PROGRAMME KÕIGILE

TARTU
1989

TARTU ÜLIKOOL
Arvutuskeskus

PROGRAMMEERIMISKEEL C

Programme kõigile

Koostanud T. Kelder
Ü. Kaasik

TARTU 1989

Kinnitatud matemaatikateaduskonna nõukogus
28. septembril 1989. a.

Käesolev väljaanne kujutab endast programmeerimiskeele C teatmikku, mis annab täieliku ülevaate nii keelest kui ka programmeerimisstiilist selles keeles (lähemal ajal ilmub veel teatmiku teine osa, kus tutvustatakse keele C standardset teeki). Ülevaade ei lähtu mingist konkreetsest realisatsioonist ja operatsioonisüsteemist, samuti on püütud hoiduda ka arvutist sõltuvatest mõistetest. Eraldi tuuakse välja keele C muutused, mis tulenevad ANSI poolt soovitatud standardist. Kogu käsitluses eeldatakse, et lugeja on juba tuttav programmeerimiskeelte põhimõistetega ja vähemalt ühe Algoli-laadse keelega nagu Pascal, Ada või PL/I. Peab ühtlasi aga arvestama, et tegemist ei ole keele C õpikuga. Keelt kirjeldatakse järjekorras "alt üles", mis ei sobi just kõige esimeseks tutvumiseks keelega. Samuti ei ole püütud saavutada käsitluse ranget järjepidevust: mõiste kirjeldamisel võidakse kasutada (näidetes) ka veel kirjeldamata mõisteid. Mõningad käsitletavad küsimused pakuvad huvi mitte niivõrd programmeerijale kui keele C realiseerijale, kuigi kirjeldus pole piisavalt range ja üksikasjalik, et tema järgi oleks võimalik keelt C realiseerida.

1. S I S S E J U H A T U S

1.1. Keele C koht

Programmeerimiskeel C kuulub Algoli-laadsete keelte hulka. Ta sarnaneb enam keeltega PL/I, Pascal või Ada ja vähem keeltega BASIC, FORTRAN või Lisp.

Keele C töötas välja 1972. aasta paiku Dennis Ritchie firmast Bell Laboratories. Keele C eelkäijateks võib lugeda keeli BCPL (Martin Ricards, 1967) ja B (Ken Thompson, 1970). Kuigi keelt C peetakse universaalset tüüpi programmeerimiskeeleks, on teda enam kasutatud siiski süsteemprogrammeerimisalastes ülesannetes. Näiteks populaarne operatsioonisüsteem UNIX on kirjutatud C-s ja algselt oli ka keel C realiseeritud ainult operatsioonisüsteemis UNIX. Praeguseks on aga keel C realiseeritud juba enamiku levinud arvutitüüpide ja operatsioonisüsteemide jaoks ning on üsna populaarne eriti nende programmide koostamiseks, mida tahetakse üle kanda eri tüüpi arvutitele.

Keele C populaarsusel on rida põhjusi. Esiteks annab ta suhteliselt täieliku vahendite komplekti laia klassi ülesannete lahendamiseks. Keeles C on küllalt arenenud andmetüübid nagu viidad, struktuurid ja sõned, rikkalik operatsioonide valik ning kaasaegsed juhtimisstruktuurid. Lisaks keelele endale on vastavas funktsiooniteegis vahendid sisendiks/väljundiks, mälujaotuseks, tekstitöötluseks ja muuks sarnaseks.

Teiseks tuleb arvestada, et C on suhteliselt väike keel. Keeles C kirjutatud programmid on efektiivsed, kuna C andmetüübid ja operatsioonid on küllalt vahetult arvutil realiseeritavad. Semantiline erinevus keele C operatsiooni ja arvutioperatsiooni vahel on suhteliselt väike.

Kolmandaks on keeles C kirjutatud programmid hästi üle kantavad erinevatele arvutisüsteemidele ning keeles on võimalik välja eraldada arvutitüübist sõltuvad osad.

Tuleb arvestada ka seda, et seoses operatsioonisüsteemi UNIX laia levikuga leidub maailmas juba palju keeles C kirjutatud programme ja seda keelt kasutavaid programmeerijaid.

Kahjuks toovad mõningad C populaarsust tagavad omadused kaasa ka probleeme. Näiteks C programmeerimissüsteemi väiksus tuleneb muuhulgas range tüübikontrolli puudumisest, see aga võimaldab vigu programmeerimisel. Sageli tuleb "heade" programmide kirjutamiseks järgida stilistikareegleid, mis ei ole otseselt ette kirjutatud kompilaatori poolt. Teise probleemina võib nimetada asjaolu, et programmide ülekantavuse tagamiseks on mõned C operatsioonid ning tüübid sisemiselt kirjeldamata, mis lubab erinevusi konkreetsetes rakendustes, s.t. teatud keelekonstruktsioonide korral jääb interpretatsioon realiseerija otsustada.

Kokkuvõttes on C keel, milles vilunud programmeerija saab kirjutada kiiresti, efektiivselt ja elegantselt, sageli nimetatakse keelt C professionaalse programmeerimise keeleks.

1.2. Mis defineerib C

Siintoodav käsitlus järgib eeskätt teatmikku: S. P. Harbison, G. L. Steele "C. A Reference Manual" Second edition. Prentice-Hall, 1987. Lisaks on kasutatud järgmisi materjale:

B. W. Kernighan, D. M. Ritchie "The C Programming Language". Prentice-Hall, 1978. Leidub ka tõlge vene keelde: Б. Керниган, Д. Ритчи, Ф. Фьюэр "Язык программирования Си. Задачи по языку Си", Москва, 1985.

B. C. Hunter "Understanding C". Berkeley, 1984.

Microsoft C Compiler. Language Reference. Microsoft Corporation 1987.

Проект Государственного Стандарта Союза СССР. "Язык программирования Си для персональных ЭВМ". Первая редакция, рассылаемая на отзыв. 1987.

Иванов А.Г. "Язык программирования Си. Предвадительное описание". Прикладная информатика, вып. 1, 1985 с. 68 - 111.

ANSI, X3, Information Processing Systems, X3Y11. C-Language Informaytion Bulletin, July 1, 1985.

Программное обеспечение персональных профессиональных ЭВМ Единой Системы. Система Программирования СиМ86. Версия 2. Руководство Пользователя. 1988.

Kuigi C õpikuid ja kirjeldusi leidub suurel hulgal, puudub praktiliselt keele praeguse seisu korrektne kirjeldus. Erinevad realisatsioonid erinevad omavahel peaaegu alati mõningate detailide poolest kas rohkem või vähem.

Traditsiooniliselt valitakse esituse aluseks B. Kernighani ja D. Ritchie autorikirjeldus, kuigi vaevalt et leidub täpselt sellele kirjeldusele vastavat kompilaatorit. Enamasti tuuakse ikka sisse mõningaid laiendusi ja/või kitsendusi ning kuigi nende osas on erinevates realisatsioonides püütud saavutada ühtlust, pole see alati kaugeltki täielik. Autorikirjeldusele vastavat keelt nimetame edaspidi originaal-C.

Teiseks oluliseks keele C defineerimise allikaks on konkreetsed kompilaatorid: tuleb kirjutada programm keeles C, kompileerida see ja vaadata, mis juhtub. Taoline lähenemisviis on enamasti mõistlik, sest paljude C realisatsioonide aluseks on nn. Portable C Compiler (PCC), mis on ka ise realiseeritud eri tüüpi arvutite jaoks.

1982. aastal moodustati ANSI komitee keele C standardiseerimiseks. Selle komitee poolt koostatud C standardi projekt võttis kokku küllalt palju keele laiendusi. Käesolevas väljaandes nimetame seda keelt ANSI C.

1.3. C-programmeerimisest

Programm keeles C ehk C-programm koosneb ühest või mitmest lähtefailist. Iga lähtefail sisaldab teatava osa programmist, tavaliselt mingi arvu välisfunktsioone ja välisandmete kirjeldusi. Lähtefailid kasutavad sageli nn. päisfaile, mis annavad teistes failides määratud välisfunktsioonide ja -andmete kirjeldused. Ühes lähtefailidest peab olema antud funktsioon nimega main, millest algab programmi täitmine.

Iga lähtefail kompileeritakse C-kompilaatori abil eraldi. Sel etapil kontrollitakse programmi keelelist korrektsust ja kui vigu ei leita, siis loob kompilaator vastava objektfaili ehk objektmooduli. Leitud keelevigadest annab kompilaator kasutajale teada veateadete abil ning ühtlasi võib objektmoodul jääda ka loomata.

Kui kõik lähtefailid on kompileeritud ning vastavad objektmoodulid loodud, siis antakse nad üle komplekteerimisprogrammile nimega linker, mis kogub kokku kõik objektmoodulid ja lisab neile teegimoodulid, mis sisalduvad C-programmeerimissüsteemi standardteegis. Sel etapil selgub ka, kas kõik programmi kogumiseks vajalikud komponendid on olemas. Tavaliselt ei spetsifitseeri keel linkerit: kasutatakse vastava operatsioonisüsteemi standardprogramme. Komplekteerimise tulemus on programm, mida saab välja kutsuda täitmiseks.

Kuigi igas programmeerimissüsteemis tuleb keele C korral need sammud läbida, on programmeerija sellekohased konkreetsed tegevused arvutist ja operatsioonisüsteemist sõltuvad. Seepärast me edasises ei pööragi tähelepanu programmi kompileerimise ja komplekteerimise konkreetsetele küsimustele.

Vaatleme järgmist näidet. Sisaldagu fail proga.c teksti

```
#include <stdio.h>
hello()
{
printf("Tervist!\n");
}
printarv()
{
int i;
for(i = 1; i < 10; ++i)
printf("%d\n",i);
}
```

failis progb.c aga olgu tekst

```
main()
{
hello();
printarv();
}
```

Need failid moodustavad lihtsa C-programmi. Fail proga.c sisaldab kaks funktsiooni: neist hello trükib (teegifunktsiooni printf abil) lause Tervist! ja printarv trükib arvud 1 kuni 9. Võtmesõnaga for algav rida on tsükkel, mille toimel järgmist rida täidetakse muutuja i väärtustel 1,..., 9. Fail

progb.c sisaldab funktsiooni main, millest algab C-programmi täitmine (ta pöördub funktsioonide hello ja printarv poole).

Selle programmi kompileerimiseks, komplekteerimiseks, nimega program varustamiseks ja täitmiseks tuleb Microsoft C korral operatsioonisüsteemis MS-DOS sisestada käsuread:

```
cc proga;
cc progb;
link progb+proga,program;
program
```

1.4. Keele kirjeldamise süntaks

Keele kirjeldamisel süntaksivalemites esitame **poolpaksu kirjaga** terminaalsed sümbolid - sõnad ja märgid, mis tuleb keeles anda täpselt nii, nagu valemites kirjutatud. Mitteterminaalsed sümbolid ehk defineeritavad mõisted esitame tavalise kirjaga (võivad koosneda tähtedest, numbritest ja sidekriipsust) nii, et nad selgitavad oma semantikat, näiteks

avaldis argumentide-loetelu deklaraator2

Süntaksivalemi kuju on järgmine: defineeritav mõiste antakse omaette real koos järgneva kooloniga, järgnevatel ridadel antakse võimalikud alternatiivsed tähendused, näiteks

```
sümbol:
    trükisümbol
    võtmesümbol
```

Lühiduse mõttes võib võimalikud tähendused loetleda ka ühel real, selle tähisena järgneb koolonile sõna variandid:

```
number: variandid
    0 1 2 3 4 5 6 7 8 9
```

Mittekohustusliku (s.o. võimalik, et puuduva) osa määrame ära allakriipsutusega, mida kasutatakse nii terminaalsete kui ka mitteterminaalsete sümbolite korral, näiteks

```
funktsiooni-deklaraator:
    deklaraator ( parameetrid )
algväärtus:
    avaldis
    { algväärtuste-loetelu , }
```

2. K E E L E C L E K S I K A

2.1. Tähestik

Lähtefail keeles C koosneb keele tähestikku kuuluvaist sümboleist. Tähestik jaguneb põhitähestikuks ja lisatähestikuks. Keele C põhitähestiku hulka kuuluvad:

1) viiskümmend kaks ladina tähestiku suur- ja väiketähte

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

2) kümme kümnendnumbrit

0 1 2 3 4 5 6 7 8 9

3) tühikusümbol ␣

4) kakskümmend üheksa erisümbolit

! # % ^ & * ( ) - _ + = ~ [ ] \ ¦ ; : ' " { } , . < > / ?

Lisatähestiku sümbolite hulka kuuluvad:

1) viis formaatimissümbolit, vastavad ASCII sümbolitele tagasilüke BS (backspace), horisontaaltabulaator HT (horizontal tab), vertikaaltabulaator VT (vertical tab), lehevahetus FF (form feed) ja kelgutagastus CR (carriage return);

2) erisümbolid, mis ei kuulu põhitähestikku

$ @ ‘

Formaatimissümboleid tõlgendatakse tühikutena ning neil ei ole programmis muud tähendust. Lisatähestikku kuuluvaid erisümboleid võib kasutada vaid kommentaarides ning sümbolkonstantides ja sõnedes. Käesolevas kirjelduses eeldame, et keele C lisatähestikku kuuluvad ka eesti tähestiku tähed:

õ ä ö ü Õ Ä Ö Ü

mis võimaldab neid kasutada näidetes kommentaaride ja sõnede koosseisus näidete parema loetavuse tagamiseks.

Nagu siit võib näha, on keele C tähestik märksa laialdasem enamiku programmeerimiskeelte tähestikest. See võib tuua kaasa mõningaid raskusi programmide vormistamisel juhul, kui arvutiseadmed ei võimalda sisestada kõiki sümboleid. ANSI C toob ära kokkulepped mõningate sümbolite asendamiseks sümbolikolmikutega (trigraafidega), mis võimaldab programme kirjutada väiksema tähestiku abil.

Arvutitähestik, s.t. sümbolite hulk, mida aktsepteeritakse C-programmide täitmisel, ei pea tingimata kokku langema keele C tähestikuga. Sümbolid arvutitähestikus on esindatud kas nende ekvivalentidega keele tähestikus või siis spetsiaalsete võtmesümbolitega, mis algavad langkriipsuga \.

Lisaks keele tähestiku sümboleile peab arvutitähestik sisaldama veel vähemalt:

1) sümboli null, mis kodeeritakse kui väärtus 0;

2) sümboli realüke (newline) kui rea lõputunnuse.

Sümbolit null kasutatakse sõnede lõputunnusena, realüke aga jagab sisendil või väljundil sümbolitevoo ridadeks. Ka arvutitähestik sisaldab formaatimissümbolid, mille kujutamiseks programmis tuuakse sisse spetsiaalsed võtmesümbolid.

Sümbolit tühik ning formaatimissümboleid (realüke ja tabulatsioonisümbolid) kokku kutsutakse tühisümboleiks. Neid sümboleid programmis üldiselt ignoreeritakse peale juhu, kus nad eraldavad lekseeme, mille vahel pole muid eraldajaid või kui nad esinevad sõnedes ja/või sümbolkonstantides. Tühisümboleid võib kasutada ka programmi loetavuse parandamiseks.

Realükkesümbol newline märgib programmirea lõppu, kuid mõningates realisatsioonides tähistavad rea lõppu veel sümbolid kelgutagastus, reavahetus ja vertikaaltabulaator. Lähteteksti jaotus ridadeks on oluline keele C preprotsessoris, mis töötab ridadega. Rea lõputunnusele järgnev sümbol loetakse järgmise rea esimeseks sümboliks. Kui see sümbol omakorda on rea lõputunnus, siis loetakse rida tühjaks.

Eraldi kokkuleppena tähendab rea lõputunnusele vahetult eelnev sümbol \ seda, et tuleb ignoreerida nii sümbol \ kui ka järgnev rea lõpusümbol. See võimaldab lähteteksti ridu kokku võtta üheks reaks, näiteks pikkade sõnede korral.

Keel C ei kitsenda maksimaalset reapikkust, kuigi paljud realisatsioonid seavad piiri, tavaliselt 100 kuni 500 sümbolit. Mõistlik on kasutada mitte pikemaid kui 80-sümbolilisi ridu, võimaldamaks programmi ekraanil normaalselt kujutada.

Igal sümbolil arvutitähestikust on oma konkreetne kood, s.t. mingi sisemine arvuline väärtus, mis võib eri arvutites olla erinev. Kood on oluline, sest selle järgi teisendatakse

sümbolid täisarvudeks. Keeles C nõutakse, et põhitähestiku kõigil sümboleil oleks erinevad positiivsed koodid.

Üks C-programmeerimisel levinud viga seisneb mingi konkreetse kooditabeli eeldamises. Näiteks arvatakse, et avaldis

```
'Z' - 'A' + 1
```

arvutab tähtede arvu alfabeedis. Avaldis määrab tähtede arvu küll siis, kui arvutis kasutatavaks koodiks on ASCII (avaldise väärtus on sel juhul 26). Kui aga kasutatavaks koodiks on näiteks EBCDIC, siis saame avaldise väärtuseks 41, kuna tähestik selles koodis ei ole kodeeritud järjestikku.

2.2. Kommentaarid

Kommentaarid keeles C algavad sümbolipaariga /* , lõpevad sümbolipaariga */ ning võivad sisaldada keele tähestiku suvalisi sümboleid. Kogu kommentaari tõlgendatakse kui tühisümbolit. Kommentaare kasutatakse programmi loetavuse parandamiseks, näiteks:

```
void Squares ()   /* funktsioonil pole argumente */
{
int i;
/* trükitakse täisarvud 0, ..., 10 ja nende ruudud */
for (i=0; i <= 10; ++i)
printf("%d ruudus on %d\n",i,i*i);
}
```

Kommentaare käsitleb tühisümbolitena ka preprotsessor. Seega rea lõpp kommentaaris ei lõpeta preprotsessirida. Näiteks kolm järgmist direktiivi #define on samaväärsed:

```
#define  ten  (2 * 5)
#define  ten  /* ten:
                  one greater than nine
              */ (2 * 5)
#define  ten  (2/*/*/*/*/*/5)
```

Enamik realisatsioone (ka originaal-C ja ANSI C) ei luba üksteisesse sisestatud kommentaare. Ja kui ka lubatakse (nagu Microsoft C ja Lattice C korral), ei ole seda soovitatav kasutada (pealegi pole kaalukaid argumente nende kasuks).

2.3. Lekseemid

Sümbolid, millest moodustub C-keelne programm kogutakse lekseemidesse. Eristatakse viit lekseemiklassi: operaatorid, eraldajad, nimed, võtmesõnad ja konstandid.

Lekseemi eraldamisel tekstist leiab kompilaator alati võimalikest pikima lekseemi, näiteks sõna external tõlgendatakse ühe lekseemina, aga mitte kahena - võtmesõnana extern ja nimena al. Lekseemide eraldajateks võivad olla tühisümbolid ja kommentaarid. Segaduste vältimiseks peab tühisümbol eraldama nime, võtmesõna, täis- või reaalarvkonstandi järgnevast nimest, võtmesõnast, täis- või reaalarvkonstandist.

2.4. Operaatorid ja eraldajad

Lihtoperaatorid ehk ühesümbolilised tehtemärgid on:

```
!  %  ^  &  *  -  +  =  ~  |  .  <  >  /  ?
```

Liitoperaatorid on aga sümbolipaarid või kolmikud:

```
->  ++  --  <<  >>  <=  >=  ==  !=  &&  ||
+=  -=  *=  /=  %=  <<=  >>=  &=  ^=  |=
```

Muude eraldajatena peale operaatorite esinevad sümbolid:

```
(  )  [  ]  {  }  ,  ;  :
```

Täpsemalt, originaal-C ning sealt tulenevalt ka paljud realisatsioonid loevad liitomistamistehte märgid

```
+=  -=  *=  /=  %=  <<=  >>=  &=  ^=  |=
```

kaheks eraldi lekseemiks - operatsiooni ja omistamise sümbol. Seega võib kirjutada ka näiteks

```
summa + = osa;
```

Siiski tuleb niisuguste operaatorite ühe lekseemina kirjutamist lugeda paremaks stiiliks ning mõned uuemad kirjeldused nagu näiteks ANSI C seda ka nõuavad.

2.5. Nimed

Nimi ehk identifikaator koosneb tähtedest ja numbritest, kusjuures esimene sümbol on täht (allkriips _ loetakse tähtede hulka). Nimedena ei tohi kasutada võtmesõnu.

```
nimi:
     allkriips
     täht
     nimi järgmine-sümbol
järgmine-sümbol:
     allkriips
     täht
     number
täht: variandid
     A B C D E F G H I J K L M
     N O P Q R S T U V W X Y Z
     a b c d e f g h i j k l m
     n o p q r s t u v w x y z
allkriips:
     _
number: variandid
     0 1 2 3 4 5 6 7 8 9
```

Nimedes loetakse suur- ja väiketähed erinevateks, seega nimed abc ja aBc on erinevad.

Keeles C puudub kitsendus nimede pikkuse kohta. Küll aga eristatakse nimesid tavaliselt teatud arvu esimeste sümbolite põhjal. Originaal-C ja ka paljude varasemate realisatsioonide korral oli selleks arvuks 8. ANSI C arvestab kuni 31 esimest sümbolit. Eri tingimused seatakse tavaliselt välisnimedele, kuna nad sõltuvad ka linkerist. Näiteks võib välisnimi olla piiratud 6 arvestatava sümboliga ning suur- ja väiketähed võivad olla samaväärsed. Eelistada tuleks pikemate nimede kasutamist, tagamaks programmi parem loetavus.

Üksikud realisatsioonid lubavad mitmekeelsete programmide võimaldamiseks nimedes ka mõningaid teisi sümboleid (näiteks $). Taolised programmid aga on konkreetsest realisatsioonist sõltuvad ning selliseid nimesid tuleks vältida.

Kuigi mitte otseselt keele koostisosana on praktikas välja kujunenud teatud hulk reegleid nimede kasutamise kohta. Neid reegleid järgivad paljud keeles C programmeerijad, kuna see hõlbustab programmide mõistmist ja nende ülekandmist ühelt arvutilt teisele.

Halvaks stiiliks tuleb pidada niisuguste nimede kasutamist, mis erinevad ainult neisse kuuluvate tähtede suuruse poolest, nagu näiteks count ja Count. Üldiselt on märgata tendentsi, eriti UNIX-keskkonnas, tähistada preprotsessimakrode nimed suurtähtedega ja kasutada kõigis ülejäänud nimedes vaid väiketähti. Tüüpiline näide:

```
#define TABLESIZE 100
...
int i, squares[TABLESIZE];
```

Väga pikad nimed tehakse loetavamaks kas siis tähesuuruste vaheldumise või allkriipsu kasutamisega. Tunduvalt raskem on välja lugeda nime averylongidentifier kui näiteks nime AVeryLongIdentifier või a_very_long_identifier.

Nagu juba öeldud, on välisnimed sageli enam kitsendatud kui ülejäänud nimed. Sageli lisab kompilaator välisnimedele vaikimisi mingi sümboli (näiteks paljudes realisatsioonides pannakse iga välisnime ette sümbol _). Samuti võivad välisnimedes suur- ja väiketähed kokku langeda.

Paljudes realisatsioonides on allkriipsuga algavad nimed reserveeritud süsteemi sisemiseks kasutamiseks ja ehkki programmeerijal ei ole keelatud neid kasutada, tuleks niisuguseid nimesid siiski vältida.

Kui C-kompilaator võimaldab pikki nimesid, programmeerimissüsteem aga ainult lühikesi välisnimesid, siis võib programmeerija preprotsessori abil need lühikesed (ja ebaülevaatlikud) nimed maskeerida. Näide:

```
#define error_handler eh73
extern void error_handler();
int *p;
...
if(!p) error_handler("Viga! Nullviida esinemine.\n");
```

2.6. Võtmesõnad

Omaette lekseemiklassi moodustavad keeles C reserveeritud nimed ehk võtmesõnad, mida ei või kasutada kui tavalisi identifikaatoreid. Need nimed on järgmised:

auto	**do**	**for**	**return**	**typedef**
break	**double**	**goto**	**short**	**union**
case	**else**	**if**	**sizeof**	**unsigned**
char	**enum**	**int**	**static**	**void**
continue	**extern**	**long**	**struct**	**while**
default	**float**	**register**	**switch**	

Originaal-C ei kasutanud siinloetletuist võtmesõnu enum ja void. Varem võtmesõnadena kasutatud asm, fortran ning entry on aga praeguseks enamikus realisatsioonides kõrvale jäetud. ANSI C lisab võtmesõnad const, signed ja volatile. Mõningates realisatsioonides on täiendavate võtmesõnadena kasutusel veel near ning far.

Võtmesõnad on kasutatavad makrodirektiivides nimedena, kuid seda tuleb pidada halvaks stiiliks. Sellise stiili ühe mõistliku kasutamise näitena võib tuua makro

```
#define void int
```

mis on sobiv kasutamiseks siis, kui konkreetne C-kompilaator ei tunne andmetüüpi void.

2.7. Konstandid

Eristatakse nelja erinevat tüüpi konstante: täisarvkonstandid, reaalarvkonstandid, sümbolkonstandid ja sõned:

konstant:
- täisarvkonstant
- reaalarvkonstant
- sümbolkonstant
- sõne

Selliseid lekseeme kutsutakse mõnikord ka literaalideks, et eristada neid konstantsete väärtustega objektidest (s.t. objektidest, mille väärtusi ei saa muuta). Näitena viimaste kohta võib tuua loenditüübi konstandid ehk loendikonstandid, mis aga leksikaliselt kuuluvad nimede klassi. Me kasutame edaspidi terminit konstant mõlemas tähenduses.

Iga konstant on iseloomustatud tema tüübi ja väärtusega. Erinevat tüüpi konstantide esitusviisid võtame vaatlusele järgnevates alalõikudes.

Täisarvkonstante võib esitada kas kümnend-, kaheksand- või kuueteistkümnendkujul.

1) Kümnendkonstant on mittetühi kümnendnumbrite järjend, kus esimene number ei ole 0.

2) Kaheksandkonstant koosneb sümbolist 0, millele võib järgneda kaheksandnumbrite (0 kuni 7) järjend. Originaal-C lubab kaheksandkonstandis kasutada ka numbreid 8 ja 9, mis väljendavad kaheksandväärtusi 10 ning 11, kuid uuemates realisatsioonides see enam lubatud ei ole ning seda tuleb pidada ebasoovitatavaks.

3) Kuueteistkümnendkonstant koosneb sümbolist 0, millele järgneb sümbol x või X ja sellele omakorda kuueteistkümnendnumbrite järjend. Kuueteistkümnendnumber on üks numbreist 0 kuni 9 ja tähtedest a kuni f (A kuni F), mis väljendavad kümnendväärtusi 10 kuni 15.

Üksikut numbrit 0 võib pidada nii kümnend- kui kaheksandkonstandiks. Täisarvkonstandile vahetult järgnev sümbol l (või L) osutab, et konstandi väärtus on tüüpi long.

täisarvkonstant:
    kümnendkonstant tüübimarker
    kaheksandkonstant tüübimarker
    kuueteistkümnendkonstant tüübimarker
kümnendkonstant:
    mittenull
    kümnendkonstant number
kaheksandkonstant:
    0
    kaheksandkonstant 8-number
kuueteistkümnendkonstant:
    baasimarker
    kuueteistkümnendkonstant 16-number
mittenull: variandid
    1 2 3 4 5 6 7 8 9
8-number: variandid
    0 1 2 3 4 5 6 7
16-number: variandid
    0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

baasimarker: variandid

0x 0X

tüübimarker: variandid

l L

Täisarvkonstandi väärtus on alati positiivne (kui ei ole ületäitumist). Kui konstandile eelneb märk miinus, siis pole tegemist konstandi vaid avaldisega. Väärtus on üldreeglina tüüpi int. Väärtuse tüübiks loetakse long int siis kui:

1) kümnendkonstandi väärtus on suurem kui maksimaalne võimalik positiivne täisarvuline väärtus tüüpi int;

2) kaheksand- või kuueteistkümnendkonstandi väärtus on suurem kui maksimaalne võimalik väärtus tüüpi unsigned int;

3) konstant lõpeb sümboliga l või L.

Kui konstandi väärtus ületab maksimaalse võimaliku väärtuse tüübist long int (või unsigned long int mõningates realisatsioonides), siis on tulemus määramata. Enamik kompilaatoreid ei hoiata kasutajat sellise situatsiooni korral, vaid omistavad konstandile mingi suvalise väärtuse. Programmeerija võib suured konstandid määrata makrodena, et muuta nende väärtusi ülieminekul ühest arvutist teise, näiteks:

```
#define MAXPOSINT 0077777
#define MAXNEGINT 0100000
#define MAXPOSLONG 0x37777777
#define MAXNEGLONG 0x80000000
```

Illustreerime täisarvkonstandi omadusi keeles C järgneva näitega. Eeldame, et meil on tegemist realisatsiooniga, mis kasutab tüübi int jaoks 16-bitist täiendesitust ning tüübi long jaoks 32-bitist täiendesitust (selline esitusviis on väga levinud). Vaatleme mõningate konstantide tegelikku matemaatilist väärtust, tüüpi ja C-väärtust (sulgudes on antud defineerimata, kuid enamikes realisatsioonides tõepärased väärtused). Huvipakkuvaim selle tabeli juures on asjaolu, et konstandid matemaatilise väärtusega 2^{15}-1 ja 2^{16} vahel osutuvad positiivseteks siis, kui nad on antud kümnendkonstandina, kuid negatiivseteks kaheksand- ning kuueteistkümnendkonstantidena. ANSI C laiendab mõnevõrra täisarvkonstandi süntaksit ja kasutab veidi erinevaid tüübireegleid.

C konstant	Õige väärtus	C tüüp	C väärtus
0	0	int	0
32767	$2^{15}-1$	int	32767
077777	$2^{15}-1$	int	32767
32768	2^{15}	long	32768
0100000	2^{15}	(int)	(-32768)
65535	$2^{16}-1$	long	65535
0xFFFF	$2^{16}-1$	(int)	(-1)
65536	2^{16}	long	65536
0x10000	2^{16}	long	65536
2147483647	$2^{31}-1$	long	2147483647
0x7FFFFFFF	$2^{31}-1$	long	2147483647
2147483648	2^{31}	(long)	(-2147483648)
0x80000000	2^{31}	long	-2147483648
4294967295	$2^{32}-1$	(long)	(-1)
0xFFFFFFFF	$2^{32}-1$	long	-1
4294967296	2^{32}	(long)	(0)
0x100000000	2^{32}	(long)	(0)

Reaalarvkonstant on alati kümnendalusel ja võib koosneda täisosast, kümnendpunktist, murdosast ning eksponendist. Kas täisosa või murdosa võib puududa, kuid mitte mõlemad; kas kümnendpunkt või eksponent võib puududa, kuid mitte mõlemad:

reaalarvkonstant:
    numbrid eksponent
    punktnumbrid eksponent
eksponent:
    e märk numbrid
    E märk numbrid
märk: variandid
    + -
punktnumbrid:
    numbrid .
    numbrid . numbrid
    . numbrid
numbrid:
    number
    numbrid number

Mõned näited reaalarvkonstantide kohta:

```
0.      3e1      3.14159    .0      1.0E-3
1e-3    .00034   2E+9       125.    12.053E+2
```

Reaalarvkonstant on alati positiivne (kui pole ületäitumist). Konstandile eelnev märk - muudab selle avaldiseks.

Reaalarvkonstandi tüüp on alati double ja tema väärtus sõltub tüübi double sisemisest esitusest. Kui konstant on selleks liiga suur või liiga väike, et olla esitatud tüübis double, siis on tema väärtus defineerimata. Enamik kompilaatoreid programmeerijat sellisest situatsioonist ei hoiata.

ANSI C laiendab reaalarvkonstandi süntaksit ja võimaldab ka konstante tüüpi float.

Sümbolkonstant on sümbol apostroofide vahel, näiteks 'A' (eriline võtmesümbolite mehhanism võimaldab esitada konstantidena ka neid sümboleid, mis pole vahetult sisestatavad):

```
sümbolkonstant:
        ' sümbol '
sümbol:
        trükisümbol
        võtmesümbol
```

Trükisümbol on keele C tähestiku sümbol, millel on vaste arvutitähestikus. Erandi moodustavad apostroof, langkriips ja reaalüke, mida saame esitada üksnes võtmesümbolitena.

Sümbolkonstandi tüüp on int ja väärtus vastava sümboli kood. Toome mõned näited sümbolkonstantide väärtuste kohta (sulgudes) eeldades, et kasutatavaks koodiks on ASCII:

```
'a' (97)     'A' (65)      '%'   (37)     ' '  (32)
'?' (63)     '8' (56)      '\r'  (13)     '\0'  (0)
'\23' (19)   '"' (34)    '\377' (255)     '\\' (92)
```

Hea programmeerimisstiili kohaselt loetakse trükisümbolite hulka ainult need, millel tegelikult on trükipilt ning lisaks veel tühik. Formaatimissümbolid on sobivam esitada võtmesümbolitena (mõned kompilaatorid võivad seda ka nõuda).

Kui sümbolkonstandis kasutatakse selliseid sümboleid või võtmesümboleid, milledel pole vastet arvutitähestikus, siis on tulemus realisatsioonist sõltuv. ANSI C näiteks keelab sellised sümbolkonstandid.

Mõnedes realisatsioonides on lubatud ka mitmesümbolilised sümbolkonstandid nagu 'ABC'. Niisugusel juhul sõltub lubatud sümbolite arv tavaliselt sellest, mitu baiti võtab enda alla tüüp int. Peab arvestama, et selline konstant ei määra mitte sümbolite järjendit, vaid täisarvulise väärtuse, mis saadakse vastavatest baitidest mingis (realisatsioonist sõltuvas) järjestuses kujutatuna. Nii näiteks 'ab' võib olla kas täisarv 0x6162 või täisarv 0x6261. Nende sümbolkonstantide kasutamine on oluliselt masinsõltuv ja seepärast tuleks neist hoiduda.

Sõne on jutumärkides asuv sümbolite (võimalik et tühi) järjend. Sümboliteks võivad olla nii trükisümbolid kui ka võtmesümbolid analoogiliselt sümbolkonstantidega:

```
sõne:
     " sümbolid "
sümbolid:
     sümbol
     sümbolid sümbol
```

Sõnede korral kuuluvad trükisümbolite hulka kõik keele C tähestiku sümbolid, millel on ekvivalent arvutitähestikus, (erandiks on jutumärgid, langkriips ja realüke, mida saab esitada võtmesümbolitena). Sõne peab paiknema lähteprogrammi ühel real, kuid kui rea viimaseks sümboliks on langkriips \, siis langkriipsu ja rea lõpusümbolit ignoreeritakse, mis võimaldab anda sõne ka mitmel real (mõned realisatsioonid ignoreerivad sel juhul ka jätkurea algtühisümboleid). Näiteid sõnede kohta:

```
""
"See on sõne"
"\""
"Kommentaar algab sümbolitega '/*'\n"
"See on näide sõnest\
mitmel real."
```

Igale n sümbolist koosnevale sõnele vastab programmi täitmise ajal n+1 baidi pikkune staatilise mälu blokk, kus esimesed n baiti on initsialiseeritud sõne sümbolitega ja viimaseks on sümbol '\0' - sõne lõputunnus.

Sõne tüübiks on massiiv sümboleist, pikkusega n+1 sümbolit, seega operatsioon sizeof("abcd") annab väärtuse 5. Kui sõne esineb avaldises, siis teisendatakse selle tüüp tüübiks viit sümbolile ja väärtuseks on viit sõne esimesele sümbolile. See võimaldab viitu algväärtustada sõnedega, näiteks:

```
char *p = "abcdef";
```

Enamiku realisatsioonide korral eraldatakse igale sõnele eraldi mälu - ka siis kui sõnede kirjapilt on identne. Tuleb pidada ebasoovitatavaks sõne sisu muutmist programmis, kuigi äsjases näites on see võimalik (kasvõi *(p+1)= 'g';). ANSI C paigutab sõned mälupiirkonda, kus neid ei saagi muuta. Sõne sisu muutmine võib aga tekkida ka kaudselt, kui sõne antakse argumendina ette mingile teegifunktsioonile, mis seda muudab. Sellistest ebameeldivustest hoidumiseks võib sõnede asemel kasutada sõnedega algväärtustatud massiive:

```
char  p1[] = "abcdef"; /* p1[i] on muudetav */
char *p2="abcdef"; /* p2[i] ei pruugi olla muudetav */
```

Võtmesümboleid võib kasutada sümbolkonstantides ja sõnedes. Nad võimaldavad kujutada sümboleid, millede otsene esitamine programmis on võimatu või ebamugav. Võtmesümboleid on kaht liiki - tähelised ja numbrilised. Tähelised võtmesümbolid nimetavad mõningaid formaatimissümboleid, numbrilised kujutavad soovitud sümboli koodi:

```
võtmesümbol:
        \ võti
võti:
        võtmemärk
        koodi-tähis
võtmemärk: variandid
        b f n r t v \ ' "
koodi-tähis:
        8-number
        8-number 8-number
        8-number 8-number 8-number
```

Mõned realisatsioonid võimaldavad ka veel muid võtmemärke (näiteks \e ja \a). Mõnikord on võimalik koodi tähist anda ka kujul \xnn, kus nn on kaks kuueteistkümnendnumbrit.

Kui langkriipsule järgneb mingi muu sümbol, on tulemus defineerimata, kuid enamik kompilaatoreid jätab sel juhul langkriipsu lihtsalt vahele.

Võtmemärgid kujutavad mõningaid formaatimis- või erisümboleid masinsõltumatul moel. Nende tähendus on järgmine:

b - tagasilüke (backspace)
f - lehevahetus (form feed)
n - realüke (newline)
r - kelgutagastus (carriage return)
t - horisontaaltabulaaator (horizontal tab)
v - vertikaaltabulaator (vertical tab)
\ - langkriips (et maskeerida teda võtmesümbolist)
' - apostroof
" - jutumärgid

Kasutamise näitena anname väikese programmi, mis loendab sisendil ridu. Funktsioon getchar loeb sisendi järjekordse sümboli, kuni sisendi lõpuni, millest signaliseerib funktsiooni väärtus -1 (tavaliselt defineeritakse kui EOF).

```
/* loendatakse read sisendil */
#define EOF -1
main()
{
int next_char;       /* järjekordne sisendi sümbol */
int num_lines;       /* ridade arv */
while((next_char  =  getchar()) != EOF)
        /* iga sümboli jaoks kuni sisendi lõpuni */
if(next_char == '\n')  /* kas on rea lõputunnus */
      ++num_lines;     /* suurendame ridade arvu */
/* trükime saadud ridade arvu välja */
printf ("loeti %d rida\n",num_lines);
}
```

Suvalise sümboli võime esitada andes tema koodi ühe kuni kolme kaheksandnumbri abil, näiteks \0, \141 jne. Selline esitusviis on muidugi masinsõltuv. Lisaks tuleb arvestada, et koodiosa lõppeb kas esimese mitte-kaheksandnumbriga või peale kolmandat numbrit. Näiteks \0111 on kaks sümbolit \011 ja 1, \080 aga kolm sümbolit \0, 8 ja 0.

3. P R E P R O T S E S S O R

3.1. Preprotsessoridirektiivid

Keele C preprotsessor annab mõningad lihtsad makrovahendid lähteteksti töötlemiseks enne kompileerimist. Mõningates realisatsioonides on preprotsessor eraldi programm, mis loeb lähtefailist ja kirjutab oma töö tulemuse kettale. Enamasti on aga preprotsessor ühendatud kompilaatoriga kui selle esialgne faas ja eraldi faili preprotsessis ei moodustata.

Kogu preprotsessi juhitakse spetsiaalsete preprotsessori juhtridade ehk direktiivide abil. Preprotsessoridirektiiv on rida, mis algab sümboliga # ja millele järgneb direktiivi nimi (mõnes realisatsioonis on enne sümbolit # ning sümboli # ja nime vahel lubatud tühisümbolid). Standardsed preprotsessoridirektiivid on järgmised:

- #define - makroasendus;
- #undef - makroasenduse tühistamine;
- #include - teksti lisamine teisest failist;
- #if - teksti tingimuslik kompileerimine, sõltuvalt mingi konstantavaldise väärtusest;
- #ifdef - tingimuslik kompileerimine, sõltuvalt mingi makronime defineeritusest;
- #ifndef - vastandvõimalus direktiivile #ifdef;
- #else - direktiividega #if, #ifdef ja #ifndef määratud teksti alternatiivi määramine;
- #endif - tingimusliku kompileerimise lõputunnus;
- #line - reanumbrite lisamine teadetesse.

Järgmised kaks direktiivi pole küll üldiselt kasutusel, kuid leiduvad paljudes kompilaatorites:

- #elif - asendab järjestikused #else ja #if;
- #if defined - katab direktiivid #ifdef ja #ifndef.

Preprotsessoridirektiivide süntaks on sõltumatu keele C süntaksist. Makroasendusega teksti kantavad lõigud ei pea preprotsessoridirektiivi koosseisus olema keele C süntaksi mõttes korrektsed. ANSI C kasutab lisaks nimetatutele veel preprotsessoridirektiive #pragma ja #error.

3.2. Preprotsessori leksika

Makroasenduste kindlaksmääramisel jagab ka preprotsessor lähteteksti lekseemideks, kasutades samu lekseemi eraldamise reegleid nagu kompilaatorgi. Preprotsessor tunneb ära nimed, konstandid, sõned ja kommentaarid. Makroasendusi ei teostata kommentaarides, sümbolkonstantides ja sõnedes (kuigi mõnes realisatsioonis kontrollitakse parameetrite leidumist ka parameetritega makroasenduse osaks olevates sõnedes).

Direktiivi nimele järgnev osa reast võib vajaduse korral sisaldada argumente, tühisümboleid ja kommentaare. Kui direktiiv ei nõua argumente, siis ei või neid ka olla (üldiselt ei ignoreeri preprotsessor direktiivi mittevajalikku osa, vaid loeb selle veaks). Mõningail juhtudel lubatakse kommentaare ainult preprotsessoridirektiivi lõppu. Kui direktiivi viimaseks sümboliks on langkriips \, siis jäetakse see ja temale järgnev reavahe, s.t. preprotsessoridirektiiv jätkub järgmisel real. Muuhulgas ei vaadelda järgmise rea esimese sümbolina esinevat sümbolit # sel juhul preprotsessoridirektiivi tunnusena. Näiteks

```
#define err(flag,msg)   if (flag)
    printf(msg)
```

on samaväärne direktiiviga

```
#define err(flag,msg)   if (flag)   printf(msg)
```

3.3. Makroasendused

Preprotsessoridirektiiv #define määrab makronime (identifikaatori), millega seostatakse lekseemide jada - direktiivi #define ülejäänud osa. Kui preprotsessor leiab selle nime lähtetekstist või ka teiste preprotsessoridirektiivide argumentide osast, siis ta asendab selle direktiivis #define antud lekseemide jadaga. Kui makros #define on nimi antud parameetritega, siis makrolaiendis asendatakse parameetrid nende tegelike väärtustega.

Preprotsessor ei erista võtmesõnu teistest nimedest ning seega on põhimõtteliselt võimalik kasutada ka võtmesõnu mak-

ronimedena, kuigi seda tuleb lugeda halvaks programmeerimisstiiliks. Makronimesid ei tunne preprotsessor ära kommentaarides, sõnedes ja sümbolkonstantides.

Preprotsessoridirektiivil #define on kaks erikuju, mille erinevus algab sellest, kas makronimele vahetult järgneb avav sulg (või mitte. Lihtsal vormil pole avavat sulgu, s.t. direktiiv esitatakse kujul:

```
#define nimi asendus
```

Sellisel kujul antud makrol pole parameetreid. Kui tekstist leitakse nimi, siis ta lihtsalt asendatakse vastava asendusega (ehk makrolaiendiga).

Parameetriteta makroasenduse peamiseks kasutusviisiks on programmis vajalikele põhilistele konstantidele nime omistamine. See võimaldab esiteks muuta programmi loetavamaks ja teiseks esitada konstandid programmis ainult üks kord, mis tunduvalt kergendab programmi modifitseerimist juhul, kui selle konstandi väärtust on vaja muuta. Mõningaid näiteid niisuguste makrode kohta:

```
#define TRUE         1
#define FALSE        0
#define EOF          (-1)
#define BLOCK_SIZE   0x100 /* kettabloki pikkus */
#define TRACK_SIZE   (16 * BLOCK_SIZE) /* raja pikkus */
#define READ_ACCESS    0x01
#define WRITE_ACCESS   0x02
#define APPEND_ACCESS 0x04
```

Nagu näeme algab makrolaiend esimesest makronimele järgnevast mittetühjast sümbolist, ilma mingite spetsiaalsete sümboliteta. Levinud veaks on liigsete sümbolite asetamine makrolaiendisse, näiteks kui pannakse omistamismärk = nime ja makrolaiendi vahele:

```
#define NUMBER = 5
```

Iseenesest ei ole siin midagi veel valesti. Küsimus on aga selles, et programmeerija kipub mitte pidama võrdusmärki makrolaiendi osaks, mis annab võimaluse nii süntaktilisteks kui ka semantilisteks vigadeks. Näiteks

```
count = NUMBER;
```

laiendatakse peale makroasendust vigasele kujule

```
count = = 5;    /* süntaktiliselt vigane */
```

Süntaksiviga ei pruugi aga alati ka esineda, näiteks

```
result = count+NUMBER;
```

laiendatakse kujule

```
result = count+=5;  /* tõenäoliselt vigane */
```

mis on süntaktiliselt õige, kuid vaevalt, et sooviti muutujale count lisada konstanti 5 nagu nüüd välja tuleb. Asja halb külg on siin selles, et kirjapilt, mis annaks programmeerija poolt (tõenäoliselt) soovitud tulemuse, on enne makroasendust keele C süntaksi mõttes vigane:

```
count NUMBER;
```

laiendatakse kujule

```
count = 5;
```

Samal põhjusel ei soovitata panna makrolaiendi lõppu semikoolonit, sest lähtekeele konstruktsioonid oleks siis enne makroasendust süntaktiliselt vigased.

