

Konekielinen ohjelmointi

Muistitilan käyttö ja tiedon sijainti suoritusaikana

Muuttujat

Tietorakenteet: yksi- ja moniulotteiset taulukot, tietueet

Kontrolli: valinta, toisto

Koodin optimointi

Tarkistukset

Copyright Teemu Kerola 2004

Tässä luennossa tutustutaan konekielisen ohjelmoinnin perusasioihin, kuten muuttujien ja yksinkertaisten tietorakenteiden määrittelyyn ja aritmeettisten lausekkeiden toteutukseen. Käymme myös läpi peruskontrollirakenteet: valinnan ja toiston. Aliohjelmien ja käyttöjärjestelmäpalvelujen toteutus käsitellään seuraavalla luennolla. Tutustumme tässä myös optimoinnin peruspiirteisiin ja kuinka koodissa pitäisi toteuttaa tietorakenteiden käytön tarkistukset. Monimutkaisemmista tietorakenteista käymme läpi tietueet, listat ja moniulotteiset taulukot. Ensi alkuun käsittelemme kuitenkin tiedon sijaintia järjestelmässä.

Tiedon sijainti suoritusaikana

Muistissa ("keskusmuistissa")

- ohjelman data-alueella vai koodialueella (konekäskyn osoiteosassa?)
- iso Esim. 1 GB eli 256M 32-bittistä sanaa TTK-91: 512 sanaa
- hidas Esim. 10 ns muistin hakuaika

Rekisterissä (konekäskyssä viitattavassa laiterekisterissä)

- konekäskyssä viitattavassa laiterekisterissä (R4?)
- suorittimen sisäisessä työrekisterissä (TR? IR?)
- pieni Esim 256B eli 64 kpl 32-bittistä sanaa TTK-91: 8 kpl + PC + TR ...
- nopea Esim. 1 ns

Probleema: milloin muuttujan X arvo pidetään muistissa ja milloin rekisterissä?

- missä päin muistia? miten siihen viitataan?
- missä rekisterissä? milloin?

Copyright Teemu Kerola 2004

Ohjelman suoritusaikana käyttämä tieto voi olla useassa eri paikassa ja tällä on paljon merkitystä ohjelman suoritussnopeuteen. Keskusmuisti on käskyjä suorittavien piirien kannalta aika hidas, mutta sillä on hyvänä puolena suuri koko. Nykyaikaisessa pöytäkoneessa voi olla esimerkiksi 1 GB muistia, mutta tiedon hakemiseen muistista voi kulua esimerkiksi 10 ns. Muistin toimintaa tosin käytännössä nopeuttaa automaattisesti toimiva välimuistilaitteisto, mutta unohtakaamme se nyt toistaiseksi. Ttk-91 koneen muisti on oletusarvoisesti 512 32-bittistä sanaa.

Tiedon sijainti suoritusaikana

Muistissa ("keskusmuistissa")

- ohjelman data-alueella vai koodialueella (konekäskyn osoiteosassa?)
- iso Esim. 1 GB eli 256M 32-bittistä sanaa TTK-91: 512 sanaa
- hidas Esim. 10 ns muistin haku aika

Rekisterissä (konekäskyssä viitattavassa laiterekisterissä)

- konekäskyssä viitattavassa laiterekisterissä (R4?)
- suorittimen sisäisessä työrekisterissä (TR? IR?)
- pieni Esim 256B eli 64 kpl 32-bittistä sanaa TTK-91: 8 kpl + PC + TR ...
- nopea Esim. 1 ns

Probleema: milloin muuttujan X arvo pidetään muistissa ja milloin rekisterissä?

- missä päin muistia? miten siihen viitataan?
- missä rekisterissä? milloin?

Copyright Teemu Kerola 2004

Rekisterit ovat nopeita, mutta niitä on rakenteellisista syistä hyvin vähän. Jos rekisterikanta rakennettaisiin kovin suureksi, niin (a) toteutusmenetelmän vuoksi niistä tulisi hitaampia ja (b) niihin viittaamiseksi (osoittamiseksi) tarvittaisiin enemmän bittejä ja siten koodin koko kasvaisi. Todellisissa koneissa on tyypillisesti 32-128 32-bitin rekistereitä, joista osaa voi käyttää pareittain 64-bitin rekistereinä. Ttk-91:ssä on vain kahdeksan käskyssä viitattavaa rekisteriä, joten rekisteriviite eli rekisterin indeksi voidaan koodata konekäskyyn kolmella bitillä.

Tiedon sijainti suoritusaikana

Muistissa ("keskusmuistissa")

- ohjelman data-alueella vai koodialueella (konekäskyn osoiteosassa?)
- iso Esim. 1 GB eli 256M 32-bittistä sanaa TTK-91: 512 sanaa
- hidas Esim. 10 ns muistin hakuaika

Rekisterissä (konekäskyssä viitattavassa laiterekisterissä)

- konekäskyssä viitattavassa laiterekisterissä (R4?)
- suorittimen sisäisessä työrekisterissä (TR? IR?)
- pieni Esim 256B eli 64 kpl 32-bittistä sanaa TTK-91: 8 kpl + PC + TR ...
- nopea Esim. 1 ns

Probleema: milloin muuttujan X arvo pidetään muistissa ja milloin rekisterissä?

- missä päin muistia? miten siihen viitataan?
- missä rekisterissä? milloin?

Copyright Teemu Kerola 2004

Meillä on siis valittavan joko iso ja hidas muisti tai pieni ja nopea rekisterijoukko. Tämä antaa paljon mahdollisuuksia optimointiin, mutta myös ohjelmien toteuttamiseen vähemmän fiksulla tavalla. Ohjelmointikielten kääntäjät käyttävät usein jopa 50% ajastaan siihen, että optimoidaan ohjelman eri muuttujien arvojen sijaintipaikkoja ohjelman suoritusaikana. Esimerkiksi, jonkin loopin aikana muuttuja X arvo kannattaa pitää rekisterissä, mutta muulloin kyseinen arvo on fiksumpaa pitää muistissa, koska kyseisellä rekisterillä on parempaakin käyttöä.

Tiedon viittaamistavat

Tieto on muistissa

- muistiosoitteen (esim 0x6F123456 tai 3459321) avulla
- symbolin (esim. X tai HenkNr) avulla symbolista konekieltä käytettäessä
- rekisterissä tai muistissa olevan osoitteen avulla
- suoritusaikana laskettavan osoitteen avulla

Tieto on välimuistissa

- samalla tavalla kuin jos tieto olisi muistissa
- viittaushetkellä ei tiedetä, löytyykö tieto välimuistista vai keskusmuistista tai kuinka kauan tiedon viittaamiseen kuluu aikaa

Tieto on rekisterissä

- konekielessä rekisterin osoitteen (esim. 6 tai 18) avulla
- symbolisessa konekielessä rekisterin nimen (esim. R3, FP, F5, I2) avulla

Tieto on konekäskyssä (vakio tai osoiteosassa)

- yleensä käskyssä on vain yksi paikka vakiotiedolle

Copyright Teemu Kerola 2004

Muistissa olevaan tietoon viitataan suoraan tunnetun muistiosoitteen perusteella. Esimerkiksi voimme hakea tietoa muistipaikasta 0x6F123456 tai 3459321. Symbolista konekieltä käytettäessä useimmiten käytetään sen sijaan symbolista osoitetta, mikä tietenkin käännoäsaikana sitten vaihdetaan numeeriseen muistiosoitteeseen. On myös mahdollista, että tiedon osoite on valmiina jossakin rekisterissä tai tunnetussa muistipaikassa. Usein osoite myös lasketaan suoritusaikana eri komponenteista konekäskyssä olevien tietojen perusteella. Joskus osoitteen laskentaan tarvitaan useampia konekäskyjä, kuten esimerkiksi moniulotteisten taulukoiden yhteydessä näemme.

Tiedon viittaamistavat

Tieto on muistissa

- muistiosoitteen (esim 0x6F123456 tai 3459321) avulla
- symbolin (esim. X tai HenkNr) avulla symbolista konekieltä käytettäessä
- rekisterissä tai muistissa olevan osoitteen avulla
- suoritusaikana laskettavan osoitteen avulla

Tieto on välimuistissa

- samalla tavalla kuin jos tieto olisi muistissa
- viittaushetkellä ei tiedetä, löytyykö tieto välimuistista vai keskusmuistista tai kuinka kauan tiedon viittaamiseen kuluu aikaa

Tieto on rekisterissä

- konekielessä rekisterin osoitteen (esim. 6 tai 18) avulla
- symbolisessa konekielessä rekisterin nimen (esim. R3, FP, F5, I2) avulla

Tieto on konekäskyssä (vakio tai osoiteosassa)

- yleensä käskyssä on vain yksi paikka vakiotiedolle

Copyright Teemu Kerola 2004

Käytännössä muistiin kohdistuva tiedonhaku oikeasti toteutuu välimuistista 90-99% tapauksista. Välimuisti toimii kuitenkin täysin autonomisesti, eikä edes suoritin tiedä ennen tiedon viittaamista, löytyykö tieto välimuistista vai ei. Välimuistin toimintaa tutkitaan tarkemmin myöhemmillä kursseilla eikä siihen puututa tällä kurssilla paljoakaan. Yleispiirteenä välimuisti (tai oikeammin useat välimuistit) saavat keskusmuistin näyttämään nopeamalta kuin se onkaan. Välimuistiin ei voi suoraan viitata. Ttk-91 esimerkkikoneessa ei ole välimuistia lainkaan.

Tiedon viittaamistavat

Tieto on muistissa

- muistiosoitteen (esim 0x6F123456 tai 3459321) avulla
- symbolin (esim. X tai HenkNr) avulla symbolista konekieltä käytettäessä
- rekisterissä tai muistissa olevan osoitteen avulla
- suoritusaikana laskettavan osoitteen avulla

Tieto on välimuistissa

- samalla tavalla kuin jos tieto olisi muistissa
- viittaushetkellä ei tiedetä, löytyykö tieto välimuistista vai keskusmuistista tai kuinka kauan tiedon viittaamiseen kuluu aikaa

Tieto on rekisterissä

- konekielessä rekisterin osoitteen (esim. 6 tai 18) avulla
- symbolisessa konekielessä rekisterin nimen (esim. R3, FP, F5, I2) avulla

Tieto on konekäskyssä (vakio tai osoiteosassa)

- yleensä käskyssä on vain yksi paikka vakiotiedolle

Copyright Teemu Kerola 2004

Yleensä kaikki laskutoimituksissa käytettävä tieto pyritään pitämään rekistereissä, koska silloin se on nopeasti saatavilla. Rekisteriin osoitetaan symbolisessa konekielessä aina rekisterin nimen perusteella. Todellisissa koneissa rekistereitä on useita eri tyyppisiä ja rekisterin tyyppi on koodattu sen symboliseen nimeen. Esimerkiksi rekisteri F5 voisi olla liukulukurekisteri (floating point -rekisteri). Tällaista rekisteriä voisi sitten käyttää ainoastaan liukulukuoperaatioiden yhteydessä.

Tiedon viittaamistavat

Tieto on muistissa

- muistiosoitteen (esim 0x6F123456 tai 3459321) avulla
- symbolin (esim. X tai HenkNr) avulla symbolista konekieltä käytettäessä
- rekisterissä tai muistissa olevan osoitteen avulla
- suoritusaikana laskettavan osoitteen avulla

Tieto on välimuistissa

- samalla tavalla kuin jos tieto olisi muistissa
- viittaushetkellä ei tiedetä, löytyykö tieto välimuistista vai keskusmuistista tai kuinka kauan tiedon viittaamiseen kuluu aikaa

Tieto on rekisterissä

- konekielessä rekisterin osoitteen (esim. 6 tai 18) avulla
- symbolisessa konekielessä rekisterin nimen (esim. R3, FP, F5, I2) avulla

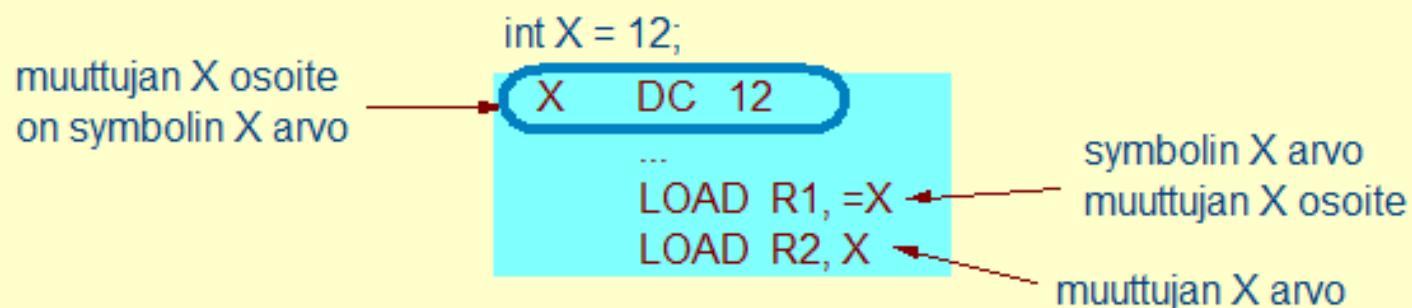
Tieto on konekäskyssä (vakio tai osoiteosassa)

- yleensä käskyssä on vain yksi paikka vakiotiedolle

Copyright Teemu Kerola 2004

Viitattu tieto voi myös olla suoraan konekäskyssä sen vakio-osassa. Tietty vakiotyyppinen tieto on usein järkevää laittaa suoraan konekäskyyn, koska se tulee haetuksi muistista automaattisesti jo käskyn noutamisen yhteydessä. Tällaista tietoa ei voi muuttaa, joten se ei voi olla mikään muuttujan arvo. Se voisi olla vaikkapa vakio 1 konekäskyssä, joka lisää muuttujan arvoon luvun 1. Konekäskyssä oleva tieto on eräs muoto muistissa pidettävästä tiedosta. Ttk-91:ssä vakio-osa on vain 16-bittiä, joten siihen ei mahdu kovin suuria vakioita.

Tieto ja sen osoite



Muuttujan X osoite on 230

Muuttujan X arvo on 12

Symbolin X arvo on 230

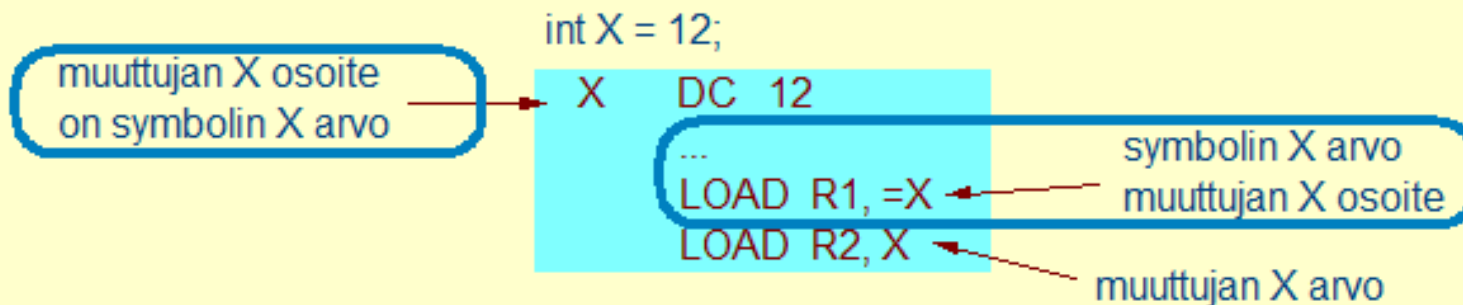
- symbolit ovat yleensä olemassa vain käännösaikana
- virheilmoituksia varten symbolitaulua pidetään joskus mukana myös suoritusaikana

muisti	
	230
	12345
	12556
	128765
	12222
X = 230:	12
	12998

Copyright Teemu Kerola 2004

On hyvin tärkeää ymmärtää, mikä ero on itse tiedolla ja tiedon osoitteella. Konekielisessä ohjelmoinnissa käsittelemme koko ajan molempia. Tässä esimerkissä muuttuja X on talletettu muistipaikkaan 230 ja sen arvona on luku 12. Tämä tilanne on voinut syntyä vaikkapa DC-määrittelystä, jossa varataan tilaa muuttujalle X ja alustetaan se arvoon 12.

Tieto ja sen osoite



Muuttujan X osoite on 230

Muuttujan X arvo on 12

Symbolin X arvo on 230

- symbolit ovat yleensä olemassa vain käännösaikana
- virheilmoituksia varten symbolitaulua pidetään joskus mukana myös suoritusaikana

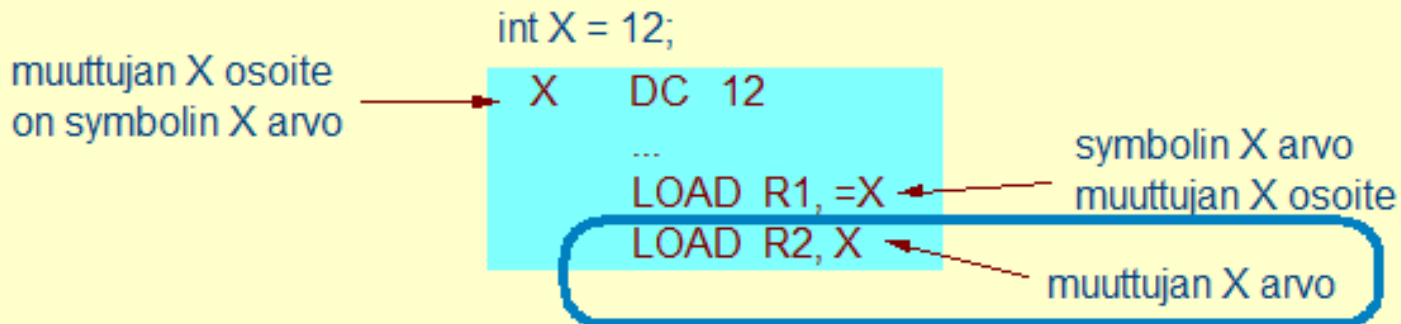
X = 230:

muisti	
230	
12345	
12556	
128765	
12222	
12	
12998	

Copyright Teemu Kerola 2004

Tässä konekäskyssä X:n käyttäminen välittömänä operandina merkitsee, että rekisteri R1 saa uudeksi arvokseen symbolin X arvon sellaisenaan eli luvun 230. Tämä on muuttujan X osoite. Kun muuttuja määritellään DC-käskyllä, niin siihen liittyvällä symbolilla on arvonaan muuttujan osoite.

Tieto ja sen osoite



Muuttujan X osoite on 230

Muuttujan X arvo on 12

Symbolin X arvo on 230

- symbolit ovat yleensä olemassa vain käännösaikana
- virheilmoituksia varten symbolitaulua pidetään joskus mukana myös suoritusaikana

X = 230:

muisti	
230	
12345	
12556	
128765	
12222	
12	
12998	

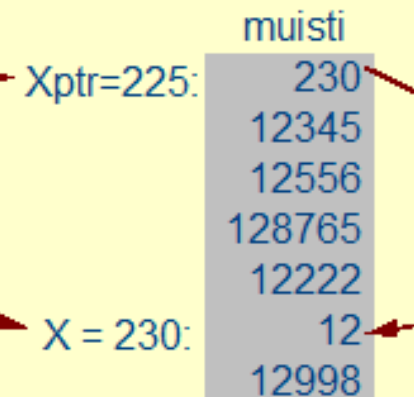
Copyright Teemu Kerola 2004

Kun muuttujaan X viitataan suoraan muistiinviittausmoodia käyttäen, toiseksi operandiksi saadaan muuttujan X arvo 12, eli muistipaikan 230 sisältö. Kokonaislukuarvoisella muuttujalla X on siis osoite 230 ja tämänhetkinen arvo 12. X:n osoite säilyy samana, mutta sen arvo voi vaihtua ohjelman suorituksen edetessä. Muuttuja X on talletettu kääntäjän valitsemaan osoitteeseen 230. Ohjelmoija ei voi vaikuttaa siihen, mihin muistiosoitteeseen kukin DC-muuttuja sijoitetaan - eikä hän yleensä haluakaan. Kaikki muistiosoitteet ovat samanvertaisia tässä suhteessa.

Tieto, tiedon osoite ja tiedon osoitteen osoite

```
Xptr  DC  0  
X     DC  12
```

```
...  
LOAD R1, =X      ; R1 ← 230  
STORE R1, Xptr    ←  
LOAD R2, X        ; R2 ← 12  
LOAD R3, @Xptr    ; R3 ← 12
```



Muuttujan X osoite on 230

Muuttujan X arvo on 12

Osoitinmuuttujan (pointterin) Xptr osoite on 225

Osoitinmuuttujan Xptr arvo on 230

- osoitinmuuttujan arvo on jonkin tiedon osoite
- vrt. kokonaislukumuuttujan arvo on kokonaisluku

Osoitinmuuttujan Xptr osoittaman tiedon arvo on 12

```
C-kieli: X = *ptrSumma; /* osoitinmuuttujan osoittaman muuttujan arvo */  
         *ptrSumma += 5; /* lisää osoitinmuuttujan osoittaman muuttujan arvoa */
```

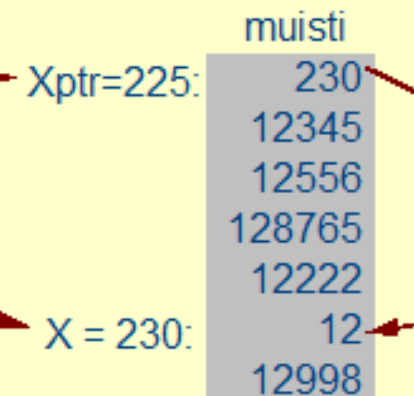
Copyright Teemu Kerola 2004

Tiedon osoite on siis myöskin oma tietotyyppinsä ja se voidaan tallettaa muistiin samalla tavalla kuin mikä tahansa muukin tieto. Tiedon osoite on oikeastaan 32 bittinen etumerkitön kokonaisluku, mutta ttk-91:ssä se talletetaan muistiin tavallisena kokonaislukuna. Konekielen tasolla käsiteltävään tietoon liittyy siis kolme perustavaa laatua olevaa eri käsitettä: tieto, tiedon osoite ja tiedon osoitteenosoite eli tietoon viittaavan osoitinmuuttujan osoite.

Tieto, tiedon osoite ja tiedon osoitteen osoite

```
Xptr  DC  0  
X     DC  12
```

```
LOAD R1, =X    ; R1 ← 230  
STORE R1, Xptr  ; R1 ← 230  
LOAD R2, X      ; R2 ← 12  
LOAD R3, @Xptr  ; R3 ← 12
```



Muuttujan X osoite on 230

Muuttujan X arvo on 12

Osoitinmuuttujan (pointterin) Xptr osoite on 225

Osoitinmuuttujan Xptr arvo on 230

- osoitinmuuttujan arvo on jonkin tiedon osoite
- vrt. kokonaislukumuuttujan arvo on kokonaisluku

Osoitinmuuttujan Xptr osoittaman tiedon arvo on 12

```
C-kieli: X = *ptrSumma; /* osoitinmuuttujan osoittaman muuttujan arvo */  
         *ptrSumma += 5; /* lisää osoitinmuuttujan osoittaman muuttujan arvoa */
```

Copyright Teemu Kerola 2004

Muuttujan X osoite saadaan työrekisteriin R1 käyttämällä toisena operandina symbolin X arvoa ja se talletetaan osoitinmuuttujan Xptr arvoksi tavallisella STORE-käskyllä. Osoitinmuuttujat kannattaa nimetä jollakin erityisellä tavalla, jotta niitä käytettäessä muistaa niiden sisältävän itse tiedon asemesta tiedon osoitteen. Tässä esimerkissä tunnuksen loppuosa 'ptr' indikoi, että kyseessä on osoitinmuuttuja.

