujul (täitesümbolina kasutatakse nulli, nagu siin viimases lahtris):

```
LAADIGEAPEL
SINIDTYNNID
ESSEVENNADK
ARAMAZOVID0
```

Šifreerimiseks kasutatakse *F*-stiili (esimesena muutub veeruindeks) ning sõnum kodeeritakse „ülalt alla ja vasakult paremale“ stringiks

```
LSEAAISR...LDK0
```

Sellise koodi murdmine on triviaalne. Kui murdjatel on piisavalt statistilist materjali saadetud salakirjade näol, siis tähtede esinemissageduste statistilise analüüsiga tehakse kindlaks nii sõnumi keel kui ka šifreerimisviis, käesoleval juhul inglise keeles *simple transpositions*[1], ning lugemisel alustatakse oletusest, et tegemist on kaherealise tabeliga, kui see midagi mõistlikku ei anna, siis kolmerealisega jne – kuni „veergepidi“ loetud tekst on semantiliselt aktsepteeritav (ja seega dešifreeritud). Seda krüptimismeetodit illustreerib järgmine programm:

[1] Lihtne transponeerimine.

```
//transp.c :: simple transpositions  06.09.09
#include <stdio.h>
#include <stdlib.h>

    char kiri[80], sala[80], des[80];
    int i,j,k,m,n;
    div_t J;

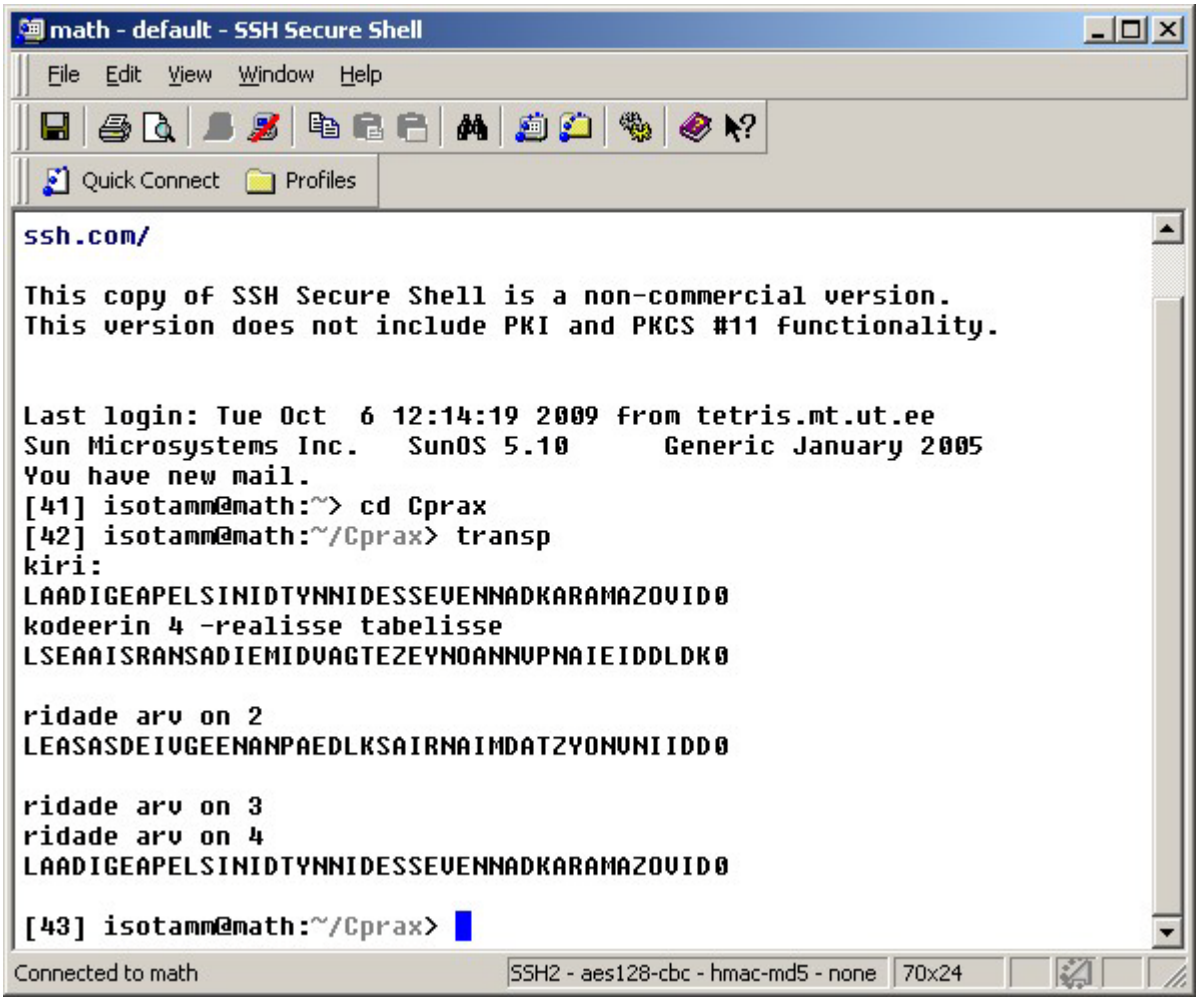
    void kooder(int ra){
        printf("kodeerin %d -realisse tabelisse\n",ra);
        J=div(n,ra);
        m=J.quot;
        if(J.rem!=0){
          printf(„vale pikkus\n“);
          abort( );
          }
        memset(sala,'\0',80);
        k=0;
        for(i=0;i<m;i++){
           for(j=0;j<ra;j++){
              sala[k]=kiri[i+j*m];  //C-stiil --> F-stiil
              k++;
              }
           }
        puts(sala);
        getchar( );
    }

    int murdja(int ra){
        printf("ridade arv on %d\n",ra);
        J=div(n,ra);
        if(J.rem!=0) return(0);  //tabel peab olema „täis“
        m=J.quot;
        memset(des,'\0',80);
        k=0;
        for(i=0;i<m;i++){
           for(j=0;j<ra;j++){
              des[i+j*m]=sala[k];  //F-stiil --> C-stiil
              k++;
              }
           }
       puts(des);
       getchar( );
       return(1);
    }

int main( ){
    int ra;
    printf("kiri:\n");
    gets(kiri);
```

```
    n=strlen(kiri);
    kooder(4);
    for(ra=2;ra<5;ra++) murdja(ra);
}
```

Ning selle lahendamiskäiku illustreerib järgmine slaid (joonis 7.8.a):



Joonis 7.8.a. Naiivne kood ja selle murdmine.

Karmides sõjatingimustes osutus see „naiivne meetod“ ootamatult efektiivseks, kuivõrd niimoodi kodeeritud esimeste sõnumite eetrisse jõudmisel polnud „murdjatel“ mingit alust oletada, kas tegemist on koodiraamatute uue versiooniga, ja kui sõnum püüti kinni kas Saksamaalt või N. Liidust, siis kas on kasutatud *Enigma*t või *Vernam*i koodi uue võtmega, või sõnumisaatjad lihtsalt lollitavad püüdjaid. Meenutagem, et militaarside standard on sõnumite edastamine viiemärgiliste „gruppidena“ morsekoodis. Ning sõnumi dešifreerimiseks on vaja teada „rea pikkust“.

7.9. Vektoresitusest veel

7.9.1. Vektoresituse kontrollimine

Käesolevas lühikeses alapunktis näitame, et *C*-kompilaator kasutab tõepoolest massiivi kujutamiseks *vektorit* ning ei „valmista ette“ mingilgi moel viidastruktuure massiivi hierarhilistele alamstruktuuridele. Meile juba tuttav näide on järgmine: kirjeldatud on maatriks *int m*[3][3] ja elementide väärtused 0..8. Pilt on järgmine:

	0	1	2
0	**0**	**1**	**2**
1	**3**	**4**	**5**
2	**6**	**7**	**8**

Kirjutamaks *m* elementi „legaalselt“ üle – kusjuures pole kuidagigi läbi nähtav massiivi füüsiline kujutamine mälus – tuleb kasutada tsüklit tsüklis (näiteks liidame iga elemendi väärtusele ühe juurde):

```
for(i=0; i<3; i++){
        for(j=0; j<3; j++){
                m[i][j]=m[i][j]+1;  //seda saab teisiti ka teha, tehke!
                }
        }
```

Kui *C*-kompilaatorid ei loo mingeid salastruktuure (ja miks nad peaksidki?), siis loovad nad *n*-mõõtmelise massiivi jaoks *vektori*, ja kui see nii on, siis ülaltoodud omistamisi saab teha ka ühekordse tsükliga: kirjeldame viida (*int*–muutujale või *int*–vektorile)

```
int *v;
```

ning omistame ta algväärtuseks viida maatriksi *m* „vasaku ülanurga“ elemendile.

```
v=&m[0][0];
```

Omistamine (ja trükkimine) käib nii:

```
for(i=0;i<9;i++){
     v[i]=i+1;
     printf("%d ",v[i]);
     }
```

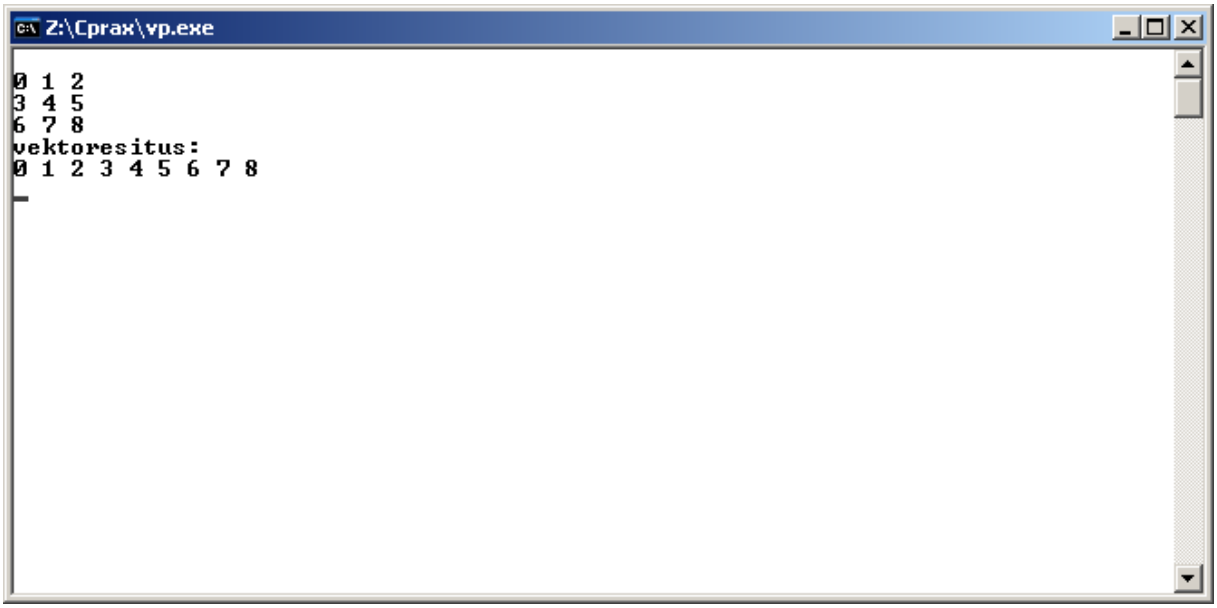
Lugejad võivad seda kontrolli korrata – tõepoolest, maatriks on mälus kujutatud vektorina *v*[9] elementidega 1..9. Üksiti peaks see veenma lugejat, et tegemist pole *viidastruktuuriga*, kuivõrd lihttsükkel 0..8 saab kätte *vektori* kõik elemendid.

7.9.2. Vektoresituse kasutamine

Teades, et *C*-kompilaator kujutab mitmemõõtmelist massiivi mälus vektorina, võime mitmeid ülesandeid oluliselt lihtsustada. Näiteks, *n*-mõõtmelise massiivi elementide järjestamine nende väärtuste järgi, elementide väärtuste unikaalsuse kontroll, mingi väärtuse otsimine, *min*- ja *max*-väärtuste otsimine, kõikide elementide summa leidmine jne. Selle lähenemisviisi näitena kasutame järgmist programmi (lahendamist vt. joon. 7.9.2.a):

```
//vp.c : 2D-massiivi järjestamine elementide väärtuste järgi
#include <stdio.h>
#include <stdlib.h>

int main( ){
      int  m[3][3]={{7,3,8},{0,5,1},{4,2,6}};   /*maatriksi  m  deklareerimine
                  ja initsialiseerimine*/
      int *v;  //C-stiil: massiivi vektoresituse viit
      v=&m[0][0];  //viida v väärtustamine
      int i,j,k;
/*m[3][3] elementide järjestamine mullimeetodil[1]: kasutame vektorit v
            [0..8] */
      for(i=0;i<8;i++){
            for(j=i+1;j<9;j++){
                  if(v[i]>v[j]){
                        k=v[i];
                        v[i]=v[j];
                        v[j]=k;
                        }
                  }
            }
//näitame järjestamise toimimist m[3][3] peal
      for(i=0;i<3;i++){
            printf(„\n“);    //reakaupa trükk
            for(j=0;j<3;j++) printf("%d ",m[i][j]);
            }
      printf("\n");  //sama, vektori v trükk (m[i][j]::v[i])
      for(i=0;i<9;i++) printf("%d ",v[i]);
      printf("\n");
}
```



Joonis 7.9.2.a**.** Maatriksi *C*-esitus (esimesena muutub reaindeks).

[1] Harjutus: kirjutage programm, mis järjestab *n*-mõõtmelise (*n*=1, 2, 3, ..) massiivi *n* indekseid (*i,j*, ..) kasutades.

7.10. Massiivi kujutamine *Iliffe*'i vektoritega

7.10.1. Viitade vektorid

Vektoreid käsitledes tõime näiteid, kus vektori elemendid on *int*-tüüpi (tavaliselt sedatüüpi välja pikkus on 4 baiti) ja *char*-tüüpi – välja pikkusega 1 bait. Ent tegelikult võib vektor olla *mistahes defineeritud* tüüpi, näiteks, *short*, *double*, aga tüübiks võib olla ka viit mingit muud tüüpi objektile, näiteks vektorile või struktuurile. Sisuliselt: deklaratsiooni *char b*[20] väärtuseks on viit baitide vektorile (võimalik, et stringile) ja *char *c* väärtuseks on *viit* ühele baidile (või nende vektorile, kui nii omistatakse).

Muutuva reapikkusega kahemõõtmelise massiivi näitena toovad *Kernighan* ja *Ritchie* [K&R, lk. 113] ära sellise:

```
static[1] char *name[ ]={
        „Illegal month“,
        „January“,“February“,“March“,“April“,“May“;“June“,“July“,“August“,“September“,
        „October“,“November“,“December“
};
```

Tuginedes senistele teadmistele tõdegem, et *name* defineerib 13-väljalise[2] viitade vektori, ja iga viit viitab unifitseerimata pikkusega stringile.

Mingem edasi, vaadates, kuidas edastatakse käivitatavale *C*-programmile *käsurida*[3] (*Command Line*) näiteks *UNIX*-keskkonnas:

```
[43] isotamm@math:~/Cprax>spuu loomad
```

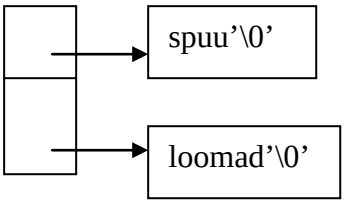
Välja kutsutakse programm *spuu* parameetriga *loomad.* Vastu võtmaks parameetreid, tuleb *C*-programmi „*main*“-moodul vormistada nii[4]:

```
int main(int argc, char *argv[ ][5]){ ... }
```

C virtuaalarvuti (vt. näiteks [PK, lk. 76 jm.]) edastab käivitatud programmile *argc* väärtusena käsurea komponentide arvu; meie näites on see 2: esimene on programmi nimi (ülalpool, *spuu*) ja teisest alates tekstid, mida interpreteeritakse parameetrite väärtustena, ülaltoodud näites on tolleks väärtuseks sõna „loomad“. Niisiis, argumentide loetelu kujutatakse kahemõõtmelise massiivina *argv*, mille ridade arvu määrab *argc* (selle tuvastab virtuaalarvuti) ning iga rida on (lõputunnusega) string. (igas reas on viit argumendile). Meie näitele vastab joonis 7.10.1.a.

---

[1] *static* tähendab, et nii märgistatud algväätustatud andmeid ega andmestruktuure ei saa lahendamise ajal muuta,
[2] 13 on looksulgude {..} vahel esitatud stringide arv.
[3] Vt. ka [K&R, lk. 114].
[4] Argumentidele võib panna suvalisi nimesid, näiteks *int main*(*int pa, char *pl*[ ]). Siiski, programm on loetavam, kui kasutame standardnimesid.
[5] Võrdväärne variant on *char **argv* – ent ainult selles kontekstis, kus kahetasemeline andmestruktuur („massiiv“) *argv* on juba loodud.



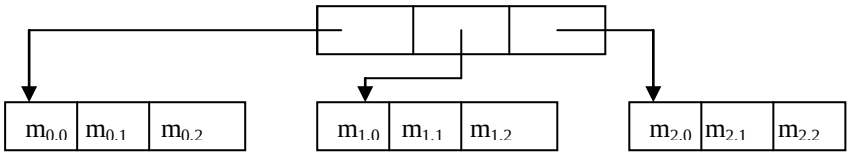
Joonis 7.10.1.a. Käsurea „*spuu loomad*“ täitmise ajal transleeritud andmestruktuur.

Niisiis, *argv*[0] ja *argv*[1] väärtused on viidad stringidele „*spuu*“ ja „*loomad*“.

7.10.2. Viidad: kahemõõtmeline massiiv

Meenutagem, et pesamasinate ajastul (vt. näit. [Isotamm, PK], 2. peatükk) olid viidad igati tavalised, normaalsed (tegelikult – vältimatud) andmetüübid, kuni programmeeriti masinkoodis või assembleris. Mõistagi kasutati dünaamilist mälujaotust, kirjutades vastavad moodulid ise või kasutades (hiljem) masinorienteeritud operatsioonisüsteeme, nt. *IBM OS*is makrosid *GETMAIN* ja *FREEMAIN*. Esimeste kõrgtaseme-keelte ilmumine muutis asja, põhjused olid erinevad. *FORTRAN*il polnud dünaamilist mälujaotust: ta oli ja on orienteeritud teadusarvutustele, kus on vähe ja lihtsaid (andmestruktuuride mõttes) andmeid – lihtmuutujad või konstandid ja kuni 3-mõõtmelised homogeensed arvulised massiivid, ja *viitu* polnud *sisuliselt* vaja. *Algol*-60 disainiti publitseerimaks (eeskätt *FORTRAN*is kirjutatud) programmide algoritme ja ehkki algoritmi üldisuse huvides oli lubada dünaamilisi massiive, ei võimaldanud ta „programmeerijal“ (loe: *Algol*-keelse algoritmi publikatsiooni kirjutajal) tekitada viidastruktuure (ahelad, puud jne. sj. ka välisaheldusega paisktabeleid) ning nii kadusidki viidad teaduspublikatsioonidest ja koos sellega enamikust esimestest protseduurorienteeritud keeltest. Ent süsteemprogrammeerijad jätkasid süsteemse tarkvara kirjutamist masinorienteeritud keeltes ja loomulikult jätkasid ka tööd viitadega (kuni nad, so. süsteemprogrammeerijad ise) tegid oma vajadustele vastavad keeled – *Chuck Moore* keele *Forth* (vt. näit. [Isotamm, PK], lk. 139 jj.) ning *D. Ritchie* keele *C* – kui mainida üldtuntud keeli.

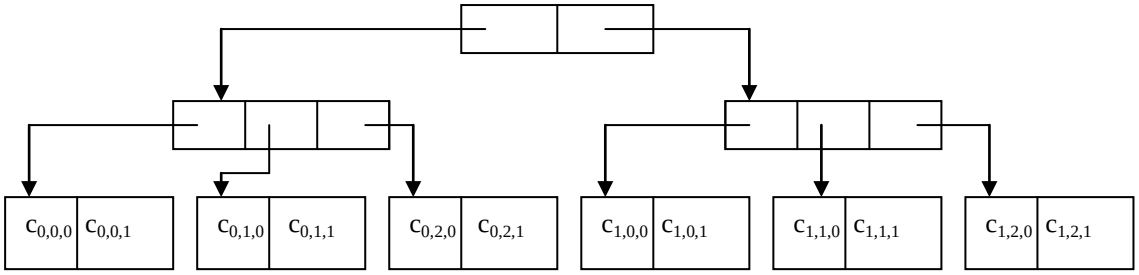
Eelnev lõik peaks juhatama sisse *n*-mõõtmelise massiivi kujutamise ja adresseerimise viitu kasutava meetodi; dokumenteeritult oli selle autoriks *John K. Iliffe* (vt. [Iliffe_1], [Iliffe_2]), aastal 1961. Tema „vektorite süsteemi“ loomise põhjus peitus tõigas, et kaasajal oli kaks olulist kitsaskohta: masinaaeg ja operatiivmälu maht. Eelmises jaotises käsitlesime varianti, kus massiiv kujutatakse mälus vektorina ja indekseerimiseks kasutatakse *korrutamistehet*, ent teatavasti on see (ajaliselt) „kallis“ tehe. Ja ehkki ka mälu oli defitsiitne, tekkis – kui see defitsiit pole kriitiline – *Iliffe*'i varianti kasutades võimalus oluliseks ajavõiduks[1].



Joonis7.10.2.a.. Massiiv *m*[3][3];

[1] Dilemma oli selline: vektori-variant on mälusäästlik, ent korrutamiste tõttu aeglane, *Iliffe*'i variant on kiire, aga vajab üpris palju lisamälu.

Ülalpool kasutasime massiivi *int m*[3][3], lisaks vaatame massiivi *char c*[2][3][2][1]. *Iliffe*'i vektoreid kasutades kujutatakse neist esimest mälus nii nagu joonisel 7.10.2.a, teist aga nii, nagu joonisel 7.10.2.b.



Joonis 7.10.2.b. Massiiv *c*[2][3][2];

Kui me *C*-keeles kirjutame *int m*[3][3]; või *char c*[2][3][2]; siis kõik *C*-kompilaatorid kujutavad neid massiive *RC*-vektoritena[2] ja selleks, et massiive kujutada *Iliffe*'i vektoritega, tuleb seda ise programmeerida. Näiteks võime oma staatilise 3×3 *int*-maatriksi *m* saada mällu teha järgmise viidastruktuurina:

```
int *rida1=m[0];  //siin kasutame ära oma maatriksi int m[3][3], rida1 on viit m esimesele
//reale.
int *rida2=m[1];
int *rida3=m[2];

int rida1[3]={0,1,2};  //võime ka iga rea väärtustada veergude kaupa
int rida2[3]={3,4,5};
int rida3[3]={6,7,8};

int *m1[3]={rida1, rida2, rida3};
```

Viimane näiterida tähendab, et *m*1 on vektor, mille elementideks on kolm viita *int*-arvudele (või nende vektoritele, nagu see meil tegelikult on). Esmapilgul võib tunduda üllatav, et me ei pruugi ilmutatud kujul näidata, kui palju neid viitu on – võime kirjutada ka

```
int *m1[ ]={rida1, rida2, rida3};
```

ent nonde arvu määrab üheselt massiivi initsialiseerimine ({ ja } vahel).

Edasi, kui kirjeldame muutuja *int **mp*[3], siis selle väärtuseks võime omistada viida, mis omakorda viitab *int*-tüüpi muutujale või vektorile. Niisiis, võime omistada *mp*=*m*1 ning saame nii *m*1 kui ka *mp* elemente indekseerida just sama moodi nagu tegime oma maatriksiga *m*; näiteks

```
printf("%d %d\n",m1[0][1],mp[1][0]);
```

[1] Siin on see kirjeldus esitatud „sissejuhatavana", kolmemõõtmelisi massiive käsitleme pisut hiljem.
[2] Meenutame, et *RC* tähendab *row-column*-stiili: kiiremini muutub teine indeks.
[3] Samas, me ei saa „kahe tärniga" kirjeldatud kahemõõtmelist massiivi initsialiseerida kujul
int **m={{0,1,2},{3,4,5},{6,7,8}}; – eelnevalt peame vastava struktuuri looma, ja alles siis saame teda kasutada. Vahet pole, *kes* selle loob, vt. näiteks *main*-mooduli parameetritega varianti, kus *char *argv*[ ] ≡ ***argv*.

Allpool on toodud programmi *katsa.c* tekst, mis peaks hõlbustama viimaste näidete kontrollimist.

```
//katsa.c: pointers array
#include <stdio.h>
#include <stdlib.h>

int main( ){
        int i;
        int **mp;
        int *v;
        int m[3][3]={{0,1,2},{3,4,5},{6,7,8}};  //maatriksi kirjeldamine ja initsialiseerimine
        int *r1=m[0];  //m[0] on viit maatriksi m esimesele reale
        int *r2=m[1];
        int *r3=m[2];
        int *m1[ ]={r1,r2,r3};   //m1 esitab maatriksit m viidastruktuurina
        printf("%d\n",m1[1][1]);  //indeksid toimivad täpselt samuti nagu m puhulgi
        mp=m1;  //vt. kirjeldusi!
        printf("%d\n",mp[1][1]);  //indeksid „töötavad“ ka siin
        v=&m[0][0];  //näide, et m on mälus vektorina
        for(i=0;i<9;i++) printf("%d ",v[i]);
        getchar( );       //DOS-akna peetamiseks
}
```

Peatugem veel ühel seigal. Kui me kasutame *n*-mõõtmeliste massiivide ($n \geq 2$) kirjeldamiseks varianti, mis nõuab massiivi kujutamist vektorina (so, kirjutades näiteks `int m[3][3];`) tehakse alati vektor pikkusega $\prod_{i=0}^{n-1} \dim_i$, kus dim_i on massiivi *i*-nda taseme indeksite arv ja *n* on massiivi mõõdete arv; näiteks meie *m* puhul määratakse, et vektor koosneb üheksast (3×3) neljabaidisest elemendist. Ja samamoodi on „*Knuth*i massiivi“ (vt. osa 7.6) elementide arv 3×5×11×3=495. Mis tähendab, et maatriksis peavad kõik read olema ühepikkused (mis on ka ainumõeldav, kuivõrd rea- ja veeruindeksite abil arvutatakse „lahtri“ suhtaadress vektoris.

Aga juhul, kui me moodustame mitmemõõtmelise massiivi viidastruktuurina, see kitsendus enam ei kehti (äsjatoodud näite jaoks tähendab see, et iga rida võib olla unikaalse pikkusega). Järgmises jaotises toome paar asjakohast lihtsat – kahemõõtmeline massiiv seda ju on – näidet, viidates klassikutele, kelleks meie kontekstis on kahtlemata *Kernighan* ja *Ritchie*.

7.10.3. Viidad: kolmemõõtmeline massiiv

Joonisel 7.10.2.b on kujutatud kolmemõõtmeline massiiv `char c[2][3][2]`. Siin esitame näiteprogrammi kolmD.c teksti. See programm genereerib massiivi *c* kolmel moel: kõigepealt moodustatakse ta staatilise *C*-massiivina nimega *c*: kompilaator genereerib *RC*-esituse, so. kujutab massiivi elemente ühes vektoris – meie kirjutame viida sellele vektorile muutujasse *V*, teiseks „programmeeritakse ta sisse“ vektorhaaval, nimega *cppp*1 ja kolmandaks, moodustame dünaamiliselt massiivi *cppp*.

```
//kolmD.c : pointers and arrays.  18.11.08
#include <stdio.h>
#include <stdlib.h>

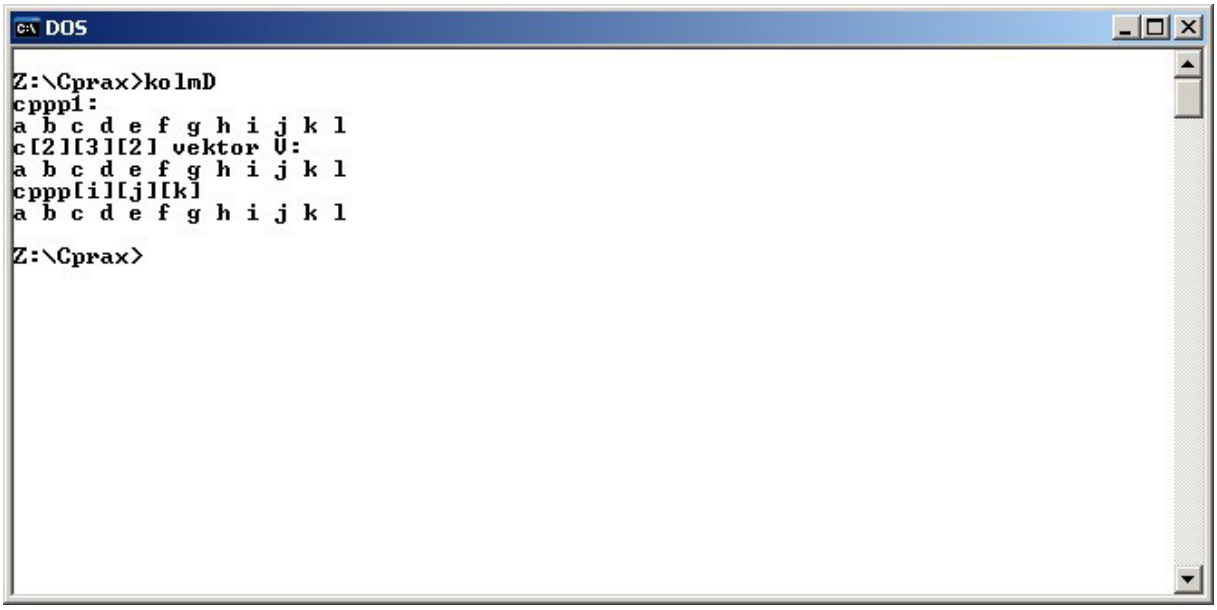
int main( ){
        int i,j,k;
        char C;   //c[i][j][k] väärtus: sümbol 'a', 'b' jne
        char *V;  //3D-massiivi vektoresitus
        char ***cppp,**cpp,*cp;  //3D-tasemete viitade vektorid
        char c[2][3][2];   //"legaalne" (staatiline) 3D-massiiv
// massiivi c "sisse kirjutamine" viidastruktuurina[1]
        char cp1[ ]={'a','b'};  //c[0][0]
        char cp2[ ]={'c','d'};  //c[0][1]
        char cp3[ ]={'e','f'};  //c[0][2]
        char cp4[ ]={'g','h'};  //c[1][0]
        char cp5[ ]={'i','j'};  //c[1][1]
        char cp6[ ]={'k','l'};  //c[1][2]
        char *cpp1[ ]={cp1,cp2,cp3};  //c[1]
        char *cpp2[ ]={cp4,cp5,cp6};  //c[2]
        char **cppp1[ ]={cpp1,cpp2};  //c
        printf(„cppp1:\n“);  //indeksid i, j ja k töötavad!
        for(i=0;i<2;i++){
                for(j=0;j<3;j++){
                        for(k=0;k<2;k++) printf("%c ",cppp1[i][j][k]);
                        }
                }
        printf(„\n“);
        V=&c[0][0][0];  //V on 3D-massiivi c esimese elemendi aadress
        C='a';  //C väärtuseks saab int 97  (kümnendsüsteemis)
 //vektorina V esitatud 3D-massiivi elementide väärtusteks saavad sümbolid 'a','b',...
        printf("c[2][3][2] vektor V:\n");
        for(i=0;i<12;i++){     //12=2×3×2
                V[i]=C;
                printf("%c ",V[i]);
                C++;
                }
        printf("\n");
// massiivi c[2][3][2]; ruumi dünaamiline reserveerimine
```

[1] Pangem tähele, et viidastruktuuri kirjelduse „parempoolseimat tärni“ asendab initsialiseerivates kirjeldustes sulupaar ’[ ]’. Seda, et mõlemad variandid kirjeldavad identset struktuuri, näitlikustab nt. legaalne omistamine *cppp=cppp1*. Ja et *x on sama, mis x[ ] – ainsa erinevusega, et ainult viimane variant võib olla *lvalue* rollis (so, *x={..} pole lubatud ja x[ ]={..} on lubatud). *Argumendi kirjeldusena* on nad võrdväärsed. Vt. *main*-mooduli parameetreid, kus *int argc* väärtus on tekstikujul esitatud argumentide loendi elementide arv ja sellele loendile saame *main*-mooduli argumentide hulgas viidata kas **argv*[ ] või ***argv*.

```
// cppp={cpp1, cpp2};
        cppp=(char ***)malloc(2*(sizeof (char **)));
        for(i=0;i<2;i++){
           cppp[i]=(char **)malloc(3*(sizeof (char *)));   //võtab mälu cppi-le, salvestab viidad
                for(j=0;j<3;j++){
                    cppp[i][j]=(char *)malloc(2*(sizeof(char *)));  //võtab mälu cpi-le, salvestab
                    }
                }

//dünaamilise 3D-massiivi initsialiseerimine c[2][3][2] väärtustega 'a',...,'l'
         for(i=0;i<2;i++){
        for(j=0;j<3;j++){
                 for(k=0;k<2;k++) cppp[i][j][k]=c[i][j][k];
                }
        }
        printf("cppp[i][j][k]\n");  //cppp indekseeritud väljatrükk
        for(i=0;i<2;i++){
                for(j=0;j<3;j++){
                        for(k=0;k<2;k++) printf("%c ",cppp[i][j][k]);
                        }
                }
        getchar( );
}
```

Joonisel 7.10.3.a on toodud programmi *kolmD.c* lahendamise väljatrükk.



Joonis 7.10.3.a. Programmi *kolmD.c* lahendustulemused.

7.10.4. Massiivid ja viidad

Viidad ja massiivid on *C*-keeles lahutamatult seotud. Teema lõpetuseks: *Thompson* ja *Ritchie* said endi käsutusse tolleaegse „unelmate masina“, mis toetas masinkoodi tasemel kõikvõimalikke manipulatsioone viitadega (vt. [Isotamm PK], lk. 128 jj.]) ning operatsioonisüsteemi ülekandmiseks mõeldud keel suutis uusi võimalusi hästi kasutada (kusjuures mitte kõiki, aparatuurne magasin – näiteks – *C*-autoreid ei huvitanud). Samas jääb mulje, et nende keeledisaini-alane kogemus ja eelistused tuginesid *Algol*-60 ideaalidele, kus teatavasti viidatüüpi ei olnud. Ning *Ritchie* ja *Thompson* tekitasid oma '*' ja '&'-sümboolikaga, aga ka mitme-tärnidefinitsioonidega paraja segaduse. Sellest annavad tunnistust pikad asjakohased seletused, [K&R] teevad seda lk. 93 –126, teisedki [O'Reilly], [Jensen] umbes samas mahus[1]. „Normaalset konstruktsiooni“ peaks olema võimalik defineerida ja näitlikustada (ülimalt) paari lehekülje tekstiga.

Siinkirjutaja leitud materjalidest tundub arusaadavaimana *Viktor Leppikson*i [Leppikson C, lk. 114 jj.] selgitus, et „deklaraatorisse lisatud tärnid ja nurksulud on vajalikud defineeritava muutuja tüübi täpsustamiseks“ ning et „üks ja sama deklaraator võib olla modifitseeritud niihästi tärnide kui ka nurksulgudega. Kuna tärni prioriteeti loetakse nurksulgude omast madalamaks, siis on lausega *double *pm*[5]; defineeritud massiiv (ja mitte viit) *pm*, (- - -), elementideks on *double*-tüüpi suurusi sisaldavatele mäluväljadele osutavad viidad.“

Lõpetagem lõbusa näitega. Tekst on pärit ajakirjast „Arvutustehnika ja andmetöötlus“, 2'94, lk. 12[2], tõlgitud artiklist „*Programmeerimiskeelte arengu tragöödia*“ [Gutknecht], kus on lõik „Lugeja meeleheaks sooviksin ma arutluse lõpetada tõeliselt huvitava ülesandega, mis C-keeles programmeeritult veel eriti huvipakkuvaks osutub. Ülesandeks on kirjutada võimalikult lühike reprodutseeruv (st iseennast väljastav) programm. Järgnevas on toodud reprodutseeruv üherealine *C*-programm, mis osutub lugejale parajaks pähkliks:

```
main(){char*c="main(){char*c=%c%s%c;printf(c,34,c,34,10);}%c";
print(c,34,c,34);}
```
//tsitaadi lõpp: “

Et *gcc* seda programmi aktsepteeriks, tuleb *main*-moodul teha *int*-funktsiooniks ja asja huvides parandada veel paar viga; meie jaoks tarvitamiskõlblik tekst on järgmine:

```
int main( ){char*c="int main( ){char*c=%c%s%c;printf(c,34,c,34,10);}%c";
      printf(c,34,c,34,10);}
```

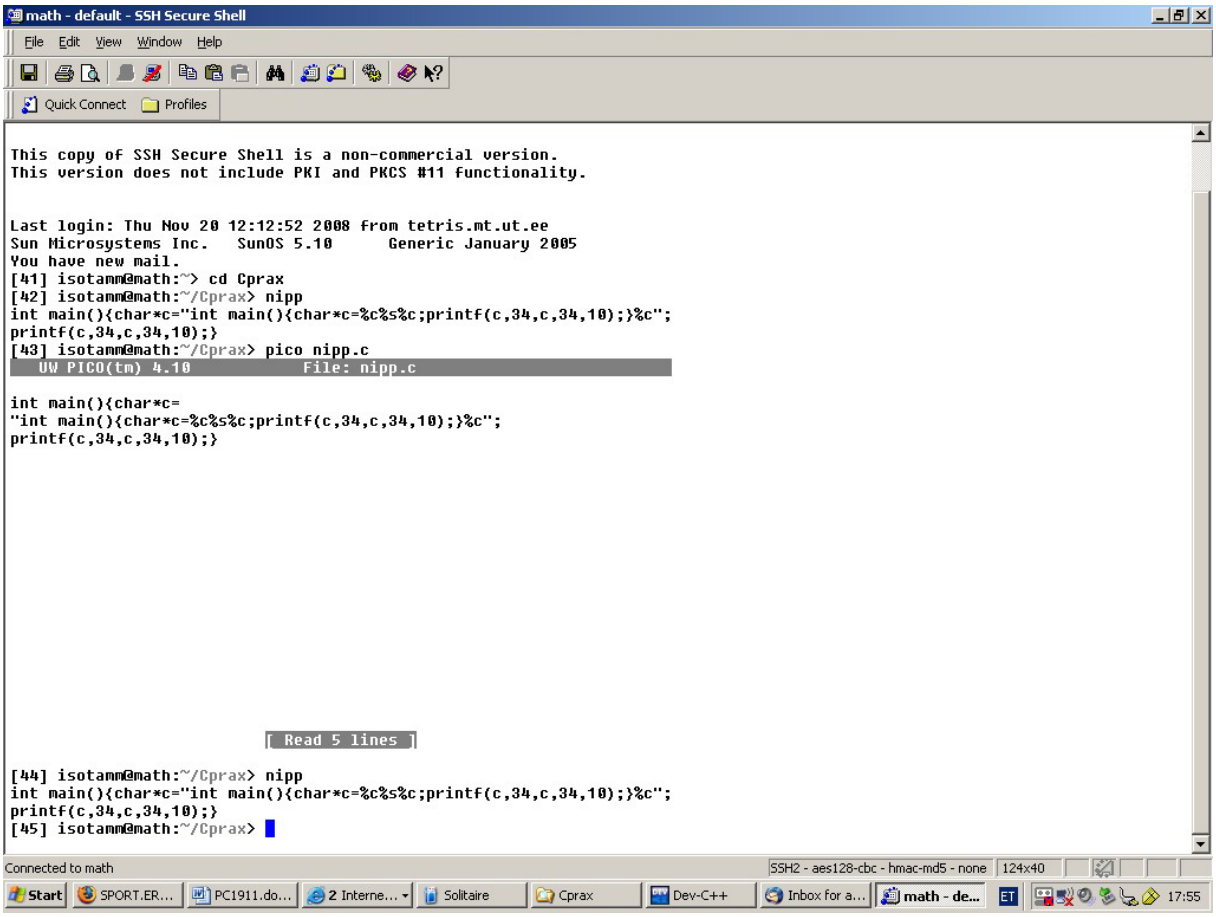
ning selle töötulemus on järgmisel joonisel (7.10.4.a).

Loodetavasti näeb *C*-keelt omandav programmeerija (aga need ongi meie sihtgrupp) selle programmi „nipi“ üsna ruttu läbi, aga – kinnitamiseks temasuguste enesekindlust ja aitamaks vähemkogenumaid – kommenteerime seda teksti. Esitame selle teksti liigendatult:

---

[1] Töö viitade ja viidatavate väärtustega on loomulikult ja seetõttu elegantselt lahendatud näiteks keeles *Forth* (vt. näit. [PK], lk. 139 .. 155).

[2] Tõlkinud *Anu Oja*, artiklist *Jürg Gutknecht, The Tragedy of Programming Language Development, Structured Programming, nr. 14, 1993, lk. 49–55* ja siintoodud programmi eest avaldatakse originaalis tänu *Clemens Szyperski*le. Märkigem, et *Jürg Gutknecht*i arvates seisnes *tragöödia C*-keele „distsiplineerimatuses“ ja helgem tulevik objektorienteeritud distsipliinis (konkreetselt, keeles *C++*).



Joonis 7.10.4.a. Reprodutseeruva programmi *nipp.c* tekst ja lahendamistulemus.

```
int main( ){
      char *c="int main( ){char*c=%c%s%c;printf(c,34,c,34,10);}%c";
      printf(c,34,c,34,10);
}
```

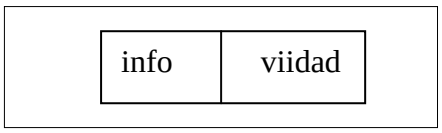
Programmi ainus operaator on *printf*. Meenutagem tema formaalset kirjeldust:

int printf(const char *format, ...)

Formaadi annab *string c*, kusjuures see „trükitakse“ välja nagu ta on esitatud ning teksti lisatakse järjekorras formaadis näidatud parameetrite loendi elementide (teisendatud) väärtused. Ja „34“ on sümboli ’“’ kood (’\“’) ning „10“ on reavahetuse kood (’\n’), *ASCII*s.

8. Ahelad

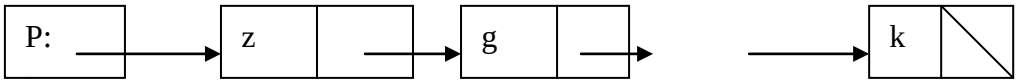
Ahel (ingl.k. *linked* list, varasemas kirjanduses ka *chain*, vene k. список связей, ka цепь)[1] on lihtne lineaarne viidastruktuur, „mille korral järjendi iga element on varustatud veel viidaga selle järjendi järgmisele[2] elemendile“ (vt. [Kaasik, PL], lk. 6). Ahel kui *andmestruktuur* koosneb samatüüpi *lülidest*; *lüli* koosneb loogilisel tasemel kahest osast: oma-info ja viit (või viidad) teis(t)ele lüli(de)le. Lihtahela lüli on põhimõtteliselt järgmine:



„Info“ võib paikneda suvalisel arvul järjestikustel väljadel, ent selleks võib olla ka viit lüliga seotud objektile. „Viidad“ on lüli formaadis esile tõstetud kui viit ahela lülile: *lihtahela* puhul on üks viit – viit järgmisele lülile ning *topeltseotud* ahela lülis on kaks viita: üks endiselt järgmisele ning teine – eelmisele lülile. Välja, kus paikneb viit ahele esimesele lülile, nim. ahela *peaks.* Ahela kirjeldamiseks piisab ta lüli formaadi kirjeldamisest, näiteks võib lihtahela lüli kirjeldus olla selline:

```
struct lüli_1{
        char label;
        struct lüli_1 *next;
        };
```

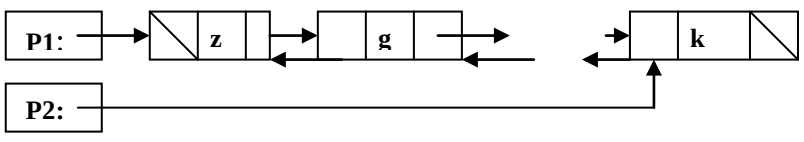
Sellele kirjeldusele vastava ahela pea võime deklareerida näiteks nii: `struct lüli_1 *P;` ning näite-ahel võib olla järgmine:



Topeltseotud ahela lüli võiks nii kirjeldada:

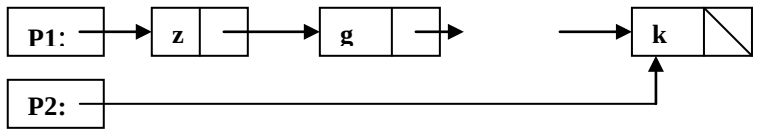
```
struct lüli_2{
        struct lüli_2 *prev;
        char label;
        struct lüli_2 *next;
        };
```

Lihtsustamaks töid ahelaga on otstarbekas kasutada topeltseotud ahela puhul kahte *pead*: üks viitab esimesele ja teine viimasele lülile (`struct lüli_2 *P1; struct lüli_2 *P2;`). Näiteid toome järgmises jaotises, kus käsitleme *magasini*, aga ka pisut hiljem, *tabelite* jaotises. Näitepilt:

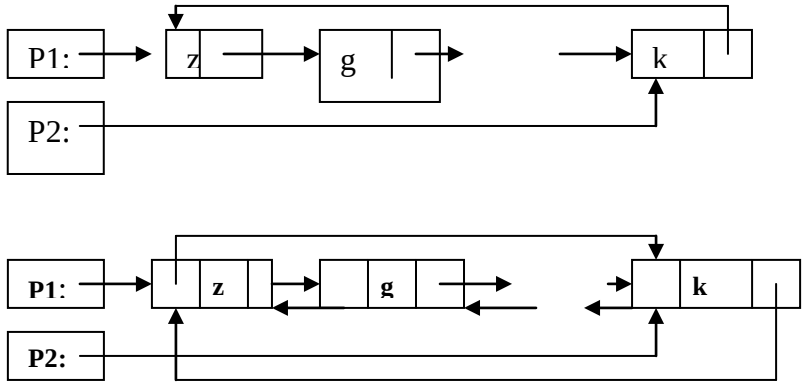


[1] Vt. näit. [MOS], lk. A38.
[2] Nii on *lihtahela* korral, teine võimalus on *topeltseotud ahel*, me käsitleme seda ka.

Mõistagi, kahte ahelapead võime kasutada ka lihtahela puhul (vt *järjekord* magasinide-jaotises):



Äsjatoodud piltidel kasutasime naaberlüli puudumisel tühiviita (*NULL*, pildil diagonaaliga „kast“). Ent kasutatavad on ka variandid, kus tühja parempoolse naabri viida kohale pannakse viit esimesele lülile, topeltseotud ahelas ka viit esimesest lülist viimasele. Andmekirjelduses see arusaadavalt ei kajastu; nii moodustuvad *ringahelad*:



Selmet kirjeldada ahela päid (*P*1 ja *P*2) eraldiseisvate muutujatena, võime kirjeldada ahela *deskriptori*, kus võib olla muudki kasutatavat informatsiooni, näiteks sellise:

```
struct listD{
        struct lüli_2 *P1;  //viit esimesele lülile
        struct lüli_2 *P2;  //viit viimasele lülile
        struct lüli_2 *Pc;  //viit aktiivsele (jooksvale) lülile
        int n;   //lülide arv : muutuja!
        int i;   //“jooksva lüli“ järjekorranumber (1..n)
}
```

Niimoodi saaksime opereerida paralleelselt paljude ahelatega väiksema riskiga eksida nende viitade kasutamisel. Ent „ilmutatud kujul“ ahelapäid ega ka deskriptorit pole alati tarviski luua – ahel võib olla mingi teise andmestruktuuri osis ning vajalikud viidad on siis juba selle „ülemstruktuuri“ komponendid. Ühe sellise andmestruktuuri näiteks sobib programmeerimiskeel *Lisp*, kus nii programm kui ka töödeldavad andmed on esitatud omavahel seotud lihtahelatena (vt. näit. [Isotamm, PK], lk. 174 jj.). Loomuldasa on ahel *dünaamiline* andmestruktuur[1], so., selline, mis luuakse ja millega manipuleeritakse (lisatakse või eemaldatakse lülisid) täitmise ajal (ingl. k. *runtime*)[2].

[1] Samas, ahel on kasutatav ka staatilise mälujaotusega keeltes. Vt. näit. [Isotamm, PK, lk. 118 „*FORTRAN* ja viidasüsteemid“]

[2] Seda võimaldab ka *Lisp*: ootuspäraselt töödeldavate andmete puhul, ent programmi täitmise ajal saab genereerida programmi lisateksti, mis pärast erifunktsiooniga transleerimist on interpreteeritav võrdväärselt esialgse programmiga, ning samuti täitmise ajal saab kustutada juba mittevajalikuks osutuvaid programmiahelaid.

Lihtahela kasutamise näiteks toome programmi, mis loeb klaviatuurilt täisarve ning moodustab neist dünaamilise[1] järjendi. Kui me annaks sellele moodulile arve ette varem moodustatud *vektorist*, siis võiksime seda programmi kasutada ka *pistemeetodil järjestamise* (*insertion sort*, сортировка вставками) näitena (tulemuste trükki vt. joonis 8.a).

```
//dynaamiline järjestamine "insertion sort".   09.01.09
#include <stdio.h>
#include <stdlib.h>

struct lyli{
      int key;
      struct lyli *next;
      };

struct lyli *P=NULL;    //ahela pea

int main( ){
   int a;                        //järjekordne võti
   char arv[10];                 //sisendpuhver
   struct lyli *New,*L,*Pr;  //pistetav, võrreldav, eelmine (Pr->New->L)

dai: printf("\nanna arv  ");
   gets(arv);
   if(strlen(arv)==0) goto ots;
   a=atoi(arv);   //sümbolid -> int
   New=(struct lyli *)malloc(sizeof(struct lyli));
   memset(New,'\0',sizeof(struct lyli));
   New->key=a;

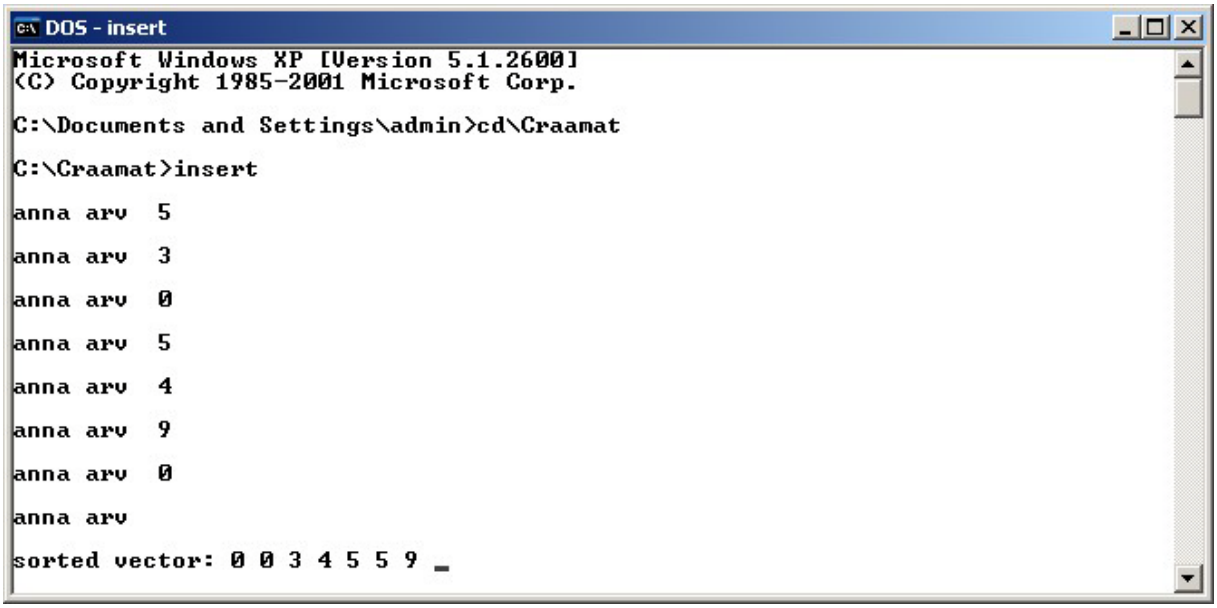
//otsin ahelast kohta.
   L=Pr=P;
   if(L==NULL){     //ahel oli tühi
     P=New;
     goto dai;
     }

koht:  if(a<L->key){
         if(L==P){       //uus läheb esimeseks
            New->next=L;
            P=New;
            goto dai;
            }
         Pr->next=New;   //uus Pr ja L vahele
```

[1] So. töö käigus on elemendid alati mittekahanevalt ja seejuures stabiilselt järjestatud

```
          New->next=L;
          goto dai;
          }

kas:    if(L->next==NULL){       //ahela lõpp, uus sinna
          L->next=New;
          goto dai;
          }
       Pr=L;                          //pistmiskohta pole veel leidnud
       L=L->next;
       goto koht;
ots: L=P;
     printf("\nsorted vector: ");
ring: if(L==NULL){
         getchar( );
         P=NULL;
         goto dai;     //uut järjendit tegema
         }
     printf("%d ",L->key);
     L=L->next;
     goto ring;
}
```



Joonis 8.a. Pistemeetodil moodustatud ahela „info“ väljatrükk.

D. Knuth [Knuth I, lk. 296] toob valiku näiteoperatsioone tööks ahelaga:

1. *k*-nda lüli kättesaamine;
2. *k*-nda lüli järele uue lüli lisamine;
3. *k*-nda lüli eemaldamine ahelast;
4. kahe ahela ühendamine;

5. ahela „lõhkumine“ kaheks ahelaks;
6. ahela kopeerimine;
7. ahela lülide loendamine;
8. ahela lülide järjestamine mingi „info“-osa välja(de) väärtus(t)e järgi;
9. leida ahelast lüli mingi „info“-osa välja(de) väärtus(t)e järgi.

Kui kasutamegi ahelat dünaamilise andmestruktuurina, siis tundub loomulikuna, et lüli „kättesaamiseks“ kasutame *k* väärtusi kas 1 („anna esimene“) või *n* (*k*=1, 2, ...,*n*) – „anna viimane“; uue lüli lisamine toimub üldjuhul ahela lõppu (*k*=*n*), eemaldamisel võiks *k* olla kas „jooksva lüli“ järjekorranumber (või kas 1 või *n*). Ahela lülide loendamine on triviaalne; seda tööd võiksime vältida, kasutades *deskriptorit* (vt. *listD*→*n*, mida muudetakse iga kord, kui ahelasse lisatakse või sealt eemaldatakse lüli).

Ahela lülide järjestamine on mõneti kahtlase väärtusega toiming[1], mõtet võiks sellel olla vaid juhul, kui ahel on „valmis“, so. saanud *staatiliseks.* Kui siiski, sest tavaliselt kasutame ahelaid sellise info salvestamiseks, millel on (mingi või oluline) roll lüli lisamise *ajal.* Me käsitleme noid asju järgmises jaotises (*magasinid*).

Ent lüli otsimine „info“ järgi on mõnedel juhtudel mõttekas tegevus, näiteks siis, kui ahelaga kujutatakse *prioriteetidega järjekorda* ning erinevate prioriteetidega liikmed on ühises järjekorras[2].

Kuivõrd kõik need ülaltoodud operatsioonid on programmeerija jaoks triviaalselt realiseeritavad, ei näitlikusta me neid *C*-tekstidega (seda võivad lugejad ise „näpuharjutuseks“ teha). Mõistagi, dünaamilise (so, normaalse) ahela puhul tuleb uue lüli jaoks küsida dünaamilist mälu (näit. funktsiooniga *malloc*), ja kui eemaldatud lüli enam vaja pole, siis vabastada mälu (funktsiooniga *free*).

Ahel on andmestruktuur, mille abil saame esitada *kõiki andmeagregaate*, alates *vektorist* ja lõpetades *tabeliga.* Kahtlejatele: vektor on *indekseeritav* andmestruktuur, ent kuskil pole öeldud, et indekseerimine peaks toimuma *C*-keele või masinkoodi-tasemel, see võib samahästi olla realiseeritud kui protseduur, mis „liigub *i*-ndale lülile, kuni $i < n$“, ja *i* on lüli järjekorranumber ($i = 0..n-1$). Iseasi on efektiivsus, ent samamoodi „iseasi“ on toimetulek lahendamisaegses dünaamilises keskkonnas.

Samamoodi, ahelate abil, võime mälus kujutada *n*-mõõtmelisi massiive; elemendi (2-mõõtmelise *int*-massiivi „lahtri“) kirjeldus võiks olla järgmine:

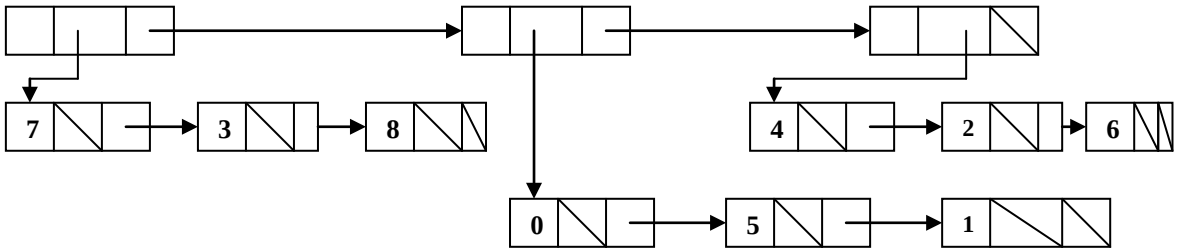
```
struct lahter{
        int value;      //tühi, kui lahtrist algab alamahel (alluv ≠ ∅ )
        struct lahter *alluv;
        struct lahter *naaber;
        };
```

[1] Siiski, kui abstraktne andmestruktuur *tabel* (võtmega varustatud kirjete hulk) on esitatud kirjete ahelana, on järjestamine ilmselt mõttekas tegevus (kui soovime koostada *järjestatud tabeli*).
[2] Näiteks, kui *N*-ajal seni rahulikult sabas olnud sõja- või tööaumärkidega veteranid said väljaspool järjekorda oma viinerikilo või voodilinad kätte siis, kui kaup hakkas otsa lõppema: prioriteedid lõid järjekorra kaheks (vt [Luik]).

Meile juba tuttava kahemõõtmelise arvumassiivi *int m*[3][3]

	0	1	2
0	**7**	**3**	**8**
1	**0**	**5**	**1**
2	**4**	**2**	**6**

esitus ahelate abil võib olla järgmine (vasakus ülanurgas olevale elemendile p.o. viit väljaspool struktuuri):



Ilmselt struktuurielement *value* esitab massiivi „lahtri“ väärtust siis, kui viit „alluv“ = ∅.

Arvatavasti peame osutame lugeja tähelepanu esialgu märkamatule seigale: kui me deklareerime, näiteks 3×3-maatriksi *C*-vahenditega, siis nii see ka on ja jääb: maatriksis on alati 3 ri-a ja igas 3 veergu. Ent kui me teeme *n*-mõõtmelise massiivi ahelate abil, siis on meil, kasutajatel, vabad käed nii massiivi mõõtmete arvuga manipuleerimiseks kui ka iga mõõtme siseselt alamstruktuuri tekitamiseks – ahelate abil genereerime meile vajaliku andmestruktuuri *programselt.* Näiteks nii, et maatriksi 2. reas on mingil hetkel 12 ja 3. reas 7 veergu. Mõistagi peaks selline „isetegevus“ olema kajastatud samuti dünaamilises deskriptoris.

Keerulisemate andmestruktuuride realiseerimist, so. „füüsilist esitust“ ahelate abil püüame seletada järgmistes jaotistes, alates *puid* käsitlevast jaotisest.