Teine makroasenduse variant on defineerida makronimi koos sulgudesse asetatud ja komadega eraldatud parameetritega. Avav sulg peab siin vahetult järgnema makronimele:

```
#define  nimi(nimi1,nimi2,...,nimiN)  asendus
```

Makro parameetrid peavad olema üksteisest erinevad nimed. Kõik parameetrid ei pea ilmtingimata asenduses esinema, kuid loomulik on nende esinemine.

Programmi tekstis tuleb selliselt defineeritud makronimi anda koos järgneva avava suluga, üksteisest komadega eraldatud tegelike parameetritega ja lõpetava suluga, näiteks:

```
#define sum(x,y)  ((x) + (y))

...

return sum(a+3,b);
```

Makronime ja avava sulu vahel võib sellises makroväljakutses esineda tühisümboleid, samuti võivad tühisümbolid olla tegelike parameetrite koostisosaks.

Makrol võib olla ka null parameetrit. Sellisel juhul tuleb vastav nimi anda küll järgnevate sulgudega, kuid tühja tegelike parameetrite osaga. See võimaldab makrodena esitada argumentideta funktsioone:

```
#define  getchar()  getc(stdin)
...
while((c=getchar()) != EOF) ...
```

Makro tegelik parameeter võib sisaldada sulgusid (eeldades, et need on balansseeritud) ja komasid, kui need asuvad mingeis sisemistes sulgudes. Koma või sulg võivad esineda ka sümbolkonstandi või sõne koostisosana, sellised sulud muidugi ei pea olema balansseeritud. Tuleb arvestada, et loogelised ning nurksulud tegelikes parameetrites ei pea olema balansseeritud ega maskeeri ka komasid. Mõned näited:

```
#define   sum(x,y)  ((x) + (y))
...
result = sum(f(a,b) , g(a,b));
...
#define  insert(stmt)  stmt
insert({ a=1; b=1; })      /* see on korrektne */
insert({ a=1, b=1; }) /* vigane, koma maskeerimata */
insert({ (a=1, b=1); })  /* koma maskeeritud */
```

Makroasendusel asendatakse kogu makro koos tegelike parameetritega pärast viimaste töötlust makrolaiendiga. Tegelike parameetrite töötlus toimub järgnevalt. Iga tegelik parameeter seatakse vastavusse oma formaalse parameetriga ja makrolaiendis asendatakse viimase iga esinemine tegeliku parameetriga. Saadud makrolaiend asendabki makroväljakutse.

Vaatleme järgmist näidet, mis võimaldab programmis kõik seal esinevad spetsiaalse kujuga tsüklilaused asendada vastavate konstruktsioonidega incr, milles on määratud tsüklimuutuja, algväärtus ja lõppväärtus:

```
#define incr(v,l,h)  for((v) = (l); (v) <= (h); (v)++)
main()
{
 int i;
 /* trükime arvude 1, ... , 20 kuubid */
 incr(i,1,20)
 printf("Arvu %d kuup on %d\n",i,i*i*i);
}
```

Makrotöötlusega saab see tekst (kommentaarideta) kuju:

```
main()
{
int i;
for((i) = (1); (i) <= (20); (i)++)
printf("Arvu %d kuup on %d\n",i,i*i*i);
}
```

Pärast makrolaiendi sisseviimist teksti jätkub preprotsess taas selle makrolaiendi algusest, mis võimaldab korduvaid makroasendusi. Siin tuleb silmas pidada, et direktiivi #define makrolaiendit ennast ei töödelda makroasenduse mõttes, see on võimalik ainult pärast asendust. Olgu meil näiteks antud järgmised makromäärangud:

```
#define plus(x,y)  add(y,x)
#define add(x,y)   ((x) + (y))
```

makro plus(plus(a,b),c) laiendatakse järgmise skeemi järgi:

```
plus(plus(a,b),c)
add(c,plus(a,b))
((c) + (plus(a,b)))
((c) + (add(b,a)))
((c) + (((b) + (a))))
```

Tuleb aga silmas pidada, et korduvalt teostatakse ainult makroasendusi, mitte kogu preprotsessi, s.t. kui makrolaiend ise määrab preprotsessoridirektiivi, siis jääb see täitmata ja antakse muutmatul kujul kompilaatorile edasi. Asi lõpeb süntaksiveaga, sest sümbol # määrab preprotsessoridirektiivi ainult enne makrotöötlust.

Peab märkima, et võimalik on ka rekursiivne makroasendus: makro, mis makrolaiendis sisaldab iseennast, näiteks

```
#define repeat(x) x repeat(x)
```

Sellisel juhul jääb makrotöötlus lõpmatusse tsüklisse. Enamik kompilaatoreid seda viga ei avasta ja töötlus toimub kuni mingi muu tõenäolise veani (näiteks täitub reapuhver).

Tuleb veel märkida, et mõningad realisatsioonid annavad kasutaja käsutusse nn. standardmakrod, mida ei saa kasutada preprotsessoridirektiivis #undef. Näiteks makro __FILE__ tähistab sageli kompileeritava faili nime ja __LINE__ jooksvat reanumbrit (sobivad kasutada näiteks väljatrükkides). Ana-

loogiliselt annavad mõningad realisatsioonid standardmakrod ka selleks, et kindlaks teha, millises süsteemis toimub kompileerimine. Nii näiteks võib UNIX-kompilaatoris olla antud standardmakro unix, mille olemasolu kontrollimise teel saab programmi sisse tuua konkreetselt UNIX-ist sõltuvaid programmilõike, näiteks:

```
#ifdef unix
  /* spetsiaalselt UNIX-ist sõltuv osa */
...
#endif
```

Preprotsessoridirektiiviga #undef tühistatakse direktiivis #define antud määrang. Selle direktiivi kuju on

```
#undef  nimi
```

Direktiivis #undef toodud nimi ei pea kindlasti olema defineeritud. Pärast direktiivi #undef on ta aga kindlasti defineerimata ja teda võib kasutada uuesti direktiivis #define. Tuleb märkida, et mingi makronime ümberdefineerimine ilma vahepealse direktiivita #undef on mõnes realisatsioonis lubatud, teistes aga toob kaasa vea, seepärast on parim praktika seda igal juhul kontrollida, näiteks:

```
#ifdef TABLESIZE
#undef TABLESIZE
#endif
#define TABLESIZE  ...
```

Makroasenduses on tegemist puhtalt tekstuaalse asendusega. Lekseemideks eraldamine kompilaatori poolt toimub alles pärast seda. Selle asjaolu arvestamata jätmine võib mõnikord tuua kaasa ootamatuid tagajärgi. Olgu antud makromäärang:

```
#define SQUARE(x)   x * x
```

siis makro, millega soovitakse leida argumendi ruut, näiteks SQUARE(5) asendatakse tekstilõiguga 5 * 5. Kuid näiteks

```
SQUARE(z+1)
```

annab meile ilmselt ebaõige tulemuse

```
z+1 * z+1
```

Selle vältimiseks tuleb makrolaiendis panna makro iga parameeter sulgudesse. Vaadeldes aga näidet

```
(short) SQUARE(z+1)
```

saab selgeks, et sulgudesse tuleb asetada ka kogu makrolaiend. Seega korrektne makromäärang ruudu leidmiseks on

```
#define SQUARE(x)  ((x) * (x))
```

Mõningaid probleeme tekitavad makrod ka kõrvalefektiga avaldistes. Selle illustreerimiseks vaatleme näiteks funktsiooni, mis leiab oma argumendi ruudu:

```
int square(x)
int x
{
return x * x;
}
```

ning kaht programmilõiku (makro SQUARE on endine):

```
a = 3;
b = square(a++);
```

ja

```
a = 3;
b = SQUARE(a++);
```

Esimesel juhul (ruut leitakse funktsiooniga) on pärast lõigu täitmist b väärtus 9 ja a väärtus 3. Teisel juhul aga (tegemist on makroga) laiendatakse teine rida kujule

```
b = ((a++) * (a++));
```

kust nähtub, et pärast selle lõigu täitmist võib a väärtus olla 5 ja b väärtus 12 (just nimelt võib olla, kuna keeles C võib kompilaator avaldisi väärtustada mitmes järjekorras).

Mõningad (kuid mitte kõik) kompilaatorid asendavad makro formaalse parameetri tegelikuga ka siis, kui see parameeter esineb asenduse koosseisus olevas sõnes. See võib kaasa tuua rea probleeme ja kasulikum on sellest hoiduda. Kompilaatorid ei kontrolli alati makroasenduses sõnet piiravate jutumärkide paarsust, kuid lasta asendada n.-ö. poolikut sõnet on samuti väga küsitava väärtusega.

3.4. Failide lisamine teksti

Preprotsessoridirektiivi #include toimel paigutatakse selle direktiiviga määratud faili sisu programmi teksti direktiivi #include asemele. Direktiivis #include võib faili

nimi olla antud kahel viisil. Kui esimeseks mittetühjaks sümboliks peale direktiivi nime on ", siis peab ka viimane sümbol olema "; kui esimene sümbol on <, siis peab viimane sümbol olema >. Mõlemal juhul määravad nende eraldajate vahel olevad sümbolid lisatava faili nime (mille konkreetne formaat sõltub operatsioonisüsteemist). Seega tuleb direktiiv #include esitada ühel kahest võimalikust kujust:

```
#include   "faili_nimi"
#include   <faili_nimi>
```

Üldiselt näitavad need erinevad kujud, kust antud faili tuleb otsida juhul, kui tema asukoht ei ole nimega üheselt ära määratud. Esimesel juhul otsitakse faili samast kataloogist kust kompileeritavat lähtefailigi, teisel juhul aga teatavast standardsest "süsteemikataloogist". Tähenduse erinevus on tavaliselt selles, et esimesel juhul võiks olla tegemist programmeerija enda poolt kirjutatud teiste programmidega, teisel juhul aga standardsete teegiprogrammidega. Mõned realisatsioonid ei erista neid kahte võimalust, samuti lubatakse mõnel juhul esitada faili nime üldse ilma spetsiaalsete eraldajateta.

Direktiiviga #include lisatud fail võib omakorda sisaldada direktiive #include. Sellise sisalduvuse sügavus sõltub küll realisatsioonist, kuid pole tavaliselt kunagi väiksem kui viis või kuus.

3.5. Tingimuslik kompileerimine

Tingimusliku kompileerimise direktiivid võimaldavad sõltuvalt mingitest tingimustest osa teksti failist kas programmi lülitada või vahele jätta.

Direktiivid #if, #else ja #endif. Võimalus ridade programmi lülitamiseks teatud tingimusel on siin järgmine:

```
#if konstantavaldis
    tekst1
#else
    tekst2
#endif
```

Konstantavaldis võib olla saadud makroasenduste tulemusena ja peab määrama aritmeetilise väärtuse. Programmilõigud tekst1 ja tekst2 esitavad suvalist teksti, sealhulgas ka teisi preprotsessoridirektiive, näiteks direktiivi #if. Direktiiv #else ja järgnev tekst2 võib ka puududa (sellega samaväärne on tühi tekst2). Kui kummaski neist tekstidest leidub omakorda direktiive #if, siis peavad nad olema balanseeritud vastavate direktiividega #endif.

Vaadeldavate direktiivide täitmisel väärtustatakse kõigepealt konstantavaldis. Kui saadud väärtus erineb nullist, siis lisatakse programmi tekst1 ja jäetakse vahele tekst2 (kui ta esineb). Kui konstantavaldise väärtus on null, siis jäetakse tekst1 vahele ja lisatakse tekst2 (kui ta esineb). Kui tekstilõik jäetakse vahele, siis jäävad töötlemata ka temas esinevad preprotsessoridirektiivid (sealhulgas näiteks ka kõik makrod #define).

Kui konstantavaldise väärtustamisel leitakse nimi, mis ei ole määratud makrona, siis loetakse tema väärtuseks null. Nagu edasises näeme tähendab see, et direktiivid #ifdef nimi ja #if nimi on samaväärsed juhul, kui nimi on määratud makroasenduses nullist erineva konstandina.

Preprotsessoridirektiivi #elif toetavad ANSI C ning veel mõned realisatsioonid kui lühendit järjestikustele direktiividele #else ja #if. Näiteks konstruktsioon

```
#if konstantavaldis1
    tekst1
#elif konstantavaldis2
    tekst2
...
#elif konstantavaldisN
    tekstN
#else
    viimane_tekst
#endif
```

moodustab lüliti, kus programmi teksti lülitatakse tervest hulgast võimalustest üks - see mille korral vastav konstantavaldis on esimesena nullist erinev ning jäetakse vahele

kõik ülejäänud. Kui kõigi konstantavaldiste väärtused on nullid, kuid leidub (mittekohustuslik) #else, siis lülitatakse programmi viimane_tekst. Konstantavaldiste väärtustamine toimub samuti kui direktiivi #if korral. Põhimõtteliselt pole direktiivil #elif iseseisvat tähtsust, kuna sama konstruktsiooni saab (küll mõnevõrra kohmakamalt) väljendada ka direktiividega #if, #else ja #endif.

Direktiive #ifdef ja #ifndef võib kasutada koos direktiividega #else ja #endif samuti kui direktiivi #if. Erinevusena on nendes direktiivides tingimuseks nimi, mille määratust või mittemääratust makrona kontrollitakse. Direktiiv

```
#ifdef nimi
```

on samaväärne direktiiviga #if 1 siis, kui nimi on määratud makrona ja direktiiviga #if 0, kui ta seda ei ole. Direktiivi #ifndef tähendus on vastupidine. Tuleb märkida, et nime määratus tähendab siin ainult esinemist mingi direktiivi #define makronimena (või standardmakrona) ega sõltu sellest, kas programmis on või ei ole määratud sellenimeline muutuja.

Direktiivide #ifdef üheks levinud kasutusviisiks on mitmele arvutile mõeldud rakenduste kirjutamine. Näiteks võib lisada programmi sobiva järgnevatest direktiividest #define:

```
#define  VAX    1
#define  PDP    1
#define  IBM360 1
```

Siis järgmine programmilõik on masinsõltumatu:

```
#ifdef  VAX
/* spetsiaalkood VAX jaoks */
...
#endif
#ifdef   PDP
/* spetsiaalkood PDP jaoks */
...
#endif
#ifdef   IBM360
/* spetsiaalkood IBM jaoks */
...
#endif
```

Vaadeldavate direktiivide teiseks laialdaseks kasutamisvõimaluseks on mõningatele konstantidele päisfailis vaikimisi väärtuste andmine vaid siis, kui programmeerija pole seda ümber määranud. Oletame näiteks, et päisfail table.h määrab mingi sisemise tabeli:

```
/* päisfail table.h - tabeli määramine */
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
struct table { ... };
struct table internal_table[TABLE_SIZE];
```

Kui programmeerija ei ole rahul vaikimisi määratud tabeli suurusega 100, siis saab ta selle ümber määrata:

```
#define TABLE_SIZE 500
#include <table.h>
...
```

ANSI C toob sisse konstruktsiooni defined, mida võib kasutada direktiivides #if ja #elif. Tema kuju on järgmine:

```
#if defined(nimi)
#elif defined(nimi)
```

Konstruktsiooni defined(nimi) väärtus on 1 siis, kui nimi on defineeritud makrona ja 0 vastasel korral. See konstruktsioon asendab sisuliselt direktiivid #ifdef ja #ifndef.

3.6. Ridade ilmutatud nummerdamine

Direktiiv #line teatab kompilaatorile, millise faili ja millise reaga on momendil tegemist. Sellel direktiivil on mõtet siis, kui fail pole kirjutatud vahetult, vaid genereeritud mingi automaatse vahendi (näiteks spetsiaalse makroprotsessori) abil. Sellisel juhul saab direktiividega #line viidata tegelikule lähtefailile ja tema reanumbritele. Direktiivil #line on järgmine kuju:

```
#line täisarv "faili_nimi"
```

Direktiiv teatab kompilaatorile, et programmi järgmine rida on genereeritud lähtefaili faili_nimi reast näidatud reanumbriga täisarv. Mõningates realisatsioonides direktiiv #line puudub, mõnedes töötab ta aga kui tühidirektiiv.

4. K I R J E L D U S E D

4.1. Kirjelduste liigitus

Nimed väljendavad keeles C järgmist liiki objekte:

- muutujad
- funktsioonid
- tüübid
- struktuuride ja ühendite komponendid
- loendikonstandid
- märgendid
- makrod

Kõigist loetletud objektiklassidest räägime hiljem üksikasjalikumalt. Välja arvatud makrod ja märgendid seotakse nimed ülejäänud objektitüüpidega kirjeldustes. Kirjeldus on keelekonstruktsioon, mis seob programmis mingi nime mingi konkreetse C-keelse objektiga nagu muutuja, funktsioon või tüüp. Kirjeldused jagunevad kahte klassi - definitsioonid millega kirjeldatavad objektid ka luuakse ja deklaratsioonid, mis teatavad kusagil mujal defineeritud objektidest. Defineerida saab igat objekti kogu programmi jooksul ainult ühes kohas, deklaratsioone võib aga olla palju, näiteks välismuutuja deklaratsioonid eri lähtefailides.

Muutujad, funktsioonid ja tüübid esitatakse kirjelduseosas, mida nimetame deklaraatoriks; tüübinimed, struktuuride ja ühendite komponendid ning loendikonstandid esitatakse kirjelduse koosseisu kuuluvas tüübikirjeldajas (ehk tüübispetsifikatsioonis). Märgendid defineeritakse nende esinemistega funktsioonides lausete koosseisus ning makrod - preprotsessoridirektiividega #define.

Kirjelduste tähenduse selgitamisel keeles C tekivad mõningad raskused. Mitteharjumuspärane süntaks võib kohutada algajaid, kuna kirjeldatava objekti tegelik olemus on ära peidetud kahte eri konstruktsiooni - tüübikirjeldajasse ja deklaraatorisse. Näiteks kirjeldab deklaratsioon

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

meile viida funktsioonile, mille väärtus on tüüpi int.

Kirjeldused võivad programmis asuda mitmetel erinevatel kohtadel. Asukohast sõltuvad ka kirjeldatavate objektide omadused. Iga programm keeles C koosneb teatud hulgast väliskirjeldustest. Väliskirjeldused võivad määrata funktsioone, muutujaid jms. Iga funktsioon esitab oma parameetrite kirjeldused ning sisu, milles sisalduvad blokid võivad omakorda sisaldada sisemisi kirjeldusi. Järgnevad süntaksivalemid määravad kirjelduste asukoha C-programmis. Mõned konstruktsioonid, mis antud vaatenurgast ei paku veel huvi, jätame siinkohal vahele, märkides neid kolme punktiga.

```
C-programm:
        väliskirjeldused
väliskirjeldused:
        väliskirjeldus
        väliskirjeldused väliskirjeldus
väliskirjeldus:
        algväärtus-kirjeldus
        funktsiooni-definitsioon
funktsiooni-definitsioon:
        kirjeldajad deklaraator funktsiooni-sisu
kirjeldajad:
        mäluklassi-kirjeldaja
        tüübi-kirjeldaja
        kirjeldajad mäluklassi-kirjeldaja
        kirjeldajad tüübi-kirjeldaja
funktsiooni-sisu:
        kirjeldused  blokk
kirjeldused:
        kirjeldus
        kirjeldused kirjeldus
kirjeldus:
        kirjeldajad deklaraatorid ;
deklaraatorid:
        deklaraator
        deklaraatorid , deklaraator
blokk:
        { algväärtus-kirjeldused  laused }
```

```
laused:
        lause
        laused lause
lause:
        blokk
        ...
algväärtus-kirjeldused:
        algväärtus-kirjeldus
        algväärtus-kirjeldused algväärtus-kirjeldus
algväärtus-kirjeldus:
        kirjeldajad deklaraatorid-väärtused ;
deklaraatorid-väärtused:
        deklaraator-väärtus
        deklaraatorid-väärtused , deklaraator-väärtus
deklaraator-väärtus:
        deklaraator algväärtustaja
deklaraator:
        nimi
        ...
algväärtustaja:
        = algväärtus
algväärtus:
        avaldis
        ...
```

Nagu näha on kõigil kirjeldustel peale funktsiooni definitsiooni ühesugune süntaks. Semantikareeglid keelavad küll mõned süntaktiliselt õiged kirjeldused, kuid need reeglid võtame vaatlusele mõnevõrra hiljem.

4.2. Mõnda terminoloogiast

Enne kui hakata selgitama kirjelduste konkreetset tähendust selgitame mõningaid mõisteid.

Kirjelduse ja kirjeldatava nime skoop ehk kehtivuspiirkond on programmiosa, mille jooksul see kirjeldus on aktiivne. Skoobiks võib olla blokk, funktsioon või programmi suurem osa. Nimedel on keeles C üks järgmistest skoopidest.

a) Väliskirjelduses määratud nime skoop paikneb kirjelduse asukohast failis kuni faili lõpuni.

b) Funktsiooni parameetrikirjelduses määratud nime skoop paikneb kirjelduse asukohast kuni funktsiooni lõpuni.

c) Bloki alguses kirjeldatud nime skoop paikneb kirjelduse asukohast kuni bloki lõpuni.

d) Märgendi skoop on funktsioon, kus see märgend esineb.

e) Makronime skoobiks on programmiosa teda määravast direktiivist #define kuni määrangut tühistava direktiivini #undef või siis kuni faili lõpuni.

Öeldakse, et kirjeldus (nimi) on nähtav mingis kontekstis, kui nime kasutamine põhineb sellele kirjeldusele. Kirjeldus võib olla nähtav kogu oma skoobi ulatuses, kuid teda võib varjata mingi teine kirjeldus, mis määrab sama nime ja mille skoop katab osa tema skoobist. Näiteks järgmises programmilõigus kaetakse täisarvulise muutuja x määrang sisemise kirjeldusega, mis määrab x reaalarvulise muutujana:

```
int x = 10; /* x on välimisel tasemel täisarvuline */
main()
{
double x;    /* sisemine kirjeldus varjab välimise */
... sin(x) ... /* x on siin reaalarvuline muutuja */
}
```

Funktsiooni parameetrite kirjeldused varjavad vastavad väliskirjeldused ja kirjeldused bloki alguses varjavad kirjeldused väljaspoolt blokki. Üldiselt kirjeldus varjab teise siis, kui ta määrab sama nime, mis kuulub samasse nimeklassi ja mille skoop sisaldub teise skoobis.

Keeles C nagu ka paljudes teistes programmeerimiskeeltes võib ühel nimel olla programmi samal kohal mitu erinevat tähendust. Millisele nime tähendusele viidatakse selgub viitamise kontekstist. Erineva tähendusega nimede kasutamise määrab nime kuuluvus konkreetsesse nimeklassi. Iga nimeklassi nimedel on oma skoop ja nähtavus. Näiteks võib üks ning sama nimi olla kasutusel muutujanimena ja tüübinimena. Nende nimede skoobid ei varja üksteist, kuna nimele viitamise kontekstist on alati selge, millist tähendust silmas peetakse.

Tavaliselt eristatakse keeles C viit eraldi nimeklassi.

a) Preprotsessori makronimed. Kuna nad töödeldakse eri etapina enne kompileerimist, siis on nad sõltumatud kõigist teistest nimedest (ja ka võtmesõnadest).

b) Märgendid, mille definitsioonides järgneb nimele vahetult eraldaja : , mis võimaldab märgendeid ära tunda. Märgendile viidatakse ainult suunamislauses goto.

c) Struktuuri-, ühendi- ja loenditüüpide nimed, millele viidatakse alati läbi võtmesõnade struct, union või enum.

d) Komponentide nimed, mis igas struktuuris või ühendis moodustavad eraldi nimeklassi. Seega üks ja sama nimi võib olla suvaliste struktuuride või ühendite komponendi nimeks. Sellistele nimedele viitamine toimub alati läbi komponendi valiku operaatori . või -> .

e) Kõik ülejäänud (muutujate, funktsioonide, tüüpide ja loendikonstantide) nimed moodustavad ühise nimeklassi.

Nimede siintoodud klassijaotus erineb veidi originaal-C nimeklassidest, kus märgendid kuuluvad muutujanimedega ühte klassi (seda järgivad ka praegu mitmed kompilaatorid). Teiseks moodustavad komponentide nimed originaal-C korral kokku ühe nimeklassi: välja arvatud väga kunstlikud erijuhud ei lubata kahel struktuuril või ühendil samanimelisi komponente. Selline kitsendus on aga tarbetu ja ebamugav ning enamik kompilaatoreid seda enam ei järgi.

Eri realisatsioonides võib täheldada veel mõningaid erinevusi nimede jaotamisel klassidesse. Mõnikord loetakse struktuuri-, ühendi- ja loenditüüpide nimed samasse klassi muutujanimedega, mõnikord aga moodustavad nad kolm omaette nimeklassi (viimane on õigustatud, kuna viitamise kontekstist järeldub siin alati, millist liiki nimega on tegemist).

Lisaks nimeklassile, skoobile ja nähtavusele seotakse nimede ja kirjeldustega veel eksisteerimisaja ehk ekstendi mõiste. See on erinevalt teistest mitte kompileerimis- vaid täitmisaja termin ja määrab perioodi, mille vältel vastav objekt programmi täitmisel tegelikult eksisteerib. Ekstent määratakse ainult muutujate ja funktsioonide, mitte teiste nimede jaoks. Ekstendid jagunevad järgmistesse liikidesse.

a) Staatiline ekstent tähendab eksisteerimist kogu programmi täitmisaja vältel. Keeles C on staatilise ekstendiga kõik funktsioonid ja kõik väliskirjeldustes määratud muutujad. Blokis kirjeldatud muutujatel võib olla staatiline ekstent sõltuvalt kirjeldamisviisist.

b) Objektil on lokaalne ekstent siis, kui see objekt luuakse funktsiooni või blokki sisenemisel ja ta hävib sealt väljumisel. Kui lokaalse ekstendiga muutujale on määratud algväärtus, siis muutuja algväärtustatakse alati tema loomisel. Lokaalse ekstendiga on funktsiooni parameetrid. Blokis kirjeldatud muutujal on lokaalne ekstent sõltuvalt kirjeldamisviisist. Kõiki lokaalse ekstendiga muutujaid kutsutakse keeles C automaatmuutujaiks.

c) Objektil on dünaamiline ekstent siis, kui selle objekti loomine või hävitamine toimub dünaamiliselt, s.o. vahetult programmeerija poolt. Dünaamilisi objekte loovad keele C mõningad teegifunktsioonid (näiteks malloc), keeles endas ei ole vahendeid nende tekitamiseks.

4.3. Kirjeldamise üldküsimused

Välja arvatud mõningad erisituatsioonid ei tohi keeles C nime kasutada enne selle kirjeldamist. Nimetame nime kirjeldamispunktiks seda kohta programmitekstis, kus vastav nimi kui lekseem esineb teda määrava kirjelduse koosseisus. Nimi on kasutatav kohe alates oma kirjeldamispunktist. Näiteks täisarvulise muutuja intsize saame järgnevas näites algväärtustada tema enda pikkusega, sest algväärtustamine toimub kirjeldamispunktist tagapool:

```
static int intsize = sizeof(intsize);
```

Nime kasutamist tekstis tema kirjeldamispunktist eespool nimetame ettepoole viitamiseks. Keeles C on lubatud ettepoole viitamine kahel juhul. Esiteks võib märgendinime kasutada suunamislauses enne selle märgendi defineerimist programmis. Teiseks lubab enamik realisatsioone struktuuri-, ühendi- või loenditüübi nime mõningates kontekstides kasutada enne selle tüübi täielikku kirjeldamist (seda võimalust vaatleme lähe-

malt seoses vastavate tüüpide käsitlemisega). Järgnevas näites on aga ettepoole viitamine ebakorrektne:

```
typedef struct { int value; cell *next; } cell;
```

sest nime cell kirjeldamispunktiks on tema viimane esinemine ja seega osutub selle kasutamine struktuuri sees lubamatuks ettepoole viitamiseks.

Ühe ja sama nime mitmekordne kirjeldamine samas nimeklassis kas ülemisel tasemel või siis samas blokis on vigane. Selliseid kirjeldusi nimetame konfliktseiks. Järgmises näites on nime howmany kaks kirjeldust konfliktsed, kuid nime str kaks kirjeldust ei ole, sest objektid kuuluvad eri nimeklassidesse:

```
extern int howmany;
extern char str[10];
typedef double howmany();
extern struct str { int a,b; } x;
```

On aga siiski kaks korduvaid kirjeldusi lubavat erandit. Esiteks tohib olla antud kuitahes palju välisobjekti deklaratsioone, eeldades et kõikides neis määratakse nimele sama tüüp. Näiteks välise teegifunktsiooni kahekordne deklareerimine pole vigane. Teiseks, kui on antud välisobjekti deklaratsioon, siis võib sellele järgneda ka selle objekti definitsioon. See lahendab probleemid, mis võivad tekkida ettepoole viitamise keelamisest. Olgu meil näiteks kaks funktsiooni f ja g, mis kumbki pöördub teise poole. Selleks, et üks funktsioon saaks teise poole pöörduda, peab teise funktsiooni nimi olema selleks momendiks määratud:

```
extern double f();
double g(x,y)
double x,y;
{ ... f(x-y) ... }
double f(z)
double z;
{ ... g(z,z/2.0) ... }
```

Sellise mehhanismi puuduseks on asjaolu, et ettepoole viidatav nimi peab olema kindlasti defineeritud kui extern, s.t. tal ei või olla mäluklassi static.

Et nime skoop algab tema kirjeldamispunktist ja mitte näiteks bloki algusest, võib tekkida mõnevõrra veider situatsioon, kus bloki sees on ühel ja samal nimel mitu tähendust. Näiteks järgmine programmilõik

```
{
int i = 0;
...
    {
    int j = i;
    float i = 10.0;
    ... }
}
```

on korrektne, sest kuni i kirjeldamiseni sisemises blokis kehtib välimise bloki definitsioon, mis määrab i täisarvuliseks muutujaks. Siiski on niisuguste kirjelduste kasutamine kahtlane ja sellest tuleks hoiduda.

Muutujale mälu eraldamine ei pruugi alati tähendada tema algväärtustamist. Enamikku muutujaid keeles C saab algväärtustada, s.t. siduda nad avaldistega, mis määravad muutuja algväärtuse, mis omistatakse muutujale temale mälu eraldamisel. Kui algväärtust muutujale pole antud, siis on nende väärtus peale mälu eraldamist määramata.

On tähtis meeles pidada, et staatilised (ehk staatilise ekstentiga) muutujad algväärtustatakse programmi täitmisel vaid üks kord ja nad säilitavad oma väärtuse ka väljaspool nende skoopi. Näiteks järgmises blokis on nii muutuja L kui S algväärtustatud väärtusega 0. Mõlema muutuja skoobiks on blokk, kuid muutujal S on staatiline, muutujal L aga lokaalne (automaatne) ekstent:

```
{
static int S = 0;
auto   int L = 0;
L = L + 1; S = S + 1;
printf("L = %d, S = %d\n",L,S);
}
```

Igal sisenemisel sellesse blokki suurendatakse muutujate L ja S väärtusi ühe võrra ning need väärtused trükitakse. Kui

vaadeldavasse blokki pöördutakse korduvalt, siis väljatrükitavad väärtused kujunevad järgnevateks:

```
L = 1, S = 1
L = 1, S = 2
L = 1, S = 3
...
```

Automaatmuutujate korral on veel oluline teada, et nende algväärtustamine toimub ainult siis, kui blokki sisenetakse selle algusest. Järelikult sisenemisel blokki kas suunamislause goto või lüliti switch abil jäävad kõik automaatmuutujad algväärtustamata. Seega on järgmine näide vigane (sest muutuja sum jääb algväärtustamata):

```
goto L;
...
{
static int vector[9] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = 0; /* sisenemisel goto L; jääb täitmata */
L: for(i = 0; i < 9; ++i) sum += vector[i];
... }
```

Lõpuks veel üks märkus välisobjektide deklaratsioonide ehk extern-deklaratsioonide kohta. Sellised objektid võivad olla defineeritud mingis teises lähtefailis ning vastavuse antud deklaratsiooni ning mingis teises lähtefailis defineeritud objekti vahel loob alles linker. Enamik kompilaatoreid käsitleb extern-deklaratsioone samamoodi kõigi teistega, lugedes näiteks järgmise programmilõigu vigaseks (sest selles on E kirjeldatud sisemises blokis, s.t. tema skoop ei levi üle bloki piiride):

```
{
  {
  extern E;
  E = 0;
  }
E = 1;
}
```

Siiski loeb mõni realisatsioon kõik extern-deklaratsioonid globaalseiks ja viimase programmilõigu seega korrektseks.

4.4. Mäluklassikirjeldajad

Vaatleme nüüd eraldi kirjelduste kõiki koostisosi, s.t. mäluklassikirjeldajaid, tüübikirjeldajaid, deklaraatoreid ja algväärtustajaid.

Mäluklassikirjeldaja määrab kirjeldatava objekti ekstendiliigi, kusjuures ühes kirjelduses võib olla antud ülimalt üks mäluklassikirjeldaja. Ehkki originaal-C ei nõua mäluklassikirjeldaja eelnemist tüübikirjeldajale on niisugune järjekord siiski saanud ainuvalitsevaks (ja võib mõnes realisatsioonis olla ka nõutav):

mäluklassi-kirjeldaja: variandid

auto extern register static typedef

Märgime, et mitte iga mäluklassikirjeldaja ei ole lubatud kirjelduse igas kontekstis. Nende kirjeldajate tähendused on järgmised.

1) Mäluklass auto on lubatud ainult mingi bloki alguses ja määrab muutujale lokaalse (automaatse) ekstendi. Et lokaalne ekstent määratakse bloki alguses defineeritavaile muutujaile ka vaikimisi, siis kirjeldajat auto kasutatakse programmides harva.

2) Mäluklass extern teatab, et tegemist on välise objekti deklareerimisega. Muutuja saab staatilise ekstendi, tema definitsioon võib asuda samas lähtefailis kuskil mujal või siis mingis teises lähtefailis.

3) Mäluklassi register võib kasutada lokaalsete muutujate või funktsiooni parameetrite defineerimisel. Tähendus on sama mis mäluklassil auto, kuid lisaks teatatakse kompilaatorile, et võimaluse korral tuleb muutuja paigutada arvuti registrisse.

4) Mäluklassi static võib kasutada funktsioonide ja muutujate defineerimisel. Funktsioonide korral (ekstent on neil alati staatiline) määrab ta, et funktsiooni nimi ei levi väljapoole antud lähtefaili piire (ega saa teatavaks linkerile). Muutujate korral määrab static muutujaile staatilise ekstendi ning samuti selle, et vastava muutuja nimi on lokaalne antud lähtefaili piires.

5) Võtmesõna typedef asub küll süntaktiliselt mäluklassi kohal, kuid tähendab seda, et määratakse mitte muutujat või funktsiooni, vaid tüüpi. Selle tüübi nimi antakse kirjelduses kirjeldatava nime kohal ning seda võib edasistes kirjeldustes kasutada tüübikirjeldaja koosseisus.

Mäluklassiga register seostub rida kitsendusi. Nimelt saab arvuti registritesse paigutada ainult kindlat tüüpi objekte - milliseid just, see sõltub arvutist ja realisatsioonist (üldiselt on objektid tüüpi int siin alati lubatud). Realisatsioonist sõltuv on ka aktsepteeritavate registrimuutujate arv funktsioonis. Kui see arv ületatakse, siis tõlgendatakse klassi register nagu klassi auto. Lisaks tuleb märkida, et registrimuutujail pole aadressi, seega operatsioon & pole nende suhtes rakendatav.

Kui kirjelduses mäluklassi ei ole antud, siis ta määratakse vaikimisi sõltuvalt kontekstist järgmiselt.

a) Väliskirjeldustele (seega ka funktsioonide definitsioonidele) määratakse staatiline ekstent ning vastavad nimed on nähtavad ka väljaspool antud lähtefaili (ja saavad teatavaks linkerile).

b) Funktsiooni parameetrite ainuke lubatud mäluklass on register. Seega vaikimisi eeldatakse "mitte register".

c) Bloki alguses antud kirjeldustes eeldatakse vaikimisi extern funktsioonide puhul ja auto ülejäänud juhtudel.

Vaatamata neile reeglitele on heaks stiiliks esitada mäluklass extern ilmutatult ka seal, kus ta võetaks vaikimisi. Teisest küljest on üldine praktika jätta ära mäluklass auto.

Näitena mäluklassikirjeldajate tüüpilisest kasutamisest toome funktsiooni, mis otsib sõnest parameetri abil ette antud sõna. Et põhifunktsioon substr peab olema nähtav teistele kasutajatele, siis saab ta vaikimisi mäluklassi extern. Abifunktsioonil streq on tähtsust ainult antud failis, seega sobib tema mäluklassiks static. Funktsioonis streq kasutatakse intensiivselt lokaalseid muutujaid p1 ja p2, seepärast on need defineeritud mäluklassiga register. Et funktsiooni substr lokaalse muutuja p kasutamise intensiivsus ei ole nii oluline, siis on talle vaikimisi antud mäluklass auto:

```
char *substr(string,word)
char *string, *word;
{
char *p = string;
while ( p )
     if( streq (p++, word ) return p;
return 0;
}
static int streq(str1, str2)
char *str1, *str2;
{
register char *p1 = str1, *p2 = str2;
while (*p1 && *p2)
      if(*p1++ != *p2++) return 0;
return 1;
}
```

4.5. Tüübikirjeldajad

Tüübikirjeldaja annab infot kirjeldatavate nimede tüüpide kohta. Seejuures ei määra ta tüüpi täielikult, kuna tüüpi täpsustab kirjelduse järgmine osa - deklaraator. Kõrvalefektina võib tüübikirjeldaja määrata struktuuri-, ühendi- või loenditüübi nime, struktuuride ja ühendite komponente ning loendikonstante. Nagu juba märgitud, on üldlevinud esitusviis, kus tüübikirjeldaja järgneb mäluklassikirjeldajale:

```
tüübi-kirjeldaja:
     täisarvutüübi-kirjeldaja
     reaalarvutüübi-kirjeldaja
     loenditüübi-kirjeldaja
     struktuuritüübi-kirjeldaja
     ühenditüübi-kirjeldaja
     tüübi-void-kirjeldaja
     typedef-nimi
```

Tüübikirjeldajaid kirjeldame detailsemalt järgmises peatükis. Siin toome vaid mõningad üldist laadi märkused. Kõigepealt mõned näited tüübikirjeldajate kohta:

```
void
int
unsigned long int
my_struct_type
union { int a; float b; }
enum { red, blue, green }
```

Kui tüübikirjeldaja puudub, siis loetakse see vaikimisi kirjeldajaks int. Sageli jäetakse tüübikirjeldaja ära funktsioonide definitsioonides. Varasemais kompilaatoreis, milledel puudus andmetüüp void, oli funktsioonile tüübikirjeldaja andmata jätmine sageli tunnuseks, et see funktsioon ei väljasta väärtust. Viimasel ajal aga on selle jaoks kasutusel andmetüüp void ja seda tuleb ka soovitada:

```
void sort(v,n)
int v[],n;
{ ... }
```

Kui kompilaator ei tunne võtmesõna void, siis võib selle defineerida kui int sünonüümi:

```
/* teeme void int sünonüümiks */
typedef void int;
```

Keele C süntaks nõuab kirjelduses vähemalt ühe kirjeldaja (kas mäluklassi- või tüübikirjeldaja) olemasolu. See on tingitud keele ühesuse nõuetest. Näiteks funktsiooni deklaratsioonis:

```
extern int f();
```

saaksime mõlema kirjeldaja ärajätmisel (mis vaikimisi reegleid kasutades oleks siin võimalik) konstruktsiooni

```
f();
```

mis aga ei erine enam pöördumisest funktsiooni f poole. Mõlemad kirjeldajad võib siiski ära jätta (ja kasutada vaikimisi võetavaid kirjeldajaid) funktsiooni definitsioonis, kuna seal järgnev parameetrite kirjeldamine ning funktsiooni sisu eristavad antud konstruktsiooni üheselt.

Selgitame lõpuks veel ühte peensust tüübikirjeldajate kasutamisel. Struktuuride, ühendite ja loendite definitsioonid võimaldavad kõrvalefekti - vastavale tüübile nime andmist. Näiteks

```
struct S { int a,b; }
```

defineerib struktuuritüübi S, mille komponentideks on kaks täisarvulist välja a ja b. Seda nime S saab kasutada edasistes kirjeldustes näiteks muutuja aaaa defineerimisel:

```
struct S aaaa;
```

Sellega seoses võib niisugustes kirjeldustes täielikult ära jätta deklaraatorid ja kasutada ainult kõrvalefekti:

```
struct S { int a,b; };
```

Seda kirjeldust võimaldab C grammatika ja seega ka kompilaatorid, kuid kompilaatorid ei erista enamasti antud variandi mõttetut kasutamist - kirjeldusi, kus pole ei deklaraatoreid ega kõrvalefekti:

```
struct { int a,b; };
```

Teine viga, mida kompilaator siin tõenäoliselt ei avasta on mäluklassi kasutamine. Mäluklass kuulub tegelikult eksisteerivale objektile. Kui aga anda näiteks kirjeldused

```
static struct S { int a,b; };
struct S x,y;
```

siis võib arvata, nagu kuuluksid x ja y mäluklassi static. Tegelikult kandub eelmisest kirjeldusest üle ainult tüüp ja seega pole mäluklassil seal mõtet.

4.6. Deklaraatorid

Deklaraator on kirjelduse osa, mis esitab kirjeldatava nime ja võib täpsustada ka tema tüüpi. Märgime, et sarnast konstruktsiooni teistes programmeerimiskeeltes ei leidu.

```
deklaraator:
        lihtdeklaraator
        ( deklaraator )
        funktsiooni-deklaraator
        massiivi-deklaraator
        viida-deklaraator
```

Lihtdeklaraator määrab aritmeetilist, loendi-, struktuuri- või ühenditüüpi muutuja nime:

```
lihtdeklaraator:
        nimi
```

Kui T on mingi tüübikirjeldaja ja id identifikaator (nimi), siis kirjeldus

```
T id;
```

tähendab, et id on tüüpi T muutuja. Mõned näited:

```
int i;                  /* i on täisarvuline muutuja */
float velocity; /* velocity on reaalarvuline muutuja*/
struct S { int a; float b; } a_and_b;
        /* a_and_b on kahe komponendiga struktuur */
```

Lihtdeklaraatorit kasutatakse juhul, kui tüübikirjeldaja määrab kogu tüübiinfo. See on niiviisi aritmeetiliste tüüpide, struktuuride, loendite ja ühendite, samuti aga ka defineeritud tüüpide (typedef abil) ja void korral. Viidad, massiivid ja funktsioonid nõuavad keerukamat deklaraatorit, mis saadakse lihtdeklaraatorist spetsiaalsete operatsioonide rakendamise teel.

Viidadeklaraator määrab viidatüüpi muutuja:

```
viida-deklaraator:
        * deklaraator
```

Olgu D mingi deklaraator, mis sisaldab nime id. Kui kirjeldus T D; määrab, et id on tüüpi "... T", siis kirjeldus

```
T * D;
```

määrab, et id on tüüpi "... viit tüübile T". Mõned näited:

```
int x;      /* x on täisarvuline muutuja */
int * x;    /* x on viit täisarvule */
int * * x; /* x on viit täisarvu viidale */
int x[];    /* x on massiiv täisarvudest */
int * x[]; /* x on massiiv viitadest täisarvudele */
int x();    /* x on täisarvulise väärtusega funktsioon*/
int  *x(); /* x on funktsioon väärtusega viit
              täisarvule */
```

ANSI C laiendab viida mõistet, tuues sisse ka viida konstandile (vt. lk. 171).

Massiivideklaraator määrab massiivitüüpi objekti:

```
massiivi-deklaraator:
        deklaraator [ avaldis ]
```

Olgu D mingi deklaraator, mis sisaldab nime id. Kui kirjeldus T D; määrab, et id on tüüpi "... T", siis kirjeldus

T (D) [e] ;

määrab, et id on tüüpi "... massiiv tüüpi T elementidest". Mõned näited:

```
int x;          /* x on täisarvuline muutuja */
int (x) [];     /* x on massiiv täisarvudest */
int * x;        /* x on viit täisarvule */
int * x [];     /* x on massiiv viitadest täisarvudele */
int (* x) [];/* x on viit täisarvude massiivile */
int (x[])[]; /* x on massiiv täisarvude massiividest*/
```

Sulud võib siin sageli ära jätta, kui arvestada deklaraatorite konstrueerimise operatsioonide prioriteete.