Tieto, tiedon osoite ja tiedon osoitteen osoite

```
Xptr  DC  0  
X      DC 12
```

```
...  
LOAD R1, =X      ; R1 ← 230
```

```
STORE R1, Xptr
```

```
LOAD R2, X      ; R2 ← 12
```

```
LOAD R3, @Xptr  ; R3 ← 12
```

muisti

230
12345
12556
128765
12222
12
12998

Xptr=225:

X = 230:

Muuttujan X osoite on 230

Muuttujan X arvo on 12

Osoitinmuuttujan (pointterin) Xptr osoite on 225

Osoitinmuuttujan Xptr arvo on 230

- osoitinmuuttujan arvo on jonkin tiedon osoite
- vrt. kokonaislukumuuttujan arvo on kokonaisluku

Osoitinmuuttujan Xptr osoittaman tiedon arvo on 12

C-kieli: `X = *ptrSumma; /* osoitinmuuttujan osoittaman muuttujan arvo */`
`*ptrSumma += 5; /* lisää osoitinmuuttujan osoittaman muuttujan arvoa */`

Copyright Teemu Kerola 2004

Epäsuora tiedonosoitusmoodi on tarkoitettu juuri osoitinmuuttujien osoittaman tiedon käsittelyyn. Rekisteriin R3 ei siis ladata osoitinmuuttujan Xptr arvoa 230 vaan osoitinmuuttujan Xptr osoittaman tiedon arvo 12. Osoitinmuuttujat voivat sisältää minkä tahansa tiedon osoitteen, esimerkiksi muuttujan osoitteen, taulukon tai jonkun muun tietorakenteen osoitteen, tai käskyn tai aliohjelman osoitteen. Olioihin viitataan aina niiden osoitteen perusteella ja oliossa sen metodit on esitetty metodien osoitteina.

Tieto, tiedon osoite ja tiedon osoitteen osoite

```
Xptr  DC  0
X      DC 12

...
LOAD R1, =X      ; R1 ← 230
STORE R1, Xptr    ←
LOAD R2, X        ; R2 ← 12
LOAD R3, @Xptr    ; R3 ← 12
```

muisti

230
12345
12556
128765
12222
12
12998

Muuttujan X osoite on 230

Muuttujan X arvo on 12

Osoitinmuuttujan (pointterin) Xptr osoite on 225

Osoitinmuuttujan Xptr arvo on 230

- osoitinmuuttujan arvo on jonkin tiedon osoite
- vrt. kokonaislukumuuttujan arvo on kokonaisluku

Osoitinmuuttujan Xptr osoittaman tiedon arvo on 12

C-kieli: `X = *ptrSumma; /* osoitinmuuttujan osoittaman muuttujan arvo */`
`*ptrSumma += 5; /* lisää osoitinmuuttujan osoittaman muuttujan arvoa */`

Copyright Teemu Kerola 2004

Javaassa ei ole explisiittisiä osoitinmuuttujia, koska niiden käyttö on 'vaarallista' eli hyvin virhealtista. C-kieli sen sijaan tukee osoitinmuuttujia ja niiden käyttöä ihan ohjelmointikielen tasolla. Aritmeettisessa lausekkeessa ennen muuttujan nimeä oleva `**`-merkki tarkoittaa, että kyseessä on osoitinmuuttuja ja että halutaan käyttää osoitinmuuttujan osoittaman tiedon arvoa osoitinmuuttujan arvon asemesta. Tässäkin esimerkissä `**`-merkin unohtaminen voi johtaa aika hankalasti havaittavaan virheeseen.

Osoitinmuuttujat

Muuttujia samalla tavoin kuin kokonaislukuarvoiset muuttujatkin

- osoitinmuuttujalla on osoite ja arvo



Osoitinmuuttujan arvo on jonkin tiedon osoite muistissa

- globaalin, kaikkialla ohjelmassa viitattavissa olevan, tiedon osoite
 - muuttuja, taulukko, tietue, olio

dataosoite

- käskyn, aliohjelman tai metodin osoite
 - osoite ohjelmakoodiin

koodiosoite

- keosta (heap) dynaamisesti suoritusaikana varatun tiedon osoite
 - Pascalin tai Javan **new**-operaatio palauttaa keosta varatun muistialueen osoitteen (tai virhekodein, jos esim. keossa ei ole tilaa)

dataosoite

```
ptrRec = new (struct typeRec);
```

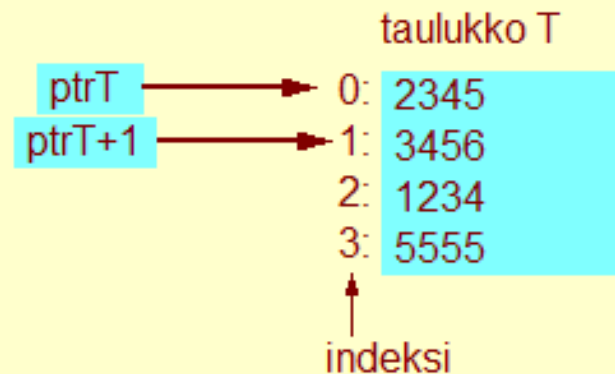
Copyright Teemu Kerola 2004

Osoitinmuuttujat eli pointterit ovat siis muuten aivan tavallisia muuttujia, mutta niiden tietotyyppinä on muistiosoite. Osoitinmuuttujien arvot voidaan asettaa normaalisti, mutta osoitteen käyttöhetkellä niiden tulee olla laillisia tämän ohjelman muistiosoitteita. Niiden käyttötapoja on moninaisia. Esimerkiksi, taulukoita voidaan käsitellä normaaliin tapaan indekseillä indeksoidun tiedonosoitusmuodon perusteella tai sitten niitä voidaan käsitellä pointterien avulla, jolloin seuraavan taulukon alkion osoite saadaan edelliseen alkioon osoittavaa pointteria lisäämällä.

Osoitinmuuttujat

Muuttujia samalla tavoin kuin kokonaislukuarvoiset muuttujatkin

- osoitinmuuttujalla on osoite ja arvo



Osoitinmuuttujan arvo on jonkin tiedon osoite muistissa

- globaalin, kaikkialla ohjelmassa viitattavissa olevan, tiedon osoite
 - muuttuja, taulukko, tietue, olio
- käskyn, aliohjelman tai metodin osoite
 - osoite ohjelmakoodiin

dataosoite

koodiosoite

- keosta (heap) dynaamisesti suoritusaikana varatun tiedon osoite
 - Pascalin tai Javan **new**-operaatio palauttaa keosta varatun muistialueen osoitteen (tai virhekodein, jos esim. keossa ei ole tilaa)

dataosoite

```
ptrRec = new (struct typeRec);
```

Copyright Teemu Kerola 2004

Ohjelmille varataan muistitilaa joko ohjelman latausvaiheessa tai sitten suoritusaikana. Suoritusaikana varattavaa pitkäaikaisesti käytössä olevaa muistitilaa allokoidaan keosta (heap), joka on käyttöjärjestelmän ylläpitämä osa muistia. Keossa olevaa muistia otetaan käyttöön joko käyttöjärjestelmäpalveluilla (kuten. 'malloc' Linuxissa) tai ohjelmointikielten omilla operaatioilla (kuten. Pascalin tai Javan New-operaatio). Ttk-91 kone ei tue keon käyttöä eikä siten dynaamisesti varattavia muistialueita.

Globaali eli kaikkialla käytettävissä oleva data

Globaalit muuttujat ja muut globaalit tietorakenteet sijaitsevat (kääntäjän sijoittamana) ttk-91 koneen muistissa heti ohjelmakoodin jälkeen

"globaali datasegmentti sijaitsee heti koodisegmentin jälkeen"

- muuttujat

```
int X = 25;  
short Y;  
float Ft;
```

```
Char Ch;  
char Str = "Pekka";  
boolean fBig;
```

- tilan varaus

X	DC	15	; alkuarvo 15
Taulu	DS	20	; 20 kokonaislukua
fBig	DC	1	; true = 1, false = 0

- viittaaminen

```
LOAD   R1, X  
STORE  R2, Taulu(R1)  
LOAD   R4, fBig  
JNZER  R4, Iso
```

Copyright Teemu Kerola 2004

DC- ja DS-valekäskyillä määritelty globaali data sijoitetaan muistiin heti ohjelmakoodin jälkeen. Globaali data on pääohjelmatasolla määritelty data, mikä on viitattavissa kaikkialla ohjelmakoodissa. Korkean tason kielissä globaalia dataa on kaikki pääohjelmatasolla määritellyt muuttujat ja tietorakenteet. Toisaalta, globaalia dataa eivät ole esimerkiksi aliohjelmien tai metodien paikalliset muuttujat tai keosta suoritusaikana varatut tietorakenteet. Niiden sijainti muistissa on muualla: pinossa tai keossa. Ttk-91:ssä on tietenkin vain kokonaislukuarvoisia rakenteita, mutta niillä voidaan kyllä hyvin toteuttaa esimerkiksi korkean tason kielen totuusarvotyyppiset muuttujat.

Globaali eli kaikkialla käytettävissä oleva data

Globaalit muuttujat ja muut globaalit tietorakenteet sijaitsevat (kääntäjän sijoittamana) ttk-91 koneen muistissa heti ohjelmakoodin jälkeen

"globaali datasegmentti sijaitsee heti koodisegmentin jälkeen"

- muuttujat

```
int X = 25;  
short Y;  
float Ft;
```

```
Char Ch;  
char Str = "Pekka";  
boolean fBig;
```

- tilan varaus

X	DC	15	; alkuarvo 15
Taulu	DS	20	; 20 kokonaislukua
fBig	DC	1	; true = 1, false = 0

- viittaaminen

```
LOAD    R1, X  
STORE   R2, Taulu(R1)  
LOAD    R4, fBig  
JNZER   R4, Iso
```

Copyright Teemu Kerola 2004

Symbolisella konekielellä globaalit tietorakenteet varataan siis DC- ja DS-pseudokäskyillä. Emme käsittele jatkossa lainkaan tilannetta, missä globaaleja tietorakenteita voisi varata puhtaassa konekieleessä, koska tällaista tilannetta ei tule. Tilanvarausten käsittely symbolitaulun avulla on sen verran monimutkaista. Jos tilanne haluttaisiin ratkaista ilman valmista symbolisen konekielen kääntäjää, niin ratkaisu olisi helpointa tehdä niin, että ensin toteutetaan symbolitaulu ja symbolisen konekielen kääntäjä! Lisäksi pitäisi ottaa huomioon lataajan toiminta, jotta tiedetään, minne päin muistia globaalit muuttujat voisi sijoittaa. Summa summarum, tyydytään nyt käyttämään symbolista konekieltä ihan kiltisti.

Globaali eli kaikkialla käytettävissä oleva data

Globaalit muuttujat ja muut globaalit tietorakenteet sijaitsevat (kääntäjän sijoittamana) ttk-91 koneen muistissa heti ohjelmakoodin jälkeen

"globaali datasegmentti sijaitsee heti koodisegmentin jälkeen"

- muuttujat

```
int X = 25;  
short Y;  
float Ft;
```

```
Char Ch;  
char Str = "Pekka";  
boolean fBig;
```

- tilan varaus

X	DC	15	; alkuarvo 15
Taulu	DS	20	; 20 kokonaislukua
fBig	DC	1	; true = 1, false = 0

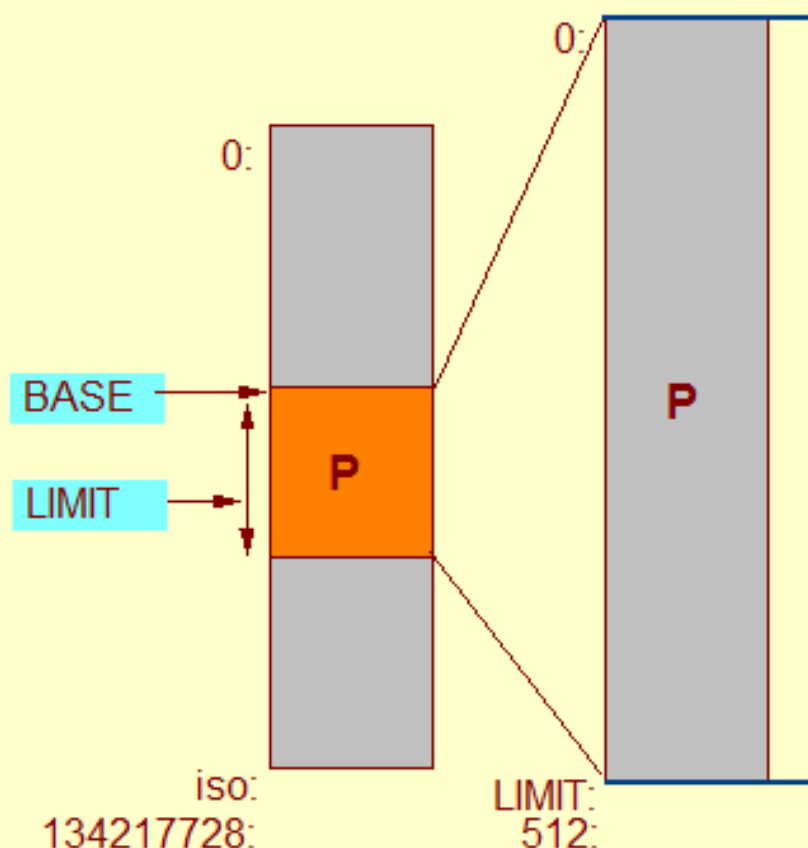
- viittaaminen

```
LOAD    R1, X  
STORE   R2, Taulu(R1)  
LOAD    R4, fBig  
JNZER   R4, Iso
```

Copyright Teemu Kerola 2004

Globaaleihin tietorakenteisiin on helppo viitata, koska niille on annettu symboli niiden määrittelyn yhteydessä. Symbolien arvona on kyseisten tietorakenteiden osoitteet muistissa, joten kyseisiä symboleja käytetään suoraan globaalin datan osoitteina siihen viitattaessa. Aliohjelmien paikallinen data ja keosta suoritusaikana varattu data on hankalammin viitattavissa, koska niihin täytyy viitata aina jonkin vasta suoritusaikana ratkenneen osoitteen perusteella. Symbolien arvothan määritellään jo käännösaikana. Näistä puhutaan lisää myöhemmillä luennoilla.

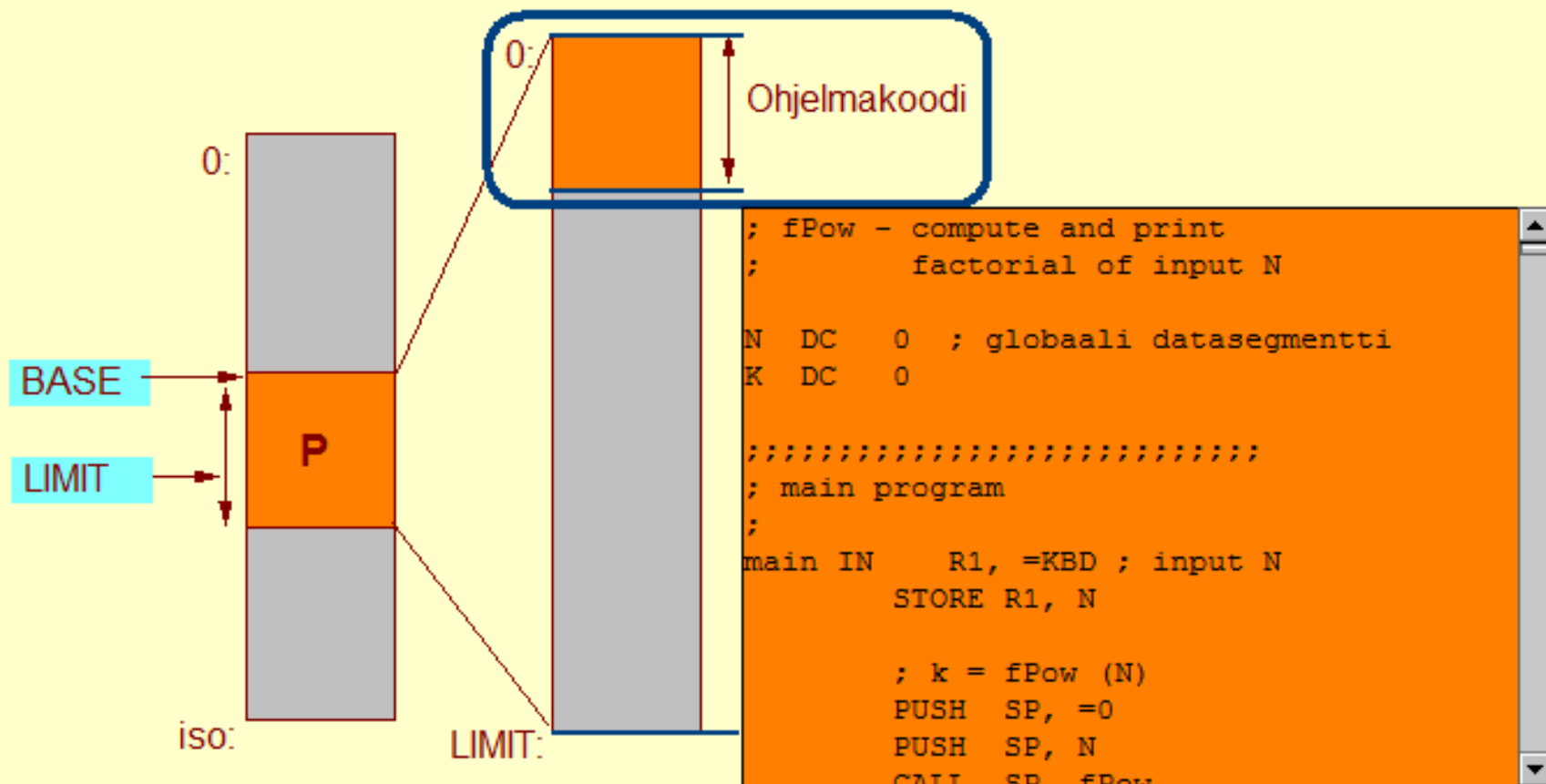
Muistitilan käyttö yhdelle ttk-91 ohjelmalle P



Copyright Teemu Kerola 2004

Mikään ohjelma ei saa koko tietokoneen muistia itselleen. Osa muistista on varattu käyttöjärjestelmäohjelmille ja -datalle. Osa on varattu muille ohjelmille. Ttk-91 koneessa yhden ohjelman käyttöön varattu muistialue on merkitty muistinhallinnan erikoisrekistereillä BASE ja LIMIT, joita ainoastaan käyttöjärjestelmä voi asettaa etuoikeutetuilla konekäskyillä etuoikeutetussa suoritustilassa. BASE-rekisteri osoittaa käytössä olevan muistialueen alkuun ja LIMIT-rekisteri kertoo sen pituuden. Kaikki ohjelman käyttämät osoitteet ovat suhteellisia BASE-rekisterin arvoon ja ovat siis puoliavoimella välillä [0, LIMIT). LIMIT-arvo on yleensä 512, mikä hyvin riittää meidän esimerkiohjelmille.

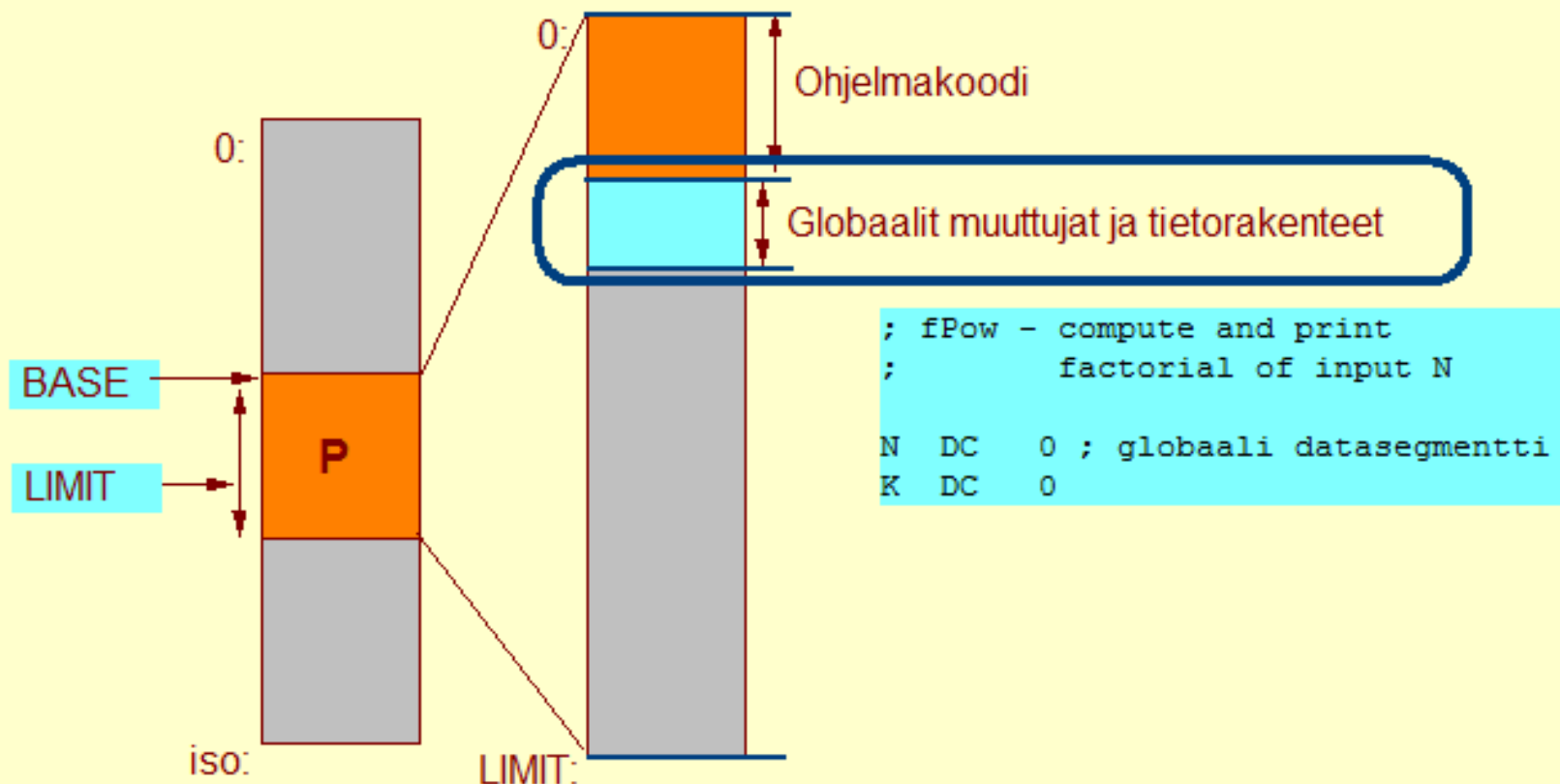
Muistitilan käyttö yhdelle ttk-91 ohjelmalle P



Copyright Teemu Kerola 2004

Ohjelmakoodi sijoitetaan ohjelman muistialueen alkuun. 'Koodisegmentti' alkaa siis ohjelman osoitteesta 0 ja loppuu viimeiseen konekäskyyn. Kaikki ohjelman suoritettava koodi on tällä alueella. Joissakin järjestelmissä koodisegmentti on rajattu tarkemmin, jolloin jokaisen suoritettavan konekäskyn todella täytyy sijaita koodialueella. Joissakin koneissa koodialue on myös suojattu muutoksilta, joten sinne ei voi tallettaa dataa suoritusajana. Ttk-91:ssä ja sen kuvitellussa käyttöjärjestelmässä ei ole mitään tällaisia rajoitteita tai suojauksia. Ohjelmakoodia voi siis jopa muuttaa suoritusajana!