9. Magasinid

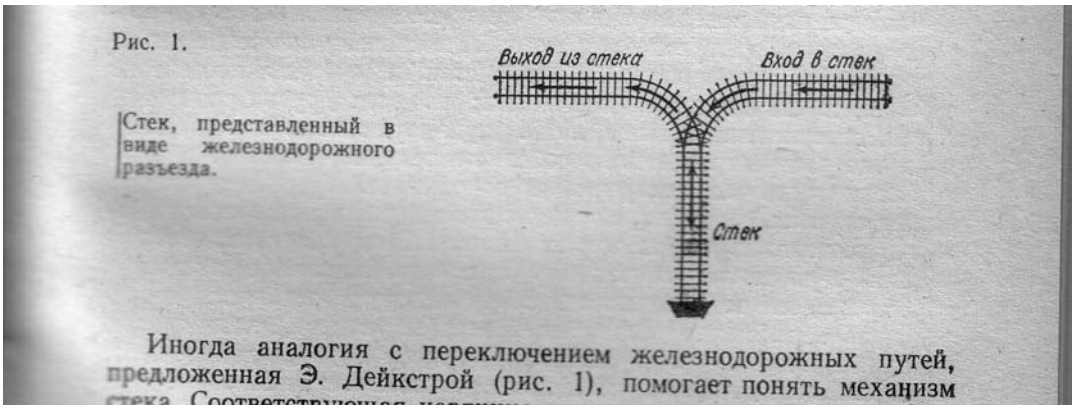
9.1. Sissejuhatus

Magasin (ingl. k. *stack*, vene k стэк) on „asi“, mille liigitamine *abstraktsete andmestruktuuride* hulka – ehkki nii on tehtud[1] – tekitab kõhklusi. Meie arvates on ta pigem *võte* või *printsiip* vms. Magasin realiseeritakse tavaliselt vektori abil (või mitme vektori abil mitmetraktilise magasini puhul) või mingit tüüpi ahela abil ning mingit omaette andmestruktuuri magasin ei evi.

Sarnaselt *vektorile*, millel on masinkoodi-tugi (algaadress+(indeks)), on (vähemalt) *PDP*-11 masinatest alates realiseeritud nn. *aparatuurne magasin* (vt. näit. [Isotamm, PK] lk. 128 jj.) – mis ei tähenda, et oleks konstrueeritud mingi alternatiiv lineaarsele mälule, vaid tähendab tõika, et mingi osa operatiivmälust on eraldatud magasinile ning selle mäluosaga manipuleerimist toetatakse masinkoodi-käskude tasemel: register *stack pointer*, operatsioonid *push* ja *pop*, mis lisavad ja eemaldavad elemendi ning modifitseerivad magasiniviita[2].

Ülalmainitud aparatuurne magasin „teenindab“ protsessori toel ainult ühte magasinitüüpi, mida tänapäeval tihtipeale samastataksegi mõistega *stack*: selle akronüüm on *LIFO-stack* (*Last-In-First-Out*[3]).

Sedatüüpi magasini näitlikustama sobib, vähemalt poiste jaoks, (pool)automaatrelva padrunipide: esimesena salve pandud padrun teeb viimase ning viimasena lisatud padrun teeb esimese paugu. Programmeerimises kasutatakse *LIFO*-tüüpi magasini kõige laialdasemalt, hiljem toome asjakohaseid näiteid. *Knuth* [Knuth I, lk. 297] illustreerib *LIFO*-stack'i *Dijkstra* (temast tuleb hiljem pikemalt juttu) raudtee-pildiga[4]:



Joonis 9.1.a. *LIFO*-magasini illustratsioon.

Programmeerija (sh. translaatoritegija) jaoks ongi see ainus „toetatav“ magasinitüüp, ent lisaks *LIFO*le on olulised veel järgmised:

- *FIFO* (*First In First Out*) – *järjekord.* Tänapäeval näeme sedalaadi andmestruktuure vahetevahel (tipptunnil) poekassas, lennujaama *check-in*is jmt. kohtades.Programmee-

[1] Vt. näit. [A,H&U, lk. 57]
[2] Hiljem esitame nende operatsioonide võimalikud algoritmid *C*-keeles erinevate magasinitüüpide jaoks.
[3] „Viimasena sisse, esimesena välja“.
[4] Tagapool me esitame nii sootuks elulähedasema algoritmi rongide koostamiseks, Selle pildi oma sobib veduri panekuks endist esimest vagunit lükkama, endine viimane vagun sõidab kõige ees.

rimises on järjekord oluline ajajaotusrežiimiga operatsioonisüsteemides, sh. tavaliselt kui *queue* – seletame pisut hiljem, ent mõeldavad on teat. rikkumised (vt. [Luik]).

- *queue* („järjekord, saba“) – ent mitte „normaalse saba“ mõttes, vaid kui selline järjekord, kust kliendid ei lahku, vaid asuvad pärast teenindamist uuesti „sappa“. Juba viidatud N. ajal moodustus selline *queue* mõnel heal päeval, kui Pälsoni (*Pepleri*) keldripoodi oli toodud kohviube, klausliga, et 200g nina peale. Mõne ringiga (juurdetulijad lasti mõistagi solidaarselt vahele) võis juhtuda, et taskukohane kogus oligi kõigil käes. Suurarvutite ajajaotusega operatsioonisüsteemid andsid ressursse (protsessoriaeg, operatiivmälu ja välisseadmed) ringjärjekorra-klientidele mingiks ajakvandiks, eelistades kõrgema prioriteediga kliente.
- *deque* – magasin, kus juurdepääs on *mõlemis otsas*, nii, et nood otsad toimivad kui *LIFO*-magasinid. Ja kuivõrd me juba ilukirjanduslike näidete juurde juhtusime, siis sedasorti magasin võiks olla illustreeritud poega, kus kinnipanekuni on aega 10 minutit, uks on veel lahti, aga ette tulevad prioriteetsemad kunded (*LIFO*) ja saba lõpus annavad mõned alla (ka *LIFO*, ainult et teisest otsast). Selgituseks: kui kell on kukkunud, siis müük lõpetatakse (N. ajal lõppes suvalise alkohoolse joogi müük kell 19.00).

9.2. LIFO-tüüpi magasin

Tundub, et termin *stack* (v.k. стэк) on globaalses teaduskirjanduses kinnistunud tähistama just *LIFO*-magasini. Näiteks kirjutas *Terrence Pratt*[1] [Pratt, lk. 22]: „...üht ja sedasama mõistet on tähistatud paljude erinevate terminitega, näiteks andmestruktuuri, mida meie nimetame „*stack*iks“ (стэк, meie märkus), on nimetatud ka „magasin-nimistuks“ ja „*LIFO*-nimistuks“ (- - -) Termin *stack* on praegu ilmselt selle andmestruktuuri tähistamiseks kõige kasutatavam.[2]“ Meil on nähtavasti vedanud, kuivõrd meil on olemas üldmõiste *magasin* ja selle liigid – mujal tundub *üldmõiste* puuduvat.

Toimetaja märkus. Leidub siiski mitmeid teisigi käsitlusi, milles kasutatakse termineid *LIFO*- ja *FIFO*-magasin. Üldiselt on aga selgem lähtuda mõistest *dünaamiline andmestruktuur* (*järjend*), mille erijuhud (magasin, järjekord jt.) on määratud eeskätt sellega, kuidas on defineeritud elementide lisamise ja eemaldamise operatsioonid. Vt. ka [Kiho03, lk. 26].

Sissejuhatus oli mõeldud hoiatamaks lugejat, et kui kirjanduses käsitletakse *stack*i, siis (kui pole selgelt teisiti öeldud) mõeldakse *LIFO*-tüüpi magasini.

Joonis 9.2.a. *Friedrich Ludwig Bauer.*

[1] Raamatu originaal, *Terrence W. Pratt, Programming Languages. Design and Implementation, University of Texas at Austin*“ ilmus 1975. aastal.

[2] Osundatud raamatu tõlke 85. lk. lisab *Pratt* veel kaks nimevarianti: *пушдаун-список* ja *магазин.*

Selles alapunktis mõtlemegi *magasini* all *LIFO-tüüpi* magasini, ja selle võttis 1951. aastal kasutusele saksa arvutiteadlane *Friedrich Ludwig Bauer* (vt.joon. 9.2.a), kes projekteeris ning realiseeris avaldiste töötlemise tolleaegse arvuti *Stanislaus* jaoks.

Friedrich Ludwig Bauer sündis 10. juunil 1924. a. Regensburgis, lõpetas Ludwig-Maximiliani Ülikooli, praegu on ta Müncheni Ülikooli emeriitprofessor. Lisaks magasini leiutamisele osales *Bauer* ka *Algol-58* ja *Algol-60* meeskondades, 1968. aastal pakkus ta välja termini *software engineering*[1] (tarkvaratehnika) ja oli mõjukaim isik *arvutiteaduse* (*computer science*) kui iseseisva distsipliini õppekavade avamiseks Saksa ülikoolides. Seni viimase suure tööna ilmus tema sulest 1983. aastal mahukas raamat „*Cryptology – Methods and Maxims*" [Bauer]. Krüptoloogia-teema huvitav tutvustus sisaldub *Baueri* kaasautorlusel 1975. a. ilmunud raamatus [Bauer jt.].

Ent naaskem *magasini enda* juurde. *LIFO*-tüüpi magasin võib olla realiseeritud kaheti: kas *vektori* – see on aparatuurse magasini ainus mõeldav vorm – või *lihtahela* kujul; rakendusprogrammis võime kasutada mõlemat varianti.

Alustagem **vektoresitusest**. Vektori definitsiooni kohaselt on kõik tema elemendid sama tüüpi[2] ja järelikult ka ühepikkused. Magasinil on põhiparameetriteks talle eraldatud mäluvälja *pikkus*, so. maksimaalne elementide arv, ja jooksev viit magasini „aktiivsesse piirkonda"; selle järgi toimub magasinist lugemine (=eemaldamine) ja lisamine ning kontrollimine: kui viit modifitseerimise käigus osutab magasinipiirkonna algusest „ettepoole", antakse signaal „tühi" (*empty stack*) ja kui tahetakse kirjutada aadressile väljaspool piirkonda, on signaaliks „ületäitumine" (*stack overflow*).

Kui me tegutseme masinorienteeritud keskkonnas (näit. *Inteli assembler*) või magasinorienteeritud keeles (nagu *Forth*), siis pole meie jaoks oluline, *kuidas* magasiniviita nihutatakse – oluline on, et operatsioonid *push* („lükka sisse") ja *pop* („lükka välja") töötavad. Ent kui me programmeerime magasini ise, on valida kahe variandi vahel: magasiniviit on valmis kas kirjutamiseks (lugemiseks peame teda nihutama koha võrra tagasi) või lugemiseks (kirjutamiseks peame teda nihutama koha võrra edasi). Oma näidetes realiseerime esimese variandi, põhjendusega, et algselt on magasin tühi ning viida abil saame sinna kirjutada esimese elemendi. Juhime tähelepanu seigale, et *pop* ei kustuta magasini tipmist elementi, ta teeb selle koha ainult *ülekirjutatavaks* järgmise *push*i poolt ja kättesaamatuks järgmisele *pop*ile.

Esimeses näites kujutame magasini *string*ina *stack* ning lisaks kasutame *FIFO*-tüüpi magasini *bin*; programm saab ette sümbolkujul (mittenegatiivseid) kümnendtäisarve, teisendab need kahendarvudeks ja trükib sümbolkujul välja. Idee sellise näiteprogrammi kirjutamiseks saime *V. Leppikson*i raamatust [Leppikson C, lk. 11], kus märgita täisarvude masinkuju tutvustamise käigus anti teisenduse $10 \rightarrow 2$ algoritm järgmise näitega, teisendatakse arvu 37_{10}:

37 : 2 = 18 jääk 1
18 : 2 = 9 jääk 0
9 : 2 = 4 jääk 1
4 : 2 = 2 jääk 0
2 : 2 = 1 jääk 0
1 : 2 = 0 jääk 1

[1] Selle valdkonnaga tutvumiseks võiks teiste hulgas sobida ka mahukas (793 lk.) raamat [Pressman], kus lk. 23 tunnustatakse *Bauer*i terminiloomet (tõsi, *Roger Pressman* nimetab teda ameerikapärase familiaarsusega *Fritz Bauer*iks).
[2] Oluline on, et *C*-kontekstis kuulub nende hulka ka *viidatüüp.*

ning kahendarvu saame, kirjutades jäägid välja vastupidises järjekorras: 100101. Teisendus on hõlpsasti tehtav *LIFO*-magasini abil, esimene jääk pannakse magasini „põhja“ ja viimane – „tippu“; väljundrea saamiseks teeme tsükli, mille „kehas“ on sisuliselt kaks operaatorit: *char s=pop*(stack) ja *push*(*s, bin*). Tegelikus programmis me ilmutatud kujul *push*i ja *pop*i ei kasuta. Jagamiseks kasutame teegi *stdlib.h* funktsiooni *div* kirjeldusega

```
div_t  div(int num, int denom);
```

kui muutuja *dec* on kirjeldatud `div_t dec;` siis kahe täisarvu jagatis *dec* arvutatakse kui *num* : *denom* ja ta väärtus on kaheväljaline struktuur: jagatise täisosa on väljal *dec.quot* ning jääk – *dec.rem*. Mõlemad on *int*-tüüpi. Niisiis, **programm:**

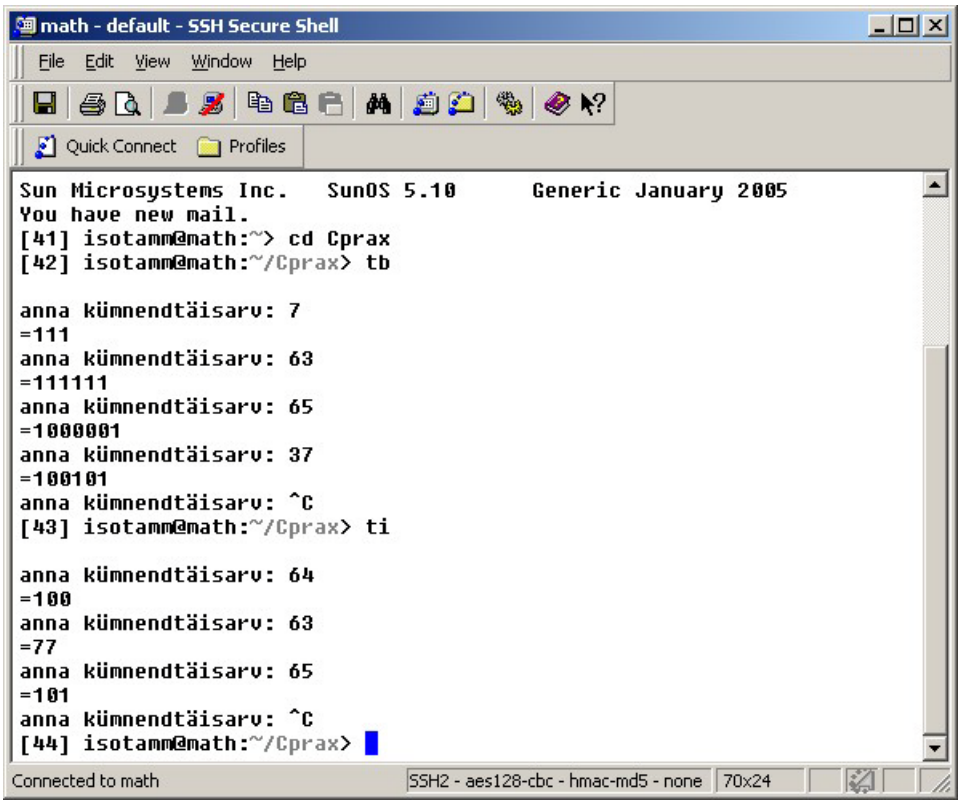
```
//tentobin.c  sümbolkujul :: kümnendarv --> kahendarv.  10.12.08  (execute tb)
#include <stdio.h>
#include <stdlib.h>

int main( ){
   div_t dec;
   int i,j,sp;
   char stack[32];   //LIFO-stack :: täidame „alt-üles“
   char bin[33];     //FIFO-stack :: täidame „vasakult-paremale“
   memset(bin,'\0',33);    //väljundstringi nullimine
   int arv;      //siia loetakse arv funktsiooniga scanf
ring: printf("\nanna kümnendtäisarv: ");
     scanf("%d",&arv);
     sp=31;   //LIFO-magasini „põhja“ indeks
conv: dec=div(arv,2);   //(dec) .quot on täisosa ja .rem on jääk
     stack[sp]=dec.rem+48;   //jääk (sümbolkujul) magasini: numbri 0 ASCII-kood on 48
     sp--;    //uus kirjutamisindeks, liigume „alt üles“
     arv=dec.quot;    //arv=jagatise täisosa
     if(arv>0) goto conv;  //teisendamine jätkub. Kui arv==1, siis quot=0 ja rem=1
     j=0;
//kirjutamine LIFO → FIFO.  sp algväärtus on stack'i (LIFO) kirjutamisindeks, bin on
//järjekord: (FIFO, täidetakse vasakult paremale. i algab kasutatud magasinitipust (sp+1)
     for(i=sp+1; i<32; i++,j++) bin[j]=stack[i];   //push(bin) ja pop(stack)
     bin[j]='\0';  //stringi lõputunnus paika
     printf("=%s",bin);
     goto ring;  //töö lõpetab Ctrl+C
}
```

Teine näiteprogramm teisendab samuti „10 → 2“, ent resultaadina „pakib“ bitthaaval kokku *int*-väärtuse – sobides loodetavasti ka bitioperatsioonide (nihutamine ja disjunktsioon) näiteks.

```
//tentoint.c  bittesitus :: kümnendarv --> kahendarv.  10.12.08  (execute ti)
#include <stdio.h>
#include <stdlib.h>
```

```
int main( ){
   div_t dec;
   int i,sp;
   char stack[32];
   int intarv;     //siia kogun kahendarvu
   int  arv, bitt;
ring: printf("\nanna kümnendtäisarv: ");
      scanf("%d",&arv);
      sp=31;
      intarv=0;
conv: dec=div(arv,2);
      stack[sp]=dec.rem;
      sp--;
      arv=dec.quot;
      if(arv>0) goto conv;
```



Joonis 9.2.b. Teisendused „10→2“ ja „10→8“.

```
 //LIFO tipust võetav väärtus (0 või 1) kirjutatakse „intarv'u“ bitthaaval
     for(i=sp+1; i<32; i++){
       bitt=stack[i]<<(31-i);      //biti nihutamine vasakule (...2, 1, 0 kahendkohta)
       intarv=intarv|bitt;     //vasakpoolseima biti lisamine kogutavasse int-arvu
       }
     printf("=%o",intarv);  //kaheksandarvu trükk. 16-ndarvu trükiks: %x
     fflush(stdin);   //puhastame sisendi
```

```
    goto ring;
}
```

Meie lugejale pakume välja „näpuharjutuse“: kirjutage programmid „10 → 3 kuni 10 → 9, parem, kui objektsüsteem (millisesse süsteemi teisendada) oleks parameetrina ette antav. Ning kindlasti tuleks kirjutada programmid tagasiteisenduseks näiteks, „2 → 10“ vms. Ideid sellisteks teisendusteks võib saada näiteks raamatust [Leppikson C], lk. 11..12.

Kolmas näide on mõnevõrra keerulisem, siin tuleb aritmeetiline[1] (sulg)avaldis viia *inverteeritud Poola kujule.* Seda tööd tegeva programmi tutvustus ja tekst on lisas 7

Avaldiste (aritmeetilised või loogilised või nende kombinatsioonid) puhul on olulisim, kuidas keeles on fikseeritud elementaaravaldis, võimalikud on 3 kuju (näitetehe on muutujate *a* ja *b* liitmine)[2]:

- *infiks* (tehtemärk paikneb operandide vahel – just nii nagu (kooli)matemaatikast harjunud oleme: *a* + *b*;
- *prefiks*: + *a b* (nii on kombeks funktsionaalsetes keeltes: „+” on kahe argumendi – *a* ja *b* funktsioon);
- *postfiks*: *a b* +.

Enamik masinast sõltumatuid programmeerimiskeeli toetavad avaldiste *infiks*-kuju, kusjuures tehetele on keele kirjelduses tavaliselt[3] omistatud *prioriteedid, Algol-60* tehetel on need kahanevas järjekorras sellised:

priorit.	**tehted**	**Semantika**
7	↑	astendamine (*C* funktsioon *pow*)
6	× / ÷	korrutamine, jagatise täisosa leidmine, jäägiga jagamine (*C*: *div*)
5	+ –	liitmine ja lahutamine
4	< ≤ = ≥ > ≠	võrdlustehted (*C* liitsümbolid: <=, ==, >=, !=)
3	¬	eitus (unaarne tehe, *C* !)
2	∧	konjunktsioon (*C* „&“ (bitikaupa) või „&&“)
1	∨	disjunktsioon (*C* „|“ (bitikaupa) või „||“)
0	( ) [ ]	sulud

Tehete sooritamise järjekorda saab tavaliselt muuta sulgude abil. Näiteks, kui me kirjutame *A*+*B* * *C*–*D*, siis leitakse kõigepealt *B* * *C*, ent korrutustehe sooritatakse viimasena, kui kirjutame (*A*+*B*) * (*C*–*D*). Sama tavaliselt sooritatakse ühesuguse prioriteediga tehted järjekorras vasakult paremale; nii on see ka *C*-keeles.

Äsjatoodud avaldise *Poola kuju*[4] (avaldise suluvaba prefiks-esitus) on järgmine:

× + *A B* – *C D*

---

[1] Samamoodi saab toimida ka loogilise avaldisega.

[2] Vanas eesti koolimatemaatikas nimetati “liitmist” “kokkulöömiseks”. Kui *a* on üks telliskivi, *b* teine telliskivi ja “+” on “löö kokku”, siis *prefiks* annab eeskirja: “löö kokku, võta 1. kivi, võta 2. kivi”, *infiks* “võta esimene kivi, löö kokku, võta teine kivi” ja *postfiks* – “võta esimene kivi, võta teine kivi, löö kokku”.

[3] Prioriteete pole näiteks *APL*-is.

[4] Ingl. k. *Polish Form*, vene k. *Польская запись*. Selle autor on Poola matemaatik *Jan Łukasiewicz* (vt näit. [Isotamm, PK] lk. 226)

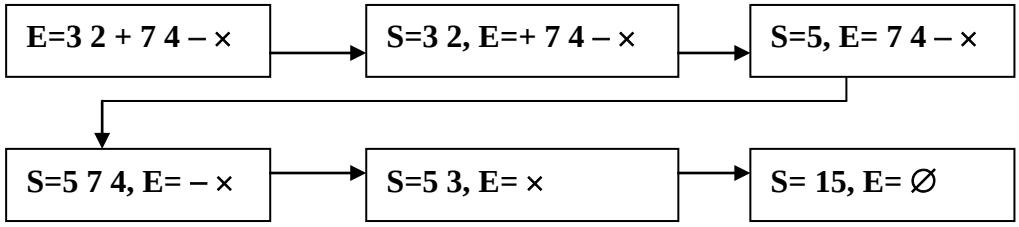
Poola kuju variant *CPF* (*Cambridge Polish Form*[1]) on programmeerimiskeele *Lisp* programmide (operaatorid ja andmed) esitamise ainus aktsepteeritav moodus. Ülaltoodud avaldis näeb *CPF*is välja nii[2]

(×(+ A B)(– C D))

Allikate (vt. [RPN]) järgi „keerasid“ *F. L. Bauer* ja *E. Dijkstra* Poola kuju „ümber“: *prefiks*-avaldisest sai suluvaba *postfiks*-avaldis (*inverteeritud Poola kuju*, ingl. k. *Reverse Polish notation*, vene k. обратная Польская запись). Meie näite see kuju on järgmine:

A B + C D – ×

Postfiks-kuju ei kasutata tavaliselt programmeerimiskeelte avaldiste süntaksis (erand on näiteks *FORTH*), ent sellel kujul esitatud avaldisi on *LIFO*-tüüpi magasini abil väga lihtne lahendada. Näiteks, olgu *A*=3, *B*=2, *C*=7 ja *D*=4. Siis võime oma näite kirjutada kujul
3 2 + 7 4 – ×;
Magasini abil arvutatakse avaldise väärtus välja nii: avaldist töödeldakse vasakult paremale, operandid pannakse magasini, tehe sooritatakse magasini tipmisele eelneva elemendi ja tipmise elemendi vahel (lahutamine ja jagamine pole kommutatiivsed tehted) ning resultaat kirjutatakse magasini operandide asemele. Illustreerime seda järgmisel joonisel (avaldist tähistame *E* ja magasini *S*-ga), seisud on esitatud järjekorras vasakult paremale ja ülalt alla:



Joonis 9.2.c. *Inverteeritud Poola kujul* avaldise väärtuse leidmine.

Teadaolevalt kasutati esimeses *Algol*-60 translaatoris *infiks*-avaldise teisendamiseks *inverteeritud Poola kujule Dijkstra* „sorteerimisjaama algoritmi“ (ingl. k. *railway shunting yard*), mis järgib lihtsaid reegleid [RPN]:

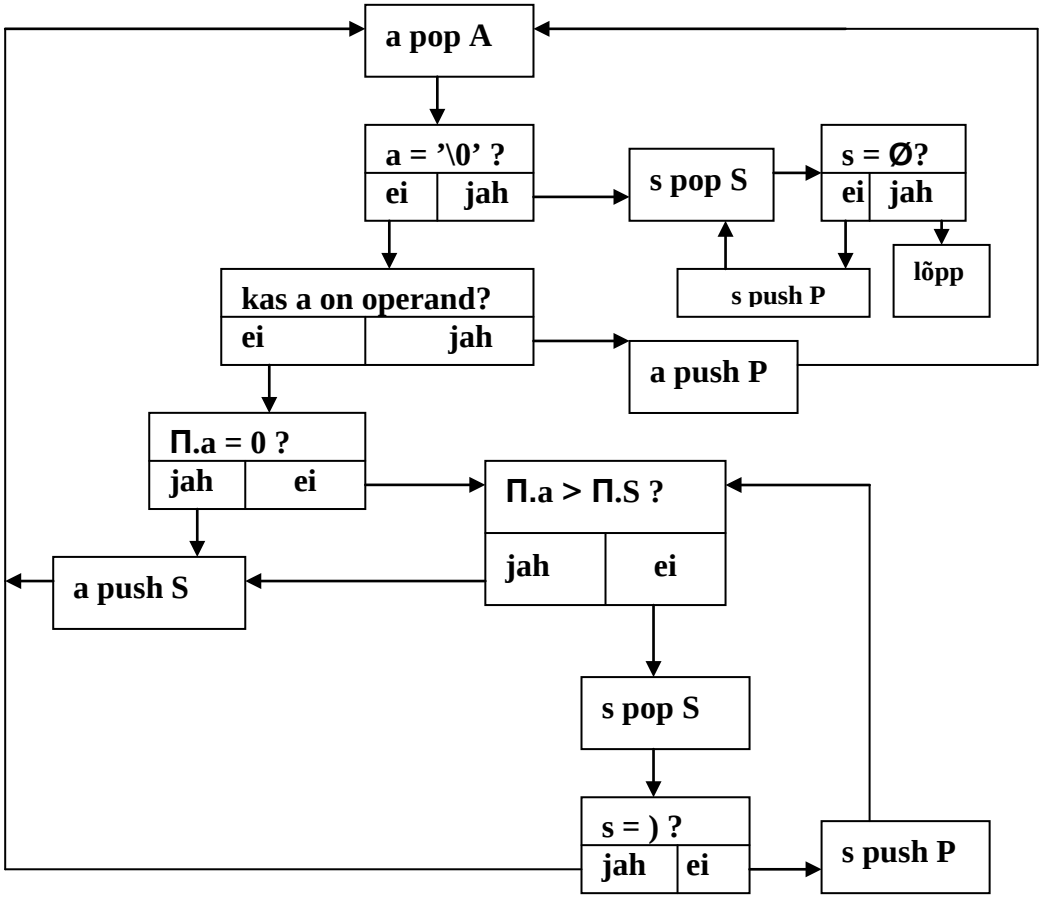
- Operandid lähevad sisendjärjekorrast väljundjärjekorra lõppu (mõlemad on *FIFO*-tüüpi).
- Tehete jaoks kasutatakse *LIFO*-magasini (“tupik“). Alustav sulg (sisendjärjekorrast) läheb alati sinna ning lõpetav sulg viib tupikust kõik sümbolid kuni alustava sulu leidmiseni väljundjärjekorda; sulgusid endid sinna ei kanta.
- Muu eraldaja *e* lisamisel tupikusse võrreldakse *e* prioriteeti tupiku tipmise sümboli *x* omaga. Kui *e* prioriteet pole *x* omast väiksem, siis kantakse *e* tupikusse, vastasel korral viiakse *x* väljundisse (kuivõrd kõrgema prioriteediga tehe tuleb sooritada *enne* madalama prioriteediga tehtest); seda korratakse seni, kuni tohib *e* magasini lisada.

[1] Vt. näit. [Isotamm,m PK] lk. 224 jj.)
[2] See pole *Lisp*, seal on tehted ümber nimetatud funktsioonide nimedega, näit. ’+’ (so, liitmine) on *PLUS*( ).

- Kui sisend on ammendatud, siis viiakse tupikust kõik sinna jäänud sümbolid väljundisse.

Järgnevatel joonistel esitame *Dijkstra algoritmi* plokkskeemi (joonis 9.2.d) ja seda algoritmi järgiva teisenduse pildi (joonis 9.2.f). Kasutame kolme magasini: *FIFO*-tüüpi magasinis *A* on *infiks*-avaldis lõputunnusega '\0' , *FIFO*-tüüpi magasini *P* kogutakse avaldise inverteeritud Poola kuju ning eraldajate töötluseks kasutatakse *LIFO*-tüüpi magasini *S.* Operaator *a pop A* tähendab, et *A*-st eemaldatakse tipmine element ja selle väärtus omistatakse muutujale *a*, ning *a push S* tähendab, et *a* kirjutatakse magasini *S* tipmiseks elemendiks *a push P* tähendab *a* kirjutamist *P* lõppu. Π.*a* on *a* prioriteet (kui *a* on eraldaja) ning Π.*S* on magasini *S* tipmise elemendi prioriteet.

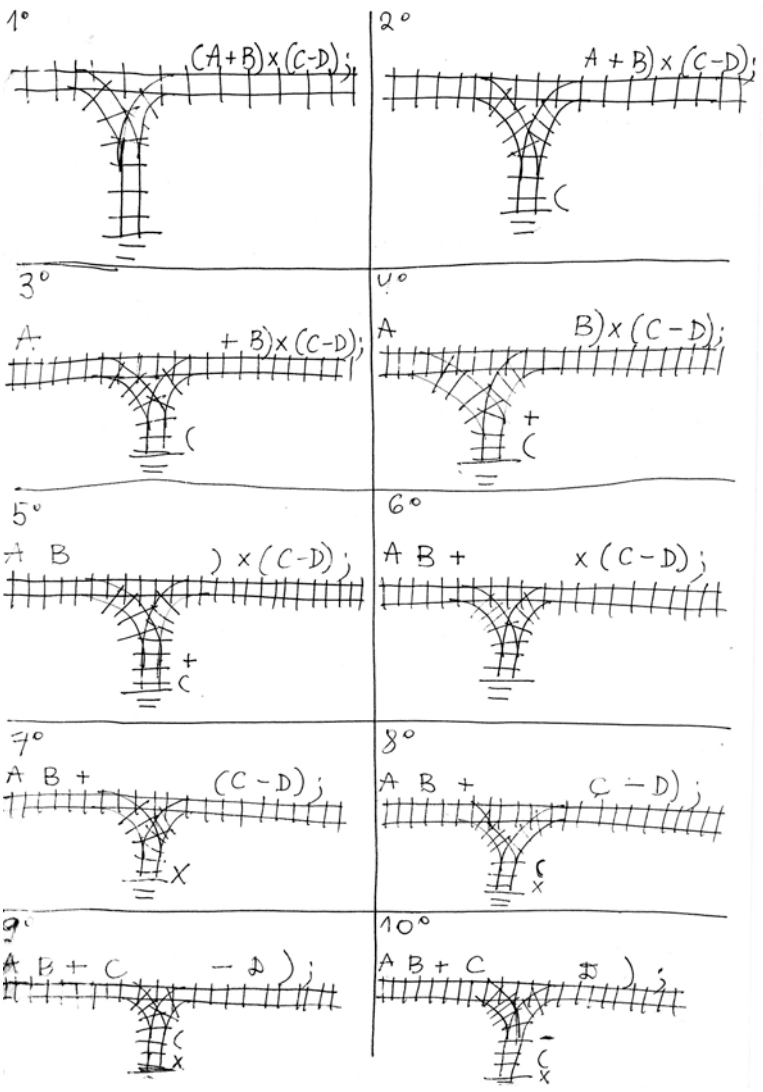
Joonisel 9.2.f on magasin *A* tupikust (magasin *S*) paremal ning magasin *P* vasemal; „vaguneid" liigutatakse paremalt vasakule – kui nad on operandid, ja tupikusse ning sealt välja vasakule, kui prioriteedid, lõpetav sulg või lõputunnus seda nõuavad.



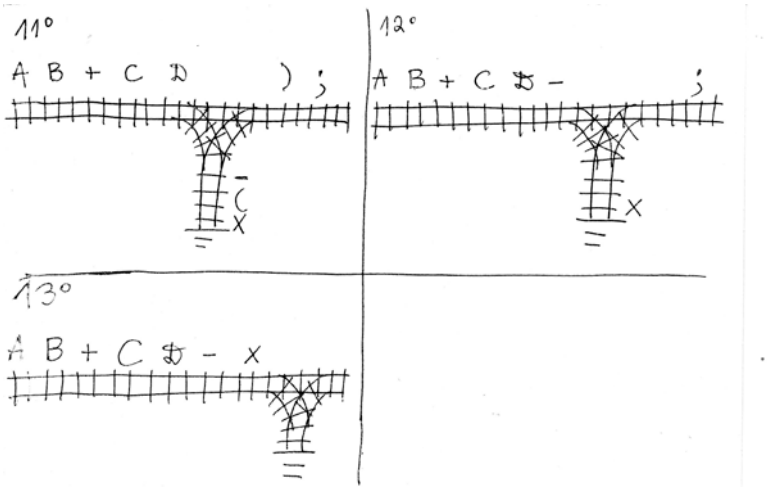
Joonis 9.2.d. *Dijkstra* sorteerimisjaama-algoritm.

Joonis 9.2.e. *Edsger Wyte Dijkstra* (1930 – 2002).

Dijkstra oli hollandi matemaatik ja arvutiteadlane. Märksõnad: inverteeritud Poola kuju, mille ta töötas 1960-ndate alguses välja koos *Friedrich L. Bauer*iga, on ainult üks seik, millega *Dijkstra* on arvutiteaduse ajalukku läinud. [EWD] andmetel on tema tähtsamateks saavutusteks *Dijkstra algoritmina* tuntud lühima tee leidmise meetod graafis (sorteerimisjaama-algoritm on *Dijkstra's shunting yard algorithm*), aluse panemine *struktuurprogrammeerimise* paradigmale (*goto*-vaba stiil)[1], multiprogrammeerimise süsteemi *THE* loomine, paralleelprogrammeerimisse *semafori* kontseptsiooni lisamine ja osalemine *Algol*-60 esimese kompilaatori kirjutamisel. Wikipedia artiklis [EWD] on muuhulgas tsiteeritud *Dijkstra*t: „Arvutiteadus pole rohkem teadus arvutitest kui astronoomia on teadus teleskoopidest". *Edsger W. Dijkstra* on töötanud *Eindhoven*i tehnikaülikoolis, *Burroughs*i korporatsioonis ning *Texase* ülikoolis (*Austin*) [EWD].



[1] Selles valdkonnas on tuntud *Dijkstra* 1968. a. märtsis *CACM*-is ilmunud artikkel „Go To Statement Considered Harmful" ning *Dijkstra, , C. A. R. Hoare* ja *Ole-Johan Dahl*i raamat „*Structured Programming*", 1972. (vt. [EWD]).



Joonis 9.2.f. Avaldise viimine *inverteeritud Poola kuju*le *Dijkstra* algoritmiga.

Võrrelgem ülaltoodud raudtee-varianti joonisel 9.1.a tooduga: lisandunud on *otsetee* sisendist väljundisse – mis ongi *Dijkstra* teene, ehkki pärisraudteel oli see koosseisude ümbermängimise meetod tuntud tolle „pärisraudtee“ algusest saadik, rongikoosseisu lisamisel mängisid oma osa *prioriteedid*. Ent: vististi on kõik toiumivad tehnolahendused „loodusest maha kirjutatud“ – samuti „vististi“ ainus erand on *ratas*, mida peetakse geniaalseimaks leiutuseks. Sest looduses selle prototüüpi lihtsalt pole.

Lisas 7 on toodud programm interaktiivseks aritmeetiliste konstantavaldiste viimiseks inverteeritud Poola kujule ning nende väärtuste väljaarvutamiseks, järgides ülaltoodud algoritme (sj. magasinide *vektor*esitust).

Niisiis, selles alajaotises vaatlesime magasini *vektoresitust*, mis muide on ka ainus aparatuurse magasini tüüp (*LIFO*). **Teine variant** on rakendusprogrammeerija jaoks magasini esitamine **lihtahela** kujul. Lisas 7 toodud programmis interpreteeritakse *postfiks*-avaldist vektormagasini *double Ls*[16] abil (moodul on *felix*). Allpool kasutame selle asemel dünaamilist andmestruktuuri, ahela lüli kirjeldab struktuur *lifo* ning selle kasutamiseks tuleb lisada programmile *IPoola.c* moodulid *makelyli* (mis annab mälu uue magasinielemendi jaoks), *apush* (kirjuta *LIFO*-magasini) ja *apop* (anna ja eemalda lüli magasini tipust).

```
struct lifo{
        double a;           //magasinielemendi väärtus
        struct lifo *next;  //magasiniaknast järgmine
};
```

Ning programmi *IPoola.c* tuleb lisada globaalne parameeter ahela pea jaoks, näiteks selline:

```
struct lifo *Pea=NULL;  //initsieerime tühja ahela
struct lifo *makelyli(void){
        struct lifo *L;
        L=malloc(sizeof (struct lifo));
        L->next=NULL;
        return(L);
}
```

```
void apush(double x){       //push: tee uus lüli ja pane ta LIFO-esimeseks, vana on 2.
        struct lifo *new;
        new=makelyli( );
        new->a=x;
        new->next=Pea;
        Pea=new;
}

double apop(void){
        struct lifo *first;
        double x;
        first=Pea;
        if(first==NULL){
                printf(„\nempty stack“);
                getchar( );
                abort( );
                }
        x=first->a;
        Pea=first->next;
        free(first);   //vabastan seni esimeseks olnud lüli mälu
        return(x);
}
```

Ja moodul *felix* on järgmine: //“felix“ nagu „vändaga aritmomeeter“

```
//Poola kuju interpretaator. O[„ut“] on postfiks-järjekord (FIFO-magasin)

int  felix(struct Sd *O){
    double x,y;
    int j=0,k;    //j on postfiks-stringi indeks: sümbolid, inv. Poola kuju. k: arvu kogumine
    char s, arv[16];  //siia kogun O->stack'ist (invert.Poola kuju)  arvu. next...' '.
    Pea=NULL;
next: s=O->stack[j];
     if((isdigit(s))||((s=='-')&&isdigit(O->stack[j+1]))||(s=='.')){
//arvuks lähevad number, unaarne miinus ja kümnendpunkt. Viimase väärkäsitlust ei käita.
//arvu ülekanne puhvrisse
        k=0;
        for(;(arv[k]=O->stack[j])!=' ';j++,k++);
        arv[k]=' ';
        j++;
        apush(atof(arv));
        goto next;
        }
    else s=O->stack[j];
    if(s=='\0'){
```

```
        printf("\navaldise väärtus = %2.2f",apop( ));
        return(1);
        }
    if(s==' '){
         j++;
         goto next;
         }
    y=apop( );
    x=apop( );
    switch(s){
           case '+': apush(x+y); break;
           case '-': apush(x-y); break;
           case '*': apush(x*y); break;
           case '/': if(y==0.0){
                       printf("\nmina ei oska nulliga jagada");
                       return(0);
                       }
                     apush(x/y); break;
        }
     j++;
     goto next;
    }
```

Sel variandil on mitmeid eeliseid aparatuurset malli järgiva meetodi[1] ees. Olulisim on see, et magasini ületäitumist ei pruugigi tulla, kui operatsioonisüsteemil on anda piisavalt palju mälu[2]. Näiteks *Java* virtuaalmasinal tekivad rekursiivsete algoritmide puhul probleemid, kui rekursiivseid pöördumisi on „liiga palju“: virtuaalmasin (*JVM*[3]) kasutab fikseeritud pikkusega magasini – mida saab pikendada, ent mitte lahendamise ajal.

Mainigem, et *IBM*i „suurarvutite“ (jutumärgid on lisatud tänaste mikroarvutite parameetreid teades) alamprogrammidesse pöördumistega kaasnev säilituspiirkondade *LIFO*-magasin oli rajatud samuti (topeltseotud) ahelatele (vt. näit. [Isotamm, PK], lk. 74).

Niisiis, *LIFO*-magasini jaoks on defineeritud ainult kaks operatsiooni: *push* ja *pop* (lisa ja eemalda). Ent sedatüüpi magasinile orienteeritud keeltes (näiteks, *Forth* või *java baitkood*) on palju neid elementaaroperatsioone kasutavaid „pealisehitusi“, näiteks operaator *swap*, mille põhimõtteline algoritm on järgmine:

```
a=pop; b=pop; push a; push b;
```

resultaadina vahetatakse *LIFO*-magasini kaks tipmist elementi. Nondest keeltest vt. näiteks [Isotamm, PK lk. 139 – 156 ja 273 jj.].

---

[1] Sel juhul on alati tegemist staatilise andmestruktuuriga: aparatuurse magasini „sügavus“ (võimalik elementide arv) on kas operatsioonisüsteemi või siis rakendusprogrammi „sisse kirjutatud“ ja täitmise ajal seda muuta ei saa.
[2] Iga uue lüli jaoks tuleb võtta dünaamilist mälu, *C*-variandis funktsiooniga *malloc* (või *calloc*).
[3] *JVM = Java Virtual Machine* (*Java virtuaalmasin*). Kompaktne magasin ei saa üle piiride minna, ent *ahela* abil tehtav magasin saab „elada“, kuni operatiivmälus on veel kasutamata ahelakvante (pikkuse määrab lüli formaat).

Ääremärkusena: lihtahela-jaotises toodud dünaamilise järjendi loomine *pistemeetodil* kasutas ahelat, ent seda ei saa me kuidagi siduda ühegi magasinitüübiga, kuivõrd lisamine ei toimu (tingimata) ahela algusse – nagu eeldab *LIFO* – ega lõppu – nagu nõuab *FIFO* –, vaid *sobivasse kohta.* Ent, ahelat vaatame läbi alates magasini tipust ja liikudes „allapoole“ (sealtoodud algoritm muud käsitlust ei võimaldagi, kuivõrd lihtahelas pole tagasiviitu).

Kordame: magasin (*stack,* стэк) on eeskätt tuntud ja kasutatud kui *LIFO*-tüüpi mehhanism. Hea tahtmise korral võime vektori järjestiktöötlust käsitleda nagu *FIFO*-magasini töötlemist (umbes nii, et kui vektori-indeks on juba suurenenud, siis tagasivõtmist ei ole). Tegelikud *FIFO*-kasutajad on *operatsioonisüsteemid*, mis peavad panema ressursse tahtvaid protsesse „sappa“, saamaks oma käsutusse soovitud ressursse (operatiivmälu, protsessoriaeg, õigus pöörduda välisseadmete poole jmt.).

Ja tegelikult, operatsioonisüsteemid ei kasuta magasini ei *LIFO*- ega ka *FIFO*-variandis, vaid *ringahelana* (*queue*). Me võiks seda programmeerida nii, et ahela pea viitab *aktiivsele* elemendile ning uus element pannakse selle *ette,* et tema teenindamine oleks (senikehtivas järjekorras) viimane (peame silmas *prioriteetideta* järjekorda).

Veelkord, *LIFO*-magasin on paljudel juhtudel asendamatu, *FIFO* on tõlgendamise küsimus (et kas on magasin või pelgalt vektor), ja ringahela kasutusala on (vist) marginaalne – niivõrd, kuivõrd on seda operatsioonisüsteemide kirjutamine.

10. Puud

Tsiteerigem *Jüri Kiho*t [Kiho 03, lk. 27]: „Puuks nimetatakse lõplikku tippude hulka, mis on kas tühi või milles üks tipp – juur ehk juurtipp – on välja eraldatud ning ülejäänud tipud on jaotatud $m \geq 0$ mittelõikuvaks alamhulgaks T_1, T_2, ..., T_m, millest igaüks on omakorda puu; alamhulkasid T_1, T_2, ..., T_m nimetatakse puu juure *alampuudeks. ... Kahendpuuks* nimetatakse lõplikku tippude hulka, mis on kas tühi või milles üks tipp – *juur* ehk *juurtipp* – on välja eraldatud ning ülejäänud tipud on jaotatud ülimalt kaheks mittelõikuvaks alamhulgaks T_v ja/või T_p, millest igaüks on omakorda kahendpuu; alamhulkasid T_v ja T_p nimetatakse vastavalt juure *T vasakuks* ja *paremaks* alampuuks“.

Kirjanduses on puu tippude subordinatsiooni mitmeti tähistatud, näiteks:

- patriarhaalne tähistamine: *juur* on isa (*parent*) ja alluvad on lapsed, omavahel vennad;
- bürokraatlik tähistamine:*juur* on ülemus, alumisel tasemel on alluvad (omavahel naabrid).

Mitterippuvatel tippudel on vähemalt üks alluv; alluvateta tipud on *lehed* e. *rippuvad tipud.*

Puud (ja eriti kahendpuud) on paljude algoritmide jaoks sobivaimad andmestruktuurid. Toogem mõned näited, alustagem *järjestamis*- ja *otsimispuust,* ja seda näitega: loeme klaviatuurilt mistahes kirjeldusega, **char*, *int* vms. „võtmeid“ ja teeme kahendpuu: esimene loetud „võti“ saab kahendpuu juure märgendiks, ja edasi toimitakse rekursiivselt nii: kui järgmine „võti“ on mingis mõttes „väiksem“ juurest, siis otsitakse talle kohta „vasakus“ alampuus, muidu aga „paremas“.

Programm lõpetab puu ehitamise siis, kui võtme asemel antakse „tühi *Enter*“, seejärel trükitakse võtmed kasvavas ja kahanevas järjekorras. Võtmeid võrdleme puus olevatega funktsiooni *strcmp*(cs,ct) abil, see tagastab *int*-väärtuse *r*: *r*<0, kui cs<ct, *r*=0, kui cs=ct ja *r*>0, kui cs>ct. Operandid on mõistagi stringid.

„Otsimispuu“-ülesande lahendamispilt on allpool. Programmi kutsume välja nii:

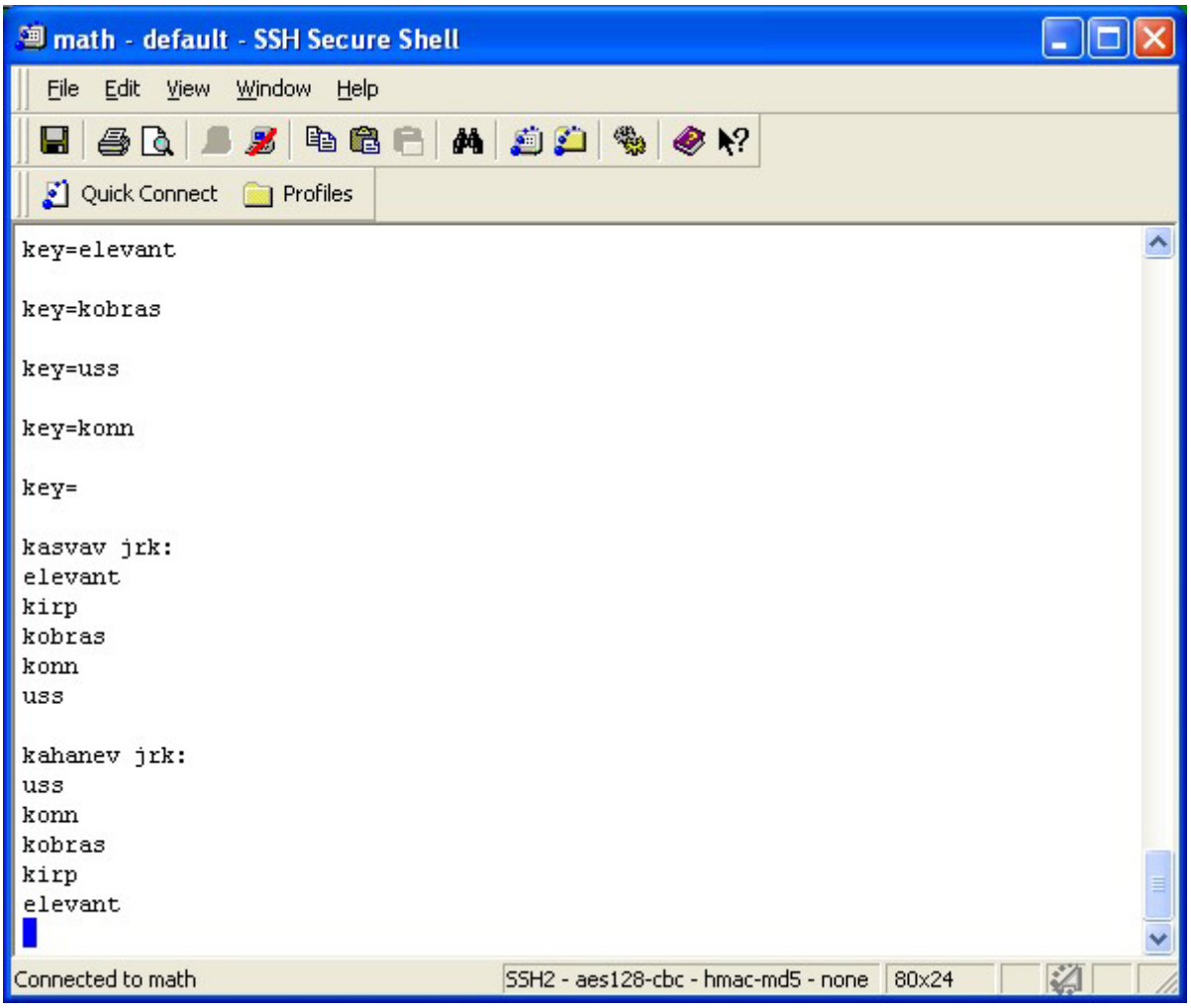
```
<prog-nimi> <failinimi>   (näiteks [44] isotamm@math:~/Cprax> spuu zoo);
```

Kui tolle nimega („zoo“) faili pole, tuleb ta luua, muidu aga sisse lugeda. Fail on viitadega seotud tippude hulk (puu).

Dialoogi käigus ehitatakse (või täiendatakse) puud. Kui mõni võti juba on puus, siis trükitakse olemasolev tipp (võti ja viidad) välja ning uut tippu puusse ei lisata[1].

Pärast puu trükkimist tuleb ta kirjutada (üldjuhul modifitseeritult) kettale tagasi, sama nimega.

[1] See on tavapärane käitumismall *tabelite* puhul: võtmed *peavad* olema *unikaalsed* (vt. relatsiooniline andmebaasimudel, infosüsteemide või andmebaaside kursuse materjalidest, seal on relatsioon lihtne, ainult aatomitest koosnev kirje).



Joonis 10.a. Otsimispuu.

```
//spuu.c Sisestab otsimispuusse võtmeid, väljastab kasvavas ja kahanevas järjekorras.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

FILE *mf=NULL;   //faili viida[1] kirjeldamine
struct top{
        char key[32];
        struct top *v;    //viit vasakule alluvale või tühi
        struct top *p;   //viit paremale alluvale või tühi
        };
char key[32];            //sisendpuhver
//uue tipu tegija
struct top *new(char *k){
```

[1] Faili viit viitab struktuurile, kus on kirjas faili karakteristikud: puhvri asukoht, kirjutamise/lugemise järg, kas fail on avatud lugemiseks või tohib sinna ka kirjutada, jmt. (vt. [K&R], lk. 160). Kirjeldama peame alati „tühiviita“ väärtusega NULL; kui faili avamine õnnestub, siis NULL-väärtus kirjutatakse üle.

```
	struct top *t;
	t=malloc(sizeof(struct top));
	memset(t,'\0',sizeof (struct top));
	strcpy(t->key,k);
	return t;
	}

//kirjutab puu kettale (esimesel pöördumisel saab ette juure aadressi)
void wp(struct top *t){
	fwrite(t,sizeof(struct top),1,mf);
	if(t->v != NULL) wp(t->v);
	if(t->p != NULL) wp(t->p);
	}

//loeb puu kettalt, kirjutab vanad viidad tegelikega üle.
struct top  *rp( ){
	struct top *t;
	t=malloc(sizeof(struct top));     //mälueraldus loetavale tipule
	fread(t,sizeof(struct top),1,mf);   //tipp loetakse kettalt mällu
	if(t->v != NULL)t->v=rp( );
	if(t->p != NULL)t->p=rp( );
	return(t);
	}

//puu läbimine ja trükk „inorder v -> p“  (võtmete kasvav järjekord)
void inovp(struct top *t){
	if(t->v != NULL) inovp(t->v);
	printf("%s\n",t->key);
	if(t->p != NULL) inovp(t->p);
	}

// puu läbimine ja trükk „inorder p -> v“  (võtmete kahanev järjekord)
void inopv(struct top *t){
	if(t->p != NULL) inopv(t->p);
	printf("%s\n",t->key);
	if(t->v != NULL) inopv(t->v);
	}

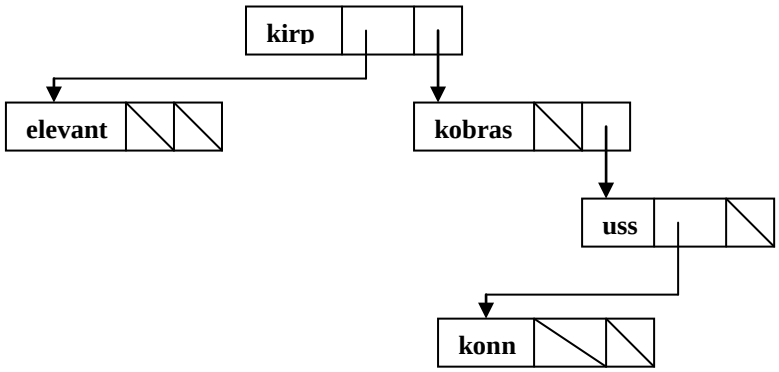
int main(int arc,char **arv){   //teine parameeter võiks olla ka nii kirjutatud: *argv[ ]){
	struct top *tipp;
	struct top *juur=NULL;
	struct top *jooksev;
	int r;
	for(r=0;r<arc;r++) printf("%s\n",arv[r]);    //trükin progr. nime ja puu-nime
	if(arc!=2){
```

```
        printf("command line must be spuu <puunimi>\n");
        return(1);
        }
    mf=fopen(arv[1],"rb");    //avan faili parameetrina antud nimega (binary, lugemiseks)
    if(mf==NULL)goto ring;  //faili kettal ei ole: mine „ring“
    juur=rp( );               //loen puu mällu
    inovp(juur);              //trükin võtmed kasvavas järjekorras
ring: printf("\nkey=");       //hakkan uusi võtmeid sisestama
    gets(key);
    if(strlen(key)==0) goto ots;   //“tühi Enter“
    tipp=new(key);            //teen uue „lahtise“ tipu
    jooksev=juur;             //hakkan talle puus kohta otsima
    if(juur!=NULL) goto otsi;   //kui puu on juba olemas, mine „otsi“
    juur=tipp;                 //uus tipp saab juureks
    goto ring;
otsi: r=strcmp(key,jooksev->key);
    if(r==0){
        //selle märgendiga tipp on juba puus
        printf("%s: v:%p p:%p\n",key,jooksev->v,jooksev->p);
        free(tipp); kustutan „lahtise“ tipu: võti on puus, ent tipp on tehtud
        goto ring;
        }
    if(r<0){
        if(jooksev->v != NULL){
            jooksev=jooksev->v;
            goto otsi;
            }
        jooksev->v=tipp;   //uus läheb vasakusse alampuusse ja pole enam „lahtine“
        goto ring;
        }
    if(r>0){
        if(jooksev->p != NULL){
            jooksev=jooksev->p;
            goto otsi;
            }
        jooksev->p=tipp;  //seni „lahtine“ uus tipp -> parem alampuu
        goto ring;
        }
ots:  remove(arv[1]);    //kustutan kettalt „vana puu“ (kui ta seal oli)[1]
    mf=fopen(arv[1],"wb");  //avan kirjutamiseks samanimelise „uue“ binary puu-faili
    wp(juur);   //kirjutan puu mälust kettale
    fflush(mf);   //väljundpuhvri „jõuga“ tühjendamine
    fclose(mf);
```

[1] „Vana puu“ on (kui ta kettal oli) juba mälus, ja antud seansi lõpus kirjutatakse ta (koos lisatud tippudega) välja.

```
p:      printf("\nkasvav jrk:\n");    //puu võtmete väljatrükid
        inovp(juur);
        printf("\nkahanev jrk:\n");
        inopv(juur);
}
```

Paar lehekülge tagasi esitatud *spuu* lahendamise pildilt võib aru saada, et võtmed esitati järjekorras *kirp, elevant, kobras, uss* ja *konn.* Puu, mille ehitas meie programm *spuu*, näeb meile vastuvõetaval kujul seansi lõpus välja järgmine:



Joonis 10.b. Otsimis- ja järjestamispuu.

Niisiis, alustasime kahendpuu-temaatikat kohe programmiga – teine variant olnuks alustada kahendpuu läbimise viisidega ja viidastruktuuride kirjutamisega kettale ja nende lugemisega kettalt. Me kõhklesime üsna kaua, kuidas toimida, ja arvasime lõpuks, et valitud sissejuhatus on ehk pisut intrigeerivam.

Ent mõningate mõistete, põhimõtete ja käitumiseeskirjade käsitlemisest me ükskõik kumma variandi puhul ei pääse. Alustagem sellest, et kahendpuu läbimise (=töötlemise) jaoks on võimalikud kolm rekursiivset algoritmi; kõik nad alustavad puu *juurest*:

- eesjärjekord (juur → vasak → parem, või juur → parem → vasak), Ingl. k.[1] *preorder*;
- keskjärjekord (*inorder*, vasak → juur → parem, või parem → juur → vasak);
- lõppjärjekord (*postorder*, vasak → parem → juur või parem → vasak → juur).

Näiteks, kui me läbime oma *spuu*-kahendpuu noil viisidel ja igas alampuu juurtipus[2] väljastame tipu märgendi, saame järgmised jadad:

- eesjärjekord: kirp, elevant, kobras, uss, konn (või kirp, kobras, uss, konn, elevant);
- keskjärjekord: elevant, kirp, kobras, konn, uss (või uss, konn, kobras, kirp, elevant);
- lõppjärjekord: elevant, konn, uss, kobras, kirp (või konn, uss, kobras, elevant, kirp).