Kui nurksulgudes esineb täisarvuline konstantavaldis e, siis määrab ta elementide arvu massiivis. Massiivi esimese elemendi indeks on alati 0, seega kirjeldus int A[3]; määrab massiivile elemendid A[0], A[1] ja A[2]. Elementide arv peab olema positiivne, ehkki mõni kompilaator seda ei kontrolli.

Mitmemõõtmelised massiivid, nagu juba näitest näha võis, defineeritakse kui massiivide massiivid. Massiivi elementide arvu (konstantavaldise) võib ära jätta järgmistel juhtudel.

1) Kui massiiv kirjeldatakse funktsiooni parameetrina.

2) Kui deklaraatorile järgneb algväärtustaja, mis niisugusel juhul määrab massiivi elementide arvu.

3) Kui tegemist pole massiivi definitsiooni vaid deklaratsiooniga, siis elementide arv määratakse kusagil mujal.

Samad reeglid kehtivad ka n-mõõtmelise massiivi esimese mõõtme kohta (viimase n-1 mõõtme rajad peavad olema antud).

```
static int vector[5]; /* raja peab olema */
char prompt[] = "Yes or No?";  /*  raja 11 määratakse
                                  algväärtustajast */
extern  matrix[][10];  /* deklaratsioon, kus esimene
                                  raja  võib puududa */
```

Funktsioonideklaraator määrab funktsioonitüüpi objekti:

```
funktsiooni-deklaraator:
        deklaraator ( parameetrid )
parameetrid:
        nimi
        parameetrid , nimi
```

Olgu D mingi deklaraator, mis sisaldab nime id. Kui kirjeldus T D; määrab, et id on tüüpi "... T", siis kirjeldus

```
T (D) ();
```

määrab, et id on tüüpi " ... tüüpi T väärtusega funktsioon". Mõned näited:

```
int x;      /* x on täisarvuline muutuja */
int (x)(); /* x on täisarvulise väärtusega funktsioon */
int * x;    /* x on viit täisarvule */
int * (x ()) /* x on funktsioon väärtusega viit
                täisarvule */
int (* x)(); /* x on viit täisarvulise väärtusega
                funktsioonile */
int * x[];  /* x on massiiv viitadest täisarvule */
int (* x[])(); /* x on massiiv viitadest täisarvuliste
                väärtustega funktsioonidele */
```

Sulud võib siin ära jätta ainult kahes esimeses näites.

Funktsioonideklaraator võib sulgudes sisaldada formaalsete parameetrite loetelu, kuid ainult funktsiooni definitsiooni (aga mitte deklaratsiooni) korral:

```
void f(x,y) /* kahe parameetriga funktsioon */
int x,y;
{ ... }
extern void g(); /* funktsiooni deklaratsioon, kus
                    parameetriloetelu peab puuduma */
int (*h)();      /* defineeritakse viit funktsioonile,
          mitte funktsioon, parameetriloetelu puudub */
```

Mõningad viimased realisatsioonid (ja ka ANSI C) toovad sisse funktsioonide nn. prototüübikirjeldused (vt. lk. 166), mis võimaldavad kompilaatoril teostada tüübikontrolli funktsioonide väljakutsumise korral.

Kõiki eelpool kirjeldatud deklaraatorite liike võib omavahel kombineerida, nagu seda juba näidetest näha oli. Kirjeldame veel ühe näitena 5-elemendilise massiivi viitadest täisarvulise väärtusega funktsioonidele:

```
int (* arr[5]) ();
```

Kuigi suvaline kombineerimine on süntaktiliselt lubatud, esitab keel C deklaraatoreile siiski mõningad kitsendused.

1) Tüübi void korral on lubatud ainult funktsioon väärtusega tüübist void (ANSI C lubab viitu tüübile void).

2) Pole lubatud massiivid funktsioonidest, kuid võivad esineda massiivid viitadest funktsioonidele.

3) Pole lubatud funktsioon, mille väärtuseks on massiiv, kuid väärtuseks võib olla viit massiivile.

4) Pole lubatud funktsioon, mille väärtuseks on funktsioon, kuid väärtuseks võib olla viit funktsioonile.

Deklaraatorite kombineerimisel on olulised nende prioriteedid - funktsioonide ja massiivide deklaraatorid on kõrgema prioriteediga kui viidadeklaraatorid. Seega * x() on samaväärne mis * (x()) ja tähendab "funktsioon viidatüüpi väärtusega", aga mitte "viit funktsioonile".

Väga keeruliste deklaraatorite moodustamine võib muuta kirjeldused ebaülevaatlikeks. Heaks programmeerimisstiiliks on siin mõningatele vaheetappidele omaette nime määramine kirjeldusega typedef. Nii näiteks tuleks kirjeldada muutujat x mitte kujul

```
int *(*(*(* x)()) [10]) ();
```

millest on otseselt peaaegu võimatu aru saada, vaid kujul

```
typedef int *(*function_ptr)();
typedef function_ptr (* routines)[10];
routines (* x) ();
```

Muutuja x on viit funktsioonile, mille väärtuseks on viit 10-elemendisele massiivile viitadest funktsioonidele millede väärtusteks on viidad täisarvudele.

Deklaraatorite tähendust aitab mõista järgmine lihtne reegel: kui rakendada deklaraatoris antud operatsioone avaldises esinevale nimele, siis saame tulemuseks tüübikirjeldajaga määratud tüüpi väärtuse. Seega näiteks kirjeldus

```
int *(* x) [4];
```

tähendab, et avaldise

```
*(* x) [i]
```

väärtus on tüüpi int. Siin rakendati nimele x kõigepealt viida järgi võtmist, sii elemendi valikut ja lõpuks jälle viida järgi võtmist - seega x on viit massiivile, mille elementideks on viidad täisarvudele.

4.7. Algväärtustajad

Iga muutuja deklaraatorile võib kirjelduses teatavasti (vt. lk. 36) järgneda algväärtustaja, millega määratakse ära väärtus, mis vastavale muutujale omistatakse tema eksisteerimisaja alguses:

algväärtus:
    avaldis
    { algväärtuste-loetelu ⌴ }
algväärtuste-loetelu:
    algväärtus
    algväärtuste-loetelu , algväärtus

Lõpukoma sulgudes ei muuda algväärtustamise tähendust.

Võimalik algväärtuse kuju sõltub muutuja tüübist ja mäluklassist ning sellest, kas on tegemist väliskirjeldusega või kirjeldusega bloki alguses (ehk sisekirjeldusega). Üldiselt peab iga staatilise ekstendiga muutuja algväärtus olema konstantavaldis, s.t. avaldis, mille väärtust on võimalik välja arvutada kompileerimisajal. Igale staatilise ekstendiga muutujale, millele pole antud algväärtust, omistatakse vaikimisi algväärtus null.

Automaatmuutujat võib algväärtustada suvalise avaldisega. Algväärtustamine toimub blokki sisenemisel, vastava koodi genereerib kompilaator. Kui automaatmuutujale pole antud algväärtust, omandab see juhusliku (defineerimata) algväärtuse. Funktsiooni formaalsetel parameetritel ei saa olla algväärtustajaid.

Skalaarsete muutujate algväärtustamise kuju on järgmine:

deklaraator = avaldis

Kui aritmeetiline muutuja kuulub staatilisse mäluklassi, siis peab avaldis olema konstantavaldis; automaat- ja registrimuutujate korral võib tegu olla suvalise avaldisega. Mõned kompilaatorid nõuavad, et staatiliste muutujate korral peab avaldis olema muutujaga sama tüüpi, mõnikord aga lubatakse suvalist aritmeetilist tüüpi avaldist; automaat- ja registrimuutujate korral võib alati tegu olla suvalist tüüpi avaldisega. Mõned näited:

```
static int Count = 4 * 200;
extern int getchar();
main()
{
int ch=getchar();
... }
void process(k)
double k;
{
static double epsilon = 1.0 e-6;
auto float factor = k * epsilon;
... }
```

Originaal-C lubab aritmeetiliste muutujate algväärtusi esitada ka loogelistes sulgudes, kuid need sulud on loogiliselt mittevajalikud ja seepärast tuleks neist hoiduda (mõned realisatsioonid seda ka ei luba).

Staatilist tüüpi viitade algväärtustamisel võib kasutada konstantavaldist, auto ja register tüüpi viitade algväärtustamisel aga suvalist avaldist, kusjuures avaldise väärtuseks peab olema sama tüüpi viit. Viidatüüpi konstantavaldise elemendid võivad olla järgmised.

1) Konstant 0, mida (erandina) võib kasutada suvalist tüüpi viidana tähenduses, et ei viidata millelegi reaalsele.

2) Funktsiooni nimi tüübiga funktsioon tüüpi T väärtusega, mis teisendatakse tüüpi viit tüüpi T väärtusega funktsioonile ja võib algväärtustada sama tüüpi viitu, näiteks:

```
extern int f();
static int (*fp)() = f;
```

3) Staatilise massiivi nimi tüübiga massiiv tüüpi T elementidest, mis teisendatakse tüüpi viit tüübile T ja võib algväärtustada sama tüüpi viitu, näiteks:

```
char array[100];
char *cp = array;
```

4) Operatsioon & juba defineeritud staatilise muutuja nime ees väljendamaks vastavat tüüpi viita, näiteks:

```
static short s;
short *sp = &s;
```

5) Operatsioon & rakendatuna staatilise massiivi elemendile, näiteks:

```
float Power[10];
float *fp = &Power[5];
```

6) Ilmutatud tüübiteisenduse abil viidana esitatud täisarv, näiteks:

```
long *PSW = (long *) 0xffffff0;
```

(mõned realisatsioonid sellist tüübiteisendust ei võimalda).

7) Sõne, mis teisendatakse tüüpi viit sümbolile ja mis võib algväärtustada sama tüüpi viita, näiteks:

```
char *message = "Type <CR> to begin ";
```

Lisaks võib variantide 3 - 7 elemente kasutada koos neile liidetud või lahutatud täisarvulise konstantavaldisega:

```
static short s;
short *sp = &s + 3, *msp = &s - 3;
```

Massiivi algväärtus esitatakse loogelistes sulgudes. Kui I_j (j=0,1, ... ,n-1) on tüüpi T objektide algväärtused, siis

$$\{ I_0, I_1, \ldots, I_{n-1} \}$$

on algväärtuseks n-elemendilisele massiivile tüüpi T elementidest. Element indeksiga j saab algväärtuse I_j, näiteks

```
int array[4] = { 0, 1, 2, 3 };
```

Sama reegli kohaselt saab algväärtustada ka mitmemõõtmelisi massiive või siis struktuurimassiive:

```
int arr3m[4][2][3] =
      { { {  0,  1,  2 }, {  3,  4,  5 } },
        { {  6,  7,  8 }, {  9, 10, 11 } },
        { { 12, 13, 14 }, { 15, 16, 17 } }
        { { 18, 19, 20 }, { 21, 22, 23 } } };
struct { int a; float b; } a[3] =
      { { 1, 2.5 } , { 2, 3.9 } , { 0, -4.0} };
```

Originaal-C ja ka enamik realisatsioone lubavad algväärtustada ainult staatilisi massiive. Üksikud realisatsioonid on sellest kitsendusest loobunud. ANSI C võimaldab küll algväärtustada automaatseid massiive, kuid ainult konstantsete algväärtustega.

Täiendusena kehtivad massiivide algväärtustamisel järgmised reeglid. Algväärtustajaid võib loetelus olla vähem kui

massiivis elemente, sel juhul ülejäänud elemendid saavad algväärtuse 0. Kui massiivil leidub algväärtus, siis võib tema raja jätta andmata: massiivi võetakse nii palju elemente nagu on antud algväärtusi. Sõnedega võib algväärtustada sümbolimassiive: massiivi esimese elemendi algväärtuseks saab sel juhul sõne esimene sümbol, teise elemendi algväärtuseks teine sümbol jne. Et sõne lõppu lisatakse alati sümbol '\0', peab massiiv olema selle sümboli võrra pikem. Kui algväärtusi on rohkem kui massiivis elemente, siis annab enamik kompilaatoreid vea. Näiteid:

```
int arr[5]={ 1, 2, 3 }; /* on samaväärne järgnevaga */
int arr[5]={ 1, 2, 3, 0, 0 };
int squares[] = { 0, 1, 4, 9 } /* raja on 4 */
char x[5] = "ABCD"; /* on samaväärne järgnevaga */
char x[5] = { 'A', 'B', 'C', 'D', '\0' };
char str[] = "ABCDEF"; /* raja on 7 */
```

Loenditüüpi muutujate algväärtustajas peab avaldis olema vastavat loenditüüpi. Mõned kompilaatorid lubavad siin ka avaldist loogelistes sulgudes.

Staatilise muutuja algväärtus peab tingimata olema konstantavaldis, s.t. vastavat loenditüüpi konstant; automaatja registrimuutujaid võib algväärtustada suvalise seda tüüpi avaldisega (reaalselt siiski ainult kas selle tüübi konstant või muutuja). Näiteid:

```
static enum E { a, b, c } x = a;
auto enum E y = x;
```

Mõned realisatsioonid ei erista loenditüüpi täisarvutüübist, s.t. võimaldavad algväärtustada loenditüüpi muutujaid täisarvuliste avaldistega. Kuigi see võib olla lubatud, tuleb seda pidada halvaks stiiliks, äärmisel juhul võiks kasutada ilmutatud tüübiteisendust.

Struktuuri algväärtus tuleb esitada loogelistes sulgudes. Koosnegu näiteks struktuuritüüp T kokku n komponendist vastavalt tüüpi T_j (j=1,2,...,n) ja olgu I_j algväärtus tüüpi T_j objektile. Siis

$$\{ I_1, I_2, \ldots, I_n \}$$

on algväärtuseks objektile tüüpi T.

Staatilisi struktuure on lubatud algväärtustada, kusjuures komponendi algväärtuse lubatav kuju on sama, mis seda tüüpi muutuja algväärtusel. Mõningad realisatsioonid ei luba algväärtustada struktuuri koosseisus olevaid bitivälju. Automaatseid (või ka registri) struktuure üldreeglina algväärtustada ei ole lubatud (ANSI C lubab algväärtustada automaatseid struktuure, kuid ainult konstantsete algväärtustega). Näide struktuuri algväärtustamisest:

```
struct S { int a; char b[5]; double c; };
struct S x = { 1, "abcd", 45.0 };
```

Kui algväärtusi struktuuri kõigi komponentide jaoks ei jätku, siis ülejäänutele omistatakse algväärtus null (samuti nagu massiivide korral). Liiga palju algväärtusi on viga.

Ühendite algväärtustamine. Originaal-C ja paljud realisatsioonid ei võimalda algväärtustada ühendeid, eelkõige sel ilmsel põhjusel, et ei ole teada millisele ühendi komponendile algväärtus omistatakse. Mõningad realisatsioonid, mis algväärtustamist võimaldavad, algväärtustavad ühendi esimese komponendi (siin võib esineda kitsendusi, näiteks lubatakse algväärtustada ainult staatilisi ühendeid). Toome ühe näite:

```
enum Greek { alpha, beeta, gamma };
union U { struct { enum Greek tag; int size; } I;
          struct { enum Greek tag; float size; } F; };
static union U x = { alpha, 42 };
```

Massiivide ja struktuuride algväärtustamisel võib ära jätta osa sisemisi sulgusid (ehkki sulgudega on programm ülevaatlikum). Siin kehtivad järgmised reeglid.

1) Ära ei või jätta kõige välimist sulupaari.

2) Kui algväärtuste loetelu sisaldab korrektse arvu algväärtusi, siis võib sisemised sulupaarid ära jätta.

Sisemiste sulgudeta algväärtuste loeteluga liitobjekti algväärtustamisel võetakse loetelust vajalik arv algväärtusi esimese komponendi jaoks, järgnevad algväärtused initsialiseerivad teist komponenti jne. Näiteks:

```
int matrix [2][3] = { 1, 2, 3, 4, 5, 6 };
/* see on samaväärne kui {{1, 2, 3},{4, 5, 6}} */
```

Ülearuseid sulgusid algväärtuses tavaliselt veaks ei peeta.

4.8. Kontekstuaalsed kirjeldused ja välisnimed

Ainsa erandina nimede otsese kirjeldamise üldisest reeglist loetakse avaldises esinev kirjeldamata nimi, millele järgneb avav sulg (kontekstuaalselt kirjeldatuks kui tüüpi int väärtusega funktsioon. Seega kui f on kirjeldamata, siis

```
{   ... f(i,j) ...   }
```

eeldab vaikimisi nagu oleks programmis leidunud kirjeldus

```
extern int f();
```

Peab aga märkima, et funktsioonide kirjeldamatajätmine on halb praktika. Nimelt kiputakse unustama, et see kehtib ainult tüüpi int väärtusega funktsioonide kohta. Kui aga jätta defineerimata mingit muud tüüpi väärtusega funktsioon, on tulemuseks tõenäoliselt viga programmi täitmisel, näiteks

```
{
double x,y;
...
x = sqrt(y); /* tulemus on ettearvamatu, kuna */
... }   /* sqrt väärtust tõlgendatakse kui täisarvu */
```

Vaatleme lõpuks mõningaid välisnimedega seotud probleeme. Välisnimi on nimi, mis erinevates lähtefailides peab tähistama ühte ning sama objekti (kusjuures seosed loob linker). Tekib aga küsimus, millises lähtefailis asub see objekt tegelikult ja mis siis, kui ta ühes lähtefailis saab ühe algväärtuse, teises aga teise? Ilmselt võib definitsioone (milles objekt tegelikult luuakse) kogu programmis olla ainult üks, deklaratsioone (mis seovad nime kuskil mujal defineeritud objektiga) aga palju. Mida siis lugeda definitsiooniks? Kahjuks lähenevad erinevad realisatsioonid asjale erinevalt, nii et põhiliselt esineb neli meetodit.

1) Algväärtustajaga väliskirjeldus loetakse definitsiooniks. Antud nimega võib algväärtuse siduda vaid ühes kohas.

2) Kõikides deklaratsioonides peab olema näidatud mäluklass extern: ilma selleta loetakse väliskirjeldus definitsiooniks. Definitsioon võib, kuid ei pea sisaldama algväärtustajat. Algväärtus ja extern välistavad teineteist. Seda kõige levinumat varianti kasutavad ka originaal-C ja ANSI C.

3) Ühisvälja kasutamisel ei loeta sõltumata mäluklassist extern ühtegi kirjeldust otseselt definitsiooniks. Algväärtus võib esineda vaid üks kord mäluklassist extern sõltumata. Linker eraldab välisobjektidele mälu ühisväljana, andes sama nime jaoks mälu üksainus kord. See on programmeerijale kõige valutum, kuid süsteemile enam nõudeid esitav meetod.

4) Enamik UNIX-realisatsioone kasutab järgmist meetodit:

a) Kui puudub extern aga leidub algväärtustaja, siis on tegemist definitsiooniga. Kui niisuguseid definitsioone on mitu, siis annab linker vea.

b) Kui puudub extern ja algväärtustaja, siis kasutatakse ühisväljameetodit. Neid kirjeldusi võib olla kuitahes palju.

c) Kui leidub extern, siis on tegemist deklaratsiooniga (algväärtustajat ei tohi olla). Kui seda kirjeldust programmis ei kasutata, siis linkeri jaoks välisviita ei looda.

Võimalikult paljude realisatsioonidega kooskõla saavutamiseks võib programmeerija ühes kohas programmis anda välisnime ilma mäluklassita extern ja algväärtustajaga (ka algväärtuse 0 korral). Kõigis ülejäänud kohtades anda nimi ilma algväärtustajata, kasutades mäluklassi extern.

Loomulikult peavad kõik ühe ja sama välisnime kirjeldused määrama sama tüüpi objekti. Kui see nii ei ole, siis sellist viga ei ole kompilaator eri failide korral võimeline avastama (vea avastab linker või veelgi tõenäolisemalt selgub see täitmisel, kus teda on raske lokaliseerida).

Enamik uuemaid realisatsioone ignoreerib välisnimede kirjeldusi, kui programmis selle välisnime poole kordagi ei pöörduta. Näiteks kui programmis on kirjeldus

```
extern double fft();
```

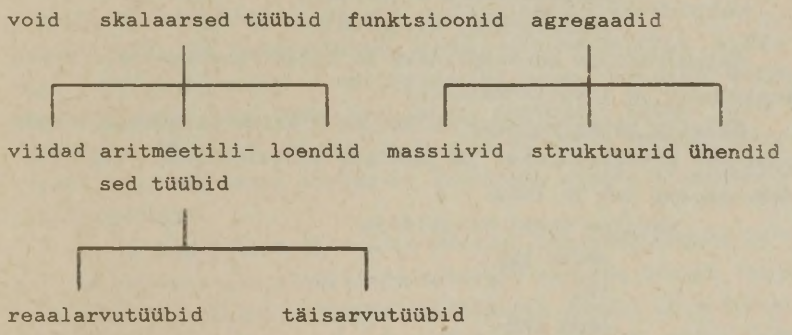
kuid pole ühtegi pöördumist funktsiooni fft poole, siis vastavat välisviita ei looda ja komplekteerimisel funktsiooni fft (mis võib asuda teegis) juurde ei lisata. See võimaldab koguda teegifunktsioonide kirjeldused ühte päisfaili, mida võib direktiiviga #include lisada kõigile programmidele: juurde võetakse ainult tegelikult vajalikud funktsioonid. Peab aga märkima, et mõned realisatsioonid tekitavad välisviida iga programmis esineva välisnime jaoks.

5. TÜÜBID KEELES C

Tüüp tähendab programmeerimiskeeles mingit väärtuste hulka koos nende väärtustega sooritatavate operatsioonide hulgaga. Näiteks täisarvutüüp koosneb täisarvulistest väärtustest mingis vahemikus ja operatsioonidest nende väärtustega nagu liitmine, lahutamine, korrutamine jne. Reaalarvutüüp erineb täisarvutüübist nii võimalike väärtuste kui ka operatsioonide poolest, liitmine on siin näiteks ujupunktesituses arvude liitmine.

Me ütleme, et muutuja või avaldis "on tüüpi T", kui tema väärtus kuulub tüübi T väärtuste hulka. Muutujate tüübid määratakse muutujate kirjeldustes. Avaldiste tüübid tulenevad operatsioonide definitsioonidest.

Keeles C on laialdane valik tüüpe, nagu mitmesuguse esitusega täisarvud, reaalarvud, viidad, loendid, struktuurid, ühendid ja funktsioonid. Lisaks on kasutusel spetsiaalne tüüp void, millel pole ühtegi väärtust. Seda tüüpi kasutatakse niisuguste funktsioonide kirjeldamisel, mis ei väljasta väärtust. Kokkuvõtlikult esitame keele C võimalikud tüübid järgmise tabelina:



Järgnevas vaatleme üksikasjalikult kõiki neid tüüpe. Iga tüübi jaoks määrame, kuidas vastavat tüüpi objekti kirjeldada, milline on tüübi väärtuste hulk, millised on kitsendused ja vastavat tüüpi objektidega lubatud operatsioonid.

5.1. Täisarvutüübid

Keeles C on kasutusel laiem valik täisarvutüüpe ja nendega sooritatavaid operatsioone kui enamikus teistes programmeerimiskeeltes. See on tingitud asjaolust, et C kajastab võimalikult vahetult enamikus arvutites esinevaid sõnapikkusi ja aritmeetilisi operatsioone. Täisarvutüübid esindavad keeles C järgmisi objekte.

1) Märgiga või märgita täisarvulised väärtused koos tavaliste aritmeetiliste ja võrdlusoperatsioonidega.

2) Bitivektorid koos bitikaupa loogiliste operatsioonidega AND, OR ja XOR, bitiinversiooniga (bitikaupa eitusega) ning nihutamistega vasakule ja paremale.

3) Tõeväärtused koos loogilise liitmise, loogilise korrutamise ja eitusega. Väärtust 0 tõlgendatakse kui FALSE ja muid väärtusi kui TRUE (1 on TRUE "kanooniline" väärtus).

4) Sümbolid, mis on esitatud nende koodidena.

Tavaliselt jagatakse täisarvutüübid kolme klassi: märgiga täisarvud, märgita täisarvud ja sümbolid:

täisarvutüübi-kirjeldaja:
märgiga-tüübi-kirjeldaja
märgita-tüübi-kirjeldaja
sümbolitüübi-kirjeldaja

Täisarvutüüpi konstantideks on täisarvkonstandid, pikad konstandid ja sümbolkonstandid.

Märgiga täisarvutüüpe on keeles C kolme pikkusega, mida pikkuste mittekahanevas järjekorras kirjeldatakse võtmesõnadega short, int ja long:

märgiga-tüübi-kirjeldaja:
short int
int
long int

Kirjeldaja short int on samaväärne kirjeldajaga short ja long int samaväärne kirjeldajaga long. Mõned näited:

```
auto short i, j;
long int l;
static int k;
```

Need kolm tüüpi ei pea aga igas realisatsioonis tingimata esitama erineva pikkusega täisarve - pikkused sõltuvad konkreetsest arvutist. Üldnõue on siiski, et short ei oleks kunagi pikem kui int ja int pikem kui long.

ANSI C kasutab märgiga täisarvutüüpide määramiseks spetsiaalset võtmesõna signed (vt. lk. 170).

Märgiga täisarvude sisemine esitus sõltub arvutitüübist. Paljudel arvutitel on sümbol esitatud 8 bitiga, lühikesed täisarvud 16 bitiga ja pikad täisarvud 32 bitiga, kusjuures arvutisõna võib olla nii 16 kui 32 bitti. Sellest tulenevalt on ka märgiga täisarvudel keeles C analoogiline esitus (kuigi see keelest endast ei tulene ega pruugi nii olla kõigi realisatsioonide korral).

1) Tüüp short esitub tavaliselt vähemalt 16 bitil. Kui niisugusest vahemikust piisab, siis võib short leida kasutamist suurte täisarvumassiivide säilitamisel. Et aga aritmeetilistes operatsioonides iga tüüpi short väärtus teisendatakse automaatselt tüüpi int, siis pole erilist mõtet lihtmuutujail tüüpi short int.

2) Tüüp long esitub tavaliselt vähemalt 32 bitil, andes täisarvude suurima ulatuse. Mõnedes realisatsioonides (näiteks 16-bitistele mikroprotsessoritele) on operatsioonid pikkade täisarvudega tunduvalt aeglasemad kui tavalistega.

3) Tüübi int esitus vastab tavaliselt arvutisõna pikkusele. Sellega seoses on operatsioonid nende arvudega kõige efektiivsemad, kuid tüübi int esitus varieerub laias ulatuses, olles 32 bitti ühtedes ja ainult 16 bitti teistes arvutites, mis toob kaasa mõningaid probleeme seoses programmide ülekantavusega.

Arvude kujutamise diapasoon märgiga täisarvutüüpide korral ei sõltu mitte ainult kasutatavate bittide arvust vaid ka kodeerimisviisist. Näiteks täiendkoodi korral on n-bitiste arvude kujutamise diapasoon -2^{n-1} kuni $2^{n-1}-1$, pöördkoodi korral aga on alumine piir $-(2^{n-1}-1)$.

Et tagada programmide maksimaalne masinsõltumatus võib programmeerija ise defineerida talle sobivad tüübinimed vastavalt sisulistele vajadustele, näiteks:

```
typedef  short  part_number;
typedef  int    order_quantity;
typedef  long   purchase_order;
...
purchase_order back_order(part, number)
    part_number part;
    order_quantity number;
   { ... }
```

Nüüd võib arvutilt arvutile ülekandmisel jätkuda ainult vastavate tüübidefinitsioonide muutmisest, jättes programmi enda muutmatuks. Lisaks suureneb nii ka programmi loetavus.

Märgita täisarvutüüp hõlmab väärtusi vahemikust 0 kuni 2^n-1, kus n on selle tüübi sisemise esituse pikkus bittides. Originaal-C ja ka paljud teised realisatsioonid võimaldavad ainult ühe märgita täisarvutüübi unsigned ehk unsigned int. Viimasel ajal aga antakse keeles täielik vastavus märgita ja märgiga täisarvutüüpide vahel, seega:

märgita-tüübi-kirjeldaja:

unsigned short int

unsigned int

unsigned long int

Aritmeetilised tehted märgita täisarvudega teostatakse mooduli 2^n järgi. Näiteks suurimale võimalikule arvule ühe liitmine annab tulemuseks arvu 0. Kui tehtes osalevad märgiga ja märgita täisarvud, siis märgiga täisarvud teisendatakse märgita täisarvudeks. See teisendus võib kaasa tuua üllatusi. Näiteks teades, et märgita täisarvud on positiivsed, võidakse arvata, et alati on tõene järgmine tingimus:

```
unsigned int u;
...
if ( u > -1) ...
```

Tegelikult on see praktiliselt kõigi realisatsioonide korral just väär, sest arvu -1 teisendamine märgita täisarvuks annab tulemuseks suurima võimaliku märgita täisarvu.

Sümbolitüüp keeles C on täisarvuline tüüp, kus väärtusteks on sümbolite täisarvulised koodid ja neid võib vabalt kasutada täisarvulistes operatsioonides:

sümbolitüübi-kirjeldaja:

unsigned char

Originaal-C ja paljud realisatsioonid ei luba kasutada võtmesõna unsigned. Sel juhul sümbolite tõlgendamine märgiga või märgita täisarvudena sõltub realisatsioonist. Näiteid:

```
static char buffer[80];
char c;
char *bp = buffer;
```

Sõltumata sümbolite tõlgendamisest märgiga või märgita täisarvudena on garanteeritud, et keele C põhitähestiku kõigi sümbolite väärtused on positiivsed (ülejäänud sümbolite väärtused võivad olla kas positiivsed või negatiivsed). Järgmine pisiprogramm teeb kindlaks, millise sümbolitüübiga on 8-bitise sümboli korral tegemist - kas märgiga, märgita või "pseudo-märgita" sümbolitega (viimane juht tähendab, et kõigil sümboleil on mittenegatiivsed väärtused, kuid teisendusreeglid vastavad märgiga täisarvule):

```
int main()
{
char c = 255;
if( c/-1 == 1) printf("Märgiga sümbolid\n");
else if (c/-1 == 0) printf("Märgita sümbolid\n");
else if(c/-1==-255) printf("Pseudo-märgita sümbolid\n");
else printf("??? Midagi muud c/-1 = %d\n",c/-1);
}
```

Sümboli märgiga või märgita esitus on tähtis ka paljude teegifunktsioonide korral. Näiteks failist sümboli lugemise funktsioon getchar annab faili lõputunnuseks -1 (tavaliselt määratakse see kui EOF). Vaatleme järgmist programmi:

```
#define EOF -1
extern int getchar();
extern void putchar();
void copy_char()
   {
   char ch;     /* vigane! */
   while((ch = getchar()) != EOF) putchar(ch);
   }
```

See programm kahjuks ei tööta siis, kui sümbolid on märgita või pseudo-märgita. Selgituseks eeldame et sümboli pikkus on 8 bitti, täisarvu pikkus aga 16 bitti. Seepärast omistamisel ch=getchar(); omistatakse täisarv sümbolile ja kui täisarvuks on -1, siis saab muutuja ch tavaliselt väärtuse 255. Edasi võrreldakse antud väärtust suurusega -1. Märgita või pseudo-märgita sümboli korral ei saada siin võrdust kunagi ja programm jääb lõpmatusse tsüklisse. Korrektseks lahenduseks on muutuja ch defineerimine kui int, mitte kui char.

Teiseks sümbolitega seotud probleemiks on nende sisemise esituse pikkus. Ehkki 8-bitine sümbol on levinuim, kasutavad mõned arvutid siiski ka 7- ja 9-bitiseid sümboleid. Seda tuleb arvestada siis, kui kasutatakse sümboleid "väga lühikeste" täisarvudena, et pikkades massiivides mälu kokku hoida.

5.2. Reaalarvutüübid

Keeles C on võimalik kasutada kaht tüüpi reaalarve ehk ujupunktarve, nimetame neid tavaliseks ja topelttäpsusega esituseks, tüübinimedega vastavalt float ja double:

reaalarvutüübi-kirjeldaja: variandid

float double

Varasemais realisatsioonides on double sünonüümina kasutusel long float, kuid viimastes realisatsioonides (ka ANSI C korral) on see võimalus kõrvale jäetud. Toome mõned näited:

```
double d;
static double pi;
float coefficients[10];
```

Kuna kõik tüüpi float väärtused teisendatakse aritmeetilistes operatsioonides väärtusteks tüüpi double, siis tüüpi float objektidel on mõtet ainult massiividena ja struktuuride koosseisus mälu kokkuhoidmiseks. Siiski mõned uuemad realisatsioonid ja ANSI C kasutavad ka float-aritmeetikat.

Keel C ei määra reaalarvutüüpide konkreetseid esitusi - need sõltuvad arvutist (mõnes arvutis võivad sisemised esitused ka kokku langeda). Mõistlik on eeldada, et tüübi float väärtused moodustavad tüübi double väärtuste alamhulga.

Reaalarvutüüpi operandidega on võimalikud kõik tavalised aritmeetilised operatsioonid: liitmine, lahutamine, korrutamine, jagamine, aga samuti võrdlemis- ja loogilised operatsioonid ning teisendused suvalisse aritmeetilisse tüüpi.

Keeles puuduvad kokkulepped täis- ja reaalarvuliste tüüpide omavahelise vahekorra kohta, s.t. me ei tarvitse olla kindlad, et teisendades näiteks tüüpi double objekti tüüpi long ja seejärel tagasi tüüpi double, saame sama väärtuse.

Reaalarvutüüpi konstante esindavad keeles C reaalarvkonstandid, mille tüübiks loetakse double. ANSI C toob sisse andmetüübi long double (tüübi long float asemel), mis on potentsiaalselt veel suurema ulatuse ja täpsusega kui double.

5.3. Viidatüübid

Iga tüübi T jaoks peale tüübi void võib keeles C formeerida tüübi viit tüübile T, mille väärtused osutavad tüübi T väärtustele (võib öelda, et viidatüübid väljendavad aadresse). Näiteks, et defineerida ip kui viit tüübile int ja cp kui viit tüübile char tuleb kirjutada

```
int *ip;
char *cp;
```

Viidad on C-programmides väga intensiivselt kasutusel nii keele C algselt süsteemprogrammeerimisele orienteeritud iseloomu kui ka viitade ja massiivide ühtse kasutamise tõttu. Viitadega on seotud kaks operatsiooni - aadressi võtmine & ja sisu võtmine viida järgi * (vt. lk. 106-107). Näiteks:

```
int i, j, *ip;
ip = &i;    /* ip viitab nüüd muutujale i */
i = 22;
j = *ip;    /* muutuja j väärtuseks on 22 */
*ip = 17;   /* muutuja i väärtus on 17 */
```

Igal viidatüübil on spetsiaalväärtus "viit eimillelegi", mida väljendatakse täisarvkonstandina 0 ja kirjeldatakse tavaliselt makrona NULL. Et väärtus 0 tähendab ka tõeväärtust FALSE, saab lihtsalt kontrollida, kas viit viitab millelegi:

```
if (ip) i = *ip;
```

Viidaaritmeetika olemasolu on üheks keele C kasulikuks omaduseks. Kui p on avaldis tüüpi viit tüübile T ning i on täisarvuline avaldis, siis p + i ja p - i on sama viidatüüpi, kusjuures viidatakse esialgsest objektist i objekti taha- või ettepoole. Arvutiaadresside terminites tuleb uue viidatava objekti aadressi saamiseks korrutada täisarvu i tüüpi T objekti pikkusega ja juurde liita aadressile p. Kui p ja q on kaks viidatüüpi avaldist tüüpi T objektidele, siis vahe p - q on täisarvulist tüüpi ning väljendab nende kahe viida vahele jäävat tüüpi T objektide arvu.

Ülejäänud võimalikeks operatsioonideks viidatüüpi väärtustega on omistamine, võrdlusoperatsioonid (mida tuleb tõlgendada viidatavate objektide mälus paiknemise järgi), loogiline liitmine ja korrutamine ning teisendamine teistesse viidatüüpidesse või täisrvulistesse tüüpidesse.

Iga arvuti korral ei saa eeldada, et kõikidel viitadel (viitadel suvalisele tüübile) oleks sama sisemine esitus. Kui baitadresseerimisega arvuteis on tavaline, et suvalise baidi aadressi saab kujutada arvutisõnana, siis sõnaadresseerimisega arvuteis võib sõna 0-nda baidi ja ülejäänud baitide aadresside kujutamine olla erinev. Teiseks tuleb arvestada, et kindlad andmetüübid võivad operatsioonides nõuda teatavat rajastamist kas siis näiteks sõna või topeltsõna piirile. Seega võivad siin tekkida järgmised raskused.

1) Viida teisendamine teist tüüpi viidaks võib kaasa tuua tema esituse muutuse.

2) Eri tüüpi viitade võrdlemine ei pruugi taanduda lihtsaks aritmeetiliseks võrdlemiseks.

3) Viitade ja täisarvude omavaheline teisendamine võib kaasa tuua etteaimamatuid resultaate.

Eriti mikroprotsessorite korral võib kasutusele tulla pika ja lühikese aadressi mõiste. Nende esitamiseks toovad mõned realisatsioonid viitadega seoses sisse võtmesõnad near ja far, mis väljendavad lühikest ja pikka aadressi.

Sõltumata viida jaoks valitud esitusest peab olema täidetud teatud hulk tingimusi, millest C-programmid oluliselt sõltuvad. Esiteks, kui rajastamisnõue tüübi T jaoks on tuge-

vam kui tüübi S jaoks, siis viita tüübile T saab teisendada viidaks tüübile S ja tagasi ilma infokaota. Teiseks, viit sümbolile on nõrgima rajastamisnõudega, s.t. suvalist viita saab teisendada viidaks sümbolile ja tagasi ilma infokaota. Mõned kompilaatorid loevad veaks viida teisendamise teist tüüpi viidaks juhul, kui see võib kaasa tuua rajastamisvea.

5.4. Massiivid

Kui T on suvaline tüüp peale funktsiooni, siis saab kirjeldada massiivi tüüpi T elementidest. Kõigi massiivide indeksite alumine raja keeles C on 0, seega kui massiiv on defineeritud kirjeldusega int M[4]; siis kuuluvad massiivi elemendid M[0], M[1], M[2] ja M[3]. Massiivi pikkus on alati võrdne tema elementide pikkuste summaga.

Keeles C on andmetüüpide "viit tüübile T" ja "massiiv tüüpi T elementidest" vahel tihe seos. Esiteks, kui massiivi nimi esineb avaldises, siis ta teisendatakse viidaks selle massiivi esimesele elemendile (indeksiga 0):

```
int a[10], *ip;
ip = a;
```

Nüüd viitab ip massiivi esimesele elemendile, seega omistamine ip = a; kus a on massiiv, osutub samaväärseks omistamisega ip = &a[0]; Ainukeseks erandiks sellise teisenduse juures on operatsioon sizecf (vt. lk. 104), mis rakendatuna massiivi nimele annab kogu massiivi pikkuse, mitte massiivi esimesele elemendile osutava viida pikkuse.

Teiseks on massiivist elemendi valik väljendatav viidaaritmeetika terminites. Nimelt massiivi a i-nda elemendi valiku operatsioon a[i] on keeles defineeritud samaväärsena avaldisega *(a+i) ehk eelmise märkuse põhjal avaldisega *(&a[0] + i). Sellepärast võib ka viitadele rakendada massiivi elemendi valiku operatsiooni (muidugi peab programmeerija hoolitsema selle eest, et viit ikka tegelikult osutaks mingite objektide massiivile):

```
double d, *dp;
d = dp[4];
```

Mitmemõõtmelised massiivid kirjeldatatakse kui massiivid massiividest. Massiivi mõõtmete arvul ei ole sealjuures piirangut. Massiivid paigutatakse arvuti mällu nii, et kõige kiiremini muutub viimane indeks. Näiteks kirjeldus

```
int t[2][3];
```

määrab kaheelemendilise massiivi kolmeelemendilistest täisarvulistest massiividest, mille elemendid paiknevad arvuti mälus (aadresside suurenemise järjekorras) t[0][0], t[0][1], t[0][2], t[1][0], t[1][1], t[1][2]. Vahekord massiivide ja viitade vahel jääb kehtima ka mitmemõõtmeliste massiivide korral. Näiteks äsjakirjeldatud massiivi t korral on avaldis t[1][2] samaväärne avaldisega *(*(t+1)+2). Siin t kui kaheelemendilise massiivi nimi teisendub viidaks kolmeelemendilisele massiivile. Liites täisarvu 1 saame viida järgmisele seilisele massiivile (ehk kahemõõtmelise massiivi järgmisele reale). Viida järgi sisu võtmise operatsioon annab selle massiivi enda, mis omakorda kui andmetüüp massiiv teisendatakse viidaks tema esimesele elemendile. Liites juurde konstandi 2 saame viida selle elemendile indeksiga 2 ja lõpuks viida järgi sisu võtmine annab meile vastava täisarvu.

Massiivi defineerimisel peab massiivi elementide arv olema teada. Tegelikult elemendi valiku operatsioonis ei kontrollita elemendi kuulumist massiivi. Seega võib kirjeldamisel massiivi ainukese või vasakpoolseima raja ära jätta juhul, kui pole tegemist massiivi defineerimisega vaid deklareerimisega ja samuti juhul, kui on tegemist funktsiooni formaalse parameetriga (raja võib ära jätta ka siis, kui ta määratakse algväärtustajast). Näiteks järgmises funktsioonis sum antakse välismassiivi a raja ette parameetrina:

```
extern int a[];
int sum(n)
int n;
{
int i, s=0;
for (i = 0; i < n; ++i) s += a[i];
return s;
}
```

Küllaltki levinud on funktsiooni argumendiks oleva massiivi raja üleandmine teise parameetrina:

```
int sumarray(array,arraylen)
int array[],arraylen;
{ ... }
```

Selles näites on korrektsem defineerida array mitte massiivi vaid viidana, sest massiivi nimi avaldises teiseneb viidaks ja seega antakse ka funktsiooni väljakutsumisel üle viit.

Nagu juba öeldud, on sizeof ehk massiivi pikkuse määramine ainukeseks operatsiooniks massiiviga tervikuna (pikkus määratakse baitides, mitte massiivi elementides). Kõigis ülejäänud kontekstides vaadeldakse massiivi kui viita. Seejuures on korrektne tõlgendada seda konstantse viidana, sest massiivile kui tervikule pole võimalik omistada.

5.5. Loenditüübid

Loenditüübid on sisse toodud uuematesse realisatsioonidesse teiste programmeerimiskeelte nagu Pascal ja Ada vastavate tüüpide analoogidena. Originaal-C loenditüüpe ei toeta. Kahjuks aga ei käsitle loenditüüpe aktsepteerivad C-kompilaatorid neid alati ühtmoodi. Igal juhul määrab loenditüüp teatava täisarvude hulga, mis kirjelduses esitatakse identifikaatorite - loendikonstantide abil. Näiteks kirjeldus

```
enum kala{ haug, luts, kilu, lest } minukala,sinukala;
```

määrab uue loenditüübi nimega enum kala, mille konstantideks (võimalikeks väärtusteks) on haug, luts, kilu ja lest. Samuti määratakse kaks seda tüüpi muutujat minukala ja sinukala, millele saab omistada väärtusi loendikonstantide hulgast:

```
minukala = kilu;
sinukala = lest;
```

Loenditüüpi muutujale saab omistada ka teise sama loenditüüpi muutuja väärtuse ning võrrelda omavahel sama tüüpi muutujaid või muutujaid loendikonstantidega.

Loenditüüpide realiseerimisel seatakse loendikonstandile vastavusse mingi täisarv. Need täisarvud valitakse vaikimisi automaatselt, kuid valitavad arvud võib ka ette anda:

```
enum kala { haug=1, luts=7, kilu=4, lest=0 };
```

Anname nüüd loenditüübi kirjeldamise täieliku süntaksi:

```
loenditüübi-kirjeldaja:
        loenditüübi-definitsioon
        loenditüübi-viide
loenditüübi-definitsioon:
        enum nimi { konstandiloetelu }
loenditüübi-viide:
        enum nimi
konstandiloetelu:
        konstandi-kirjeldus
        konstandiloetelu , konstandi-kirjeldus
konstandi-kirjeldus:
        loendikonstant
        loendikonstant = avaldis
loendikonstant:
        nimi
```

Muutujad ja teised loenditüüpi objektid võib kirjeldada kas tüübi endaga samas kirjelduses või siis järgnevates kirjeldustes, kasutades loenditüübi viidet. Viimane on aga võimalik ainult siis kui loenditüübi kirjelduses on antud tüübi nimi. Näiteid:

```
enum color { red, blue, green } favorite;
enum color acceptable,non_accept;
enum { a, b, c, d } enumarray[10];
```

Loenditüübi nimi kuulub ühte nimeklassi struktuuri- ja ühenditüüpide nimedega. Tema skoop on sama, mis oleks sellel kohal kirjeldatud muutujal. Loendikonstandid kuuluvad samasse nimeklassi kui muutujanimed, funktsioonide nimed ja defineeritud tüübinimed. Ka nende skoop on sama mis sellel kohal defineeritud muutujal:

```
int a = 12;
{
enum { a, b }; /* uus a definitsioon varjab välise a */
float  b;  /* !!! see on vigane kirjeldus, sest nimi  b
              kordub samas blokis  */
... }
```

Tavaliselt on loenditüüpi muutujate sisemine esitus sama mis tüübil int, kuigi mõnikord võidakse kasutada ka esitust short või long. Loendikonstantidele antakse täisarvulised väärtused järgmiste reeglite kohaselt.