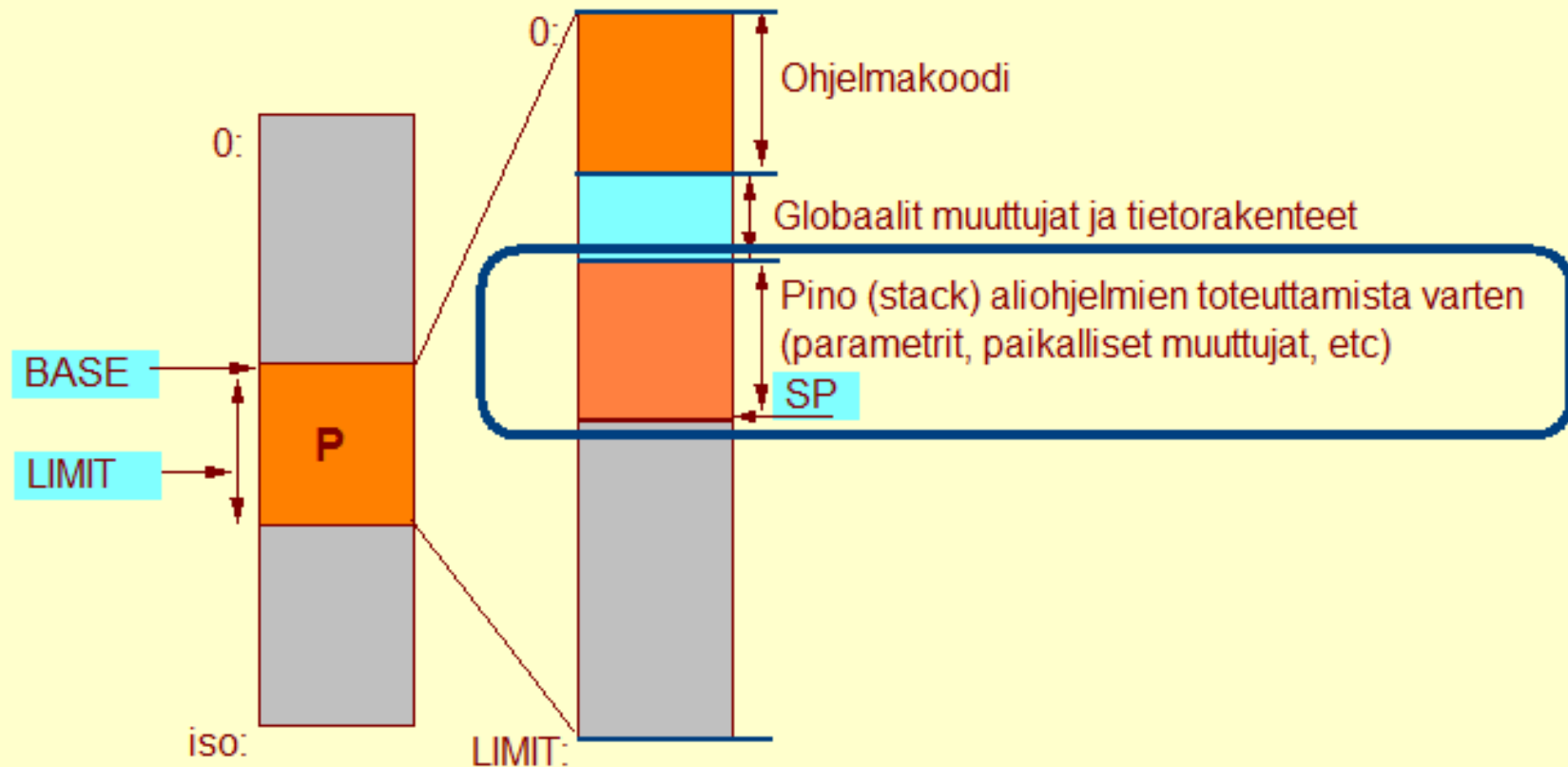
Muistitilan käyttö yhdelle ttk-91 ohjelmalle P



Copyright Teemu Kerola 2004

Symbolisen konekielen kääntäjä sijoittaa kaikki DC- ja DS-pseudokäskyillä varatun datan heti ohjelmakoodin jälkeen. 'Globaali datasegmentti' sijaitsee siis välittömästi koodisegmentin jälkeen. Joissakin koneissa globaali datasegmentti on osoitettu tarkemmin erikoisrekistereillä, mutta ttk-91:ssä se on vain muistialue koodin jälkeen. Varsinaisen tilanvarauksen tekee lataaja kääntäjän antamien ohjeiden perusteella. Lataaja myös alustaa globaalin datan kääntäjän ilmoittamien tietojen mukaiseksi. Ohjelman tulisi itse (siis koodissa) alustaa kaikki muu data (esim. DS-määritetty) ennen sen käyttöä.

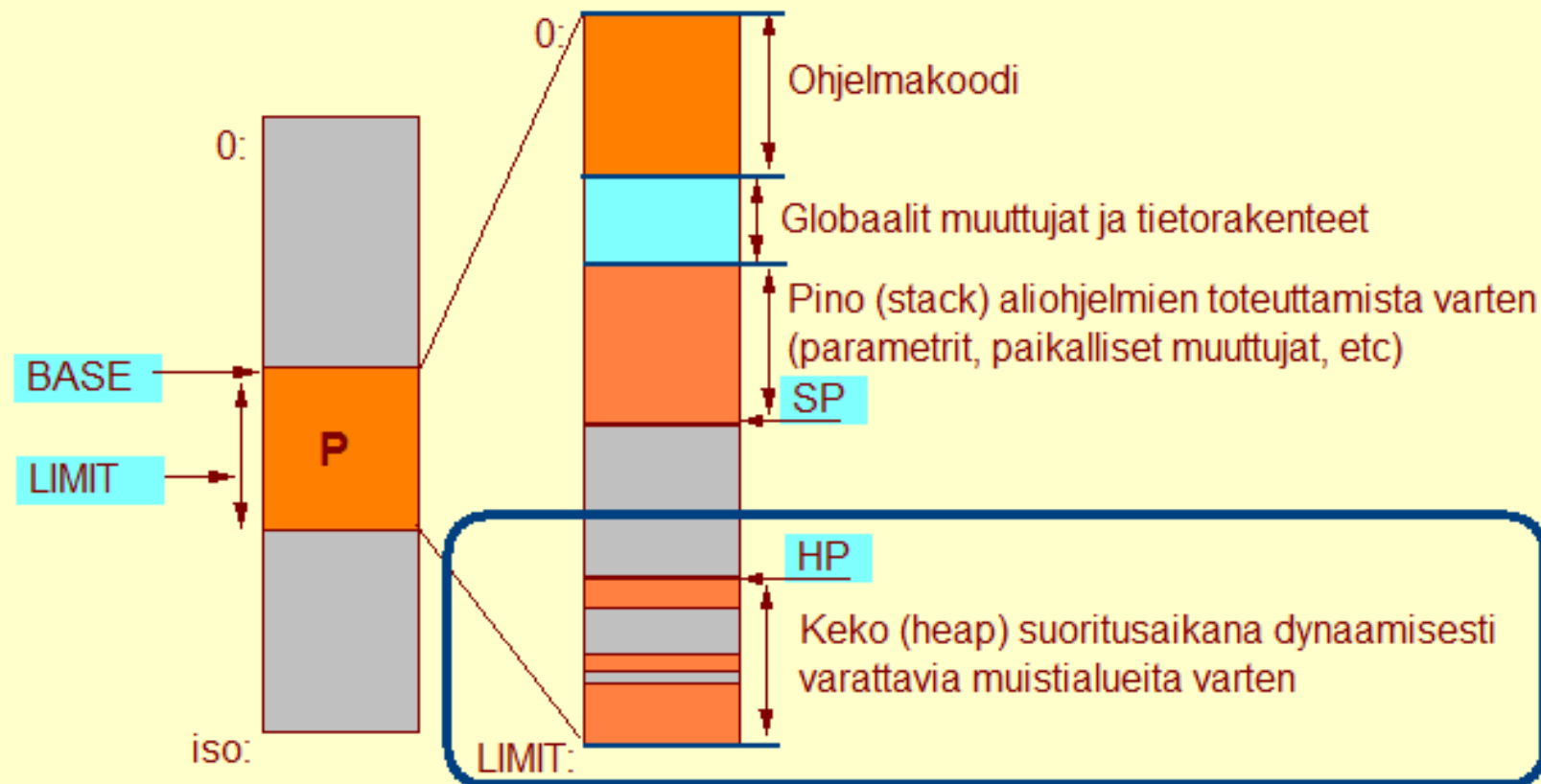
Muistitilan käyttö yhdelle ttk-91 ohjelmalle P



Copyright Teemu Kerola 2004

Pino (stack) on ttk-91-koneessa heti globaalin datasegmentin jälkeen. Sen alkukohta on kääntäjän määrittelemä vakio (koodisegmentin koko + datasegmentin koko). Pinon koko vaihtelee suoritusaikana, mutta pinorekisteri (SP eli rekisteri R6 ttk-91:ssä) osoittaa sen 'pinnalle' koko ajan. Pino kasvaa aina aliohjelmaa kutsuttaessa ja se pienenee sieltä palatessa. Pinoon talletetaan kaikki aliohjelmien toteutuksessa tarvittava tieto, joka käsitellään tarkemmin seuraavalla luennolla. Joissakin koneissa pinoa käytetään myös laskutoimitusten välitulosten tallettamiseen, mutta me käytämme sitä ttk-91:ssä ainoastaan aliohjelmien toteuttamiseen. Välitulokset talletetaan rekistereihin.

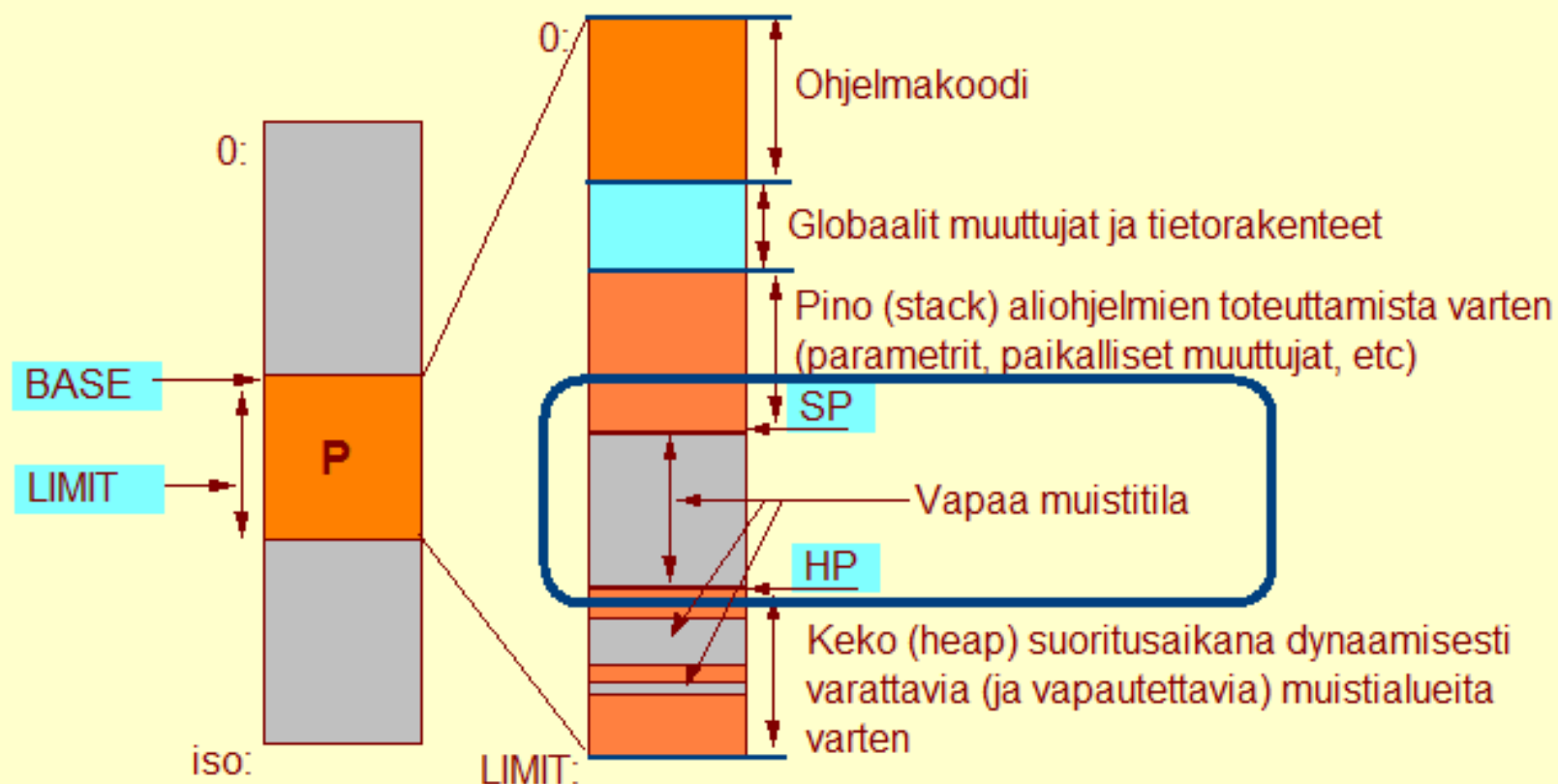
Muistitilan käyttö yhdelle ttk-91 ohjelmalle P



Copyright Teemu Kerola 2004

Keko sijaitsee yleensä ohjelmalle varatun muistialueen lopussa, samalla alueella kuin pinokin. Keon loppuosoite on siten vakio (tässä muistipaikka LIMIT-1), mutta sen alkuosoite vaihtelee dynaamisesti suoritusaikana. Keon alkuun osoittaa kekorekisteri HP (heap pointer). Kun ohjelma varaa muistitilaa keosta, tila löytyy joko keon sisältä aikaisemmin vapautetulta alueelta tai sitten keon alusta kekoa kasvattamalla. Jos HP 'törmää' SP'hen (tai päinvastoin), ohjelman suoritus keskeytyy 'stack overflow' virheeseen, mikä on hyvin tuttu ainakin rekursiota opetteleville ohjelmoijille. Ttk-91 koneessa ei ole kekoa eikä siis HP rekisteriäkään.

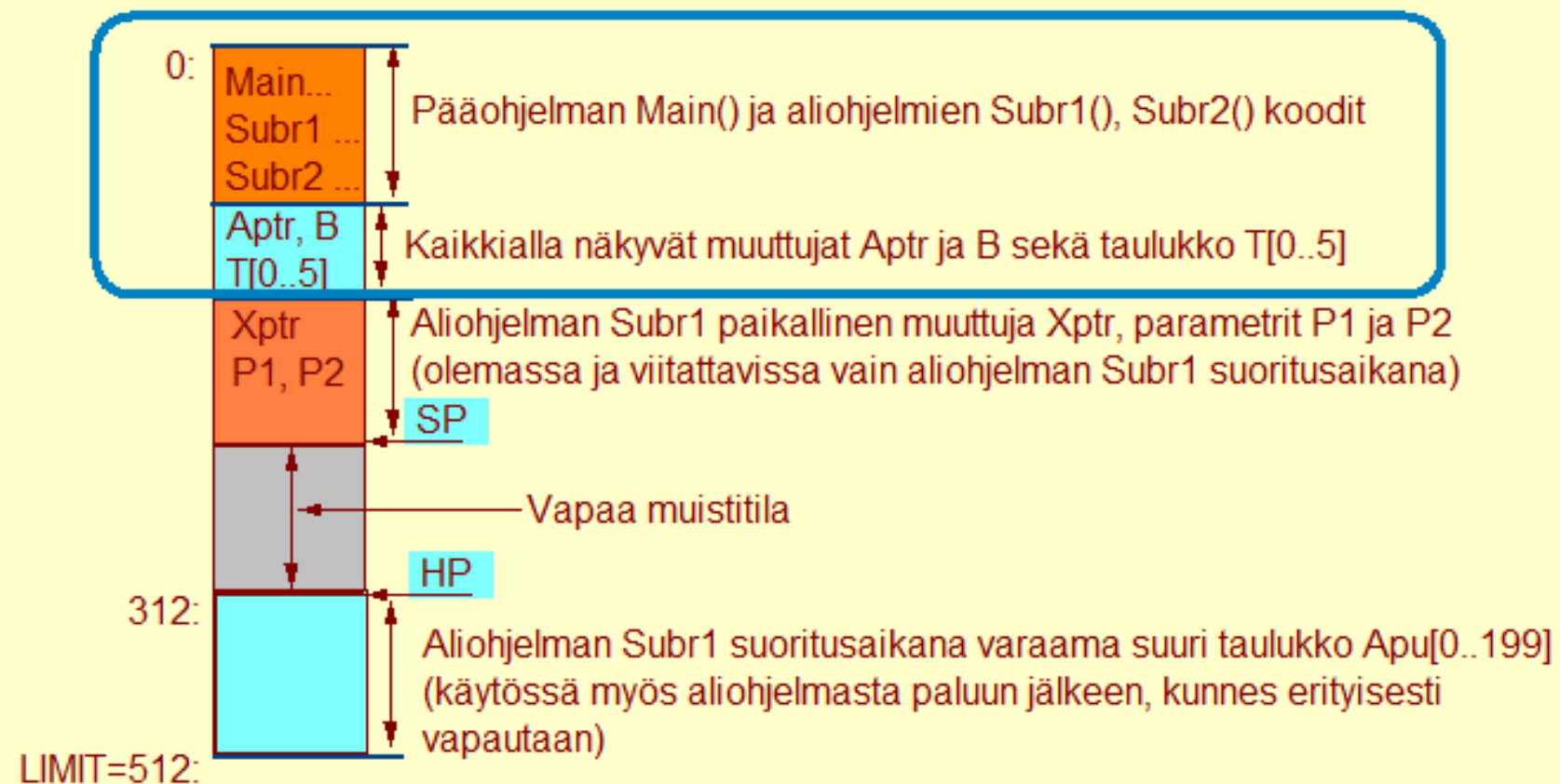
Muistitilan käyttö yhdelle ttk-91 ohjelmalle P



Copyright Teemu Kerola 2004

Ohjelmalle varatusta muistialueesta vapaana on siis kaikki SP:n ja HP:n väliin jäävä muisti ja keon sisällä oleva suoritusaikana joskus varattu ja sitten myöhemmin vapautettu muistialue. Keon sisälle voi ohjelman pitkäaikaisen suorituksen aikana kertyä paljonkin kovin pieniä vapaita muistialueita, joita muistinhallinta tarpeen tullen yhdistelee. Yhdistely on iso toimenpide ja muistinhallinta keskeyttää ohjelman normaalin suorituksen sen ajaksi. Toinen vaihtoehto on vain todeta ohjelmalle, että vapaata muistia ei ole tarpeeksi, vaikka sitä oikeasti onkin, mutta useana käyttökelvottoman pieninä paloina. Ttk-91:ssä ei siis ole kekoa, joten vapaa muisti on kaikki SP:n ja LIMIT'in välissä oleva muistitila.

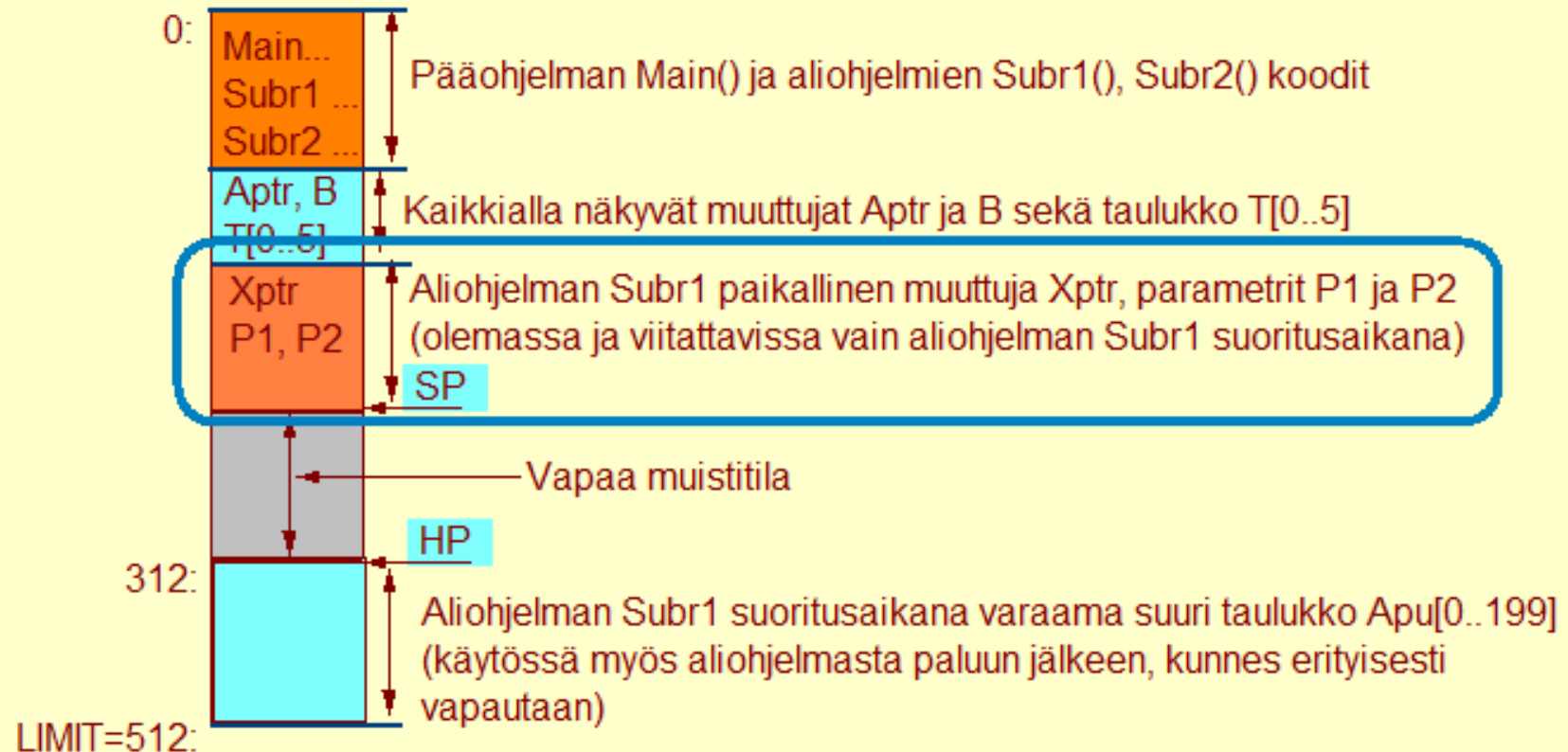
Esimerkki: muistin käyttö aliohjelman Subr1 suorituksen aikana



Copyright Teemu Kerola 2004

Tarkastellaan muistin käyttöä vielä yhden konkreettisen esimerkin varjolla. Oletetaan, että ohjelmassa on pääohjelma ja kaksi aliohjelmää: Subr1 ja Subr2. Suoritusajana sekä pääohjelman että aliohjelmien kaikki koodi on sijoitettu alkamaan muistipaikasta 0. Pääohjelman ensimmäinen suoritettava käsky eli rekisterin PC alkuarvo on 0. Ohjelmassa on pääohjelmatasolla määritelty kaksi muuttujaa (Aptr ja B) sekä 6-alkioinen taulukko T. Nämä on sijoitettu muistiin heti koodialueen jälkeen. Globaalin data-segmentin koko on siis 8 sanaa.

