PROGRAMEERIMISKEEL C

Veiko Sinivee

Sisukord

Sissejuhatus 1
Programmeerimiskeel C 1
C - keele programmi ehitusest 4
Abiprogramm MAKE 13
Enamkasutatavad funktsioonid 16
Funktsioonid printf() ja scanf() 17
Andmetüüpide konverteerimisfunktsioonid 21
Muutujad 22
Konstandid 25
Andmetüübid 26
Täisarvud 28
Murdarvud 30
Sümbolid 31
Tuletatud tüübid 32
Massiivid 33
Viidad 34
Andmestruktuurid 36
Funktsioonide tüübid ja viidad 39
Avaldised ja laused 40
Tehtemärgid 42
Programmi töö juhtimine 46
Tingimuslause if 47
Tingimuslause switch...case 48
While tsükkel 50
For tsükkel 51
Käsk goto 52
Funktsioonide parameetrid 53
Makrod ja sümboolsed konstandid 57
Mälu haldamine 59
Mälumudelid ja protsessori ehitus 64
Mälumudelid 67
Programmi laadimine 70
Assembleri kasutamine 75
Katkestused 85
Katkestustele vastamine 93
Standardsed funktsioonid 97

Sisend-/väljundfunktsioonid 99
Puhverdatud funktsioonid 99
Puhverdamata funktsioonid 107
Näiteprogramm HEXEDIT 114
Graafilised funktsioonid 125
Joonelemendid, pikselid ja rasterpildid 129
Pindelemendid 134
Graafiline tekst ja aknad 138
Näiteprogramm GRADEMO 141
Mälu reserveerimine 146
Kontrollfunktsioonid 152
Operatsioonisüsteemi funktsionid 155
Failid ja kataloogid 159
Matemaatilised funktsioonid 163
Tarkvara loomine 165
Nõudmiste analüüs 167
Andmevoogude analüüs 169
Programmi struktureerimine 171
Andmevoogudel põhinev struktureerimine 173
Kodeerimine 174
Vigade otsimine 177
Optimeerimine 179
Programmi optimeerimise võimalused 181

Jooniste loetelu

Joonis 1: Programeerimiskeele C programmi struktuur 5
Joonis 2: Määratud integraali arvutamine ristkülikureegli järgi............................12
Joonis 3: Programmi parameetrite salvestamine 47
Joonis 4: Mälu haldamine operatsioonisüsteemis DOS 60
Joonis 5: Programmi segmentide jaotus 61
Joonis 6: Protsessori i8086 ehitus 65
Joonis 7: VROOMi jaoks vajalikud lisanduvad programmisegmendid 72
Joonis 8: Funktsiooni pinuraami ehitus 77
Joonis 9: Ekraanile kuvatud sümboli atribuudi bittide tähendused 110
Joonis 10: Andmetekulg protsessorist ekraanile 125
Joonis 11: Funktsioonid line(), lineto() je linerel() 131
Joonis 12: Nurgad ja ellipsid 134
Joonis 13: Mustrielemendi ehitus 136
Joonis 14: Faili aja ja kuupäeva salvestamine 160
Joonis 15: Andmevoogudel baseeruv analüüsimeetod 168
Joonis 16: Andmevoogudel baseeruva analüüsi näide 169
Joonis 17: Transformatsioonivoo struktureerimine...173
Joonis 18: Vastasmõjuvoo struktureerimine 173
Joonis 19: Turbo Profileri aken 179

Tabelite loetelu

Tabel 1: Käsusümbolid (*escape sequences*)........18
Tabel 2: Tüübi määrused........19
Tabel 3: Formaadi lipud........20
Tabel 4: Täisarvude tüübid........29

Tabel 5: Murdarvude tüübid........30
Tabel 6: Lausete tüübid........41
Tabel 7: Ühekohalised tehtemärgid........42
Tabel 8: Kahekohalised tehtemärgid........43
Tabel 9: Tehtemärkide prioriteet ja assotsiatiivsus........45
Tabel 10 : Programeerimiskeele C mälumudelid........69
Tabel 11: Funktsiooni väärtuse väljakutsujale loovutamise viis........78
Tabel 12: Registrite CS ja DS sisu erinevate mälumudelite puhul........81
Tabel 13: Funktsiooni *fopen()* parameetri mode võimalikud väärtused........104
Tabel 14: Sümboolsed konstandid tekstivärvuse määramiseks........111
Table 15: Tekstirezhiimide omadused........112
Tabel 16: Graafikarezhiimid, lahutusvõimed ja värvid........126
Tabel 17: Funktiooni *putimage()* parameetri *options* väärtused........132
Tabel 18: Täitemustritele vastavad sümboolsed konstandid........136
Tabel 19: Borland C/C++ graafilised šriftid........139
Tabel 20: Graafilise teksti joondamine........139
Tabel 21: Ekraanileheküljed ja graafikarezhiimid........141
Tabel 22: Sümbolite klassifitseerimise funktsioonid........153
Tabel 23: Andmetüüpide minimaalsed ja maksimaalsed väärtused........155
Tabel 24: Borland C/C++ matemaatikafunktsioonid........163

Sissejuhatus

See raamat käsitleb programmeerimiskeeli C ja C++. Raamat on mõeldud kõigile, kes soovivad õppida tundma nimetatud programmeerimiskeeli või värskendada oma teadmisi nendest. Raamatus antakse vaid programmeerimiskeeltest C ja C++ ülevaade ning käsitletakse üksnes enamkasutatavaid funktsioone ja programmeerimistehnikaid. Seepärast ei sobi käesolev raamat kasutamiseks käsiraamatuna. Igal translaatoril on oma funktsioonide kogumik, millest vaid osa on sarnased kõigi teiste translaatorite funktsioonidega. Siin käsitletakse vaid selliseid funktsioone. Ülejäänud funktsioonid ja nende kasutamine on kirjas teie translaatori käsiraamatutes.
Raamatus üritatakse piirduda standardsete funktsioonidega. Kuna aga näiteks graafikafunktsioonid ei ole C-keeles standardiseeritud, siis on osa näiteprogramme transleeritavad vaid siinkasutatud translaatori - Borland C/C++ 2.0 abil.
Näiteprogrammide tekstid on toodud otse vastava teema juures. Nende paremaks eristamiseks muust tekstist on sel puhul kasutatud `courier` šrifti. See šrift on natuke nurgelisem ja seega ka paremini loetavam, mis on programmi tekstide juures üsna tähtis. C ja C++ keele võtmesõnad on trükitud **rasvaselt**. Funktsioonide nimed on paremaks eristamiseks trükitud *kaldkirjas*. Failide nimed on sisestatud SUURTÄHTEDES. Käskude osad, mille kasutamine ei ole kohustuslik, ümbritsetakse kandiliste sulgudega, näiteks - [-c -o -i]. Käskude osad, mis ei ole ei võtmesõnad ega valikud ja mille sisu te seega peate ise teadma, ümbritsetakse nurksulgudega, näiteks: CC [<valikud>] <faili nimi>. Mitme võimaliku valiku puhul, millest vaid üks on sobiv, ümbritsetakse kõik valikud looksulgudega ja eraldatakse üksteisest püstkriipsudega, näiteks { a | b | c}.
Raamatu esimene osa käsitleb programmeerimiskeelt C ja teine programmeerimiskeelt C++. Raamatust arusaamiseks ei ole vajalik omada eelteadmisi nimetatud programmeerimiskeeltes.

Programmeerimiskeel C

Programmeerimiskeel C loodi firma AT&T uurimisasutuses ja tema autoriks on Dennis Ritchie. Samas asutuses loodi ka uus operatsioonisüsteem *UNIX*, mis erinevalt teistest selleaegsetest operatsioonisüsteemidest ei olnud programmeeritud masinkoodis, vaid hoopis programmeerimiskeeles C. See võimaldas nimetatud operatsioonisüsteemi kiiresti porteerida väga mitmesugustele arvutitele. See kõik tingis ka programmeerimiskeele C kiire leviku. Varem loodi suurem osa operatsioonisüsteemidest vastavalt iga arvuti eriomadustele. Seetõttu olid ka konkreetse arvuti jaoks loodud programmid kasutatavad vaid sellel arvutil või paremal juhul antud firma arvutitel. Arvutiehitusest sõltumatu ope-

ratsioonisüsteem (UNIX) lõi nagu omamoodi vahelüli arvuti ja programmide vahel ja võimaldas luua programme, mis töötasid kõikidel arvutitel, milles see operatsioonisüsteem oli installeeritud. Kuna aga programmid sageli kasutasid masinate eriomadusi (töökiiruse tõstmiseks), siis oli vaja arvutiehitusest sõltumatut programmeerimiskeelt. Selle tühiku täitiski programmeerimiskeel C.
C on vägagi paindlik programmeerimiskeel. Osa programmistidest arvavad, et C on tegelikult vaid natuke mugavam assembler. Programmeerimiskeeles C on võimalik kodeerida mitmeid probleeme, mis ei ole teistes programmeerimiskeeltes ilma assembleri abita lahendatavad. Selle näiteks võiks tuua eraldi bittide mõjutamise ja katkestuste (*interrupts*) kasutamise. Nii saab kodeerida masinalähedasi probleeme ja siiski kindlustada programmi hea porteeritavuse (implementeerimise teistsuguse ehitusega arvutil ainult uuesti transleerimise abil).
See paindlikkus tekitab aga ka mitmeid probleeme. Programmeerimiskeel C lubab kodeerida väga keerukaid probleeme lühikesel kujul, kuid seejuures tuleb täpselt teada, mida tehakse. Just see paindlikkus takistab translaatoril mitmete võimalike vigade äratundmist. Seega peab programmeerija ise teadma, kas koostatud programm sellisel kujul ka midagi mõistlikku teeb.
C keel on ka üsna lühike oma sõnastuses. See võimaldab kirjutada keeruka programmi kiiresti ja vähese vaevaga. Teiset küljest on lühikeste funktsiooninimede meelespidamine raskem, ja unustada ei tohi ka seda, et see, mis kiiresti paberile visatud (arvutisse toksitud), on sageli ka üsna vigane.
Nagu juba öeldud, võimaldab programmeerimiskeel C mitmeid probleeme üsna arvutilähedaselt kodeerida. Sedamoodi loodud masinkood on ka sageli kiirem kui teistes keeltes kodeeritud programm. Teisest küljest on programmi töökiiruse tõstmiseks sageli olulisem valida sobivam algoritm (lahendusviis), kui otsida parem translaator.
Nii nagu teistegi keelte puhul, ilmus ka peale programmeerimiskeele C loomist hulganiselt C erivariante. Iga firma soovis realiseerida mingeid erisoove ja lisavõimaluste implementeerimise kaudu oma translaatorit paremini turustada. See aga ei ole kasutajale enamasti sugugi vajalik. Hulga tähtsam on, et loodud programmi õnnestuks transleerida ka teiste C keele translaatorite abil. Kui see ei ole võimalik, siis ei saa seda programmi ka teistsuguse ehitusega arvutitele porteerida. Seepärast loodi C keele standard ANSI (American National Standardisation Institute) X3J11. Peale seda on ilmunud mitmeid teisigi standardeid, mis lisavad loodule uusi võimalusi. Nimetatud standard määrab kindlaks C keele programmi põhiehituse, operaatorid, võtmesõnad ja palju muud. Sellest standardist kinni pidades saab garanteerida, et enamik translaatoreist saavad selle programmiga hakkama.
Selles raamatus käsitletakse ka mõningaid Borland C/C++ translaatori eriomadusi, ilma milleta ei saaks osa teemasid käsitleda või mis on muul viisil programmistile vajalikud.

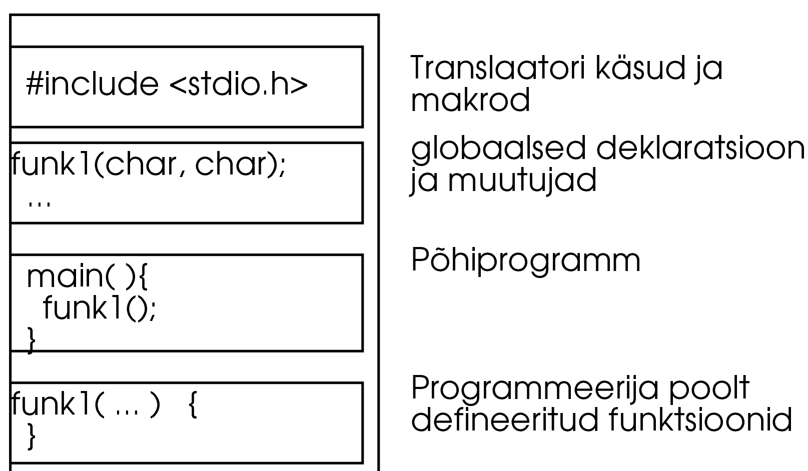
C - keele programmi ehitusest

Programmeerimiskeel C on protseduraalse iseloomuga, s.o. enamus tööst tehakse funktsioonides ja loodud programm transleeritakse otse masinkoodi. Peale protseduraalsete programmeerimiskeelte on olemas ka funktsionaalsed (*Lisp*), loogilised (*Prolog*) ja objektorienteeritud (*Smalltalk*) programmeerimiskeeled. Olgu juba ette ära öeldud, et C++ ei ole puhas objektorienteeritud programmeerimiskeel, vaid hübriidkeel, s.o. midagi protseduraalsete ja objektorienteeritud keelte vahepealset.
Funktsioonid on programmi osad, mis täidavad mingit eriülesannet. Funktsioon omab nime, mille abil teda saab välja kutsuda. Programmeerimiskeele C funktsioonid vastavad näiteks keele BASIC alamprogrammidele (SUBROUTINE). Funktsiooni juurde kuuluv programmilõik on ümbritsetud looksulgudega. Kui teie programm peab sageli täitma sarnaseid ülesandeid, siis on kasulik see ülesanne kodeerida omaette funktsioonina. Nii loodud programm on lühem, kuna sama ülesande mitmekordsel täitmisel ei pea vastavat programmilõiku mitu korda programmi sisestama, vaid selle asemel kasutatakse funktsiooni. Peale selle on niimoodi programmi lihtsam koostada. Suurem probleem jaotatakse omaette ülesanneteks, mis igaüks moodustavad eraldi funktsiooni.
Funktsioonid võivad omada parameetreid, s.o. väärtusi, mis selle tööd mõjutavad ja mis antakse funktsioonile üle seda välja kutsuva funktsiooni poolt. Funktsioon võib omada ka väärtust - nn. funktsiooni oma väärtust. See väärtus on sageli funktsiooni poolt lahendatava ülesande tulemus ja ta antakse peale funktsiooni töö lõppu funktsiooni välja kutsunud funktsioonile tagasi. Funktsioon näeb seega välja järgmiselt:

```
int  funktsioon1(int parameeter1, int parameeter2)
{
  return parameeter1 + parameeter2;
}
```

Antud näitefunktsioon on tüübist *int* (täisarv).
Iga programm peab sisaldama funktsiooni, mille nimi on *main()*. Programmi töö alguses laadib operatsioonisüsteem selle programmi mällu ja kutsub välja programmis sisalduva funktsiooni *main()*. Funktsioon *main()* võib omakorda vajaliku töö täitmiseks kutsuda välja teisi funktsioone .
Translaatori komplektis on teeke enamkasutatavate funktsioonidega. Programmeerimiskeel C ei sisalda mingeid käske teksti väljastamiseks ekraanile, failide avamiseks jms. Kõigi nende ülesannete täitmiseks on olemas vastavad funktsioonid. Need funktsioonid on juba masinkoodi transleeritud ja teekidesse kogutud. Selleks, et teie programm neid kasutada saaks, tuleb vastavad teegid transleeritud programmile lisada. Seda tööd teeb *linker*.
Nagu juba öeldud, on kõik funktsioonid võrdväärsed. Kui nüüd translaator leiab programmitekstis mingi funktsiooni nime, siis otsib ta vastavast failist selle funktsiooni definitsiooni. Kui translaator seda failist ei leia, siis otsib ta

kasutatud funktsiooni programmiga lingitavatest teekidest. Kui ka seal vastavat funktsiooni ei leidu, siis väljastatakse veateade. Translaator üritab ka kontrollida, kas funktsioonile anti üle õige arv õiget tüüpi parameetrid. Selleks aga peab translaator teadma, millist tüüpi parameetreid antud funktsioon vajab. Nimetatud andmed kogutakse sageli päisefailidesse.
Päisefailid .H sisaldavad ka muid vajalikke andmeid. Neid faile ei transleerita, vaid kasutatakse üksnes andmete üleandmiseks. Enamasti sisestatakse päisefailid programmi translaatori käsu *#include* abil. Joonisel 1. näete programmeerimiskeeles C koostatud programmi struktuuri.



Joonis 1: Programmeerimiskeele C programmi struktuur

Jooniselt on näha, et programmi alguses on translaatori käsk *#include <stdio.h>*. Selle käsuga teatatakse translaatorile, et ta peab faili enne transleerimist lisama päisefaili nimega STDIO.H. Nimetatud päisefail sisaldab paljude vajalike funktsioonide deklaratsioone, nende poolt kasutatavaid konstante jms. Translaatori teine levinud käsk on *#define*. Selle käsuga defineeritakse sümboolseid konstante ja makrosid.
Sellele järgneb rida *funk1(char, char);*. See on funktsiooni deklaratsioon. Selline rida teatab translaatorile, et kusagil selles failis on defineeritud sellenimeline funktsioon ja et sellel on kaks parameetrit tüübist *char*. Kuna rea lõpus on kohe semikoolon ja mingeid looksulge ei järgne, siis teab translaator, et see ei ole veel vastava funktsiooni definitioon. Leidnud nüüd selle funktsiooni nime funktsioonist *main(),* saab translaator ka kohe kontrollida, kas üleantud parameetrid olid õiget tüüpi.
Selle koha peal defineeritakse sageli ka mõned uued tüübid ja muutujad. Muutujad on nimelised mälupesad, millesse võite salvestada oma töö tulemused. Muutujad peavad alati omama mingit tüüpi ja translaator kontrollib, kas te omistate neile väärtusi sobivast tüübist. Kui te defineerite nimetatud kohal failis muutuja, siis on too muutuja globaalne, s.o. te saate tema väärtust lugeda ja ka muuta igast funktsioonist. Kui te aga deklareerite muutuja mingis funkt-

sioonis, siis kehtib nn. "nähtavuse" printsiip. Seda muutujat saate nüüd kasutada ainult selles funktsioonis.
Programmeerimiskeeles C on kõik funktsioonid võrdväärsed. Erinevalt programmeerimiskeelest PASCAL ei saa te siin defineerida mingi funktsiooni sees omakorda uusi funktsioone, mis oleksid ka kättesaadavad ainult sellele funktsioonile. Kordloodud funktsioonid on C - keeles kasutatavad kõigi teiste funktsioonide poolt. Ainult funktsiooni *main()* ei tohi enam teiste funktsioonide poolt välja kutsuda.
Näites INTEGRAL.C on toodud lihtsa C programmi tekst.

INTEGRAL.C

```
1./*****************************************************/
2./***                                              ***/
3./***   Integral.c                                 ***/
4./***                                              ***/
5./***   See on lihtne programm, mis iseloomustab    ***/
6./***   C - keele programmide põhistruktuuri. Programm ***/
7./***   arvutab  määratud integraali kasutaja       ***/
8./***   poolt sisestatud rajades.                   ***/
9./*****************************************************/

10./*----------------< Päisefailid >---------------------*/

11.#include <stdio.h>    /* seda päisefaili on vaja
12.                          mitmete teegis defineeritud
13.                          funktsioonide kasutamiseks    */
14.#include <math.h>   /* selles päisefailis on defineeritud
15.                            matemaatilised funktsioonid */
16.#include <ctype.h>    /* toupper() */
17.#define MINPIIR 0.0
18.#define MAXPIIR M_PI / 2.0
19.#define EPS 0.0001

20./*----------------< Globaalsed muutujad >--------------*/

21.typedef double   REAL;  /* esialgu kasutame tüüpi double.
22.                          hiljem ehk ka long double, kui
23.                          soovime suuremat täpsust */
24./*----------------< Funktsioonide prototüübid >--------*/
25.REAL IntFunc(REAL);     /* integreeritava funktsiooni
                            prototüüp */
                         /* interpolatsiooni funktsioon */
26.REAL Ristkuelik(REAL, REAL, unsigned long);

27./*-----------------------------------------------------*/
28./*---                                              ---*/
29./*---   IntFunc()                                  ---*/
30./*---                                              ---*/
31./*---   arvutab integreeritava funktsiooni väärtuse ---*/
32./*---   kohal x.                                   ---*/
33./*-----------------------------------------------------*/
34.   REAL IntFunc(REAL x)
35.   {
36.   return 5.0 * exp(2.0 * x) / (exp(M_PI) - 2.0) * cos(x);
37.   }   /*  IntFunc() */


38./*-----------------------------------------------------*/
39./*---                                              ---*/
40./*---   Ristkuelik()                               ---*/
```

```
41./*---                                              ---*/
42./*---  arvutab funktsiooni IntFunc määratud inte-   ---*/
43./*---  graali soovitud rajades ja nõutud täpsusega. ---*/
44./*---  Nagu funktsiooni nimest näha, ei ole C-keeles---*/
45./*---  funktsioonide ja muutujate nimedes lubatud   ---*/
46./*---  kasutada täppidega tähti jms.                ---*/
47./*--------------------------------------------------*/
48.REAL Ristkuelik(REAL MinRaja, REAL MaxRaja, unsigned long n)
49.   {
50.    REAL    h, sum;
51.    int     i;

52.    sum = 0.0;
53.    h = (MaxRaja - MinRaja) / n;
54.    for(i = 0; i < n; i++) {
55.      sum += IntFunc(MinRaja + i * h);
56.    }
57.    return sum * h;
58.   }   /* Ristkuelik */

59./*==================================================*/
60./*===                                            ===*/
61./*===   main()                                   ===*/
62./*===                                            ===*/
63./*=== See funktsioon on programmi peamine funkt-    ===*/
64./*=== sioon. Igas programmis peab olema funktsioon  ===*/
65./*=== main(). Programmi algul stardib operatsiooni- ===*/
66./*=== süsteem  selle funktsiooni ja selle lõppedes  ===*/
67./*=== on ka programm lõppenud.                    ===*/
68./*==================================================*/
69.   int main( void )
70.   {
71.    char       Vastus = 'n';
72.    unsigned long Num;
73.    REAL      MinRaja, MaxRaja, Tulemus, Epsilon, Temp;

74.    /* teatame kõigepealt, mida see programm teeb */
75.    puts("See programm arvutab funktsiooni: ");
76.    puts("5 * e**2x / (e**Pi - 2) * cos(x)) ");
77.    puts("määratud integraali teie poolt sisestatud rajades ja
   täpsusega.");
78.    puts("Programmi lõpetamiseks vajutage klahvile 'j'");
79.    /* initsialiseerime vajalikud muutujad */
80.    MinRaja = MINRAJA;
81.    Tulemus = Temp = 0.0;
82.    MaxRaja = MAXRAJA;
83.    Epsilon = EPS;
84.    Num = 1;
85.    /* alustame arvutamisega */
86.    do {
87.   /* Loeme sisse uued rajad ja täpsuse */
88.      puts("Sisestage uued rajad ja täpsus");
89.      printf("Alumine raja [%f]: ", MinRaja);
90.      scanf("%f", &MinRaja);
91.      printf("Ülemine rada [%f]: ", MaxRaja);
92.      scanf("%f", &MaxRaja);
93.      printf("Täpsus [%f]: ", Epsilon);

94.      scanf("%f", &Epsilon);
95.     /* interpoleerime seni, kuni soovitud täpsus on saavutatud */
96.      do {
97.        Tulemus = Temp;
98.        Temp = Ristkuelik(MinRaja, MaxRaja, Num);
99.        Num = Num * 2;
100.     } while(Epsilon < fabs(Tulemus - Temp));
101.     /* trükime tulemuse ja anname kasutajale võimaluse
102.        arvutusi uute väärtustega korrata */
103.     printf("\n\nIntegral = %f", Temp);
104.     puts("\nSoovite te uuesti arvutada (J/E)?");
105.     fflush(stdin);
```

```
106.     scanf("%1c", &Vastus);
107.   } while (toupper(Vastus) == 'J');
108.   return 0;
109. }
```

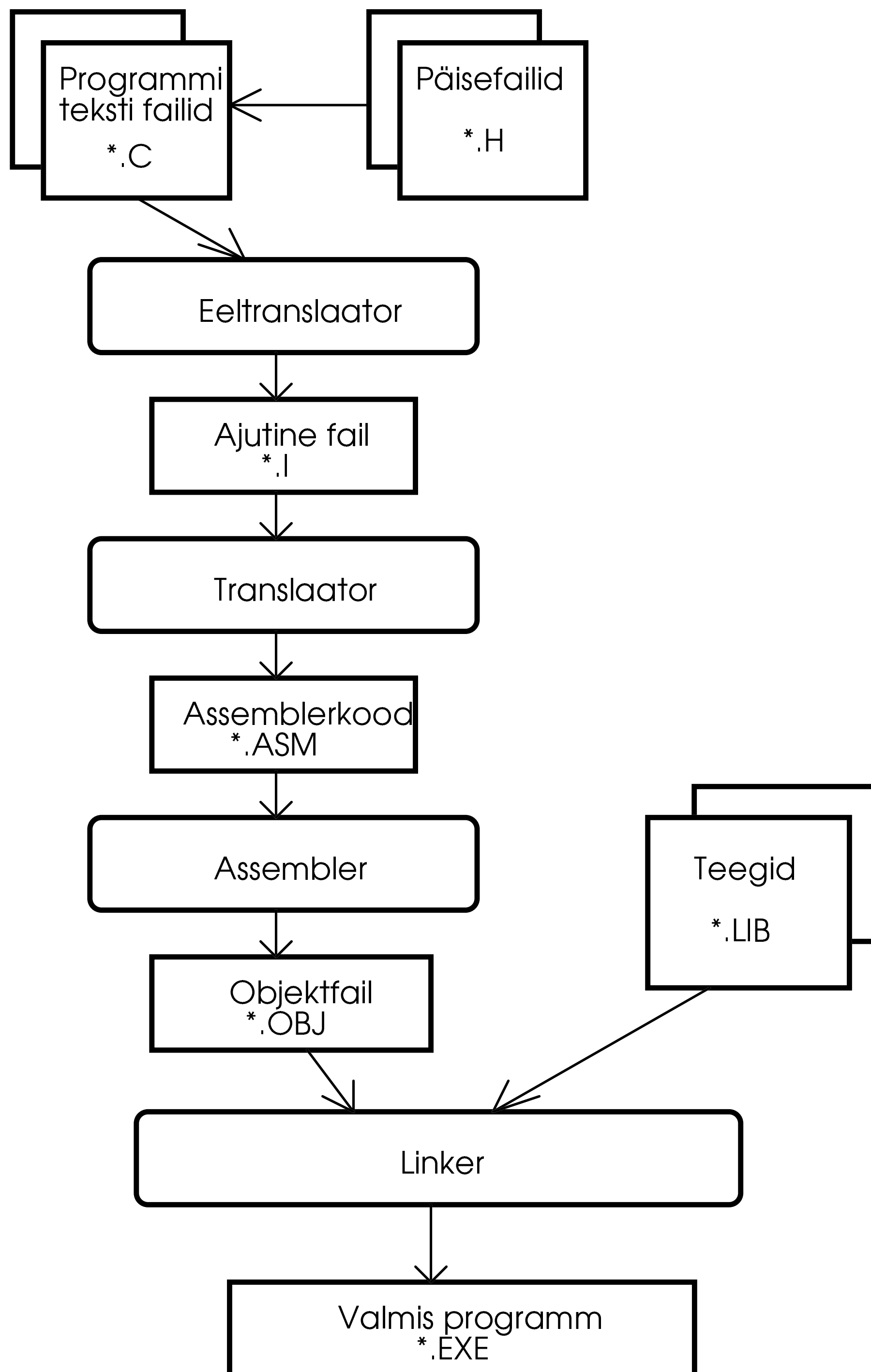
Programmeerimiskeel C kasutab kommentaaride jaoks sümboleid /* ja */. Translaator ignoreerib kõike, mis on piiratud nende sümbolitega. Programmeerimiskeel C++ kasutab kommentaaride jaoks veel sümbolit //. See sümbol määrab kommentaari, mis ulatub rea lõpuni. Sellist kommentaari on lihtsam sisestada, kuna tuleb määrata vaid kommentaari algus, kuid ta on ka mitmeti piiratud. Sageli ostub kasulikuks ümbritseda mingi programmilõik kommentaarisümbolitega, et seda sel kombel mitte transleerida (välja kommenteerida). Nii saab programmi tööd jätkata ilma selle programmilõiguta ja nimetatud lõiku ei ole vaja kustutada. Sageli on vaja mingi osa ühest reast välja kommenteerida. Sel juhul tuleb kasutada sümboleid /* ja */. Näiteprogrammis INTEGRAL.C kasutatakse kommentaare mitmete programmi osade tähenduse selgitamiseks ja erinevate osade alguse ja lõpu märkimiseks. Kommentaarid muudavad programmi teksti paremini loetavaks. Read 1 kuni 9 on kommentaarid ja sisaldavad andmeid selle kohta, mida antud programm teeb ja demonstreerib.

Read 10 kuni 19 sisaldavad translaatori käske. Selles osas lisatakse käsu *#include* abil kolm päisefaili, mis sisaldavad programmis kasutatud standardsete funktsioonide definitsioone. Ilma nende päisefailidete ei oskaks translaator programmi õigesti luua ega teaks, milliseid teeke on vaja. Enne transleerimist kasutab translaator käskude täideviimiseks eeltranslaatorit (precompiler). Käsu *#include* puhul lisab eeltranslaator kogu vastava päisefaili sisu määratud kohta programmi tekstis. Selliselt muudetud programmi tekst salvestatakse ajutisse faili, mis antakse nüüd üle tegelikule translaatorile.

Peale selle defineeritakse siin kolm sümboolset konstanti. Käsu *#define* abil saab defineerida makrosid ja sümboolseid konstante. Sümboolne konstant vastab mingile konstantsele väärtusele. Eeltranslaator asendab programmi tekstis vastavate konstantide nimed nagu näiteks MINRAJA, tema väärtusega, mis siin on 0.0. Sümboolsed konstandid muudavad teie programmi teksti paremini loetavaks. Nii on näiteks konstandi nimi MAKSUPROTSENT palju väljendusrikkam kui arv 0.35. Peale selle saab niiviisi konstante defineerida ühes kohas, kus neid on lihtsam muuta. Kui nimetatud protsent muutub, on vaja ainult muuta vastavat definitsiooni. Vastasel juhul peaksite otsima üles kõik kohad oma programmi tekstis, kus te olete seda konstanti kasutanud ja ta seal asendama. Suurema programmi puhul on see tülikas ja võib kergesti vigu tekitada. Lihtsam on lasta asendused teha translaatoril.

Read 20 kuni 23 sisaldavad globaalseid muutujaid ja definitsioone. Toodud programm ei vaja mingeid globaalseid muutujaid. Mõned üksikud siiski defineeritakse funktsioonis *main()*, kuid need on tegelikult funktsiooni *main()* kohalikud muutujad. Globaalsete muutujate kasutamine on sageli vigade põhjustajaks, kuna nende väärtusi tohib muuta iga funktsioon ja seega võivad funktsioonid üksteise tööd mõjutada. Muutujate asemel defineeritakse siin uus andmetüüp - REAL.

Read 24 kuni 26 sisaldavad funktsioonide prototüüpe ehk deklaratsioone. Funktsiooni deklaratsioon võib asuda suvalisel kohal programmi tekstis, kuid ta peab asuma enne vastava funktsiooni definitsiooni ja enne iga funktsiooni, mis seda funktsiooni kasutab. Seega on soovitav lisada funktsioonide definitsioonid faili algusesse. Samal põhjusel tuleks ka päisefailid sisestada faili algusesse, kuna funktsioonide definitsioone on ka päisefailides.
Funktsiooni deklaratsioon näeb välja nagu funktsioon, kuid tal puudub kood ja ta lõpeb kohe peale parameetrite loetelu semikooloniga. Ka parameetrite nimede lisamine ei ole vajalik, ainult parameetri tüüp on tähtis. Selles programmis ei ole need deklaratsioonid küll vajalikud, kuna funktsioonide definitsioonid annavad translaatorile samasugused andmed. Funktsioonide definitsioonid on aga siin sisestatud just sellises järjekorras, et definitsioon paikneb enne funktsioone, mis seda funktsiooni kasutavad. Keerukamas programmis on selle üle raskem arvet pidada ja seepärast on soovitav luua iga funktsiooni jaoks deklaratsioon programmi algul või sobivas isiklikus päisefailis. Muid programmilõike käsitleme järgnevates peatükkides.
Kui olete programmi tekstifaili (INTEGRAL.C) sisestanud, võite programmi transleerimiseks kasutada suvalist C translaatorit. Joonisel 2 näete C - keele programmi transleerimist.



Joonis 2: Programmeerimiskeele C programmi transleerimine

Programmi transleerimiseks võib kasutada ainult C - keele translaatorit, näiteks:

```
BCC   [<valikud>] INTEGRAL.C
```

Sel juhul kutsub translaator (BCC.EXE) kõigepealt välja eeltranslaatori (CPP.EXE), mis sisestab teksti vajalikud päisefailid ja asendab sümboolsete konstantide makrode nimed vastavate väärtustega. Tulemus salvestatakse ajutisse faili. Nüüd transleerib BCC.EXE faili assemblerkoodi, seejärel

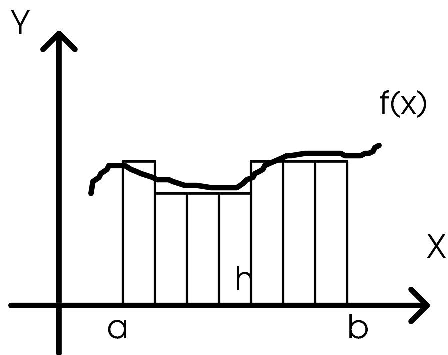
masinkoodi ja lingib teekidega. Tulemuseks on valmis programm - INTEGRAL.EXE.
Kui te soovite näiteks oma makrode kontrollimiseks näha eeltranslaatori töö tulemust, siis kasutage käsureal enne faili nime valikut **-i**. Sel puhul ei kustuta translaator loodud ajutist faili peale töö lõppu. Sageli kasutatakse ka valikut **-c**, mis sunnib translaatorit oma tööd peale objektfaili loomist lõpetama. Sel juhul tuleb ise objektfail teekidega linkida. Translaator kasutab valmis faili jaoks sama nime, mis oli *.C failil, lisades vaid uue nimelaiendi .EXE. Mitme faili transleerimisel valitakse programmi nimeks esimese faili nimi. See ei ole aga alati nii. Näiteks operatsioonisüsteemi UNIX C - translaatorid panevad loodud faili nimeks A.OUT. Kui te soovite anda valmis failile mingi teise nime, siis kasutage valikut **-o <uus nimi>**. Näiteks, nimetamaks valmis programmi INTR.EXE:

```
BCC   -o INTR           INTGRAL.C
```

Tähtis valik on ka **-v**, mis lisab objektfailile kodeeritud kujul andmeid muutujate jms. kohta ja võimaldab seega loodud programmist hiljem siluriga vigu otsida. Osa translaatoreid kasutab objektfaili andmete sisestamiseks siluri jaoks valiku **-v** asemel valikut **-g** .
Kui teie programm koosneb mitmest eraldi *.C ja *.H failist, siis on sageli sobiv kasutada valikut **-c**. Kui te nüüd muudate ainult üht faili, siis tuleb üksnes see fail uuesti transleerida ja teiste objektfailide ja teekidega uuesti linkida.
Programm INTEGRAL.EXE arvutab funktsiooni:

$$Y = \frac{5 * e^{2x}}{(e^{\pi} - 2) * \cos(x)}$$

määratud integraali rajades 0 kuni π/2. Määratud integraal on teatavasti võrdne funktsioonialuse pindalaga nimetatud rajades. Selle pindala arvutamiseks kasutatakse siin lihtsat ristkülikumeetodit, mispuhul kogu pindala jaotatakse elementaarseteks ristkülikuteks ja nende pindalad summeeritakse.



Joonis 3: Määratud integraali arvutamine ristkülikureegli järgi

Programmi tulemuseks peaks olema:

```
D:\SAMPLES\CSAMPLES\INTEGRAL>integral
See programm arvutab funktsiooni:
5 * e**2x / (e**Pi - 2) * cos(x))
määratud integraali teie poolt sisestatud rajades ja täpsusega.
Programmi lõpetamiseks vajutage klahvile 'j'
Sisestage uued rajad ja täpsus
Alumine raja [0.000000]: 0.0
Ülemine raja [1.570796]: 1.570796
Täpsus [0.000100]: 0.000001


Integraal = 1.000090
Soovite te uuesti arvutada (J/E)?
e

D:\SAMPLES\CSAMPLES\INTEGRAL>
```

Abiprogramm MAKE

Kui te kasutate mitut *.C faili, siis märkate varsti, et on üpris tülikas kutsuda translaator koos kõigi valikutega välja iga faili jaoks. Valikute sisestamisel võib tekkida vigu. Kuna te iga kord peale programmi muutmist peate selle testimiseks sisestama ühed ja samad käsud, siis on lihtsam luua käsufail *.BAT ja viia sinna kõik transleerimiseks vajalikud käsud. See aga tähendaks, et iga väikesegi muudatuse korral transleeritakse kõik *.C failid uuesti, mis aga ei ole hoopiski vajalik.
Nimetatud probleemi lahendamiseks kasutatakse programmi MAKE. Lihtsaim võimalus programmi MAKE kasutamiseks on sisestada käsk:

```
MAKE  INTEGRAL.C
```

Programm MAKE kutsub nüüd välja translaatori BCC.EXE. See käsk on seega võrdväärne käsuga:

```
BCC   INTEGRAL.C
```

Üldjuhul on aga vaja programmi MAKE töö juhtimiseks luua eraldi fail. Selle faili nimeks peaks olema MAKEFILE. Kui te soovite, võite anda nimetatud programmile mistahes teise nime, näiteks INTEGRAL.MAK, kuid siis peate kasutama valikut -f.

```
MAKE  -f INTEGRAL.MAK  [<tulemus>]
```

Programm MAKE üritab nüüd teha soovitud tulemuse saavutamiseks vajalikud tööd. Selleks sisaldab fail MAKEFILE (või INTEGRAL.MAK) reegleid, mis määravad, millised tulemused on üldse võimalikud, kuidas neid luua ja mil moel need üksteisest sõltuvad. Tulemused on tavaliselt translaatori poolt loodavate failide nimed, näiteks INTEGRAL.EXE või INTEGRAL.OBJ. Programm MAKE loob nüüd tulemused, mis veel ei eksisteeri või mis on vanemad kui nende osad. Kui näiteks tulemus PROG.EXE sõltub failidest FILE1.OBJ ja FILE2.OBJ ning te olete just muutnud faili FILE2.C, siis üritab programm MAKE luua kõigepealt faili FILE2.OBJ ja seejärel PROG.EXE. Faili FILE1.C aga ei transleerita, kuna seda ei ole muudetud ja vastav objektfail on seega kasutamiseks kõlbulik. Selgitamaks, milliseid faile on vaja transleerida, võrdleb programm MAKE nende salvestamisaegu. Nimetatud aegu võite vaadelda näiteks käsu DIR abil. Tulemused (*.OBJ, *.EXE jne.), mille osad on vanemad, tuleb uuesti luua. Enne mingi tulemuse loomist kontrollitakse ka selle osade kõlblikkust ja vajaduse korral luuakse enne osad.
Allpool on toodud lihtne fail programmi MAKE jaoks.

INTEGRAL.MAK

```
#Kommentaarid algavad kaksiktrelli sümboliga.
#Alates sellest sümbolist on kogu ülejäänud reaosa kommentaar
      #kommentaar võib alata suvaliselt kohalt reas.

#integral.exe sõltub failist integral.obj.,sellele
#järgnevad reeglid, mis määravad, kuidas luua faili integral.exe
#iga reegel algab taandrealt, UNIX-süsteemides täpselt ühe <TAB>-.
#sümboli kauguselt
integral.exe: integral.obj
      tlink /Lc:\borlandc\lib c0s.obj integral.obj, integral.exe, , ,

integral.obj: integral.c
      bcc -c -Id:\borlandc\include integral.c
```

Esimene tulemus on INTEGRAL.EXE, mis on oma komponentidest eraldatud kooloniga. Temale järgnevad reeglid selle faili loomiseks. Nagu näha, on selleks vaja vaid linkida vastav objektfail. Järgmine reegel aga määrab, kuidas luua fail INTEGRAL.OBJ.
Iga reegel peab algama taandrealt. DOSi süsteemis kasutatavate translaatorite puhul sobib taandreaks tavaliselt mõni tühik ja/või tabulaator. Operatsioonisüsteemis UNIX kasutatavad translaatorid aga nõuavad enamasti taandreana täpselt ühe tabulaatori. Seega on 8 tühiku kasutamine viga ja seejuures veel väga raskesti avastatav.
Iga reegel ja kommentaar peab mahtuma ühele reale. Kui see ei õnnestu, siis tuleb kas enne otse klahvile <RETURN> vajutamist sisestada sümbol \ (backslash) või kasutada makrosid.
Makrod tuleb defineerida samas failis enne nende kasutamist. Makro omab nime ja tähendust. Trasnleerides näiteks suurest hulgast *.C failidest koosnevat programmi, võiks defineerida järgmised makrod:

```
OBJS = C0S.OBJ FILE1.OBJ FILE2.OBJ FILE3.OBJ
SOURCES = FILE1.C FILE2.C FILE3.C
```

Nüüd võib neid kasutada translaatori ja linkeri käsureal:

```
tlink /Ld:\borlandc\lib §(OBJS), prog.exe, , ,
bcc -c -Id:\borlandc\include $(SOURCES)
```

Sama kehtib ka valikute *-Id:\borlandc\include* ja */Ld:\borlandc\lib* kohta. Need valikud määravad, millisest kataloogist translaator ja linker peavad otsima oma päisefaile ja teeke. Kui te aga määrate nimetatud kataloogid juba failis AUTOEXEC.BAT keskonnamuutujate *INCLUDE* ja *LIB* abil, siis on need valikud muidugi üleliigsed. Näiteks

```
...
SET INCLUDE=d:\borlandc\include; d:\borlandc\classlib\include
SET LIB=d:\borlandc\lib;d:\borlandc\classlib\lib;
...
```

Siin kasutatakse ka linkerit TLINK.EXE. Tavaliselt on linkeri nimeks LINK.EXE. Linkeri TLINK.EXE käsk peab omama järgmist süntaksi:

```
TLINK [<valikud>] <obj failid>, <exe fail>, [<map fail>], [<teegid>],
[<def fail>]
```

Linkeri põhilisi valikuid juba käsitlesime, ülejäänud ei leia nii sageli kasutamist. Võimalike valikute loetelu leiate translaatori käsiraamatust (Borland C puhul: "Tools & Utilities Guide").
Map-fail sisaldab andmeid loodud programmi segmentide aadresside kohta. Seda te tavaliselt ei vaja. *Def*-fail on vajalik vaid Windowsi programmide puhul.
Näidatud reeglid olid nn. otsesed reeglid (*explicit*), kuna nad kirjeldasid otseselt ühe kindla faili loomist. Programm MAKE tunneb aga ka kaudseid reegleid, mis kirjeldavad terve grupi sarnaste failide loomist. Selline reegel võiks välja näha järgmiselt:

```
.c.obj:
      BCC    -c   $.
```

See reegel määrab, et iga .OBJ faili saab luua vastavanimelisest .C failist, kui seda transleerida käsuga *BCC -c <faili nimi>.C*. Viimane sümbol **$.** määrab hetkelise failinime tüve. Kui hetkel uuritakse võimalusi faili FILE1.OBJ loomiseks, siis on selleks tüveks FILE1. Seega tähendab sümbol **$.** selles kohas FILE1.C.
Tulemused ei pea alati olema just failide nimed. Näiteks reegel:

```
clear:
      erase *.obj
      erase *.bak
```

loob koos sellele järgneva käsuga võimaluse puhastada antud kataloog peale korrektse programmi loomist kasututest objekt- ja reservfailidest.

```
      MAKE  -f INTEGRAL.MAK   clear
```

Programm MAKE omeb veel teisi ja palju keerukamaid võimalusi, kuid need ei ole sageli standardiseeritud ja seepäraste me neid siin ei vaatle.
Borland C/C++ omanikel on lihtsam kasutada Borlandi programmeerimiskeskkonda (IDE). Sel juhul tuleks algul valida menüüst *Project* käsk **Open...**, ning luua uus projektfail *.PRJ. Seepeale avaneb tööpiirkonna allosas uus aken, kuhu saab menüü *Project* käsu **Add...** abil lisada uusi faile. Projekti tuleb lisada kõik vajalikud *.C failid ja omaloodud teegid, kui programm neid vajab. Päisefaile ei tohi aga projekti lisada, kuna nad sisestatakse hoopiski *#include* käsu abil. Te ei saa projektiakent otseselt editeerida, vaid võite ainult menüü *Project* käskude abil sinna faile lisada ja sealt eemaldada. Projektiakna mõtteks on näidata andmeid iga faili kohta. Projektiaken sisaldab iga faili kohta nime, kataloogi, ridade arvu, teksti ja andmete suuruse baitides. Kui te nüüd kasutate mitmesuguste valikute tegemiseks menüü *Options* käske, siis salvestatakse need valikud projektifailis. Projektifail ei kujuta endast ASCII faili, vaid on fail

kahendkoodides, mida te võite abiprogrammi PRJ2MAK.EXE abil konverteerida vastavasse .MAK-faili.

Enamkasutatavad funktsioonid

Paljud C - keele programmid kasutavad päisefaili STDIO.H. See fail sisaldab paljusid enamkasutatavaid funktsioone andmete väljastamiseks ekraanile ja faili ning andmete lugemiseks klaviatuurilt. Nende hulka kuuluvad: *printf()*, *scanf()*, *puts()* ja *gets()*. Kuna enamus teie poolt loodud programme tõenäoliselt peab mingeid parameetreid kasutajalt nõudma või vähemalt väljastama arvutuste tulemused ekraanile, siis käsitleme siin põhilisi sisestus- ja väljastusfunktsioone enne keerukamaid teemasid.
Funktsioon *puts()* (Put String) väljastab ekraanile üheainsa rea ja nihutab kursori kohe järgmisele reale. Kui kursor on jõudnud ekraani viimasele reale, siis nihutatakse kõik read ühe rea võrra ülespoole ja ekraani allosas tekib uus tühi rida. Seejuures peab väljastatav string olema lõpetatud nulliga, nagu C - keeles tavaline.

```
int puts(const char *string);
```

Kui vigu ei tekkinud, on funktsiooni väärtus mingi positiivne arv, vastasel juhul on funktsiooni väärtus võrdne sümboolse konstandiga *EOF*. Funktsioon *puts()* väljastab andmeid väljundvoole (*stream*) *stdout* ja seega mitte tingimata ekraanile, seda käsitleme peatükis Vood.
Täielikuks vastandiks on funktsioon *gets()*, mis loeb sisse ühe rea klaviatuurilt.

```
char * gets(char *buffer);
```

Funktsioon loeb klaviatuurilt niikaua, kuni rida lõpetatakse klahvile <RETURN> vajutamisega. Seejuures väljastatakse loetud tähed ka ekraanile. Kasutaja võib klahvi <Backspace> abil sisestatud sümboleid kustutada ja vajaduse korral uusi sisestada enne klahvile <RETURN> vajutamist. Sisseloetud tähed kopeeritakse parameetri *buffer* poolt osutatud puhvrisse ja viimase tähe järele lisatakse lõpumärgiks null. Funktsiooni väärtuseks on viit puhvrile või vea puhul sümboolne konstant NULL.
Üheainsa tähe (sümboli) väljastamiseks ekraanile või selle sisselugemiseks võite ksutada makrosid *putchar()* ja *getchar()*.

```
int putchar(int c);
int getchar( void );
```

Nimetatud makrod kasutavad tegelikult funktsioone *putc()* ja *getc()*, mis väljastavad või loevad sümboleid sisend - või väljundvoogudest.
Nii võite midagi ekraanile väljastada või sealt sisse lugeda, kuid need on vaid sümbolid. Teie programm aga arvutab kuueteistkümnendsüsteemis ja seega on

need vaja arvutustulemuste väljastamiseks enne konverteerida ASCII sümboliteks (tähed ja numbrid). Sama kehtib ka klaviatuurilt sisseloetud numbrite kohta. Ka need on esialgu vaid sümbolid.
Sümbolite konverteerimiseks on kaks võimalust:

- kasutada funktsioone printf() ja scanf(), mis konverteerivad sümboleid otse väljastamise (või lugemise) käigus ka soovitud formaati.
- kasutada sümbolite konverteerimiseks spetsiaalseid funktsioone *atoi()*, *atof()* jne.

Funktsioonid printf() ja scanf()

Funktsioon *printf()* väljastab soovitud andmed ekraanile (tegelikult väljundvoole - *stdout*), konverteerides need vajaduse korral soovitud formaati.

```
int printf(const char *format [, argument] ...);
```

See funktsioon ei oma kindlat arvu parameetreid. Vajalik on ainult esimene parameeter - *format*. Sellele võib aga järgneda suvaline arv parameetreid vastavalt sellele, kui palju just on vaja. Muutuva parameetrite arvuga funktsioonide kasutamine on üks C - keele plusse, millega näiteks programmeerimiskeel PASCAL ei saa hakkama. Selliseid funktsioone on küll raskem programmeerida, kuid neid on hulga mugavam kasutada.
Lihtsaim võimalus kasutada funktsiooni *printf()* on ühe stringi väljastamine:

```
printf("Hello world!");
```

Funktsioon *printf()* ei nihuta kursorit peale stringi väljastamist automaatselt järgmisele reale. Järgmine *printf()* jätkab samast kohast, kus eelmine lõpetas. Kui te soovite nihutada kursori uuele reale, siis peate kasutama spetsiaalseid sümboleid **\n** (escape sequences). Funktsioon käsitleb sümbolit \ teistest sümbolitest erinevalt. Nimetatud sümbol märgib mingi käsu algust, mille täpse tähenduse määrab sellele järgnev sümbol. Programmeerimiskeel C tunneb järgmisi käsusümboleid:

Sümbolid	Tähendus
\n	Uus rida
\t	Horisontaalne tabulaator
\a	Helisignaal (bell)
\b	Kustuta eelmine sümbol (<Backspace>)
\f	Uus lehekülg
\v	Vertikaalne tabulaator
\r	Reavahetus
\?	Küsimärk
\'	Ülakoma. Ilma sümbolita \ tähendaks ülakoma stringi lõppu.
\"	Jutumärk. Ilma sümbolita \ tähendaks jutumärk stringi lõppu.
\\	Üksainus sümbol \. Ilma eelneva sümbolita \ tähendaks see käsu algust.
\ddd	ASCII sümbol. Seejuures tähendab *ddd* selle sümboli numbrit kaheksandsüsteemis.
\xddd	ASCII sümbol. Seejuures tähendab *ddd* selle sümboli numbrit kuueteistkümnendsüsteemis.

Tabel 1: Käsusümbolid (escape sequences)

Nagu näete, tuleb string ümbritseda jutumärkidega ja selle sees ei tohi esineda jutumärke. Kui see on aga vajalik, siis tuleb enne jutumärki sisestada sümbol \.
Funktsiooni *printf()* esimene parameeter - *format* - võib sisaldada formaadimääranguid, mis omavad järgmist kuju:

%[<lipud>] [<laius>] [**.**<täpsus>] [{**F** | **N** | **h** | **l** | **L**}] tüüp

Kui funktsioon *printf()* leiab väljastamisel mingi formaadimäärangu, siis valib ta parameetrile *format* järgneva esimese argumendi, konverteerib ta vastavalt leitud formaadimäärangule ja väljastab antud kohas ekraanile. Järgmise formaadimäärangu asemel väljastatakse järgmine parameeter jne. Kui argumente on rohkem kui formaadimääranguid, siis liigseid parameetreid lihtsalt ignoreeritakse. Kui aga formaadimääranguid on rohkem kui parameetreid, siis tekib andmete väljastamisel viga. Selline viga võib tingida programmi töö enneaegse lõpetamise või süsteemi kokkuvarisemise.
Formaadimäärang algab alati sümboliga %, millele järgneb vähemalt soovitavat andmetüüpi määrav sümbol. Näiteks:

```
int         palk = 1000;
char        nimi[20] = "Jaanus";
...
printf("Eelmises kuus teenis %s \t %d  EEK\n", nimi, palk);
```

Tulemuseks on:

```
Eelmises kuus teenis Jaanus         1000 EEK
```

Programmeerimiskeel C tunneb järgmisi tüübimääranguid:

Tüüp	Tähendus
i	Positiivne või negatiivne täisarv
d	Positiivne või negatiivne täisarv
u	Positiivne täisarv
x	Arv kuueteistkümnendsüsteemis (0-9 ja a-f)
X	Arv kuueteistkümnendsüsteemis (0-9 ja A-F)
o	Arv kaheksandsüsteemis
c	Täht või sümbol
s	String (tähtede jada), mille lõpus on null.
f	Ühekordse täpsusega kümnendmurd stiilis: [-]dddd.dddd. Koma asemel kasutatakse punkti (ameerika stiil).
e	Kahekordse täpsusega kümnendmurd stiilis: [-]d.dddd e [aste]. Koma asemel kasutatakse punkti (ameerika stiil). Näiteks -1.0056e6 s.o. -1.0056 * 1000000.
E	Sama mis eelmine, ainult et sümboli **e** asemel kasutatakse enne astet sümbolit **E.**.
g	Kahekordse täpsusega kümnendmurd stiilis e või E vastavalt sellele, kumb on lühem. Stiili e kasutatakse vaid siis, kui aste on väiksem kui -4 või suurem või võrdne formaadimäärangus toodud täpsusest. Liigseid nulle ei trükita ja koma trükitakse vaid siis, kui murdosa ei ole null.
G	Sama mis eelmine, ainult et sümboli **g** asemel kasutatakse sümbolit **G**.
n	Argument osutab täisarvule (muutuja), millesse salvestatakse hetkel väljastatud sümbolite arv.
p	Argument on pikk viit (mingile muutujale vms.). Ekraanile trükitakse selle viida aadress stiilis xxxx:yyyy. Seejuures tähendab xxxx viida segmenti ja yyyy tema aadressi segmendis. Mõlemad aadressi pooled väljastatakse kuueteistkümnendnumbritena. Määrang %hp tähendab lühikest viita ja seega trükitakse vaid aadressi teine pool.

Tabel: Tüübimäärangud

Tüübimäärangule võib eelneda üks järgmistest tähtedest, mis täpsustavad argumendi tüüpi:

- F - pikk viit
- N - lühike viit
- h - lühike täisarv
- l - pikk täisarv
- L - kahekordse täpsusega kümnendmurd

Otse peale sümbolit **%** võib mitmesuguste lippudega määrata konverteeritud numbrite formateerimist ekraanil. Järgnevas tabelis näete võimalikke lippude tähendusi. Te võite neist lippudest määrata ühe või mitu ja suvalises järjekorras.

Formaadi lipp	Tähendus
-	Joondab tulemuse vasakule ja lisab vajaduse korral paremale tühikuid. Kui seda lippu ei ole märgitud, siis joondatakse tulemus paremale ja lisatakse vajaduse korral vasakule tühikuid või nulle.
+	Nii positiivsed kui negatiivsed numbrid algavad vastava märgiga. Tavaliselt plussmärki ei väljastata.
tühik	Positiivne väärtus algab plussmärgi asemel tühikuga .
#	Takistab liigsete nullide eemaldamist. Näiteks tüübi määrangute e, E, f, g, ja G puhul ei eemaldata lõpust liigseid nulle. Määrangute x ja X puhul lisatakse arvu ette null.

Tabel 2: Formaadi lipud

Formaadi lippudele võib järgneda mingi positiivne arv - laius - mis määrab väljastatava arvu kohtade hulga. Seda võimalust kasutatakse sageli murdarvude väljastamisel nende organiseerimiseks tabeli tulpadesse. Kui väljastatava arvu kohtade hulk on väiksem kui pakutud kohtade arv, siis lisatakse vasakule või paremale (vastavalt sellele, kas lippu "-" on kasutatud või mitte) tühikuid. Kui enne laiust sisestatakse null, siis kasutatakse tühikute asemel nulle. Kui väljastatav arv omab rohkem kohti kui soovitud, siis trükitakse need ikkagi välja. Kui laiuse asemel sisestada tärn "*", siis võetakse laiuse väärtus parameetrile *format* järgnevatest parameetritest. Selline parameeter peab olema täisarv ja asuma loetelus enne formateeritavat väärtust.
Laiusele võib järgneda *täpsuse* määrang. Täpsuse määrangu mõju sõltub argumendi tüübist. Täisarvude puhul (tüübid, i, d, u, o, x, X) määrab see väljastatavate kohtade minimaalse arvu. Kui arvus on vähem kohti, siis lisatakse vasakule nulle. Kui aga arvus on rohkem kohti, siis trükitakse kõik kohad välja. Tüüpide **f**, **e** ja **E** puhul määrab *täpsus* komale järgnevate kohtade arvu. Kui täpsust eraldi ei määrata, siis on täpsus 6 kohta. Viimane koht on ümardatud. Kui täpsus on 0 või täpsuse punktile ei järgne mingit numbrit, siis trükitakse välja vaid arvu täisarvulised kohad. Tüüpide **g**, **s** ja **G** puhul määrab täpsus kohtade üldarvu.
Andmete sisestamiseks kasutatakse funktsiooni *scanf().*

```
int scanf(const char *format [, argument] ...);
```

Funktsioon loeb sisendvoost (tavaliselt klaviatuurilt) sisestatud sümboleid ja võrdleb neid parameetri *format* poolt määratuga. Sobivad sümbolid sisendvoost konverteeritakse vajalikku tüüpi ja salvestatakse parameetriloetelus määratud muutujatesse. Parameetri *format* poolt osutatud string võib sisaldada tühikuid, tabulaatoreid, lihtsat teksti ja formaadimääranguid. Tühikud ja tabulaatorid võr-

reldavas stringis seatakse vastavusse suvalise arvu tühikute või tabulaatoritega sisendvoos. Lihtsa teksti puhul peab kehtima täpne vastavus. Kui funktsioon ei leia sisendvoost sümbolit, mis asub võrreldavas tekstis, siis ta lõpetab oma töö. Kui funktsioon aga leiab võrreldavas tekstis formaadimäärangu, siis üritab ta konverteerida sisendvoos leitud sümboleid vastavasse formaati ja salvestada parameetrite loetelus järgmisesse muutujasse. Kui see ei õnnestu, siis lõpetab funktsioon oma töö. Märkida tuleb, et funktsiooni *scanf()* parameetriteloetelu peab sisaldama muutujate aadresse, mitte väärtusi nagu funktsiooni *printf()* puhul.
Näiteks:

```
int   nNum;
float fNum1, fNum2;
char  nimi[20];
...
scanf("Numbreid %d, Nimi %s, Protsent %f teine %f", &nNum,
             nimi, &fNum1, &fNum2);
...
```

Kasutaja sisestab nüüd:

```
Numbreid       10, Nimi Jaan,  Protsent 0.1 Teine  0.2
```

Tulemuseks on:

```
nNum = 10   nimi = "Jaan"  fNum1 = 0.1  fNum2 = sama, mis ennegi
```

Funktsioon *scanf()* täitis praegusel juhul esimesed kolm muutujat õigete väärtustega, kuid lõpetas töö enne viimasele muutujale väärtuse omistamist, kuna kasutaja sisestas "Teine" ja mitte "teine". Programmeerimiskeel C eristab nimelt suur- ja väiketähti üksteisest.

Andmetüüpide konverteerimisfunktsioonid

Te võite ka lugeda kogu kasutaja poolt sisestatud rea mingisse puhvrisse (näiteks funktsiooniga *gets()*) ja seejärel üritada seda konverteerida soovitud formaati. Selleks võite kasutada funktsioone:

```
int atoi(const char *buffer);
long atol(const char *buffer);
float atof(const char *buffer);
long double _atold(const char *buffer);
```

Kõigi nende funktsioonide jaoks on vaja sisestada päisefail STDLIB.H translaatorikäsuga *#include*. Viimase kahe funktsiooni jaoks tuleb sellele lisaks sisestada ka veel päisefail MATH.H.
Need funktsioonid konverteerivad kuni sajast sümbolist koosneva stringi sellele vastavaks numbriks. Stringi lõpus peab olema null. Funktsioonid lõpetavad oma töö esimese sümboli juures, mida nad enam ei oska konverteerida, kusjuures

selleks võib olla null või mingi teine mittenumbriline sümbol. Kui funktsioon ei suuda konverteerida puhvri sisu soovitud formaati, siis on funktsiooni väärtuseks null. Funktsioonid *atof()* ja *_atold()* eeldavad, et murdarv on sisestatud järgmises formaadis:

[<tühikud>] [<märk>] <täisarvulised kohad> . <murdarvulised kohad> [{ e | E | d | D} [märk] <aste>]

Kõik nimetatud funktsioonid ignoreerivad üleliigseid tühikuid.
Näiteprogrammis INTEGRAL kasutati veel funktsiooni *fflush()*. See funktsioon on seotud andmete lugemisega voogudest. Funktsioon *fflush()* tühjendab vastava voo seal salvestatud sümbolitest. Selles programmis on ta vajalik, sundimaks funktsiooni *scanf()* lugema klaviatuurilt uusi sümboleid.