1) Kui konstandiga on ilmutatult seotud täisarvuline konstantavaldis, siis konstandile omistatakse selle avaldise väärtus. Sõltuvalt realisatsioonist võib avaldises olla lubatud ka juba eespool defineeritud loendikonstantide kasutamine, näiteks:

```
enum boys { Bill = 10, John = Bill+2, Fred = John+3 };
```

2) Kui ilmutatult teisiti pole määratud, siis esimesele loendikonstandile omistatakse väärtus 0.

3) Kui ilmutatult teisiti pole määratud, siis loetelus järgmisele konstandile omistatakse eelmise konstandi väärtusest ühe võrra suurem väärtus.

Ilmutatult võivad loendikonstantidele omistatavad väärtused olla ka negatiivsed, samuti ei ole keelatud rohkem kui ühele konstandile sama väärtuse omistamine, kuigi seda tuleb pidada halvaks stiiliks. Samuti loetakse halvaks stiiliks ka loenditüüpi muutujate ja konstantide kasutamist täisarvulistes avaldistes, kuigi mõned kompilaatorid seda lubavad.

5.6. Struktuuritüübid

Struktuuritüüp keeles C on suurel määral sarnane tüüpidega, mida mõningates teistes programmeerimiskeeltes nimetatakse kirjeteks (record): struktuur on kogum nimedega varustatud komponentidest, mis võivad üldiselt olla erinevat tüüpi. Struktuurid annavad programmeerijale teatava võimaluse abstraktsete andmetüüpide realiseerimiseks. Tüüpiline näide on kompleksarvu defineerimine:

```
struct complex { double real; /* reaalosa */
                 double imag; /* imaginaarosa */ };
struct complex x, y;
```

Nii defineeritud tüübi complex jaoks võib funktsioonidena sisse tuua kompleksarvu tekitamise ja operatsioonid kompleksarvudega (vaatame näitena korrutamist):

```
struct complex new_complex(r, i)
          double r, i;
{
struct complex new;
new.real = r; new.imag = i;
return new;
}
struct complex complex_mult(a,b)
  struct complex a, b;
{
  struct complex product;
  product.real = a.real * b.real - a.imag * b.imag;
  product.imag = a.real * b.imag + a.imag * b.real;
  return product;
}
```

Pärast neid sissejuhatavaid näiteid anname nüüd struktuuritüübi kirjeldamise täieliku süntaksi:

```
struktuuritüübi-kirjeldaja:
     struktuuritüübi-definitsioon
     struktuuritüübi-viide
struktuuritüübi-definitsioon:
     struct nimi { s-komponendid }
struktuuritüübi-viide:
     struct nimi
s-komponendid:
     s-komponendi-kirjeldus
     s-komponendid s-komponendi-kirjeldus
s-komponendi-kirjeldus:
     tüübi-kirjeldaja s-deklaraatori-loetelu ;
s-deklaraatori-loetelu:
     s-deklaraator
     s-deklaraatori-loetelu , s-deklaraator
s-deklaraator:
     deklaraator
     bitiväli
bitiväli:
     deklaraator : avaldis
```

Samuti kui loenditüüpide korral võib vastavat tüüpi objekte kirjeldada kas samas kirjelduses struktuuritüübi endaga või järgnevates kirjeldustes struktuuritüübi viite abil:

```
struct complex { double real, imag; } x, y;
struct complex *cp,z[100];
```

Paljudes realisatsioonides on lubatud kasutada struktuuritüüpide nn. osalist kirjeldamist. See toimub veel defineerimata struktuuri viite abil kahes kontekstis: viitade defineerimisel antud struktuurile ja tüübinimede (typedef-nimede) defineerimisel, kus konkreetse tüübi struktuur pole oluline. See võimaldab anda üksteisele viitavaid struktuure, millede sissetoomine oleks muidu kohmakas, näiteks:

```
struct P { ... struct Q *qp; ... };
struct Q { ... struct P *pp; ... };
```

Struktuuridega lubatavad operatsioonid on eri kompilaatorite korral mõnevõrra erinevad. Kõik realisatsioonid toetavad komponendi valiku operatsioone . ja ->. Enamik uuemaid realisatsioone võimaldavad struktuuri omistamist struktuurile, struktuuri funktsiooni tegeliku parameetrina ja funktsioone, mille väärtuseks on struktuur (originaal-C neid võimalusi ei luba). Otsesed võrdlusoperatsioonid struktuuride vahel pole lubatud. Aadressioperatsioon & on lubatud nii struktuuri enda kui ka tema komponentide suhtes (peale bitiväljade, mis ei ole arvutis vahetult adresseeritavad).

Struktuuri komponendid võivad olla suvalist tüüpi peale funktsiooni ja tüübi void. Struktuur ei või sisaldada komponendina iseennast, kuid võib sisaldada viita iseendale, näiteks on järgmine kirjeldus lubatav:

```
struct table { int a; struct table *next; };
```

Ühe struktuuritüübi komponentide nimed moodustavad omaette nimeklassi, s.t. nad peavad olema kõik omavahel erinevad, kuid võivad kokku langeda muutujate ning funktsioonide nimedega, defineeritud tüübinimedega ja loendikonstantidega. Näiteks järgmised kirjeldused on kõik lubatavad:

```
int x;
struct A { int x; double y; } y;
struct B { int y; double x; } z;
```

Muutuja x on selles näites korraga deklareeritud kolmes kontekstis: kui täisarvuline muutuja, struktuuri täisarvuline komponent ja struktuuri reaalarvuline komponent. Konflikti siin ei teki, kuna viitamine on igas kontekstis erinev, vastavalt x, y.x ja z.x. Märgime, et originaal-C määras kõikide struktuuritüüpide komponentide nimed ühte nimeklassi, kuid praegu seda kitsendust enam praktiliselt ei järgita.

Kui struktuuri komponendiks on omakorda struktuur, mis määrab ka struktuuritüübi nime, siis selle nime skoop on sama, mis välise struktuuritüübi nimel kui ta oleks (või on) esindatud. Nimede S ja T skoop järgmises näites on sama:

```
struct S { struct T { int a,b; } x; ... };
```

Elementide paigutuse kohta struktuuri sisemises esituses kehtib reegel, et enne defineeritud komponent asub eespool, s.t. kui p ja q on viidad struktuuri kahele komponendile, siis seos p < q kehtib parajasti siis, kui selle komponendi kirjeldus, millele viitab p on struktuuri kirjelduses antud eespool. Kuna struktuuri komponendid võivad olla eri tüüpi ja erisuguste rajastamisnõuetega, siis võib struktuuris komponentide vahele jääda kasutamata "auke", seega võib struktuuri pikkus olla suurem komponentide pikkuste summast.

Rajastamisnõudeid tuleb arvestada ka siis, kui struktuuridest moodustatakse keerukamaid objekte. Üldreeglina peab struktuuridest moodustatud massiivi pikkus võrduma struktuuri pikkuse ja massiivi elementide arvu korrutisega. Seetõttu paigutatakse struktuur alati mällu nii, et tema algusaadress rahuldab komponentide tugevaimat rajastamisnõuet ja pikkus on selle nõude kordne. Näiteks arvuteis, kus nõutakse pika reaalarvu rajastamist 8-le on järgmise struktuuri pikkus 24 baiti, sõltumata komponentide kirjeldamise järjekorrast:

```
struct S { double value; char name[10]; };
```

Keeles C saab struktuuri täisarvulisi komponente pakkida ka väiksemale või mittestandardse pikkusega mäluväljale. Selliseid komponente nimetame bitiväljadeks ja nende defineerimisel antakse komponendi nime järel koolon ja täisarvuline konstantavaldis, mis määrab välja pikkuse bittides:

```
struct S { unsigned a:4; unsigned b:5, c:7; };
```

Bitivälju kasutatakse masinsõltuvates programmides, mis võivad nõuda andmete esitamist konkreetsele riistvarale vastaval kujul. Nende programmide ülekandmisel ühest arvutist teise tuleb andmestruktuurid tõenäoliselt ümber defineerida.

Bitiväljade korral lubab originaal-C kasutada vaid tüüpi unsigned, kuid mõni kompilaator lubab ka tüüpi int, tõlgendades seda nagu sümboleid kas märgita, märgiga või pseudomärgita maneeris. Enamasti piirab kompilaator bitivälja maksimaalpikkuse tüübi int või unsigned pikkusega. Eri kompilaatorid võivad bitiväljadele mälu eraldada nii arvutisõna algusest kui ka lõpust alates. Kui järjekordne väli ei mahu enam antud sõnasse, siis paigutatakse ta järgmisesse.

Süntaks lubab ka nimeta bitiväljasid, mida võib kasutada väljade sobivamaks paigutamiseks, kuid millele programmis ei saa viidata. Eritähendus on nimeta väljal pikkusega 0, mis nõuab järgmise välja paigutamist järgmisesse sõnasse:

```
struct S { unsigned a:4, :2, b:5, :0, c:4; };
```

Bitiväljadel puudub aadressi ja viida mõiste ning neist ei saa moodustada massiive. Avaldistes tõlgendatakse bitivälja täisarvulise väärtusena. Üldiselt võtab bitiväljadele juurdepääs tavaliste muutujatega võrreldes rohkem aega.

5.7. Ühenditüübid

Ühenditüübi kirjeldamise süntaks on järgmine:

```
ühenditüübi-kirjeldaja:
        ühenditüübi-definitsioon
        ühenditüübi-viide
ühenditüübi-definitsioon:
        union nimi { u-komponendid }
ühenditüübi-viide:
        union nimi
u-komponendid:
        u-komponendi-kirjeldus
        u-komponendid u-komponendi-kirjeldus
u-komponendi-kirjeldus:
        tüübi-kirjeldaja deklaraatorid ;
```

Komponentide kirjeldamine langeb kokku struktuurikomponentide kirjeldamisega, kuid pole lubatud bitiväljad. Samuti nagu struktuuride puhul võib ka ühenditüübi viidet kasutada teistes kirjeldustes. Ühendi komponendid võivad olla suvalist tüüpi peale funktsiooni ja tüübi void. Ka ühend ei tohi sisaldada komponendina iseennast (küll aga viita iseendale). Nagu struktuuride korral moodustavad ka ühe ühenditüübi komponentide nimed omaette nimeklassi.

Ühendit võib vaadelda kui struktuuri, mille kõikide komponentide aadressid langevad kokku nii omavahel kui ka ühendi algusaadressiga, sõltumata sellest, mis tüüpi komponendiga on tegemist. Näiteks kirjelduse

```
union U { ... int C; ... } object, *p=&object;
```

toimel kehtivad järgmised võrdused:

```
(union U *) &(p->C) == p
            &(p->C) == (int *) p
```

Ühendi pikkuseks on tema pikima komponendi pikkus, millele võib olla lisatud teatav arv baite selleks, et rajastamistingimuste kohaselt oleks ühenditest moodustatud massiivi pikkus võrdne ühendi pikkuse ja massiivi elementide arvu korrutisega. Samuti peab ühend paiknema mälus nii, et oleks tagatud rajastamistingimuste täidetus kõige tugevama rajastamistingimusega komponendi jaoks. Näiteks kui tüüp double peab antud realisatsioonis olema rajastatud 8-le, siis on järgmise ühendi pikkus 16 baiti:

```
union U { double value; char name[10]; };
```

Ühenditüüp keeles C sarnaneb "variantidega kirjele" mõnedes teistes programmeerimiskeeltes, näiteks keeles Pascal. Samuti nagu struktuuril on ka ühendil teatud hulk nimelisi komponente, kuid tegelik väärtus saab korraga olla vaid ühel komponendil. Näiteks võib järgmiselt kirjeldatud ühendil u olla kas täis- või reaalarvuline väärtus, kuid ainult üks neist igal konkreetsel momendil:

```
union datum { int i; double d; } u;
```

Selle ühendi täisarvulisele väärtusele viidatakse kujul u.i ja reaalarvulisele väärtusele kujul u.d, kusjuures vastutus õige komponendi aktiivsuse eest lasub programmeerijal.

Ühendeid saab kasutada masinsõltumatul moel kui järgida mõningaid stiilireegleid. Näiteks tuleb pidada väga halvaks stiiliks väärtuse omistamist ühendi ühele komponendile ja järgnevalt teise komponendi kasutamist. Isegi kui see antud realisatsiooni ning arvuti korral ei osutu veaks, võib ta seda olla mingis teises realisatsioonis:

```
/* see on halva stiili näide !!!! */
union U { long c; double d; } x;
long l;
x.d = 1.0e10;
l  =  x.c; /* mõnel juhul võib see  meetod  reaalarvu
              mantissi eraldamiseks ju nutikas näida */
```

Üheks levinud meetodiks ühenditega korrektsel manipuleerimisel on lisada neile nn. tüübimuutuja, mille väärtus määrab momendil aktiivse komponendi. Vaatleme seda näite varal:

```
#define IVAL 1
#define FVAL 2
#define SVAL 3
union node { int count;
             double value;
             char name[20]; };
struct table { int tag;
               union node data; } tab[100];
...
if (tab[i].tag == IVAL) ...
else if(tab[i].tag == FVAL) ...
else if(tab[i].tag == SVAL) ...
```

Mõnes realisatsioonis ei lubata ühendeid algväärtustada, teistes saab anda esimesele komponendile algväärtuse.

5.8. Funktsioonid

Funktsioon on objekt, mis väljakutsumisprotsessis väljastab mingi kindlat tüüpi väärtuse. Funktsioon on keeles C ainukeseks alamprogrammi liigiks, kusjuures programm keeles C moodustubki funktsioonidest. Ütleme, et funktsioon on tüüpi T, kui tema poolt väljastatavad väärtused on tüüpi T.

Keeles C võivad funktsioonid olla suvalist tüüpi peale tüübi massiiv või funktsioon, s.t. funktsiooni väärtuseks ei tohi olla funktsioon ega massiiv.

Funktsioonitüüpi objekte saab programmis kirjeldada kahel viisil. Funktsiooni definitsioon tekitab funktsioonitüüpi objekti, defineerib tema formaalsed parameetrid, väärtuse tüübi ning esitab funktsiooni sisu (näiteks selle funktsiooni väärtuse leidmise eeskirja). Vaatleme näitena funktsiooni square, mis väljastab oma parameetri ruudu:

```
int square(x)
int x;
{
return x * x;
}
```

Funktsiooni deklaratsioon loob välisviida funktsioonile, mis on defineeritud kusagil mujal (näiteks teises lähtefailis). Deklaratsioonis teatatakse vaid funktsiooni tüüp, kuid ANSI C ja paljud viimased realisatsioonid võimaldavad määrata ka formaalsete parameetrite tüübid (vt. lk. 165):

```
extern int square();
```

Ainsaks operatsiooniks funktsioonidega, mis ei too kaasa automaatset tüübiteisendust tüübist funktsioon tüüpi viit funktsioonile, on funktsiooni poole pöördumine ehk funktsiooni väljakutse. Näiteks deklaratsioonid

```
extern int f(), (*fp)(), (*apf[])();
```

määravad täisarvulise funktsiooni f, viida fp täisarvulisele funktsioonile ja massiivi apf viitadest täisarvulistele funktsioonidele. Neid nimesid saab funktsiooni poole pöördumiseks kasutada näiteks järgmisel kujul:

```
int i;
i = f(14);
i = (*fp)(j,k);
i = (*apf[j])(k);
```

Funktsiooni poole pöördumisel ei kontrollita küll tegelike parameetrite arvu, kuid nende tüüpe võidakse kontrollida, kui funktsiooni deklaratsioonis on antud formaalsete parameetrite tüübid.

Funktsiooni nime kasutamisel avaldises mingis muus kontekstis peale funktsiooni poole pöördumise teisendatakse ta automaatselt tüübist "funktsioon, mis väljastab tüüpi T väärtuse" tüüpi "viit funktsioonile, mis väljastab tüüpi T väärtuse". See on kasutatav funktsiooni nime üleandmisel tegeliku parameetrina ning funktsiooni nime omistamisel (algväärtustamisel) vastavat tüüpi viidale:

```
extern int f();
int (*fp)();
...
fp = f;
g(f);
```

Funktsioonidele ei ole rakendatav operatsioon sizeof.

5.9. Andmetüüp void

Suhteliselt uus täiendus keelele C on andmetüüp void, millel pole ühtegi objekti ega järelikult ka ühtegi operatsiooni. Keeles C võib defineerida funktsiooni tüüpi void, millel pole väärtus, s.t. kogu funktsiooni sisuline tegevus avaldub kõrvalefektis:

tüübi-void-kirjeldaja:

void

Mõned kompilaatorid annavad veateate juhul, kui programmis arvutatakse tarbetu väärtus, s.t. väärtus, mida ei kasutata. Teate vältimiseks võib kasutada ilmutatud tüübiteisendust tüüpi void. Mõned näited:

```
extern void f();
extern int g();
...
f();
(void) g();
```

ANSI C võimaldab kasutada veel tüüpi void * , s.t. viita andmetüübile void kui universaalset viita, millisesse tüüpi on võimalik teisendada suvalist teist viita. Tavaliselt kasutatakse keeles C sellise universaalse viidana viita sümbolile, s.t. tüüpi char *.

5.10. Defineeritavad tüübinimed

Kui kirjelduses on mäluklassi kohal võtmesõna typedef, siis deklaraatoris olev nimi määrab sünonüümi kirjelduses näidatavale tüübile. Seda nime võib edasistes kirjeldustes kasutada tüübikirjeldaja asemel:

typedef-nimi:
nimi

Näiteks kirjeldus

```
typedef int *ptr, (*func)();
```

määrab nime ptr kui tüübi "viit täisarvule" sünonüümi ja annab nime func tüübile "viit täisarvulisele funktsioonile".

Niisugused defineeritavad tüübinimed ei määra sisuliselt uusi tüüpe vaid võimaldavad keerukate tüüpide korral kasutada lihtsamaid ja mõistetavamaid tähistusi. Näiteks eelmise näite põhjal võib anda kirjeldused:

```
ptr link, *indirect_link;
func my_func, my_vector[10];
```

Halvaks programmeerimisstiiliks on typedef abil määratud tüübi andmine koos täiendava tüübikirjeldajaga (ehkki kompilaator ei pruugi seda alati veaks pidada), näiteks:

```
typedef long int bigint;
unsigned bigint x; /* tõenäoliselt vigane !!! */
```

Seoses defineeritavate tüübinimedega tekivad ka mõningad probleemid. Näiteks võib selliselt määrata funktsioone:

```
tydedef double DblFunc();
```

kuid kahjuks ei saa nime DblFunc korrektselt kasutada funktsiooni defineerimiseks. Funktsiooni definitsioon nõuab parameetriloetelu, kui aga esitada kirjeldus

```
DblFunc fabs(x)
double x;
{ ... }
```

siis on nimi fabs määratud kui funktsioon, mille väärtus on omakorda funktsioon, kuid seda loetakse keeles C vigaseks.

Kirjeldusega typedef määratud nimesid võib omakorda kasutada ka edasistes kirjeldustes typedef. Tüübinimede skoobi kohta kehtivad samad reeglid mis teiste nimede jaoks.

5.11. Tüüpide ekvivalentsus

Selgitame siin (ja samuti ka veel järgmises peatükis) lähemalt, mida tegelikult tähendab keeles C väide, et "kaks objekti on üht ja sama tüüpi".

Kaks tüüpi "viit tüübile T" ja "viit tüübile S" on samad parajasti siis, kui tüübid T ja S on samad. Tüübid "funktsioon tüüpi T väärtusega" ja "funktsioon tüüpi S väärtusega" on samad parajasti siis, kui tüübid T ja S on samad.

Tüübid "n-elemendiline massiiv tüüpi T elementidest" ja "m-elemendiline massiiv tüüpi T elementidest" on samad siis, kui on samad tüübid S ja T, ning n väärtus võrdub m väärtusega. Sellega seoses peab märkima, et tüübid "viit 5-elemendilisele massiivile täisarvudest" ja "viit 10-elemendilisele massiivile täisarvudest" ei ole samad.

Iga loendi, struktuuri või ühendi definitsiooniga määratakse omaette tüüp, kuigi näiteks kahe struktuuri komponendid võivad nimede ja tüüpide poolest kokku langeda. Loendi, ühendi või struktuuri viide määrab sama tüüpi objektid kui vastav definitsioon. Seega järgmises näites on x ja y eri tüüpi muutujad, kuid u ja v sama tüüpi:

```
struct { int a,b; } x;
struct { int a,b; } y;
struct S { int a,b; } u;
struct S v;
```

Et defineeritud tüübinimed ei määra uusi tüüpe vaid ainult sünonüüme vanadele, siis muutujad i ja j on järgmises näites sama tüüpi:

```
typedef int count;
int i;
count j;
```

5.12. Abstraktsed deklaraatorid

Kahes kontekstis tuleb keeles C osutada tüüpi ilma konkreetset seda tüüpi objekti määramata. Need on operatsioon sizeof (vt. lk. 104) ja ilmutatud tüübiteisendus (vt. lk.

103), mil kasutatakse konstruktsiooni, mida nimetatakse tüübinimeks (mitte ära segada defineeritava tüübinime või agregaaditüübi nimega). Selle konstruktsiooni väliskuju sarnaneb kirjeldusega, kus aga määratav nimi deklaraatoris on ära jäetud. Sellist erikujulist deklaraatorit nimetatakse abstraktseks deklaraatoriks:

```
tüübinimi:
        tüübi-kirjeldaja abstraktne-deklaraator
abstraktne-deklaraator:
        tühi-abstraktne-deklaraator
        mittetühi-abstraktne-deklaraator
tühi-abstraktne-deklaraator:

mittetühi-abstraktne-deklaraator:
        ( mittetühi-abstraktne-deklaraator )
        abstraktne-deklaraator ()
        abstraktne-deklaraator [ avaldis ]
        * abstraktne-deklaraator
```

Eeskätt just mitmemõttelisuse vältimiseks nõutakse siin, et sulgudes olev abstraktne deklaraator ei oleks tühi. Mõned näited tüübinimede kohta:

```
int           - tüüp int
float *       - tüüp "viit tüübile float"
char (*) ()   - tüüp "viit funktsioonile väärtusega char"
int * ()      - tüüp "funktsioon väärtusega viit tüübile
                int"
int *[5]      - tüüp "massiiv viiest viidast tüübile int"
int (*(*())() - tüüp "viit funktsioonile väärtusega viit
                funktsioonile väärtusega int"
```

Tüübinimed peavad keele C süntaksi kohaselt nii ilmutatud tüübiteisenduse kui ka operatsiooni sizeof korral asuma sulgudes. Kui tüübinimi määrab struktuuri, ühendi või loendi, siis tuleb eeldada, et see tüüp on varem defineeritud, sest esinemine tüübinimena ei kirjelda vastavat tüüpi:

```
i=sizeof(struct  S  { int a,b; } );
/* võib olla õige, kuid vähemalt veider, kuna nimi  S
   ei ole järgnevas määratud */
```

6. T Ü Ü B I T E I S E N D U S E D

6.1. Sisemise esituse küsimusi

Enamikus programmeerimiskeeltes jääb andmetüüpide sisemine esitus programmeerija eest varjatuks. Üldiselt ei ole ka keeles C hädavajalik seda teada, kuid mõningal juhul saadakse efektiivsed programmid nimelt tänu konkreetse sisemise esituse arvestamisele. Teisest küljest tähendab orienteeritus konkreetsele esitusele ka sõltuvust sellest esitusest. Vaatame seepärast mõningaid probleeme, mis tekkivad konkreetse sisemise esituse valikul.

Kõik andmeobjektid on programmi täitmise ajal esitatud arvuti mälus kui teatud hulk abstraktseid ühikuid ehk baite. Iga selline ühik koosneb kindlast hulgast kahendkohtadest ehk bittidest. Definitsiooni kohaselt loeme andmeobjekti pikkuseks teda moodustavate baitide arvu. Samuti fikseeritakse, et tüüpi char andmeelemendi (sümboli) pikkus on 1.

Et kõik sama tüüpi objektid võtavad enda alla ühe palju mälu, siis võime rääkida ka tüübi pikkusest kui kõigi selle andmetüübi objektide pikkusest. Seega on pikkuse leidmise operatsioon sizeof kasutatav ka tüüpide korral. Ütleme, et tüüp on pikem mingist teisest tüübist siis, kui nende tüüpide objektide pikkused rahuldavad vastavat seost. Toome näite, mis leiab mõnede põhitüüpide pikkused:

```
main()
{
printf("Tüüpide pikkused:\n");
printf("char=%d\tshort=%d\tint=%d\tlong=%d\t\
float=%d\tdouble=%d\n",
sizeof(char),sizeof(short),sizeof(int),sizeof(long),
sizeof(float),sizeof(double) );
}
```

Sisemise esituse valik sõltub oluliselt adresseerimisstruktuurist. Keele C jaoks on loomulik niisugune struktuur, kus iga bait on eraldi adresseeritav. Sel juhul on suurema andmeühiku aadressiks tema esimese baidi aadress. Paraku ei

ole aga sellega veel selge, millisele andmeühiku osale see esimene bait vastab. Näiteks mõnedes arvutites on esimesel positsioonil täisarvu madalaim bait, teistes aga kõrgeim. Programmid, mis vahetult arvestavad seda baitide järjekorda on oluliselt masinsõltuvad. Sama kehtib ka bitiväljade kohta struktuurides: mõned realisatsioonid eraldavad bitid väljadesse vasakult paremale, teised paremalt vasakule.

Mõned arvutid võimaldavad suvalist tüüpi andmeelementidel paikneda arvutimälus suvalistel positsioonidel, teistes aga on oluline, et kindlat tüüpi andmeühikud asuksid kindlatel positsioonidel. Niisuguseid nõudeid nimetatakse rajastamisnõueteks. Näiteks võidakse nõuda, et 4-baidise täisarvu algusaadress jaguks neljaga ja 8-baidise reaalarvu aadress kaheksaga. Nende nõuete täitmatajätmine võib kaasa tuua vea programmi täitmisel või vähendada programmide efektiivsust.

Programmeerija ei pea hoolitsema rajastamisnõuete täitmise eest - seda teeb kompilaator. Küll aga saab viida ilmutatud tüübiteisenduste abil rajastamisnõudeid rikkuda. Kui teisendada viit nõrgema rajastamisnõudega tüübile viidaks tugevama rajastamisnõudega tüübile, siis võib esineda kahte liiki vigu. Esiteks, kui kompilaator jätab selle teisenduse jõusse, siis võib täitmisel tekkida viga riistvara tasemel. Teiseks, kui kompilaator seab rajastamisnõude õigeks, siis saame olukorra, kus pärast tagasiteisendamist lähtetüübiks ei näita saadud viit enam samale kohale (vt. ka lk. 67).

Keeles C ei eeldata, et mingi konkreetne täisarvutüüp oleks küllalt pikk sisaldamaks viidaväärtust. Paljudes realisatsioonides on kõigil viitadel esitus sama pikkusega kui tüübil int. Kuna int on ka vaikimisi eeldatav tüüp, siis suhtuvad mõned programmeerijad muretult viitade ja täisarvude vahelisse teisendamisse. Näiteks jäetakse sageli funktsiooni kirjelduses tüübikirjeldaja ära ka siis, kui funktsiooni väärtuseks ei ole täisarv vaid viit. Siiski leidub arvuteid, kus tüübi int pikkusest ei piisa igat tüüpi viitade esitamiseks. Samuti ei ole viit funktsioonile alati teisendatav muud tüüpi viitadeks. Seepärast tuleb viida puhul funktsioonile korrektselt näidata funktsiooni väärtuse tüüp.

6.2. Põhitüüpide teisendused

Mingit tüüpi väärtuse teisendamine mingisse teise tüüpi võib toimuda mitmesugustel juhtudel:

- avaldises võib olla antud ilmutatud tüübiteisendus;
- mingi operand võidakse aritmeetilisest või loogilisest operatsioonist tulenevalt teisendada mingisse teise tüüpi selle operatsiooni ettevalmistamise käigus;
- üht tüüpi väärtuse omistamine teist tüüpi objektile toob kaasa vastava tüübiteisenduse;
- funktsiooni tegelik parameeter võidakse enne funktsiooni poole pöördumist teisendada ühest tüübist teise;
- funktsiooni väärtus võidakse funktsioonist naasmisel teisendada vastavasse tüüpi.

On fikseeritud kitsendused, mis piiravad objektide teisendamist ühest tüübist teise. Need kitsendused ei ole aga ühesugused kõigi loetletud liikide korral. Vaatlemegi nüüd võimalike teisenduste hulka mitmesugustes situatsioonides.

Väärtuse teisendamine ühest tüübist teise võib kaasa tuua esituse muutmise. Kui tüüpidel on erinevad pikkused, siis muutub kindlasti esitusviis, kuid see võib muutuda näiteks ka täisarvu teisendamisel reaalarvuks, kuigi tüüpide pikkused on samad. Teisest küljest ei pruugi aga tüübi int teisendus tüübiks unsigned int kaasa tuua esituse muutmist.

Mõnel juhul on esituse muutus triviaalne, taandudes näiteks biti paljundamisele või nullide lisamisele, kuid mõnikord (näiteks reaal- ja täisarvu korral) võib esituse muutus olla komplitseeritud. Märgime, et tüüpide kokkulangemisel on alati võimalik väärtuse teisendamine ühest tüübist teise (vastavate juhtumite loetelu oli toodud eelmises peatükis).

Teisendused täisarvutüüpi on võimalikud kõigist aritmeetilistest tüüpidest ja viidatüüpidest, mõnes realisatsioonis saab täisarvutüüpi teisendada ka loenditüüpidest.

Teisendamisel täisarvutüüpidest peab põhireeglina väärtuse matemaatiline tähendus jääma võimaluse korral muutmatuks. Näiteks kui mingi märgita tüüpi väärtus on 15, siis peab sama väärtus tekkima ka teisendamisel märgiga tüüpi.

Kui väärtus pole uues tüübis esitatav, siis on kaks võimalust. Teisendamisel märgiga tüüpi on tulemuseks ületäitumine ja väärtus uues tüübis ei ole tehniliselt määratud. Kui aga uueks tüübiks on märgita tüüp, siis peab tulemus olema lähteväärtusega võrdne mooduli 2^n järgi, kus n on tulemustüübi kahendkohtade arv. Sellest järeldub, et märgiga täisarvu teisendamisel sama pikkusega märgita täisarvuks ei ole täiendkoodi korral vaja muuta esitust, s.t. uut väärtust väljendab sama bitikonfiguratsioon.

Kui märgita täisarv teisendatakse sama pikkusega märgiga täisarvuks, siis pole täiendkoodi korral samuti vaja esitusviisi muuta. Kui lähteväärtus on liiga suur tema esitamiseks märgiga arvuna, peaks tulemuseks olema ületäitumine, kuid tegelikult annab enamik realisatsioone tulemuseks negatiivse arvu ja programmeerijad on sellega küllaltki harjunud.

Lähtetüübist pikemas tulemustüübis ei pruugi lähteväärtus olla kujutatav vaid sellisel juhul, kui negatiivne väärtus teisendatakse märgita väärtuseks. Siin teostatakse tavaliselt vahepealne teisendus sama pikkusega märgiga tüüpi.

Kui tulemustüüp on lähtetüübist lühem, siis taandub märgita täisarvude korral teisendus lihtsalt osa bittide ärajätmisele vasakult, s.t. teisendus toimub mooduli 2^n järgi, kus n on tulemustüübi pikkus. Samuti toimitakse märgiga tüüpides täiendkoodi korral.

Teisendused reaalarvutüüpidest täisarvuks peavad (võimaluse korral) andma tulemuseks lähteobjekti õige matemaatilise väärtuse. Kui lähteväärtusel on murdosa, siis tavaliselt lõigatakse see ära. Juhtum, mil lähteväärtust pole ka ligikaudu võimalik esitada täisarvuna (sest reaalarvu esitusulatus on suurem) jääb konkreetse realisatsiooni otsustada.

Teisendused viidatüüpidest toimuvad nii nagu sama pikkusega märgita täisarvutüübist. Oluline on see, et viidaväärtus 0 peab teisendamisel täisarvutüüpi samuti saama väärtuse 0. Tavaliselt eeldatakse, et viida teisendamisel tüüpi long ja tagasi pole infokadu, kuigi keel seda otseselt ei nõua.

Teisendused reaalarvutüüpi on võimalikud ainult aritmeetilistest tüüpidest.

Teisendamisel tüübist float tüüpi double peab matemaatiline väärtus säilima. Kui teisendamisel tüübist double tüüpi float lähteväärtus on tüübi float kujutuspiirides, siis valitakse tulemuseks üks kahest väärtusest, mille vahel asub lähteväärtus (realisatsioonist sõltuvalt ümardatakse kas alla või üles). Kui lähteväärtus pole kujutuspiirides, siis on tulemus defineerimata (võib tekkida üle- või alatäitumine).

Teisendamisel täisarvutüüpidest peab väärtus võimaluse korral jääma muutmata. Kui aga näiteks reaalarvu mantissi jaoks on määratud vähem bitte kui täisarvule, siis peab tulemuseks olema üks kahest lähisväärtusest, mille vahel asub täisarvuline väärtus. Kui täisarvuline väärtus on väljaspool reaalarvude kujutuspiire, siis pole tulemus defineeritud.

Teisendus stuktuuri või ühendi tüüpi on võimalik ainult samast tüübist. Selles teisenduses esitus ei muutu.

Teisendamisel loenditüüpi on reeglid samad, mis täisarvutüüpides. Mõned sellised teisendused (näiteks reaalarvu teisendus loenditüüpi) osutavad halba programmeerimisstiili.

Teisendada viidatüüpi saab viitu ja täisarve. Erijuhtudel teisendatakse viidatüüpi ka massiive ja funktsioone.

Suvalist tüüpi nullviita võib teisendada suvalist tüüpi viidaks, kusjuures väärtuseks on nullviit. Väärtus tüübist viit tüübile S saab teisendada tüüpi viit tüübile T suvaliste tüüpide S ja T korral. Tuleb aga arvestada, et rajastamisnõuded võivad tulemuse muuta mittevastuvõetavaks.

Täisarvulise konstandi 0 võib alati teisendada viidatüüpi - tulemuseks on nullviit, mis ei viita ühelegi reaalsele objektile. Nullist erinevad täisarvud võivad olla teisendatavad viidatüüpi, kuid tulemused on masinsõltuvad. Tegelikus teisenduses vaadeldakse viita kui märgita täisarvutüüpi ja teisendus toimub analoogiliselt.

Kui avaldises esineb väärtus tüüpi massiiv tüüpi T elementidest, siis alati (peale operatsiooni sizeof) teisendatakse ta tüüpi viit tüübile T.

Välja arvatud funktsiooni poole pöördumine teisendatakse avaldises esinev väärtus tüübiga tüüpi T väärtusega funktsioon alati tüüpi viit tüüpi T väärtusega funktsioonile.

Teisendused massiivi- ja funktsioonitüüpi pole lubatud. Seega järgmine näide on vigane:

```
extern int f();
double d;
d =((double ())f) (); /* vigane !!! */
```

(reaalarvuks saab teisendada funktsiooni väärtust, mitte aga teisendada funktsiooni reaalarvuliseks).

Teisendada tüüpi void saab suvalist tüüpi väärtust, kuigi tulemus pole kuskil kasutatav. See teisendus võib olla kasulik mõnes realisatsioonis teatamaks kompilaatorile, et avaldise väärtust ei kasutata (muidu oleks võimalik veateade).

6.3. Teisenduste liigid

Kõik ülalpool loetletud teisendused on teostatavad ilmutatud tüübiteisenduste abil. Mõned näited:

```
int i, *ip;
char c, *cp;
float f;
double d;
enum E { red = 1, blue = 2, green = 3 } color;
...
ip = (int * ) cp;
c = (char) f;
cp = (char *) ip;
i  = (int) color;
color = (enum E) ((int) d);
```

Lihtsa omistamisavaldise (vt. lk. 120) korral peavad parema ja vasema poole tüübid olema samad või siis teisendatakse parem pool vasaku poole tüüpi. Lubatavad teisendused kõikvõimalike hulgast anname järgnevas tabelis:

Vasaku poole tüüp	Parema poole tüüp
suvaline aritmeetiline tüüp	suvaline aritmeetiline tüüp
suvaline viidatüüp	konstant 0
viit tüübile T	massiiv tüüpi T elementidest
viit funktsioonile	funktsioon

Kõik ülejäänud teisendused võivad kaasa tuua kas hoiatuse või vea, kuigi mõned kompilaatorid lubavad siin samu teisendusi nagu ilmutatud tüübiteisenduste korral.

Harilikud unaarsed teisendused määravad, kuidas teisendatakse üksikut operandi enne operatsiooni sooritamist. Teisendused teostatakse vaikimisi unaarsete operatsioonide !, -, ~ ja * operandiga, samuti ka eraldi binaarse operatsiooni << või >> kummagi operandiga ning funktsiooni poole pöördumisel iga tegeliku parameetriga.

Unaarsete teisenduste ideeks on taandada suur arv aritmeetilisi ja viidatüüpe väiksemaks arvuks tüüpideks, millede jaoks on realiseeritud vastavad operatsioonid. Teisendused jagunevad kahte liiki. Aritmeetilised tüübid teisendatakse sama liiki pikemasse tüüpi, näiteks tüüp short tüüpi int ja tüüp float tüüpi double. Viidasarnased operandid teisendatakse tegelikeks viitadeks. Anname vastavad teisendused tabelina (kõik ülejäänud tüüpi operandid jäävad muutmata):

Operandi lähtetüüp	Tüüp pärast teisendust
char, short	int
unsigned char	unsigned
unsigned short	unsigned
float	double
massiiv tüüpi T elem.	viit tüübile T
funktsioon tüüpi T	viit funktsioonile tüüpi T

Loetletud teisendused on erinevates C realisatsioonides kõige enam kasutusel. ANSI C toob veidi teistsuguse nimekirja, mida kasutatakse ka teistes viimase aja realisatsioonides. Näiteks annavad mõned kompilaatorid võimaluse välja lülitada teisendus float -> double. See ei sobi originaal-C seisukohtadega, kuid võib olla kasulik arvutusülesannetes.

Harilikud binaarsed teisendused määravad, kuidas vaikimisi teisendatakse binaarse operatsiooni operande enne operatsiooni teostamist. Need teisendused eelnevad enamikele binaarseist operatsioonidest, samuti rakendatakse neid tingimusliku operatsiooni teisele ja kolmandale operandile. Teisenduste eesmärgiks on taandada tüübid sellisteks, et

nende jaoks oleksid realiseeritud operatsioonid. Kui operatsioonil on eri tüüpi operandid, siis teisendatakse need ühisesse tüüpi ja sinna kuulub ka operatsiooni tulemus.

Harilikes binaarsetes teisendustes rakendatakse kõigepealt mõlemale operandile harilikke unaarseid teisendusi ja seejärel järgmist algoritmi.

1) Kui üks operandidest pole aritmeetilist tüüpi või on mõlemad sama tüüpi, siis täiendavaid teisendusi ei teostata.

2) Vastasel korral, kui üks operand on tüüpi double, siis teisendatakse ka teine operand tüüpi double.

3) Vastasel korral, kui üks operand on tüüpi unsigned long int, teisendatakse ka teine tüüpi unsigned long int.

4) Vastasel korral, kui operandid on tüüpi unsigned int ja long int, siis teisendatakse nad tüüpi unsigned long int.

5) Vastasel korral, kui üks operand on tüüpi long int, siis teisendatakse ka teine operand tüüpi long int.

6) Vastasel korral, kui üks operand on tüüpi unsigned int, siis teisendatakse ka teine operand tüüpi unsigned int.

7) Vastasel korral on mõlemad operandid tüüpi int ja lisateisendusi ei ole vaja.

Reegel 4 puudub mõnes realisatsioonis, siis kombinatsiooni long int ja unsigned int korral teisendatakse operandid tüüpi long int. Kui realisatsioonis puudub unaarne teisendus float -> double, siis muutuvad ka binaarsed teisendused.

Funktsiooni poole pöördumisel rakendatakse tegelikule parameetrile harilikke unaarseid teisendusi. Isegi kui puudub teisendus float -> double, teisendatakse tüüpi float parameeter siiski tüüpi double, kuna teegifunktsioonid eeldavad vaid tüüpi double argumente.

Realisatsioonides, mis võimaldavad funktsioonide kirjeldamisel anda prototüüpe (nagu ANSI C), ei teostata mitte harilikke teisendusi vaid tegelikud parameetrid teisendatakse määratud tüüpidesse. Kui prototüübikirjeldus antud funktsiooni jaoks puudub, siis kasutatakse harilikke teisendusi.

Funktsiooni väärtusele rakendatakse kõigepealt harilikke unaarseid teisendusi ja seejärel teisendatakse ta tüüpi, mis on määratud funktsiooni kirjelduses (vt. 9. ptk.).

7. A V A L D I S E D

7.1. Üldisi märkusi

Keele C rikkalik operatsioonide hulk katab enamiku vastavatest arvutioperatsioonidest. Käesolevas peatükis esitame avaldiste süntaksi ja kirjeldame iga operatsiooni.

Objekt on piirkond arvuti mälus, millele saab viidata ja kuhu saab omistada väärtusi. Nimetame l-väärtuseks avaldist, mis viitab mingile objektile ja mille abil saab ka objekti väärtust muuta (mõnikord nimetame l-väärtuseks sellise avaldise väärtust). Nimetus l-väärtus tuleneb sellest, et neid avaldisi saab kasutada vasakul (left) pool omistamismärki. Analoogiliselt avaldisi, mis võimaldavad vaid objektile viidata, kuid mitte omistada, nimetatakse vahel r-väärtusteks, kuna neid võib kasutada ainult paremal pool omistamismärki.

Aritmeetilist, viida-, loendi-, struktuuri- ja ühenditüüpi muutujate nimed on l-väärtused. Funktsioonide, massiivide ja loendikonstantide nimed aga seda ei ole. Mõned operatsioonid mitte-l-väärtustega tekitavad l-väärtusi. Näiteks ei ole massiivi nimi l-väärtus, kuid elemendi valiku operatsioon muudab selle l-väärtuseks, s.t. pole võimalik omistada massiivile, küll on aga võimalik omistada massiivi elemendile (kui see omakorda pole massiiv). Peale nimede võivad l-väärtusi produtseerida veel järgmised avaldiseliigid.

1) Elemendi valik e[k] võib anda l-väärtuse sõltumata sellest, kas avaldised e ja k on l-väärtused.

2) Avaldis sulgudes on l-väärtus parajasti siis, kui see avaldis ilma sulgudeta on l-väärtus.

3) Komponendi otsese valiku operatsioon e.nimi võib anda l-väärtuse vaid siis, kui e on l-väärtus.

4) Viida järgi komponendi valiku operatsioon e -> nimi võib anda l-väärtuse sõltumata sellest, kas e on l-väärtus.

5) Viida järgi väärtuse leidmise operatsioon *e võib anda l-väärtuse sõltumata sellest, kas e on l-väärtus.

Muud avaldiseliigid ei tekita l-väärtusi, näiteks ei ole seda omistamisoperatsiooni tulemus või funktsiooni väärtus.

Mõningad operatsioonid nõuavad operandidena l-väärtusi:

1) unaaroperatsiooni & operand peab olema l-väärtus;

2) operatsioonide ++ ja -- kasutamisel nii prefiks- kui postfikskujul peavad operandideks olema l-väärtused;

3) kõigi omistamisoperatsioonide vasakpoolseteks operandideks peavad olema l-väärtused.

Teised operatsioonid niisugust laadi nõudeid ei esita.

7.2. Operatsioonide prioriteedid ja järgnevus

Järgnevalt me esitame operatsioonid ja vastavad avaldised prioriteetide kahanemise järjekorras. Igal operatsioonil on oma prioriteeditase ja väärtustamise järjekord. Kui avaldises pole antud sulge, siis enne täidetakse kõrgema prioriteediga operatsioon, kui aga operatsioonide prioriteedid on samad, siis täidetakse neid operatsioone väärtustamise järjekorras kas vasakult paremale või paremalt vasakule. Vastavalt nimetatakse operatsioone kas vasakassotsiatiivseiks või paremassotsiatiivseiks. Kõigil ühe prioriteeditasemega operatsioonidel on ka sama assotsiatiivsus. Näiteks avaldises

```
a * b + c
```

täidetakse enne korrutamisoperatsioon, kuna tema prioriteet on kõrgem. Samuti avaldises:

```
a += b != c
```

täidetakse enne operatsioon !=, kuna tema prioriteet on kõrgem kui operatsioonil +=. Seevastu järgmises avaldises

```
a + b - c
```

on operatsioonidel + ja - sama prioriteet ning nende vasakassotsiatiivsuse tõttu täidetakse nad vasakult paremale, seega liitmisoperatsioon enne. Teisest küljest aga avaldises

```
a = b += c
```

on operatsioonid = ja += sama prioriteediga kuid paremassotsiatiivsed, seega täidetakse nad järjekorras += ja =.