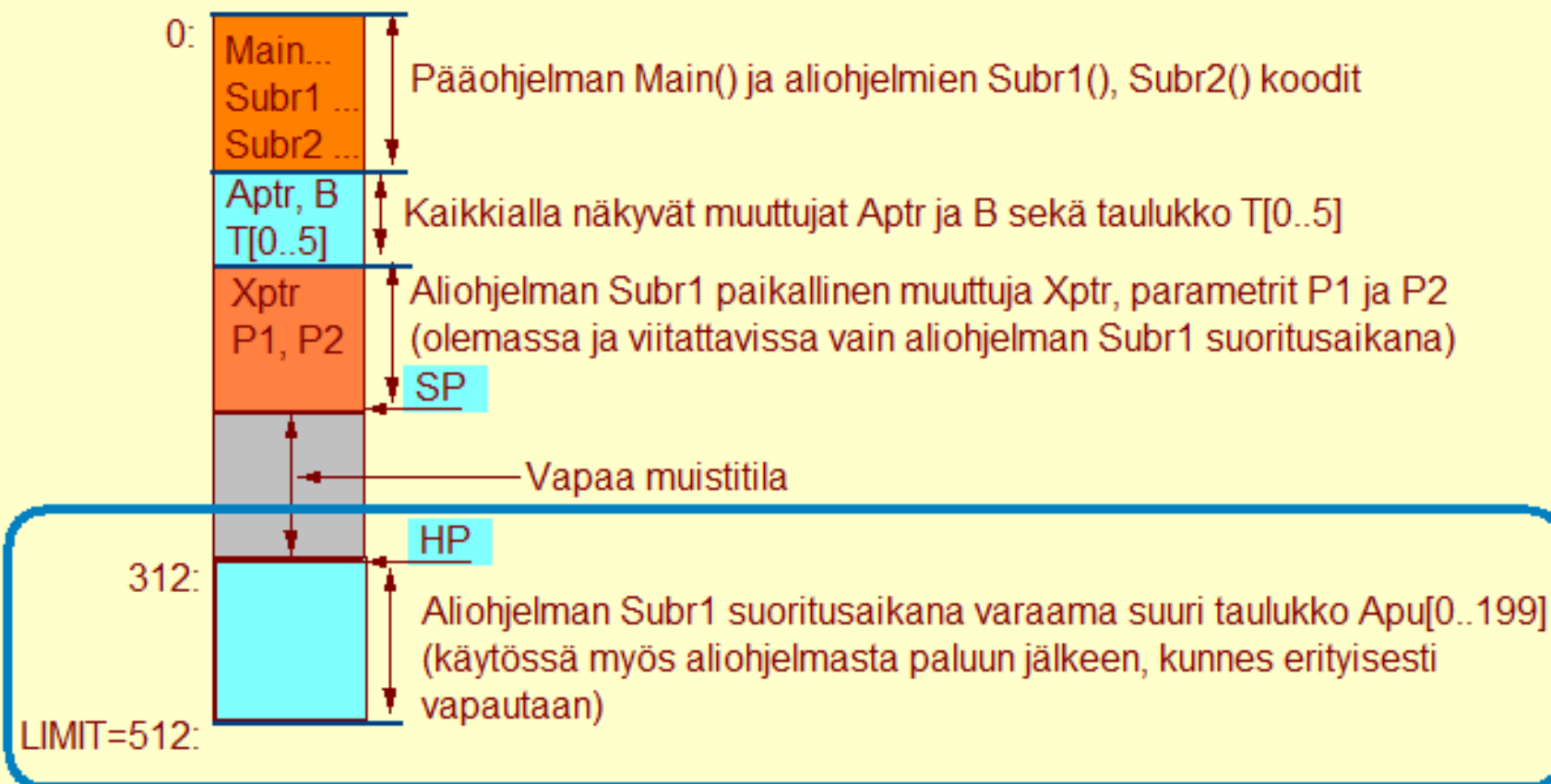
Esimerkki: muistin käyttö aliohjelman Subr1 suorituksen aikana



Copyright Teemu Kerola 2004

Oletetaan nyt, että pääohjelma on kutsunut suoraan aliohjelmää Subr1. Aliohjelman Subr1 suorituksen aikana sen paikallinen muuttuja Xptr ja sen parametrit on talletettu pinnoon. Niiden tarkkaa sijaintia muistissa ei ole etukäteen määritetty, mutta niiden sijainti on tunnettu suhteessa pinorekisterin nykyiseen arvoon. Itse asiassa, eri kutsukerroilla aliohjelman Subr1 paikallinen data voi sijaita eri kohtaa pinossa, mutta kyseisen datan sijainti pinorekisterin suhteen on aina sama. Huomaa, että aliohjelman Subr2 paikallisia tietoja ei ole lainkaan muistissa tällä hetkellä.

Esimerkki: muistin käyttö aliohjelman Subr1 suorituksen aikana



Copyright Teemu Kerola 2004

Aliohjelma Subr1 varaa ensimmäisellä kutsukerralla suuren työtaulukon Apu, joka allokoidaan keosta. Aliohjelman Subr1 on viisasta tallettaa taulukon Apu osoite johonkin globaaliin muuttujaan (esim. Aptr), jos se haluaa pystyä viittaamaan siihen myös seuraavalla kutsukerralla. Jos taulukon Apu osoite pidetään ainoastaan aliohjelman paikallisessa muuttujassa (esim. Xptr), joka häviää aliohjelmasta paluun yhteydessä, niin taulukko Apu jää varatuksi, mutta siihen ei päästä enää käsiksi.

Muistissa oleva data

Globaali data

- varataan ohjelman latauksen yhteydessä
- viitattavissa (käytettävissä) aina
- viittaus tapahtuu symbolin nimen (staattisen osoitteen) avulla

```
int X; float Y;
```

Dynaaminen keosta varattava data

- varataan tarvittaessa keosta suoritusaikana
- vapautetaan kun ei enää tarvita, tai
muistinhallinta vapauttaa automaattisesti ei-käytössä olevat alueet
- viitattavissa (käytettävissä) aina
- viittaus allokointiaikana palautetun osoitteen perusteella

```
pS = new (struct typeS);
```

Aliohjelmien paikallinen data

- varataan pinosta aliohjelman kutsun yhteydessä
- vapautetaan aliohjelmasta poistuttaessa
- viitattavissa (käytettävissä) vain aliohjelman suorituksen aikana
- viittaus aliohjelman sisällä suhteellisen osoitteen (SP tai FP) perusteella

```
parametrit  
paikalliset muuttujat
```

Copyright Teemu Kerola 2004

Ohjelman data-alue on siis kolmessa osassa. Globaali data on varattu ohjelmalle jo latauksen yhteydessä ja sen sijainti on siis etukäteen tunnettu. Tällaiseen tietoon on helppo viitata symbolisessa konekielessä määriteltyjen symbolien avulla.

Muistissa oleva data

Globaali data

- varataan ohjelman latauksen yhteydessä
- viitattavissa (käytettävissä) aina
- viittaus tapahtuu symbolin nimen (staattisen osoitteen) avulla

```
int X; float Y;
```

Dynaaminen keosta varattava data

- varataan tarvittaessa keosta suoritusaikana
- vapautetaan kun ei enää tarvita, tai
muistinhallinta vapauttaa automaattisesti ei-käytössä olevat alueet
- viitattavissa (käytettävissä) aina
- viittaus allokointiaikana palautetun osoitteen perusteella

```
pS = new (struct typeS);
```

Aliohjelmien paikallinen data

- varataan pinosta aliohjelman kutsun yhteydessä
- vapautetaan aliohjelmasta poistuttaessa
- viitattavissa (käytettävissä) vain aliohjelman suorituksen aikana
- viittaus aliohjelman sisällä suhteellisen osoitteen (SP tai FP) perusteella

```
parametrit  
paikalliset muuttujat
```

Copyright Teemu Kerola 2004

Suoritusaikana mikä tahansa pitkäaikasta käyttöä vaativa tietorakenne tulee varata keosta. Allokoinnin yhteydessä varausrutiini palauttaa varatun tietorakenteen osoitteen ja kaikki viitteet tietorakenteeseen tapahtuvat sen jälkeen tuon osoitteen avulla. Yleensä osoite talletetaan johonkin globaaliin osoitinmuuttujaan. Varattu tietorakenne säilyy varattuna, kunnes se joko explisiittisesti tai automaattisesti vapautetaan. Muistatthän, että Ttk-91 koneessa ei ole kekoa.

Muistissa oleva data

Globaali data

- varataan ohjelman latauksen yhteydessä
- viitattavissa (käytettävissä) aina
- viittaus tapahtuu symbolin nimen (staattisen osoitteen) avulla

```
int X; float Y;
```

Dynaaminen keosta varattava data

- varataan tarvittaessa keosta suoritusaikana
- vapautetaan kun ei enää tarvita, tai
muistinhallinta vapauttaa automaattisesti ei-käytössä olevat alueet
- viitattavissa (käytettävissä) aina
- viittaus allokointiaikana palautetun osoitteen perusteella

```
pS = new (struct typeS);
```

Aliohjelmien paikallinen data

- varataan pinosta aliohjelman kutsun yhteydessä
- vapautetaan aliohjelmasta poistuttaessa
- viitattavissa (käytettävissä) vain aliohjelman suorituksen aikana
- viittaus aliohjelman sisällä suhteellisen osoitteen (SP tai FP) perusteella

```
parametrit  
paikalliset muuttujat
```

Copyright Teemu Kerola 2004

Aliohjelmien kaikki paikallinen data allokoidaan pinosta ja tällaiseen tietoon viitataan sitten tiedon suhteellisen osoitteen perustella. Tiedon sijainti tunnetaan joko SP'hen tai FP'hen (R7) suhteutettuna. On huomattava, että kaikki aliohjelmien oma data häviää täysin aliohjelmasta paluun yhteydessä. Aliohjelmien paikalliset rakenteet ovat siis olemassa vain aliohjelmien suoritusaikana. Pinon käyttö aliohjelmien toteutuksessa käydään läpi seuraavalla luennolla.

Tiedon sijainti laitteistossa suoritusaikana

Rekisteri (nopea)

- kääntäjä päättää, milloin muuttujan arvo pidetään rekisterissä (ja missä rekisterissä)

Välimuisti (nopea)

- muistinhallintalaitteisto hoitaa suoritusaikana automaattisesti joillekin muistialueille

Muisti (hidas)

- kääntäjä valitsee sijaintipaikan
 - ◆ globaali data datasegmenttiin, varataan ja alustetaan ohjelman latauksen yhteydessä
 - ◆ pienet vakiot voidaan sijoittaa myös konekäskyjen vakio-osaan koodisegmentissä
- ohjelma sijoittaa suoritusaikana
 - ◆ aliohjelmien paikallinen data ja parametrit pinossa
- ohjelmointikielen tai käyttöjärjestelmä palveluohjelma sijoittaa suoritusaikana
 - ◆ dynaaminen data keossa

Levy, levypalvelin (aivan liian hidas)

- vaatii käyttöjärjestelmän palveluohjelmien apua tiedon kopiointiin muistiin
- mitä tehdä (mahdollisesti hyvin pitkänä) odotusaikana?

Copyright Teemu Kerola 2004

Ohjelman suoritusaikana viittaama tieto voi olla useassa fyysisesti eri tyyppisessä muistissa. Rekisteri on kaikkein nopeinta muistia, mutta niitä on vain muutama koko laitteistossa. Niiden käytön suhteen pitää siis olla säästäväinen. Kääntäjä päättää, minkä muuttujan arvoa kulloinkin pidetään missäkin rekisterissä ja milloin ei missään. Esimerkiksi jonkin globaalin muuttujan X arvo kannattaa pitää rekisterissä tiukan silmukan aikana, jos joka kierroksella X:ään viitataan. Muulloin taas X:n arvo pysyy hyvässä tallessa muistissa. Konekielellä ohjelmoitaessa ohjelmoija tietenkin itse tekee nämä päätökset.

Tiedon sijainti laitteistossa suoritusaikana

Rekisteri (nopea)

- kääntäjä päättää, milloin muuttujan arvo pidetään rekisterissä (ja missä rekisterissä)

Välimuisti (nopea)

- muistinhallintalaitteisto hoitaa suoritusaikana automaattisesti joillekin muistialueille

Muisti (hidas)

- kääntäjä valitsee sijaintipaikan
 - ◆ globaali data datasegmenttiin, varataan ja alustetaan ohjelman latauksen yhteydessä
 - ◆ pienet vakiot voidaan sijoittaa myös konekäskyjen vakio-osaan koodisegmentissä
- ohjelma sijoittaa suoritusaikana
 - ◆ aliohjelmien paikallinen data ja parametrit pinossa
- ohjelmointikielen tai käyttöjärjestelmä palveluohjelma sijoittaa suoritusaikana
 - ◆ dynaaminen data keossa

Levy, levypalvelin (aivan liian hidas)

- vaatii käyttöjärjestelmän palveluohjelmien apua tiedon kopiointiin muistiin
- mitä tehdä (mahdollisesti hyvin pitkänä) odotusaikana?

Copyright Teemu Kerola 2004

Välimuisti on lähes yhtä nopea kuin rekisteri, mutta sen toiminta on täysin autonomista. Joskus muistista haettu alkio löytyy välimuistista ja joskus ei. Välimuistin toiminta on aika monimutkaista ja onkin parempi, että ohjelmoijan ei tarvitse murehtia siitä lainkaan. Hänelle riittää tieto, että välimuisti (tai oikeammin usea eri tasoinen välimuisti) nopeuttaa muistin toimintaa huomattavasti. Käytännössä välimuistin toiminnan tehokkuuteen voi vaikuttaa hyvällä ohjelmointikurilla, jos ohjelmoija ymmärtää välimuistin toimintaperiaatteet oikein hyvin. Emme puutu välimuistiin tällä kurssilla paljoakaan, eikä ttk-91:ssä ole välimuistia.

Tiedon sijainti laitteistossa suoritusaikana

Rekisteri (nopea)

- kääntäjä päättää, milloin muuttujan arvo pidetään rekisterissä (ja missä rekisterissä)

Välimuisti (nopea)

- muistinhallintalaitteisto hoitaa suoritusaikana automaattisesti joillekin muistialueille

Muisti (hidas)

- kääntäjä valitsee sijaintipaikan
 - ◆ globaali data datasegmenttiin, varataan ja alustetaan ohjelman latauksen yhteydessä
 - ◆ pienet vakiot voidaan sijoittaa myös konekäskyjen vakio-osaan koodisegmentissä
- ohjelma sijoittaa suoritusaikana
 - ◆ aliohjelmien paikallinen data ja parametrit pinossa
- ohjelmointikielen tai käyttöjärjestelmä palveluohjelma sijoittaa suoritusaikana
 - ◆ dynaaminen data keossa

Levy, levypalvelin (aivan liian hidas)

- vaatii käyttöjärjestelmän palveluohjelmien apua tiedon kopiointiin muistiin
- mitä tehdä (mahdollisesti hyvin pitkänä) odotusaikana?

Copyright Teemu Kerola 2004

Pääosa viitatus datasta on muistissa. Kääntäjä valitsee, sijoitetaanko pieni vakio data-alueelle vai konekäskyjen vakio-osaan. Pinosta varataan muistialueita vain pinon pinnalta aliohjelmakutsujen yhteydessä. Varauksen tekee kutsukohdassa oleva koodi. Erilaiset palveluohjelmat hoitavat keon käsittelyn, eikä ohjelmoijalla ole mahdollisuutta vaikuttaa, mistä päin kekoa hänen dynaamisesti varaamansa muistialue allokoidaan. Yleisesti ottaen, kaikki ohjelman käyttämä tieto on muistissa ja se kopioidaan rekistereihin ainoastaan tiedon manipuloinnin ajaksi.

Tiedon sijainti laitteistossa suoritusaikana

Rekisteri (nopea)

- kääntäjä päättää, milloin muuttujan arvo pidetään rekisterissä (ja missä rekisterissä)

Välimuisti (nopea)

- muistinhallintalaitteisto hoitaa suoritusaikana automaattisesti joillekin muistialueille

Muisti (hidas)

- kääntäjä valitsee sijaintipaikan
 - ◆ globaali data datasegmenttiin, varataan ja alustetaan ohjelman latauksen yhteydessä
 - ◆ pienet vakiot voidaan sijoittaa myös konekäskyjen vakio-osaan koodisegmentissä
- ohjelma sijoittaa suoritusaikana
 - ◆ aliohjelmien paikallinen data ja parametrit pinossa
- ohjelmointikielen tai käyttöjärjestelmä palveluohjelma sijoittaa suoritusaikana
 - ◆ dynaaminen data keossa

Levy, levypalvelin (aivan liian hidas)

- vaatii käyttöjärjestelmän palveluohjelmien apua tiedon kopiointiin muistiin
- mitä tehdä (mahdollisesti hyvin pitkänä) odotusaikana?

Copyright Teemu Kerola 2004

Osa ohjelman viittaamasta tiedosta voi kaikesta huolimatta sijaita levyllä esimerkiksi sen vuoksi, että keskusmuisti on liian pieni kaikelle ohjelman käsittelemälle tiedolle. Levy on kuitenkin aivan liian hidas media ohjelman suorituksen kannalta, eikä ohjelman suoritus voi edetä normaalisti levyltä luvun yhteydessä. Käyttöjärjestelmä puuttuu asiaan ja hoitaa tiedon kopioinnin levyltä muistiin. Tämän jälkeen käyttöjärjestelmä antaa jollain tavalla alkuperäiselle ohjelmalle tiedon asiasta ja ohjelman suoritus voi jatkua normaalisti. Mutta tietenkin viitattu data sijaitsee nyt muistissa eikä levyllä.

Konekielisen ohjelmoinnin peruskäsitteet

Aritmeettinen lauseke

- miten tehdä laskutoimitukset?

$X = 2 + 4 * Y$

$H := 3.8765 + 34.56/Z;$

Yksinkertaiset tietorakenteet

- yksiulotteiset taulukot, tietueet

Kontrolli - mistä seuraava käsky?

- valinta: if-then-else
- toisto: for-silmukka, while-silmukka
- aliohjelmat, virhetilanteet

Monimutkaiset tietorakenteet

- listat, moniulotteiset taulukot
- pinot, keot

Copyright Teemu Kerola 2004

Ohjelmoinnissa on oikeastaan vain muutamia peruskäsitteitä. Eri korkean tason kielissä nämä käsitteet toteutetaan ehkä hivenen eri muodossa käyttäen erilaisia syntakseja, mutta perusteiltaan ainakin kaikki lohkorakenteiset kielet ovat hyvin saman kaltaisia. Toisaalta, on olemassa myös luonteeltaan erilaisia ohjelmointikieliä, kuten funktionaaliset kielet ja tekoälykielet, mutta emme käsittele niitä nyt tällä kertaa. Perustyyö kaikessa laskennassa on suorittaa laskutoimituksia, joten tietenkin meidän tulee osata toteuttaa erilaiset aritmeettiset lausekkeet myös konekielellä.

Konekielisen ohjelmoinnin peruskäsitteet

Aritmeettinen lauseke

- miten tehdä laskutoimitukset?

```
int Tbl[20];  
...  
Tbl[3] = 54;
```

Yksinkertaiset tietorakenteet

- yksiulotteiset taulukot, tietueet

```
struct { int x; int y; } S;  
...  
Xcoord = S.x;  
Ycoord = S.y;
```

Kontrolli - mistä seuraava käsky?

- valinta: if-then-else
- toisto: for-silmukka, while-silmukka
- aliohjelmat, virhetilanteet

Monimutkaiset tietorakenteet

- listat, moniulotteiset taulukot
- pinot, keot

Copyright Teemu Kerola 2004

Yksiulotteiset taulukot ja yksinkertaiset tietueet muodostavat rakenteellisen tiedon perusteet, ja kaikki lohkorakenteiset ohjelmointikielet tukevat niiden käyttöä. On siis olennaista ymmärtää, miten nämä rakenteellisen tiedon peruskäsitteet toteutetaan konekielen tasolla.

Konekielisen ohjelmoinnin peruskäsitteet

Aritmeettinen lauseke

- miten tehdä laskutoimitukset?

```
x = 5;  
y = 67;  
z = x;
```

```
if (X < 45)  
    Y = 20;  
else  
    Y = 0;
```

Yksinkertaiset tietorakenteet

- yksiulotteiset taulukot, tietueet

```
while (X > 0) {  
    Y += 1;  
    X /= 2;  
}
```

```
for (i=0; i<10; ++i)  
    Sum += Tbl[i];
```

Kontrolli - mistä seuraava käsky?

- valinta: if-then-else
- toisto: for-silmukka, while-silmukka
- aliohjelmat, virhetilanteet

```
X = MyFunction ( X, 34);  
System.out.println("Hello");
```

Monimutkaiset tietorakenteet

- listat, moniulotteiset taulukot
- pinot, keot

```
Y = 1;  
X = Y-1;  
Z = 45 / X; // oops
```

Copyright Teemu Kerola 2004

Normaalitapauksessa konekielessä nyt suoritettavan konekäskyn jälkeen suoritetaan nykykäskyä seuraavassa muistipaikassa oleva käsky. Tämä oletusarvo pienentää koodia huomattavasti, koska useimmiten seuraavaksi suoritettavan konekäskyn osoitetta ei tarvitse erikseen speksata. On kuitenkin paljon tavalliseen ohjelmointiin liittyviä tilanteita ja myös erikoistilanteita, joissa seuraavaksi suoritettava käsky löytyykin muualta (kuin nykyisen käskyn jälkeen). Käymme läpi kaikki nämä eri tilanteet kontrollin siirtymisessä.

Konekielisen ohjelmoinnin peruskäsitteet

Aritmeettinen lauseke

- miten tehdä laskutoimitukset?

```
int [ ] [ ] matriisi = { { 1, 2 }, { 3, 4 }, { 5, 6 } };  
int [ ][ ][ ] matr3d = new int [ ][ ][ ];
```

...

```
X = matriisi [1] [2];  
X = matr3d [i] [6] [k];
```

Yksinkertaiset tietorakenteet

- yksiulotteiset taulukot, tietueet

```
public class Person {  
    public String name;  
    public int age;  
}
```

```
Person sanna = new Person ("Sanna", 10);
```

```
ageNow = sanna.age;
```

Kontrolli - mistä seuraava käsky?

- valinta: if-then-else
- toisto: for-silmukka, while-silmukka
- aliohjelmat, virhetilanteet

```
public class Stack {  
    int SP; int Size; ...  
    public int push(...);  
    public int pop(...);  
    public int create (...);  
}
```

Monimutkaiset tietorakenteet

- listat, moniulotteiset taulukot
- pinot, keot

Copyright Teemu Kerola 2004

Yksinkertaisten rakenteiden jälkeen saamme hivenen tuntumaa monimutkaisempiin tieto- ja koodirakenteisiin. Esimerkiksi moniulotteinen taulukko on yksiulotteisen taulukon laajennus, mutta sen toteutus on usein selvästi monimutkaisempaa, koska sille ei ole yleensä laitteistotukea sopivan tiedonosoitusmoodin muodossa. Toisaalta, mikä tahansa monimutkainenkin tietorakenne voidaan aina kuvata yksinkertaisiin perustyyppeihin ja toteuttaa näiden avulla.

Aritmeettinen lauseke

lauseke korkean
tason kielellä

```
int a, b, c;
```

```
...
```

```
b = 34;
```

```
a = b + 5 * c;
```

Copyright Teemu Kerola 2004

Aritmeettisessa lausekkeessa voi olla mukana muuttujia tai vakioita. Tässä esimerkissä meillä on kolme muuttujaa (a, b ja c) ja kaksi yksinkertaista sijoituslauseetta. Ensimmäisessä lauseessa muuttujalle b sijoitetaan uusi vakioarvo 34. Toisessa lauseessa muuttujalle a sijoitetaan aritmeettisen lausekkeen arvo, joka tietenkin täytyy laskea ensin. Esimerkissä kaikki muuttujat ja vakiot ovat kokonaislukuarvoisia, mutta lausekkeet lasketaan vastaavasti myös muille tietotyypeille.

Aritmeettinen lauseke

lauseke korkean
tason kielellä

```
int a, b, c;
```

```
...
```

```
b = 34;
```

```
a = b + 5 * c;
```

tilan varaus

A	DC	0
B	DC	0
C	DC	0

Copyright Teemu Kerola 2004

Muuttujat a, b ja c otaksutaan tässä pääohjelmatasolla määritellyksi. Niiden tilanvaraus tehdään tällöin DC-pseudokäskyllä. DC-valekäskyllä allokoidulle datalle pitää aina antaa jokin alkuarvo, ja tässä kukin muuttujista alustetaan nolaksi.