Muutujad

Muutujad on nimelised mälupesad, millesse võite salvestada mingeid väärtusi. Muutujad omavad nime, tüüpi, kehtivust, nähtavust ja salvestamise klassi.
Muutujate nimed võivad koosneda suvalisest tähtede, numbrite ja alljoonte "_" jadast. Muutuja nime esimene sümbol peab olema kas täht või allkriips. Samad reeglid kehtivad ka funktsioonide nimede kohta. Täppidega tähtede ä, Ä, ü, Ü, ö, Ö, õ ja Õ kasutamine nimedes ei ole lubatud. Üldiselt ei soovitata mingi muutuja või funktsiooni nime alustada alljoonega "_", kuna selliseid nimesid kasutavad sageli teekide funktsioonid ja nende poolt kasutatavad muutujad. Kui teie muutuja nimi langeb kokku mingi ettedefineeritud muutuja nimega, siis võib see tekitada raskesti avastatavaid vigu. Peale selle ei tohi te kasutada muutuja või funktsiooni nimena translaatori poolt kasutatavaid võtmesõnu. Borland C/C++ translaator tunneb järgmisi võtmesõnu:

```
asm         _asm         __asm        auto         break        case
cdecl       _cdecl       __cdecl      char         class        const
continue    _cs          __cs         default      delete       do
double      _ds          __ds         else         enum         _es
__es        _export      __export     extern       far          _far
__far       _fastcall    __fastcall   float        for          friend
goto        huge         _huge        __huge       if           inline
int         interrupt    _interrupt   interrupt    loadds       _loadds
long        near         _near        __near       new          operator
pascal      _pascal      __pascal     private      protected    public
register    return       _saveregs    __saveregs   _seg         __seg
short       signed       sizeof       _ss          __ss         static
struct      switch       template     this         typedef      union
unsigned    virtual      void         volatile     while
```

Muutujategi puhul tehakse vahet nende definitsiooni ja deklaratsiooni vahel. Muutujat defineerides määrate tema tüübi ja reserveerite talle antud blokis mälu. Seda tehes võite talle kohe omistada ka mingi väärtuse. Näiteks:

```
int         nNum = 15;
float       fNum1,
            fNum2;
```

Ülaltoodud näide defineerib ühe täisarvu ja kaks murdarvu ning omistab täisarvule väärtuse 15. Muutuja definitsioon sisaldab muutuja tüübi nime (int, *float*, jne.), millele järgneb muutuja nimi. Ühel real saab defineerida mitu muutujat, eraldades need üksteisest komadega. Definitsiooni lõpetab semikoolon. Muutujate nime ja tüübi vahel võib olla suvaline arv tühikuid, tabulaatoreid ja reavahetuse märke.
Muutuja deklaratsioon erineb definitsioonist vaid võtmesõna **extern** poolest.

```
extern int  nNum;
```

Selline deklaratsioon teatab translaatorile, et sellise nime ja tüübiga muutuja on juba defineeritud mingis teises failis. Deklaratsioone kasutatakse mitmest *.C failist koosneva programmi transleerimisel. Ilma sellise deklaratsioonita ei saaks seda muutujat teisest .C failist lugeda ega tema väärtust muuta. Deklaratsioonis ei tohi muutujale mingit väärtust omistada.
Kehtivus (*scope*) määrab, millistes programmi osades saab antud muutujat kasutada. Programmeerimiskeel C tunneb nelja liiki kehtivusi:

- Üldine (*file scope*) - Üldised muutujad defineeritakse väljaspool kõiki blokke ja funktsioone. Üldine muutuja defineeritakse tavaliselt faili alguses. Sellised muutujad on kehtivad kõikides blokkides ja funktsioonides antud faili (*.C) piires. Kui seda muutujat on vaja kasutada ka mõnes teises failis defineeritud funktsioonis, siis tuleb ta tolles failis uuesti defineerida, kasutades seejuures võtmesõna **extern**.
- Funktsioonisisene (*function scope*) - Funktsioonisisesed objektid on märgendid (labels), mida kasutatakse käsuga **goto**. Märgend defineeritakse kujul <nimi>: Märgid peavad olema funktsiooni piires erinevate nimedega.
- Funktsiooni prototüübisisene - Funktsiooni prototüübis tuleb määrata parameetrite tüübid ja samas võib neile anda ka nimed. Antud nimed peavad selle prototüübi piires olema üksteisest erinevad. Kuna nad kehtivad ainult prototüübi piires ja prototüüp ei tohi mingit koodi omada, siis on prototüübis mõttetu muutujatele nimesid omistada (ainult vast nende tähenduse märkimiseks, et programmi tekst oleks paremini loetav).
- Kohalik kehtivus (*block scope*) - Kohalikud muutujad defineeritakse bloki alguses ja nad on kehtivad kuni bloki lõpuni. Kui antud blokk sisaldab veel muidki alamblokke, siis on antud muutuja ka neis kehtiv.

Programmeerimiskeel C++ tunneb veel kehtivust klassi piires.
Nähtavus on tihedalt seotud kehtivusega ja sageli langeb sellega kokku, kuid vahel võivad nad ka erineda. Kui mingis blokis on defineeritud kohalik muutuja, siis on see muutuja kehtiv ka tolle bloki alamblokkides. Kui aga üks neist alamblokkidest defineerib samanimelise muutuja, siis on ülemise bloki muutuja küll kehtiv, kuid mitte nähtav. Näiteks:

```
...
int func1(int a, char c){ /* selles funktsioonis on i ja ch koha- */
   int      i;            /* likud muutujad. Kehtivad on ka */
```

```
   char     ch;           /* parameetrid a ja c. Parameetrite */
                          /* väärtusi saab praegu küll lugeda, aga */
                          /* mitte muuta (kui nad ei ole viidad) */
   ...
   {                      /* siin algab uus blokk, mis */
      double i;           /* defineerib uue muutuja i */
      i = 0.1;            /* praegu on ülemise bloki muutuja i
                          /* küll kehtiv, kuid mitte nähtav */
      ch = c + 'A';       /* ülemise bloki muutuja ch on  */
                          /* aga kehtiv ja nähtav */
   }
}
...
```

Muutujad omavad salvestamisklassi. See atribuut määrab vastava mälupesa reserveerimise (*allocation*) ja vabastamise (*free*) korra. Erineva salvestamisklassiga muutujate mälupesad reserveeritakse erinevat tüüpi mälus ja nad omavad erineva pikkusega eluiga (*lifetime or duration*). Programmeerimiskeel C tunneb kolme salvestamisklassi:

- Pidev (*static*) - Pidevad objektid salvestatakse programmi andmesegmendis. Nende jaoks vajalikud mälupesad reserveeritakse kohe programmi alguses ja säilitatakse kuni programmi lõpuni. Kui mingi funktsioon muudab pideva muutuja väärtust, siis jääb see väärtus kehtima ka peale funktsiooni töö lõppu. Kõik üldised (*global*) muutujad on pidevad. Mingi kohaliku muutuja salvestamiseks pidevana tuleb kasutada võtmesõna **static** või **extern**.
- Ajutine (*local*) - Ajutised objetid salvestatakse pinus (*stack*) või võimaluse korral protsessori registrites. Seega peavad nad olema kohalikud muutujad. Te võite kohaliku muutuja puhul kasutada võtmesõna **auto** tema ajutise iseloomu määramiseks, kuid see on tarbetu, kuna translaator loob sellised muutujad niigi ajutistena. Ajutine muutuja tuleb defineerida vastava bloki alguses. Tema mälupesa reserveeritakse blokki sisenemisel ja vabastatakse kohe peale bloki töö lõppu. Seega kõlbavad ajutised muutujad vaid vahetulemuste salvestamiseks, mitte aga lõpliku tulemuse üleandmiseks. Ka funktsioonide parameetrid on ajutised. Ajutisele muutujale võib veel anda võtmesõnaga **register** uue salvestamisklassi. Sel juhul üritab translaator salvestada antud muutujat mingis protsessori registris. Võtmesõna **register** on ainult viide translaatorile, mitte aga käsk. Kui translaatoril see ei õnnestu, siis salvestab ta antud muutuja ikkagi pinus. Kasutage võtmesõna **register** vaid üksikute muutujate puhul, mille väärtusi sageli muudetakse või loetakse.
- Dünaamiline - Dünaamilised muutujad salvestatakse põhimälus. Siin ei olegi tegemist otseselt muutujatega, vaid reserveeritud mälublokkidega. Dünaamilise mälu reserveerimiseks tuleb kasutada C- keele mälu reserveerimise funktsioone *malloc(), calloc()* jne. Antud funktsioonid loovutavad programmile viida reserveeritud mälule, mille kaudu selle mälu sisu saab muuta ja lugeda. Peale töö lõppu selle mälublokiga tuleb ta ise mälust eemaldada funktsiooni *free()* abil.

Muutujad on algselt kehtivad vaid selles failis, kus nad defineeritakse. Nende kasutamiseks ka teistes *.C failides tuleb nad seal uuesti deklareerida, kasu-

tades võtmesõna **extern**. Selline deklaratsioon toimub tavaliselt faili alguses, kuid võib toimuda ka mingis funktsioonis. Sel juhul oleks teises failis defineeritud üldine muutuja antud failis kasutatav vaid tolle funktsiooni kohaliku muutujana. Kuna te võite vaid üldisi muutujaid uuesti deklareerida, siis on ka kohaliku muutujana deklareeritud **extern** muutuja pideva iseloomuga.

Konstandid

Konstandid on sümbolite jadad, mis märgivad kindlaid väärtusi. Konstante saab kas omistada otse muutujatele või kasutada neid avaldistes. Programmeerimiskeel C tunneb nelja liiki konstante:

1. Täisarvud - Täisarvuliste konstantide märkimiseks kasutatakse numbrilisi sümboleid 0 - 9 ja tähti a - f ning A - F. Täisarvulised konstandid võivad olla kümnend-, kaheksand- ja kuueteistkümnendsüsteemis. Kümnendsüsteemis konstandid võivad sisaldada suvalisi numbreid, seejuures ka nulli, näiteks 112 või 0, kuid ei tohi alata nulliga. Kaheksandsüsteemis konstandid peavad algama nulliga ja võivad sisaldada numbreid 0 - 7, näiteks 0235 või 076. Kuueteistkümnendsüsteemis konstandid peavad algama sümbolitega **0x** ja võivad sisaldada kõiki numbreid ning tähti a - f , A - F. Näiteks 0x1AC4 või 0x123.
2. Murdarvud - Murdarvulised konstandid koosnevad täisarvulisest osast, kümnendpunktist (ameerika stiilis kasutatakse siin täisarvulise ja murdarvulise osa eraldamiseks koma asemel punkti), murdarvulisest osast, astmest ja vajaduse korral sümbolitest f, F, l või L. Näiteks 23.76e3 (= 23.76 * 1000). Nulliga võrduva täisarvulise osa võib ära jätta. Sel juhul algab konstant punktiga, näiteks 0.15 on sama, mis .15. Nulliga võrduva murdarvulise osa võib samuti ära jätta, kuid punkt tuleb lisada, kui te ei soovi täisarvulist konstanti sisestada. Näiteks 12.0 asemel 12.. Negatiivse konstandi ees peab olema märk "-", kuid positiivse konstandi eest võib märgi "+" ära jätta. Aste algab sümboliga "e" või "E", millele järgneb astendaja. Ka siin võib positiivse astme puhul märgi sisestada või mitte. Näiteks 1.23e-5, 23.7E18 ja .56E-12. Tavaliselt on murdarvulised konstandid tüübist *double*, kuid te võite need tähe **f** või **F** lisamisega muuta tüübiks *float* ning tähtede **l** või **L** abil tüübiks *long double*. Tüüpe käsitletakse järgmises peatükis.
3. Sümbolid - Sümbolid on suvalised tähed ja teised ASCII sümbolid, mis on ümbritsetud ülakomadega, näiteks 'a', 'G', '5' või '\n'. Märkida tuleks, et sümbol '\n' on üks ja mitte kaks tähte, kuna ta märgib tegelikult ASCII sümbolit järjekorranumbriga 13. Nimetatud sümbol ise ei oleks aga ekraanil otseselt kujutatav. Käsukoodide loetelu leiate tabelist 1 leheküljel 16. Suvalisi ASCII sümboleid saab asendada neile vastava konstandiga, kasutades sümbolit \ ja vajaliku sümboli numbrit kaheksand- või kuueteistkümnendsüsteemis, näiteks \03 või \x12. Seda võimalust kasutatakse sageli näiteks täppidega tähtede väljastamiseks ekraanile. Tavaliselt on sümbolkonstandid

tüübist **int** ja nende salvestamiseks kasutatakse 16 bitti. Ülemine bait täidetakse nullide või ühtedega vastavalt sellele, kas te olete valinud menüüst ***O**ptions* | ***C**ompiler* | ***C**ode* Generation tüübi *char* jaoks *unsigned* või *signed*. Programmeerimiskeeles C++ on sümbolkonstandid tavaliselt tüübist **char** (8 bitti). Peale selle tunneb Borland C++ translaator ka nn. kahest tähest koosnevaid sümboleid. Siin lihtsalt kasutatakse ära 16 biti olemasolu ja salvestatakse sinna korraga kaks tähte. Selline konstant ümbritsetakse samuti ülakomadega, näiteks 'AC' või A5'.

4. Tähejadad (*strings*) - Tähejadad võivad sisaldada suvalise arvu tähti, mis on ümbritsetud jutumärkidega, näiteks: "See \t on lihtne konstant". Tähejadad salvestatakse massiividesse, millest tuleb juttu tüüpide juures. Tähejadad võivad sisaldada ka numbrite sümboleid, kuid mittekujutatavate sümbolite asemel tuleb jadasse sisestada vastava sümboli number koos sümboliga\. Näiteks: "Mul \0245nnestus seda teha!". Sümbol \245 vastab tähele 'õ'.
5. Loendused (*enumerations*) - Loendused defineerivad uue andmetüübi koos tema võimalike väärtustega. Näiteks *enum viljad { kirsid, pirnid, ploomid };* . See näide defineerib täisarvulise tüübi *viljad*, mis võib omada loetelus toodud väärtusi. Tegelikult salvestatakse antud tüübist muutuja kui päris tavaline täisarv ja loetletud nimedele seatakse vastavusse mingid numbrid. Tavaliselt alustatakse nimedele väärtuste omistamist nullist. Te võite aga seda muuta, omistades neile ise definitsioonis mingeid väärtusi, näiteks: *enum viljad { kirsid, pirnid=4, ploomid=-2 };* Translaator omistab nüüd nimele kirsid ikkagi numbri 0 ja ülejäänutele teie poolt pakutud numbrid. Selliseid konstante (loendatud nimesid) võite kasutada igal pool, kus numbrilisi konstantegi.

Andmetüübid

Programmeerimiskeel C nõuab iga muutuja jaoks selle andmetüübi määramist. Tüüp määrab muutuja salvestamiseks vajaliku mälupesa suuruse ja sinna salvestatud andmete tähenduse. Programmeerimiskeel C tunneb kolme põhilist andmetüüpi - sümbolid, täisarvud ja murdarvud. Iga tüüp koosneb omakorda mitmetest alamtüüpidest. Lisaks sellele saab sama tüüpi muutujaid koguda massividesse (*array*), opereerida nende aadressidega (*pointers and references*) ja luua tuntud tüüpide alusel uusi tüüpe (*typedefs*). Erinevaid tüüpe saab samuti koguda andmestruktuuridesse (structures and unions) ja luua niimoodi uusi tüüpe. Programmeerimiskeel C++ lisab sellele kõigele veel klassid, mis tähendavad tegelikult andmete ja nende töötlemiseks sobivate funktsioonide kogumist ühte tüüpi.

Põhilised tüübid vajavad teatud kindla hulga mälu. Kõige väiksem andmeühik on bitt. Üks bitt võib omada vaid väärtusi 1 ja 0. Seega võib üks bitt sisaldada vastust mingile küsimusele stiilis "ja või ei". Programmeerimiskeel C võimaldab ka üksikute bittide väärtusi muuta, kuid bitt ei ole siiski põhitüüp. Väikseim adresseeritav mäluühik on bait, mis koosneb kaheksast bitist. Baidi

sees omab iga bitt kindlat positsioon, mis määrab tema tähenduse. Esimene bitt omab järjekorranumbrit 0 ja viimane 7.
Meie tavaline arvude süsteem on kümnendsüsteem. Ka siin omab iga positsioon erinevat tähendust. Näiteks arv 235 on väljendatav kujul:

$$2 * 10^2 + 3 * 10^1 + 5 * 10^0 .$$

Iga koht on kümne aste ja seega on see arv kümnendsüsteemis. Arvuti aga kasutab kahendsüsteemi. Iga koht on kahe aste ja võib omada vaid väärtusi 1 ja 0. Nimelt siin tulebki mängu bitt. Bitt on arvuti arvudesüsteemi aluseks. Iga suurem andmetüüp koosneb järjestatud bittidest, mis määravad arvu 2 astmete väärtused. Näitena toodud arv 235 muundatakse arvuti sees kujule 11101011, mis tähendab:

$$1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$$

Nimetatud arv on väljendatav ka ühe baidi abil, kuna $2^8 - 1 = 255$, mis on suurim ühe baidi abil väljendatav arv. Suuremate arvude jaoks on vajalik rohkem baite. Ühe sümboli jaoks aga jätkub ühest baidist. Tähed ja muud sümbolid salvestatakse arvutis numbrite abil. Igale sümbolile seatakse vastavusse mingi kindel arv vahemikus 0 - 255. Näiteks tähele 'A' vastab arv 65. Erinevaid koode on palju, neist kõige tuntum on ASCII kood.
Lõputute nullide ja ühtede jadadega opereerimine on raske ja seepärast kasutatakse veel kaheksand- ja kuueteistkümnendsüsteemi. Nende süsteemide alused on arvu 2 astmed ja seega on ühest süsteemist teise konverteerimine üsna lihtne.
Nagu juba mainitud, on igat tüüpi muutuja jaoks vajalik kindel arv baite. See arv on iga tüübi jaoks ühel protsessoril konstantne väärtus, kuid ärge lootke, et see on nii iga protsessori tüübi puhul. Kui te soovite teada mingi tüübi suurust baitides (näiteks faili salvestamisel või sealt lugemisel), siis kasutage programmi porteeritavuse huvides operaatorit *sizeof()*. Lisaks sellele vahetavad Inteli protsessorid suuremat tüüpi muutuja salvestamisel järjestikused baidid omavahel (*byte swapping*). See teema aga kuulub rohkem masinkoodi piiresse.
Programmeerimiskeel C kontrollib küll muutujatele väärtuste omistamisel tüübi sobivust, kuid ei keela omistada ka ebasobivat tüüpi väärtust. Võimaluse korral konverteerib translaator selle väärtuse enne muutujale omistamist vajalikku tüüpi. Te võite aga ka ise väärtuste tüüpe konverteerida, kasutades vajaliku tüübi nime ümarsulgudes. Näiteks:

```
int   i;           /* täisarv */
float f1, f2;      /* kümnendmurrud */
...
f1 = 1.3;
f2 = 2.5;          /* summa on tegelikult 3.8 */
        /* enne omistamist konverteerime summa täisarvuks ==> 3 */
        /* murdosa lihtsalt eemaldatakse. Ümardamiseks peate */
        /* defineerima vajaliku funktsiooni */
i = (int) (f1 + f2);
```

Seega on programmeerimiskeeles C võimalik ka näiteks liita tähtedele arve ja neist arve lahutada. PASCAL - keeles ei ole see nii lihtsalt võimalik. Selline tüüpide segamine on ohtlik ja võib tekitada palju vigu, kuid ettevaatlikul kasutamisel võib see tunduvalt tõsta programmi töökiirust.

Täisarvud

Täisarvud baseeruvad kõik tüübil **integer** (*int*). Arvutis kasutatavad täisarvud erinevad teataval määral matemaatikast tuntud täisarvudest. Matemaatikas on täisarvude hulk loenduv, kuid lõpmata suur. Igale täisarvule võib leida uue täisarvu, mis on temast ühe võrra suurem või väiksem. Arvutis kasutatavate täisarvude hulk on aga piiratud, kuna iga arvu jaoks on ette nähtud vaid teatud kindel hulk baite. Seepärast suudab üks muutuja tüübist **integer** salvestada väärtusi vaid kindlast täisarvude osahulgast. Nii on loodud hulk tüübi **integer** alamtüüpe, mis suudavad salvestada suuremaid või väiksemaid arve. Järgnevas tabelis näete tüübi **integer** alamtüüpe koos vastavate nimede, vajalike baitide arvu ja võimalike väärtuste piirkonnaga.

Tüübi nimed	Suurus baitides	Väärtuste piirkond
int	2	-32 768 kuni 32 767
unsigned int ehk *unsigned*	2	0 kuni 65 535
short int ehk *short*	2	-32 768 kuni 32 767
unsigned short int ehk *unsigned short*	2	0 kuni 65 535
long int ehk *long*	4	- 2 147 483 648 kuni 2 147 483 647
unsigned long int ehk *unsigned long*	4	0 kuni 4 294 967 295

Tabel 3: Täisarvude tüübid

Tabelis 4 lähtutakse oletusest, et te kasutate Intel 80286 tüüpi protsessoriga arvutit. Teistsuguse protsessoriga arvutitel on need andmetüübid küll olemas, kuid nende suurused ja seega ka võimalike väärtuste piirkonnad võivad toodud andmetest erineda. Näiteks i80386 tüüpi protsessoriga arvutitel on tüübi *int* suuruseks 4 baiti.

Tüüp *int* on defineeritud nii, et ta reserveerib antud arvutil sobivaima suurusega mälupesa. Intel 80286 tüüpi protsessoriga arvutid on 16-bitised, s.o. nende andme- ja aadressisiin on 16 biti laiused. Seega suudavad nad korraga toimeta-da mälust protsessorisse 16 bitti. Neljabaidise andmehulga jaoks on vaja and-meid transportida kahes jaos. Kui nüüd andmesegmendis on enne 16-bitist *int* tüüpi muutujat salvestatud üks ühebaidine muutuja ja sellele järgnev *int* algab

seepärast paarituarvuliselt aadressilt, siis tuleb see toimetada protsessorisse hoolimata oma sobivast suurusest ikkagi kahes jaos. See on tingitud Intel protsessorite ehitusest. Probleemi lahendamiseks kasutatakse sageli nn. *align* käsku, mis sunnib translaatori salvestama suuremaid andmetüüpe alustades paarisarvuliselt aadressilt. Vajaduse korral jäetakse muutujate vahele lihtsalt tühje kohti. Borland C translaatori puhul kasutage sel puhul menüü *Options | Compiler | Code Generation* dialoogis valikut **Word alignment**.
Andmetüübi suuruse hankimiseks baitides kasutage operaatorit *sizeof()*. Näiteks:

```
i = sizeof(int)   /* mis peaks olema 2 */
```

Need väärtused on erinevatel arvutitel erinevad. Kindel on vaid see, et:

sizeof(short) <= sizeof(int) <= sizeof(long) , ning
sizeof(unsigned short) <= sizeof(unsigned) <= sizeof(unsigned long).

Täisarvude piirkonna piiratusest tuleneb see, et lisades suurimale täisarvule ühe on tulemuseks vastava tüübi väikseim arv. Näiteks:

```
int   i;
...
i = 32767;   /* = sümboolne konstant INT_MAX --> LIMITS.H */
i = i + 1;   /* i on nüüd - 32 768, mitte +32 768 !!! */
```

Seda viga nimetatakse piiri ületamiseks (*owerflow*). Osa translaatoreid väljastab vastaval juhul veateate, osa vaid siis, kui seda otse nõutakse (vastava translaatori käsuga) ja osa üldse mitte. Siin tuleks kasutada päisefailis LIMITS.H defineeritud sümboolseid konstante ja ise kontrollida, et piiri ei ületataks.

Murdarvud

Murdarvud on vajalikud mitmesuguste probleemide lahendamisel. Programmeerimiskeel C tunneb vaid kümnendmurde. Murdarvud salvestatakse arvutis koprotsessori (x87) formaadis. See on määratud IEEE (s.o. *Institute of Electronics and Electrical Engineering*) formaatidega nr. 754 ja 845. Selle kohaselt võivad arvud omada formaati: *float* (4 baiti), *double* (8 baiti) ja *long double* (10 baiti). Iga murdarv jagatakse mantissiks ja astmenäitajaks. Mõlemad omavad märki (+ või -). Enne salvestamist arv normeeritakse. Näiteks 123.456 --> 1.23456 * 10^2. Seega on antud arvu astmenäitaja 2. Astmele liidetakse enne salvestamist mingi kindel konstant (*bias*), et kõik astmed oleksid positiivsed. Suurimale negatiivsele astmele vastab järelikult konstant 1. Üldiselt vastab igale positiivsele astmele aste + *bias*. Nimetatud konstant ehk *bias* on iga formaadi puhul erinev:

float: bias = 127, eksponendi jaoks kasutatakse 8 bitti. Ülejäänud 24 bitti lähevad mantissi kirja panemiseks. Mantiss normaliseeritakse nii, et esimene bitt oleks alati 1 ja see esimene jäetakse salvestamata (on ju ta väärtus alati teada). Selle asemel salvestatakse mantissi esimese bitina kogu arvu märk (1, kui töödeldav murdarv on negatiivne). Suurim negatiivne aste on -126 ja suurim positiivne aste 127 ;

double: bias = 1023, astmenäitaja jaoks kasutatakse 11 bitti ja mantissi jaoks 53 bitti. Suurim negatiivne aste on -1022 ja positiivne 1023. Kogu arvu jaoks kasutatakse 64 bitti;

long double: on teistest erinev. Siin salvestatakse alati ka arvu esimene bitt. Ta võib olla ka 0 (näiteks 0,00001*10 astmes -201. Sellega saavutatakse suurem täpsus arvu 0 ümber . Kogu murdarvu märk salvestatakse nüüd astmenäitaja alguses. Bias = 16383. Suurim negatiivne aste on -16382 ja suurim positiivne aste 16383. Arv salvestatakse 10 baidiga.

Murdarvu tüübi nimi	Suurus baitides	Piirkond
float	4	3.4 x 10^{-38} kuni 3.4 x 10^{38}
double	8	1.7 x 10^{-308} kuni 1.7 x 10^{308}
long double	10	3.4 x 10^{-4932} kuni 3.4 x 10^{4932}

Tabel 4: Murdarvude tüübid

Tänu astmete kasutamisele saab kümnendmurru kujul salvestada suuremaid arve kui täisarvu kujul ja tänu murdarvulisele osale ka suurema täpsusega. Paraku on ka murdarvud piiratud ja peale selle ei ole nende hulk arvteljel pidev, vaid sisaldab vastavalt tüübile suuremaid või väiksemaid auke. See on tingitud mantissi piiratusest. Tüüp *float* suudab salvestada seitsme-, *double* viieteistkümne- ja *long double* üheksateistkohalisi arve. Viimane tüüp on ette nähtud matemaatikaprotsessori 80x87 jaoks. Programmeerimiskeel C sisaldab

murdarvude jaoks teeke paljude funktsioonidega nagu näiteks *sin()*, *cos()*, *tan()* jne. Matemaatikaprotsessor oskab neid funktsioone ka iseseisvalt lahendada. Kui seda protsessorit ei ole, siis on võimalik vastavaid funktsioonie lahendada ka põhiprotsessori abil, mis aga teeb seda sadu kordi aeglasemalt. Kui teil on matemaatikaprotsessor, siis kasutage teda kindlasti. Selleks on vaid vaja translaatorile selle protsessori olemasolust teatada. Borland C keskkonna puhul on võimalus menüü *Options | Compiler | Advanced Code Generation* abil ekraanile ilmunud dialoogis valida **emulation** (matemaatiliste funktsioonide lahendamine põhiprotsessori abil) ja **80x87** (matemaatikaprotsessori kasutamine) vahel. Matemaatikaprotsessori kasutamine ei nõua programmis mingeid muudatusi. Kui te olete transleerinud oma programmi nii, et ta kasutaks matemaatikaprotsessorit, siis ei suuda too programm enam töötada, arvutitel, mis sellist protsessorit ei oma. Sellisel juhul tuleks programm uuesti transleerida ilma matemaatikaprotsessori kasutamiseta.

Sümbolid

Sümboleid salvestatakse arvutis tavaliselt ühe baidi abil. Igale tähele vastab teatav kindel kood. Need koodid on kindlaks määratud ASCII tabelis. Peale ASCII on olemas ka teisi koode nagu näiteks EBEDIC, mida kasutatakse suurarvutitel jms. Osa UNIX-süsteemidest tunneb küll ASCII koodi, kuid kasutab igas baidis tähe jaoks vaid seitset bitti, võimaldades nii kodeerida vaid 128 tähte ja sümbolit. Käesolevas raamatus käsitleme vaid standardset ASCII koodi.
Standardne ASCII kood on aga vaid esimese 128 sümboli osas alati muutumatu. Ülejäänud sümbolite osas sõltub tegelikult ekraanile väljastatav sellest, millise alamkoodi (code page) te valisite. Tavaliselt kasutatav alamkood 437 (ameerika) ei sisalda paljusid vajalikke sümboleid nagu näiteks tähti õ ja Õ. Rahvusvaheline alamkood (850) sisaldab ka neid tähti, kuid vähem joon-elemente. Nimetatud erinevuste tõttu on programmeerimiskeeles C muutujate ja funktsioonide nimedes lubatud kasutada vaid sümboleid koodidega vahemikus 33 - 128, s.o. a -z, A - Z ja 0 - 9.
Sümbolite salvestamiseks kasutatakse vaid tüüpi *char*. Näiteks:

```
char  a, b;
...
a = 'A';   /*omistame muutujale a sümboli 'A' --> kood: 65 */
b = a + 3; /*sümboleid ja täisarve võib liita ja lahutada ==>b ='D'*/
```

Sel põhimõttel saab ka näiteks ühekohalisi numbreid konverteerida vastavateks sümboliteks :

```
char  a,    b;
...
a = 3;                /* number 3 */
b = a + '0';          /* sümbol '3' */
```

Selline konverteerimine ei ole aga päris korrektne. Mõned operatsioonisüsteemid kasutavad kahebaidilisi sümboleid. Ka sümbolite koodid võivad muutuda. Seepärast on parem kasutada standardseid C - keele funktsioone *atoi()*, *atof()* jt.
Sümboleid jaotatakse kindlatesse gruppidesse: väiketähed, suurtähed, numbrid jne. Päisefail CTYPE.H sisaldab nende sümbolite tüüpide selgitamiseks mitmete funktsioonide prototüüpe, nagu näiteks *isalpha()*, *isdigit()* ja *ishexdigit()*. Neid funktsioone kasutatakse sageli näiteks translaatorite programmeerimisel.
Sümboleid salvestatakse sageli sümbolite jadana (string). Selline jada võib sisaldada suvalise hulga sümboleid. Jada lõpus peab olema sümbol koodiga null. Teised programmeerimiskeeled käsitlevad stringe erinevalt. Programmeerimiskeel C sisaldab palju funktsioone stringide töötlemiseks. Nende prototüübid on salvestatud päisefailis STRING.H. Stringid koosnevad ainult sümbolitest. Teiste andmetüüpide suuremal hulgal salvestamiseks kasutatakse massiive.

Tuletatud tüübid

Peale põhitüüpide on programmeerimiskeeles C võimalik defineerida ka uusi tüüpe. Need tüübid peavad rajanema põhitüüpidel. Näiteks ei tunne programmeerimiskeel C loogiliste muutujate jaoks mingit eraldi tüüpi. Muutuja tunnistatakse tõeseks, kui tema väärtus ei ole võrdne nulliga ja vääraks, kui ta väärtus on null. Seega võiksite defineerida loogiliste muutujate jaoks uue tüübi järgmisel kujul:

```
#define FALSE       0         /* ainult väärtus 0 tähendab väära */
#define TRUE        !FALSE    /* kõik ülejäänud väärtused vastavad
                                       tõesele */
typedef      BOOL    char;    /* kasutame selle andmetüübi jaoks
                                võimalikult vähe mälu */
```

Siin kasutati uue tüübi defineerimiseks võtmesõna *typedef*. Loodud tüüp langeb täielikult kokku ühe põhitüübiga ja tema mõtteks on vaid programmi teksti muutmine paremini loetavaks. Edaspidi võib juba muutujate loomiseks kasutada tüüpi BOOL. Lisaks sellele defineeriti veel muutujate ja funktsioonide väärtuste võrdlemiseks kaks sümboolset konstanti TRUE ja FALSE .
Tuletatud tüüpide hulka kuuluvad massiivid, viidad, funktsioonide väärtused ja andmestruktuurid.

Massiivid

Massiivid (array) ei kuulu enam põhitüüpide hulka. Massiiv sisaldab mingit hulka muutujaid, mis kõik omavad sama tüüpi. See tüüp võib olla üks põhi-

tüüpidest (täisarv, murdarv, sümbol) või mingi muu tuletatud tüüp. Massiivi defineerimisel lisatakse elementide arv muutuja nime järele kandilistesse sulgudesse. Numbri asemel võib sisestada ka konstantse väärtusega valemi. Näiteks:

```
int   nNumbers[20];                   /* kakskümmend täisarvu */
char  Name[10], FamiliyName[2 * 10]; /* kaks sümbolite jada isiku
                          ees- ja perekonnanime salvestamiseks */
#define LINES  25;
#define COLS   80;
char   Display[LINES][COLS];
```

Nüüd ei määra muutuja nimi enam üht konkreetset muutujat, vaid tervet hulka. Massiivi ühe konkreetse elemendi väärtuse lugemiseks või muutmiseks tuleb lisada massiivi nimele soovitud elemendi järjekorranumber kandilistes sulgudes. Meeles tuleb aga pidada, et massiivi esimese elemendi number on 0, mitte 1 ja viimase oma (elementide hulk - 1). Selle reegli rikkumisega võite tekitada raskesti avastatava vea, mida translaator ei avasta. Kui te salvestate midagi ülemises näites defineeritud muutuja nNumbers elementi numbriga 20, siis kirjutate sellega üle temale järgneva muutuja Name esimesed kaks baiti (1 integer sisaldab 2 baiti).
Korrektsed tehted oleksid:

```
nNumbers[3] = 356;
nNumbers[16] = nNumbers[2] + 56;
strcpy(Name, "Peeter"); /* täidab massiivi Name esimesed kuus kohta
                    nimetatud sümbolitega ja sisestab seitsmendasse
                    kohta, s.o. kohta nr.6 lõpu tähisena 0*/

Name[0] = 0;        /* sisestab esimesse positsiooni arvu null.
                       See on lihtsaim viis jada tühjendada */
```

Nagu juba näha, kasutatakse massiive sageli sümbolite jadade (stringide) salvestamiseks. Sümbolite jada vajab alati lõpumärgi jaoks ühe positsiooni rohkem, kui seda on stringi pikkus. Seda saab kasutada kiireks stringi tühjendamiseks. Kui esimesse positsiooni sisestada arv null, siis stringide jaoks loodud funktsioonid enam ei uurigi muude positsioonide väärtust.
Massiivid võivad sisaldada ka teisi massiive, s.o. olla mitmedimensionaalsed. Selleks lisatakse massiivi nimele veel ühed (või mitmed) kandilised sulud koos vastava elementide hulgaga, näiteks:

```
int   M[4][2];          /* 4 x 2 massiiv */
```

Arvuti mälus salvestatakse selline n - dimensionaalne massiiv järjestikku, alustades kõige vasakpoolsemast dimensioonist. Massiiv *M* salvestatakse seega kui neli kaheliikmelist gruppi, näiteks: M(1,1), M(1,2), M(2,1), M(2,2), M(3,1), M(3,2), M(4,1), M(4,2).
Tuletatud tüüpide puhul eristatakse täieliku (*complete*) ja ebatäieliku (*incomplete*) definitsiooni vahel. Täieliku definitsiooni puhul märgitakse ära kõik antud tüübi vajalikud andmed. Üleval toodud näited sisaldavad vaid täielikke definitsioone.

Ebatäieliku definitsiooni puhul tuuakse vaid uue tüübi nimi või veel mõned andmed, kuid ei tooda küllaldaselt andmeid soovitud tüübi suuruse määramiseks. Massiivi puhul oleks ebatäielik näiteks selline definitsioon, mis ei sisalda massiivi elementide arvu. Ebatäieliku definitsiooni kasutamiseks on vaja elemendi suurus hiljem ikkagi kuidagi määrata. Näiteks:

```
char  Name[] = "Peeter";
```

Toodud näites jäeti stringi pikkus määramata, kuna me ta kohe täidame teatud kindla pikkusega konstandi abil ja seega on ka elemendi suurus määratud. Selles näites oleks elemendi pikkuseks 7 (6 tähte ja lõpumärgina null). Selle pikkuse arvutab translaator. Näidatud kujul salvestatakse programmis sageli vajalikke stringe, mille suurus ja sisu ei muutu. Võimalik oleks ju ka elemendi suurus kohe määrata, kuid selleks tuleks ise kõik sümbolid üle lugeda. Lihtsam on lasta seda teha translaatoril. Kui aga määrata element palju suurem kui vaja, siis raiskame asjatult ruumi. Ebatäielikku definitsiooni kasutatakse sageli ka funktsiooni parameetri tüübi määramiseks, et võimaldada anda üle suvalise suurusega parameeter.
Mitmedimensionaalsete massiivide puhul võib jätta määramata vaid esimese dimensiooni. Näiteks:

```
int nNumbers [][3];  /* n x 3 massiiv */
```

Viidad

Viidad on muutujad, mis sisaldavad teise muutuja või muutujate hulga mälupesa aadressi. Te võite muuta viida enda sisu, mispeale viit näitab mingit teist mälupesa, või viida poolt määratud mälupesa sisu. Viimasel juhul jääb viida oma sisu muutumatuks. Viida defineerimiseks kasutatakse sümbolit *. Näiteks:

```
int   *pNumber, i;      /* Täisarv i ja viit täisarvule pNumber */
...
i = 3;                  /* Omistame täisarvule i mingi väärtuse */
pNumber = &i;           /* Omistame viidale pNumber i aadressi.
                           Viit näitab nüüd arvule 3. */
*pNumber = 45;          /* Muudame nüüd viida poolt näidatud mälu-
                           pesa, s.o. täisarvu i sisu.
                           Muutuja i on nüüd võrdne 45 ga */
```

Toodud näites defineeritakse täisarv *i* ja viit *pNumber*. Seejärel kasutatakse muutuja *i* aadressi hankimiseks sümbolit **&**, mis omistatakse muutujale *pNumber*. Nüüd on muutuja *pNumber* sisuks *i* aadress ja ta näitab seega arvule 3. Kui muudame nüüd *pNumber* poolt osutatud mälupesa, muudame ka *i* sisu.
Viitadega tuleb olla väga ettevaatlik, kuna just nendega töötamisel tekib eriti palju vigu. Teisest küljest annavad just viidad programmeerimiskeelele C tema suure paindlikkuse ja sageli ka kiiruse.

Viitasid on kahte tüüpi: lühikesed (*near*) ja pikad (*far*). Lühikesed viidad sisaldavad vaid muutuja aadressi vastava andmesegmendi piires (*offset*), pikad aga sisaldavad ka segmendi aadressi. Kas viit on lühike või pikk, sõltub programmi transleerimisel kasutatud mälumudelist. Mälumudelitest ja mälu haldamisest tuleb juttu natuke hiljem. Viida pikkuse määramiseks kasutage vajaduse korral võtmesõnu **near** ja **far**. Üldiselt ei ole see vajalik, kuna iga mälumudeli puhul on sarnased viidad ühesuguse pikkusega ja vajaduse korral saab ka translaator nende konverteerimisega hakkama.
Igast tüübist saab sümboli ***** abil luua temale vastava viida. Viida tüübi määramine võimaldab translaatoril kontrollida selle viidaga sooritatud tehete õigsust. Viitasid kasutatakse sageli ka andmeblokkide või massiivide täitmisel väärtustega. Sel puhul on tähtis, et viit oleks tuletatud sobivast põhitüübist. Vastavalt viida tüübile võivad samad tehted viidga omada erinevat mõju. Kui te näiteks defineerite viida tüübile *char*, siis muutub viida sisu temale arvu 1 liitmisel ühe võrra. Täisarvule (*integer*) osutavale muutujale ühe liitmisel muutub tema sisu aga kahe võrra. See on tingitud sellest, et nimetatud tüübid omavad erinevaid pikkusi (baitides). Viidale mingi arvu liitmine ei tähenda aga otseselt tema väärtuse muutmist, vaid hoopiski viida nihutamist soovitud suunas määratud arvu positsioonide võrra. Kuna need positsioonid on erinevat tüüpi viitade puhul erineva pikkusega, siis on ka viida väärtuste otsesed muutused erinevad. Näiteks:

```
int   *pi, i[10];
char  *pc, c[10];
...
pi = &i[5];
pc = &c[5];
pi = pi + 1;      /* Viit pi nihkus massiivis i ühe positsiooni
                     võrra  edasi ja näitab nüüd muutujale i[6].
                     Tema sisu aga muutus kahe võrra. */
pc = pc + 1;      /* Viit pc nihkus massiivis c ühe positsiooni
                     võrra edasi ja näitab samuti muutujale c[6].
                     Tema sisu aga muutus vaid ühe võrra. */
pi = (int *)((char *)pi + 2); /* nüüd nihkus pi samuti ühe
                                 positsiooni võrra edasi. */
*pi = *pi + 2;          /* nüüd muudame otseselt viida sisu. Ka
                           seekord  on tulemuseks viida nihkumine
                           ühe positsiooni võrra */
```

Iga viida ja muutuja tüüpi saab muuta, lisades valemisse soovitud muutuja ette ümarsulgudes vajaliku tüübimäärangu. Täpsemalt öeldes saab muuta vaid avaldise hetkelist tüüpi, muutuja tüüp jääb samaks. Toodud näites konverteeritakse viida *pi* sisu kõigepealt viidaks tüübile *char* ja nihutatakse seda kahe positsiooni võrra. Nüüd konverteeritakse ta tagasi viidaks tüübile *int* ja omistatakse saadud väärtus muutujale *pi*. Kuna tüüp *int* on täpselt kaks korda pikem kui tüüp *char*, siis on viit *pi* nihkunud vaid ühe koha võrra.
On olemas ka andmetüüp *void*, mis tähendab tegelikult "mitte midagi" või "määramatu". Te ei saa luua muutujat tüübist *void*, kuid te võite luua viida tüübile *void*. Sellist viita ei saa esialgu kasutada, kuna translaator ei oska teda käsitleda. Taolisele viidale võib aga omistada mingi teise viida sisu. Nüüd on ka tollel viidal kindel tüüp ja seda tohib kasutada. Seda kasutatakse sageli

funktsioonide defineerimisel, kui nimetatud funktsioonile soovitakse anda üle suvalist tüüpi viitasid. Alles funktsiooni kasutamisel üleantud viit määrab sellise parameetri tüübi. Tüübist *void* tuletatud viidale omistamisel ei ole vaja kasutada tüübi konverteerimist. Tüüpi *void* kasutatakse ka funktsiooni väärtuse tüübina juhul, kui funktsioon ei loovuta oma kasutajale mingit väärtust või funktsiooni parameetriteloetelus, kui funktsioon ei vaja mingeid parameetreid.
Massiivid on üsna sarnased viitadega. Massiivi nimi on tegelikult viit massiivi algusesse. Seega võite kasutada massiivi nime just nagu viitagi. Näiteks:

```
int   nNumbers[10];
...
*(nNumbers + 4) = 356;  /* sama, mis nNumbers[4] = 356 */
```

Seda kasutatakse sageli stringide puhul. Iga stringe töötlev funktsioon aktsepteerib parameetrina ka sümbolitemassiivi nime. Mingit tüüpide konverteerimist ei ole siin vaja.

Andmestruktuurid

Massiivide abil võite salvestada vaid teatud hulga üht tüüpi muutujaid. Keerukamate andmestruktuuride salvestamiseks tuleks aga luua uus tüüp. Selleks võite kasutada võtmesõnu **struct** ja **union**. Uue tüübi loomisega taotletakse enamasti kokkukuuluvate andmete üheskoos salvestamist. Andmestruktuuri defineerimiseks kasutatakse järgmist sõnastust:

```
struct <uue tüübi nimi> {
  <väljad koos tüüpidega>;
} <seda tüüpi muutujate nimed>;
```

Näiteks:

```
struct Isik {
   char         Eesnimi[10];        /* isiku nimi ja perekonnanimi */
   char    Perekonnanimi[20];
   unsigned long Telefon;           /* telefoninumber */
   char    Aadress[30];             /* aadress - 29 sümbolit */
} Toomas, Peeter;         /* kaks isikut kõigi loetletud andmetega */
...
struct  Isik            Perekond[5]; /* struktuur viie elemendiga */
```

Eelmise lõiguga defineeriti uus tüüp *Isik*, mis on 10 + 20 + 4 + 30 = 64 baidi pikkune. Uue tüübi defineerimisel ei ole ilmtingimata vaja määrata tüübi nime, kuid kui te seda ei tee, siis ei saa te ka loodud tüüpi hiljem uute muutujate defineerimisel kasutada. Sel juhul jääks vaid võimalus luua kohe tüübi defineerimisel paar seda tüüpi muutujat, nagu siinloodud muutujad *Toomas* ja *Peeter*. Looksulgude vahel sisestatakse üksteise järel kõik antud andmestruktuuri elemendid. Need elemendid salvestatakse arvuti mälus samas järjekorras.

Sellist tüüpi muutuja erinevate elementide väärtuste muutmiseks või lugemiseks tuleb kasutada operaatorit **.** Näiteks:

```
strcpy(Peeter.Eesnimi, "Peeter");
Peeter.Telefon = 523467;
strcpy(Toomas.Aadress, "Tallinn EE-0026, Akadeemia 15-42");
```

Kui te aga defineerite viida sellisele andmestruktuurile, siis tuleb teil kasutada erinevate elementide muutmiseks või lugemiseks operaatorit **->**. Näiteks:

```
struct Isik                   *minu_tuttav;
unsigned long     telefon;
...
minu_tuttav = &Peeter;          /* viit muutujale Peeter */
telefon = minu_tuttav->Telefon; /* telefon on nüüd 523467 */
```

Tüübi *Isik* defineerimise järel defineerisime ka viie elemendiga struktuuri *Perekond*. See on struktuur, mille iga element on 64 baidi pikkune ja sisaldab omakorda elemente. Iga elemendi muutmiseks tuleb kõigepeal kasutada elemendi järjekorranumbrit ja seejärel peale punkti (või noolt) soovitud elemendi nime, näiteks:

```
struct Isik tuttavad;
char        nimi[10];
...
strcpy(Perekond[4].Eesnimi, "Toomas");
tuttavad = &Perekond;
strcpy(nimi, (tuttavad+4)->Eesnimi);  /* nimi on nüüd "Toomas" */
```

Loodud andmetüübi suuruse määramiseks kasutage operaatorit *sizeof()*, näiteks *sizeof(Isik)*.
Uue andmetüübi defineerimisel võib peale põhiliste andmetüüpide kasutada veel nn. bitivälju (*bitfields*). Bitiväli defineeritakse järgmiselt:
<tüübi nimi> <välja nimi> : <laius bittides>
Tüübi nimi määrab selle põhitüübi, mida kasutatakse bitivälja salvestamiseks. Võimalik on kasutada vaid tüüpe *char*, *unsigned char*, *int* ja *unsigned int*. Kui välja nime ei määrata, siis see väli küll luuakse, kuid tema väärtust ei saa kasutada. Seda kasutatakse sageli bitivälja defineerimise puhul protsessori registrite jaoks juhul, kui osa bitte ei oma mingit tähendust. Laius määrab elemendi laiuse bittides. Bitiväli võib olla 1 kuni 16 biti laiune. Näiteks:

```
struct bitid {
  int  esimene_bitt : 1;           /* üks bitt */
  int   : 3;                       /* midagi ebahuvitavat */
  int muud_bitid : 12;             /* vajalikud 12 bitti */
};
```

Andmestruktuurid võivad sisaldada ka omakorda struktuure. Sel juhul tuleb need struktuurid defineerida enne neid kasutavat struktuuri. Näiteks:

```
struct Aadress{
  char Taenav[10], Linn[10], Riik[10];
  unsigned long telefon;
  unsigned maja, korter;
};
```

```
...
struct Isik {
  char  Eesnimi[10], Perekonnanimi[20];
  struct Aadress  aadress;
} Peeter;
```

Peale võtmesõna **struct** on võimalik uut andmestruktuuri luua ka võtmesõnaga **union**. Sel puhul kehtivad kõik samad reeglid, ainult et loodud andmetüübi pikkus ei ole enam võrdne kõigi tema elementide pikkuste summaga, vaid kõige suurema elemendi pikkusega. Näiteks:

```
struct LihtnePunkt {
  char            taepne;   /* FALSE (<0) kui lihtsad koordinaadid */
  unsigned  x, y;           /* suvalise ekraanipunkti koordinaadid */

};

struct TaepnePunkt {
  char      taepne;         /* TRUE (>0) kui täpsed koordinaadid */
  unsigned long  x, y;      /* täpsed koordinaadid nagu näiteks
                               tehnilisel joonisel */
};

union Koodinaadid {
  struct LihtnePunkt  lp;   /* millist neist kasutame, sõltub */
  struct TaepnePunkt  tp;   /* olukorrast */
} Punkt1, Punkt2;

union Koordinaadid            k;
...
if(k.taepne) {
  k.x = 3;
  k.y = 4;
}
else {
  k.x = 3.56;
  k.y = 4.23;
}
```

Tüübi *Koordinaadid* pikkuseks on praegu 9 baiti, mis on võrdne ka tüübi *TaepnePunkt* pikkusega. Vastavalt olukorrast võime nüüd kasutada üht neist tüüpidest. Paraku reserveeritakse sellisele muutujale alati 9 baiti mälu. Selleks, et teada, millist tüüpi praegu kasutame, sisaldavad eri tüübid lippu *taepne*. Näidatud on ka, kuidas sellist muutujat kasutada.

Funktsioonide tüübid ja viidad

Iga funktsioon omab isiklikku tüüpi. Peale töö lõppu loovutab funktsioon seda tüüpi muutuja teda välja kutsunud funktsioonile. Selleks kasutab ta võtmesõna **return**. Nimetatud võtmesõna abil on võimalik funktsiooni tööd ka enneaegselt lõpetada (näiteks mingi vea puhul) ja loovutada väljakutsujale mingi teine väärtus. Näiteks:

```
/* funktsioon loovutab oma väljakutsujale viida sümbolite jadale */
char* Func1(char *pStr1, char *pStr2)
{
  if((pStr1== NULL) || (pStr2 == NULL))
```

```
    return NULL;  /* kui üks viitadest kuhugi ei osuta,
                     siis loovuta NULL */
  else            /* vastasel juhul lisa teine jada esimese lõppu */
    return strcat(pStr1, pStr2);
}
...
char *pStr1, *pStr2, *pStr3;
...
pStr3 = Func1(pStr1, pStr2) + 5;      /* funktsiooni kasutamine
                                         valemi osana */
```

Programmeerimiskeeles C on funktsioon nagu omapärast tüüpi muutujaks, mis loovutab oma väärtuse vaid peale hetkelist "mõtteaega". Muus osas on funktsioon nagu harilik muutuja ja teda saab seega sisestada avaldise suvalisse ossa, välja arvatud omistustehte vasakule poolele. Funktsioonile ei saa ju mingit väärtust omistada. Selle asemel võib ta aga ise omistada muutujatele väärtusi.
Kui funktsiooni (tema väärtuse) tüübiks on *void*, siis ei loovuta funktsioon oma väljakutsujale mingeid andmeid ja seega ei saa teda ka enam kasutada avaldise osana.
Te võite defineerida viitasid funktsioonidele. Selline viit erineb harilikust viidast selle poolest, et tema poolt osutatud mälupesa väärtust ei saa enam otse lugeda. Te võite aga sellise viida abil vastava funktsiooni välja kutsuda ja tema tulemuse omistada mingile muutujale. Funktsioonide viitasid kasutatakse näiteks sageli menüüde implementeerimiseks. Iga menüükäsk omaks viita vastavale funktsioonile ja selle menüükäsu valimisel kutsutaksegi too funktsioon välja. Viita funktsioonile võiks defineerida järgneva kirjega:

```
<funktsiooni väärtuse tüüp> (*<viida nimi>)(<parameetrite loetelu>);
```

Näiteks:

```
char* (*pFunc1)(char*, char*);
...
pFunc1 = Func1;   /* funktsioon Func1() on defineeritud
                              eelmises näites */
...
              /* vananenud viis funktsiooni viida kasutamiseks */
pStr3 = (*pFunc1)(pStr1, pStr2) + 5;
              /* uuem (ANSI standard) viis funktsiooni viida
                 kasutamiseks */
pStr3 = pFunc1(pStr1, pStr2) + 5;
```

Funktsiooni viitasid võib salvestada massividesse (jällegi oleks sobivaks näiteks menüüde implementeerimine). Massiivi suurust määrav number nurksulgudes ei tohiks aga olla sellise rea lõpus, vaid kohe peale massiivi nime:

```
<funktsiooni väärtuse tüüp> (*<viida nimi>[<massiivi
suurus>])(<parameetrite loetelu>);
```

Näiteks:

```
void (*MenuFunc[10])(void);
```

Avaldised ja laused

C - programmid koosnevad hulgast avaldistest (*expression*) ja lausetest (*statement*), mis täidavad mingit ülesannet. Avaldis on kombinatsioon muutujatest ja funktsioonidest ning neid siduvatest tehtemärkidest (*operators*) Lihtsaim avaldis on üksikmuutuja. Tehtemärkide abil kombineeritakse muutujate ja funktsioonide väärtused, luues nõnda uusi väärtusi, mis on avaldise enda väärtuseks. Avaldis võib omakorda sisaldada ka teisi avaldisi. Selliste avaldiste väärtused kombineeritakse samuti tehtemärkide abil muutujate ja funktsioonide väärtustega peamise avaldise väärtuse arvutamiseks. Avaldised oleksid näiteks:

```
a + b
a
c = d
func1(pStr1, pStr2)
(c + d) * funk2(e - f )
```

Avaldise väärtuse arvutamisega alustatakse tema vasakust äärest ja lõpetatakse paremal äärel. Osa tehtemärke omab suuremat prioriteeti kui teised. Nii näiteks arvutatakse korrutis enne summat. Arvutamise järjekorra muutmiseks võib kasutada ümarsulge.
Lause võib sisaldada üht või mitut avaldist. Põhiliseks lause ja avaldise erinevuseks on see, et lause lõpetab mingi tehte. Lause lõpumärgina kasutatakse programmeerimiskeeles C semikoolonit. Nii lause kui ka avaldis võivad hõlmata mitu rida. Translaator uurib avaldisi niikaua, kuni ta leiab lause lõpumärgi. Seejärel ta kas loob vastava koodi või teatab esinevatest vigadest. Avaldis ei saa eksisteerida üksinda, ta peab alati olema osa mingist lausest.
Ka lauseid võib grupeerida. Selleks kasutatakse looksulge. Kõikjale, kuhu saab sisestada ühe lause, võib sisestada ka lausete grupi. Borland C/C++ translaator tunneb järgmisi lausetüüpe:

Tüüp	Tähendus
Lihtne lause	Üks või enam avaldist, mille lõpetab semikoolon
Keerukas lause	Grupp lauseid looksulgudes. Grupi alguses võib defineerida muutujaid.
Märge	Märgend (label) programmi töö juhtimiseks võtmesõnade **switch** ja **goto** abil.
Valik	Programmi töö juhtimine vastavalt mingitele tingimustele. Sellise lause loovad võtmesõnad **switch**, **if** ja **else**.
Loendus	Programmi töö täitmine seni, kuni mingi tingimus on muutnud oma väärtust. Selline lause luuakse võtmesõnade **while**, **do** ja **for** abil.
Hüpe	Programmi töö muutmine ilma tingimuseta. Selline lause luuakse võtmesõnade **continue**, **break**, **return** ja **goto** abil.
Assembler	Otsese assemblerkoodi sisestamine võtmesõna **asm** abil.
Definitsioon	Lause muutujate defineerimiseks ja neile vajaduse korral algväärtuste omistamiseks.

Tabel 5: Lausete tüübid

Tehtemärgid

Tehtemärgid omavad programmeerimiskeeles C suurt tähtsust. Nende abil luuakse avaldisi ja lauseid. Tehtemärgid jaotatakse ühe- ja kahekohalisteks. Ühekohalised tehtemärgid mõjutavad vaid ühe muutuja väärtust ja asuvad selle ees või järel. Järgmises tabelis näete kõiki võimalikke ühekohalisi tehtemärke.

Tehtemärk	Tähendus
+	Pluss. Märgib positiivset arvu. See tehtemärk ei ole tavaliselt vajalik.
-	Miinus. Märgib negatiivset arvu. Muudab tema järel seisva muutuja märki.
!	Loogiline eitus. Muudab tema järel seisva (loogilise) muutuja tõeväärtust.
&	Aadress. Hangib tema järel seisva muutuja mälupesa aadressi.
*	Viit. Hangib tema järel seisva viida poolt määratud mälupesa sisu või määrab muutuja defineerimisel, et tegemist on viidaga.
++	Suurendamine (*increment*). Suurendab tema ees- või järel seisva muutuja väärtust. Kui tehtemärk ++ seisab muutuja ees, siis suurendatakse muutuja väärtust enne teiste tehete sooritamist. Kui ta aga seisab muutuja järel, siis suurendatakse muutuja väärtust alles peale teiste tehete sooritamist.
--	Vähendamine (*decrement*). Vähendab tema ees- või järel seisva muutuja väärtust. Kui tehtemärk -- seisab muutuja ees, siis vähendatakse muutuja väärtust enne teiste tehete sooritamist. Kui ta aga seisab muutuja järel, siis vähendatakse muutuja väärtust alles peale teiste tehete sooritamist.
~	Bitikaupa ühendkomplekt. Muudab iga 0 biti väärtuse 1-ks ja iga 1 omakorda 0-ks.

Tabel 6: Ühekohalised tehtemärgid (järg)

Kahekohaline tehtemärk ühendab vastava tehte abil kaks mingit väärtust ja loob uue väärtuse. Tabelis 8 näete kõiki tuntud kahekohalisi tehtemärke.

<table>
<tr><th>Tehtemärk</th><th>Tähendus</th></tr>
<tr><td></td><td>Matemaatilised tehted</td></tr>
<tr><td>+</td><td>Liitmine.</td></tr>
<tr><td>-</td><td>Lahutamine.</td></tr>
<tr><td>*</td><td>Korrutamine.</td></tr>
<tr><td>/</td><td>Jagamine.</td></tr>
<tr><td>%</td><td>Jagatise jääk (modulus)</td></tr>
<tr><td></td><td>Omistamistehted</td></tr>
<tr><td>=</td><td>Omistamine. Omistab vasakul pool seisvale muutujale paremal pool seisva avaldise tulemuse. Vasakul pool peab olema muutuja, et talle saaks midagi omistada. Paremal pool võib olla avaldis. Muus osas on omistamine päris harilik tehe. Te võite ühes lauses teostada mitu omistamist. Näiteks: a = b + (c = d - 34);</td></tr>
<tr><td>+=</td><td>Summa omistamine. Suurendab vasakul pool seisva muutuja väärtust paremal pool seisva avaldise väärtuse võrra.</td></tr>
<tr><td>-=</td><td>Vahe omistamine. Vähendab vasakul pool seisva muutuja väärtust paremal pool seisva avaldise väärtuse võrra.</td></tr>
<tr><td>*=</td><td>Korrutise omistamine. Korrutab vasakul pool seisva muutuja väärtuse paremal pool seisva avaldise väärtusega ja omistab tulemuse uuesti vasakul pool seisvale muutujale.</td></tr>
<tr><td>/=</td><td>Jagatise omistamine. Jagab vasakul pool seisva muutuja väärtuse paremal pool seisva avaldise väärtusega ja omistab tulemuse uuesti vasakul pool seisvale muutujale.</td></tr>
<tr><td>%=</td><td>Jagatise jäägi omistamine. Jagab vasakul pool seisva muutuja väärtuse paremal pool seisva avaldise väärtusega ja omistab jagatise jäägi vasakul pool seisvale muutujale.</td></tr>
<tr><td>&=</td><td>Bitikaupa loogilise JA omistamine. Ühendab vasakul pool seisva muutuja bitid ühekaupa loogilise JA (AND) abil paremal pool seisva avaldise väärtuse bittidega ja omistab tulemuse uuesti vasakul pool seisvale muutujale.</td></tr>
<tr><td>|=</td><td>Bitikaupa loogilise VÕI omistamine. Ühendab vasakul pool seisva muutuja bitid ühekaupa loogilise VÕI (OR) abil paremal pool seisva avaldise väärtuse bittidega ja omistab tulemuse uuesti vasakul pool seisvale muutujale.</td></tr>
<tr><td>^=</td><td>Bitikaupa loogilise XOR omistamine. Ühendab vasakul pool seisva muutuja bitid ühekaupa loogilise XOR abil paremal pool seisva avaldise väärtuse bittidega ja omistab tulemuse uuesti vasakul pool seisvale muutujale.</td></tr>
</table>

Tabel 7: Kahekohalised tehtemärgid

Tehtemärk	Tähendus
	Omistamistehted
<<=	Bitikaupa vasakule poole nihutatud väärtuse omistamine. Nihutab vasakul pool seisva muutuja bitte paremal pool seisva avaldise väärtuse võrra kohti vasakule ja omistab tulemuse uuesti vasakul pool seisvale muutujale. Täisarvu nihutamine ühe koha võrra vasakule on võrdne tema korrutamisega kahega, toimub aga palju kiiremini.
>>=	Bitikaupa paremale poole nihutatud väärtuse omistamine. Nihutab vasakul pool seisva muutuja bitte paremal pool seisva avaldise väärtuse võrra kohti paremale ja omistab tulemuse uuesti vasakul pool seisvale muutujale. Täisarvu nihutamine ühe koha võrra paremale on võrdne tema jagamisega kahega, toimub aga palju kiiremini.
	Loogilised tehted
&&	Loogiline JA (AND).
||	Loogiline VÕI (OR).
	Võrdlustehted
==	Võrdne
!=	Mittevõrdne
<	Väikesem
>	Suurem
<=	Väikesem või võrdne
>=	Suurem või võrdne
	Tehted eraldi bittidega
<<	Vasakule poole nihutamine.
>>	Paremale poole nihutamine.
&	Bitikaupa loogiline AND
|	Bitikaupa loogiline OR
^	Bitikaupa loogiline XOR
	Andmestruktuuride osade kasutamine
.	Andmestruktuuri (*struct*) elemendi väärtus.
->	Välja kasutamine andmestruktuurile osutava viida abil.
	Tingimus
a ? b : c	Kui avaldise *a* väärtus on tõene, siis arvutatakse avaldise *b* väärtus, vastasel juhul avaldise *c* väärtus.
	Koma
,	Eraldab kahte avaldist, mis ei ole üksteisega seotud. Selle märgi abil on võimalik sisestada kahte või enamat lauset kohta, kus on lubatud vaid üks. Laused täidetakse üksteise järel vasakult paremale.