Toome nüüd keele C operatsioonide (ja vastavate avaldiseliikide) täieliku loetelu prioriteediklasside kahanevas järjekorras lisades ka assotsiatiivsuse märgi L või R vastavalt kas vasak- või paremassotsiatiivsuse tähenduses:

prioriteet	operatsioon	tähendus
primaarsed ja postfiksoperatsioonid		
16	nimi konstant	lihtosutus
16L	a[k]	elemendi valik
16	f(...)	pöördumine funktsiooni poole
16L	.	komponendi valik nime järgi
16L	->	komponendi valik viida järgi
15	++ --	postfiks suurend. ja vähend.
unaaroperatsioonid		
14	++ --	prefiks suurend. ja vähend.
14	sizeof	suurus
14	(tüübinimi)	ilmutatud tüübiteisendus
14	~	bitiinversioon
14	!	loogiline eitus
14	-	unaarne miinus
14	&	aadressi võtmine
14	*	kaudsustamine
binaar- ja ternaaroperatsioonid		
13L	* / %	multiplikatiivsed
12L	+ -	aditiivsed
11L	<< >>	nihked vasakule ja paremale
10L	< > <= >=	võrdlusoperatsioonid
9L	== !=	võrdus ja mittevõrdus
8L	&	bitikaupa korrutamine
7L	^	bitikaupa mitteekvivalents
6L	¦	bitikaupa liitmine
5L	&&	loogiline korrutamine
4L	¦¦	loogiline liitmine
3R	? :	tinglik (ternaar) operatsioon
2R	= += -= *= /= %=	omistamisoperatsioonid
2R	<<= >>= &= ^= ¦=	
1L		komaoperatsioon

Igal operatsioonil on antud prioriteediklass ja assotsiatiivsus. Binaarsed ja ternaarsed operatsioonid on vasakassotsiatiivsed peale tingliku operatsiooni ja omistamiste, mis on paremassotsiatiivsed. Mõnikord kirjeldatakse unaarja postfiksoperatsioone kui paremassotsiatiivseid, kuid niisugune tõlgendus taandub tegelikult sellele, et postfiksoperatsioonide prioriteet on kõrgem. ANSI C lisab toodud nimekirja unaarse operatsiooni + ja sõnede konkatenatsiooni.

Mõningates operatsioonides nagu näiteks liitmine või lahutamine võib juhtuda, et operatsiooni tulemus (avaldise väärtus) ei mahu tulemustüübi väärtuste hulka. Sellist situatsiooni nimetatakse ületäitumiseks (või ka alatäitumiseks). Keel C ei määra üldiselt käitumist ületäitumise korral. Üheks võimaluseks on produtseerida mingi antud tüübi korrektne väärtus. Teine võimalus on katkestada programmi töö avariiga. Kolmandaks võimaluseks on mingi masinsõltuva jälgimisinfo väljastamine, et oleks võimalik teha kindlaks ületäitumise põhjus.

Mõningate teiste operatsioonide korral defineeritakse, et operatsiooni tulemus on määramata mingite kindlat tüüpi operandide korral, s.t. garanteeritakse, et mingi väärtus töötatakse välja, kuid see võib olla ettearvamatu. Näiteks kui jagamisel on parempoolne operand null, siis on käitumine defineerimata (nagu ületäitumise korral), kui aga nihutusoperatsioonis << või >> parempoolne operand on negatiivne või liiga suur, siis annab operatsioon küll väärtuse, kuid see väärtus on määramata.

Traditsiooniliselt ignoreerivad kõik C realisatsioonid märgiga täisarvude ületäitumist selles mõttes, et operatsiooni tulemuseks jääb see väärtus, mis töötati välja arvuti poolt. Paljudes täiendkoodi kasutavates arvutites on tulemuse madalamsd bitid ka ületäitumise korral õiged. Reaalarvude üle- ja alatäitumist tavaliselt aga analüüsitakse, sealhulgas on võimalik ka avariilõpp. Märgita arvude korral antakse alati resultaat, mis on õige mooduli 2^n järgi. Näiteks kui märgita väärtusest 4 lahutada märgita väärtus 7, siis 16-bitise aritmeetika korral on tulemuseks märgita väärtus 65533.

7.3. Primaaravaldised

Primaaravaldise liigid on järgmised:

primaaravaldis:
- nimi
- konstant
- sulgavaldis

Funktsiooni poole pöördumise, elemendi valiku ja komponendi valiku operatsioonid, mida sageli vaadeldakse primaaravaldisi andvatena, on siin viidud postfiksavaldiste alla.

Nime väärtus sõltub tema tüübist, mis määratakse selle nime kirjeldusega. Aritmeetiliste muutujate, viitade, loendi-, struktuuri- ja ühendimuutujate nimede väärtusteks on vastavat tüüpi objektid, kusjuures avaldis kujutab endast l-väärtust. Loendikonstandi väärtuseks on täisarv, mis on seotud konstandiga, see ei ole l-väärtus. Vaatleme näidet:

```
typedef enum { red, blue, cyan,
               magenta, green, yellow } colortype;
static colortype complementary(color)
colortype color;
{
switch (color) {
       case red:      return cyan;
       case blue:     return yellow;
       case green:    return magenta;
       case cyan:     return red;
       case yellow:   return blue;
       case magenta:  return green;
       }
}
```

Loendikonstantidena tuuakse kuus värvust. Vastavalt funktsiooni parameetriks oleva loendimuutuja väärtusele väljastatakse vastav täiendvärvus loendikonstandina (vt. lk. 141).

Massiivi nime väärtuseks on algselt massiiv, mis pole l-väärtus. Kõikjal avaldises (peale operatsiooni sizeof) teisendatakse see väärtus vastavalt harilikele unaarsetele teisendustele viidaks massiivi elemendi tüübile:

```
extern void PrintMatrix();
int Matrix[10][10], total_length, row_length;
total_length = sizeof Matrix;
row_length = sizeof Matrix[0];
PrintMatrix(Matrix);
```

Operatsioonid sizeof leiavad siin vastavalt kogu maatriksi ja tema ühe rea pikkuse. Funktsioonile üle antav tegelik parameeter Matrix teisendatakse viidaks.

Funktsiooni nime väärtuseks on funktsioon, mis pole l-väärtus. Enamikus kontekstides teisendatakse see väärtus harilike unaarsete teisenduste kohaselt viidaks vastavat tüüpi funktsioonile. Ainsaks erandiks on funktsiooni poole pöördumine, milles funktsiooni nime väärtuseks jääb funktsioon:

```
void plot(f, x0, x1)
double (*f)(), x0, x1; /* f on viit funktsioonile */
{ ... }     /* trükitakse funktsiooni f väärtused */
double fn(x)  /* mingi konkreetne f */
double x
{
extern double sin(),cos();
return(sin(x) - 2.0 * cos(x));
}
void main()
{
plot(fn, 0.01, 100.0); /* fn teisendatakse viidaks */
... }
```

Märgendit, tüübinime, struktuuri või ühendi komponendi nime, samuti struktuuri-, ühendi- või loenditüübi nime primaaravaldisena kasutada ei saa. Mõned neist nimedest võivad olla avaldises kasutatavad spetsiaalkonstruktsioonides. Näiteks struktuuride ja ühendite komponentide nimed võivad olla komponendi valiku operatsioonide . ja -> operandiks, tüübinimesid saab kasutada operatsiooni sizeof operandina jne.

Konstant avaldisena ei ole kunagi l-väärtus. Konstandi väärtus on kas vastavat aritmeetilist tüüpi või sõne korral tüüpi massiiv sümboleist, mis harilike unaarsete teisenduste kohaselt muudetakse avaldises tüüpi viit sümbolile.

Sulgavaldis koosneb avavast sulust, suvalist liiki avaldisest ja sulgevast sulust:

```
sulgavaldis:
    ( avaldis )
```

Sulgavaldise väärtus ja tüüp langevad kokku sulgudes antud avaldise omadega (samuti ka see, kas ta on l-väärtus või mitte). Sulgavaldist kasutatakse arvutusjärjekorra määramiseks juhul, kui prioriteetidest tulenev järjestus ei rahulda. Samuti võib sulge kasutada avaldiste loetavamaks muutmiseks. Peab arvestama (vt. lk. 125), et sulud ei tarvitse alati määrata arvutusjärjekorda. Näide sulgavaldise kohta:

```
x1 = (-b + discr)/(2 * a);
```

7.4. Postfiksavaldised

Postfiksavaldised jagunevad järgmiselt:

```
postfiksavaldis:
    primaaravaldis
    elemendi-valik
    komponendi-valik
    pöördumine-funktsiooni-poole
    postfiks-suurendamine
    postfiks-vähendamine
```

Elemendi ja komponendi valikut ning funktsiooni poole pöördumist vaadeldakse mõnikord kui primaaravaldisi, kuid nende süntaks sarnaneb rohkem postfiksavaldistele.

Massiivi elemendi valik koosneb postfiksavaldisest, avavast nurksulust, avaldisest ja sulgevast nurksulust. Postfiksavaldis (tavaliselt, kuid mitte tingimata massiivi nimi) viitab massiivi algusele ning nurksulgudes olev avaldis määrab viidaaritmeetika järgi täisarvulise nihke:

```
elemendi-valik:
    postfiksavaldis [ avaldis ]
```

Keeles C on avaldised e1[e2] ja *((e1) + (e2)) definitsiooni kohaselt ekvivalentsed. Avaldistele e1 ja e2 rakendatakse harilikke binaarseid teisendusi ning saadav tulemus on l-väärtus. Märgime, et operatsiooni * operandiks peab olema

viit, mis on summa korral võimalik vaid siis, kui ühe liidetava tüübiks on viit ja teine on täisarvuline. Sama kehtib ekvivalentsi kohaselt ka avaldise e1[e2] korral, kuid siin nõutakse, et viidatüüpi oleks avaldis e1. Mõned näited:

```
char buffer[90], *bptr = buffer;
int i = 89;
buffer[0] = '\0';
bptr[i] = bptr[0];
```

Nimetatud ekvivalentsist järeldub, et massiivi elementide indeksid algavad nullist, näiteks 90-elemendilise massiivi indeksid muutuvad rajades 0 kuni 89. Eelmise näite muutuja bptr on viit, seega l-väärtus ja talle võib omistada massiivi elemendi aadressi, näiteks bptr = &buffer[6]; (pärast seda on bptr[-4] väärtus sama mis avaldisel buffer[2]). Siit nähtub, et mõnes kontekstis on mõtet ka negatiivsetel indeksitel. Teisest küljest buffer pole avaldisena l-väärtus ja temale ei saa omistada. Massiivi nime võib vaadelda kui konstantset viita, mis viitab alati samale kohale.

Mitmemõõtmelise massiivi elemendi valik toimub valikuoperatsiooni korduva rakendamisega (mitte aga mitme avaldise kirjutamisega nurksulgudesse). Vaatleme näitena programmilõiku kahemõõtmelise ühikmaatriksi moodustamiseks:

```
#define SIZE 10
{
double matrix[SIZE][SIZE];
int i,j;
/* see ei ole kiireim meetod ühikmaatriksi jaoks */
for (i=0; i < SIZE; ++i)
  for (j=0; j < SIZE; ++j)
       matrix[i][j] = ((i == j) ? 1.0 : 0.0 );
}
```

Halvaks programmeerimisstiiliks on komaavaldise kasutamine nurksulgudes - paljudes programmeerimiskeeltes tähendab see elemendi valikut mitmemõõtmelisest massiivist ja võib raskendada programmi mõistmist. Näiteks järgmises avaldises valitakse element indeksiga 2*k ühemõõtmelisest massiivist:

```
massiiv[k=n+1,2*k]
```

Kui komaavaldis on elemendi valiku operatsioonis tõepoolest vajalik (kuigi on raske leida selle kohta mõistlikku näidet), siis on soovitatav panna ta sulgudesse, seega:

```
massiiv[(k=n+1,2*k)]
```

Elemendi valiku operatsioonis saab mitmemõõtmelise massiivi ilmutatud tüübiteisendusega viidaks muuta sisuliselt ühemõõtmeliseks (selline kunstlikuna näiv vahend võib anda efektiivsema programmi). Siinjuures tuleb meeles pidada, et massiivid paiknevad mälus ridade kaupa. Illustreerimiseks teisendame veidi eelmist ühikmaatriksi moodustamise näidet:

```
#define SIZE 10
{
double matrix[SIZE][SIZE];
int i; /* see programm peaks olema eelmisest kiirem */
/* kõigepealt nullime kogu maatriksi */
for(i=0; i < SIZE * SIZE; ++i)
   ((double *) matrix) [ i ] = 0.0;
/* seame nüüd paika diagonaali elemendid */
for(i=0; i < SIZE * SIZE; i += (SIZE + 1))
   ((double *) matrix) [ i ] = 1.0;
}
```

Komponendi valiku operatsioone kasutatakse struktuuride ja ühendite komponentidele viitamiseks:

```
komponendi-valik:
     otsene-komponendi-valik
     kaudne-komponendi-valik
otsene-komponendi-valik:
     postfiksavaldis . nimi
kaudne-komponendi-valik:
     postfiksavaldis -> nimi
```

Komponendi otseseks valimiseks tuleb kirjutada postfiksavaldis, punkt ja nimi. Postfiksavaldise tüübiks peab seejuures olema struktuur või ühend ja nimi peab olema vastava struktuuri või ühendi komponendi nimi. Avaldise väärtuseks ja tüübiks tuleb vastava komponendi väärtus ja tüüp. Tulemuse väärtus on l-väärtus siis, kui seda on postfiksavaldise väärtus ja komponent ei ole massiiv. Tuleb märkida, et esi-

mene tingimus ei ole täidetud näiteks funktsiooni väärtustena esinevate struktuuride või ühendite korral.

Komponendi kaudse valimise avaldis koosneb postfiksavaldisest, liitoperatsioonitähisest -> ja nimest. Postfiksavaldise tüübiks peab olema viit struktuurile või ühendile ja nimi peab olema vastava struktuuri või ühendi komponendi nimi. Avaldise väärtuseks ja tüübiks tuleb vastava komponendi väärtus ja tüüp. See väärtus on l-väärtus siis, kui vastav komponent ei ole massiiv. Definitsiooni kohaselt on avaldis e -> nimi ekvivalentne avaldisega (*e).nimi. Mõned näited:

```
struct { float x,y; } Point, *Point_pt;
Point.x = 0.0; /* seab x väärtuseks 0.0 */
Point_pt= &Point /* Point_pt viitab muutujale Point */
Point_pt -> y = 0.0 /* seab y väärtuseks 0.0 */
```

Funktsiooni pole pöördumise avaldis koosneb postfiksavaldisest, avavast sulust, mittekohustuslikust komadega eraldatud avaldiste loetelust ja sulgevast sulust:

```
pöördumine-funktsiooni-poole:
        postfiksavaldis ( avaldiste-loetelu )
avaldiste-loetelu:
        omistamisavaldis
        avaldiste-loetelu , omistamisavaldis
```

Postfiksavaldise tüübiks peab olema funktsioon väärtusega T, kus T on mingi tüüp. Funktsiooni poole pöördumise kui operatsiooni väärtuse tüübiks on sel juhul T, operatsiooni tulemuseks aga väljakutsutavas funktsioonis tagastatud väärtus, mis pole kunagi l-väärtus. Kui tüübiks T on void, siis funktsioonil ei ole väärtust.

Mõned realisatsioonid lubavad funktsiooni poole pöördumisel postfiksavaldise tüübina ka tüüpi viit funktsioonile:

```
int (*f)();
f(x,y);      /* pöördumine läbi viida */
```

Sellise vabaduse põhjenduseks on soov hoiduda liiga keerukatest konstruktsioonidest, antud juhul peaks see olema

```
(*f)(x,y);
```

Avaldiste loetelus toodud avaldised määravad funktsiooni tegelikud parameetrid. Nende avaldistega teostatakse funkt-

siooni parameetritele vastavad teisendused, kuid üldiselt mingeid muid teisendusi ja kontrolle ei teostata. Näiteks ei pea kompilaator veateate või mingi muu tegevusega reageerima sel juhul, kui tegelike parameetrite arv või nende tüübid ei vasta funktsiooni definitsioonis antuile. Üldjuhul ei pruugi niisugune informatsioon isegi olla kompilaatorile kättesaadav (kui aga on kättesaadav, siis pole hoiatav veateade või muu reageerimine ka keelatud).

Pärast argumentide teisendamist tehakse neist koopiad väljakutsutava funktsiooni jaoks, s.t. parameetrite üleandmine toimub väärtuse järgi. Väljakutsutavas funktsioonis on parameetrite nimed l-väärtused, kuid neile omistamine muudab ainult formaalse parameetri väärtust ja ei mõju tegelikule parameetrile. Vaatleme näiteks järgmist programmilõiku:

```
double power4(y)  /* leiab arvude neljanda astme */
double y;
{
y *= y;       /* tõstame y ruutu */
return y * y; /* ja veel kord ruutu */
}
void main()
{
double x,z;
...
x = 3.0;
z = power4(x);
... }
```

Funktsioon power4 kasutab oma formaalset parameetrit ühtlasi vahetulemuse säilitamiseks. Nii piisab ainult kahest korrutamistehtest (y * y * y * y jaoks on vaja kolm). Funktsiooni väljakutsumine funktsioonis main tegeliku parameetri x väärtust ei muuda: pärast funktsioonist power4 naasmist on x väärtus 3.0 ja z väärtus 81.0.

Tuleb arvestada, et kui parameetriks on viit, siis tema väärtuse kopeerimisel ei kopeerita objekti, millele ta viitab. Seega parameetritena viitade kasutamisel saab väljakutsutav funktsioon muuta väljakutsuva funktsiooni objekte.

Kui väljakutsutava funktsiooni tüübiks ei ole void, kuid väljakutsuv programm funktsiooni väärtust ei kasuta, siis võib sellest kompilaatorile teatada ilmutatud tüübiteisendusega tüüpi void (muidu annab mõni kompilaator veateate):

```
{ ...
(void) strcat(word,suffix);
... }
```

Komaavaldist saab funktsiooni tegeliku parameetrina kasutada vaid siis, kui asetada ta sulgudesse:

```
{
int i,j;
...
f( ( i=1, i), (j=1, j)); /* korrektne, kuid veider */
... }
```

Postfiksne suurendamis- ja vähendamisoperatsioon on kõrvalefektiga operatsioonid:

```
postfiks-suurendamine:
    postfiksavaldis ++
postfiks-vähendamine:
    postfiksavaldis --
```

Operand peab olema l-väärtus ja võib olla suvalist skalaarset tüüpi. Sellele l-väärtusele vastava objekti väärtusele liidetakse (või vähendamise korral lahutatakse temast) konstant 1. Avaldise väärtuseks jääb operandi vana väärtus, mis pole l-väärtus. Operandile ja konstandile 1 rakendatakse enne liitmist (lahutamist) harilikke binaarseid teisendusi ja enne omistamist objektile omistamisega seotud teisendusi.

Kui operatsiooni käigus tekib ületäitumine, siis on käitumine märgiga täisarvutüüpide ja reaalarvutüüpide korral määramata. Suurima võimaliku märgita väärtuse suurendamisel on tulemuseks null ja märgita nulli vähendamisel suurim märgita väärtus. Kui operandiks on viit mingile tüübile T, siis pärast operatsiooni viitab see järgmisele (eelmisele) tüüpi T objektile, avaldise väärtuseks on aga viida väärtus enne modifitseerimist. Üldlevinud võte on postfiksse suurendamise või vähendamise kasutamine massiivi elementide skaneerimisel. Järgnevas näites arvutatakse sõne pikkus:

```
int strlen(cp)
char *cp;
{
int count = 0;
while(*cp++) count++;
return count;
}
```

7.5. Unaaravaldised

Unaaravaldiste tüübid on järgmised:

unaaravaldis:
- postfiksavaldis
- ilmutatud-tüübiteisendus
- operatsioon-sizeof
- unaarne-miinus
- loogiline-eitus
- bitiinversioon
- aadressi-leidmine
- kaudsustamine
- prefiks-suurendamine
- prefiks-vähendamine

Unaaroperatsioonide prioriteet on madalam kui postfiks-operatsioonidel, kuid kõrgem kui binaar- ja ternaaroperatsioonidel. Näiteks avaldis *a++ arvutatakse kui *(a++), mitte kui (*a)++. ANSI C toob sisse veel unaarse plussi.

Ilmutatud tüübiteisendus koosneb avavast sulust, tüübinimest, sulgevast sulust ja unaaravaldisest:

ilmutatud-tüübiteisendus:
(tüübinimi) unaaravaldis

Selle operatsiooniga teisendatakse operandiks oleva avaldise väärtus tüüpi, mille määrab tüübinimi sulgudes (tulemus ei ole l-väärtus). Ilmutatult on teostatavad kõik eelmises peatükis vaadeldud tüübiteisendused. Toome näite:

```
extern char *alloc();
struct S *p;
p = (struct S *) alloc(sizeof(struct S));
```

Mõned realisatsioonid ignoreerivad (ebakorrektselt) mõningaid tüübiteisendusi. Oletame näiteks, et tüübi unsigned short pikkus on 16 bitti, tüübi unsigned pikkus aga 32 bitti. Siis avaldise

(unsigned)(unsigned short)0xFFFFFF

väärtuseks peab tulema 0xFFFF, kuna teisendus tüüpi unsigned short nõuab väärtuse lõikamist 16 bitile. Mõni kompilaator aga jätab siin väärtuse 0xFFFFFF muutmata. Maksimaalse ülekantavuse tagamiseks võib soovitada sel juhul mitte ilmutatud tüübiteisendust vaid omistamist vahemuutujale.

Operatsioon sizeof leiab objekti või andmetüübi pikkuse. Operatsioonil on kaks lubatud kuju - võtmesõnale sizeof võib järgneda kas tüübinimi sulgudes (nagu ilmutatud tüübiteisenduses) või avaldis:

operatsioon-sizeof:
sizeof (tüübinimi)
sizeof unaaravaldis

Kui sizeof operand on tüübinimi sulgudes, siis tulemus annab seda tüüpi objekti pikkuse, näidates kui palju mäluühikuid (baite) võtab enda alla selle tüübi objekt. Tüüp ei tohi seejuures olla ilmutatult andmata rajadega massiiv, funktsioon ega void. Kui tüübinimena anda struktuuri, ühendi või loenditüübi definitsioon (mida tuleb pidada halvaks programmeerimisstiiliks), siis tasub meeles pidada, et leitakse vaid antud tüübi pikkus, tüüp ise aga jääb määramata.

Kui sizeof operandiks on avaldis, siis saame sama tulemuse nagu juhul, kui operandiks oleks selle avaldise tüüp. Operatsiooni sizeof väärtustamisel ei teostata mingeid teisendusi, kui aga avaldises esinevad operatsioonid toovad endaga kaasa tüübiteisendusi, siis kajastuvad need teisendused ka operatsiooni sizeof väärtuses. Seega kui sizeof operandiks on massiivi nimi, saame tulemuseks kogu massiivi pikkuse (kuna tavalist teisendamist viidaks siin ei toimu). Teisest küljest, kui muutuja v on tüüpi short, siis sizeof v väärtus on sama mis avaldisel sizeof(short), sizeof (v+0) väärtus on aga sama mis avaldisel sizeof(int), sest liitmistehtes teostatakse harilik tüübiteisendus short -> int.

Operatsiooni sizeof tulemus arvutatakse kompileerimisajal. Seega operatsioone vastavas avaldises analüüsitakse ainult tüüpide seisukohalt, täidetavat koodi ei kompileerita ning kõrvalefektid jäävad seega täitmata. Näiteks avaldisega

```
k = sizeof (j++);
```

antakse muutujale k mingi väärtus, kuid muutuja j väärtust ei suurendata. Seepärast ei ole kõrvalefektiga avaldiste kasutamisel operatsioonis sizeof mõtet.

Mõni realisatsioon ei luba operatsiooni sizeof rakendada bitiväljadele. Teised küll lubavad, kuid annavad tulemuseks selle välja tüübi pikkuse, mis aga ei sõltu välja pikkusest. Seega sizeof rakendamisest bitiväljadele peaks hoiduma.

Originaal-C ei määra operatsiooni sizeof tulemuse tüüpi. Tavaliselt võetakse sõltuvalt operandi pikkusest tüübiks kas unsigned või unsigned long.

Lõpuks märgime keele C olulist mitmesust, mis võib esile tulla operatsiooni sizeof korral. Vaatleme näiteks avaldist

```
sizeof(long)-2
```

mida võib tõlgendada nii kujul sizeof((long)-2) kui ka kujul (sizeof(long))-2. Eri realisatsioonid lahendavad selle vastuolu üldiselt suvaliselt.

Unaarne miinus sooritab aritmeetilist tüüpi operandi nn. aritmeetilise negatsiooni. Teostatakse harilikud unaarteisendused ning operatsiooni tulemus ei ole l-väärtus:

```
unaarne-miinus:
    - unaaravaldis
```

Operatsiooni -e, kus e on avaldis, võime tõlgendada kui avaldist 0 - (e), sest mõlemal neil on sama väärtus. Märgiga täisarvutüüpide ja reaalarvutüüpide korral võib tekkida ületäitumine, mispuhul käitumine on määramata. Märgita operandi k korral on tulemuseks alati väärtus 2^n-k, kus n on tulemustüübi bittide arv. Tulemus pole sel juhul kunagi negatiivne, kuid iga märgita täisarvu x korral x + (-x) võrdub nulliga.

Loogilise eituse ehk negatsiooni operand võib olla suvalist skalaarset tüüpi:

```
loogiline-eitus:
    ! unaaravaldis
```

Operandile rakendatakse harilikke unaarseid teisendusi ning tulemuse tüübiks on alati int. Tulemuse väärtuseks on 1 siis, kui operandi väärtus on 0 (ka viitade ja reaalarvutüüpide korral) ja 0 siis, kui operandi väärtus erineb nullist. Avaldise ! (x) väärtus langeb kokku avaldise (x) == 0 väärtusega. Toome näite:

```
if (! x) ...
```

Bitiinversioon arvutab operandi n.-ö. pöördesituse:

```
bitiinversioon:
        ~ unaaravaldis
```

Operand võib olla suvalist täisarvutüüpi ja talle rakendatakse harilikke unaarseid teisendusi. Tulemuse iga bitt on operandi vastava biti negatsioon. Tulemus ei ole l-väärtus. Näiteks:

```
/* nullime muutuja address kolm madalamat bitti */
#define LOW_BITS 7
unsigned address;
...
address &= ~LOW_BITS;
```

Et erinevad realisatsioonid võivad märgiga täisarve kujutada erinevalt, siis on bitiinversiooni kasutamine märgiga täisarvutüüpide korral üldiselt masinsõltuv. Seetõttu võib operatsiooni ~ soovitada ainult märgita tüüpide korral.

Aadressi leidmise operatsiooni & tulemuseks on viit tema operandile, milleks peab olema l-väärtus:

```
aadressi-leidmine:
        & unaaravaldis
```

Kui operandi tüüp on T, siis operatsiooni tulemuse tüüp on viit tüübile T. Operandile ei rakendata mingeid teisendusi, tulemus ei ole l-väärtus.

Originaal-C ei võimalda aadressi määrata operandile mäluklassiga register. Mõned kompilaatorid siiski võimaldavad seda, kuna mäluklass register on ainult soovitus kompilaatorile ja mitte tingimusteta määrang. Sel juhul toob aadressi võtmine registrimuutujast kaasa mäluklassi register asendamise mäluklassiga auto. Soovitatav on seega operatsiooni & kasutamisest registrimuutujatele hoiduda.

Operatsiooni & operandiks ei või olla massiivi nimi ega funktsiooni nimi, sest need avaldised pole l-väärtused. Meenutame, et enamikus kontekstides teisendatakse funktsiooni nimi automaatselt viidaks funktsioonile ja massiivi nimi viidaks massiivi esimesele elemendile. Näide:

```
extern int i, f();
int *ip, (*fp)();
ip = &i /* operatsioon & on vajalik */
fp = f;   /* operatsioon & pole vajalik */
```

Kuigi mõned kompilaatorid võimaldavad operatsiooni & ka funktsiooni või massiivi nime korral, tõlgendades seda kui tühioperatsiooni, pole selline kasutamine siiski korrektne.

Kaudsustamisoperatsioon * leiab viida järgi väärtuse. Seega võib operatsioone & ja * lugeda pöördoperatsioonideks:

kaudsustamine:
    * unaaravaldis

Operandile rakendatakse harilikke unaarseid teisendusi, millest reaalselt mõjub massiivide ja funktsioonide teisendamine viitadeks. Operandi (teisendatud) väärtuse tüüp peab olema viit tüübile T ja tulemus on T tüüpi l-väärtus, mis nimetab objekti, millele viitab operand. Näiteks:

```
int i,*p;
p = &i;     /* p viitab muutujale i */
*p = 10;    /* muutuja i väärtus on nüüd 10 */
```

Kaudsustamisoperatsiooni rakendamine nullviidale võib kaasa tuua määramata käitumise, ka programmi avariilõpu.

Prefiksne suurendamis- ja vähendamisoperatsioon on nagu vastavad postfiksoperatsioonid kõrvalefektiga operatsioonid:

prefiks-suurendamine:
    ++ unaaravaldis
prefiks-vähendamine:
    -- unaaravaldis

Operand peab olema l-väärtus ja võib olla suvalist skalaarset tüüpi. Operatsiooniga liidetakse operandile (või lahutatakse temast) konstant 1 ja tulemus omistatakse tagasi l-väärtusele. Avaldise väärtuseks on saadud uus väärtus, mis pole l-väärtus. Operatsioon ++(x) on identne operatsiooniga

(x)+=1 ja --(x) operatsiooniga (x)-=1 (vt. lk. 122). Operandidele rakendatakse harilikke binaarseid teisendusi ning enne omistamist sellele vastavaid teisendusi. Näide:

```
static uniqint()
/* väljastatakse järjestikused naturaalarvud */
{
static int count = 0;
return ++count;
}
```

Kui operatsioon tekitab ületäitumise, siis on käitumine märgiga täisarvude ja reaalarvude korral määramata. Suurima märgita väärtuse suurendamise tulemuseks on 0, märgita nulli vähendamise tulemuseks suurim märgita väärtus. Kui operand on tüüpi viit tüübile T, siis tulemus viitab järgmisele (eelmisele) tüüpi T objektile. Laused ++a; ja a++; on samaväärsed (samuti --a; ja a--;), kuid tavaliselt eelistatakse viimast, ehkki esimene versioon võib mõnes realisatsioonis töötada kiiremini.

Vaatleme veel näitena programmilõiku, mis pöörab sõne ümber, s.t. annab ta vastupidises järjekorras:

```
void strrev(s1,s2)
char *s1,*s2;
{
/* sõne ümberpööramine */
char *p = s1;
while (*p++) ; /* leiame lähtesõne lõpu */
--p;  /* võtame viida tagasi enne lõpunulli */
/* salvestame sümbolid vastupidises järjekorras */
while (p > s1 ) *s2++ = *--p;
*s2 = '\0' ;  /* lõpetav null */
}
```

7.6. Binaaravaldised

Binaaravaldiseks nimetatakse avaldist, mis koosneb kahest binaarse operatsiooniga seotud avaldisest. Prioriteetide kahanemise järjekorras vaatleme järgmisi binaaravaldisi:

multiplikatiivne avaldis, aditiivne avaldis, nihkeavaldis, võrdlusavaldis, võrdusavaldis, bitikaupa-AND avaldis, bitikaupa-XOR avaldis ja bitikaupa-OR avaldis.

Kõik keele C binaarsed operatsioonid on vasakassotsiatiivsed. Seega kuna operatsioonid * ja % on võrdse prioriteediga, siis avaldist x*y%z arvutatakse kujul (x * y)%z, mitte kujul x * (y%z). Kõigis binaarsetes operatsioonides väärtustatakse operandid enne operatsiooni teostamist, kuid väärtustamise järjekord ei pruugi olla kindlaks määratud.

Multiplikatiivsed operatsioonid korrutamine, jagamine ja jäägi leidmine on kõik sama prioriteediga:

```
multiplikatiivne-avaldis:
        unaaravaldis
        multiplikatiivne-avaldis mult-op unaaravaldis
mult-op: variandid
        * / %
```

Binaarne operatsioon * tähistab korrutamist, kus mõlemad operandid peavad olema aritmeetilist tüüpi. Operandidele rakendatakse harilikke binaarseid teisendusi, tulemuse tüüp on teisendatud operandide ühine tüüp. Tulemus ei ole l-väärtus. Täisarvuliste operandidega teostatakse täisarvuline, reaalarvulistega aga reaalarvuline korrutamine.

Ületäitumise korral on käitumine märgiga täisarvuliste ja reaalarvuliste operandide puhul defineerimata. Märgita täisarvuliste operandide korrutis on ületäitumise korral õige mooduli 2^n järgi, kus n on vastava tüübi bittide arv.

Operatsioon * eeldatakse olevat kommutatiivne ja assotsiatiivne. Kompilaator võib korduvaid korrutamisi ümber järjestada, hoolimata isegi ilmutatult antud sulgudest. Näiteks võidakse avaldist a * (b * c) * d väärtustada ka järjekorras (c * a) * (b * d). Kui operandide väärtustamise järjekord on oluline (näiteks kõrvalefektide tõttu), siis tuleb selline avaldis anda koos vahepealse omistamisega abimuutujale.

Binaarne operatsioon / tähistab jagamist, kus mõlemad operandid peavad olema aritmeetilist tüüpi. Operandidele rakendatakse harilikke binaarseid teisendusi. Tulemus pole l-väärtus, tema tüübiks on teisendatud operandide ühine tüüp.

Reaalarvuliste operandide korral teostatakse reaalarvuline jagamine. Täisarvuliste operandide korral, kui operandid ei jagu täpselt on tulemuseks üks kahest täisarvulisest väärtusest, mille vahele jääb jagatise õige väärtus. Kui mõlemad operandid on positiivsed, siis jagamise tulemuseks on nullile lähem väärtus, s.t. toimub murdosa äralõikamine. Sama kehtib ka märgita täisarvude jagamise korral. Kui vähemalt üks operandidest on negatiivne, siis sõltub tulemus realisatsioonist. Seepärast tuleb maksimaalse masinsõltumatuse saavutamiseks hoiduda negatiivsete operandidega täisarvulisest jagamisest.

Kui jagamisel tekib ületäitumine märgiga täisarvuliste või reaalarvuliste operandide korral, siis on käitumine defineerimata. Märgita täisarvude korral ületäitumist tekkida ei saa. Reaktsioon nulliga jagamisel on nii täis- kui reaalarvuliste operandide korral masinsõltuv.

Binaarne operatsioon % tähistab esimese operandi teisega jagamisel tekkiva jäägi leidmist. Operandid peavad olema täisarvutüüpi, neile rakendatakse harilikke binaarseid teisendusi ja tulemuse tüübiks on teisendatud operandide ühine tüüp. Tulemus ei ole l-väärtus.

Kui b ei ole võrdne nulliga, siis avaldise (a/b)*b + a%b väärtus võrdub avaldise b väärtusega, seega jäägi leidmine sõltub täisarvulisest jagamisest. Kui mõlemad operandid on positiivsed, siis on jäägi leidmine samaväärne matemaatilise funktsiooniga mod. Sama kehtib ka märgita operandide puhul. Kui vähemalt üks operandidest on negatiivne, siis sõltub tulemus realisatsioonist samuti nagu täisarvulise jagamise korral. Seepärast tuleb maksimaalse masinsõltumatuse saavutamiseks hoiduda jäägi leidmise operatsiooni kasutamisest negatiivsete operandidega.

Kui jäägi leidmisel tekib ületäitumine märgiga täisarvude korral, siis on käitumine määramata. Märgita operandide puhul ületäitumist tekkida ei saa. Kui teise operandi väärtuseks on null, siis on reaktsioon masinsõltuv.

Vaatleme näitena funktsiooni kahe täisarvu suurima ühisteguri leidmiseks Eukleidese algoritmi abil:

```
unsigned syt(x,y)
unsigned x,y;
{  unsigned temp;
while(y != 0)
    {
     temp = y;
     y = x % y;
     x = temp;
    }
return x;
}
```

Aditiivsed operatsioonid liitmine ja lahutamine on ühesuguse prioriteediga (ning vassakassotsiatiivsed):

```
aditiivne-avaldis:
    multiplikatiivne-avaldis
    aditiivne-avaldis ad-op multiplikatiivne-avaldis
ad-op: variandid
    + -
```

Binaarne operatsioon + tähistab liitmist. Operandidele rakendatakse harilikke binaarseid teisendusi. Operandid peavad olema kas mõlemad aritmeetilist tüüpi, või siis üks operand viidatüüpi ja teine täisarvutüüpi (muud tüüpi operandid pole lubatud). Operatsiooni tulemus ei ole l-väärtus.

Kui operandid on mõlemad aritmeetilist tüüpi, siis on tulemuse tüübiks teisendatud operandide ühine tüüp. Täisarvuliste operandide korral teostatakse täisarvuline liitmine, reaalarvuliste operandide korral reaalarvuline liitmine.

Kui operandideks on viit p mingile tüübile T ja täisarvuline väärtus k, siis operatsiooni tulemuseks on viit tüübile T, mis viitab k objekti edasi viida p esialgsest väärtusest. Näiteks p+1 viitab järgmisele ja p + (-1) eelmisele objektile võrreldes objektiga, millele viitab p. Korrektne on selline operatsioon siis, kui p viitab mingile massiivi elemendile. Seepärast ei tohi p siin olla viit funktsioonile, kuna funktsioonidest ei saa moodustada massiive.

Kui liitmisel tekib ületäitumine, siis on käitumine märgiga täisarvude ja reaalarvude korral määramata. Käitumine

ületäitumisel on määramata ka siis, kui üks operand on viit. Märgita täisarvude korral on tulemus õige mooduli 2^n järgi, kus n on vastava märgita tüübi bittide arv.

Operatsioon + eeldatakse olevat kommutatiivne ja assotsiatiivne. Kompilaator võib korduvaid liitmisi ümber järjestada ka siis, kui on antud sulud (vt. lk. 109).

Binaarne operatsioon - tähistab lahutamist. Operandidele rakendatakse harilikke binaarseid teisendusi. Mõlemad operandid peavad olema kas aritmeetilist tüüpi või siis võib vasakpoolne operand olla viidatüüpi ja parempoolne täisarvutüüpi. Lõpuks võivad mõlemad operandid olla ka viidad sama tüüpi objektidele. Mingid muud operandide tüübid pole lubatud. Tulemus ei ole l-väärtus.

Kui operandid on mõlemad aritmeetilist tüüpi, siis on tulemuse tüübiks teisendatud operandide ühine tüüp. Täisarvuliste operandide korral teostatakse täisarvuline ja reaalarvuliste operandide korral reaalarvuline lahutamine. Märgime, et ühest märgita täisarvust teise märgita täisarvu lahutamise tulemus on märgita ja seega ei saa olla negatiivne. Küll aga kehtivad märgita täisarvu tüüpi avaldiste korral seosed (a + (b - a)) == b ja (a - (a - b)) == b.

Täisarvulise väärtuse lahutamine viidast on analoogiline täisarvulise väärtuse liitmisega viidale. Kui viidast p tüübile T lahutatakse täisarvuline väärtus k, siis tulemuseks on viit sama tüüpi objektile, mis viitab esialgsest k objekti tagasi. Näiteks viitab avaldis p-1 eelmisele ja p - (-1) järgmisele objektile võrreldes objektiga, millele viitab p. See lahutamine on korrektne vaid siis, kui p viitab mingile massiivi elemendile. Seega juhul kui on tegemist viidaga funktsioonile, pole täisarvulise väärtuse lahutamine temast korrektne, kuna funktsioonidest ei saa moodustada massiive.

Kui p ja q on viidad sama tüüpi objektidele, siis p - q on täisarvulist tüüpi ning tema väärtus k selline, et q + k võrdub viida p väärtusega. Väärtuse tüüp võib sõltuvalt realisatsioonist olla nii int kui long. Selline lahutamine on korrektne vaid juhul kui mõlemad viidad p ja q viitavad ühe ja sama massiivi elementidele. Kui ükskõik kumma viida väär-

tus on null, siis pole tulemus defineeritud. Viitu funktsioonidele üksteisest lahutada ei saa.

Kui lahutamisel tekib ületäitumine, siis on käitumine märgiga täisarvutüüpide ja reaalarvutüüpide korral määramata. Sama kehtib ka siis, kui ükskõik kumb operand on viit. Märgita täisarvude korral on tulemus õige mooduli 2^n järgi, kus n on vastava tüübi bittide arv.

Nihkeoperatsioon << tähistab nihutamist vasakule ja >> nihutamist paremale. Need operatsioonid on ühesuguse prioriteediga ja vasakassotsiatiivsed:

```
nihkeavaldis:
        aditiivne-avaldis
        nihkeavaldis nihkeop aditiivne-avaldis
nihkeop: variandid
        <<  >>
```

Mõlemad operandid peavad olema täisarvutüüpi ja neile rakendatakse eraldi harilikke unaarseid teisendusi (nihkeoperatsioonide korral ei rakendata harilikke binaarseid teisendusi). Tulemus ei ole l-väärtus ja tema tüübiks on teisendatud vasakpoolse operandi tüüp.

Vasakpoolne operand defineerib nihutatava väärtuse ning parempoolne nihke suuruse, s.t. kahendkohtade arvu, mille võrra vasakpoolset nihutatakse. Operatsioon << nihutab vasakpoolset operandi vasakule, vasakpoolsed bitid hävivad ja parempoolsed vabanevad bitid täidetakse nullidega. Operatsioon >> nihutab vasakpoolset operandi paremale ja parempoolsed bitid hävivad. Millega täidetakse vasakpoolsed vabanevad bitid sõltub vasakpoolse operandi tüübist. Kui see on märgita tüüp, siis täidetakse vasakpoolsed bitid nullidega. Märgiga tüübi korral sõltuvalt realisatsioonist kas lisatakse vasakule nulle või paljundatakse vasakpoolseimat bitti.

Nihutamisoperatsioonide tulemus pole defineeritud siis, kui parempoolne operand osutub negatiivseks. Sama kehtib siis, kui parempoolse operandi väärtus on küll positiivne, kuid suurem vasakpoolse operandi tüübi pikkusest bittides. Märgime, et parempoolse operandi väärtuseks võib olla 0 ja siis lihtsalt mingit nihutamist ei toimu.

Järjestikuseid nihutamisoperatsioone täidetakse vasakult paremale. Järgmine avaldis näiteks eraldab 16-bitise b korral 8 keskmist bitti:

```
b << 4 >> 8
```

Ehkki näide on semantiliselt korrektne, pole selline avaldis siiski hästi loetav; seepärast on parem kasutada sulgusid:

```
(b << 4) >> 8
```

Märgiga vasaku operandi korral sõltub paremale nihutamine realisatsioonist. Originaal-C kirjelduses ei konkretiseerita, kas vabanevad vasakpoolsed bitid täidetakse nullidega või paljundatakse vasakpoolseimat bitti. Enamikus realisatsioonides täiendkoodi kasutavatel arvutitel on rakendatud teist võimalust. Sel juhul on nihutamine paremale samaväärne kahe astmega jagamisega. Masinsõltumatuse saavutamiseks tuleb kasutada ainult märgita operandide paremale nihutamist.

Võrdlusoperatsioonide tehtemärkideks on <, <=, > ja >= :

```
võrdlusavaldis:
        nihkeavaldis
        võrdlusavaldis võrdlusop nihkeavaldis
võrdlusop: variandid
        < <= > >=
```

Operandidele rakendatakse harilikke binaarseid teisendusi. Operandid võivad olla kas mõlemad aritmeetilist tüüpi või siis mõlemad sama tüüpi viidad. Tulemus ei ole l-väärtus, tema tüüp on int ja väärtus kas 0 või 1.

Täisarvuliste operandide korral teostatakse (kas märgiga või märgita) täisarvuline võrdlemine, reaalarvuliste operandide korral aga reaalarvuline võrdlemine. Viitade korral sõltub tulemus viidatavate objektide paiknemisest aadressiruumis. Masinsõltumatu on selline võrdlemine vaid juhul, kui mõlemad viidad viitavad sama massiivi elementidele, kusjuures relatsiooni "suurem kui" mõistetakse kujul "vastaval objektil on massiivis suurem indeks".

Operatsioon < tähendab "väiksem", <= tähendab "väiksem või võrdne", > tähendab "suurem" ja >= tähendab "suurem või võrdne". Nende operatsioonide tulemuseks on 1 kui vastav relatsioon operandide vahel kehtib ja 0 vastasel korral.

Binaarsed võrdlusoperatsioonid on kõik sama prioriteediga ja vasakassotsiatiivsed. Seega on näiteks mõeldav avaldis kujul 3 < x < 7, kuid see ei ole matemaatiline kirjaviis - avaldis ei kontrolli siin, kas x on 3 ja 7 vahel. Operatsioonid täidetakse järjekorras (x < 3) < 7. Esimene võrdlus annab tulemuse 0 või 1, mis on seitsmest väiksem ja kogu avaldise väärtus tuleb seega alati 1. Soovides kontrollida kas x on 3 ja 7 vahel tuleb kasutada kaht võrdlust, mida võib siduda kas bitikaupa või loogilise korrutamise operatsiooniga, seega kas 3 < x & x < 7 või 3 < x && x < 7 .