Aritmeettinen lauseke

lauseke korkean
tason kielellä

```
int a, b, c;
```

```
...
```

```
b = 34;
```

```
a = b + 5 * c;
```

tilan varaus

A	DC	0
B	DC	0
C	DC	0

aritmeettisen lausekkeen koodi

```
LOAD  R1, =34  
STORE R1, B
```

```
...
```

```
LOAD  R1, B  
LOAD  R2, C  
MUL   R2, = 5  
ADD   R1, R2  
STORE R1, A
```

Copyright Teemu Kerola 2004

Sijoituslause toteutetaan konekielellä yksinkertaisesti 'pyöräyttämällä' sijoitettava arvo jonkun vapaan rekisterin kautta muistissa olevaan muuttujaan. Vakio '34' talletetaan tässä konekäskyn vakio-osaan, josta se saadaan suoraan käyttöön 'välitön operandi' tiedonosoitusmoodin avulla. Muuttujaan b viitataan tavallisella suoralla muistiosoituksella.

Aritmeettinen lauseke

lauseke korkean
tason kielellä

```
int a, b, c;
```

...

```
b = 34;
```

```
a = b + 5 * c;
```

tilan varaus

A	DC	0
B	DC	0
C	DC	0

aritmeettisen lausekkeen koodi

```
LOAD  R1, =34  
STORE R1, B
```

```
LOAD  R1, B  
LOAD  R2, C  
MUL   R2, =5  
ADD   R1, R2  
STORE R1, A
```

Copyright Teemu Kerola 2004

Aritmeettinen lauseke pitää ensin tietenkin laskea käytössä olevia semanttisia sääntöjä noudattaen. Esimerkiksi, useimmissa ohjelmointikielissä tämän lausekkeen kertolasku tulee suorittaa ennen yhteenlaskua. Yksinkertainen kääntäjä tuo ensin muuttujan B arvon rekisteriin R1 ja sitten laskee kertolaskun '5*C' rekisteriin R2, ja lopulta laskee summan rekisteriin R1 ja vie sen muuttujan A arvoksi. Toiminta on hyvin suoraviivaista eikä loppujen lopuksi ole juurikaan vaikeampaa kirjoittaa konekielellä kuin korkean tason kielellä. Eniten aikaahan on kuitenkin mennyt sen ajattelemiseen, että pitää laskea lausekkeen 'B+5C' arvo.

Aritmeettinen lauseke

lauseke korkean
tason kielellä

```
int a, b, c;
```

...

```
b = 34;
```

```
a = b + 5 * c;
```

tilan varaus

A	DC	0
B	DC	0
C	DC	0

aritmeettisen lausekkeen koodi

```
LOAD  R1, =34  
STORE R1, B
```

...

```
LOAD  R1, B  
LOAD  R2, C  
MUL   R2, =5  
ADD   R1, R2  
STORE R1, A
```

Optimointikriteereitä

käännökseen kuluva aika

koodin suoritus aika

konekäskyjen lukumäärä

muistiviitteiden lukumäärä

käytettyjen rekistereiden lukumäärä

optimoitu koodi?

```
LOAD  R1, =5  
MUL   R1, C  
ADD   R1, B  
STORE R1, A
```

Copyright Teemu Kerola 2004

Sama aritmeettinen lauseke voidaan laskea konekielellä usealla eri tavalla. Mikä tapa on paras? Useimmat opiskelijoiden ohjelmat käännetään monta kertaa, kunnes ne lopulta toimivat oikein ja sitten suoritetaan yhden kerran. Tässä tapauksessa olisi ehkä parasta käyttää kääntäjän optioita, joilla käännös tapahtuu nopeasti, vaikka itse generoitu koodi olisikin ehkä hidas suorittaa. Teollisuuden sovelluksissa tilanne on usein päin vastainen. Sama ohjelma voidaan suorittaa tuhansia kertoja päivässä, joten on tärkeitä, että sen suoritus aika olisi mahdollisimman lyhyt. Suoritusajan optimointi ei ole helppoa, mutta suoritettujen käskyjen tai muistiviitteiden lukumäärä on hyvä indikaattori.

Globaalin taulukon tilan varaus ja käyttö

(yksittäisiä lauseita,
ei järkevää koodia!)
korkean tason
kielellä

```
int X, Y;  
int Taulu[30];  
...  
X = 5;  
Y = Taulu[X];  
Y = Taulu[3];  
Taulu[X] = Y;
```

Copyright Teemu Kerola 2004

Kaikkialla käytössä olevat yksiulotteiset taulukot kuuluvat ohjelmoijan peruskalustoon. Tässä esimerkissä meillä on kaksi (kokonaislukuarvoista) muuttujaa (X ja Y) ja yksi (kokonaislukuarvoinen) taulukko Taulu, jossa on 30 alkia. Alkiot on oletusarvoisesti numeroitu nollasta alkaen: Taulu[0], ..., Taulu[29]. Taulukon normaalikäyttöön kuuluu sen indeksointi vakiolla tai muuttujan arvolla sekä taulukon alkioden luku ja kirjoitus.

Globaalin taulukon tilan varaus ja käyttö

(yksittäisiä lauseita,
ei järkevää koodia!)
korkean tason
kielellä

```
int X, Y;  
int Taulu[30];  
...  
X = 5;  
Y = Taulu[X];  
Y = Taulu[3];  
Taulu[X] = Y;
```

tilan varaus

X	DC	0
Y	DC	0
Taulu	DS	30

```
LOAD R1, =0 ; initialize Taulu to zeroes  
LOAD R2, =0  
INIT STORE R1, Taulu[R2]  
ADD R2, =1  
COMP R2, =30  
JLES INIT
```

Copyright Teemu Kerola 2004

Kaikkialla käytössä olevan taulukon tilanvaraus tehdään yksinkertaisesti DS-valekäskyllä. Symbolisessa konekielessä ei ole mitään konstia kertoa kääntäjälle taulukon alkuarvoista, joten oletusarvoisesti ne ovat määrittelemättömiä. Kääntäjä voi kuitenkin alustaa ne esimerkiksi nolliksi, mutta tähän ei tulisi luottaa. Kaikki taulukot pitää siis muistaa alustaa ohjelmakoodissa.

Globaalin taulukon tilan varaus ja käyttö

(yksittäisiä lauseita,
ei järkevää koodia!)
korkean tason
kielellä

```
int X, Y;  
int Taulu[30];  
...  
X = 5;  
Y = Taulu[X];  
Y = Taulu[3];  
Taulu[X] = Y;
```

tilan varaus

X	DC	0
Y	DC	0
Taulu	DS	30

koodi

```
LOAD R1, =5  
STORE R1, X  
  
LOAD R1, X  
LOAD R2, Taulu(R1)  
  
LOAD R1, =3  
LOAD R2, Taulu(R1)  
  
LOAD R2, Y  
LOAD R1, X  
STORE R2, Taulu(R1)
```

Copyright Teemu Kerola 2004

Taulukkoon viittaminen tapahtuu kahdessa vaiheessa. Ensimmäisessä vaiheessa johonkin indeksirekisteriin sijoitetaan taulukon indeksi, ja toisessa vaiheessa tehdään itse taulukon muistiviite tuon indeksirekisterin avulla indeksoitua muistinosoitusblokkia käyttäen. Taulukot sijaitsevat aina muistissa ja niistä tuodaan käsittelyyn työrekestereihin aina vain yksi alkio kerrallaan. Taulukon alkion arvojen talletus muistiin tapahtuu vastaavasti yksi alkio kerrallaan. Uusi arvo lasketaan ensin työrekesteriin ja alkion indeksi indeksirekisteriin, jonka jälkeen arvo talletetaan muistiin indeksoitua tiedonosoitusblokkia käyttäen. Taulukoiden käsittelyn voi tehdä myös osoitinmuuttujien avulla, mutta siitä sitten myöhemmin lisää.

Globaalin taulukon tilan varaus ja käyttö

(yksittäisiä lauseita,
ei järkevää koodia!)
korkean tason
kielellä

```
int X, Y;  
int Taulu[30];  
...  
X = 5;  
Y = Taulu[X];  
Y = Taulu[3];  
Taulu[X] = Y;
```

tilan varaus

X	DC	0
Y	DC	0
Taulu	DS	30

koodi

```
LOAD R1, =5  
STORE R1, X  
  
LOAD R1, X  
LOAD R2, Taulu(R1)  
  
LOAD R1, =3  
LOAD R2, Taulu(R1)  
  
LOAD R2, Y  
LOAD R1, X  
STORE R2, Taulu(R1)
```

optimoiva kääntäjä
osaasi jättää tämän
käsken pois.

Copyright Teemu Kerola 2004

Tässäkin esimerkissä näkyy hyvin, kuinka yksinkertainen kääntäjä kääntää korkean tason kielen lauseet yksi kerrallaan konekieleksi. Optimoiva kääntäjä osaasi ottaa huomioon myös muista korkean tason kielen lauseista generoidut käskyt. Vaikka tämän yhden käsken jättäminen pois voi tuntua hyvin suoraviivaiselta ja itsestään selvältä asialta, niin käytännössä kääntäjän optimoivaa koodingenerointilogiikkaa suunniteltaessa havaitsee pian, että yleisessä tapauksessa tämänkin asian huomaaminen ja ohjelmoiminen on aika monimutkaista.

Globaalin tietueen tilanvaraus ja käyttö

korkean tason
kielellä

```
int X, Y;  
struct Tauno {  
    int Pituus;  
    int Paino;  
}  
  
..  
X = Tauno.Paino;  
Tauno.Pituus = 175;
```

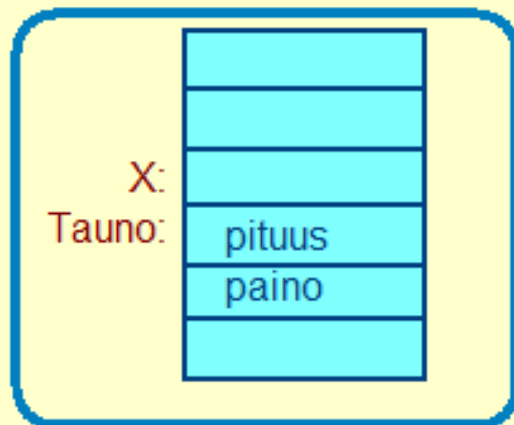
Copyright Teemu Kerola 2004

Tietueet ovat toinen ohjelmoijan perustietorakenne. Toisin kuin taulukossa, tietueen sisällä sen alkiot eli kentät voivat olla keskenään erilaisia, vaikka ne meidän ttk-91 esimerkeissä ovatkin aina samaa int-tyyppiä. Kaikilla samaa tyyppiä olevilla tietueilla on siis sama rakenne ja samat kentät. Tässä esimerkissä meillä on globaali tietue Tauno, jolla on kaksi kenttää: Pituus ja Paino. Kenttiin pystytään viittaamaan korkean tason kielessä niiden nimen perusteella.

Globaalin tietueen tilanvaraus ja käyttö

korkean tason
kielellä

```
int X, Y;  
struct Tauno {  
    int Pituus;  
    int Paino;  
}  
..  
X = Tauno.Paino;  
Tauno.Pituus = 175;
```



Tietueen Tauno kentän Pituus
suhteellinen osoite tietueessa Tauno = 0

Tietueen Tauno kentän Pituus
osoite = symbolin Tauno arvo (+ 0)

Tietueen Tauno kentän Paino
suhteellinen osoite tietueessa Tauno = 1

Tietueen Tauno kentän Paino
osoite = symbolin Tauno arvo + 1

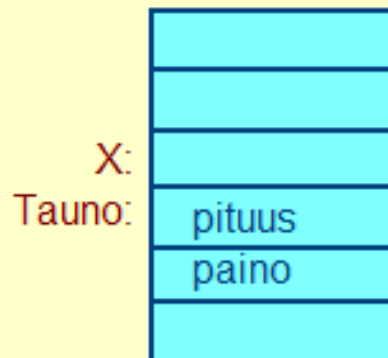
Copyright Teemu Kerola 2004

Tietue on monisanainen tieto ja se talletetaan muistiin samalla tavalla kuin muukin monisanainen tieto: peräkkäisiin muistipaikkoihin. Kääntäjä valitsee kenttien sisäisen järjestyksen tietueen sisällä, mutta järjestyksellä ei juurikaan ole väliä. Tässä esimerkissä tietueen Tauno kentän Pituus suhteellinen osoite tietueen sisällä on 0 ja kentän Paino suhteellinen osoite on 1. Tietueen osoite eli tässä symbolin Tauno arvo on kentän Pituus osoite, joka on siis aivan eri asia kuin kentän Pituus suhteellinen osoite tietueen sisällä.

Globaalin tietueen tilanvaraus ja käyttö

korkean tason
kielellä

```
int X, Y;  
struct Tauno {  
    int Pituus;  
    int Paino;  
}  
..  
X = Tauno.Paino;  
Tauno.Pituus = 175;
```



tilan varaus

X	DC	0
Tauno	DS	2
Pituus	EQU	0
Paino	EQU	1

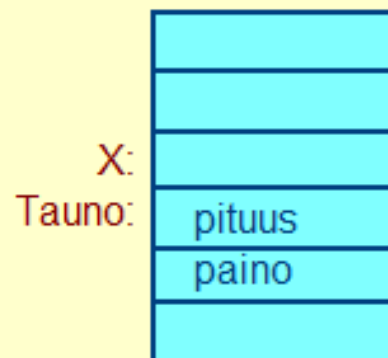
Copyright Teemu Kerola 2004

Symbolisen konekielen tasolla globaalille, kaikkialla käytössä olevalle tietueelle Tauno varataan tilaa DS-valekäskyllä ja symboleille Pituus ja Paino annetaan arvoksi tietueen kenttien suhteelliset sijainnit. Jos samantyyppisiä tietueita varattaisiin suoritusajana keosta, niin myös niiden pituus- ja painokenttien suhteelliset sijainnit tietueen sisällä olisivat samat kuin tässä määritellyt. Oikeasti siis kullekin tietuetyypille määritellään kunkin kentän suhteelliset sijainnit, jotka ovat samoja kaikille kyseisen tyyppin tietueille.

Globaalin tietueen tilanvaraus ja käyttö

korkean tason
kielellä

```
int X, Y;  
struct Tauno {  
    int Pituus;  
    int Paino;  
}  
  
..  
X = Tauno.Paino;  
Tauno.Pituus = 175;
```



tilan varaus

X	DC	0
Tauno	DS	2
Pituus	EQU	0
Paino	EQU	1

koodi

```
LOAD R1, =Tauno  
LOAD R2, Paino(R1)  
STORE R2, X  
  
LOAD R2, =175  
STORE R2, Pituus(R1)
```

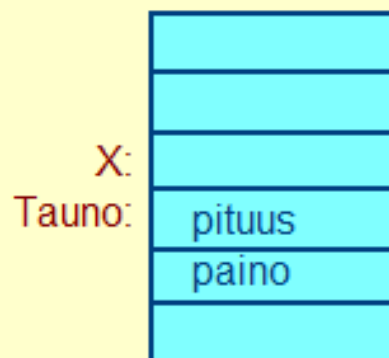
Copyright Teemu Kerola 2004

Tietueihin viitataan samalla indeksoidulla tiedonosoitusmoodilla kuin taulukoihinkin, mutta nyt moodia käytetään eri tavalla. Kun taulukoiden yhteydessä taulukon osoite oli käskyn osoiteosassa ja taulukon sisäinen osoite eli indeksi oli indeksirekisterissä, niin tietueiden kohdalla tilanne on päin vastoin. Tietueen osoite ladataan aina ensin johonkin indeksirekisteriin ja tietueen sisäinen osoite eli tietueen kentän suhteellinen osoite annetaan käskyn vakio-osassa. Esimerkissä rekisteri R1 osoittaa siis kyseessä olevaan tietueeseen ja sen kenttiin viitataan kenttien suhteellisten osoitteiden Pituus ja Paino avulla, indeksoiden niitä tietueen osoitteella.

Globaalin tietueen tilanvaraus ja käyttö

korkean tason
kielellä

```
int X, Y;  
struct Tauno {  
    int Pituus;  
    int Paino;  
}  
..  
X = Tauno.Paino;  
Tauno.Pituus = 175;
```



tilan varaus

X	DC	0
Tauno	DS	2
Pituus	EQU	0
Paino	EQU	1

```
LOAD  R1, =Tauno  
LOAD  R2, Paino(R1)  
STORE R2, X
```

```
LOAD  R2, =175  
STORE R2, Pituus(R1)
```

Copyright Teemu Kerola 2004

On merkillepantavaa, että yhdellä samalla indeksoidulla tiedonosoitusmoodilla voidaan toteuttaa molemmat tärkeimmät rakenteisen tiedon viittaustavat: taulukot ja tietueet. Taulukoissa tietorakenteen osoite on käskyssä vakiona ja taulukon sisäinen osoite eli usein muuttuva indeksi on indeksirekisterissä. Tietueisiin viitattaessa tietueen osoite on osoitinmuuttujan arvona rekisterissä ja tietueen sisäinen kentän osoite on käskyn vakio-osassa. Tietyissä konekäskyissä viitataan aina samaan kenttään, vaikka itse tietue vaihtuisikin. Tämä on tietenkin vain perustapa viitata sekä taulukoihin että tietueisiin - monenlaisia muita variantteja on myös käytössä.

Kontrollin siirto muualle kuin seuraavaan käskyyn

Ehdoton kontrollin siirto

- JUMP, CALL, EXIT, SVC, IRET

JUMP	LOOP
CALL	SP, Subr
EXIT	SP, =2
SVC	SP, ReadFile
IRET	SP, =1

Ehdollinen kontrollin siirto perustuen työrekisterin arvoon

- JZER, JNZER, JPOS, JNPOS, JNEG, JNNEG

Ehdollinen kontrollin siirto perustuen tilarekisterin vertailubittien arvoon

- JEQU, JNEQU, JGRE, JNGRE, JLES, JNLES
- aseta arvo ensin COMP-käskyllä (normaali tilanne)
- aseta arvo ALU-operaatioiden yhteydessä (ttk-91 "piirre"), jokaista ALU-käskyä "seuraa" implisiittinen vertailu "COMP Rj, =0"

ADD, SUB, MUL,
DIV, NOT, AND,
OR, XOR, SHL,
SHR, SHRA

ADD R1, =5

$R1 \leftarrow R1 + 5$

$G \leftarrow 1$, jos ja vain jos $R1$ 'n uusi arvo > 0

$E \leftarrow 1$, jos ja vain jos $R1$ 'n uusi arvo $= 0$

$L \leftarrow 1$, jos ja vain jos $R1$ 'n uusi arvo < 0

Copyright Teemu Kerola 2004

Kontrollin siirrolla tarkoitetaan sitä, mistä seuraava konekäsky löytyy kun nykyinen käsky on suoritettu loppuun. Oletusarvoisestihan se löytyy aina nykykäskyä seuraavasta muistipaikasta. Mielenkiintoiset tilanteet tulevat tietenkin vain siitä, kun kontrolli siirtyy jonnekin muualle. Yksinkertaisin tapa on ehdoton kontrollin siirto muualle, vaikka siirtymän kohde vaihtelisikin kuten esimerkiksi aliohjelmasta paluu -käskyn EXIT yhteydessä. Kontrolli kyllä palaa aina kutsukohtaa seuraavaan konekäskyyn, mutta aliohjelman kutsukohtiahan saattoi olla hyvinkin monta eri puolella ohjelmaa.

Kontrollin siirto muualle kuin seuraavaan käskyyn

Ehdoton kontrollin siirto

- JUMP, CALL, EXIT, SVC, IRET

Ehdollinen kontrollin siirto perustuen työrekisterin arvoon

- JZER, JNZER, JPOS, JNPOS, JNEG, JNNEG

JZER R1, Loop
JNEG R3, Else

Ehdollinen kontrollin siirto perustuen tilarekisterin vertailubittien arvoon

- JEQU, JNEQU, JGRE, JNGRE, JLES, JNLES
- aseta arvo ensin COMP-käskyllä (normaali tilanne)
- aseta arvo ALU-operaatioiden yhteydessä (ttk-91 "piirre"),
jokaista ALU-käskyä "seuraa" implisiittinen vertailu "COMP Rj, =0"

ADD, SUB, MUL,
DIV, NOT, AND,
OR, XOR, SHL,
SHR, SHRA

ADD R1, =5

$R1 \leftarrow R1 + 5$

$G \leftarrow 1$, jos ja vain jos $R1$ 'n uusi arvo > 0
 $E \leftarrow 1$, jos ja vain jos $R1$ 'n uusi arvo $= 0$
 $L \leftarrow 1$, jos ja vain jos $R1$ 'n uusi arvo < 0

Copyright Teemu Kerola 2004

Ehdollinen kontrollin siirtymä eli ehdollinen hyppykäsky voidaan toteuttaa kahdella tavalla. Useissa tilanteissa päätös tehdään vertailemalla jonkin työrekisterin arvoa nolnaan. Sitä varten on ihan omat konekäskynsä.

Kontrollin siirto muualle kuin seuraavaan käskyyn

```
COMP R3, R4  
JNLES Else
```

↗ vertaa?

Ehdoton kontrollin siirto

- JUMP, CALL, EXIT, SVC, IRET

```
COMP R3, R4  
MUL R2, R2 ; tulos >= 0  
JNLES Else ; hyppää aina!
```

Ehdollinen kontrollin siirto perustuen työrekisterin arvoon

- JZER, JNZER, JPOS, JNPOS, JNEG, JNNEG

```
COMP R0, =20  
JGRE Loop
```

Ehdollinen kontrollin siirto perustuen tilarekisterin vertailubittien arvoon

- JEQU, JNEQU, JGRE, JNGRE, JLES, JNLES
- aseta arvo ensin COMP-käskyllä (normaali tilanne)
- aseta arvo ALU-operaatioiden yhteydessä (ttk-91 "piirre"), jokaista ALU-käskyä "seuraa" implisiittinen vertailu "COMP Rj, =0"

```
ADD, SUB, MUL,  
DIV, NOT, AND,  
OR, XOR, SHL,  
SHR, SHRA
```

```
ADD R1, =5
```

```
R1 ← R1+5
```

```
G ← 1, jos ja vain jos R1'n uusi arvo > 0
```

```
E ← 1, jos ja vain jos R1'n uusi arvo = 0
```

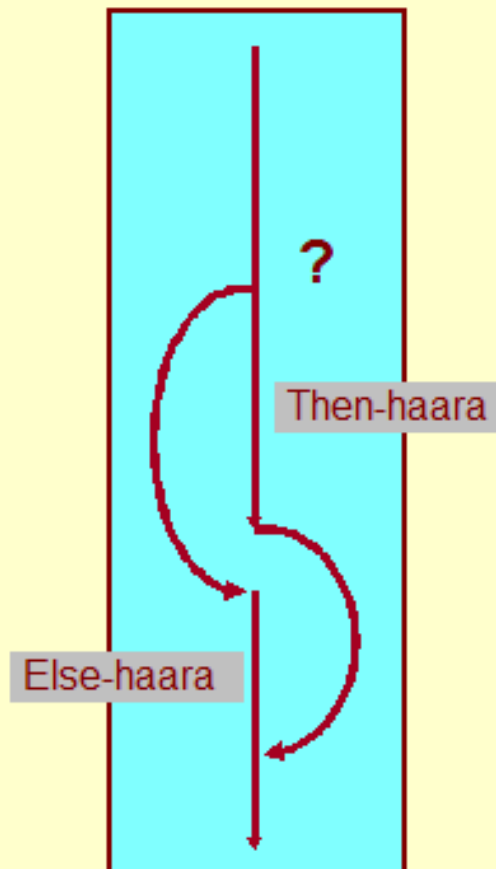
```
L ← 1, jos ja vain jos R1'n uusi arvo < 0
```

Copyright Teemu Kerola 2004

Yleisempi vertailuun perustuva kontrollin siirto tehdään kahdessa vaiheessa. Ensin tehdään vertailu kahden arvon välillä ja tulos talletetaan tilarekisteriin. Seuraavassa vaiheessa sitten tarkastellaan tilarekisterissä olevaa vertailun tulosta ja haaraudutaan sen mukaisesti. Tilannetta vähän monimutkaistaa ttk-91 koneen 'piirre', jossa jokainen ALU-operaatio suorittaa oletusarvoisesti tuloksen vertailun nolleen ja asettaa tilarekisterin vertailubitit vastaavasti. Tämähän on ihan turhaa, koska meillä on erikseen yllämainitut konekäskyt juuri tätä tilannetta varten.