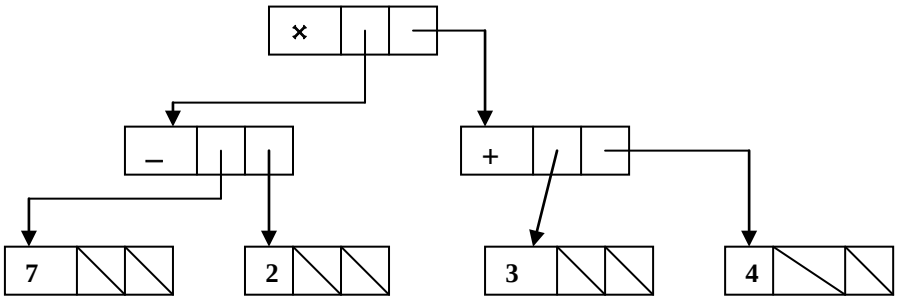
Kahendpuu läbimise viiside ingliskeelsed nimetused on vahetult seotud *avaldiste* esitusviiside ja nende nimetustega; neid viise on kolm. Kasutame näideteks aritmeetilisi avaldisi, ehkki sa-

[1] Kirjanduses kohtume sageli teistmoodi terminoloogiaga: *preorder, postorder ja endorder*. Pisut tagapool põhjendame, miks me eelistame skeemi *preorder*, *inorder* ja *postorder*.
[2] Puu *leht*, *rippuv* (alluvateta) *tipp* on *tühja alampuu juur*.

mahästi sobiksid ka loogilised avaldised või aritmeetiliste ja loogiliste avaldiste kombinatsioonid:

- *prefiks*-kuju: +ab; (kõik tehted on funktsioonid ning operandid on nende argumendid);
- *infiks-kuju*: a+b;
- *postfiks*-kuju: ab+ (nii on see magasinorienteeritud keeltes[1] nagu *Forth*, vt [Isotamm PK], lk. 139 jj.).

Edasi, kooliaritmeetika tuginedes me teame, mis järjekorras tuleb arvutada (näiteks aritmeetilist) sulgavaldist. Me võime sellise avaldise üles joonistada *kahendpuu* kujul: juure märgendiks paneme *viimasena* sooritatava tehte märgi[2] ning juure vasaku alampuu ehitame tollest märgist vasemale jäävast avaldise osast ning parema alampuu paremale jäävast osast, kasutades rekursiivselt sama loogikat. Näiteks, konstantavaldise (7–2)×(3+4) kahendpuu on järgmine[3]:



Joonis 10.c. Avaldise (7 – 2)×(3+4) kahendpuu.

Läbime äsjatehtud kahendpuu kolmel viisil (sh. sümmeetriliselt), kirjutades endiselt igas „jooksvas juurtipus“ välja ta märgendi:

- eesjärjekord: × – 7 2 + 3 4 (või : × + 4 3 – 2 7);
- keskjärjekord: 7 – 2 × 3 + 4 (või 4 + 3 × 2 – 7);
- lõppjärjekord: 7 2 – 3 4 + × (või 4 3 + 2 7 – ×).

Näeme, et *preorder* andis avaldise *prefiks-*, *inorder – infiks-* ja *postorder postfiks*-kuju. Kõikidel variantidel on programmeerija jaoks olemas kasulikud rakendused (mõned esinevad ka meie äsjatoodud programmis *spuu.c*). Toome mõned näited.

Eesjärjekord tundub kõige loomulikumana kahendpuu *kirjutamiseks kettale*: alustame puu juurest ning kirjutame rekursiooni kasutades kettale esmalt ta vasaku alampuu ja seejärel parema. Kettale kirjutatud *viidad* evivad vaid *lipu* tähendust: kui viit on *NULL*, siis alampuud kettal pole, muidu aga teame, mis järjekorras ta kettale kirjutati ning oskame uute viitadega puu täita just sellise infoga (k.a. alampuud), nagu tal enne kettale kirjutamist operatiivmälus oli. Mõistagi loeme puu kettalt sama *eesjärjekorda* kasutades.

[1] Need on orienteeritud *LIFO-stack*ile.
[2] Kui võimalusi on rohkem kui üks, siis valime neist suvalise.
[3] juhime tähelepanu seigale, et mitterippuvate tippude märgendid on tehtemärgid ja rippuvate tippude (lehtede) omad on operandid

Keskjärjekord teenis meie näiteprogrammis puu järjestatult väljastamise huve: valemiga $v \rightarrow j \rightarrow p$ saame väljatrüki võtmeväärtuste kasvavas ja $p \rightarrow j \rightarrow v$ – kahanevas järjekorras.

Lõppjärjekorra kasutamist oleme juba tutvustanud *LIFO*-tüüpi magasini käsitledes. Seal tutvusime *Dijkstra* algoritmiga, mis viib sulgavaldise inverteeritud Poola kujule, siin nägime, et see on triviaalselt tehtav, kui meil on avaldise kahendpuu – ent viimase tegemine *ei ole* triviaalne[1]. Ent kui meil on käes avaldise inverteeritud Poola kuju, siis *LIFO*-magasini abil selle interpreteerimine leidmaks avaldise väärtust on elementaarne.

„Normaalses" olukorras, kus puu sügavus (tasete arv) on „mõistlik" ja operatiivmälu on „mõistliku" mahuga (jätame jutumärkides esitatu lahti seletamata)) on puu läbimine suvalisel moel ülilihtne programmeerida *rekursiivselt.* Näiteks nii (kõik kolm moodulit töötavad „$v \rightarrow p$"- variandis ning ainus tegevus on tipu märgendi trükkimine):

```
void preorder(struct tipp *t){
        printf("%c ",t->l);
        if(t->v != NULL) preorder(t->v);
        if(t->p != NULL) preorder(t->p);
        }

void inorder(struct tipp *t){
        if(t->v != NULL) inorder(t->v);
        printf("%c ",t->l);
        if(t->p != NULL) inorder(t->p);
        }

void postorder(struct tipp *t){
        if(t->v != NULL) postorder(t->v);
        if(t->p != NULL) postorder(t->p);
        printf("%c ",t->l);
        }
```

Näeme, et „tegevuse" koht liigub „ülevalt alla", esmalt *pre,* siis *in* ja lõpuks *post*: trükkimise „koht" on vastavalt 1., 2. ja 3. (viimane) operaator.

Kõik ülaltoodud kolm algoritmi on *rekursiivsed,* neid on lihtne programmeerida ja üldjuhul töötavad nad vastuvõetava ajaga. Ent rekursiooniga kaasneb üsna palju mahukat ja kasutaja eest varjatud tööd, ükskõik, kas seda tööd teeb operatsioonisüsteem või keele (meie jaoks, konkreetse *C*-kompilaatori loodud) *virtuaalmasin* (vt. näit. [Isotamm PK], lk. 75 jm.). Iga rekursiivse pöördumisega kaasnev operatsioonide hulk on üsna suur: säilituspiirkonnale dünaamilise mälu eraldamine, säilituspiirkonda registrite jooksvate seisude ja lokaalsete muutujate väärtuste salvestamine, naasmisaadressi salvestamine ning säilituspiirkondi siduvate viitade loomine ja salvestamine säilituspiirkonda. Rekursioonisammult väljumine tegeleb samade asjadega peegelpildis[2]: registrite taastamine, säilituspiirkonna mälu vabastamine jne. Niisiis,

[1] Programmi teksti teisendamiseks *analüüsi puuks* on olemas lineaarse keerukusega ($O(n)$, n on programmi *lekseemide* arv) algoritmid, ent nende „käsitsi" programmeerimine pole kuigi lihtne. Vastavaid tehnikaid tutvustatakse loengukursuse „Automaadid, keeled ja translaatorid (MTAT.05.085)" raames.

[2] Rekursioonist väljudes tehakse sama arv tehteid nagu sinna minnes.

rekursioon on programmeeritult lihtne ja läbipaistev, ent leidub juhte, kus me ei saa seda kasutada ning peame (kahend)puud läbima teisiti, näiteks *LIFO*-tüüpi magasini abil. *Ees-* ja *keskjärjekorras* selle töö tegemine on triviaalne, ent *lõppjärjekorras* pole: selle jaoks me ei tea head algoritmi, mis töötaks ühetraktilise magasiniga. Üks võimalik variant on järgmine:

```
/* order.c : kahendpuu läbimine magasini abil 18.03.00 . Kasutame „vana terminoloogiat“:
inorderi asemel „postorder“ ja postorderi asemel „endorder“[1]  */
#include <stdio.h>
#include <stdlib.h>

#define n 4

struct tipp{
        char label;         /* märgend    */
        struct tipp *v;     /* vasak viit */
        struct tipp *p;     /* parem viit */
        };

/* kahetraktiline magasin: mõlemat vektorit indekseerime sama indeksiga */
struct tipp *M1[n];         /* tippude meelespidamiseks */
int M2[n];                  /* parem lipp. 1, kui parem haru on läbitud (endorder) */

/* puu  „programmeerime sisse“  */
struct tipp c={'c', (struct tipp *)NULL, (struct tipp *)NULL};
struct tipp e={'e',(struct tipp *)NULL,(struct tipp *)NULL};
struct tipp f={'f',(struct tipp *)NULL,(struct tipp *)NULL};
struct tipp b={'b',&c,(struct tipp *)NULL};
struct tipp d={'d',&e,&f};
struct tipp a={'a',&b,&d};


/*                  PUU

                                    a
                                   /   \
                                  b     d
                                 /     /  \
                                c     e    f

*/
```

[1] Vt. toimetaja märkust [Knuth I, lk. 395]: *preorder* (прямой порядок), *postorder* (обратный порядок) ja *endorder* (концевой порядок).

```
/* magasini nullimine */
void null_stack(void){
        int i;
        for(i=0;i<n;i++){
                M1[i]=(struct tipp *)NULL;
                M2[i]=0;
                }
}

/* kahendpuu läbimine lõppjärjekorras  c b e f d a. Ette juur, v->p->j */
int endorder(struct tipp *t){
        int i;                  /* magasiniindeks */
        struct tipp *aken;   /* aktiivne tipp */
        null_stack( );
        printf("\nendorder:  ");
        aken=t;
        i=0;
vasak: M1[i]=aken;
        if(aken->v!=(struct tipp *)NULL){
                aken=aken->v;
                i++;
                goto vasak;
                }
        if(aken->p!=(struct tipp *)NULL){
                M2[i]=1;
                aken=aken->p;
                i++;
                goto vasak;
                }
/* tipumärgendi trükk */
labor: printf("%c ",aken->label);
        M2[i]=0;   /* magasini abil tagasi */
        i--;
        if(i<0) return(1);
        aken=M1[i];
        if(M2[i]==0){
                M2[i]=1;
                if(aken->p!=(struct tipp *)NULL){
                        aken=aken->p;
                        i++;
                        goto vasak;
                        }
                }
        goto labor;
}
```

```
/* kahendpuu läbimine eesjärjekorras  a b c d e f. Ette juur, j->v->p */
int preorder(struct tipp *t){
        int i;    /* magasiniindeks */
        struct tipp *aken;   /* aktiivne tipp */
/* magasini nullimine */
        null_stack( );
        aken=t;
        i=0;
        printf("\npreorder:  ");
parem: if(aken->p!=(struct tipp *)NULL) M1[i]=aken->p;
        printf("%c ",aken->label);
        if(aken->v!=(struct tipp *)NULL){
                aken=aken->v;
                i++;
                goto parem;
                }
mag: if(M1[i]==(struct tipp *)NULL){
                i--;
                if(i<0)  return(1);
                goto mag;
                }
        aken=M1[i];
        M1[i]=(struct tipp *)NULL;
        goto parem;
}

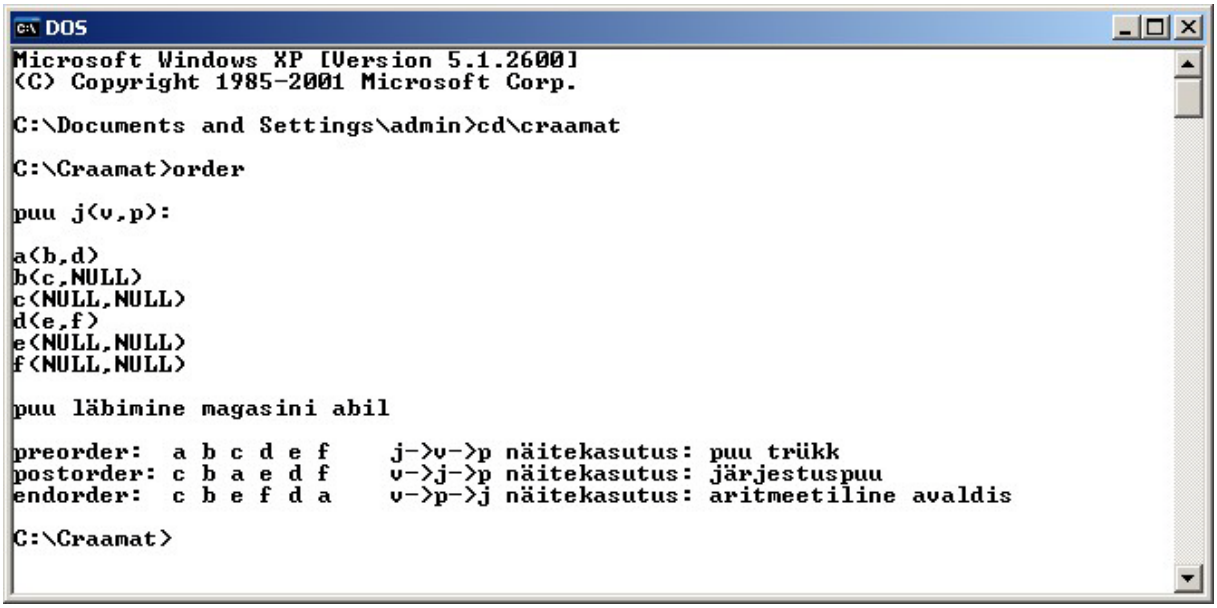
/* kahendpuu läbimine keskjärjekorras c b a e d f. Ette juur. v->j->p */
int postorder(struct tipp *t){
        int i;    /* magasiniindeks */
        struct tipp *aken;   /* aktiivne tipp */
/* magasini nullimine */
        null_stack( );
        aken=t;
        i=0;
        printf("\npostorder: ");
ws:     if(aken->v!=(struct tipp *)NULL){     /* write stack */
        M1[i]=aken;
        aken=aken->v;
        i++;
        goto ws;
        }
labor:  printf("%c ",aken->label);
        if(aken->p!=(struct tipp *)NULL){
                aken=aken->p;
```

```
                goto ws;
                }
mag: if(M1[i]==(struct tipp *)NULL){
                i--;
                if(i<0)  return(1);
                M1[i+1]=(struct tipp *)NULL;
                goto mag;
                }
        aken=M1[i];
        M1[i]=(struct tipp *)NULL;
        goto labor;
}

/* rekursiivne puutrükkal. Eesjärjekord j->v->p */
void p_tree(struct tipp *t){
        struct tipp *p;
        if(t!=(struct tipp *)NULL){
                printf("%c(",t->label);
                if(t->v!=(struct tipp *)NULL){
                        p=t->v;
                        printf("%c,",p->label);
                        }
                else printf("NULL,");
                if(t->p!=(struct tipp *)NULL){
                        p=t->p;
                        printf("%c)\n",p->label);
                        }
                else printf("NULL)\n");
        p_tree(t->v);
        p_tree(t->p);
        }
}

int main( ){
        printf("\npuu j(v,p):\n\n");
        p_tree(&a);      /* trükib puu eesjärjekorras (preorder) */
        printf("\npuu läbimine magasini abil\n");
        preorder (&a);  printf("   j->v->p näitekasutus: puu trükk");
        postorder(&a); printf("   v->j->p näitekasutus: järjestuspuu");
        endorder(&a);  printf("   v->p->j näitekasutus: aritmeetiline avaldis");
        printf("\n");
}
```

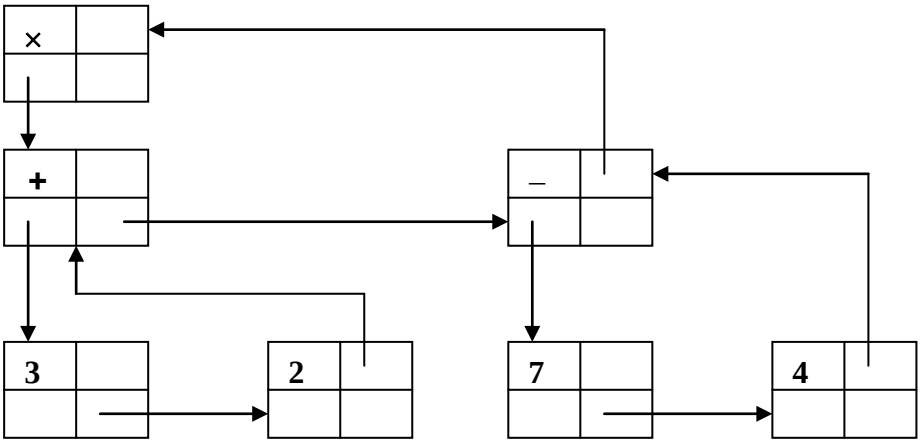
Seni oleme tutvunud kahendpuu läbimise rekursiivsete algoritmidega ning sellistega, mis kasutavad ühe- või kahetraktilist *LIFO*-tüüpi magasini. Allpool näitame, kuidas saab kahend-



Joonis 10.d. Programmi *order* lahendusprotokoll.

puud läbida lõppjärjekorras ilma rekursiooni ega magasinita; selleks kasutame raamatu [Isotamm, PK] lehekülgedel 227...229 esitatut.

Reeglina kasutavad masinast sõltumatud programmeerimiskeeled avaldiste *infiks*-kuju ja realiseerida[1] on triviaalne *postfiks*-kujul esitatud avaldist. Süntaksorienteeritud transleerimismeetodid võimaldavad ehitada süntaksi analüüsi käigus avaldise kahendpuu, mille läbimine lõppjärjekorras annab *inverteeritud Poola kuju* – ent sel juhul pole viimast vajagi, kuivõrd me võime avaldist interpreteerida juba puu läbimise käigus. Näitame, kuidas seda võiks teha; ka-



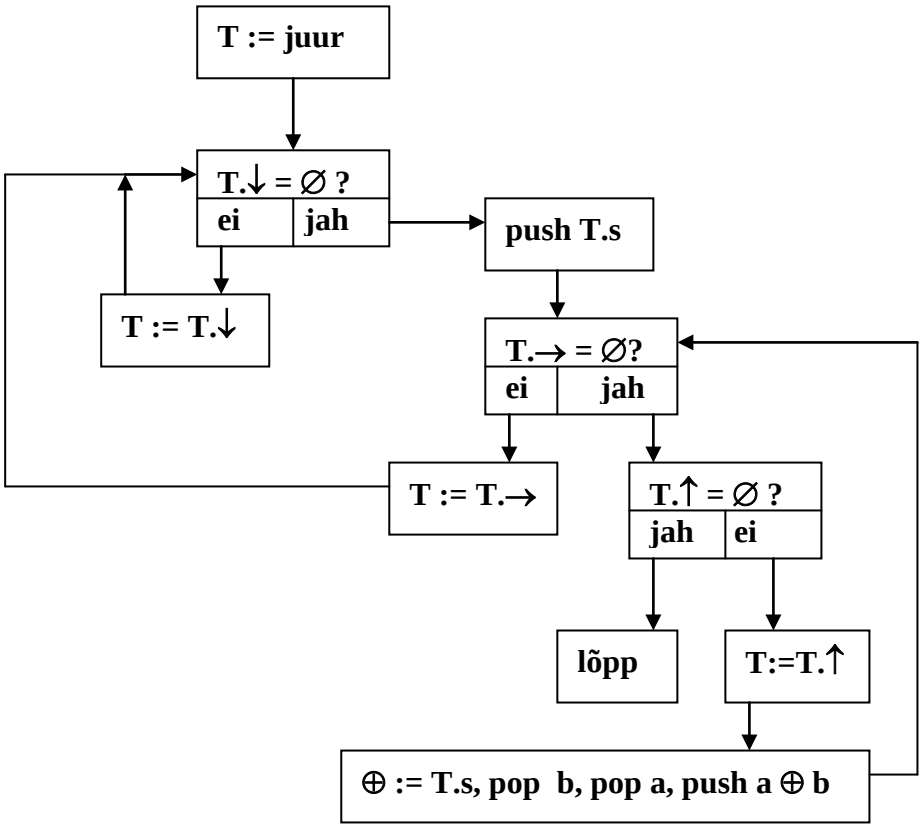
Joonis 10.e. Avaldise (3+2)×(7–4) „puu”.

sutame selleks avaldist (3+2)×(7–4), mille kahendpuu on joonisel 10.e ning puu ise on kujutatud graafina (vältimaks magasini kasutamist puu läbimisel)[2].

[1] Kas interpreteerida väärtuse leidmiseks või siis käskude genereerimiseks, so. kompileerimiseks

[2] Selle tehnika mõtles välja *Mati Tombak* 1970-ndate algupoolel (kui selle peale on varemgi tuldud, siis tuleks märkida, et *Tombak* tuli selle peale teistest sõltumatult). *Knuth*i raamatutes seda võimalust ei käsitleta.

Joonisel 10.f on esitatud plokkskeem aritmeetilise avaldise väärtuse leidmiseks avaldise kahendpuu läbimise käigus. Väärtuse arvutamiseks kasutatakse *LIFO*-tüüpi magasini; algoritmis me teda „nimepidi" ei kutsu, ent kasutame funktsioone *push* (pane magasini) ja *pop* (võta magasinist). „Aktiivse" tipu tähis on *T,* tipu-väljad on (vasakult paremale ja ülalt alla) T.s, T.↑, T.↓ ja T.→ ning jooksvad operandid on *a* ja *b.*



Joonis 10.f. Aritmeetilise avaldise väärtuse arvutamise algoritm.

Mõistagi, rangelt võttes pole meil õigust *ülesviidaga* tuunitud andmestruktuuri enam nimetada kahendpuuks, kuivõrd viimane on üheselt defineeritud kui *atsükliline*[1] *orienteeritud* [2] *graaf, mille ühestki tipust ei välju üle kahe kaare* (vt. näit. [Kiho 03], lk. 27 jj. või [Buldas jt.], lk. 31 jj.). Me võime omavolitsemist püüda vabandada nii, et kuulutame ülesviida *struktuurivälise*ks informatsiooniks, mida me pragmaatilistel kaalutlustel kasutame. Ja veel, „kolmeviidalises variandis" kasutasime – kui tippu kirjeldasime struktuuri *struct tipp*{...} abil – viitu *struct tipp *alamahel, struct tipp *naaber* ja *struct tipp *ylemus*.

Vahekokkuvõtteks: translaatoris kasutatav programmi puu[3] moodustub nii, et alampuude struktuur on ootuspärane *puu* struktuur, ent probleeme võib tekkida järjestamis- (ja otsimis)puude puhul: drastilisel juhul saame puu asemel *lihtahela*, nii juhtub näiteks, kui võtmeväärtused laekuvad *järjestatult.* Järjestatud *vektorist* otsimiseks on olemas kiire algoritm (*ka-*

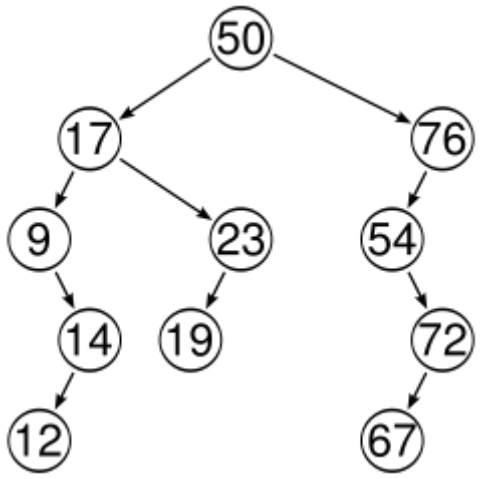
[1] Puus viitade abil liikudes on välistatud suvalisse lähtetippu tagasi jõudmine, teisisõnu: suvalisse tippu (va. juur) suundub parajasti üks kaar ja juurtippu ei suundu ainsatki.
[2] Viitade abil saame liikuda ainult ühes suunas: ülemuselt alluvale.
[3] Mille üks lihtne erijuht on *avaldise* puu.

hendotsimine), ent see pole rakendatav ahelate puhul. Ning suvalise võtme otsimiseks *n*-tipulisest puust kulutame ootuspärase keskmise $log_2(n)$ sammu asemel $n/2$ sammu – kui lootsime ehitada puud, ent saime lihtahela.

Reaalsete ja mahukate andmetöötlusülesannete[1] puhul kasutatakse „kontrollimatu“ kahendpuu asemel rafineeritumaid mooduseid, mis püüavad garanteerida otsimispuu dünaamilise (so, lahendamisaegse) tasakaalustatuse (jämedalt: igal tasemel on vasakus ja paremas alampuus umbes sama palju tippe). Õpikuisse on neist meetodeist jõudnud eeskätt *AVL*-puude[2] algoritm ja *B*-puude[3] sõnaline ja joonistega illustreeritud tutvustus (detailsed algoritmid on liiga mahukad).

Nende ridade kirjutamise ajal olime pisut keerulise valiku ees: kas maksta lõivu pealkirja „teisele poolele“ (*A&A*) või pidada prioriteetseks selle pealkirja esimest poolt (*C*). Valik oli raske, sest nonde meetodite algoritmide tutvustamine ja tutvustamise „saatmine“ *C*-programmidega (mida iseenesest polnuks raske teha, tsiteerides raamatut [Binstock&Rex], kokku lk. 278 – 361) kallutanuks *meie* raamatu liigselt *A&A*-valdkonda – ent sealt saame pelgalt näiteid, ei enamat. Liiati on noid meetodeid lakooniliselt, aga arusaadavalt kirjeldanud *Jüri Kiho* [Kiho 03, lk. 33 jj.].

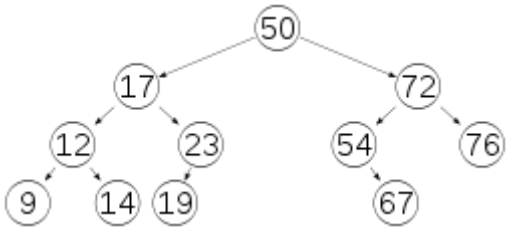
AVL-puu on kahendotsimise puu, mida moodustamise käigus „mängitakse“ programselt ümber nii, et „vasaku ja parema alampuu kõrgused erinevad ülimalt 1 võrra ja mõlemad on *AVL*-puud“ [Kiho 03, lk. 33]. Seejuures, algoritm pole üleliia komplitseeritud ega seda realiseeriv *C*-programm ei tuleks üleliia pikk.



Joonis 10.g. Tasakaalustamata otsimispuu.

B-puude esimene silmatorkav erinevus senikäsitletud otsimispuudega võrreldes seisneb selles, et nad *pole kahendpuud*, vaid nende mitterippuvates tippudes on võtmete muutuva pikkusega plokid. Neid kasutatakse laialdaselt info- ja andmebaasisüsteemides võimaldamaks kas kettalt

[1] Näiteks andmebaasi- ja infosüsteemid.
[2] *AVL* on akronüüm kahe vene programmeerija, *G. M. Adelson-Velski* ja *J. M. Landis*e nimede - - ; s. 8. tammikuuta (jaanuaril) 1922) on venäläinen matemaatikko ja tietojenkäsittelytieteilijä. Hän ja Jevgeni Landis keksivät AVL-puun vuonna 1962. Hän oli myöhemmin 1960-luvulla mukana kehittämässä ensimmäistä tietokoneshakkia“.
[3] *Balanced Tree*, vt. [Binstock&Rex, lk. 293 jj.]



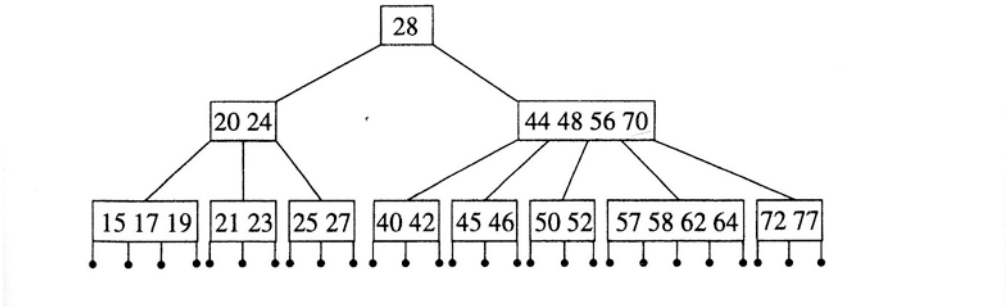
Joonis 10.h. Pärast joonise 10.g puu „ümbermängimist“ saadud *AVL*-puu [AVL].

või võrgust operatiivmällu tuua andmeid suurte plokkide kaupa (mis on olulise tähtsusega, kuivõrd väliskeskkonna poole pöördumine on ajaliselt kallis operatsioon).

B-puu autorid on *Rudolf Bayer* ja *Edward M. McCreight* (1962) ning kui nimetuse *B*-d tõlgendatakse üldiselt kui *Balanced* (tasakaalustatud), siis vahel ka kui *Bayer*i initsiaali. Ning seda andmestruktuuri kasutavat järjestamis- ja otsimismeetodit tutvustavast raamatust [Kiho 03] laename järgmise faksiimile-tsitaadi[1] (lk. 35):

m**-järku B-puuks** nimetatakse m-rajalist otsimispuud, mille korral
1. kõik lehed on samal tasemel ega sisalda kirjeid;
2. igal vahetipul peale juure on vähemalt $\lceil m/2 \rceil$ ja ülimalt m alluvat ning juure aste on vähemalt 2 ja ülimalt m.

Näide viiendat järku B-puu kohta on estitatud joonisel 2.6. B-puu lehed moodustavad fiktiivse taseme, edaspidi neid tühje lehti joonisel ei kujutata.



Joonis 10.i. *J. Kiho B*-puu näide.

Kuivõrd meie raamatu sihtrühm on Tartu Ülikooli Matemaatika-informaatikateaduskonna üliõpilased, siis neile soovitame (kui huvi tekkis) lugeda täiendavalt *Jüri Kiho* raamatut ja edasi näiteks raamatut [Binstock&Rex] tutvumaks puu moodustamisega, sh. lisamis- ja eemaldamisoperatsioonidega *C*-d kasutades. Viimases raamatus on vastavad algoritmid ja nende realisatsioon toodud lehekülgedel 293 – 359. Ent neile, kellel pole kas aega, võimalust või pragmaatilist huvi teemaga süvitsi minekuks, toome mõned kommentaarid *Kiho* tsitaadile.

Puu tipu *aste* on ta alluvate arv, puu *i*-nda tasemel on tipud, mille kaugus juurest on *i*, ja *m*-rajaline tähendab, et ühelegi tipule ei allu rohkem kui *m*-elemendiline plokk. Iga ploki piires on võtmed järjestatud kasvavas järjekorras[2], ja nagu äsjatoodud jooniselt nägime, on lõpptaseme võtmed (olenemata plokikuuluvusest) samamoodi järjestatud (15, 17, ...72, 77).

[1] Palume vabandust neilt, keda eksitab tsitaadilt loetav joonise number 2.6 või lubadus „edaspidi“. Neile soovitame lugeda *J. Kiho* originaalramatut.
[2] Infosüsteemides peavad kirjete võtmed olema (reeglina) unikaalsed.

11. Graafid

11.1. Sissejuhatus

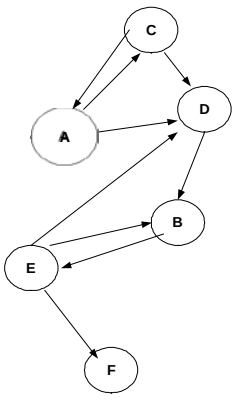
Kuivõrd oleme juba tutvunud *puu*-struktuuriga, siis juhatagemgi graafi-teema sisse, tuginedes meie *puu*-teadmistele. Puu on tippude hulk, millest igaühel võib olla kas 0 või rohkem alluvat, null või rohkem naabrit, ja ainult üks ülemus. Puu juur on ainus ilma ülemuseta tipp. Ja alluvateta tipud on puu *lehed.* Kui nimetame tippude-vahelisi *viitu* (ülemus-alluv, alluv-naaber, alluv-ülemus) *kaarteks*, ja lubame kahe tipu vahele rohkem kui ühe viida (viidad on näiteks ülemus-alluv, alluv-ülemus), siis tegelemegi andmestruktuuriga *graaf*. Kusjuures, lõppjärjekorras kahendpuu läbimise algoritme käsitledes märkisime, et meie üks „puu“ pole päris-puu (seal on tsüklid ja tegu on tegelikult *graaf*iga). Meenutagem, et kasutasime kolme viidaga tippe vältimaks puu läbimisel nii rekursiooni kui ka magasini. Niisiis, *graaf* on *tippudest* ja *kaartest* koosnev andmestruktuur, kus suvaline tipp võib olla kaartega vahetult seotud suvalise hulga teiste selle graafi tippudega. Erinevalt puust pole graafil erilist tippu – juurt – millest peaks alustama graafi läbimist (ehk töötlust), alguspunkti valik on vaba.

Graaf on matemaatikute jaoks huvipakkuv objekt. Meie ülikoolis on graafiteooriat õpeatnud näiteks *Mati Kilp* (matemaatikutele ja majandusüberneetikutele), matemaatilisest aspektist käsitleb graafe näiteks õppevahend [Buldas jt.] ja andmestruktuuride ning algoritmide aspektist *Jüri Kiho* [Kiho 03, lk. 93 jj.]. Viimativiidatud allika järgi on enimkäsitletud graafialgoritmid

- graafi tippude topoloogiline sorteerimine;
- lühimad teed graafis;
- graafi läbimine;
- minimaalne toes.

Kuivõrd graafide-alane kirjandus on meie üliõpilastele piisavalt kättesaadav ning seda teemat käsitletakse põhjalikult vastavateemalis(t)es erikursus(t)es, siis võime loodetavasti eriliste kahjudeta jätta ülalloetletud algoritmid käsitlemata; kompensatsiooniks vaatleme üht reaalset ülesannet, mille andmestruktuuriks on *graaf.* See ülesanne on lahendatud „alltöövõtu korras“ (koos *Mati Tombak*uga) kartograafia- ja IT-firmale *Regio* ja lahendus on rakendatud praktikas.

Graafi (kui abstraktse andmestruktuuri) kujutamiseks mälus on üldiselt tunnustatud kaht varianti: *maatriksesitust* ja *viidastruktuuri.*



Joonis 11.1.a. Orienteeritud graaf.

Ülaloleval joonisel (11.1.a) on kujutatud kuuetipuline graaf[1], tipumärgenditega *A..F*. Nooled tippude vahel tähistavad neid ühendavaid *kaari* ning noole teravik osutab ainuvõimalikule liikumissuunale. Allpool esitame sama graafi maatriksina (joonisel 11.1.b).

	A	B	C	D	E	F
A			1	1		
B					1	
C	1			1		
D		1				
E		1		1		1
F						

Joonis 11.1.b. Graafi maatriksesitus

Graafi kujutatakse maatriksi abil teoreetiliselt *Boole*'i maatriksina ($n \times n$ bitti), tegelikes rakendustes on ta tänapäeval (kui mälu maht pole enam esimene kitsendaja) baitmaatriks. Meie joonist tuleks lugeda nii: kui rea *X* veerus *Y* on väärtus 1, siis graafis on tee tipust *X* tippu *Y*, ja kui see väärtus on 0, siis pole. Näiteks, tipust *D* saame tippu *B* ja tipust *F* pole kuhugi minna; joonisel tähistab tühi lahter 0-väärtust.

11.2. Rahvaloenduse-ülesanne

AS Regio võitis riigihanke konkursi käesoleva sajandi alul teostatud riigi rahvaloenduse tarkvara väljatöötamiseks. Konkreetsemalt, Eestimaa oli jagatud valimisringkondadeks (Jõgeva, Tallinn-Mustamäe, Järvakandi vald jne) ning lähteandmeteks olid statistikaameti „oletused" ringkondade elanike arvu kohta Need olid üpris detailsed: iga ringkond oli jagatud (*Regio* siseterminoloogia kohaselt) *kvantideks* oletatava loendavate arvuga 0..368 inimest ning noist *kvantidest* tuli moodustada (taas *Regio* terminoloogiat kasutades) etteantud arv *klastreid* (igas oli hinnanguliselt mitte vähem kui 302 ja mitte rohkem kui 368 inimest) ning nood klastrid pidid olema *sidusad*: territoriaalselt polnud lubatud, et klaster koosneks kahest eraldipaiknevast „saarest". Klastrite arvu andis ette statistikaamet ning sellest tuli kinni pidada: elanike hinnangulise arvu järgi moodustati ringkonna loendamisjaoskonnad koos nende eelarvetega ning sinna mänguruumi ei jäetud. Ülesanne on lahendatav, kui kvantide kaalud[2] on võimalikult väikesed. Näiteks, kui ringkonnas on 4 kvanti ja iga kaal on 200, siis pole midagi teha.

Regio realiseeris peatöövõtjana ringkondade kartograafilise esituse, umbes nii, nagu allpool esitatud joonisel, kus kvandi numbriga on seotud eeldatav loendatavate arv („kaal", see on joonisel sulgudes), interaktiivse kvantide käsitsi-ühendamise, jaoskonna dokumentatsiooni (sh. kaardi) genereerimise; alltöövõtu korras pidime meie (siinkirjutaja ja *Mati Tombak*) kirjutama kvantide ühendamise programmi.

Teoreetiliselt on see programm triviaalne kombinatoorikaülesanne: tuleb leida kõik *m* kombinatsiooni *n* kvandi ühendamisest[3] nii, et *m* on etteantud klastrite arv ja et klastritele esitatud

[1] Kujutatud graaf on *orienteeritud* (e. *suunatud*). Teine variant on selline, kus me kujutame kaari noolte asemel joontega, ja joon tippude *X* ja *Y* vahel tähendab, et saame liikuda nii $X \to Y$ kui ka $Y \to X$.

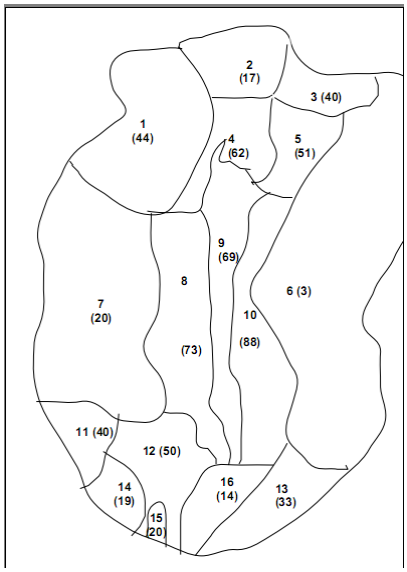
[2] Kaal on eeldatatav loendatavate arv.

[3] $C_n^m = n!/m!(n-m)!$ Ajaliselt eksponentsiaalseks hindas keerukust *Mati Tombak*.

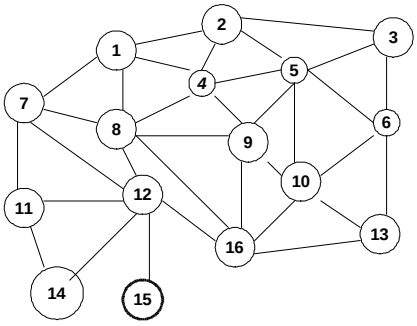
tingimused oleksid täidetud. Reaalselt oli kvante mõnekümnest kuni mitmesajani[1]; kõigi tingimusi rahuldavate kombinatsioonide leidmise ajalise keerukuse hinnang on *eksponentsiaalne*: c^n, kus c on rutiinsete sammude arvu väljendav konstant ning n on kvantide arv.

Meid kaasati tingimusega, et meie programm suudaks jaotamisülesande lahendada ülimalt viie minutiga (programm pandi tööle loomuldasa aeglase dialoogiprogrammi taustal[2]) ning allakirjutanu tolleagse 25-*MHz*-kohverarvutil ei kulunud meil ühegi *Regio* näite lahendamiseks üle mõne sekundi; empiiriliseks kiirushinnanguks kujunes $c \times n$ (ja ajaliseks keerukuseks $O(n)$).

Joonisel 11.2.a on ühe väljamõeldud loendusringkonna kvantide „kaart"; kvante on 16, kaaludega 3 kuni 88. Joonis 11.2.b kujutab kvantide kaardile vastavat orienteerimata graafi.



Joonis 11.2.a. Kvantide kaart.



Joonis 11.2.b. Kvantide graaf.

[1] Näiteks, Keila 380 kvanti tuli paigutada 30 klastrisse. Kombinatoorika-kiirushinnang on c^{380}.
[2] *Regio* väljamõeldud ja hästi toimiv süsteem oli selline: meie tegime käivitades kettale faili *klaster.tmp*, mille nimetasime ümber *klaster.prn*, kui ülesanne oli lahendatud. Põhiprogramm pidi seega perioodiliselt kontrollima, kas too .tmp-fail on veel kettal, kui ei ole, siis on alamülesanne lõpetanud. Ja veel, programmi projekteerimiseks, kirjutamiseks ja silumiseks oli meil aega 10 päeva.

Lähteandmed edastati *ASCII*-koodis tekstifailina, mille struktuur oli järgmine:

min = <min>;
max = <max>;
n = <kvantide arv>;
m = <klastrite arv>;
<kvandi nr.> ; <kaal> ; <naaberkvant1> , <naaberkvant2>...;

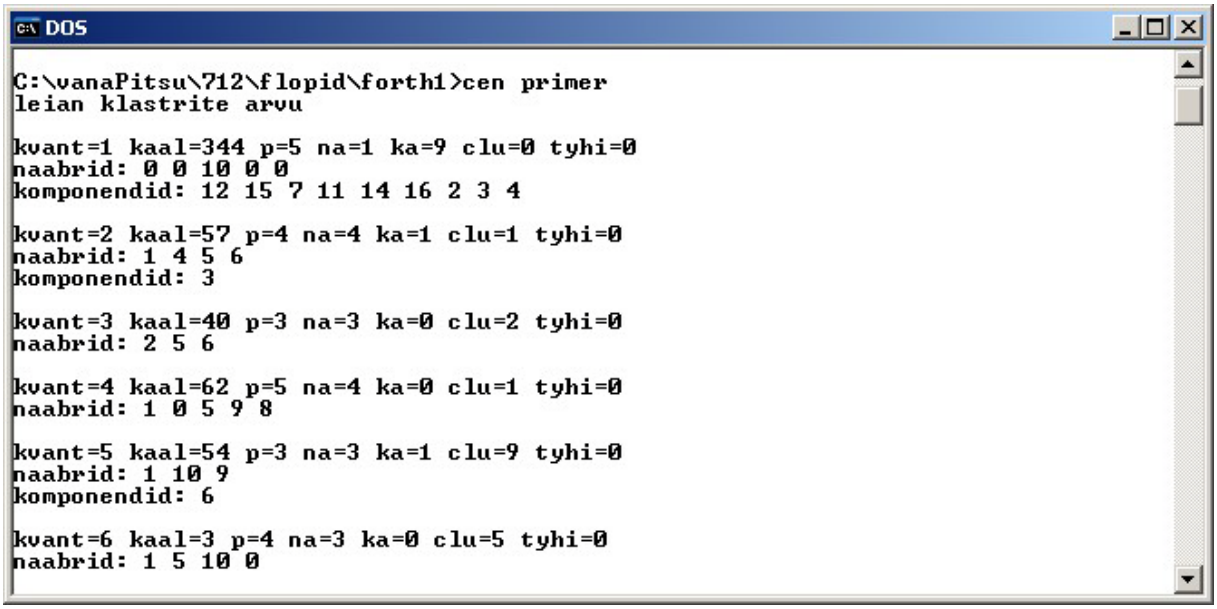
Kommentaariks: *min*=302 ja *max*=368 olid küll fikseeritud, ent võimaldasid mõistlikke erandeid. Näiteks, Ruhnu või Piirissaare[1] jaoks oli *min* ilmselgelt liiga suur, samas aga polnuks mõistlik neid ühendada kaugel asuvate naabritega, ja *max* näiteks Lasnamäe 370 eeldatava asukaga maja jaoks oli mõistlik samuti korrigeerida. Klastrite nõutavaks arvuks sai 2 lähtudes kaalude summast 661. Meie näite lähtefail (süsteemse nimega *sisend.prn*) on järgmine:

min=302;
max=368;
n=16;
m=2;
1;44;2,4,8,7;
2;17;1,4,5,3;
3;40;2,5,6;
4;62;1,2,5,9,8;
5;51;2,3,6,10,9,4;
6;3;3,5,10,13;
7;20;1,8,12,11;
8;73;1,4,9,12,7;
9;69;4,5,10,16,8;
10;88;5,6,13,16,9;
11;40;7,12,14;
12;50;11,7,8,16,15,14;
13;33;10,6,16;
14;19;11,12;
15;38;12;
16;14;12,9,10,13;

Lahendiks on tekstifail *klastrid.prn*: klastri number ja komponentide loetelu.

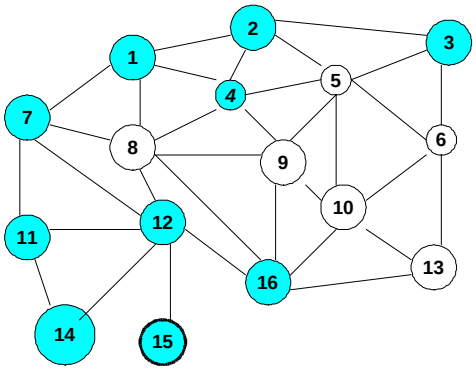
1;12,15,7,11,14,16,2,3,4,1;
2;13,9,8,5,6,10;

[1]Alalisi elanikke oli 2009. a. alguses Ruhnul 56 ja Piirissaarel 106. Ruhnlasi oli kõige rohkem 1842.a. – 389 (suurem kui meie *max*!). Allikas: *Internet.*



Joonis 11.2.c. Klastrite moodustamise programmi silumisaegne kontrolltrükk.

Kui liidame komponentide kaalud kokku, siis saame klastri 1 kaaluks *k* 344 ning klastril 2 – 317; mõlemad mahuvad nõutud piiresse ($302 \le k \le 368$). Demonstreerimaks meie lugejatele, et jaoskonnad on leitud tõepoolest programselt, tõime ülalasuval pildil ära ühe silumisetapi kontrolltrükifragmendi, mis ei aita küll mõista resultaadini jõudmist; sellest teeme põgusalt juttu tagapool.



Joonis 11.2.d. Klastrite kvandid.

Meie ülesande andmestruktuur on niisiis *graaf*, mille tipud on lähtekujul *kvandid* atribuutidega kvandi number, kaal, naabrite arv ja viit naabrite vektorile. Lisaks neile kirjeldati struktuuris *kvant* välju, mida modifitseeriti lahendamise ajal, kus mõned kvandid kvalifitseeriti klastriteks ja ülejäänud muutusid klastrite komponentideks. Struktuuri *kvant* kirjeldus on järgmine:

```
struct kvant{
        int kv;         /* kvandi number */
        int kaal;       /* elanike oletatav arv */
        char p;         /* naabrite vektori pikkus (konstant) */
```

```
    char na;       /* naaberkvantide arv (võib kahaneda klasterdamise käigus) */
    int *naabrid;   /* 1..na: naabrite numbrid */
    char ka;       /* komponentide arv; algväärtus=0: ka>0, kui kvant saab klastriks */
    int *komp;      /* 1..ka */
    int klaster;   /* klaster, kuhu kvant liideti. Klastril 0 */
    char tyhi;     /* 1, kui iseseisev klaster, 0, kui kustutatud */
    };
```

Teine oluline struktuur oli *parm*, mida me kasutasime oma moodulite tarbeks ning põhiprogrammile ei edastanud:

```
struct parm{
    double ka;    /* etteantud klastrite arv */
    int N;        /* kvantide arv */
    double KS;    /* kaalude summa */
    double min_ka;    /* klastrite min-arv = KS/k_max */
    double max_ka;    /* klastrite max-arv = KS/k_min */
    int max_p;        /* maksimaalne kvandi naabrite arv */
    int C;            /* viimase lahenduse klastrite arv */
    };
```

Programm lõpetab töö niipea, kui *C=ka*.

Niisiis, „klasterdamise“ käigus saab mõnest kvandist „klaster-kvant“, millega liidetakse naabreid: kvante või teisi naaberklastreid. Samas säilitab too klaster ka oma kvandi-tunnused: ümbermängimise käigus võib tema komponente või teda ennast viia naaberklastri koosseisu. Välja *tühi* väärtus on 1 siis, kui töötluse lõppedes on klaster-kvant alles; selliste arv on tolleks hetkeks *m* (etteantud klastrite arv). Väljundisse saadetud klastri viimane komponent on kvant, mille ümber moodustus klaster, komponentide vektoris **komp* teda pole. Meie äsjatoodud näites on esimese klastri „algkvant“ kvant nr. 1 ja teisel klastril 10.

Saavutamaks vastuvõetavat lahendusaega, pidime paratamatult oma ülesande jaoks kasutama *heuristikat*; sellest tehnikast on põgusalt kirjutatud lisas 8.

Meie kogemused näitavad, et heuristilist lahendust pole võimalik välja töötada tarkvaraarenduse läbiproovitud meetoditega – et on töörühm, sellel on ülemus ja asetäitjad, on „algoritmeerijad“ ja „kodeerijad“[1], on ülalt-alla liikuvad plaanid, algoritmid ja ajagraafikud ning altüles liikuvad aruanded ja seletuskirjad.

Esiteks, sedatüüpi algoritmi pole võimalik „projektülesande“ moodi välja töötada, algoritmi tegemine ja täiustamine toimub programmeerimise ja silumise käigus, tihti „katse-eksitus“-meetodil.

[1] Nende ridade autorile ei meeldi see termin. Professionaalne programmeerija pole pelgalt „kodeerija“ (loe: etteantud algoritmi madalatasemeline tõlkija etteantud programmeerimiskeelde), hea programmeerija tahab ette lähteandmete ning väljundi spetsifikatsioone ning lühiseletust, kuidas saab ülesande loogikast lähtuvalt sisendist väljundi. Ja kõrgtaseme keeles kirjutatud programmitekst pole „kood“, kood on ikkagi ehe masinkood (kompileeritud programm).

Teiseks, üksi on võimalik seda tööd teha, ent sootuks viljakam on kahe sama taseme kolleegi koostöö, mille käigus „kakeldakse läbi“ võimalikud jätkuvariandid, ja kui vaja (ja on vähegi aega), siis proovitakse neid kõiki. Ülemus-alluva suhted ja suurem meeskond on reeglina välistatud. Mõlemad osalised on nii „algoritmijad“ kui ka „programmeerijad“[1].

Kolmandaks, mentaalselt raskeim on selle „koha“ leidmine, millest alustada. See on ka punkt, millel on otsene seos meie ülesandega. Meie jaoks oli selge, et peame mingi (kombinatoorika-välise) plaani järgi hakkama kvante klastriteks ühendama, ja oli vaja otsustada, millisest kvandist alustada (mäletatavasti pole graafil „juurt“). Meie arvates oli sobiv variant alustada neist kvantidest, millel on minimaalne arv naabreid, näiteks, kui kvandil on ainult üks naaber, siis on ainus võimalus need kaks kvanti ühendada. Ja nii edasi: kahe naabriga kvandid tuleb emma-kumma naabriga ühendada varem kui kolme naabriga kvandid, ja nii edasi. Ehk – kui vaatame ka meie pilte – siis alustame „linna“ äärtest ning liigume „kesklinna“ suunas.

Me alustasime kahe tsükliga üle kvantide (alustades minimaalsest naabrite arvust), esimesel korral ühendasime kvante, kui nende kaalude summa ei olnud alla klastri minimaalkaalu (*lim*=302). Kui pärast seda tsüklit saavutasime klastrite nõutud arvu, oligi lahend käes. Kui aga ei, siis teisel korral tegime (juba ühendatud „pisiklastrite“ ja olemasolevate klastrite kvantide jaoks) uue tsükli (piirsummaga *lim*=368).

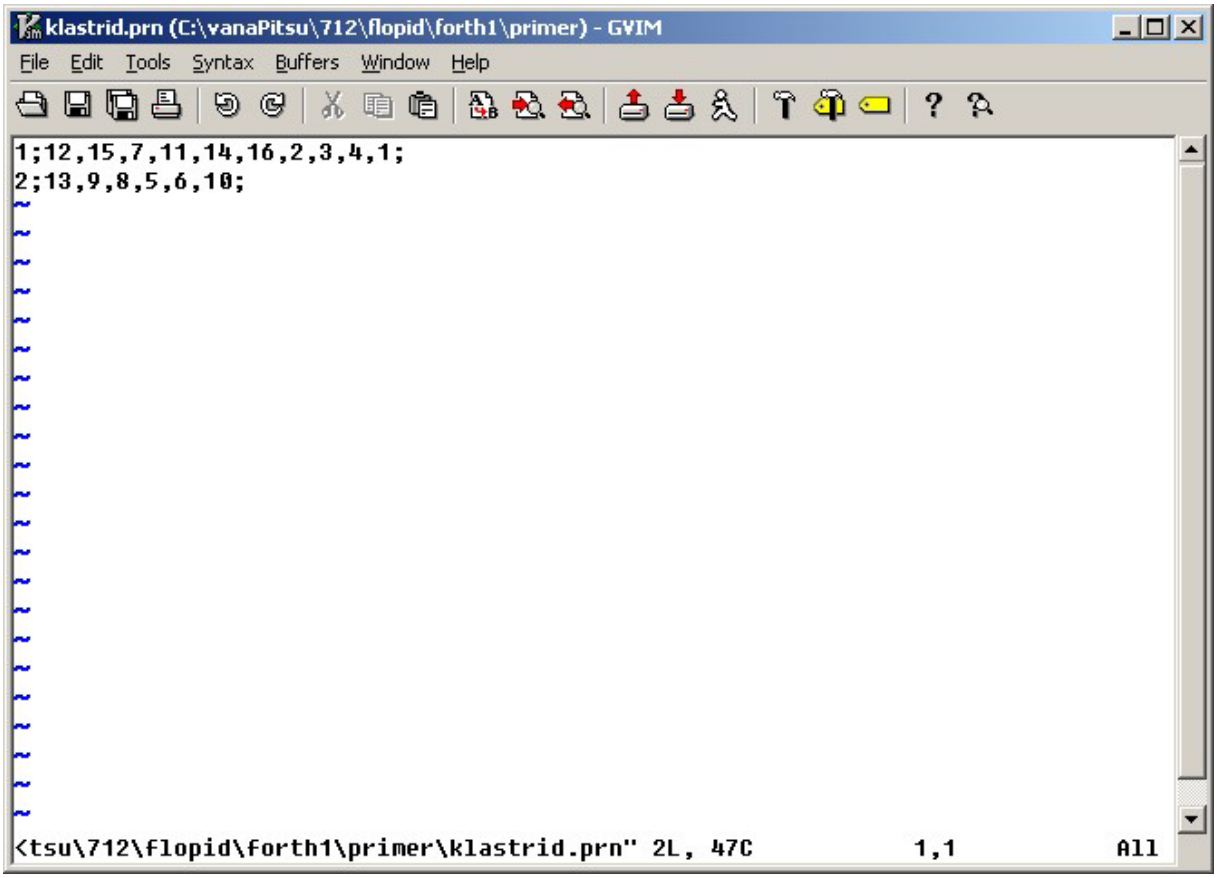
Kasutasime moodulit *join*(*i*,*lim*) ja mainitud tsüklid olid põhimõtteliselt sellised (*lim* on esimesel korral 302 ja teisel 368) ning *maxnab* on maksimaalne ühe kvandi naabrite arv antud graafis (see tuvastatakse kvantide graafi moodustamise ajal; meie näites on selle muutuja väärtus 6: nii palju naabreid on tippudel 5, 8 ja 12):

```
for(i=1;i<=maxnab;i++) join(i,lim);
```

Kui ka nüüd klastreid oli rohkem vajalikust, siis hakkasime jaotust ümber mängima: tõstsime „väiksematest“ (kaalu mõttes) klastritest[2] kvante piirnevatesse klastritesse üle, hoolitsedes, et viimased „lõhki“ ei läheks. Seejuures tuli jälgida, et programm ei jääks tsüklisse: ühel ringil tõstame kvandi klastrist *A* klastrisse *B*, järgmisel tõstame selle kvandi tagasi jne. Kui kõik sellised elementaarsed ümberjaotamised tulemuseni ei viinud, siis pandi nõutud klastrite arv „jõuga“ klappima, ühendades vajalik arv klastreid omavahel – jälgides et „uutel suurtel“ klastritel oleks võimalikult palju teistesse klastritesse kuuluvaid naaberkvante. Seejärel hakati „uusi suuri“ klastreid kvanthaaval väikeste naabrite vahel ära jagama, jälgides, et „doonor“ jääks nõutavatesse piiridesse ning „vastuvõtja“ saavutaks noisse piiridesse jääva kaalu. Vajadusel „aeti naaber lõhki“, et tasakaalustamine viia kaugemale. Algoritm toimis; paraku ei pea me otstarbekaks seda nende kaante vahel ära tuua, ehkki *Regiol* – nagu me 2009. a. veebruaris küsisime – poleks midagi selle vastu. Kogu siintutvustatud ülesande keskkond ja liides põhi- ja taustaprogrammi vahel on avaldatud samuti *Regio* loal.

Programmi *C*-teksti maht on 21,6 kilobaiti ning *Dev-C++* abil 2009. a. veebruaris kompileeritud[3] *.exe*-faili maht on 212 kilobaiti.

[1] *Andrei Jeršov* ütles omal ajal umbes nii, et „anda oma tehniline projekt realiseerimiseks võõrastesse kätesse on sama hea tegu, nagu lihase lapse andmine lastekodusse“, (*Jeršovist* loe näit. . [Isotamm, PK] lk. 199 jj.)
[2] Eesmärk oli väike klaster naabrite vahel ära jagada.
[3] 1999. aastal kasutasime *djgpp*-keskkonda.



Joonis 11.2.e. Näiteülesande resultaatfaili ekraanipilt.

Resümeerigem. Me lahendasime *eksponentsiaalse* keerukushinnanguga ülesande *lineaarse* kiirushinnanguga, ent *me ei töötanud välja algoritmi*: viimane on midagi üldisemat kui konkreetse ülesande lahendamiseks sobiv *programm. Algoritm* on alati üldine ja tõestatav. Ja kui heuristikat kasutades õnnestub leida *üldine algoritm*, siis pole see enam *heuristiline.*

12. Tabelid

Tabel (*table*, таблица) on kirjetest (*record*, запись) koosnev abstraktne andmestruktuur. *Kirje* omakorda on abstraktne andmestruktuur, mis koosneb kahest „loogilisest väljast“[1]: *kirje võtmest* ja võtmega seotud andmetest (informatsioonist). Juurdepääs tabeli kirjetele toimub eranditult võtmete abil ja reeglina on nõutud võtmete *unikaalsus*[2] nagu näiteks infosüsteemides, mis kasutavad relatsioonilist mudelit.

12.1. Võtmed

Tabel on ülejäänud abstraktsete andmestruktuuride hulgast kõige kasutatavam *andmebaasi-* (*resp. info*)süsteemide rakendustes, seda nii *hierarhilise, võrk-* kui ka (eriti) *relatsioonilise* ja *objektorienteeritud* andmemudeli puhul. Neis süsteemides on keskne mõiste *objekt* [3]. Refereerigem *Uno Merestet* [Mereste I][4].

Aspirant Ain Isotamme visioon oma teaduslikust juhendajast tema 55. sünnipäeval.