Tuleb olla ettevaatlik võrdlusoperatsioonidega eri tüüpi operandide vahel. Vaatleme kurioosset näidet:

```
-1 < (unsigned) 0
```

Võiks arvata, et tulemus on 1 (tõene), sest -1 on ju väiksem kui 0. Et aga teine operand on märgita null, siis teisendatakse ka esimene operand märgita arvuks, tõenäoliselt suurimaks märgita väärtuseks. Seega on toodud avaldise väärtus 0.

Mõned realisatsioonid lubavad võrrelda viitu täisarvudega, taandades selle märgita või märgiga arvude võrdlemisele. Lubatavusele vaatamata tuleks neist võrdlustest hoiduda.

Võrdusoperatsioonid "võrdne" ja "mittevõrdne" eraldame ülejäänud võrdlustest, kuna nende prioriteet on madalam:

```
võrdusavaldis:
    võrdlusavaldis
    võrdusavaldis võrdusop võrdlusavaldis
võrdusop: variandid
    == !=
```

Operandidele rakendatakse harilikke binaarseid teisendusi. Tulemus pole l-väärtus, tema tüüp on int ja väärtus 1 või 0.

Operandid võivad olla kas mõlemad aritmeetilist tüüpi, mõlemad sama tüüpi viidad või siis üks operandidest on viit ja teine täisarvuline konstantavaldis väärtusega 0.

Täisarvuliste operandide korral teostatakse täisarvuline võrdlemine, reaalarvuliste operandide korral reaalarvuline võrdlemine. Viitade võrdlemine kontrollib, kas viidad osutavad samale objektile või on mõlema väärtus 0 (viida väärtus võrdub nulliga siis, kui viit ei osuta ühelegi objektile).

Operatsioon == kontrollib operandide võrdumist, operatsioon != aga mittevõrdumist. Tulemus on 1 siis, kui operandid rahuldavad vastavat tingimust ja 0 vastasel korral.

Operatsioone = ja == tuleb eristada: esimene neist on omistamisoperatsioon, teine aga kontrollib võrdumist. Eristamise tähtsus tuleneb sellest, et paljudes programmeerimiskeeltes tähistatakse võrdusoperatsiooni ühe võrdusmärgiga =. Halvaks programmeerimisstiiliks tuleb pidada omistamise kasutamist seal, kus "traditsiooniliselt" on võrdlus. Näiteks

```
if(x = next()) ...
```

võib olla korrektne, kuna lauses if on lubatud lihtsalt avaldis, seega ka omistamine. Siiski jääb kahtlus, kas pole tegemist trükiveaga ja tegelikult peaks olema kaks võrdusmärki. Selle kahtluse saab kõrvaldada kirjutades ilmutatult välja mittevõrdumise nulliga, mida lauses if tegelikult ka kontrollitakse. Seega samaväärne kuid arusaadavam on

```
if ( (x = next()) != 0) ...
```

Binaarsed võrdusoperatsioonid on vasakassotsiatiivsed. Seepärast ei saa avaldist x == y == 7 tõlgendada matemaatiliselt (et nii x kui y väärtus on 7). Antud avaldis täidetakse kujul (x == y) == 7. Siin esimese operatsiooni tulemus on kas 1 või 0, seega mitte kunagi 7 ja kogu avaldise väärtus on alati 0. Keerulisemaid võrdlusi saab moodustada loogiliste ja bitikaupa operatsioonide abil.

Bitikaupa loogilised operatsioonid on vasakassotsiatiivsed ning hõlmavad bitikaupa loogilise korrutamise (bitikaupa-AND), bitikaupa mitteekvivalentsi (bitikaupa-XOR) ja bitikaupa loogilise liitmise (bitikaupa-OR):

```
bitikaupa-AND-avaldis:
     võrdusavaldis
     bitikaupa-AND-avaldis & võrdusavaldis
bitikaupa-XOR-avaldis:
     bitikaupa-AND-avaldis
     bitikaupa-XOR-avaldis ^ bitikaupa-AND-avaldis
bitikaupa-OR-avaldis:
     bitikaupa-XOR-avaldis
     bitikaupa-OR-avaldis | bitikaupa-XOR-avaldis
```

Operandid peavad kõigis neis operatsioonides olema täisarvulised, neile rakendatakse harilikke binaarseid teisendusi. Tulemus ei ole 1-väärtus ja tema tüübiks on teisendatud operandide ühine tüüp.

Tulemuse iga bitt saadakse teisendatud operandide vastavate bittidega näidatud loogilist operatsiooni sooritades. Seega näiteks 0x53 & 0x62 väärtuseks on 0x42 , 0x14 ^ 0x12 väärtuseks 0x06 ja 0x12 | 0x14 väärtuseks 0x16.

Kõik bitikaupa loogilised operatsioonid on kommutatiivsed ja assotsiatiivsed. Kompilaator võib korduvaid ühesuguseid bitikaupa operatsioone ümber järjestada ka siis, kui ilmutatult on antud sulud. Näiteks avaldis a & (b & c) & d võidakse väärtustada järjekorras (c & a) & (b & d). Kui operandide väärtustamise järjekord avaldises on (näiteks kõrvalefektide tõttu) oluline, siis tuleb niisugused avaldised esitada mitme avaldisena koos vahepealsete omistamistega abimuutujaile.

Bitikaupa operatsioone võidakse vajaduse korral kasutada koos tõeväärtustega (0 või 1) konstrueerimaks keerulisemaid tingimusi. Näiteks

```
if(a < b & b < c) ...
```

Tegelikult tuleb siin aga eelistada loogilise korrutamise && kasutamist, eriti siis, kui pole kindel, kas operandide võimalikud väärtused on vaid 0 ja 1. Näiteks kui muutuja v väärtus on 2 ja muutuja p väärtus 4, siis v & p väärtus tuleb 0, kuid v && p väärtus 1.

Operatsioon | erineb loogilisest liitmisest || vaid selle poolest, et tema mõlemad operandid igal juhul väärtustatakse (kui loogilise liitmise || korral esimene operand on nullist erinev, siis teist ei väärtustata). Teisest küljest on loogilise liitmise || tulemus alati kas 0 või 1, mis aga operatsiooni | korral nii ei ole.

Et erinevates arvutites võivad märgiga täisarvud olla realiseeritud erinevalt, siis osutuvad bitikaupa operatsioonid märgiga täisarvude korral masinsõltuvateks. Sõltumatuse tagamiseks arvutist on seega soovitatav kasutada bitikaupa operatsioonides ainult märgita operande.

7.7. Loogilised avaldised

Loogiliste avaldiste moodustamisel kasutatakse binaarseid operatsioone && ja ||, mis on süntaktiliselt vasakassotsiatiivsed, kuid operatsiooni && prioriteet on kõrgem:

loogiline-AND-avaldis:
    bitikaupa-OR-avaldis
    loogiline-AND-avaldis **&&** bitikaupa-OR-avaldis
loogiline-OR-avaldis:
    loogiline-AND-avaldis
    loogiline-OR-avaldis || loogiline-AND-avaldis

Nende operatsioonide operandid võivad olla suvalist skalaarset tüüpi. Kahe operandi tüüpide vahel ei ole siduvaid tingimusi. Operatsiooni tulemus ei ole l-väärtus, ta on alati tüüpi int ja tema väärtus on kas 0 või 1.

Mõlema operatsiooni täitmisel väärtustatakse kõigepealt vasakpoolne operand. Kui loogilise korrutamise korral saadud väärtus on (operatsiooni == mõttes) null, siis parempoolset operandi ei väärtustata ja tulemuseks on 0. Kui vasakpoolse operandi väärtus erineb nullist, siis väärtustatakse ka parempoolne operand. Kui selle väärtus on omakorda 0, siis on null ka tulemus. Vastasel korral on tulemus 1. Kui loogilise liitmise korral vasakpoolse operandi väärtustamisel saadud väärtus erineb nullist, siis parempoolset operandi ei väärtustata ja tulemuseks on 1. Vastasel korral väärtustatakse parempoolne operand ja kui tema väärtus on 0, siis on 0 ka tulemus, vastasel korral on tulemus 1. Mõned näited:

a	b	a && b	a || b
1	0	0	1
0	-1	0	1
1	1	1	1
34.5	'\0'	0	1
0.0	'\0'	0	0
&x	&y	1	1
&x	0	0	1

Erinevalt bitikaupa operatsioonidest, on väärtustamise järjekord vasakult paremale siin garanteeritud.

7.8. Tingimusavaldis

Eraldajad ? ja : tähistavad tingimuslikku avaldist või ternaarset paremassotsiatiivset operatsiooni, mille prioriteet on madalam kui loogilistel operatsioonidel:

```
tingimusavaldis:
    loogiline-OR-avaldis
    loogiline-OR-avaldis ? avaldis : tingimusavaldis
```

Esimene operand määrab, kumb teisest või kolmandast operandist tuleb väärtustada. Kui esimese operandi väärtus erineb nullist (operatsiooni == mõttes), siis väärtustatakse ainult teine operand ja tulemuseks on teise operandi (võimalik, et teisendatud) väärtus. Kui esimese operandi väärtus on null, siis väärtustatakse ainult kolmas operand ja tulemuseks on kolmanda operandi (võimalik, et teisendatud) väärtus. Tingimusoperatsiooni tulemus ei ole l-väärtus.

Tingimusoperatsioon on paremassotsiatiivne oma esimese ja kolmanda operandi suhtes, seega avaldis

```
a ? b : c ? d : e ? f : g
```

arvutatakse kui

```
a ? b : (c ? d : (e ? f : g))
```

Kirjeldame näitena signum-funktsiooni, mis väljastab 1, -1 või 0, kui argument on positiivne, negatiivne või null:

```
int signum(x)
int x;
{
return (x > 0) ? 1 : (x < 0) ? -1 : 0;
}
```

Tingimusoperatsiooni teine operand võib olla suvaline avaldis, kuna sümbolipaar ? : eraldab selle sulgudena. Kolmas operand võib aga ilma sulgudeta sisaldada vaid kõrgema prioriteediga operatsioone (mõnes realisatsioonis kehtib see ka teise operandi jaoks). Vigane on näiteks avaldis

```
a ? b = c : c = b
```

Kolmanda operandi suhtes korrektne (ja teise operandi suhtes selgem) on anda need operandid sulgudes:

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

Tingimusoperatsiooni esimene operand võib olla suvalist skalaarset tüüpi. Teise ja kolmanda operandi valikuks on lubatud järgmised variandid.

1) Mõlemad operandid on aritmeetilist tüüpi. Sel juhul rakendatakse neile harilikke binaarseid teisendusi ja tulemuse tüübiks on teisendatud operandide ühine tüüp.

2) Mõlemad operandid on pärast teisendusi (näiteks massiiv -> viit) sama tüüpi viidad. Tulemuse tüübiks on seda ühist tüüpi viit.

3) Mõlemad operandid on ühte tüüpi (ka tüüpi struktuur, ühend või void). Tulemus on sel juhul sama tüüpi.

4) Üks operand on viit, teine aga konstantavaldis väärtusega 0. Tulemus on sel juhul viidatüüpi.

On kujunenud stiiliks panna esimene operand sulgudesse, kuigi see ei ole kohustuslik. Peab ühtlasi märkima, et mitte kõik realisatsioonid ei luba tingimusoperatsiooni operandidena struktuuri, ühendit ja tüüpi void.

7.9. Omistamisavaldised

Omistamine on keeles C operatsioon (mitte lause), millel on alati väärtus. Omistamisavaldis koosneb kahest operandist, mis on eraldatud omistamismärgiga. Operatsiooni = nimetame lihtomistamiseks, kõiki ülejäänuid liitomistamisteks:

```
omistamisavaldis:
        tingimusavaldis
        unaaravaldis omist-op omistamisavaldis
omist-op: variandid
        = += -= *= /= %= <<= >>= &= ^= |=
```

Omistamisoperatsioonid on kõik ühesuguse prioriteediga ja paremassotsiatiivsed. Seega avaldis x *= y = z arvutatakse kui x *= (y = z). Paremassotsiatiivsus võimaldab teistest programmeerimiskeeltest tuntud korduvaid omistamisi, näiteks

```
a = b = c = d + 7
```

Et seda avaldist tõlgendatakse kujul

```
a = (b = (c = d + 7))
```

siis omistab ta muutujatele a, b ja c väärtuse d + 7.

Omistamisoperatsiooni vasakpoolseks operandiks peab olema l-väärtus, millele omistatakse parempoolse operandi teisendatud väärtus, mis ühtlasi on ka operatsiooni tulemus ja ei ole ühegi omistamise korral l-väärtus.

Lihtomistamist tähistab omistamismärk = . Parempoolse operandi väärtus omistatakse vasakpoolsele operandile. Operandid võivad olla mõlemad aritmeetilist tüüpi, siis teisendatakse parempoolne operand vasakpoolse tüüpi. Operandid võivad olla ka sama tüüpi viidad, struktuurid või ühendid. Lõpuks võib vasakpoolne operand olla viidatüüpi, parempoolne aga konstantavaldis väärtusega 0. Viimane omistamine tähendab nullviida loomist, mis ei viita ühelegi objektile.

Omistamisoperatsiooni tulemuse väärtuseks ja tüübiks on teisendatud parempoolse operandi väärtus ja tüüp.

Omistamisoperatsiooniga ei saa omistada tervet massiivi teisele massiivile järgmistel põhjustel. Esiteks, massiivi nimi ei ole l-väärtus ega või seega asuda vasakul pool omistamismärki. Teiseks, kui massiivi nimi esineb paremal pool omistamismärki, siis teisendatakse ta tüüpi viit. Massiivi nime saab aga omistada viidale ja pärast omistamist viitab see massiivi esimesele elemendile, näiteks:

```
int a[10], *p;
p = a;
```

Küll on (mõneti kurioosselt) võimalik omistada massiivi massiivile, kui see massiiv on struktuuri komponendiks:

```
struct matrix { double contents[10][10]; };
struct matrix a,b;
...
a = b; /* selline omistamine on lubatud */
```

Originaal-C ja mitmed realisatsioonid ei luba struktuuride või ühendite omistamist. On ka realisatsioone, mis võimaldavad omistamist struktuuride kuid mitte ühendite korral.

Liitomistamisoperatsioon tähistusega op= tähendab, et avaldis a op= b on samaväärne avaldisega a = (a) op (b), kus avaldis a väärtustatakse vaid üks kord. Täpsemalt, liitomistamisel väärtustatakse vasak ja parem operand, kusjuures vasak peab olema l-väärtus. Seejärel pärast harilikke binaar-

seid teisendusi operatsiooni op tarvis leitakse selle operatsiooni tulemus, mis omistatakse vasaku operandiga määratud l-väärtusele (pärast omistamisele vastavaid teisendusi). Teisendatud väärtus jääb ka liitomistamise väärtuseks.

Operatsioonide += ja -= korral peavad kas mõlemad operandid olema aritmeetilist tüüpi või siis on vasakpoolne operand viidatüüpi ja parempoolne täisarvutüüpi.

Operatsioonide *= ja /= korral peavad mõlemad operandid olema aritmeetilist tüüpi.

Operatsiooni %= korral peavad mõlemad operandid olema täisarvutüüpi.

Operatsioonide <<= ja >>= korral peavad mõlemad operandid olema täisarvutüüpi. Masinsõltumatuse saavutamiseks on soovitatav, et vasakpoolne operand oleks märgita tüüpi.

Operatsioonide &=, ^= ja |= korral peavad mõlemad operandid olema täisarvutüüpi. Masinsõltumatuse saavutamiseks on soovitatav, et operandid oleksid märgita tüüpi. Näiteid:

```
int a;
double b;
...
a += 5;
b /= 2.0;
```

7.10. Komaavaldis

Komaavaldis koosneb kahest avaldisest, mille vahel on koma. Operatsioon on süntaktiliselt vasakassotsiatiivne:

```
komaavaldis:
        omistamisavaldis
        komaavaldis , omistamisavaldis
avaldis:
        komaavaldis
```

(viimane definitsioon märgib, et komaavaldis lõpetab meie süntaksivalemid kogu avaldise jaoks).

Komaavaldise väärtustamisel arvutatakse esmalt vasakpoolse operandi väärtus ning seejärel parempoolse operandi väärtus. Operatsiooni tulemus ei ole l-väärtus, tema tüübiks

ja väärtuseks jääb parempoolse operandi tüüp ja väärtus. Komaavaldis võimaldab tõlgendada komadega eraldatud avaldiste järjendit kui ühte avaldist, mille tüübiks ja väärtuseks on kõige parempoolsema operandi tüüp ja väärtus (eelmiste operandide tähtsus seisneb nende kõrvalefektis).

Keeles C kasutatakse sümbolit koma ka mõnedes teistes kontekstides. Mitmesuse vältimiseks tuleb sellistel juhtudel komaavaldis panna sulgudesse. Näiteks avaldist

```
f(a, b = 5, 2 * b, c)
```

käsitletakse kui nelja parameetriga funktsiooni poole pöördumist. Kui teine koma on mõeldud komaavaldise koosseisus, siis tuleb see avaldis panna sulgudesse, seega

```
f(a, (b = 5, 2 * b), c)
```

Nüüd on tegemist kolme parameetriga funktsiooni poole pöördumisega, kus teine tegelik parameeter on komaavaldis.

Komaavaldist ei saa kasutada ilma sulgudeta järgmistes kontekstides: funktsiooni tegelike parameetrite loetelu, bitivälja pikkus struktuurikomponendi kirjelduses, loendikonstandi väärtus loenditüübi kirjelduses ning algväärtus algväärtuste loetelus. Meenutame, et koma on parameetrieraldajaks ka preprotsessori makros. Erinevalt koma muudest kasutusviisidest garanteerib komaavaldis, et operandid väärtustatakse vasakult paremale (näiteks funktsiooni tegelike parameetrite loetelu seda ei taga).

Anname komaavaldise näitena programmilõigu, mis pöörab massiivi ringi - vahetab esimese ja viimase elemendi, teise ja eelviimase jne. (komaavaldised on lause for koosseisus):

```
void revers(array,n)
int array[],n;        /* n on massiivi elementide arv */
{
int i,j,temp;
if(n < 2) return;
for(i = 0, j = n - 1; i < j; ++i, --j)
  {
  temp=array[i]; array[i] = array[j]; array[j] = temp;
  }
}
```

7.11. Konstantavaldised

Keeles C kasutatakse mitmetel juhtudel konstantavaldist, s.t. avaldist, mille väärtuse saab leida kompileerimisajal. Niisugusteks juhtudeks on:

1) kontrollitav väärtus preprotsessoridirektiivis #if;
2) massiivi raja;
3) märgendi case väärtused lauses switch;
4) bitivälja pikkus;
5) loendikonstandi väärtus;
6) algväärtus staatilisele või välismuutujale.

Igas sellises kontekstis on tingimused konstantavaldise kuju jaoks veidi erinevad. Kõigil juhtudel kehtib aga tingimus, et konstantavaldise väärtus on identne sama avaldise väärtusega täitmisajal (see on oluline eriti nn. cross-kompilaatorite korral, kus kompileerimisaja arvutisüsteem võib erineda täitmisaja arvutisüsteemist).

Iga konstantavaldise koosseisu võivad kuuluda täisarvja sümbolkonstandid. Kasutada võib binaarseid operatsioone * / % + - << >> == != < <= > >= & ^ | && || unaarseid operatsioone - ~ ! ning tingimusoperatsiooni ? : Operandide rühmitamiseks võib kasutada sulgusid.

Ülejäänud konstanditüüpide ja operatsioonide kasutamine konstantavaldises sõltub kontekstist.

1) Preprotsessoridirektiivis #if võib kasutada operatsiooni defined.

2) Massiivi raja, märgendi case, välja pikkuse ja loendikonstandi väärtuse esitamisel võib kasutada loendikonstante, operatsiooni sizeof ja ilmutatud tüübiteisendusi täisarvutüüpi (operatsiooni sizeof operandiks ei pea olema konstantavaldis, kuna oluline on vaid selle avaldise kompileerimisajal määratav tüüp).

3) Staatiliste ja välismuutujate algväärtustes võib lisaks kasutada veel reaalarvkonstante ja suvalisi tüübiteisendusi. On lubatud unaarse operatsiooni & (aadressi leidmine) rakendamine staatilisele või välismuutujale, samuti massiivi elemendile. Võib kasutada ka funktsioonide ja massii-

vide nimesid (mis väljendavad vastavate objektide aadresse) ning sõnesid. Semantiliselt on staatilise või välismuutuja algväärtus kas konstant või varem defineeritud staatilise või välisobjekti aadress pluss või miinus konstant.

Originaal-C ei luba konstantavaldises eitusoperatsiooni. Mõnes realisatsioonis pole lubatud kasutada ilmutatud tüübiteisendusi, mõnes seevastu on lubatud komaavaldise kasutamine (see on mõnevõrra veider, kuna komaavaldise vasaku operandi mõte on kõrvalefektis, konstantavaldisel aga kõrvalefekti olla ei saa).

Konstantavaldise süntaks langeb kokku avaldise süntaksiga, välja arvatud kõrvalefektiga operatsioonid. Kitsendused konstantavaldises on semantilised, mitte süntaktilised.

7.12. Väärtustamise järjekord

Operatsioonide käsitlemisel nimetasime, et kompilaator võib avaldise operande ümber järjestada ja ka vabalt valida väärtustamise järjekorra. Siin kehtivad järgmised reeglid.

Funktsiooni tegelikke parameetreid ja binaarse operatsiooni operande võib kompilaator väärtustada suvalises järjekorras. Mõningad kompilaatorid valivad siin küll järjekorra vasakult paremale, kuid seda ei või näiteks kõrvalefektideks ära kasutada - vastavad võtted on realisatsioonist sõltuvad. Lisaks eeldatakse, et binaarsed operatsioonid +, *, &, ^, | on kommutatiivsed ja assotsiatiivsed matemaatilises mõttes ning kompilaator võib seda omadust arvestada. Näiteks kui muutujad a, b, c ja d on kõik sama aritmeetilist tüüpi, siis võib kompilaator avaldise (a + b) + (c + d) väärtustada suvalises järjekorras, kasvõi (a + d) + (b + c).

Üldiselt on matemaatiline assotsiatiivsus ja kommutatiivsus arvutis kehtiv märgita operandide korral. See ei pruugi aga kehtida märgiga operandide korral, näiteks liidetavate summeerimine ühes järjekorras võib kaasa tuua ületäitumise, teises järjekorras aga mitte. Kompilaatorid seda asjaolu muidugi ei kontrolli. Juhul kui väärtustamise järjekord on oluline, tuleb kasutada abimuutujaid:

```
{
int a, b, c, d, q;
int temp1, temp2;
...
/* leiame q = (a+b)+(c+d) nimelt selles järjekorras */
temp1 = a + b;
temp2 = c + d;
q = temp1 + temp2;
... }
```

Väga oluliseks tuleb pidada kitsendust, mille kohaselt siis, kui mingiks assotsiatiivseks ja kommutatiivseks operatsiooniks ettevalmistamine toob kaasa (vastavalt harilikele binaarseile teisendustele) operandide tüübiteisendused, ei või kompilaator enam operande suvaliselt ümber järjestada. Näiteks kaks järgmist avaldist ei ole samaväärsed:

```
x = (1.0 + -3) + (unsigned) 1; /* tulemus on -1.0 */
x = 1.0 + (-3 + (unsigned) 1); /* tulemus on suur */
```

Esimene avaldis annab tulemuseks -1.0. Teises avaldises aga teisendatakse -3 märgita arvuks 2^n-3, kus n on märgita tüübi bittide arv, liidetakse 1 ja tulemus teisendatakse reaalarvuks. Lõpptulemuseks on reaalarv 2^n-1 . Loomulikult ei pruugi tulemus olla programmeerija poolt soovitud ja seega peab kompilaator nii suuri erinevusi tulemuste vahel arvestama.

Nagu öeldud, on funktsiooni tegelike parameetrite väärtustamise järjekord suvaline, kuid ühe parameetri väärtustamisel peab kompilaator selle väärtustama täielikult, seejärel väärtustama täielikult mingi järgmise parameetri ja nii edasi, kuni kõik parameetrid on väärtustatud. Samad kitsendused kehtivad ka binaarse operatsiooni operandide korral - operandid võidakse väärtustada suvalises järjekorras, kuid kogu operand täielikult. Kõrvalefektidega avaldises on selline kitsendus oluline. Vaatame järgmist näidet:

```
char *x[10], **p = x;
...
if(strcmp(*p++, *p++) == 0) printf("Sõned võrduvad");
```

Funktsioon strcmp võrdleb kaht sõnet. Lause if võrdleb, kas sõnemassiivi kaks järjestikust elementi võrduvad omavahel.

Siin on oluline, et funktsiooni üks parameeter oleks täielikult välja arvutatud (tehtud nii kaudsustamine * kui ka kõrvalefekt ++) enne, kui hakatakse väärtustama järgmist.

7.13. Tarbetud väärtused

Kolmes kontekstis võivad tekkida sellised väärtused, mida edaspidi ei kasutata. Nendel juhtudel ütleme, et väärtused on tarbetud. Niisugusteks juhtudeks on:

1) avaldislause;
2) komaavaldise esimene operand;
3) algväärtustus ja sammuavaldised lauses for.

Kui tarbetud väärtused tekivad kõrvalefektita avaldises, siis võib kompilaator seda pidada veaks. Kõrvalefektid tekivad avaldistes, kus on tegemist funktsiooni poole pöördumiste ja omistamistega. Mõned näited:

```
extern void f();
f(x);    /* need avaldised on kõrvalefektiga ja */
i++;     /* seega ei kutsu esile veateadet */
a = b;
```

Järgmistes avaldistes aga esinevad tarbetud väärtused:

```
extern int g();
g(x); /* funktsiooni poole pöördumisel võib olla kõr-
      valefekt, kuid funktsiooni väärtus on tarbetu */
x + 7;     /* kõrvalefekt puudub */
x + (a *= 2); /* sulgudes oleval avaldisel on kõrval-
        efekt, kuid kogu avaldise väärtus on tarbetu */
```

Programmeerija võib ilmutatud tüübiteisenduse abil tüüpi void teatada kompilaatorile tarbetutest väärtustest. See on mõistlik funktsioonide korral, mille väärtusi võidakse mõnes kontekstis kasutada, mõnes aga mitte:

```
extern int g();
(void) g(x);
```

Kompilaatorid, mis ei tunnista andmetüüpi void, ei anna ka vigu tarbetute väärtuste korral. Oluline on siinjuures see, et kui kompilaator avastab tarbetu väärtuse, siis võib ta selle arvutamiseks vajalikku koodi mitte genereerida.

8. L A U S E D

8.1. Lausete liigid

Keeles C kasutatakse paljudest programmeerimiskeeltest tavapäraseks saanud komplekti lauseid, nagu tingimuslik lause, tsüklid ja ka otsene suunamine goto:

```
lause:
        avaldislause
        märgendatud-lause
        blokk
        tingimuslik-lause
        tsükkel
        lüliti
        katkestamislause
        jätkamislause
        naasmislause
        suunamislause
        tühilause
```

Laused keeles C sarnanevad küll enamike Algoli-laadsete programmeerimiskeelte lausetega, kuid võib siiski rõhutada mõningaid iseärasusi. Nagu keeltes Pascal ja Ada on järjestikuste lauseto eraldajaks keeles C semikoolon. Seejuures ei ole semikoolon aga mitte üldiselt lause lõpu tunnuseks vaid konkreetsete lausete süntaktiliseks koostisosaks. Näiteks blokk, mis on eraldatud loogeliste sulgudega { ... } ei vaja lõppu semikoolonit. Keeles Pascal kirjutatud laused

```
t := b;
begin b := a end;
a := t;
```

esitatakse keeles C kujul

```
t = b;
{ b = a; }
a = t;
```

Semikoolon järgneb omistamisele b = a; ja mitte sulule }.

Teiseks keele C iseärasuseks on see, et tingimust määrav avaldis (näiteks tingimuslikus lauses) tuleb paigutada sul-

gudesse. Nimetame niisuguseid avaldisi kontrollavaldisteks ehk ka tingimusteks. Sulgudesse asetamine võimaldab loobuda spetsiaalsetest eraldavatest võtmesõnadest nagu then või do. Näiteks programmifragment, mis keeles Pascal on antud kujul

```
if x = y then
  while x <> z do
    process(x);
```

näeb keeles C välja järgnevalt:

```
if (x == y)
  while (x != z)
    process(x);
```

Kõikide kontrollavaldiste korral tõlgendatakse väärtust 0 kui tõeväärtust FALSE ja nullist erinevat väärtust kui TRUE. Kontrollavaldise e tüüp peab olema selline, et avaldis (e) != 0 oleks lubatav ja väärtustatav. Kui avaldise väärtus on 1, siis öeldakse, et tingimus on täidetud, kui aga 0, siis mitte. Avaldis e võib olla täisarvu-, reaalarvu- või viidatüüpi, mõnes realisatsioonis ka loenditüüpi.

8.2. Avaldislause

Suvaline avaldis, millele järgneb semikoolon, on lause:

```
avaldislause:
        avaldis ;
```

Avaldislause täitmine seisneb avaldise väärtustamises, kusjuures avaldise väärtus on tarbetu. Seepärast on avaldislauses loomulikud kõrvalefektid nagu funktsiooni poole pöördumine, omistamine, suurendamine jms. Mõned näited:

```
speed = distance/time;/* omistamine muutujale speed */
++count;              /* muutuja count suurendamine */
printf("Type any key \n"); /* väljundoperatsioon */
(x < y) ? ++x: ++y;   /* tingimuslik suurendamine */
```

Viimane näide on küll korrektne, kuid traditsiooniliselt kasutatakse sel juhul lauset

```
if(x < y) ++x; else ++y;
```

Kompilaator võib avaldise tarbetu väärtuse arvutamiseks koodi mitte genereerida (vt. lk. 127).

8.3. Märgendatud laused

Märgend on konstruktsioon, mida kasutatakse lauses märkimaks, et sellele lausele saab anda juhtimise kas suunamislause või lülitiga. Märgendeid on kolme liiki. Märgend-nimi on kasutatav igas lauses ja ta seostatakse suunamislausega. Märgendid case ja default on kasutatavad vaid lülitis:

```
märgendatud-lause:
        märgend : lause
märgend:
        märgend-nimi
        märgend-case
        märgend-default
märgend-nimi:
        nimi
märgend-case:
        case avaldis
märgend-default:
        default
```

Märgend ei ole eraldi konstruktsioon vaid esineb alati mingi lause koosseisus. Kui märgend osutub vajalikuks seal, kus lauset pole, siis võib kasutada tühilauset (näiteks suunamisel bloki lõppu). Märgend-nime vaatleme lähemalt suunamislause, märgendeid case ning default lüliti käsitlemisel.

8.4. Blokk

Blokk ehk liitlause koosneb (võimalik, et tühjast) kirjelduste osast, millele järgneb (võimalik, et tühi) lausete osa. Konstruktsioon asub loogelistes sulgudes (vt. lk. 35).

Blokk võib programmis esineda suvalisel kohal, kus võib esineda lause. Kui blokis puuduvad kirjeldused, siis on ta lihtsalt lausete järjend. Sisekirjelduste korral määrab blokk ühtlasi temas kirjeldatavate nimede skoobi.

Bloki täitmine algab kirjelduste järjestikuse täitmisega (mälu eraldamine vms.), millele järgneb lausete järjestikune täitmine kuni bloki lõpuni. Juhtimine võib blokist väljuda

ka suunamis-, naasmis-, katkestamis- või jätkamislause toimel. Samuti saab blokki siseneda suunamislause või lülitiga, kuid sellisel sisenemisel jäävad blokis defineeritavad automaat- ja registrimuutujad algväärtustamata, kuigi kirjeldustes võivad algväärtused esineda. Bloki täitmine algab sel juhul lausest, millele juhtimine anti ja lõpeb tavaliselt.

Kui märgendamata blokk on lüliti sisuks, siis saab seda täita ainult suunamisega märgendile. Seega niisuguses blokis kirjeldatud automaat- ja registrimuutujaid ei algväärtustata kunagi ning järelikult pole neil algväärtustel ka mõtet.

Blokis kirjeldatud nime skoop ulatub vastava nime kirjeldamispunktist kuni bloki lõpuni. Selline nimi on nähtav kogu oma skoobi ulatuses, kui teda ei varja sama nime teine kirjeldus mingis sisemises blokis.

Iga blokis ilma mäluklassi näitamata kirjeldatud nimi kuulub tüübi funktsioon korral mäluklassi extern ja kõigil muudel juhtudel mäluklassi auto. Mäluklass extern on ühtlasi ainuke lubatav mäluklass, mida võib kasutada blokis kirjeldatud funktsioonide korral.

Kui blokis deklareeritakse muutuja mäluklassiga extern, siis muutujale mälu antud kohal ei eraldata ja tema algväärtustamist ei lubata. Vastava nime definitsioon peab esinema kusagil mujal, võimalik et mõnes teises lähtefailis.

Kui blokis defineeritakse muutuja mäluklassiga register või auto, siis eraldatakse talle uuesti mälu igal blokki sisenemisel ja vabastatakse blokist väljumisel. Muutujaile antud algväärtused arvutatakse uuesti igal normaalsel sisenemisel blokki. Algväärtuste puudumisel saavad need muutujad kontseptuaalselt defineerimata algväärtuse, samuti ei säili nende muutujate väärtus ühest blokki sisenemisest teiseni.

Kui blokis defineeritakse muutuja mäluklassiga static, siis nagu kõigile staatilistele muutujatele eraldatakse talle mälu vaid üks kord, programmi täitmise alguses. Kui muutujal leidub algväärtus, siis algväärtustamine toimub samuti vaid programmi täitmise alguses. Staatilised muutujad säilitavad oma väärtuse ka väljaspool antud blokki, s.o. blokki pöördumiste vaheajal ja ühest blokki pöördumisest teiseni.

Ilma kirjeldusteta blokke kasutatakse laialdaselt mitmesuguste juhtimiskonstruktsioonide koosseisus, näiteks kui tingimuslikult või tsüklis tuleb täita mitu lauset:

```
if(error)
  {
  ++error_count;
  print_error_message();
  }
```

Kirjeldustega blokk toob lisaks sisse piiratud nähtavusega lokaalsed muutujad. Selline võte võib teha programmi paremini loetavaks ja arusaadavamaks:

```
if (first) /* tühjendame massiivi */
  {
  int i;
  for (i = 0; i < 10; ++i)  a[i] = 0;
  first = 0;    /* kustutame lipu väärtuse */
  }
```

Keeles C on lubatud ka tingimusteta suunamine bloki sisse, kuid seda tuleb pidada halvaks programmeerimisstiiliks (paljudes keeltes on need suunamised keelatud). Lisaks tekivad ka raskused algväärtustega. Vaatleme järgmist näidet:

```
{
extern int a[100];
int i, sum = 0;
L: for(i = 0; i < 100; ++i)
   sum += a[i];
... }
```

Kui siin antakse juhtimine vahetult märgendile L, siis jääb muutuja sum algväärtustamata ja programm töötab valesti.

8.5. Tingimuslik lause

Tingimuslik lause esineb kahel kujul: kas koos haruga else või ilma. Mõlemal juhul algab lause võtmesõnaga if, millele järgneb tingimus (avaldis sulgudes) ja lause. Edasi võib mittekohustuslikult esineda võtmesõna else ning lause. Märgime, et keeles C puudub võtmesõna then.

tingimuslik-lause:

if (avaldis) lause1 else lause2

Tingimusliku lause täitmisel väärtustatakse kõigepealt sulgudes olev avaldis. Kui tema väärtus on nullist erinev, siis täidetakse lause1, vastasel korral jäetakse lause1 vahele ja kui esineb else, siis täidetakse lause2. Kui else puudub ja avaldise väärtus on 0, siis läheb juhtimine tingimuslikule lausele vahetult järgnevale lausele.

Üsna levinud on tingimuslike lausete järjestikune esitamine, nii et võtmesõnale else järgneb omakorda tingimuslik lause. See konstruktsioon sarnaneb sisuliselt lülitile:

```
if (avaldis1) lause1
  else if (avaldis2) lause2
    ...
    else lauseN
```

Toome veel kord näitena signum-funktsiooni, mille väärtuseks on 1, -1 või 0 sõltuvalt sellest, kas argumendi väärtus on positiivne, negatiivne või null (vt. ka lk. 119):

```
int signum(x)
int x;
{
if (x > 0) return 1;
else if(x < 0) return -1;
else return 0;
}
```

Et tingimusliku lause koosseisus võib jälle esineda tingimuslik lause, siis tekib küsimus, millise if juurde kuulub mingi else. Näiteks vaadeldes programmilõiku:

```
if((k >= 0) && (k < TABLE_SIZE))
  if( table[k] >= 0)
  printf("Element indeksiga %d on %d\n", k, table[k]);
  else
  printf("Viga, indeks %d masiivist väljas\n", k);
```

võib veateatest järeldada, et else kuulub loogiliselt kokku välimise lausega if. Mitmesuse vältimiseks eeldatakse keeles C aga, et else kuulub alati lähima if juurde, kus veel puudub else. Seega tuleb seesmine if paigutada blokki:

```
if((k >= 0) && (k < TABLE_SIZE))
  {
  if( table[k] >= 0)
  printf("Element indeksiga %d on %d\n", k, table[k]);
  }
  else
  printf("Viga, indeks %d masiivist väljas\n", k);
```

8.6. Tsüklid

Keeles C leiduvad järgmised tsüklilaused ehk tsüklid:

tsükkel:
- tsükkel-while
- tsükkel-do
- tsükkel-for

Tsükli while korral kontrollitakse kordamistingimust tsüklisammu alguses, tsükli do korral tsüklisammu lõpul. Tsükkel for võimaldab defineerida tegevused tsükliks ettevalmistamisel, kordamistingimused ja ühelt sammult teisele üleminekul sooritatavad tegevused. Tsüklilause koosseisu kuulub alati mingi teine lause, mida nimetame tsükli sisuks.

Tsükkel while koosneb võtmesõnast while, millele järgneb avaldis sulgudes ja lause:

tsükkel-while:

while (avaldis) lause

(erinevalt keelest Pascal ei kasutata siin võtmesõna do).

Selle tsükli täitmisel väärtustatakse kõigepealt sulgudes olev avaldis (tingimus). Kui saadud väärtus erineb nullist, siis täidetakse tsükli sisuks olev lause ja kogu protsess kordub. Kui tingimuse väärtus on 0, siis antakse juhtimine tsüklile vahetult järgnevale lausele. Erinevatel väärtustamistel võib tingimuse väärtus olla erinev, kas avaldise enda kõrvalefekti või tsükli sisu täitmise tulemusel.

Tsükli while täitmine lõpeb siis kui tingimuse väärtus on 0. Juhtimise võib tsükli sisust välja anda ka naasmis-, suunamis- või katkestamislausetega. Jätkamislausega saab tsükli täitmist modifitseerida.

Vaatleme näitena funktsiooni power täisarvulise muutuja x tõstmiseks mittenegatiivsele täisarvulisele astmele y:

```
int power(x,y)
int x,y;
{
int base = x;
int exponent = y;
int z = 1;
while (exponent > 0)
   {
   if(exponent % 2) /* kas exponent on paaritu */
     z *= base;     /* korrutame z ja base */
   base *= base;    /* tõstame base ruutu */
   exponent /= 2;   /* jagame exponent kahega */
   }
return z;           /* z väärtus on nüüd x astmel y */
}
```

Mõnikord võib tsükli sisuks olla ka tühilause. Näiteks järgmine fragment leiab sõne lõpuaadressi:

```
while (*char_pointer++) ;
```

Viita tõstetakse edasi seni, kuni leitakse väärtus 0, mis ongi sõne (ja ka tsükli) lõputunnuseks. Samamoodi võib sõne ühest kohast teise kopeerida:

```
while (*dest_ptr++ = *source_ptr++) ;
```

Sümboleid kopeeritakse siin kuni lõpetava nullini, mis samuti kopeeritakse. Programmeerija peab muidugi hoolitsema, et tulemusväli oleks piisavalt pikk sõne mahutamiseks.

Tsükkel do koosneb võtmesõnast do, millele järgneb lause, võtmesõna while, avaldis sulgudes ja semikoolon:

```
tsükkel-do:
     do lause while ( avaldis ) ;
```

Selle tsükli täitmisel täidetakse kõigepealt võtmesõnale do järgnev lause (tsükli sisu). Seejärel väärtustatakse sulgudes olev tingimus. Kui selle väärtus erineb nullist, siis protsess kordub. Kui aga väärtus on 0, siis antakse juhtimine tsüklile vahetult järgnevale lausele. Tingimuse väärtus võib ka siin olla erinevatel väärtustamistel erinev.

Tsükli do täitmine lõpeb siis, kui tingimuse väärtus on 0. Juhtimise võib tsükli sisust välja anda ka naasmis-, suunamis- või katkestamislausetega. Jätkamislause abil saab tsükli sisuks oleva lause täitmist modifitseerida.

Tsükli do põhiliseks erinevuseks tsüklist while on tingimuse väärtustamine tsüklisammu lõpul. Tsükli sisu täidetakse siin seega vähemalt üks kord, kuid tsüklis while võidakse tsükli sisu ka mitte täita.

Tsükkel do keeles C sarnaneb teiste programmeerimiskeelte (nagu Pascal) konstruktsioonile repeat - until. Erinevalt keelest Pascal lõppeb tsükli täitmine siin aga siis, kui tingimus on väär, mitte siis kui ta on tõene. See ühtlustab tsükli do tsüklitega while ja for.

Näitena toome programmilõigu, mis loeb ja töötleb sümboleid ning lõpetab töö siis, kui on töödeldud sümbol realüke:

```
int ch;
do {
   ch = getchar();
   process(ch);
   }
while (ch != '\n');
```

Sama näite saab vähem ilmekalt esitada ka tsükli while abil:

```
int ch;
while(ch = getchar(ch),process(ch), ch != '\n');
```

Ka tsükli do sisuks võib olla tühilause:

```
do ; while  (avaldis);
```

Et aga see lause on samaväärne lausega

```
while (avaldis) ;
```

siis tühja sisuga tsüklit do praktiliselt ei kasutata.

Tsükkel for on keeles C mõnevõrra üldisem teistes programmeerimiskeeltes kasutatavatest iteratsioonitsüklitest:

tsükkel-for:

for (avaldis1 ; avaldis2 ; avaldis3) lause

Tsükkel for algab võtmesõnaga for, millele järgneb kolm semikoolonitega eraldatud mittekohustuslikku avaldist sulgudes ja lõpuks lause. Kuigi avaldised on mittekohustuslikud, peavad eraldavad semikoolonid ja sulud igal juhul esinema.

Tüüpilise kasutuse korral initsialiseerib esimene avaldis tsüklimuutuja, teine on kordamistingimus ja kolmas muudab tsüklimuutuja väärtust (näiteks suurendab mingi sammu võrra). Semantiliselt võib esimene ja kolmas avaldis olla suvaline, teine aga selline, mida on mõtet võrrelda nulliga.

Tsükli for täitmine toimub järgnevate sammude kaupa.

1) Kui leidub esimene avaldis, siis ta väärtustatakse, kusjuures avaldise väärtus ise on tarbetu.

2) Kui leidub teine avaldis, siis ta väärtustatakse. Kui saadud väärtus on 0, siis sellega lõpeb ka kogu tsükli täitmine. Kui väärtus erineb nullist või kui teine avaldis puudub, siis jätkatakse tsükli täitmist.

3) Täidetakse sulgudele järgnev lause - tsükli sisu.

4) Kui leidub kolmas avaldis, siis ta väärtustatakse, kusjuures tema väärtus on tarbetu.

5) Tsükli for täitmist jätkatakse sammust 2.

Tsükli for täitmine lõpeb siis, kui avaldis2 annab väärtuse 0. Juhtimise võib tsükli sisust välja anda ka naasmis-, suunamis- või katkestamislausetega. Jätkamislausega võib tsükli sisuks oleva lause täitmist modifitseerida.

Tsükkel for on (välja arvatud reaktsioon jätkamislausele) samaväärne järgmise programmilõiguga:

```
{
avaldis1;
while (avaldis2) {
                 lause
                 avaldis3;
                 }
}
```

Kui tsüklis puudub avaldis1 või avaldis3, siis nad jäetakse vahele ka selles skeemis. Kui puudub avaldis2, siis loetakse kordamistingimus täidetuks ja seega võib avaldis2 asemel kasutada nullist erinevat konstanti, näiteks while(1).

Üldkasutatavaks "lõputuks tsükliks" on for(;;) ... Muidugi saab ka seda tsüklit katkestada naasmis-, suunamis- või katkestamislausete abil. Lõputu tsüklina on samuti kasutatav ka while (1) ... , kuid tsükkel for on enamlevinud.

Tsükli for tüüpilise kasutamise näitena toome fragmendi, mis trükib täisarvud 0 - 9 ja nende ruudud:

```
for(j = 0; j < 10; ++j)
  printf("%d  %d\n",j ,j * j);
```

Teise näitena kirjutame tsükli for abil ümber funktsiooni power (vt. lk. 135) täisarvude astendamiseks:

```
int power(x,y)
int x,y;
{
int base = x, exponent, z = 1;
for(exponent = y; exponent > 0; exponent /= 2)
   {
   if(exponent % 2)  z *= base;
   base *= base;
   }
return z;
}
```

Tsüklimuutuja exponent on kirjeldatud väljaspool tsüklit, sest tsüklis for ei saa olla kirjeldusi. Kirjeldused võivad esineda küll tsükli sisu moodustava bloki koosseisus, kui aga muutuja exponent kirjeldada selles blokis, siis pole ta sulgudes olevates avaldistes veel määratud.