If-then-else -valinta

```
if (a < b)  
    x = 5;  
else  
    x = y;
```

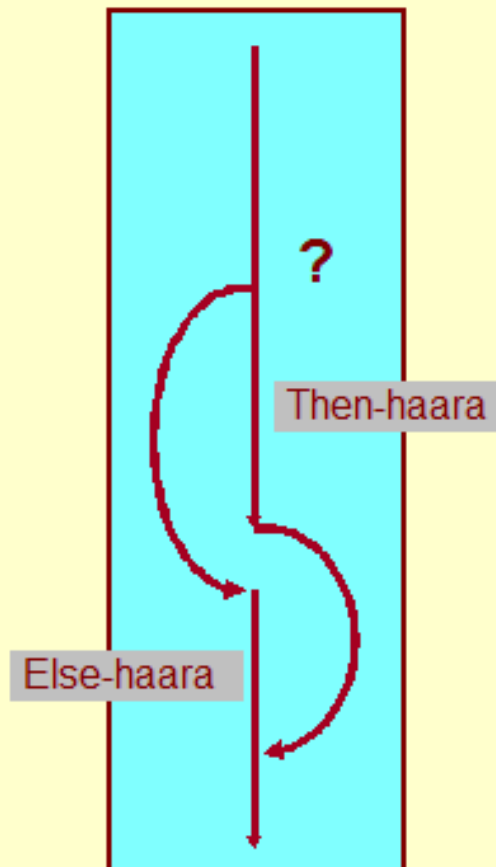


Copyright Teemu Kerola 2004

Normaali if-then-else -rakenne on yleisin korkean tason kielten kontrollirakenne. Koodia suoritetaan tässä 'ylhäältä alaspäin', jolloin jossain vaiheessa tehdään vertailu. Jos vertailun tulos on epätosi, haaraudutaan else-haaraan. Muutoin suoritetaan heti vertailun jälkeen oleva then-haara, minkä jälkeen suoritus jatkuu else-haaran jälkeen. Tästä on tietenkin myös yksinkertaisempi muoto, jossa else haara puuttuu.

If-then-else -valinta

```
if (a<b)  
  x = 5;  
else  
  x = y;
```



```
LOAD R1, A  
COMP R1, B  
JNLES Else  
  
Then LOAD R1, =5  
      STORE R1, X  
      JUMP Done  
  
Else  LOAD R1, Y  
      STORE R1, X  
  
Done NOP
```

Copyright Teemu Kerola 2004

Korkean tason kielen koodi käännetään suoraviivaisesti, yksi lause kerrallaan. Vertailemme ensin muuttujan A arvoa muuttujan B arvoon ja haaraudumme sitten else-haaraan, jos A ei ole pienempi kuin B. Muutoin jatkamme koodin suoritusta then-haarasta, jossa on osoitteena symboli 'Then' ihan selvyiden vuoksi. Koodissahan ei viitata tuohon symboliin lainkaan. Lopulta suoritus jatkuu aina Done-symbolin kohdalta, joko hyppäämällä sinne then-haaran lopusta tai vain jatkamalla suoritusta else-haaran jälkeen. Tämä on hyvä esimerkki NOP-käskyn käytöstä. Vaikka NOP-käsky ei teekään mitään, sille voidaan antaa osoite!

If-then-else -valinta

```
if (a<b)
  x = 5;
else
  x = y;
```

Else-haara

Then-haara

koodi: 9 sanaa

suoritus:
then-haara: 7 käskyä
else-hara: 6 käskyä

```
LOAD R1, A
COMP R1, B
JNLES Else
```

Then LOAD R1, =5
STORE R1, X
JUMP Done

Else LOAD R1, Y
STORE R1, X

Done NOP

koodi: 6 sanaa

suoritus:
then-haara: 6 käskyä
else-hara: 5 käskyä

```
LOAD R2, Y
LOAD R1, A
COMP R1, B
JNLES Else
```

Then LOAD R2, =5

Else STORE R2, X

Copyright Teemu Kerola 2004

Optimoiva kääntäjä osaisi kääntää tämän ehdollisen lauseen siten, että siitä generoituisi vähemmän konekäskyjä ja että koodi luultavasti suoritaisi nopeammin kuin ilman optimointia generoitu koodi. Optimoitu koodi on kuitenkin ihmisen vaikeampaa lukea ja ymmärtää. Tässä on hyödynnetty sitä asiaa, että molemmissa kontrollihaaroissa talletetaan samaan muuttujaan X. Koodissa arvataan ensin, että X'ään tullaan tallettamaan Y'n arvo. Talletettava arvo R2:ssa muutetaan then-haarassa 5'ksi, jos arvaus oli väärin. Koodissa on säästetty kokonaan then-haaran lopussa oleva hyppykäsky.

Monivalintalaus (case, switch)

```
switch (lkm) {  
    case 4: x = 11;  
           break;  
    case 0: break;  
    default: x = 0;  
           break;  
}
```

Copyright Teemu Kerola 2004

If-then-else -lauseen laajennus on monivalinta-lause eli case-lause eli switch-lause. Eri korkean tason ohjelmointikielissä se voi olla vähän eri muodossa, mutta perusidea on silti sama. Meillä on muuttuja tai lauseke, ja sen arvosta riippuen kontrolli siirtyy oikeaan case-tapaukseen. Case'n jälkeen suoritus jatkuu koko case-lauseen jälkeen. C- ja Java-kielissä tämä on switch-lause ja siirtymä switch-lauseen loppuun pitää eksplisiittisesti muistaa laittaa break-lauseella jokaiseen case'en. Joissakin toisissa ohjelmointikielissä (esim. Pascalissa) tätä siirtymää switch-lauseen loppuun ei tarvitse eksplisiittisesti mainita, vaan se tapahtuu automaattisesti.

Monivalintalause (case, switch)

```
switch (lkm) {  
    case 4: x = 11;  
           break;  
    case 0: break;  
    default: x = 0;  
           break;  
}
```

```
Switch  LOAD R1, Lkm  
  
Vrt4    COMP R1, =4  
        JNEQU Vrt0  
        LOAD R2, =11  
        STORE R2, X  
        JUMP  Cont  
  
Vrt0    COMP R1, =0  
        JNEQU Def  
        JUMP  Cont  
  
Def     LOAD R1, =0  
        STORE R1, X  
  
Cont    NOP
```

Copyright Teemu Kerola 2004

Case-lause koodataan konekielellä suoraviivaisesti yksi case-kohta kerrallaan, tyyliin if-then-elseif-then-elseif-then jne. Kussakin case'ssa ensin tutkitaan, halutaanko juuri tämä case suorittaa. Jos ei, niin siirrytään seuraavaan case'en. Jos kohta on oikea, niin suoritetaan tämän case-kohdan koodi ja sitten jatketaan suoritusta case-lauseen jälkeen. Case-lauseessa voi useiden ohjelmointikielten yhteydessä olla oletusarvoinen kohta, joka suoritetaan siinä tapauksessa, että mikään nimetty case ei 'täppää'.

Monivalintalause (case, switch)

```
switch (lkm) {  
    case 4: x = 11;  
           break;  
    case 0: break;  
    default: x = 0;  
           break;  
}
```

Onko Case-tapausten järjestyksellä väliä?

1000 casea, oikea löytyy keskimäärin 500
vai 10 yrityksellä?

Switch LOAD R1, Lkm

Vrt4 COMP R1, =4
 JNEQU Vrt0
 LOAD R2, =11
 STORE R2, X
 JUMP Cont

Vrt0 COMP R1, =0
 JNEQU Def
 JUMP Cont

Def LOAD R1, =0
 STORE R1, X

Cont NOP

Copyright Teemu Kerola 2004

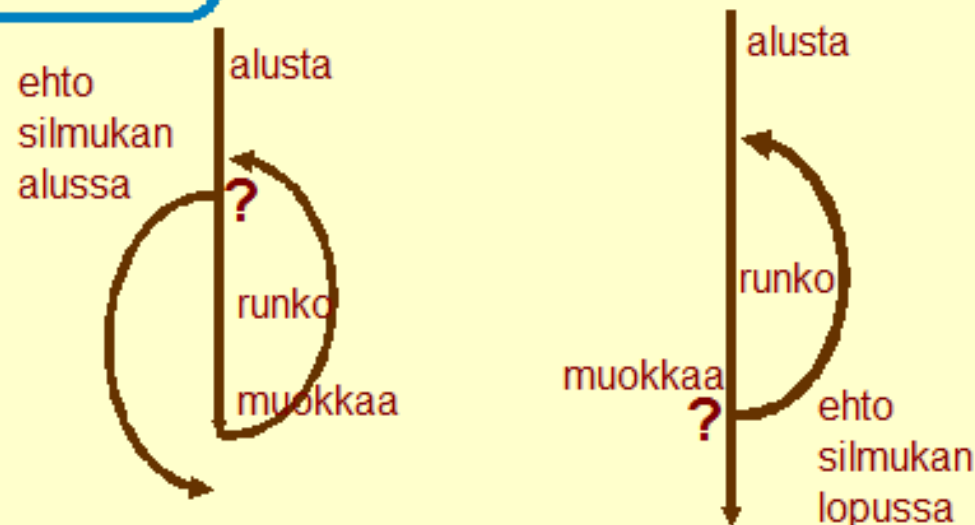
Case-lause on siis oikeasti oikein pitkä if-then-elseif-then-elseif-then-else lause, jossa suoritetaan ehkä suurikin määrä vertailuja, kunnes oikea case-tapaus löytyy. Viisas ohjelmoija sijoittaaakin case-tapauksensa sellaiseen järjestykseen, että oikea case löytyy keskimäärin mahdollisimman vähillä vertailuilla. Case-tapauksiahan voi tilanteesta riippuen olla satoja tai jopa tuhansia. Joissakin tapauksissa optimoiva kääntäjä voi auttaa tässä. Joissakin tapauksissa apuna voi käyttää myös ns. hyppytaulua (jump table), jonka avulla oikea case löytyykin taulukon avulla eikä vertailujen perusteella.

Toistolause (for-lause)

For-step-until -silmukka
Do-until -silmukka
Do-while -silmukka
While-do -silmukka
...

```
for (i=0; i<10; i++) { ... };  
while (flag) { .. };  
do { ... } until X > 2345;
```

```
while (not done) do  
begin .... end;
```



Copyright Teemu Kerola 2004

Korkean tason kielissä on paljon erilaisia toistolauseita, esimerkiksi for-lauseita, do-until -lauseita ja while-do -lauseita. Lauseiden kielioppi eli syntaksi ja merkitys eli semantiikka vaihtelevat hieman eri kielissä, mutta ei paljon. Esimerkiksi, joissakin kielissä for-lauseen yhteydessä koodirunko suoritetaan aina vähintään yhden kerran, mutta yleensä näin ei ole. Joissakin kielissä toistolauseen sisällä voi määritellä paikallisia muuttujia, mutta me emme käsittele tätä ohjelmointikielten piirrettä lainkaan. Otaksumme jatkossa, että kyse on ainoastaan kontrollin toteuttamisesta.

Toistolause (for-lause)

For-step-until -silmukka

Do-until -silmukka

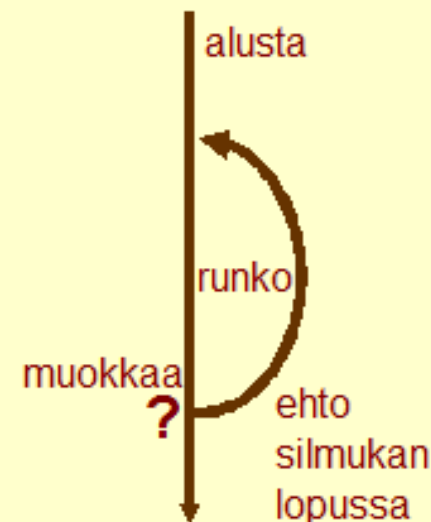
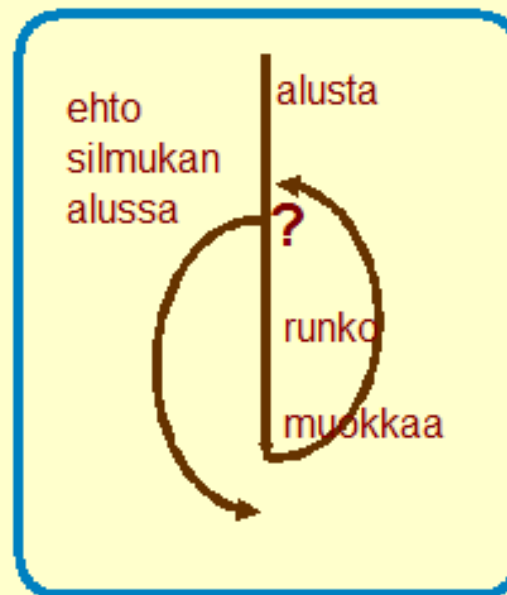
Do-while -silmukka

While-do -silmukka

...

```
for (i=0; i<200; i++) {  
    ...  
};
```

```
i = 0;  
while (i<200) {  
    ...  
    i++  
};
```



Copyright Teemu Kerola 2004

Konekielen tasolla nämä kaikki erilaiset toistolauseet redusoituvat kahteen perustyyppiin. Ensimmäisessä tyyppissä lauseessa tehdään alustuksia ja sen jälkeen on ehtolause. Jos ehto pitää paikkansa, niin suoritetaan toistolauseen runko, modifioidaan jotain toistolauseen ehtomuuttujia ja palataan takaisin ehtolauseeseen. Jos ehto ei pidä paikkansa, niin poistutaan toistolauseesta. Tässä tyyppissä on varsin mahdollista, että toistolauseen runkoa ei suoriteta edes yhtä kertaa. Tämä sopii esimerkiksi C'n tai Javan for- tai while-semantiikkaan.

Toistolause (for-lause)

For-step-until -silmukka

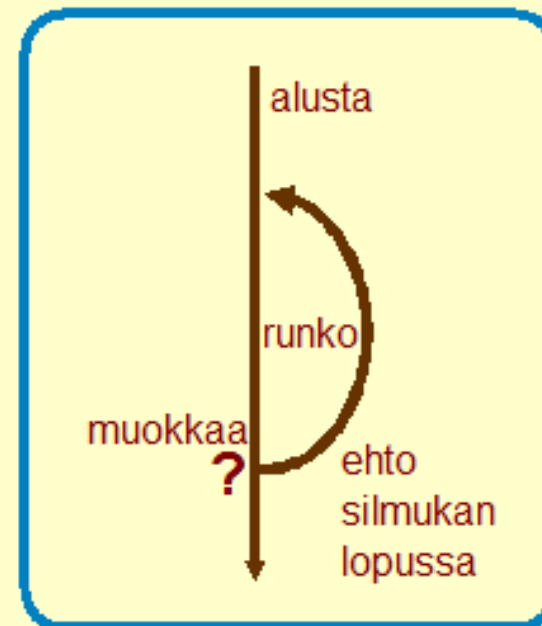
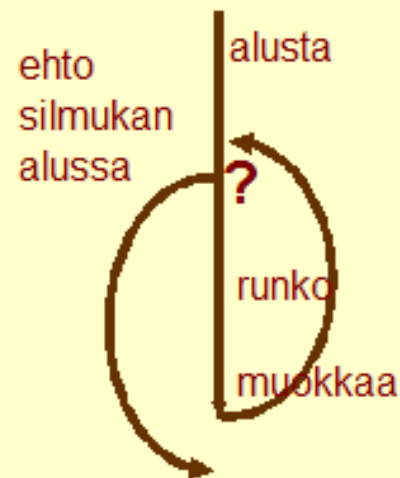
Do-until -silmukka

Do-while -silmukka

While-do -silmukka

...

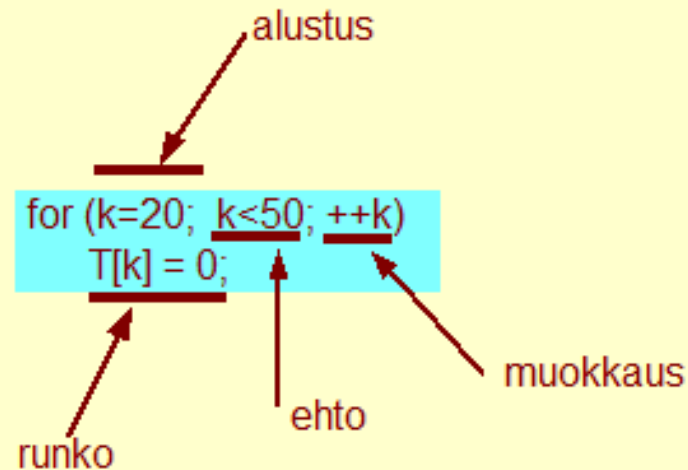
```
i = 0;  
do {  
    ...  
    i++  
} while i<200;
```



Copyright Teemu Kerola 2004

Toisessa toistolauseen tyypissä silmukan suoritusehto on silmukan lopussa. Tässä tapauksessa silmukan runko suoritetaan aina vähintään yhden kerran. Tämä sopii esimerkiksi joidenkin Fortran-kielten for-semantiikkaan ja Javan tai C'n do-while -semantiikkaan.

For-step-until ja muut vastaavat toistolauseet



Copyright Teemu Kerola 2004

For-lause voidaan kääntää konekielelle paloittain. Toistolauseet koostuvat kaikki neljästä osasta: toistomuuttujien alustuksesta, ehtolausekkeesta, varsinaisesta työn tekevästä toistolauseen rungosta ja toistomuuttujien arvojen muokkaamisesta joka iteraatiokerralla. Toteutamme nämä kaikki osat siis yksi kerrallaan.

For-step-until ja muut vastaavat toistolauseet

```
for (k=20; k<50; ++k)  
    T[k] = 0;
```

K DC 0

LOAD R1, =20
STORE R1, K

Copyright Teemu Kerola 2004

Tässä esimerkissä alustus toteutetaan ensin asettamalla toistomuuttujalle K alkuarvo 20.

For-step-until ja muut vastaavat toistolauseet



```
for (k=20; k<50; ++k)  
  T[k] = 0;
```

```
K    DC    0
```

```
...  
LOAD R1, =20  
STORE R1, K
```

```
Loop LOAD R2, =0  
      LOAD R1, K  
      STORE R2, T(R1)
```

Copyright Teemu Kerola 2004

Toistolauseen runko toteutetaan seuraavaksi. Työrekisteriin R2 ladataan alkuarvo nolla ja indeksirekisteriin R1 indeksimuuttuja K. Lopuksi taulukon T alkioon K talletetaan uusi alkuarvo 0. Huomaa, että tässä toteutetaan for-loopin Fortran-tyyppistä semantiikkaa, jossa loopin runko suoritetaan aina ainakin yhden kerran. Jos kyseessä olisi esimerkiksi C-ohjelma, niin ehto pitäisi tarkistaa ennen rungon ensimmäistä suorituskertaa.

For-step-until ja muut vastaavat toistolauseet



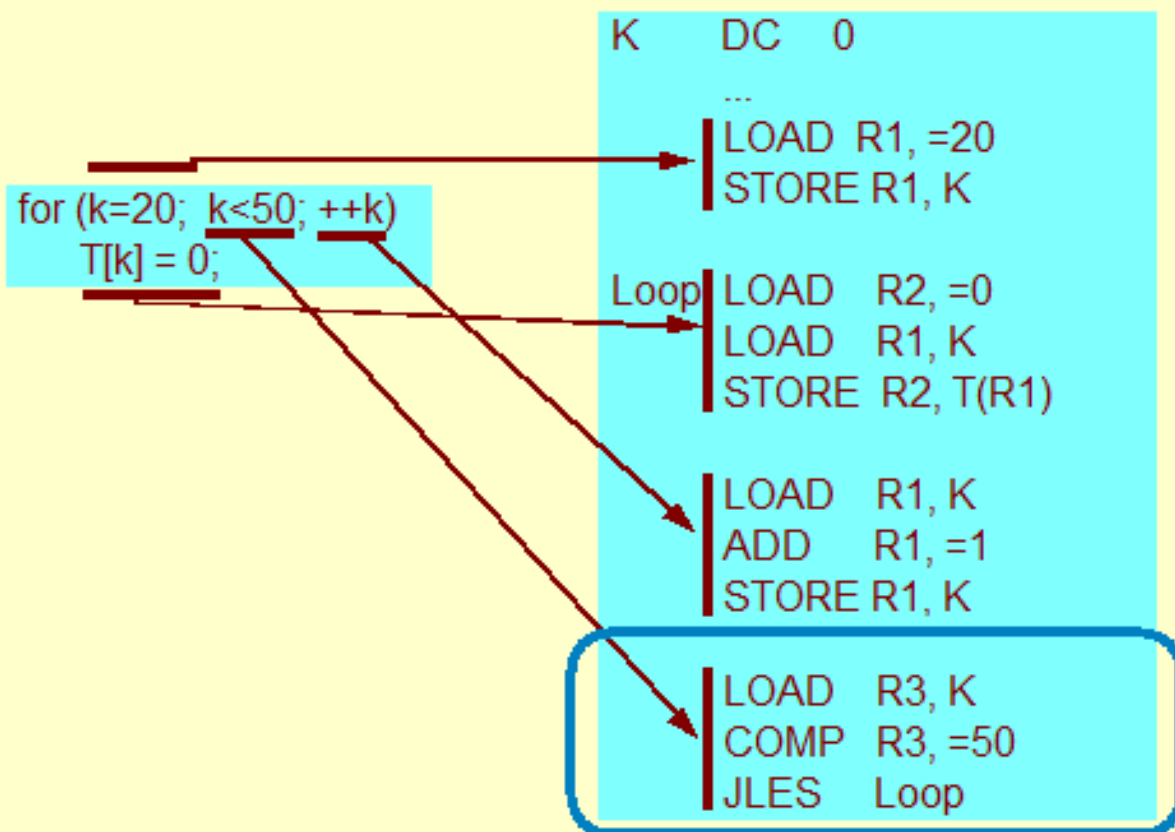
for (k=20; k<50; ++k)
T[k] = 0;

K DC 0
...
LOAD R1, =20
STORE R1, K
Loop LOAD R2, =0
LOAD R1, K
STORE R2, T(R1)
LOAD R1, K
ADD R1, =1
STORE R1, K

Copyright Teemu Kerola 2004

Seuraavaksi muokataan loopin toistomuuttujaa K kasvattamalla sen arvoa yhdellä.

For-step-until ja muut vastaavat toistolauseet



Copyright Teemu Kerola 2004

Lopuksi tehdään ehdon tarkistus ja hypätään toistolauseen rungon alkuun, jos K'n arvo on edelleen pienempi kuin 50.

For-step-until ja muut vastaavat toistolauseet



```
for (k=20; k<50; ++k)
    T[k] = 0;
```

Olisiko parempi pitää k:n arvo rekisterissä?
Miksi? Milloin?

Mikä on k:n arvo lopussa? Onko sitä?

Entä jos toisenlainen for-semantiikka?

```
K    DC    0
    ...
    LOAD R1, =20
    STORE R1, K

Loop LOAD  R2, =0
    LOAD  R1, K
    STORE R2, T(R1)

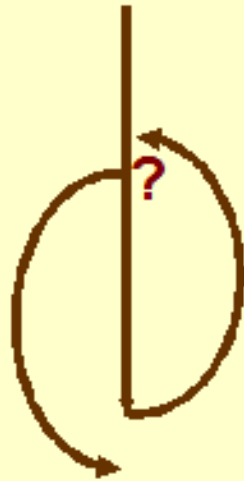
    LOAD  R1, K
    ADD   R1, =1
    STORE R1, K

    LOAD  R3, K
    COMP  R3, =50
    JLES  Loop
```

Copyright Teemu Kerola 2004

Tämä on tietenkin vain for-loopin yksinkertainen perusratkaisu. Todellisuudessa asiaa mutkistaa korkean tason ohjelmointikielen semantiikka ja erilaiset optimoinnit. Esimerkiksi, koodin voisi tehdä huomattavasti nopeammaksi, jos K:n arvo pidettäisiin koko ajan jossakin rekisterissä, mutta toisaalta tämä varaisi yhden rekisterin K:ta varten. Jotkut ohjelmointikielet vaativat loop-semantiikassaan, että tässä tapauksessa K:n arvo loopin jälkeen olisi 50, kun taas jotkut muut kielet nimenomaan antavat tilaa optimoinnille ja määrittelevät semantiikan, jossa K:n arvo loopin jälkeen on tuntematon.