Joonis 12.1.a. Pilt *Uno Merestest* pärineb raamatust [Mereste 2, lk. 427]

Statistiline vaatlus on andmete hankimine uuritava objekti kohta. Objekt on nähtus – „ese“ (inimene, auto, pank jne) või protsess (liiklus, tootmine, raharinglus jne). Protsess on vaadeldav üksiknähtuste (sündmuste) kogumina. Üksiknähtust iseloomustab tema *kvaliteet* (omaduste e. tunnuste e. atribuutide komplekt) ja kvantiteet – tunnuste mõõdetavad (registreeritavad) väärtused. Vaatlusega pole võimalik hõlmata kõiki objekti tunnuseid, vaid ainult neid, mis on olulised vaatluse eesmärki silmas pidades. Tavaliselt kogutakse andmeid massnähtuste (nähtuste kogumite) kohta; üksiknähtusi käsitletakse kogumi liikmetena. Objekti iseloomustama valitud tunnuseid klassifitseeritakse järgmiselt;

- *Atributiivsed* (e. *kvalitatiivsed*) tunnused: sellise tunnuse väärtus iseloomustab objekti püsivat olekut (näit. inimese sugu või rahvus) või tinglikult püsivat olekut (näit. inimese nimi või aadress). Kvalitatiivse tunnuse väärtusega ei saa sisuliselt teha aritmeetilisi tehteid (kui tal on arvuline väärtus, on see formaalselt küll võimalik, ent sisutühi – näiteks, mida annab telefoninumbrite aritmeetiline keskmine).

[1] Loogiline väli võib olla mistahes moel struktureeritud.
[2] Tabelis ei tohi sel juhul olla kaht ühesuguse võtmega kirjet. Kui seda tingimust pole objektiivselt võimalik täita, siis moodustatakse sama võtmega, ent erineva semantikaga kirjetest omaette „alamandmestruktuur“, näiteks ahel.
[3] Laiemas, mitte pelgalt objektorienteeritud paradigma mõttes.
[4] Teeme refereeringu, korrates oma [Isotamm, AA&P, lk. 11 jj.] kunagist teksti.

Neid tunnuseid võime omakorda jaotada kahte klassi; *piiratud väärtusvaruga* tunnused, sh. *alternatiivsed tunnused* (*sugu* on kas mees või naine, *tõeväärtus* on kas tõene või väär) ja ülejäänud – need, mille *väärtusvaru võimsus* (võimalike väärtuste hulk) on suurem kui 2, ent on ette teada (näiteks Eestis kasutatavad auto- või telefoninumbrid) või need, mille väärtusvaru võimsus pole ei fikseeritud ega ka fikseeritav (näiteks, vastsündinutele pandavad nimed + senipandud nimed).

- *Kvantitatiivsed* tunnused, mille väärtused on mõõdetavad ja alati arvuliselt väljendatavad, näiteks inimese vanus, laste arv, kasv, kaal, pangaarve seis, kettaheite rekord jne. Reeglina pole need väärtused konstandid, vaid muutujad, ja nende kui operandidega sooritatud aritmeetilistel tehetel on tunnetuslik väärtus. Sedatüüpi tunnuseid liigitatakse omakorda alamklassidesse

  Sõredad (diskreetsed) tunnused võivad omandada vaid *täisarvulisi* väärtusi ja nad on kas olemuslikult sõredad nagu laste arv või tinglikult sõredad nagu inimese vanus: kombeks on seda registreerida teatud vanusest alates täisaastates. Edasi, *pidevad tunnused* võivad evida *oma väärtusvaru piires* mistahes reaalarvulisi väärtuseid (näiteks, kettaheite tulemuste teoreetiline väärtusvaru on meetrites 0.5 m kuni 74,05 m) ja ka pidevaid tunnuseid saame jagada kahte alamklassi: *olemuselt pidevad* on tavaliselt ajas muutuvad tunnused (nagu vanus) ja *tinglikult pidevad* (nagu keskmine laste arv või keskmine kiirus).

Ent naaskem *tabeli* juurde, lõpetades *Mereste* refereerimise. Kirje võtmeks saab olla ainult *kvalitatiivne* tunnus, so, tüüp või osiste kombinatsioon on kas *int* või *char*. Niisiis, välistatud on *reaalarvulised* võtmed; lisaks äsjatoodud sisulistele kaalutlustele (*real*-tüüpi atribuut on sisuldasa *muutuja*) tuleb silmas pidada, et *reaalarvuline väärtus* pole mingist täpsuspiirist alates ümardamata võrreldav: näiteks, sisestades *scanf*-funktsiooni abil arvu 0.99 ja jagades *a/b* (*a*=99.0 ja b=100.0) ei pruugi need arvud mingist komakohast alates kattuda.

Nii juurdepääs tabeli kirjetele kui ka kirjete töötlemine, näiteks, kirjete väljastamine järjestatult võtmeväärtuste kasvavas või kahanevas järjekorras toimub *võtmete* abil – ent loomulikult ei välista see printsiip tabeli töötlemist mingil muul moel. Näiteks, kui tabelis on maailma kõigi aegade 100 parimat kettaheitjat, siis on selle tabeli võtmeks loomulikult sportlase nimi (ja eesnimi), ent tabeli kasutajat võib huvitada nende kirjete järjestatus mitte nimede, vaid tagajärgede järgi: tabeli korrastamine pelgalt *võtme järgi* ei pruugi olla ainus võimalus; me (programmeerijad) võime teha *tabelile* suvalisi *pealisehitusi*[1]. Me käsitleme neid asju osas „multijuurdepääs“.

Võtmeteema lõpetame repliigiga: *universaalsed* (kitsendusteta) järjestusmeetodid ei piira võtme tüüpi. Järjestada suudavad need meetodid ka siis, kui võtme tüüp on *real*. Ent nagu selles jaotises püüdsime näidata, ei kvalifitseeru seda tüüpi atribuudid *kirje võtmeks*. Ja seega, kui meil on vaja järjestada *tabeli kirjeid* võtmeväärtuste järgi, siis me pole oma järjestusmeetodite valikus tingimata seotud ajalise keerukuse $O(n \times (\log n))$-hinnanguga, ning võime kasutada ka veelgi kiiremaid erivariante (mis reaalarvude järjestamiseks pole sobivad).

[1] Näiteks, teha spordiedetabeleid tulemuste paremusjärjekorras (jooksudes mingi tunnuse kasvavas ja muudel aladel kahanevas järjekorras), ent need tulemused on alati *kvantitatiivsete* tunnuste väärtused.

12.2. Tabeli füüsiline esitus

Tabeli füüsiline kujutamine arvuti mälus on täiesti reglamenteerimata, oluline on vaid see, et too esitus võimaldaks interpreteerida kirjeid kui tabeli elemente ning garanteerida neile juurdepääsu võtmete abil. *C*-keeles tundub loomulikuna *kirje* kujutamine *struktuurina* ning sel juhul saame *tabelit* kujutada kui vektorit, mille elementideks on viidad kirjetele, või kui ahelat, kus „info“ on viit kirjele, või kui puud, kus tipu info on viit kirjele. Või *n*-mõõtmelise massiivi või graafina, kus massiivi element või graafi tipu „info“ on viit kirjele.

Loomuldasa on tabel dünaamiline andmestruktuur, so. operatsioonideks tabeliga on kirje lisamine, kirje otsimine, kirje kustutamine (ja miks ka mitte kirjete väljastamine). Tabelite edasises käsitluses tähistame kirjete (jooksvat) arvu *n*-ga. Ja ehkki teisedki operatsioonid pole väheolulised, on tabeli tähtsaim karakteristik *otsimiskiirus*: sammude arv, millega saame tabelist kätte *k*-nda kirje (mille võti määrab kirje indeksiks tabelis *k*, $0 \le k \le n-1$) või tõdemuse, et selle võtmega kirjet tabelis pole. Ülesannet pisut lihtsustades käsitleme edasises tabeleid staatilistena, so, „valmis“ andmehulkadena.

Otsimiskiirus sõltub sellest, kuidas me (lihtsustatult – staatilise) tabeli enne kirjete otsima hakkamist korrastame ja mis otsimisalgoritme me tollest korrastatusest johtuvalt saame kasutada.

12.3. Läbivaadatav (korrastamata) tabel

Kui peame kinni oma naiivsest eeldusest, et me opereerime staatilise tabeliga, mis on moodustunud kirjete laekumise juhuslikus järjekorras, siis otsimise seisukohalt on meil esimeses lähenduses tegemist *korrastamata* e. *läbivaadatava* tabeliga: kui otsime kirjet võtmega *v* ja kui see on (juhuslikult) tabeli esimese kirje võti, siis saame kirje kätte 1 sammuga. Aga kui see *v* on tabeli viimase kirje (indeksiga $n-1$) võti, siis *n* sammuga, ehk kui me otsime tabelist (teades võtmete väärtusvaru) kõiki *n* võtit, siis peame tegema $\approx n^2/2$ sammu. Eelnevat meenutades: ajalise keerukuse hinnang kõikide võtmete otsimisel korrastamata tabelist on niisiis $O(n^2)$. Kirje otsimise (ja leidmise) puhul on olulised funktsioonid

- anna kirje võtme järgi: töötab tsükli puhul üle kogu tabeli kiirushinnanguga $n(n-1)/2$;
- anna esimene kirje: korrastamata tabeli puhul mõttetu;
- anna viimane kirje: samuti sisutühi;
- anna võtme suhtes eelmine kirje: sisutühi;
- anna võtmest järgmine kirje: sisutühi.

Esitame siinkohal programmi, mis loeb kirjed klaviatuurilt ning moodustab neist *tabeli*. Lihtsaim moodus on teha seda lihtahela abil. Näide on kahekordselt laenatud: „loomatabelitabeli“ originaali idee on pärit ühest *Toomas Mikli* (üks *SODI* põhitegijatest, 90-ndate esimese poole *TTÜ* professor) seminariesinemisest, mida me kasutasime oma raamatus [Isotamm AA&P, lk. 36]. See programm teeb *unikaalsete võtmetega* tabeli: kui otsimisvõtmega seotud kirje on juba tabelis, siis väljastatakse temaga seotud andmed (ja sammude arvu kirje leidmiseni, „s=..“), ja kui *ei ole*, siis lisatakse ahela lõppu (sinnani jõuame võtit otsides) uus lüli ning küsitakse dialoogi arendades kirje muude atribuutide väärtused. Ja seetõttu, et tabel on unikaalsete võtmetega, ei saa me uut kirjet lisada ei ahela algusse ega ka (kohe) lõppu, ning programmeerimise lihtsuse huvides on ahela pea rollis esimesena loodud lüli.

```
//ahel.c Sisestab ahelasse (tabel!) kirjeid. 16.01.09

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

FILE *mf=NULL;

struct loom{
      char key[32];   //kirje võti
      char nimi[10];  //võtmega seotud andmed
      char karv[10];
      char iseloom[10];
      };

struct lyli{
          struct loom *rec;
          struct lyli *next;
          };

char key[32];    //sisendpuhver

//teeb uue lüli. Ette saab võtme.
struct lyli *new(char *k){
   struct lyli *t;
   struct loom *e;
   t=malloc(sizeof(struct lyli));
   memset(t,'\0',sizeof(struct lyli));
   e=malloc(sizeof(struct loom));
   memset(e,'\0',sizeof(struct loom));
   t->rec=e;
   strcpy(e->key,k);
   printf(" nimi: ");
   fflush(stdin);  //puhastan sisendpuhvri ja küsin atribuutide väärtused
   gets(e->nimi);
   printf(" karv: ");
   gets(e->karv);
   printf(" iseloom: ");
   gets(e->iseloom);
   return t;
}

//kirjutab ahela kujul esitatud tabeli kettale
void wahel(struct lyli *t){
```

```
        fwrite(t,sizeof(struct lyli),1,mf);
        fwrite(t->rec,sizeof(struct loom),1,mf);
        if(t->next != NULL) wahel(t->next);
        }

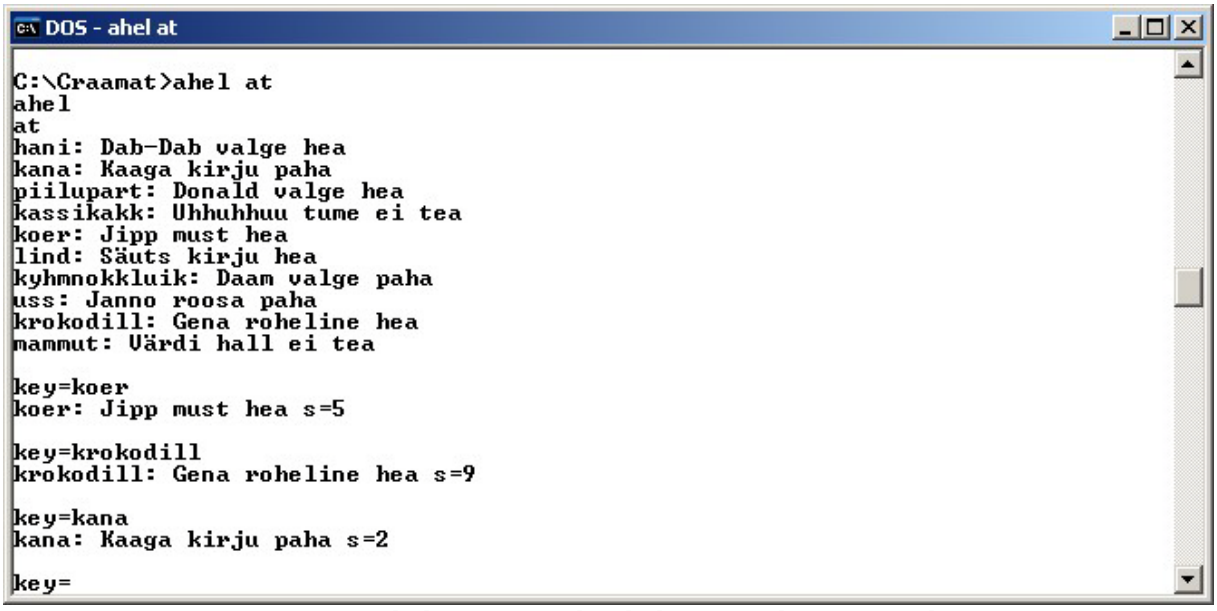
//loeb tabeli kettalt mällu
struct lyli *rahel( ){
        struct lyli *t;
        struct loom *e;
        t=malloc(sizeof(struct lyli));
        fread(t,sizeof(struct lyli),1,mf);
        e=malloc(sizeof(struct loom));
        t->rec=e;
        fread(t->rec,sizeof(struct loom),1,mf);
        if(t->next != NULL)t->next=rahel( );
        return(t);
        }

//väljastab aheldatud kirjed ekraanile
void pr_ahel(struct lyli *t){
   struct loom *e;
        e=t->rec;
        printf("%s: %s %s %s\n",e->key,e->nimi,e->karv,e->iseloom);
        if(t->next != NULL) pr_ahel(t->next);
}

int main(int arc,char *argv[ ]){
        struct lyli *tipp;
        struct loom *e;
        struct lyli *P=NULL;
        struct lyli *jooksev;
        int r, s;   //s=otsisammude arv võtme leidmiseni
        for(r=0;r<arc;r++) printf("%s\n",argv[r]);
        if(arc!=2){
             printf("command line must be ahel <tabeli nimi>\n");
             abort( );
             }
        mf=fopen(argv[1],"rb");  //avame tabeli „binary“-failina ainult lugemiseks
        if(mf==NULL) goto ring;
        P=rahel( );  //loeme „vana tabeli“ kettalt
        pr_ahel(P);
ring:   printf("\nkey=");
        gets(key);
        if(strlen(key)==0) goto ots;
        s=0;   //sammude arv võtme leidmiseni (s=1,2,...
```

```
        jooksev=P;
        if(P!=NULL) goto otsi;
        tipp=new(key);
        P=tipp;
        goto ring;
otsi: e=jooksev->rec;
      r=strcmp(key,e->key);
        if(r==0){               //leidsin!
                printf("%s: %s %s %s s=%d\n",key,e->nimi,e->karv,e->iseloom,s+1);
                goto ring;
                }
        if(jooksev->next != NULL){
                jooksev=jooksev->next;
                s++;
                goto otsi;
                }
        tipp=new(key);
        jooksev->next=tipp;
        goto ring;
ots:    pr_ahel(P);     //tabeli väljastamine kirjehaaval ekraanile
        remove(argv[1]);   //kustutan kettalt „vana tabeli“
        mf=fopen(argv[1],"wb");  //avan sama nimega uue, kirjutamiseks
        wahel(P);    //kirjutan uue tabeli kettale
        fflush(mf);
        fclose(mf);
}
```

Joonisel 12.3.a on selle programmi (*ahel*.c) ühe lahendamisseansi pilt (varem selle programmiga tehtud tabeli nimi on *at*).



Joonis 12.3.a. Korrastamata (ahelaga esitatud) tabeli manipuleerimise pilt.

12.4. Järjestatud (sorteeritud) tabel

Otsimiskiirus muutub drastiliselt, kui me järjestame kirjed võtmeväärtuste (tavaliselt kasvavas) järjekorras. Sel juhul on meil tegemist *järjestatud tabeliga* ning operatsioonid

- anna võtme järgi: töötab iga võtme jaoks keerukushinnanguga kuni $O(\log n)$;
- anna esimene kirje: annab;
- anna viimane kirje: annab;
- anna eelmine kirje (võtme abil leitud kirje suhtes): annab;
- anna võtme abil leitud kirjest järgmine kirje: annab.

Eelmine näiteprogramm (korrastamata tabel) illustreeris üksiti ka tabeli organiseerimist *ahela* abil. Järgmised kaks programmi kasutavad tabeli moodustamise teisi meetodeid, esimene neist teeb tabeli kahend(otsimis)puuna ja töötab nagu meie „ahela"-programm: kui võti on tabelis, siis väljastatakse kirje ja kui pole, siis lisatakse kirje (küsides atribuutide väärtusi). Teine meetod eeldab, et tabel on kujutatud kui võtmeväärtuste järgi järjestatud viitade vektor.

12.4.1. Tabel kui kirjete otsimispuu

Modifitseerigem peatüki 10 *Puud* alguses esitatud programmi *spuu.c* nii, et otsimispuu oleks moodustatud kui *tabel* (igas tipus on pelga *võtme* asemel viit *kirjele*):

```
//opuu.c Sisestab otsimispuusse (tabel!) kirjeid ja „otsib".  16.01.09

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

FILE *mf=NULL;
//tabeli kirje kirjeldus
struct loom{
     char key[32];   //võti: näiteks karu, ahv, vihmauss jne
     char nimi[10];  //see ja järgmised väljad on „võtmega seotud info"
     char karv[10];
     char iseloom[10];
     };

struct top{
        struct loom *rec;
        struct top *v;
        struct top *p;
        };

char key[32];

//teeb puu uue tipu
struct top *new(char *k){
```

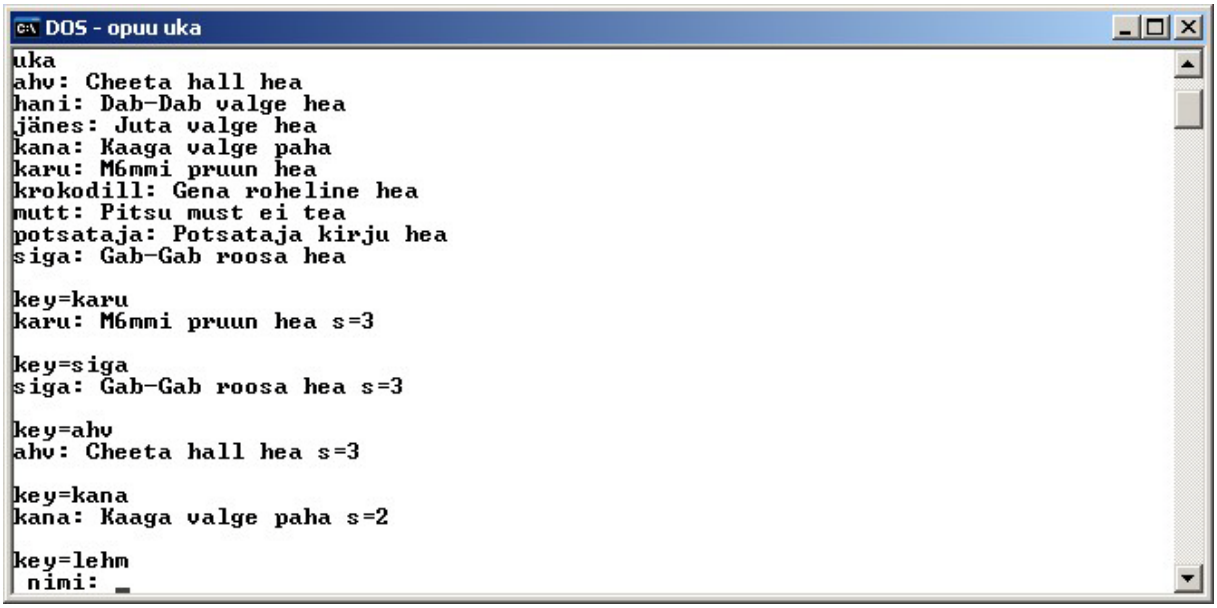
```
    struct top *t;
    struct loom *e;
    t=malloc(sizeof (struct top));
    memset(t,'\0',sizeof (struct top));
    e=malloc(sizeof (struct loom));
    memset(e,'\0',sizeof (struct loom));
    t->rec=e;
    strcpy(e->key,k);
    printf(" nimi: ");  //küsin atribuutide väärtusi
    gets(e->nimi);
    printf(" karv: ");
    gets(e->karv);
    printf(" iseloom: ");
    gets(e->iseloom);
    return t;
}
//kirjutab puu eesjärjekorras ja tipphaaval kettale
void wp(struct top *t){
        fwrite(t,sizeof (struct top),1,mf);
        fwrite(t->rec,sizeof (struct loom),1,mf);
        if(t->v != NULL) wp(t->v);
        if(t->p != NULL) wp(t->p);
        }

//loeb puu eesjärjekorras ja tipphaaval mällu
struct top  *rp( ){
        struct top *t;
        struct loom *e;
        t=malloc(sizeof (struct top));
        fread(t,sizeof (struct top),1,mf);
        e=malloc(sizeof (struct loom));
        t->rec=e;
        fread(t->rec,sizeof (struct loom),1,mf);
        if(t->v != NULL)t->v=rp( );
        if(t->p != NULL)t->p=rp( );
        return(t);
        }

void inovp(struct top *t){
    struct loom *e;
        if(t->v != NULL) inovp(t->v);
        e=t->rec;
    printf("%s: %s %s %s\n",e->key,e->nimi,e->karv,e->iseloom);
        if(t->p != NULL) inovp(t->p);
        }
```

```
int main(int arc,char *argv[ ]){
        struct top *tipp;
        struct loom *e;
        struct top *juur=NULL;
        struct top *jooksev;
        int r;
        for(r=0;r<arc;r++) printf("%s\n",argv[r]);  //lugejale meeldetuletamiseks
        if(arc!=2){
            printf("command line must be opuu <puunimi>\n");
            abort( );
            }
        mf=fopen(argv[1],"rb");  //“binary“-fail lugemiseks
        if(mf==NULL)  goto ring;  //seda faili pole veel tehtud, teeme nüüd
        juur=rp( );     //loeme faili kettalt ja
        inovp(juur);   //trükime välja
ring:   printf("\nkey=");
        gets(key);
        if(strlen(key)==0) goto ots;   //“tühi Enter“
        jooksev=juur;
        if(juur!=NULL) goto otsi;
        tipp=new(key);  //teeme seni tühja puu juure
         juur=tipp;
        goto ring;
otsi:
         e=jooksev->rec;
         r=strcmp(key,e->key);   //võrdlen otsimisvõtit jooksva tipu omaga
        if(r==0){
                printf("%s: %s %s %s\n",key,e->nimi,e->karv,e->iseloom);
                goto ring;
                }
        if(r<0){
                if(jooksev->v != NULL){
                        jooksev=jooksev->v;
                        goto otsi;
                        }
                tipp=new(key);
                 jooksev->v=tipp;
                goto ring;
                }
        if(r>0){
                if(jooksev->p != NULL){
                        jooksev=jooksev->p;
                        goto otsi;
```

```
                    }
                tipp=new(key);
                 jooksev->p=tipp;
                goto ring;
                }
ots:
        inovp(juur);    //trükin uue puu
         remove(argv[1]);  //kustutan lugemiseks avatud originaali kettalt
        mf=fopen(argv[1],"wb");
        wp(juur);               //ja kirjutan mälust kettale sama nimega uue puu
        fflush(mf);
        fclose(mf);
}
```



Joonis 12.4.1.a. Ekraanipilt programmi *opuu.c* dialoogist.

12.4.2. Tabel kui (võtmete järgi) järjestatud viidavektor

Kui järjestatud tabel on kujutatud viitade vektorina[1], siis saame kasutada *kahendotsimist.* All-pool kasutatud kahendotsimise *C*-algoritm pärineb raamatust [K&R, lk. 134].

```
//binsearch.c :: vt. K&R, lk. 134.   15.01.09.  Test on sisse programmeeritud.

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

[1] Kuidas sellist viidavektorit saada näiteks ahela abil moodustatud tabelist loodame näidata pisut hiljem.

```
struct loom{
      char key[32];
      char nimi[10];
      char karv[10];
      char iseloom[10];
      };

//NB! struktuuri initsialiseerimisel pole vaja ise hoolitseda kirjete eristamise eest
struct loom T[6]={
      "hiir","Piiks","hall","paha",
      "kana","Kaaga","kirju","paha",
      "koer","Jipp","must","hea",
      "mutt","Pitsu","must","ei tea",
      "part","Dab-Dab","valge","hea",
      "siga","Gab-Gab","roosa","hea",
      };
 int s;   //otsimissammude arv

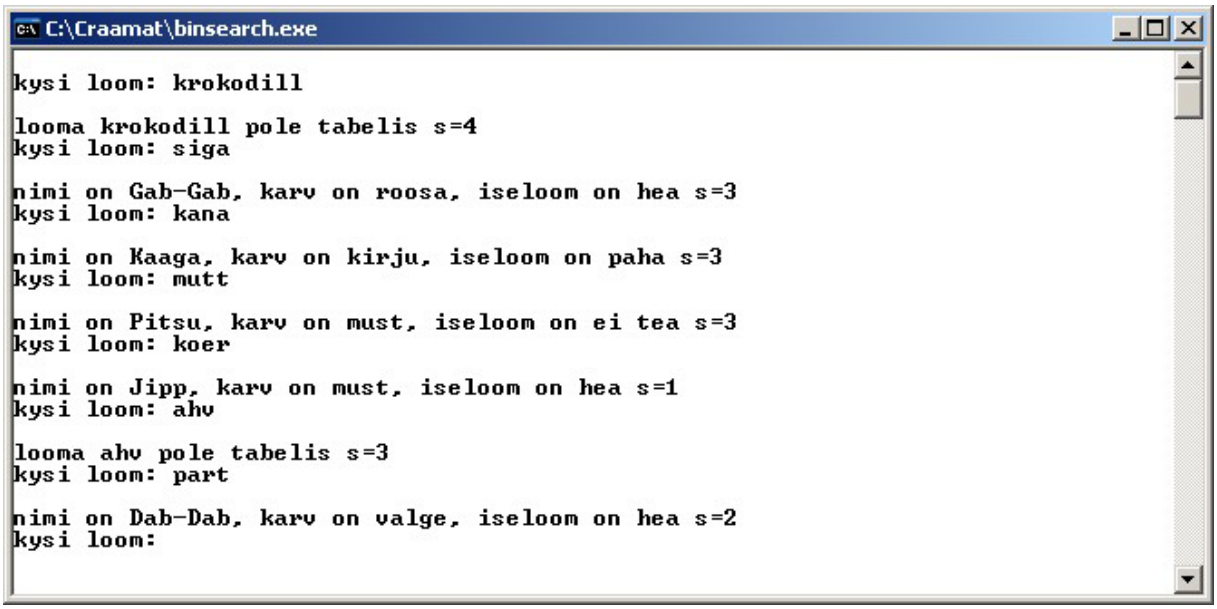
struct loom *binsearch(char *word, struct loom T[ ], int n){
   int cond;
   int low,high,mid;
   s=1;
   low=0;
   high=n-1;
   while(low<=high){
        mid=(low+high)/2;
        if((cond=strcmp(word,T[mid].key))<0){
                high=mid-1;  //NB! T[mid].key
                s++;
                }
        else if(cond>0){
                low=mid+1;
                s++;
                }
            else return T+mid;    //leitud! mõtle: T+mid ja mitte T[mid]
        }
   return NULL;    //pole tabelis
}

int main( ){
   char otsi[32];
   struct loom *kirje;

ring: printf("\nkysi loom: ");
     gets(otsi);
```

```
    if(strlen(otsi)==0) return 0;    //tyhi Enter: seansi lõpp
    kirje=binsearch(otsi,T,6);
    if(kirje==NULL) printf("\nlooma %s pole tabelis s=%d",otsi,s);
    else printf("\nnimi on %s, karv on %s, iseloom on %s s=%d",
            kirje->nimi,kirje->karv,kirje->iseloom,s);
    goto ring;
}
```

Järjestatud tabelist kahendotsimise *idee* on lihtne ja seda näitlikustatakse tihti sellega, mida me ise teeme, kui otsime meile võõra keele „Võõras – Eesti“-sõnaraamatust mingi sõna eestikeelset vastet (eeldades, et meil pole aimugi, kuidas jaotuvad sõnad algustähtede järgi, ent teades, et selle „võõrkeele“ rääkijad kasutavad ladina tähestikku ja järjestavad tähti nii nagu meiegi): lööme sõnaraamatu silma järgi keskelt lahti ja vaatame, kas otsitav sõna algab sõnaraamatus eespool või tagapool algava tähega (kui ta on avatud leheküljel, siis on ta käes ja lõpetame otsimise). Järgmisel sammul on sõnastik meie jaoks ainult see eelmise sammu pool, kus me otsimist jätkame (eespoolest või tagapoolest) – igal sammul kahaneb otsimisruum poole võrra. Suvalise võtme otsimise sammude arv on suurusjärgus $log_2(n)$.



Joonis 12.4.2.a. Kahendotsimine viitade vektorist (*s* on otsimissammude arv).

Niisiis, kui *tabel* on organiseeritud kui *kirjeviitade vektor* (nagu viimases näites) või kui *otsimispuu*, siis võtme järgi otsimise kiirushinnang on ligikaudu *kahendlogaritm* kirjete arvust. Samas, kui *staatiline* tabel moodustati kirjete laekumise juhuslikus (võtmete mõttes) järjekorras ja programmeerija valis tabeli füüsiliseks esituseks *ahela*, siis (kui tal õnnestuski see järjestada) pole kahendotsimine kasutatav[1].

[1] Ahela järjestamise otstarbekuse asjus avaldasime skepsist ka ülalpool.

12.5. Paisktabel

Järjestatud tabel on asendamatu andmete organiseeritud väljastamise puhul (igasugused edetabelid, sõnaraamatud, registrid jmt.). Ent kui ei kogu tabeli väljastamine ega ka võtme abil leitud kirje „naabrid“ pole päringus olulised, siis võtme järgi otsingu kiiruse logaritmhinnang võib osutuda liiga suureks. *Kui* meid huvitab vaid otsivõtmega kirje leidmine (või mitteleidmine) ja mitte järjestatud väljastamine, esimene, eelmine või järgmine või viimane kirje, siis on mõttekas teha tabel *paisktabelina*[1] (*hash-table*, хэш-таблица, slängis ka *häštabel*).

Paisksalvestuse idee on lihtne: tabeli pikkus on *n*, kirjete arv on *m* võtmetega *key*[1..*m*] ja kiire otsimise huvides on hea, kui $n>m$ (ilmselt on neil tingimustel tabel moodustatud kui kirjeviitade vektor). Kirje võti k_i $(0 \le i \le m-1)$ teisendatakse mingi funktsiooniga[2] $f(k_i, n)$ kirje paisktabeli-indeksiks a_i $(0 \le a_i \le n-1)$. Paraku pole teada ühtegi (võtmetele) oluliste kitsendusteta paiskfunktsiooni, mis välistaks *põrked* (*collision*), so., pole välistatud situatsioon, kus $a_i = a_j$, kui $k_i \ne k_j$. Põrke puhul kasutatakse *põrkefunktsiooni* $g(\ )$ ning paiskaadress a_i leitakse üldjuhul kui $a_i = f(k_i, n) + g(j, k_i, n)$, kus j on põrke likvideerimise samm ($j = 1,2,...$) ja põrkesituatsioonis käitumise järgi eristataksegi paisksalvestusmeetodeid; neid töötati mööda nud sajandi keskel välja märkimisväärne hulk.

12.5.1. Ajaloost

Paisksalvestuse ajaloo lühiülevaate teeme *D. Knuth*ile [Knuth III, lk. 642..643] tuginedes. Tema andmetel oli vististi esimene programmeerija, kes selle idee peale tuli, *Hans Peter Luhn, IBM*: 1953. aastal lasi ta oma firma sisekäibesse ettekande, kus käsitleti *välisaheldusega* paisksalvestust (*Knuth* nendib, et tõenäoliselt oli see ka esimene kord, kui kasutati *ahelaid*); samale mehele kuulub ka optimaalsete otsimis-kahendpuude idee (vt. [Knuth III], lk. 519 – 521). Umbes samal ajal tuli *IBM*is üks teine töörühm nn. *lahtise adresseerimise* meetodile[3].

Me käsitleme neid variante tagapool, ent siinkohal nentigem, et *välisaheldus* on üldjuhul kiireim meetod, ent pikka aega oli ta nii publikatsioonides kui ka kasutatavuselt tagaplaanil. Miks nii?

Põhjus peitub arvatavasti programmeerimiskeelte arengus. Kaugel 1953. aastal programmeeriti põhiliselt masinkoodis või paremal juhul assembleris ning dünaamilise mälujaotusega polnud mingeid probleeme, sellega tegelesid programmeerijad ise või kasutasid selleks sobivat varemloodud moodulit. Aasta hiljem tuli staatilise mälujaotusega *FORTRAN*, kus dünaamikat sai vaid imiteerida staatilise massiivi piires[4]. Pisut hiljem tuli algoritmide publitseerimiseks mõeldud *ALGOL*, millel on küll dünaamilised massiivid, ent puudub *viidatüüp* ja seega ei saa kasutaja viidastruktuure luua. Mainekad programmeerimisajakirjad, eeskätt *CACM*, hakkasid publitseerima algoritme ainult *ALGOL*is ning dünaamilised andmestruktuurid jäid paratamatult väljapoole teadusringkondade (publitseerimis)huvi.

---

[1] Paisksalvestuse eestikeelne terminoloogia on *Leo Võhandu* looming. Muuhulgas pakkus ta *display* venekeelseks vasteks sõna смотрило, ent kasutusele see siiski Venes ei tulnud (ehkki tekitas elevust).
[2] Seda nimetatakse *paiskfunktsiooniks* (*hash function*, функция расстановки, хэш- функция).
[3] Sel juhul likvideeritakse põrkeid tabeli *sees* (vt. näit. [Kiho, lk. 46 jj.]). Miks seda meetodit *lahtiseks* (*open addressing*, открытая адресация с линейным опробованием) nimetatakse, jääb siinkirjutaja jaoks lahtiseks.
[4] Vt. [Isotamm, PK] lk. 118).

Ent jätkakem *Knuth*iga[1]. Esimene avaldatud artikkel paisksalvestusest on pärit 1956. aasta detsembrist ning selle autor on *Arnold I. Dumey* [wDumey]. 1957. aastal avaldas asjakohase artikli *W. W. Peterson* [2]; mõlemad mehed on eelkõige tuntud kui krüptoloogid, *Dumey* ka kui kirju postiindeksite järgi sorteeriva seadme kaasautor. Vene üks tuntumaid programmeerijaid, *Andrei Petrovitš Jeršov* (tema pilti vt. näit. [Isotamm, PK] lk. 200) avaldas sõltumatult eeltoodud autoritest 1957. aastal artikli *lineaarsest paiskamisest*, kus ta näitas, et kui $m/n < 2/3$, siis keskmine otsisammude arv jääb alla kahe.

Werner Buchholz[3] publitseeris 1963. aastal esimese ülevaateartikli *paiskfunktsioonidest. Knuth* nendib, et ehkki 1968. aastaks oli paisksalvestus saanud üldkasutatavaks, oli seda tutvustavaid või midagi uut lisavaid publikatsioone üsna vähe. Mainitud 1968. aastal publitseeriti *Robert Morris*e ülevaateartikkel juba mainitud ja mainekas ajakirjas *CACM* (*Communications of ACM*) paisksalvestusmeetoditest ning – huvitava seigana – see artikkel „legaliseeris“ seni programmeerijate slängi kuulunud termini *hash-coding* (verbina *to hash*)[4].

12.5.2. Paiskfunktsioonid

Tuletagem meelde tõika, et tabeli kirje võti *key* võib olla kas mittenegatiivne täisarv (*C*-programmi jaoks *int*-tüüpi) või tekst. Kuivõrd meil prevaleerib (seni) *int*-arvude 4-baidine esitus, siis tuginegem sellele. Kui kirje võti ongi kuni 32-bitine märgita täisarv või kuni 4-sümboliline tekst, siis võime teda esimeses lähenduses käsitledagi kui paiskfunktsiooni tegelikku argumenti *key*', seejuures teksti käsitleme funktsiooni väärtuse arvutamisel *kahendarvuna* (näiteks, võtit KOER interpreteerime kui 16-ndarvu 4b4f4552 – kui järgime *ASCII*-koodi)[5]. Kui aga võtme väärtus ei mahu 4-baidisele väljale, tuleb ta 4-baidiseks kokku pakkida. Mitmetel põhjustel peetakse vastuvõetavaks pakkimismeetodiks *bitikaupa loogilist mittesamaväärtustamist* (*xor = exclusive or*, *C*-keeles tähistatud kui „^”). Pakkimist (ja võimalik, et pakitud arvuga veel millegi ette võtmist) – võtmeväärtuse teisendamist – tähistame kui $key \rightarrow key'$.

Paiskfunktsioonile esitatakse kaks nõuet (vt. näit [Knuth III], lk. 608):

1. Paiskaadressi arvutamine peab toimuma võimalikut kiiresti ja
2. Funktsioon peab genereerima võimalikult vähe põrkeid; loomulik oletus on, et paiskaadressi arvutamises peavad osalema kõik võtme bitid.

Internetist ja raamatukogudest leiab neist funktsioonidest üpris palju teavet, ent juba *Knuth*i ülevaateraamatu koostamise ajaks olid “kristalliseerunud” peamised kaks erinevat lähenemist: *jagamisvariant* ja *korrutamisvariant.*

[1] Kui allpool mainitud autoreid pole siin viidatud, siis vastavad allikad leiate *Knuthi* tekstist.

[2] *W. W. Peterson* - Addressing for random access storage. IBM Journal of Research and Development 1:130--146, 1957 . (Viide leitud *BibNetWiki* vahendusel.) Ta käsitles põhjalikult lahtist adresseerimist.

[3] Sellele mehele võlgneme tänu *baidi* (*byte*) mõiste eest: olles üks arvuti *IBM Stretch* väljatöötajaist, mõtles ta välja selle termini tähistamaks minimaalset välisseadmetega info vahetamise välja. Algselt tähistas *bait* 4 bitti, siis 7-t ja *IBM System/*360-st alates sai standardiks 8 bitti.

[4] Loodetavasti andestab meiegi lugeja, kui näiteks programmide kommentaarides seda terminoloogiat kasutame.

[5] Mõistagi, mingeid aritmeetikatehteid 4-baidise *char*-tüüpi väljaga teha ei saa. Väljapääs on näiteks selle teksti baithaaval transformeerimises märgita *int*-muutujaks (vt. funktsiooni *text_to_key*(..) siintoodud näites).

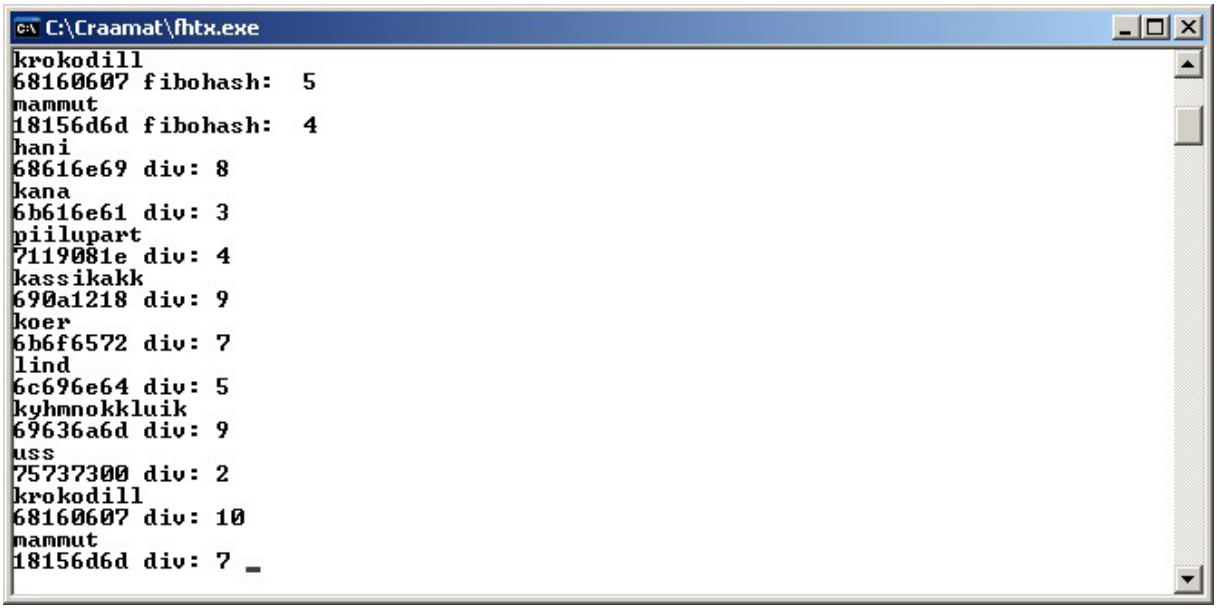
Jagamisvariandi puhul valitakse tabeli pikkus *n* (sinna mahtuvate kirjete arv) kui esimene algarv[1], mis on "sobival moel"[2] suurem kui paisatavate võtmete arv *m*. Kirjet saab adresseerida kui $T[i \times p]$, kus i ($0 \le i < n$) on kirje tabeli-indeks ja p on kirje pikkus baitides Paiskaadress a_i leitakse sel juhul nii: a_i = jääk(key'$_i$ /n). Too jääk on garanteeritult tabeli piiresse mahtuv indeks, so. $0 \le a_i < n$.

C-keeles saame *int*-arvude *a* ja *b* jagatise *jäägi c*, kasutades tehet *c=a%b* (*a/b* resultaat on jagatise täisosa). Näiteprogramm:

```
//fhtx.c :: Jagamis-hash-funktsioon tekstidele 23.01.09
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

char *T[10]={"hani","kana","piilupart","kassikakk","koer","lind",
          "kyhmnokkluik","uss","krokodill", "mammut"};
//suvalise pikkusega teksti pakkimine 4-baidiseks märgita int-arvuks
unsigned int text_to_key(char *t){
    int i;
    unsigned int k=0;
    printf("\n%s\n",t);
    while(*t != '\0'){
       for(i=0;i<4 && *t !='\0';i++,*t++) k^=*t<<(3-i)*8;
       }
    printf("%x ",k);
    return(k);
}
//n on tabeli pikkus (algarv)
int div_hash(unsigned int k, int n){
   printf("div: %d ",k%n);  //paiskaadress on teisendatud võtme ja tabeli pikkuse jagatise jääk
   return(k%n);
    }
//paisktabeli pikkus on 13
int main( ){
   int i,a,b;
   unsigned int key;
   for(i=0;i<10;i++){
      key=text_to_key(T[i]);   //võtmeväärtus -> märgita täisarv
      a=div_hash(key,13);
      }
   getchar( );
}
```

[1] Teatavasti saame seda tingimust rahuldavat algarvu programselt ja lihtsalt leida.
[2] *Sobival moel* sõltub valitud meetodist. Lahtise adresseerimise korral peaks suhe *m/n* olema ligikaudu 2/3, välisaheldus töötab vastuvõetava kiirusega ka siis, kui $m \approx n$ (mõistliku paiskfunktsiooni korral).



Joonis 12.5.2.a. Jagamismeetodil leitud paiskaadressid *div*.

Korrutamisvariant

Kui *jagamisvariandi* puhul lähtusime üldiselt tunnustatud seisukohast, et võimalikke põrkeid saab vältida, valides tabeli pikkuseks *n* algarvu, siis *korrutamisvariandi* puhul soovitatakse tavaliselt kaht varianti: esiteks, tabeli pikkus *n* on arv 2^m ning paiskaadressina kasutatakse *m* bitti (manipuleeritud) võtmest, teiseks – tabeli pikkus on suvaline (tingimusel, et paisatavate kirjete arv poleks suurem kui tabeli pikkus).

Esimesel juhul kasutatakse paiskaadressi leidmiseks (pakitud) võtmeväärtuse *k' ruutu k'*× *k'* – sellepärast, et võtme väärtuse võimalikult paljud bitid osaleksid paiskaadressi leidmisel. Bitieralduse-meetod *kitsendab* oluliselt kirjete võtmete väärtusvaru lubatavat hulka, kuivõrd neljabaidiste *int*-arvude ruutu tõstmisel on reaalne *ületäitumise* oht (vanadel *C*-kompilaatoritel oli võimalus ületäitumist blokeerida, uutel enam mitte). Seega võiksime tekstilisi võtmeid esmalt kokku pakkida 16-bitisele *märgita int*-väljale, ja nonde väärtuste ruutude jaoks kasutada 32-bitist välja. Ent (suhteliselt) väikeste *int*-tüüpi võtmete puhul on see variant täiesti kasutatav.

Allpooltoodud algoritmi esitus pärineb *TTÜ* kolleegi *Viktor Leppiksoni* materjalidest.

```
//fhf.c :: "ruudu keskelt" hash-function  23.01.09
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double A;
int HT[32];
int K[20]={77,62,1,9,98,3,33,23,12,13,
        54,76,5,8,27,2,81,87,15,22};
int mask=0;
int barv=0;  //maski bittide arv
//n: paisktabeli pikkus (eeldame 2 astet): väljund
```

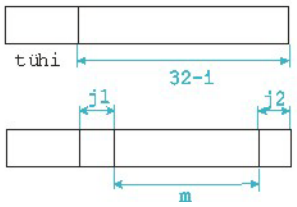
```
int tee_mask(int n){
   int i;
   int yx=1;
   for(i=0;i<32;i++){
      if((yx & n)== 1)break;
      n>>=1;
      barv++;
      }
   for(i=0;i<barv;i++){
      mask |= yx;
      yx <<= 1;
      }
   return(barv);
}
```

Joonis 12.5.2.b. Viktor Leppikson.

Paigutusfunktsioon ruutkeskmise meetodil (middle square method):

75A

```
int k; // võti
int m; // tabeli pikkus on 2^m
int i,j1,j2; // abimuutujad
unsigned int kk,bit=0x80000000; // abimuutujad
unsigned int mask; // konstant, mille m madalamat bitti
                   // on ühed, ülejäänud nullid,
                   // näiteks m=10 puhul on mask 0x3FF
int h; // paigutusfunktsiooni väärtus
kk=k*k; // võtame ruutu
for (i=0; !(bit & kk); i++, bit>>=1);
j1=(32-i-m)/2;
j2=32-i-m-j1;
h=((mask<<j2) & kk)>>j2;
```




```
k=2837, kk=8048569=7ACFB9(16)
tabeli pikkus 2 astmes 8
00000000 01111010 11001111 10111001

00000000 00000000 11111111 00000000
00000000 00000000 11001111 00000000

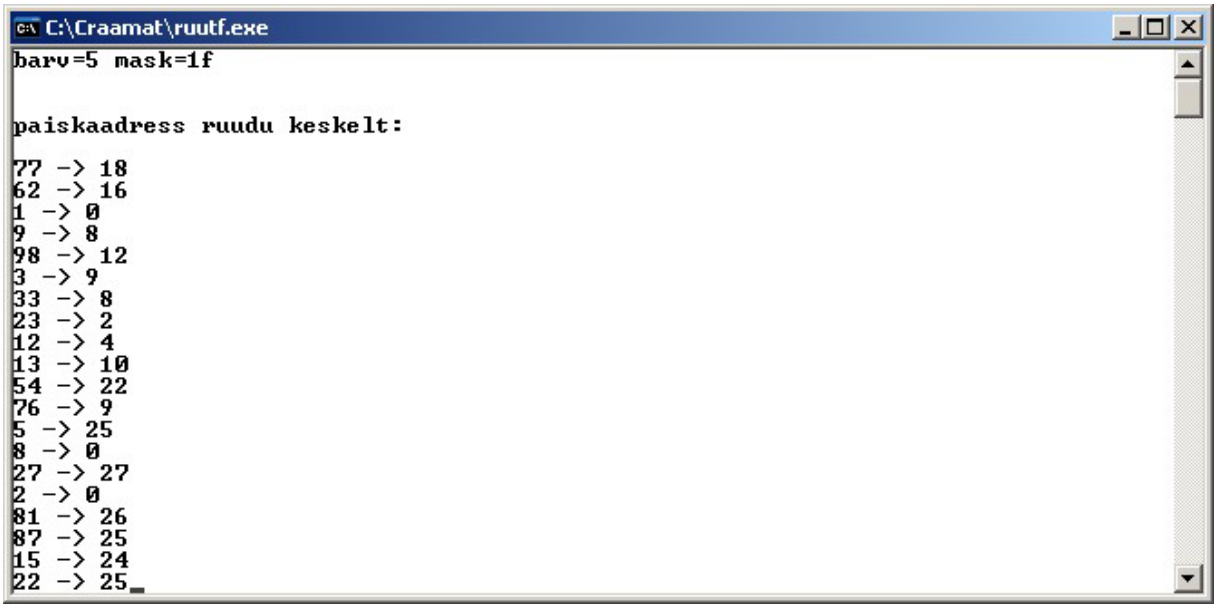
00000000 00000000 00000000 11001111
saame 207
```

Joonis 12.5.2.c.Paiskaadress võetakse “ruudu keskelt” (vt. [Leppikson A&A], fig75A.html)

```
//vt. V.Leppikson, fig75A.html
int ruutf(int k, int n){
   int a,i,j1,j2,kk;
  unsigned int bit=0x80000000;
   kk=k*k;
   for(i=0; !(bit & kk); i++, bit>>=1);
   j1=(32-i-barv)>>1;
   j2=32-i-barv-j1;
   a=((mask << j2) & kk) >> j2;
   return(a);
}

int main( ){
   int i,a;
```

```
    printf("barv=%d mask=%x\n",tee_mask(32),mask);
    getchar( );
    printf("\npaiskaadress ruudu keskelt:\n");
    for(i=0;i<20;i++){
       a=ruutf(K[i],32);
       printf("\n%d -> %d",K[i],a);
       }
    getchar( );
}
```



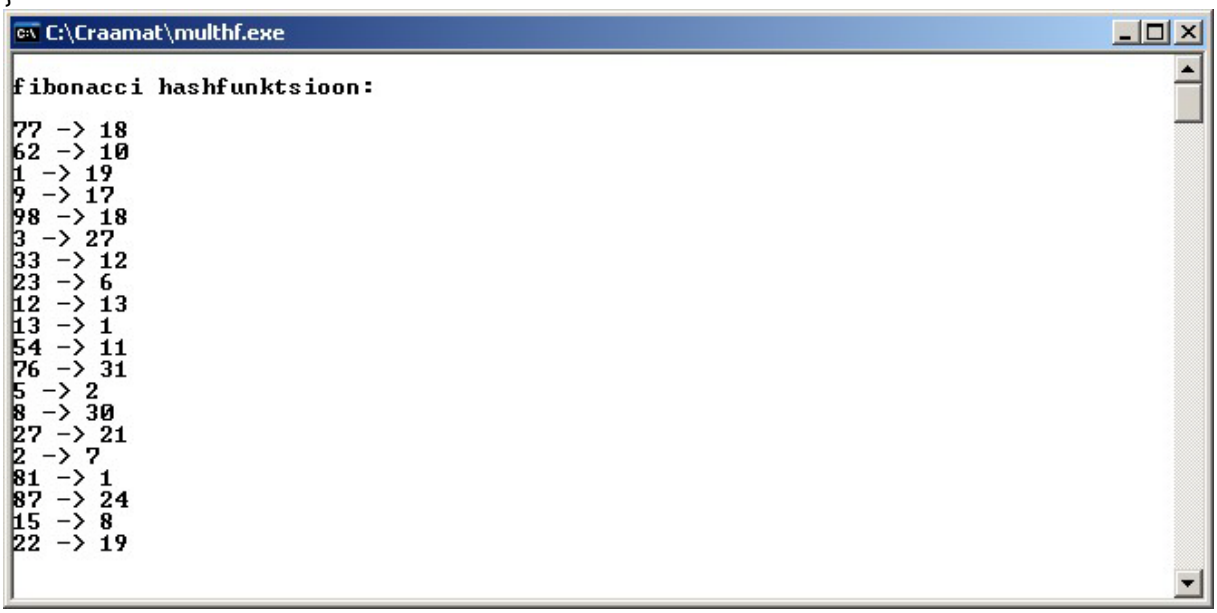
Joonis 12.5.2.d. Programmi *fhf.exe* lahendusprotokoll.

Teisel juhul kasutatakse *korrutamismeetodit* nii, et (võimalik, et modifitseeritud) võtmeväärtused kujutatakse reaalarvudena ning iga üksiku võtme paiskaadressi leidmiseks korrutatakse seda mingitel kaalutlustel valitud konstandiga *c* vahemikust 0..1. Korrutisega toimitakse põhimõtteliselt samuti nagu seda tehti jagamismeetodi puhul ([Knuth III], lk. 605), so. reaalarvulisest „vahevõtmest“ saadakse *int*-tüüpi indeks vahemikust 0..*n*–1.

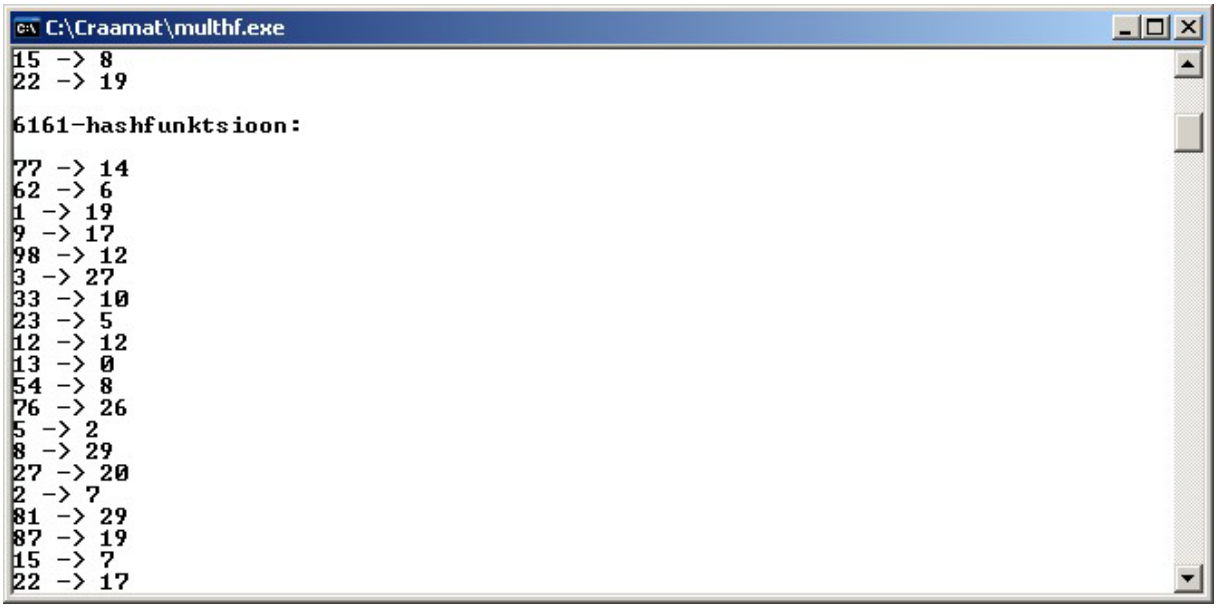
*D. Knuth*il [Knuth III, lk. 606 jj.] on toodud koos põhjendustega kolm võimalikku varianti *c* valimiseks; esimene neist põhineb *kuldlõikel* ja viimase arvutamine *Fibonacci arvudel*; see on kahe lõigu *a* ja *b* proportsioon (jagatis) väärtusega $(\sqrt{5}-1)/2 = 0.6180334$. *Knuth* näitab, et selle kordaja kasutamine „paiskab suhteliselt väheerinevad“ võtmed üksteisest „suhteliselt kaugele“. Lugejale soovitame tutvuda nii kuldlõike-*c* kui ka kahe muu *Knuth*i refereeritud konstandi ($c = 0.6161616161$ ja $c = 0.6125423371$) teoreetiliste põhjendustega viidatud raamatust. Aga siin esitame neid kolme konstanti kasutava programmi teksti (moodul *multf* on laenatud *V. Leppikson*ilt [Leppikson, A&A]), ning slaidid lahendusprotokollidega.

```
//multhf.c :: korrutamismeetodi hash-funktsioonid  23.01.09
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int HT[32];
```

```
int K[20]={77,62,1,9,98,3,33,23,12,13,54,76,5,8,27,2,81,87,15,22};
 int multf(int k, int n, double A){
    int a;
    double dummy;  //meie ei kasuta, see on modf( ) väljundparameeter
    a= (int) floor(n*modf(A*k,&dummy));  //V. Leppikson fig77C.html[1]
    return(a);
}
```

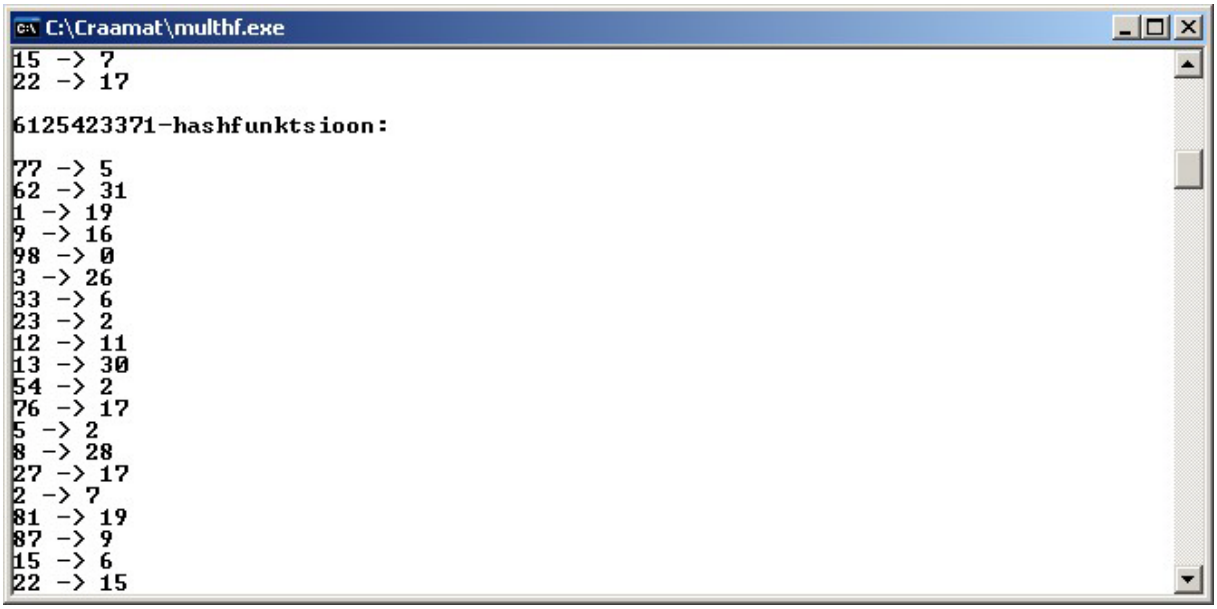


Joonis 12.5.2.e. Paiskaadressid „*Fibonacci* funktsiooniga“.



Joonis 12.5.2.f. Paiskaadressid „*c=6161*...- funktsiooniga“.

[1] *V. Leppikson*: „Korrutusmeetodil on paigutusfunktsiooni väärtuseks $A \times k$ murdosa, mis on läbi korrutatud tabeli pikkusega ja ümardatud ülespoole täisarvuks“ [Leppikson A&A, fig77C.html].



Joonis 12.5.2.g. Paiskaadressid „*c=6125..*- funktsiooniga“.