Tabel 8: Kahekohalised tehtemärgid (järg)

Tehtemärgid omavad erinevat prioriteeti. Korrutamine on kõrgema prioriteediga kui liitmine ja seda arvestatakse ka valemite väärtuste arvutamisel. Ümarsulgudega saab tehete järjekorda muuta. Sama prioriteediga tehete puhul sooritatakse tehted vasakult paremale.
Lisaks sellele omavad tehtemärgid assotsiatiivsust (*associativity*). Vastavalt assotsiatiivsusele täidetakse antud tehe kas suunas vasakult paremale või paremalt vasakule. Tabelis 9 näete tehtemärkide prioriteete ja assotsiatiivsust. Tabelis esimesel real asuvad tehtemärgid omavad kõrgeimat prioriteeti. Selle järgnevad teisel real asuvad tehtemärgid jne. Viimasel real asuvad tehtemärgid on kõige madalama prioriteediga.

<table>
<tr><th>Tehtemärgid</th><th>Assotsiatiivsus</th></tr>
<tr><td>() [] -> .</td><td>Vasakult paremale</td></tr>
<tr><td>! ~ ++ -- +</td><td>Paremalt vasakule</td></tr>
<tr><td>- * & (tüübi konverteerimine) sizeof</td><td>Paremalt vasakule</td></tr>
<tr><td>* / %</td><td>Vasakult paremale</td></tr>
<tr><td>+ -</td><td>Vsakult paremale</td></tr>
<tr><td><< >></td><td>Vasakult paremale</td></tr>
<tr><td>< <= > >=</td><td>Vasakult paremale</td></tr>
<tr><td>== !=</td><td>Vasakult paremale</td></tr>
<tr><td>&</td><td>Vasakult paremale</td></tr>
<tr><td>^</td><td>Vasakult paremale</td></tr>
<tr><td>|</td><td>Vasakult paremale</td></tr>
<tr><td>&&</td><td>Vasakult paremale</td></tr>
<tr><td>||</td><td>Vasakult paremale</td></tr>
<tr><td>?:</td><td>Paremalt vasakule</td></tr>
<tr><td>= += -= *= /= %=</td><td>Paremalt vasakule</td></tr>
<tr><td>&= ^= |= <<= >>=</td><td>Vasakult paremale</td></tr>
</table>

Tabel 9: Tehtemärkide prioriteet ja assotsiatiivsus

Operaatori (<uus tüüp>) abil on võimalik küll avaldise tüüpi konverteerida, kuid see pole alati vajalik. Sarnaseid tüüpe oskab translaator ka ise konverteerida. Suuruselt väikesema andmetüübi konverteerimisel täpsemaks tüübiks ei teki probleeme. Liigsed baidid täidetakse vastavalt väärtuse märgile kas nullide või ühtedega. Täpsema väärtuse konverteerimisel vähemtäpsemaks võib aga tekkida probleeme, kuna osa saavutatud täpsusest läheb kaduma. Sel puhul väljastab translaator tavaliselt hoiatuse.
Matemaatiliste tehete puhul konverteeritakse väiksemad tüübid nagu *char*, *short* ja *enum* kõigepealt tüübiks *int* ja sooritatakse seejärel antud tehe. Kui tehe seob kahte erinevast tüübist väärtust, siis konverteeritakse kõigepealt

ebatäpsem tüüp täpsemaks ja sooritatakse seejärel tehe. Tulemuse tüüp langeb kokku täpsema tüübiga. Näiteks täisarvu ja murdarvu korrutamisel konverteeritakse täisarv murdarvuks ja seejärel korrutatakse. Kui üks väärtustest on tüübist *long double* ja teine tüübist *double*, siis konverteeritakse viimane enne tehte sooritamist tüüpi *long double*.

Programmi töö juhtimine

Programm on niisiis lausete ja funktsioonide kogumik. Viimased omakorda koosnevad samuti lausetest ja võivad ka teisi funktsioone kasutada. Programmi töö algab funktsiooni *main()* väljakutsumisega. Funktsioon *main()* võib samuti omada parameetreid:

```
int main(int argc, char **argv, char **envp)
{
...
  return 0;
}
```

Kui te programmi startides sisestate programmi nime järele mingid parameetrid, siis kopeerib operatsioonisüsteem need parameetrid eraldi mälublokki ja annab programmile üle sellele mälublokile näitava viida. See viit ongi parameeter *argv*. Näiteks abiprogramm PRINT vajab selle faili nime, mida trükkida soovitakse. Programmile üleantud parameetreid käsitletakse kui sümbolite jadasid. Kui te soovite programmile mingeid arve üle anda, siis peab programm vastavad sümbolid ise arvudeks konverteerima. Programmi parameetrite kasutamise lihtsustamiseks jaotatakse parameetrid eraldi stringideks. Seejuures kasutatakse kahe parameetri eraldusmärgina tühikuid. Saadud sõnad salvestatakse kahemõõtmelisse sümbolitemassiivi. Tolle massiivi iga element sisaldab ühe sõna. Esimene element (numbriga 0) sisaldab programmi nime. Sellele järgnevad parameetrid. Parameetrite arv antakse programmile üle parameetris *argc*. Näiteks kui te kasutate komprimeerimisprogrammi ARJ kataloogis C:\SAMP asuvate failide salvestamiseks kettale A: arhiivi SAMPLES.ARJ:

```
C:\>ARJ  -a  A:\SAMPLES  C:\SAMP\*.*
```

Nimetatud käsk jaotatakse neljaks parameetriks ja salvestatakse järgmisel kujul:

```
argv
argv(0) A R J
argv(1) - a
argv(2) A : \ S A M P L E S
argv(3) C : \ S A M P \ * . *
```

Joonis 3: Programmi parameetrite salvestamine

Parameeter *argc* omab praegusel juhul väärtust 4. Iga elemendis salvestatud sõna on lõpetatud nulliga. Viit *argv[4]*, millele enam ühtki parameetrit ei vasta ja temale järgnevad viidad omavad väärtust NULL.
Parameeter *envp* osutab samalaadsele massiivile, kuhu on salvestatud kõik keskkonnamuutujad. Keskkonnamuutujate hulka kuuluvad kõik failis AUTOEXEC.BAT käsuga SET loodud muutujad. Näiteks muutuja *PATH* on keskkonnamuutuja.
Funktsiooni main() väärtuseks on tavaliselt 0. Iga nullist erinevat väärtust käsitletakse kui veakoodi. Te võite aga luua programmi, mis vastavalt oma töö tulemusele loovutab operatsioonisüsteemile mingi väärtuse. Seda väärtust saab siis kasutada mingis .BAT programmis.
Iga funktsiooni ja bloki sees täidetakse laused järgemööda, kuni jõutakse võtmesõnani **return** või bloki lõpuni. Kui te soovite funktsiooni või bloki tööd mingite tingimuste abil mõjutada, siis tuleks bloki tööd juhtida programmeerimiskeel C tingimuslausetega.

Tingimuslause if

if-lause on lihtsaim tingimuslause. See lause võib esineda kahel kujul.

```
if(<tingimus>)
  <lause või blokk>
```

ja

```
if(<tingimus>)
  <lause või blokk>
else
  <lause või blokk>
```

Esimesel juhul täidetakse **if**-lausele järgnev lause või lausete blokk vaid siis, kui tingimuse väärtus on nullist erinev (TRUE). Kui tingimuse väärtus on null (FALSE), siis jätkatakse funktsiooni või bloki tööd kohe peale **if**-lauset paiknevatest lausetest. Teisel juhul täidetakse tingimuse kehtides esimene -

võtmesõnale **if** järgnev lausete blokk ja vastasel juhul võtmesõnale **else** järgnev lausete blokk.
Tingimusena kasutatakse tavaliselt mingit loogilist avaldist näiteks a + b > c või a == b, kuid tingimusena võib kasutada ka suvalist valemit või muutujat. Tähtis on see, et iga avaldise nullist erinev väärtus vastab tõeväärtusele TRUE ja null tõeväärtusele FALSE. Näiteks:

```
if(a > b)          /* arvutab kahe arvu (a ja b) maksimumi */
  c = a;           /* ja omistab selle väärtuse muutujale c */
else
  c= b;
```

Tingimuslauset **if ... else** kasutatakse nii sageli, et selleks otstarbeks on loodud isegi omaette operaator **?:**. Näiteks:

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

Nii võtmesõnale **if** kui ka **else** tohib järgneda vaid üks lause. Suurem hulk lauseid tuleb ümbritseda looksulgudega. Näiteks:

```
if(pString1)       /* sama, mis if(pString1 != NULL) */
  {                /* juhul, kui viit pString1 osutab mingile
                      stringile */
  strcpy(pString2, pString1); /* siis kopeeri see jada
                                       puhvrisse, */
  pString2 += strlen(pString2);     /* millele osutab viida
                                       pString2 ja */
  }         /* nihuta viit pString2 uuesti puhvri lõppu */
```

Programmi teksti parema loetavuse huvides nihutatakse võtmesõnadele **if**, **else** ja muudele tingimuslausetele järgnevad laused kahe koha võrra paremale. Nii on lihtsam jälgida, millised laused tingimuse kehtides täidetakse ja millised mitte. Borland C/C++ editor aitab sellist teksti kujundamist. Kui te rea lõpus vajutate klahvile <ENTER>, siis nihutab editor kursori uuel real sama palju paremale, kui eelmisel real. Bloki lõpus vajutage klahvile <Backspace>. Nüüd nihutab editor kursori niipalju tagasi, kuipalju oli nihutatud paremale sellele blokile eelnev blokk.

Tingimuslause switch...case

Kui uuritav tingimus võib omada enam kui kahte väärtust, siis võite kasutada suuremat hulka **if...else** lauseid, näiteks:

```
if(nNum == 1) {          /* kui tingimuse väärtus on 1 */
...
}
else if(nNum == 2) {     /* kui tingimuse väärtus on 2 */
...
}
...                      /* kui tingimuse väärtus on ... */
else {                   /* kõigil muudel juhtudel */
...
}
```

Lihtsam ja ülevaatlikum oleks aga kasutada lauset **switch...case**. Nimetatud lause omab järgmist süntaksi:

```
switch(<tingumus>) {
  case <väärtus 1>:
     <laused>
     break;
  case <väärtus 2>:
     <laused>
     break;
    ...
  default:
     <laused>
}
```

Programm arvutab võtmesõnale **switch** järgneva avaldise väärtuse ja võrdleb seda kõigi võtmesõnadele **case** järgnevate väärtustega. Kui tingimuse väärtus langeb mõnega neist kokku, siis täidetakse sellele võtmesõnale **case** järgnevad laused. Kui üks neist lausetest on võtmesõna **break**, siis jätkatakse programmi täitmist lausetest, mis asuvad peale **switch** bloki lõppu. Kui tingimuse väärtus ei lange kokku ühegi võtmesõnadele **case** järgnevate väärtustega, siis täide-takse võtmesõnale **default** järgnevad laused. Kui lause **switch...case** ei sisalda võtmesõna **default** ja tingimuse väärtus ei langenud kokku ühegi võtme-sõnadele **case** järgneva väärtusega, siis jätkatakse programmi täitmist peale **switch...case** bloki lõppu. Kahe järjestikuse **case** bloki vahel peaks alati olema võtmesõna **break**. Kui see puudub, siis täidetakse peale esimese **case** bloki lausete täitmist ka teise **case** bloki laused, hoolimata sellest, et tingimuse väärtus ei lange kokku teisele võtmesõnale **case** järgneva väärtusega. Vahel kasutatakse seda lause **switch...case** omadust sarnast töötlemist vajavate tingimuste käsitlemisel. Näiteks:

```
...
char  ch;
...
switch(ch = getchar()) {  /* loe klaviatuurilt sisestatud sümbol */
  case 'q':               /* kasutaja sisestas sümbli 'Q' või 'q' */
  case 'Q':                   /* = Quit. Lõpeta programm */
    EndProgramm(...);
    break;
  case 'c':                   /* kasutaja soovis arvutada c =
Calculate */
  case 'C':                   /* sisestatud avaldise väärtuse vms. */
    Calculate(...);           /* arvuta uued väärtused ja
                                        esita nad ekraanile */
                              /* selle asemel, et seda siin teha,
                                        me "unustame" */
                              /* võtmesõna break ja laseme
                                        seda teha järgmisel blokil */
  case 's':
  case 'S':
    ShowResults(...);         /* väljasta tulemused ekraanile */
    break;
  ...                         /* muud käsud */
  default:                    /* tundmatu käsk */
    printf("\nTundmatu käsk!");
    printf("\nTuntud käsud on: \n");       /* näita tuntud käske */
    ...                             /* ja seleta, mis nad teevad */
}
...
```

Toodud näide demonstreerib kasutaja poolt klaviatuurilt sisestatud käskude lugemist ja täitmist. Nii võiks välja näha näiteks osa tabelarvutusprogrammist, kus peale valemite sisestamist tabeli lahtritesse võimaldatakse kasutajale mitmesuguste käskude abil nende valemite väärtusi arvutada vms. Iga käsu sisestamiseks peab kasutaja vajutama mingile klahvile. Programm loeb neid klaviatuurilt sisestatud sümboleid ja täidab neile vastavad käsud. Toodud näide täidaks vaid ühe käsu ja jätkaks peale seda programmi täitmisega peale **switch...case** lauset. Tegelikus programmis ümbritsetakse selline lause veel näiteks **while** tsükliga, mis sunniks programmi seda lauset korduvalt täitma ja uuesti käske lugema, kuni kasutaja on sisestanud mingi erilise käsu (näiteks 'Q' => Quit), mis lõpetab tsükli täitmise.

While tsükkel

While tsüklit kasutatakse mingi lause või lausete bloki korduvaks täitmiseks niikaua, kuni mingi tingimuse väärtus on nullist erinev (TRUE). **While** tsükkel võib esineda kahel kujul:

```
while(<tingimus>)
  <lause või blokk>
```

ja

```
do
  <lause või blokk>
while(<tingimus>)
```

Esimest tüüpi tsükkel kontrollib tingimuse väärtust ja täidab lauseid niikaua, kuni tingimuse väärtus on muutunud nulliks (FALSE). Igakord enne lause või lausete täitmist kontrollitakse tingimuse väärtust. Ka teist tüüpi tsükkel täidab lauseid seni, kuni tingimuse väärtus on muutunud nulliks, kuid siin kontrollitakse tingimuse väärtust alles peale lausete täitmist. Seega täidetakse teist tüüpi tsükli laused vähemalt korra ka siis, kui tingimus on otsekohe null. Mõlemat tüüpi tsükli puhul peab lausete hulgas olema üks lause, mis muudab tingimuse väärtust. Vastasel juhul ei muutu tingimus kunagi ja programm jääbki töötama. Sel juhul aitab ainult <RESET> nupule vajutamine või klahvikombinatsiooni <Ctrl>+<Alt>+<Del> kasutamine. Borland C/C++ keskkonnast starditud programmi saab peatada ka klahvide <Ctrl>+<Break> või <Ctrl>+<C> abil. Siis katkestab translaator programmi töö.

Näited:

```
char  ch;
...
do {
  Claculate(...);             /* arvuta uued väärtused */
  ShowResults(...);           /* esita nad ekraanile */
      /* küsi, kas kasutaja soovib veel kord arvutada */
  puts("Soovite te veel kord arvutada? (J/E)");
      /* aktsepteeri vaid klahvidele 'J' ja 'E' vajutamine käsuna */
  while(('J' != (ch = toupper(getchar()))) && ('E' != ch))
      /* muude klahvide puhul teata, millised käsud on olemas */
      /* ja ürita uuesti */
    puts("Jah v\0245i ei? (J/E)");
```

```
      /* täida tsüklit niikaua, kuni kasutaja
                 on vajutanud klahvile 'E' */
      /* kuid siiski vähemalt korra */
} while(ch != 'E');
```

Iga tsükkel võib sisaldada võtmesõnu **break** ja **continue**. Võtmesõna **break** katkestab tsükli täitmise ja jätkab programmi täitmist ridadest, mis paiknevad peale tsükli lõppu. Mitme üksteise sisse paigutatud tsükli puhul lõpetab võtmesõna **break** antud tsükli täitmise ja jätkab välimise tsükli täitmisega. Võtmesõna **continue** seevastu jätkab selle sama tsükli täitmisega tsükli algusest. Võtmesõna järel **continue** asetsevaid lauseid sel juhul ei täideta. Näiteks:

```
int   i, j;
...
i = j = 0;  /* nagu juba öeldud, on omistamine päris harilik tehe */
while(i < 10) {
  i++;
  if(i < 5)     /* niikaua, kui i on väiksem viiest, ära täida */
    continue;  /* muid tehteid vaid alga uuesti tsükli algusest */
  j = 0;
  while(1) {    /* lõputu tsükkel */
    j++;        /* suurenda j väärtust */
    if(j >= 5) /* kui j väärtus on suurem või võrdne viiega, siis */
      break;    /* katkesta tsükli töö
                         ja jätka välimise tsükli tööd */
  }
}
```

For tsükkel

Tsükleid **while** ja **do...while** kasutati ennekõike tingimuste puhul, kus korduste arv ei olnud teada. Seepärast tuli kas enne või peale lausete täitmist tingimuse väärtust kontrollida. Tsüklit **for** võib samuti kasutada tundmatu arvu kordustega tsükli täitmiseks, kuid tavaliselt kasutatakse teda tuntud arvuga korduste puhul. Tsükli **for** kuju on järgmine:

```
for(<valem1> ; <valem2>; <valem3>)
  <lause või lausete blokk>
```

Enne lause või lausete bloki täitmist täidetakse *valem1*. Seejärel täidetakse lauset või lausete bloki niikaua, kuni *valem2* väärtus on nullist erinev. Iga kord peale lause või lausete bloki täitmist arvutatakse *valem3* väärtus. Viimane peaks muutma *valem2* väärtust nii, et tsükkel omaks lõpliku arvu kordusi. Tavaliselt omistatakse valemis1 mingile muutujale algväärtus, mille väärtust *valemis2* kontrollitakse ja *valemis3* suurendatakse. Näiteks:

```
int   i;
...
for(i = 0; i < 5; i++)
  printf("Hello World!\n");
```

Toodud näide trükib ekraanile viis korda teksti "Hello World!". Muutuja i esimene väärtus on 0 ja viimane 4. Iga kord peale väljatrükki suurendatakse i

väärtust. Seega täidetakse tsüklit viis korda. Kindla arvu kordustega tsükleid täidab programm natuke kiiremini.
Tsükli **for** osad *valem1*, *valem2* ja *valem3* sisaldavad tavaliselt vaid ühe valemi, kuid nad võivad ka koosneda mitmest valemist, mis sel juhul eraldatakse komadega. Nimetatud osad võivad ka täiesti puududa. Ning lõpuks on **for** tsüklit võimalik kasutada ka määramata arvu korduste puhul. Näiteks:

```
for( ; ; )        /* lõputu tsükkel */
  printf("Mind ei saa katkestada!\n");

char  Buf[10], *pTmp1, *pTmp2;
char  name[] = "Toomas";
...
for(pTmp1 = name, pTmp2 = Buf; *pTmp1 ; *pTmp2++ = *pTmp1++);
```

Toodud näide kopeerib sümbolite jada *name* puhvrisse *Buf*. Selleks omistatakse *valemis1* kahele viidale algväärtused. *Valemis2* kontrollitakse esimese viida sisu. Niipea, kui see on null (C - keeles on null sümbolite jada lõpumärgiks), lõpetatakse tsükli täitmine. *Valemis3* kopeeritakse esimese viida poolt näidatud sümbol teise viida poolt määrtud kohta ja seejärel nihutatakse mõlemat viita ühe sammu võrra edasi. Siinkohal tuleks meenutada, et kui operaator ++ asus peale muutujat, siis sooritatakse tema tehe alles peale teisi tehteid. Seega täidetakse *valemis3* kõigepealt omistamine ja alles seejärel nihutatakse viitasid edasi. Kuna viidaoperaator * omas kõrgemat prioriteeti kui omistamine, siis omistatakse viitade poolt näidatud sümbolid, mitte viitade sisud (aadressid). Selles **for** tsüklis täidab *valem3* kõik vajaliku ja seega pole mingeid lauseid vaja. Kuna midagi peab sisestama, siis sisestame siia lihtsalt ühe semikooloni - tühja lause. See näide demonstreerib ka **for** tsükli kasutamist määramata arvu korduste puhul. Sümbolite jada "Toomas" sisaldab küll 6 sümbolit, kuid seda ju programm ei tea.

Käsk goto

Juba programmeerimiskeelest BASIC tuntud käsk **goto** on ka C - keeles olemas, kuid seda ei soovitata kasutada. See käsk muudab programmi teksti sageli raskesti loetavaks. Käsu **goto** kasutamiseks tuleb defineerida märk ja seejärel kasutada selle märgi nime käsuga **goto**.

```
goto <märgi nimi>
  <laused>
<märk>:
  <laused>
```

Märkide nimed on kehtivad vaid hetkelises blokis või funktsioonis. Ka käsuga **goto** saab luua tsükleid, näiteks:

```
int   i=0;
...
start:
```

```
  i++;
  if(i < 5)
    goto start;
```

Toodud näites jätkab käsk **goto** programmi täitmist märgist *start* niikaua, kuni muutuja *i* väärtus on väiksem viiest. Nagu juba öeldud, ei soovitata selliste tsüklite loomist. Kõiki käsuga **goto** loodud tsükleid jms. saab luua ka võtmesõnade **while**, **do...while**, **for**, **break** ja **continue** abil. Ainus koht, kus käsku **goto** vahel kasutatakse, on mitme üksteise sisse loodud tsükli seest "väljahüppamine" vea puhul. Ka funktsioonides, kus erinevate vigade puhul osaliselt järgneb sama vastus, võib käsu **goto** abil programmi teksti tunduvalt lühendada ja lihtsustada. Näiteks:

```
char* func1(int i, int j)
{     /* hangib i * j suuruse välja jaoks sobiva mälubloki */
  char            *pBuf;
  int       k, l;

  for(k = 0; k < i; i++) {
    for(l = 0; l < j; j++) {
      /* suurenda mälublokki */
      if(NULL == (pBuf = realloc(pBuf, k * l, sizeof(char))))
      /* kui mälu ei jätku, siis jätka märgist "error" */
        goto error;
    }
  }
  goto end; /* "hüppa" märgini end kui vigu ei ole */
error:             /* tehtud on mingi viga - mälu ei jätkunud */
  free pBuf;             /* vabasta kogu mälublokk ja */
  return NULL;           /* loovuta väärtus NULL teatamaks veast */
end:                     /* loovuta viit hangitud mälublokile */
  return pBuf;
}
```

Funktsioonide parameetrid

Programmeerimiskeele C funktsioonid näevad välja järgmiselt:

```
/* funktsiooni deklaratsioon ehk prototüüp */
<väärtuse tüüp> <nimi>(<parameetrite loetelu>);
...
/*funktsioonid definitsioon */
<väärtuse tüüp> <nimi>(<parameetrite loetelu>)
{
  <kohalike muutujate defineerimine>
  <laused>
  return <arvutatud väärtus>
}
...
/* funktsiooni kasutamine mingis muus funktsioonis */
...
<nimi>(<tegelikud parameetrid>);
...
```

Funktsioon luuakse selliselt, et see arvutaks mingi uue väärtuse, lähtudes mingitest üleantud parameetritest. Vastasel juhul tuleks luua eraldi funktsioon iga võimaliku parameetrite kombinatsiooni kohta. Muidugi on mõeldavad ka funktsioonid, mis mingit väärtust ei arvuta (näiteks ekraanile väljastamine) ja funktsioonid, mis mingeid parameetreid ei vaja. Andmevahetuseks välja kut-

sutava funktsiooni ja teda kasutava funktsiooni vahel võib kasutada funktsiooni parameetreid või globaalseid muutujaid. Globaalsete parameetrite kasutamine on küll kõige kiirem viis andmevahetuseks, sest siis ei ole vaja üleantavaid väärtusi uude kohta kopeerida; kuid see viis on ka kõige ohtlikum. Globaalsete muutujate kasutamine muudab programmi halvasti struktureerituks ja lihtsustab vigade teket. Selliste parameetrite kasutamine raskendab ka loodud funktsioonide kasutamist mingis teises programmis. Teise programmi looja (tõenäoliselt teine programmeerija) peab sel juhul teadma, milliseid muutujaid ta peab oma programmis defineerima nende funktsioonide jaoks. Ta ei tohi kasutada muid samanimelisi muutujaid jne.

Kõige parem on, kui funktsiooniga vahetatakse andmeid ainult üle tema parameetrite. Siis on funktsioon nagu kindla sisuga moodul, mis loovutab vastavalt parameetritele kindla väärtuse ja ei mõjuta mingil määral oma "ümbrust".

Funktsiooni parameetrid võivad olla väärtused (value parameters) või viidad (reference parameters). Väärtusparameetrite puhul kopeeritakse funktsioonile üle antud muutuja väärtus kohalikku muutujasse - parameetrisse. Funktsiooni sees saab seda väärtust kasutada nagu iga teistki kohalikku muutujat, ainult et talle ei saa midagi omistada. Väärtusparameetrite abil ei saa mingeid väärtusi väljakutsujale tagastada. Selleks tuleb kasutada kas osuteid või funktsiooni väärtust.

Ajutiste vahetulemuste salvestamiseks võite kasutada funktsiooni kohalikke muutujaid. Peale funktsiooni töö lõppu eemaldatakse aga need muutujad mälust. Te ei tohi kunagi loovutada funktsiooni väärtusena funktsiooni oma kohaliku muutuja aadressi, sest selleks ajaks, kui vastuvõtja seda aadressi kasutab, on tema sisu juba ammu üle kirjutatud. Funktsiooni väärtus on samuti väärtusparameeter. Temale üleantud kohaliku muutuja või valemi väärtus kopeeritakse väljakutsujale tagasi.

Viitade kasutamisel kopeeritakse parameetrisse mingi muutuja aadress. Funktsioon on sunnitud sellise parameetri tegeliku väärtuse kasutamiseks valemis lisama parameetri ette operaatori *. Samal kombel saab sellise muutuja väärtust ka muuta. Tehtud muudatus salvestatakse nüüd kohe antud muutujas. Seega saab viitparameetrite abil ka väärtusi väljakutsujale tagastada. Seda kasutatakse sageli siis, kui funktsioon peab tagastama enam kui ühe väärtuse ja seega on funktsiooni enda väärtusest vähe. Viitparameetrite kasutamine on soovitav ka funktsioonile suurema hulga andmete üleandmisel. Siis ei ole enam vaja neid andmeid kuhugi kopeerida. Väiksemate andmekoguste üleandmisel on aga soovitav kasutada väärtusparameetreid, sest nende kasutamine toimub kiiremini.

Näiteks:

```
int funk1(int a,        /* väärtusparameeter */
          int* b)       /* viitparameeter */
{
  int i;     /* kohalik muutuja. Tema väärtuse tagastamiseks tuleb */
             /* funktsiooni väärtust või mingit viitparameetrit */

  for(i = 0; i < a; i++)
    *b += i * 2;  /* iga viitparameetri sisuga tehtud muudatus on */
```

```
                    /* otsekohe üle antud */
  return i;         /* nii antakse üle kohaliku muutuja väärtus */
}
```

Osa funktsioone omab muutuvat parameetrite arvu. Näiteks funktsioon *printf()* võib omada üht või mitut parameetrit. Muutuva arvu parameetrite kasutamine on veel üks C - keele tugevaid külgi. Sel kombel saab luua väga paindlikke funktsioone, mida saab rakendada väga erinevates olukordades. Programmeerimiskeeles PASCAL ei ole muutuva arvu parameetrite kasutamine võimalik. See on tingitud parameetrite üleandmise järjekorrast. Selle asemel on funktsiooni väljakutsumine programmeerimiskeeles PASCAL natuke kiirem ja loodud programm on natuke väiksem. Borland C translaator suudab aga päris harilikus C - keele funktsioonis kasutada programmeerimiskeele PASCAL parameetrite üleandmise korda. Selleks tuleb vaid enne funktsiooni nime lisada võtmesõna **PASCAL**, näiteks:

```
int PASCAL funk1(int a, char b)
{
  ...       /* funktsioonis endas ei muutu midagi */
            /* kogu muutunud üleandmise korraldab translaator */
}
```

Võtmesõna **PASCAL** vastandiks on võtmesõna **cdecl**, mis määrab, et funktsioon kasutab C - keele parameetrite üleandmise korda. Microsoft Windowsi loomisel kasutati suuremas hulgas funktsioonides programmeerimiskeele PASCAL parameetrite üleandmise korda ja säästeti sellega kuuldavasti umbes 10% koodi. Tavaliselt ei ole siiski mõtet kasutada PASCAL parameetrite üleandmise korda. Eriti suurt kiiruse suurenemist sellega ei saavuta. Kõige efektiivsem viis programmi kiiruse tõstmiseks on ikka ainult parema algoritmi kasutamine.

Muutuva hulga parameetritega funktsioonid (võtmesõna **cdecl**) on sageli väga kasulikud. Sama tulemust saab sageli saavutada ka ühe konstantse hulga parameetritega funktsiooni korduva kasutamisega, kuid see ei ole nii efektiivne. Muutuva hulga parameetritega funktsiooni loomiseks tuleb viimase määratud parameetri järele sisestada parameetriteloetelusse kolm punkti - **...** . Näiteks:

```
int funk1(int i, ... ); /* üks täisarv ja tundmatu arv
                                 muid parameetreid */
```

Kuidagi tuleb funktsioonile ka teatada, kui palju ja millist tüüpi parameetreid talle üle anti. Muidu ei oska funktsioon neid parameetreid kasutada. Üks võimalus on kasutada formaadimäärangut nagu näiteks funktsioonis *printf()*. Sel juhul määrab esimene parameeter nii parameetrite hulga kui ka nende tüübi. Selline funktsioon on väga paindlik, kuid tema loomine on üsna keerukas. Nii kasutatakse veel kahte võimalust: esimene, kus parameetrite üldarvu määrab esimene parameeter ja teine, kus viimane parameeter omab erilist väärtust, näiteks nulli. Viimast võimalust kasutatakse palju operatsioonisüsteemi UNIX ühe graafilise kasutuskeskkonna *Open Look* funktsioonides. Millist võimalust

te ka ei kasutaks, tuleb "liigsete" parameetrite lugemiseks kasutada vastavaid C - keele makrosid.

```
void va_start(va_list ap, viimane_kindel);
<tüüp> va_arg(va_list ap, <tüüp>);
void va_end(va_list ap);
```

Makro *va_start()* seab viida *ap* esimesele "liigsele" parameetrile. Selleks, et ta teaks, millisest parameetrist alates parameetrite hulk võib muutuda, tuleb sellest makrole parameetri *viimane_kindel* abil teatada. Muutuva hulga parameetritega funktsioonid võivad sisaldada nii määratud kui määramatuid parameetreid. Määratud parameetrid peavad asuma kõik parameetrite loetelu alguses. Kui te näiteks defineerite funktsiooni

```
int funk1(int i, ...)
```

siis tuleks makrole *va_start()* loovutada teise parameetrina *i*. Nüüd näitab viit *ap* esimesele määramata parameetrile. Iga määramatu parameetri lugemiseks tuleb kasutada makrot *va_arg()*. Selle makro teine parameeter määrab, millist tüüpi muutujat hetkel loetakse ja makro väärtuseks on selle parameetri väärtus. See makro nihutab ka viida *ap* edasi järgmise määramatu parameetrini. Peale viimase määramatu parameetri lugemist tuleks kasutada makrot *va_end()*. Kui te seda ei tee, siis võib programmi käigus tekkida hulganiselt seletamatuid vigu. Translaator selle makro puudumist ei märka.
Näited:

```
      /* esimene määratud parameeter määrab parameetrite üldarvu */
int Korrutis(int n, ...)      /* korrutab n täisarvu */
{
  int             k = 0, i;
  va_list         ap;

  va_start(ap, n); /* sea viit esimesele määramatule parameetrile */
  for(i = 0; i < n; i++)      /* korruta parameetrid */
    k *= va_arg(ap, int);
  va_end(ap);
  return k;                   /* loovuta tulemus */
}
...
int   k;
...
k = Korrutis(3, 5, 6, 1);
...
      /* sama näide - viimane parameeter omab erilist väärtust */

int Korrutis(int a, ...)      /* korrutab n täisarvu */
{
  int             k, i;
  va_list         ap;

  va_start(ap, a); /* sea viit esimesele määramatule parameetrile */
  k = a;
  while(0 != (i = va_arg(ap, int))) /* korruta parameetrid */
    k *= i;
  va_end(ap);
  return k;                   /* loovuta tulemus */
}
```

Makrod ja sümboolsed konstandid

Eeltranslaatori käsuga *#define* saab defineerida nii sümboolseid konstante kui ka makrosid. Selle käsu üldine kuju on järgmine:

```
#define <konstandi nimi> <väärtus>
```

Kohates sellist käsku, asendab eeltranslaator kõikjal programmi tekstist leitud vastavanimelised konstandid toodud väärtusega. Siinkohal on tegemist vaid teksti asendamisega. Kuidas sellest tekstist aru saada, seda peab juba tegelik translaator teadma.
Sümboolseid konstnte kasutasime juba näiteprogrammis INTEGRAL.

```
17.   #define           MINPIIR           0.0
18.   #define           MAXPIIR           M_PI / 2.0
19.   #define           EPS               0.0001
```

Me oleksime võinud neid konstante ju ka otse programmi teksti sisestada, kuid nii on ülevaatlikum ja hiljem lihtsam nende konstantide väärtusi muuta. Märkida tuleb, et sümboolsete konstantide defineerimisel ei ole vaja lisada rea lõppu semikoolonit. Kui te seda siiski teete, siis käsitleb eeltranslaator seda semikoolonit kui makro osa ja sisestab ta igale poole teksti, kus te olete makrot kasutanud. See aga takistaks teil seda makrot mingis avaldises kasutamast, kuna lisatud semikoolon lõpetaks lause sellel kohal.
Makrosid defineeritakse samamoodi.

```
#define <makro nimi>(<parameetrite loetelu>) <asendav avaldis>
```

Makro on nagu tilluke funktsioon, mis erineb tegelikust funktsioonist selle poolest, et tema väljakutsumise puhul programmi käik ei muutu, vaid makro sisu lisatakse sellesse kohta programmi teksti. Kui te kasutatate makrot programmis mitu korda, siis lisatakse vastav kood ka sama palju kordi programmi teksti. Seepärast ei ole ka mõtet pikki makrosid luua. Nende mitmekordse kasutamise puhul läheks programm üsna suureks. Makro omab aga seda eelist, et tema kasutamine ei nõua alamprogrammi väljakutsumist koos parameetrite üleandmisega jne. ning on seetõttu kiirem. Lihtne makro näeks välja järgmiselt:

```
#define max(a, b) (a > b) ? a : b /* arvutab kahest arvust suurema */
#define min(a, b) (a < b) ? a : b /* arvutab kahest arvust väiksema*/
```

Makrod *min()* ja *max()* on juba defineeritud päisefailis STDLIB.H. Makro *max()* puhul asendatakse iga avaldis stiilis max(x, y) tekstiga (x > y) ? x : y. Makrode kasutamisel tuleb meeles pidada, et siin ei ole tegu funktsiooni kasutamisega, vaid teksti asendamisega. Kui *max()* oleks funktsioon, siis oleks valemis:

```
int   k, j = 0, i = 1;
...
k = max(i++, j);
```

tulemuseks 2 ja peale sellise funktsiooni kasutamist oleks muutuja i väärtuseks 1. Enne funktsiooni kasutamist suurendatakse ju muutuja i väärtust ühe võrra ja kuna 2 > 0, siis oleks tulemuseks 2. Kuna aga tegemist on teksti asendamisega, siis oleks peale asendamist avaldise välimus järgmine:

k = (i++ > j) ? i++ : j;

Nüüd täidetakse valemit i++ juba kaks korda ja seepärast on nii k kui ka i väärtusteks 3. Niisiis ei tohi (juhul kui te ei tea, kuidas see makro on implementeeritud) kasutada makro parameetritena avaldisi. Kui avaldis on nii pikk, et ta enam ühele reale ei mahu, siis tuleb otse enne klahvile <ENTER> vajutamist sisestada sümbol \. See sümbol (backslash) sunnib translaatorit ignoreerima järgnevat realõpumärki. Samal kombel saab ka teha üherealisest kommentaarist kaherealise.
Kui lisada makros parameetri nime ette sümbol #, siis tingib see parameetri väärtuse ilmumise asendatud tekstis jutumärkides. Näiteks:

```
#define TEXT(a) #a
...
printf("%s\n", "Tere hommikust" TEXT("Toomas"));
```

Tulemuseks oleks :

```
Tere hommikust "Toomas"
```

Defineeritud makro asendab kõigepealt teksti järgmiselt: "Tere hommikust" ""Toomas"". Iga järjestikku asuv jutumärkide paar defineerib ühe stringi. Praegusel hetkel oleks defineeritud kolm stringi ja nimi Toomas jääks üldse välja. Eeltranslaator on aga küllalt intelligentne lisamaks siia veel kaks sümbolit\. Seega on tulemuseks hoopiski: "Tere hommikust" "\"Toomas\"". Nüüd käsitletakse kahte jutumärki lihtsate sümbolitena ja stringi sel kohal ei lõpetata. Tulemuseks on aga ikkagi kaks stringi ja funktsiooni *printf()* formaadimäärangus oli määratud vaid üks. Kui nende kahe stringi vahel oleks koma, siis oleks tegu kahe erineva parameetriga ja viimast tõepoolest ekraanile ei ilmuks. Kuna neid aga ei ole komaga eraldatud, siis liidab translaator need kaks stringi üheks.
Lõpuks on võimalik makro parameetreid sümbolitega ## ühte liita. Operaator ## liidab kaks parameetrit üheks stringiks. Näiteks:

```
#define JADA(a, b)  a ## b
...
printf("Tulemus on: %d\n", JADA(2, 4));
...
Tulemus on: 24
```

Siin moodustati kahest arvust uus arv. Tegemist ei olnud aga mingi matemaatilise tehtega. Nii saab ka kahest stringist moodustada uue stringi, mis võib olla ka näiteks funktsiooni või makro nimi. Näiteks:

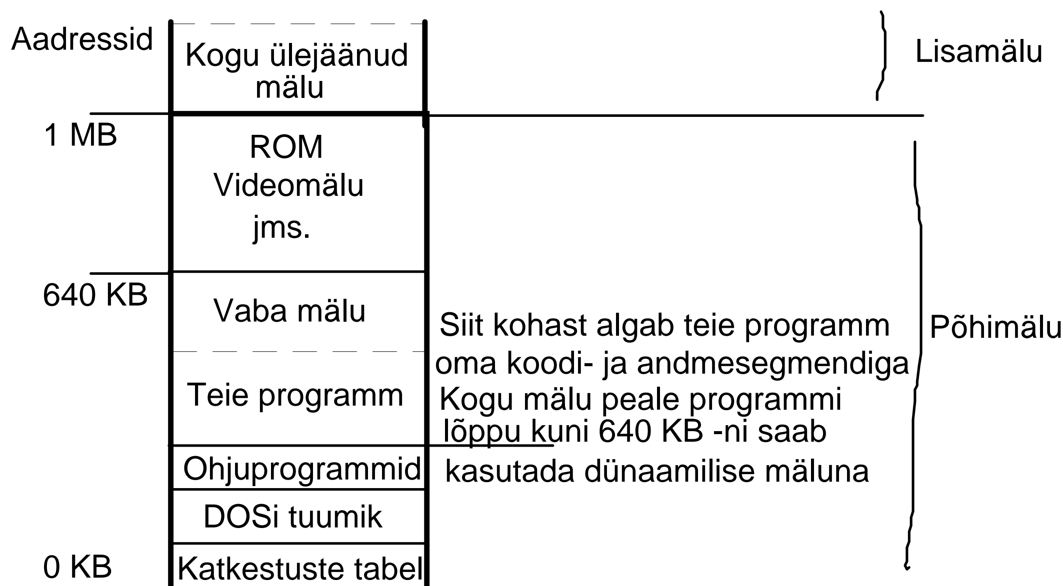
```
printf("Tulemus on: %d\n", JADA("mi", "n(2, 4)"));
...
```

```
Tulemus on 4
```

Seekord moodustab makro *JADA()* kahest üleantud stringist uue stringi: min(2, 4), mis on omakorda makro. Eeltranslaator asendab seejärel ka selle makro temale vastava avaldisega ja tulemuseks on, et 4 > 2.

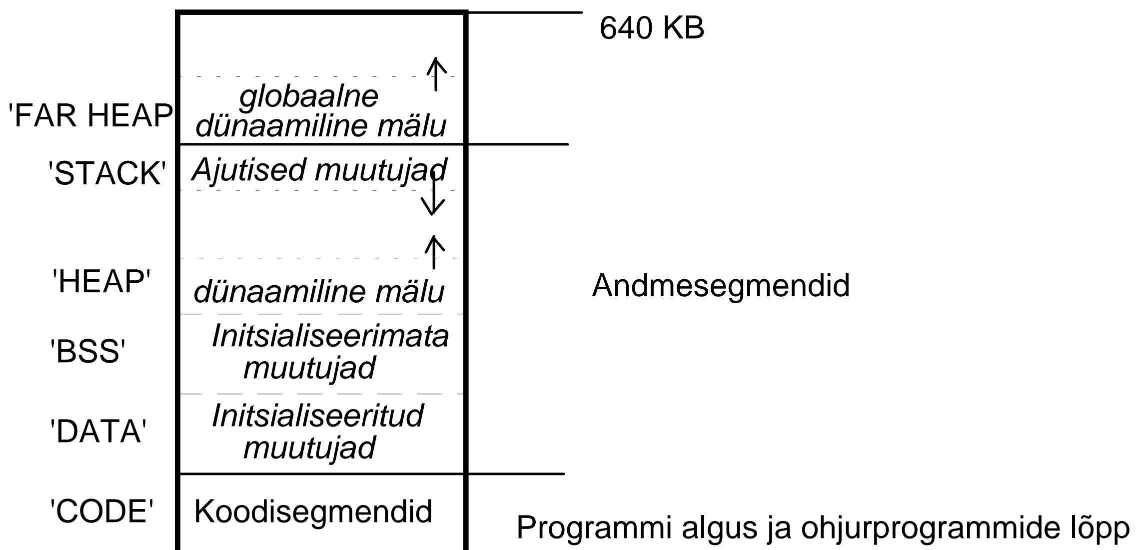
Mälu haldamine

Lihtsaim viis andmete salvestamiseks on defineerida nende jaoks sobivad muutujad. Kui te aga ei tea täpselt, kui palju andmeid teie programm vajab, siis ei saa te ka defineerida õiget hulka muutujaid. Kui te loote näiteks programmi, mis haldab teie tuttavate nimesid, aadresse ja telefoninumbreid ning salvestab vajalikud andmed mingisse faili, siis võite defineerida vastava andmestruktuuri ja oma andmesegmendis näiteks 5000 sellisest struktuurist koosneva massiivi. Tõenäoliselt ei ole teil rohkem kui 5000 tuttavat. Ikkagi on selline programm piiratud võimetega. Ta ei suuda mingil juhul lugeda rohkem kui 5000 sissekandest koosnevat andmefaili. Peale selle salvestatakse andmesegmendi sisu ka moodustatud *.EXE failis, nii on tulemuseks üsna suur *.EXE fail, mis oma suurust kuidagi ei õigusta. Kui te sellise programmi stardite, siis reserveerib ta suure hulga mälu, mida ta tegelikult ei vaja. Kuna enamus selle programmi andmefaile on tõenäoliselt tunduvalt väikesemad kui 5000 sissekannet, siis on suur osa mälust pidevalt asjatult reserveeritud. Operatsioonisüsteemis DOS ei ole see veel nii suur probleem, kuna te nagunii saate korraga kasutada ainult üht programmi. Kui te aga loote programmi mingi operatsioonisüsteemi jaoks, mis võimaldab mitmiktegumreshiimi nagu näiteks OS/2 või UNIX, siis peaks iga programm üritama süsteemi ressurssidega kokkuhoidlikult ümber käia, et korraga suudaksid töötada võimalikult palju programme.
Selleks võiks niisuguste andmestruktuuride jaoks reserveerida mälu dünaamiliselt. Operatsioonisüsteem DOS suudab otseselt kasutada vaid 1 MB mälu. See mälu jaotatakse järgmiselt:



Joonis 4: Mälu haldamine operatsioonisüsteemis DOS

Nagu näete, võib teie programm operatsioonisüsteemis DOS kasutada kogu mälu alates ohjurprogrammide lõpust kuni 640 KB piirini. Uuemates DOSi versioonides kasutatakse vabu mälublokke 640 KB ja 1 MB vahel ohjurprogrammide ja ühe osa DOSi tuumiku jaoks nii, et paremal juhul jääb teie programmile kasutada ligi 600 KB. Kuna DOS kasutab protsessorit reaalrezhiimis, siis ei suudeta adresseerida rohkem mälu kui 1MB. Selles rezhiimis kasutab protsessor 16-bitiseid aadresse. Selliselt vastab protsessori töö täpselt kunagi laialtlevinud i8086 protsessorile. Sellest ajast on pärit ka operatsioonisüsteem DOS koos oma piirangutega.
Programm koosneb ühest või enamast koodi- ja ühest või enamast andmesegmendist. Koodisegmendid sisaldavad masinkoodis programmi teksti: avaldisi, lauseid ja funktsioone. Andmesegmentidesse salvestatakse muutujate väärtused. Seejuures eristatakse initsialiseeritud (muutujad, millele omistatakse kohe mingi väärtus) ja initsialiseerimata muutujate vahel.



Joonis 5: Programmi segmentide jaotus

Joonisel 5 näete programmi segmentide jaotust operatsioonisüsteemis DOS. Joonise vasakul äärel näete nimesid, mida translaator omistab vastavatele segmentidele. Kui te ei soovi just otse assembleris programmeerida, siis ei oma need nimed suuremat tähtsust. Kohe peale ohjurprogrammide lõppu laaditakse programmi koodisegmendid. Seejärel tulevad andmesegmendid. Andmesegmendi alguses salvestatakse initsialiseeritud muutujad ja konstandid. Peale neid reserveeritakse ruumi initsialiseerimata muutujatele. Programmi *.EXE faili andmesegmendis reserveeritakse ruumi vaid muutujatele. Kui te aga laadite programmi mällu, siis suurendab operatsioonisüsteem andmesegmenti. Andmesegmendi lõppu luuakse piirkond ajutiste muutujate jaoks - pinu (stack). Kui te näiteks kutsute programmis välja mingi funktsiooni, siis salvestatakse tema kohalikud muutujad pinusse. Pinu "kasvab" andmesegmendi lõpust alguse poole. Kui väljakutsutud funktsioon kutsub omakorda välja mingeid funktsioone, siis salvestatakse nende kohalikud muutujad pinusse peale esimese funktsiooni muutujaid. Niipea, kui hetkel aktiivne funktsioon oma töö lõpetab, vabastatakse ka tema kohalike muutujate jaoks kasutatud mälu ja seega väheneb pinu suurus. Tema alumine piir nihkub ülespoole.

Dünaamilist mälu reserveeritakse kahes kohas: programmi andmesegmendis peale initsialiseerimata muutujate piirkonna lõppu (HEAP) ja peale programmi kõikide segmentide lõppu üldisest vabast mälust (FAR HEAP). Dünaamilise mälu reserveerimine programmi andmesegmendist on natuke kiirem. Selleks kasutatakse päisefailis ALLOC.H (või STDLIB.H) defineeritud funktsioone *malloc()*, *calloc()*, *realloc()* ja *free()*.

```
void* malloc(size_t size);
void* calloc(size_t nitems, size_t size);
void* realloc(void* pBuf, size_t size);
void free(void* pBuf);
```

Funktsioon *calloc()* reserveerib kohalikust andmesegmendist küllaldaselt mälu *nitems* elemendist koosnevale andmestruktuurile, mille iga element on *size*

baidi suurune. Uue mälubloki suurus on seega vähemalt *nitems* * *size* baiti. Tegelikult võib mälublokk olla veel natuke suuremgi. Selle põhjuseks on see, et mälu reserveeritakse 16-baidiste osade kaupa. Iga bloki suurus on seega 16-ga jaguv arv ja kui ta seda ei ole, siis seda suurendatakse natuke. Neid arvutusi ei pea te muidugi ise tegema. Funktsioonides kasutatud uus tüüp *size_t* on defineeritud kui täisarv ja seda kasutatakse mälublokkide suuruse määramiseks. Kui reserveerimine õnnestus, siis loovutab funktsioon oma väljakutsujale viida antud mälublokile. Selle viida abil saab nüüd mälubloki sisu muuta ja lugeda. Kui reserveerimine ebaõnnestus, siis on funktsiooni väärtuseks NULL. Seda väärtust peab ilmtingimata kontrollima, kuna nimetatud funktsiooni ebaõnnestmine on mälu vähesuse puhul küllaltki tõenäoline. Peale reserveerimist täidab funktsioon *calloc()* uue mälubloki sisu nullidega. Nii kindlustatakse, et programm ei vahetaks selles mälupiirkonnas juhuslikult olnud väärtusi enda poolt sisestatuga. Näide:

```
/* avab faili ja loeb sealt k sissekannet tüübist isik */
/* funktsiooni väärtuseks on viit mälublokile kui
      kõik õnnestus ja NULL vea puhul */
char* ReadFile(char *pFileName)
{
  int  k;
  struct isik *pIsikud;
...
      /* ava fail ja salvesta sissekannete arv muutujasse k */
...
      /* ürita reserveerida küllaldaselt mälu sissekannete jaoks */
      /* kui tulemuseks on NULL, siis lõpeta funktsiooni töö */
      /* saadud viit on tüübist void* ja tuleb enne omistamist */
      /* tüüpi struct isik * konverteerida */
      /* isikute arv on teada -> k, iga elemendi suuruse määrame */
      /* operaatoriga sizeof() */
  if(NULL == (pIsikud = (struct isik *)calloc(k, sizeof(isik))))
    return NULL;
      /* kõik on korras! Nüüd on aeg asuda antud blokki täitma */
      ...
  return pIsikud;
}
...
/* Nüüd võib kasutada mälublokki nagu päris tavalist massiivi */
int   i, k;       /* k väärtus tuleb enne välja arvutada */
char  pBuf[20];
...
for(i = 0; i < k; i++) {
  strcpy(pBuf, pIsikud[i].name);
  ...
}
```

Funktsioon *malloc()* reserveerib kohalikust mälusegmendist *size* baidi suuruse mälubloki, kuid ei täida tema sisu nullidega. Siin ei arvuta funktsioon bloki suurust ja seega tuleb argumendina sisestada sobiv avaldis. Vahel ei ole mälubloki nullidega täitmine ka vajalik. Näiteks siis, kui teil on vaja ajutiselt salvestada sümbolite jada, kus ei ole tegemist teatud hulga kindla suurusega andmestruktuuridega ja seega ei ole vaja ka funktsiooni abi mälubloki suuruse arvutamiseks. Vajalik suurus on sümbolite arv + 1 (lõpumärgi jaoks). Kui te kopeerite sümbolid reserveeritud mälublokki funktsiooniga *strcpy()*, siis lisab

nimetatud funktsioon ise jada lõppu nulli ja seega ei ole bloki initsialiseerimine lihtsalt vajalik.
Funktsioon *realloc()* muudab juba allokeeritud mälubloki suurust. Funktsiooni esimene parameeter on viit mälublokile, mille suurust tuleb muuta ja teine parameeter määrab tolle bloki uue suuruse. Funktsiooni väärtuseks on viit uuele blokile. Enamasti on see võrdne vana viidaga, kuid juhul kui blokki ei saanud suurendada, sest talle järgnev mälu oli juba mingi muu bloki poolt reserveeritud, siis reserveeritakse sobiva suurusega blokk uues kohas ja kopeeritakse vana bloki sisu uude blokki. Seega võib uus viit vanast erineda. Kui funktsioon ei suutnud mälu vähesuse tõttu blokki suurendada, siis on funktsiooni väärtuseks NULL. Sel juhul on vana viit veel kehtiv ja osutab vanale blokile, mis omab endist suurust.
Kord reserveeritud mälublokid tuleb kindlasti enne programmi lõppu mälust eemaldada. Selleks kasutatakse funktsiooni *free()*, millele antakse üle viit mälust eemaldatavale blokile. Peale bloki mälust eemaldamist ei tohi antud viita enam kasutada, enne kui talle on omistatud mingi teine väärtus, sest ta näitab veel piirkonnale, mis enam programmile ei kuulu. Kui te ei eemalda kõiki reserveeritud blokke mälust enne programmi lõppu, siis ei tee seda ka operatsioonisüsteem, sest ta ei tea, kellele need kuuluvad ja kas neid veel vajatakse. Nii tekivad mälus blokid, mida keegi ei kasuta ja mis takistavad järgmistel programmidel selle mälu kasutamist.
Kõik nimetatud funktsioonid reserveerivad mälu programmi kohalikust andmesegmendist. See on suhteliselt kiire, kuid sellised blokid ei tohi ületada suuruselt 64 KB. Seejuures tuleb meeles pidada, et seda mälu vajab ka programmi pinu. Kui pinu enam suureneda ei saa, kuna te olete reserveerinud kogu mälu andmesegmendist, siis lõpeb programmi töö veateatega. Kui te vajate väikest hulka mälu vaid ajutiselt, siis reserveerige see kohalikust andmesegmendist, sest see on kiirem, ja vabastage ta kohe, kui te teda enam ei vaja. Kui te aga peate salvestama suure hulga andmeid, siis tuleks reserveerida mälu üldisest vabast mälust (FAR HEAP). Selleks kasutatakse funktsioone:

```
void far* farcalloc(unsigned long nitems, unsigned long size);
void far* farmalloc(unsigned long nsize);
void far* farrealloc(void far* pBuf, unsigned long newsize);
void farfree(void far pBuf);
```

Need funktsioonid on sarnased eelpool toodud funktsioonidele, ainult et nad reserveerivad mälu üldisest vabast mälust ja reserveeritud blokid võivad olla suuremad kui 64 KB. Loovutatud viidad on nüüd pikad viidad (sisaldavad peale blokisisese aadressi ka segmendi aadressi) ja nende salvestamiseks tuleb kasutada ka sobivaid muutujaid. Kui te ei kasuta selliste funktsioonide juures pikki viitasid, vaid omistate saadud viida lihtsale lühikesele muutujale, siis salvestatakse vaid saadud viida viimane osa, s.o. segmendisisene aadress, segmendi aadress aga "unustatakse". Kui te nüüd hiljem seda viita kasutate, siis oletab programm, et lühike viit näitab programmi andmesegmendis asuvatele andmetele ja see tekitab vea.

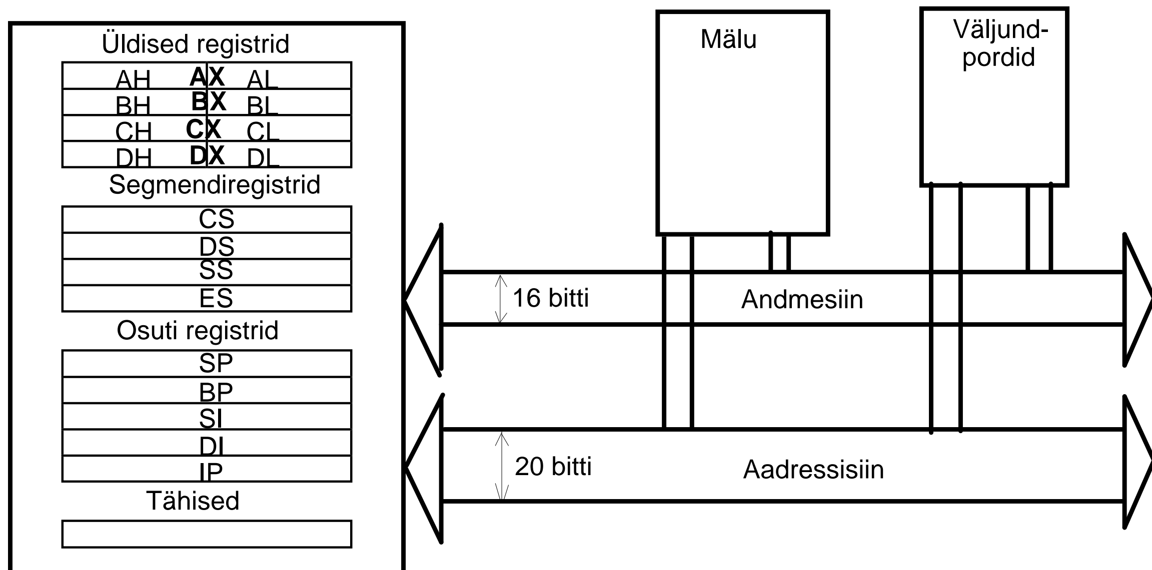
Teadmaks, kui palju te võite mälu reserveerida, kasutage funktsioone *coreleft()* või *farcoreleft()*.

```
      /* tiny, small ja medium mälumudeliga programmides */
unsigned coreleft( void );
      /* compact, large ja huge mälumudeliga programmides */
unsigned long coreleft( void );
      /* kasutatav kõikides programmides väljaarvatud tiny
mälumudeliga
            programmides */
unsigned long farcoreleft( void );
```

Funktsioon *coreleft()* teatab, kui palju mälu on programmi andmesegmendis veel vaba. Funktsioon *farcoreleft()* hangib samad andmed üldise vaba mälu kohta. Nagu näha, on funktsiooni *coreleft()* tulemus sõltuv programmi mälumudelist.

Mälumudelid ja protsessori ehitus

Operatsioonisüsteem DOS loodi algselt Intel 8086 tüüpi protsessoritega arvutitel (IBM XT ja temaga ühilduvad). Sellest on tingitud hulgaliselt Intel-protsessoritega arvutitel kasutatava C - keele eripärasid, mida nüüd seletame.
Protsessor i8086 (ja kõik temaga ühilduvad) omasid 14 registrit, 16-bitist andme- ja 20-bitist aadressisiini. Kõik registrid on 16-bitised. Need on nagu omamoodi protsessorisisesed mälupesad, kuhu saab salvestada väärtusi ja nendega opereerida. Nimetatud registrid jaotatakse omakorda üldisteks-, segmendi-, viida- ja tähiseregistriteks. Üldiseid registreid kasutatakse suuremas osas operatsioonides. Nende sisuga saab sooritada matemaatilisi-, loogilisi- ja muid tehteid. Segmendiregistrites salvestatakse segmentide algusaadresse. Viidaregistrid on vajalikud kahe segmendi vahel andmete vahetamisel ja paljudel muudel juhtudel. Tähisteregister jagatakse bittide kaupa eraldi tähisteks. Igaüks neist omab mingit omaette tähendust. Tähised näitavad matemaatiliste operatsioonide puhul tehtud vigadele, märgivad tingimuste tulemusi jne.