Tsüklimuutujaks ei pruugi olla tingimata täisarvuline muutuja. Järgmises näites juhitakse tsüklit viida abil:

```
struct list { struct list *link; int data; } ;
void print_dub(p)
struct list *p;
{
for(; p; p = p->link)
   {
   struct list *q;
   for(q = p->link; q; q = q->link)
      if(q->data == p->data)
        { printf("Korduv väärtus %d\n",p->data);
        break; }
```

Struktuur list määrab lineaarse ahela, mis sisaldab ühe seosevälja - viida järgmisele elemendile ja mingi sisu. Funktsioon print_dub trükib välja sisu korduvad väärtused, s.t. väärtused, mis kordavad juba leitud väärtusi. Esimene tsükkel kasutab tsüklimuutujana funktsiooni parameetrit p - viita ahela elemendile, mis muutub üle kogu ahela. Iga elemendi korral vaatab sisemine tsükkel ahela üle antud punktist kuni lõpuni. Kui leitakse korduv väärtus, siis trükitakse ta välja ning sisemine tsükkel katkestatakse.

Näitena üksteisesse sisestatud tsüklitest for toome funktsiooni, mis järjestab täisarvuliste elementidega massiivi mittekahanevasse järjekorda:

```
/* massiivi v[0], ... ,v[n-1] järjestamine */
void instsort(v, n)
register int v[], n;
{
register int i, j, temp;
if(n < 2) return;
for(i = 1; i < n; ++i)
   {
   temp = v[i];
   for(j = i - 1; j >= 0 && v[j] > temp; --j)
      v[j+1] = v[j];
   v[j+1] = temp;
   }
}
```

Välimine tsükkel täidetakse muutuja i kasvavatel väärtustel 1 kuni n-1. Igal sammul on elemendid v[0] kuni v[i-1] juba järjestatud. Sisemine tsükkel, mida täidetakse muutuja j kahanevatel väärtustel alates väärtusest i-1, asetab v[i] temale kuuluvale kohale. Sisemise tsükli kordamistingimuse kontrollimiseks kasutatakse loogilist operatsiooni &&.

Siin esitatud järjestusmeetod on lihtne ja piisavalt efektiivne väikeste massiivide (umbes kuni 20 elementi) järjestamiseks. Suurte massiivide jaoks jääb ta aeglaseks, kuna algoritmi töötamisaeg on võrdeline massiivi elementide arvu ruuduga. Mõningase modifikatsiooniga, tuues sisse veel kol-

manda täiendava tsükli, saame tunduvalt efektiivsema algoritmi. Järgmine programmilõik realiseerib selle nn. Shelli järjestusmeetodi:

```
/* massiivi v[0], ... ,v[n-1] järjestamine */
void shellsort(v, n)
register int v[], n;
{
register int gap, i, j, temp;
gap = 1;
/* töötame välja kontrollimissammu */
do gap = 3*gap +1; while(gap <= n);
for(gap /= 3; gap > 0; gap /= 3)
   for(i = gap; i < n; ++i)
      {
      temp = v[i];
      for(i = gap; (j >= 0) && (v[j] > temp); j -= gap)
         v[j+gap] = v[j];
      v[j+gap] = temp;
      }
 }
```

Siin on algoritm esitatud veidi modifitseeritud kujul (D. Knuth), algselt oli muutuja gap algväärtus n/2 ja ta jagati igal sammul kahega. Katsed näitasid, et kolmega jagamine on efektiivsem.

Peab märkima, et Shelli meetodis on kaks sisemist tsüklit praktiliselt identsed eelmise meetodiga. Vahe on siin selles, et järjekordne element liigub oma kohale suuremate hüpetega, hüppe suuruse määrab gap. Katsed tõestasid, et Shelli järjestusmeetodi kiirus on n^2 asemel järku $n^{1.25}$, kus n on järjestatavate elementide arv.

Mõnikord on kasulik juhtida tsüklit mitme muutuja abil. Tavaline on sel juhul komaoperatsiooni kasutamine, mis võimaldab omistada mitmele muutujale väärtuse ühes avaldises:

```
/* väljastada 1, kui kaks argumendina antud sõnet on
   võrdsed ja 0 vastasel korral */
int string_eq(s1, s2)
char *s1, *s2;
```

```
{
char *p1, *p2;
for( p1 = s1, p2 = s2; *p1 && *p2; ++p1, ++p2)
   if(*p1 != *p2) return 0;
return *p1 == *p2;
}
```

Toodud näites juhitakse tsüklit kahe viidamuutuja abil, mis muutuvad üle võrreldavate sõnede. Avaldis ++p1,++p2 tõstab mõlemat viita ühe sümboli võrra edasi. Kui mingil kohal leitakse erinevad sümbolid, siis naasmislause return katkestab tsükli ja väljastab väärtuse 0. Kui leitakse ükskõik kumma sõne lõputunnus 0 (seda kontrollib avaldis *p1 && *p2), siis tsükkel lõpeb ja väljastatav avaldise *p1 == *p2 väärtus on 0 või 1 sõltuvalt sellest, kas sõned lõpevad samal kohal.

8.7. Lüliti ja märgendid case ning default

Lause switch ehk lüliti on konstruktsioon, mis realiseerib ühe suunamise teatavast võimalike suunamiste hulgast vastavalt juhtavaldise ehk selektori väärtusele. Tema kasutamine on lähedane valikulause kasutamisele keeles Pascal või Ada, kuid sisuliselt sarnaneb lüliti keeles C rohkem keele FORTRAN arvutatavale suunamislausele:

lüliti:

switch (avaldis) lause

Lüliti koosneb võtmesõnast switch, millele järgneb avaldis sulgudes ja lause. Viimast nimetatakse ka lüliti sisuks, tavaliselt on ta blokk, kuid süntaktiliselt pole see kohustuslik.

Lüliti sisu iga lause või sisu tervikuna võib olla märgendatud mingi märgendiga case või default (vt. lk. 130). Lausel võib olla ka mitu märgendit case ja lisaks veel märgend default. Märgendid case ja default lüliti sisus peavad rahuldama järgmisi tingimusi.

1) Kõik märgendite case koosseisus olevad konstantavaldised peavad pärast harilikke unaarseid teisendusi olema sama tüüpi kui selektor (avaldis sulgudes).

2) Ühe lüliti koosseisus olevate märgendite case konstantavaldiste väärtused peavad olema kõik erinevad.

3) Ühe lüliti koosseisus võib olla vaid üksainus märgend default.

Märgendid case ja default võivad esineda ainult lülitis.

Selektori lubatav tüüp võib erinevates realisatsioonides erineda. Originaal-C lubab vaid tüüpi int (pärast harilikke unaarseid teisendusi). Kui kompilaatoris on realiseeritud loenditüüp, siis võib selektor olla veel loenditüüpi ja märgendites case võib kasutada loendikonstante. Mõnikord lubatakse kasutada ka tüüpi long. Mitteskalaarsed tüübid, reaalarvutüübid ja viidatüübid selektori tüübina ei ole lubatud.

Lüliti täitmine toimub järgmiselt.

1) Väärtustatakse selektor.

2) Kui selektori väärtus osutub võrdseks mingi märgendi case konstantavaldise väärtusega, siis antakse juhtimine selle märgendiga case märgendatud lausele.

3) Kui selektori väärtus ei ole võrdne ühegagi märgendite case konstantavaldiste väärtustest, kuid leidub märgend default, siis antakse juhtimine lausele märgendiga default.

4) Kui puudub ka märgend default, siis jääb lüliti sisu vahele ja juhtimise saab lülitile vahetult järgnev lause.

Kui juhtimine on antud mingi märgendi case või default järgi, siis seejärel toimub lausete täitmine loomulikus järjekorras kuni lüliti sisu lõpuni (teisi märgendeid case või default ignoreeritakse). Juhtimist saab anda lüliti sisust välja naasmis-, suunamis-, katkestamis- või jätkamislausega.

Tavalise lüliti kasutamisstiili korral on lüliti sisuks blokk, milles leiduvad märgenditega case ja default märgistatud laused ja sisu täitmine katkestatakse katkestamislausega. Vaatleme näitena järgmist lülitit:

```
switch (x) {
    case 1: printf("*");
    case 2: printf("**");
    case 3: printf("***");
    case 4: printf("****");
    }
```

Kui muutuja x väärtus on 2, siis trükitakse üheksa tärni: juhtimise saab lause printf("**"); kuid seejärel täidetakse veel laused printf("***"); ning printf("****"); Kui tahame lõpetada trüki peale üht lauset, siis tuleb juhtimine lülitist välja anda katkestamislausega:

```
switch (x) {
    case 1: printf("*");    break;
    case 2: printf("**");   break;
    case 3: printf("***");  break;
    case 4: printf("****"); break;
    }
```

Kuigi viimane katkestamislause pole loogiliselt vajalik, on ta siiski hea programmeerimisstiili näide - programmi on parem mõista ja vajaduse korral hõlpsam lisada uusi harusid.

Toodud näited demonstreerivad lüliti tüüpilist kasutamist. Keel C ei nõua, et lüliti sisu oleks blokk või et märgendid case ja default asuks ainult selle bloki kõige ülemisel tasemel või et nad oleks mingis kindlas järjekorras ja et nad üldse esineks. Sellest hoolimata võib hea programmeerimisstiili kohaselt soovitada just niisugust kasutamist.

Variante omavahel ühendada ja segada ei ole soovitatav. Ühe enam-vähem mõistliku näitena toome järgmise fragmendi, kus veateadete moodustamisel kasutatakse varianti fatal selleks, et trükkida prefiks veateatele ja väärtustada katkestuslipp, kusjuures veateate trükk toimub variandis error:

```
enum error_type (info, warn, error, fatal } err_flag;
...
/*trükkida veateade ja suurendada vigade loendajaid*/
switch (err_flag)
   {
   case info: printf("Info"); ++info_count; break;
   case warn: printf("Warning"); ++warn_count; break;
   case fatal: printf("Fatal "); exit_flag = 1;
   /* juhtimine läheb siin otse */
   case error: printf("Error"); ++err_count; break;
   }
   ...
```

8.8. Katkestamis- ja jätkamislause

Katkestamis- ja jätkamislause (laused break ja continue) leiavad kasutamist tsüklite ja esimene ka lüliti täitmise juhtimisel. Neid lauseid tuleb eelistada suunamislausele, kuigi viimasega saab organiseerida sarnaseid operatsioone:

katkestuslause:

break ;

jätkamislause:

continue ;

Katkestamislause koosneb võtmesõnast break ning sellele järgnevast semikoolonist. See lause katkestab kõige sisemise (milles ta ise asub) lause while, do, for või switch täitmise ja annab juhtimise katkestatud tsüklile või lülitile vahetult järgnevale lausele. Katkestamislause võib esineda ainult nimetatud lausete koosseisus.

Jätkamislause koosneb võtmesõnast continue ning sellele järgnevast semikoolonist. Lause annab juhtimise tsükli while või do tingimust väärtustavale osale, tsüklis for aga sulgudes olevat kolmandat avaldist väärtustavale osale. Jätkamislauset võib kasutada ainult tsüklites, tal pole tähendust lülitis (võib seal asuda, kui see on tsükli koostisosa).

Selgitame katkestamis- ja jätkamislausete tähendust järgmise skeemi abil. Vaatleme tsüklilauseid ja lülitit:

```
while (avaldis) lause
do lause while (avaldis);
for (avaldis1; avaldis2; avaldis3) lause
switch (avaldis) lause
```

Täiendame neid lauseid, lisades märgendid B ja C, tühilaused ning loogelised sulud:

```
{ while (avaldis) { lause C:;} B:; }
{ do { lause C:; } while (avaldis); B:; }
{ for (avaldis1; avaldis2; avaldis3) { lause C:;} B:;}
{ switch (avaldis) lause B:; }
```

Nüüd on katkestamislause toodud lausetes (tsükli või lüliti sisudes) igaühes samaväärne lausega goto B; ja jätkamislause samaväärne lausega goto C;

Katkestamislauset kasutatakse keeles C tsüklite ja lülitite juhtimiseks väga laialdaselt. Vaatleme selle kohta veel järgmist näidet:

```
/* loeme sisendilt sümboleid, kuni väli saab täis
   või kuni sisendi lõpuni */
static char array[100];
int i, c;
for(i = 0; i < 100; ++i)
   {
   c = getchar();
   if(c == EOF) break; /* sisendi lõpu korral */
   array[i] = c;
   }
```

Märgime, et väljumine katkestamislausega break on tsükli ebanormaalne lõpp (ja see on ka heaks stiiliks), tsükkel ise aga juhib oma normaalset lõppu.

Jätkamislause kasutamine on tunduvalt vähemal määral õigustatud ja sellega ei maksa liialdada. Vaatleme näitena fragmenti, mis töötleb kõik mittetühjad sisendread, kui nad ei alga sümboliga # :

```
extern char buffer[];
...
for(;;)
   {
   gets(buffer); /* loeme rea */
   if(!buffer[0]) continue;
   if(buffer[0] == '#') continue;
   process();
   }
```

Siin on jätkamislausetega liialdatud ja programmilõik oleks selgem järgmisel kujul:

```
extern char buffer[];
...
for(;;) {
   gets(buffer);
   if(buffer[0] && (buffer[0] != '#')) process();
   }
```

8.9. Naasmislause

Naasmislause ehk lause return pöördub tagasi funktsioonist, võimalik et koos funktsiooni väärtuse leidmisega:

naasmislause:

return avaldis ;

Naasmislause koosneb võtmesõnast return, millele järgneb mittekohustuslik avaldis ja (kohustuslik) semikoolon. Selle lausega antakse juhtimine funktsioonist tagasi väljakutsunud funktsiooni, vahetult funktsiooni poole pöördumise järele.

Kui juhtimine funktsioonis jõuab välimise bloki lõppu, siis on see samaväärne avaldiseta naasmislause täitmisega.

Kui naasmislauses pole antud avaldist või kui funktsiooni täitmine lõpeb ilma selle lauseta, siis funktsioonil pole väärtust. Funktsiooni poole pöördumise operatsiooni väärtus on sel juhul defineerimata. Kui naasmislauses leidub avaldis, siis see väärtustatakse ning teisendatakse vastava funktsiooni tüüpi. Saadud väärtus ongi funktsiooni poole pöördumise operatsiooni väärtuseks. Täpsemalt räägime funktsioonide väljakutsumisest ja väärtusest järgmises peatükis.

8.10. Suunamislause

Suunamislause ehk lause goto on määratud juhtimise üleandmiseks funktsiooni suvalisest kohast suvalisele lausele:

suunamislause:

goto märgend-nimi ;

Suunamislause koosneb võtmesõnast goto (kindlasti kirjutatult ühe sõnana), millele järgneb nimi ja semikoolon. Nimi peab olema sama funktsiooni mingi lause märgendiks (märgend-nimeks). Suunamislause täitmisel antakse juhtimine selle nimega märgendatud lausele. Igal lausel võib olla ka mitu märgendit, kuid ühe funktsiooni piirides peavad kõik märgendid olema unikaalsed. Märgend-nimed moodustavad omaette nimeklassi, ega ole konfliktis muude nimedega programmis.

Suunamislausega võib juhtimise anda suvalisele lausele, ka märgendada saab suvalist lauset. Siiski soovitame otseste

suunamiste korral (mis reeglina muudavad programmi halvasti loetavaks) kinni pidada järgmistest reeglitest.

1) Mitte anda suunamist tingimuslikku lausesse väljastpoolt seda tingimuslikku lauset.

2) Mitte suunata tingimusliku lause ühest harust teise.

3) Mitte anda suunamist tsükli või lüliti sisusse väljastpoolt seda tsüklit või lülitit.

4) Mitte anda suunamist blokki väljastpoolt seda blokki.

Heaks programmeerimisstiiliks on katkestamis-, jätkamis- ja naasmislausete, samuti ka tingimuslike lausete, tsüklite ja lülitite eelistamine otsesele suunamisele.

Üheks tõenäoliselt otstarbekaks suunamislause kasutamise kohaks on väljumine korraga mitmest üksteisesse sisestatud tsüklist mingis eriolukorras (nagu näiteks faili lõpp vms.).

8.11. Tühilause

Tühilause koosneb ainult semikoolonist:

```
tühilause:
        ;
```

Tühilause toimel ei tehta midagi. Tal on põhiliselt kaks kasutamiskohta. Esiteks võib tühilausest koosneda tsükli sisu, kui kogu tegevus teostatakse tsükli päises. Vaatleme näitena sõne lõpu leidmist:

```
char *p;
...
while (*p++) ;
```

Tühilause teine mõeldav kasutamiskoht on bloki viimase lausena, võimaldamaks juhtimise andmist bloki lõppu (märgend peab kuuluma lause koosseisu ja blokki lõpetavat sulgu ei saa märgendada):

```
if( e )
  { ...
  goto L;
   ...
  L: ; }
  else ...
```

9. F U N K T S I O O N I D

9.1. Funktsioonide defineerimine

Funktsioon on ainukeseks alamprogrammiliigiks keeles C. Funktsiooni definitsiooniga määratakse programmis uus alamprogramm ja antakse selle kohta järgmine informatsioon:

1) funktsiooni väärtuse tüüp;
2) funktsiooni nimi;
3) formaalsete parameetrite arv ja tüübid;
4) funktsiooni nime nähtavus väljaspool antud faili;
5) funktsiooni sisu, s.t. funktsiooni poole pöördumisel täitmisele võetavad laused.

Funktsiooni definitsiooni ei tohi ära segada funktsiooni deklaratsiooniga, mis ainult teatab kusagil mujal defineeritud funktsiooni nime ja väärtuse tüübi. Märgime, et funktsiooni definitsioonis (vt. lk. 35) võivad puududa nii mäluklassikirjeldaja kui ka tüübikirjeldaja.

Lubatavad mäluklassid funktsiooni definitsioonis on vaid static ja extern. Mäluklassi extern korral (ja vaikimisi) on funktsiooni nimi välisnimi, s.t. saab teatavaks linkerile ja nähtav ka teistes lähtefailides. Mäluklassi static korral ei ole funktsiooni nimi nähtav väljaspool antud lähtefaili.

Funktsiooni mäluklassist ei sõltu tema nime nähtavus antud lähtefailis. Funktsiooni nimi on nähtav tema kirjeldamispunktist kuni faili lõpuni, seejuures ka defineeritavas funktsioonis eneses. Keel C võimaldab igal funktsioonil rekursiivselt pöörduda iseenda poole.

9.2. Funktsioonide tüübid

Funktsiooni definitsioonis (samuti nagu ka deklaratsioonis) määravad tüübikirjeldaja ja deklaraator koos funktsiooni (väärtuse) tüübi. Kui tüübikirjeldaja puudub, siis loetakse funktsiooni tüübiks int.

Funktsiooni definitsioonis peavad tüübikirjeldaja ja deklaraator koos määrama kirjeldatava nime tüübiks funkt-

sioon tüüpi T väärtusega, kus T on suvaline tüüp, peale tüüpide massiiv ja funktsioon (mõnede realisatsioonide korral ka Ühend). Seega funktsiooni väärtuseks ei või olla massiiv ega funktsioon, kuid võib olla viit massiivile või funktsioonile. Näiteks määrab definitsioon

```
int (*g())[]
{ ... }
```

funktsiooni väärtuse tüübiga viit täisarvulisele massiivile.

Funktsiooni deklaraator peab sisaldama konstruktsiooni d(...), kus d on funktsiooni nimi. Kui funktsioonil on formaalsed parameetrid, siis peab sulgudes asuma nende nimede loetelu. Toome veel mõned funktsioonide tüüpide näited:

void f() - f on parameetriteta ja väärtuseta funktsioon;

int g(x,y) - g on funktsioon kahe parameetriga x ja y ning väärtusega tüüpi int;

int (*(*d(w))[])() - d on funktsioon ühe parameetriga w ning väärtusega tüüpi viit massiivile, mille elementideks on viidad täisarvuliste väärtustega funktsioonidele.

9.3. Formaalsete parameetrite definitsioonid

Funktsiooni definitsioonis jaguneb formaalsete parameetrite defineerimine kahte ossa. Parameetrite nimed loetletakse deklaraatori koosseisus, see loetelu määrab ka parameetrite arvu. Parameetrikirjelduste osas kirjeldatakse suvalises järjekorras parameetrite tüübid. Näiteks kui funktsioonil on kolm parameetrit tüüpidega int, double ja viit täisarvule, siis võib funktsiooni definitsioon olla järgmine:

```
void f(x,y,z)
int x, *z;
double y;
{ ... }
```

Funktsiooni formaalse parameetri ainukeseks mäluklassiks võib olla register. See teatab kompilaatorile, et võimaluse korral tuleb vastavad parameetrid kanda arvuti registreisse. Loomulikult sõltub mäluklassi register kasutamise võimalikkus ka parameetri tüübist (vt. lk. 44).

Parameetri tüüp võib olla suvaline, välja arvatud tüübid void ja funktsioon. Kui parameetri tüübiks on massiiv (mõnede realisatsioonide korral ka funktsioon), siis see korrigeeritakse kompilaatori poolt viidaks. Mõned realisatsioonid ei luba funktsiooni formaalse parameetrina ühendit.

Mõnes realisatsioonis võib formaalsete parameetrite kirjeldused jätta andmata tüüpi int parameetrite korral: kui parameetri nimi esineb parameetriloetelus ja vastava parameetri kirjeldus puudub, siis loetakse parameetri tüübiks int. Kuigi selline kirjeldamata jätmine võib olla lubatud, tuleb seda siiski pidada halvaks programmeerimisstiiliks.

Parameetrite kirjelduste osas võib anda ka struktuuride, ühendite ja loendite tüüpide definitsioone, selliste tüübinimede skoobiks on kogu funktsioon. Kuid tüübinimede defineerimise vajadus antud kohal on küsitav ja tõenäoliselt halb programmeerimisstiil. Vaatleme järgmist näidet:

```
int process(r)
struct record { int a; int b; } *r;
{ ... }
```

Struktuuri record kirjeldus on lubatav, kuid nimi record on nähtav vaid funktsioonis. Funktsiooni tegelik parameeter aga peab olema sama tüüpi kui formaalne parameeter (ehkki kompilaator seda ei pruugi kontrollida). Seepärast tuleb sedasama struktuuri kirjeldada veel seal, kus antud funktsiooni poole pöördutakse. Järelikult tuleb sisuliselt üks ja sama mõiste defineerida kahes kohas, mis on alati vigade allikaks siis, kui tekib vajadus programmi modifitseerida. Funktsiooni parameetrite kirjelduses võib kasutada struktuuritüübi viidet.

9.4. Parameetrite tüüpide korrigeerimine

Keel C määrab teatud hulga reegleid, mille põhjal funktsiooni parameetrite tüüpe korrigeeritakse, vähendamaks parameetrite erinevate võimalikkude tüüpide arvu. Korrigeerimine toimub kaheti. Tegelikele parameetritele rakendatakse funktsiooni poole pöördumisel lk. 90 nimetatud teisendusi ja samal viisil korrigeeritakse formaalsete parameetrite tüüpe.

Tüüpi char, short ja float parameetrid korrigeeritakse vastavalt tüüpi int, int ja double. Seega funktsiooni

```
void f(c)
char c;
{
int i;
i = c;
... }
```

käsitletakse täpselt samuti kui funktsiooni

```
void f(c)
int c;
{
int i;
i = c;
... }
```

Kui formaalse parameetri tüüp on massiiv tüüpi T elementidest, siis korrigeeritakse see tüüpi viit tüübile T. Massiivide ja viitade vastavusest sõltuvalt jääb see teisendus programmeerijale märkamatuks. Vaatleme näiteks funktsiooni:

```
int sumarray(a,n)
int a[], n;
{
int sum = 0, i;
for(i = 0; i < n; ++i)
    sum += a[i];
return sum;
}
```

Siin võib ilma funktsiooni sisus midagi muutmata anda formaalsete parameetrite kirjelduse ka kujul int *a, n; Kuigi massiivide nimed üldiselt ei ole l-väärtused, siiski vaadeldakse vastavalt tüübikorrigeerimisele formaalsete parameetritena kirjeldatud massiivide nimesid l-väärtustena.

Tavaliselt ei ole lubatud formaalsed parameetrid tüüpi funktsioon. Siiski aktsepteerivad mõned realisatsioonid selliseid parameetreid (sageli küll koos hoiatava veateatega), korrigeerides vastava parameetri tüübiks viit funktsioonile. Selline korrigeerimine arvestab analoogilist teisendust te-

geliku parameetri jaoks. Igal juhul tuleb eelistada vastavate parameetrite kirjeldamist viitadena funktsioonile:

```
extern int h();
...
f(h);
...
void f(g)
int (*g)();
{
int i;
i = (*g)();
... }
```

9.5. Parameetrite üleandmine

Keeles C toimub parameetrite üleandmine väärtuse järgi. See tähendab, et tegeliku parameetri väärtusest luuakse koopia väljakutsutava funktsiooni lokaalses mälupiirkonnas. Formaalse parameetri nime võib kasutada omistamisoperatsiooni vasakpoolse operandina, kuid muutus kajastub ainult vastava tegeliku parameetri lokaalses koopias.

Kui programmeerija soovib, et väljakutsutav funktsioon muudaks väljakutsuva funktsiooni objekte, tuleb argumentidena anda nende objektide aadressid. Järgmises näites toodud funktsioon swap ei tööta korrektselt, kuna parameetrite üleandmine toimub väärtuse järgi:

```
void swap(x,y)
/* vahetada parameetrite väärtused, ebakorrektne! */
int x, y;
{
int temp;
temp = x; x = y; y = temp;
}
...
swap(a,b); /* a ja b väärtusi ei vahetata */
```

Korrektses versioonis tuleb parameetrid kirjeldada viitadena ja tegelike parameetritena anda aadressid:

```
void swap(x,y)
/* vahetada parameetrite väärtused, korrektne! */
int * x, * y;
{
int temp;
temp = *x; *x = *y; *y = temp;
}
...
swap(&a,&b); /* a ja b väärtused vahetatakse */
```

Tavaliselt realiseerivad kompilaatorid parameetrite üleandmise piirkonna magasinina, kuid see ei ole kohustuslik. Oluline on vaid, et funktsioonid võimaldaksid rekursiooni. Et formaalsele parameetrile saab rakendada aadressi leidmise operatsiooni &, siis peavad parameetrid asuma adresseeritavas mälus. Loomulikult annab see operatsioon viida tegeliku parameetri koopiale, mitte parameetrile enesele.

Keeles C puuduvad vahendid muutuva parameetrite arvuga funktsioonide kirjeldamiseks. Standardteegis olevate funktsioonide abil on see siiski teostatav. Mõningad realisatsioonid (näiteks ANSI C) võimaldavad funktsiooni kirjelduses erikonstruktsiooniga näidata, et parameetrite arv on muutuv.

9.6. Formaalsete ja tegelike parameetrite vastavus

Enamik levinud programmeerimiskeeli (nagu Pascal ja Ada) eeldavad ranget tüübikontrolli funktsiooni formaalsete ning tegelike parameetrite vahel. Sellele vastavalt peavad kokku langema formaalsete ja tegelike parameetrite arv, samuti ka üksikute parameetrite tüübid. Keeles C selline kontroll puudub, mille põhjenduseks võib tuua järgmist.

Süntaks ei võimalda funktsiooni parameetrite tüüpe esitada funktsiooni deklaratsioonis ja seega on range tüübikontroll teises sisendfailis defineeritud funktsiooni kohta võimatu. Tüübikontrolli puudumine annab aga programmeerijale teatava vabaduse keelereeglite teadlikuks rikkumiseks, eriti muutuva parameetrite arvuga funktsioonide realiseerimisel. Näitena tüübikontrolli puudumise ohtudest vaatleme fragmenti

```
double sqrt(x)
double x;
{ ... }
long hypothenuse(x,y)
long x,y;
{
return (sqrt(x * x + y * y));
}
```

Kompileerimisel ei anta siin veateadet, kuigi funktsiooni sqrt parameeter on kirjeldatud tüüpi double, pöördumisel on aga tüüp long. Parameetrit ei teisendata ka õigesse tüüpi - funktsioon sqrt annab täitmise ajal lihtsalt vale tulemuse.

Nagu juba märgitud, puudub keeles C vahend muutuva parameetrite arvuga funktsioonide masinsõltumatul viisil kirjeldamiseks. Mõned sellised funktsioonid (näiteks printf) kuuluvad keele C standardteeki, kuid nad sõltuvad konkreetsest realisatsioonist. Sellised funktsioonid kasutavad väga spetsiifilisi parameetrite üleandmise tingimusi ja nõuavad mingit täiendavat mehhanismi parameetrite arvu määramiseks igal konkreetsel pöördumisel (näiteks funktsiooni printf korral määratakse parameetrite arv formaadisõnega).

ANSI C kasutab tugevamat kontrolli funktsiooni parameetrite üle nn. prototüübikirjelduste mehhanismi abil, mis võimaldab ka deklaratsioonides näidata parameetrite tüübid.

9.7. Funktsiooni väärtused

Funktsiooni väärtus võib olla suvalist tüüpi peale tüüpide funktsioon ja massiiv (mõned realisatsioonid ei luba ka ühendeid). Funktsiooni väärtuse määrab naasmislauses esinev avaldis. Kui seal avaldis puudub või funktsioonist väljutakse bloki lõpu kaudu, jääb funktsiooni väärtus määramata.

Funktsiooni väärtus pole l-väärtus, ta antakse väljakutsuvasse funktsiooni tagasi "väärtuse järgi" ja seega ei saa funktsiooni poole pöördumine olla omistamise vasakpoolseks operandiks. Keel ei määra, kuidas nimelt toimub funktsiooni väärtuse ülekandmine väljakutsuva funktsiooni mälupiirkonda.

9.8. Tegeliku ja kirjeldatud väärtuse vastavus

Naasmislause ilma avaldiseta kujul return; on keeles lubatud sõltumata sellest, kas funktsiooni tüübiks on void või mitte. Nii saavutatakse sõltumatus ka selliste kompilaatorite suhtes, mis ei aktsepteeri andmetüüpi void. Kui funktsiooni tüüp ei ole void, kuid funktsioon ei väljasta väärtust, siis on tema väärtused juhuslikud ja järelikult ei saa sellise funktsiooni poole pöörduda väärtust eeldavas kontekstis. Ilma väärtuseta naasmist funktsioonist tuleb soovitada ainult siis, kui funktsiooni tüübiks on void.

Kui funktsiooni tüüp on void, siis naasmislause esinemine koos avaldisega on viga. Juhul kui selle avaldise tüüp on omakorda void, nagu järgmises kurioosses näites:

```
void f()
{
extern void g();
...
return g();
}
```

ei pruugi kompilaator viga avastada. Vigane on ka void tüüpi funktsiooni poole pöördumine väärtust eeldavas kontekstis.

Kui funktsiooni kirjeldatud väärtus on tüüpi T, mis pole void, siis naasmislauses esitatud avaldise väärtus peab olema teisendatav tüüpi T omistamisel kasutatavate teisenduste abil. Näiteks tüüpi int funktsiooni lõpetavaks lauseks võib samaväärselt olla ükskõik milline järgnevatest lausetest:

```
return 32.5;
return (int) 32.5;
return 32;
```

Varasemates realisatsioonides, mis ei toeta andmetüüpi void, on üldlevinud stiiliks jätta ilma väärtuseta funktsioonide definitsioonides tüübikirjeldaja ära (seega omistatakse neile funktsioonidele vaikimisi tüüp int):

```
main()
{ ... }
```

9.9. Funktsioon main

Iga keeles C kirjutatud programm peab sisaldama parajasti ühe funktsiooni nimega main, mille täitmisest algab programmi täitmine. Enamik realisatsioone võimaldab funktsiooni main esitada kas ilma parameetriteta või kahe parameetriga, osa realisatsioone aga ka kolme parameetriga. Vastavalt vajadusele võib funktsiooni main defineerida sobival kujul:

```
void main()   void main(argc,argv)   void main(argc,argv,env)
              int argc;               int argc;
              char *argv[];           char *argv[], *env[];
```

Parameetrite nimed ei ole reserveeritud, kuid üldkasutatavad. Parameetrid seab täitmise ajal programmeerimissüsteem ja nad kajastavad programmile antavat globaalset infot. Parameetrite esitamise konkreetne viis sõltub nii operatsiooni- kui ka programmeerimissüsteemist. Parameetritel on C-programmis järgmine tähendus.

Parameeter argc on tüüpi int ja annab programmi argumentide arvu. Et tavaliselt on esimeseks argumendiks täidetava programmi nimi, siis on argc väärtus alati positiivne.

Parameeter argv on massiiv viitadest argumendisõnedele. Tavaliselt viitab argv[0] programmi nimele ning argv[1], ..., argv[argc-1] ülejäänud argumentidele. Tavaliselt (kuid mitte alati) on argv[argc] nullviit.

Kui funktsioonil main on lubatud ka kolmas parameeter, siis see on viidamassiiv nn. keskkonnamuutujatele. Need on süsteemi globaalsed muutujad, mis esitatakse sõnedena kujul "nimi=väärtus". Programm võib kontrollida nimede väärtusi ja valida vastavalt oma tegevusviisi. Keskkonnamuutujate otsene muutmine, nagu funktsiooni parameetrite korral tavaliselt, ei anna tulemust. Vahendid keskkonna muutmiseks programmi poolt antakse standardteegi koosseisus.

Funktsiooni main tüübiks määratakse tavaliselt kas void või int. Kas funktsiooni main väärtusel on mõtet, see sõltub konkreetsest programmeerimissüsteemist. Tavaliselt on funktsiooni main poolt väljastatav väärtus 0 programmi normaallõpu, nullist erinev väärtus aga ebanormaalse lõpu tunnuseks.

10. A N S I C

Ameerika Rahvuslik Standardite Instituut (ANSI) formeeris 1982. aastal alamkomitee keele C standardiseerimiseks. Standardiseerimisele kuulusid keel C ise, tema funktsioonide teek, kompileerimisaja ja täitmisaja keskkonnad. See komitee esitas 1986. aastal keele C standardi projekti (Draft proposed ANSI C), millele vastavat keelt oleme selles ülevaates nimetanud ANSI C. Kuigi see standard eksisteerib tänase päevani vaid projekti tasemel, on ta avaldanud suurt mõju keele C viimastele realisatsioonidele, mis üha rohkem selle standardiga arvestavad.

Suures osas aktsepteerib ANSI standardi projekt keeles C programmeerimisel välja kujunenud praktikat. Sisse on toodud keele mõningaid laiendusi (võeti näiteks üle mõningad konstruktsioonid keelest C++). Täienduste ideeks on programmeerijale masinsõltumatute programmide kirjutamise võimaldamine, kuid mitte selleks kohustamine. Standardi projektis ilmneb tendents keele C arenemisele tugeva tüübikontrolliga (strong typing) keelte suunas.

Kogu senise ülevaate jooksul oleme võrdlevalt viidanud ANSI C iseärasustele. Selles peatükis võtame kokku põhilised erinevused originaal-C ja teiste enam levinud realisatsioonide suhtes. Selle peatüki alapunktid järgivad eespool kasutatud jaotust peatükkideks.

10.1. Leksikaelemendid

Põhilised leksikaalsed erinevused keeles ANSI C on järgmised: tuuakse sisse teatavad sümbolikolmikud (ehk trigraafid) mõningate sümbolite tähistamiseks; võetakse kasutusele kolm uut võtmesõna volatile, const ja signed; laiendatakse konstantide süntaksit; liitomistamisoperatsioon loetakse kindlasti üheks lekseemiks.

Tähestiku osas määratakse eriliste sümbolikolmikute (trigraafide) hulk, et teksti saaks sisestada ka väiksema sümbolihulgaga seadmelt. Trigraaf algab kahe küsimärgiga:

Trigraaf	tähendus
??(	[
??<	{
??/	\
??'	^
??=	#
??)	]
??>	}
??!	|
??-	~

Trigraafide asendus lähtetekstis toimub enne leksikaalset analüüsi ja isegi enne võtmesümbolite (\x) töötlemist. Aktsepteeritakse ainult ülalloetletud trigraafe. Uus võtmesümbol \? on kasutatav selleks, et esitada programmis teksti, mis langeb kokku mõne trigraafiga.

Nimede pikkust ANSI C ei piira. Nimesid eristatakse esimese 31 sümboli põhjal. Suur- ja väiketähed loetakse erinevaiks. Realisatsioonid võivad välisnimede esitamist piirata kuni 6 arvestatava sümbolini ning eeldada, et välisnimedes ei eristata suur- ja väiketähti. See sõltub vastava programmeerimissüsteemi (linkeri) jaoks kehtivatest kitsendustest.

Võtmesõnadele on lisatud const, volatile ja signed ning välja jäetud entry, fortran ja asm. ANSI C nagu enamik viimaseid realisatsioone toetab võtmesõnu enum ja void.

Täisarvkonstandi süntaksit on laiendatud. Lisaks sufiksile L (long) võib kasutada sufiksit U (unsigned). Need sufiksid on kasutatavad ka koos ja suvalises järjekorras:

```
tüübimarker:
        long-marker unsigned-marker
        unsigned-marker long-marker
long-marker: variandid
        l L
unsigned-marker: variandid
        u U
```

Toome mõned näited:

```
  100u      34LU     32767ul
```

Täisarvkonstandi tüüp määratakse järgmiste reeglitega:

1) kui on antud mõlemad tüübimarkerid, siis konstandi tüüp on unsigned long int;

2) kui on antud ainult long-marker, siis konstandi tüüp on long int juhul, kui tema väärtus kuulub vastava tüübi väärtuste hulka ja unsigned long int vastasel korral;

3) kui on antud ainult unsigned-marker, siis konstandi tüüp on unsigned int juhul, kui tema väärtus kuulub vastava tüübi väärtuste hulka ja unsigned long int vastasel korral;

4) kaheksand- või kuueteistkümnendesituses täisarvkonstandil on esimene tüüp loetelust int, unsigned int ja unsigned long int, mille väärtuste hulka ta väärtus kuulub;

5) kümnendesituses täisarvkonstandil on esimene tüüp loetelust int, long int ja unsigned long int, mille väärtuste hulka tema väärtus kuulub.

Põhiliseks erinevuseks varasematest realisatsioonidest on siin see, et konstandid võivad olla ka märgita tüüpi (isegi ilma sufiksiteta) ja seepärast võivad nad vastavalt harilikele binaarsetele teisendustele muuta avaldise väärtuse märgita väärtuseks. Oletame et long int esitatakse 32-bitilises täiendkoodis, siis järgmine programmifragment võib sellise erinevuse avastada:

```
#define K 0xFFFFFFFF  /* -1 32-bitises täiendkoodis */
void main()
{
if(0 < K) printf("K on märgita - ANSI C\n");
else printf("K on märgiga - traditsiooniline\n");
}
```

Reaalarvkonstant võib peale tüüpi double olla veel tüüpi long double ja float. Seda tähistavad uued sufiksid F või f (tüüp float) ja L või l (tüüp long double). Kirjeldaja long float ANSI C puhul puudub:

```
reaalarvkonstant:
     numbrid eksponent float-marker
     punktnumbrid eksponent float-marker
float-marker: variandid
     f F l L
```

Sõned, mille vahel leiduvad ainult tühisümbolid ühendatakse ANSI C korral üheks sõneks vaid ühe lõpunulliga. Selline ühendamine muudab pikkade sõnede kirjutamisel tarbetuks rea jätkamiskokkuleppe langkriipsu abil. Näiteks:

```
static char helptext[] = "Type:\n"
                         " h for help\n"
                         " q for quit\n";
```

Järgmised kaks muutust võivad kajastuda juba olemasolevais programmides. Esiteks võivad kaks identselt kirjutatud sõnet asuda mälus ühel ja samal kohal. Teiseks võivad sõned olla paigutatud mällu, kuhu ei ole lubatud kirjutamine. Süsteemi käitumine juhul, kui programm üritab sõnet modifitseerida ei ole defineeritud. Toome väikese programmilõigu sõnede olemuse kindlakstegemiseks:

```
char *string1, *string2;
void main()
{
string1 = "abcd";
string2 = "abcd";
if(string1   ==   string2)
   printf("Sõned jagavad mälu\n");
else printf("Sõned on erinevas mälus\n");
string1[0] = '1'; /* võib kaasa tuua avariilõpu */
if(string1[0] == '1')
   printf("Sõnet saab modifitseerida\n");
else printf("Sõnet ei saa modifitseerida\n");
}
```

Praktikas esinevad need kaks omadust sageli korraga: kui sõned jagavad mälu, siis pole nad modifitseeritavad. Modifitseerimiskeeldu saab garanteerida algväärtustades sõnega massiivi, mis on määratud tüüpi const:

```
const char message[] = "Some errors\n";
```

Võtmesümbolite hulka on lisatud \a (alert) ja \? (küsimärk). Võtmesümbolit \a tõlgendatakse kui kuuldavat signaali välisseadmel (ASCII "bell" 07). Küsimärki on vaja võtmesümbolina esitada seal, kus tuleb maha suruda küsimärkide tõlgendmine trigraafi koosseisu.

Numbrilised võtmesümbolid võib peale kaheksandkuju esitada ka kuueteistkümnendkujul. Langkriipsule järgneb siis sümbol x (väiketäht) ja kuni 3 kuueteistkümnendnumbrit:

koodi-tähis:

8-number
8-number 8-number
8-number 8-number 8-number
x 16-number
x 16-number 16-number
x 16-number 16-number 16-number

10.2. Preprotsessor

Preprotsessoridirektiivi tunnusele # võivad reas eelneda ja järgneda tühisümbolid. Kui rida sisaldab peale tühisümbolite ainult sümbolit #, siis sellist rida ignoreeritakse.

ANSI C poolt on sisse toodud kaks uut konstruktsiooni, mis võimaldavad juhtida makroasendusi kahest aspektist: makroparameetrite asendusi sõnedes ja kahe lekseemi ühendamist.

Makrolaiendis tõlgendatakse makroparameetri nimele vahetult eelnevat üksikut sümbolit # kui unaarset operatsiooni järgmiselt: parameeter asendatakse jutumärkidesse asetatud tegeliku parameetriga. Arvestades sõnede ühendamist saab selle operatsiooniga määrata sõne koosseisus asendatavad parameetrid. Näiteks kui on antud järgmine makromäärang:

```
#define TEST(a,b) printf(#a "<" #b "=%d\n", (a) < (b))
```

siis asendatakse makroväljakutse

```
TEST(0,0xFFFFFFFF);
```

sõnede ühendamist arvestades järgmise konstruktsiooniga:

```
printf("0<0xFFFFFFFF=%d\n", (0) < (0xFFFFFFFF));
```

Kahe lekseemi ühendamist üheks saab makrolaiendis juhtida binaarse operatsiooniga ## . Pärast makroasendust ühendatakse operatsiooni ## operandid üheks lekseemiks, näiteks:

```
#define TEMP(i) temp ## i
TEMP(1) = TEMP(2);
```

asendamisel saadakse

```
temp1 = temp2;
```

Standardmakrodest tunneb ANSI C preprotsessor järgmisi (parameetriteta, nimi algab ja lõpeb kahe allkriipsuga):

__LINE__ - Makro väärtuseks on jooksva sisendfaili jooksva rea number kümnendkonstandina.

__FILE__ - Makro väärtuseks on jooksva sisendfaili nimi sõnena.

__DATE__ - Makro väärtuseks on kompileerimiskuupäev sõnena kujul "Mmm dd yyyy".

__TIME__ - Makro väärtuseks on kompileerimise kellaaeg sõnena kujul "hh:mm:ss".

__STDC__ - See makro on ANSI C korral defineeritud ja tal on nullist erinev väärtus.

Standardmakrosid ei saa tühistada ega ümber defineerida.

Failide lisamise direktiivi #include süntaks on nüüd:

direktiiv-include:

 # include < sümbolid >

 # include " sümbolid "

 # include nimi

Sümbolite hulka võivad kuuluda suvalised keele C tähestiku sümbolid peale realükke. Mitmesuse välistamiseks ei või sümbolile < järgnevate sümbolite hulgas leiduda sümbolit > ja sümbolile " järgnevate hulgas sümbolit ".

Oluline on siin, et faili nime määravad sümbolid ei pea olema seotud keele C süntaksiga. Isegi loetletutest teisel juhul, mis sarnaneb sõnele keeles C, ei arvestata võimalikke võtmesümboleid (küll aga arvestatakse trigraafe).

Direktiivi #include parameetriks võib olla nimi vaid siis, kui see on makronimi, mille asendamine annab ühe esimesest kahest vormist. Et makrolaiend koosneb lekseemidest, siis kitsendab see veidi nii määratud faili nimesid.

Makroasenduses ei laiendata makrolaiendit sellesama makroga (s.t. puudub vahetu rekursioon). See võimaldab makrolaiendis kasutada makronime ennast. Vaatleme näitena funktsiooni sqrt võimalikku ümberdefineerimist makrona:

```
#define sqrt(x)  ((x) < 0 ? sqrt(-x): sqrt(x))
```

Makrot võib ümber defineerida vahepealse defineeringu tühistamiseta, kui määrang täht-tähelt ühtib seni kehtivaga.