```
int main( ){
   int i,a;
   double fA;
   fA=(sqrt(5)-1)/2;  //kuldlõige: 0.6180339887 Knuth III, lk. 606
   printf("\nfibonacci hashfunktsioon:\n");
   for(i=0;i<20;i++){
      a=multf(K[i],32,fA);
      printf("\n%d -> %d",K[i],a);
      }
   getchar( );
   printf("\n6161-hashfunktsioon:\n");
   for(i=0;i<20;i++){
      a=multf(K[i],32,0.6161616161);
      printf("\n%d -> %d",K[i],a);
      }
   getchar( );
   printf("\n6125423371-hashfunktsioon:\n");
   for(i=0;i<20;i++){
      a=multf(K[i],32,0.6125423371);
      printf("\n%d -> %d",K[i],a);
      }
   getchar( );
}
```

Vaevalt, et me oleme õigustatud neist „testidest“ midagi järeldama, ent tehkem ikkagi üks puhtillustratiivne kokkuvõte (lisame siia käsitsi tehtud andmed nende võtmetega saadud põrgetest, kui tabeli pikkus olnuks algarv 31 ja funktsiooniks jagamismeetod):

meetod	***n***	***m***	**põrkeid**
ruudu keskelt	**32**	**20**	**5**
fibonacci	**32**	**20**	**2**
6161...	**32**	**20**	**4**
6125..	**32**	**20**	**5**
jääk (k%31)	**31**	**20**	**4**

Tabel 12.5.2.a . Põrgete arv (kirjete arv *m*=20) väikese *int*-võtmega (*n*=) 32-kirjeses tabelis.

Loomulikult ei maksa seda tabelit eriti tõsiselt võtta: katseandmeid on *liiga vähe* üldistamaks meetoditele antavaid hinnanguid; aga: siin võib olla idee (üliõpilase) uurimustööks, mille teema on „paiskfunktsioonide testimine ja nende efektiivsuse võrdlus“. Meie korrutusmeetodinäidete andmed olid väikesed *int*-võtmeväärtused ja neid oli vähe; tabeli *täitumuskoefitsient u* (*u*=kirjete arv *m* / tabeli pikkus *n*) oli *ca* 60%.

Muud variandid peale *jagamis*- ja *korrutamis*meetodi-paiskfunktsiooni kirjutamiseks on mõistagi ka olemas, Internetist leiame fraasi „*hash function*“ abil hulganisti viiteid[1], ja sugugi kõik nad pole liigitatavad ei *jagamis*- ega ka *korrutamismeetodi* alla. Allpool toome vaid ühe võimaluse (laename sellegi *V. Leppikson*ilt [Leppikson A&A, fig76.html]): see on „voltimismeetod“. Selle meetodi idee seisneb selles, et (kokkupakitud 4-baidine) võti *c* pakitakse omakorda kokku, tabeli pikkus *n* on arvu 2 aste ($n=2^m$, *m* peab selle meetodi jaoks olema suurem kui 8) ning paiskaadressi *a* saamiseks kasutatav pakkimisskeem on järgmine:

$a=(c[0] \wedge c[1]) + ((c[2] \wedge c[3]) << (m - 8))$;

Seda paiskfunktsiooni kasutav programm võib olla näiteks järgmine:

```
//voltftx.c :: voltimis-hash-funktsioon tekstidele 25.01.09
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

unsigned int mask=255;   //000000ff
unsigned int mask0=0;
unsigned int mask1=0;
unsigned int mask2=0;
unsigned int mask3=0;
unsigned int a,c0=0,c1=0,c2=0,c3=0;
char *T[10]={"kana","hani","piilupart","kassikakk","koer","lind",
```

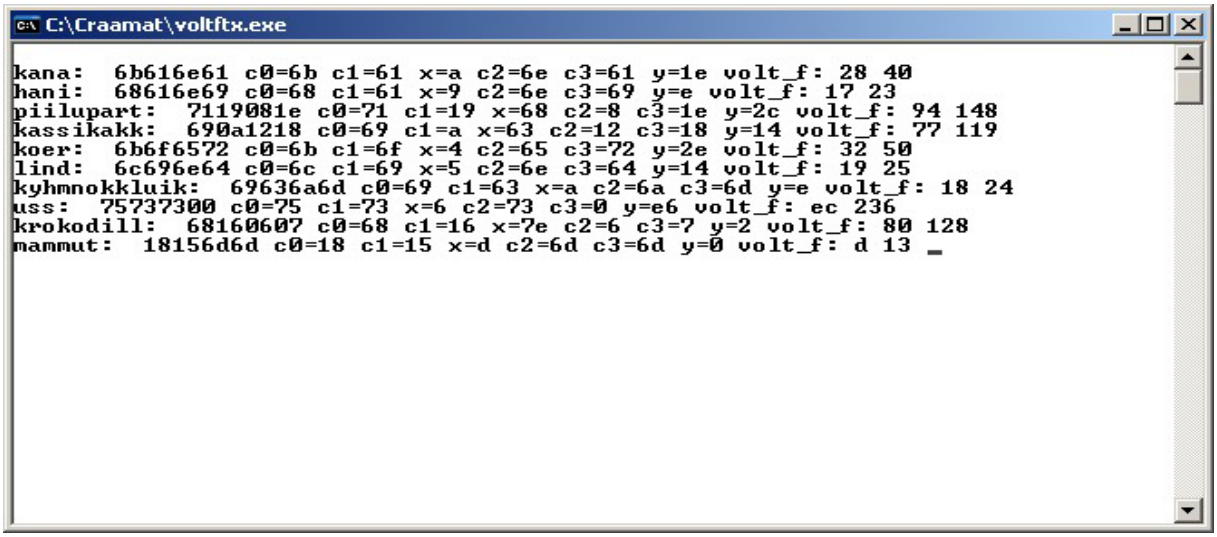
[1] Siiski on vajalik teatud ettevaatus, kuivõrd *hash function* tähistab nii paisktabeli võtme teisendamist paiskaadressiks kui ka teksti krüpteerimise algoritmi (viimasel juhul ei pea me funktsiooni nii konstrueerima, et ta väärtused kuuluksid vahemikku 0..*n*-1). Paiskaadressi tagasiteisendus on reeglina võimatu – me ei saa paiskaadressi järgi taastada kirje võtit; šifreerimise puhul on ülesanne teine – kodeerida suvalise pikkusega tekst nii, et šifrit teadmata oleks selle dekodeerimine nii raske kui võimalik, ent õige šifri abil tuleb tagada esialgse teksti taastamine. Eestikeelses terminoloogias tehakse nende asjade vahel õnneks vahet: esimene on *paiskfunktsioon* ja teine on *räsifunktsioon*. Aga et see inglise keeles teisiti on, on ehk seletatav meile juba teada tõigaga, et nii *A. I. Dumey* kui ka *W. W. Peterson* olid eeskätt krüptoloogid ja alles teises järjekorras programmeerijad. Nende jaoks oli ekskurss paisksalvestusse põgus ja elegantne „kõrvalepõige“ oma põhitemaatikast.

```
            "kyhmnokkluik","uss","krokodill",
            "mammut"};

unsigned int text_to_key(char *t){
     int i;
     unsigned int k=0;
     printf("\n%s: ",t);
     while(*t != '\0'){
        for(i=0;i<4 && *t !='\0';i++,*t++)k^=*t<<(3-i)*8;
        }
     printf(" %x ",k);
     return(k);
}

int volt_f(unsigned int k, int n, int m){
    unsigned int x,y;
    c0=(mask0&k)>>24; printf("c0=%x ",c0);
    c1=(mask1&k)>>16; printf("c1=%x ",c1);
    x=c0^c1; printf("x=%x ",x);
    c2=(mask2&k)>>8; printf("c2=%x ",c2);
    c3=mask3&k; printf("c3=%x ",c3);
    y=((c2^c3)<<(m-8)); printf("y=%x ",y);
    a=x+y;  //V. Leppikson, fig76C.html
    printf("volt_f: %x %d ",a,a);
    return(a);
}

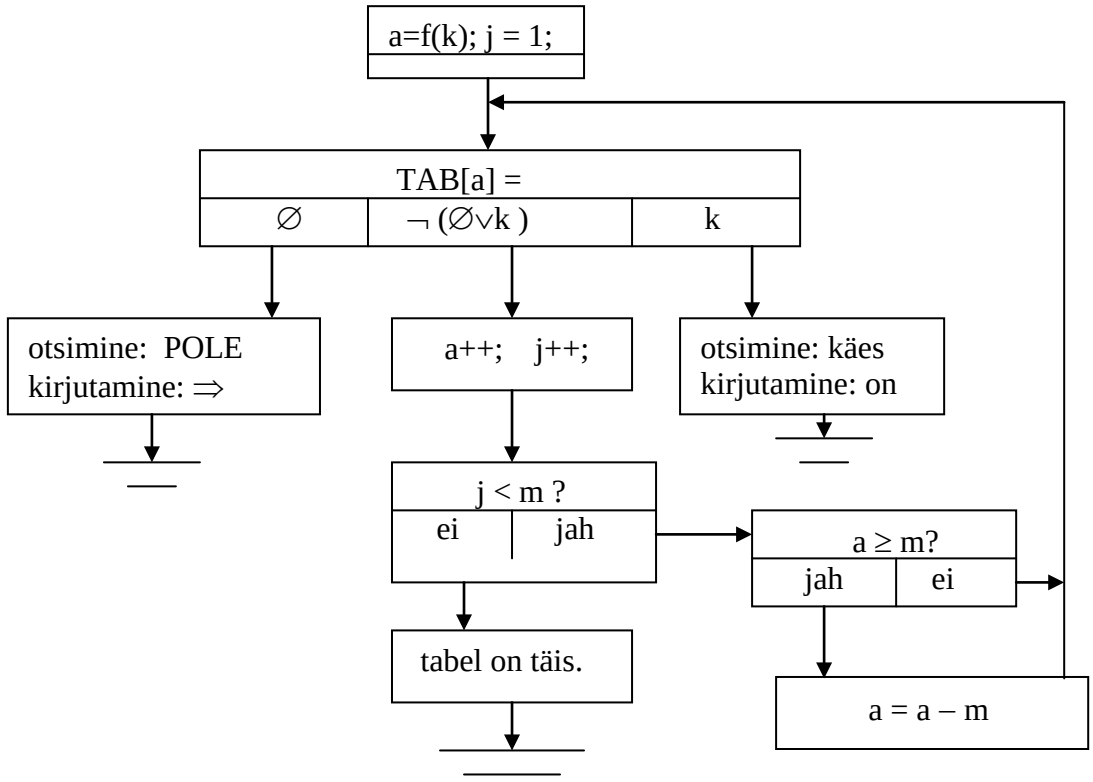
int main( ){
    int i;
    unsigned int key;
    mask3=mask;       //000000ff
    mask2=mask3<<8;   //0000ff00
    mask1=mask2<<8;   //00ff0000
    mask0=mask1<<8;   //ff000000
    for(i=0;i<10;i++){
        key=text_to_key(T[i]);
        a=volt_f(key,512,9);
        }
   getchar( );
}
```



Joonis 12.5.2.h. Voltimismeetodi lähte-paiskaadressid.

Mõistagi, selle mooduli resultaate pole meil millegagi võrrelda, kuivõrd paiskame 10 võtit 512 võtit mahutavasse tabelisse (täitumuskordaja u = 0,02), ent lahendusprotokoll näitab, et see funktsioon töötab nõuetekohaselt (paiskab tabeli sisse: $0 \le a < 512$ ja põrkeid pole). Resümeerigem: paiskfunktsioon $a_i = f(k_i, n)$ annab väljundisse paiskaadressi a_i ja ei enamat; kui see aadress on hõivatud teise võtme k_j poolt ($k_i \neq k_j$), siis see pole paiskfunktsiooni probleem.

12.5.3. Lahtise adresseerimise meetod



Joonis 12.5.3.a. Lineaarne paiskamine.

„Lahtine adresseerimine“ tähendab, et paisktabelile reserveeritakse sidus mäluväli pikkusega $n \times p$ baiti (n on tabeli pikkus arvestusega, et $n > m$, kus m on salvestatavate kirjete arv ning p on kirje pikkus baitides), ja põrke korral otsitakse kirjele mingit algoritmi järgides uus koht.

Ülalpool (joonis 12.5.3.a) esitasime ühe „lahtise adresseerimise“ algoritmi (plokkskeemina), ja nimelt neist lihtsaima, „lineaarse paiskamise“ oma (vt.[Knuth III], lk. 615 – 616).

Paiskfunktsioone tutvustades oli meil üks näide, kus leidsime jagamismeetodil 10 looma paiskaadressid. Paiskame nad tabelisse, järgides võtmete etteandmise mainitud näite järjekorras ja käitudes põrgete korral ülaltoodud algoritmi kohaselt. Saame järgmise tabeli:

i	loom	nimi	värv	iseloom
0				
1				
2	uss	Janno	roosa	paha
3	kana	Kaaga	kirju	paha
4	piilupart	Donald	valge	hea
5	lind	Säuts	kirju	hea
6				
7	koer	Jipp	must	hea
8	hani	Dab-Dab	valge	hea
9	kassikakk	Uhhuhhuu	tume	ei tea
10	kyhmnokkluik	Daam	valge	paha
11	krokodill	Gena	roheline	hea
12	mammut	Värdi	hall	ei tea

Tabel 12.5.3.a. Lineaarse paiskamise algoritmiga saadud paisktabel.

Kirjed indeksitega 2 – 9 on oma „õigetel kohtadel“. Kühmnokkluik (paiskaadressiga 9) paigutati esimesele vabale kohale, indeksiga 10. Krokodilli paiskaadress on 10, tema pandi 11. reale. Ent mammutile (lähte-paiskaadressiga 7) koha leidmiseks pidime teda (so. võtit) võrdlema järjekorras võtmetega „koer“, „hani“, „kassikakk“, „kyhmnokkluik“ ja „krokodill“ kuni leidsime tühja, 12. rea. Viimane seik sobib illustreerima lineaarse paiskamise suurimat puudust – *kobardumist*[1] – kus tabelisse tekivad pikad kirjetega tihedalt täidetud piirkonnad, mis tuleb neil juhtudel, kui järjekordse kirje lähtepaiskaadress satub sellisesse piirkonda, uue koha otsimiseks läbida[2]. Ja lihtne on näha, et kobarad kasvavad seda kiiremini, mida suurem on tabeli täitumus[3].

[1] Vene keeles скучивание.
[2] Algoritmi järgides uus kirje, mille võtme järgi saame lähtepaiskaadressi vahemikku 7..12, paigutatakse tabeli 0-ndale reale. Järgmine samasugune läheb 1., veel järgmine 6. reale ja suvalise järgmise kirje jaoks on *tabel täis.*
[3] Lahtine adresseerimine on piisavalt efektiivne meetod kuni täitumuskoefitsiendini 0,7; alates 0,8-st kasvab otsimisaeg „drastiliselt“ (vt. [wHT]).

Plokkskeemilt näeme, et algoritm arvestab võimalusega, et tabel saab täis. Sellise situatsiooni lahendamiseks (ja ennetamiseks) soovitatakse mingist täitumisprotsendist alates teha *ümberpaiskamine* (*rehash*): võtta rohkem dünaamilist mälu kui senine tabel vajas, paisata *kõik* kirjed uude tabelisse ümber ja vana tabel kustutada.

„Lahtist adresseerimist" kasutavaid meetodeid on välja töötatud üsna palju, väljavõtteliselt tutvustab neid *D. Knuth* [Knuth III, lk. 619 – 642]. Nende ridade autoril on kogutud ka paar plokkskeemidena kujutatud algoritmi[1]: *lineaarne jagatismeetod* ja *lineaarse jagatismeetodi modifikatsioon.* Lisaks neile mainigem veel paar meetodivarianti: *pseudojuhuslik* ja *ruutjagatismeetod,* Kõik nad on alates 60%-sest täitumusest alates kiiremad *lineaarsest* meetodist ning parima kiirushinnanguga on mainitutest *lineaarse jagatismeetodi modifikatsioon* (95%-se täitumuse korral *ca* 1,5 korda kiirem *pseudojuhuslikust* meetodist ja teistest *jagatismeetoditest* ning viis korda kiirem siintutvustatud *lineaarsest* meetodist.

Mainigem, et see kiire *lahtise adresseerimise* meetod tõstab tabeli täitmise (uute kirjete lisamise) ajal otsimiskiiruse parandamise huvides kirjeid tabelis füüsiliselt ringi. Mis tähendab, et otsimiskiiruse tõstmise huvides kaotatakse salvestamiskiiruses; seega sobib see meetod „staatilistele tabelitele", mida pärast täitmist ei kavatseta enam modifitseerida.

Paisksalvestuse tutvustamist alustasime tõdemusega, et kirje reaalne paiskaadress a_i arvutatakse kahe funktsiooni abil: paiskfunktsioon *f* annab *lähtepaiskaadressi* ja (vältimatud) põrked lahendatakse *põrkefunktsiooni g* abil: $a_i = f(k_i, n) + g(j, k_i, n)$. Paiskfunktsioone käsitlesime ülalpool, siin toome mõned näited põrkefunktsioonidest.

- *lineaarse paiskamise* põrkefunktsioon on $c \times j$ (c on „tühja koha" otsimise sammu kordaja; meie plokkskeemis $c=1$, aga alati $c>0$ ja j on „samm");
- *lineaarse jagatismeetodi* (n on algarv) põrkefunktsioon g on $j \times P_i$, kus P_i=täisosa(k_i'/n). Paiskfunktsioon f=jääk(k_i'/n). Kui $P_i=0$, siis $g_i=c$ ($c>0$);
- *lineaarse jagatismeetodi modifikatsiooni* (n on algarv, f=jääk(k_i'/n)) põrkefunktsioon P_i on jääk($k_i'/n-2$)+1, mille kordaja on taas vaba koha otsimise sammude loendaja j ($j=0, 1, ...$).

Kõigi nendel meetoditel, mis likvideerivad põrkeid tabeli sees, on üks ühine nõrk koht: kirje eemaldamine tabelist on komplitseeritud. *Knuth* käsitleb seda probleemi [Knuth III] lehekülgedel 625 ja 626. Probleemi tõsiduse lihtsaks näiteks oletame, et kustutame oma näitetabelist „koera": tabeli rida indeksiga 7 on pärast seda tühi[2]. Kui pärast seda otsime tabelist „mammutit" (lähtepaiskaadressiga 7), siis kuivõrd see rida on tühi, siis „lugemisel" anname signaali „pole tabelis", salvestamisel teeme tema kirje sinna – ent „mammut" on tabelis täiesti olemas, 12. real. Ning oleme rikkunud (tavaliselt kehtivat) reeglit, et võtmed peavad tabeli kirjetel olema unikaalsed, ja raiskame tabeliruumi „rippuva kirjega", milleni pole enam mingit võimalust jõuda.

[1] Autor ei pea otstarbekas nende kaante vahel neid meetodeid tutvustada, ent huvilised üliõpilased võivad algoritme siinkirjutaajalt alati küsida.

[2] Just sama õnnetus juhtuks, kui „koera" asemel kustutame näiteks „kassikaku" või „krokodilli". „Koera" puhul juhtub see lihtsalt varem. Nentigem, et dünaamilise tabeli pidamiseks „avatud" paiskamine ei sobi.

12.5.4. Välisaheldus

See on meetod, kus tabelit kujutatakse põhimõtteliselt *viidavektorina struct lyli *HT*[*n*], kus *lyli* kirjeldus võib olla näiteks selline:

```
struct lyli{
        struct kirje *rec;        //viit kirjele: võti+info
        struct lyli *collision;    //põrkeviit
        };
```

Tuletagem meelde, et paisksalvestus mõeldi välja masinorienteeritud programmeerimise ajal, sootuks varem, kui põrkeid tabeli piires likvideerivad meetodid. Ja et ka 100%-se täitumuse korral (muidugi, kui paiskfunktsioon *f* on „mõistlik“[1]) ei ületa keskmine otsisammude arv hinnangut 1,5. Ja et ületäitumis- ja ümberpaiskamise probleeme ei pruugi tähele panna seni, kuni otsikiirus on veel talutav. Ja kirje kustutamine ei tekita mingeid probleeme. Ja *põrkefunktsiooni* sisuliselt pole, põrge lahendatakse ahela lõppu[2] uue lüli lisamisega.

Selle meetodi ainus miinus on lisamälu-vajadus ahelate jaoks, ning pool sajandit tagasi oli see üsna mõjuv argument: mälu oli vähe ja see oli kallis.

Välisahelduse tutvustamiseks naaskem oma äsjase tabeli juurde, ent „alustagem algusest“: oletagem, et kirjed sisestati – selles järjekorras, nagu me nende paiskaadresse leidsime – läbivaadatavasse (so, korrastamata) tabelisse järgmise programmiga:

```
//maketab.c 31.01.09

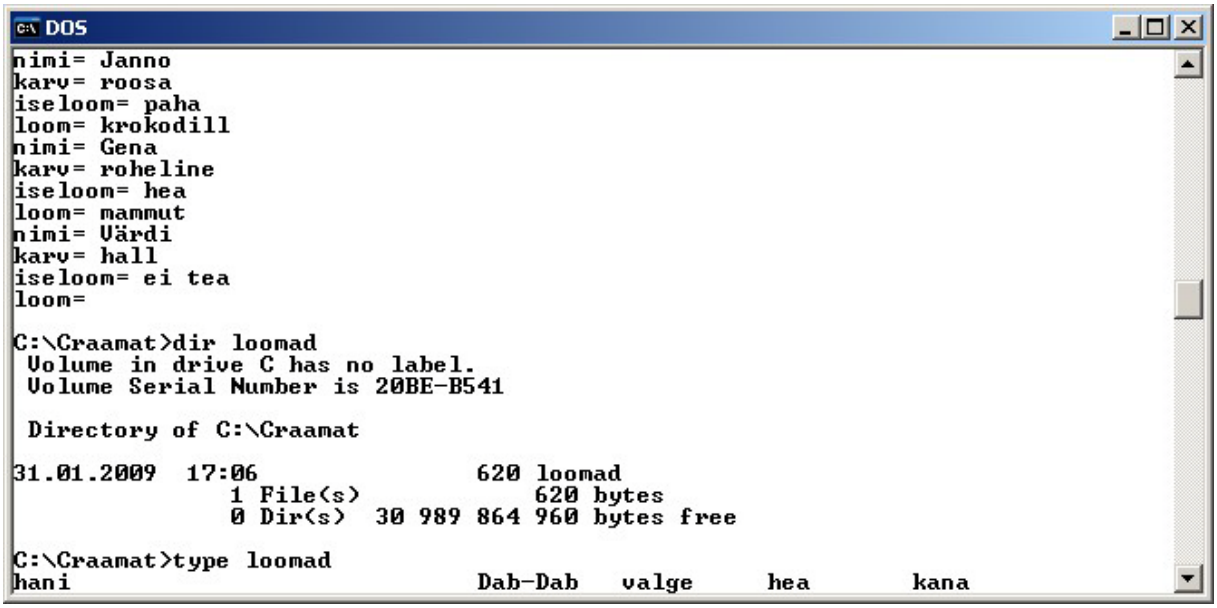
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct loom{
   char key[32];
   char nimi[10];
   char karv[10];
   char iseloom[10];
};
struct loom *puhver;
FILE *loomatabel=NULL;

int main(int argc,char **argv){
   if(argc != 2){
      printf("parameetrina tahan tabeli nime");  //..>maketab loomad
      abort( );
```

[1] Vähimmõistlikuim paiskfunktsioon tagastab suvalise võtme puhul paiskaadressina ühe ja sama *konstandi*.
[2] „Lõppu“ põhjusel, et me peame ahela läbima veendumaks, et uut (või otsi-)võtit seal veel ei ole.

```
      }
   loomatabel=fopen(argv[1],"w");
   puhver=malloc(sizeof(struct loom));
ring: memset(puhver,'\0',sizeof(struct loom));
     printf("loom= ");
     gets(puhver->key);
     if(strlen(puhver->key)==0) goto kirjuta;
     printf("nimi= ");
     gets(puhver->nimi);
     printf("karv= ");
     gets(puhver->karv);
     printf("iseloom= ");
     gets(puhver->iseloom);
     fwrite(puhver,sizeof(struct loom),1,loomatabel);
     goto ring;
kirjuta:
     fflush(loomatabel);
     fclose(loomatabel);
}
```



Joonis 12.5.4.a. Fragment loomade-faili moodustamise dialoogist ja järgnevast.

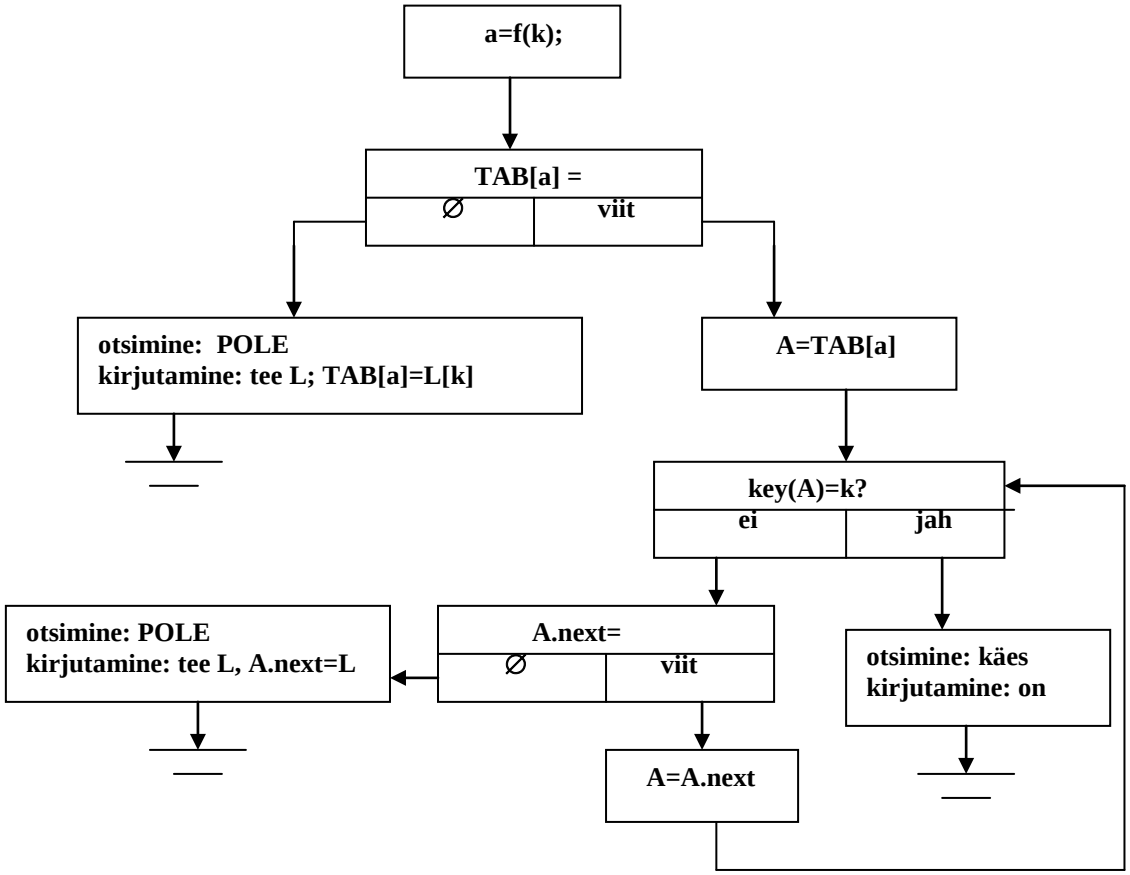
Näeme, et ülaltoodud programm teeb alati uue tabeli: teda ei huvita, kas „loomatabel“ võis olla juba varem moodustatud ja ei paku võimalust „vana tabeli“ täiendamiseks. Modifitseeritavaks tegemine on muidugi lihtne; jätame programmi sellekohase täiendamise lugejatele näpuharjutuseks.

Programmiga *maketab* peetud dialoogi käigus kettale kirjutatud tabel on meile vastuvõetaval kujul selline:

i	loom	nimi	värv	iseloom
0	hani	Dab-Dab	valge	hea
1	kana	Kaaga	kirju	paha
2	piilupart	Donald	valge	hea
3	kassikakk	Uhhuhhuu	tume	ei tea
4	koer	Jipp	must	hea
5	lind	Säuts	kirju	hea
6	kyhmnokkluik	Daam	valge	paha
7	uss	Janno	roosa	paha
8	krokodill	Gena	roheline	hea
9	mammut	Värdi	hall	ei tea

Tabel 12.5.4.a. Kirjete juhuslikus sisestamise-järjekorras salvestatud läbivaadatav tabel.

Seda tabelit kasutame tabeli-teema näitlikustamiseks selle (ala)peatüki lõpuni, ent alustagem välisahelduse tutvustamisega. Algoritmi plokkskeem on järgmine (*L* on uus viidatav lüli) ning „kirjeid eirava“ (so., kirje asemel opereerime ainult võtmega) programmi teksti toome kohe pärast plokkskeemi, ja pildi selle programmi lahendamisprotokollist pärast programmi teksti.



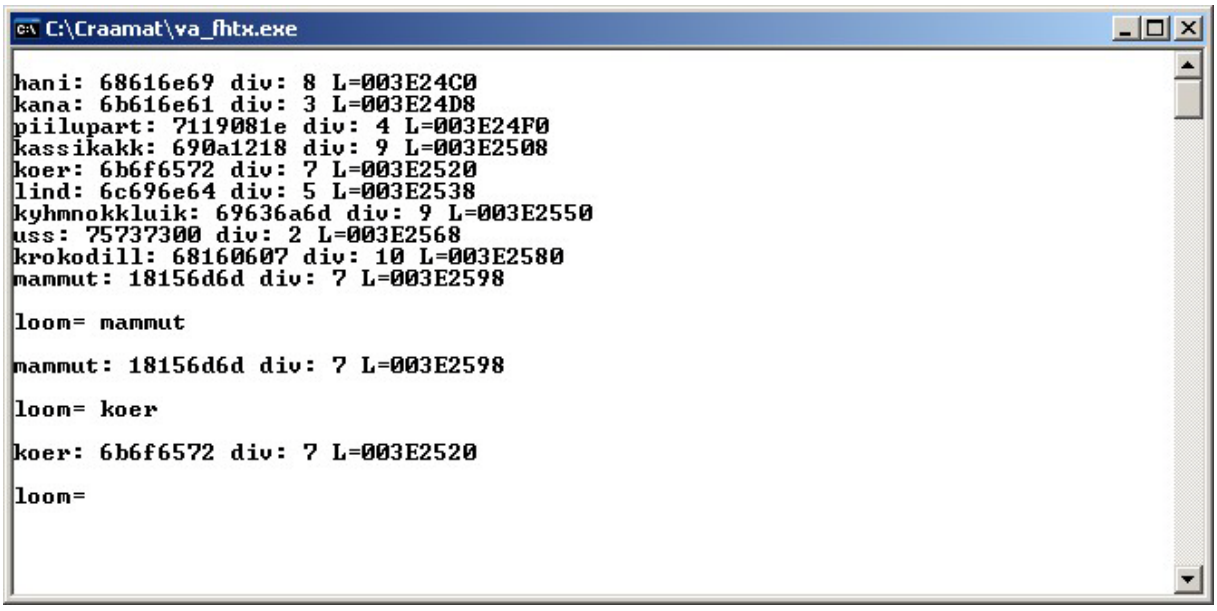
Joonis 12.5.4.b. Välisaheldusega paiskamine.

```
//va_fhtx.c :: jagamis-hash-function tekstidele, välisaheldus 30.01.09
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
char *T[10]={"hani","kana","piilupart","kassikakk","koer","lind",
            "kyhmnokkluik","uss","krokodill","mammut"};
 struct lyli{
    char key[10];
    struct lyli *collision;
};
struct lyli *HT[13];
unsigned int text_to_key(char *t){
    int i;
    unsigned int k=0;
    printf("\n%s: ",t);
    while(*t != '\0'){
       for(i=0;i<4 && *t !='\0';i++,*t++)k^=*t<<(3-i)*8;
       }
    printf("%x ",k);
    return(k);
}

int div_hash(unsigned int k, int n){
   printf("div: %d ",k%n);
   return(k%n);
}

int main( ){
   int i,a;
   char t[20];
   struct lyli *uus,*L;
   unsigned int key;
   for(i=0;i<13;i++) HT[i]=NULL;
   for(i=0;i<10;i++){
      key=text_to_key(T[i]);
      a=div_hash(key,13);
      L=HT[a];
      if(L!=NULL){
         on: if(strcmp(T[i],L->key)==0){
            printf("L=%p\n",L->key);
            goto next;
            }
```

```
            if(L->collision!=NULL){
               L=L->collision;
               goto on;
               }
            uus=malloc(sizeof(struct lyli));
            memset(uus,'\0',sizeof(struct lyli));
            strcpy(uus->key,T[i]);
            L->collision=uus;
            goto next;
            }
         uus=malloc(sizeof(struct lyli));
         HT[a]=uus;
         memset(uus,'\0',sizeof(struct lyli));
         strcpy(uus->key,T[i]);
         next: printf("L=%p ",uus);
         }
      getchar( );
}
```



Joonis 12.5.4.c. Välisahelduse näitedialoogi pilt (*va_fhtx.c*).

Äsjaesitatud programm (ja pilt) on mõeldud näitlikustama välisahelduse ideed ennast, mitte aga tabeli kirjete paiskamist. Kirjete nn. „lahtise paiskamise“ näite tõime pisut varem; nägime, et selline tabel on kompaktne, ent kiiruse huvides tuleb mõned read (kirjete ruumid) paratamatult tühjaks jätta. Välisaheldus „raiskab mälu“ teisiti: me ehitame paisktabeli kirjete tabeli „kõrvale“ (või „peale“). Välisahela lülis peab olema minimaalselt kaks välja: põrkeviit ning kirje indeks (järjekorranumber) tabelis või *viit* kirjele. Otsimiskiiruse huvides lisatakse lülisse veel üks väli – kirje võtme jaoks (viimane on sel juhul dubleeritud: üks eksemplar kirjes, teine paisktabeli ahela lülis).

Järgmine programm kasutab kolmeväljalist lüli-formaati, loeb kettalt (korrastamata) tabeli, teeb selle kirjete otsimiseks paisktabeli ning võimaldab dialoogi vormis tabelist kirjeid otsida.

```
//va_file.c :: jagamis-hash-funktsioon kirjete tekst-võtmetele, välisaheldus 31.01.09
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/stat.h>

struct lyli{
    char key[10];
    struct loom *eluk;
    struct lyli *collision;
};

struct loom{
      char key[32];
      char nimi[10];
      char karv[10];
      char iseloom[10];
};

struct loom *T;      //tabel
struct loom *kirje;  //tabeli kirje

FILE *tabel=NULL;     //tabeli-fail
struct lyli *HT[13];  //paisktabel,
struct stat stbuf;    //<sys/stat.h>
char kRida[32];
int pikkus;   //tabeli pikkus baitides
int N;        //kirjete arv

unsigned int text_to_key(char *t){
    int i;
    unsigned int k=0;
    while(*t != '\0'){
       for(i=0;i<4 && *t !='\0';i++,*t++)k^=*t<<(3-i)*8;
       }
    return(k);
}

int div_hash(unsigned int k, int n){
   return(k%n);
}
```

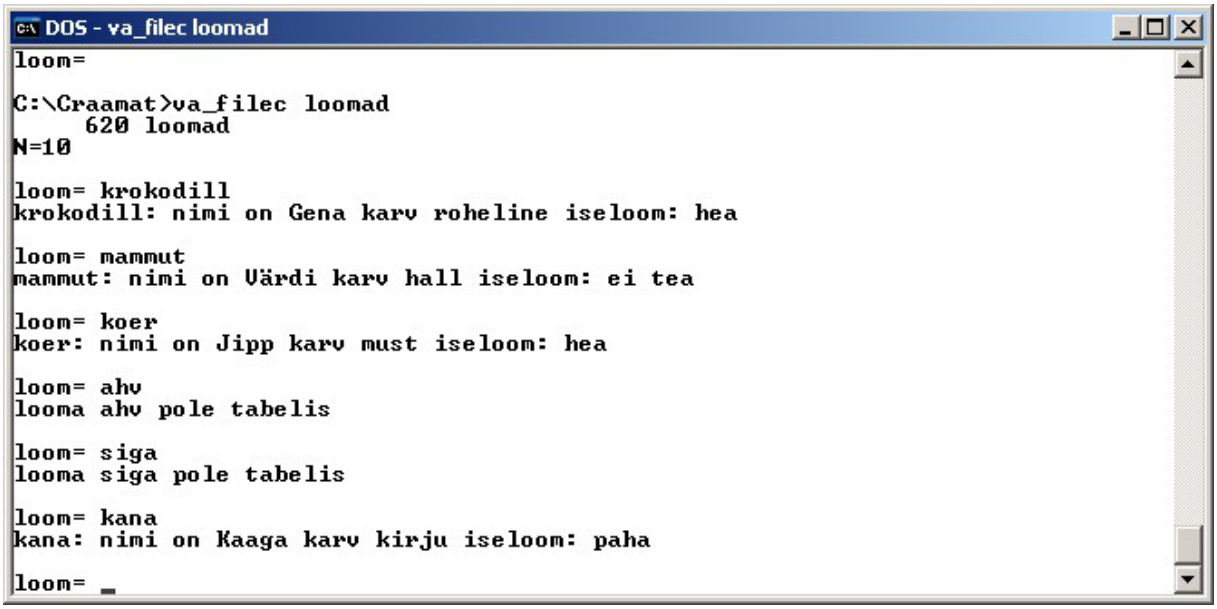
```
int main(int argc,char **argv){
   int i,a;
   char t[20];
   struct lyli *uus,*L;
   unsigned int key;
   if(argc!=2){
     printf("missing table");
     return(0);
     }
   tabel=fopen(argv[1],"r");
   sprintf(kRida,argv[1]);
   stat(kRida,&stbuf);    //anna faili-info
   printf("%8ld %s\n",stbuf.st_size,kRida);
   pikkus=stbuf.st_size;

   N=pikkus/sizeof(struct loom); printf("N=%d\n",N);  //kirjete arv
   T=(struct loom *)malloc(N*sizeof(struct loom));
   fread(T,pikkus,1,tabel);

   for(i=0;i<13;i++) HT[i]=NULL;
   for(i=0;i<N;i++){
      kirje=&T[i];
      key=text_to_key(kirje->key);
      a=div_hash(key,13);
      L=HT[a];
      if(L!=NULL){
         on: if(strcmp(kirje->key,L->key)==0){
            L->eluk=kirje;
            goto next;
            }
          if(L->collision!=NULL){
            L=L->collision;
            goto on;
            }
         uus=malloc(sizeof(struct lyli));
         memset(uus,'\0',sizeof(struct lyli));
         strcpy(uus->key,kirje->key);
         uus->eluk=kirje;
         L->collision=uus;
         goto next;
         }
      uus=malloc(sizeof(struct lyli));
      HT[a]=uus;
      memset(uus,'\0',sizeof(struct lyli));
```

```
        strcpy(uus->key,kirje->key);
        uus->eluk=kirje;
        next: ;
        }

ring: fflush(stdin);
      printf("\nloom= ");
      gets(t);
      if(strlen(t)==0) return(1);
      key=text_to_key(t);
      a=div_hash(key,13);
      L=HT[a];
      kason: if(L==NULL){
            printf("looma %s pole tabelis\n",t);
            goto ring;
            }
      if(strcmp(t,L->key)==0){
            kirje=L->eluk;
            printf("%s: nimi on %s karv %s iseloom: %s\n",
                  kirje->key,kirje->nimi,kirje->karv,kirje->iseloom);
            goto ring;
            }
      L=L->collision;
      goto kason;
}
```



Joonis 12.5.4.d. Kirjete tabelile „peale ehitatud“ paisktabeliga peetud dialoog.

Nagu näeme, pole seegi programm *üldine*, kuivõrd paisktabeli pikkus (13) on „sisse programmeeritud“, ent loodetavasti need lugejad, kes kavatsevad siit eeskuju võtta, programmeerivad

ise mooduli, mis leiab kirjete tegelikust arvust (*N*) lähtudes sellest suurema algarvu[1], või sellest suurema kahe astmena avalduva arvu (viimasel juhul tuleks paiskfunktsioon asendada „korrutamisvarianti“ kasutavaga).Joonisel 12.5.4.d on „pilt“ selle programmi lahendamisseansist

12.6. Otseadresseeritav tabel

Paisksalvestuse puhul tõdesime, et paiskaadress a_i arvutatakse kui funktsioon (töödeldud) võtmeväärtusest, mida korrigeeritakse põrke korral põrkefunktsiooniga *g*( ). Otseadresseerimine on „põrkevaba paisksalvestus“: $a_i=f(k_i)$, kusjuures $a_i \neq a_j$, kui $i \neq j$. Mõistagi, see tingimus on saavutatav ainult võtmetele *k* kehtestatud üpris rangete kitsenduste korral.

Nagu ka paisksalvestuse puhul võivad otseadresseeritava tabeli võtmed olla kas *int*-tüüpi täisarvud või *tekst*. Viimasel juhul peab olema eelnevalt fikseeritud tekstiliste võtmeväärtuste *väärtusvaru* ning efekti saavutamiseks peab eksisteerima kiire funktsioon, mis saab argumendina ette võtme ja tagastab ta järjekorranumbri, mis saab kirje indeksiks. Näiteks nii, et *mast*={*risti, ruutu, ärtu, poti*} indeksitega 0..3 (so, näit. võti *ärtu* tagastab indeksi 2).

Täisarvuliste (int-tüüpi) võtmete puhul on peamiseks kitsenduseks võtmeväärtuste amplituudi (*kmax – kmin*) mahtumine kättesaadavasse operatiivmällu. Näidet vt. jaotises „luuresort“; *sageduste vektori* elemendiks võib võtmeväärtuste sageduse asemel samahästi olla viit antud võtmega kirjele (vektor ise peab olema kirjeviitade-tüüpi selmet olla baidi- või poolsõnapikkuste elementidega).

Ja kui oleme tutvunud järgmise jaotisega (*multijuurdepääs*), mis lubab sekundaarseid võtmeid (unikaalsed põhijuurdepääsu- (primaarsed) võtmed identifitseerivad üheselt iga kirje, ent sekundaarsetega võib seotud olla kuitahes palju kirjeid; võti võib olla näiteks *sugu*: *M* või *N*), siis võime „sageduste vektori“ elemendiga siduda kirjeviitade ahela.

On lihtne näha, et suvalise võtmega otsimiseks kulub sellises tabelis alati 1 samm (saame teada, kas võti on (ja saame kätte temaga seotud informatsiooni) või pole tabelis).

Ja veel, kui võtmed on naturaalarvud vahemikust *min...max*, siis on otseadresseeritavas tabelis kehtivad kõik *järjestatud tabeli* funktsionaalsused (anna esimene, anna viimane, anna võtme järgi, anna eelmine, anna järgmine).

Otseadresseeritava tabeliga manipuleerimise ajaline keerukus on alati $O(n)$, kus *n* on kirjete arv, ning võtme otsimiste arv on üks (1).

12.7. Multijuurdepääs

Tabeli haldamiseks (lisamine, kustutamine, päringutele vastamine) on vältimatu, et iga kirje (reaalse maailma *objekt*) on üheselt identifitseeritav, ja seda *võtme* abil. Ent tegelikes rakendustes võib juhtuda, et me peame võtmetega tabatud kirjeid käitlema erinevalt. Näiteks, võtmega seotud kirje kiireks leidmiseks (või signaaliks: „pole tabelis“) sobib parimini paisktabel, kui me just otseadresseerimist kasutada ei saa.

[1] Kui *N* on suurusjärgus mõni tuhat, on „odav“ (so, kiire) variant kasutada „sisseprogrammeeritud“ (kettal asuvat) algarvude tabelit. Arvutamine võib ebasoodsal juhul olla kiiruse *kuuphinnanguga* [Laur] ning ajalises mõttes me võidame, kui tabeli pikkuseks valime mingi muu arvu.

Ent kui me tahame väljastada tabeli võtmeväärtuste kasvavas või kahanevas järjekorras, siis paisktabel seda ei võimalda. Selleks otstarbeks võime konstrueerida „tabeli peale“ otsimispuu ning seda soovikohaselt läbides saame väljastada tabeli võtmeväärtuste kasvavas või kahanevas järjekorras. Lisajuurdepääsude loomisel pole me tegelikult mingilgi moel piiratud; kui vaja, võime sellise juurdepääsutabeli teha ka siis, kui lisajuurdepääsu võti on tabelis salvestatud kui ujupunkt-arv – me võime ta ümardada vastuvõetava lävendini ning kasutade noid võtmeväärtusi kui nõuetekohaseid. Ja lisajuurdepääs(ud) pole loomulikult kitsendatud ka võtmete unikaalsuse nõudega.

Allpool toodud programm opereerib meie viimaste näidete „loomade-tabeliga“ ning teeb põhijuurdepääsu (välisaheldusega *hash*, võti on loomaliik (krokodill, hani jne)) ja mitu lisajuurdepääsu.

```
//multi.c :: põhijuurdepääs ja lisajuurdepääsud korrastamata tabeli kirjetele.  3.02.09

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/stat.h>

//tabeli kirje kirjeldus
struct loom{
      char key[32];    //nt. alligaator, kapsauss, ahv
      char nimi[10];   //nt. Muki või Miisu
      char karv[10];   //nt. must, valge, kirju
      char iseloom[10];  //hea, paha või "ei tea"
};

//põhijuurdepääs on välisaheldusega hash, ahela lüli kirjeldus
struct lyli{
    char key[32];       //eesel, kana, mammut
    struct loom *eluk;  //viit korrastamata tabeli kirjele ("reale")
    struct lyli *collision;  //põrkeviit või NULL
};

//põhijuurdepääsu võtmega pealisehitus: sortimispuu tipp
struct top{
         struct loom *rec;  //viit korrastamata tabeli kirjele ("reale")
         struct top *v;     //viit vasakule alluvale
         struct top *p;     //viit paremale alluvale
};

//lisajuurdepääsude (lubatud on korduvad võtmeväärtused) ahelate lüli
struct kett{
   struct loom *rec;    //viit korrastamata tabeli kirjele ("reale")
```

```
   struct kett *next;   //viit järgmisele või NULL
};

//lisajuurdepääs loomade "karva" järgi: otsimispuu tipp
struct top1{
        char *karv;        //dubleerib välja loom->karv
        struct kett *lyli; //sama-"karva" loomade kirjeviitade ahela pea
        struct top1 *v;    //viit vasakule alluvale
        struct top1 *p;    //viit paremale alluvale
};

//lisajuurdepääs loomade iseloomu järgi: ahela lüli
struct link{
    char *iseloom;     //dubleerib välja loom->iseloom
    struct kett *list;   //sama iseloomuga loomade kirjeviitade ahela pea (LIFO)
    struct link *next; //viit järgmisele lülile või NULL
};

//lisajuurdepääs loomade nimede järgi: ahela lüli
struct link_nimi{
    char *nimi;          //dubleerib välja loom->nimi
    struct loom *rec;  //viit korrastamata tabeli kirjele ("reale")
    struct link_nimi *next;  //viit järgmisele lülile või NULL
};

struct loom *T;      //tabel
struct loom *kirje;  //tabeli kirje

FILE *tabel=NULL;     //tabeli-faili "pide"
struct lyli *HT[13];  //pealeehitatava põhijuurdepääsu paisktabeli deklareerimine
struct stat stbuf;    //<sys/stat.h> :: siit saame tabeli ketta-atribuudi: pikkus
char kRida[32];      //pseudo-käsurida
int pikkus;           //tabeli pikkus baitides
int N;               //kettalt loetud kirjete arv
unsigned int key;     //pakitud hash-tabeli võti
struct lyli *uus,*L;   //põhijuurdepääsu hash-tabeli lülid
char t[20];              //dialoogi-sisendpuhver
struct top *juur=NULL;    //põhijuurdepääsu dubleeriva sortimispuu juur
struct top1 *root=NULL;  //lisajuurdepääsu (võti=karv) sortimispuu juur
struct link *IL=NULL;    //lisajuurdepääsu (võti=IseLoom) ahela pea
struct link_nimi *LN=NULL;  //lisajuurdepääsu (võti=nimi) ahela pea


//tekstilise võtmeväärtuse *t pakkija märgita int-arvuks k'
unsigned int text_to_key(char *t){
```

```
    int i;
    unsigned int k=0;
    while(*t != '\0'){
       for(i=0;i<4 && *t !='\0';i++,*t++)k^=*t<<(3-i)*8;
       }
    return(k);
}

//jagamismeetodi paiskfunktsioon: a=k'/n
int div_hash(unsigned int k, int n){
   return(k%n);
}

//korrastamata tabeli peale põhijuurdepääsu-paisktabeli (HT) ehitaja
void make_HT(void){
   int i,a;

   for(i=0;i<13;i++) HT[i]=NULL;  //HT pikkus on "sisse programmeeritud"
   for(i=0;i<N;i++){
      kirje=&T[i];              //"kirje" väärtuseks saab tabeli rea aadress
      key=text_to_key(kirje->key);  //tekstiline võti --> pakitud int-väärtus
      a=div_hash(key,13);   //a=paiskaadress
      L=HT[a];     //L on lähtepaiskaadressil olev viit ahelale või NULL
      if(L!=NULL){         //koht on hõivatud: põrge või korduv võtmeväärtus
         on: if(strcmp(kirje->key,L->key)==0){   //sama, viga
            printf("\nvõti %s kordub",kirje->key);
            goto next;
            }
          if(L->collision!=NULL){   //vaatame põrkeahela läbi
            L=L->collision;
            goto on;
            }
         uus=malloc(sizeof(struct lyli));  //lisan põrkeahelasse uue lüli
         memset(uus,'\0',sizeof(struct lyli));
         strcpy(uus->key,kirje->key);
         uus->eluk=kirje;
         L->collision=uus;
         goto next;
         }
      //HT[a]==NULL: teen välisahela esimese lüli
      uus=malloc(sizeof(struct lyli));
      HT[a]=uus;
      memset(uus,'\0',sizeof(struct lyli));
      strcpy(uus->key,kirje->key);
      uus->eluk=kirje;
```

```
      next: ;
      }
}

//päring põhijuurdepääsu (välisaheldusega hash) kasutades. Dialoog.
int loom(void){
   int a;
ring: fflush(stdin);  //puhastan sisendpuhvri
     printf("\nloom= ");
     gets(t);  //sisestan klaviatuurilt otsitava "looma"
     if(strlen(t)==0) return(1);    //tühi "Enter" lõpetab dialoogi
     key=text_to_key(t);    //teisendan sisestatud tekstilise võtme
     a=div_hash(key,13);    //a on lähte-paiskaadress
     L=HT[a];
kason:
     if(L==NULL){
          printf("looma %s pole tabelis\n",t);
          goto ring;   //küsi veel
          }
     if(strcmp(t,L->key)==0){    //leidsin
          kirje=L->eluk;        //viit tabeli reale
          printf("%s: nimi on %s karv %s iseloom: %s\n",
                kirje->key,kirje->nimi,kirje->karv,kirje->iseloom);
          goto ring;  //küsi veel
          }
     L=L->collision;   //otsin põrkeahelast
     goto kason;
}

//teen uue tipu põhijuurdepääsu-otsimispuule
struct top *new(struct loom *e){
   struct top *t;
          t=malloc(sizeof(struct top));
          memset(t,'\0',sizeof(struct top));
   t->rec=e;
          return t;
}

//teen otsimispuu (OP) korrastamata tabeli T peale. Korduvad väärtused
//(kirje->rec==rec->key) on välistatud (põhijuurdepääsu poolt)
void make_OP(void){
    int i,r;
    struct top *tipp, *jooksev;
    struct loom *rec;
```

```
    for(i=0;i<N;i++){
      kirje=&T[i];       //viit tabeli i-nda rea algusele
      tipp=new(kirje);   //teen otsimispuu uue tipu
      jooksev=juur;      //otsima hakkan juurest alustades
      if(juur!=NULL) goto otsi;
      juur=tipp;         //puu oli tühi
      goto next;
otsi:  rec=jooksev->rec;  //viit korrastamata tabeli järjekordsele kirjele
      r=strcmp(kirje->key,rec->key);
      if(r<0){
         if(jooksev->v!=NULL){
           jooksev=jooksev->v;
           goto otsi;
           }
         jooksev->v=tipp;
         goto next;
         }
      if(jooksev->p!=NULL){
           jooksev=jooksev->p;
           goto otsi;
           }
         jooksev->p=tipp;
      next:;
      }
}

//teen uue tipu lisajuurdepääsu ("karv") -otsimispuule
struct top1 *new1(void){
   struct top1 *t;
        t=malloc(sizeof(struct top1));
        memset(t,'\0',sizeof(struct top1));
        return t;
}

//teen "karva"-lisajuurdepääsu otsimispuu. Sama võtmeväärtusega kirjed on
//"aheldatud"
void make_OP1(void){
    int i,r;
    struct top1 *tipp, *jooksev;  //lisajuurdepääsu puu tipp
    struct loom *rec;   //viit korrastamata tabeli järjekordsele kirjele
    struct kett *uus;   //samade võtmeväärtustega kirjete ahela pea

    for(i=0;i<N;i++){
      kirje=&T[i];
      uus=malloc(sizeof(struct kett));  //kirjete ahela lüli
```

```
        uus->rec=kirje;
        uus->next=NULL;
        jooksev=root;
        if(root!=NULL) goto otsi;
        root=tipp=new1( );     //teen otsimispuu juure
        root->karv=kirje->karv;
        root->lyli=uus;
        goto next;
otsi:   r=strcmp(kirje->karv,jooksev->karv);
        if(r==0){    //lisan lüli LIFO-stacki
             uus->next=jooksev->lyli;
             jooksev->lyli=uus;
             goto next;
             }
        if(r<0){
            if(jooksev->v!=NULL){
              jooksev=jooksev->v;
              goto otsi;
              }
            tipp=new1( );
            tipp->karv=kirje->karv;
            jooksev->v=tipp;
            tipp->lyli=uus;
            goto next;
            }
        if(jooksev->p!=NULL){
            jooksev=jooksev->p;
            goto otsi;
            }
//olukord, kus jooksev->p==NULL & r>0.
        tipp=new1( );
        tipp->karv=kirje->karv;
        jooksev->p=tipp;//
        tipp->lyli=uus;
        next: ;
        }
}

//"Inorder: ->v j ->p" Põhijuurdepääsu kasutav, järjestatud kirjete väljastamine
void inovp(struct top *t){
    struct loom *e;
          if(t->v != NULL) inovp(t->v);
          e=t->rec;
          printf("%s: %s %s %s\n",e->key,e->nimi,e->karv,e->iseloom);
          if(t->p != NULL) inovp(t->p);
```

```
}

//"Inorder: ->v -> j ->p", lisajuurdepääs, trükib sama võtmeväärtusega kirjete
//primaarsed võtmed.
void inovp1(struct top1 *t){
   struct loom *e;
   struct kett *ahel,*rec;

   if(t->v != NULL) inovp1(t->v);
   ahel=t->lyli;
   printf("karv=%s :: ",t->karv);
tr: e=ahel->rec;
   printf("%s, ",e->key);
   ahel=ahel->next;
   if(ahel!=NULL) goto tr;
   printf("\n");
   if(t->p != NULL) inovp1(t->p);
}

//teen iseloomude ahela
void make_IL(void){
    int i,r;
    struct loom *rec;    //viit korrastamata tabeli järjekordsele kirjele
    struct link *zveno;  //IL-ahela lüli (iseloom, kirjete ahel, järgmine lüli)
    struct kett *uus;    //kirjete ahela (LIFO) pea

    for(i=0;i<N;i++){
      kirje=&T[i];
      uus=malloc(sizeof(struct kett));
      uus->rec=kirje;
      uus->next=NULL;
      zveno=IL;;
      if(zveno!=NULL) goto otsi;
      IL=zveno=malloc(sizeof(struct link));
      zveno->iseloom=kirje->iseloom;
      zveno->list=uus;
      zveno->next=NULL;
      goto next;
otsi:  r=strcmp(kirje->iseloom,zveno->iseloom);
      if(r==0){
          uus->next=zveno->list;
          zveno->list=uus;
          goto next;
          }
       if(zveno->next!=NULL){
```

```
            zveno=zveno->next;
            goto otsi;
            }
         zveno->next=malloc(sizeof(struct link));
         zveno=zveno->next;
         zveno->iseloom=kirje->iseloom;
         zveno->list=uus;
         zveno->next=NULL;
         next: ;
        }
}

//väljastab iseloomude ahelad
int tryki_IL(void){
    struct link *lyli;
    struct loom *eluk;
    struct kett *vv;

    lyli=IL;
ring:
    if(lyli==NULL) return(1);
    printf("\n\n%s -iseloomuga loomad:\n",lyli->iseloom);
    vv=lyli->list;
pr:  eluk=vv->rec;
    printf("%s ",eluk->key);
    vv=vv->next;
    if(vv!=NULL) goto pr;
    lyli=lyli->next;
    goto ring;
};

//teen nimedede ahela: pistemeetodil sorteerimine
void make_LN(void){
    int i,r;
    struct link_nimi *eile,*nyyd,*homme; //IL-ahela lülid: eelmine, jooksev,
                                         //järgmine

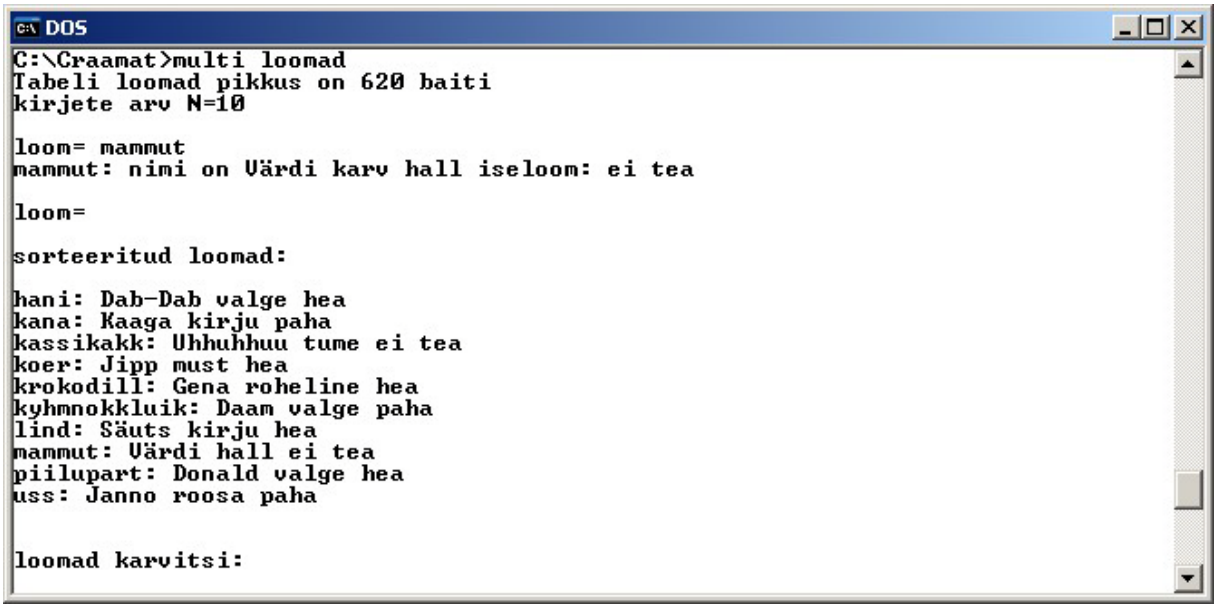
    for(i=0;i<N;i++){
      kirje=&T[i];
      nyyd=malloc(sizeof(struct link_nimi));
      nyyd->nimi=kirje->nimi;
      nyyd->rec=kirje;
      nyyd->next=NULL;
      eile=LN;
      if(eile!=NULL) goto start;
```