Joonis 6: Protsessori i8086 ehitus

Joonis 6 ei ole muidugi täiuslik. Tegelikult ei ole andmesiin ja aadressisiin ühendatud mälu ja väljundportidega otse, vaid üle siinikontrolleri. Peale seda omavad suurt tähtsust ka katkestustekontroller, kell ja muud arvuti osad.
Selleks, et mingi mälupesa sisu lugeda või muuta, tuleb aadressisiinile väljastada tema aadress ja kas lugeda andmesiinilt tema väärtus või väljastada tema uus väärtus. Kuna kõik registrid on 16-bitised, siis saab ühe registri sisuga adresseerida vaid 2^{16} = 65.535 = 64 KB mälu. Protsessor i8086 suudab aga kasutada kuni 1 MB mälu. Selleks jaotab ta mälu eraldi osadesse - segmentidesse. Ühte segmendiregistrisse salvestatakse nüüd soovitava segmendi algusaadress ja seejärel piisab ühestainsast registrist, määramaks aadressi segmendi piires (tema algusest lugedes). Lõplik aadress luuakse nii, et segmendi aadress (samuti 16-bitine arv) nihutatakse 4 biti võrra vasakule (sama, mis 16-ga korrutamine) ja talle liidetakse aadress segmendi piires (*offset*). Tulemuseks on 20-bitine arv, mis ongi mälupesa aadress. Kuna 2^{20} = 1 MB, siis suudab protsessor i8086 adresseerida kuni 1 MB mälu. Olgu näiteks soovitud mälupesa segmendi aadress A000 hex ja aadress segmendi piires 12D0 hex. Nihutades segmendi aadressi 4 biti võrra vasakule, saame A0000 hex ja liites sellele aadressi 12D0, saame lõpliku aadressina A12DO hex. Kuna segmendiregistrite sisu korrutatakse alati 16-ga enne liitmist segmendi siseseaadressiga, siis tuleneb sellest, et minimaalne segmendi suurus on 16 baiti. Tõsiasjast, et segmendisisene aadress peab mahtuma 16-bitisesse registrisse tuleneb aga, et segmendi maksimaalne suurus saab olla vaid 64 KB. Üks kilobait 1 KB = 1024 (ehk 2^{10}) baiti, mitte 1000 baiti.
Uuemad arvutid kasutavad protsessoreid i80286, i80386 ja i80486. Protsessor i80286 omab samuti 16-bitist andmesiini, kuid 24-bitist aadressisiini ja suudab seega adresseerida kuni 16 MB mälu. Protsessor i803886 ja i80486 omavad nii 32-bitist andme- kui ka aadressisiini ja suudavad seega adresseerida otseselt

(ilma segmenteerimiseta) kuni 4 GB (gigabaiti) ja segmentide kasutamisel kuni 64 TB (terabaiti) mälu. Peale selle on nende protsessorite registrid kõik 32 biti laiused. Operatsioonisüsteemis DOS töötavad aga ka need protsessorid reaal-rezhiimis, mis teeb nad täpselt sarnaseks oma eelkäija i8086-ga. Selles tööreshiimis ei suuda nad kasutada oma suuri andme- ja aadressisiine. Kuna selles raamatus käsitletakse vaid operatsioonisüsteemis DOS kasutatavaid C - keele variante, siis piirdume siin 8086 poolt loodud adresseerimisega. Protsessorite 80386 ja 80486 32 biti laiuseid registreid saab samuti kasutada vaid erirezhiimis. Nende nimedele lisatakse siis täht 'E' (*extended*), näiteks EAX, EBX. Reaalrezhiimis kasutatakse vanade tuntud nimede AX, BX all vaid nende registrite alumisi osasid (esimest 16 bitti) .
Adresseerimisel on vaja niisiis kahe registri koostööd. Ühes asub segmendi- ja teises segmendisisene aadress. Segmendiaadresside jaoks kasutatakse segmendiregistreid. Register CS (*code segment*) sisaldab hetkelise koodisegmendi aadressi ja register IP (*instruction pointer*) näitab selle segmendi sees järgmisele täidetavale masinkoodis käsule. Register DS (*data segment*) näitab programmi andmesegmendile. Andmesegmendist andmeid lugedes või sinna kirjutades võib kasutada segmendisisese aadressi jaoks suvalist üldist registrit või ka otsest aadressi masinkoodi osana. Register SS (*stack segment*) näitab programmi pinu sisaldavale segmendile. Tavaliselt on see programmi andmesegment. Pinusegmenti täidetakse ülevalt allapoole. Register SP (*stack pointer*) näitab pinusegmendi viimasele sissekandele (lõpust lugedes). Kui pinusse midagi sisestatakse, siis vähendatakse registri SP väärtust. Programmid kasutavad veel registrit BP (*base pointer*), osutamaks funktsiooni kohalike muutujate algusesse pinul. Selle registri abil loetakse funktsioonide kohalikke muutujaid. Registrit ES (*extra segment*) kasutatakse tavaliselt andmete transporteerimiseks mingist muust segmendist programmi andmesegmenti. Seejuures osutab register SI (*source index*) kopeeritavate andmete algusele ja register DI (*destination index*) andmete jaoks ettenähtud puhvrile. Protsessor 8086 tunneb paari operatsiooni, mis transporteerivad määratud arvu baite nõutud kohast teise ja seejuures ka iseseisvalt muudavad registreid DI ja SI. Transporteeritavate baitide arv salvestatakse seejuures registrisse CX (*count*).
Üldised registrid AX, BX, CX ja DX lubavad kasutada oma ülemisi ja alumisi 8-bitiseid osasid ka eraldi. Nende nimed on AH (*high*) ja AL (*low*) ning vastavalt BH, BL jne. Need 8-bitised registrid on pärit veel protsessori 8086 eelkäijalt, mis oli 8-bitine protsessor. Selleks, et lihtsustada tolle protsessori jaoks loodud tarkvara ülekandmist protsessorile 8086, oli vaja võimaldada nonde 8-bitiste registrite kasutamist.
Programmeerimiskeel C tunneb kolme tüüpi viitasid: *near*, *far* ja *huge*. Lühikesed viidad (*near*) on 16-bitised väärtused ja sisaldavad vaid segmendisisese aadressi. Neid on lihtne kasutada ja nende kasutamine on ka kiirem. Kui *near* viit näitab mingile andmestruktuurile, siis oletatakse, et see andmestruktuur asub programmi oma andmesegmendis ja segmendiregistrina kasutatakse registrit DS. Nii on see alati programmi staatiliste (globaalsete) muutujatega ja

programmi andmesegmendist reserveeritud mälublokkidega. Kui *near* viit näitab mingile funktsioonile, siis kasutatakse registrit CS.
Pikad (*far*) viidad on 32-bitised muutujad. Nad sisaldavad nii segmendi-, kui ka segmendisisese aadressi. Nende kasutamine on natuke aeglasem, kuna see nõuab segmendiregistrite CS või DS sisu muutmist. Mälu reserveerimisel üldisest mälust (FAR HEAP) on nad aga ilmtingimata vajalikud.
Viimane tüüp (*huge*) märgib viitasid, mis samuti nagu *far* viidad sisaldavad nii segmendi-, kui ka segmendisisese aadressi. Nende viitade nihutamine kontrollib aga programmi segmendipiire. Tavaline pikk viit võib küll näidata teise segmenti, kuid niipea kui teda nihutatakse üle tolle segmendi piiri, muutub vaid viida segmendisisene osa ja seega näitab viit uuesti tolle segmendi algusesse. Kui te soovite kasutada suuremaid mälublokke kui 64 KB, siis peate kasutama *huge* viitasid. Nende viitade nihutamisel üle 64 KB piiri suurendadatse ka viida segmendiaadressi määravat osa. Teine probleem pikkade viitadega (*far*) on see, et kaks *far* viita võivad näidata samasse mälupunkti, kuid omada erinevaid segmendi- ja segmendisiseseid aadresse. Seega ei saa neid osuteid võrrelda loogiliste tehetega. Nimetatud *huge* tüüpi viitade sisu on aga alati normaliseeritud, s.o. see on viidud teatud standardsele kujule. Seepärast omavad kaks samasse mälupunkti osutavat *huge* viita alati sama väärtust.
Viidad tüübist *huge* nõuavad aga erilist programmiloogikat, mis nende sisu pidevalt normaliseerib ja segmendipiiride ületamisel õige aadressi arvutab. Seda kõike teeb translaator, kuid need "üleliigsed" operatsioonid teevad *huge* viitade kasutamise natuke aeglasemaks.
Milliseid viitasid te kasutate, sõltub valitud mälumudelist. Iga mälumudel määrab teatud kindlad viidapikkused andmete ja koodi jaoks. Te võite aga alati ise mingi viida tüübi määrata, kasutades võtmesõnu: **near**, **far** ja **huge**. Näiteks:

```
char  near  *pTmp1;
char  far   *pTmp2;
char  huge  *pTmp3;
```

Mälumudelid

Kõige väikesem mälumudel on *tiny*. Selle mälumudeli puhul eksisteerib vaid üksainus (kuni 64 KB suurune) segment nii programmi koodi, andmete kui ka pinu jaoks. See mälumudel vastab täpselt DOSi *.COM failide ehitusele. Sellises programmis saab kasutada otsest adresseerimist, s.o. aadress võib olla otseselt masinkoodi osa. Kuna aadress omab vaid segmendisisest osa, siis ei ole aadresside ümberarvutamine programmi alguses (*relocation*) vajalik. Seepärast töötavad *.COM failid tunduvalt kiiremini kui teised programmid.
Kõik muud mälumudelid vastavad *.EXE failidele. Neist väikseim on *small* mälumudel. Mälumudeli *small* puhul omab programm täpselt ühe (kuni 64 KB suuruse) andme- ja ühe samasuure koodisegmendi. Programmi pinu asub tema andmesegmendis. Kuna programm koosneb nüüd vähemalt kahest segmendist, siis tekib programmi mällu laadimisel probleem. Vastavalt sellele, kui palju

ohjurprogramme on laaditud, võidakse programm laadida mälus erinevatesse kohtadesse. Ka *small* mälumudeli puhul ei ole see suurem probleem, kuna te kasutate tavaliselt lühikesi viitasid. Kui te aga kasutaksite mingi muutuja jaoks pikka viita, siis peaks tema aadress koodisegmendis sisaldama ju ka segmendi aadressi. Seda me aga ei tea, sest nagu juba öeldud, laaditakse programm iga kord mälu erinevasse kohta ja seega on segmendiaadressid alati erinevad. Seepärast täidab translaator *.EXE faili pikkade viitade segmendiosad nullidega. Faili algusesse luuakse aga nn. viitade tabel. Selles tabelis on märgitud kõikide pikkade viitade aadressid failis. Kui nüüd operatsioonisüsteem loeb sellise programmi mällu, siis omistab ta registritele mingid kindlad väärtused. Seejärel uurib ta programmi viitade tabelit ja sisestab kõigi pikkade viitade segmendiossa õiged väärtused. Alles nüüd alustatakse programmi täitmist. Seda protsessi nimetatakse aadresside ümberarvutamiseks (*relocation*) ja selletõttu on *.EXE failide startimine natuke aeglasem. Kui nii võtta, siis on aga iga uus programmeerimistehnika seotud osa töö üleandmisega translaatorile ja teeb seega programmi aeglasemaks. Tänapäeva arvutid on aga küllalt kiired selliste tööde jaoks. Kõige kiirema programmi saaks luua kahtlemata kogu programmi käsitsi masinkoodis kirjutades. Paraku vajab vähegi kasutajasõbraliku programmi käsitsi masinkoodis (või assembleris) kirjutamine nii palju aega ja vaeva, et see lihtsalt ei tasu ära. Peale selle on nii palju tõenäolisem mitmesuguste vigade tekkimine kui translaatori kasutamisel.

Mälumudeli *medium* puhul kasutab programm kõigi funktsioonidele näitavate viitade puhul pikki (*far*) ja andmete puhul lühikesi (*near*) viitasid. Seega võib mälumudelit *medium* kasutav programm sisaldada kuni 64 KB andmeid ja kuni 1 MB koodi. Selline mudel sobib suurtele keerukatele programmidele, mis aga vajavad vähe andmeid.

Mälumudel *compact* on mudeli *medium* täpne vastand. Siin kasutatakse andmete jaoks pikki ja koodi jaoks lühikesi viitasid. Sellist mälumudelit kasutav programm võib kasutada kuni 64 KB koodi (üks segment) ja kuni 1MB andmeid (rohkem segmente). Mälumudel *compact* sobib suhteliselt väikestele programmidele, mis aga töötlevad suuri andmekoguseid.

Mälumudel *large* kasutab nii koodi, kui ka andmete jaoks pikki (*far*) viitasid. Programm võib omada hulgaliselt koodi ja andmesegmente. See võimaldab kasutada nii koodi kui ka andmete jaoks kuni 1 MB mälu.

Suurim mälumudel on *huge*. See mälumudel kasutab nii koodi kui ka andmete jaoks *huge* viitasid. Tema põhiline erinevus mudelist *large* on see, et *large* mälumudeliga programmid võivad küll omada enam andmesegmente, kuid nad tohivad salvestada oma globaalsed (*static*) muutujad vaid ühes neist. Seega on globaalsete muutujate maht piiratud 64 KB-ga. *Huge* mälumudeliga programmide puhul seda piirangut ei ole.

Alltoodud tabelis näete üldistatud kokkuvõtet C - keele mälumudelitest.

Mälumudel	Andmed		Kood	
	Viidad	Maht	Viidad	Maht
Tiny	near	kuni 64 KB	near	kuni 64 KB

Small	near	kuni 64 KB	near	kuni 64 KB
Medium	near	kuni 64 KB	far	kuni 1 MB
Compact	far	kuni 1 MB	near	kuni 64 KB
Large	far	kuni 1 MB	far	kuni 1 MB
Huge	huge	kuni 1 MB	huge	kuni 1 MB

Tabel 10: Programmeerimiskeele C mälumudelid

Programmeerimiskeeles C on võimalik kasutada ühes programmis korraga ka erinevaid mälumudeleid. Selline olukord võib tekkida, kui te soovite oma programmis kasutada mingi muu mälumudeliga kompileeritud objektfaili või teeki. Probleemi lahendamiseks tuleb oma programmi sisestada eraldi deklaratsioon iga taolises objektfailis või teegis defineeritud funktsiooni ja muutuja jaoks. Selles deklaratsioonis tuleb võtmesõnade **near**, **far** ja **huge** abil määrata tolle funktsiooni tüüp sõltuvalt tema mälumudelist. Näiteks:

```
/* Teek.C Selles failis defineerime paar funktsiooni, mida hiljem  */
/* soovime kasutada põhiprogrammis. Selle faili transleerime aga  */
/* mälumudeli LARGE abil. Kõik funktsioonid ja andmed vajavad */
/* pikki viitasid */
      /* sümbolite väli. Kuna kasutatakse mälumudelit LARGE, */
      /* siis on tema nimi tegelikult pikk viit */
char  szName[10] = "Peeter";
      /* lihtne funktsioon. Kuna kasutatakse mälumudelit LARGE, */
      /* siis on tema nimi tegelikult pikk viit */
int Func1(char * a, int b)
{
   ....           /* tee midagi */
  return 2;
}
```

```
/* Program.C See on põhiprogramm. Selle faili transleerime mälu- */
/* mudeli SMALL abil. Seega vajavad kõik andmed ja funktsioonid   */
/* vaid lühikesi viitasid */
      /* loome vajalikele muutujatele ja funktsioonidele */
      /* sobivad deklaratsioonid vastavad kasutatud mälumudelile */
extern far *szName;
extern int far Func1(char far * a, int b);
      /* nüüd kasutame neid */
void main( void )
{
  char            szBuf[10];
  int             i;

  fstrcpy((char far*)szBuf, szName);
  i = Func1((char far*)szBuf, 3);
...
}
```

Programmi laadimine

Enamus PC - arvuteid omavad juba 80286, 80386 või 80486 protsessoreid. Enamasti on neil mälu ka tunduvalt rohkem kui 1 MB. Operatsioonisüsteemi DOS tõttu ei suuda nad seda aga kasutada. Lisanduv mälu võib olla kas lisa-mälu või laiendatud mälu.

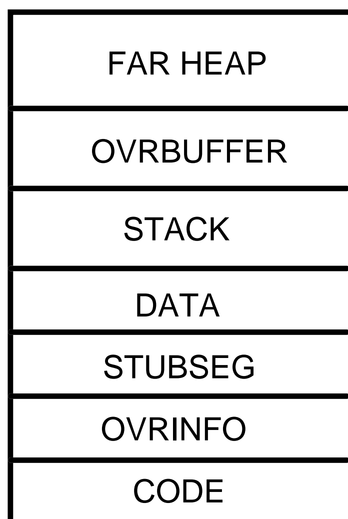
Lisamälu (*extended memory*) saab kasutada vaid eespoolnimetatud protsessoritega arvutitel. See mälu algab 1 MB piirist ja jätkub vastavalt protsessori võimalustele kuni 16 MB või 4 GB - ni. Operatsioonisüsteemide OS/2 ja UNIX all saab seda mälu ka vabalt kasutada ning mingit piiri 1 MB kohal ei eksisteeri. Operatsioonisüsteemis DOS on aga lisamälu kasutamiseks vaja protsessor viia enne töörezhiimi *protected mode*. Seda on võimalik teha ühe katkestuse abil. Kuna aga DOS selles töörezhiimis ei tööta, siis peab programm nüüd ise mälu haldamisega tegelema. Ka muid DOSi teenuseid, nagu näiteks failide lugemine vms., ei saa selles rezhiimis kasutada. Seega saaks DOSi programm kasutada laiendatud mälu vaid oma andmete jaoks. Programm peaks mäluvähesuse puhul viima protsessori töörezhiimi *protected mode*, kopeerima hetkel ebavajalikud andmed mingisse segmenti laiendatud mälus ja seepeale lülitama protsessori kohe tagasi reaalrezhiimi. Nüüd võib kopeeritud andmebloki mälust eemaldada ja seda mälu muuks otstarbeks kasutada. Kõik see on üsna keeruline ja nõuab põhjalikke teadmisi operatsioonisüsteemidest ja protsessori ehitusest. Selle probleemi lahendamiseks on loodud hulgaliselt spetsiaalseid teeke ja ohjurprogramme, mis sisaldavad vajalikke funktsioone ja lihtsustavad tunduvalt lisamälu tarbivate programmide loomist. Siinkohal tuleks nimetada näiteks *Phar Lap DOS Extender* teeki, kuid unustada ei tohiks ka vabalt kopeeritava tarkvarana (*Public Domain*) pakutavat *EMX* teeki. Vabalt kopeeritava tarkvarana on saada ka *GNU C/C++* translaator, *GDB* silur, hulgaliselt muid kasulikke teeke ja *Emacs* editor.

Laiendatud (*expanded*) mälu loodi 8086 tüüpi protsessoritega, s.t. IBM XT-ga ühilduvate arvutite jaoks. Laiendatud mälu asub väljaspoolt 1 MB piiri. XT tüüpi arvutitel tuleb kasutada lisakaarte laiendatud mäluga, mis sisaldavad ka selle mälu kasutamiseks spetsiaalset kontrollerit. Arvutitel 80286 ja paremate protsessoritega kasutatakse tavaliselt arvuti lisamälu koos spetsiaalse ohjurprogrammiga, mis võimaldab kasutada seda mälu nagu laiendatud mälu. Siiski on ka neil arvutitel võimalik kasutada lisakaarte lisamäluga. Igal juhul luuakse lisamälu kasutamiseks nn. "aken" - s.o. mälublokk programmi poolt adresseeritavas piirkonnas, mis aga tegelikult vastab mälule lisakaardil või ülalpool 1 MB piiri. Sellesse "aknale" seatakse programmi soovil lisakaardi kontrolleri või ohjurprogrammi abiga vastavusse mingi mälublokk laiendatud mälust. Iga kord, kui te sellesse mälublokki midagi kirjutate või sealt loete, muudab lisakaardi kontroller või ohjurprogramm seda operatsiooni, nii et tegelikult kirjutatakse või loetakse need andmed hoopiski lisamälu vastavast blokist. Programmist võib spetsiaalsete funktsioonide abil vahetada "aknale" vastavusse seatud lismälu blokki ja sooritada lisamäluga muid operatsioone.

Kolmas võimalus suurte programmide loomiseks, mis 640 KB piire enam ei mahu, on jagada programm segmentidesse (*overlays*) ja laadida neid siis mällu vastavalt vajadusele. Selline programm ei asu kunagi mälus tervikuna. Tavaliselt jagatakse sellised programmid osadesse (*units*), millest igaüks salvestatakse omaette *.OVL faili . Põhiprogramm jääb ikkagi *.EXE faili. Nimetatud *.EXE fail peab sisaldama vähemalt programmi *main()* funktsiooni. Lisaks sellele lisab linker *.EXE faili veel programmi ülejäänud osade (*units*)

mällu laadimiseks vajaliku programmi koodi. Kui nüüd selline programm starditakse, siis laaditakse mällu kõigepealt tema *.EXE fail ja reserveeritakse küllaldaselt suur puhver suurima *.OVL faili jaoks. Niipea kui programm kutsub välja mingi *.OVL faili salvestatud funktsiooni, loeatakse see *.OVL fail nimetatud puhvrisse. Sellel tehnikal on üks viga: *.OVL failis salvestatud funktsioon saab välja kutsuda vaid põhiprogrammis või samas *.OVL failis salvestatud funktsioone. Kui ta üritaks kutsuda välja mingis muus *.OVL failis salvestatud funktsiooni, siis tuleks enne mälust eemaldada tema oma *.OVL fail, et saaks mällu laadida uut *.OVL faili.
Borland C/C++ translaator sisaldab spetsiaalset programmiosade haldus-loogikat: *VROOM* ehk *Virtual Run-time Object-Orientated Memory Manager*. Kui te kasutate VROOM abi, siis ei pea te enam pikalt mõtlema, millisesse programmi ossa te ühe või teise funktsiooni salvestate. VROOM lihtsalt ei eemalda programmi osa mälust, vaid salvestab ta kõvakettale ajutisse faili. Nii saab ühest programmi osast (VROOM all nimetatakse neid segmentideks - *segments*) ka teises programmi osas salvestatud funktsioone välja kutsuda. Kui teie arvuti omab lisa- või laiendatud mälu, siis kasutab VROOM ajutise faili loomise asemel seda mälu programmiosade ajutiseks salvestamiseks. See muidugi tõstab tunduvalt programmi töökiirust.
Ka VROOM -i kasutamise puhul jäävad kehtima mõned piirangud. Programmi *main()* funktsioon ei tohi asuda *.OVL failis. Sama kehtib iga funktsiooni kohta, mida kasutatakse katkestustele vastamiseks. Katkestuste kasutamist käsitletakse järgmistes peatükkides. Üldiselt tuleks iga funktsioon, millelt oodatakse suurt töökiirust, salvestada põhiprogrammi. Eraldi *.OVL failidesse tuleks aga salvestada funktsioonid, mida vajatakse vaid puhuti, nagu näiteks trükkimisega seotud funktsioonid. Ka siin tuleks üritada vältida teistes programmiosades asuvate funktsioonide kasutamist mingis *.OVL failis salvestatud funktsiooni poolt. See tähendaks ju mingi programmiosa lugemist kõvakettalt, mis on muidugi aeglane. Üldiselt tuleks programmiosad luua nii, et iga osa tegeleks mingi kindla ülesannete kompleksiga nagu näiteks trükkimine. Kui programm soovib väljastada mingeid andmeid trükkalile, siis ta kutsub välja vastava funktsiooni trükkimisega tegelevast programmiosast. Kõik abifunktsioonid, mida too funktsioon vajab, peaksid asuma samas *.OVL failis, et ei oleks vaja pidevalt midagi kõvakettalt lugeda. Niipea kui trükkimine on lõpetatud, eemaldatakse see programmiosa mälust ja vabanenud mälu kasutatakse muuks otstarbeks.
VROOM võib luua programmiosade jaoks natuke suurema puhvri kui seda on programmi suurim osa ja seega lugeda korraga mällu rohkem kui ühe segmendi. Selle puhvri suurust saab reguleerida globaalse muutuja `_ovrbuffer` väärtuse muutmisega. Nii saate reguleerida oma programmi töökiirust programmi töö ajal. Mida suurem puhver, seda rohkem segmente saab korraga mällu laadida ja seda vähem peab segmente salvestama ajutiselt kõvakettale või lisamällu. Kui te suurendate nimetatud puhvrit aga vaid teatud hulga kilobaitide võrra, nii et sellest veel ei jätku rohkemate segmentide laadimiseks,

siis see teie programmi tööd ei kiirenda. Selle puhvri suurusega tuleb niisiis lihtsalt katsetada.
VROOM vajab aga lisanduvaid programmisegmente, mida näete järgmisel joonisel.



Joonis 7: VROOMi jaoks vajalikud lisanduvad programmisegmendid

Linker lisab programmile segmendid *OVRINFO* ja *STUBSEG*. Esimene neist sisaldab andmeid, mida VROOM vajab programmiosade haldamiseks. Teises segmendis - *STUBSEG* on iga segmendi jaoks eraldi kirjas teda puudutavad andmed. Kolmanda segmendi *OVRBUFFER* suurust saab, nagu juba öeldud, muuta globaalse muutujaga `_ovrbuffer`.
Programmi jagamiseks osadeks salvestage iga osa jaoks määratud funktsioonid eraldi *.C faili. Avage menüüst ***O**ptions* / ***C**ompiler* / ***C**ode Generation* dialoog *Code Generation* ja valige sobiva lülitiga programmi osadeks jagamine (overlay support). Seejärel valige projektiaknas iga eraldi ossa salvestatav *.C fail ning avage tema valikute dialoog klahvide kombinatsiooniga <Ctrl>+<O>. Selles dialoogis vajutage lülitile *Overlay this module*.
Borland C/C++ translaator ei salvesta programmi eri osi eraldi *.OVL faili, vaid loob neist ühe *.EXE faili, mille osi siis järjekorras mällu laaditakse. Osadeks jaotatav programm peab kasutama mälumudeleid: *compact*, *large* või *huge*. Programmid mälumudelitega *tiny, small* ja *medium* ei saa kasutada VROOM teenuseid.

OSAPROG.C

```
/*****************************************************************/
/***                                                          ***/
/***   OsaProg.C                                              ***/
/***                                                          ***/
/***   Demonstreerib programmi jaotamist osadeks, mida  saab    ***/
/***   VROOM abil vastavalt vajadusele mällu laadida.         ***/
/***   See fail sisaldab programmi põhifaili.                 ***/
```

```
/***                                                            ***/
/******************************************************************/

#include <stdio.h>

      /* programmi osades asuvad funktsioonid */
extern void osafunc1( void );
extern void osafunc2( void );

int main()
{
  printf("\nPoehiprogramm alustab tööd");
  /* kasutame esimese osa funktsioone */
  osafunc1();
  /* kasutame teise osa funktsioone */
  osafunc2();
  printf("\nProgrammi töö on tehtud");
  return 0;
}
```

OSA1.C

```
/******************************************************************/
/***                                                            ***/
/***  Osa1.C                                                    ***/
/***                                                            ***/
/***  See fail sisaldab programmi OsaProg.EXE esimese osa       ***/
/***  funktsioone.                                              ***/
/***                                                            ***/
/******************************************************************/

#include <stdio.h>

void osafunc1 ( void )
{
  printf("\nProgramm töötab oma esimeses osas");
}
```

OSA2.C

```
/******************************************************************/
/***                                                            ***/
/***  Osa2.C                                                    ***/
/***                                                            ***/
/***  See fail sisaldab programmi OsaProg.EXE esimese osa       ***/
/***  funktsioone.                                              ***/
/***                                                            ***/
/******************************************************************/

#include <stdio.h>

void osafunc2 ( void )
{
  printf("\nProgramm töötab oma teises osas");
}
```

Assembleri kasutamine

Assemblerit kasutatakse sageli keerukate probleemide jaoks, mida ei saa kõrgemas programmeerimiskeeles lahendada. Selliste probleemide hulka kuuluvad tavaliselt katkestustega tegelevad funktsioonid jms. Ka suurimat töökiirust nõudvates funktsioonides on sageli vaja kasutada assemblerit.
Tavaliselt kirjutatakse vaid osa programmist assembleris. Assemblerkoodi sidumiseks C - keele programmiga luuakse siis üks terve funktsioon assembleris ja kasutatakse seda C - keele programmis nagu harilikku funktsiooni. Selleks tuleb teada, kuidas C - keeles funktsioonile parameetreid üle antakse.
Borland C/C++ translaator lubab teil kasutada assemblerkoodi ka otse C - keele failis. Selleks on vaja kasutada vaid võtmesõna **asm**. Sellised assemblerkäsud võivad omada järgmist kuju:

```
asm <assembleri käsk>
asm <assembleri käsk> ;
asm <assembleri käsk> ; asm <assembleri käsk>
asm {
...
  <assembleri käsk>
  <assembleri käsk>
...
}
```

Näiteks:

```
asm  MOV  AX, 6
asm  mov  bx, ax;  int 21h
asm {
  push ds
  mov ax, [bp+8]
  inc ax
}
```

Assembleris märgib semikoolon kommentaari algust. Kui te aga sisestate C - keele faili assemblerit, siis märgib semikoolon ühe assemblerkäsu lõppu ja võimaldab teil seega sisestada ühele reale mitu assemblerkäsku.
Looksulud koos võtmesõnaga **asm** määravad ühe assemblerkeele bloki. Sellise bloki sees ei ole võtmesõna **asm** kasutamine iga rea alguses enam vajalik.
C - keele faili sisestatud assemblerkäskudes võite kasutada kõiki antud blokis kehtivaid C - keele muutujaid.
Sellise faili transleerimiseks, mis sisaldab assemblerkäske C - keele seas, võite kasutada kas Turbo Assemblerit või Borland C/C++ programmeerimiskeskkonna (IDE) oma assemblerit. Esimesel juhul tuleb teil kasutada translaatori valikut **-B** või sisestada oma programmi teksti eeltranslaatori käsk `#pragma inline`. Vastasel juhul kasutatakse Borland C/C++ programmeerimiskeskkonna oma assemblerit. Viimane on ka tunduvalt kiirem meetod. Borland C/C++ programmeerimiskeskkonna oma assembler lubab teil kasutada enamikku assemblerkäskudest. Te võite isegi defineerida muutujaid assembleristiilis käskudega DB, DW ja DT, kuid te ei saa kasutada üldisi assemblerkäske nagu näiteks SEGMENT ja ASSUME. Kui te olete valinud menüüst

***O**ptions* | ***C**ompiler* | ***C**ode Generation* valiku 80186 või 80286 protsessori käsud, siis võite neid käske kasutada ka selle programmi sees. Te ei saa aga kasutada 80386 käske C - keele faili sisestatud assemblerkäskudes. Protsessori 80386 käskude kasutamiseks tuleb luua assembleris terve funktsioon ja transleerida ta eraldi näiteks Turbo Assembleriga ning seejärel kasutada seda funktsiooni oma C - keele programmis.

Kui te soovite C - keele faili salvestatud assemblerkäskude transleerimiseks kasutada Turbo Assemblerit või mingit muud assemblerit, siis tuleb ka peale translaatori käsku **-B** või eeltranslaatori käsku `#pragma inline` määrata, milist assembleri kasutada. Selleks valige menüüst *Options* käsk *Transfer*. Avanenud dialoogis määrake assembleri asukoht (kataloog, kuhu on installeeritud Turbo Assembler või mingi muu assembler). Peale normaalset installeerimist on need andmed tavaliselt korras ja ei vaja muutmist.

C - keele faili sisestatud assemblerkäsud peavad sobima antud programmiga. Te peate arvesse võtma kasutatud mälumudelit ja kasutama oma assemblerkäskudes sobivaid osuteid. Assemblerkäskudes võib kodeerida ka tsükleid ja hüppeid (JMP, JE, JB jne.), kuid tsükli või hüppe märgiks peab olema C - keele märgend (*label*). Selline märgend ei saa asuda assemblerkäskude bloki sees.

C - keeles kasutatakse programmi spetsiaalsete andmete salvestamiseks registreid. Segmendiregistreid võite te küll muuta, kuid nende sisu tuleb enne järgmiste C - keele lausete täitmist taastada. Üldisi registreid AX, BX, CX, DX, segmendiregistrit ES ja tähiseid võite muuta, kuidas soovite, ilma nende sisu taastamise üle muret tundmata. Registreid CS ja SS ei tohiks üldse otseselt muuta, neid registreid tohib muuta ainult kaudsete käskudega (JMP, JE, PUSH, POP jne.).

Kui Borland C/C++ programmeerimiskeskkonna assemblerist ei peaks jätkuma, siis tuleks luua eraldi *.ASM failid ja kasutada nende transleerimiseks Turbo Assemblerit. See nõuab põhjalikke teadmisi C - keele funktsioonide kasutamisest. Assemblerfunktsioonid peavad käituma täpselt nii nagu normaalsed C - keele funktsioonid, et neid saaks kasutada C - keele programmis.

Pprogrammeerimiskeeles C kasutatakse parameetrite üleandmiseks funktsioonidele pinu. Parameetrid salvestatakse pinusse alates kõige viimasest ja lõpetades esimesega. Seejärel salvestatakse pinusse registri IP sisu ehk koha aadress, kust programm peaks peale funktsiooni lõpetamist jätkama. Kui väljakutsutav funktsioon omas pikka viita, s.t. tuleb vahetada ka registri CS sisu, siis salvestatakse ka tolle registri vana sisu pinusse. Funktsiooni parameetrid ja tagasipöördumiseks vajalik aadress moodustavad nn. pinuraami (*stack frame*). Iga funktsioon omab oma pinuraami. Parameetrite ja kohalike muutujate pinult lugemiseks kasutatakse registrit BP (*base pointer*). Pinuraamis salvestatakse eelmise funktsiooni registri BP väärtus tagasipöörumiseks vajalikku aadressi järel. Sellele järgnevad antud funktsiooni kohalikud muutujad. Joonisel 8 näete tüüpilist pinuraami ehitust:

Pinu algus ↑

...
teine parameeter
esimene parameeter
tagasipöördumise aadressi CS
tagasipöörumise aadressi IP
eelmise funktsiooni BP
esimene kohalik muutuja
teine kohalik muutuja
...

eelmise pinuraami lõpp

selle pinuraami lõpp

Andmesegmendi algus ↓

Joonis 8: Funktsiooni pinuraami ehitus

Väljakutsutav funktsioon peab kohe alguses salvestama registri BP pinusse. Seejärel võib ta reserveerida pinul ruumi oma kohalike muutujate jaoks. Kui te kasutate C - keele programmis asemblerkäske, siis võite lasta kõigi nende probleemidega tegelda translaatoril. Järgmises näites näete assemblerkäskude kasutamist ühes C - keele programmis.

ASMDEMO1.C

```
/*******************************************************/
/***                                                 ***/
/***    AsmDemo1.C                                   ***/
/***                                                 ***/
/***    Näitab, kuidas kasutada  C - keele programmis  ***/
/***    assemblerkäske ilma eraldi *.ASM faili loomata.***/
/*******************************************************/

#include <stdio.h>

/*-----------------------------------------------------*/
/*---                                               ---*/
/*---   SumArray()                                  ---*/
/*---                                               ---*/
/*---   Liidab n täisarvu ja loovutab tulemuse.      ---*/
/*---   Kuna programm transleeritakse small mälumudeli---*/
/*---   abil, siis on parameetrid jaotatud järgmiselt:---*/
/*---     [BP+4]  -  nCount                         ---*/
/*---     [BP+6]  -  nFirst ehk esimene liidetav     ---*/
/*---     [BP+8]  -  teine liidetav                 ---*/
/*---     [BP+10] -  kolmas liidetav jne.           ---*/
/*-----------------------------------------------------*/
int SumArray(int nCount, int nFirst, ... )
{
  asm MOV AX, nFirst  /* tulemuseks on nFirst kui arve on vaid üks */
  asm MOV CX, nCount  /* täpselt nCount arvu tuleb liita */

  asm MOV BX, BP      /* registris BX salvestame järgmise
                           liikme numbri */
  asm ADD BX, 8
```

```
sum:
  asm ADD AX, [BX]    /* järgmisena liidame [BP+8],
                          siis [BP+10] jne */
  asm ADD BX, 2       /* suurendame BX väärtust */
  asm LOOPNZ sum      /* liidame niikaua, kuni CX on null */
      /* tulemus on juba registris AX, mis on ka tema oige koht */
}


/*=====================================================*/
/*===                                               ===*/
/*===   Main()                                      ===*/
/*===                                               ===*/
/*===   Kasutab funktsiooni SumArray() abi n täisarvu ===*/
/*===   liitmiseks.                                 ===*/
/*=====================================================*/
int main( void )
{
      /* liidame kuus ühte ja trükime nende väärtuse ekraanile */
  printf("6 x 1 = %d", SumArray(6, 1, 1, 1, 1, 1, 1));
  return 0;
}
```

Sel kombel on kõige lihtsam assemblerit C - keele programmiga siduda. Nii võiks näiteks implementeerida funktsiooni, mis tegeleb väga suurt kiirust nõudvate töödega. Samas näete ka muutuva hulga parameetritega funktsioonide loomist. Selles assemblerfunktsioonis ei vaja me enam `va_` makrosid, vaid kasutame parameetreid otse registri BP abil.
Funktsiooni tulemuseks on int, mis loovutatakse väljakutsujale registris AX. Kuna nimetatud arv summeerimise lõpus juba asub nimetatud registris, siis ei olegi vaja midagi muud teha.
Programmeerimiskeeles C on kindlad reeglid selle kohta, kuidas tagastada funktsiooni väärtus väljakutsujale. Erinevad väärtused antakse tagasi erineval viisil. Alljärgnevas tabelis näete lühikest ülevaadet funktsiooni väärtuse loovutamisest väljakutsujale

Tüüp	Loovutamise koht
unsigned char	AX
char	AX
enum	AX
unsigned short	AX
short	AX
unsigned int	AX
int	AX
unsigned long	DX:AX (DX = viimased 2 bait, AX = esimesed 2 baiti)
long	DX:AX

Tabel 11: Funktsiooni väärtuse väljakutsujale loovutamise viis

Tüüp	Loovutamise koht
float	Matemaatikaprotsessori register TOS
double	Matemaatikaprotsessori register TOS
long double	Matemaatikaprotsessori register TOS
near *	AX
far *	DX:AX
struct (1-2 baiti)	AX (esimene bait asub AL -s)
struct (4 baiti)	DX:AX
struct (3 või enam kui 4 baiti)	Kopeeritakse andmesegmenti ja loovutatakse viit sellele andme-struktuurile.

Tabel 12: Funktsiooni väärtuse väljakutsujale loovutamise viis (järg)

Kui te soovite kasutada täielikult assembleris programmeeritud funktsioone, siis tuleb ise tegelda ka pinuraami loomisega, vajalike registrite väärtuste säilitamisega ja muu sellisega, millega tavaliselt tegeleb translaator. Kui te peate muutma mingit programmile vajalikku registrit, siis säilitage tema väärtus kõigepealt PUSH käsuga pinus ja peale registri kasutamist taastage tema väärtus POP käsuga. Tähele tuleb panna, et kõik PUSH ja POP käsud täidetaks õiges järjekorras. See väärtus, mis viimasena pinusse salvestatati, tuleb taastada esimeses järjekorras.
C - keele translaator lisab iga funktsiooni nime ette sümboli "_". Seda tuleb assemblerprogrammis arvesse võtta ja seal ise funktsiooni nime ette too sümbol lisada. C - keele programmis saab siis kasutada selle funktsiooni nime ilma tolle sümbolita, kuna translaator selle talle hiljem lisab.
Loodud *.ASM faili võite samuti lisada oma projektfaili. Kontrollige kindlasti *:ASM failide valikuid projektfailis (klahvikombinatsioon <Ctrl>+<O>). Avanenud dialoogis peab olema *.ASM faili translaatoriks määratud Turbo Assembler või mingi muu sobiv assembler.
Näiteprogrammis ASMDEMO2.C näete puhta assemblerfunktsiooni kasutamist C - keele programmis. Kuna too *.ASM fail on kompileeritud LARGE mälumudeliga (ainult selleks, et teha asju natuke keerukamaks) ja *.C fail SMALL mälumudeliga, siis tuleb *.C faili lisada tolle assemblerfunktsiooni deklaratsioon, et translaator teaks, kuidas talle parameetreid üle anda.

ASMDEMO2.C

```
/*******************************************************/
/***                                                ***/
/***   ASM2.C                                       ***/
/***                                                ***/
/***   See programm näitab, kuidas kasutada täielikult ***/
/***   assemblerkoodis programmeeritud funktsioone.    ***/
/*******************************************************/

/*----------------< Päisefailid >-----------------------*/
#include <stdio.h>

/*----------------< Funktsioonide prototüübid >----------*/
```

```
/* see funktsioon on defineeritud assemblerfailis RUUTJUUR.ASM */
extern long double far  ruutjuur(long double x);

/*=======================================================*/
/*===                                                 ===*/
/*===   Main()                                        ===*/
/*===                                                 ===*/
/*===   Selles programmis arvutame arvu Pi väärtuse   ===*/
/*===   suurima voimaliku täpsusega. Selleks kasutame ===*/
/*===   failis RUUTJUUR.ASM defineeritud samanimelist ===*/
/*===   funktsiooni.                                  ===*/
/*=======================================================*/
int main( void )
{
  printf("Arvu Pi väärtus on: %10.15Le", ruutjuur(2.0));
  return 0;
}
```

RUUTJUUR.ASM

```
;*******************************************************
;*                                                     *
;*   RUUTJUUR.ASM                                      *
;*                                                     *
;*   Selles failis defineerime assemblerfunktsiooni    *
;*   _ruutjuur. Pinul asub alates [BP+6] funktsiooni   *
;*   ainus argument, mis on 10-baidine kümnendmurd.    *
;*   Funktsioon arvutab oma argumendi ruutjuure, kasuta-  *
;*   des selleks kas matemaatikaprotsessorit voi       *
;*   Borland C/C++ teeki EMUL.LIB, mis jäljendab matemaa- *
;*   tiakaprotsessorit.                                *
;*   Selle funktsiooni kuju C - keeles on:             *
;*      long double far _ruutjuur(long double x);      *
;*                                                     *
;*******************************************************

      EMUL             ;kasuta Borland C/C++ teeki kui matemaatika-
                               ;protsessorit ei ole
      DOSSEG      ;kasuta DOS standardset segmentide järjestust
      .MODEL large             ;kasuta LARGE mälumudelit
      .CODE                    ;siit algab funktsiooni kood
      PUBLIC  _ruutjuur   ;C - keeles lisatakse funktsiooni nimele
                               ;peale linkimist sümbol "_"
_ruutjuur  PROC
      push  bp                 ;loo korralik pinuraam
      mov   bp, sp
      fld   tbyte ptr [bp+6]   ;laadi parameeter matemaatika-
                               ;protsessori registrisse TOS
      fsqrt                    ;arvuta ruutjuur
                               ;tulemus jääb registrisse TOS
      pop   bp
      ret                      ;ja ongi koik
_ruutjuur  ENDP
      END
```

Kui te lingite C keele faili mingi *.ASM failiga, siis saab mõlemas failis kasutada teises failis defineeritud muutujaid, täpselt nii nagu ka mitme *.C faili puhul. C keele failis defineeritud muutujaid saab *.ASM failis kasutada, kui nad deklareerida seal võtmesõna **extern** abil. Samuti võib defineerida muutujaid ka *.ASM failis (võtmesõnaga **public**) ja neid hiljem *.C failis kasutada,

kui nad deklareerida seal võtmesõnaga **extern**. Ainult HUGE mälumudeliga programmides võib andmetega natuke probleeme tekkida.
Nagu juba peatükis *Mälujaotus* öeldud, on HUGE ja LARGE mälumudeli põhiline erinevus selles, et HUGE mälumudeliga programm võib omada enam kui 64 KB initsialiseeritud ja initsialiseerimata globaalseid andmeid. Kõigi muude mälumudelite puhul osutab register DS ühele segmendile, mis sisaldab kõiki globaalseid andmeid (nn. DGROUP segment). Dünaamilisi andme-segmente võib rohkem olla. Järgmises tabelis näete registrite sisu erinevate mälumudelite puhul.

Mälumudel	registri CS sisu	registri DS sisu
TINY	_TEXT	DGROUP
SMALL	_TEXT	DGROUP
COMPACT	_TEXT	DGROUP
MEDIUM	<faili nimi>_TEXT	DGROUP
LARGE	<faili nimi>_TEXT	DGROUP
HUGE	<faili nimi>_TEXT	<faili nimi>_DATA

Tabel 13: Registrite CS ja DS sisu erinevate mälumudelite puhul

Nagu näete, ei ole mälumudelite TINY, SMALL, COMPACT, MEDIUM ja LARGE globaalsete andmetega puhul mingit probleemi. Te peate nad lihtsalt deklareerima*.ASM failis võtmesõnaga **extern** ja seejärel võite kohe tarvitada tolle muutuja nime tema sisu kasutamiseks, näiteks:

```
      *.C failis:
...
int   i, j;
extern  char  *nimi, buf[20];
...
strcpy(buf, nimi);       /* kasutame *.ASM failis
                         defineeritud muutujat */

      *.ASM failis
...
      EXTRN _i : WORD
      EXTRN _j : WORD
      .DATA
      DB   nimi   "Jaanus", 0 ;nimi, mille lõppu lisame nulli
                              ;C keele funktsioonide jaoks
...
      MOV   AX, i        ;kasutame *.C failis defineeritud
      MOV   BX, j        ;muutujaid
```

See kõik on võimalik just sellepärast, et DS osutab segmendile DGROUP. Mälumudeli HUGE puhul omab aga iga fail oma andmesegmenti, mis ka sisaldab tema globaalseid muutujaid. Iga faili jaoks on olemas segment nimega <faili nimi>_DATA, mis sisaldab selles failis defineeritud initsialiseeritud muutujaid ja segment nimega <faili nimi>_BSS selles failis defineeritud initsialiseerimata muutujate jaoks. Assemblerprogrammis tuleks siis oma

andmesegmendi kasutamiseks kõigepealt laadida selle segmendi aadress registrisse DS, näiteks:

```
       .MODEL HUGE
       .FARDATA
       DW    number      235
...
       .CODE
       PUBLIC      _func1
_func1       PROC
       push  bp           ;loome pinuraami
       mov   bp, sp
       push  ds           ;säilitame registri DS endise väärtuse
       mov   ax, @fardata ;loeme oma andmesegmendi aadressi
       mov   ds, ax       ;registrisse DS
       mov   ax, number   ;teeme midagi selle muutujaga
       ...
       pop   ds
       pop   bp
       ret
_func1       ENDP
       END
```

Kui te aga HUGE mälumudeli puhul tahaksite *.ASM failis kasutada temaga lingitud *.C failis defineeritud muutujaid, siis tekib küsimus - millises segmendis need muutujad on salvestatud ja kust saada segmendi aadressi. Selle probleemi jaoks on vaid üks õige lahendus - kasutada **SEG** käsku koos muutujanimega ja lasta translaatoril selle probleemiga tegeleda. Kui ülemises näites defineeritud muutuja *number* oleks olnud defineeritud mingis muus failis, siis oleks tulnud teha järgmist:

```
       ...
       push ds
       mov  ax, SEG _number
       mov  ds, ax
       mov  ax, _number
       ...
       pop  ds
       ...
```

Assemblerfailis kasutatavad C - keele funktsioonid tuleb samuti deklareerida võtmesõnaga EXTRN. Kui funktsiooni või muutuja deklaratsioonile või definitsioonile lisada sümbol "C", siis teab Turbo Assembler, et seda muutujat tuleb kasutada C - keele reeglite järgi ja lisab talle ise ka sümboli "_". Nüüd saab ka assemblerfailis kasutada C - keele funktsioone käsu CALL abil. Te võite kas ise salvestada vajalikud parameetrid pinule ja seejärel tolle funktsiooni välja kutsuda, või kasutada sümbolit "C" ja reastada lihtsalt funktsiooni parameetrid temast paremale.
Kohalike muutujate kasutamiseks tuleb nihutada pinuviita (SP) muutujatele ruumi tegemiseks allapoole. Nüüd saab registrit BP kasutada nende muutujate väärtuste lugemiseks ja muutumiseks. Enne funktsiooni lõppu tuleb need muutujad aga pinult eemaldada ja alles seejärel taastada endine BP. Järgmises programmis näete, kuidas kasutada assemblerprogrammis C - keele funktsioone ja kohalikke muutujaid.

ASMDEMO3.C

```
/*****************************************************/
/***                                             ***/
/***   AsmDemo3.C                                ***/
/***                                             ***/
/***   Näitab globaalsete andmete vahetamist C - keele ***/
/***   ja assemblerfunktsioonide vahel ning C - keele  ***/
/***   funktsioonide kasutamist assemblerfunktsioonis. ***/
/*****************************************************/

/*----------------< Päisefailid >------------------------*/

#include    <stdio.h>

/*----------------< Globaalsed muutujad >----------------*/

struct MyData {
  char      name[20];
  int amount;
} Data[] = {
  "Peeter", 4,
  "Jaanus", 5,
  "Toomas", 21,
  NULL, 0,
};

char  szFormat[] = "%s omab %d protsenti aktsiatest\n";


/*----------------< Funktsioonide prototüübid >----------*/

extern void ShowData(struct MyData *);
int CalcPercent(int, int);


/*===================================================*/
/*===                                           ===*/
/*===   main()                                  ===*/
/*===                                           ===*/
/*===   Selles programmis kasutatakse failis    ===*/
/*===   ASMBAR.ASM defineeritud funktsioone     ===*/
/*===   soovitud andmete esitamiseks ekraanil   ===*/
/*===   graafilise ringdiagrammina.             ===*/
/*===================================================*/
int main( void )
{
  ShowData(&Data);           /* esita diagramm ekraanile */
  return 0;
}     //main





/*---------------------------------------------------*/
/*---                                           ---*/
/*---   CalcPercent()                           ---*/
/*---                                           ---*/
/*---   Arvutab, kuipalju protsente moodustab arv     ---*/
/*---   nNum arvust nWhole. Seda funktsiooni kasu-    ---*/
/*---   tatakse assemblerfunktsioonis ShowData().     ---*/
/*---------------------------------------------------*/
int CalcPercent(int nNum, int nWhole)
{
  return (int)((double)nNum /  (double)nWhole * 100.0);
```

```
}     /* CalcPercent() */
```

SHOWDATA.ASM

```
;***********************************************************
;***                                                     ***
;***   ShowData.ASM                                      ***
;***                                                     ***
;***   Sisaldab assemblerfunktsiooni ShowData(), mis     ***
;***   esitab  ekraanile talle üle antud andmed.         ***
;***   Lisaks sellele näidatakse selles failis, kuidas  ***
;***   kasutada assemblerfunktsioonis kohalikke          ***
;***   muutujaid.                                        ***
;***********************************************************

      DOSSEG
      .MODEL SMALL
        ;koiki neid C funktsioone saab kasutada assemblerfunktsioonis
      EXTRN C printf:PROC          ;standardne C funktsioon
      EXTRN C CalcPercent:PROC     ;teises failis defineeritud
                                   ;funktsioon
      .DATA
      EXTRN C szFormat             ;formaadispetsifikatsioon
      .CODE
      PUBLIC C ShowData
      ;Pinuraamil on järgmised andmed:
      ;  [BP-4]   -     kohalik muutuja - viit struktuurile
      ;  [BP-2]   -     kohalik muutuja - kogu summa
      ;  [BP+0]   -     vana BP
      ;  [BP+2]   -     tagasipöördumise aadress
      ;  [BP+4]   -     andmestruktuuri MyStrct aadress
ShowData    PROC  C
      push bp
      mov  bp, sp
      sub  sp, 4  ;seda ruumi pinul on vaja kohalike muutujate jaoks
      mov  bx, [BP+4]            ;hangi andmestruktuuri aadress
      mov  word ptr [BP-2], 0    ;initsialiseeri muutuja
getwhole:
      mov  ax, word ptr [BX+20]  ;hangi antud isiku aktsiate arv
      add  [BP-2], ax            ;liida tema väärtus kohalikule
                                      ;muutujale
      add  bx, 22                ;nihuta viita edasi
      cmp  ax, 0      ;kontrolli, kas see on viimane arv
      jnz getwhole    ;kui see ei ole viimane arv,
                                 ;siis hangi järgmine
      mov  ax, [bp+4]
      mov  [bp-4], ax            ;initsialiseeri viit
show:
      mov  bx, [bp-4]   ;BX voib muutuda, aga kohalik muutuja mitte
                              ;seepärast säilitame aadressi pinul
      mov  ax, [bx+20]
      cmp  ax, 0              ;kontrolli, kas see oli viimane isik
      jz   getout
      call CalcPercent C, ax, [bp-2]            ;arvuta protsent
      call printf C, OFFSET szFormat, bx, ax  ;väljasta tulemused
      add [bp-4], 22                      ;nihuta viita edasi
      jmp show
getout:
      add  sp, 4    ;enne sai SP -st 4 lahutatud, nüüd tuleb
                    ;see uuesti liita
      pop  bp
      ret
ShowData    ENDP
      END
```

Katkestused

Arvutid kasutavad katkestusi (*interrups*) mitmesugusteks ülesanneteks. Näiteks kui te vajutate mingile klahvile, saadab klaviatuuriprotsessor põhiprotsessorile ühe katkestuse. Katkestus on signaal, mis sunnib põhiprotsessorit oma tööd hetkeks katkestama ja täitma mingit muud ülesannet. Selleks salvestab protsessor oma hetkelise töö jätkamiseks vajalikud andmed ja asub seejärel sellele katkestusele vastavat ülesannet täitma. Mälu alguses, esimeses 1024 baidis asub tabel, mis sisaldab pikki viitasid iga katkestuse puhul täidetavale funktsioonile. Iga sissekanne sellesse tabelisse sisaldab 4 baiti (pika viida suurus). Seega sisaldab tabel täpselt 256 sissekannet. Kui te nüüd vajutasite näiteks mingile klahvile, siis katkestab protsessor oma hetkelise töö ja täidab sellele katkestusele (katkestus number 9) vastava ülesande. Selleks hangib ta katkestuste tabelist vastava (üheksanda) sissekande ja jätkab tööd sellelt aadressilt. Nimetatud aadressil asub tavaliselt operatsioonisüsteemi või BIOSi funktsioon, mis uurib järele, millisele klahvile vajutati ja väljastab vastava sümboli ekraanile. Peale selle funktsiooni täitmist jätkab protsessor oma endist tööd.

BIOS on kogumik masinkoodis funktsioone, mis täidavad mitmesuguseid lihtsaid ülesandeid nagu näiteks ekraanile väljastamine ja klaviatuurilt lugemine. BIOSi funktsioonid on salvestatud arvuti ROM (*Read Only Memory*) mälus. Sellises mälus salvestatut saab vaid lugeda, kuid mitte muuta. Seepärast ei ole arvutit startides vaja BIOSi mitte kettalt mällu lugeda, vaid ta juba on mälus. Arvuteid on väga mitmesuguseid ja seepärast loodi BIOS, mis kindlustab, et üks ja sama tarkvara saab töötada ilma muutusteta väga erinevatel arvutitel. Programm, mis soovib näiteks väljastada ekraanile mingit sümboli (tähe või numbri), salvestab vajalikud andmed sobivates registrites ja kutsub välja vastava BIOSi funktsiooni. Selleks peab ta aga kasutama sobivat katkestust. Nüüd täidab BIOSi funktsioon selle ülesande vastavalt antud arvuti ehitusele. Programm ei pea enam teadma, millist aadressi selles arvutis kasutatakse ühe või teise seadme jaoks, vaid ainult seda, milline katkestus seda tööd teeb ja milliseid parameetreid ta vajab. Need andmed on aga standardiseeritud ja nii saab sama programm töötada väga erinevatel arvutitel ilma nende ehitusest väga palju teadmata.

Katkestusi on väga mitmesuguseid. Osa neist on reserveeritud operatsioonisüsteemi jaoks. Need katkestused uuendavad näiteks arvuti kella seisu, värskendavad mälu (*RAM refresh cycles*), loevad mitmesuguste seadmete (klaviatuuri, hiire jt.) seisu ja väljastavad andmeid mitmesugustele seadmetele (ekraan, trükkal) jne.

Katkestused on jaotatud järgmistesse gruppidesse:

1. **Katkestused: 0 - 15** - Need katkestused kutsutakse välja arvuti poolt. Sellesse gruppi kuuluvad klaviatuuri ja muude seadmete jaoks vajalikud katkestused, ning mitmed süsteemi enda jaoks vajalikud katkestused. Süsteem omab iga sellise katkestuse jaoks sobiva BIOSi funktsiooni, mis on kõik salvestatud arvuti ROM -is. Operatsioonisüsteem (DOS) asendab sageli osa

neist funktsioonidest oma funktsioonidega arvuti töö kontrollimiseks. Ka programm võib osa neist funktsioonidest asendada enda poolt loodutega .

2. **Katkestused: 16 - 31** - Neid katkestusi kasutavad nii operatsioonisüsteem kui ka programm mitmesuguste seadmete (näiteks ekraani) kasutamiseks. Igale sellisele katkestusele vastab üks BIOSi funktsioon.
3. **Katketused: 32 - 63** - Neid katkestusi kasutab operatsioonisüsteem. Programmid võivad nende katkestuste kaudu kasutada operatsioonisüsteemi funktsioone, näiteks failide avamiseks ja lugemiseks.
4. **Katketused: 64 - 95** - Neid katkestusi kasutavad operatsioonisüsteem ja mitmed ohjurprogrammid oma ülesannete täitmiseks. Igale katkestusele vastab kas üks BIOSi või mingi ohjurprogrammi funktsioon.
5. **Katkestused: 96 - 102** - Need katketused on reserveeritud programmide jaoks. Kuidas programmid neid kasutavad, ei ole kindlaks määratud.
6. **Katkestus: 103** - Seda katkestust kasutab laiendatud mälu ohjurprogramm.
7. **Katketused: 112 - 119** - Neid katketusi kasutavad ohjurprogrammid
8. **Katkestused: 128 - 240** - Neid katkestusi kasutab teie arvuti ROM -is salvestatud programmeerimiskeel BASIC, kui teie arvuti ROM üldse sisaldab keelt BASIC.
9. **Katkestused: 241 - 255** - Neid katkestusi esialgu ei kasutata. Võibolla omistatakse neile mingi tähendus tulevikus.

Mingi katkestuse kasutamiseks tuleb lihtsalt täita sobivad registrid vajalike väärtustega ja kutsuda nimetatud katkestus välja. Selleks sisaldab Borland C/C++ translaator funktsioone *int86()*, *int86x()*, *intdos()* ja *intdosx()*.

```
int int86(int nIntNr, union REGS *inregs, union REGS *outregs);
int int86(int nIntNr, union REGS *inregs, union REGS *outregs, union
SREGS *segregs);
int intdos(union REGS *inregs, union REGS *outregs);
int intdosx(union REGS *inregs, union REGS *outregs, union SREGS
*segregs);
```

Kõik nimetatud funktsioonid kasutavad andmestruktuuri REGS:

```
union REGS {
  struct WORDREGS x;
  struct BYTEREGS h;
};
```

mis koosneb omakorda kahest andmestruktuurist:

```
struct BYTEREGS {
  unsigned char al, ah, bl, bh;
  unsigned char cl, ch, dl, dh;
};
```

ja

```
struct WORDREGS {
  unsigned int ax, bx, cx, dx;
  unsigned int si, di, cflag, flags;
};
```

Nimetatud funktsioonid ja andmestruktuurid on defineeritud päisefailis DOS.H. Need andmestruktuurid jäljendavad protsessori registrite struktuuri. Programm, mis soovib mingit katkestust kasutada, defineerib oma andmesegmendis kaks muutujat tüübist REGS. Ühe neist täidab ta antud katkestuse jaoks sobivate väärtustega. Seejärel kutsub ta välja näiteks funktsiooni *int86()*. Funktsiooni

esimene parameeter määrab, millise järjekorranumbriga katkestust välja kutsuda. Teine ja kolmas parameeter sisaldavad viitasid REGS tüüpi andmestruktuuridele. Esimese andmestruktuuri sisu kopeeritakse enne katkestuse kasutamist protsessori registritesse. Teine struktuur täidetakse peale katkestust protsessori registrites leitud väärtustega. Need väärtused määravad katkestuse tulemuse. Funktsiooni int86() viimased kaks parameetrit võivad osutada ühele ja samale andmestruktuurile. Sel juhul kirjutatakse funktsioonile üleantud struktuuri sisu üle uute väärtustega. Üleantud väärtuste säilitamine ei ole aga tavaliselt vajalik ja see aitab mälu kokku hoida.
Kuna võtmesõna **union** tähendab, et need struktuurid kasutavad üht ja sama mälupiirkonda, siis võite sobivat struktuuri kasutades muuta nii tervet registrit AX (või muid üldisi registreid) kui ka tema osasid AL ja AH.
Struktuur WORDREGS sisaldab veel elementi *cflags*, mis ei vasta ühelegi protsessori registrile. Protsessori register *flags* (tähised) sisaldab mitmeid ühebitised tähiseid. Üks neist - CF (*carry flag*) märgib katkestuste puhul tavaliselt viga. Selle tähise väärtus on ka veel eraldi salvestatud nimetatud elemendis *cflag* ja see on ka funktsiooni enda väärtuseks. Niisiis, kui funktsiooni väärtus (või tema struktuuri WORDREGS elemendi *cflag* väärtus) on nullist erinev, siis on tekkinud mingi viga. Päisefailis DOS.H on defineeritud ka globaalne muutuja `_doserrno`, mille väärtus määrab konkreetse vea. See muutuja on täisarv, mis võib omada päisefailis DOS.H defineeritud väärtusi.
Funktsioon *int86x()* vajab veel neljandat parameetrit, mis osutab andmestruktuurile SREGS:

```
struct SREGS {
  unsigned es;
  unsigned cs;
  unsigned ss;
  unsigned ds;
};
```

Selle parameetri abil võib muuta registrite DS, ES sisu ka enne katkestuse väljakutsumist. Funktsioon *int86()* seevastu kasutab programmi oma DS väärtust.
Funktsioonid intdos() ja intdosx() ei vaja katkestuse numbrit, kuna nad kasutavad vaid üht kindlat katkestust - 21hex, mis on ette nähtud operatsioonisüsteemi DOS teenuste kasutamiseks.
Järgmises näiteprogrammis näete, kuidas nimetatud funktsioone kasutada. See fail sisaldab funktsioone hiire kasutamiseks graafilistes ja tekstirezhiimis töötavates programmides. Siindefineeritud funktsioone kasutame järgmiste peatükkide näiteprogrammides. Lisatud on ka lihtne näiteprogramm MOUDEMO, mis kasutab osa neist funktsioonidest. Neid funktsioone saab kasutada ka tekstirezhiimis töötavates programmides. Hiirekursori koordinaadid antakse aga ka tekstirezhiimis pikselitena. Seega tuleb neid tekstirezhiimis jagada vastavalt ekraani lahutusvõimele sobiva arvuga (tavaliselt 8). Tulemuse täisarvuline osa määrab sobiva koordinaadi tekstirezhiimis.