While-do -lause

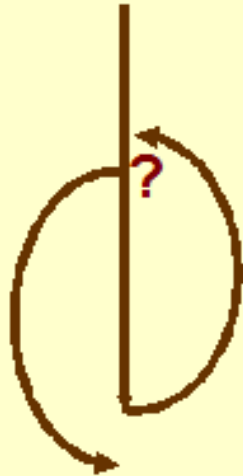


```
X = 12345;  
Xlog = 1;  
Y = 10;  
while (Y<X) {  
    Xlog++;  
    Y *= 10;  
}
```

Copyright Teemu Kerola 2004

While-looppi toteutetaan vastaavasti hyvin suoraviivaisesti perustapauksessaan. Tässä esimerkissä laskemme yksinkertaisessa loopissa triviaalilla tavalla muuttujan X 10-kantaisen logaritmin.

While-do -lause



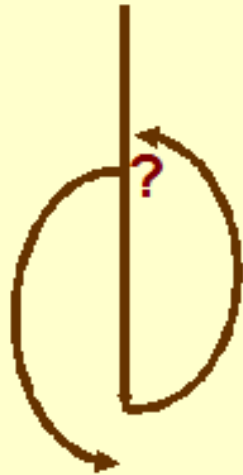
```
X = 12345;  
Xlog = 1;  
Y = 10;  
while (Y<X) {  
    Xlog++;  
    Y *= 10;  
}
```

```
LOAD  R1, =12345  
STORE R1, X  
LOAD  R1, =1 ; R1 on Xlog  
LOAD  R2, =10 ; R2 on Y
```

Copyright Teemu Kerola 2004

Ennen itse looppia alustamme silmukan omat muuttujat, Xlog'in ja Y'n, jotka tällä kertaa talletetaan rekistereihin R1 ja R2. Kääntäjä on siis jotenkin päättänyt, että tässä kohtaa koodia on järkevää uhrata niukoista rekistereistä kaksi kappaletta näiden muuttujien arvon tallettamiseen. Etuna Xlog'in ja Y'n arvojen rekistereissä pitämiseen on tietenkin se, että niitä ei tarvitse yhtä mittaa silmukan aikana lukea muistista tai kirjoittaa sinne. Toisaalta silmukan lopussa arvot tulee kuitenkin tallettaa muistiin, jotta rekisterit R1 ja R2 voidaan vapauttaa toisiokäyttöön.

While-do -lause



```
X = 12345;  
Xlog = 1;  
Y = 10;  
while (Y<X) {  
    Xlog++;  
    Y *= 10;  
}
```

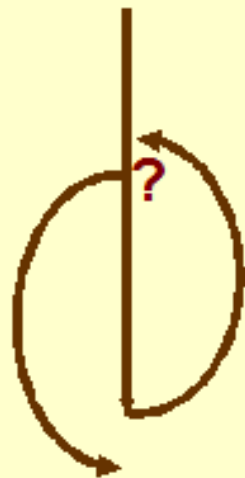
```
LOAD  R1, =12345  
STORE R1, X  
LOAD  R1, =1 ; R1 on Xlog  
LOAD  R2, =10 ; R2 on Y
```

```
While COMP  R2, X  
      JNLES Done
```

Copyright Teemu Kerola 2004

While-loopin semantiikan mukaan ehto toteutetaan ennen silmukan runkoa. Tässä tapauksessa vertailemme siis Y'n eli 10 potenssiin Xlog'in arvoa X'ään. Jos vertailun tulos on pienempi, niin työ jatkuu. Huomaa, että konekielessä muuttujan Xlog arvo pidetään rekisterissä R1, mutta X'n arvo on muistissa. Olisi ollut myös aivan mahdollista pitää X'nkin arvo rekisterissä, mutta ilmeisesti kääntäjä tällä kertaa oli päättänyt, että kaikki muut rekisterit olivat jo tärkeämmässä käytössä.

While-do -lause



```
X = 12345;  
Xlog = 1;  
Y = 10;  
while (Y<X) {  
    Xlog++;  
    Y *= 10;  
}
```

```
LOAD  R1, =12345  
STORE R1, X  
LOAD  R1, =1 ; R1 on Xlog  
LOAD  R2, =10 ; R2 on Y
```

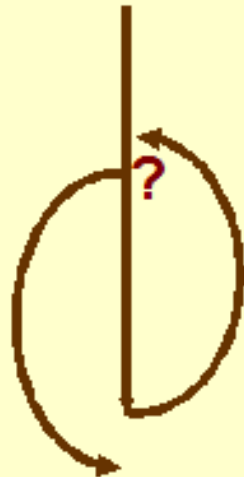
```
While COMP R2, X  
      JNLES Done
```

```
ADD    R1, =1  
MUL    R2, =10  
JUMP   While
```

Copyright Teemu Kerola 2004

Silmukan rungossa lisäämme ykkösen laskettavaan logaritmiin Xlog ja Päivitämme Y'n eli 10 potenssiin Xlog'in arvon. Nämä molemmat operaatiot on nyt hyvin helppo tehdä, koska molemmat arvot ovat valmiina rekistereissä, ja ne voi myös jättää rekistereihin.

While-do -lause



```
X = 12345;  
Xlog = 1;  
Y = 10;  
while (Y<X) {  
    Xlog++;  
    Y *= 10;  
}
```

```
LOAD  R1, =12345  
STORE R1, X  
LOAD  R1, =1 ; R1 on Xlog  
LOAD  R2, =10 ; R2 on Y
```

```
While COMP  R2, X  
      JNLES  Done
```

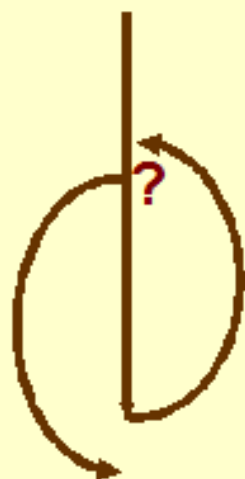
```
ADD    R1, =1  
MUL    R2, =10  
JUMP   While
```

```
Done STORE R1, Xlog ; talleta tulos  
      STORE R2, Y
```

Copyright Teemu Kerola 2004

Silmukan jälkeen meillä on nyt ylimääräinen vaihe, jota ei ollut aikaisemmin. Olimme päättäneet pitää silmukan muuttujat rekistereissä, joten ne pitää nyt tallettaa muistiin jatkokäyttöä varten. Rekisterit R1 ja R2 vapautuvat täten muihin tarkoituksiin. Toisaalta, on luultavaa, että laskennan välitulosta Y ei enää missään vaiheessa tarvita, joten sen muistiin tallettaminen on turhaa. Silmukan tuloksen Xlog ja välituloksen Y välisen hyvin tärkeän eron huomaaminen on kuitenkin aika vaikeata, joten kääntäjä luultavasti tallettaisi molemmat arvot muistiin.

While-do -lause



```
X = 12345;  
Xlog = 1;  
Y = 10;  
while (Y<X) {  
  Xlog++;  
  Y *= 10;  
}
```

Mitä pidetään muistissa ja mitä rekisterissä?

Missä rekisterissä?

```
LOAD  R1, =12345  
STORE R1, X  
LOAD  R1, =1 ; R1 on Xlog  
LOAD  R2, =10 ; R2 on Y
```

```
While COMP  R2, X  
      JNLES Done
```

```
ADD    R1, =1  
MUL    R2, =10  
JUMP   While
```

```
Done STORE R1, Xlog ; talleta tulos  
      STORE R2, Y
```

Copyright Teemu Kerola 2004

Silmukoissa on usein ongelmana päättää, mitkä muuttujat pidetään rekisterissä silmukan aikana ja mitkä muistissa. Hyvänä nyrkkisääntönä voi pitää, että usein viitattava tieto kannattaa aina pitää rekisterissä. Tiedon pitämisessä rekisterissä on se hyvä puoli, että tiedon viittaamisaika voi olla hyvinkin vain 10- tai 20-osa muistissa olevaan tietoon verrattuna. Huonona puolena on se, että rekistereiden vähyyden vuoksi jokin muu tieto täytyy sitten pitää muistissa. Lisäksi päätös täytyy lähes aina tehdä vajavaisen päättelytiedon perusteella hyvin nopeasti. Ei siis ihme, että nämä päätökset eivät ole aina oikein.

Koodin generointi

$X = Y + Y / 8;$

Kääntäjän viimeinen vaihe

- voi olla 50 % käännösajasta
- vain tämä vaihe täytyy muuttaa, kun kääntäjä portataan uudelle suorittimelle

Tavallisen koodin generointi

- alustukset, lauseet, kontrollirakenteet

Optimoidun koodin generointi

- käännös kestää (paljon) kauemmin - kertamaksu
- suoritus tapahtuu (paljon) nopeammin - moninkertainen hyöty (toivottavasti)
- milloin globaalin/paikallisen muuttujan X arvo kannattaa pitää rekisterissä?
- missä rekisterissä X:n arvo kannattaa pitää?
- monisanainen tieto (taulukot, tietueet) yleensä aina muistissa

Copyright Teemu Kerola 2004

Koodin generointi on kääntäjän viimeinen vaihe. Tämä on ainoa kohta kääntäjässä, jossa tarvitaan tietämystä kohdelaitteistosta ja sen konekielestä. Tämä tarkoittaa, että kääntäjä voidaan muuntaa eli 'portata' uudelle laitteistolle sopivaksi ainoastaan toteuttamalla uudelleen sen koodingenerointiosa eli 'backend'. Koodin generointi ei ole helppoa, koska todelliset laitteistot ovat monimutkaisia ja tehokkaan koodin toteuttamiseen tarvitaan syvällisempää tietämystä laitteistosta kuin ainoastaan sen konekäskykanta.

Koodin generointi

$X = Y + Y / 8;$

```
LOAD  R1, Y
LOAD  R2, Y
DIV    R2, =8
ADD    R1, R2
STORE R1, X
```

Kääntäjän viimeinen vaihe

- voi olla 50 % käännösajasta
- vain tämä vaihe täytyy muuttaa, kun kääntäjä portataan uudelle suorittimelle

Tavallisen koodin generointi

- alustukset, lauseet, kontrollirakenteet

Optimoidun koodin generointi

- käännös kestää (paljon) kauemmin - kertamaksu
- suoritus tapahtuu (paljon) nopeammin - moninkertainen hyöty (toivottavasti)
- milloin globaalin/paikallisen muuttujan X arvo kannattaa pitää rekisterissä?
- missä rekisterissä X:n arvo kannattaa pitää?
- monisanainen tieto (taulukot, tietueet) yleensä aina muistissa

Copyright Teemu Kerola 2004

Tavallinen koodin generointi on silti aika suoraviivaista. Erilaiset alustukset, lauseet ja lausekkeet sekä kontrollirakenteet toteutetaan konekielellä hyvin samalla tavalla kuin mitä tässäkin esimerkissä näytettiin. Lähtökohtana todelliselle koodin generoinnille ei korkean tason kielen kääntäjässä ole kuitenkaan korkean tason kielen koodi, vaan siitä eri vaiheiden avulla aikaansaatu geneerisen ohjelmointikielen koodin puumainen tietorakenne tai geneerisen koneen konekieli eli välikieli.

Koodin generointi

$X = Y + Y / 8;$

```
LOAD  R1, Y
LOAD  R2, Y
DIV   R2, =8
ADD   R1, R2
STORE R1, X
```

Kääntäjän viimeinen vaihe

- voi olla 50 % käännösajasta
- vain tämä vaihe täytyy muuttaa, kun kääntäjä portataan uudelle suorittimelle

Tavallisen koodin generointi

- alustukset, lauseet, kontrollirakenteet

```
LOAD  R1, Y
LOAD  R2, R1
SHRA  R2, =3
ADD   R1, R2
STORE R1, X
```

Optimoidun koodin generointi

- käännös kestää (paljon) kauemmin - kertamaksu
- suoritus tapahtuu (paljon) nopeammin - moninkertainen hyöty (toivottavasti)
- milloin globaalin/paikallisen muuttujan X arvo kannattaa pitää rekisterissä?
- missä rekisterissä X:n arvo kannattaa pitää?
- monisanainen tieto (taulukot, tietueet) yleensä aina muistissa

Copyright Teemu Kerola 2004

Optimoidun koodin generointi voi olla 10 tai 100 kertaa niin vaativaa kuin yksinkertaisen koodin generointi. Erilaisia optimointimahdollisuuksia on hyvin paljon ja niihin liittyy täsmällinen tieto sekä ohjelmasta että tietokonelaitteistosta. Esimerkiksi, voi olla tärkeää tietää, kuinka kauan muistista haku täsmälleen kestää tai kuinka todennäköistä on, että joku kontrollihaarautuma tapahtuu tai ei. Jos sitten samasta laitteistosta tulee uusi versio, jossa muistin hakuaika onkin yhden kellopulssin nopeampi, niin tällä voi olla vaikutusta generoitavaan koodiin ja ohjelma tulisi kääntää uudelleen tälle laitteistolle.

Optimoitu for-lause

```
for (k=0; k<50; ++k)  
    T[k] = 0;
```

```
K    DC    0  
    ...  
    LOAD  R1,=0  
    STORE R1,K  
  
Loop LOAD  R2,=0  
    LOAD  R1,K  
    STORE R2,T(R1)  
  
    LOAD  R1,K  
    ADD   R1,=1  
    STORE R1,K  
  
    LOAD  R3,K  
    COMP  R3,=50  
    JLES  Loop
```

Koodin koko (32b sanoja):	12
Suoritettuja käskyjä:	452
Suoritusaikaisia muistiviitteitä:	703 = 452+251
muuttujalla K arvo lopussa?	on

Copyright Teemu Kerola 2004

Tarkastellaan esimerkin vuoksi, kuinka aikaisemmin esitettyä for-lauseetta voisi optimoida. Looppia on kuitenkin tässä muokattu sen verran, että taulukko konaisuudessaan alustetaan nolaksi, alkaen siis indeksistä nolla. Optimoinnissa on aina ideana saada jokin asia paremmaksi. Mitä haluamme siis optimoida? Tästä koodista voidaan laskea muutamia perusmittoja. Koodi vie muistissa tilaa 12 sanaa (1 sana dataa ja 11 koodia). Suoritusaikana siitä suoritetaan aina samat 452 käskyä: 9 sanan looppia 50 kertaa plus 2 sanan alustus. Muistiviitteitä tulee 703, joista 251 on dataviitteitä ja pääosa 452 koodiviitteitä. Jokainen käskyhän pitää aina hakea muistista.

Optimoitu for-lause

```
for (k=0; k<50; ++k)
    T[k] = 0;
```

```
        LOAD R1, =0 ; K
        LOAD R2, =0 ; vakio 0
Loop    STORE R2, T(R1)
        ADD  R1, =1
        COMP R1, =50
        JLES Loop
```

Mitä eroja? Onko tämä OK?

Koodin koko (32b sanoja):	6	vs.	12
Suoritettuja käskyjä:	202	vs.	452
Suoritusaikaisia muistiviitteitä:	252	vs.	703
muuttujalla K arvo lopussa?	ei	vs.	on

```
K      DC    0
...
        LOAD R1, =0
        STORE R1, K

Loop    LOAD  R2, =0
        LOAD  R1, K
        STORE R2, T(R1)

        LOAD  R1, K
        ADD   R1, =1
        STORE R1, K

        LOAD  R3, K
        COMP  R3, =50
        JLES  Loop
```

Copyright Teemu Kerola 2004

Koodia voidaan optimoida aika yksinkertaisesti käyttämällä enemmän rekistereitä. Nyt tosin muuttujan K arvo ei jää muistiin loopin jälkeen - itse asiassa muuttujalle K ei edes varata muistitilaa missään vaiheessa. Koodin koko on nyt vain puolet alkuperäisestä, mutta suoritusajana säästö on vielä enemmän. Käskyjä suoritetaan vain 45% ja muistiviitteitä vain 36% alkuperäisestä. Jos tämä skaalataan johonkin todelliseen ohjelmaan niin kyseessä voisi olla gigatavujen tilansäästö ja/tai ohjelman suoritusajan pieneneminen 12 tunnista 5 tuntiin. Haittana on käännösajan piteneminen ehkä 5- tai 20-kertaiseksi.

Optimoitu for-lause

```
for (k=0; k<50; ++k)
    T[k] = 0;
```

```
        LOAD  R1, =49 ; i
        LOAD  R2, =0  ; vakio 0
Loop    STORE R2, T(R1)
        SUB   R1, =1
        JNNEG Loop
```

Mitä eroja? Onko tämä OK?

Koodin koko (32b sanoja):	5	vs.	12
Suoritettuja käskyjä:	152	vs.	452
Suoritusaikaisia muistiviitteitä:	202	vs.	703
muuttujalla K arvo lopussa?	ei	vs.	on

```
K      DC    0
      ...
      LOAD  R1, =0
      STORE R1, K

Loop   LOAD  R2, =0
      LOAD  R1, K
      STORE R2, T(R1)

      LOAD  R1, K
      ADD   R1, =1
      STORE R1, K

      LOAD  R3, K
      COMP  R3, =50
      JLES  Loop
```

Copyright Teemu Kerola 2004

Koodia voi vieläkin optimoida ihan konekielen tasollakin. Jos loopin muuntelumuuttujan arvo käydäänkin läpi suuremmasta pienempään, niin loppumisehto voidaan nyt testata yhdellä käskyllä edellämainitun kahden käskyn asemesta. Säästämme siis yhden käskyn eli 25% koodista jokaisella silmukkakierroksella. Koodin koko on nyt vain 42% alkuperäisestä. Suoritusaikaisia käskyjä on nyt vain yksi kolmasosa ja muistiviitteitä vain 29% alkuperäisistä. Haittana on tietenkin käännösajan piteneminen vielä entisestään.

Virhetilanteisiin varautuminen

Suoritin tarkistaa automaattisesti käskyn suoritusaikana ja aiheuttaa keskeytyksen tarvittaessa (ja sitä kautta käyttöjärjestelmän palvelupyynnön)

- nopea: tarkistus samaan aikaan käskyn suorituksen kanssa
- automaattinen, ohjelmoijan ei tarvitse muistaa
- integer overflow, divide by zero, ...

```
ADD    R1, R2
DIV    R2, R3
DIV    R2, =0
```

Generoidut käskyt tarkistavat jonkin asian ja aiheuttavat käyttöjärjestelmän palvelupyynnön tarvittaessa

- hidas: konekäskyt tekevät tarkistuksia, ei tuottavaa työtä
- manuaalinen, koodi pitää laittaa paikalleen
- index out of bounds, bad methop, bad input, ihan mitä vain!!!

```
STORE R2, Taulu(R1)
```

```
; --- taulukkoviite kohtaan Taulu[R1], taulukon indeksi R1:ssä -----
                JNNEG R1, UpperLimit
                SVC    SP, =BadIndex ; <===== virhe R1 < 0
UpperLimit COMP R1, Tsize
                JLES   IndexOK
                SVC    SP, =BadIndex ; <===== virhe R1 >= Tsize
IndexOK        STORE R2, Taulu(R1)
```

Copyright Teemu Kerola 2004

Kaikkiin virheisiin pitää aina varautua. Peruslähtökohta on, että ohjelman suorituksen aikana ei voi tapahtua mitään, mihin ei oltaisi varauduttu. Virhetilanteet eivät siis koskaan ole yllättäviä itse tyypiltään. Ainoa, mikä niissä voi yllättää, on niiden tapahtuma-aika. Virheisiin varaudutaan ohjelmassa kahdella tavalla. Osan virheistä havaitsee laitteisto eli konekäskyjen suorituspiirit automaattisesti, mutta iso osa virheistä havaitaan sitä varten generoidulla koodilla.

Virhetilanteisiin varautuminen

Suoritin tarkistaa automaattisesti käskyn suoritusaikana ja aiheuttaa keskeytyksen tarvittaessa (ja sitä kautta käyttöjärjestelmän palvelupyynnön)

- nopea: tarkistus samaan aikaan käskyn suorituksen kanssa
- automaattinen, ohjelmoijan ei tarvitse muistaa
- integer overflow, divide by zero, ...

```
ADD    R1, R2
DIV     R2, R3
DIV     R2, =0
```

Generoidut käskyt tarkistavat jonkin asian ja aiheuttavat käyttöjärjestelmän palvelupyynnön tarvittaessa

- hidas: konekäskyt tekevät tarkistuksia, ei tuottavaa työtä
- manuaalinen, koodi pitää laittaa paikalleen
- index out of bounds, bad methop, bad input, ihan mitä vain!!!

```
STORE R2, Taulu(R1)
```

```
; --- taulukkoviihte kohtaan Taulu[R1], taulukon indeksi R1:ssä -----
                JNNEG R1, UpperLimit
                SVC    SP, =BadIndex ; <==== virhe R1 < 0
UpperLimit COMP R1, Tsize
                JLES   IndexOK
                SVC    SP, =BadIndex ; <==== virhe R1 >= Tsize
IndexOK  STORE R2, Taulu(R1)
```

Copyright Teemu Kerola 2004

Jokaiseen konekäskyyn liittyy ennalta määritetty ja käskyjen suorituspäireihin valmiiksi toteutettu joukko virheentarkistuksia. Jos virhe havaitaan, niin laitteisto ilmaisee siitä keskeytyssignaalilla (arvo 1 jossakin johtimessa), minkä seurauksena kontrolli siirtyy käyttöjärjestelmärutiinille, joka huolehtii virheen käsittelystä. Jos virhe on paha, kuten esimerkiksi nollalla jako, niin ohjelman suoritus keskeytetään. Muita tällaisia virhetilanteita on esimerkiksi aritmetiikkaoperaation ylivuoto, jossa operaation tulosta ei voidakaan esittää tulosrekisterin 32 bitillä.

Virhetilanteisiin varautuminen

Suoritin tarkistaa automaattisesti käskyn suoritusaikana ja aiheuttaa keskeytyksen tarvittaessa (ja sitä kautta käyttöjärjestelmän palvelupyynnön)

- nopea: tarkistus samaan aikaan käskyn suorituksen kanssa
- automaattinen, ohjelmoijan ei tarvitse muistaa
- integer overflow, divide by zero, ...

```
ADD    R1, R2
DIV    R2, R3
DIV    R2, =0
```

Generoidut käskyt tarkistavat jonkin asian ja aiheuttavat käyttöjärjestelmän palvelupyynnön tarvittaessa

- hidas: konekäskyt tekevät tarkistuksia, ei tuottavaa työtä
- manuaalinen, koodi pitää laittaa paikalleen
- index out of bounds, bad methop, bad input, ihan mitä vain!!!

```
STORE R2, Taulu(R1)
```

```
; --- taulukkoviite kohtaan Taulu[R1], taulukon indeksi R1:ssä -----
                JNNEG R1, UpperLimit
                SVC    SP, =BadIndex ; <==== virhe R1 < 0
UpperLimit COMP R1, Tsize
                JLES   IndexOK
                SVC    SP, =BadIndex ; <==== virhe R1 >= Tsize
IndexOK  STORE R2, Taulu(R1)
```

Copyright Teemu Kerola 2004

Mitä tahansa loogista virhetilannetta voidaan testata ohjelmallisesti. Alla olevassa esimerkissä tutkitaan ennen taulukkoviitteen toteuttamista, että taulukon indeksi on todella annetuissa rajoissa. Jos ei ole, niin kutsutaan käyttöjärjestelmäpalvelua 'BadIndex' hoitamaan asia. Taulukkoviittausten indeksitarkistusten puute on suurimpia virusten hyödyntämiä järjestelmäheikkouksia, koska tarkistusten puuttuessa indeksille voidaan antaa jokin hyvin suuri arvo ja sen avulla sitten muuttaa ihan muualla olevaa tietoa. Tässä esimerkissä yhden viitteen tekemiseksi tarvitaan aina suorittaa kolme tarkistuskäskyä, mikä on aika kallis hinta yhdestä taulukkoviitteestä.