```
        LN=nyyd;
        goto next;
start:  r=strcmp(nyyd->nimi,eile->nimi);
        if(r<0){
          nyyd->next=eile;
          if(eile==LN) LN=nyyd;
          goto next;
          }
        homme=eile->next;
        if(homme==NULL){
            nyyd->next=eile->next;
            eile->next=nyyd;
            goto next;
            }
        r=strcmp(nyyd->nimi,homme->nimi);
            if(r<0){
              eile->next=nyyd;
              nyyd->next=homme;
              goto next;
              }
           eile=eile->next;
           goto start;
         next: ;
        }
}

//väljastab nimede ahelad
int tryki_LN(void){
    struct link_nimi *lyli;
    struct loom *eluk;
    lyli=LN;
ring:
    if(lyli==NULL) return(1);
    eluk=lyli->rec;
    printf("%s on %s\n",eluk->nimi,eluk->key);
    lyli=lyli->next;
    goto ring;
};

//multi.exe käivitatakse nii: ..>multi loomad
int main(int argc,char **argv){
   int i,a;
   if(argc!=2){
     printf("missing table");
     abort( );
```

```
    }
  tabel=fopen(argv[1],"r");
  sprintf(kRida,argv[1]);
  stat(kRida,&stbuf);    //anna faili-info
  printf("Tabeli %s pikkus on %d baiti\n",kRida,stbuf.st_size);
  pikkus=stbuf.st_size;
  N=pikkus/sizeof(struct loom);
  printf("kirjete arv N=%d\n",N);
  T=(struct loom *)malloc(N*sizeof(struct loom));
  fread(T,pikkus,1,tabel);
//teen põhijuurdepääsu paisktabeli
  make_HT( );
//päringud loomipidi
  loom( );
//teen lisajuurdepääsu: loomade otsimispuu
  make_OP( );
//väljastan tabeli võtmeväärtuste kasvavas järjekorras
  printf("\nsorteeritud loomad:\n\n");
  inovp(juur);
```

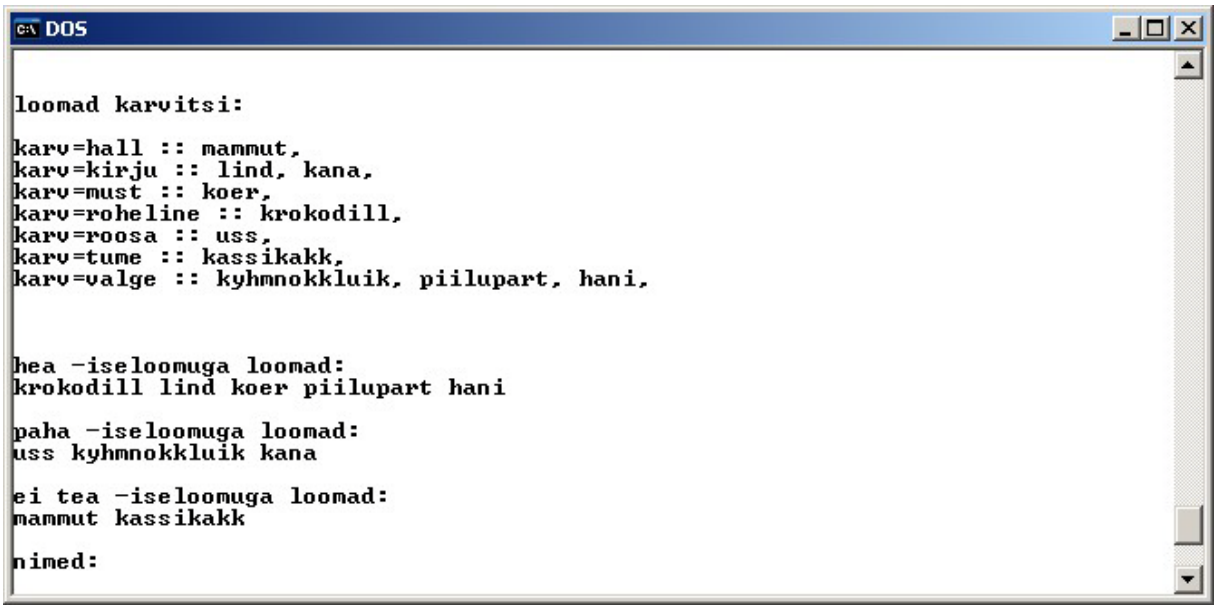


Joonis 12.7.a. Päring ja järjestatud väljastamine põhijuurdepääsu järgi.

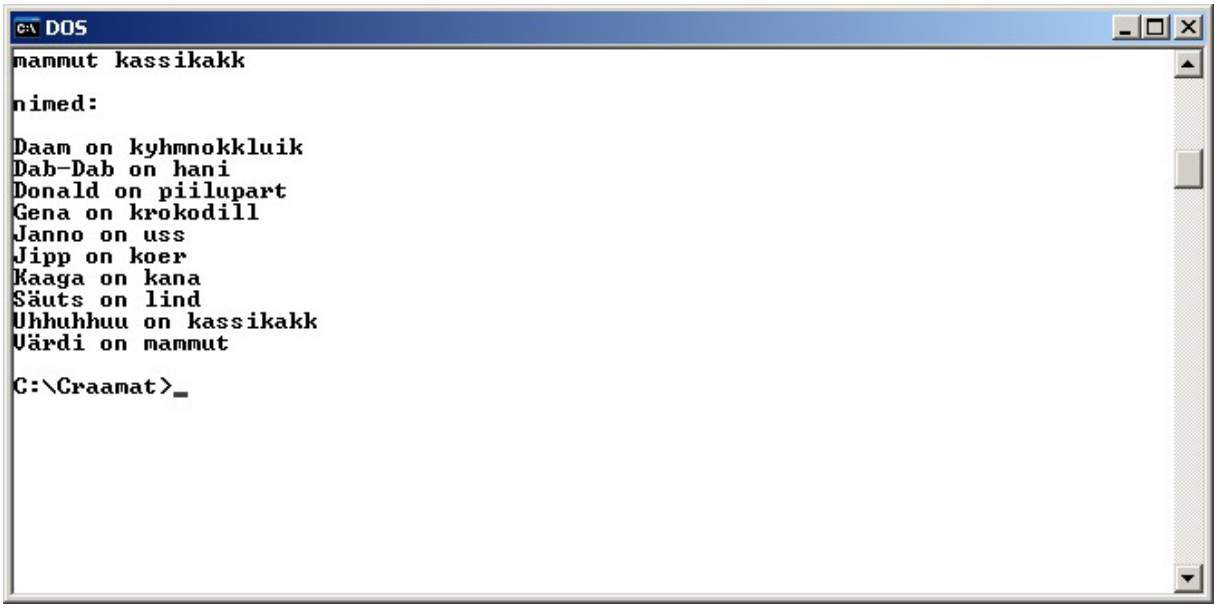
```
//väljastan värvitsi
  printf("\n\nloomad karvitsi:\n\n");
  make_OP1( );  //teen "karva"-puu
  inovp1(root);
//väljastan iseloomutsi
  make_IL( );
  printf("\n");
  tryki_IL( );
  printf("\n");
```

```
    printf("\nnimed: \n\n");
    make_LN( );
    tryki_LN( );
}
```

Multijuurdepääsu-teema lõpetamiseks nendime, et juurdepääsutabelite lisamist kitsendas eeskätt pikka aega suurimaks probleemiks olnud operatiivmälu maht. Kuni tabel (vaevu) sinna mahtus, siis tõsteti järjestamise käigus kirjeid tabelis „füüsiliselt“ ringi, kui ruumi oli pisut rohkem, siis „paisati“ kirjeid tabeli sees (ja vajadusel kirjutati ringi). Arusaadavalt polnud noil aegadel võimalust genereerida lisajuurdepääsu(de) tabelit või tabeleid – mälu oli niigi *liiga vähe*.

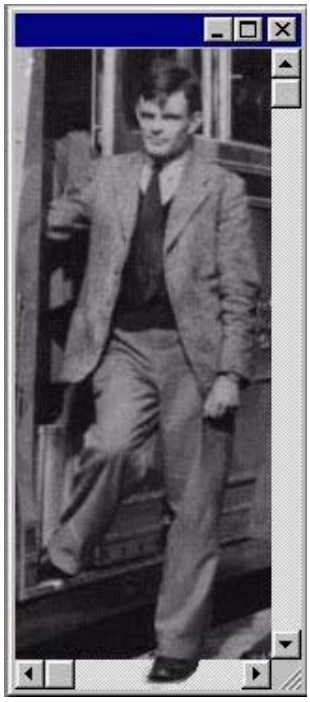


Joonis 12.7.b. Väljundid lisajuurdepääsude „karv“ ja „iseloom“ järgi.



Joonis 12.7.c . Väljund pistemeetodil järjestatud „nimede“ järgi.

Lisa 1. Turingi auhinna laureaadid 1966 – 2008[1]



Britt *Alan Mathison Turing* (23.06.1912 – 7.06.1954) on tunnustatud kui arvutiteaduse teerajaja. Eesti Entsüklopeedias [EE9, lk.625] on temast kirjutatud järgmist: „Esitas 1936 – 37 algoritmi mõiste kirjutuse abstraktse arvuti („*T. masina*“) kujul“ . Allikas [wAT] lisab, et „Teise maailmasõja ajal tegeles *Turing* Bletchley Parkis krüptograafiaga. Muuhulgas osales ta sakslaste *Enigma* koodi murdmisel. Pärast sõda projekteeris ta Londonis Rahvuslikus Füüsikalaboratooriumis ühe esimese programmeeritava elektronarvuti. Tema artikkel "*Computing machinery and intelligence*“ oli teedrajav tehisintellekti alal.

1951. astal valiti *Turing* Londoni Kuningliku Seltsi liikmeks.

Niisiis, *Alan Turing* evib teeneid järgmistes arvutiteaduse valdkondades:

- algoritmiteooria, abstraktsed arvutid;
- krüptograafia;
- elektronarvutite arhitektuur ja programmeerimine;
- tehisintellekt.

Joonis L1.a. *Alan Turing*.

Kõike seda arvestades asutati just tema nimeline teadusauhind arvutiteaduse alal, olles omataoliste hulgas mainekaim[2]. Auhinna määrab mõjuka ühenduse *Association for Computer Machinery*[3] spetsiaalne komitee.

Tänaseks on välja jagatud 43 *Turing*i auhinda, aastatel 1966 kuni 2008 (vt. [wTuring]). Allpool esitame laureaatide nimekirja valdkonniti; hoiatame, et see on mõneti meelevaldne[4].

Turingi auhinnad valdkonniti 1966 – 2008

Transleerimine

1. 1966 *Alan J. Perlis* (kompilaatorite konstruktsioon)

[1] Vt. [wTuring]. Selle lisa olemasolu peaks õigustama tõik, et paljusid autasustatuid kohtame selle raamatu lehekülgedel. Ja enamikku ülejäänuist muude kursuste materjalidega tutvudes – paraku, tihtipeale teadmata nende positsiooni teadusmaailmas.

[2] Mitmed allikad võrdlevad *Turing*i auhinda *Nobel*i premiaga, ent seda saab teha vaid väljavalitu au vaatevinklist: esmakordselt maksti laureaadile raha alles tunamullu, $250.000; rahastajateks hakkasid *Intel Corporation* ja *Google Inc.*

[3] See organisatsioon annab mh. välja ajakirja *Communications of ACM.*

[4] Meelevaldsus avaldub enamasti siis, kui laureaat kvalifitseerub mitmes valdkonnas; neil puhkudel on rubriiki paigutamine põhjendatud eeskätt sellega, millest auhinna määrajad on juhindunud. Näiteks, kas *N. Wirth* on programmeerimiskeelte või transleerimismeetodite autoriteet (Turingi auhinna sai ta ikkagi *EULER*i, *ALGOL-W*, *MODULA* ja *PASCAL*i eest 1984. aastal). Ent: teame ka ta teeneid *eelnevusmeetodi* formaliseerimise (so, transleerimise) vallas.

2. 1978 *Robert W. Floyd* (süntaksi analüüs, programmeerimiskeelte semantika, verifitseerimine, algoritmide analüüs, programmide süntees)
3. 2006 *Frances E. Allen* (optimeerimine, paralleelprotsessing)

Riistvara

1. 1967 *Maurice V. Wilkes* (arvuti *EDSAC*)
2. 1970 *James H. Wilkinson* (lineaaralgebra, vigade analüüs kiiretel numbrilistel arvutitel)
3. 1987 *John Cocke* (*RISC*-arhitektuur)
4. 1992 *Butler W. Lampson* (hajuskeskkonnad)
5. 1997 *Douglas Engelbart* (interaktiivsed meetodid)
6. 2004 *Vinton G. Cerf, Robert E. Kahn* (internetikeskkond, *TCP/IP*-protokoll)

Krüptograafia, loogika, keerukus

1. 1968 *Richard Hamming* (automaatkodeerimine, vigade avastamise ja parandamise koodid)
2. 1976 *Michael O. Rabin, Dana S. Scott* (mittedeterminineeritud automaadid)
3. 1982 *Stephen A. Cook* (ajaline keerukus)
4. 1991 *Robin Milner* (*LCF, ML, CCS*: operatsiooni- ja denotatsioonsemantika seosed)
5. 1993 *Juris Hartmanis, Richard E. Stearns* (ajalise keerukuse teooria)
6. 1995 *Manuel Blum* (keerukuse ja krüptograafia seosed)
7. 1996 *Amir Pnueli* (programmide tõestamine)
8. 2000 *Andrew Chi-Chih Yao* (keerukus ja krüptograafia)
9. 2002 *Ronald L. Rivest, Adi Shamir, Leonard M. Adleman* (avalik võti)
10. 2007 *Edmund M. Clarke, Allen Emerson, Joseph Sifakis* (efektiivne süsteemide kontroll)

Tehisintellekt

1. 1969 *Marvin Minsky* (TI)
2. 1971 *John McCarthy* (TI; meie jaoks: *Lisp*)
3. 1975 *Allen Newell, Herbert A. Simon* (TI propageerimine)
4. 1994 *Edward Feigenbaum, Raj Reddy* (TI propageerimine)

Algoritmid ja andmestruktuurid, programmeerimiskeeled

1. 1972 *Edsger W. Dijkstra* (*Algol*, artiklid graafiteooriast ja programmeerimiskeelte filosoofiast)
2. 1974 *Donald E. Knuth* („*The Art of Computer Programming*“)
3. 1977 *John Backus* (*FORTRAN, BNF*)
4. 1979 *Kenneth E. Iverson* (*APL*)
5. 1980 *C. Antony R. Hoare* (programmeerimiskeelte disaini põhimõtted)
6. 1983 *Ken Thompson, Dennis M. Ritchie* (*UNIX*)
7. 1984 *Niklaus Wirth* (*EULER, ALGOL-W, MODULA, PASCAL*)
8. 1985 *Richard M. Karp* (ajaline keerukus, võrgualgoritmid)
9. 1986 *John Hopcroft, Robert Tarjan* (algoritmide ja andmestruktuuride uurimine)
10. 1988 *Ivan Sutherland* (arvutigraafika)
11. 1989 *William Kahan* (ujupunkt-arvutused)
12. 1990 *Fernando J. Corbató* (*CTSS* ja *Multics*)
13. 1999 *Frederick P. Brooks, Jr.* (*OS/360*)
14. 2001 *Ole-Johan Dahl, Kristen Nygaard* (*Simula*)
15. 2003 *Alan Kay* (*Smalltalk*)

16. 2005 *Peter Naur* (*ALGOL-60, BNF*)
17. 2008 *Barbara Liskov* (andmeabstraktsioonid, hajusarvutused)

Andmebaasid

1. 1973 *Charles W. Bachman* (*Bachman*i skeemid)
2. 1981 *Edgar F. Codd* (relatsiooniline mudel)
3. 1998 *Jim Gray* (infosüsteemid)

Mõned märkused. **Esiteks**, auhinna määramise aasta ning laureaatide auhindamisväärseks tunnistatud töö(de)ga tegelemise aastaid võrreldes tuleb tõdeda, et *ACM* tegutseb üldjuhul samamoodi nagu *Nobeli* preemia omistamise komisjonid: tööpanuse tegeliku kaalu hindamine nõuab aastatepikkust ootamist ning jälgimist. Toogem mõned näited:

M. V. Wilkes ehitas *EDSAC*i 1949. ja auhinna sai 1967. aastal, *J. Backus*e *FORTRAN*i (1954) ja auhinna teenimise (1977) vahel oli 23 aastat, *Brooks Jr.* juhtis *OS/360* tegemist 1950-ndate lõpus ning pärjati 1999. aastal, *Dahl* ja *Nygaard* „ootasid" tunnustust 1967. aastast kuni 2001. aastani, *Alan Kay* (*Smalltalk*) „tegi" 1971. a. ja auhinna sai 2003, ja senini on ületamatu *Peter Naur*i ajanihe: *Algol*-60 ja *BNF* on dateeritavad 1960. aastasse, ja *Turingi* auhinna sai ta 2005. aastal.

Teiseks, auhinnatute seas on mitmeid arvutiteaduse üldtunnustatud autoriteete, kelle konkreetset valituksosutumise „põhjust" on raske formuleerida, ent kelle auhinnatute hulka valimine tundub täiesti loomulikuna, isegi vältimatuna, näiteks *Dijkstra, Knuth, Floyd, Hoare* või *Wirth*.

Kolmandaks, üsna tihti on *ACM*i komisjon teinud suhteliselt kiire ja meie retrospektiivis „õige" otsuse: *McCarthy, Codd, Thompson* ja *Ritchie* või *Cocke.*

Lõpuks, mõneti kahtlane on, kas kümmekond aastat hiljem oleks näiteks *Bachman* (hierarhilise andmemudeli visualiseerimine, avaldatud 1969, auhind 1973) väljavalituks osutunud. Ja nende ridade autori arvates sobinuks *Turing*i auhinna laureaatide hulka väga hästi ka *Chuck Moore* (*Forth*) või *Alfred Aho* ja *Jeffrey D. Ullman* (süntaksorienteeritud transleerimine) või *Andrei Jeršov.*

Aga loodetavasti said lugejad (meie tudengid, eeskätt) ettekujutuse arvutiteaduses valitsevatest aktsepteeritavatest trendidest ning sellest võib kasu olla oma suuna valimisel: *kus* on suurim šanss valituks osutuda? Ja vastupidi: *kus* pole siiani kedagi sõelale jäänud? Miks mitte ei võiks olla selleks *mina* esimesena?

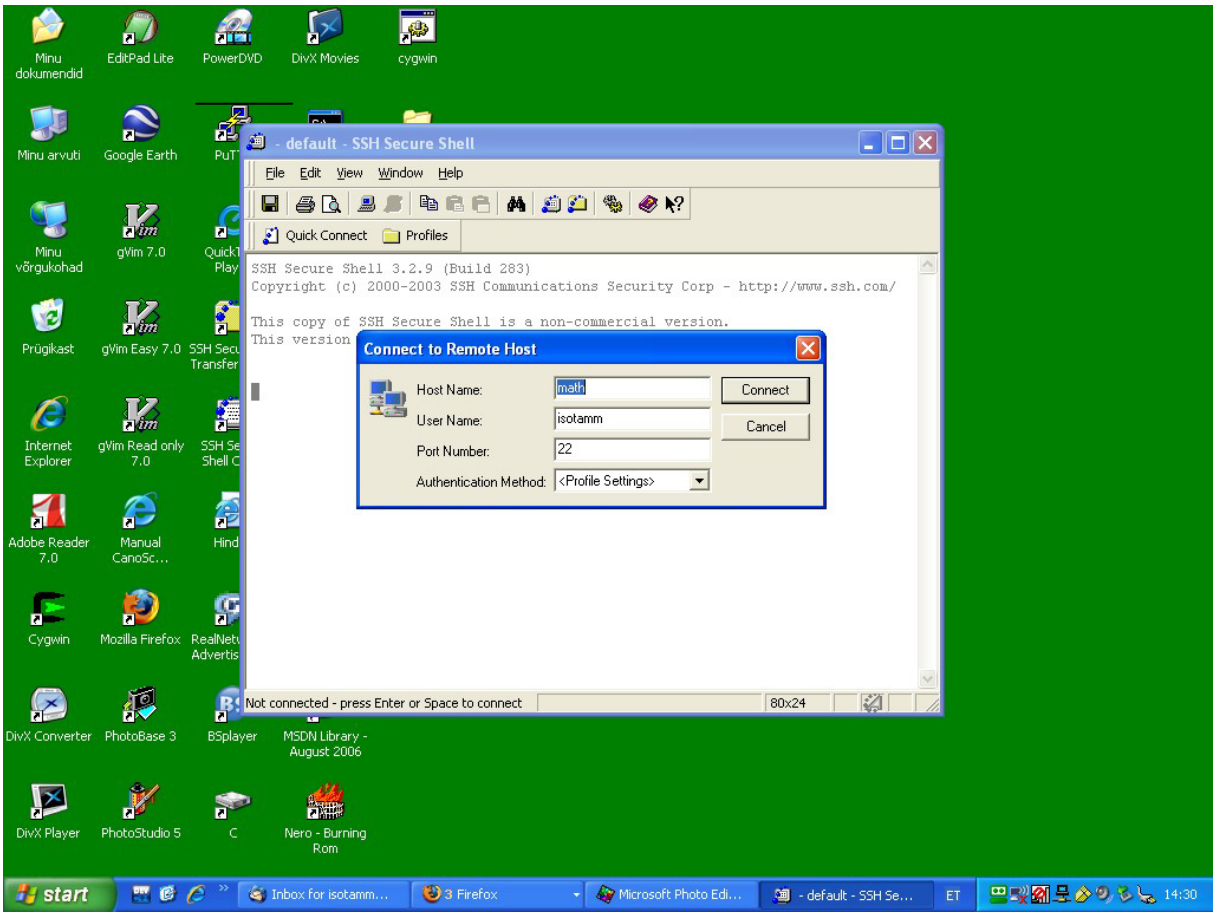
Lisa 2. Mõned töökeskkonnad

Praktikumides töötame *UNIX*-keskkonnas, ja igal õppuril on vaba voli valida endale olemasolevate hulgast meelepäraseim variant. Meie praktikumikohas (J. Liivi 2–004) on valikuteks *C*-kompilaatorid *gcc* kolme pealisehitusega: kaks ilmutatud kujul *UNIX*-orienteeritut (sisenemine *SSH Secure Shell Client*-ikooni abil või *Cygwin*iga), kolmas (*Dev-C++*) kujutab endast *Windows*i ja *UNIX*i liidest. Mistahes variandi puhul saab kasutada igaüks omale eraldatud ruumi *math*-serveris.

gcc

Sisenemiseks kasutame ikooni *SSH Secure Shell Client*. Töö alustamiseks tuleb esiteks sisse logida (vt. joonis L2.a). Valime menüüst nupu *Quick Connect*, mispeale avaneb dialoogiaken *Connect to Remote Host*. Väljale *Host Name* sisestame *math* ning väljale *User Name* oma kasutajanime (mis on meil nii e-meili kasutamiseks kui ka ÕISi sisenemiseks) ning vajutame *Connect*.

Järgmisena avaneb parooliaken – kirjutame sinna oma kasutajanimega seotud salasõna (vt. joonis L2b). Kui kõik on korras, siis avaneb *UNIX*-keskkond (vt. joonis L2.c). Süsteem ootab käsurealt korraldusi.



Joonis L2.a. *Quick Connect.*

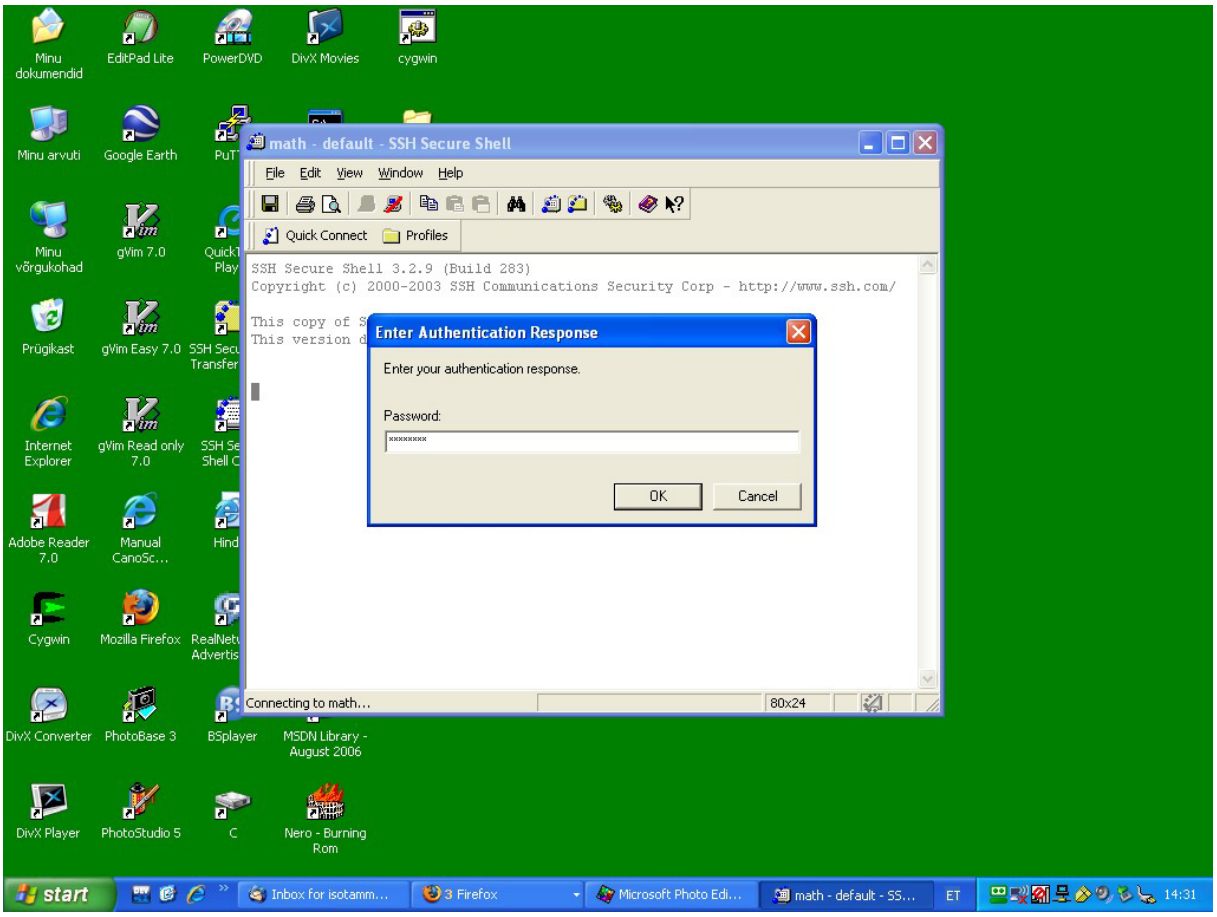
Joonisel L2.d on käsuga *ls* ekraanile toodud kasutaja *isotamm* kausta sisu. Suvalises keskkonnas töötamisel on oluline saada infot süsteemsete võtmesõnade kohta. *UNIX*is on selleks kaks võimalust:

man <võtmesõna> või
info <võtmesõna>

Kui *man* näitab manuaalist sellist alajaotust, mis ei mahu ühele ekraanile, siis järje vaatamiseks tuleb vajutada „pikka latti“ – *tühik* (*space*). Manuaali vaatamise katkestab klahv *q*.

Joonisel L2.e on käsu *man pico* resultaat. *Man* on sõna „manuaal“ prefiks, *pico* aga primitiivne tekstitoimeti, millega on mugav oma *C*-programme sisestada ja parandada. *Pico* käivitamiseks tuleb anda käsk *pico <nimi>.c*; kui sellenimelist faili kettal polnud, siis see luuakse, muidu avatakse. Teksti salvestamiseks tuleb kõigepealt sisestada *Ctrl+O*, selle peale küsib *pico* ekraani alaservas *save File Name to write:* <failinimi.c>, kinnituseks tuleb vajutada *Enter*; *pico*st väljumiseks tuleb sisestada *Ctrl+X*. *Pico* võimalustest saab ülevaate ka tema aknast, vajutades *Ctrl+H*.

NB! *C*-programmide nimelaiend peab olema kindlasti *.c*. Kui kirjutada *.C*, siis kompilaator peab seda teksti *C++*-programmiks – mis aga on suure tõenäosusega paljude sisuliselt olematute süntaksivigade allikas.



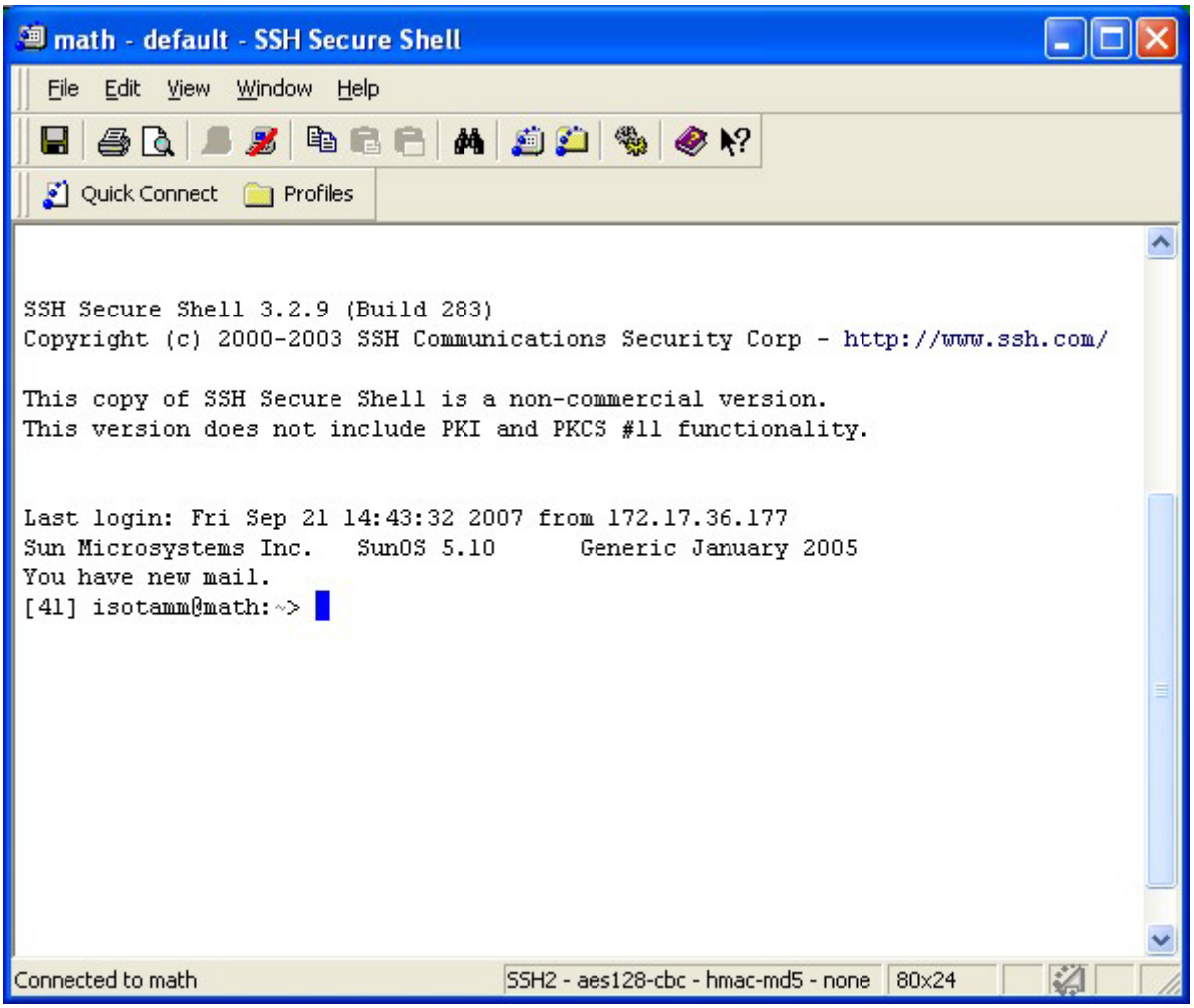
Joonis L2.b. *Password.*

***C*-programmi** kirjutades tuleb järgida teksti jäika struktuuri.

- Hea stiil on alustada teksti kommentaariga (markerid /*…*/ või // kuni reavahetuseni), näiteks nii:

  /* programm myfirst.c, alustatud 20.09.07. Loeb sisse sümbolite vektori, järjestab selle leksikograafiliselt („tähestiku järjekorras“) ja prindib ekraanile */.
- Makrod *#include* (ja vajadusel *#define*).
- Kirjeldused: tavaliselt globaalsed muutujad ja alamprogrammide kirjeldused. Kui viimaseid pole, siis tuleb alamprogrammide tekste kirjutada nende kasutamise järjekorras. *Main*-moodulit eelkirjeldada ei saa.
- Alamprogrammide tekstid – esmalt lokaalsete muutujate kirjeldused ja seejärel operaatorid.
- *Main*-mooduli tekst – esmalt lokaalsete muutujate kirjeldused ja siis operaatorid. Lühemate programmide jaoks piisab ainult *main*-moodulist.

Hea tava on teksti kirjutamisel kasutada programmi struktuuri jälgimist hõlbustavat „treppimist“ *tab*-klahvi abil (madalama taseme operaatorite plokk kirjutatakse taandega).



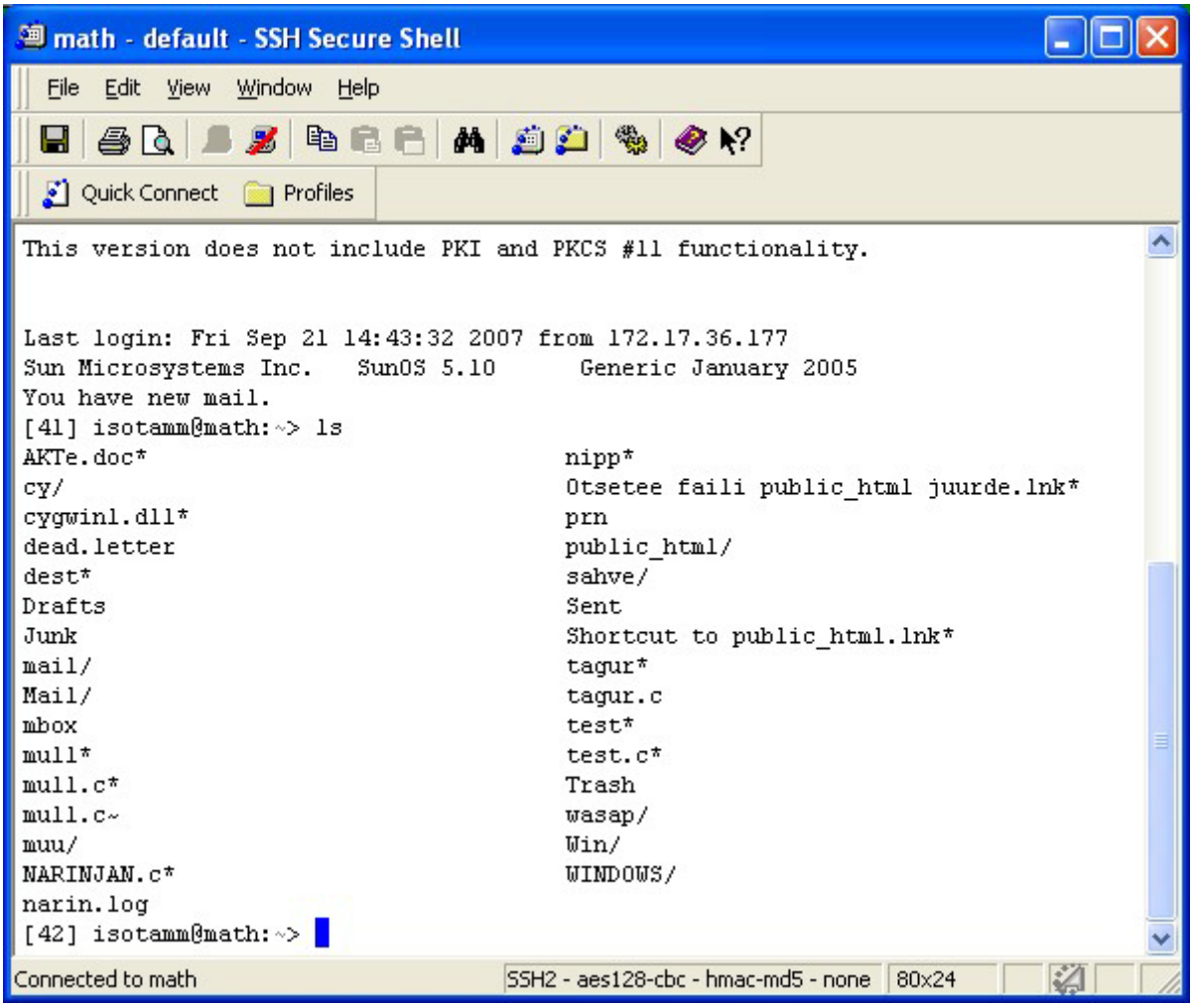
Joonis L2.c. *UNIX prompt.*

Praktikumides kasutame kompilaatorit *gcc* (*Gnu Compilers Collection*), mille töö juhtimise kõigist võimalustest võib saada ettekujutuse käsu *man gcc* abil. (Muide, *man* töötab tavaliselt ka siis, kui küsida *C* standardfunktsioonide kohta, proovige mäiteks *man printf*.)

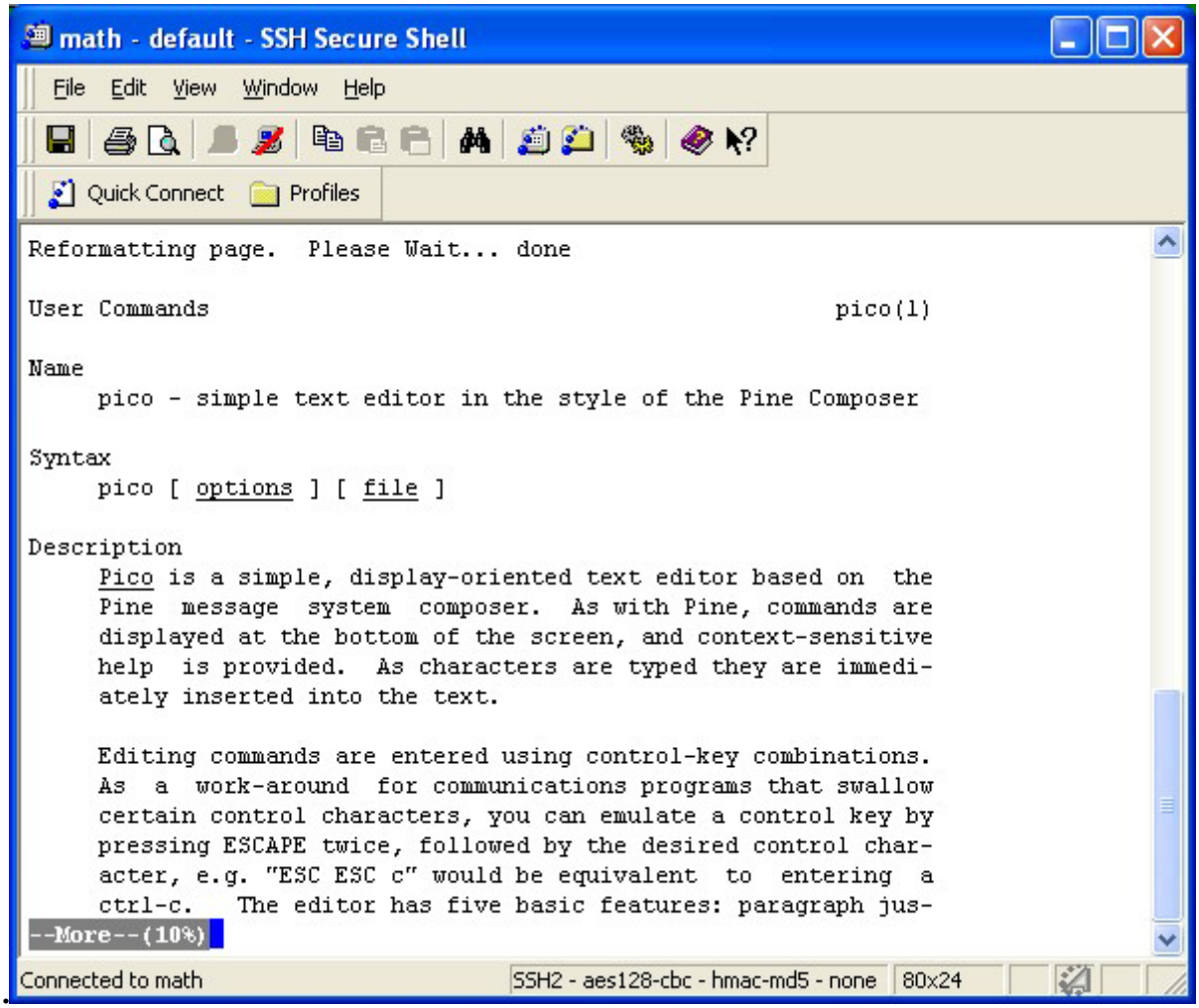
Kuni me töötame ühe-tekstifaili režiimis, on mugav(aim?) variant kaheparameetriline:

- *-o <laiendita nimi>* : objektfail, mida oskab käivitada *gcc* virtuaalarvuti;
- *<nimi>.c* : lähtefaili nimi, nimelaiend määrab kompilaatori (*C, C++, FORTRAN* vm. *C* jaoks *c*, *C++*le *C* või *cpp* või *cc* jne). Näiteks, kui tahame kompileerida programmi nimega *mull*, siis anname käsu g*cc –o mull mull.c*

Edukal kompileerimisel ei avastata süntaksivigu. Tavaliselt avastatakse, ja siis *gcc*-käsule järgnevatel ekraaniridadel on veateated: rea järjekorranumber ja vihje vea iseloomule. *Pico* paraku ei näita reanumbreid, mis võib teha süntaksivea lokaliseerimise vaevanõudvaks. Alternatiiviks võiks olla teise tekstitoimeti *vim* kasutamine – see *näitab* reanumbreid. Informatsiooni *vim*i kohta saab – nagu juba teame – päringuga *man vim* (või *info vim*). *Kui* meil õnnestus kompileerida süntaksivigadeta programm, siis saame ta käsurealt käivitada lihtsalt – näiteks, programmi *mull* (*gcc –o mull mull.c*) käsuga *mull.*



Joonis L2.d. Käsk *ls*.



Joonis L2.e. Käsuga *man pico* saadud informatsiooni avaleht.

Cygwin

Cygwin on *Windows*i jaoks loodud *UNIX*i (või *Linux*i) -sarnane keskkond, mis koosneb kahest osast:

- *DLL* (*cygwin1.dll*), mis emuleerib *UNIX*i/*Linux*i *API*-funktsionaalsust
- vajalike vahendite kogum, mis järgivad *UNIX*i/*Linux*i „väljanägemist ja hinge“ (*look and feel*).

Cygwin DLL töötab x86-protsessori nii 32-bitise kui ka 64-bitise versiooniga *Windows*i keskkonnas. Rõhutatakse, et *Cygwin* pole võimeline jooksutama *Linux*i (*UNIX*i) rakendusi *Windows*i all; kõik rakendused (meie jaoks *C*-programmid) tuleb *Cygwin*i (või *SSH* vmt.) keskkonnas eelnevalt (uuesti) transleerida. Ülaltoodu on pärit saidilt

http://www.cygwin.com/ (vaadatud 13.12.07). Kasulikuks võib osutuda järgmine viit: http://www.cygwin.com/faq/faq.using.html (12.12.07).

Ka selles keskkonnas saame kasutada *gcc*-kompilaatorit samasuguse käsureaga nagu *SSH*-versiooni puhul, näiteks:

gcc –o bw tagur.c

ent erinevalt *SSH*-st kompileeritakse *Cygwin*i kasutades fail *bw.exe* ning selle käivitamiseks peame pisut rohkem kirjutama (*prompt* on siin *$*):

$ ./bw

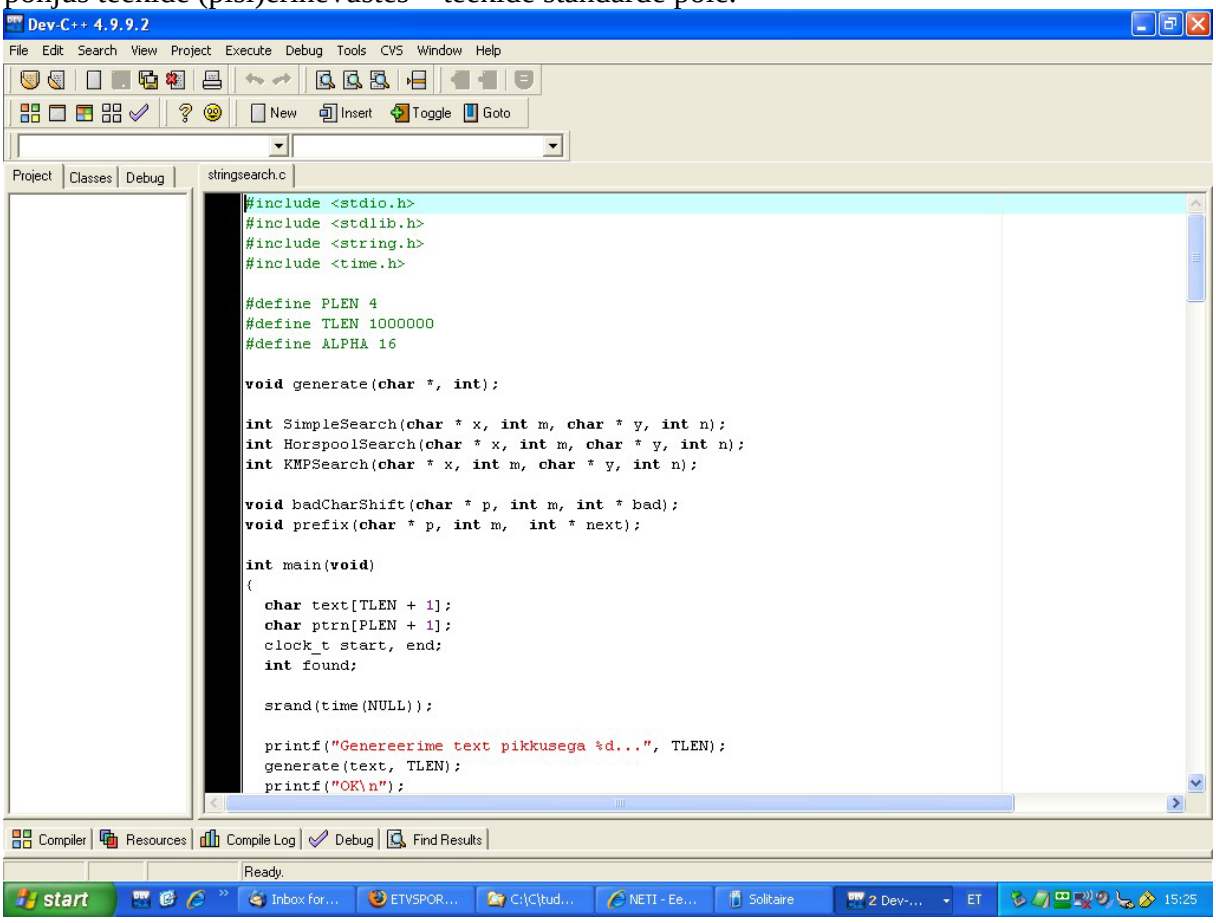
Tundub, et siin toimivad juba tuttavad asjad, *pico*, *man*, *info* jm. Keskkonda illustreerib ekraanipilt joonisel L2.f :

Joonis L2.f. *Cygwin.*

Dev-C++

Kolmas võimalus töötada simuleeritud *UNIX*-keskkonnas on kasutada firma *BloodshedSoftware* vabavaralist integreeritud arenduskeskkonda *Dev-C++*, mis on installeeritud vähemasti meie J. Liivi 2 – 004 arvutiklassi töökohtadele. Kompilaator (nimega *Mingw*) saab hakkama nii *C*- kui ka *C++*-programmidega; pakett on kasutajasõbralik pealisehitus *gcc*-le (*GNU Com-piler Collection*). Süsteem töötab järgmiste *Windows*i versioonidega: 95/98/NT/2000/XP ning on kaitstud *GNU* litsentsiga. Autorid on *Colin Laplace*, *Mike Berg*, *Hongli Lai*, *Mingw*-kom-pilaatori tegid *Mumit Khan*, *Jan Jaap van der Heidjen*, *Colin Hendrix* ja „*GNU coders*“ (allikas: http://www.bloodshed.net/devcpp.html vaadatud 6.12.07).

Allpool on joonis L2.g – slaid töökeskkonnast, avatud on *Dmitri Melnikov*i alamstringide otsimise programm. Nagu ka *Cygwin*, kompileerib *Dev-C .exe*-faili. Kogemused näitavad, et *Dev-C* on kapriissem keskkond kui „puhas“ *gcc* – võib juhtuda, et veatult kompileerunud programm „jooksis“ lahendamise ajal „kinni“, ent *gcc*-ga seda ei juhtunud. Arvatavasti on põhjus teekide (pisi)erinevustes – teekide standarde pole.[1]



Joonis L2.g. *Dev-C++*

[1] Kuid põhjus võib olla ka selles, et see projekt jäi lõpuni teostamata ning mõned asjad jäid seetõttu ka pooleli. Meie kogemus näitab, et kui *Dev-C* hakkab ebaadekvaatselt käituma, siis on abi programmi sulgemisest (ilma tekstifaili „näppimiseta“) ja uuesti käivitamisest.

Kasutamine on lihtne ja loogiline. Uue programmi kirjutamiseks liikuge **`File`**→**`new`**→ **`source file`**, kirjutage tekst ja salvestage „File"-menüüst „save as", seejärel valige **`Execute`** →**`Compile`**, kui leiti vigu, näete neid allservas avanenud vigade aknas (seal saate klõpsates liikuda kahtlasele tekstireale ja kohe näpukad ära parandada), kui vigu enam polnud, siis käivitage programm (silumiseks, esimeses lähenduses) **`Execute`**→**`Run`**. Kena on harvapakutav lisavõimalus programmi teksti väljatrükkimiseks – võite lasta lisada ridade järjekorranumbrid.

Ent, nagu juba ülalpool märgitud, programmi töö võib päädida pseudoveaga: pealisehituseta *gcc*-kompilaatorite väljundkood võib sama teksti „kompilaati" täita normaalselt

djgpp

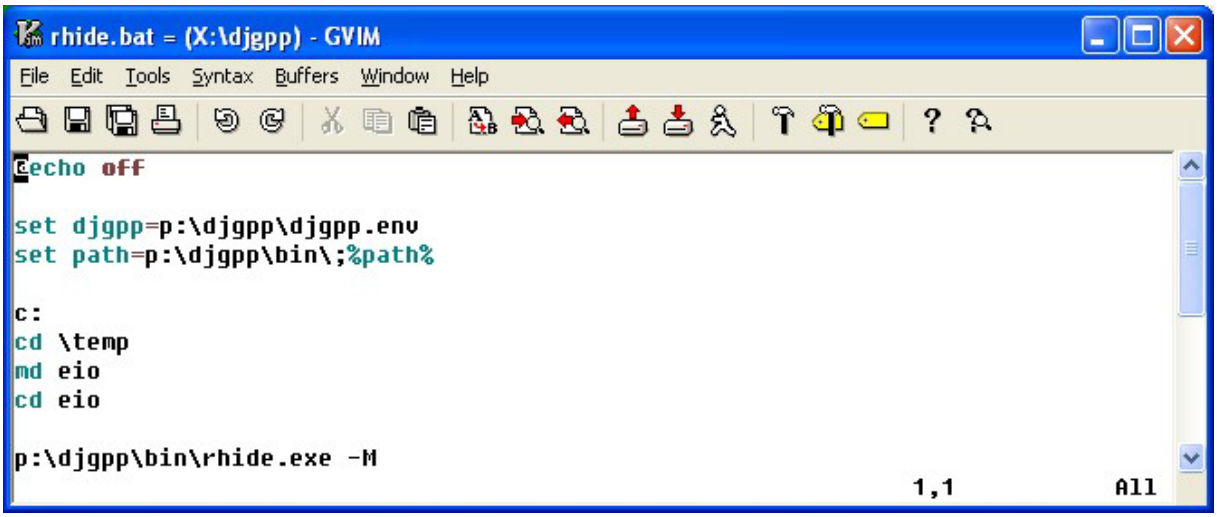
Järgmised kaks keskkonda on avalikult „*Microsoft*-orienteeritud". Meie jaoks on oluline, et needki on vabavaralised (ja evivad ahvatlevaid lisavõimalusi – juurdepääsu protsessori funktsioonidele). *Djgpp* on akronüüm sõnadest *DJ's Gnu Programming Platform*; *Gnu*-projekt algatati *MIT* egiidi all *Cambridge*'is (*Massachusetts*). Millegipärast (aga miks ka mitte!) valiti projekti nimeks targa näoga gnuu-antiloobi oma; joonisel L2.h olev pilt on saadud, kasutades linki http://en.wikipedia.org/wiki/Massachusetts_Institute_of_Technology.

Joonis L2.h. Gnuu-antiloop.

Djgpp-projekti alustati 1989. aastal ning ta hõlmab 32-bitisele protsessorile orienteeritud transleerimiskeskkondi keeltele *C, C++, ObjC, Ada* ja *Fortran* (operatsioonisüsteem on *Microsofti DOS* või mõni muu, mis suudab jooksutada *MS DOS exe-faile* – näiteks *Microsoft Windows* või *IBM OS*/2 (vt. http://en.wikipedia.org/wiki/DJGPP , vaadatud 11.08.09). Mainigem veel, et *DJ* tuleneb projekti eestvedaja *DJ Delorie* nimest vt. http://www.delorie.com/djgpp/ ja http://www.delorie.com/djgpp/doc/ug/intro/what-is-djgpp.html

Meie praktikumides on *djgpp* kättessadav *Samba* kaudu ning *C*-kompilaatori käivitamine „ehedas" *djgpp*-keskkonnas on üpris tülikas. Sestap soovitame kasutada *Robert Hoehne* (Saksamaa, Dittmannsdorf) tehtud liidest *RHIDE* (*Robert Hoehne Integrated Development Environment*), mille saame käivitada *Samba* teegist *djgpp*, valides *GCC CommandLine* ja seal *rhide*. See *bat*-fail teeb kasutaja-kettale teegi nimega *temp.eio* ning aktiveerib *Windows*i-eelse graafikaga akna, kus navigeerida saab vahel hiirega, vahel aga ainult klahvidega (nooled, Enter, funktsionaalklahvid *Fx*). Lihtprogrammide (ainult üks *.c*-ressurss) jooksutamine on lihtne: *C*-tekst viige tollesse *temp.eio*-teeki, käivitage taas *rhide*, valige (hiirega!) *File→open* ja sealt oma *.c*-fail ning valige *run*(→run). *RHIDE* teeb kõik: transleerib ja komplekteerib. Kui midagi oli süsteemi arvates pahasti, siis ekraani alaservas avatud uues aknas on vastavad teated. Kui aga kõik oli (süsteemi arvates) korras, siis uus *run→run* käivitab kompileeritud faili (*DOS-aknas*). Tagasi saab, sisestades sõna *exit*.

Djgpp-platvorm on oluline mõistmaks *C* kui süsteemiprogrammeerimise keele olemust. Nimelt annab see vahetu juurdepääsu *BIOS*- ja *DOS*-funktsioonidele ja seega vahenditele, mida *UNIX*- ja *Windows*-orienteeritud keskkonnad ja rakenduskeeled kasutaja eest peidavad. Selles rakenduses on kasutatavad teegid *dos.h* ja *bios.h.* Huviline saab leida muudki kasulikku, uurides teeke Solarise *djgpp* alt, näiteks, kui ta huvitub protsessi aja mõõtmisest.



Joonis L2.i. *rhide.bat* (avatud *gvim*iga).

DOS-näiteks kasutame kümnenditagust[1] arvutiarhitektuuri kirjeldust:

```
struct WORDREGS{
        unsigned int ax;
        unsigned int bx;
        unsigned int cx;
        unsigned int dx;
        unsigned int si;
        unsigned int di;
        unsigned int flags;
        };
struct BYTEREGS{
        unsigned char al,ah;   //WORDREGSi väljad ax...dx on REGSis ülekattes
        unsigned char bl,bh;  //BYTEREGSiga
        unsigned char cl,ch;
        unsigned char dl,dh;
        };
union REGS{
        struct WORDREGS x;
        struct BYTEREGS h;
        };
union REGS regs;
```

Madalaima taseme funktsiooni kasutamise näiteks toome *C*-programmi, mis trükib operatsioonisüsteemilt küsitud mälumahu:

[1] Tänast seisu saab vaadata teegist *djgpp* – väljade pikkusatribuudid pole enam need. Tänapäeval prevaleeriva 32-bitise arhitektuuri ajal on struktuur *WORDREGS* 32-bitiste *int*-komponentide kogum ja *BYTEREGS*i komponentide formaat on 16-bitine *short int.* (Slaidid *RHIDE*-ekraanidelt äpardusid: neid ei õnnestunud *Windows*-masinas klahvi *PrtSc* abil kopeerida.)