MOUFUNC.H

```
/*******************************************************/
/***                                               ***/
/***   MouFunc.H                                   ***/
/***                                               ***/
/***   See fail sisaldab sümboolseid konstante ja   ***/
/***   funktsioonide deklaratsioone faili MOUFUNC.C ***/
/***   jaoks.                                      ***/
/*******************************************************/

#include <dos.h>

#define MOU_NBUTTON          0
#define MOU_LBUTTON          1
#define MOU_RBUTTON          2
#define MOU_BBUTTON          3
#define MOU_GETLBUTTONINFO   0
#define MOU_GETRBUTTONINFO   1

#ifndef __MOUFUNC_C_

extern int MouInit( void );
extern void MouShow( void );
extern void MouHide( void );
extern int MouGetInfo(int *, int *);
extern int MouGetPressed(int, int*, int*);
extern int MouGetReleased(int, int*, int*);
extern void MouGotoXY(int, int);
extern void MouSetAlNumCursor(int, char, char, char, char);

#endif
```

MOUFUNC.C

```
/*****************************************************/
/***                                             ***/
/***   MouFunc.C                                 ***/
/***                                             ***/
/***   See fail sisaldab funktsioone hiire kasuta-   ***/
/***   miseks. Kui te soovite neid funktsioone oma   ***/
/***   programmis kasutada, siis tuleb teil lisada   ***/
/***   oma programmi päisefail MOUFUNC.H.            ***/
/***   Lisaks sellele tuleb teil installeerida mingi ***/
/***   standardne hiire ohjurprogramm. Siintoodud    ***/
/***   funktsioonid kasutavad vaid ohjurprogrammi    ***/
/***   teenuseid.                                    ***/
/*****************************************************/


#include  __MOUFUNC_C_
#include  "moufunc.h"

/*-----------------------------------------------------*/
```

```
/*---                                                ---*/
/*---   MouInit()                                    ---*/
/*---                                                ---*/
/*---   See funktsiooni initsialiseerib hiire        ---*/
/*---   ohjurprogrammi. Funktsiooni väärtus on nullist---*/
/*---   erinev, kui initsialiseerimine onnestus,     ---*/
/*---   vastasel juhul null.                         ---*/
/*------------------------------------------------------*/
int MouInit( void )
{
  union REGS  myregs;

  myregs.x.ax = 0;              /* katkestuse nr 33 funktsioon 0 */
  int86(0x33, &myregs, &myregs);  /* algväärtus ei oma tähendust */
  return myregs.x.ax;             /* tulemus asub registris AX */
}     /* MouInit */


/*------------------------------------------------------*/
/*---                                                ---*/
/*---   MouShow()                                    ---*/
/*---                                                ---*/
/*---   See funktsioon kuvab hiirekursori ekraanile. ---*/
/*---   Enne kui te midagi ekraanile esitate voi     ---*/
/*---   mingit ekraani osa rasterpildina salvestate, ---*/
/*---   oleks mottekas hiirekursor ekraanilt eemal-  ---*/
/*---   dada ja peale seda uuesti ekraanile esitada. ---*/
/*---   Hiirekursori ekraanilt eemaldamiseks         ---*/
/*---   kasutage funktsiooni MouHide().              ---*/
/*------------------------------------------------------*/
void MouShow( void )
{
  union REGS  myregs;

  myregs.x.ax = 1;              /* katkestuse nr 33 funktsioon 1 */
  int86(0x33, &myregs, &myregs);
}     /* MouShow */

/*------------------------------------------------------*/
/*---                                                ---*/
/*---   MouHide()                                    ---*/
/*---                                                ---*/
/*---   Eemaldab hiirekursori ekraanilt.             ---*/
/*------------------------------------------------------*/
void MouHide( void )
{
  union REGS  myregs;

  myregs.x.ax = 2;              /* katkestuse nr 33 funktsioon 2 */
  int86(0x33, &myregs, &myregs);
}     /* MouHide */



/*------------------------------------------------------*/
/*---                                                ---*/
/*---   MouGetInfo()                                 ---*/
/*---                                                ---*/
/*---   See funktsioon hangib hiirekursori hetkelise ---*/
/*---   positsiooni ekraanil ja uurib, kas mingi     ---*/
/*---   hiireklahv on alla vajutatud. Funktsiooni    ---*/
/*---   tulemuseks on konstant, mis määrab, milline  ---*/
/*---   hiireklahv (kui ükski) on alla vajutatud.    ---*/
/*---   See voib olla üks järgmistest väärtustest:   ---*/
/*---   MOU_NBUTTON (0) - ükski klahv ei ole all.    ---*/
/*---   MOU_LBUTTON (1) - vasak klahv on all         ---*/
/*---   MOU_RBUTTON (2) - parempoolne klahv on all.  ---*/
/*---   MOU_BBUTTON (3) - molemad klahvid on all.    ---*/
/*---   Viimane väärtus vastab kolmeklahvilise hiire ---*/
/*---   puhul keskmisele klahvile.                   ---*/
```

```
/*-------------------------------------------------------*/
int MouGetInfo(int *pRow, int *pColumn)
{
  union REGS  myregs;

  myregs.x.ax = 3;               /* katkestuse nr 33 funktsioon 3 */
  int86(0x33, &myregs, &myregs);
  *pRow = myregs.x.dx;
  *pColumn = myregs.x.cx;
  return myregs.x.bx;
}     /* MouGetInfo */



/*-------------------------------------------------------*/
/*---                                                 ---*/
/*---   MouGetPressed()                               ---*/
/*---                                                 ---*/
/*---   See funktsioon uurib, kui mitu korda soovitud ---*/
/*---   hiireklahvi on vajutatud alates sellest       ---*/
/*---   hetkest, kui seda funktsiooni viimati         ---*/
/*---   kasutati ja millises punktis teda viimati     ---*/
/*---   vajutati. Kui funktsiooni esimene parameeter  ---*/
/*---   on null, siis uuritakse vasaku klahvi andmeid,---*/
/*---   kui aga 1, siis parema klahvi andmeid.        ---*/
/*-------------------------------------------------------*/
int MouGetPressed(int nButton, int *pRow, int *pColumn)
{
  union REGS  myregs;

  myregs.x.ax = 5;               /* katkestuse nr 33 funktsioon 5 */
  myregs.x.bx = nButton;         /* milline klahv meid huvitab */
  int86(0x33, &myregs, &myregs);
  *pRow = myregs.x.dx;
  *pColumn = myregs.x.cx;
  return myregs.x.bx;            /* kui mitu korda seda klahvi */
                                 /* on vajutatud */
}     /* MouGetPressed */



/*-------------------------------------------------------*/
/*---                                                 ---*/
/*---   MouGetReleased()                              ---*/
/*---                                                 ---*/
/*---   See funktsioon uurib, kui mitu korda soovitud ---*/
/*---   hiireklahvi on vabastatud alates sellest, kui ---*/
/*---   seda funktsiooni viimati kasutati ja millises ---*/
/*---   punktis ta viimati vabastati.                 ---*/
/*-------------------------------------------------------*/
int MouGetReleased(int nButton, int *pRow, int *pColumn)
{
  union REGS  myregs;

  myregs.x.ax = 6;               /* katkestuse nr 33 funktsioon 4 */
  myregs.x.bx = nButton;         /* milline klahv meid huvitab */
  int86(0x33, &myregs, &myregs);
  *pRow = myregs.x.dx;
  *pColumn = myregs.x.cx;
  return myregs.x.bx;            /* kui mitu korda seda klahvi */
                                 /* on lahti lastud */
}     /* MouGetReleased */


/*-------------------------------------------------------*/
/*---                                                 ---*/
/*---   MouGotoXY()                                   ---*/
/*---                                                 ---*/
/*---   Nihutab hiirekursori soovitud punkti.         ---*/
/*-------------------------------------------------------*/
```

```
void MouGotoXY(int nRow, int nColumn)
{
  union REGS  myregs;

  myregs.x.ax = 4;              /* katkestuse nr 33 funktsioon 4 */
  myregs.x.dx = nRow;
  myregs.x.cx = nColumn;
  int86(0x33, &myregs, &myregs);
}     /* MouGotoXY */


/*---------------------------------------------------*/
/*---                                             ---*/
/*---   MouSetAlNumCursor()                       ---*/
/*---                                             ---*/
/*---   See funktsioon määrab hiirekursori kuju ja    ---*/
/*---   välimuse, kui ekraan on tekstirezhiimis.      ---*/
/*---   Parameetrid nScreenChar ja nScreenAttr        ---*/
/*---   määravad, kui palju hiirekursori alla jäänud  ---*/
/*---   tähest peab jääma nähtavaks. Väärtus 0xFF     ---*/
/*---   jätab kogu tähe muutumatuks, väärtus 0        ---*/
/*---   kustutab ta täielikult. Nimetatud parameetrid ---*/
/*---   liidetakse loogilise AND operatsiooniga ja    ---*/
/*---   saadud tulemusele liidetakse parameerite      ---*/
/*---   nCursorChar ja nCursorAttr väärtused loogilise---*/
/*---   XOR operatsiooniga.                           ---*/
/*---------------------------------------------------*/
void MouSetAlNumCursor(int fSoftware, char nScreenChar, char
nScreenAttr,
                              char nCursorChar, char nCursorAttr)
{
  union REGS  myregs;

  myregs.x.ax = 10;             /* katkestuse nr 33 funktsioon 10 */
  myregs.x.bx = fSoftware;
  myregs.h.cl = nScreenChar;
  myregs.h.ch = nScreenAttr;
  myregs.h.dl = nCurorChar;
  myregs.h.dh = nCursorAttr;
  int86(0x33, &myregs, &myregs);
}     /* MouSetAlNumCursor */
```

MOUDEMO.C

```
/*****************************************************/
/***                                             ***/
/***   MouDemo.C                                 ***/
/***                                             ***/
/***   See fail demonstreerib mitmete failis MOUFUNC.C ***/
/***   defineeritud funktsioonide toimet.          ***/
/*****************************************************/

#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "moufunc.h"
/*===================================================*/
/*===                                             ===*/
/*===   Main()                                    ===*/
/*===                                             ===*/
/*===   Initsialiseerib graafilise rezhiimi ja        ===*/
/*===   proovib mitmeid failis MOUFUNC.C defineeritud ===*/
/*===   funktsioone. Seejärel proovitakse neid ka     ===*/
/*===   tekstrezhiimis.                               ===*/
```

```
/*=========================================================*/
int main( void )
{
   int       gdriver = DETECT, gmode, errorcode;
   int       x, y, button;
   char  buf[30];
   int       coord[8];

   initgraph(&gdriver, &gmode, "E:\\BORLANDC\\BGI");
   errorcode = graphresult();
   if (errorcode != grOk)
   {
      printf("Graafika viga: %s\n", grapherrormsg(errorcode));
      printf("Vajutage suvalisele klahvile:");
      getch();
      exit(1);
   }
   if(!MouInit()) {
     printf("Ei suutnud hiire ohjurprogrammi initsialiseerida!\n");
     printf("Vajutage suvalisele klahvile:");
     getch();
     exit(1);
   }
   outtextxy(30, getmaxy() - 30,
      "Vajutage suvalisele klahvile programmi lopetamiseks");
   MouShow();       /* esita hiirkursor ekraanile */
   setfillstyle(SOLID_FILL, 0);
   coord[0] = 100;              /* see ristkülik on vaja täita */
   coord[1] = 200;              /* tagaplaani värviga iga kord */
   coord[2] = getmaxx() - 50;   /* enne uue teksti väjastamist,*/
   coord[3] = coord[1];         /* selleks et puhastada ekraan */
   coord[4] = coord[2];
   coord[5] = 250;
   coord[6] = coord[0];
   coord[7] = coord[5];

   do {
     button = MouGetInfo(&y, &x);
     fillpoly(4, coord);
     MouHide(); /* varja hiirekursor enne teksti vms väljastamist */
     switch(button) {
       case MOU_NBUTTON:
       sprintf(buf,
       "Hiirekursor asub punktis: X = %d, Y = %d", x, y);
       break;
       case MOU_LBUTTON:
       sprintf(buf,
       "Hiire vasakut klahvi vajutati punktis: X = %d,Y = %d", x, y);
       break;
       case MOU_RBUTTON:
       sprintf(buf,
       "Hiire paremat klahvi vajutati punktis: X = %d,Y = %d", x, y);
       break;
       case MOU_BBUTTON:
       sprintf(buf,
       "Hiire molemat klahvi vajutati punktis: X = %d,Y = %d", x, y);
       break;
     };
     outtextxy(110, getmaxy() / 2, buf);
     MouShow();  /* nüüd kuva uuesti hiirekursor ekraanile */
   } while(!kbhit());
   closegraph();
   exit(0);
   return 0;
}     /* main */
```

Katkestustele vastamine

Eelmises peatükis nägime, kuidas kasutada süsteemi katkestusi. Vahel on aga vajalik defineerida funktsioon, mida operatsioonisüsteem kutsub välja ühe kindla katkestuse puhul. See funktsioon võib kas täielikult asendada operatsioonisüsteemi poolt selle katkestuse jaoks defineeritud funktsiooni või vaid osaliselt muuta nimetatud funktsiooni tööd. Nagu nägime, on osa katkestusi reserveeritud programmide jaoks. Nende puhul ei ole tavaliselt vaja muretseda muude selle katkestuse jaoks defineeritud funktsioonide pärast. Kuid tavaliselt kasutatakse mingit süsteemile vajalikku katkestust. Nii näiteks kasutab süsteem ühte kindlat katkestust arvuti kella näidu uuendamiseks. See katkestus genereeritakse arvuti poolt automaatselt 18,2 korda sekundis. Te võite kasutada seda katkestust mingiks oma programmi vajaduseks.
Operatsioonisüsteemi jaoks vajaliku katkestuse kasutamiseks tuleb defineerida oma funktsioon, mis seda katkestust töötleb. Selleks otstarbeks tuleb kasutada võtmesõna **interrupt**. Selline funktsioon peab omama järgmisi parameetreid:

```
void interrupt IntFunk1(int bp, int di, int si, int ds, int es,
                        int dx, int cx, int bx, int ax, int ip,
                        int cs, int flags)
{

}
```

Te ei pea defineerima kõiki siintoodud parameetreid. Te võite osa viimastest parameetritest ära jätta, kui te neid ei vaja, kuid te ei tohi muuta parameetrite järjekorda. Kui te vajate registrit ES, siis peate defineerima ka kõik temale eelnevad parameetrid.
Funktsioonis võite muuta mõningaid või kõiki neid registreid. Nii muudate nende registrite väärtusi pinul ja funktsiooni töö lõppedes salvestatakse need väärtused pinult registritesse. Mõningaid registreid ei tohi muuta, nende registrite hulka kuulub näiteks CS. Sellest oli juba juttu.
Katketusfunktsiooniga võib teha hulga kasulikke töid, kuid operatsioonisüsteemi DOS funktsioonide kasutamisega tuleks olla ettevaatlik. Operatsioonisüsteem DOS ei ole mitmiktegumrezhiimiga operatsioonisüsteem ja ta kasutab oma funktsioonides ainult üht muutujate komplekti. Seega ei tohi neid muutujaid enne muuta, kui antud funktsioon on oma töö lõpetanud. Katkestused aga toimuvad ettearvamatul ajahetkel. Kuna katkestusfunktsioon ei saa kunagi päris kindel olla, et programm hetkel seda sama DOSi funktsiooni ei kasuta, siis ei tohiks ta DOSi funktsioone kasutada.
Peale sobiva katkestusfunktsiooni defineerimist tuleb ta installeerida. Selleks tuleks kirjutada antud funktsiooni aadress katkestustetabeli vastavasse lahtrisse. Enne seda aga tuleks salvestada selle lahtri endine sisu sobivasse muutujasse. Peale programmi lõppu tuleks lahtri endine sisu taastada. Katkestusfunktsioon peaks enne oma muudatuste tegemist kutsuma välja vana katkestusfunktsiooni. Nii kindlustatakse, et teiste seda katkestust kasutavate programmide töö ei

muutuks. Kui te kasutate näiteks kellakatkestust, siis tuleks arvestada seda, et ka teised programmid ja operatsioonisüsteem ise vajavad seda katkestust.
Borland C/C++ sisaldab kahte funktsiooni katkestustetabeli lahtrite sisu muutmiseks ja lugemiseks.

```
void interrupt (*getvect(int nIntNr)) ();
setvect(int nIntNr, void interrupt (*isr)());
```

Funktsioon *getvect()* loovutab soovitud katkestustetabeli lahtri sisu ja funktsioon *setvect()* asendab selle lahtri sisu uue väärtusega.
Järgmises näiteprogrammis asendatakse klaviatuurikontrolleri katkestus, näitamaks iga klahvi ASCII - ja Scan - koodi.

KBDINT.C

```
/*****************************************************/
/***                                              ***/
/***   KbdInt.C                                   ***/
/***                                              ***/
/***   Selles programmis asendatakse klaviatuuri      ***/
/***   katkestusele vastav funktsioon isikliku        ***/
/***   funktsiooniga, mis näitab teatud andmeid       ***/
/***   iga vajutatud klahvi kohta.                    ***/
/*****************************************************/

/*----------------< Päisefailid >------------------------*/

#include   <dos.h>
#include   <conio.h>
#include   <stdio.h>


/*----------------< Globaalsed muutujad >----------------*/

void interrupt (far *oldkbdint)();   /* vana katkestusfunktsioon */
char        cChar, cScanCode;

/*----------------< Funktsioonide prototüübid >----------*/

void interrupt newkbdint();

/*====================================================*/
/*===                                            ===*/
/*===   main()                                   ===*/
/*===                                            ===*/
/*===   Asendab alguses klaviatuuri vana katkestus-   ===*/
/*===   funktsiooni uuega ja hiljem vastupidi.        ===*/
/*===   Programmi töö lopetamiseks vajutage klahvile  ===*/
/*===   <ESC>.                                   ===*/
/*====================================================*/
int main( void )
{
  /* hangi klaviatuuri  vana katkestusfunktsiooni aadress */
  oldkbdint = getvect(0x09);
  /* asenda see uue funktsiooniga */
  setvect(0x09, newkbdint);
    /* oota niikaua, kuni loetud sümbol on null */
  while(27 != getch()) {
    printf("Vajutati klahvile sümboliga: %c ja koodiga: %d\n",
            cChar, cScanCode);
  }
  /* taasta vana olukord enne programmi loppu */
  setvect(0x09, oldkbdint);
```

```
  return 0;
}     /* main */

/*---------------------------------------------------------*/
/*---                                                   ---*/
/*---   NewKbdInt()                                     ---*/
/*---                                                   ---*/
/*---   asendab klaviatuuri vana katkestusfunktsioo-    ---*/
/*---   ni. Koigepealt kutsub see funktsioon välja      ---*/
/*---   vana funktsiooni, et mitte häirida teiste       ---*/
/*---   programmide tööd ja et saada oige klaviatuuri ---*/
/*---   kood. Seejärel trükib ta ekraanile vajutatud    ---*/
/*---   klahvi koodi.                                   ---*/
/*---------------------------------------------------------*/
void interrupt newkbdint()
{
   int            tail;
   /* kutsu välja vana katkestusfunktsioon */
   (*oldkbdint)();
   /* vana katkestusfunktsioon on juba täitnud klaviatuuripuhvri */
   /* hangi klaviatuuripuhvri lopp */
   tail = peek(0x0040, 0x001C);
   tail -= 2;
   if(tail < 30)
     tail = 60;
   /* loe justvajutatud klahvi ASCII kood ja Scan - kood */
   cChar = peek(0x0040, tail);
   cScanCode = peek(0x0040, tail + 1);
}     /* newkbdint */
```

Standardsed funktsioonid

Selles peatükis käsitletakse C - keele standardseid funktsioone ja päisefaile. Programmeerimiskeeles C ei lisata programmile mingeid standardseid funktsioone. Kõik funktsioonid on salvestatud vastavatesse teekidesse. Nende kasutamiseks tuleb nimetatud teek programmiga siduda (linkida) ja lisada programmile vastav päisefail eeltranslaatori käsuga *#include*. Viimane sisaldab antud teegi funktsioonide deklaratsioone, mitmeid vajalikke sümboolseid konstante, makrosid ja ka teegile vajalikke globaalseid muutujaid. Mõnevõrra keeruline on sobivate päisefailide ja teekide leidmine. Päisefailid on jaotatud teemade järgi. Iga päisefail sisaldab teatud ülesannete grupiga tegelevaid funktsioone. Kuna aga igasse taolisesse gruppi kuulub suurel hulgal funktsioone, siis on loodud ka sellised päisefailid, mis sisaldavad vaid ühte osa nendest funktsioonidest. Üldiselt on soovitav programmi lisada võimalikult vähe päisefaile. See kiirendab translaatori tööd ja vähendab programmi suurust. Nii on osa funktsioonide deklaratsioonid enamates päisefailides. Mõned funktsioonid vajavad teiste funktsioonide abi ja nii sisaldavad ka nende päisefailid kas otse vastavate funktsioonide deklaratsioone või on vajalik päisefail neisse omakorda *#include* käsuga lisatud. Järgnevates peatükkides käsitleme funktsioone vastavalt nende tööülesannetele.

Teegid on jagatud nii mälumudelite kui ka teemade järgi. Kui te vaatate oma kataloogi `\BORLANDC\LIB` sisu, siis näete seal hulganiselt *.LIB (teegid) ja *.OBJ (objektkood) faile.

Iga programmiga seotakse vähemalt üks *.OBJ ja üks *.LIB fail. Selline *.OBJ fail sisaldab programmi alguses täidetavat koodi. See kood kopeerib programmi nime ja talle üleantud parameetrid programmi andmesegmenti. Andmesegmendist saab neid parameetreid lugeda funktsiooni *main()* parameetrite *argc* ja *argv* abil. Vastavalt mälumudelile listakse programmile kas C0S.OBJ, C0M.OBJ, C0C.OBJ, C0L.OBJ või C0H.OBJ. Seejuures näitab objektfaili nime esimese osa viimane täht (S, M, C, L või H), millise mälumudeliga on tegemist. Kõik standardsed funktsioonid on kogutud teeki C<mälumudelit näitav täht>.LIB. See ei tähenda aga, et programmile lisatakse kogu selle teegi sisu. Programmi koodisegmenti kopeeritakse vaid nende funktsioonide kood, mille deklaratsioon on lisatud programmi.
Matemaatilised funktsioonid on salvestatud eraldi teeki MATH<mälumudelit näitav täht>.LIB. Ka need teegid erinevad vastavalt kasutatud mälumudelile. Kui teie arvuti ei sisalda matemaatikaprotsessorit (koprotsessorit), siis tuleb programmiga siduda ka teek: EMU.LIB. See teek on vajalik matemaatiliste funktsioonide kasutamiseks ilma koprotsessori abita. Kui te aga omate koprotsessorit, siis siduge programmiga hoopiski FP87.LIB. See teek kasutab matemaatiliste funktsioonide jaoks koprotsessorit.
Graafikafunktsioone kasutavad programmid vajavad veel teeki GRAPHICS.LIB ja osadesse jaotatavad programmid vajavad teeki OVERLAY.LIB. Kui te soovite luua Microsoft Windowsi programme, siis tuleb teil valida teised *.OBJ ja *.LIB failid, mis enamast sisaldavad oma nimes tähte W, näiteks C0WS.OBJ ja CWS.LIB.
Kui te kasutate oma programmide transleerimiseks Borland C/C++ programmeerimiskeskkonda, siis võite määrata kõik need valikud kindlaks menüü *Options* käskudega. Lihtsa programmi jaoks aitab, kui määrata menüü *Options* dialoogis *Application,* et tegemist on DOSi *.EXE programmiga ja valida seejärel sobiv mälumudel menüü *Options | Compiler* dialoogist *Code Generation*. Tehtud valikud salvestatakse *.PRJ faili. Nüüd "teab" translaator ise, millist C0x.OBJ ja Cx.LIB faili programmiga siduda. Kui teie programmi kasutab graafika- või matemaatilisi funktsioone, siis tuleb ka vastav teek temaga siduda. Seda saate määrata menüü *Options | Compiler* dialoogist *Code Generation*.
Allpool näete üht programmi MAKE andmefaili, mis sobib üsna suurele hulgale programmidele. Teil tuleb lihtsalt määrata sobivad parameetrid. Siintoodud fail on määratud näiteprogrammi INTEGRAL transleerimiseks.

MAKEFILE

```
#-----------------------------------------------------------
#---  MAKEFILE
#---
#--- Andmefail programmile MAKE translaatori juhtimiseks
#-----------------------------------------------------------

#Asendage järgmistel ridadel määratud parameetrid
#teie programmi jaoks sobivatega
```

```
SOURCE = INTEGRAL                #siia tuleb teie programmi nimi
MODEL = s                        #Esialgu SMALL mälumudel.
INCPATH = E:\BORANDC\INCLUDE     #kus asuvad teie päisefailid
LIBPATH = E:\BORLANDC\LIB        #ja teegid
OBJS = C0$(MODEL).OBJ $(SOURCE).OBJ
LIBS = EMU.LIB MATH$(MODEL) C$(MODEL).LIB  #kui te vajate teisi
teeke,
                                 #siis lisage nende nimed siia
CFLAGS = -c -I$(INCPATH)
LFLAGS = /v/x/c/P-/L$(LIBPATH)

$(SOURCE).EXE:  $(SOURCE).OBJ $(SOURCE).C
      TLINK $(LFLAGS) $(OBJS), $(SOURCE).EXE, , $(LIBS),


$(SOURCE).OBJ:    $(SOURCE).C
      BCC  $(CFLAGS)  $(SOURCE).C
```

Sisend-/väljundfunktsioonid

Programmeerimiskeel C sisaldab väga mitmesuguseid sisend-/väljundfunktsioone. Tavaliselt kasutatakse puhverdatud ehk voogudel baseeruvaid funktsioone, kuid on olemas ka lihtsaid funktsioone, mis väljastavad otseselt väljundseadmele või sellelt loevad. Lisaks neile on olemas veel graafilised funktsioonid. Esimesed kaks gruppi on üsna standardsed, kuid viimased, s.o. graafilised funktsioonid on iga translaatori puhul erinevad.

Puhverdatud funktsioonid

Sisend-/väljundfunktsioonide abil väljastatakse andmeid ekraanile, trükkalile ja teistele seadmetele ning loetakse uusi andmeid klaviatuurilt ja teistelt sisend-seadmetelt.
Arvutatud andmeid võib salvestada faili või ka failist uusi andmeid lugeda. Fail on suurem kogum andmeid, mis salvestatakse teie arvuti mingile kettaseadmele. Selline andmeblokk omab nime, suurust ja kindlat asukohta, s.o. kataloogi. Ühes kataloogis ei tohi asuda kahte samanimelist faili. Operatsioonisüsteem DOS määrab, et failide nimed võivad koosneda vaid kuni kaheksast sümbolist, millele võib järgneda punkt ja seejärel jälle kuni kolm sümbolit. Seejuures määravad esimesed kaheksa (või vähem) sümbolit faili tegeliku nime ja viimaseid kolme kasutatakse tavaliselt faili tüübi määramiseks. Faili nimes võib kasutada kõiki numbreid ja inglise tähestiku tähti. Seega ei ole näiteks sümbolid ö, ä, ü ja õ faili nimedes lubatud (DOSi puhul). Lubatud on aga veel sümbolid ~, -, _, $, §, %, #, *, {, }, !. kuid vaid üks punkt.
Kuna sisend- ja väljundseadmed on tunduvalt aeglasemad kui arvuti protsessor, siis loob programm puhverdatud funktsioonide puhul iga seadme ja avatud faili jaoks eraldi mälupuhvri. Seadmele väljastamise puhul salvestatakse väljastatavad andmed kõigepealt sellesse puhvrisse ja alles puhvri täitumise puhul väljastatakse nad üheskoos seadmele. Ka seadmelt või failist lugemise puhul salvestatakse loetud andmed esialgu puhvrisse ja antakse nad alles küllaldase hulga loetud andmete puhul edasi protsessorile. Sellised puhvrid võivad õige kasutamise puhul tunduvalt suurendada programmi töökiirust. Neid puhvreid nimetatakse voogudeks. Vood sisaldavad peale puhvri veel muidki andmeid avatud faili või seadme kohta. Voog sisaldab näiteks andmeid puhvri suuruse kohta ning tähiseid, mis määravad, kas voole väljastamisel on tekkinud mingi viga või mitte. Failidega seotud vood sisaldavad veel tähiseid, mis määravad hetkelise positsiooni failis (järgmise loetava või kirjutatava tähe järjekorranumber faili algusest loetuna) ja tähise, mis määrab, kas voog on jõudnud väljastamisega faili lõppu.
Borland C/C++ translaator avab programmi jaoks viis standardset voogu:

- stdin - Seda voogu kasutatakse andmete lugemisel klaviatuurilt
- stdout - Seda voogu kasutatakse andmete väljastamiseks ekraanile

- stdprn - Seda voogu kasutatakse trükkalile väljastamiseks
- stderr - Seda voogu kasutatakse veateadete väljastamiseks. Tavaliselt väljastatakse veateated ekraanile.
- stdaux - Seda voogu kasutatakse väljundpordile (COM1 - COM4) väljastamiseks.

Kui te kasutate väljastamiseks voogusid, siis annab see teile võimaluse programmi poolt väljastatavaid andmeid ekraani asemel ka faili salvestada või ka lugeda sisestatavaid andmeid klaviatuuri asemel failist. Vood varjavad programmi eest selle, kuhu andmeid tegelikult väljastatakse ja mis kujul seda tehakse. Vood on nummerdatud järgmiselt: *stdin* = 0, *stdout* = 1, *stderr* = 2. Neid numbreid saab kasutada koos sümbolitega > ja < väljastatavate ja sisestatavate andmete ümberjuhatamiseks. Näiteks käsk DIR väljastab kataoogi faililoetelu tavaliselt ekraanile. Käsk

```
DIR > kataloog.txt
```

aga salvestab faililoetelu faili nimega *kataloog.txt*. Lisades sümbolile > numbri, saab määrata, millist voogu ümber juhatada ja kuhu. Kasulik on see näiteks translaatori poolt väljastatavate veateadete ümberjuhatamiseks faili, et neid saaks ka hiljem lugeda. Näiteks:

```
BCC -c hello.c  2> errors.txt
```

Nii saab neid ka hiljem rahus lugeda. Kui veateated aga otse ekraanile väljastatakse, siis võib olla raske neid meeles pidada. Peale selle väljastab translaator vahel nii palju veateateid, et esimesed juba ekraanilt kaovad, kui viimased on veel väljastamata.

Sümboli | abil saab aga ühe programmi väljundit (stdout) siduda teise programmi sisendiga (stdin). Näiteks:

```
DIR | more
```

Siin loob käsk DIR faililoetelu hetkelises kataloogis ja annab need andmed üle programmile MORE, mis esitab neid lehekülje kaupa ekraanile. Sellised programmid töötavad teatavas mõttes filtrina, mis loevad oma sisendvoolt (stdin) andmeid, töötlevad neid ja väljastavad väljundvoole (stdout).

Programmeerimiskeel C tunneb kahesugusid voogusid:

- Tekstivood - Sellised vood on seotud teksti sisaldava faili või ekraaniga. Teksti puhul omavad osa sümbolitest erilist tähendust. Näiteks sümbolid ^M (ASCII nr: 10) ja ^J (ASCII nr: 13) koos tähendavad reavahetust ja failist lugemisel asendatakse need sümbolid sümboliga \n. Tabulaatorid asendatakse vastava arvu tühikutega jne.
- Numbrilised vood - Numbriliste voogude puhul ei oma ükski sümbol erilist tähendust ja seepärast neid ka ei muudeta.

Puhverdatud sisend-/väljundfunktsioonide kasutamiseks tuleb programmi lisada eeltranslaatori käsuga päisefail STDIO.H.

Peatükis *Enamkasutatavad funktsioonid* käsitletud funktsioonid *printf()* ja *scanf()* väljastavad oma andmed voole *stdout* ja loevad voolt *stdin*.

```
int printf(const char *format [, ...]);
int scanf(const char *format [, ...]);
```

Need funktsioonid loevad ja kirjutavad formateeritud andmeid sisend- ja väljundvoole. Päisefailis STDIO.H on deklareeritud ka funktsioonid:

```
int fprintf(FILE *stream, const char *format [, ...]);
int fscanf(FILE *stream, const char *format [, ...]);
```

mis võimaldavad lugeda andmeid mingilt voolt. Selleks tuleb vaid soovitud voog esimese parameetriga määrata. Iga voog kirjeldatakse andmestruktuuri FILE abil. Voog peab enne nende funktsioonide kasutamist olema seotud mingi seadmega. Need funktsioonid loevad või kirjutavad siis sellele seadmele ja kui tegemist on failiga, siis nihutavad nad ka vastavalt edasi hetkelise positsiooni viita.
Funktsioonid *puts()* ja *gets()* on samuti defineeritud päisefailis STDIO.H.

```
int puts(const char * string);
char *gets(char * buffer);
```

Need funktsioonid loevad või kirjutavad terve rea sisend - või väljundvoole. Olemas on ka samalaadsed funktsioonid, mis kirjutavad või loevad terve rea suvalisele määratud voole.

```
inf fputs(FILE *stream, const char *string);
char * fgets(FILE *stream, char *buffer);
```

Päisefailis STDIO.H on defineeritud ka funktsioon *perror()*, mis väljastab mingi sümbolite jada voole *stderr*.

```
int perror(const char *string);
```

Päisefail STDLIB.H sisaldab muutuja *errno* definitsiooni. Seda muutujat kasutavad paljud funktsioonid. Kui funktsioon ei suuda oma ülesannet mingil põhjusel täita, siis ei väljasta ta mitte veateadet (sest te võibolla ei taha seda teadet inglisekeelsena väljastada), vaid salvestab selle vea numbri muutujasse *errno*. Päisefailis STDLIB.H defineeritud muutuja *sys_errlist* sisaldab samuti vigade lühikesi kirjeldusi. Iga kirjeldus on lühike string, mille järjekorranumber massiivis *sys_errlist* vastab selle vea numbrile.
Olles lugenud ühe rea soovitud voolt, võite nüüd kasutada formateeritud andmete lugemiseks või kirjutamiseks sobivaid funktsioone.

```
int sprintf(char *buffer, const char *format [, ...] );
int sscanf(char *buffer, const char *format [, ...]);
```

Need funktsioonid on olemas ka natuke teisel kujul:

```
vprintf(char *format, va_list arguments);
vscanf(char *format, va_list arguments);
```

```
vfprintf(FILE *stream, char *format, va_list arguments);
vscanf(FILE *stream, char *format, va_list arguments);
vsprintf(char *buffer, char *format, va_list arguments);
vsscanf(char *buffer, char *format, va_list arguments);
```

Need funktsioonid erinevad eelpooltoodud funktsioonidest vaid selle poolest, et neile ei saa soovitud parameetreid kohe üle anda, vaid viimane parameeter on viit tegelikele parameetritele. Nimetatud funktsioone on väga mugav kasutada parameetrite sorteerimiseks muutuva parameetrite arvuga funktsioonides. Näiteks oleks sobiv töödelda programmi parameetreid nende funktsioonide abil funktsioonis *main()* .
Üheainsa sümboli (tähe) väljastamiseks või lugemiseks sisend- või väljundvoolt saab kasutada järgmisi makrosid:

```
int getchar( void );
int putchar(int  c);
```

Need makrod kirjutavad ja loevad voogudele *stdin* ja *stdout*, ning on defineeritud järgmiste makrode abil, mis loevad ja kirjutavad suvalisele määratud voole.

```
int getc(FILE *stream);
int putc(int c, FILE *stream);
```

Funktsioonid *fgetc()* ja *fputc()* saavad hakkama sama ülesandega ning vajavad samu parameetreid. Ainus vahe on selles, et tegemist on funktsioonide, mitte makrodega.

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

Funktsioonide kasutamine muudab programmi töö natuke aeglasemaks, kuid makrode massiivne kasutamine muudab programmi suuremaks. Kuna kõvaketas on niikuinii palju aeglasem, siis on soovitav kasutada vastavate makrode abil funktsioone *fgetc()* ja *fputc()* .
Makrod *fgetchar()* ja *fputchar()* väljastavad ja loevad tähti voogudelt *stdin* ja *stdout*. Need makrod kasutavad aga oma töö tegemiseks makrode *getc()* ja *putc()* asemel funktsioone *fgetc()* ja *fputc()* ning vähendavad seega pisut programmi suurust. Makrod *fgetchar()* ja *fputchar()* ei ole aga defineeeritud ANSI C standardis.

Funktsioonid *getw()* ja *putw()* loevad ja kirjutavad numbreid määratud voole.

```
int getw(FILE *stream);
int putw(int w, FILE *stream);
```

Needki funktsioonid ei ole ANSI C standardis defineeritud. Need funktsioonid ei formateeri loetavaid numbreid ja neid ei tohiks kasutada tekstirezhiimis voogude puhul. Nende funktsioonidega saab lugeda ja kirjutada vaid numbrilistele voogudele.

Andmeid võib lugeda või kirjutada ka blokkide kaupa.

```
size_t fread(char *buffer, size_t size, size_t n, FILE *stream);
```

Funktsioon *fread()* loeb viimase parameetriga määratud voolt *n* andmeblokki suurusega *size* baiti ja salvestab nad parameetri *buffer* poolt osutatud puhvrisse. See fuktsioon ei sobi eriti hästi standardsetelt voogudelt lugemiseks, vaid on rohkem ette nähtud failide jaoks .
Funktsioon *fwrite()* kirjutab *n size* baidi suurust andmeblokki parameetriga *stream* määratud voole.

```
size_t fwrite(char *buffer, size_t size, size_t n, FILE *stream);
```

Tüübimäärang *size_t* on tegelikult sama, mis *int* ehk täisarv.
Selleks, et lugeda või salvestada andmeid faili, tuleb fail esmalt avada. Selleks kasutatakse funktsiooni *fopen()*.

```
FILE *fopen(const char *filename, const char *mode);
```

See funktsioon avab parameetriga *filename* määratud faili ja seob ta vastava vooga. Parameeter *mode* määrab, kas tegemist on teksti- või numbrilise vooga, ning kas sellest failist saab vaid lugeda, kirjutada või teha mõlemat. Kui avatav fail ei asu hetkelises kataloogis, siis tuleb määrata faili täielik asukoht parameetriga *filename*. Parameeter *mode* võib sisaldada järgmisi sümboleid:

Väärtus	Tähendus
r	Fail avatakse ainult lugemiseks.
w	Fail avatakse ainult kirjutamiseks. Kui faili ei eksisteeri, siis ta luuakse.
a	Fail avatakse kirjutamiseks ja hetkeline väljastamispositsioon viiakse kohe faili lõppu. Peale faili avamist sinna kirjutatud andmed liidetakse seega seal juba olevatele andmetele. Kui faili ei eksisteeri, siis ta luuakse.
r+	Fail avatakse nii lugemiseks kui kirjutamiseks. Kui faili ei eksisteeri, siis loovutab funktsioon väärtuse NULL.

w+	Fail avatakse nii lugemiseks kui kirjutamiseks. Kui fail on juba olemas, kustutatakse see ja luuakse uus sellenimeline fail.
a+	Fail avatakse nii lugemiseks kui kirjutamiseks ja hetkeline positsioon viiakse kohe faili lõppu. Kui faili ei eksisteeri, siis ta luuakse.
t	Selle sümboli lisamine eelmistele määrab, et fail avatakse tekstirezhiimis.
b	Selle sümboli lisamine eelmistele määrab, et fail avatakse numbrilises rezhiimis.

Tabel 14: Funktsiooni fopen() parameetri mode võimalikud väärtused

Olles faili avanud, võib eelpoolmainitud funktsioone kasutada temaga seotud voole väljastamiseks või sealt lugemiseks. Peale andmevahetust tuleb fail sulgeda funktsiooniga *fclose()*.

```
int fclose(FILE *stream);
```

Kui te olete avanud mitu faili, siis võite funktsiooniga *fcloseall()* sulgeda kõik avatud failid.

```
int fcloseall( void );
```

See funktsioon ei kuulu aga ANSI C standardisse.
Funktsioon *freopen()* suleb parameetriga *stream* määratud vooga seotud faili ja seob ta parameetriga *filename* määrtud failiga.

```
FILE * freopen(const char *filename, const char *mode, FILE *stream);
```

Seda funktsiooni kasutatakse tavaliselt standardsete voogude nagu *stdin*, *stderr* ja *stdout* sidumiseks mingi failiga. Seejärel saab ka funktsioonidega *printf()* ja *scanf()* faili väljastada või sellest lugeda.

Makro *remove()* kustutab parameetriga *filename* määratud faili.

```
int remove(const char *filename);
```

See makro kasutab tegelikult funktsiooni *unlink()*, mis aga ei ole standardne funktsioon.

```
int unlink(const char *filename);
```

Funktsioon *rename()* muudab määratud faili nime.

```
int rename(const char *oldname, const char *newname);
```

Hetkelise väljastamis- või lugemispositsiooni hankimiseks kasutage funktsiooni *ftell()*.

```
long ftell(FILE *stream);
```

Funktsiooni *ftell()* väärtus on hetkelise positsiooni asukoht baitides faili algusest lugedes. Saadud andmeid saab kasutada funktsiooniga *fseek()* hetkelise positsiooni muutmiseks.

```
int fseek(FILE *stream, int offset, int origin);
```

Funktsioon *fseek()* nihutab faili (voo) *stream* hetkelist positsioon täisarvu *offset* võrra, lugedes parameetriga *origin* määratud kohast. See parameeter võib omada järgmisi väärtusi:

- SEEK_SET (0) - faili algusest loetuna
- SEEK_CUR (1) - hetkelisest positsioonist loetuna
- SEEK_END (2) - faili lõpust loetuna

Sama tööga saavad hakkama ka funktsioonid *fgetpos()* ja *fsetpos()*.

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

Tüüp *pos_t* on failis (voos) hetkelise positsiooni salvestamiseks ettenähtud tüüp.
Funktsioon *rewind()* nihutab hetkelise positsiooni faili algusesse ja kustutab faili lõppu ja väljastamise viga näitavad tähised.

```
void rewind(FILE *stream);
```

Funktsioon feof() uurib, kas voog on jõudnud lugemisega faili lõppu.

```
int feof(FILE *stream);
```

Kui hetkeline positsioon on jõudnud faili lõppu, siis on selle funktsiooni väärtus nullist erinev, vastasel juhul null. Kirjutamiseks avatud faili puhul ei oma faili lõpp suurt tähendust, kuna iga kirjutamisega fail suureneb. Ainult lugemiseks avatud faili puhul aga tähendab faili lõppu jõudmine, et iga järgmine lugemisoperatsioon loovutab veateate.
Funktsioon *ferror()* uurib, kas voos on väljastamisel või lugemisel tekkinud mingi viga.

```
int ferror(FILE *stream);
```

Funktsioon *clearerr()* kustutab määratud voo veatähise.

```
void clearerr(FILE *stream);
```

Vahel ei mahu kõik andmed enam mällu ja siis tuleb osa andmeid salvestada ajutiselt faili . Selleks luuakse ajutine fail. Selleks, et olla kindel, et sellenimelist faili selles kataloogis veel ei ole ja et seega mingeid vajalikke andmeid üle ei kirjutata, kasutatakse funktsiooni *tmpnam()*.

```
char *tmpnam(char *buffer);
```

Funktsioon *tmpnam* salvestab loodud nime parameetri *buffer* poolt osutatud puhvrisse. Kui see parameeter on NULL, siis salvestatakse nimi kuskil mujal programmi andmesegmendis ja loovutatakse sellele nimele näitav viit. Seda nime saab seejärel kasutada funktsioonis *fopen()* vajaliku faili avamiseks.
Funktsioon *tmpfile()* täidab kõik nimetatud ülesanded

```
FILE *tempfile( void );
```

See funktsioon loob kõvakettal (hetkelise kataloogis) ajutise faili, millele antakse selline nimi, mida selles kataloogis veel ei esine ja loodud fail avatakse numbrilises rezhiimis ("w+b"). Niipea kui see fail suletakse, kustutatakse ta ka automaatselt kettalt. Kui te ajutisi faile ei kustuta, siis kustutatakse nad peale programmi lõppu automaatselt.
Ajutiste failide kustutamiseks võib kasutada funktsiooni *rmtmp()*.

```
int rmtmp( void );
```

See funktsioon kustutab hetkelises kataloogi kõik hetkel avatud ajutised failid, mis on loodud funktsiooniga *tmpfile()*.

Puhverdatud funktsioonide puhul ei ole programmeerijal vaja puhverdamise üksikasjade üle pead murda, kuid vajaduse korral on ka neid parameetreid võimalik määrata. Funktsiooniga *setbuf()* saab omistada voole uue puhvri.

```
void setbuf(FILE *stream, char *buffer);
```

Muidugi tuleb see puhver kõigepealt mälust reserveerida. Kui parameeter *buffer* on NULL, siis määratud voos enam puhverdamist ei kasutata. Kui nimetatud parameeter aga viitab mingile programmi poolt kasutatud mälublokile, siis peab viimane olema vähemalt *BUFSIZ* baidi suurune. Konstant *BUFSIZ* on defineeritud päisefailis STDIO.H.
Funktsioon *setvbuf()* võimaldab soovitud puhverdamise liiki veelgi täpsemalt määrata.

```
int setvbuf(FILE *stream, char *buffer, int type, size_t size);
```

Selle funktsiooni kasutamise puhul ei ole enam vaja kõigepealt reserveerida sobiva suurusega puhvrit. Kui parameeter *buffer* on NULL, siis reserveerib funktsioon *setvbuf()* parameetri *size* baidi suuruse puhvri. Parameeter *type* määrab soovitud puhverdamise tüüpi ja võib omada järgmisi väärtusi:

- _IOFBF - Täielik puhverdamine kogu puhvri ulatuses.
- _IOLBF - Voog puhverdatakse reakaupa. Andmeid kogutakse puhvrissse kuni rea lõpumärgini ja alles seejärel väljastatakse seadmele või antakse programmile edasi.
- _IONBF - Puhverdamist ei kasutata

Nimetatud väärtused on samuti sümboolsed konstandid ja on defineeritud päisefailis STDIO.H.

Puhverdamine võib ka tekitada raskesti leitavaid vigu. Kui te väljastate ja loete ühelt ja samalt voolt, siis võib järgnev lugemisoperatsioon loovutada need andmed, mida just äsja kirjutati (mitte aga need, mida te vajate) või lugeda endisi andmeid (enne kui uued andmed on tegelikult kettale salvestatud. Seepärast on soovitav enne voolt lugemist kindlustada, et puhver oleks tühi. Selleks kasutatakse funktsiooni *fflush()*.

```
int fflush(FILE *stream);
```

See funktsioon tühjendab voopuhvri, s.t. kirjutab kõik veel salvestamata andmed seadmele või faili. Funktsioon *fflushall()* täidab sama ülesande kõigi hetkel avatud voogudega.

```
int fflushall( void );
```

Puhverdamata funktsioonid

Puhverdatud funktsioonid on väga töökindlad. Nad kasutavad BIOSi ja DOSi funktsioone ning töötavad seega igal arvutil, millele DOS on installeeritud. Sageli on nad aga natuke aeglased, seda eriti ekraanile väljastamisel. Mitte puhverdamine, aga just BIOSi ja DOSi funktsioonid on ekraanile väljastamisel veidi aeglased. Nad ootavad enne väljastamist, kuni ekraanipinda uuendav elektronkiir on antud kohast möödunud. See tehnika väldib vanematel kuvaritel (CGA) nn. lumesaju efekti tekkimist, kuid EGA ja VGA kuvaritel on ta täiesti tarbetu. Seepärast kasutavad paljud programmid tõsiasja, et tekstirezhiimis salvestatakse ekraanisisu teatud mälupiirkoda, mille aadress on täpselt teada. Mustvalgetel monitoridel algab ekraanipuhver aadressilt B000:0000 hex, graafilistel monitoridel aga aadressilt B800:0000 hex. VGA ja EGA monitoridel algab tekstipuhver aadressilt A000:0000, väljaarvatud siis, kui nad on CGA rezhiimis, millal ka nende tekstipuhver algab aadressilt B800:0000. Ekraanipuhvri aadressi saab uurida järgi ka DOSi andmesegmendist või spetsiaalse funktsiooni abil.

Otseselt ekraanimälusse kirjutamiseks ei ole aga hoopiski vaja kasutada assemblerkeelt. Ka programmeerimiskeel C sisaldab funktsioone, mis kirjutavad otseselt ekraanile. Nende funktsioonide deklaratsioonid asuvad päisefailis CONIO.H.

Funktsioonid *cgets()* ja *cputs()* kirjutavad soovi korral ekraanile kas otse või BIOSi funktsiooni abil sõltuvalt globaalse muutuja *directvideo* väärtusest.

```
char *cgets(char *buffer);
int cputs(char *string);
int   directvideo;
```

Muutuja *directvideo* väärtus on algselt 0, mis tähendab et kasutatakse BIOSi funktsioone. Kui te soovite otse ekraanile väljastada, siis omistage sellele muutujale mingi nullist erinev väärtus.

Need funktsioonid omavad ka muid erinevusi. Enne funktsiooni *cgets()* kasutamist seadke puhvri *buffer* esimese elemendi (buffer[0]) väärtuseks maksimaalne sümbolite arv, mis puhvrisse mahub. Funktsioon *cgets()* salvestab tegelikult loetud sümbolite arvu puhvri teise elementi (buffer[1]) . Sümbolid salvestatakse alates kolmandast elemendist. Seega peab puhver mahutama peale sümbolite veel kaks elementi ja lõpumärgina nulli. Funktsioon *cputs()* ei konverteeri üleantud revahetusmärke '\n' DOSi failide vastavateks sümbolitepaariks ASCII(10) ja ASCII(13). Nimetatud funktsioonid kirjutavad ainult ekraanile või loevad klaviatuurilt ja nende sisendit või väljundit ei saa faili ümber juhatada. Need funktsioonid ei vasta ka ANSI C standardile.
Sama võib mainida ka funktsioonide *cscanf()* ja *cprintf()* kohta. Ka nemad ei ole standardsed, kuid on üsna kasulikud.

```
int cscanf(const char *format, [, argument ...]);
int cprintf(const char *format, [, argument ...]);
```

Need funktsioonid kasutavad samu parameetreid, nagu eelmises peatükis vaadeldud vastavad funktsioonid, kuid kirjutavad otse ekraanile või loevad klaviatuurilt vastavalt muutuja *directvideo* väärtusele.
Päisefail CONIO.H sisaldab veel palju muid kasulikke funktsioone tekstirezhiimis programmide jaoks. Enamik neist funktsioonidest ei ole standardsed. Need funktsioonid väljastavad otse ekraanile või loevad klaviatuurilt. Ekraanile väljastamisel kasutatakse nn. akent. Tavaline VGA kuvar mahutab harilikus tekstirezhiimis 25 rida teksti, millest igaüks on 80 sümboli laiune. Vastavalt kuvari ja graafikakaardi tüübile võib kasutada olla ka suurema või väikesema lahutusvõimega tekstirezhiime. Te võite ka ise piirata programmi poolt väljastatud andmeid mingisse ekraanipiirkonda. Selleks tuleb taoline piirkond (aken) luua funktsiooniga *window()*.

```
void window(int x1, int y1, int x2, int y2);
```

See funktsioon loob ekraanile piirkonna (akna), mille vasakpoolne ülemine nurk asub punktis (x1, y1) ja parempoolne alumine nurk punktis (x2, y2). Kuvari vasakpoolne ülemine nurk omab koordinaate (1, 1). X- telg on suunatud vasakult paremale ja Y - telg ülevalt alla. Näiteks käsk *window(1, 1, 80, 25)* loob tavaliselt tervet ekraani hõlmava akna. Kõik järgnevalt väljastatud andmed esitatakse sellesse aknasse. Kui andmete väljastamisega on jõutud akna viimasele reale, siis nihutatakse akna sisu automaatselt ühe rea võrra ülespoole ja akna alumisele äärele ilmub uus vaba rida. Kui funktsioonile *window()* üleantud koordinaadid on kehtetud (näiteks liiga suured), siis ei oma funktsioon mingit toimet ja akna endine suurus jääb kehtima. Väikseim võimalik aken on ühe rea ja ühe tulba laiune. Selle funktsiooniga saate korraga luua vaid ühe akna, mis siis kehtib kõigile ekraanile väljastavatele funktsioonidele. Kui te olete, näiteks, loonud ekraani keskel väikese akna, siis peate muu ekraanipiirkonna muutmiseks looma uue akna, mis seda piirkonda sisaldab.
Aknas kehtivad uued koordinaadid. Akna vasakpoolne ülemine nurk omab alati koordinaate (1, 1). Funktsioonid *cputs()* ja *cprintf()* kasutavad nn. hetkelist

positsiooni, s.t. nad väljastavad oma andmed eelmiste andmete järele ning nihutavad hetkelise positsiooni enda poolt väljastatud andmete lõppu. Hetkelise positsiooni saab teada funktsioonide *wherex()* ja *wherey()* abil.

```
int wherex( void );
int wherey( void );
```

Need koordinaadid kehtivad hetkelise akna sees ja mitte kogu ekraani ulatuses. Hetkelist posistsiooni saab muuta funktsiooniga *gotoxy()*.

```
void gotoxy(int x, int y);
```

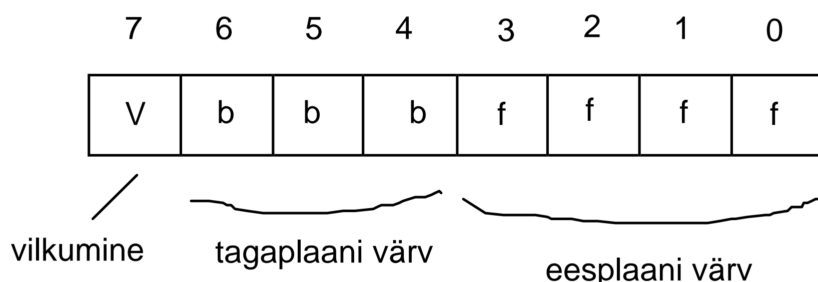
Seejärel väljastavad järgmised funktsioonid oma andmed ekraanile alates sellest aknapunktist.
Ekraanile esitatud andmed võivad olla normaalse heledusega, heledamad või tumedamad. Heledusega näidatakse tavaliselt teksti, mis omab mingit erilist tähendust. Funktsioonid *normvideo()*, *highvideo()* ja *lowvideo()* muudavad järgnevalt väljastatava teksti heldust.

```
void normvideo( void );
void highvideo( void );
void lowvideo( void );
```

Funktsioon *highvideo()* muudab temaga väljastatava teksti heledamaks, *lowvideo()* muudab selle tumedamaks ja *normvideo()* taastab hariliku heleduse.
Tekstirezhiimis kasutatakse kahte värvi: eesplaani - ja tagaplaani värvi. Eesplaani värviga esitatakse sümbolid ja tagaplaani värviga täidetakse sümbolite vahed. Need värvid võivad olla erinevad iga ekraanile esitatava sümboli jaoks. Nende värvide muutmiseks kasutatakse funktsioone *textcolor()*, *textbackground()* ja *textattr()*.

```
void textcolor(int newcolor);
void textbackground(int newcolor);
void textattr(int newattr);
```

Sümbolid reserveerivad ekraanimälus kaks baiti. Esimene neist sisaldab sümboli atribuudi ja teine tema ASCII koodi. Sümboli atribuut määrab nii sümboli eesplaani kui ka tagaplaani värvi ning selle, kas sümbol vilgub või mitte. Joonisel 9 näete sümboli atribuudi bittide tähendust.



Joonis 9: Ekraanile kuvatud sümboli atribuudi bittide tähendused

Nagu näete, on eesplaani värvi määramiseks 4 bitti, aga tagaplaani värvuse jaoks vaid 3 bitti. Seega saab määrata kuni 16 erinevat eesplaani ja 8 tagaplaani värvi. Kui atribuudi viimane bitt on 1, siis tekst vilgub ekraanil. Te võite soovitud andmed seada korraga funktsiooniga *textattr()* või kasutada funktsioone *textcolor()* ja *textbackground()* ning värvide määramiseks failis CONIO.H defineerituid sümboolseid konstante. Neid konstante näete tabelis 14.

Konstant	Number	Tähendus
BLACK	0	Must. Sobib nii ees- kui tagaplaani jaoks.
BLUE	1	Sinine. Sobib nii ees- kui tagaplaani jaoks.
GREEN	2	Roheline. Sobib nii ees- kui tagaplaani jaoks.
CYAN	3	Tumelilla. Sobib nii ees- kui tagaplaani jaoks.
RED	4	Punane. Sobib nii ees- kui tagaplaani jaoks.
MAGENTA	5	Roosa. Sobib nii ees- kui tagaplaani jaoks.
BROWN	6	Pruun. Sobib nii ees- kui tagaplaani jaoks.
LIGHTGRAY	7	Helehall. Sobib nii ees- kui tagaplaani jaoks.
DARKGRAY	8	Tumehall. Sobib ainult eesplaani jaoks.
LIGHTBLUE	9	Helesinine. Sobib ainult eesplaani jaoks.
LIGHTGREEN	10	Heleroheline. Sobib ainult eesplaani jaoks.
LIGHTCYAN	11	Helelilla. Sobib ainult eesplaani jaoks.
LIGHTRED	12	Helepunane. Sobib ainult eesplaani jaoks.
LIGHTMAGENTA	13	Heleroosa. Sobib ainult eesplaani jaoks.
YELLOW	14	Kollane. Sobib ainult eesplaani jaoks.
WHITE	15	Valge. Sobib ainult eesplaani jaoks.
BLINK	128	Vilkuv. Sümboli muutmiseks vilkuvaks liitke see konstant eesplaanivärvusele .

Tabel 15: Sümboolsed konstandid tekstivärvuse määramiseks

Niimoodi määratud värvused kehtivad kõigile järgnevalt väljastatavatele andmetele. Iga kord, kui soovite väljastada andmeid teiste värvidega, peate neid muutma ja hiljem jälle tagasi muutma.
Hetkeliste värvuste, atribuudi jms. teada saamiseks kasutage funktsiooni *gettextinfo()*.

```
void gettextinfo(struct text_info *andmed);
```

Selle funktsiooni ainus parameeter osutab andmestruktuurile *textinfo*, kuhu vajalikud andmed salvestatakse.

```
struct text_info {
  unsigned char winleft;        /* akna vasak piir (x1) */
  unsigned char wintop;         /* akna ülemine piir (y1) */
  unsigned char winright;       /* akna parempoolne piir (x2) */
  unsigned char winbottom;      /* akna alumine piir (y2) */
  unsigned char attribute;      /* teksti atribuut */
  unsigned char normattr;       /* harilik atribuut */
  unsigned char currmode;       /* hetkeline tekstirezhiim:
```

```
                                    BW40, BW80, C40, C80, or C4350 */
  unsigned char screenheight;   /* ekraani kõrgus  */
  unsigned char screenwidth;    /* ekraani laius */
  unsigned char curx;           /* hetkeline X koordinaat */
  unsigned char cury;           /* hetkeline Y koordinaat */
};
```

Selle sturktuuri element *currmode* näitab, millises tekstirezhiimis ekraan parajasti asub. Tekstirezhiimid erinevad lubatud värvuste ja korraga esitatavate sümbolite hulga poolest. Järgmises tabelis näete teksti kuvamisrezhiimide lühiiseloomustust.

Tekstirezhiim	Omadused
BW40	Mustvalge rezhiim. Ekraanile mahub 25 rida, igas 40 sümbolit. See rezhiim on tüüpiline CGA monitoridele.
BW80	Mustvalge rezhiim. Ekraanile mahub 25 rida, igas 80 sümbolit. See rezhiim on tüüpiline mustvalgetele EGA ja VGA monitoridele.
C40	Värviline rezhiim. Ekraanile mahub 25 rida, igas 40 sümbolit.
C80	Värviline rezhiim. Ekraanile mahub 25 rida, igas 80 sümbolit.
C4350	Värviline rezhiim. Ekraanile mahub 43 (EGA) või 50 (VGA) rida, igas 80 sümbolit.
LASTMODE	Viimatikasutatud tekstirezhiim.

Tabel 16 : Tekstirezhiimide omadused

Nimetatud tekstirezhiime saate valida funktsiooniga *textmode()*.

```
void textmode(int newmode);
```

Funktsioon viib ekraani soovitud tekstirezhiimi (kui antud monitor seda suudab) ja taastab harilikud tekstiatribuudid. Seejuures määratakse selline aken, mis hõlmab tervet ekraani, määratakse harilik tekstiheledus (nagu funktsiooniga *normvideo()*) ja harilikud värvid. Seda funktsiooni tohib kasutada vaid siis, kui ekraan on juba tekstirezhiimis. Kui te olete enne viinud ekraani graafilisse rezhiimi, siis tuleb kõigepealt taastada tekstirezhiim funktsiooniga *restorecrtmode()* ja alles siis kasutada funktsiooni *textmode()*.
Päisefail CONIO.H sisaldab ka rohkesti funktsioone ekraani osaliseks või täielikuks puhastamiseks.

```
void clrscr( void );
void clreol( void );
void delline( void );
```

Funktsioon *clrscr()* puhastab kogu hetkelise akna sisu. Funktsioon *clreol()* puhastab ekraani hetkelisest positsioonist kuni rea lõpuni, nihutamata seejuures hetkelist positsiooni. Funktsioon *delline()* puhastab kogu hetkelise rea ja nihutab alumisi ridu ühe rea võrra ülespoole.

Funktsioon *insline()* seevastu sisestab hetkelisele positsioonile uue tühja rea ja nihutab kõik ülejäänud read ühe võrra allapoole.

```
void insline( void );
```

Üksiku sümboli lugemiseks klaviatuurilt kasutage funktsioone *getch()* ja *getche()*.

```
int getch( void );
int getche( void );
```

Mõlemad funktsioonid loevad üheainsa sümboli otse klaviatuurilt, kuid funktsioon *getche()* väljastab loetud sümboli ekraanile ja funktsioon *getch()* mitte. Kui see sümbol osutus ebavajalikuks või teda on vaja uuesti töödelda, siis võib ühe sümboli funktsiooniga *ungetch()* "klaviatuurile taastada" .

```
int ungetch(int ch);
```

See funktsioon salvestab talle üleantud sümboli eraldi puhvris, nii et järgmine funktsioon *getch()* või *getche()* loovutab just selle sümboli. Nii saab salvestada aga ainult ühe sümboli. Seda funktsiooni kasutatakse sageli näiteks translaatorite loomisel.
Funktsioon *kbhit()* uurib, kas kasutaja on vajutanud mingile klahvile, kuid ei eemalda seda sümbolit operatsioonisüsteemi vastavast puhvrist.

```
int kbhit( void );
```

Kui te teate, et kasutaja on vajutanud mingile klahvile, siis võite selle sümboli lugeda funktsiooniga *getch()* või *getche()*. Nii võite luua tsükli, mis kontrollib, kas mingit klahvi on vajutatud ja alles seejärel loeb selle sümboli klaviatuuripuhvrist.
Ühe sümboli väljastamiseks ekraanile kasutage funktsiooni *putch()*.

```
int putch(int ch);
```

Selleks, et lihtsustada ekraanile mitmete nihutatavate akende moodustamist, sisaldab Borland C/C++ ka funktsioone *gettext()* ja *puttext()*.

```
int puttext(int x1, int y1, int x2, int y2, const char *puhver);
int gettext(int x1, int y1, int x2, int y2, char *puhver);
```

Funktsioon *gettext()* kopeerib ekraaniristküliku (x1, y1) (x2, y2) parameetri *puhver* poolt osutatud puhvrisse. Iga sümboli jaoks on vaja 2 baiti (tähe kood ja atribuut). Puhver peab seega mahutama ((x2 - x1) * (y2 - y1)) * 2 baiti. Kopeeritud andmed salvestatakse puhvrisse järjestikku rida rea järel. Funktsioon *puttext()* kopeerib sellise puhvri sisu uuesti ekraanile. Kui iga aken salvestab oma tagapõhja enne selle muutmist ja taastab ta ekraanilt kustutamisel, siis saab neid aknaid ekraanil nihutada ilma seda muutmata. Seda aga vaid tingimusel, et aknad ei kattu või et selline teisi aknaid varjav aken saab ekraanil viibida vaid ajutiselt (dialoog).