Uutest preprotsessoridirektiividest toob ANSI C sisse #elif, #error ja #pragma ning operatsioonidest defined (direktiivi #elif ja operatsiooni defined vaatlesime lk. 31).

Direktiiv #error produtseerib kompileerimisaja veateatena oma parameetrina antud sõne. Seda direktiivi kasutatakse peamiselt preprotsessis avastatud vastuoludest teatamiseks:

```
#if SIZE % 256 != 0
#error "SIZE must be a multiple of 256!"
#endif
```

Direktiivi #pragma kasutatakse kompilaatorile realisatsioonist sõltuva info edastamiseks. Näiteks võib määrata edasiseks tööks mingi parameetri. Direktiivis antud infole ei seata kitsendusi. Kui info ei vasta realisatsioonile, siis kompilaator ignoreerib teda. Samuti võivad erinevad realisatsioonid tõlgendada sama informatsiooni erinevalt, mistõttu direktiivi #pragma tuleks kasutada tingimuslikult:

```
#if defined(TCC) && defined(__STDC__)
#pragma inline(myfunc)
#endif
```

10.3. Kirjeldused

Skoobid ja nimeklassid. Märgend-nimed moodustavad omaette nimeklassi. Struktuuride, ühendite ja loendite tüübinimed kuuluvad ühte nimeklassi. Iga struktuuri või ühendi komponentide nimed moodustavad omaette nimeklassi.

Välisnimed alluvad tavalistele skoobireeglitele. Järgmised kirjeldused näiteks ei tekita viga kompileerimisel, ehkki välisnimi X on kirjeldatud kahte tüüpi. Siin võib tekkida küll viga täitmise ajal (seepärast annab hea kompilaator siin hoiatava veateate):

```
if(test) {
  extern int X;
  return X; }
else {
  extern double X;
  return X; }
```

Formaalsete parameetrite nimed loetakse kirjeldatuks funktsiooni sisu moodustavas kõige välimises blokis, s.t. sama nime kirjeldamine selles blokis on viga. See välistab parameetrinimede juhusliku varjamise, mis võib ette tulla eriti juhul, kui parameetrid jäävad ilmutatult kirjeldamata:

```
int f(x,y)
{
int x,y;    /* siin annab ANSI C vea */
... }
```

Tüübinime kasutamine struktuuritüübi ja ühenditüübi definitsioonis annab vastavale struktuuri- või ühenditüübile nime. Selle nime skoop ulatub kirjeldamispunktist kuni kas bloki lõpuni, või siis väliskirjelduse korral faili lõpuni. Uus definitsioon sisemises blokis varjab antud nime.

Struktuuri- ja ühenditüübi osalist defineerimist, s.t. veel defineerimata struktuuri- või ühenditüübi viite andmist on lubatud kasutada sellises kontekstis, kus ei ole oluline vastava objekti pikkus. Seda saab kasutada viitade defineerimisel antud tüübile või siis sünonüümi andmisel antud tüübile konstruktsiooni typedef abil. Vastava tüübi täielik definitsioon peab järgnema hiljem sama skoobi piirides. Nii saab kasutada üksteisele viitavaid struktuure, näiteks:

```
struct cell;
struct header { struct cell *first; ... };
struct cell { struct header *head; ... };
```

Mittetäielik definitsioon struct cell; varjab iga teise nime cell kirjelduse samas nimeklassis. Järgnev struktuuri header määrav kirjeldus kasutab struktuuri cell osalist määrangut. Struktuuri cell täielik määrang järgneb sama skoobi piires.

Deklaraatorite kirjelduses on tehtud mõned muutused.

1) Viidadeklaraator võib sisaldada tüübikirjeldajaid (tüübimodifikaatoreid):

```
viida-deklaraator:
        * tüübikirjeldajad deklaraator
tüübikirjeldajad:
        tüübikirjeldaja
        tüübikirjeldajad tüübikirjeldaja
```

Viidadeklaraatoris võivad tüübikirjeldajateks olla uued kirjeldajad const ja volatile. See võimaldab kirjeldada "konstantseid viitu" ja "viitu konstantsetele objektidele".

2) Funktsioonideklaraator võib sisaldada parameetrikirjeldusi, ka muutuva parameetrite arvu spetsifikatsiooni:

funktsiooni-deklaraator:
    deklaraator (parameetrid)
    deklaraator (parameetritüüpid)
parameetritüübid:
    parameetri-kirjeldus
    parameetritüübid , parameetri-kirjeldus
parameetri-kirjeldus:
    kirjeldajad deklaraator
    kirjeldajad abstraktne-deklaraator

Vältimaks mitmesust parameetrite ja parameetritüüpide vahel ei tohi parameetri nimi ühtida mingi kehtiva tüübinimega.

Ainukeseks lubatavaks mäluklassiks parameetrikirjelduses on register, mida aga funktsiooni definitsioonist erinevas kontekstis ignoreeritakse. Seega prototüübis kasutatav mäluklass register parameetrite üleandmise viisi ei määra ja tal on mõtet ainult funktsiooni definitsioonis.

Parameetrite tüüpide loetelu sisaldavat funktsioonideklaraatorit nimetame funktsiooni prototüübiks. See määrab funktsiooni parameetrite arvu ja tüübid. Kui funktsiooni kirjelduses pole antud parameetrite tüüpe, siis selle funktsiooni poole võib pöörduda suvalise arvu ja suvalist tüüpi argumentidega (nagu keeles C tavaliselt). Prototüüpe võib kasutada nii funktsioonide definitsioonides (siis jäävad ära parameetrite tüübikirjeldused), funktsioonide deklaratsioonides kui ka liitdeklaraatori koosseisus. Mõned näited:

```
extern double sqrt(double x);
int abs(int i)
{
return i < 0 ? -i: i;
}
void (*func_array[3])(int order, double epsilon) =
                { f1, f2, f3 };
```

Funktsioonide prototüübid on ANSI C üks olulisemaid keele C laiendusi. Selline laiendus annab keelele võimaluse funktsiooni argumentide tüüpide ja arvu kontrolliks. Prototüüp suurendab programmi loetavust ja vähendab vigade võimalust. Ilma prototüübita vorm on jäänud keelde vaid selleks, et tagada juba varem loodud programmide korrektsus.

Kui prototüüpe ei kasutata, siis:

- funktsioone saab kirjeldada kontekstuaalselt, s.o. nende poole pöördumisega;
- funktsiooni parameetritele rakendatakse pöördumisel tavalisi teisendusi;
- ei kontrollita parameetrite arvu ega tüüpe; funktsioonil võib olla suvaline arv suvalist tüüpi parameetreid.

Vastupidiselt sellele, kui kasutatakse prototüüpe, siis:

- funktsioonid peavad olema enne nende poole pöördumist ilmutatult kirjeldatud;
- parameetrid teisendatakse (nagu omistamisel) prototüübi poolt kirjeldatud tüüpidesse;
- parameetrite arv ja tüübid peavad kokku langema kirjeldatud tüüpidega (või olema nendeks teisendatavad), vastupidine juhtum annab vea; muutuva parameetrite arvu näitamiseks on olemas erivahendid.

Prototüübi kuju sõltub sellest, kas funktsioonil pole parameetreid, on fikseeritud arv parameetreid või on muutuv arv parameetreid.

1) Kui funktsioonil parameetrid puuduvad, siis tema prototüübi parameetritüüpide loetelu sisaldab vaid ühe elemendi - tüübikirjeldaja void:

```
extern int random(void);
static void print_header(void);
```

2) Kui funktsioonil on fikseeritud arv parameetreid, siis kirjeldatakse nad prototüübis. Kui tegu on funktsiooni deklaratsiooniga, siis võivad parameetrite nimed esineda või ka mitte esineda (esinemisel neid ignoreeritakse):

```
extern double atan2(double, double);
extern char *strncpy(char *destin,
                     char *source, int count);
```

3) Kui funktsioonil on muutuv arv parameetreid, siis osa esimesi neist võib kirjeldada, järgnev koma ja kolm punkti tähistavad suvalist arvu järgneda võivaid parameetreid. Prototüübis peab esinema vähemalt üks kirjeldatud parameeter:

```
extern int fprinf( FILE file, char *format,... );
```

Prototüüpe võib kasutada ka liitdeklaraatoris, nagu näiteks ANSI C teeki kuuluva funktsiooni signal kirjelduses:

```
void(*signal(int sig, void (*func)(int sig))(int sig);
```

Siin kirjeldatakse nimi signal kui funktsioon, millel on kaks parameetrit: täisarvuline parameeter sig ja parameeter func - viit tüüpi void funktsioonile, millel on üks täisarvuline parameeter sig. Funktsiooni signal väärtuse tüüp ühtib tema teise parameetri tüübiga: on viit tüüpi void funktsioonile ühe täisarvulise parameetriga sig. Parema arusaadavuse huvides võib funktsiooni signal kirjeldada järgnevalt:

```
typedef void (*sig_handler) (int sig);
sig_handler signal(int sig, sig_handler func);
```

On lubatud ka ühes funktsiooni kirjelduses prototüübi esinemine, teises aga puudumine. Näiteks deklaratsioonis

```
typedef (*sig_handler2)();
sig_handler2 signal2(int sig, sig_handler2 func);
```

on funktsioonil signal2 prototüüp, aga funktsioonil, millele viitab tema teine parameeter prototüüpi ei ole.

Funktsiooni definitsioonis võib funktsiooni deklaraator olla nii tavalises kui ka prototüübivormis. Kaks järgmist funktsiooni definitsiooni näiteks on ligikaudu samaväärsed:

```
int f(int i, int j) { ... } /* prototüübiga vorm */
int f(i,j) int i,j; { ... } /* tavaline vorm */
```

Me ütleme, et need definitsioonid on ligikaudu samaväärsed, kuna prototüübi olemasolust tulenevad väikesed erinevused.

1) Kui funktsioon on defineeritud prototüübiga, siis see prototüüp nõuab, et kõik ülejäänud sellesama funktsiooni deklaratsioonid peavad olema antud samasuguse prototüübiga.

2) Kui funktsioon on defineeritud ilma prototüübita, kuid prototüüp on määratud mingi selle funktsiooni varasema deklaratsiooniga, siis parameetrite kirjeldused funktsiooni definitsioonis peavad vastama sellele prototüübile.

Teine reegel võimaldab olemasolevaid programme täiendada prototüüpidega. Selleks lisatakse prototüübiga funktsiooni-kirjeldus, funktsiooni definitsiooni pole aga vaja muuta.

Kui leidub ühe funktsiooni mitu prototüüpi, siis peavad nad kokku langema, seejuures peavad omavahel võrduma ka massiivide rajad (peale esimese) ning samuti peavad funktsiooni väärtused olema sama tüüpi (koos vastava prototüübiinfoga, kui see esineb). Parameetrite nimed, kui nad esinevad, peavad samuti kokku langema. Näiteks järgmistes paarides antud parameetrite tüübid on ülalloetletud mõttes kõik erinevad:

```
int                 short
int *               short *
int ()              int (double x)
int (*) [20][20]    int (*) [10][40]
```

Kui toimub pöördumine prototüübiga esitatud funktsiooni poole, siis tegelike parameetrite arv peab kokku langema prototüübis antud parameetrite arvuga ja nende tüübid peavad võimaldama omistamisele vastavaid teisendusi prototüübiga antud tüüpidesse. Enne pöördumist teostatakse vastavad teisendused. Kui parameeter kuulub prototüübi muutuva osa parameetrite hulka, siis selle parameetri jaoks teostatakse funktsiooni poole pöördumisel tavaline teisendus.

ANSI C määrab (realisatsioonidele) täpse vahekorra prototüübiga funktsiooni poole pöördumise ja ilma prototüübita funktsiooni poole pöördumise vahel. Funktsiooni poole pöördumine väljaspool mingit selle funktsiooni prototüübi skoopi peab olema täpselt samasugune, nagu pöördumine fikseeritud parameetrite arvuga prototüübiga, kus parameetrite tüübid vastavad täpselt tegelike parameetrite tüüpidele pärast neile rakendatavaid tüübiteisendusi. Kui näiteks on antud pöördumine kirjeldamata funktsiooni poole:

```
process(a, b, c, d);
```

kus tegelike parameetrite tüübid on

```
short a; struct {int a,b;} b; float *c; float d;
```

siis peab pöördumine olema täpselt samasugune, nagu oleks antud prototüüp

```
int process(int, struct {int a,b;}, float *, double);
```

Tuleb märkida, et see kokkulepe ei määra prototüüpi mõne järgneva pöördumise jaoks. Näiteks võib edaspidi leiduda pöördumine kujul:

```
process (x, y, z);
```

kus muutujad x, y, ja z on kõik tüüpi float. Seda pöördumist tõlgendatakse nii, nagu oleks olnud tegemist prototüübiga

```
int process(double, double, double);
```

sõltumata eelnevast pöördumisest. Muidugi võib see tuua kaasa vea täitmisajal, kuid ANSI C ei loe prototüübi mittekasutamisel muutuvat parameetrite arvu ja erinevaid tüüpe veaks.

Algväärtustajad. ANSI C kohaselt võib ühendeid algväärtustada. Algväärtuseks peab olema avaldis, mis on lubatav ühendi esimese komponendi algväärtustamiseks, näiteks:

```
union U { double d; long q; } x = 0.0;
union V { struct { int a; union U *b;} s;
          struct { union U *b; int a;} t; } y = {0, &x};
```

Automaatseid andmeagregaate võib algväärtustada, kuid algväärtustavad avaldised peavad rahuldama samu tingimusi, mis staatiliste agregaatide korral (lubatud on ainult konstantavaldised). Selline (üsna suvaline) kokkulepe lihtsustab mõneti realisatsiooni ja välistab "koledad" algväärtused.

Agregaatide algväärtustamisel antav algväärtuste loetelu peab olema täielikult loogelistes sulgudes vastavalt väärtustatavate objektide tasemele, või peavad olema ära jäetud kõik sulud peale kõige välimiste. See kokkulepe kõrvaldab mitmesuse algväärtuste mittetäieliku loetelu tõlgendamisel.

Kui algväärtuste loetelu on liiga lühike liitobjekti algväärtustamiseks, siis algväärtustatakse ülejäänud komponendid vastavat tüüpi nullidega. Kui staatilistel objektidel pole antud algväärtusi, siis algväärtustatakse nad vastavat tüüpi nullidega. Järgmised algväärtustajad on samaväärsed:

```
struct S { int a; float b; char *c;};
struct S x = {0};
struct S x = { 0, 0.0F, (char *) 0 };
```

Välisnimed. Mäluklassiga extern kirjeldatud nimel on samasugune skoop nagu kõigil teistel nimedel. ANSI C kasutab välisnime korral puuduva mäluklassi meetodit (vt. lk. 57).

10.4. Tüübid

ANSI C laiendab tüüpide koosseisu järgmiselt:

1) märgita täisarvutüübil võib olla igasugune pikkus;

2) märgiga täisarvutüüpe võib anda ilmutatult võtmesõnaga signed (kasutatav ka sümbolitüübi ja bitiväljade korral);

3) lisatud on uus reaalarvutüüp long double; ära on jäetud long float kui double sünonüüm;

4) lisatud on kaks uut tüübitäiendit const ja volatile;

5) lisatud on (paljude realisatsioonide eeskujul) tüübid enum ja void.

Täisarvutüüpides lisandus märgiga tüübi kirjeldus, märgiga täisarvutüübi kirjeldaja uus süntaks on järgmine:

märgiga-tüübi-kirjeldaja:

signed
signed int
signed short int
signed long int

sümbolitüübi-kirjeldaja:

char
unsigned char
signed char

Lisaks keelele teatava sümmeetria andmisele on märgiga tüüpidel eriline tähtsus sümbolite ja bitiväljade korral. Kui char on realiseeritud märgita tüübina, siis signed char toob sisse märgiga sümbolid. Kui char on märgiga tüüp, siis unsigned char võimaldab kasutada märgita sümboleid. Bitiväljade puhul (kus märgi olemasolu langeb tavaliselt kokku sama omadusega tüübi char korral) saab opereerida samamoodi. Tüüp signed on sünonüüm tüübile signed int ehk int. Näiteks kui tüüp char kasutab 8 bitti, siis pärast programmifragmenti

```
unsigned char uc = -1;
signed char sc = -1;
char c = -1;
int i = uc, j = sc, k = c;
```

on muutuja i väärtus alati 255 ja j väärtus -1. Muutuja k väärtus võib sõltuvalt realisatsioonist olla 255 või ka -1.

Reaalarvutüüpidele on lisatud long double ja ära on jäetud tüüp long float kui double sünonüüm:

reaalarvutüübi-kirjeldaja:

float

double

long double

Uuemad arvutid teostavad tehteid tavaliste, topelttäpsusega ja laiendatud reaalarvudega (enamasti vastavalt 32, 64 ja 128 bitti). Seetõttu on keelde lisatud laiendatud reaalarvu tüüp. Mõnes realisatsioonis võivad küll tüübid double ja long double, või siis float ja double kokku langeda.

Võtmesõna const on uus tüübikirjeldaja, mida kasutatakse koos teiste tüüpidega, sealhulgas struktuuride, ühendite ja loenditega, või üksikult (siis const on const int sünonüüm):

konstantse-tüübi-kirjeldaja:

const

Kui muutuja on võtmesõnaga const konstantseks kuulutatud, siis tema väärtust ei saa muuta: talle ei saa omistada ega kasutada teda suurendamis- ja vähendamisoperatsioonides. Tüübikirjeldaja const abil saab viida puhul defineerida nii konstantset viita kui ka viita konstantsele objektile:

```
int * const const_pointer; /* konstantne viit */
const int *pointer_to_const; /* viit konstandile */
```

Ilmekamaks muudab selle näite kirjelduse typedef kasutamine:

```
typedef int * intptr;
const intptr pointer_to_const;
intptr const const_pointer;
```

Viita konstandile saab muuta, sellele viidale vastavat sisu aga mitte. Viida konstandile saame ka siis, kui rakendame konstantsele muutujale aadressioperatsiooni &:

```
const int *pc; /* viit konstantsele täisarvule */
int *p, i;
const int ic;
pc = &ic;
pc = p = &i;
*p = 5; /* see omistamine on lubatav */
*pc = 5; /* see aga annab vea! */
```

Objekti tüübiga "viit tüübile const T" ei saa otseselt omistada objektile tüübiga "viit tüübile T". Omistamisel tuleb kasutada ilmutatud tüübiteisendust. Jätkame eelmist näidet:

```
pc = &i;    /* lubatav */
pc = p;     /* lubatav */
p  = &ic;   /* vigane */
p  = pc;    /* vigane */
p = (int *) &ic; /* lubatav */
p = (int *)pc;   /* lubatav */
```

Keelereeglid tüübi const jaoks ei ole muidugi täielikud, s.t. ei välista võimalust vastava objekti muutmiseks (näiteks võib viit konstantsele objektile olla funktsiooni parameetriks, funktsioonis aga muudetakse sisu selle viida järgi). Realisatsioon võib konstantsed objektid paigutada mällu, mida pole võimalik modifitseerida ja katse selliste objektide väärtusi muuta võib kaasa tuua programmi avariilõpu.

Võtmesõna volatile on tüübikirjeldaja, mida kasutatakse koos teiste tüüpidega (sealhulgas const, struktuurid, ühendid, loendid) või üksikult (on siis volatile int sünonüüm):

volatile-tüübi-kirjeldaja:

volatile

Tüübikirjeldaja volatile teatab kompilaatorile, et selle tüübiga objekt ei ole realisatsiooni poolt juhitav globaalsetes optimisatsioonides. Selgitame seda põhimõtet lähemalt.

ANSI C toob sisse järjestuspunkti mõiste. Järjestuspunkt eksisteerib iga sellise avaldise täitmise lõpul, mis ei ole mingi teise avaldise osaavaldiseks. Seega eksisteerib järjestuspunkt avaldislause, juhtkonstruktsioonide juhtavaldise, naasmislause avaldise ja algväärtustusavaldise lõpul. Lisaks on järjestuspunktid funktsiooni poole pöördumisel iga argumendi väärtustamise lõpul, loogilistes operatsioonides && ja || iga operandi väärtustamise lõpul, tingimusavaldises enne operatsiooni ? ning komaavaldises pärast iga operandi väärtustamist.

Tüüpi volatile objektide muutmist või viitamist neile ei tohi optimiseerida üle järjestuspunktide, optimiseerimine järjestuspunktide vahel on lubatud. Vaatleme näidet:

```
extern int f();
auto int i,j;
i = f(0);
while(i)
  {
  if(f(j * j)) break;
  }
```

Et muutuja i tsükli sees ei muutu, siis võib optimiseeriv kompilaator selle fragmendi esitada kujul:

```
if(f(0))
  {
  i = j * j;
  while(!f(i));
  }
```

Esimene omistamine muutujale i jäeti ära ning muutujat i kasutatakse vahetulemuse j * j säilitamiseks, mida väärtustatakse ainult üks kord väljaspool tsüklit. Kui aga muutujad i ja j on defineeritud tüüpi volatile

```
auto volatile int i,j;
```

siis selliseid optimiseerimisi sooritada ei lubata.

On lubatud defineerida viita volatile tüübiga objektile. Sel juhul võib optimiseerida viida kasutamist, kuid mitte objekti kasutamist, millele osutab viit. Samuti nagu const korral võib objekti tüübiga "viit tüübile volatile T" omistada objektile tüüpi "viit tüübile T" ainult ilmutatud tüübiteisenduse abil.

Tavaliselt kasutatakse volatile tüüpi objekte juurdepääsuks konkreetsest riistvarast sõltuvaile mäluaadressidele ja registreile, samuti aga ka asünkroonsete protsesside juhtimisel. Sel juhul on garanteeritud nende objektide muutmise või neile viitamise kindel järjekord.

Universaalsed viidad. ANSI C lisab andmetüübi void * kui universaalse viida. Sellistele viitadele ei saa rakendada kaudsustamise või elemendi valiku operatsioone * ja [], kuid neid võib ilma kitsendusteta omistada suvalist tüüpi viidale ja vastupidi. Kuigi selguse mõttes on soovitatav omistamisel lisada ilmutatud tüübiteisendus, pole see kohustuslik:

```
void *gen_ptr;
int *int_ptr;
char *char_ptr;
gen_ptr = int_ptr;  /* lubatav */
int_ptr = gen_ptr;  /* lubatav */
int_ptr = char_ptr  /* vigane */
int_ptr = (int *) char_ptr; /* lubatav */
```

Universaalsed viidad on olulise tähendusega funktsioonide prototüüpides. Kui funktsiooni parameetriks või väärtuseks võib olla suvalist tüüpi viit, siis tuleb ta prototüübis kirjeldada kui void * (muidu peaks tegelik parameeter olema täpselt prototüübis näidatud tüüpi viit, sest viidad ei ole omistamisel üksteiseks teisendatavad). Näiteks on sõnesid kopeeriv teegifunktsioon strcpy seega kirjeldatav kui

```
char *strcpy(char *s1, const char *s2);
```

Funktsioon memcpy parameetriteks ja väärtuseks võivad olla suvalist tüüpi viidad ja seepärast tuleb tema kirjelduses kasutada tüüpi void * :

```
void *memcpy(void *p1, const void *p2, int n);
```

10.5. Teisendused

Omistamisel kasutatakse lk. 88 toodud teisendusreegleid. Eri tüüpi viidad ei ole üksteisele omistatavad. Selliseks omistamiseks saab kasutada universaalset viita void *. Täiendavad kitsendused tulevad sisse veel viitade korral const ja volatile tüüpi objektidele (neid vaatlesime vastavate tüüpide juures).

Harilikud unaarsed teisendused. Tüüpi float ei teisendata tüübiks double: operatsioone teostatakse ka tüüpi float operandidega.

Täisarvutüüpide korral kehtivad järgmised reeglid.

1) Kõik täisarvutüübid, mille pikkus on väiksem kui tüübil int (ja ka nende märgita teisendid) teisendatakse tüüpi int. Märgime, et ehkki tüüpide short ja char pikkus on enamasti väiksem kui tüübil int, ei pea see nii olema igas realisatsioonis.

2) Kõik märgiga täisarvutüübid, mille pikkus on sama kui tüübil int teisendatakse tüüpi int. Kõik sellise pikkusega märgita täisarvutüübid teisendatakse analoogiliselt tüüpi unsigned int.

3) Realisatsioonides, kus sümbolid ja bitiväljad on märgiga teisendatakse nad vastavasse märgiga põhitüüpi (int). Kui sümbolid ja bitiväljad on märgita, siis teisendatakse nad märgita põhitüüpi (unsigned int).

Harilike binaarsete teisenduste korral rakendatakse kummalegi operandile kõigepealt harilikke unaarseid teisendusi. Seejärel toimitakse järgmise algoritmi kohaselt.

1) Kui ükskõik kumb operandidest pole aritmeetilist tüüpi või kui mõlemad operandid on sama tüüpi, siis mingeid lisateisendusi ei teostata.

2) Vastasel korral, kui üks operandidest on tüüpi long double, siis teisendatakse ka teine operand tüüpi long double.

3) Vastasel korral, kui üks operand on tüüpi double, siis teisendatakse ka teine operand tüüpi double.

4) Vastasel korral, kui üks operand on tüüpi float, siis teisendatakse ka teine operand tüüpi float.

5) Vastasel korral, kui üks operand on tüüpi unsigned long int, siis teisendatakse ka teine operand tüüpi unsigned long int.

6) Vastasel korral, kui üks operand on tüüpi long int, siis teisendatakse ka teine operand tüüpi long int.

7) Vastasel korral, kui üks operandidest on tüüpi unsigned int, siis teisendatakse ka teine operand tüüpi unsigned int.

8) Vastasel korral on mõlemad operandid tüüpi int ja mingeid lisateisendusi pole vaja.

Kui funktsioonil pole antud prototüüpi, siis teostatakse funktsiooni tegelike parameetritega harilikud unaarsed teisendused. Kooskõla tagamiseks varasemate realisatsioonidega teisendatakse siin tüüp float tüüpi double. Kui aga on antud funktsiooni prototüüp, siis teisendatakse tegelikud parameetrid vastavalt sellele.

10.6. Avaldised

Muudatused avaldistes on ANSI C korral tingitud põhiliselt masinsõltumatute ja paremini loetavate programmide saamise taotlusest.

Operatsioonides . ja -> peab vasakpoolne operand olema esimesel juhul kindlasti kas struktuur või ühend ja teisel juhul viit struktuurile või ühendile. Erandina võib kasutada ilmutatult teisendatud nullviita.

Funktsiooni poole pöördumisel f(...) võib avaldise f tüübiks olla ka "viit funktsioonile...", mis automaatselt teisendatakse tüüpi funktsioon.

Operatsiooni sizeof tulemus võib olla tüüpi unsigned int või ka unsigned long int. Realisatsiooni poolt valitud tüüp on kirjelduse typedef abil defineeritud kui size_t standardses päisfailis stddef.h.

Aadressi leidmise operatsiooni & võib ANSI C kohaselt rakendada ka funktsioonile, kus tulemuse tüüp on viit funktsioonile. Sama operatsiooni võib rakendada ka massiivile, tulemuse tüüp on sel korral viit massiivile või siis viit viidale. Selline võimalus enamikus teistes realisatsioonides puudub.

Unaarne pluss on täiendav unaarne operatsioon:

unaarne-pluss:

+ unaaravaldis

Unaaravaldist +e võib tõlgendada kui lühemat kirjutusviisi avaldisele 0 + (e). Kõrvalefektina keelab unaarne pluss sellise optimiseerimise, mis järjestab ümber e alamavaldised ja avaldisse e mittekuuluvad alamavaldised. Unaarne pluss annab programmeerijale teatava võimaluse määrata väärtustamise järjekorda. On näiteks teada, et reaalarvude liitmine pole arvutis tegelikult assotsiatiivne. Seega ei pruugi avaldise ((X + Y) - X) - Y väärtus kirjutatud järjekorras väärtustatuna võrduda avaldise (X + Y) - (X + Y) väärtusega, s.o. nulliga. Selleks, et keelata kompilaatoril esimesena antud avaldise optimiseerimine, võib kirjutada

+((+(X + Y)) - X) - Y

Liitmine ja lahutamine. Kahe sama tüüpi viida lahutamise tulemuse tüübiks on täisarvutüüp, mis on defineeritud kui tüüp ptrdiff_t standardses päisfailis stddef.h. Viitu tüüpi void * ei saa üksteisest lahutada, samuti ei saa neile liita ega neist lahutada täisarvulist avaldist.

Võrdlused void * tüüpi viitade vahel ei ole lubatud. Võrdust saab kontrollida järgmiste operandide korral:

1) kui mõlemad operandid on aritmeetilist tüüpi;

2) kui mõlemad operandid on sama tüüpi viidad;

3) kui üks operand on viit tüüpi void * või konstant 0 ja teine operand suvalist tüüpi viit.

Komaoperatsioon konstantavaldises ei ole lubatud. Sama agregaadi kahe komponendi või elemendi aadresside vahe on konstant ja teda võib kasutada konstantses algväärtustajas.

10.7. Laused

Lüliti selektor võib olla suvalist täisarvutüüpi, talle rakendatakse harilikke unaarseid teisendusi. Konstantavaldised märgendeis case peavad seejuures olema teisendatavad samasse tüüpi.

Avaldis naasmislauses teisendatakse funktsiooni kirjelduses antud tüüpi täpselt samuti nagu omistamisel vastavat tüüpi objektile. Kui teisendamine ei ole teostatav saame tulemuseks vea.

10.8. Standardteek

Ehkki keele C standardteeki vaatleme üksikasjalikumalt teatmiku teises osas, teeme siin seoses ANSI C käsitlusega mõningaid märkusi. Eeskätt on oluline, et ANSI C standardiseerib ka teegi, mis oli tegelikult erinevates realisatsioonides kõige vähem kooskõlastatud osa. (See on ka loomulik, kuna teegi koosseisus olevad vahendid sõltuvad oluliselt konkreetsest operatsioonisüsteemist ja arvutist.)

Kõik teegifunktsioonid ja muutujad on ANSI C korral jaotatud teatud hulgaks klassideks, kusjuures igale klassile

vastab oma päisfail, kus on antud kõik selle klassi funktsioonide prototüübid ning nendega seotud tüüpide ja konstantide kirjeldused. Päisfailide nimed on n.-ö. "sisemised", s.t. kehtivad kõikide realisatsioonide korral sõltumata faili nime tegelikust kujust antud operatsioonisüsteemis.

Kõik teegi koosseisus olevad nimed on reserveeritud. Programmeerija ei tohi neid kasutada välisnimedena. Lisaks on süsteemile reserveeritud kõik nimed, mis algavad allkriipsuga _ .

Prototüüpide lisamiseks programmi peab programmeerija direktiividega #include oma programmi koosseisu juurde võtma vajalikud süsteemsed päisfailid. Direktiivide #include järjekord pole oluline.

ANSI C sätestab, et kõik funktsioonidena kirjeldatud teegivahendid on tegelikult ka realiseeritud funktsioonidena, ehkki efektiivsemate programmide saamise huvides on mõningad neist päisfailides makrodena üle määratud. Vaatleme seda situatsiooni hüpoteetilise näite varal. Oletame, et teegis on funktsioon nonzero, mis väljastab väärtuse 1 siis, kui tema tegeliku parameetri väärtus pole 0 ja väärtuse 0 vastasel korral. Päisfailis võib olla antud nii prototüüp kui ka makro:

```
extern int nonzero (int x);
#define nonzero(x) ((int)(x)? 1: 0)
```

Kui programmeerija tahab olla kindel, et ta kasutab nimelt funktsiooni, siis võib ta makro tühistada:

```
#ifdef nonzero
#undef nonzero
#endif
```

Funktsioonid on enamikus realiseeritud selliselt, et argumentide teisendusi peale harilike unaarsete teisenduste ei vajata. See võimaldab funktsioone kasutada ilma prototüüpideta ja seega saavutakse ühilduvus varasemate realisatsioonidega. See ühilduvus aga pole võimalik muutuva arvu parameetritega funktsioonide korral nagu printf, scanf jne. ANSI C puhul saab neid funktsioone korrektselt kasutada vaid koos prototüüpidega.

Lisa. Keele C süntaks

C-programm: (lk. 35)

    väliskirjeldused

väliskirjeldused:

    väliskirjeldus
    väliskirjeldused väliskirjeldus

väliskirjeldus:

    algväärtus-kirjeldus
    funktsiooni-definitsioon

funktsiooni-definitsioon:

    kirjeldajad deklaraator funktsiooni-sisu

kirjeldajad:

    mäluklassi-kirjeldaja
    tüübi-kirjeldaja
    kirjeldajad mäluklassi-kirjeldaja
    kirjeldajad tüübi-kirjeldaja

funktsiooni-sisu:

    kirjeldused blokk

kirjeldused:

    kirjeldus
    kirjeldused kirjeldus

kirjeldus:

    kirjeldajad deklaraatorid ;

deklaraatorid:

    deklaraator
    deklaraatorid , deklaraator

blokk:

    { algväärtus-kirjeldused laused }

algväärtus-kirjeldused: (lk. 36)

    algväärtus-kirjeldus
    algväärtus-kirjeldused algväärtus-kirjeldus

algväärtus-kirjeldus:

    kirjeldajad deklaraatorid-väärtused ;

deklaraatorid-väärtused:

    deklaraator-väärtus
    deklaraatorid-väärtused , deklaraator-väärtus

deklaraator-väärtus:
    deklaraator algväärtustaja
algväärtustaja:
    = algväärtus
deklaraator: (lk. 47)
    lihtdeklaraator
    (deklaraator)
    funktsiooni-deklaraator
    massiivi-deklaraator
    viida-deklaraator
lihtdeklaraator:
    nimi
funktsiooni-deklaraator: (lk. 49)
    deklaraator (parameetrid)
parameetrid:
    nimi
    parameetrid , nimi
massiivi-deklaraator: (lk. 48)
    deklaraator [avaldis]
viida-deklaraator:
    * deklaraator
algväärtus: (lk. 52)
    avaldis
    { algväärtuste-loetelu , }
algväärtuste-loetelu:
    algväärtus
    algväärtuste-loetelu , algväärtus
mäluklassi-kirjeldaja: variandid (lk. 43)
    auto extern register static typedef
tüübi-kirjeldaja: (lk. 45)
    täisarvutüübi-kirjeldaja
    reaalarvutüübi-kirjeldaja
    loenditüübi-kirjeldaja
    struktuuritüübi-kirjeldaja
    ühenditüübi-kirjeldaja
    tüübi-void-kirjeldaja
    typedef-nimi

täisarvutüübi-kirjeldaja: (lk. 60)
    märgiga-tüübi-kirjeldaja
    märgita-tüübi-kirjeldaja
    sümbolitüübi-kirjeldaja
märgiga-tüübi-kirjeldaja:
    short int
    int
    long int
märgita-tüübi-kirjeldaja: (lk. 62)
    unsigned short int
    unsigned int
    unsigned long int
sümbolitüübi-kirjeldaja: (lk. 63)
    unsigned char
reaalarvutüübi-kirjeldaja: variandid (lk. 64)
    float double
loenditüübi-kirjeldaja: (lk. 70)
    loenditüübi-definitsioon
    loenditüübi-viide
loetenditüübi-definitsioon:
    enum nimi { konstandiloetelu }
loenditüübi-viide:
    enum nimi
konstandiloetelu:
    konstandi-kirjeldus
    konstandiloetelu , konstandi-kirjeldus
konstandi-kirjeldus:
    loendikonstant
    loendikonstant = avaldis
loendikonstant:
    nimi
struktuuritüübi-kirjeldaja: (lk. 72)
    struktuuritüübi-definitsioon
    struktuuritüübi-viide
struktuuritüübi-definitsioon:
    struct nimi { s-komponendid }
struktuuritüübi-viide:

struct nimi

s-komponenendid:

s-komponendi-kirjeldus

s-komponendid s-komponendi-kirjeldus

s-komponendi-kirjeldus:

tüübi-kirjeldaja s-deklaraatori-loetelu ;

s-deklaraatori-loetelu:

s-deklaraator

s-deklaraatori-loetelu , s-deklaraator

s-deklaraator:

deklaraator

bitiväli

bitiväli:

deklaraator : avaldis

ühenditüübi-kirjeldaja: (lk. 75)

ühenditüübi-definitsioon

ühenditüübi-viide

ühenditüübi-definitsioon:

union nimi { u-komponendid }

ühenditüübi-viide:

union nimi

u-komponendid:

u-komponendi-kirjeldus

u-komponendid u-komponendi-kirjeldus

u-komponendi-kirjeldus:

tüübi-kirjeldaja deklaraatorid ;

tüübi-void-kirjeldaja: (lk. 79)

void

typedef-nimi: (lk. 80)

nimi

avaldis: (lk. 122)

komaavaldis

komaavaldis:

omistamisavaldis

komaavaldis , omistamisavaldis

omistamisavaldis: (lk. 120)

tingimusavaldis

```
     unaaravaldis omist-op omistamisavaldis
omist-op: variandid
     = += -= *= /= %= <<= >>= &= ^= |=
unaaravaldis:                                    (lk. 103)
     postfiksavaldis
     ilmutatud-tüübiteisendus
     operatsioon-sizeof
     unaarne-miinus
     loogiline-eitus
     bitiinversioon
     aadressi-leidmine
     kaudsustamine
     prefiks-suurendamine
     prefiks-vähendamine
tingimusavaldis:                                 (lk. 119)
     loogiline-OR-avaldis
     loogiline-OR-avaldis ? avaldis : tingimusavaldis
loogiline-OR-avaldis:                            (lk. 118)
     loogiline-AND-avaldis
     loogiline-OR-avaldis || loogiline-AND-avaldis
loogiline-AND-avaldis:
     bitikaupa-OR-avaldis
     loogiline-AND-avaldis && bitikaupa-OR-avaldis
bitikaupa-OR-avaldis:                            (lk. 116)
     bitikaupa-XOR-avaldis
     bitikaupa-OR-avaldis | bitikaupa-XOR-avaldis
bitikaupa-XOR-avaldis:
     bitikaupa-AND-avaldis
     bitikaupa-XOR-avaldis ^ bitikaupa-AND-avaldis
bitikaupa-AND-avaldis:
     võrdusavaldis
     bitikaupa-AND-avaldis & võrdusavaldis
võrdusavaldis:                                   (lk. 115)
     võrdlusavaldis
     võrdusavaldis võrdusop võrdlusavaldis
võrdusop: variandid
     == !=
```

võrdlusavaldis: (lk. 114)
    nihkeavaldis
    võrdlusavaldis võrdlusop nihkeavaldis
võrdlusop: variandid
    < <= > >=
nihkeavaldis: (lk. 113)
    aditiivne-avaldis
    nihkeavaldis nihkeop aditiivne-avaldis
nihkeop: variandid
    << >>

aditiivne-avaldis: (lk. 111)
    multiplikatiivne-avaldis
    aditiivne-avaldis ad-op multiplikatiivne-avaldis
ad-op: variandid
    + -
multiplikatiivne-avaldis: (lk. 109)
    unaaravaldis
    multiplikatiivne-avaldis mult-op unaaravaldis
mult-op: variandid
    * / %
ilmutatud-tüübiteisendus:
    (tüübinimi) unaaravaldis
tüübinimi: (lk. 82)
    tüübi-kirjeldaja abstraktne-deklaraator
abstraktne-deklaraator:
    tühi-abstraktne-deklaraator
    mittetühi-abstraktne-deklaraator
tühi-abstraktne-deklaraator:
mittetühi-abstraktne-deklaraator:
    (mittetühi-abstraktne-deklaraator)
    abstraktne-deklaraator ( )
    abstraktne-deklaraator [avaldis]
    * abstraktne-deklaraator
operatsioon-sizeof: (lk: 104)
    sizeof (tüübinimi)
    sizeof unaaravaldis

unaarne-miinus: (lk. 105)
    - unaaravaldis
loogiline-eitus:
    ! unaaravaldis
bitiinversioon: (lk. 106)
    ~ unaaravaldis
aadressi-leidmine:
    & unaaravaldis
kaudsustamine: (lk. 107)
    * unaaravaldis
prefiks-suurendamine:
    ++ unaaravaldis
prefiks-vähendamine:
    -- unaaravaldis
postfiksavaldis: (lk. 97)
    primaaravaldis
    elemendi-valik
    komponendi-valik
    pöördumine-funktsiooni-poole
    postfiks-suurendamine
    postfiks-vähendamine
elemendi-valik:
    postfiksavaldis [avaldis]
komponendi-valik: (lk. 99)
    otsene-komponendi-valik
    kaudne-komponendi-valik
otsene-komponendi-valik:
    postfiksavaldis . nimi
kaudne-komponendi-valik:
    postfiksavaldis -> nimi
pöördumine-funktsiooni-poole: (lk. 100)
    postfiksavaldis (avaldiste-loetelu)
avaldiste-loetelu:
    omistamisavaldis
    avaldiste-loetelu , omistamisavaldis
postfiks-suurendamine: (lk. 102)
    postfiksavaldis ++

postfiks-vähendamine:
    postfiksavaldis --
primaaravaldis: (lk. 95)
    nimi
    konstant
    sulgavaldis
sulgavaldis: (lk. 97)
    (avaldis)
nimi: (lk. 12)
    allkriips
    täht
    nimi järgmine-sümbol
järgmine-sümbol:
    allkriips
    täht
    number
täht: variandid
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
    a b c d e f g h i j k l m n o p q r s t u v w x y z
allkriips:
    _
number: variandid
    0 1 2 3 4 5 6 7 8 9
konstant: (lk. 14)
    täisarvkonstant
    reaalarvkonstant
    sümbolkonstant
    sõne
täisarvkonstant: (lk. 15)
    kümnendkonstant tüübimarker
    kaheksandkonstant tüübimarker
    kuueteistkümnendkonstant tüübimarker
kümnendkonstant:
    mittenull
    kümnendkonstant number
mittenull: variandid
    1 2 3 4 5 6 7 8 9

kaheksandkonstant:

0

kaheksandkonstant 8-number

8-number: variandid

0 1 2 3 4 5 6 7 8

kuueteistkümnendkonstant:

baasimarker

kuueteistkümnendkonstant 16-number

16-number: variandid

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

baasimarker: variandid

0x 0X

tüübimarker: variandid

l L

reaalarvkonstant: (lk. 17)

numbrid eksponent

punktnumbrid eksponent

eksponent:

e märk numbrid

E märk numbrid

märk: variandid

+ -

punktnumbrid:

numbrid .

numbrid . numbrid

. numbrid

numbrid:

number

numbrid number

sümbolkonstant: (lk. 18)

' sümbol '

sümbol:

trükisümbol

võtmesümbol

trükisümbol:

täht

number

```
     erisümbol
erisümbol: variandid                                    (lk. 8)
     ! ␣ # ^ & * ( ) - _ + = ~ [ ]
     % ¦ ; : { } , . < > / ? $ @ `
sõne:                                                   (lk. 19)
      " sümbolid "
sümbolid:
     sümbol
     sümbolid sümbol
võtmesümbol:
     \ võti
võti:
     võtmemärk
     koodi-tähis
võtmemärk: variandid
     n t b r f v \ ' "
koodi-tähis:
     8-number
     8-number 8-number
     8-number 8-number 8-number
laused:                                                 (lk. 128)
     lause
     laused lause
lause:
     avaldislause
     märgendatud-lause
     blokk
     tingimuslik-lause
     tsükkel
     lüliti
     katkestamislause
     jätkamislause
     naasmislause
     suunamislause
     tühilause
avaldislause:                                           (lk. 129)
     avaldis ;
```

märgendatud-lause: (lk. 130)

    märgend : lause

märgend:

    märgend-nimi
    märgend-case
    märgend-default

märgend-nimi:

    nimi

märgend-case:

    case avaldis

märgend-default:

    default

tingimuslik-lause: (lk. 133)

    if (avaldis) lause **else** lause

tsükkel: (lk. 134)

    tsükkel-while
    tsükkel-do
    tsükkel-for

tsükkel-while:

    while (avaldis) lause

tsükkel-do: (lk. 135)

    do lause **while** (avaldis) ;

tsükkel-for: (lk. 136)

    for (avaldis ; avaldis ; avaldis) lause

lüliti: (lk. 141)

    switch (avaldis) lause

katkestamislause: (lk. 144)

    break ;

jätkamislause:

    continue ;

naasmislause: (lk. 146)

    return avaldis ;

suunamislause:

    goto märgend-nimi ;

tühilause: (lk. 147)

S i s u k o r d

ЯЗЫК ПРОГРАММИРОВАНИЯ Си. Программы для всех.
Составители Тынис К е л ь д е р, Юло К а а з и к.
На эстонском языке. Тартуский университет.
ЭССР, 202400, г. Тарту, ул. Юликооли, 18.
Vastutav toimetaja J. Tapfer. Paljundamisele antud 17.10.1989.
Formaat 60x84/16. Rotaatoripaber. Masinakiri. Rotaprint.
Tingtrükipoognaid 11,16. Arvestuspoognaid 10,54.
Trükipoognaid 12,0. Trükiarv 200. Tell. nr. 712. Hind 35 kop.
TÜ trükikoda. ENSV, 202400 Tartu, Tiigi t. 78.

35 kop.