```
//djgpp evib DOS-väratit: dos.h ja bios.h
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>

#define MEM 0x12  //funktsiooni identifikaator: anna mälu maht

int main( ){
      union REGS regs;    //defineeritud: djgpp::dos.h
      unsigned int size;
      int86(MEM,&regs,&regs);
      size=regs.x.ax;
      printf("Memory size is %d Kbytes\n",size);
        getchar( );
        exit(0);
}
```

Funktsioonidega *int*86( ) ja *int*21( ) saame kasutada pea kõiki madalaima, so, protsessori taseme *BIOS*i ja *DOS*i funktsioone. Nende kirjeldused (ja vahel ka parameetrid) on kättesaadavad näiteks linkidelt :
http://www.htl-steyr.ac.at/~morg/pcinfo/hardware/interrupts/inte1at0.htm (15.12.07)
http://www.bioscentral.com/misc/interrupts.htm (6.12.07)
http://www.beyondlogic.org/interrupts/interupt.htm (6.12.07)

Huvilistele soovitame tutvuda *djgpp include*-failidega *dos.h*, *bios.h* ja *keys.h*, kus leiduvad kõikvõimalike liht- ja funktsionaalsete klahvide (*Ctrl*, *Alt*) kombinatsioonide koodid.

Turbo-C (*TC*)

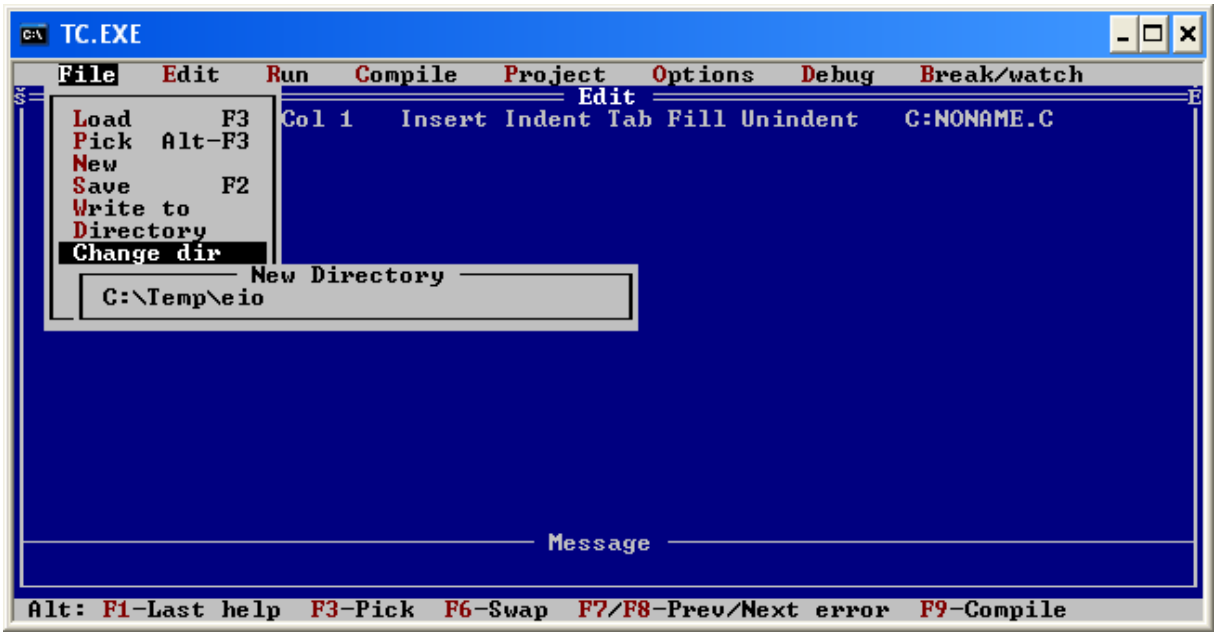
Süsteem on „allalaaditav“ saidilt http://www.pitt.edu/~stephenp/misc/downloadTC.html .

Vabavara. See keskkond võiks sobida neile, kes tahaks tunnetada „nostalgilist hõngu“: hiir ei funktsioneeri, kõik tehakse ära kursorijuhtimise klahvidega (nooled igas suunas, PgUp, PgDown, F-klahvid), aga funktsionaalsus on mugav. Projektifail on lihtne (ühe-programmi režiimis polegi teda vaja), ja kui ära harjuda, siis silumine on lihtsam kui enamarenenud keskkondades.

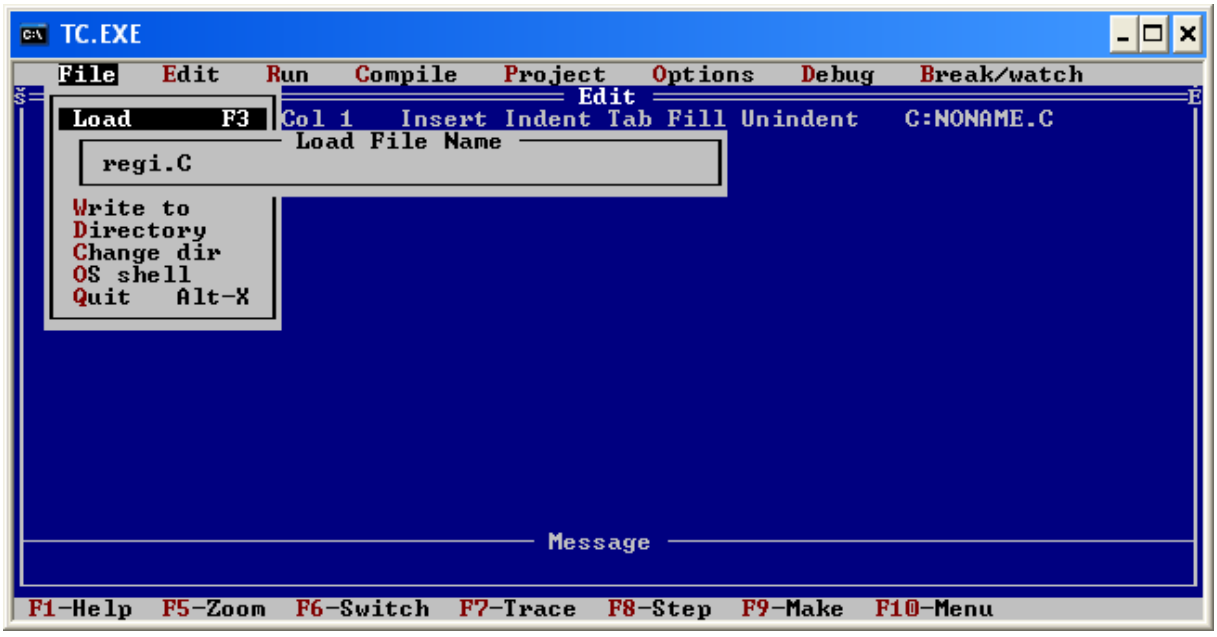
Navigeerimine on võimalik rippmenüüde esitähtede (punased), kindlam aga F10 vahendusel.

TC toetab ligipääsu protsessori funktsioonidele (so. omab päisefaile *dos.h* ja *bios.h*) – just niisamuti nagu *djgpp*. *NB*! *TC* ei saa aru „moodsamast kommentaaristiilist“ – kui kasutate „//“, siis saate hulganisti arusaamatuid veateateid.

Järgnevad slaidid joonistel L2.j ja L2.k juba tuttava programmi *regi.c* töötlemisest *TC* keskkonnas.



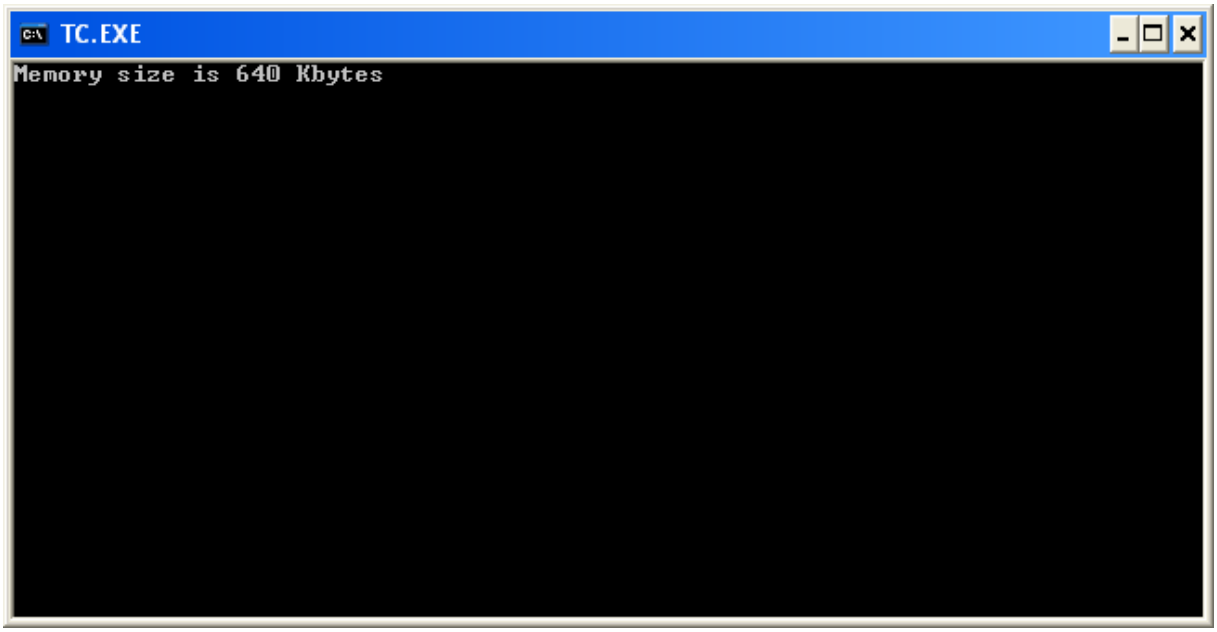
Joonis L2.j. *TC*: faili lokaliseerimise algus.



Joonis L2.k. Faili nime sisestamine.

Seepeale tuleb *F10* abil saada uuesti kontrolli alla juhtmenüü ja valida rippmenüü *Compile*:

Joonis L2.l. Kompileerimine.



Joonis L2.m. *F10 →Run→Run.*

Ilmselt on tulemuseks saadud mälumaht väär – tuletab meelde *PC AT* maksimumi, aga – see pole antud juhul kuigi oluline (usutavasti kasutab *TC union regs*i jaoks 16-bitist kirjeldust ja nii jäävad kõrgemad järgud „mängust välja“).

Lisa 3. Bootstrapping

Bootstrap on inglise keeles „saapatripp“, aas saapasääre tagaküljel hõlbustamaks jalgatõmbamist; inglased kasutavad seda sõna ka ülekantud tähenduses: midagi tegema lihtsalt ja ise (vt. [Webster's], lk. 191). Arvutiteaduses on sel terminil kaks tähendust: esiteks on see *alg-laadur*, lühike programm (mikroarvutitel püsimälus), mis laadib tegeliku laaduri operatsioonisüsteemi või rakenduse järgnevaks laadimiseks (vt. [Tavast&Hanson], lk. 26). Teiseks kasutatakse *bootstrapping*ut tähistamaks programmeerimiskeskkonna järkjärgulist arendamist, liikudes lihtsamalt keerulisemale. *Wikipedia* [bootstrapping] toob sellise näite: alustatakse elementaarse tekstiredaktori kirjutamisest, seda kasutatakse assembleri tegemisel, viimast kasutatakse kõrgema taseme programmeerimiskeele objektkoodina ja nii edasi, kuni graafilise keskkonna ja komplitseeritud programmeerimiskeelte realiseerimiseni välja.

Kompilaatorite tegemisel tähistab *bootstrapping* tehnikat, kus mingi osa kompilaatorist kirjutatakse objektkeeles (näiteks, mingi osa *C* kompilaatorist kirjutatakse *C*-keeles). Refereerigem taas [bootstrapping]: tavaliselt kasutatakse järgmisi variante.

- Keele *X* translaatori esmavariant kirjutatakse keeles *Y* (viimane võib olla ka masinkood või assembler); näiteks *Niklaus Wirth* olevat esimese *Pascal*-translaatori kirjutanud *FORTRAN*is.
- *X* esmaversioon kujutab endast selle *X*-i alamhulka, mille jaoks juba on kompilaator; nii olevat tehtud mitmed *Java* „ülemhulgad“.
- *X* kompilaator luuakse *cross-compile*ri tehnikat kasutades: sel puhul kompileeritakse ühel platvormil oleva kompilaatori abil kompilaatori kood teise riistvaraplatvormi jaoks (vt. [CD], lk. 87) – seda kasutatakse tavaliselt ka *C* realiseerimisel.
- Kirjutatakse *X* kompilaatori esimene versioon ning kasutatakse teda iseenese optimeerimiseks
- Kompilaator kompileeritakse *baitkoodi* (vt. näit. [Isotamm, PK] lk. 209 jj.) tagamaks ta realiseerimist teistel platvormidel.

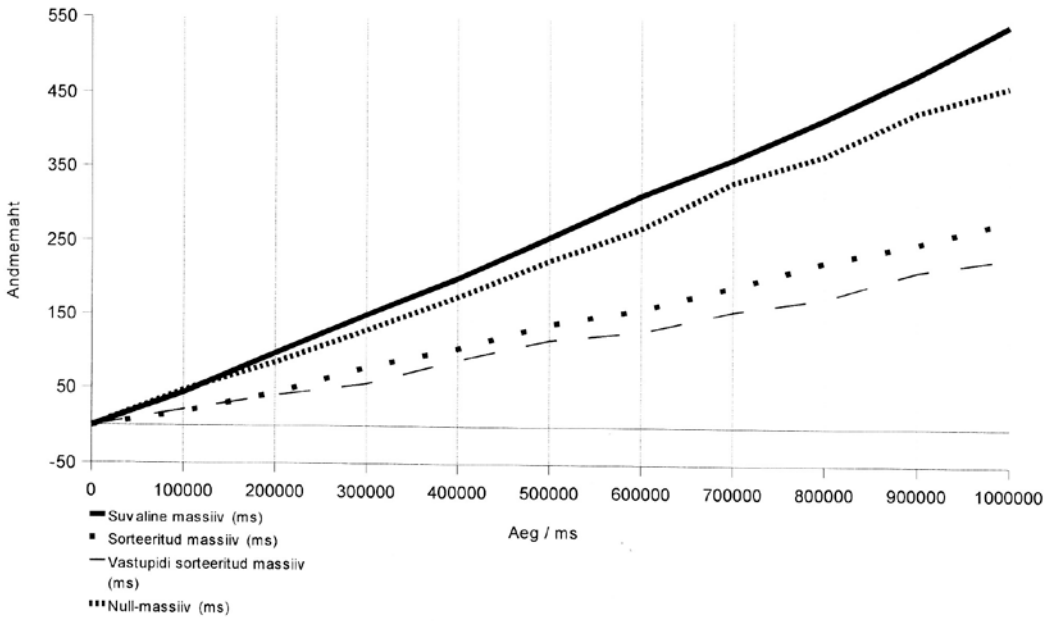
*Bootstrapping*ut on ühel või teisel moel kasutatud teiste hulgas *PL\I*, *BASIC*u, *C*, *Pascal*i, *Haskell*i, *Modula*-2 ja *Scheme* realiseerimisel.

Lisa 4. Kiir- ja ühildusmeetodite testid[1]

Test 1. Mediaan-lahkmega kiirmeetod

Andmemaht	Suvaline massiiv (ms)	Sorteeritud massiiv (ms)	Vastupidi sorteeritud massiiv (ms)	Null-massiiv (ms)
0	0	0	0	0
100000	43.6	18.2	21.4	46.8
200000	96.6	43.6	40.6	84.2
300000	149.6	78	56	128.2
400000	200.2	103.4	87.8	174.6
500000	256.2	137.6	115.8	225
600000	312.8	159	128	269
700000	362.6	190.4	156.6	331
800000	418.6	225	175	368.6
900000	478.4	250.2	212.6	428.2
1000000	543.4	281.2	231	462.4

Tabel 2: Mediaan-lahkmega kiirmeetodi tulemused



Joonis 2: Mediaan-lahkmega kiirmeetodi tulemused

Tsiteerigem *Uno Merestet* [Mereste II, lk. 71 jj.]: „Mediaan on korrastatud rea keskmine liige, millest mõlemale poole jääb võrdne arv liikmeid. Teisiti ongi mediaani nimetatud ka *keskliikmeks*.

Kui korrastatud reas on liikmeid *paaritu arv*, siis on mediaani väga hõlpus leida. Näiteks rea

*S*1 = 5, 8, 9, 13, 16, 18, 20, 25, 39

kus on 9 liiget, on mediaan (tähis *Me*) 16, so. 5. liige ükskõik kummast rea otsast lugedes.

[1] Vt. [Vigonski], lk. 13 – 16. Jooniste numeratsioon pärineb originaalist.

Kui tähistada rea liikmete arvu *n*, siis on mediaani järjekorranumber reas (*n*+1)/2.

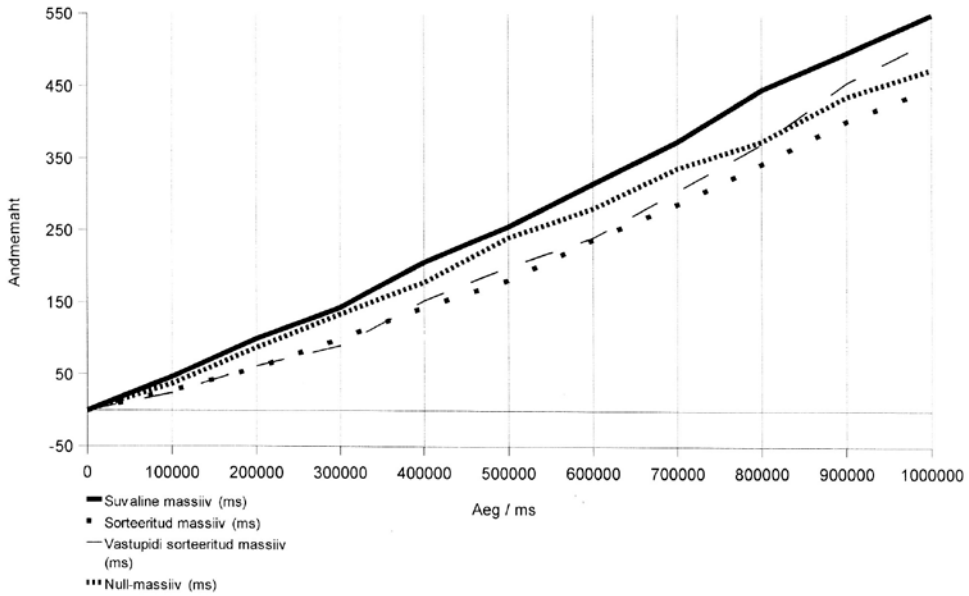
Reas, kus on *paarisarv liikmeid*, saadakse mediaani järjekorranumbriks selline arv, mis on küll rea keskel, ent kahe keskmise liikme vahel[1]. Rea alguses ja lõpus olevate variantide suurused ning omavaheline jaotumine mediaanile mõju ei avalda. Seepärast on mediaan lahtiste äärerühmadega variatsioonridades eriti sobiv asendama aritmeetilist keskmist.

Mediaani tüüpilisus on alati tunduvalt suuremal määral tagatud kui aritmeetilisel keskmisel (- - -) aritmeetiliseks keskmiseks osutub tihti just selline tunnuse väärtus, mida reas ei esine-gi ja mis pole seega reale üldse omane. Mediaaniga seda juhtuda ei saa.“

Test 2. Suvalise lahkmega kiirmeetod

Andmemaht	Suvaline massiiv (ms)	Sorteeritud massiiv (ms)	Vastupidi sorteeritud massiiv (ms)	Null-massiiv (ms)
0	0	0	0	0
100000	47	27.8	24.6	37.4
200000	100	59.2	62	87.6
300000	143.6	100.2	90.2	134.2
400000	206.4	143.8	152.8	178.4
500000	256.2	181.2	200	240.8
600000	316	237.8	240.4	281.2
700000	375	287.4	306	337.6
800000	447	343.6	371.6	375.2
900000	497	403.4	456.4	437.6
1000000	549.8	450	515.4	474.6

Tabel 3: Suvalise lahkmega kiirmeetodi tulemused



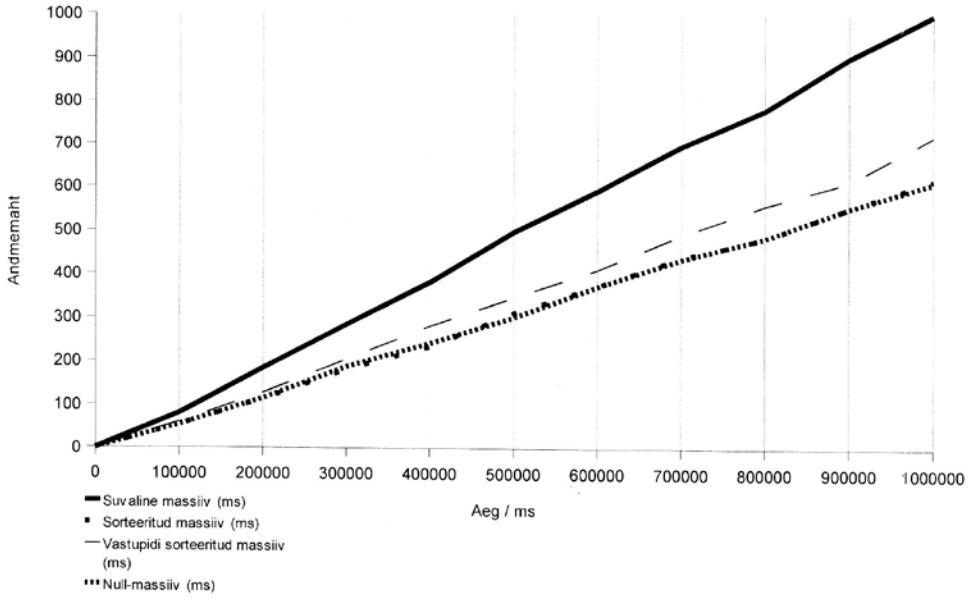
Joonis 3: Suvalise lahkmega kiirmeetodi tulemused

[1] Sel juhul peetakse mediaaniks nonde kahe väärtuse aritmeetilist keskmist.

Test 3. Ühildusmeetod

Andmemaht	Suvaline massiiv (ms)	Sorteeritud massiiv (ms)	Vastupidi sorteeritud massiiv (ms)	Null-massiiv (ms)
0	0	0	0	0
100000	81.2	54.4	60.7	54.4
200000	184.3	113.9	128.2	115.6
300000	284.5	181.3	204.5	187.5
400000	381.3	234.3	281.3	242
500000	498.2	311.2	346.8	301.6
600000	592	373.3	411	372
700000	696.7	439.2	492.1	436
800000	781.3	484.4	559.3	485.6
900000	903.1	554.7	614.2	554.5
1000000	999.8	618.6	720.6	612.5

Tabel 4: Ühildusmeetodi tulemused



Joonis 4: Ühildusmeetodi tulemused

Meenutagem, et „sorteeritud massiiv“ tähendab vektori järjestatust mittekahanevas ning „vastupidi sorteeritud massiiv“ järjestatust mittekasvavas järjekorras. (*Simon Vigonski* sorteerib vektorit mittekahanevalt). Ning „null-massiiv“ on tegelikult „mõnitamine“: meetodile antakse ette vaid null-väärtustest koosnev vektor, kus poleks üldse vaja midagi teha.

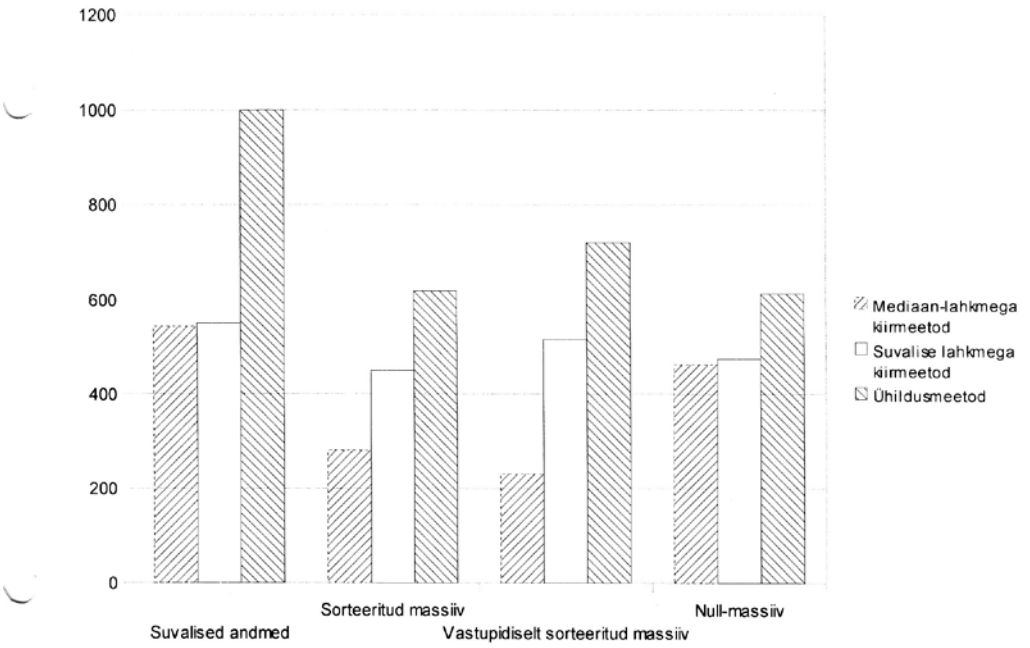
Järgmisel leheküljel veendume, et *Hoare*'i *kiirmeetod* on tõepoolest nähtavalt kiirem *v. Neumann*i *ühildusmeetodist* ja selle meetodi populaarsus (vt. *Astrachan*i tabelit) on põhjendatud. Mõlemat meetodit on püütud korduvalt paremaks teha (saavutamata küll logaritmhinnangust paremat ajalise keerukuse hinnangut). Me ei tea, kas *C. A. R. Hoare* pidas silmas just neid edasiarendusi, kui ta ütles[1], et *me peaksime loobuma pisioptimeerimisest, millele kulutame 97% ajast; selles peitubki kogu kurja juur.*“

[1] [Hoare] andmeil eitas *Sir Hoare*, et ta oleks kunagi nii ütelnud, ent *Knuth* olla ütelnud, et on küll.

Koondtulemused suurima testi puhul

	Mediaan-lahkmega kiirmeetod (ms)	Suvalise lahkmega kiirmeetod (ms)	Ühildusmeetod (ms)
Suvalised andmed	543.4	549.8	999.8
Sorteeritud massiiv	281.2	450	618.3
Vastupidiselt sorteeritud massiiv	231	515.4	720.6
Null-massiiv	462.4	474.6	612.5

Tabel 5: Koondtulemused 1 000 000 elemendilise testi jaoks



Joonis 5: Koondtulemused 1 000 000 elemendilise testi jaoks

Lisa 5. Automaat ja rooma numbrid

Tekstitöötlus on sisuldasa töö stringiga, so. sümbolitest koosneva vektoriga. Sageli on ülesanne püstitatud nii, et string tuleb sümbolhaaval, vasakult paremale, läbi vaadata, „lugemispea" ees on jooksev sisendsümbol ning selle koodi põhjal teeb algoritm mingi otsuse. Kui see sümbol on „vale", siis töö katkeb veasituatsiooni fikseerimisega, tavaliselt ta seda pole ning algoritm võib sümbolit lihtsalt aktsepteerida ja jätkata tööd järgmisest sümbolist, ent sümbol võib põhjustada ka mingit tegevust, sh. programmi lülitumist teisele režiimile. Neid režiime nimetatakse *olekuteks* ja minimaalselt on neid kaks: *alg-* ja *lõppolek* (viimane saabub veasituatsioonis või lõputunnuse lugemisel). Sedatüüpi algoritmi võime pliiatsi ja paberi abil kujutada tabelina[1], kus juhtveerus (mis esitab „ridade nimesid") on sisendsümbol, juhtreas („veergude nimed") on olekud ning lahtrites tegevused, sh. ka oleku muutmine.

Erialakirjanduses nimetatakse selliseid tabeleid vahel *otsustustabeliteks* (*decision table,* таблица решения)[2] või *lõpliku olekute hulgaga automaadi* skeemiks. Viimane termin on tegelikule programmeerijale pisut kõrgelennuline: *Chomsky* klassifikatsiooni[3] 3. klassi grammatikad kirjeldavad *regulaarseid keeli,* mis defineerivad *regulaaravaldisi* ning neid tuvastavad mitut liiki automaadid (*determineeritud*, *mittedeterminee*ritud, *magasinmäluga* jmt.). Ent primitiivse automaadi stringi töötlemiseks võime programmeerida ka ilma vastavat teooriat tundmata.

Nende ridade autor on seda tehnikat kasutanud mitme „päris"-ülesande lahendamisel (näiteks, *Boole*'i funktsiooni lahendite otsimise programmi sisendi programmeerimisel, kus tuli funktsiooni tekst teisendada puustruktuuriks), aga ka *A&A* praktikuminäidetena. Lihtsad näited on positsioonilise kümnendarvu teisendamine „rooma kujule" ja vastupidi; allpool teeme nendega tutvust. Ent sissejuhatuseks värskendagem mälu rooma numbrite[4] alal. [EE 8] kirjutab, et need on „liitmispõhimõttel rajanev arvusüsteem, milles kasut. erisuguseid sümboleid kümnendkohtade (I=1, X=10, C=100, M=1000) ja nende poolte tarvis (V=5, L=50, D=500). Tei-

	1	**× 10**	**× 100**	**× 1000**
1	I	X	C	M
2	II	XX	CC	MM
3	III	XXX	CCC	MMM
4	IV	XL	CD	MMMM
5	V	L	D	MMMMM
6	VI	LX	DC	MMMMMM
7	VII	LXX	DCC	MMMMMMM
8	VIII	LXXX	DCCC	MMMMMMMM
9	IX	XC	CM	MMMMMMMMM

Tabel L5.a. Rooma numbrid.

[1] *Tabel* tavakeeles, mitte abstraktse andmestruktuuri mõttes.
[2] Tavaliselt küll juhul, kui juhtreas pole *olekud,* vaid on *loogilised tingimused*.
[3] Vt. näit. [Isotamm PK], lk. 234.
[4] Vt. ka [EE 8, lk. 189] ja [Truu]. Viimane esitab lisaks huvitavad ja meie omadest sootuks erinevad teisendusalgoritmid, milledega soovitame lugejal kindlasti tutvuda.

sed naturaalarvud kirjutatakse selliseid sümboleid sobivalt kõrvutades. Kui suurema väärtusega sümbol paikneb väiksema väärtusega sümboli ees, siis on koguväärtus nende väärtused liidetud, kui väiksem, siis lahutatud; nt. VI=6, IV=4, XI=11, IX=9, CX=110, XC=90.“

Teisendusautomaati esitab järgmine tabel:

S \ olek	start	i	v	x	l	c	d	m
I	o=i ic=1 a=1	ic++; kui c<4 a++, muidu viga	Kui ic<3 {ic++ a++} muidu viga	ic=1 a++ o=i	ic=1 a++ o=i	ic=1 a++ o=i	ic=1 a++ o=i	ic=1 a++ o=i
V	o=v ic=0 a=5	Kui ic=1{ a+=3 lõpp} muidu viga	viga	a+=5 o=v	a+=5 o=v	a+=5 o=v	a+=5 o=v	a+=5 o=v
X	o=x xc=1 a=10	Kui ic=1{ a+=8 lõpp} muidu viga	viga	xc++ kui xc<4{ a+=10} muidu viga	xc=1 a+=10 o=x	xc=1 a+=10 o=x	xc=1 a+=10 o=x	xc=1 a+=10 o=x
L	o=l a=50	Kui ic=1{ a+=48 lõpp} muidu viga	viga	Kui xc=1{ a+=30 xc=0 o=x} muidu viga	viga	a+=50 o=l	a+=50 o=l	a+=50 o=l
C	o=c cc=1 a=100	Kui ic=1{ a+=98 lõpp} muidu viga	viga	Kui xc=1{ a+=80 xc=0 o=x} muidu viga	viga	Kui cc<3{ a+=100 cc++} muidu viga	a+=100 cc++ o=c	a+=100 cc++ o=c
D	o=d a=500	Kui ic=1{ a+=498 lõpp} muidu viga	viga	Kui xc=1{ a+=480 xc=0 o=x} muidu viga	viga	Kui cc=1{ a+=300} muidu viga	viga	a+=500 o=d
M	o=m a=1000	Kui xc=1{ a+=998 lõpp} muidu viga	viga	Kui xc=1{ a+=980 xc=0 o=x} muidu viga	viga	Kui cc=1{ a+=800} muidu viga	viga	a+=1000
muu	viga	viga	viga	viga	viga	viga	viga	viga
Lõpp	print(a)	print(a)	print(a)	print(a)	print(a)	print(a)	print(a)	print(a)

int a: kogutav arv, ic: i-de, xc: x-de ja cc: c-de loendaja; o=olek

Tabel L5b. Automaat *rooma → araabia.*

Tänapäeval on „rooma numbrid“ marginaalses rollis.Vahel kasutatakse neid raamatu-peatükkide nummerdamiseks, omal ajal kirjutati tõsimeeli, et niimoodi nummerdatakse tänapäeval partei kongresse, ja üldiselt saavad kõik aru, kui nii nummerdame sajandeid (meil käib XXI).

Ülaltoodud tabeli realiseerib järgmine programm[1]:

```
/* araabia.c: teisendab rooma-arvu araabia kujule */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char rooma[20];
char olek, k;
int a,ic,xc,cc,i,j,n;

int start(char key){
      int flag=1;
      switch(key){
        case 'I': olek='i'; ic=1; a=1; break;
        case 'V': olek='v'; ic=0; a=5; break;
        case 'X': olek='x'; xc=1; a=10; break;
        case 'L': olek='l'; a=50; break;
        case 'C': olek='c'; cc=1; a=100; break;
        case 'D': olek='d'; a=500; break;
        case 'M': olek='m'; a=1000; break;
        default:  flag=0;
        }
      return(flag);
}

int yks(char key){
      int flag=1;
      if((ic>1)&&(key!='I')) return 0;
      switch(key){
            case 'I': ic++;
                  if(ic<4) a++;
                  else flag=0;
                  break;
            case 'V': if(ic==1){ a+=3; ic=0; olek='e'; }
                    else flag=0;
                    break;
            case 'X': if(ic==1) a+=8;   ic=0; olek='e'; break;
            case 'L': if(ic==1) a+=48;  ic=0; olek='e'; break;
            case 'C': if(ic==1) a+=98;  ic=0; olek='e'; break;
            case 'D': if(ic==1) a+=498; ic=0; olek='e'; break;
            case 'M': if(ic==1) a+=998; ic=0; olek='e'; break;
            default: flag=0; break;
            }
      return(flag);
}

int viis(char key){
      int flag=1;
      if(key=='I'){
          if(ic<3){ a++; ic++;}
          else flag=0;
          }
      else flag=0;
      return(flag);
}
```

[1] Mainigem, et töötleme *stringi,* vasakult paremale.

```
int kymme(char key){
      int flag=1;
      switch(key){
             case 'I': ic=1; a++; olek='i'; break;
             case 'V': ic=0; a+=5; olek='v'; break;
             case 'X': if(xc<3){ a+=10; xc++; }
                   else{ flag=0; xc=0;}
                   break;
             case 'L':
                  if(xc==1){ a+=30; olek='l'; xc=0;}
                  else flag=0;
                  break;
             case 'C':
                  if(xc==1){ a+=80; olek='c'; xc=0;}
                  else flag=0;
                  break;
             case 'D':
                  if(xc==1){ a+=480; olek='d'; xc=0;}
                  else flag=0;
                  break;
             case 'M':
                  if(xc==1){ a+=980; olek='m'; xc=0;}
                  else flag=0;
                  break;
            default: flag=0; break;
            }
      return(flag);
}

int fifty(char key){
      int flag=1;
      xc=0; ic=0;
      switch(key){
             case 'I': ic=1; a++; olek='i'; break;
             case 'V': a+=5; olek='v'; break;
             case 'X': xc=1; a+=10; olek='x'; break;
             case 'L': flag=0; break;
             case 'C': flag=0; break;
             case 'D': flag=0; break;
             case 'M': flag=0; break;
             default: flag=0; break;
             }
      return(flag);
}

int sada(char key){
      int flag=1;
      xc=0; ic=0;
      switch(key){
             case 'I': ic=1; a++; olek='i'; break;
             case 'V': a+=5; olek='v'; break;
             case 'X': xc=1; a+=10; olek='x'; break;
             case 'L': a+=50; olek='l'; break;
             case 'C':
                        if(cc<3){ a+=100; cc++;}
                        else flag=0;
                        break;
             case 'D':
                        if(cc==1){ a+=300; olek='d';}
                        else flag=0;
```

```
                        break;
              case 'M' : if(cc==1){ a+=800; olek='m';}
                        else flag=0;
                        break;
              default: flag=0; break;
              }
      return(flag);
}

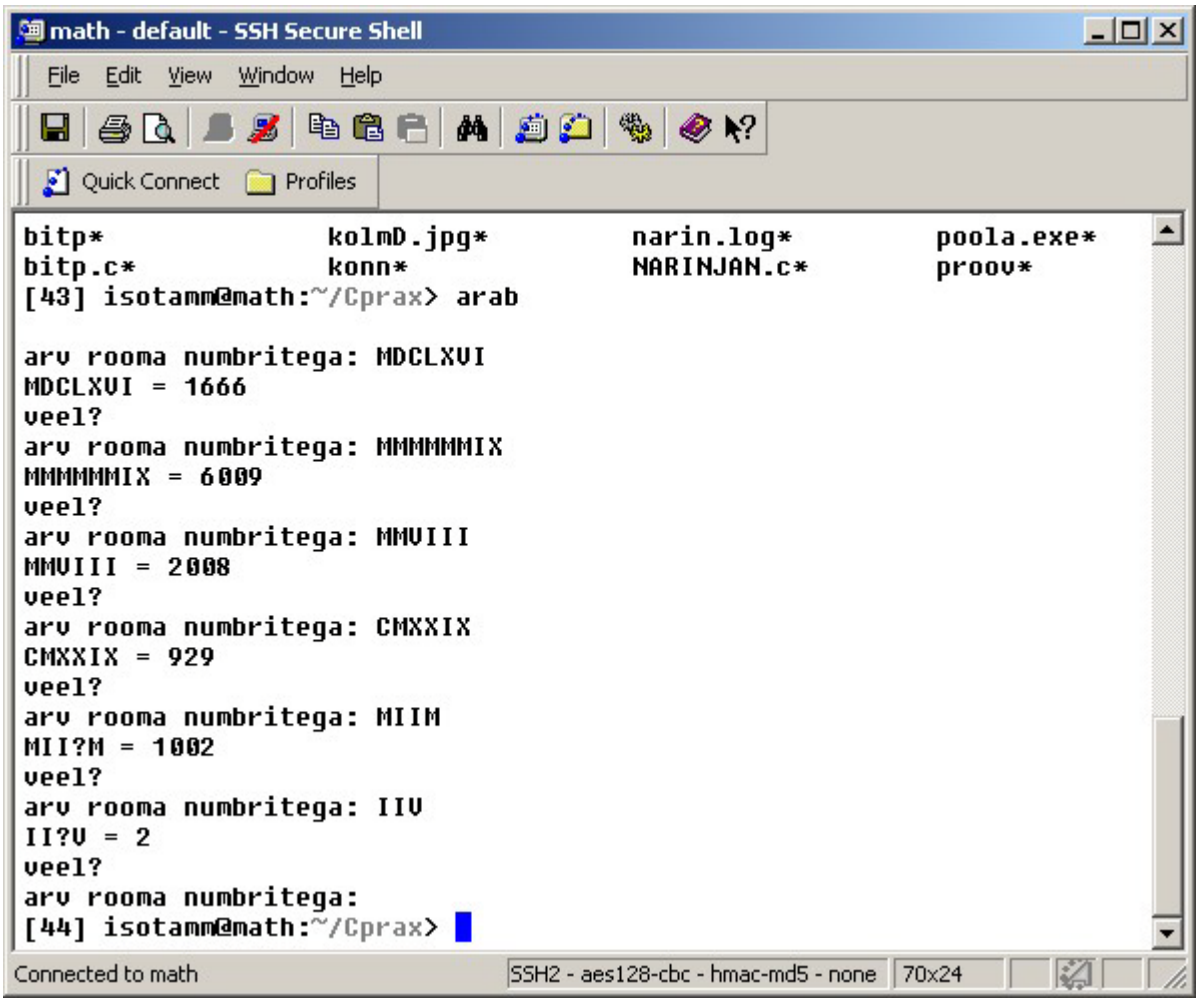
int viissada(char key){
      int flag=1;
      xc=0; ic=0; cc=0;
      switch(key){
            case 'I': ic=1; a++; olek='i'; break;
            case 'V': a+=5; olek='v'; break;
            case 'X': xc=1; a+=10; olek='x'; break;
            case 'L': a+=50; olek='l'; break;
            case 'C':
                  if(cc<3){ a+=100; cc++; olek='c';}
                  else flag=0;
                   break;
             case 'D': flag=0; break;
             case 'M': flag=0; break;
             default: flag=0; break;
             }
      return(flag);
}

int tuhat(char key){
      int flag=1;
      xc=0; ic=0; cc=0;
      switch(key){
             case 'I': ic=1; a++; olek='i'; break;
             case 'V': a+=5; olek='v'; break;
             case 'X': xc=1; a+=10; olek='x'; break;
             case 'L': a+=50; olek='l'; break;
             case 'C':
                  if(cc<3){ a+=100; cc++; olek='c';}
                  else flag=0;
                  break;
             case 'D': a+=500; olek='d'; break;
             case 'M': a+=1000; break;
             default: flag=0; break;
             }
      return(flag);
}

void makeit(void){
     for(i=0; i<n; i++){
           k=rooma[i];
           switch(olek){
              case 's': if(start(k)==0) goto p; break;
              case 'i': if(yks(k)==0) goto p; break;
              case 'v': if(viis(k)==0) goto p; break;
              case 'x': if(kymme(k)==0) goto p; break;
              case 'l': if(fifty(k)==0) goto p; break;
              case 'c': if(sada(k)==0) goto p; break;
              case 'd': if(viissada(k)==0) goto p; break;
              case 'm': if(tuhat(k)==0) goto p; break;
              case 'e': goto p;
```

```
            }
        }
p:      for(j=0; j<i; j++) printf("%c",rooma[j]);
        if(i<n) printf("?");
        for(j=i; j<n; j++) printf("%c",rooma[j]);
        printf(" = %d \n",a);
}

int main( ){
ring: printf("\narv rooma numbritega: ");
      gets(rooma);
      n=strlen(rooma);
      if(n==0) return(0);
      a=0;
      olek='s';
      ic=0; xc=0; cc=0;
      makeit( );
      printf("veel?");
      goto ring;
}
```



Joonis L5.a. *Rooma → araabia*. Veakohta tähistab „?“.

Teisendus ***araabia → rooma*** on sootuks lihtsam, esitame siin *kuni* neljakohaliste positsioonide kümnendarvude teisendamise programmi (so, sobib arvude 1..9999 jaoks).

```
//rom.c teisendab kuni 4-kohase kümnendarvu "rooma kujule". 7. nov. 2007
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

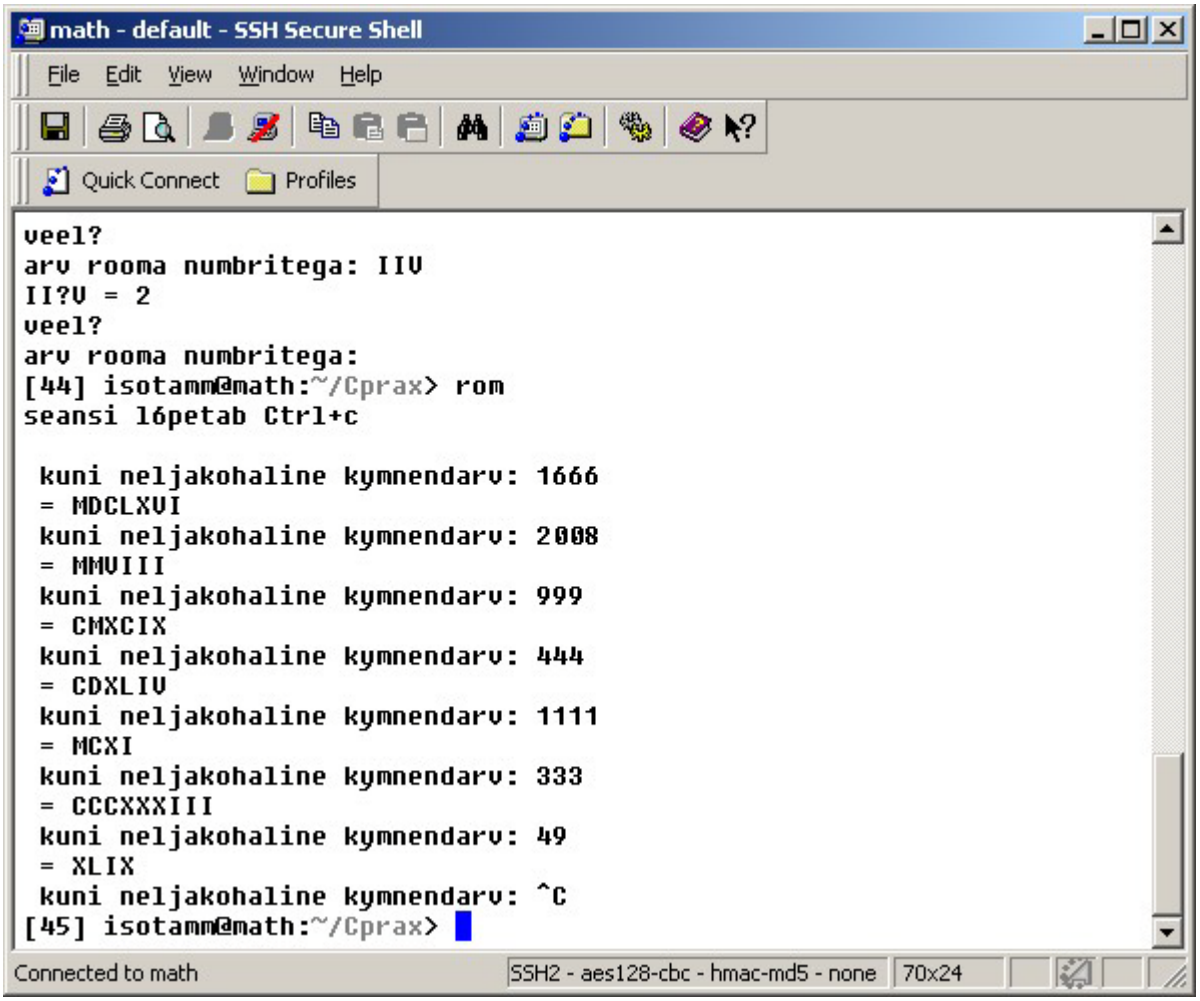
char a[5];  /* araabia */
int n,m,olek,i;
char d;
void suur(void){
   int j;
   m=atoi(&d);    //atoi teisendab väljalt d sümbolkuju-arvu int-muutujaks[1]
   for(j=0; j<m; j++) printf("M");
}

void rooma(char i,char v,char x){
        switch(d){
                case '1': printf("%c",i); break;
                case '2': printf("%c%c",i,i); break;
                case '3': printf("%c%c%c",i,i,i); break;
                case '4': printf("%c%c",i,v); break;
                case '5': printf("%c",v); break;
                case '6': printf("%c%c",v,i); break;
                case '7': printf("%c%c%c",v,i,i); break;
                case '8': printf("%c%c%c%c",v,i,i,i); break;
                case '9': printf("%c%c",i,x); break;
                }
        }

int main( ){
        printf("seansi l6petab Ctrl+c\n");
R:      printf("\n kuni neljakohaline kymnendarv: ");
        scanf("%4s",&a);    //"%4s": scanf loeb pikemast sisendreast esimesed 4 sümbolit
        fflush(stdin);
        n=strlen(a);
        printf(" = ");
        i=0;
        for(olek=n; olek>0; olek--){    //teisendust alustame vasakult
                if((d=a[i])=='0') goto next;
                if(isdigit(d)==0) goto viga;  //isdigit kontrollib, kas sümbol (d) on number
                switch(olek){
                        case 1: rooma('I','V','X'); break;   //1..9
                        case 2: rooma('X','L','C'); break;  //10..90
```

[1] *atoi* asemel võinuks me kirjutada *m=d*–48 (*ASCII* koodid: '0'=48,..,'9'=57).

```
                        case 3: rooma('C','D','M'); break;  //100..900
                        case 4: suur( ); break;     //tuhanded: 1..9
                        }
next:           i++;
                }
        goto R;
        viga: printf("\n%c pole number\n",d);
        goto R;
}
```



Joonis L5.b. araabia → rooma.

Lisa 6. Pseudo-tehisintellekt

Tehisintellekti temaatikast[1] on tänini muuhulgas aktuaalne tarkvara loomine, mille abil saaks võimalikuks kasutaja ja programmi teineteist mõistev dialoog loomulikus (vene, eesti jne) keeles. Sisuldasa tähendab see keele semantikale tuginevate andmestruktuuride loomist ning nende interpreteerimist teksti *mõttest* aru saamiseks, kasutades nii leksikat, süntaksit, idioomide sõnastikku jne. Ja ehkki selles valdkonnas on tänaseks üllatavalt palju ära tehtud[2], pole veel toimivaid programme enama kui tehnilise toortõlke jaoks, ilukirjanduse, poeesia ja argoo adekvaatsest tuvastamisest ja tõlkimisest on asi veel vististi mõnda aega kaugel.

1978. a. organiseeris akad. *Enn Tõugu* N. Liidu selle valdkonna autoriteete kaasates (ja talle omaselt meie noortele osavõttu võimaldades) kevadtalvel Pühajärve lähistel asunud puhkebaasis seminari, mida juhtis *NL TA* akadeemik *Pospelov* ja osales palju arvutilingviste, tehisintellekti asjatundjaid jmt, lisaks paar programmeerijat (nende ridade autor liigitab end viimaste hulka) ja kokkutuleku üheks teemaks oli just *päringud loomulikus keeles*[3].

Esimese tööpäeva õhtul, pärast sauna ja vabas õhkkonnas, tegi kõigile nalja *NL TA Novosibirski Filiaali* noor teadur *Aleksandr Narinjani*[4], kes väitis, et võime ülejäänud päevad suusatada ja saunatada jne., kuivõrd ta selle seminari probleemid eelmisel nädalal juba ära lahendas. Ja ta jahmatas kohe ka enamikku osalejaid, tuues kõigile näha paarikilose laitrüki-paki (2×A4, kokkulapatud lint), kus oli trükitud kasutaja ja programmi dialoogi logi. Kasutaja[5] võis küsida mida tahes *Narinjani* Arvutuskeskuse-kolleegide kohta, ja – usutavasti on see õigesti meeles – esimene küsimus oli (*Narinjani* lubas kasutada nii vene kui ka inglise keelt, ja trükiseade sai mõlemaga suurtähti kasutades hakkama):

A НУ-КА TELL ME СВОЛОЧ СКОЛЬКО MONEY ПОЛУЧАЕТ ЕРШОВ[6]

ja vastus tuli:

АКАДЕМИК АНДРЕЙ ПЕТРОВИЧ ЕРШОВ ПОЛУЧАЕТ 500 РУБЛЕЙ ЗА МЕСЯЦ

Ja selliseid „protokolle“ oli palju, kõik „õiged ja asjakohased“. Naer asendus jahmatusega[7] ... *Narinjani* oli loomulikult reserveeritud, naeratas ja andis mõista, et „pole tänu väärt“. Siinkirjutajale hakkas asi nalja tegema, ja sobival hetkel saime *Saša*ga nelja silma all vestelda; näitasin talle jämedat plokkskeemi võimaliku lahendi kohta. *Narinjani* vaatas seda natukese aega, hakkas naerma, ja palus, et ma enne homset teda ei paljastaks, ta tahtis ise etendust anda.

---

[1] Vt. näit. [Isotamm, PK] lk. 174 jj.
[2] Vt. näit. TÜ töörühma, mida juhivad akadeemik *Haldur Õim* ja prof. *Mare Koit*, publikatsioone.
[3] Vt. ka *Haldur Õim*, Filoloogi mälestused sellest, kuidas eesti keel ja arvuti Tartus kokku said [PoSa, lk. 87 – 95].
[4] Sama mees kirjutas kümmekond aastat hiljem õige, valulise ja julge märgukirja [Narinjani]. Siinkirjutaja arvas, et Saša on armeenlane, ent meie kolleeg, pärslane *Vezal Vojdani*, arvas, et tegu on pigem tema rahvuskaaslasega – mis ongi väga tõenäoline. *Haldur Õim*u andmeil on *Narinjani* täna Moskvas Venemaa Tehisintellekti Instituudi direktor ([PoSa] , lk. 94).
[5] Mõistagi, NL TA Novosibirski filiaali arvutuskeskuse masinasaalis, portatiivseid arvuteid tol ajal ei olnud.
[6] Ligikaudne tõlge: „kuule tõbras, kui palju raha Jeršov saab“, vastus: „akadeemik A. P. J. saab 500 rbl. kuus“.
[7] Pole vist vaja ütelda, et keegi ei arvanud, et tegu oleks tõepoolest tehisintellektliku lahendusega; küsimus oli, *kuidas* ta seda teeb.

Kui siinkirjutaja sai ettepaneku hakata andama *A&A* praktikume, meenus talle see „Narinjani laadanipp“ ning talle tundus, et see võiks pakkuda huvi ka meie tudengitele. Allpool esitame selle minimaalse versiooni *C*-teksti; kogu töö on üles ehitatud *string.h* funktsioonile *strstr*:

```
char *strstr(const char *cs, const char *ct),
```

mis tagastab viida *ct* esimesele leidumisele *cs*-is või viida *NULL*, kui *ct* pole *cs*-i alamstring. Programmi *narinjani* saab käivitada kas ilma parameetriteta (sel juhul ei tehta logifaili) või parameetriga, milleks on loodava logifaili nimi. Alustame ühe sellise logi tutvustamisega[1]:

```
Andmebaas:

insener Mati Oja, palk 7000, telefon 333867
programmeerija Rein Kraav, palk 8000, telefon 333009
suhtekorraldaja Ele Mann, palk 5000, telefon 333542
direktor Aivar Kivi, palk 25000, telefon 333543
administraator Annely Katus, palk 10000, telefon 333667
logistik Sven Ivanov, palk 6000, telefon 333509

to end: single 'Enter'

olen volitatud vastama Teie kysimustele kolmes kohalikus keeles

Kysige palun!
A nu-ka tell me svoloch skolko poluchajet kivi
Dr. Kivi, palk, 25000

Kysige palun!
but Kraav, how much he gets?
hr. Kraav, palk, 8000

Kysige palun!
kuda talle helistada
hr. Kraav, telefon, 333009

Kysige palun!
attehh, aga mannile?
prl. Mann, telefon, 333542

Kysige palun!
agawhouvastamlogistikom rabotajet,a?
hr. Ivanov, amet, logistik

Kysige palun!
palju ta pappi saab
hr. Ivanov, palk, 6000

Kysige palun!
miks nii vähä!
vabandust, aga see on konfidentsiaalne informatsioon

Kysige palun!
loodetavasti saite kogu vajaliku informatsiooni!
```

C-programm ise on järgmine:

[1] Ühe lahendusseansi „ekraanipilt“ on toodud *Cygwin*i tutvustavas lisas 2.

```
/* narinjani.c :: pseudotehisintellekt "Narinjani laadanipp". 2.11.00. */

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

//c_key: eeskätt veerutunnused. Piisab prefiksist!

char *c_key[10][26]={
{"pere","last","fami"},  //perekonnanimi: 0 :: veeruindeks
{"nimi","first","name","imja","zovut","zvat"},  //eesnimi: 1
{"amet","posit","dolz","teeb","zanim","making","busy","tegele","rabota","post"}, //amet: 2
{"palk","salary","raha","papp","kroon","money","dollar","bucks","saa",  //palk: 3
"poluch","dengi","deneg","zar","bash","pay","rubl","teeni","puru",
"tasku","pocket","karman","plekk","palju","skol","how"},
{"telefon","phone","numb","nomer","pozvon","helista","call","traat"},  //telefon: 4

//“pärisveerud“ on indeksitega 0..4. Ülejäänud (5..9) on pigem dialoogi toetavas rollis

{},  //reservis: 5 (ei läinud vaja)
{"kes","who","kto","kelle","kogo","chei","chja","kis","mis"},  // rida: 6 (häälestab reale)
{"ta","tema","his","her","jego","jejo"},       // omastav!: 7  (kinnitus: rida on fikseeritud)
{"miks","why","pochemu","zachem"},                    /* no comment: 8 */
{"loll","kura","fool","stupid","fuck","jeb","dura","hu","per","pos"}};   /* labasused: 9 */

//veeruvektorite pikkused c_lim : veerutuvastuste võtmesõnade arvud c_key’s

int c_lim[10]={
   3,    //0: perekonnanimi
   6,    //1: eesnimi
   10,  //2: amet
   25,  //3: palk
   8,   //4: telefoninumber
   0,   //5: reservis
   7,   //6: reatuvastaja (isik)
   6,   //7: myra: jätkab jooksva persooniga
   4,   //8: kysimus yle piiride: legaalsed on vaid need, millele leidub vastus tabelis.
   10  //9: labasused
};

//reatunnused  (objekti fiksaator: objekte on 6, stringid peaksid aitama lisaks tabeli enda
//andmetele määrata reaindeksit)

char *r_key[6][10]={
{"engin","masin","inz","oja","mati"},  /* r=0, näit. masinamees, inzhener, nimed väikselt */
```

```
{"prog","anal","kraav","rein"},  /* r=1  progeja, programmist, analüütik... */
{"ylemus","boss","chief","pea","nach","dire","hozj","kivi","aivar"},  /* r=2; pealik, peamees */
{"suht","PR","pub","rekl","aja","mann","ene"},  /* r=3 PR-mees, publicity, ajakirjandus...*/
{"admin","valits","hald","komand","katus","annely"},  /* r=4 valitsejanna, haldur, .. */
{"logi","trans","komm","com","ivanov","sven"}};  /* r=5  */

int r_lim[6]={5,4,9,7,6,6};  //reavektorite pikkused -- võimalike alternatiivsete võtmete
//arvud

char *vp[]={"perekonnanimi","eesnimi","amet","palk","telefon"};  //vastuse jaoks: näit.
//küsitakse „pappi“, aga vastuses on korrektselt „palk“

//andmebaasitabel "ise"  (viimast veergu ei avalikustatud, veerud ja nende järjekord pole
//„päris need“

char *tabel[6][6]={
        {"Oja","Mati","insener","7000","333867","hr."},
        {"Kraav","Rein","programmeerija","8000","333009","hr."},
        {"Kivi","Aivar","direktor","25000","333543","Dr."},
        {"Mann","Ene","suhtekorraldaja","5000","333542","prl."},
        {"Katus","Annely","administraator","10000","333667","pr."},
        {"Ivanov","Sven","logistik","6000","333509","hr."}
};

char *K;   //sisendrida ja redigeeritud sisendrida (sinna satuvad ainult numbrite ning suur-
//ja väiketähtede koodid)
FILE *Logi=NULL;  //logifaili "pide"
int logi;  //kui on logifaili vaja, siis logi=1

//call: narinjani  /   narinjani <logifaili_nimi> „tr“ dubleerib vajadusel konsooli logisse

void tr(char *p){
    printf(p);
    if(logi==1) fprintf(Logi,p);
    }

int main(int argc,char *argv[ ]){
int i,j,r,v,x,y,z,fv,fr,s,u,t=1;
logi=0;   //logifaili ei taha või selle tekitamine ebaõnnestus
K=(char *)malloc(256);
if(K==NULL) return(1);
if(argc==1) goto letsgo;  //logifaili ei ole

Logi=fopen(argv[1],"w");
if(Logi!=NULL) logi=1;  //ekraanidialoog kopeeritakse logifaili
```

```
letsgo: y=-1; z=-1;

//“tabel“ ekraanile (ja logisse, kui vaja)
tr("Andmebaas:\n\n");
tr("insener Mati Oja, palk 7000, telefon 333867\n");
tr("programmeerija Rein Kraav, palk 8000, telefon 333009\n");
tr("suhtekorraldaja Ele Mann, palk 5000, telefon 333542\n");
tr("direktor Aivar Kivi, palk 25000, telefon 333543\n");
tr("administraator Annely Katus, palk 10000, telefon 333667\n");
tr("logistik Sven Ivanov, palk 6000, telefon 333509\n\n");
tr("to end: single 'Enter'\n\n");
tr("olen volitatud vastama Teie kysimustele kolmes kohalikus keeles\n\n");

 xnew: memset(K,'\0',256);  //sisendpuhvri nullimine
        v=-1; r=-1;  //tabeli veerud ja read: undefined
        fv=fr=0;   //vanu veeru- ja reaindekseid pole
        if(t==1)   //eelmine kysimus fikseeris objekti/see on esimene
                tr("Kysige palun!\n");
        else tr("kuidas palun?\n");   //objekt (rida) fikseerimata

        gets(K);  //loe küsimus
        if(strlen(K)==0){    //tyhi 'Enter', end of the show
                tr("loodetavasti saite kogu vajaliku informatsiooni!\n");
                if(logi==1){
                        fflush(Logi);   //väljundpuhvri sundtühjendamine
                        fclose(Logi);
                        }
                return(1);
                }
//filter: ainult numbrid ja "aaped" (suured ja pisikesed)[1] : ASCII koodid. ’0’=48, ’z’=122.
if(logi==1) fprintf(Logi,"%s\n",K);
s=0;
for(u=0;u<256;u++){
  if((K[u]>=48&&K[u]<=57)||(K[u]>=65&&K[u]<=90)||(K[u]>=97&&K[u]<=122)){
        K[s]=K[u];
        s++;
        }
  }
K[s]='\0';   //lõputunnus paika
//otsin veerutunnuseid
for(i=0;i<10;i++){
   for(j=0;j<c_lim[i];j++){
```

[1] Filter eirab kõiki *ASCII* sümboleid peale 0..9A..Za..z. Niisiis, me võime küsimusi esitada näiteks ka „s õ r e n d a tul t“, ent mitte ridu vahetades (*gets* loeb stringi kuni reavahetuseni).