Päisefailis CONIO.H on defineeritud ka funktsioonid sisend- ja väljundportide kasutamiseks. Neid porte ei tohi ära vahetada järjestikuste ja paralleelsete portidega (COM1 ... COM4, LPT, LPT2). Arvuti protsessor on otseselt seotud vaid mäluga. Kõigi sisend- ja väljundseadmetega on aga protsessor seotud portide kaudu. Nii on ka graafikakaart, kettaajurite kontroller jms. seotud teatud portidega. Pordid on nummerdatud. Arvuti omab 256 porti. Port on nagu teatud kindel aadress, mille väljastatud andmed antakse üle temaga seotud seadmele. Tavaliselt ei vaja te otsest portidele väljastamist või neilt lugemist. Hiire kasutamine nõuab andmete lugemist pordilt (mis on omakorda sisend-väljundkontrolleri kaudu seotud ühe järjestikulise pordiga). Kogu selle töö teeb aga hiire ohjurprogramm. Nii on hiire kasutamiseks küllaldane ohjurprogrammi funktsioonide kasutamine kindlate katkestuste abil

```
int inp(unsigned portno);
unsigned inpw(unsigned portno);
int outp(unsigned portno, int value);
unsigned outpw(unsigned portno, unsigned value);
```

Näiteprogramm HEXEDIT

Järgmine näiteprogramm näitab puhverdatud ja puhverdamata tekstirezhiimi funktsioonide kasutamist ja palju muudki. See programm on tilluke editor kuueteistkümnendkoodis failide jaoks, mis võimaldab teil neid faile lugeda, editeerida ja salvestada. Editor kasutab käskude sisestamiseks ja kursori positsioneerimiseks ekraanil ka hiirt. Ekraani üleval ääres asub viie käsuga menüü: *Ava, Salvesta, Muuda nimi, Trüki ja Lõpp*. Neid käske saab kasutada, kui vajutada klahvile <ALT>, hoida seda allavajutatuna ja vajutada vastava menüükäsu nime esimesele tähele. Näiteks <ALT>+<A> avab uue faili. Menüükäskude esimesed tähed on nende paremaks eristamiseks esitatud ekraanile punase värviga ja suurtähtedega. Ekraani viimast rida kasutage teadete väljastamiseks. Kui näiteks soovitud nimega faili ei leita, siis ilmub sellele reale vastav teade. Menüükäskude *Ava* ja *Muuda nimi* puhul kasutatakse seda rida ka uue faili nime sisse lugemiseks. Raamiga ümbritsetud piirkond ekraani keskosas moodustab editoriakna. Selles aknas esitatakse sisseloetud faili sümbolid ekraanile kuueteistkümne kaupa reas. Iga rea alguses asub aadresss, mis määrab selle rea esimese sümboli kauguse faili algusest. Sellele järgnevad kuusteist sümbolit kuueteistkümnendarvudena. Rea lõpus näidatakse need samad sümbolid ka ASCII tähtedena. Sümbolite asemel, mis ei vasta tähtedele või numbritele, esitatakse selles veerus punktid.
Hiire vasaku klahviga võite klõpsutada mingi menüükäsu nime kohal, mis vastab sama käsu valimisele klaviatuurilt. Kui klõpsutate hiire vasaku klahviga editoriaknas, siis nihutatakse kursor antud kohta.
Kui te vajutate mingile klahvile (ilma <ALT> klahvi kinni hoidmata), siis sisestatakse vastav sümbol ekraanile kursori asukohta ja kursor nihutatakse ühe positsiooni võrra paremale. Kui kursor on jõudnud rea lõppu, niis nihutatakse ta järgmise rea algusesse. Te ei saa sümboleid failist kustutada, vaid peate nad vajaduse korral uute sümbolitega üle kirjutama.

Kursori nihutamiseks editoriaknas võite kasutada ka nooleklahve ning klahve <PgUp>, <PgDn>, <Home> ja <End>.
Hiire kasutamiseks liideti sellesse programmi ka eelpool toodud failid MOUFUNC.C ja MOUFUNC.H.

HEXEDIT.C

```
/*******************************************************/
/***                                                 ***/
/***   HexEdit.C                                     ***/
/***                                                 ***/
/***   See fail sisaldab näiteprogrammi HEXEDIT       ***/
/***   pohiprogrammi. Nimetatud programm demonstreerib ***/
/***   puhverdatud ja puhverdamata funktsioonide      ***/
/***   kasutamist, mälu reserveerimist ja kasutab ka  ***/
/***   eespool loodud funktsioone hiire juhtimiseks.  ***/
/***                                                 ***/
/*******************************************************/

/*----------------< Päisefailid >-----------------------*/

#include    <conio.h>
#include    <stdio.h>
#include    <string.h>
#include    <stdlib.h>
#include    "moufunc.h"

#define  ID_NOCOMMAND   0    /* sümboolsed konstandid */
#define  ID_OPEN  1       /* menüü- ja teiste käskude jaoks */
#define  ID_SAVE  2
#define  ID_SAVEAS      3
#define  ID_PRINT 4
#define  ID_QUIT  5
#define  ID_LINEUP      6
#define  ID_LINEDOWN    7
#define  ID_PAGEUP      8
#define  ID_PAGEDOWN    9
#define  ID_LEFT  10
#define  ID_RIGHT 11
#define  ID_HOME  12
#define  ID_END         13


/*----------------< Globaalsed muutujad >----------------*/

struct Menustruct {
  char name[12];
  int  nr;
  int  x1, x2;
} MENU[] = {      /* menüükäsud, käsunumbrid ja menüü koordinaadid */
  "Ava",    ID_OPEN, 0, 0,
  "Salvesta",   ID_SAVE, 0, 0,
  "Muuda nimi", ID_SAVEAS, 0, 0,
  "Trüki",      ID_PRINT, 0, 0,
  "Lopeta",     ID_QUIT, 0, 0,
  NULL,     ID_NOCOMMAND, 0, 0,
};

char  szFileName[250];       /* avatud faili nimi */
char  *pMem;                 /* viit mäluphvrile */
int   nFirstLine;            /* esimese rea number alates nullist */
int   fMouse = 1;            /* 1, kui hiire ohjurprogramm on olemas
*/
int     fDirty = 0;          /* 1, kui teksti on muudetud */
int     CurX, CurY;          /* hetkelised koordinaadid tekstis */
struct  text_info  tinf;     /* tekstirezhiimi andmed */
```

```
long  size;                   /* teksti suurus baitides */


/*----------------< Funktsioonide prototüübid >----------*/

void DrawScreen( void );      /* Joonistab menüü ja editoriakna */
int  GetCommand( void );      /* hangib uue käsu */
void DoCommand(int);          /* täidab selle käsu */
void message(char *);         /* esitab teate alumisele reale */
void RedrawText( void );      /* uuendab editoriakna sisu */
void OpenFile(char *);        /* avab faili ja loeb selle mällu */
void PrintFile( void );       /* trükib faili */
void PutCursor(int, int);     /* asetab kursori ekraanile */
int GetFilename( char * );    /* hangib uue failinime */
int WriteFile( char * );      /* salvestab faili */
void PutChar(char ch);        /* sisestab sümboli faili ja ekraanile
*/


/*=====================================================*/
/*===                                               ===*/
/*===   main()                                      ===*/
/*===                                               ===*/
/*===   Töötleb programmi parameetreid, initsialiseerib ===*/
/*===   globaalsed muutujad ja alustab käskude      ===*/
/*===   täitmist. Niipea, kui kasutaja sisestab käsu ===*/
/*===   Lopp, lopeb ka programmi töö.               ===*/
/*=====================================================*/
#pragma argsused
int main(int argc, char **argv)
{
  int command;
  nFirstLine = 0;       /* initsialiseeri muutujad */
  size = 0;
  pMem = NULL;
  CurX = CurY = 1;
  gettextinfo(&tinf);
  szFileName[0] = 0;
  directvideo = 1;     /* joonista editoriaken ekraanile */
  DrawScreen();
  if(!MouInit()) {     /* initsialiseeri hiire ohjurprogramm */
    message("Hiire ohjurprogrammi ei ole starditud!");
    fMouse = 0;
  }
  if(fMouse) {         /* inverteeritud hiirekursor */
    MouSetAlNumCursor(0, 0xFF, 0xFF, 0, 0x77);
    MouGotoXY(100, 320);
    MouShow();
  }
  if(argc > 1)         /* töötle programmi parameetreid */
    OpenFile(argv[1]);
  while(ID_QUIT != (command = GetCommand()))
    DoCommand(command);
  MouHide();     /* taasta enne programmi loppu endine olukord  */
  if(pMem)       /* vabasta reserveeritud mälu */
    free(pMem);
  textcolor(BLACK);
  textbackground(WHITE);
  clrscr();
  return 0;
}     /* main */


/*---------------------------------------------------*/
/*---                                             ---*/
/*---   GetCommand()                              ---*/
/*---                                             ---*/
/*---   Loeb sisse uue käsu klaviatuurilt.        ---*/
/*---------------------------------------------------*/
int GetCommand( void )
```

```
{
  char      buf[10];
  int x, y, ch;

  /* niikaua, kuni kasutaja ei ole vajutanud mingile klahvile,
     kontrollime, kas hiire vasakule klahvile ei ole vajutatud
     ja kui see on nii, siis üritame toelgendada seda mingi käsuna */
  while(!kbhit())
    if(fMouse) {
      if(!MouGetPressed(MOU_GETLBUTTONINFO, &y, &x))
      continue;
      x /= 8; y/= 8;    /* hiire ohjurprogramm kasutab graafilise
      rezhiimi koordinaate ja neid tuleb enne kasutamist teisendada*/
      if(y == 0) {      /* menüükäsud */
      if((x >= MENU[0].x1) && (x <= MENU[0].x2))
        return ID_OPEN;
      if((x >= MENU[1].x1) && (x <= MENU[1].x2))
        return ID_SAVE;
      if((x >= MENU[2].x1) && (x <= MENU[2].x2))
        return ID_SAVEAS;
      if((x >= MENU[3].x1) && (x <= MENU[3].x2))
        return ID_PRINT;
      if((x >= MENU[4].x1) && (x <= MENU[4].x2))
        return ID_QUIT;
      }                 /* kursori uus positsioon */
      if((x >= 10) && (x <= 58) &&
       (y >= 1) && (y <= tinf.screenheight - 3)) {
      CurY = y - 1 + nFirstLine;
      CurX = (x - 10) / 3 + 1;
      PutCursor(CurX, CurY);
      }
    }
  if(0 == (ch = getch()))  /* ALT klahvi puhul on esimene bait null*/
    ch = getch();          /* see oleks mingi käsu number */
  else {
    PutChar(ch);           /* vastasel juhul on tegemist */
    return ID_NOCOMMAND;   /* sisestatava sümboliga */
  }
  switch(ch) {
    case 38:               /* ALT+l -> lopeta programmi töö */
      return ID_QUIT;
    case 30:               /* ALT+a -> ava fail */
      return ID_OPEN;
    case 31:               /* ALT+s -> salvesta sama nime all */
      return ID_SAVE;
    case 50:               /* ALT+m -> muuda nimi ja salvesta */
      return ID_SAVEAS;
    case 20:               /* ALT+t -> trüki fail */
      return ID_PRINT;
    case 80:
      return ID_LINEDOWN;
    case 72:
      return ID_LINEUP;
    case 75:
      return ID_LEFT;
    case 77:
      return ID_RIGHT;
    case 81:
      return ID_PAGEDOWN;
    case 73:
      return ID_PAGEUP;
    case 71:
      return ID_HOME;
    case 79:
      return ID_END;
    default:                    /* tundamtu käsk -> ära tee midagi */
      return ID_NOCOMMAND;
  }
}     /* GetCommand() */
```

```
/*-------------------------------------------------------*/
/*---                                                 ---*/
/*---   DrawScreen()                                  ---*/
/*---                                                 ---*/
/*---   Joonistab ekraanile menüü ja editoriakna.     ---*/
/*-------------------------------------------------------*/
void DrawScreen( void )
{
  int             x, y;

  window(1, 1, tinf.screenwidth, tinf.screenheight);
  /* Joonista menüüriba */
  gotoxy(1, 1);
  textbackground(LIGHTGRAY);
  clrscr();                     /* täida tagapohi */
  gotoxy(1, 1);  x = 0;
  while(MENU[x].nr != 0) {      /* joonista menüükäskude nimed */
    putch(' '); putch(' ');     /* kaks tühikut */
    MENU[x].x1 = wherex();
    textcolor(RED);             /* esimene täht olgu punane */
    putch(MENU[x].name[0]);
    textcolor(BLACK);           /* ülejäänud osa on must */
    cputs(MENU[x].name + 1);
    MENU[x].x2 = wherex() - 1;
    x++;
  }
  textbackground(BLUE);
  textcolor(WHITE);             /* joonista editoriaken */
  gotoxy(1, 2);
  putch('+');
  for(x = 2; x < tinf.screenwidth; x++) putch('-');
  putch('+');
  if(szFileName[0]) {
    x = strlen(szFileName);
    gotoxy(tinf.screenwidth / 2 - x / 2 ,2);
    cputs(szFileName);
  }
  for(y = 3; y < tinf.screenheight-1; y++) {
    gotoxy(1, y);
    clreol();
    putch('¦');
    gotoxy(tinf.screenwidth, y);
    putch('¦');
  }
  gotoxy(1, tinf.screenheight-1); putch('+');
  for(x = 2; x < tinf.screenwidth; x++) putch('-');
  putch('+');
  window(2, 3, tinf.screenwidth-1, tinf.screenheight-2);
}     /* DrawScreen() */

/*-------------------------------------------------------*/
/*---                                                 ---*/
/*---   message()                                     ---*/
/*---                                                 ---*/
/*---   Näitab akna alumisel real mingit teadet.      ---*/
/*-------------------------------------------------------*/
void message(char *pString)
{
  window(1, 1, tinf.screenwidth, tinf.screenheight);
  gotoxy(2, tinf.screenheight);
  clreol();
  textcolor(RED);
  textbackground(LIGHTGRAY);
  cputs(pString);
  window(2, 3, tinf.screenwidth-1, tinf.screenheight-2);
}
```

```
/*-------------------------------------------------------*/
/*---                                                 ---*/
/*---   GetFilename()                                 ---*/
/*---                                                 ---*/
/*---   Nouab kasutajalt faili nime sisestamist alu-  ---*/
/*---   misele reale ja salvestab selle ettenäidatud  ---*/
/*---   puhvrisse.                                    ---*/
/*-------------------------------------------------------*/
int GetFilename(char *pString)
{
  char            buf[250];

  window(1, 1, tinf.screenwidth, tinf.screenheight);
  gotoxy(2, tinf.screenheight);
  clreol();
  textcolor(WHITE);
  textbackground(LIGHTGRAY);
  cputs("Failinimi: ");      /* noua uue nime sisestamist */
  buf[0] = 247;
  cgets(buf);
  if(buf[1] != 0) {         /* kui nime pikkus ei ole null siis */
    strncpy(pString, buf+2, buf[1]);  /* kopeeri ta puhvrisse */
    *(pString+buf[1]) = 0;
  }
  window(2, 3, tinf.screenwidth-1, tinf.screenheight-2);
  return buf[1];
}     /* GetFilename() */



/*-------------------------------------------------------*/
/*---                                                 ---*/
/*---   DoCommand()                                   ---*/
/*---                                                 ---*/
/*---   Täidab kasutaja poolt sisestatud käsu.        ---*/
/*-------------------------------------------------------*/
void DoCommand(int nCommand)
{
  char            szBuf[250];
  int       dif;

  switch(nCommand) {
    case ID_OPEN:       /* ava uus fail */
      szBuf[0] = 0;
      if(GetFilename(szBuf))
      OpenFile(szBuf);
      break;
    case ID_HOME:       /* nihuta kursor positsioooni (1, 1) */
      CurX = CurY = 1;
      nFirstLine = 0;
      RedrawText();
      PutCursor(CurX, CurY);
      break;
    case ID_END:        /* nihuta kursor faili loeppu */
      CurY = size / 16;
      CurX = size - CurY * 16;
      nFirstLine = max(0, CurY-10);
      clrscr();
      RedrawText();
      PutCursor(CurX, CurY);
      break;
    case ID_LEFT: /* nihuta kursor ühe positsiooni voerra vasakule */
      CurX = max(1, CurX - 1);
      PutCursor(CurX, CurY);
      break;
    case ID_RIGHT: /* nihuta kursor ühe positsiooni voerra paremale
*/
      CurX = min(16, CurX + 1);
      PutCursor(CurX, CurY);
```

```
      break;
    case ID_LINEDOWN:  /* nihuta kursor ühe positsiooni voerra alla
*/
      CurY = min(size / 16, CurY + 1);
      if((CurY - 1) > (nFirstLine + tinf.screenheight - 5)) {
      nFirstLine++;
      RedrawText();
      }
      PutCursor(CurX, CurY);
      break;
    case ID_LINEUP:   /* nihuta kursor ühe positsiooni voerra üles */
      CurY = max(1, CurY - 1);
      if(CurY - 1 < nFirstLine) {
      nFirstLine--;
      RedrawText();
      }
      PutCursor(CurX, CurY);
      break;
    case ID_PAGEUP:   /* nihuta kursor ühe lehekülje vorra üles */
      CurY = max(1, CurY - tinf.screenheight + 4);
      nFirstLine = max(1, nFirstLine - tinf.screenheight + 4);
      RedrawText();
      PutCursor(CurX, CurY);
      break;
    case ID_PAGEDOWN: /* nihuta kursor ühe lehekülje vorra alla */
      CurY = min(CurY + tinf.screenheight - 4, size / 16);
      nFirstLine = min(nFirstLine + tinf.screenheight - 4, size / 16
- 10);
      RedrawText();
      PutCursor(CurX, CurY);
      break;
    case ID_PRINT:     /* trüki fail */
      PrintFile();
      break;
    case ID_SAVE:     /* salvesta fail */
      WriteFile(szFileName);
      break;
    case ID_SAVEAS:   /* salvesta fail uue nime all */
      szBuf[0] = 0;
      if(GetFilename(szBuf)) {
      strcpy(szFileName, szBuf);
      WriteFile(szFileName);
      }
      DrawScreen();
      RedrawText();
      break;
    default:
      break;
  }
}     /* DoCommand() */


/*-------------------------------------------------------*/
/*---                                                ---*/
/*---   RedrawText()                                 ---*/
/*---                                                ---*/
/*---   Uuendab editoriakna sisu.                    ---*/
/*-------------------------------------------------------*/
void RedrawText( void )
{
  int x, i;
  char      buf[100], Tmp[10];

  if(pMem != NULL) {
    MouHide();
    textcolor(YELLOW);
    textbackground(BLUE);
                      /* igasse ritta mahub 16 elementi */
    for(x = 0; ((x + nFirstLine) * 16 + x < size) &&
            (x < tinf.screenheight - 4); x++) {
```

```
      sprintf(buf, "%06d | ", (x + nFirstLine) * 16); /* aadressid */
      for(i = 0; (i < 16) && ((x + nFirstLine) * 16 + i < size); i++)
{
      sprintf(Tmp, "%02X ", *(unsigned char*)(
                  pMem + (x + nFirstLine) * 16 + i));
      strcat(buf, Tmp);
      }
      strcat(buf, "| ");
      for(i = 0; (i < 16) && ((x + nFirstLine) * 16 + i < size); i++)
{
      if(*(pMem + (x + nFirstLine) * 16 + i) > 32) {
        sprintf(Tmp, "%c", *(pMem + (x + nFirstLine) * 16 + i));
        strcat(buf, Tmp);
      }
      else
        strcat(buf, "·");
      }
      gotoxy(1, x + 1);
      cputs(buf);
    }
    MouShow();
    PutCursor(CurX, CurY);
  }
}     /* RedrawText() */


/*-----------------------------------------------------*/
/*---                                               ---*/
/*---   OpenFile()                                  ---*/
/*---                                               ---*/
/*---   Avab määratud faili, loeb ta sisu mällu ja  ---*/
/*---   kutsub välja funktsiooni RedrawText(),      ---*/
/*---   uuendamaks editoriakna sisu.                ---*/
/*-----------------------------------------------------*/
void OpenFile(char *szName)
{
  FILE            *hFile;
  char            buf[50];
  int       i;

      /* ürita avada fail */
  if(NULL == (hFile = fopen(szName, "rb"))) {
    sprintf(buf, "Ei suuda avada faili %s !", szName);
    message(buf);
    return;
  }
      /* uuri, kui suur see fail on */
  fseek(hFile, 0L, SEEK_END);
  size = ftell(hFile);
      /* reserveeri vastav hulk mälu */
  if(pMem)
    free(pMem);
  if(NULL == (pMem = (char *)malloc(size))) {
    sprintf(buf, "Ei suuda reserveerida %d baiti mälu !", size);
    message(buf);
    fclose(hFile);
    return;
  }
      /* loe faili sisu puhvrisse */
  fseek(hFile, 0L, SEEK_SET);
      /* kontrolli, kas loeti sisse oige hulk baite */
  if(size != fread(pMem, 1, size, hFile)) {
    free(pMem);
    pMem = NULL;
    sprintf(buf, "Ei suutnud lugeda faili %s !", szName);
    message(buf);
    fclose(hFile);
  }
  sprintf(szFileName, szName);
  i = strlen(szFileName);
```

```
  window(1, 1, tinf.screenwidth, tinf.screenheight);
  gotoxy(tinf.screenwidth / 2 - i / 2, 2);
  cputs(szFileName);
  window(2, 3, tinf.screenwidth-1, tinf.screenheight-2);
  nFirstLine = 0;
  fDirty = 0;
  DrawScreen();
  RedrawText();
  fclose(hFile);
}     /* OpenFile() */



/*------------------------------------------------------------*/
/*---                                                      ---*/
/*---   PrintFile()                                        ---*/
/*---                                                      ---*/
/*---   Trükib faili standardsel trükkalil.                ---*/
/*------------------------------------------------------------*/
void PrintFile( void )
{
  int x, i;
  char      buf[100], Tmp[10];

  if(pMem != NULL) {
    fprintf(stdprn, "             %s\n\n", szFileName);
    /* igasse ritta mahub 16 elementi */
    for(x = nFirstLine; x < size; x++) {
      sprintf(buf, "%06d | ", x * 16);    /* aadressid */
      for(i = 0; i < 16; i++) {
      sprintf(Tmp, "%02X ", *(unsigned char*)(pMem + x * 16 + i));
      strcat(buf, Tmp);
      }
      strcat(buf, "| ");
      for(i = 0; i < 16; i++) {
      if(*(pMem + x * 16 + i) > 32) {
        sprintf(Tmp, "%c", *(pMem + x * 16 + i));
        strcat(buf, Tmp);
      }
      else
        strcat(buf, "·");
      }
      fprintf(stdprn, "%s \n", buf);
    }
  }
}     /* PrintFile */


/*------------------------------------------------------------*/
/*---                                                      ---*/
/*---   PutCursor()                                        ---*/
/*---                                                      ---*/
/*---   Nihutab textikursori soovitud positsiooni.         ---*/
/*------------------------------------------------------------*/
void PutCursor(int x, int y)
{
  gotoxy(10 + (x - 1) * 3 + 1, y - nFirstLine);
}     /* PutCursor */






/*------------------------------------------------------------*/
/*---                                                      ---*/
/*---   WriteFile()                                        ---*/
/*---                                                      ---*/
```

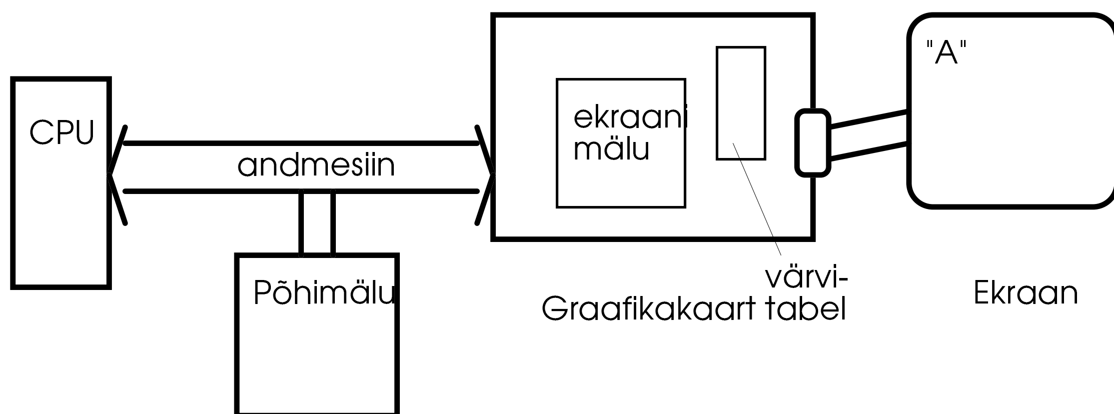
```
/*---   Salvestab hetkelise puhvri vastavasse faili.  ---*/
/*---   Vana fail kirjutatakse uuega üle.             ---*/
/*----------------------------------------------------------*/
int WriteFile(char *szName)
{
  FILE            *hFile;
  char            buf[50];
  int       i;
      /* ürita avada faili */
  if(NULL == (hFile = fopen(szName, "wb"))) {
    sprintf(buf, "Ei suuda avada faili %s !", szName);
    message(buf);
    return 0;
  }
  if(size != fwrite(pMem, 1, size, hFile)) {
    sprintf(buf, "Ei suutnud salvestada faili %s !", szName);
    message(buf);
    fclose(hFile);
    return 0;
  }
  fDirty = 0;
  message("fail salvestatud");
  fclose(hFile);
  PutCursor(CurX, CurY);
  return 1;
}     /* WriteFile() */

/*----------------------------------------------------------*/
/*---                                                 ---*/
/*---   PutChar()                                     ---*/
/*---                                                 ---*/
/*---   Sisestab sümboli ch hetkelisel positsioonil.  ---*/
/*----------------------------------------------------------*/
void PutChar(char ch)
{
  char            buf[5];
  *(pMem + (CurY - 1) * 16 + CurX - 1) = ch;
  sprintf(buf, "%02X ", *(unsigned char*)
      (pMem + (CurY - 1) * 16 + CurX - 1));
  gotoxy(10 + (CurX - 1) * 3, CurY - nFirstLine);
  textcolor(YELLOW);
  textbackground(BLUE);
  cputs(buf);
  gotoxy(59 + CurX, CurY - nFirstLine);
  putch(ch);
  if(CurX < 16) {
    CurX++;
  }
  else {
    CurX = 1;
    CurY = min(CurY + 1, size / 16);
  }
  PutCursor(CurX, CurY);
  fDirty = 1;
}     /* PutChar() */
```

Graafilised funktsioonid

Selles peatükis käsitleme Borland C/C++ graafilisi funktsioone. Need funktsioonid ei ole standardiseeritud. Seega saata neid programme kasutada vaid Borland C translaatoriga. Eelmistes peatükkides vaadeldud funktsioonid väljastasid samuti andmeid ekraanile, kuid sel korral oli ekraan jagatud 25 reast ja 80 tulbast (vastavalt tekstirezhiimile) koosnevaks väljaks. Programm sai küll määrata, millised sümbolid selle välja erinevatesse lahtritesse sisestatakse, kuid

mitte rohkem. Graafilises rezhiimis saate aga ekraanil kontrollida iga üksikut punkti (pikselit). Muidugi on selline väljastamine natuke aeglasem, kuna nüüd tuleb toimetada graafikakaardi mällu iga pikseli värvuse andmed. Tekstirezhiimi puhul oli vaja anda üle ainult ühe tähe kood ja atribuut. Kasutades neid andmeid määras graafikakaart ise vastavate pikselite värvused. Graafilises rezhiimis on selle asemel võimalik täpsemalt määrata väljastatavate andmete kuju. Ka graafilises rezhiimis saab väljastada andmeid tekstikujul. Isegi samade tuntud funktsioonide (*printf()*, *puts()*, *putch()*) kasutamine on võimalik. Sel juhul kasutatakse väljastamisel üht lihtsat šrifti. Graafilises rezhiimis on aga võimalik määrata ka mingit muud šrifti, tõmmata teksti ümber jooni, ringe ja muid graafilisi kujundeid ning kasutada suletud graafiliste kujundite puhul täitmiseks kindlaid mustreid ja värvusi.
Graafikakaart omab eraldi mälu, milles säilitatakse ekraani iga pikseli andmed. Enamus graafikamonitore on suutelised esitama suurel hulgal erinevaid värve, kuid kuna iga värvitooni salvestamiseks peab kasutama mingit arvu ja arvude suurus on piiratud, siis saab ekraanil korraga esitada vaid teatud kindlat hulka värve. Värvi määramiseks jagatakse värvitoon kolme ossa: punase- rohelise- ja kollase põhivärvi intensiivsus. Neid kolme värvi sobivalt segades saab määrata mistahes muu värvi. Iga värvitooni intensiivsus on aga mingi arv nullist kuni 256-ni. Vähendamaks ekraanimälus salvestatavate andmete hulka salvestatakse vajalikud värvitoonid nn. värvitabelis.



Joonis 10: Andmetekulg protsessorist ekraanile

Värvitabelis luuakse iga vajaliku värvi kohta eraldi sissekanne, mis sisaldab tema kolme põhivärvuse intensiivsuseid. Tavaliselt sisaldab see tabel 256 sissekannet, millest igaüks koosneb kolmest 6-bitisest osast. Määramaks ühe pikseli värvust ekraanil salvestatakse ekraanimälu vastavasse kohta värvitabeli vastava sissekande järjekorranumber. Seega on vaja ühe pikseli 256 võimaliku värvuse eristamiseks salvestada mälus vaid 1 bait.
Graafikarezhiimid erinevad tunduvalt võimalike värvide hulga ja lahutusvõime poolest. Mida suurem lahutusvõime, seda vähem värve on korraga võimalik ekraanile esitada ja vastupidi. Borland C/C++ kasutab graafika väljastamiseks ekraanile kataloogis \BORANDC\BGI asuvaid ohjurprogramme. Igale graafi-

karezhiimide grupile vastab oma ohjurprogramm. Kõik need ohjurprogrammid sisaldavad neidsamu funktsioone, kuid erinevate lahutusvõimete ja värvide hulga jaoks. Borland C/C++ 2.0 tunneb järgmisi graafikarezhiime:

Ohjurprogramm	Graafikarezhiim	Lahutusvõime	Värvide hulk
CGA	CGAC0	320 x 200	4
	CGAC1	320 x 200	4
	CGAC2	320 x 200	4
	CGAC3	320 x 200	4
	CGAHI	640 x 200	2
MCGA	MCGAC0	320 x 200	4
	MCGAC1	320 x 200	4
	MCGAC2	320 x 200	4
	MCGAC3	320 x 200	4
	MCGAMED	640 x 200	2
	MCGAHI	640 x 480	2
EGA	EGALO	640 x 200	16
	EGAHI	640 x 350	16
EGA64	EGA64LO	640 x 200	16
	EGA64HI	640 x 350	4
EGAMONO	EGAMONOHI	640 x 350	2
HERC	HERCMONOHI	720 x 348	2
ATT400	ATT400C0	320 x 200	4
	ATT400C1	320 x 200	4
	ATT400C2	320 x 200	4
	ATT400C3	320 x 200	4
	ATT400MED	640 x 200	2
	ATT400HI	640 x 400	2
VGA	VGALO	640 x 200	16
	VGAMED	640 x 350	16
	VGAHI	640 x 480	16
PC3270	PC3270HI	720 x 350	2
IBM8515	IBM8515LO	640 x 480	256
	IBM8515HI	1024 x 768	256

Tabel 17: Graafikarezhiimid, lahutusvõimed ja värvid

Esimeses tulbas näete ohjurprogrammide nimesid. Graafikafunktsioone kasutav programm peab enne graafikafunktsioonide kasutamist installeerima vastava ohjurprogrammi funktsiooni *initgraph()* abil.

```
initgraph(int far *graphdriver, int far *graphmode, char
                                         far *pathtodriver);
```

Funktsioon *initgraph()* laadib soovitud ohjurprogrammi mällu ja lülitab ekraani määratud graafilisse rezhiimi. Funktsiooni esimene parameeter osutab täisarvulisele muutujale, mis sisaldab üht tabeli 16 esimeses tulbas toodud sümboolset konstanti. Selle parameetriga määratakse soovitud ohjurprogramm. Funktsiooni teine parameeter näitab samuti ühele täisarvule, mille sisu määrab soovitud graafikarezhiimi. Tabeli 16 teises tulbas on loetletud, millised rezhiimid on iga ohjurprogrammi jaoks lubatud. Toodud nimetused on samuti sümboolsed konstandid ja neid võib otse nii ka programmis kasutada. Kui funktsiooni esimene parameeter aga osutab nullile (sümboolne konstant DETECT), siis üritab programm ise leida teie ekraani jaoks sobiva ohjurprogrammi. Sel juhul ei osutata tähelepanu funktsiooni teie poolt seatud parameetri väärtusele, vaid lülitatakse ekraan suurima võimaliku lahutusvõimega rezhiimi. Nüüd salvestab funktsioon esimese kahe parameetri poolt osutatud muutujatesse tegelikult kasutatud ohjurprogrammile ja graafikarezhiimile vastavad konstandid. Kui te kasutasite parameetrit DETECT, siis võite muutujate uute väärtuste järgi otsustada, kui palju värve te nüüd kasutada saate ja kui suur on lahutusvõime. Funktsiooni viimane parameeter viitab sümbolite jadale (stringile), mis määrab kataloogi, kus asuvad ohjurprogrammid. Kui see parameeter on NULL, siis oletatakse, et ohjurprogrammid asuvad hetkelises kataloogis. Te võite ka kasutada abiprogrammi BGIOBJ.EXE, muutmaks ohjurprogrammid objektfailideks (*.OBJ) ja sidumaks nad seejärel oma programmiga. Unustada ei tohi, et kataloogi nimedes kasutatav sümbol "\" omab C - keeles erilist tähendust ja seega tuleb näiteks kataloogi C:\BORLANDC\BGI määramiseks sisestada sümbolite jada "C:\\BORLANDC\\BGI". Seda kataloogi kasutatakse ka graafiliste šriftide *.CHR otsimiseks.
Funktsiooniga *installuserdriver() on* ka võimalik installeerida kasutaja poolt programmeeritud ohjurprogramm ja seejärel kasutada ekraani lülitamiseks graafilisse rezhiimi funktsiooni *initgraph()*, kuid seda siin ei käsitleta, kuna selle ohjurprogrammi loomine nõuaks teilt põhjalikke teadmisi assembleris.
Peale graafiliste funktsioonida kasutamist tuleks tekstirezhiim taastada funktsiooniga *closegraph()*.

```
void far closegraph( void );
```

Graafiliste funktsioonide prototüübid asuvad päisefailis GRAPHICS.H ja definitsioonid teegis GRAPHICS.LIB. Graafiliste funktsioonide kasutamiseks tuleb see teek programmiga siduda.
Päisefail GRAPHICS.H defineerib ka muutuja *grapherror*, mis on täisarv ja sisaldab veakoodi graafiliste funktsioonide töös tekkinud vea jaoks. Funktsioon *initgraph()* omistab sellele muutujale väärtuse 0. Hiljemkasutatud funktsioonid võivad tema väärtust muuta.
Ohjurprogrammid jaotatakse üldiselt kahte gruppi CGA ja EGA/VGA. Kõik esimese grupi ohjurprogrammid asuvad failis CGA.BGI. Nad omavad väikese lahutusvõimega rezhiime (320 x 200), kus kasutatakse 4 värvi sisaldavat värvitabelit. Kasutamiseks valitud graafikarezhiim CGAC0, CGAC1 jne määrab, millist värvitabelit on vaja. Need värvitabelid sisaldavad tavaliselt kolme värvi

ja neljanda (tagaplaanivärvi) saate ise määrata. Selleks kasutage funktsiooni *setbkcolor()*.

```
void far setbkcolor(int nColor);
```

Suurema lahutusvõimega rezhiimis saate kasutada vaid kaht värvi.
Ohjurprogrammi EGA puhul saate kasutada 16 värvist koosnevat värvitabelit. Neid kuutteist värvi tohite aga kõiki valida 64 võimaliku hulgast. VGA ohjurprogramm suudab samuti esitada ekraanile korraga 16 värvi, mida saab valida 262.144 võimalikust värvist. Vastavalt graafikakaardile ja monitorile võib olla rohkemgi erineva hulga värvide ja lahutusvõimetega graafika-rezhiime. Borland C/C++ sisaldab aga ohjurprogramme vaid piiratud hulga jaoks.
Värvitabeli värvuse muutmise jaoks kasutage funktsiooni *setpalette()*.

```
void setpalette(int nPaletteIndex, int nNewColor);
```

See funktsioon omistab värvitabelisse sissekande järjekorranumbriga *nPaletteIndex* parameetriga *nNewColor*. Värvitabeli esimese sissekande järje-korranumber on 0. Värvis nNewColor on mingi väärtus vastava graafikakaardi jaoks lubatud piirides. Näiteks EGA puhul peab see väärtus asuma vahemikus 0 - 63. Nende värvide jaoks on olemas ka sümboolsed konstandid nagu näiteks EGA_BLACK või CGA_GREEN, mis leiate translaatori käsiraamatust. Suurima võimaliku väärtuse parameetri *nNewColor* jaoks leiate funktsiooniga *getmaxcolor()*.

```
int far getmaxcolor( void );
```

Muutmaks värvitabeli paljusid sissekandeid, kasutage funktsiooni *setall-palette()*, mis asendab korraga kogu värvitabeli sissekanded.

```
void far setallpalette(struct palettetype far *newPalette);
```

Funktsiooni *setallpalette()* ainus parameeter on viit andmestruktuurile tüübist *palettetype*, mis sisaldab uusi värvitabeli sisssekandeid.

```
struct palettetype {
  unsigned char   size;
  signed char     colors[MAXCOLORS+1];
};
```

Selle struktuuri väli *size* määrab värvitabeli sissekannete hulga. Massiiv *colors* aga sisaldab igale sissekandele vastavat numbrit. Kui te omistate massiivi *colors* mingile elemendile väärtuse -1 ja kasutate seejärel funktsiooni *setallpalette()*, siis nimetatud värvitabeli sissekande väärtust ei muudeta. Te võite aga ka kasutada funktsiooni *getpalette(),* täitmaks seda andmestruktuuri hetkeliste värvitabeli väärtustega, muuta siis vajalikud elemendid ja kasutada uuesti funktsiooni *setallpalette()*.

```
void far getpalette(struct palettetype far *oldPalette);
```

Funktsioon *initgraph()* loob samuti ühe andmestruktuuri tüübist *palettestruct* ja täidab selle vastava graafikarezhiimi jaoks sobivate väärtustega. Selle värvitabeli andmete lugemiseks defineerige sobiv viit ja kasutage värvitabeli aadressi hankimiseks funktsiooni *getdefaultpalette()*.

```
struct far palettetype * getdefaultpalette( void );
```

Olles määranud graafilise rezhiimi ja värvitabeli, saab asuda ekraanile graafilisi kujundeid joonistama. Seejuures eristatakse joon- ja pindelementide vahel.

Joonelemendid, pikselid ja rasterpildid

Joonelemendid on jooned, ringid, ellipsid, ristkülikud jne. Kui määrata iga kujundi jaoks kõikmõeldavad graafilised parameetrid, siis omaksid need funkt-sioonid liiga palju parameetreid ja ei oleks seega ülevaatlikud. Seepärast salves-tatakse osa parameetreid, nagu näiteks ees- ja tagaplaani värv, ohjurprogrammi muutujates ja neid väärtusi kasutatakse siis iga järgneva väljastusoperatsiooni puhul. Kui te näiteks soovite joonistada punase ringjoone, siis tuleb kõigepealt määrata funktsiooniga *setcolor()* esiplaanivärviks punane (kui ta seda juba ei ole) ja joonistada seejärel ringjoon.

```
void far setcolor(int nNewColor);
```

Funktsiooni *setcolor()* parameeter *nNewColor* peab olema mingi hetkelise värvitabeli sissekande järjekorranumber. Kui hetkeline eesplaanivärvus juba oli punane, siis ei tee see funktsioon kah midagi paha. Kui te aga soovite kindel olla, siis kasutage funktsiooni *getcolor()*.

```
int far getcolor( void );
```

Üldiselt tuleb öelda, et eesplaanivärvuse uurimine võtab täpselt sama palju aega, kui tema määramine, nii et kui te just programmi kulu järgi ei oska öelda, milline on hetkeline eesplaanivärvus, siis parem muutke kohe ja ärge hakake uurima, milline värv oli.
Peale eesplaanivärvuse mõjutab joonelementide välimust veel joone tüüp. Selle atribuudi määramiseks kasutage funktsiooni *setlinestyle()*.

```
void far setlinestyle(int linestyle, unsigned pattern, int
thickness);
```

Funktioon *setlinestyle()* määrab korraga kolm atribuuti. Parameeter *linestyle* määrab joone tüübi ja võib omada järgmisi väärtusi:
- SOLID_LINE (0) - pidev joon
- DOTTED_LINE (1) - punktiirjoon
- CENTER_LINE (2) -
- DASHED_LINE (3) - kriipsjoon
- USERBIT_LINE (4) - kasutaja poolt defineeritud joonetüüp

Kui parameetri *linestyle* väärtus on USERBIT_LINE, siis määrab parameeter *pattern* joone tegeliku stiili. See parameeter on 16-bitine positiivne täisarv, mis iseloomustab mustrit. Iga 1-ga võrduv bitt selles arvus määrab, et vastav piksel ekraanil joonistatakse eesplaani värviga ning ülejäänud tagaplaani värviga. Näiteks kui parameetri *pattern* väärtus on 1 (mis kahendkoodis oleks 0000000000000001), siis ilmub ekraanile iga 16 pikseli järel üks punkt. Väärtuse 65.535 (0xFFFF) puhul aga ilmub ekraanile pidev joon.

Parameeter *thickness* määrab joone laiuse pikselites. Esialgu on lubatud vaid väärtused:

- NORM_WIDTH (1) - Ühe pikseli jämedune joon
- THICK_WIDTH (3) - Kolme pikseli jämedune joon

Funktsioon *getlinestettings()* hangib kõik nimetatud andmed ja salvestab nad andmestruktuuri *linesettingstype*.

```
void far getlinesettings( struct far linesettingstype *settings);

struct linesettingstype {
  int linestyle;
  unsigned upattern;
  int thickness;
};
```

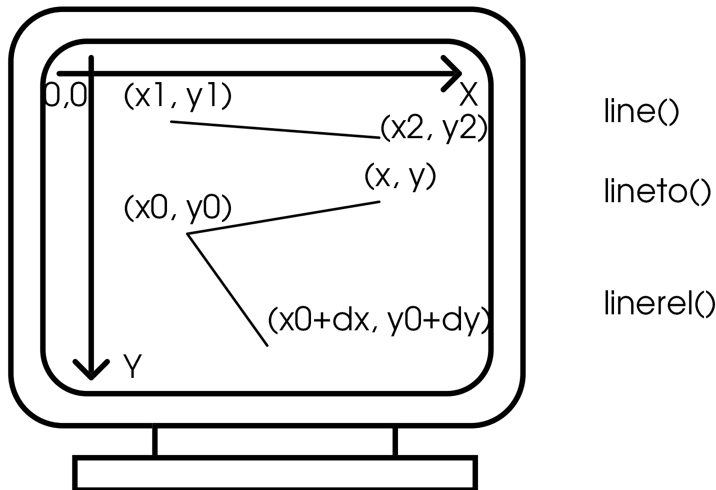
Ühe sirgjoone joonistamiseks ekraanile võib kasutada funktsioone *line()* ja *lineto()* ja *linerel()*.

```
void far line(int x1, int y1, int x2, int y2);
void far lineto(int x, int y);
void far linerel(int dx, int dy);
```

Funktsioon *line()* joonistab sirgjoone punktist (x1, y1) punkti (x2, y2), kasutades hetkelist eesplaani värvust ja joonetüüpi. Algselt on koordinaatidesüsteemi alguspunkt ekraani ülemises vaskus nurgas ja erinevalt tekstirezhiimist on esimese punkti koordinaadid (0, 0), mitte (1, 1). Seejuures on X- telg suunatud vasakult paremale ja Y - telg ülevalt alla. Koordinaatidesüsteemi ühikud vastavad pikselitele, kuid nii telgede asetust, suunda, kui ka koordinaatidesüsteem ühikute pikkusi saab muuta. Sellest täpsemalt järgnevates peatükkides. Suurima X ja Y - väärtuse saab teada funktsioonidega *getmaxx()* ja *getmaxy()*.

```
int far getmaxx( void );
int far getmaxy( void );
```

Joonisel 11 näete funktsioonide *line()*, *lineto()* ja *linrel()* mõju.



Joonis 11: Funktsioonid line(), lineto() je linerel()

Funktsioonid *lineto()* ja *linerel()* kasutavad nn. hetkelist väljastamise punkti, mis joonisel 11 on märgitud koordinaatidega (x0, y0). Algselt asub see punkt koordinaatide alguspunktis, kuid suur osa funktsioone, mis midagi ekraanile väljastavad, nihutavad selle punkti oma väljastatud andmete lõppu, et järgmised funktsioonid äsjaväljastatud andmeid üle ei kirjutaks. Selle punkti koordinaate saab uurida funktsioonidega *getx()* ja *gety()*, ning määrata funktsiooniga *moveto()*, *moverel()*.

```
int far getx( void );
int far gety( void );
void far moveto(int x, int y);
void far moverel(int dx, int dy);
```

Funktsioon *moveto()* nihutab hetkelise väljastamise punkti lihtsalt punkti (x, y), funktsioon *moverel()* seevastu liidab hetkelise punkti koordinaatidele väärtused *dx* ja *dy*, ning arvutab sel kombel uue hetkelise punkti.
Funktsioon *line()* ei mõjuta mingil määral hetkelist punkti ega kasuta ka tema andmeid. Funktsioon *lineto()* seevastu tõmbab joone hetkelisest punktist punkti (x, y) ja funktsioon *linerel()* joonistab sirge hetkelisest punktist punkti, mille koordinaadid on hetkelise punkti omadest (dx, dy) võrra suuremad. Muidugi tohivad need parameetrid (dx ja dy) olla ka negatiivsed.
Üheainsa pikseli värvuse määramiseks kasutage funktsiooni *putpixel()* ja tema värvuse uurimiseks funktsiooni *getpixel()*.

```
void far putpixel(int x, int y, int nColor);
unsigned far getpixel(int x, int y);
```

Ka graafilises rezhiimis on võimalik salvestada terve ekraanipiirkonna sisu eraldi mälublokki ja hiljem see uuesti ekraanile väljastada. Selleks kasutatakse funktsioone *getimage()* ja *putimage()*.

```
void far getimage(int x1, int y1, int x2, int y2, void far *bitmap);
void far putimage(int x1, int y1, void far *bitmap, int options);
```

Funktsioon *getimage()* koperib ekraanilt ristkülikukujulise piirkonna (x1, y1) - (x2, y2) parameetri *bitmap* poolt viidatud puhvrisse. Seejuures kasutatakse selle puhvri esimest kahte 16-bitist arvu salvestatud rasterpildi laiuse ja kõrguse salvestamiseks. Puhver peab olema küllaldaselt suur kogu informatsiooni salvestamiseks ehk: (x2 - x1) * (y2 - y1) * <bitti pikseli kohta> / 8. Selle suuruse arvutamiseks on mugavam kasutada funktsiooni *imagesize()*.

```
unsigned far imagesize(int x1, int y1, int x2, int y2);
```

Funktsioon *imagesize()* arvutab määratud suurusega rasterpildi suuruse baitides, võttes arvesse ka hetkelist graafikarezhiimi. Kui soovitud rasterpilt osutub liiga suureks (s.o. suurem kui 64 KB), siis loovutab funktsioon *imagesize()* väärtuse -1.
Funktsioon *putimage()* väljastab salvestatud rasterpildi uuesti ekraanile, alustades punktist (x, y) ja kasutades seejuures parameetriga *options* määratud valikuid. Nimetatud parameeter määrab, kuidas rasterpildi pikselite värvused seotakse ekraani pikselite värvustega ja võib omada järgmises tabelis näidatud väärtusi.

Sümboolne konstant	Tähendus
COPY_PUT	Kopeerib rasterpildi ekraanile, kirjutades selle piirkonna üle uute väärtustega.
XOR_PUT	Ühendab rasterpildi pikselite värvused ekraani antud piirkonna pikselite värvusega loogilise XOR funktsiooni abil. Selle valiku abil saab samasse kohta eelnevalt väljastatud rasterpilti uuesti ekraanilt eemaldada.
OR_PUT	Ühendab rasterpildi pikselite värvused ekraani antud piirkonna pikselite värvusega loogilise OR funktsiooni abil.
AND_PUT	Ühendab rasterpildi pikselite värvused ekraani antud piirkonna pikselite värvusega loogilise AND funktsiooni abil.
NOT_PUT	Kannab ekraanile inverteeritud rasterpildi.

Tabel 18: Funktiooni putimage() parameetri options väärtused

Mustvalge rasterpildi üksikud bitid määravad, kas antud pikseli värvus on valge (1) või must (0). Nimetatud valikud mõjutavad seda, mil kombel rasterpildi ja ekraani pikselite väärtused omavahel kombineeritakse. Kui te kasutate näiteks valikut AND_PUT, siis kombineeritakse rasterpildi ja ekraani vastavad pikselid loogilise AND funktsiooni abil. Loogilise AND funktsiooni väärtus on teatavasti vaid siis 1, kui mõlemad parameetrid on ühesugused. Seega "soosib" see valik musti pikseleid, valged pikselid esitatakse ekraanile aga vaid siis valgetena, kui nad satuvad juba valgele aluspõhjale. Loogiline OR funktsioon on selle täielik vastand. Sel juhul esitatakse valged pikselid ekraanile alati

valgetena, mustad aga ainult siis, kui nad satuvad juba mustale aluspõhjale. Loogiline XOR funktsioon on vaid siis 1, kui mõlemad parameetrid omavad üksteisest erinevaid väärtusi. Seega kui sama rasterpilt kantakse valiku XOR_PUT abil ekraanile, siis satuvad samad pikselite väärtused uuesti samale kohale ja XOR funktsioon kustutab nad. Värvilise rasterpildi puhul kombineeritakse iga värvitasandi pikselite väärtused valitud loogilise funktsiooni abil ekraani vastava värvitasandi pikselite väärtustega. Värvitasanditeks nimetatakse ekraani iga pikseli salvestamiseks kasutatud bittide väärtusi. Näiteks kui ekraan on graafilises rezhiimis, mis suudab esitada korraga 16 erinevat värvi, siis kasutatakse iga pikseli jaoks 4 bitti, mis määra-vad põhivärvuste - punane, roheline, sinine - intensiivsuse väärtused. Kui iga biti väärtus on 1, siis on punase, sinise ja rohelise segu valge. Kuna iga pikseli jaoks kasutatakse nimetatud 4 bitti, siis saab ekraani jaotada mõtteliselt neljaks eri tasandiks, mis määravad vastavalt punase- rohelise- ja sinise osapildi. Neid tasandeid nimetataksegi värvitasanditeks.
Funktioone *getimage()* ja *putimage()* kasutatakse sageli animatsiooni programmeerimiseks. Rasterpildi kopeerimine ekraanimällu on tunduvalt kii-rem, kui selle pildi arvutamine ja vastavate graafiliste kujundite joonistamine.
Ristküliku joonistamiseks kasutage funktsiooni *rectangle()*.

```
void far rectangle(int x1, int y1, int x2, int y2);
```

Nimetatud funktsioon joonistab ekraanile ristküliku punktist (x1, y1) - ülemine vasak punkt punkti (x2, y2) - alumine parempoolne punkt. Ristküliku joonistamiseks kasutatakse hetkelist eesplaani värvust, joonetüüpi ja -laiust. Ristküliku sisu ja hetkelist väljastamise punkti ei muudeta.
Suvalise hulknurga joonistamiseks võib kasutada funktsiooni *drawpoly()*.

```
void far drawpoly(int numPoints, int far *points);
```

Funktsioon *drawpoly()* joonistab hulknurga, ühendades parameetri *points* poolt osutatud punktide massiivi iga punkti järgmise punktiga. Parameeter *numPoints* määrab massiivi *points* sissekannete arvu. Massiiv *points* sisaldab *numPoints* * 2 täisarvu. Iga arvude paar määrab ühe punkti ekraanil (x, y). Suletud N - tipuga hulknurga joonistamiseks määrake N + 1 punkti, millest viimane vastab esimesele.
Ringjoone joonistamiseks kasutage funktsiooni *circle()*.

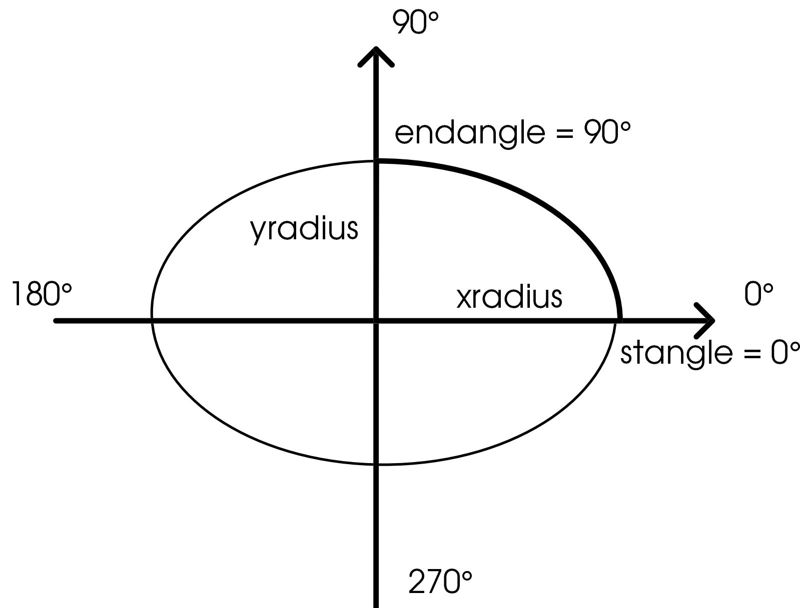
```
void far circle(int x, int y, int radius);
```

Funktsioon *circle()* joonistab ekraanile ringjoone keskpunktiga (x, y) ja raadiusega *radius*. Ellipsi joonistamiseks kasutage funktsiooni *ellipse()*.

```
void far ellipse(int x, int y, int stangle, int endangle, int
xradius, int yradius);
```

Funktsioon *ellipse()* joonistab ekraanile elliptilise kaare, millele vastava ellipsi keskpunkt asub punktis (x,y) ja mille pooltelјed on *xradius* ja *yradius*. Kaare

joonistamisega alustatakse nurgast *stangle* ja lõpetatakse nurgas *endangle*. Nurki mõõdetakse kraadides ja vastupäeva. Seda näete ka joonisel 12.



Joonis 12: Nurgad ja ellipsid

Samasuguse kaare, kuid ringikujulise, joonistab funktsioon *arc()*.

```
void far arc(int x, int y, int stangle, int endangle, int radius);
```

Kui alustada nurgast 0 ja lõpetada nurgaga 360, saab selle funktsiooniga joonistada ka suletud ringjoone.
Ka joonelementide puhul saab määrata, mil moel nad kombineeritakse ekraanil juba asuvate pikselitega. Selleks kasutage funktsiooni *setwritemode()*.

```
void far setwritemode(int mode);
```

Funktsioon *setwritemode()* parameeter *mode* võib omada väärtusi COPY_PUT ja XOR_PUT, mis omavad eespoolmainitud tähendusi.

Pindelemendid

Pindelemendid joonistavad ekraanile eesplaani värviga mingi suletud graafilise kujundi ning täidavad tema sisemuse tagaplaani värviga. Täitmiseks võib kasutada ka mitmesuguseid mustreid.
Funktsioon *setfillstyle()* määrab pindelementide täitmiseks kasutatava värvi ja mustri.

```
void far setfillstyle(int pattern, int color);
```

Graafilise rezhiimi puhul määrab ohjurprogramm tavaliselt täitmisel kasutatavaks värviks valge värvuse ja mustriks ühtlase täite. Hetkeliste täitmise väärtuste teadasaamiseks kasutage funktsiooni *getfillsettings()*.

```
void far getfillsettings( struct fillsettingstype far *fillinfo);
```

Funktsioon *getfillsettings()* täidab parameetri *fillinfo* poolt osutatud andmestruktuuri *fillsettingstype* vastavate andmetega.

```
struct fillsettingstype {
  int pattern;
  int color;
};
```

Parameeter *pattern* võib omada järgmisi väärtuseid:

Konstandi nimi	Tähendus
EMPTY_FILL	Täidab tagaplaani värviga
SOLID_FILL	Täidab ühtlaselt määratud värviga
LINE_FILL	-----
LTSLASH_FILL	//////
SLASH_FILL	\\\\\\, Jämedad jooned
BKSLASH_FILL	//////, Jämedad jooned
LTBKSLASH_FILL	\\\\\\
HATCH_FILL	xxxxx
XHATCH_FILL	xxxxx, Jämedad jooned
INTERLEAVE_FILL	Joonistab vaid iga teise horisontaalse joone kujundi sees tagaplaani värviga
WIDE_DOT_FILL	Hajutatud punktid, mis ei asu tihedalt ükstese kõrval
CLOSE_DOT_FILL	Lähestikku asuvad hajutatud punktid
USER_FILL	Kasutaja (programmeerija) poolt defineeritud täitemuster

Tabel 19: Täitemustritele vastavad sümboolsed konstandid

Nagu näete, sisaldab Borland C/C++ graafikateek juba hulgaliselt kasulikke täitemustreid. Kui neist ükski ei peaks sobima, siis võib ka defineerida oma isikliku mustri. Selleks ärge siiski kasutage funktsiooni *setfillstyle()* parameetriga USER_FILL, sest te peate kuidagi ka oma mustri defineerima. Kasutage oma mustri määramiseks funktsiooni *setfillpattern()* ja juba määratud mustri uurimiseks funktsiooni *getfillpattern()*.

```
void far setfillpattern(char far *pattern, int color);
void far getfillpattern(char far *pattern);
```

Funktsioon setfillpattern() määrab kasutaja poolt defineeritud täitemustri. Tema parameeter *pattern* osutab 8-baidisele massiivile, millest iga üksik bait määrab 8 x 8-pikselise mustrielemendi ühe horisontaalse rea. Kui näiteks

esimese baidi kolmas bitt omab väärtust 1 ja ülejäänud on nullid, siis täidetakse iga mustrielemendi esimese rea tagant viimane bitt parameetri *color* poolt määratud värviga ja ülejäänud selle rea pikselid täidetakse tagaplaani värviga.

*pattern
bait 0. = 0x30
bait 1. = 0x48
bait 2. = 0x84
bait 3. = 0x80
bait 4. = 0x80
bait 5. = 0x84
bait 6. = 0x48
bait 7. = 0x30

Joonis 13: Mustrielemendi ehitus

Joonisel 13 näete mustrielemendi ehitust. Siin on loodud tähte - "C" kujutav mustrielement. Sellise mustrielemendi määramiseks defineerige vastav massiiv ja omistage tema elementidele siintoodud väärtused, näiteks:

```
...
char  pattern[8] = { 0x30, 0x48, 0x84, 0x80, 0x80, 0x84, 0x48, 0x30
};
...
setfillpattern((char far *)pattern, getmaxcolor());
...
```

Mingi kujundi täitmisel reastatakse sellised mustrielemendid kujundi ülemisest vasakust nurgast alustades üksteise järel nii kaua, kuni kogu kujundi sisu on täidetud. Sellised mustrielemendid on nagu kahhelkivid, millega mingit pinda kaetakse.
Täidetud hulknurga joonistamiseks kasutage funktsiooni *fillpoly()*.

```
void far fillpoly(int numPoints, int far *points);
```

Selle funktsiooni parameeter *numPoints* määrab parameetri *points* poolt osutatud massiivi sissekannete arvu. Parameeter *points* osutab aga täisarvu paaride massiivile, millest igaüks määrab ühe hulknurga tipu koordinaadid. Hulknurga sisu täidetakse hetkelise täitemustri ja värviga. Kui massiivi *points* viimane koordinaatide paar (hulknurga tipp) ei lange kokku esimese paariga ja toodud koordinaadid määravad seega hoopis avatud kujundi, siis ühendab funktsioon *fillpoly()* automaatselt kujundi viimase tipu esimesega ja loob seega suletud kujundi. Funktsioon *drawpoly()* aga seda ei tee.
Täidetud ellipsi (ja ringjoone) joonistamiseks kasutage funktsiooni *fillellipse*().

```
void far fillellipse(int x, int y, int xradius, int yradius);
```

Funktsioon *fillellipse()* joonistab ellipsi keskunktiga (x, y) ja pooltelgedega *xradius* ning *yradius*. Ellipsi piirjoon joonistatakse hetkelise joone värvuse ja tüübiga, ning ellipsi sisu täidetakse hetkelise täitemustri ja -värviga.

Borland C/C++ graafiline teek sisaldab ka palju funktsioone graafikute ja diagrammide jaoks. Tulpdiagrammi loomiseks saab kasutada funktsioone *bar()* ja *bar3d()*.

```
void far bar(int x1, int y1, int x2, int y2);
void far bar3d(int x1, int y1, int x2, int y2, int depth, int
topflag);
```

Funktsioon *bar()* joonistab ekraanile täidetud ristküliku ülemise vasaku nurgaga punktis (x1, y1) ja alumise parema nurgaga punktis (x2, y2). Ristküliku sisu täidetakse hetkelise täitemustri ja -värviga, kuid ristküliku piirjoont ei joonistata. Piirjoone joonistamiseks kasutage funktsiooni *bar3d()* sügavusega (*parameeter depth*) 0.
Funktsioon *bar3d()* joonistab kolmedimensionaalse tulba sügavusega *depth*. Kui parameeter *topflag* on nullist erinev, siis on joonistatav risttahukas asetatud nii, et tema ülemine tahk on nähtav, nullise *topflag* puhul mitte. Viimasel juhul saab paremini esitada mitut tulpa üksteise otsas, mis on kasulik näiteks mitme erineva andmetejada võrdlemiseks.
Sektordiagrammi loomiseks kasutage funktsioone *pieslice()* ja *sector()*.

```
void far pieslice(int x, int y, int stangle, int endangle, int
radius);
void far sector(int x, int y, int stangle, int endangle, int xradius,
int yradius);
```

Funktsioon *pieslice()* joonistab ringi keskpunktiga (x, y) ja raadiusega *radius* sektori alustades nurgast *stangle* ja lõpetades nurga *endangle* puhul. Sektor täidetakse hetkelise täitemustri ja -värvusega.
Funktsioon *sector()* joonistab samalaadse ellipsi sektori. Vastava ellipsi keskpunkt on (x, y) ja poolteljed *xradius* ning *yradius*.
Te võite täita ka suvalise suletud kujundi, kasutades funktsiooni *floodfill()*.

```
void far floodfill(int x, int y, int border);
```

Funktsioon *floodfill()* täidab kujundi hetkelise täitevärvi ja -mustriga, alustades punktist (x, y). See punkt peab asuma suletud kujundi sees ja kogu kujund peab olema piiratud parameetriga *border* määratud värvi piiriga. Näiteks:

```
...
line(100, 100, 300, 100); /*joonistame hetkelise esiplaani*/

line(300, 100, 200, 300); /*värviga lihtsa kolmnurga*/
line(200, 300, 100, 100);
setfillstyle(LTSLACH_FILL, getmaxcolor());
           /* täidame kolmnurga soovitud värvi ja mustriga */
floodfill(200, 200, getcolor());
...
```

Graafiline tekst ja aknad

Graafilises rezhiimis saab küll kasutada teksti väljastamiseks ka lihtsaid funktsioone, kuid tavaliselt kasutatakse selleks siiski funktsioone *outtext()* ja *outtextxy()*.

```
void far outtext(char far *pString);
void far outtextxy(int x, int y, char far *pString);
```

Funktsioon *outtext()* väljastab talle üleantud sümbolitejada ekraanile, alustades hetkelisest väljastamise positsioonist ja kasutades hetkelist šrifti. Funktsioon *outtextxy()* seevastu alustab väljastamisega punktist (x,y). Funktsioon *outtext()* nihutab hetkelist väljastamise positsiooni väljastatud teksti lõppu, kui tekst väljastati normaalse suuna (HORIZ_DIR) ja harilikku joondamisega (LEFT_TEXT). Üle ekraani ulatuv tekstiosa ei esitata ekraanile. Väljastatava teksti suurus võib erineda vastavalt valitud šriftile ja suurendusele. Tekstirea laiust ja kõrgust saab arvutada funktsioonidega *textwidth()* ja *textheight()*.

```
int far textwidth(char far *pString);
int far textheight(char far *pString);
```

Funktsiooniga *settextstyle()* saab määrata väljastamisel kasutatavat šrifti, suurust ja suunda.

```
void far settextstyle(int font, int direction, int charsize);
```

Borland C/C++ sisaldab kahte liiki šrifte: rasteršrifte ja vektoršrifte. Iga šrift on salvestatud *.CHR failis. Need failid peavad asuma funktsioonis *initgraph()* määratud kataloogis. Te võite ka muuta need failid abiprogrammiga BGIOBJ.EXE objektfailideks ja siduda oma programmiga, mis muudab valmis programmi natuke suuremaks. Rasteršriftid on salvestatud piklikus mustvalges rasterpildis. Tähestiku iga sümboli jaoks on selles rasterpildis reserveeritud eraldi ruudukene. Kui nüüd vastav täht väljastatakse ekraanile, siis kopeeritakse selle ruudukese sisu ekraani vastavasse punkti ja omistatakse valgetele punktidele eesplaani- ja mustadele tagaplaanivärvus. Selline väljastamine on väga kiire, kuid neid šrifte on raske suurendada. Suurendamisel asendatakse iga piksel mitme sama värvi pikseliga ja nii lähevad tähed sakiliseks. Vektoršriftid seevastu on salvestatud joone punktide jadana. Igale tähele vastavad mitu vektorit, mis kirjeldavad, kuidas seda tähte ekraanile joonistada. Enne ekraanile väljastamist muudetakse needki šriftid rasterpildiks, mis kiirendab väljastamist. Seda aga tehakse iga kord enne uue šrifti kasutamist ja nii ei ole kartust, et tähed kaotaksid suurendamisel oma kuju.
Funktsiooni *settextstyle()* parameeter *font* määrab järgnevates väljastusoperat-sioonides kasutatava šrifti ja võib omada järgmisi väärtusi:

Konstant	Tähendus
DEFAULT_FONT	Harilik 8 x 8 pikselise ruuduga rasteršrift.
TRIPLEX_FONT	Lihtne vektoršrift
SMALL_FONT	Väikesepunktiline vektoršrift
SANS_SERIF_FONT	Ilma serifideta vektoršrift. Serifid on sümboli tippudesse joonistatud tillukesed jooned.
GOTHIC_FONT	Kalligraafilist käsikirja meenutav vektoršrift.

Tabel 20: Borland C/C++ graafilised šriftid

Parameeter *direction* määrab, millises suunas tekst väljastatakse ja ta võib omada järgmisi väärtusi:

- HORIZ_DIR - Tekst väljastatakse vasakult paremale
- VERT_DIR - Tekst väljastatakse alt üles

Viimane parameeter - *charsize* - määrab suurenduse. Kui näiteks *charsize* väärtus on 1, siis esitatakse harilik 8 x 8 rasteršrift 8 x 8-pikselise ruudu sees. Kui aga *charsize* väärtus on 2, siis on iga täheruudu suuruseks 16 x 16 jne. Sel kombel saab šrifti suurendada kuni 10-kordseks. Vektoršriftide puhul võib määrata *charsize* väärtuseks 0, suurendus seatakse aga hoopis funktsiooniga *setusercharsize()*.

```
void far setusercharsize(int multx, int divx, int multy, int divy);
```

Funktsiooni *setusercharsize()* esimesed kaks parameetrit määravad sümboli laiuse. Need parameetrid jagatakse omavahel (multx / divx) ja saadud konstandiga korrutatakse šrifti algset laiust. Samamoodi määratakse ka šrifti uus kõrgus.

Tavaliselt joondatakse väljastatav tekst vasakule ja üles, kuid funktsiooniga *settextjustify()* saab joondada teksti ka paremale, alla või keskele.

```
void far settextjustify(int horiz, int vert);
```

Funktsiooni *settextjustify()* parameetrid *horz* ja *vert* määravad vastavalt horisontaalse ja vertikaalse joondamise, ning võivad omada järgmisi väärtusi:

Parameeter	Väärtus	Tähendus
horz	LEFT_TEXT	Vasakule joondatud tekst
	CENTER_TEXT	Horisontaalselt keskendatud tekst
	RIGHT_TEXT	Paremale joondatud tekst
vert	TOP_TEXT	Üles joondatud tekst
	CENTER_TEXT	Vertikaalselt keskendatud tekst
	BOTTOM_TEXT	Alla joondatud tekst

Tabel 21: Graafilise teksti joondamine

Määratud teksti väljastamise parameetreid saab uuesti uurida funktsiooniga *gettextsettings()*, mis salvestab parameetrite hetkelised väärtused andmestruktuuri *texttypeinfo*.

```
void far gettextsettings( struct far texttypeinfo *textinfo);
struct texttypeinfo {
  int           font;
  int           charsize;
  int           direction;
  int           horiz;
  int           vert;
};
```

Kõik eespoolnimetatud funktsioonid kasutavad hetkel kehtivat koordinaatide-süsteemi. Nii nagu tekstirezhiimis, võib ka graafilises rezhiimis defineerida ekraanile piirkonna (akna), millesse järgnevad väljastusoperatsioonid esitavad oma andmeid. Selleks kasutatakse funktsiooni *setviewport()*.

```
void far setviewport(int x1, int y1, int x2, int y2, int clip);
```

Funktsioon *setviewport()* määrab akna, mille ülemine vasak nurk asub punktis (x1, y1) ja alumine parem nurk punktis (x2, y2). Kui parameeter *clip* on nullist erinev, siis lõigatakse aknasse väljastatud graafiliste kujundite üle akna piiride ulatuvad osad ära (ei väljastata ekraanile). Funktsiooni *setviewport()* parameetrid määratakse ekraanikoordinaatides. Kui parameetrite väärtused on kehtetud, siis jääb hetkeline aken (viewport) kehtima. Akna sees asub koordinaatide süsteemi alguspunkt akna ülemises vasakus tipus. X - telg on suunatud vasakult paremale ja Y - telg ülevalt alla.
Peale uue graafilise rezhiimi määramist loob funktsioon *initgraph()* automaatselt kogu ekraani pinda hõlmava akna (viewport). Hiljem võib hetkelise akna suurust uurida funktsiooniga *getviewsettings()*.

```
void far getviewsettings(struct viewporttype far *viewportinfo);
struct viewporttype {
  int left;
  int top;
  int right;
  int bottom;
  int clip;
};
```

Funktsioon *getviewsettings()* täidab andmestruktuuri *viewporttype* vastavate andmetega.
Kogu ekraani puhastamiseks (tagaplaanivärvusega täitmiseks) kasutage funktsiooni *cleardevice()* ja hetkelise akna sisu puhastamiseks funktsiooni *clearviewport()*.

```
void far cleardevice( void );
void far clearviewport( void );
```

Sõltuvalt graafikakaardist ja valitud graafilisest rezhiimist võib eksisteerida rohkem kui üks ekraanilehekülg. Ekraanilehekülg on osa graafikakaardimälust, mis suudab antud graafilises rezhiimis salvestada kogu ekraani pikselite väärtused. Ühele ekraanileheküljele väljastamine ei muuda teise lehekülje sisu. Borland C/C++ eristab aktiivse ja nähtava ekraanilehekülje vahel. Nähtav ekraanilehekülg on hetkel kasutajale nähtav ja teda määratakse funktsiooniga

setvisualpage(). Aktiivne on see ekraanilehekülg, millele järgnevad väljastusoperatsioonid esitavad oma andmed ja teda seatakse funktsiooniga *setactivepage()*.

```
void far setvisualpage(int page);
void far setactivepage(int page);
```

Järgmised graafilised rezhiimid sisaldavad mitu ekraanilehekülge:

Ohjurprogramm	Graafikarezhiim	Ekraanilehekülgi
EGA	EGALO	4
	EGAHI	2
	EGAMONOHI	2 (kui mälu on 256K)
HERC	HERCMONOHI	2
VGA	VGALO	2
	VGAMED	2

Tabel 22: Ekraanileheküljed ja graafikarezhiimid

Ekraanilehekülgi kasutatakse sageli sujuva animatsiooni programmeerimiseks. Selleks seatakse ekraan graafikarezhiimi, mis sisaldab vähemalt kahte ekraanilehekülge. Kasutajale näidatakse kõigepealt animatsiooni esimest pilti ehk kaadrit (frame). Samal ajal joonistatakse varjatud leheküljele uut kaadrit. Kui see valmis saab, siis vahetatakse ekraanilehekülgi. Nii jääb otsene uue kaadri joonistamine kasutajale nähtamatuks ja see muudab animatsiooni sujuvamaks Seda meetodit nimetatakse kahe ekraanilehekülje animatsiooniks (*double buffer animation*).

Näiteprogramm GRADEMO

Järgmine näiteprogramm demonstreerib paljude enamlevinud graafikafunktsioonide kasutamist ning kahte mainitud animatsioonimeetodit: rasterpildiga ja kahe ekraanilehekülje abil.

GRADEMO.C

```
/*********************************************************/
/***                                                  ***/
/***   GRADEMO.C                                      ***/
/***                                                  ***/
/***   Näiteprogramm GRADEMO.EXE näitab Borland C/C++  ***/
/***   graafiliste funktsioonide kasutamist.          ***/
/*********************************************************/

/*----------------< Päisefailid >------------------------*/

#include    <graphics.h>
#include    <conio.h>   /* kbhit() */
#include    <stdio.h>   /* printf() */
#include    <stdlib.h>  /* exit(), rand(), randomize() */
#include    <string.h>  /* strcpy() */
#include    <math.h>    /* sin(), cos() */
#include    <time.h>    /* time() ->
```

```
            funktsiooni randmize() jaoks */
#include    <alloc.h>   /* farfree() */

/*----------------< Globaalsed muutujad >----------------*/

int         gmode, gdriver = DETECT, errcode;
            /* määrake siin oma vastav kataloog */
char            gpath[] = "D:\\BORLANDC\\BGI";

/*----------------< Funktsioonide prototüübid >----------*/

void PutText(char *);
void JoonDemo( void );
void KaarDemo( void );
void AnimDemo1( void );
void AnimDemo2( void );

/*=====================================================*/
/*===                                               ===*/
/*===   main()                                      ===*/
/*===                                               ===*/
/*===   Initsialiseerib graafikakaardi ja kutsub    ===*/
/*===   üksteise järel välja eri teste täitvad      ===*/
/*===   funktsioonid.                               ===*/
/*=====================================================*/
int main( void )
{
      /* määra graafiline rezhiim */
  initgraph(&gdriver, &gmode, gpath);
      /* kontrollime, kas ei tekkinud viga */
  errcode = graphresult();
       /* kui tekkis viga, siis ... */
  if (errcode != grOk)
  {
    printf("Graaafika viga: %s\n", grapherrormsg(errcode));
    printf("Vajutage suvalisele klahvile!");
    getch();
    exit(1);
  }
  settextstyle(TRIPLEX_FONT, HORIZ_DIR, 1);
      /* näitefunktsioonid */
  JoonDemo();
  KaarDemo();
  AnimDemo1();
  AnimDemo2();
  /* taastame tekstirezhiimi */
  closegraph();
  return 0;
}     /* main() */



/*-----------------------------------------------------*/
/*---                                               ---*/
/*---   PutText()                                   ---*/
/*---                                               ---*/
/*---   Esitab ekraanile kaks teksti, millest esimene ---*/
/*---   näitab testi nime ja teine soovitab vajutada  ---*/
/*---   suvalisele klahvile.                        ---*/
/*-----------------------------------------------------*/
void PutText(char *pName)
{
  int  i;

  char      szMessage1[30];
  char      szMessage2[] = "Vajutage suvalisele klahvile!";
  sprintf(szMessage1, "Funktsiooni %s näide", pName);
  i = getmaxx() / 2 - textwidth(szMessage1) / 2;
  outtextxy(i, getmaxy() / 2, szMessage1);
```

```
  outtextxy(20, getmaxy() - 30, szMessage2);
  getch();
}     /* PutText() */

/*-------------------------------------------------------*/
/*---                                                 ---*/
/*---   JoonDemo()                                    ---*/
/*---                                                 ---*/
/*---   Näitab funktsioonide line(), lineto(),        ---*/
/*---   linerel() ja moveto() kasutamist.             ---*/
/*-------------------------------------------------------*/
void JoonDemo( void )
{
  int x, y, i, k;

  clearviewport();
  /* joonistab ekraanile ruudustiku */
  for(x = 20; x < getmaxx(); x += 20)
    line(x, 0, x, getmaxy());
  for(y = 20; y < getmaxy(); y += 20)
    line(0, y, getmaxx(), y);
  PutText("line()");
  /* joonistab ekraanile  hulga kolmnurki */
  clearviewport();
  x = getmaxx() / 2;
  y = getmaxy() / 2;
  i = min(x / 20, y / 20);
  for(k = 1; k < i; k++) {
    moveto(x - k * 20, y - k * 20);
    lineto(x + k * 20, y - k * 20);
    lineto(x, y + k * 20);
    lineto(x - k * 20, y - k * 20);
  }
  PutText("lineto()");
  /* joonistab ekraanile spiraali */
  clearviewport();
  i = min(y / 10, x / 10);
  moveto(x, y);
  for(k = 1; k < i; k += 2) {
    linerel(-k * 20, 0);
    linerel(0, -k * 20);
    linerel(k * 20 + 20, 0);
    linerel(0, k * 20 + 20);
  }
  PutText("linerel()");
}     /* JoonDemo() */


/*-------------------------------------------------------*/
/*---                                                 ---*/
/*---   KaarDemo()                                    ---*/
/*---                                                 ---*/
/*---   Näitab mitmesuguste koverjoonte joonistamist. ---*/
/*---   Graafilised funktsioonid kasutavad nurkade    ---*/
/*---   määramisel kraade, kuid matemaatilised        ---*/
/*---   funktsioonid radiaane.                        ---*/
/*-------------------------------------------------------*/
void KaarDemo( void )
{
  int x, y, r, i, j;
  int protsendid[4] = { 10, 50, 25, 15 };
  char  nimed[4][10] = { "Jaanus", "Toomas", "Peeter", "Sven" };

  /* joonistab hulga kontsentrilisi ringe */
  clearviewport();
  x = getmaxx() / 2;
  y = getmaxy() / 2;
  i = min(x / 20, y / 20);
  for(j = 1; j <= i; j++)
    circle(x, y, j * 20);
```

```
  PutText("circle()");
  /* joonistab kaheksa kaart, mis suunduvad ekraani */
  /* keskpunktist ääre poole */
  clearviewport();
  i = min(x , y) / 2;
  for(j = 0; j < 8; j++)
    arc(x - i * sin((double)j * 45.0 * M_PI / 180.0),
      y - i * cos((double)j * 45.0 * M_PI / 180.0),
      -45 + j * 45, 45 + j * 45, i);
  PutText("arc()");
  /* joonistab sektordiagrammi */
  clearviewport();
  for(j = i = 0; j < 4; j++) {
    setfillstyle(LTSLASH_FILL + j, getmaxcolor() - j );
    pieslice(x, y, i, i + (int)((double)
      protsendid[j] * 3.6), min(x, y) / 2);
    fillellipse(50, 100 + j * 20, 20, 10);
    outtextxy(100, 90 + j * 20, nimed[j]);
    i += (int)((double)protsendid[j] * 3.6);
  }
  PutText("pieslice()");
}     /* KaarDemo() */


/*-------------------------------------------------------*/
/*---                                                ---*/
/*---   AnimDemo1()                                  ---*/
/*---                                                ---*/
/*---   Näitab animatsiooni programmeerimist raster-  ---*/
/*---   piltide ja funktsiooni getimage() ning       ---*/
/*---   putimage() abil.                             ---*/
/*-------------------------------------------------------*/
void AnimDemo1( void )
{
  void far *pBitmap1, *pBitmap2;
  unsigned  x, y, size;

  clearviewport();
  /* joonistame tillukese UFO, mida hakkame hiljem liigutama */
  setfillstyle(SOLID_FILL, EGA_LIGHTGRAY);
  fillellipse(100, 100, 50, 20);
  fillellipse(100, 80, 30, 10);
  setfillstyle(SOLID_FILL, EGA_RED);
  fillellipse(70, 100, 5, 5);
  fillellipse(100, 110, 5, 5);
  fillellipse(130, 100, 5, 5);
  setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
  setcolor(EGA_LIGHTBLUE);
  line(80, 70, 70, 50);
  line(120, 70, 130, 50);
  setcolor(EGA_WHITE);
  setlinestyle(SOLID_LINE, 0, NORM_WIDTH);
  size = imagesize(50, 50, 150, 120);
  /* reserveerime mälu selle pildikese ja teise samasuure */
  /* pildikese jaoks, kuhu salvestame selle koha endise sisu */
  if(NULL == (pBitmap1 = malloc(size))) {
    outtextxy(100, 100, "Ei suuda reserveerida küllaldaselt mälu!");
    getch();
    exit(1);
  }
  if(NULL == (pBitmap2 = malloc(size))) {
    outtextxy(100, 100, "Ei suuda reserveerida küllaldaselt mälu!");
    getch();
    exit(1);
  }
  /* salvestame UFO esimesse puhvrisse */
  getimage(50, 50, 150, 120, pBitmap1);
  /* nüüd on koik valmis. Puhastame ekraani ja ... */
  clearviewport();
  /* enne teatame veel kasutajale,
```

```
    kuidas funktsiooni tööd lopetada */
  outtextxy(20, getmaxy() - 30, "Vajutage suvalisele klahvile!");
  /* initsialiseerime juhuarvude generaatori */
  randomize();
  /* niikaua, kui ei ole vajutatud ühelegi klahvile */
  while(!kbhit()) {
    /* hangime kaks juhuarvu, millele vastav punkt asub ekraanil */
    x = random(getmaxx() - 100);
    y = random(getmaxy() - 70);
    /* salvestame tagapohja */
    getimage(x, y, x + 100, y + 70, pBitmap2);
    /* esitame UFO ekraanile */
    putimage(x, y, pBitmap1, COPY_PUT);
    /* väljastame selle koha endise sisu uuesti ekraanile */
    putimage(x, y, pBitmap2, COPY_PUT);
  }
  /* vabastame reserveeritud mälublokid */
  farfree(pBitmap1);
  farfree(pBitmap2);
}     /* AnimDemo1() */


/*---------------------------------------------------------*/
/*---                                                   ---*/
/*---   AnimDemo2()                                     ---*/
/*---                                                   ---*/
/*---   See funktsioon näitab animatsiooni program-    ---*/
/*---   meerimist kahe ekraanilehekülje abil.           ---*/
/*---------------------------------------------------------*/
void AnimDemo2( void )
{
  int   page, x, y;
  setgraphmode(VGAMED);
  if(graphresult() != grOk) {
    printf("Ei suuda määrata vajalikku graafikarezhiimi!");
    getch();
    exit(1);
  }
  page = 0;
  randomize();
  while(!kbhit()) {
    if(page) {
      setactivepage(0);
      setvisualpage(1);
    }
    else {
      setactivepage(1);
      setvisualpage(0);
    }
    page = !page;
    clearviewport();
    x = random(getmaxx() - 100);
    y = random(getmaxy() - 70);
    /* nüüd ei pea me enam neid pildikesi mälupuhvritesse */
    /* salvestama, kuid selle asemel tuleb nad iga kord uuesti */
    /* ekraanile joonistada. Tänu kahe puhvri kasutamisele */
    /* ei märka aga kasutaja seda, kuidas üksikuid graafilisi */
    /* kujundeid ekraanile joonistatakse */
    settextstyle(TRIPLEX_FONT, HORIZ_DIR, 1);
    outtextxy(20, getmaxy() - 30, "Vajutage suvalisele klahvile!");
    setfillstyle(SOLID_FILL, EGA_LIGHTGRAY);
    fillellipse(x + 50, y + 50, 50, 20);
    fillellipse(x + 50, y + 30, 30, 10);
    setfillstyle(SOLID_FILL, EGA_RED);
    fillellipse(x + 20, y + 50, 5, 5);
    fillellipse(x + 50, y + 60, 5, 5);
    fillellipse(x + 80, y + 50, 5, 5);
    setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
    setcolor(EGA_LIGHTBLUE);
    line(x + 30, y + 20, x + 20, y);
```

```
    line(x + 70, y + 20, x + 80, y);
    setcolor(EGA_WHITE);
    setlinestyle(SOLID_LINE, 0, NORM_WIDTH);
  }
}     /* AnimDemo2() */
```

Mälu reserveerimine

Mäu reserveerimiseks kasutatakse tavaliselt funktsioone *calloc()*, *malloc()* ja *realloc()* või siis nende vastavaid funktsioone üldisest vabast mälust reserveerimiseks: *farmalloc()*, *farcalloc()* ja *farrealloc()*. Neid funktsioone käsitleti juba peatükis "Mälu haldamine". Tüüpiliselt loovutavad need funktsioonid reserveeritud mälublokile näitava viida. See viit on alati tüübist *void ** või *void far ** ja teda tuleb seepärast soovitud tüübiks konverteerida. Kui soovitud suurusega mälublokki ei õnnestunud reserveerida, siis loovutavad need funktsioonid väärtuse NULL. Saadud viita tuleb alati kontrollida, sest kui te üritate kasutada NULL viita, siis katkestab operatsioonisüsteem teie programmi töö. Funktsioonid *calloc()* ja *farcalloc()* täidavad reserveeritud mälubloki nullidega, mis ka enamiku programmide jaoks sobib. Sümbolite jada puhul on lõpumärgiks null ja ka andmestruktuuride salvestamisel kasutatakse sageli salvestatud struktuuride loendamise asemel nullidega täidetud andmestruktuuri elemendi lõpus. Nimetatud funktsioonid on defineeritud päisefailis ALLOC.H, kuid ka päisefail STDLIB.H sisaldab nende definitsioone.
Päisefailis MALLOC.H on defineeritud funktsioonid *alloca()* ja *stackavail()*.

```
size_t stackavail( void );
void * alloca(size_t size);
```

Funktsioon *alloca()* reserveerib mälu programmi pinult. Saadud viit tuleb salvestada hetkelise funktsiooni kohalikku muutujasse. Peale selle funktsiooni töö lõppu vabastatakse reserveeritud mälu automaatselt, ilma et te kasutaksite funktsiooni *free()*. Teadmaks, kui palju mälu programmi pinult veel on vaba, kasutage funktsiooni *stackavail()*. Neid funktsioone kasutavad nii DOSi, kui ka osa UNIXi translaatoreist, kuid need ei ole standardsed ANSI C funktsioonid.
Päisefail MEM.H defineerib mitmeid funktsioone mälublokkide haldamiseks. Sarnaseid ja osalt samu funktsioone on defineeritud ka päisefailis STRING.H. Päisefail STRING.H defineerib funktsioone sümbolijadade haldamiseks. Põhiline vahe on selles, et sümbolijadu töötlevad funktsioonid oletavad, et tegemist on C - keele sümbolijadadega. Viimased koosnevad üksikutest baitidest, kusjuures jada lõpus on sümbol '\0'. Mälublokke haldavad funktsioonid seevastu ei omista mingit tähelepanu üksikute baitide väärtusele, vaid vajavad mälubloki pikkust määravat parameetrit.
Mälublokke haldavad funktsioonid sisaldavad kolme funktsiooni mälu kopeerimiseks:

```
void * memccpy(void *dest, const void * src, int c, size_t n);
void * memcpy(void *dest, const void * src, size_t n);
void * memmove(void *dest, const void * src, size_t n);
```

Kõik nimetatud funktsioonid kopeerivad viida *src* poolt määratud mälublokist *n* baiti viida *dest* poolt näidatud mälublokki. Funktsioon *memccpy()* võib kopeerimise ka enne *n* baidi teisaldamist lõpetada, kui ta satub baidile väärtusega *c*, mis on sel juhul viimaseks kopeeritud baidiks. Funktsioonid *memcpy()* ja *memmove()* kopeerivad kindlasti määratud arvu baite. Kui aga kopeeritavad piirkonnad omavahel kattuvad, siis on funktsiooni *memcpy()* tulemus ebakorrektne. Funktsioon *memmove()* seevastu kopeerib ka kattuvate või osaliselt kattuvate piirkondade puhul õigesti, kuid on seetõttu aeglasem. Funktsioonide *memcpy()* ja *memmove()* tulemuseks on viit, mis langeb kokku funktsiooni esimese parameetriga - *dest*. Kui funktsioon *memccpy()* lõpetas oma töö väärtusega *c* baidi leidmise tulemusena, siis näitab funktsiooni poolt loovutatav viit sellele järgnevale baidile, vastasel juhul on funktsiooni tulemuseks NULL.
Funktsioonid *_fmemccpy()*, *_fmemcpy()* ja *_fmemmove()* täidavad samu ülesandeid üldisest mälust reserveeritud mälublokkide puhul.

```
void far * _fmemccpy(void far *dest, const void far * src, int c,
size_t n);
void far * _fmemcpy(void far *dest, const void far * src, size_t n);
void far * _fmemmove(void far *dest, const void far * src, size_t n);
```

Funktsioon *memset()* ja *_fmemset()* täidavad parameetri *ptr* poolt osutatud mälubloki esimesed *n* baiti väärtusega *c*.

```
void* memset(void* ptr, int c, size_t n);
void far* _fmemset(void far* ptr, int c, size_t n);
```

Mälublokkide võrdlemiseks võib kasutada funktsioone *memcmp()* je *memicmp()* (ning vastavaid funktsioone *_fmemcmp()* ja *_fmemicmp()*).

```
int memcmp(const void* ptr1, const void* ptr2, size_t n);
int memicmp(const void* ptr1, const void* ptr2, size_t n);
int _fmemcmp(const void far* ptr1, const void far* ptr2, size_t n);
int _fmemicmp(const void far* ptr1, const void far* ptr2, size_t n);
```

Funktsioon *memcmp()* võrdleb baidikaupa viitade *ptr1* ja *ptr2* poolt määratud mälublokkide *n* esimest baiti. Funktsiooni tulemus on:

- negatiivne - kui esimese mälubloki baitide väärtus oli väiksem
- null - kui mälublokkide sisu oli sama
- positiivne - kui esimese mälubloki baitide väärtus oli suurem

Seejuures ei mõjuta funktsioon *memcmp()* mingil määral baitide väärtust enne nende võrdlemist, funktsioon *memicmp()* seevastu võrdleb ilma väike- ja suurtähtede vahel vahet tegemata.
Mingi kindla väärtuse otsimiseks kasutage funktsiooni *memchr()* (või funktsiooni *_fmemchr()*)

```
void* memchr(void* ptr, int c, size_t n);
void far* _fmemchr(void far* ptr, int c, size_t n);
```

Funktsioon *memchr()* otsib parameetri *ptr* poolt määratud mälublokist *n* baidi ulatuses baiti väärtusega *c* ja loovutab esimesele sellisele baidile näitava viida. Kui nimetatud mälublokk sellist baiti ei sisalda, siis on funktsiooni tulemuseks NULL.
Päisefail STRING.H sisaldab kõiki eespool nimetatud funktsioone ja lisaks sellele funktsioone sümbolijadade töötlemiseks. Sümbolijadade kopeerimiseks võib kasutada funktsiooni *strcpy()*.

```
char* strcpy(char *dest, const char*  src);
char far* _fstrcpy(char far*dest, const far* src);
```

Funktsioon *strcpy()* kopeerib parameetri *src* poolt osutatud sümbolijada parameetri *dest* poolt osutatud puhvrisse ja loovutab viida nimetatud puhvrile. Esimene sümbolijada peab olema lõpetatud sümboliga '\0' (ehk arvuga 0), mis samuti kopeeritakse teise puhvrisse.
Funktsioon *stpcpy()* erineb funktsioonist *strcpy()* vaid selle poolest, et esimese sümbolijada lõpumärki ei kopeerita teise puhvrisse ja seega tuleb see vajaduse korral sinna ise lisada. Selle funktsiooniga saab aga kirjutada juba olemasoleva sümbolijada keskele mingeid sümboleid, ilma seejuures nimetatud jada lühendamata.

```
char* stpcpy(char *dest, const char*  src);
char far* _fstpcpy(char far*dest, const far* src);
```

Kindla arvu sümbolite kopeerimiseks kasutage funktsiooni *strncpy()*.

```
char* strncpy(char *dest, const char*  src, size_t size);
char far* _fstrncpy(char far*dest, const char far*  src, size_t
size);
```

Funktsioon *strncpy()* kopeerib maksimaalselt *size* baiti parameetri *src* poolt osutatud puhvrist parameetri *dest* poolt osutatud puhvrisse. Kui esimene puhver sisaldas vähem sümboleid, siis kopeeritakse vaid need. Kui ta aga sisaldab rohkem sümboleid, siis kopeeritakse vaid esimesed *size* baiti.
Funktsioon *strdup()* loob teise samasuguse sümbolitejada. Selleks reserveerib ta funktsiooni *malloc()* abil sobiva suurusega mälubloki, kopeerib soovitud sümbolitejada loodud puhvrisse ja loovutab nimetatud puhvrile näitava viida.

```
char * strdup(const char* src);
char far* _fsrtdup(const char far* src);
```

Sümblitejada pikkuse mõõtmiseks tuleks kasutada funktsiooni *strlen()*.

```
size_t strlen(const char* ptr);
size_t far _fstrlen(const char far* ptr);
```

Vajaliku sümboli otsimiseks võib kasutada funktsiooni *strchr()*.

```
char* strchr(const char* ptr, int c);
char far* _fstrchr(const char far* ptr, int c);
```

Kui funktsioon leiab soovitud sümboli määratud jadast, siis loovutab ta sellele sümbolile näitava viida, vastasel juhul aga väärtuse NULL. Seejuures loetakse lõpumärk '\0' jada osaks, näiteks:

```
char nimi[] = "Jaanus", *ptr;
...
ptr = strchr(nimi, 0);
```

loovutab sümbolitejada lõppu näitava viida. Funktsioon *strchr()* otsib niisiis soovitud sümboli esimese positsiooni määratud jadas. Kui aga nimetatud sümbol esineb määratud jadas mitu korda, siis võib vaja olla otsida ka esimesele positsioonile järgnevaid positsioone. Selleks võite kasutada samuti funktsiooni strchr(), loovutades talle igal järgmisel korral eelmisel korral saadud viida, kuni funktsiooni väärtus on NULL, näiteks:

```
char string[] = "metallitehnoloogia", *ptr;
...
ptr = string;
while(NULL != (ptr = strchr(ptr, 'l'))) {
  printf("Tähe l %d positsioon: %s\n", (int)string - (int)ptr + 1,
ptr);
}
```

Toodud näites väljastatakse tähe 'l' iga positsioon määratud sümbolijadas. Seejuures kasutatakse tõsiasja, et viidad on lihtsalt aadressid. Kui nad eelnevalt konverteerida täisarvudeks, siis saab neid üksteisest lahutada ja saadud vahe annabki otsitud sümboli positsiooni. Kuna aga C - keel alustab loendamisega nullist, mitte aga ühest, siis tuleb veel liita 1.
Viimase positsiooni leidmiseks on lihtsam kasutada funktsiooni *strrchr()*.

```
char* strrchr(const char* ptr, int c);
char far* _fstrrchr(const char far* ptr, int c);
```

Funktsioon *strrchr()* erineb funktsioonist *strchr()* vaid selle poolest, et ta alustab otsimisega sümbolitejada lõpust ja liigub alguse poole.
Paljude sümbolite otsimiseks sümbolijadast tuleks kasutada funktsiooni *strstr()*.

```
char* strstr(char* str1, char* str2);
char far* _fstrstr(char far* str1, char far* str2);
```

Funktsioon *strstr()* otsib sümbolitejadast *str1* jada *str2* esimest positsiooni ja loovutab sellele näitava viida või väärtuse NULL, kui soovitud jada ei leidunud.
Sümbolijadade võrdlemiseks kasutage funktsioone *strcmp()*, *stricmp()* ja *strcmpi()*.

```
int strcmp(const char *s1, const char *s2);
int strcmpi(const char *s1, const char *s2)
int stricmp(const char *s1, const char *s2);
int far _fstrcmp(const char far *s1, const char far *s2);
int far _fstricmp(const char far *s1, const char far *s2);
```

Kõik need funktsioonid võrdlevad üle antud sümbolijadasid ja loovutavad väärtuse, mis näitab, kas jada s1 on väiksem (negatiivne väärtus), suurem (positiivne väärtus) või võrdne (0) jadaga s2. Funktsioonid *strcmpi()* ja *stricmp()* loevad suujuures väikesed ja suured tähed võrdseks. Näiteks *stricmp("AAA", "aaa")* annab tulemuse 0, aga *strcmp("AAA", "aaa")* annab negatiivse tulemuse, kuna suurtähed omavad väiksemat ASCII koodi kui väiketähed.
Järgnevad funktsioonid täidavad sama ülesannet, kui eespool toodud funktsioonid, kuid kontrollivad vaid mõlema jada esimesi *maxlen* sümbolit.

```
int strncmp (const char *s1, const char *s2, size_t maxlen);
int strncmpi(const char *s1, const char *s2, size_t maxlen);
int strnicmp(const char *s1, const char *s2, size_t maxlen);
int far _fstrncmp (const char far *s1, const char far *s2, size_t
maxlen);
int far _fstrnicmp(const char far *s1, const char far *s2, size_t
maxlen);
```

Sümbolijadade uurimiseks kasutatakse veel funktsioone *strspn()* ja *strcspn()*.

```
size_t strcspn(const char *s1, const char *s2);
size_t strspn(const char *s1, const char *s2);
size_t far _fstrcspn(const char *s1, const char far *s2);
size_t far _fstrspn(const char far *s1, const char far *s2);
```

Funktsioon *strspn()* ostsib jadast *s1* jadas *s2* leiduvaid sümboleid ja loovutab kõige pikema jada *s1* selle osa pikkuse, mis koosneb ainult jadas *s2* leiduvatest sümbolitest. Viimaste sümbolite hulka ei loeta aga jada *s2* lõpumärki - '\0'. Funktsioon *strcspn()* seevastu loovutab jada *s1* suurima osa pikkuse, mis ei sisalda ühtegi jadas *s2* leiduvatest sümbolitest.
Programmi argumentide töötlemiseks kasutatakse sageli funktsiooni *strtok()*.

```
char* strtok(const char* str1, const char* str2);
char far* _fstrtok(const char far* str1, const char far* str2);
```

Funktsioon *strtok()* käsitleb sümbolite jada s1 kui mitmest osajadast koosnevat jada, mille osasid lahutavad jadas s2 leiduvad sümbolid. Näiteks kui te loeksite mingist *.BAT failist järgmise käsu:

```
      ARJ.EXE  a -r  -p  -va A:\ARHIIV C:\DOC\*.*
```

ja sooviksite seda käsku jaotada programmi nimest ja eraldi tähistest koosnevateks osajadadeks, siis saaksite kasutada funktsiooni *strtok()*. Nagu näete, koosneb toodud käsk osajadadest, mis on üksteisest lahutatud ühe või enama tühiku või tabulaatoriga. Seega tuleks nüüd kasutada funktsiooni *strtok()* osajadade (*token*) leidmiseks järgmiselt:

```
char  jada[] = "ARJ.EXE  a -r  -p  -va A:\ARHIIV C:\DOC\*.*", *ptr;
...
ptr = strtok(jada, " \t");
while(NULL != (ptr = strtok(NULL, " \t"))) {
  printf("Järgmine osajada on: %s", ptr);
}
```

Esimene kord funktsiooni *strtok()* kasutades loovutab funktsioon esimesele osajadale näitava viida ja sisestab jadasse selle osajada lõppu sümboli '\0'. Iga järgmine funktsiooni *strtok()* kasutamine esimese parameetriga NULL loovutab järgmisele osajadale näitava viida ja lisab tolle lõppu jällegi sümboli '\0'. See kestab kuni funktsioon *strtok()* loovutab väärtuse NULL näitamaks, et rohkem osajadasid ei leidu. Funktsioon *strtok()* niisiis purustab uuritava jada töö käigus (jaotab ta osajadadeks). Kui see jada on teile veel vajalik, siis salvestage ta sisu mujale, kasutades selleks näiteks funktsiooni *strdup()*.
Sümbolijadade ühendamiseks kasutage funktsiooni *strcat()* või *strncat()*.

```
char *strcat(char *dest, const char *src);
char far * far _fstrcat(char far *dest, const char far *src);
char *strncat(char *dest, const char *src, size_t maxlen);
char far * far _fstrncat(char far *dest, const char far *src, size_t
maxlen);
```

Funktsioon *strcat()* lisab jada *src* jada *dest* lõppu. Jada *dest* peab seega olema küllalt suur, mahutamaks mõlemat jada. Funktsioon *strncat()* kopeerib maksimaalselt *maxlen* sümbolit jadast *src* jada *dest* lõppu.
Sümbolite jada täitmiseks mingi kindla sümboliga kasutage funktsioone *strset()* ja *strnset()*.

```
char *strset(char *s, int ch);
char far * far _fstrset(char far *s, int ch);
char *strnset(char *s, int ch, size_t n);
char far * far _fstrnset(char far *s, int ch, size_t n);
```

Funktsioon *strset()* täidab kogu jada *s* sümbolitega *ch*. Funktsioon *strnset()* täidab aga vaid esimesed *n* jada *s* sümbolit sümbolitega *ch*.
Sümbolite jada sisu konvetreerimiseks võib kasutada funktsioone *strlwr()*, *strupr()* ja *strrev()*.

```
char *strlwr(char *s);
char far * far _fstrlwr(char far *s);
char *strupr(char *s);
char far * far _fstrupr(char far *s);
char *strrev(char *s);
char far * far _fstrrev(char far *s);
```

Funktsioon *strlwr()* konverteerib kõik jada *s* suurtähed (A-Z) vastavateks väiketähtedeks. Funktsioon *strupr()* seevastu konverteerib kõik jada *s* väiketähed suurtähtedeks. Ning lõpuks funktsioon *strrev()* pöörab jada ümber, s.t. viimane sümbol saab esimeseks, lõpust teine teiseks jne. Näiteks *strrev("ABC")* annab tulemuseks jada "CBA".

Kontrollfunktsioonid

Sageli võib programmistruktuurist lähtudes eeldada, et üks või teine muutuja omab antud hetkel mingit kindlat väärtust. Näiteks enne mälublokile näitava viida kasutamist võime oletada, et too viit ei oma väärtust NULL. Kui see aga nii oleks ja me ikkagi seda viita kasutaksime, siis oleks tulemuseks raske viga. Seega tuleks mainitud oletust enne viida kasutamist kontrollida ja kui ta ei kehti

ning mingit muud võimalust programmi tööd jätkata ei ole, siis tuleks programmi töö ise lõpetada. Selliseks otstarbeks on loodud päisefail ASSERT.H, mis sisaldab vaid ühe makro - *assert()*.

```
void assert(int test);
```

Makro *assert()* sisaldab tingimuslauset *if*, mis kontrollib, kas talle üle antud avaldis omab väärtust 0 (tõeväärtus VÄÄR). Kui see nii on, siis väljastab ta voole *strerr* teate:

```
Assertation failed: <test> file <faili nimi> line <rea number>
```

ning kutsub seejärel välja funktsiooni *abort()* programmi lõpetamiseks.

```
void abort( void );
```

Funktsioon *abort()* on defineeritud päisefailis STDLIB.H ning ka päisefailis PROCESS.H. See funktsioon väljastab voole *stderr* teate:

```
Abnormal programm termination
```

ja kasutab seejärel programmi töö lõpetamiseks funktsiooni *_exit()*.
Makrot *assert()* kasutatakse enamasti programmi kontrolli käigus, kui veel ei olda kindel programmi korrektses töös. Hiljem võib valmis programmist need kontrollid eemaldada, et programmi töökiirust tõsta. Selleks tuleks sisestada programmi teksti enne eeltranslaatori käsku *#include <ASSERT.H>* käsk *#define NDEBUG*. Viimane käsk määrab, et programm tuleks transleerida valmis kujul ja sel juhul asendab eeltranslaator makro *assert()* tühikuga.
Päisefail CTYPE.H defineerib sümbolite klassifitseerimiseks hulga funktsioone. Sümboleid võib jagada tähtedeks (mis omakorda jagunevad väike- ja suurtähtedeks), numbriteks, kirjavahemärkideks (punkt, koma, semikoolon jne.) ja muudeks. Nimetatud funktsioonid vajavd kõik ühte parameetrit - uuritavat sümbolit, ja loovutavad positiivse väärtuse, kui nimetatud sümbol kuulus vastavasse sümbolite klassi ning vastasel juhul nulli. Järgnevas tabelis näete nende funktsioonide ülevaadet.

Funktsioon	Sümbolite klass
int isalnum(int c);	c on harilik täht või number: A - Z, a - z, 0 - 9
int isalpha(int c);	c on harilik täht: A - Z, a - z
int isascii(int c);	c esimene bait omab väärtust piirides 0 - 127
int iscntrl(int c);	c vastab klahvile <DEL> või mingile laiendatud ASCII koodiga kontrollklahvile, mille esimene bait on null: 0x7F või 0x00 kuni 0x1F.
int isdigit(int c);	c on number: 0 - 9
int isgraph(int c);	c on ekraanile väljastatav sümbol, s.t. sama mis *isprint()*, ainult et seekord ei loeta tühikut (0x20) õigete sümbolite hulka.

Tabel 23: Sümbolite klassifitseerimisfunktsioonid

Funktsioon	Sümbolite klass
int islower(int c);	c on väiketäht: a - z

int isprint(int c);	c on ekraanile trükitav täht: 0x20 - 0x7E. Väiksemad sümbolid kujutavad endast kontrollkoode, mille väljastamine ekraanile ei esita sinna mingit sümbolit, vaid hoopis täidab mingi operatsiooni, näiteks 0x07 väljastab hoiatava helisignaali.
int ispunct(int c);	c on kirjavahemärk või kontrollkood: (isspace() || iscntrl())
int isspace(int c);	c on tühik, horisontaalne või vertikaalne tabulaator, reavahetuse märk või uue rea märk: ' ', '\t', '\v', '\n', '\r'
int isupper(int c);	c on suurtäht: A - Z
int isxdigit(int c);	c on number kuueteistkümnendsüsteemis: 0 - 9, A - F, a - f

Tabel 24: Sümbolite klassifitseerimisfunktsioonid (järg)

Päisefail CTYPE.H sisaldab ka funktsioone sümbolite konverteerimiseks soovitud sümbolite klassi.

```
int tolower(int ch);
int _tolower(int ch);
int toupper(int ch);
int _toupper(int ch);
```

Funktsioon *tolower()* konverteerib talle üleantud suurtähe vastavaks väiketäheks. Kui antud täht juba on väiketäht või ei ole üldsegi täht, siis seda ei muudeta. Makro *_tolower()* täidab sama ülesande, kuid ta ei kontrolli enne tähe tegelikku väärtust ja seega tuleks teda kasutada vaid siis, kui antud täht kindlasti on suurtäht. Ta töötab on aga kiiremini kui funktsioon *tolower()*. Sarnaselt konverteerib funktsioon *toupper()* iga väiketähe vastavaks suurtäheks ja makro *_touper()* teeb sama ilma eelneva kontrollita.
Kindlustamaks, et mingi sümbol on ASCII täht, tuleks kasutada makrot *toascii()*.

```
int toascii(int ch);
```

Makro *toascii()* kindlustab, et arvu *ch* kõik bitid, väljaarvatud esimesed 7, on nullid. Seega on tulemus mingi väärtus piires 0 - 127, mis vastab tähtede ASCII koodidele.
Arvutis kasutatavad muutujad omavad kõik kindlat pikkust ja seega on arvuti muutujate väärtused piiratud. Nende piiride ületamine võib tekitada üsna üllatavaid tulemusi. Näiteks täisarv (int) kasutab kahte baiti ja võib seega salvestada väärtusi piirides - 32.768 kuni 32.767. Kui te sellisele muutujale omistate algul väärtuse 32.767 ja seejärel liidate veel ühe, siis ei ole tulemuseks mitte 32.768 vaid hoopiski - 32.768. See on tingitud täisarvude salvestamise viisist. Pikkade täisarvude (*long*) puhul tuleb see piir hiljem, kuid ka neil arvudel on suurim võimalik väärtus. Need piirid on defineeritud sümboolsete konstantidena päisefailis LIMITS.H
Järgmises tabelis näete nimetatud konstante ja neile vastavaid andmetüüpe.

Andmetüüp	Konstant	Väärtus
char	CHAR_BIT	bittide arv (8)
	CHAR_MAX	maksimaalne väärtus (127)
	CHAR_MIN	minimaalne väärtus (-128)
unsigned char	UCHAR_MAX	maksimaalne väärtus (255)
signed char	SCHAR_MAX	maksimaalne väärtus (127)
	SCHAR_MIN	minimaalne väärtus (-128)
int	INT_MAX	maksimaalne väärtus (32.767)
	INT_MIN	minimaalne väärtus (-32.768)
unsigned int	UINT_MAX	maksimaalne väärtus (65.535)
short	SHRT_MAX	maksimaalne väärtus (32.767)
	SHRT_MIN	minimaalne väärtus (-32.768)
unsigned short	USHRT_MAX	maksimaalne väärtus (65.535)
long	LONG_MAX	maksimaalne väärtus (2.147.483.647)
	LONG_MIN	minimaalne väärtus (-2.147.483.648)
unsigned long	ULONG_MAX	maksimaalne väärtus (4.294.967.295)

Tabel 25: Andmetüüpide minimaalsed ja maksimaalsed väärtused

Tabelis toodud väärtused kehtivad aga ainult 8086 ja 80286 tüüpi protsessoriga arvutil. 80386 protsessori puhul on näiteks andmetüübi *int* piirid võrdsed tüübi *long* omadega. Seda muidugi ainult juhul, kui ka translaator on selle protsessori jaoks loodud. Nende erinevuste tõttu ongi soovitavam kasutada otseste väärtuste asemel siintoodud sümboolseid konstante. Teistsuguse protsessori puhul on piirid teistsugused, kuid ka vastavate sümboolsete konstantide väärtused on sellele vastavalt muudetud.
Murdarvude puhul kasutatakse veel konstanti HUGE_VAL, mis sümboliseerib murdarvu suurimat (ja -HUGE_VAL tema vähimat) väärtust. See konstant on defineeritud päsefailis MATH.H

Operatsioonisüsteemi funktsionid

Arvuti püsimälu (ROM) sisaldab mitmeid kasulikke funktsioone, nende kasutamine on võimalik katkestuse vahendusel. Asja lihtsustamiseks sisaldab päisefail BIOS.H vastavaid C - keele funktsioone. Need funktsioonid kasutavad oma ülesande täitmiseks tegelikult BIOSi funktsioone katkestuste vahendusel ja võimaldavad teile seega BIOSi funktsioonide kasutamise ilma assemblerkeele abita. BIOSi funktsioonide hulka kuuluvad näiteks otsene kettaseadme sektorite lugemine või kirjutamine, arvuti kella seadmine ja lugemine jms. Nimetatud funktsioonide kasutamisega tuleks olla ettevaatlik. Funktsioon *biosdisk()* näiteks loeb või kirjutab otseselt kettaseadmele, ilma seejuures kettal loodud kataloogidele või failidele tähelepanu omistamata. Selle funktsiooni vigane kasutamine võib muuta teie kettaseadme sisu mitteloetavaks.

Teisest küljest saab aga selle funktsiooniga luua mitmeid kasulikke programme, nagu näiteks kettaseadise sisu optimeerija (nagu Nortoni SpeedDisk), mille jaoks harilikud C - funktsioonid ei sobi.
Ka operatsioonisüsteem DOS sisaldab mitmeid kasulikke funktsioone, mida programmid samuti kasutavad katkestuste vahendusel. Sellesse kategooriasse kuuluvad mälu reserveerimine DOSi funktsiooniga, kettaseadise sektorite või failide kirjutamine ja lugemine, mälu sisu muutmine kindlal aadressil ning palju muud. Neilegi vastavad Borland C funktsioonid, mis on defineeritud päisefailis DOS.H. Ka nende funktsioonidega tuleks olla ettevaatlik. Peatükis "Katkestused" vaadeldud funktsioonid *int86()*, *intr()* ja *int86x()* on samuti defineeritud päisefailis DOS.H
Päisefail PROCESS.H sisaldab funktsioone teiste programmide startimiseks, lõpetamiseks ja operatsioonisüsteemi käskude kasutamiseks.
Funktsioon *system()* kasutab soovitud käsu täitmiseks operatsioonisüsteemi käsuprogrammi COMMAND.COM.

```
int system(const char *command);
```

Näiteks: system("DIR");
See funktsioon peatab hetkeks programmi töö, stardib kommandoprogrammi COMMAND.COM, laseb tal täita soovitud käsu ja jätkab seejärel programmi tööga. Kui käsk täideti, siis on funktsiooni tulemuseks 0, vastasel juhul on funktsiooni tulemuseks veakood.
Arvuti võib täita korraga mitut programmi. Kui te ühe programmiga (näiteks editoriga) hetkel töötate, siis võib arvuti kasutatada protsessori jõudehetki, täit-maks tagaplaanil veel teisi programme, mis hetkel kasutajalt mingeid juhiseid ei vaja. Seda nimetatakse mitmiktegumrezhiimiks. Programm on tegelikult vaid *.EXE fail kettal, mis kirjeldab, kuidas mingit ülesannet täita. Kui te selle programmi stardite, siis laaditakse tema kood mällu. Seda - täidetavat programmi - nimetatakse tegumiks (*instance, task*). Tegum omab kindlaid muutujate väärtusi, kuid kasutab oma ülesande täitmiseks oma programmi poolt kindlaks määratud viisi - algoritmi. Te võite ka operatsioonisüsteemis DOS laadida korraga mällu mitu programmi, kuid igal hetkel saab vaid üks neist korraga aktiivne olla ja protsessor ei suuda oma jõudehetkil teisi programme täita. Teised operatsioonisüsteemid aga, nagu näiteks UNIX ja OS/2, suudavad seda.
Mitmesuguste *exec...()* ja *spawn...()* funktsioonidega saab teisi programme või ka sama programmi veel kord mällu laadida ja uut tegumit luua. Funktsioonid *exec...()* laadivad teise programmi mällu ja stardivad selle. Funktsioonid *spawn...()* teevad sedasama ja jätkavad peale uue programmi lõppu vana programmi täitmist.
Kui te teate, kui palju argumente te uuele programmile üle andma peate, siis kasutage funktsioone:

```
int execl  (char *path, char *arg0, ..., NULL);
int execle (char *path, char *arg0, ..., NULL, char **env);
int execlp (char *path, char *arg0, ...);
```

```
int execlpe(char *path, char *arg0, ..., NULL, char **env);
```
või:
```
int spawnl  (int mode, char *path, char *arg0, ..., NULL);
int spawnle (int mode, char *path, char *arg0, ..., NULL, char
*envp[]);
int spawnlp (int mode, char *path, char *arg0, ..., NULL);
int spawnlpe(int mode, char *path, char *arg0, ..., NULL, char
*envp[]);
```

Funktsioonide *exec...()* esimene parameeter määrab starditava programmi täieliku nime koos tema kataloogiteega ja vajaduse korral ka kettaseadme nimega. Sellele järgnevad programmi parameetrid. Viimane parameeter on NULL, mis näitab, et sellele rohkem parameetreid ei järgne. Näiteks:

```
execl("C:\\NC\\ARJ.EXE", "a", "-va", "A:\\DOCS.ARJ", "C:\\DOCS\*.*",
NULL);
```

Funktsioonid *execle()* ja *spawnle()* võimaldavad muuta ka uue tegumi keskkonnamuutujaid. Keskkonnamuutujad (*environment variables*) määratakse tavaliselt failis AUTOEXEC.BAT ja muudes *.BAT failides. Nende kuju on järgmine:

```
[SET] <muutuja nimi> = <väärtus>
```

Näiteks:
```
PATH=C:\DOS;C:\NC;C:\WINDOWS;C:\BORLANDC\BIN;
```
või
```
SET LIB=C:\BORLANDC\LIB;C:\BORLANDC\CLASSLIB\LIB;
```

Keskkonnamuutujaid kasutatakse näiteks kataloogide määramiseks, kust programmid otsivad mingeid andmeid sisaldavaid faile jms. Soovikorral võite salvestada sobivad muutujad koos uute väärtustega sümbolijadadesse, mis omakorda salvestatakse massiivi ja anda sellele massiivile näitav viit üle funktsioonile *execle()* või *spawnle()*. Kui te uuele programmile mingeid keskkonnamuutujaid ei määra, siis kasutab ta vana programmi keskkonnamuutujate väärtusi.

Funktsioonid *execlp()* ja *spawnlp()* ei vaja enam starditava programmi täielikku nime, vaid kasutavad selle programmi otsimiseks keskkonnamuutuja PATH väärtust .

Funktsioonid *execlpe()* ja *spawnlpe()* lubavad määrata uusi keskkonnamuutujaid ja kasutavad ka muutujat PATH.

Funktsioonid *spawn...()* vajavad veel parameetrit *mode*, mis määrab uue ja vana programmi koostöö viisi. See parameeter võib omada üht järgnevatest väärtustest.

- P_WAIT - Vana programm ootab, kuni uus oma tööga valmis saab ja jätkab siis oma tööd.
- P_NOWAIT - Vana ja uus programm töötavad üheaegselt edasi. See nõuaks mitmiktegumrezhiimi ja ei ole esialgu operatsioonisüsteemis DOS võimalik. Selle parameetri kasutamine tekitab veateate.

- P_OVERLAY - Uus programm loetakse mällu samasse kohta, kus oli vana programm ja seega lõpetatakse vana programmi töö. See vastab funktsioonide *exec...()* tööle.

Kui te aga ei tea täpset programmi parameetrite arvu, siis kasutage funktsioone:

```
int execv  (char *path, char *argv[]);
int execve (char *path, char *argv[], char **env);
int execvp (char *path, char *argv[]);
int execvpe(char *path, char *argv[], char **env);
```

või

```
int spawnv  (int mode, char *path, char *argv[]);
int spawnve (int mode, char *path, char *argv[], char *envp[]);
int spawnvp (int mode, char *path, char *argv[]);
int spawnvpe(int mode, char *path, char *argv[], char *envp[]);
```

Nimetatud funktsioonid erinevad eespooltoodutest vaid selle poolest, et nüüd ei anta iga parameeter otse funktsioonile üle, vaid salvestatakse sümbolijadade massiivi *argv* ja antakse seejärel üle kogu massiiv. See võimaldab määrata starditava programmi parameetrite arvu ja sisu alles vana programmi käigus.

Failid ja kataloogid

Failide avamist, lugemist, kirjutamist, loomist ja kustutamist vaatlesime juba eespool. Kataloogide lugemiseks ja neis salvestatud failide pikkuste, nimede jms uurimiseks kasutatakse päisefailis DIR.H defineeritud funktsioone.

Operatsioonisüsteem DOS kasutab nn. hetkelise kataloogi mõistet. Igal hetkel on mingi kataloog aktiivne. Kui te olete näiteks avanud kataloogi C:\NC ja stardite mingi programmi, siis otsitakse seda programmi kõigepealt hetkelisest kataloogist ja seejärel keskkonnamuutujaga PATH määratud kataloogidest. Samuti loetleb käsk DIR hetkelise kataloogi failid, kui te just otse ei määra, millise kataloogi failide loetelu te soovite. Hetkelist kataloogi saab muuta käsuga CD või CHDIR. Windows lubab juba teatud piiratud mitmik-tegumrezhiimi ja seepärast peavad Windowsi programmid ise arvet oma hetkelise kataloogi üle. Kui üks neist oma hetkelist kataloogi muudab, siis ei mõjuta see Windowsi teiste programmide tööd. DOSi programmid seevastu on selles suhtes üksteisest sõltuvad.

Hetkelise kataloogi sisu lugemiseks kasutage funktsioone *findfirst()* ja *findnext()*.

```
int findfirst(const char* pathname, struct ffblk* fileblock, int
attrib);
int findnext(struct ffblk* fileblock);
```

Funktsioon *findfirst()* leiab parameetriga *pathname* määratud kataloogist (või hetkelisest kataloogist) esimese faili, mis vastab parameetriga *pathname* määratud mustrile ja omab parameetriga *attrib* määratud atribuute. Parameeter *pathname* võib sisaldada sümboleid '*' (suvaline arv tundmatuid sümboleid) ja '?' (üks tundmatu sümbol). Parameeter *attrib* määrab faili atribuudid ja võib sisaldada järgmiste konstantide kombinatsioone:

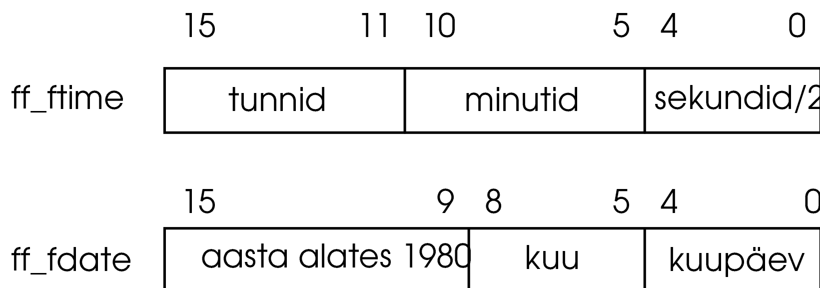
1. FA_RDONLY - faili tohib vaid lugeda
2. FA_HIDDEN - varjatud fail

3. FA_SYSTEM - operatsioonisüsteemi fail
4. FA_LABEL - ketta nimi
5. FA_ARCH - arhiveeritud (komprimeeritud) fail
6. FA_DIREC - kataloog

Leitud faili andmed andmed salvestatakse andmestruktuuri *ffblk*.

```
struct ffblk {
  char ff_reserved[21]; /* reserveeritud DOSi jaoks */
  char ff_attrib;       /* faili atribuut */
  int  ff_ftime;        /* viimase muutmise aeg */
  int  ff_fdate;        /* viimase muutmise kuupäev */
  long ff_fsize;        /* fail suurus baitides */
  char ff_fname[13];          /* faili nimi */
};
```

Elemendid ff_ftime ja ff_fdate on 16-bitised täisarvud, mis sisaldavad kodeeritult faili viimase muutmise aega ja kuupäeva. Joonisel 14 näete nende elementide erinevate bittide tähendusi.