```
        if(strstr(K,c_key[i][j])!=NULL){
                switch(i){
                        case 9: tr("palun! Te olete haritud inimene!\n");  //labasus
                                t=1;
                                 goto xnew;  //'break' pole vajalik: 'goto'
                        case 8: tr("vabandust, aga see on");
                                tr(" konfidentsiaalne informatsioon\n\n");  //nt, „miks“
                                goto xnew;
                        case 7: if(y>=0) r=y;
                                break;
                        default: v=i; z=v; fv=1; goto rida;
                        }
                }
        }
}
if(z>=0){
        v=z;
        goto rida;
        }
t=0; goto xnew;

//rea fikseerimine algab tabelisse kirjutatud lahtriväärtuste otsimisest (näit., „kelle traat on
//333509?“). vt. ka c_key[6]: „kelle“.
rida:
if(v!=6 && r>=0) goto ytle;  //legaalne veeruindeks ja reaindeks on juba fikseeritud.
//muidu otsin tabelist
for(i=0;i<6;i++){
        for(j=0;j<6;j++){
                if(strstr(K,tabel[i][j])!=NULL){   //vt. ülaltpoolt strstr( ) kirjeldust
                        r=i; x=j; fr=1;
                        goto vasta;  //leidsin!
                        }
                }
        }
//maatriksist ei leidnud, otsin reatunnuste hulgast
for(i=0;i<6;i++){
    for(j=0;j<r_lim[i];j++){
        if(strstr(K,r_key[i][j])!=NULL){
                r=i; x=2; fr=1;
                goto vasta;  //leidsin siit.
                }
        }
  }
if(y>=0){
        r=y;
```

```
        goto vasta;
        }
sorry: t=0; goto xnew;
vasta:
        if((fv==0)&&(fr==0)) goto sorry;
        y=r;

        if(v==6){
                switch(x){
                        case 0: v=2; break;
                        case 1: v=2; break;
                        case 2: v=2; break;
                        case 3: v=3; break;
                        case 4: v=0; break;
                        }
                }
ytle:
        printf("%s %s, ",tabel[r][5],tabel[r][0]);   //“hr.“, „pr.“, „prl.“, „Dr.“+nimi
        printf("%s, %s\n\n",vp[v],tabel[r][v]);     //“ametlik“ veerunimi + lahter
        if(logi==1){
                fprintf(Logi,"%s %s, ",tabel[r][5],tabel[r][0]);
                fprintf(Logi,"%s, %s\n\n",vp[v],tabel[r][v]);
                }
        t=1;
        goto xnew;
}
```

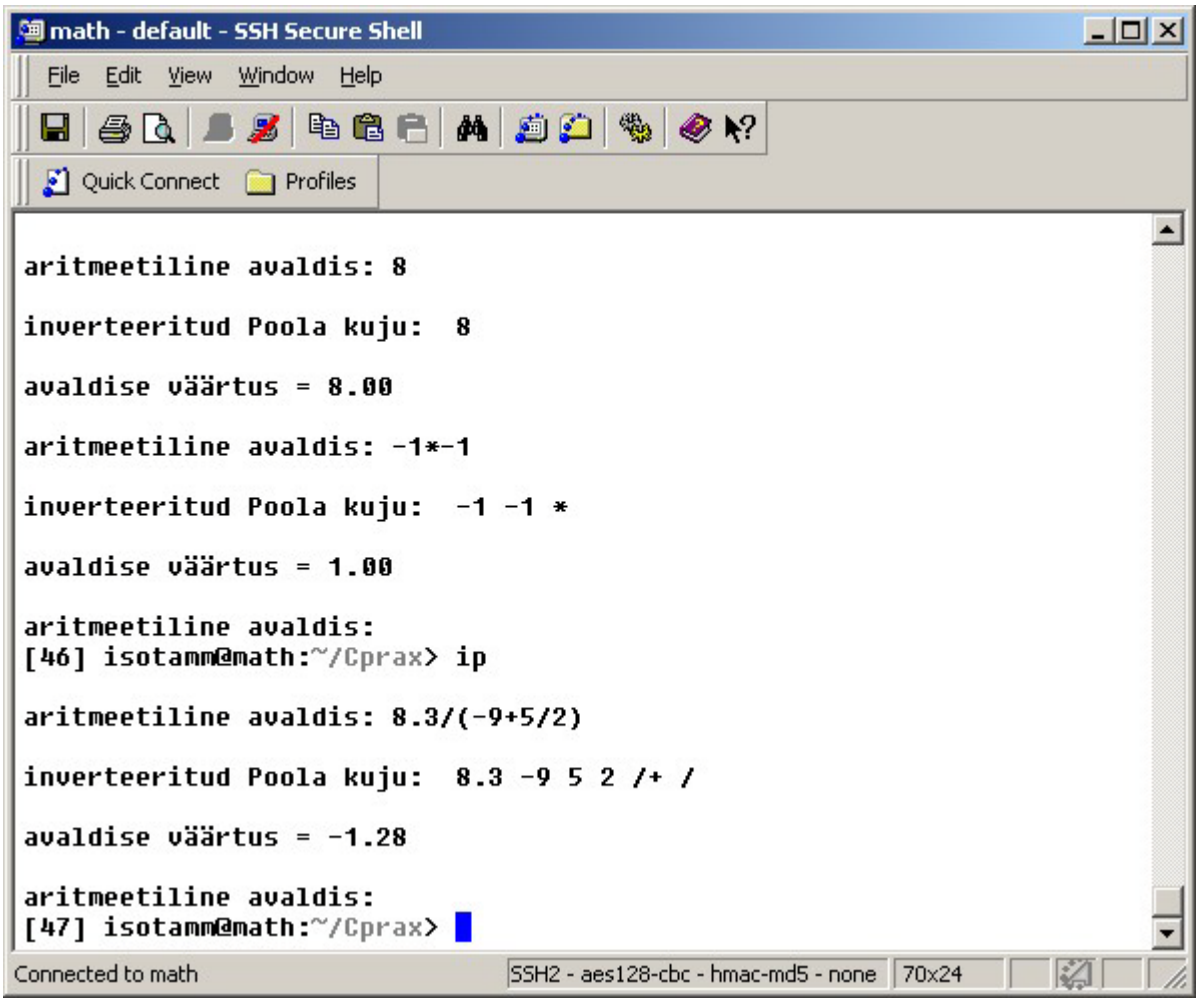
Juhime lugeja tähelepanu seigale, et äsjaesitatud algoritm on võimalikest lihtsaim, näiteks ta ei tööta, kui tabeli kõikide lahtrite väärtused pole unikaalsed. Ent see ei tohiks olla takistuseks algoritmi täiustamisele; muuhulgas oleks arvatavasti huvitav *Narinjani* lähenemist üldistada näiteks nii, et andmebaas (ja rea- ning veeruvõtmed) pole *sisse programmeeritud*, vaid on sisestatavad failist. Edasi, võiks panna programm vastama küsimustele „üle kirjete“, näiteks, „kelle palk on suurim? väikseim? mis on keskmine?“ Ja nii edasi. Mõistagi, sellised arendused ei konkureeri tehisintellekti-valdkonna tegelike ülesannetega, ent võiksid pakkuda näilisi ja seejuures toimivaid lahendusi.

Selle näite üks teema on „teksti filtreerimine“ (loe, *müra* elimineerimine, meie programmi jaoks on *müra* kõik tühikud, kirjavahemärgid jne – nagu juba öeldud, kõik, mis pole numbrite ja tähtede *ASCII*-koodid). See teema pole oluline ainult meie „laadanipis“, nende ridade autoril oli hea koostöö *Vladimir Škarupelov*iga 2007/2008 akadeemilisel aastal, kus *Vladimir* kirjutas (ja kaitses) bakalaureusetöö teemal „*Text analysis and applications*“ [Škarupelov] ja mille kohta juhendaja kirjutas oma arvamuses, et „ehkki „rämpsposti“-filtrid on viimaste aastatega efektiivsemaks muutunud, on ka seda sorti „*meile*“ tootvad illegaalid uusi võtteid leiutanud, üheks näiteks sobib oma teksti varjamine „tekstuaalse müraga“, mida nood filtrid tavaliselt ei ole suutelised elimineerima“. Näiteks: „v*i**a***g*r**a*“ = viagra. „Tekstuaalne müra“ on siin muidugi sümbol ’*’. *Vladimir*i lahendus järgis sisuldasa meie tegevust vektori *K* moodustamisel: esmases lähenduses tuleb elimineerida kõik ebaoluline.

Lisa 7. Inverteeritud Poola kuju

Selles lisas olev programm *IPoola.c* on mõeldud illustreerima abstraktse andmestruktuuri *magasin* kasutamist ning *Edsger Wyte Dijkstra* algoritmi aritmeetilise sulgavaldise viimiseks *inverteeritud Poola kujule* ja viimase interpreteerimist arvutamaks avaldise väärtust; vältimaks muutujate väärtustamist ja kasutamist (mis hägustaks iseenesest lihtsate algoritmide esitust) tegeleb see programm ainult aritmeetiliste *konstantavaldistega.* Kasutusel on 4 magasini: avaldise viimiseks postfiks-kujule kasutame kaht *FIFO*-tüüpi magasini: ühes neist on klaviatuurilt sisestatud (sulg)avaldis, teise kogutakse tema postfiks-kuju ning sulgudest vabanemiseks ja lähteavaldise komponentide järjekorra muutmiseks kasutatakse prioriteetidega *LIFO*-magasini. Need magasinid on varustatud deskriptoritega (veel kord näitamaks nende kasutamisvõimalust). Ja veel: mõlemad *FIFO*-magasinid on *stringide* jaoks.

Postfiks-kuju interpretaator kasutab ilma deskriptorita *LIFO*-tüüpi magasini. Allpool on esitatud programmi tööd kajastav ekraanipilt (kasutati *UNIX*i keskkonda).



Joonis L7.a. Ekraanipilt programmiga *ip* (lähtetekst *IPoola.c*) peetud dialoogist.

Programm IPoola.c

```
//aritmeetiline sulgavaldis --> inverteeritud Poola kuju. Dijkstra algoritm, 20.12.08
//Poola kuju interpretaator.  Programm IPoola.c.  Vektormagasinid.

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

char *pty="(-+/*)";    //aritmeetilise konstantavaldise legaalsed eraldajad
int prty[]={0,1,1,2,2,3};  //eraldajate prioriteedid (eraldajatele vasakult paremale
struct Sd{            //Stack Descriptor
      char type;     //L=LIFO, F=FIFO
      char *stack;   //viit vektorile: stack[n]
      int n;         //magasini sügavus
      int si;        //magasini indeks. Start: kui L, siis n-1, kui F, siis 0
      };

//teeb magasini-deskriptori. Argumendid: tüüp ja magasinina kasutatava vektori pikkus
struct Sd *makeD(char type, int n){
      struct Sd *descr;
      descr=malloc(sizeof(struct Sd));
      descr->type=type;
      descr->n=n;
      descr->stack=malloc(n);
      if(type=='L') descr->si=n-1;
      else descr->si=0;
      return(descr);
      }

//kirjuta magasini D element el
int push(struct Sd *D, char el){
    switch (D->type){
          case 'L': if(D->si<0){
                        overflow: printf("\nstack overflow");
                        return 0;  //tõenäoliselt viga avaldises, jätkata ei saa
                        }
                D->stack[D->si]=el;
                D->si--;
                break;
          case 'F': if(D->si==D->n) goto overflow;
                D->stack[D->si]=el;
                D->si++;
                break;
          }
```

```
    return 1;
}

//loe (ja "eemalda" magasini tipmine element
char pop(struct Sd *D){
    char s;
    switch (D->type){
        case 'L': if(D->si==D->n-1)return '\0';  //tühi
                  D->si++;
                  return(D->stack[D->si]);
                  break;
        case 'F': if(D->si==D->n)return '\0';  //tühi
                  s=D->stack[D->si];
                  D->si++;
                  return(s);
                  break;
        }
}

//Dijkstra algoritmi "tupik". Sümbol e läheb magasini või enne minekut viib magasinist
//sümboli(d) väljundritta: FIFO-magasini O.

int prtyPush(struct Sd *D, struct Sd *O, char e){
    int i,epr,xpr;    //epr, xpr: e ja x prioriteedid
    char x;

//alustav sulg '(' läheb alati magasini
     if(e=='('){
       if(push(D,e)==0) return(0);  //magasin on täis
       return(1);
       }

//kas avaldis on loetud lõpuni? string, marker on '\0'

    if(e=='\0'){
       tyhjax: x=pop(D);
             push(O,x);
             if(x=='\0')return 0;
             goto tyhjax;
             }

//lõpetav sulg: magasinist kõik -> Out kuni '('

    if(e==')'){
       out: x=pop(D);
```

```
            if(x=='(') return 1;
            push(O,x);
            goto out;
        }
//kirjutamine: magasin on tühi

    if(D->si==D->n-1){     //tühi magasin
        push(D, e);
        return 1;
        }
//kirjutamine: kui e-prioriteet<magasinitipu x prioriteet: x -> Out
     for(i=0; i<6; i++){
         if(e==pty[i]) epr=prty[i];
         }
outO: x=pop(D);     //x on eemaldatud! vajadusel kirjutan tagasi
    if(x=='\0'){   //tupik on tühi
         push(D,e);
         return 1;
         }

// x prioriteedi tuvastamine

    for(i=0; i<6; i++){
         if(x==pty[i]) xpr=prty[i];
         }

//kui sisendrea sümboli e prty pole madalam x omast: e->magasin, muidu x->Out

    if(epr>=xpr){
         push(D,x);  //kirjutangi x tagasi
         push(D,e);  //lisan e
         return 1;
         }
    else push(O,x);  //viin x tupikust väljundritta
    goto outO;
}

//sulgude paarsuse kontroll

int check(char *t){
   int i,p=0;
   for(i=0;t[i]!='\0';i++){
      if(t[i]=='(') p++;
      if(t[i]==')') p--;
      }
```

```
    if(p<0){
       printf("\nlõpetavaid sulgusid on %d rohkem",abs(p));
       goto out;
       }
    if(p>0){
       printf("\n alustavaid sulgusid on %d rohkem",abs(p));
       goto out;
       }

out: return(p);  //kui sulgude paarsus klapib, siis p=0.
}

//Poola kuju interpretaator
int  felix(struct Sd *O){
    double Ls[16],x,y;
    int i,j=0,k;
    char s, arv[16];
    i=15;  //põhja indeks
next: s=O->stack[j];
     if((isdigit(s))||((s=='-')&&isdigit(O->stack[j+1]))||(s=='.')){

//arvu ülekanne puhvrisse
         k=0;
         for(;(arv[k]=O->stack[j])!=' ';j++,k++);
         arv[k]=' ';
         j++;
         Ls[i]=atof(arv);
         i--;
         goto next;
         }
    else s=O->stack[j];
    if(s== '\0'){
          printf("\navaldise väärtus = %2.2f",Ls[i+1]);
          return(1);
         }
    x=Ls[i+2];
    y=Ls[i+1];
    switch(s){
           case '+':  Ls[i+2]=x+y; i++; break;
           case '-':  Ls[i+2]=x-y; i++; break;
          case '*':   Ls[i+2]=x*y; i++; break;
          case '/':
                   if(y==0.0){
                     printf("\nmina ei oska nulliga jagada");
                     return(0);
```

```
                    }
                Ls[i+2]=x/y;
                i++;
                break;
            }
            j++;
            goto next;
    }

int main( ){
   char x;
   int t,ti,a,i,j=0;
   struct Sd *In;       //sisendstring: aritm. konstantavaldis (FIFO)
   struct Sd *Out;      //väljundstring: inverteeritud Poola kuju (FIFO)
   struct Sd *pS;       //“tupik“:  prioriteetidega LIFO-stack
//teen magasinide deskriptorid
ring: In=makeD('F',128);  //infix-sulgavaldise tekst
     Out=makeD('F',128); //postfix-avaldise tekst (inverteeritud Poola kuju)
     pS=makeD('L',16);   //Dijkstra "tupik"
     j=0;
     printf("\naritmeetiline avaldis: ");      //küsin sisendteksti
     gets(In->stack);   //loen avaldise teksti (string)
     if(strlen(In->stack)==0) goto out;  //tühi "Enter" lõpetab seansi

//filtreerin avaldise (In->stack): jäävad ainult "nähtavad" sümbolid

     for(i=0;In->stack[i]!='\0';i++){
        x=In->stack[i];
        if(isgraph(x)){      //“isgraph(x)“ on „tõene“, kui x on trükitav sümbol (v.a. tühik)
            In->stack[j]=x;
            j++;
            }
        }
     In->stack[j]='\0';   //filtreeritud tekstile lisan lõpumarkeri

//lihtsaim korrektsusekontroll: sulgude paarsus

     if(check(In->stack)!=0) goto next;
     a=0;  //tühikut ei kirjuta väljundisse eraldajaks
toPo: x=pop(In);
     if(x=='\0'){
          if(a>0) push(Out,' ');   //tühik kirjutatakse arvu järele, sisend on loetud
          goto tup;   //tupikut tühjendama
          }
     if(!isdigit(x)){
```

```
// kui x pole number, siis unaarse tehte kontroll: '-' toimib, '+' eiran, '*' ja '/' on vead
        if((Out->si==0)||((In->si-ti)==1)){   //tühi väljund või 2 järjest. eraldajat
          switch(x){
             case '-': push(Out,x); goto toPo;  //'-' Poola kujusse
             case '+': goto toPo;  //eiran: '+' ei lähe Poola kuju stringi
             case '/': printf("unaarne jagamine");  //viga
                      goto next;   //next: küsin uut avaldist
             case '*': printf("unaarne korrutamine");  //viga
                      goto next;
                  case '(': ti=In->si; goto tup;     //paar: tehe {+,-,*,/}  ja '(' : tohib
             default: printf("vale sümbol %c\n",x);  //aktsepteerimatu sümbol sisendis
                     goto next;
             }
          }
       }
     if(isdigit(x)||x=='.'){      //kui x on number või '.', siis x kuulub „arvu“ ja --> Out
         push(Out,x);
         a=1;          //lipp: "arv", arvu kogumise lõpus kirjutan väljundisse eraldaja „tühik“
           goto toPo;
         }
     if(a>0) push(Out,' ');    //arvu lõppu tühik (atof( ) jaoks, vt. „a=1“ ülalpool
     if(x!=')') ti=In->si;     //kasutan sümbolipaaride (2 tehet) tuvastamiseks
tup:  t=prtyPush(pS,Out,x);     //kirjutan x prioriteetidega magasini
     if(t==1) goto toPo;    //avaldis pole lõpuni töödeldud: jätka
     printf("\ninverteeritud Poola kuju:  %s",Out->stack);  //väljundstring: postfiks-kuju
     getchar( );  //võimaldan vaadata ekraani kuni suvalise klahvi vajutamiseni
//lahendan  (interpreteerin inverteeritud Poola kuju)
     felix(Out);  //"felix" nagu aritmomeeter (ette saab postfix-stringi: FIFO-stacki „Out“)
     getchar( );   //Felix väljastab ise avaldise väärtuse, getchar( ) annab aega seda vaadata
next: fflush(stdin);  //valmistun uue avaldise lugemiseks, fflush puhastab süst. puhvrid
     free(In); free(Out), free(pS);  // vabastan (deskriptoritega hõivatud) mälu.
     goto ring;
out:;   //tühi operaator, lõpetan
}
```

Lisa 8. Heuristika

Ülevaade

Heuristiline programmeerimine on võte, mis aitab soodsal juhul lahendada üle jõu käiva ajalise keerukusega ülesandeid (mille jaoks on põhimõtteline algoritm teada), või ülesandeid, mille jaoks teadaolevat ja universaalset algoritmi pole, või on tõestatud, et sellist algoritmi ei eksisteeri.

Alustagem selle mõiste selgitusest. Heuristika (nimetus pärineb sõnast *heureka* – kreekakeelses originaalis ευρισκειν, eesti keeles *olen leidnud* [1]) on [ENE 3, lk. 381] järgi „avastamisõpetus, uue avastamise (teaduses) ja teadmiste omandamise (õppimises) võttestik; teadusharu, mis käsitleb avastamismeetodeid ja loova mõtlemise eripära. Uuritakse võtteid, mille abil on õnnestunud leida mingi praktilise või teoreetilise probleemi ligikaudne, kuid kiire lahendus." Meie teise[2] entsüklopeedia esimene trükk [ENE 71, lk. 11] kirjutas selle mõiste pisut rohkem lahti: „...uue avastamisel toimub nn. heuristiline hüpe, sest uus teadmine (avastus) ei ole loogiliselt tuletatav olemasolevatest teadmistest. Tähtsat osa avastuses etendab näit. intuitsioon".

Computer Dictionary [CD, lk. 172 – 173] nendib, et *heuristika* on meetod või algoritm, millega saab välja töötada programmeerimisülesande vastuvõetav lahendus formaliseerimata või „iseõppivate" variantide abil. Nende ülesannete jaoks töötatakse välja esmalt „heuristiline" lahend ja seejärel püütakse seda täiustada. Paraku, [CD] hinnangul on *heuristika* mõiste algselt üsna kitsalt (ja täpselt) määratletud, ent ülemäärase kasutamise tõttu üpris hägustunud.

Allika [Filosoofia] järgi on „heuristika protsess, nagu näiteks *katse ja eksituse meetod*, et lahendada ülesandeid, mille jaoks algoritmi ei leidu. Heuristika mingi probleemi jaoks on reegel või meetod lahendusele lähenemiseks".

George Polya, 17.12.1887 Budapest – 7.09.1985 Palo Alto (USA), ungari juut (sünnijärgse nimega *György Pollak*), kelle 1945. aastal ilmunud raamatut „*How to solve it*" [Polya] peetakse teerajajaks heuristilise programmeerimise valdkonda.

Joonis L8.a. *George Polya.*

Allpool toome ära *Polya* soovitused lahendamatuna näiva ülesande lahendamiseks:

- kui teil on raskusi probleemi mõistmisel, siis püüdke joonistada asjakohane skeem (pilt, joonis);
- kui Te ei näe lahendust, siis teades oodatavat lahendust alustagegi sellest ning vaadake, kuidas tollest lahendusest saaks jõuda „tagasi minnes" algseisu;
- kui probleem on abstraktne (üldine), siis püüdke konstrueerida konkreetne näide;

[1] 1971. a. *ENE* [ENE 71, lk. 11]: „*Archimedesele* omistatav hüüatus hüdrostaatika põhiseaduse (Archimedese seaduse) avastamisel".
[2] Esimene entsüklopeesia ilmus esimese vabariigi ajal. Teine oli niisiis ENE esimene väljaanne.

- püüdke lahendada eeskätt üldisemad asjad ja seejärel detailid.

Heuristiliste algoritmide jaoks on reeglina andmestruktuuriks *graaf* kui abstraktsete andmestruktuuride seas kõige keerukam.

Noist sissejuhatavatest määratlustest võime teha paar järeldust.

- Heuristiline algoritm saab ette sisendi ja väljundi kirjeldused, ent ta ise on „kasutaja" jaoks *must kast* (so. moodul, mille täpne algoritm pole kasutajale teada).
- Heuristiline algoritm võib anda konkreetse ülesande lahendamiseks kiire programmi, mida me ei saa (üldjuhul) üldistada antud ülesande „ülemklassile" (kui see osutub võimalikuks, lakkab too üldiseksosutunud algoritm olemast *heuristiline*).
- Heuristiline algoritm on olemuslikult *optimeeriv*, ent me ei saa vastuvõetava lahendini jõudes kunagi väita, et see lahend ongi *optimaalne*.

Peatugem kahel mõistel: *optimaalne* ja *optimeerimine.* Esmalähenduses võib tunduda, et *optimiseerimise* tulem ongi alati *optimaalne*[1] lahend, ent üldjuhul see nii ei ole. Mingi funktsiooni optimum on ühe muutuja funktsiooni $z=f(x)$ jaoks sihifunktsioonist sõltuvalt kas $z=min$ või $z=max$, ning see x väärtus[2], et z oleks kas maksimaalne või minimaalne, on üheselt leitav. Kui aga $z=f(x,y)$, siis *optimaalse* (loe: tõestatavalt parima lahendi – z on parim lahend x ja y teatud väärtuste puhul) tekivad tavaliselt probleemid. Parim lahend $z=f(x)$ pole üldjuhul $z=f(y)$, kui $z=f(x,y)$. Rohkem kui ühe muutuja funktsiooni optimum[3] on üldjuhul hägune mõiste. Internetiallikas [MO] toob kahe vastandliku optimeerimiskriteeriumi näideteks järgmisi:

- kasumi maksimeerimine ja kulude minimeerimine;
- auto võimsuse maksimeerimine ja kütusekulu minimeerimine;
- masina detailide kaalu minimeerimine ja vastupidavuse maksimeerimine.

Osundatud artiklis märgitakse, et parem lõpptulemus ühe kriteeriumi jaoks saavutatakse teise kriteeriumi arvel. Meie võime oma käesoleva lisa raames nentida, et optimeerimine toimub tihti heuristikat kasutades, ja see protsess katkestatakse ratsionaalse lahenduse leidmisel (kusjuures reeglina ei saa langetada otsust lahenduse optimaalsuse kohta).

Meile tundub, et heuristika abil lahendatavate ja andmestruktuurina graafi kasutavate ülesannete kaks peamist varianti on ratsionaalse tee leidmine graafis (arvestades lisatingimustega) ja kombinatoorikaülesanded, mida püütakse lahendada graafi „ümber mängides" (so, lähtegraafi tippude vahelisi seoseid muutes)[4].

- Esimese suuna klassikaline näide on *rändkaupmehe*-ülesanne[5] (*travelling salesman problem*): graafi tipud on „linnad" ning need tuleb läbida nii, et igas tipus ollakse vaid üks kord ning „linnad" läbitakse võimalikult lühema koguteekonnaga.

---

[1] Meie jaoks on mõeldamatu, et terminist *optimaalne* saab moodustada võrdeid („optimaalsem", „peaaegu optimaalne" jne.), oma üliõpilastele olen püüdnud õpetada, et noid võrdeid tohivad moodustada ainult ajakirjanikud ja poliitikud, ent mitte lihtrahvas.

[2] Erijuhul ka $-\infty$ või $+\infty$.

[3] Ingl.k. *multiobjective optimization* [MO].

[4] Tüüpiline abstraktne andmestruktuur heuristika jaoks on *graaf.* Erijuhtudel võib selleks olla ka *string*, näiteks mustrite sobitamise (*pattern matching*) ülesande mõnede ekstremaalsete juhtude puhul. Üllatuslikult kasutavad heuristikat ka *viirusetõrje* algoritmid, mis arvatavasti käsitlevad testitavat programmi kui baidijada (stringi) juba tuvastatud baidijärjendite (viiruste) leidmiseks.

[5] Täpset ülesande püstitust püstitust vt. [Tombak 07, lk. 72 jj.].

- Teise suuna samavõrd „klassikaline“ näide on *seljakoti*-ülesanne[1]: graafi tippudel on *kaalud* ning lähtegraafist tuleb teha alamgraaf nii, et uue graafi tippude kaalude summa oleks maksimaalne (üldjuhul ei pruugi kõik lähtegraafi tipud uude graafi mahtuda). Tunnetuslikult on „seljakott“ pisut veniv mõiste (kott on presendist ja märjana saab teda venitada, eks ole), sestap tundub paremana *kasti* analoogia: meil on laudkast, millele tuleb kaas peale naelutada, ja sinna tuleb pakkida võimalikult tihedalt väiksemaid eri mõõtu risttahukakujulisi kinninaelutatud pakke.

  Allikas [wKnap] annab „tervele mõistusele“ tugineva heuristilise lähtepunkti sellele ülesandele nii: hea lahenduse leidmine on keeruline: me ei tea kiiremat lahendusviisi kui et püüame noid pakke mahutada kasti. Enamik meist (inimestest) panevad kasti esmalt suuremad asjad, ja seejärel väiksemad, suuremate ümber. Nii me ei pruugi saavutada ideaalset pakkimist, ent tavaliselt saame täiesti rahuldava tulemuse. Seda „rusikareeglit“ järgides saame üpris suure tõenäosusega hea heuristilise „pakkimis-“algoritmi (vt. ka [Heuristics]).

  Muide, me saame selle ülesande lihtsalt rekursiivseks muuta, kui paneme paika, et kasti pakitavad „pakid“ pole kinni naelutatud, vaid on tühjad pappkastid, kuhu saame paigutada väiksemaid, samuti tühje karpe.

Ehkki iga heuristilise lahenduse programm[2] on reeglina unikaalne, on selliste lahenduste leidmisel akumuleerunud teatud eesmärgile viivad üldised lähenemisteed; noid nimetatakse tavaliselt *metaheuristika*ks,

Heuristika strateegiatest

Alustagem mõningate heuristikavaldkonna üldiselt omaksvõetud mõistete tutvustamist.

- **ahne** (*greedy*) **algoritm** on „algoritm, kus vastuse otsimisel võetakse järgmisena alati vahetult parim või kohalik lahend. Ahne algoritm leiab alati teatud ülesande üldiselt ehk globaalselt optimaalse[3] lahendi, kuid võib leida mõne teise ülesande teatud variandi mitte-optimaalse lahendi“ (vt. [gtg]). Ahne algoritm on **kiire tagurduseta**[4] algoritm hea lubatud lahendi iteratiivseks leidmiseks heuristilise valiku teel [samas].
- **pimeotsing** „on probleemi lahendamine jõumeetodil, võimaluste proovimisel ei kasutata olemasolevaid teadmisi. Heuristilise otsingu puhul kasutatakse otsuste tegemisel ära teadmisi otsinguruumi kohta“ [DBogdanov, lk. 20].
- **lokaalne otsing** (*local search*, vt. [wLocal]) on strateegia, mis otsib lahenduse jatkuteed oma lähiümbrusest alustades (meenutagem, et andmestruktuur on *graaf*); seejuures kasutatakse informatsiooni naabrite kohta ning üritatakse teha jätkamiseks ratsionaalne valik. Märgitakse, et nii saab lahendada min-max-ülesandeid, rändkaupmehe-ülesannet ja *Boole*'i funktsiooni lahendite arvu leidmise (#*SAT*) ülesannet[5].

---

[1] Ingl. k. *knapsack problem* või *rucksack problem.*

[2] Paljud autorid väldivad heuristikast kirjutades *algoritmi* mõistet; viimane on midagi kindlat, tõestatut ja universaalset. Heuristikaga saame kirjutada vastuvõetava *programmi*, ent me ei saa usutavalt kinnitada, et meie lahenduskäik võiks olla üldistatav (ent algoritm peab seda olema). Kui me edaspidi kirjutame *heuristilistest algoritmidest,* siis tasuks seda märkust meenutada,

[3] Loe: *vastuvõetava.*

[4] *Tagurdus* tähendab loobumist mõnest viimasest otsimissammust ja tagasivõtmise lähtepunktis mingi muu otsin-guteega alustamist. Vältimaks tsüklisse sattumist tuleb hüljatud otsinguteed meeles pidada.

[5] *Mati Tombak* ja allakirjutanu kasutasid heuristikat *Dedekind*i arvude leidmise kiire algoritmi väljatöötamisel.

„Aktiivsel“ tipul on tavaliselt mitu vahetut naabrit ning otsingu jätkamiseks valitakse üks neist, siit tuleneb ka meetodi nimetus *lokaalne* otsing. Kui naabrite hulgast jätkamiskandidaadi valikul kasutatakse lokaalset maksimeerimiskriteeriumi, siis on tegemist lokaalse otsingu variandiga, mida tuntakse „mägironimise“ (*hill climbing*) heuristikuna.

- **tabuotsing** (*Tabu* (*taboo*) *search*) on sisuliselt *lokaalse otsingu* erijuht. Olles tipus *X* ja valides jätkuteed naabertippude *Y*, *Z* ja *W* hulgast võib *rändkaupmees* kasutada infot naaberlinnade karakteristikute kohta, näiteks, kui *Y* ja *W* on lähedal ja neisse jõuab umbes sama ruttu, ent *Z* on igas mõttes (tee pikkus ja sõiduaeg) kaugemal, siis pannakse tipp *Z* „keelatud teede nimekirja“ (*tabu*-listi). *Z*-linna külastatakse hiljem ja üpris kindlasti teiste linnade kaudu [wTabu].

  Tabu-tehnika on kasulik vältimaks heuristilise otsingu „tsüklisse“ jäämist, ja teda kasutatakse kombinatoorikaülesannete lahenduste otsimisel [Tabu].

Kolm metaheuristikat

Järgnevas toome näiteks kolm heuristilist otsingustrateegiat (vt. [GS] ja [wMH]):

- Sipelgapesa-optimeerimine (*ant colonies optimization*): otsitakse ratsionaalseid teid graafis, katse-eksituse-meetodil, ning parematena tunduvatele kaartele omistatakse suuremaid kaale (putukaterminoloogias *feromooni*); nii on lahendatud mh. lahendamatuna tunduv *tunniplaani-ülesanne*[1]. (vt. ka [MaxMin]).
- Simuleeritud lõõmutus (*simulated annealing* (vt. [wSim])). Meetodi nimi vihjab metallurgiale: töödeldavat metalli jahutatakse protsessi kontrollides soovitud resultaadi saavutamiseni; vajadusel võib ka temperatuuri tõsta. Globaalse parameetri *T* (t°) algväärtus on *max* ning lahenduse leidmisel on see kas minimaalne või vähemalt vastuvõetav: minimeerida püütakse näiteks energiakulu. Lahendusteid otsitakse vahetust naabrusest[2] (vt. *lokaalne otsing*).
- 3D-marsruut („mägironimine“, *greedy algorithm and hill-climbing, random-restart hill climbing*)[3].

Püüdkem heuristiliste algoritmide klassi kokku võtta. Sedastagem, et see variant *võib* leida vastuvõetava algoritmi, ent ei pruugi: ta püüab lahendada ülesande etteantud piirangute raames; piiranguteks võivad olla lahendusaeg ning mitmesugused tingimused, mida lahend peab rahuldama. Näiteks, tunniplaani-ülesande „tingimused“ peavad arvestama olemasolevate ruumide ja õpetajate võimalustega ning õpilasi säästva vaheldusega (näiteks, et kõik nädala matemaatikatunnid poleks ühel päeval üksteise järel), lisaks võivad olla õpetajatel omad (nõrgad) tingimused, näiteks mõni õpetaja ei taha alustada kell 8, mõni teine aga tahab (lasteaeda lapse järele) minna hiljemalt kell 6 õhtul.

Teema lõpetuseks meie raamatus nentigem, et ei *metaheuristika* ega ka *heuristika* pole oma olemuselt ei formaliseeritavad ega ka täielikult üldistatavad. Kui see mingi heuristilise algo-

[1] *Aleksei Bogdanov* [ABogdanov] lahendas sel meetodil ülesande 200 õpilase, 400 aine (aine × klassid), 45 ajagraafiku (esmaspäev kell 8 kuni reede kell 17), 10 ruumi, 5 „eripära“, 40 õpetaja ja 5 „nõrga kitsendusega“.

[2] Viidatud allikas on näide rändkaupmehe-ülesande keerukusest: kui linnu on kõigest (n=) 20, siis võimalike teede arv on 20!=2432902008176640000, ent heuristiliselt – lokaalse otsinguga – teeme vaid $n(n-1)/2$=190 sammu

[3] *Dan Bogdanov* modelleeris maastikku ja selle läbimist mobiilse agendi jaoks graafi-andmemudelitel, esimene põhineb kolmnurkade kahendpuul ja teine ruutude neljandpuul [DBogdanov].

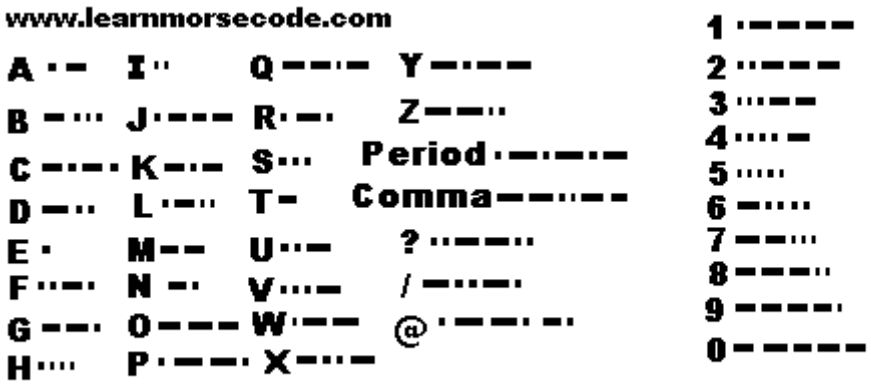
ritmi puhul õnnestub (so, me saame ta formaliseerida ja üldistada), siis lakkab ta olemast *heuristiline* algoritm. Mingi vihje heuristilise algoritmi arendamisest „normaalseks“ võib anda viimase iseloomustamine „ahne algoritmina“ (vt. näit. [Kiho 03, lk. 86]).

Kui meil on mingi ülesande lahendamiseks teadaolev vastuvõetava kiirushinnanguga algoritm, siis pole mingit põhjust heuristika rakendamiseks. Kui aga pole, siis võib (aga ei pruugi) heuristikast kasu olla.

Allikas [Heuristics] võtab probleemi kokku nii: heuristiline algoritm võib anda vastuvõetava lahenduse praktilisele ülesandele, ent ta ei taga lahenduse formaalset korrektsust. Heuristikat kasutatakse tüüpiliselt neil juhtudel, kui optimaalse lahendi leidmiseks (aja, mälu jne. suhtes) pole teada algoritmi. Arvutiteaduse põhieesmärgiks on leida tõestatult hea töökiirusega ning kvaliteediga algoritme. Heuristika „hülgab“ tavaliselt kas mõlemad sihid või ühe neist: ta võib leida väga hea lahenduse, ent pole garantiid, et resultaat tuleb samavõrra paha; heuristikaga tehtud programm võib töötada väga kiiresti, ent see ei tähenda, et mingi teise probleemi analoogiline lahendus töötaks samuti vastuvõetava ajaga.

Vististi ilmaaegu ei pannud *Knuth* oma elutöö pealkirjaks „Programmeerimis**kunst**“ – heuristiline otsing ongi rohkem kunst kui teadus (ehkki *Knuth ise* esitab ainult rangelt hinnatavaid algoritme).

Lisa 9. Morse tähestik



http://www.learnmorsecode.com/ **(13.07.09)**

Kasutatud materjalid

[A,H&U] А. Ахо, Дж. Хопкрофт, Дж. Ульман, Построение и анализ вычислительных алгоритмов, Издательство «Мир», Москва, 1979.

[Astrachan] Owen Astrachan, Bubble sort: An Archaelogical Algorithmic Analysis, Computer Science Department, Duke University, 2003.
http://www.cs.duke.edu/~ola/papers/bubble.pdf (29.12.08)

[AV] http://wapedia.mobi/fi/Georgi_Adelson-Velski (12.01.09)

[AVL] http://en.wikipedia.org/wiki/AVL_tree (13.01.09)
[bait] http://en.wikipedia.org/wiki/Byte (20.01.09)

[Bauer] http://www.answers.com/topic/friedrich-l-bauer (17.12.08)

[Bauer jt.] Friedrich L. Bauer, Rupert Gnatz, Ursula Hill, Informatik, Springer-Verlag, Berlin Heidelberg New York, 1975.

[Binstock&Rex] Andrew Binstock and John Rex, Practical Algorithms for Programmers, Addison-Wesley Publishing Company, 1995.

[B-keel]:

[Lebherz] Eric Lebherz, Hypernews Computer Language List v. 1.4, http://www.hypernews.org/HyperNews/get/computing/lang-list.html (4.05.09)

[wB] B (programming language), http://en.wikipedia.org/wiki/B_(programming_language) (8. 04.09)

[Johnson&Kernighan] S. C. Johnson, B. W. Kernighan, The Programming Language B, http://cm.bell-labs.com/cm/cs/who/dmr/bintro.html (4.05.09)

[Thompson] Ken Thompson, Users' Reference to B, http://cm.bell-labs.com/cm/cs/who/dmr/kbman.html (4.05.09)

[Kernighan] B. W. Kernighan, A Tutorial Introduction to the Language B, http://cm.bell-labs.com/cm/cs/who/dmr/btut.html (4.05.09)

[ABogdanov] Aleksei Bogdanov, Optimeerimisülesannete lahendamine sipelgakoloonia metaheuristika abil, magistritöö, Tartu Ülikool, Matemaatika-informaatikateaduskond, ATI, Tartu 2007 (käsikiri).

[bootstrapping] http://en.wikipedia.org/wiki/Bootstrapping_(compilers) (30.11.08), http://en.wikipedia.org/wiki/Bootstrapping_(computing) (1.12.08)

[B-tree] http://en.wikipedia.org/wiki/B-tree (12.01.09)

[Buldas jt.] A. Buldas, P. Laud, J. Villemson, Graafid, Tartu Ülikool, Arvutiteaduse instituut, Tartu 2003.

[CD] Computer Dictionary, Microsoft Press®, 1991.
Толовый словарь по вычислительной технике, Microsoft Press®, Русская редакция, 1995.

[chcoord] A. Isotamm, J. Jagomägi, M. Tombak, H. Türnpu, Programmipakett Chcoord1.10 kaartide matemaatiliste aluste ja geodeetiliste andmete teisendamiseks, Kaardikoja Teataja, 1, 1994, lk.11-14.

[CPL] CPL programming language,
http://knowledgerush.com/kr/encyclopedia/CPL_programming_language/ (5.05.09)

[C_S] Counting sort, http://en.wikipedia.org/wiki/Counting_sort (6.01.09)

[DBogdanov] Dan Bogdanov, Optimaalse teekonna leidmise ülesande lahendamine geomeetriliste objektide puude abil, TÜ, MIT, Arvutiteaduse instituut, semestritöö, Tartu 2004 (käsikiri).

[Denniston] Robin Denniston, 30 sala-aastat. A. G. Dennistoni töö signaalluures 1914 – 1944, „Tammerraamat“, 2009.

[Eckhouse&Morris] Р. Экхауз, Л. Моррис, Мини-ЭВМ: Организование и программирование, М., «Финансы и статистика», 1983.

[EE 8] Eesti Entsüklopeedia 8, Tallinn 1995.

[EE 9] Eesti Entsüklopeedia 9, Tallinn 1996.

[Eight_q] http://en.wikipedia.org/wiki/Eight_queens_puzzle (6.12.08)

[ENE 3] Eesti Nõukogude Entsüklopeedia, 3, Tallinn, Kirjastus „Valgus“, 1988.

[ENE 71] Eesti Nõukogude Entsüklopeedia, HERN – KIRU, Tallinn, 1971.

[EWD] http://en.wikipedia.org/wiki/Edsger_W._Dijkstra (25.12.08)

[Feather] Clive Feather, A brief(ish) description of BCPL,
http://www.lysator.liu.se/c/clive-on-bcpl.html (4.05.09)

[Filosoofia] http://wiki.zzz.ee/index.php/Filosoofia_m%C3%B5isted (15.02.09)

[GNU C L] The GNU C Library, http://www.gnu.org/s/libc/manual/html_node/index.html (30.09.07)

[Goebel] Greg Goebel, Codes, Ciphers, & Codebreaking,
http://www.vectorsite.net/ttcode.html (7.08.09)

[Gutknecht] Programmeerimiskeelte arengu tragöödia (2), tõlkinud Anu Oja, „Arvutustehnika ja andmetöötlus“, 2’94, lk. 10..15.

[GS] http://www.cc.ioc.ee/jus/gtglossary/gtglos_h.htm (15.02.09)

[GSV] http://personal.rhul.ac.uk/uhah/058/talks/inaugural.pdf (12.07.09)

[gtg] http://www.cc.ioc.ee/jus/gtglossary/gtglos_g.htm (ahne algoritm, 3.04.09)

[Heuristics] http://en.wikipedia.org/wiki/Heuristics (14.02.09)

[Hoare] http://en.wikipedia.org/wiki/C._A._R._Hoare (4.01.09)

[Iliffe_1] Iliffe vector, http://en/wikipedia/org/wiki/Iliffe_vector (26.10.08)

[Iliffe_2] wapedia: Wiki: Iliffe vector, http://wikipedia/mobi/en/Iliffe_vector (12.11.08)

[Isotamm, AA&P] Ain Isotamm, Andmed, andmemudelid ja päringukeeled, Tartu Ülikool, Majandusteaduskond, Rahanduse ja arvestuse instituut, TÜ Kirjastus, Tartu 1996,

[Isotamm, PK] Ain Isotamm, Programmeerimiskeeled, TÜ Matemaatika-informaatikateaduskond, Arvutiteaduse instituut, TÜ Kirjastus, Tartu 2007.

[IŠ]. Ain Isotamm, Nikita Šipilov, Automaadid, keeled ja translaatorid, TÜ ATI, käsikiri.

[Jensen] Ted Jensen, A Tutorial on Pointers and Arrays in C, version 1.2, Feb. 2000, http://home.netcom.com/~tjensen/ptr/ch1x.htm

[Kaasik, PL] Programmeerimisleksikon, „Programme kõigile“, koostanud Ü. Kaasik, Tartu Riiklik Ülikool, Tartu 1987.

[K&R] Brian W. Kernighan, Dennis M Ritchie, The C Programming Language, Second Edition, AT&T Bell Laboratories, Murray Hill, New Jersey, sine data.

[KjaK] T. Kelder, Ü. Kaasik, Programmeerimiskeel C, Programme kõigile, Tartu Ülikool, Arvutuskeskus, Tartu 1989.

[KjaK-2] T. Kelder, Ü. Kaasik, Programmeerimiskeele C standardfunktsioonide teek, Programme kõigile, Tartu Ülikool, Arvutuskeskus, Tartu 1990.

[Kelder] Tõnis Kelder, Rersonaalarvuti EC-1840 baastarkvara loomisest, kogumikus „40 aastat arvutuskeskust“, Tartu Ülikool, Arvutuskeskus, Tartu 1999.

[Kiho 97] J. Kiho, Algoritmid ja andmestruktuurid, Tartu Ülikool, Arvutiteaduse instituut, Tartu 1997.

[Kiho 03] Jüri Kiho, Algoritmid ja andmestruktuurid, Tartu Ülikool, Arvutiteaduse instituut, Tartu 2003.

[Kiho 05] Jüri Kiho, Algoritmid ja andmestruktuurid, Ülesannete kogu, Tartu Ülikool, Arvutiteaduse instituut, Tartu 2005.

[Knuth] http://www-cs-faculty.stanford.edu/~knuth/ (21.11.08)

[Knuth I] Д. Кнут, Искусство программирования для ЭВМ, т. 1, Основные алгоритмы, «Мир», М 1976.

[Knuth III] Д. Кнут, Искусство программирования для ЭВМ, т. 3, Сортировка и поиск, «Мир», М 1978.

[Knuth TAoCP] http://www-cs-faculty.stanford.edu/~knuth/taocp.html (22.11.08)

[Lafore] Robert Lafore, The Waite Group's Turbo C®, Programming for the PC, Revised Edition, Howard W. Sams & Company, 1989.

[Laur] Sven Laur, Avaliku võtmega krüptograafia, http://home.cyber.ee/ahtbu/aturve_9.ppt. (1.02.09).

[Lebedev] Лебедев В. Н., Введение в системы программирования, М, «Статистика». 1975.

[Lebherz] Eric Lebherz, Computing Languages List, http://www.hypernews.org/HyperNews/get/computing/lang-list.html (07.09.05)

[Leppikson A&A] V. Leppikson, „Algoritmid ja andmestruktuurid" (IAG0090), http://www.tud.ttu.ee/material/leppikson/Algoritmid%20ja%20andmestruktuurid.htm (01.07.08)

[Leppikson C] Viktor Leppikson, Programmeerimine C-keeles, Teine väljaanne, „Külim", Tallinn, *sine data.*

[L&P] Rhydian Lewis and Ben Paechter, Application of the Grouping Genetic Algorithm to University Course Timetabling, Centre for Emergent Computing, Napier University, Edinburgh EH10 5DT, UK. {r.lewis|b.paechter}@napier.ac.uk

[Litover] Mati Litover, Graafiteooria sõnastik, http://www.cc.ioc.ee/jus/gtglossary/gtglos_g.htm#graphic (19.02.09)

[Luik] Viivi Luik, Me kõik olime vabakäiguvangid, „Maaleht", *11.12.2008*

[matrix_product] http://www.tutor.ms.unimelb.edu.au/matrix/matrix_product.html (13.11.08)

[MaxMin] A MAX-MIN Ant System for the University Course Timetabling Problem – K. Socha et al – ANTS2002.

[Mereste I] U. Mereste, Statistika üldteooria I, Statistiline vaatlus, Tartu Riiklik Ülikool, Tartu 1971.

[Mereste II] U. Mereste, Statistika üldteooria II, Keskmised ja variatsiooni näitarvud, Tartu Riiklik Ülikool, Tartu 1971.

[Mereste 2] Uno Mereste, Toimunust & kaasaelatust, 2. osa, SE&JS, Tallinn 2004.

[MO] http://en.wikipedia.org/wiki/Multiobjective_optimization (15.03.09)

[MOS] Ü. Kaasik, H. Espenberg, E. Etverk, O. Rünk, A. Vihman, Matemaatika oskussõnastik, „Valgus“, Tallinn 1978.

[MSDNL] Predefined Macros, C/C++ Preprocessor Reference.

[Narinjani] Александр Семенович Нариньяни, О советской программе форсированного развития ЭВМ, Рабочая записка, 02.08.1985, http://tapemark/narod/ru/narinjani.html (28.09.08)

[Norbeciak] Maciej Norberciak, Universal Method for Solving Timetabling Problems Based on Evolutionary Approach, Proceedings of the International Multiconference on Computer Science and Information Tecnology, pp. 149 – 157, 2006.

[One-time Pad] http://users.telenet.be/d.rijmenants/en/onetimepad.htm (13.07.09)

[O'Reilly] http://www.oreilly.com/catalog/pcp3/chapter/ch13.html (4.10.2007)

[PDP-11] http://en.wikipedia.org/wiki/PDP-11 (8.04.09)

[PK] К. Хоор, Обработка записей, «Языки пропраммирования» стр. 278 – 343, «Мир», М 1972.

[Polya] http://en.wikipedia.org/wiki/How_to_Solve_It (15.02.09) George Pólya's 1945 book ***How to Solve It,*** http://www-gap.dcs.st-and.ac.uk/~history/Biographies/Polya.html (17.02.09)

[PoSa] Pool sajandit arvutit Tartu Ülikoolis, Tartu 2009.

[Pratt] Т. Пратт, Языки программирования: разработка и реализация, «Мир», Москва, 1979.

[Pressman] Roger S. Pressman, Software Engineering, A Practitioner's Approach, International Edition, 1992.

[Pritchard] http://www.cs.ucf.edu/courses/cot4810/fall04/presentations/Hash_Tables.ppt. (20.01.09)

[pronto] http://www.fm.ee/pronto/index.php?/archives/P7.html (1.01.09)

[Pöial] Jaanus Pöial, Algoritmid ja andmestruktuurid (I209),
http://enos.itcollege.ee/~jpoial/algoritmid/ (1.07.08)

[Pöial-U] Jaanus Pöial, Unix (Linux) lühikonspekt,
http://enos.itcollege.ee/~jpoial/java/Unix_lyhikonspekt.html (28.08.07)

[Rauk] Melanie Rauk, Inglise – eesti sõnaraamat koolidele, Tallinn, Valgus 1988.

[Richards] Martin Richards, BCPL, http://www.cl.cam.ac.uk/~mr10/BCPL.html (8.04.09)

[Ritchie] Dennis M. Ritchie, The Development of the C Language, http://plan9.bell-labs.com/who/dmr/chist.html (28.08.08)

[RPN] http://en.wikipedia.org/wiki/Reverse_Polish_Notation , http://en.wikipedia.org/wiki/Reverse_Polish_Notation#Converting_from_infix_notation http://en.wikipedia.org/wiki/Shunting_yard_algorithm http://www.absoluteastronomy.com/topics/Shunting_yard_algorithm (24.12.08)

[Shell] Shell sort, http://en.wikipedia.org/wiki/Shell_sort (29.12.08)

[Simonyi] Charles Simonyi, Hungarian Notation, 1999. http://msdn.microsoft.com/enus/library/aa260976.aspx (25.05.09)

[Sinivee] Veiko Sinivee, Programmeerimiskeel C, http://www.hot.ee/veiks26/cprog/cprog210.html (2.07.08)

[Sorting Algorithms] http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html (4.10.08)

[stackwiki] http://wikipedia.org/wiki/Stack_(data_structure) (22.10.08)

[sys/stat] http://opengroup.org/onlinepubs/007908775/xsh/sysstat.h.html (23.09.09)

[sys/times] http://www.opengroup.org/onlinepubs/000095399/basedefs/sys/times.h.html (21.09.09)

[Škarupelov] Vladimir Škarupelov, Text analysis and applications, Bachelor work, Tartu, 2008 (käsikiri)

[Tabu] http://www.cs.sandia.gov/opt/survey/ts.html (31.03.09)

[Tavast&Hanson] Arvi Tavast, Vello Hanson, Arvutikasutaja sõnastik inglise – eesti, ilo sõnastik, Ilo kirjastus, 2003.

[Thompson] Ken Thompson: a brief introduction, http://www.linfo.org/thompson.html (13.10.08)

[Tombak 07] Mati Tombak, Keerukusteooria, Tartu Ülikool, Tartu 2007.

[Tombak] Mati Tombak, Counting Complexity – Applications and Benchmarks, February 28, 1998 (käsikiri).

[Tombak jt] M. Tombak, A. Isotamm, T. Tamme. On logical method for counting Dedekind numbers. *Lecture Notes in Computer Science* 2138, 424-427, Springer-Verlag 2001.

[Tretjakov] Konstantin Tretjakov, Algoritmide keerukus, 6. november 2004 (käsikiri).

[Truu] Ahto Truu, Nupunurk, „Arvutustehnika ja andmetöötlus“ 1’95, lk. 32..36.

[Unix]: (kõik vaadatud viimati19.04.09)

Jaanus Pöial, Unix (Linux) lühikonspekt,
http://enos.itcollege.ee/~jpoial/java/Unix_lyhikonspekt.html

Operatsioonisüsteem UNIX, http://www.cs.ut.ee/~epuman/unix.html

UNIXi 20 käsku, Arvutid ja Internet,
http://www.vykk.vil.ee/tanel/materjalid/tigu/tigu.4.html

Basics of the Unix Philosophy, http://www.faqs.org/docs/artu/ch01s06.html

Unix commands reference ard, http://www.indiana.edu/~uitspubs/b017/

UNIX Command Summary, Stanford University,
http://unixdocs.stanford.edu/unixcomm.html

Basic UNIX commands, http://mally.stanford.edu/~sr/computing/basic-unix.html

Unics Commands, http://8help.osu.edu/wks/unix_course/intro-137.html

Unix Command Summary, http://www.math.utah.edu/lab/unix/unix-commands.html

[wUnix] http://en.wikipedia.org/wiki/Unix

[v.Neumann] http://ei.cs.vt.edu/~history/VonNeumann.html (3.01.08)

[Vigonski] Simon Vigonski, Kiir- ja ühildusmeeetodi võrdlus. Kodutöö aines „Algoritmid ja andmestruktuurid“, Tartu Ülikool, Matemaatika-informaatikateaduskond, Arvutiteaduse instituut, Tartu 2008 (käsikiri).

[VILLIS]

- Виллемс А. Л, Изотамм А. А., Организация данных в системе VILLIS. Труды ВЦ ТГУ, 1974, вып. 30.
- Виллемс А. Л, Изотамм А. А., Томбак М. О., Суперзапись – способ представления сложных объеков. Труды ВЦ ТГУ, 1974, вып. 30.
- Виллемс А. Л, Изотамм А. А., Томбак М. О., Генератор отчетов VILLIS. Труды ВЦ ТГУ, 1974, вып. 30.
- Виллемс А. Л, Изотамм А. А., Генератор отчетов VILLIS для ЭВМ «Мнск-32». Труды ВЦ ТГУ, 1976, вып. 38.
- Виллемс А. Л, Изотамм А. А., Организация данных в системе VILLIS. Труды ВЦ ТГУ, 1976, вып. 38.
- Виллемс А. Л, Изотамм А. А., Язык манипулирования данными в системе VILLIS. Труды ВЦ ТГУ, 1976, вып. 38.

[Vleck] Tom Van Vleck, Unix and Multics, http://www.multicians.org/unix.html (1.05.09)

[wArray] Array, http://en.wikipedia.org/wiki/Array (9.11.08)

[wAT] http://et.wikipedia.org/wiki/Alan_Turing (14.04.09)

[wBCPL] BCPL, http://en.wikipedia.org/wiki/BCPL (8.04.09)

[wBombe] http://en.wikipedia.org/wiki/File:TuringBombeBletchleyPark.jpg (12.07.09)

[wCrypto] http://en.wikipedia.org/wiki/Cryptanalysis_of_the_Enigma#British_bombe (12.07.09)

[wDumey] http://en.wikipedia.org/wiki/Arnold_Dumey (19.01.09)

[Webster's] Webster's II New Riverside University Dictionary, The Riverside Publishing Company, Boston, 1984.

[wEnigma] http://en.wikipedia.org/wiki/Enigma_machine (12.07.09)

[wGen] http://en.wikipedia.org/wiki/Genetic_algorithm (31.03.09)

[wHistory] http://en.wikipedia.org/wiki/History_of_cryptography (14.07.09)

[wHT] http://en.wikipedia.org/wiki/Hash_table (20.01.09)

[wKernighan] http://en.wikipedia.org/wiki/Brian_Kernighan (11.08.09)

[Wirth] Wirth N., Kolmkümmend aastat programmeerimiskeeli ja translaatoreid, tõlkinud *Jaan Penjam*, A&A, nr.1, 1993, lk. 13 – 24.

[wKnap] http://en.wikipedia.org/wiki/Knapsack_problem (2.04.09)

[wKT] http://www.en.wikipedia.org/wiki/Ken_Thompson (13.10.08)

[wLocal] http://en.wikipedia.org/wiki/Local_search_(optimization) (3.04.09)

[wMH] http://en.wikipedia.org/wiki/Metaheuristic (16.02.2009)

[wSec] http://en.wikipedia.org/wiki/Second-system_effect (06.06.09)

[wSiemens] http://en.wikipedia.org/wiki/File:Fernscheiber_01.jpg (15.07.09)

[wSim] http://en.wikipedia.org/wiki/Simulated_annealing (31.03.09)

[wTabu] http://en.wikipedia.org/wiki/Tabu_search (31.03.09)

[wThearded] http://en.wikipedia.org/wiki/Forth_virtual_machine (13.06.09)

[wTimes] http://www.opengroup.org/onlinepubs/009695399/functions/times.html (20.09.09)

[wTuring] http://en.wikipedia.org/wiki/Turing_Award (13.04.09)

[wOTP] http://en.wikipedia.org/wiki/One-time_pad (1.07.09)

[wOTPe] http://en.wikipedia.org/wiki/One-time_pad#Example (2.07.09)

[wVigenère] http://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher (29.07.09)

[Yngve_1] http://en.wikipedia.org/wiki/Victor_Yngve (6.11.08)

[Yngve_2] http://www.bookrags.com/wiki/Victor_Yngve (6.11.08)

Indeks