Joonis 14: Faili aja ja kuupäeva salvestamine

Olles täitnud andmestruktuuri *ffblk* esimese sellise faili andmetega, võib järgmise sarnase faili leidmiseks kasutada juba funktsiooni *findnext()*. Funktsiooni väärtus on 0, kui otsitav fail leidus, vastasel juhul mingi positiivne arv (veakood). Näiteks:

```
struct ffblk    info;
int    i;
...
if(0 == findfirst("C:\DOC\*.*", &info, 0))
  printf("Kataloogis C:\\DOC asuvad failid\n\n\n");
i = 0;
do {
  i++;
  printf("%s\t%d\t%c%c%c%c%c%c\n", info.ff_fname, info.ff_fsize,
    (info.ff_fattrib & FA_RDONLY) ? 'R' : ' ',
    (info.ff_fattrib & FA_HIDDEN) ? 'H' : ' ',
    (info.ff_fattrib & FA_SYSTEM) ? 'S' : ' ',
    (info.ff_fattrib & FA_LABEL) ? 'L' : ' ',
    (info.ff_fattrib & FA_ARCH) ? 'A' : ' ',
    (info.ff_fattrib & FA_DIREC) ? 'D' : ' ');
} while(0 == findnext(&info));
printf("\n\n\t\t%d faili", i);
```

Toodud programmilõik kuvab ekraanile kõik kataloogis C:\DOC asuvad failid, trükkides iga faili kohta ühe rea. Selline rida sisaldab faili nime, suuruse ja atribuudid. Umbes sama tööd teeb DOSi käsk DIR.

Aktiivse kettaajuri nime hankimiseks või vahetamiseks kasutage funktsioone *getdisk()* ja *setdisk()*.

```
int getdisk( void );
int setdisk(int drive);
```

Funktsioon *getdisk()* loovutab aktiivse kettaajuri numbri. Operatsioonisüsteem DOS nummerdab kettaid alates nullist: A = 0, B = 1, C = 2, D = 3 jne. Sama numbrit saab kasutada funktsioonis *setdisk()* aktiivse kettaajuri seadmiseks. Kui kettaajuri vahetamine õnnestus, on funktsiooni *setdisk()* tulemuseks 0, vastasel juhul veakood.

Operatsioonisüsteem DOS haldab eraldi aktiivset kataloogi iga ketta jaoks. Kui te näiteks vahetate kataloogiks C:\BORLANDC\BIN ja seejärel aktiveerite kettaajuri A ning siis sisestate käsu C:, siis vahetatakse aktiivseks kettaajuriks uuesti C, aga te ei asu enam C algkataloogis, vaid kataloogis C:\BORLANDC\BIN.

Suvalise kettaajuri (mis ka tegelikult eksisteerib) aktiivse kataloogi nime hankimiseks kasutage funktsiooni *getcurdir()*.

```
int getcurdir(int drive, char *directory);
```

Kui funktsioon *getcurdir()* leiab sellise kettaajuri, siis kopeerib ta tema aktiivse kataloogi nime viida *directory* poolt näidatud puhvrisse. Viimane peab kõigi kataloogide nimede mahutamiseks olema maksimaalselt MAXDIR baidi suurune.

Aktiivse kettaajuri aktiivse kataloogi nime hankimiseks kasutage funktsiooni *getcwd()*.

```
char *getcwd(char *buf, int buflen);
```

Funktsioon *getcwd()* kopeerib hetkelise kataloogi nime viida *buf* poolt näidatud puhvrisse ja hoidub seejuures puhvri piirist üle kirjutamast. Puhvri suurus on määratud parameetriga *buflen*.

Hetkelise kataloogi muutmiseks kasutage funktsiooni *chdir()*.

```
int chdir(const char* path);
```

Kataloogide loomiseks ja eemaldamiseks tuleks kasutada funktsioone *mkdir()* ja *rmdir()*.

```
int mkdir(const char* path);
int rmdir(const char* path);
```

Kataloogide nimede jaoks kehtivad operatsioonisüsteemi DOS poolt määratud reeglid (8+3 tähte või numbrit). Eemaldatav kataloog peab olema tühi ning ei tohi olla hetkeline kataloog või antud ketta juurkataloog.

Failide lugemise ja kirjutamisega tegelevas programmis on sageli vaja jaotada faili täielik nimi tema kettaajuri-, kataloogi- ja faili enda nimeks. Selleks sobib funktsioon *fnsplit()*. Kui te olete hankinud hetkelise kettaajuri ja kataloogi nimed ja soovite nüüd otsida teatud kindlale mustrile vastavaid faile, siis tuleks moodustada vastav muster funktsioonige *fnmerge()*. Loodud mustrit saab seejärel kasutada funktsioonides *findfirst()* ja *findnext()*.

```
void fnmerge(char* path, const char* drive, const char* dir,
            const char *name, const char* ext);
int fnsplit(const char* path, char* drive, char* dir, char *name,
            char* ext);
```

Funktsioon *fnmerge()* loob talle üleantud kettaajuri (*drive*), kataloogi (*dir*) ja faili (*name*) nimest ning faililaiendist (*ext*) faili täieliku nime ja salvestab selle viida *path* poolt näidatud puhvrisse. Funktsioon *fnsplit(),* vastupidi, jaotab talle üleantud täieliku nime eraldi osadeks ja salvestab nad vastavates puhvrites.
Soovitud nimega faili otsimiseks kasutage funktsiooni *searchpath().*

```
char * searchpath(const char* filename);
```

Funktsioon *searchpath()* otsib määratud nimega faili kõigepealt hetkelisest kataloogist ja kui teda seal ei leidu, siis keskkonnamuutuja PATH poolt määratud kataloogidest. Kui fail leitakse, siis loovutab funktsioon *searchpath()* viida tema täielikule nimele, vastasel juhul konstandi NULL.
Päisefail DIRECT.H sisaldab kataloogidega töötamiseks veel kolme funktsioni definitsiooni, mis aga kasutavad kettaajurite jaoks natuke teistsuguseid numbreid. Päisefail DIRENT.H defineerib kataloogide ja failide kasutamiseks määratud funktsioone operatsioonisüsteemi UNIX stiilis. Need funktsioonid on kasulikud ennekõike UNIXi programmide porteerimiseks DOSi.

Matemaatikafunktsioonid

Matemaatikafunktsioonide definitsioonid asuvad päisefailis MATH.H. Nende kasutamiseks tuleb programm siduda sobiva teegiga MATHx.LIB. Kui teie arvuti omab matemaatikaprotsessorit, siis lisage veel teek FP87.LIB, vastasel juhul teek EMU.LIB. Borlandi programmeerimiskeskkonda kasutades piisab vastava valiku tegemisest menüüs *Options | Complier | Code Generation*.
Borland C/C++ sisaldab ka funktsioone kompleksarvude kasutamiseks. Selleks tuleb lisada programmi teksti käsuga #include päisefail COMPLEX.H. Seal-toodud funktsioonid ei vaja mingit erilist lisateeki. Samas on defineeritud ka kaks andmestruktuuri kompleksarvude jaoks:

```
struct complex {
  double x, y;
};
struct _complexl {
  long double x, y;
};
```

Järgmises tabelis näete ülevaadet Borland C/C++ matemaatilistest funktsioonidest.

Funktsioon	Tähendus
Absoluutväärtus:	

int abs(int x);	Täisarvu absoluutväärtus
double abs(complex x);	Murdarvu absoluutväärtus
double cabs(struct complex z);	Kompleksarvu absoluutväärtus
long double cabsl(struct _complexl (z));	Suure täpsusega kompleksarvu absoluutväärtus
double fabs(double x);	Suurema täpsusega murdarvu absoluutväärtus
long double fabsl(long double @E(x));	Suurima täpsusega murdarvu absoluutväärtus
long int labs(long int x);	Suure täisarvu absoluutväärtus
Ümardamisfunktsioonid:	
double ceil(double x);	Leiab väikseima täisarvulise murdarvu, mis ei on veel suurem arvust x. Näiteks: ceil(2.5) on 3.0.
double floor(double x);	Leiab suurima täisarvulise murdarvu, mis on veel väiksem arvust x. Näiteks: floor(2.5) on 2.0.
long double ceill(long double (x));	Sama, mis ceil(), aga suurema täpsusega.
long double floorl(long double (x));	Sama, mis floor(), aga suurema täpsusega.
Trigonomeetrilised	**funktsioonid**:
double cos(double x);	cos(x), x on radiaanides
double sin(double x);	sin(x)
double tan(double x);	tan(x)
double tan2(double y, double x);	tan(x/y)
long double cosl(long double (x));	Sama mis cos(x)
long double sinl(long double (x));	Sama mis sin(x)
long double tanl(long double (x));	Sama mis tan(x)
long double tan2l(long double(y), long double (x));	Sama mis tan2(x)
double acos(double x);	arccos(x).
double asin(double x);	arcsin(x)
double atan(double x);	arctan(x)
double atan2(double y, double x);	arctan(x/y)

Tabel 26: Borland C/C++ matemaatikafunktsioonid

Funktsioon	Tähendus
Trigonomeetrilised	**funktsioonid**:
long double acosl(long double (x));	Sama mis arccos(x).
long double asinl(long double (x));	Sama mis arcsin(x)

long double atanl(long double (x));	Sama mis arctan(x)
long double atan2l(long double (y), long double (x));	Sama mis arctan2(x)
complex cos(complex z);	Kompleksarvu cos(x)
complex sin(complex z);	Kompleksarvu sin(x)
complex tan(complex x);	Kompleksarvu tan(x)
complex acos(complex z);	Kompleksarvu arccos(x)
complex asin(complex z);	Kompleksarvu arcsin(x)
complex atan(complex x);	Kompleksarvu arccos(x)
Logaritmfunktsioonid:	
double log(double x);	Murdarvu naturaallogaritm
double log10(double x);	Murdarvu kümnendlogaritm
long double logl(long double (x));	Sama mis log()
long double log10l(long double(x));	Sama mis log10()
complex log(complex x);	Kompleksarvu naturaallogaritm
complex log10(complex x);	Kompleksarvu kümnendlogaritm
Astendamine:	
double pow(double x, double y);	x astmes y
long double pow(long double (x), long double (y));	Sama, ainult suurema täpsusega
complex pow(complex x, complex y);	Kompleksarvu astendamine kompleksarvuga.
complex pow(complex x, double y);	Kompleksarvu astendamine murdarvuga
complex pow(double x, double y);	Kompleksarv arvust x astmes y
double exp(double x);	Arv e astmes x
long double exp(long double (x));	Sama mis exp()
complex exp(complex z);	e astmes kompleksarv x
double sqrt(double x);	Ruutjuur arvust x. Muude juurte jaoks kasutage astendamist juurija pöördväärtusega.
long double sqrtl(long double @E(x));	Sama mis sqrt(), ainult suurema täpsusega.
complex sqrt(complex x);	Ruutjuur kompleksarvust

Tabel 27: Borland C/C++ matemaatikafunktsioonid (järg)

Päisefail MATH.H sisaldab veel funktsioone murdarvu loomiseks mantissist ja astmest ning hulgaliselt kasulikke konstante, nagu näiteks: pi (M_PI), arv e (M_E) jms.

Tarkvara loomine

Tarkvara ei toodeta nagu teisi tooteid, vaid arendatakse. Suuremate tarkvarapakettide loomine on üsna keerukas. Umbes 25% suurtest tarkvaraprojektidest ei saa iialgi valmis. Enamus suuri tarkvaraprojekte kestavad aasta võrra kauem ja maksavad 100% rohkem, kui ette nähtud. See näitab, et tarkvaraprojektide läbiviimine on sageli halvasti organiseeritud. Mida suurem projekt, seda tähtsam on enne programmeerimist täpselt kindlaks määrata, kuidas ja mida üldse programmeerida soovitakse. Programmeerimise käigus avastatud loogilised struktuurivead võivad kogu eelneva töö asjatuks muuta.
Sageli soovivad ka kliendid lasta peale tarkvara üleandmist teha mitmesuguseid muudatusi. Nii kulutatakse sageli 60-80 % rahast hilisemate muudatuste tegemiseks (tarkvara hooldamine). Tarkvara hooldamist raskendab sageli ka asjaolu, et algsed plaanid ja vahel ka programmitekstid (*.C ja *.H failid) on kuhugi kadunud. Lisaks sellele võivad selle programmi loonud programmeerijad juba ammu töötada mingis muus firmas ja nii ei tea keegi enam täpselt, kuidas seda programmi muuta. Seda probleemi saab lahendada hästi struktureeritud programmeerimisstiili ja korraliku dokumentatsiooniga.
Tarkvara loomine jaotatakse järgmistesse etappidesse:

- Tarkvara planeerimine ja võimalikkuse kontroll
- Süsteemi ja tarkvara nõudmiste kontroll
- Andmestruktuuride, algoritmide ja programmistruktuuri loomine
- Programmi loomine
- Programmi töö kontroll ja optimeerimine
- Tarkvara hooldamine

Vastavalt projekti suurusest võib osa etappe kasutuks osutuda või vastupidi - omakorda etappideks jaguneda. Suuremate projektide puhul tuleks iga etapi lõpus tehtud otsused dokumenteerida. See lihtsustab programmi hilisemat muutmist ja kohendamist.
Tarkvara väärtuse hindamisel peetakse silmas järgmisi omadusi:

1. **Välised omadused** - Siia alla kuuluvad kasutaja poolt märgatavad omadused nagu näiteks töökiirus, kasutamise lihtsus jne.
2. **Sisemised omadused** - Neid omadusi määravad enamasti programmeerijad. Siia alla kuuluvad:
   - 2.1. **Korrektsus** - Tarkvara täidab täpselt seda ülesannet, mis tema loomisel määrati.
   - 2.2. **Robustsus** - Tarkvara suudab oma ülesannet täita ka ebanormaalsetes tingimustes (vähemalt osaliselt) .
   - 2.3. **Usaldatavus** - Hõlmab korrektuse ja robustsuse
   - 2.4. **Edasiarendatavus** - Tarkvara saab vajaduse korral lihtsalt uute nõudmiste jaoks kohendada.
   - 2.5. **Hilisem kasutatavus** - Tarkvara saab hilisemates projektides osaliselt või täielikult uuesti kasutada. Just see omadus aitab tunduvalt tõsta tarkvarafirma majanduslikku efektiivust.

2.6. **Standardsus** - Võimalus kasutada tarkvara koos muu tarkvaraga. Selleks on vaja kasutada standardseid failiformaate, protsesside vahelist andmevahetust, välisilmet ja palju muud.
2.7. **Efektiivsus** - Tarkvara oskus kasutada arvutiresursse (mälu, protsessoriaeg jms.) efektiivselt .
2.8. **Porteeritavus** - Võimalus kohendada tarkvara ilma erilise ümberprogrammeerimiseta muu arvutiarhitektuuri või operatsioonisüsteemi jaoks .
2.9. **Kontrollitavus** - Võimalus tarkvara tööd lihtsalt kontrollida.
2.10. **Kaitse** - Tarkvara omadus avaldada osa andmeid vaid kindlatele kasutajatele .

Ilmselt on osa toodud nõudmisi üksteisega osaliselt vastuolus. Näiteks porteeritavus näeb ette, et ei tohiks kasutada mingeid selle arvuti omapärasid, mis ei esine muude arvutite puhul. See aga on vastuolus efektiivusega. Siin tuleb leida sobiv tasakaal kõigi oluliste omaduste vahel.
Lihtsaim tarkvara loomiseviis on luua võimalikult kiiresti mingi programm ja siis teda proovida. Seejärel asutakse vigu parandama ja programmi soovitud suunas muutma. Selline viis sobib aga ainult väga väikeste programmide jaoks.
Täielik struktureeritud looming: analüüs - algoritmide loomine - kodeerimine - kontroll - hooldamine on aga sageli natuke liiga aeganõudev ja kohmakas. See- pärast kasutatakse sageli mingit vahepealset vormi. Üks selline viis on näiteks kiire näite meetod (*rapid prototyping*). Siin kasutatakse seda, et sageli ei tea klient veel tarkvara tellimise järguski päris täpselt, mis ta tegelikult saada tahab. Nii tehakse kliendi soovidele vastavalt kiiresti näiteprogramm, mis osaliselt juba täidab nõutud ülesanded ja omab sellist välimust (menüüd jms) nagu valmis programm seda omama saab, kuid ei ole veel optimeeritud jne. Selle näiteprogrammi alusel otsustab klient, mida (veel) vaja on. Vajaduse korral luuakse enne täielikult optimeeritud lõpliku programmi loomist veel paar näiteprogrammi. Näiteprogrammi järgi on ka lihtsam otsustada, kui palju aega ja raha on vaja antud projekti läbiviimiseks. Enne tegeliku programmi loomisega alustamist luuakse kliendiga leping, mis punktikaupa täpselt määrab kõik programmile esitatud nõudmised. Hilisemate lahkarvamuste lahendamiseks (programmi üleandmisel) kontrollitakse ülesande tätmist nimetatud lepingupunktide järgi.

Nõudmiste analüüs

Nõudmiste analüüs määrab, mida loodav tarkvarapakett tegelikult tegema peab, millises keskkonnas (operatsioonisüsteem, arvuti ehitus jne.) ta töötama hakkab ja milliseid piiranguid tuleb silmas pidada. Analüüsi tulemustest luuakse vastav dokument. Vastavalt vajadustele võib tulemuseks olla:

- lihtne dokument - määrab tarkvarapaketile esitatavad nõudmised sõnaliselt.

- näiteprogramm - loodud näiteprogramm demonstreerib esitatud nõudmisi. Nõudmised, mida veel programmeeritud ei ole, on määratud sõnaliselt .
- formaalne dokument - analüüsi tulemused on kirjeldatud mingis formaalses keeles.

Analüüsi tulemusi dokumenteerides peetakse silmas järgmisi põhimõtteid:

1. Analüüs peab määrama vaid, mida tuleb teha ja mitte kuidas seda teha.
2. Kui keskkonnatingimused tarkvara töö käigus muutuvad (näiteks tööstus-protsesside automatiseerimisel), siis ei saa tarkvara käitumist kirjeldada sisend- ja väljundandmete vaheliste matemaatiliste seoste kirjeldamisega, vaid on vaja kasutada protsessorienteeritud mudelit.
3. Analüüs peab kirjeldama kogu süsteemi neid osi, mis tarkvara tööd mõjutavad.
4. Analüüs peab kirjeldama tarkvara koostööd operatsioonisüsteemiga ja andmevahetust.
5. Loodud dokument peab olema arusaadav ka mitteprogrammeerijale.
6. Loodud dokument peab olema küllaldaselt formaalne ja täpne, et tema järgi saaks kontrollida, kas suvalise proovi tulemused vastavad soovitule.
7. Loodud dokument peab lubama osa nõudmisi ka lahtiseks jätta ja vajaduse korral uusi nõudmisi lisada.

Analüüsi tulemuste kirjeldamiseks on loodud mitmeid standardeid (IEEE ja ANSI standard Nr: 830). ANSI standardi järgi peaks tarkvaraanalüüs sisaldama järgmisi punkte:

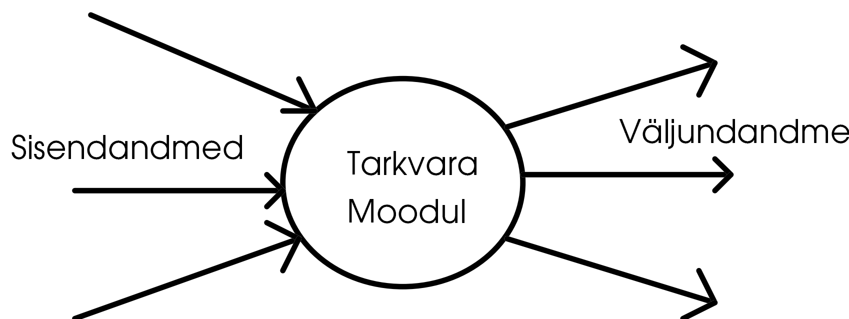
1. Sissejuhatus
    1.1. Tarkvara töökeskkond
    1.2. Üldised struktuuripõhimõtted
    1.3. Tarkvara tööd kitsendavad omadused
2. Andmete kirjeldus
    2.1. Andmevoogude kirjeldus
    2.2. Andmete kirjeldus
    2.3. Andmetestruktuuri kirjeldus
    2.4. Operatsioonisüsteemi ülesanded
3. Tarkvara töö kirjeldus
    3.1. Transformatsioonide loetelu
    3.2. Transformatsioonide kirjeldus
    3.2.1. Tehete järjekord
    3.2.2. Kitsendused, piirväärtused
    3.2.3. Tarkvarale esitatud nõudmised
    3.2.4. Struktuurile esitatud nõudmised
    3.2.5. Selgitavad diagrammid
4. Hinnang
    4.1. Tarkvara piirid
    4.2. Kontroll arvutuste üle
    4.3. Oodatav tarkvara käitumine
    4.4. Erijuhud
5. Kasutatud kirjanduse loetelu

6. Lisad

Loodud on ka ka mitmeid analüüsimeetodeid. Enamus neist kirjeldab tarkvara käitumist andmete muutmise ja struktuuri kirjeldamise abil. Osa meetodite jaoks on loodud ka vastavad programmid, mis lubavad andmeid lihtsamini sisestada, uurivad tarkvarakomponentide vahelisi seoseid ja loovad kindla probleemi puhul (andmepangad, tabelarvutus jne.) koguni valmis koodi. Siin kirjeldame vaid üht meetodit: andmevoogude analüüs.

Andmevoogude analüüs

See meetod põhineb oletusel, et tarkvarapakett koosneb moodulitest, mis kõik mingil moel sisendandmeid mõjutavad ja neist uusi andmeid arvutavad, mida siis järgmised moodulid tarbivad jne.

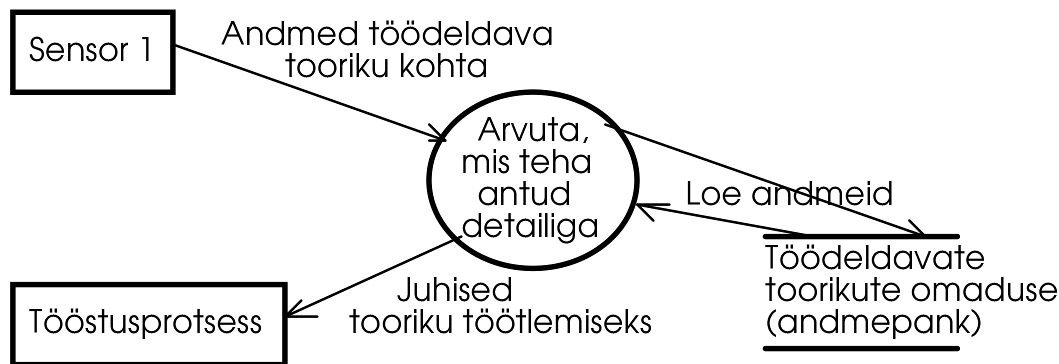


Joonis 15: Andmevoogudel baseeruv analüüsimeetod

Andmevoogudel baseeruva analüüsimeetodi puhul kujutatakse tarkvara struktuuri diagrammina, mille osad näitavad andmete liikumist ja muundamist tarkvaras. See diagramm sisaldab järgmisi elemente:

- Nooled - näitavad andmete liikumisteed. Vajaduse korral kirjutatakse noole kõrvale transporteeritavate andmete hetkeline sisu.
- Ringid - kujutavad tarkvara mooduleid, mis transformeerivad sisend-andmeid mingi seaduspärasuse järgi väljundandmeteks.
- Kaks rõhtjoont - kujutavad andmete säilitamist mingis kohas, näiteks failis.
- Ristkülikud - kujutavad andmete tekitajat või tarbijat. Selleks võib olla näiteks isik, kes klaviatuurilt mingeid andmeid sisestab, ekraan, millele andmeid väljastatakse, tööstusprotsess jne.

Näiteks:



Joonis 16: Andmevoogudel baseeruva analüüsi näide

Sellise diagrammi loomiseks leidke kõigepealt kõik andmevood. Kohtades, kus andmeid muudetakse, sisestage ring (transformatsioon) ja märkige sellesse andmete töötluse viis. Seejärel kontrollige kõiki transformatsioone ja asendage need, mis oma keerukuse tõttu ei ole kirjeldatavad ühe funktsiooniga, uute transformatsioonide ja nende vaheliste andmevoogudega. Niimoodi jätkake soovitud täpsuse saavutamiseni.

Kui osa transformatsioone analüüsi käigus pidevalt muutuvad (uued nõudmised kliendilt või analüüsi tulemused), siis võib nad asendada ühe suurema transformatsiooniga ja seejärel kirjeldada selle transformatsiooni ehitust eraldi diagrammil.

Diagrammil kasutatud andmestruktuuride nimetusi tuleks täpsemalt seletada. Selleks lisatakse diagrammile veel andmestruktuuride määrang (*data dictionary*), milles kasutatakse järgmisi elemente:

- = Andmestruktuur koosneb järgnevatest elementidest
- + eraldab andmestruktuuri komponente
- m{X} või n{X}m - Looksulgudes määratud elemendid võivad andmestruktuuris esineda *m* korda (või *n* kuni *m* korda). Kui m puudub, siis on tegemist n kuni lõpmatu arvu kordustega. Puudub aga n, siis on tegmist 0 kuni m kordusega.
- ( ) Ümarsulgudes toodud elemendid võivad andmestruktuuris esineda või ka puududa.
- _ Allajoonitult kujutatakse elemente, mille väärtuste abil eristatakse üksteisest sama tüüpi andmestruktuure .
- * * Tärnide vahele lisatakse kommentaarid, mis täpsustavad mingit andmestruktuuri elemendi eriomadust.

Näiteks:

```
Andmed töödeldava tooriku kohta = koordinaat + tüüp
koordinaat = täisarv
tüüp = täisarv
Juhised töödeldava tooriku kohta = koordinaat + käsk + parameeter
käsk = täisarv
parameeter = murdarv
Töödeldava tooriku omadused = tüüp + koostis + värvus + otstarve
koostis = 1{ materjal }10
materjal = täisarv
```

```
värvus = sümbolite jada
otstarve = sümbolite jada
```

Andmestruktuuride määrangut täpsustatakse niikaua, kuni kõik elemendid on määratud elementaarsete andmetüüpidega (murdarv, täisarv, sümbol jne.).
Ka kõigi transformatsioonide ülesandeid tuleks lähemalt kirjeldada. Selline transformatsiooni ülesannete määrang peaks ühemõtteliselt määrama, mil viisil sisendandmetest arvutatakse väljundandmed. Ülesanne määratakse valemitega või sõnaliselt. Seejuures koosneb määrang tegusõnadest, mis seovad andmestruktuuride määrangus toodud andmete nimesid. Kasutatakse ka mitmesuguseid reserveeritud sõnu, mis määravad käskudevahelisi seoseid. Määrangu alguses (ja soovitavalt ka lõpus) tuleks märkida transformatsiooni nimi või number. Määrang ei tohiks sisestada mingeid programmeerimise detaile. Näiteks:

```
BEGIN Arvuta, mis teha antud detailiga
  uuri andmeid töödeldava tooriku kohta
  IF uus detail
    loe andmepangast detaili omadused
  Väljasta juhised tooriku töötlemiseks
END Arvuta mis teha antud detailiga
```

Tarkvarale esitatavate nõudmiste analüüsiks kasutatakse veel DSSD (Data Structured Systems Development), JSD (Jackson System Development) ja palju muidki meetodeid.

Programmi struktureerimine

Struktureerimine jaotab tarkvara kogu ülesande väiksemateks osaülesanneteks, mis parandab programmi töö kontrollitavust ja vastab küsimusele, kuidas antud ülesannet lahendada. Kui analüüsis käsitleti antud probleemiosasid ja nende vahelisi põhimõtteid ning seaduspärasusi, siis struktureerimise käigus käsitletakse vastavate programmiosade vahelisi seoseid jne.
Programmi struktureerimine jaotatakse kolme etappi:

1. **Kogu struktuur** - jaotab tarkvara mooduliteks, mis tegelevad kindlate osaülesannetega
2. **Andmete struktuur** - defineerib kasutatavad andmestruktuurid
3. **Protseduuride strutuur** - jaotab moodulid omakorda protseduurideks, mis tegelevad väiksemate osaülesannetega.

Struktureerimine peaks jaotama tarkvara kindlate osaülesannetega tegelevateks mooduliteks ja näitama nendevahelisi seoseid. Seda aga eraldi andmete ja funktsioonide jaoks. Vahel on kasulik määrata kõigepealt kindlaks üldine struktuur ja asuda seejärel seda täpsustama ja täiustama.
Loodud moodulid peavad võimalikult vähe üksteisest sõltuma. Kõige parem oleks, kui üks suurem moodul sisaldab kõiki vajalikke funktsioone mingi kindla ülesande täitmiseks ja mitte midagi peale nende. Näiteks trükkimine võiks olla loodud funktsiooni *print()* abil, mis kasutab vajalikeks osaülesanneteks vaid

oma mooduli funktsioone. Osa operatsioonisüsteeme ja keskkondi (näiteks Microsoft Windows) võimaldavad laadida selliseid mooduleid mällu alles siis, kui neid tegelikult vajatakse ja peale vastava töö lõpetamist uuesti mälust eemaldada. See aitab tunduvalt mälu kokku hoida.

Andmestruktuure võib luua suvalise suuruse ja struktuuriga, kuid tavaliselt kasutatakse peale põhitüüpide vaid lihtsamaid struktuure nagu näiteks massiivid, loetelud, puud ja kataloogid. Globaalseid muutujaid tuleks vältida. Andmeid tuleks muuta vaid selleks ettenähtud funktsioonides ja parameetrite üleandmise teel, mitte aga nii, et funktsioon muudab otse globaalseid muutujaid. Funktsioon peaks olema nagu kindla käitumisega must kast, mille sisu meid enam ei huvita. Ideaalsel juhul peaks see kehtima ka moodulite kohta. Kui küllaldaselt suur hulk funktsioone vajavd samu andmeid, siis tuleks need andmed koguda kindlat tüüpi andmestruktuuri ja anda igale funktsioonile parameetrina sellisele andmestruktuurile osutav viit.

Vastavalt ANSI standardile peaks tarkvara struktuuri dokumentatsioon sisaldama järgmisi punkte:

1. Sissejuhatus
   1.1. Tarkvara ülesanded
   1.2. Arvuti ja tarkvara töö juhtimine
   1.3. Tarkvara põhiülesanded
   1.4. Kasutatud andmepangad, ekspertsüsteemid jms.
   1.5. Põhilised struktuurikitsendused
2. Muud dokumendid
   2.1. Kasutatud tarkvara dokumentatsioon
   2.2. Operatsioonisüsteemi kirjeldus
   2.3. Arvuti ja tarkvara tootja täiendused
3. Struktuuri kirjeldus
   3.1. Andmete struktuur
   3.1.1. Andmevoogude ülevaade
   3.1.2. Andmestruktuuride ülevaade
   3.2. Programmi struktuur
   3.3. Programmiosade vaheline tööjaotus
4. Moodulid
   4.1. Iga mooduli jaoks tema ülesannete kirjeldus
   4.2. Andmevahetus teiste moodulitega
   4.3. Mooduli sisemine struktuur
   4.4. Kasutatud andmestruktuurid
   4.5. Kasutatud moodulid
5. Failide struktuur ja globaalsed andmed
   5.1. Programmi välised andmestruktuurid
   5.1.1. Loogiline struktuur
   5.1.2. Andmestruktuuri elementide kirjeldus
   5.1.3. Kasutamismeetodid
   5.2. Globaalsed andmed
   5.3. Failid ja nende ehitus

6. Nõudmiste ülevaade
7. Töö kontroll
   7.1. Kontrolli suunad
   7.2. Kontrollide omavaheline sidumine ja mõju
8. Lisad

Vastavalt tarkvaraprojekti suurusele võib osa punkte osutuda üleliigseks või vajada omakorda alamjaotusi. Siin käsitleme täpsemalt andmevoogudel põhinevat struktureerimismeetodit.

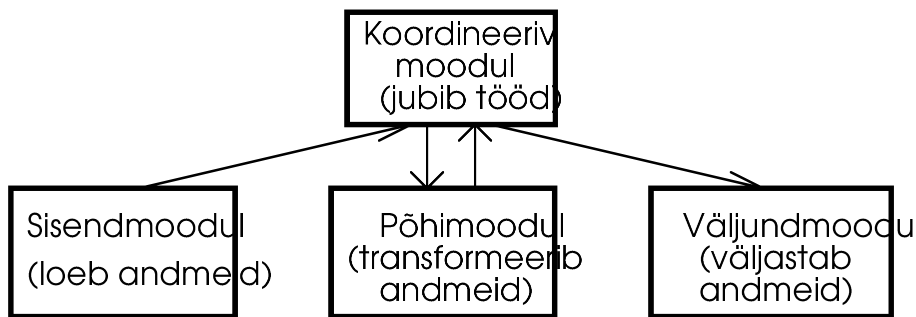
Andmevoogudel põhinev struktureerimine

Enamik struktureerimismeetodeid võimaldab analüüsi tulemustest sobiva struktuuri loomist kasutades samu põhimõtteid kui analüüsiski. Andmevoogudel põhinev struktureerimine sobib eriti hästi probleemide puhul, kus andmeid transformeeritakse järjestikku (mitte aga paralleelselt) ja kus ei esine keerukaid andmestruktuure. Sellised probleemid esinevad näiteks tööstusprotsesside juhtimisel, matemaatiliste valemite lahendamisel ja muude teaduslik - tehniliste probleemide puhul. Andmevoogudel põhinevat struktureerimismeetodit saab väikeste muudatustega kasutada ka probleemide puhul, mis nõuavad katkestuste kasutamist ja kindla ajaga vastuse arvutamist (reaalaeg).

Analüüsi tulemusena saadi andmevoogude diagramm. Sellest diagrammist täidetakse programmi struktuuri loomiseks järgmised ülesanded:

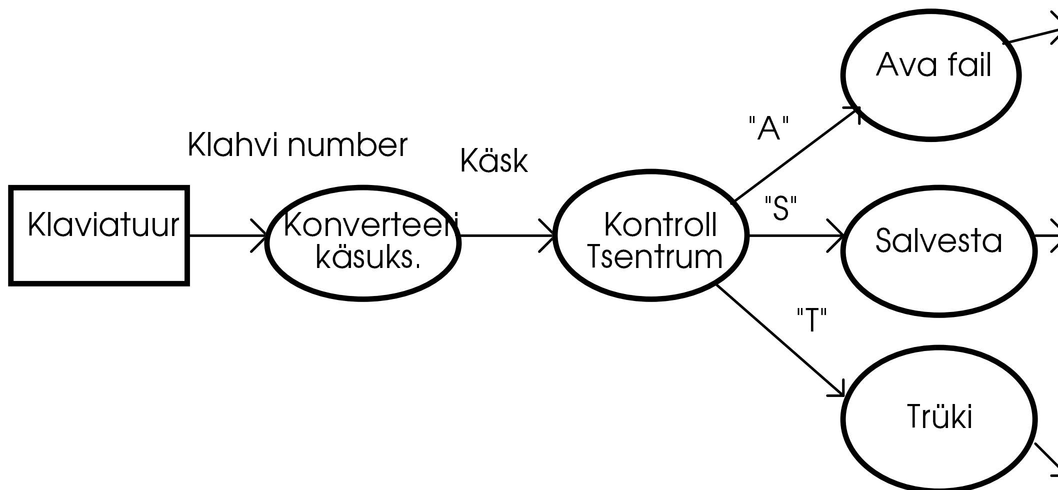
1. Määrata kindlaks andmevoo tüüp. Andmevoog võib olla tüübist transformatsioonivoog (*transformation stream*), mis muudab andmeid, või vastasmõjuvoog (*transaction stream*), mis mõjutab programmi kulgu. Enamik programme sisaldab mõlemat tüüpi voogusid.
2. Määrata kindlaks eri tüüpi voogude kokkupuutepunktid.
3. Andmevoogude diagrammist luuakse programmi struktuuri diagramm.
4. Vastasmõjuvoogusid täiustatakse kuni programmi soovitud kulu selgimiseni.
5. Saadud struktuuri täiustatakse operatsioonisüsteemi iseärasusi, programmi vajadusi ja kogemusi kasutades.

Sisendandmeid loetakse tavaliselt klaviatuurilt või failist. Igal juhul on nad mingis (välises) formaadis. Programmi sees kasutatakse andmete salvestamiseks sageli teistsugust formaati. Väljastatakse andmed omakorda uues formaadis. Seega on vajalik mingi keskne transformatsioon ühest andmeformaadist teise. Seda keskset transformatsiooni ja sisendit ning väljundit ühendab mingi transformatsioonivoog. Selline probleem struktureeritakse tavaliselt nelja mooduli abil.



Joonis 17: Transformatsioonivoo struktureerimine

Kui programm peab lugema mitmeid väliseid formaate (GIF, PCX, TIFF jne.), siis võiks sisendmooduli jaotada omakorda väiksemateks mooduliteks jne.
Vastasmõjuvoog lähtub sageli mingist transformatsioonivoost, mis näiteks loeb klaviatuurilt käske ja transformeerib neid sobivateks menüükäskudeks. Selles kohas asub niinimetatud vastasmõju tsentrum, millest jaotatakse programmile käske.



Joonis 18: Vastasmõjuvoo struktureerimine

Kontrolltsentrum aktiveerib vastvalt sisestatud käsule mingi transformatsioonivoo. Viimane täidab oma ülesande ja pöördub seejärel tagasi kontrolltsentrumi, kust siis täidetakse mingi muu käsk .

Kodeerimine

Kodeerimine peaks täpselt jälgima loodud programmi struktuuri. Parim kodeerimisstiil on lihtne ja otsejooneline. Järgnevas loetletakse mõningaid põhimõtteid.
Andmeid tuleks funktsioonile üle anda parameetrite abil. Funktsioon peaks oma töö põhilise tulemuse loovutama funktsiooni väärtusena. Muud tulemused võib tagastada viitparameetrite abil. Globaalsete andmete kasutamist tuleks vältida.
Kui funktsiooni töö võib ka ebaõnnestuda (näiteks faili avamine, mida ei eksisteeri), siis peaks funktsiooni väärtus seda väljendama. Näiteks faili avav

funktsioon loovutab faili numbri, mis ei tohi olla null, kui fail on avatud. Null näitab siis, et faili ei suudetud avada. Kui funktsiooni väärtus võib olla ka null ja negatiivne ja vea tekkimine on võimalik, siis peaks funktsiooni väärtus näitama ainult seda, kas töö õnnetus või mitte (loogiline muutuja), tegelik väärtus tuleks aga loovutada viitparameetri abil. Funktsioon peaks olema nagu "must kast", mis kindlatele sisendandmetele seab vastavusse kindlad väljundandmed. Kui ta reserveerib mälu, siis peaks ta selle ka ise enne töö lõppu vabastama. Samal tasemel olevad funktsioonid (mis üksteist vastastikku ei kasuta) ei tohiks mingil kombel üksteise tööd mõjutada suuta.
Vältida tuleks keerukaid loogilisi tingimusi ja negatiivseid tingimusi. Näiteks:
```
if((i != 0) && !(i < 0))
```
asemel hoopiski:
```
if(i > 0)
```
Tsüklid tuleks luua nii lihtsad kui võimalik ja kõik, mida ei pea tsüklis arvutama, tuleks teha väljaspool tsüklit.
Soovitav oleks kasutada vaid standardseid funktsioone ja definitsioone.
Enne matemaatilise valemi kodeerimist tuleks mõelda:

- kas teda ei saa lihtsustada
- kas tema parameetrid peavad asuma mingites kindlates piirides
- kui suur võib olla tulemus (mis tüüpi: *int, long, double* jne.)

Mitmedimensionaalseid massiive ja keerukaid andmestruktuure tuleks vältida, olgugi et C - keel seda lubab. Samuti tuleks vältida keerukat viitade - aritmeetikat.
Kui murdarve ilmtingimata vaja ei ole, siis tuleks alati kasutada täisarve, kuna nendega arvutamine on palju kiirem.
Täisarvu saab võrrelda nulliga (*do* { ... } *while*(i > 0)), kuid murdarvuga on see kahtlane. Murdarvu ei saa kunagi otse mingi kindla väärtusega võrrelda, vaid kasutada tuleks mingit piirkonda. Näiteks:

```
double   f, eps = 0.001;
...
if( (f < 1.0 + eps)  && (f > 1.0 - eps)) { /* kui f on umbes 1.0 */
   ...
}
```
Igale funktsioonile peaks eelnema kommentaaride blokk. Selline blokk võiks sisaldada järgmisi andmeid:

- parameetrite nimed, tüübid ja mõte
- mooduli nimi, mille koosseisu see funktsioon kuulub
- loomise aeg, viimase muutmise kuupäev
- milliseid funktsioone kasutatakse ja millistes päisefailides nad on defineeritud.
- milliseid globaalseid muutujaid kasutatakse ja kus nad on defineeritud
- milliseid globaalseid muutujaid muudetakse
- millised funktsioonid kasutavad antud funktsiooni

- funktsiooni tulemuse tüüp ja mõte

Samuti peaks iga faili päises olema kommentaaride blokk, mis sisaldab järgmisi andmeid mooduli kohta:

- faili nimi ja projekti nimi, mille koosseisu ta kuulub
- programmeerija(te) nimi
- loomise ja viimase muutmise kuupäev
- kasutatud translaatori nimi
- millisesse projekti ossa see moodul kuulub (põhiprogramm, failide lugemine jne.)
- defineeritud funktsioonid
- defineeritud muutujad

Ka üldine programmi tekst peaks sisaldama mõningaid kommentaare ja olema arusaadavalt kujundatud. Kommentaarid ei peaks iga üksiku lause tööd kirjeldama, kuna liigne kommentaaride hulk teeb programmiteksti, vastupidi, raskelt loetavaks. Kommentaarid peaksid lisama vaid täiendavaid andmeid, mida programmi tekstist kohe välja ei loe. Pikemate funktsioonide puhul, mis sisaldavad hulganiselt *for*, *while* ja *do-while* tsükleid, oleks mõttekas iga tsükli lõpuklambri taha kirjutada kommentaar, millise tsükli see klamber lõpetab. Näiteks:

```
for(i = 0; i < n ; i++) {
  ...
  <palju, palju lauseid>
  ...
}   /* for - i on väiksem n -st */
```

See on väga kasulik siis, kui selle tsükli algus asub mitu ekraanitäit eespool. Samuti on soovitav funktsiooni lõppu märkida kommentaariga funktsiooni nimi.

Blokkide ja tsüklite kasutamise puhul kasutatakse tsükli lausete jaoks taandrida (tavaliselt kaks tühikut), mis hõlbustab funktsiooni struktuuri jälgimist. Borland C/C++ editor lihtsustab teil sellise struktuuri loomist. Kui te tsükli alguses nihutate rida mõne positsiooni võrra paremale, siis alustab Broland C/C++ editor järgmise rea sama suure taandreaga. Bloki lõppu jõudes vajutage klahvile <Backspace>, mis nihutab kursori täpselt eelmise taandrea kohale (kaks positsiooni vasakule). Olemas on ka spetsiaalsed programmid, mis suvalise *.C faili sobivalt formateerivad ja uude faili salvestavad (*pretty-print*). Näiteks editor Emacs (mida enamsti kasutatakse UNIX arvutitel) oskab C - keele teksti juba sisestades väga ilusasti automaatselt formateerida. Paljud programmeerijatele loodud editorid võimaldavad editori käitumist ka mitmesuguste makrode abil muuta. Nende abil saab lihtsustada ennekõike keerukate kommentaariblokkide (funktsiooni ja mooduli alguses) sisestamist. Kui osa lausest ei mahu enam samale reale, siis poolitage ta tehtemärgi või sulu kohalt ja nihutage ülejääv osa tunduvalt paremale, et te ei segaks seda ära mingi bloki algusega.

Peale muutujate defineerimist (bloki alguses) oleks soovitav jätta üks tühi rida enne lausete algust. Samuti oleks soovitav sisestada paar tühje rida iga

funktsiooni definitsiooni vahele ja märkida mooduli põhiliste osade, nagu näiteks globaalsed muutujad, päisefailid jms., algus ja lõpp sobivate kommentaaridega.
Kasutage funktsioonide ja muutujate jaoks loogilisi nimesid, mis mingil määral viitaksid ka muutuja või funktsiooni mõttele ja tüübile. Näiteks võiks täisarvulise muutuja nimi alata väikse n tähega (*number*) ja murdarvuline muutuja f (*float*) tähega.

Vigade otsimine

Tegijale juhtub mõndagi ütleb vanasõna ja nii see ongi. Programmeerimisel tehakse ikka mingeid vigu. Osa neist leiab translaator kohe üles. Siia hulka kuuluvad liiga suur või väike hulk parameetreid, vale funktsiooni või muutuja nimi, puuduvad sulud jne. Osa vigu on aga loogilise iseloomuga. Nad ei takista programmi transleerimist ja ei väljasta isegi hoiatust. Näiteks:

```
int i, j;
...
for(i = 0; j < 5; i++) {  /* see tsükkel ei lõppe kunagi */
   ...
}
```

Sel juhul tuleks kasutada silurit (*debugger)*. Borland C/C++ sisaldab kahte silurit. Üks neist on osa Borland C/C++ töökeskkonnast (IDE). Enne siluri kasutamist tuleks programm transleerida koos siluri jaoks vajalike andmetega. Selleks tuleb valida menüüst ***Options*** / ***Compiler*** / ***Code Generation*** käsk: ***Debug info into OBJs***. Tavaliselt on see käsk uues projektis juba valitud.
Nüüd võite menüüst ***Run*** valitud käskudega programmi startida, teda vajalikus kohas peatada, muutujate väärtusi uurida jne.
Menüü ***Run*** käsk ***Run*** (<Ctrl>+<F9>) stardib programmi ja kui mingeid katkestuspunkte ei ole määratud, siis täidetakse programm kuni lõpuni (või esimese tõsise veani). Kui programm vajab argumente, siis valige kõigepealt menüü ***Run*** käsk ***Arguments...***, sisestage vajalikud argumendid ja startige seejärel programm uuesti.
Menüü ***Run*** käsk ***Go to cursor*** täidab programmi kuni reani, millel asub kursor ja peatub siis. Sama käsu täidab ka klahv <F4>.
Järgmise programmirea täitmiseks valige menüü ***Run*** käsk ***Step over*** (või vajutage klahvile <F7>) või ***Trace into*** (või vajutage klahvile <F8>). Need käsud erinevad oma mõjult vaid siis, kui järgmisel real mingi funktsioon välja kutsutakse. Sel juhul täidab käsk ***Step over*** väljakutsutava funktsiooni otsekohe kuni lõpuni ja peatub sellele järgneval programmireal. Käsk ***Trace into*** seevastu peatub väljakutsutava funktsiooni esimesel real ja võimaldab seega jälgida ka tolle funktsiooni käitumist.
Programmi töö peatamiseks valige menüü ***Run*** käsk ***Program reset*** või vajutage klahvidele <Ctrl>+<F2>.

Programmi peatamiseks mingis kindlas kohas viige kursor sellele reale ja valige menüü ***Debug*** käsk ***T***_oggle breakpoint_ või kasutage klahvikombinatsiooni <Ctrl>+<F8>. Nüüd peatub programm alati sellel real. Katkestuspunkti eemaldamiseks viige kursor antud reale ja valige uuesti menüü ***Debug*** käsk ***T***_oggle breakpoint._ Menüü ***Debug*** käsk ***B***_reakpoints..._ avab dialoogi, milles saab samuti seada ja eemaldada katkestuspunkte.
Menüü ***Debug*** käsk ***I***_nspect..._ toob ekraanile tillukese dialoogi, milles võite sisestada teid huvitava muutuja nime või mingi valemi. Nupule Ok vajutades näidatakse soovitud valemi väärtust. Sama menüü käsk ***E***_valuate/Modify..._ toob ekraanile dialoogi, milles võite samuti sisestada muutujate nimesid ja valemeid ning ka nende hetkelisi väärtusi muuta.
Menüü ***W***_indow_ käsk ***Watch*** toob ekraanile tillukese akna, millesse võib sisestada muutujate nimesid ja näha nende väärtusi. Muutuja või valemi sisestamiseks sellesse aknasse vajutage nupule <Ins>. Avanenud dialoogis sisestage soovitud muutuja nimi ja vajutage nupule _Ok_. Muutuja või valemi eemaldamiseks sellest aknast viige kursor sellele reale ja vajutage klahvile <Del>. Valemi või muutuja nime muutmiseks viige kursor antud reale ja vajutage nupule <Enter>. Avanenud dialoogis muutke nimi ja vajutage nupule _Ok_. Samu ülesandeid saab täita ka menüü ***Debug*** alammenüü ***W***_atches_ käskudega.
Abiprogramm _Turbo Debugger_ võimaldab täita samu ülesandeid ja lisaks sellele:

- Jälgida programmi tööd ka tagurpidi, s.o. liikuda eelmisele reale jne.
- Animeerida programmi tööd - s.o. peatuda igal real mingi kindel arv sekundeid ja jätkata seejärel programmi tööd.
- Uurida ka keerukate andmestruktuuride sisu.
- Jälgida andmesegmendi, suvalise mälu piirkonna, pinu ja protsessori registrite sisu.
- Uurida eriliste programmide nagu näiteks ohjurprogrammide tööd.

Silur _Turbo Debugger_ on eraldi programm. Programmeerimiskeskkonna silurit on lihtsam kasutada ja harilikult sellest ka jätkub. _Trubo Debugger_ on tunduvalt võimasam ja suudab rohkem, kuid temas ei saa kohe peale vea leidmist ka programmi teksti antud kohas muuta. Ta võimaldab vaid programmi tööd jälgida. _Turbo Debuggeri_ võite samuti startida käsuga "-" _Turbo_ ***D***_ebugger_ programmeerimiskeskkonna vasakpoolseimast menüüst. Kui te alati kasutate programmi töö uurimiseks silurit _Turbo Debugger_, siis võite menüü ***O***_ptions_ käsuga ***D***_ebugger_ avatud dialoogis määrata siluri tüübiks _Standalone_. Sel juhul stardivad menüü ***Debug*** ja ***Run*** vastavad käsud kohe _Turbo Debuggeri_ ja laadivad temasse antud programmi.
Turbo Debugger sarnaneb oma välimuselt üsnagi Programmeerimiskeskkonna siluriga. Ka siin on menüü ***Run***, mille käskudega saab programmi startida, peatada ja reahaaval täita. Vastavad käsud omavad samu nimesid, ainult neile vastavad klahvikombinatsioonid on natuke muutunud. Lisaks juba tuntud käskudele on ilmunud veel käsud ***U***_ntil return_, mis täidab hetkelise funktsiooni kuni lõpuni ja peatub seejärel teda väljakutsunud funktsiooni järgmisel real.

Käsk ***B**ack trace* täidab programmi ühe rea võrra tagurpidi. Käsk ***E**xecute to...* võimaldab sisestada mingi valemi või tingimuse, mida kasutatakse katkestuspunkti seadmiseks. Programmi töö peatatakse, kui see tingimus on tõene.

Katkestuspunktide jaoks on ilmunud menüü ***B**reakpoints*. Selle menüü käskudega saab lisaks harilikele katkestuspunktidele määrata ka katkestuspunkte mingi mälubloki jaoks. Niipea, kui programm seda mälu kasutab, peatatakse tema töö (käsk *Changed memory global...*). Võimalik on määrata ka mingi tingimus, mispuhul programmi töö peatatakse (käsk *Expression true global...*) ja seada katkestuspunkte protsessori abil (käsk *Hardware breakpoint...*).

Muutujate ja valemite tulemusi jälgida ja neid aknasse *Watches* lisada saab menüü ***D**ata* käskudega.

Menüü ***V**iew* käskudega saab jälgida protsessori registrite, pinu ja andmesegmendi sisu, väljakutsutud funktsioonide järjekorda ja palju muudki.

Optimeerimine

Siluriga *Turbo Debugger* saate jälgida programmi tööd ja otsida programmist mitmesuguseid vigu. Kui te olete saanud oma programmi juba töökorda, siis võib see vahel veel olla ebapiisav. Mõnikord esitatakse programmile nõudmisi, et mingi teatud programmiosa tuleb täita maksimaalselt *n* sekundiga. Sel juhul tuleb uurida, kas programm selle ülesande täidab ja kui see nii ei ole, siis üritada kasutatud algoritme või ka programmeerimisstiili ja funktsioone nii muuta, et programmi kiiremini töötaks. Selleks otstarbeks ongi loodud silur *Turbo Profiler*.

Üldiselt on inimene alati arvuti juures see kõige aeglasem osa. Seega ei ole mõtet optimeerimisega ülepingutada ja asjatult head programmistuktuuri ohverdada või mingeid erilisi trikke kasutada. Optimeerimine on vajalik vaid siis, kui programm ilmselt reageerib tunduvalt aeglasemini kui kasutaja suudab temasse andmeid sisestada või kui programmile esitatakse kindlaid nõudmisi maksimaalse tööaja suhtes.

Turbo Profiler analüüsib teie programmi ja teatab, milliseid funktsioone kõige sagemini välja kutsuti, ka kui palju aega nende jaoks kasutati (protsentuaalselt ja arvuliselt). Nähes nüüd, millistes programmiosades kõige rohkem aega kasutati, võite seda programmiosa uurides jõuda järeldusele, kuidas sama tööd kiiremini teha.

Turbo Profileri kasutamiseks tuleb transleerida programm koos siluri andmetega, kuna *Turbo Profiler* kasutab samuti neid andmeid. Lisaks sellele oleks kasulik programmi struktuuri mõnevõrra muuta. Kui *Turbo Profiler* uurib ka kasutaja vastust nõudvaid funktsioone (*scanf()*, *getch()* jne.), siis sõltub tulemus iga kord kasutaja vastamise kiirusest. Seepärast tuleks üritada programmi niimoodi struktureerida, et neid osi saaks uuringutest välja jätta.

Turbo Profileri võite startida käsuga `tprof` käsurealt või programmeerimis-keskkonnast menüü "=" käsuga *Turbo Profiler*. Nüüd avaneb *Turbo Profileri* aken, mis on jagatud horisonaalselt kaheks. Ülemine näitab uuritava programmi teksti ja alumine sisaldab uuringute tulemusi (statistikat). Algselt märgistab *Turbo Profiler* kõik programmi read. Statistikat tehakse ainult märgistatud ridade kohta. Kui reast vasakul on sümbol '=>', siis on rida märgistatud ja tema kohta tehakse statistikat.
Uuringute alustamiseks startige programm käsuga ***Run*** (klahv <F9>). Peale programmi töö lõppu ilmuvad alumises aknas statistikaanalüüsi tulemused.

```
  -   File   View   Run   Statistics   Print   Options   Window   Help           READY
 ----------Module: INTEGRAL File: ...LES\CSAMPLES\INTEGRAL\INTEGRAL.C 108--1--------
          scanf("%f", &MaxPiir);
          printf("Täpsus [%f]: ", Epsilon);
          scanf("%f", &Epsilon);
          /* interpoleerime niikaua, kuni soovitud täpsus on saavutatud */
          do {
            Tulemus = Temp;
            Temp = Ristkuelik(MinPiir, MaxPiir, Num);
            Num = Num * 2;
          } while(Epsilon < fabs(Tulemus - Temp));
          /* trükime tulemuse ja anname kasutajale v§imaluse
             arvutusi uute väärtustega korrata */

 =[■]=Execution Profile==========================================2=[↑][↓]=
   Total time: 35.716 sec       Display: Time
   % of total: 99 %              Filter: All
         Runs: 1 of 1              Sort: Frequency

  #INTEGRAL#99        18.802 sec   52% |==========================================
  #INTEGRAL#45         7.9570 sec  22% |=================
  #INTEGRAL#67         3.5051 sec   9% |========
  #INTEGRAL#46         3.3149 sec   9% |=======

 F1-Help F2-Area F3-Mod F5-Zoom F6-Next F9-Run F10-Menu
```

Joonis 19: Turbo Profileri aken

Joonisel 19 näete *Turbo Profileri* akent näiteprogrammi INTEGRAL.EXE analüüsi tulemustega. Alumine aken (*Execution Profile*) on omakorda kaheks jaotatud. Ülemises osas on esitatud andmeid kogu programmi kohta. Seal on kogu programmi tööaeg (*Total time:*), alumises aknas näidatavate andmete hulk (*Display:*) ja nende sorteerimise viis (*Sort:*).
Alumine osa on jaotatud omakorda kaheks. Vasakpoolne osa näitab iga rea kohta kui palju sellel real tööaega kulutati. Nagu näete, kulutati enamus programmi INTEGRAL.EXE ajast (52%) real 99. Vajaduse korral võite lasta näidata ka iga rea kasutamise arvu. Mõnda rida täidetakse väga mitu korda (tsüklid), kuid tema täitmine ei võta palju aega. Alumise aknaosa parempoolne osa sisaldab lihtsustatud tulpdiagrammi, mis näitab iga rea jaoks kulutatud suhtelist ajahulka.
Mingi kindla rea markeerimiseks viige kursor sellele reale ja vajutage klahvile <F2> või vajutage hiire vasakule nupule selle rea kohal. Kui te vajutate hiire paremale nupule *Turbo Profileri* ülemises aknas, ilmub ekraanile selle akna kohale menüü, mis võimaldab markeerida või eemaldada markeeringut programmi tervetelt piirkondadelt. Selle menüü alammenüü *Add areas* käsk *All routines* markeerib kõik programmi funktsioonid tervikuna. Seega analüüsitakse funktsioone vaid tervikuna, mitte aga iga funktsiooni iga rida. Ka teegifunktsioone analüüsitakse. Käsk *Modules with source* markeerib vaid need

funktsioonid, mis on antud programmis kodeeritud (seega mitte teegifunktsioone). Käsk *Routines in module* markeerib vaid antud faili iga funktsiooni ja käsk *Every line in module* markeerib antud faili iga rea. Käsk *Lines in routine* markeerib selle funktsiooni iga rea, milles asub kursor. Käsk *Current routine* analüüsib antud funktsiooni kogu tööaega. Alammenüü *Remove areas* võimaldab piirkondadelt markeeringut eemaldada.

Hiire paremale nupule vajutamine alumises aknas toob ekraanile selle akna kohaliku menüü. Selles menüüs võite valida, kas esitada iga rea kohta vaid tema jaoks kulunud aeg (*Time*), tema täitmise arv (*Counts*) või mõlemad (*Both*). Võimalik on määrata ka esitatud andmete sorteerimisviisi.

Menüü *Statistics* käsuge *Profiling options* ... avatud dialoogis saab valida analüüsitüübiks aktiivse (*Active*) või passiivse (*Passive*). Esimesel juhul peatub *Turbo Profiler* igal real, mis on markeeritud ja mõõdab tema jaoks kulunud aega ja muud. Teisel juhul teeb *Turbo Profiler* vaid "pistelist kontrolli". Esimene moodus on muidugi hulga täpsem, kuid ka tunduvalt aeglasem. Pika programmi puhul oleks kasulikum määrata kõigepealt passiivse analüüsiga aeglasemad programmiosad umbes ja siis aktiivselt analüüsida vaid neid osi.

Programmi optimeerimise võimalused

Üks võimalus programmi tööd kiirendada on kasutada kiiremaid funktsioone. Funktsioon *puts()* väljastab sümbolijada ekraanile kiiremini kui funktsiooni *putch()* kasutamine eraldi iga tähe jaoks. Makrod on tavaliselt kiiremad, kui funktsioonid, kuida nad teevad programmi suuremaks.

Teine võimalus oleks kasutada paremaid algoritme. Siin nimetaksime mõningaid lihtsamaid optimeerimisvõimalusi.

- Konstantide arvutamine transleerimise ajal - valemi konstantsed osad tuleks juba transleerimise käigus välja arvutada. Näiteks:

  ```
  y = x + 2 * M_PI;
  ```
  asemel

  ```
  #define     2PI   2.0 * M_PI
  y = x + 2PI;
  ```

- Vähem tüüpidevahelisi konverteerimisi - sisestage konstandid juba õiges tüübis. Näiteks:

  ```
  y = 2 * M_PI
  ```
  siin tuleb 2 niikuinii murdarvuks muuta, niisiis:

  ```
  y = 2.0 * M_PI
  ```

- Kustutage programmitekstist muutujad, mida mingis valemis ei kasutata.
- Kui kaks valemit sisaldavad mingit sarnast osa, siis tuleks see enne välja arvutada ja muutujasse salvestada (ainult üks kord) ja siis kasutada selle muutuja väärtust mõlemas valemis. Näiteks:

```
y1 = x * sin(fi * M_PI / 180.0);
y2 = x * cos(fi * M_PI / 180.0);
```

asemel:

```
temp = fi * M_PI / 180.0;
y1 = x * sin(fi);
y2 = x * cos(fi);
```

- Kõravaldage tsüklitest avaldised, mida seal (mitu korda) ilmtingimata ei pea arvutama. Need avaldised tuleks täita enne tsüklit.
- Kui tsükli täitmiste arv on ettearvutatav või konstantne suurus, siis kasutage *while* või *do-while* asemel *for* tsüklit.
- Kui kaks tsüklit on ühepikkused ja täidavad ülesandeid, mis üksteise tööd ei mõjuta, siis looge neist üksainus tsükkel.
- Kõrvaldage tingimused, mis kunagi tõesed olla ei saa. Näiteks:

```
int f1, f2 =0;
while(f1 && f2)
      puts("mingi tekst!");
```

  Kuna f2 on null, siis ei saagi seda tsüklit kunagi täita.
- Sagelikasutatud muutujate jaoks kasutage protsessoriregistreid. Selleks kasutage muutuja nime ees võtmesõna **register**. Näiteks:

```
register int      i;
for(i = 0; i < 100; i++) {
...
}
```

- Rekursioon (funktsioon kutsub end ise välja) tuleks alati kui võimalik asendada lihsama tsükliga. See on küll keerukam valemi järgi, kuid töötab kiiremini.

Optimeerida tuleks iga kord vaid ühe põhimõtte kaupa ja alati salvestada eelmine versioon faili juhuks, kui midagi sassi läheb. Suuremate algoritmide puhul tuleks üritada ettearvutada nende keerukus n^2, n^3 jne. ja kasutada antud olukorras sobivaimat.