Taulukon indeksitarkistus

```
for (k=20; k<50; ++k)
    T[k] = 0;
```

K	DC	0
...		
T	DS	50 ;data
Tsize	DC	50 ;pituus

```
LOAD R1, =20
STORE R1, K
```

```
Loop LOAD R2, =0
      LOAD R1, K
```

```
STORE R2, T(R1)
```

```
LOAD R1, K
ADD R1, =1
STORE R1, K
```

```
LOAD R3, K
COMP R3, =50
JLES Loop
```

Copyright Teemu Kerola 2004

Tässä esimerkissä lisäämme taulukon indeksitarkistukset aikaisemmin esitettyyn taulukon alustussilmukkaan. Indeksitarkistuksiin pitää varautua jo taulukkoa varattaessa (tai keosta allokoitaessa), koska meidän tulee johonkin tallettaa taulukon pituus tarkistuksen tekemistä varten. Muuttuja Tsize alustetaan tässä taulukon T pituudeksi. Otaksumme, että taulukon indeksi alkaa aina nollasta.

Taulukon indeksitarkistus

```
for (k=20; k<50; ++k)
    T[k] = 0;
```

K	DC	0
...		
T	DS	50 ;data
Tsize	DC	50 ;pituus

`cc -fbounds-check myprog.c`

```
LOAD R1, =20
STORE R1, K
```

```
Loop LOAD R2, =0
      LOAD R1, K
```

```
JNNEG R1, Ok
Bad  SVC  SP, =BadIndex
Ok   COMP R1, Tsize
      JNLES Bad
```

```
STORE R2, T(R1)
```

```
LOAD R1, K
ADD R1, =1
STORE R1, K
```

```
LOAD R3, K
COMP R3, =50
JLES Loop
```

Copyright Teemu Kerola 2004

Joka taulukon viittauskohtaan laitetaan nyt indeksin tarkistuskoodi ennen itse taulukkoiviittauksen tekemistä. Koodi tarkistaa nyt, että indeksi on sallituissa rajoissa eli kuuluu arvojoukkoon 0-49. Jos näin ei ole, niin käyttöjärjestelmärutiinia BadIndex kutsutaan 'hoitamaan asia'. Indeksitarkistusten tekeminen kuluttaa merkittävästi suoritusaikaa ja koko ohjelma toimii nyt jonkin verran hitaammin. Joissakin ohjelmointikielissä (esim C) on kääntäjän optio, jonka avulla voidaan valita, generoidaanko taulukkoiviittauksen rajatarkistukset vai ei.

Taulukon indeksitarkistus

```
for (k=20; k<50; ++k)
    T[k] = 0;
```

K	DC	0
...		
T	DS	50 ;data
Tsize	DC	50 ;pituus

Voisiko tarkistukset jättää joskus pois?
Milloin? Miksi?
Miten ohjelmoisit päättelyn?

```
LOAD R1, =20
STORE R1, K
```

```
Loop LOAD R2, =0
      LOAD R1, K
```

```
JNNEG R1, Ok
```

```
Bad SVC SP, =BadIndex
```

```
Ok COMP R1, Tsize
     JNLES Bad
```

```
STORE R2, T(R1)
```

```
LOAD R1, K
ADD R1, =1
STORE R1, K
```

```
LOAD R3, K
COMP R3, =50
JLES Loop
```

Copyright Teemu Kerola 2004

Joissakin tapauksissa rajatarkistukset voidaan jättää tarpeettomana pois, jos jotenkin voidaan osoittaa etukäteen (siis käännösaikana), että tässä kohtaa taulukon indeksi on aina oikein. Esimerkiksi, juuri tässä esimerkissä voidaan todeta, että silmukan muuntelumuuuttujan ja samalla taulukon indeksin arvojoukko on 20-49, mikä sisältyy taulukon T indeksin arvojoukkoon 0-49, joten tarkistuskoodia ei tarvita. Päättely muuttuu huomattavasti hankalammaksi, jos muuntelumuuuttujaa ei suoraan käytetä indeksinä, vaan indeksi lasketaan jonkin lausekkeen avulla.

Taulukon alaraja ei ole nolla

```
for (k=20; k<50; ++k)
    T[k] = 0;
```

K	DC	0
...		
T	DS	30 ; data, vain 30 alkioita
TLow	DC	20 ; alaraja
THigh	DC	50 ; yläraja+1
TSize	DC	30 ; koko

sijainti	arvo
T:	T [20]
T+1:	T [21]
...	...
T+29:	T [49]

```
LOAD R1, =20
STORE R1, K

Loop LOAD  R2, =0
      LOAD  R1, K

      SUB    R1, TLow
      STORE R2, T(R1)

      LOAD  R1, K
      ADD   R1, =1
      STORE R1, K

      LOAD  R3, K
      COMP  R3, =50
      JLES  Loop
```

Copyright Teemu Kerola 2004

Toinen taulukkojen käyttöön liittyvä monimutkaisempi tilanne ilmenee, jos taulukon alaraja ei olekaan nolla. Esimerkissä 30-alkioiselle taulukolle T varataan siis tilaa vain 30 sanaa, ja taulukon ensimmäisen alkion T[20] osoite on symbolin T arvo. Normaalitapauksessahan symboli T osoitti taulukon alkioon T[0], mutta nyt siis alkioon T[20]. Vastaavasti alkion T[21] osoite on T+1 jne. Alkion T[i] osoite on siis T+i-20. Tämä pitää nyt sitten koodata joka paikkaan, jossa taulukkoon T viitataan. Tämä monimutkaistaa koodia jonkin verran, minkä vuoksi useissa ohjelmointikielissä (esim. C tai Java) taulukon indeksit alkavat aina nollasta.

Taulukon alaraja ei ole nolla

```
for (k=20; k<50; ++k)
    T[k] = 0;
```

K	DC	0
...		
T	DS	30 ; data, vain 30 alkiota
TLow	DC	20 ; alaraja
THigh	DC	50 ; yläraja+1
TSize	DC	30 ; koko

Entä indeksitarkistukset?

sijainti	arvo
T:	T [20]
T+1:	T [21]
...	...
T+29:	T [49]

```
LOAD R1, =20
STORE R1, K

Loop LOAD  R2, =0
      LOAD  R1, K

      SUB    R1, TLow
      STORE R2, T(R1)

      LOAD  R1, K
      ADD   R1, =1
      STORE R1, K

      LOAD  R3, K
      COMP  R3, =50
      JLES  Loop
```

Copyright Teemu Kerola 2004

Tästä esimerkistä on tahallaan jätetty pois indeksitarkistukset, joista pitää tietenkin huolehtia myös joka viittauskerralla.

Moniulotteiset taulukot

sarake	0	1	2	3
T: rivi 0				
rivi 1				

Ohjelmointikieli voi suoraan tukea moniulotteisia taulukoita

$X = Tbl[i, j]$

$Y = Arr[k][6][y+2]$

Toteutus konekielitasolla (useimmissa arkkitehtuureissa) on silti yksiulotteinen taulukko

- vain yksi indeksirekisteri konekäskyssä
- määrittele yksiulotteinen taulukko, jossa tarpeeksi alkioita

LOAD R5, 'T[1, 2]' ??

T:

Moniosainen toteutus

- laske alkion osoite toteutetussa yksiulotteisessa taulukossa
- käytä indeksoitua tiedonosoitusmoodia alkioon viittaamiseksi

Moniosainen toteutus osoitinmuuttujan avulla

- laske alkion osoite muistissa johonkin osoitinmuuttujaan (rekisteriin)
- käytä epäsuoraa tiedonosoitusmoodia alkioon viittaamiseksi

Copyright Teemu Kerola 2004

Moniulotteiset taulukot on hivenen vaikeampi toteuttaa, koska laitteisto ei suoraan tue niiden käyttöä samalla tavalla kuin yksiulotteisia taulukoita. Yleensä indeksirekistereitä on vain yksi. Toisaalta, on olemassa arkkitehtuureja (esim. Digital PDP), joissa on kaksi indeksirekisteriä, jolloin myös 2-ulotteiset taulukot on helpompi toteuttaa. Oletamme nyt kuitenkin tässä, että käytössä on vain yksi indeksirekisteri. Esimerkiksi 2-ulotteinen taulukko $T[2][4]$ toteutetaan 1-ulotteisena 8-alkioisena taulukkona T, jossa kunkin alkuperäisen 2-ulotteisen taulukon alkion sijainti on helposti laskettavissa indeksien perusteella.

Moniulotteiset taulukot

sarake	0	1	2	3
T: rivi 0				
rivi 1				

Ohjelmointikieli voi suoraan tukea moniulotteisia taulukoita

$X = Tbl[i, j]$

$Y = Arr[k][6][y+2]$

Toteutus konekielitasolla (useimmissa arkkitehtuureissa) on silti yksiulotteinen taulukko

- vain yksi indeksirekisteri konekäskyssä
- määrittele yksiulotteinen taulukko, jossa tarpeeksi alkioita

LOAD R5, 'T[1, 2]' ??

T:

Moniosainen toteutus

- laske alkion osoite toteutetussa yksiulotteisessa taulukossa
- käytä indeksoitua tiedonosoitusmoodia alkioon viittaamiseksi

Moniosainen toteutus osoitinmuuttujan avulla

- laske alkion osoite muistissa johonkin osoitinmuuttujaan (rekisteriin)
- käytä epäsuoraa tiedonosoitusmoodia alkioon viittaamiseksi

Copyright Teemu Kerola 2004

Moniulotteisia taulukoita ei siis todellisuudessa toteuteta sellaisenaan tai edes sisäkkäin olevina 1-ulotteisina taulukkoina kuten esimerkiksi Javassa tehdään. Sen sijaan toteutus on lähes kaikissa nykyisissä koneissa yksi-ulotteinen taulukko, johon moni-ulotteinen taulukko on jollain tavoin matemaattisesti kuvattu. Voimme siis ajatella ja helposti käyttää 2- tai 5-ulotteisia taulukoita, vaikka niiden toteutus onkin 1-ulotteinen. Useimmiten meidän ei tarvitse edes ajatella toteutuspuolta, koska korkean tason kääntäjät huolehtivat siitä. Toisaalta, konekielen tasolla toteutus on itse tehtävä.

Moniulotteiset taulukot

sarake	0	1	2	3
T: rivi 0				
rivi 1				

Ohjelmointikieli voi suoraan tukea moniulotteisia taulukoita

$X = \text{Tbl}[i, j]$

$Y = \text{Arr}[k][6][y+2]$

Toteutus konekielitasolla (useimmissa arkkitehtuureissa) on silti yksiulotteinen taulukko

- vain yksi indeksirekisteri konekäskyssä
- määrittele yksiulotteinen taulukko, jossa tarpeeksi alkioita

LOAD R5, 'T[1, 2]' ??

T:

Moniosainen toteutus

- laske alkion osoite toteutetussa yksiulotteisessa taulukossa
- käytä indeksoitua tiedonosoitusmoodia alkioon viittaamiseksi

```
LOAD R1, =1
MUL R1, =4
ADD R1, =2
LOAD R5, T(R1)
```

Moniosainen toteutus osoitinmuuttujan avulla

- laske alkion osoite muistissa johonkin osoitinmuuttujaan (rekisteriin)
- käytä epäsuoraa tiedonosoitusmoodia alkioon viittaamiseksi

Copyright Teemu Kerola 2004

Moniulotteisten taulukoiden viittaukset tehdään aina kahdessa osassa. Ensin lasketaan viitatun alkion sijainti toteutetun 1-ulotteisen taulukon sisällä johonkin indeksirekisteriin ja sitten itse viittaus tehdään tavalliseen tapaan indeksoitua tiedonosoitusmuotoa käyttäen. Tottakai myös tässä tapauksessa pitäisi molemmat indeksit tarkistaa ennen varsinaista muistinvittautusta. Aikaisemman yhden indeksin asemesta koodin tulee tarkistaa nyt kaikki käytetyt indeksit.

Moniulotteiset taulukot

sarake	0	1	2	3
T: rivi 0				
rivi 1				

Ohjelmointikieli voi suoraan tukea moniulotteisia taulukoita

$X = \text{Tbl}[i, j]$

$Y = \text{Arr}[k][6][y+2]$

Toteutus konekielitasolla (useimmissa arkkitehtuureissa) on silti yksiulotteinen taulukko

- vain yksi indeksirekisteri konekäskyssä
- määrittele yksiulotteinen taulukko, jossa tarpeeksi alkioita

LOAD R5, 'T[1, 2]' ??

T:

Moniosainen toteutus

- laske alkion osoite toteutetussa yksiulotteisessa taulukossa
- käytä indeksoitua tiedonosoitusmoodia alkioon viittaamiseksi

Moniosainen toteutus osoitinmuuttujan avulla

- laske alkion osoite muistissa johonkin osoitinmuuttujaan (rekisteriin)
- käytä epäsuoraa tiedonosoitusmoodia alkioon viittaamiseksi

```
LOAD R1, =T
ADD R1, =4 ; rivi 0
ADD R1, =2
LOAD R5, @R1
```

Copyright Teemu Kerola 2004

Toinen vaihtoehto moniulotteisten (ja myös 1-ulotteisten) taulukoiden viittaamiseksi on ensin laskea johonkin rekisteriin viitatus alkion osoite muistissa, ja sitten viitata siihen epäsuoraa tiedonosoitusmoodia käyttäen. Viittauskohdan oikeellisuustarkistuksen voi tehdä joko tavalliseen tapaan tarkistamalla indeksien arvot, tai sitten tarkistamalla, että laskettu alkion muistiosoite (tässä R1) osuu todella tähän taulukkoon.

Esimerkki 2-ulotteisesta taulukosta

```
int [] [] T = new int [4] [3];  
...  
Y = T[i][j];
```

T	0	1	2
0			
1			
2			
3			

looginen rakenne

T:	T[0][0]
	T[0][1]
	T[0][2]
	T[1][0]
	T[1][1]
	T[1][2]
	T[2][0]
	T[2][1]
	T[2][2]
	T[3][0]
	T[3][1]
	T[3][2]

fyysinen rakenne

T	DS	12 ; 4 riviä, 3 sar.
Trows	DC	4 ; rivi-lkm
Tcols	DC	3 ; sarake-lkm

```
...  
;      Y = T[i][j]  
      LOAD R1, i  
  
      MUL   R1, Tcols  
  
      ADD   R1, j  
  
      LOAD  R2, T(R1)  
      STORE R2, Y
```

Copyright Teemu Kerola 2004

Tässä esimerkissä meillä on 2-ulotteinen taulukko T, jossa on 4 riviä ja kullakin rivillä 3 saraketta. T toteutetaan 1-ulotteisena taulukkona ja symbolin T arvona on taulukon ensimmäisen alkion osoite. Määrittelemme myös apumuuttujat Trows ja Tcols, jotka alustetaan taulukon rivien ja sarakkeiden lukumääräksi. Trows on tässä mukana täydellisyyden vuoksi. Sitä ei tässä esimerkissä enempää käytetä, mutta sitä tarvittaisiin käytännössä indeksien oikeellisuustarkistukseen. Tässä esimerkissä emme murehdi indeksitarkistuksista, vaikka oikeasti ne tietenkin kuuluvat asiaan.

Esimerkki 2-ulotteisesta taulukosta

```
int [][] T = new int [4] [3];  
...  
Y = T[i][j];
```

T	0	1	2
0			
1			
2			
3			

looginen rakenne

T:	T[0][0]
	T[0][1]
	T[0][2]
	T[1][0]
	T[1][1]
	T[1][2]
	T[2][0]
	T[2][1]
	T[2][2]
	T[3][0]
	T[3][1]
	T[3][2]

fyysinen rakenne

T DS 12 ; 4 riviä, 3 sar.
Trows DC 4 ; rivi-lkm
Tcols DC 3 ; sarake-lkm

```
...  
;      Y = T[i][j]  
      LOAD R1, i  
  
      MUL  R1, Tcols  
  
      ADD  R1, j  
  
      LOAD R2, T(R1)  
      STORE R2, Y
```

Copyright Teemu Kerola 2004

Sijoituslauseeseen 'y=T[i][j]' taulukkoviite toteutetaan nyt kaksivaiheisesti. Indeksien i ja j avulla lasketaan viitatus alkion osoite yksiulotteisessa toteutustaulukossa T ja tämä osoite talletetaan indeksirekisteriin. Varsinainen muistiviite tehdään sitten samalla tavalla kuin muutenkin yksiulotteisille taulukoille. Rivi-indeksi i tarkoittaa, että ennen viitattua riviä i meillä on i kpl kokonaisia Tcols pituisia rivejä muistissa, joissa on siis i*Tcols sanaa. Rivillä i ennen j'nnettä alkioita meillä on vielä j sanaa, joten yhteensä T:tä pitää indeksoida arvolla i*Tcols+j, kun viitataan alkioon T[i][j].

Esimerkki 2-ulotteisesta taulukosta

```
int [ ] [ ] T = new int [4] [3];
...
Y = T[i][j];
```

T	0	1	2
0			
1			
2			
3			

looginen rakenne

T:	T[0][0]
	T[0][1]
	T[0][2]
	T[1][0]
	T[1][1]
	T[1][2]
	T[2][0]
	T[2][1]
	T[2][2]
	T[3][0]
	T[3][1]
	T[3][2]

fyysinen rakenne

```
T      DS  12 ; 4 riviä, 3 sar.
Trows  DC   4 ; rivi-lkm
Tcols  DC   3 ; sarake-lkm
```

```
...
;      Y = T[i][j]
      LOAD R1, i
```

Esimerkki
i = 1, j = 2

R1 = 1

```
MUL   R1, Tcols
```

```
ADD   R1, j
```

```
LOAD  R2, T(R1)
STORE R2, Y
```

Copyright Teemu Kerola 2004

Käydään taulukkoviite läpi vielä konkreettisen esimerkin kera, jossa muuttujien i ja j arvot ovat 1 ja 2. Laskemme siis lauseketta $i * Tcols + j$. Aloitamme lataamalla R1:een rivi-indeksi i'n arvon 1. Koska taulukon indeksit tässä esimerkissä alkavat nolasta, niin viittaus riville 1 tarkoittaa samalla sitä, että muistiin talletetussa taulukossa on 1 kpl kokonaisia rivejä ennen viitattua riviä.

Esimerkki 2-ulotteisesta taulukosta

```
int [ ] [ ] T = new int [4] [3];
```

```
...
```

```
Y = T[i][j];
```

T	0	1	2
0			
1			
2			
3			

looginen rakenne

T:	T[0][0]
	T[0][1]
	T[0][2]
	T[1][0]
	T[1][1]
	T[1][2]
	T[2][0]
	T[2][1]
	T[2][2]
	T[3][0]
	T[3][1]
	T[3][2]

fyysinen rakenne

T DS 12 ; 4 riviä, 3 sar.

Trows DC 4 ; rivi-lkm

Tcols DC 3 ; sarake-lkm

Esimerkki

i = 1, j = 2

R1 = 1

```
...
; Y = T[i][j]
LOAD R1, i
```

```
MUL R1, Tcols
```

R1 = 3

```
ADD R1, j
```

```
LOAD R2, T(R1)
STORE R2, Y
```

Copyright Teemu Kerola 2004

Viitatus alkion rivi-indeksi 1 on nyt R1:ssä ja se on siis ennen viitattua riviä olevien kokonaisten rivien lukumäärä. Kun R1 kerrotaan rivin pituudella eli sarakkeiden lukumäärällä Tcols eli 3'lla, saadaan kaikkien edeltävien rivien sisältämien sanojen määrä R1:een.

Esimerkki 2-ulotteisesta taulukosta

```
int [ ] [ ] T = new int [4] [3];
...
Y = T[i][j];
```

T	0	1	2
0			
1			
2			
3			

looginen rakenne

T:	T[0][0]
	T[0][1]
	T[0][2]
	T[1][0]
	T[1][1]
	T[1][2]
	T[2][0]
	T[2][1]
	T[2][2]
	T[3][0]
	T[3][1]
	T[3][2]

fysinen rakenne

T DS 12 ; 4 riviä, 3 sar.
 Trows DC 4 ; rivi-lkm
 Tcols DC 3 ; sarake-lkm

Esimerkki
 i = 1, j = 2

```
...
Y = T[i][j]
LOAD R1, i
R1 = 1

MUL R1, Tcols
R1 = 3

ADD R1, j
R1 = 5

LOAD R2, T(R1)
STORE R2, Y
```

Tarkistukset?

0 ≤ i < 4
 0 ≤ j < 3

Copyright Teemu Kerola 2004

Kun R1'een sitten vielä lisätään sarakeindeksi j'n arvo 2 eli tällä rivillä ennen viitattua alkioita olevien sanojen lukumäärä, saadaan R1'een alkion T[1][2] suhteellinen osoite rakenteessa T. Loppu onkin sitten triviaalia. Koodi näyttää lyhyeltä, mutta siitä puuttuukin vielä kaikki välttämättömät tarkistukset.

Moniulotteiset taulukot voi tallettaa muistiin itse asiassa hyvinkin monella eri tavalla. Tärkeätä on vain jotenkin sopia tietyistä järjestyksestä, jotta taulukon alkion sijainti voidaan laskea nopeasti aina joka viittauksen yhteydessä käytetyistä taulukon indekseistä. Yleisin menetelmä lienee aiemmin esitelty rivi kerrallaan talletus, jossa taulukko talletetaan isolle yhtenäiselle muistialueelle, yksi rivi kerrallaan.

Moniulotteisten taulukoiden talletus muistiin

Talletus rivi kerrallaan

- C, Pascal, Java

Talletus sarake kerrallaan

- Fortran

Talletus riveittäin rivitaulukkoon

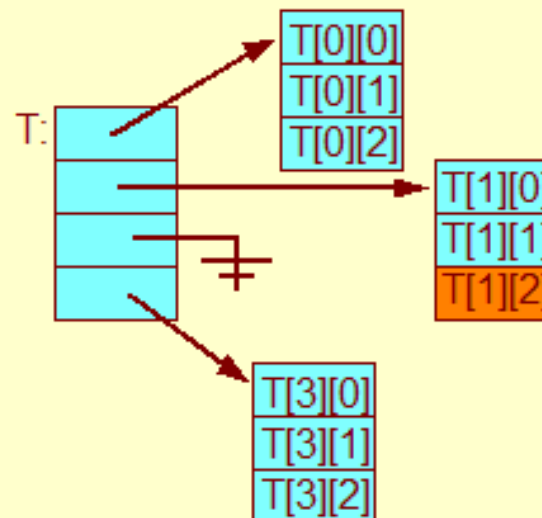
- vain rivit yhtenäisenä lohkona

Entä 3-, 4-, tai useampi- ulotteiset taulukot?

- samalla tavalla...

T	0	1	2
0			
1			
2			
3			

$X = T[1][2]$



T:	T[0][0]	T[0][1]	T[0][2]	T[1][0]	T[1][1]	T[1][2]	T[2][0]	T[2][1]	T[2][2]	T[3][0]	T[3][1]	T[3][2]

T:	T[0][0]	T[1][0]	T[2][0]	T[3][0]	T[0][1]	T[1][1]	T[2][1]	T[3][1]	T[0][2]	T[1][2]	T[2][2]	T[3][2]

```
LOAD R1, j
MUL   R1, Trows
ADD   R1, i
LOAD  R2, T(R1)
STORE R2, X
```

Copyright Teemu Kerola 2004

Aivan yhtä hyvin 2-ulotteiset taulukot voitaisiin tallettaa sarake kerrallaan yhtenäiselle muistialueelle. Esimerkiksi Fortran-kääntäjä käyttää tätä menetelmää. Käytännössä ei ole juurikaan väliä, talletetaanko 2-ulotteiset taulukot riveittäin vai sarakettain, koska useimmat 2-ulotteisia taulukoita käyttävät sovellukset käyvät taulukoita läpi sekä rivi- että sarakejärjestyksessä. Jos taas sovelluksen tiedetään lähes aina käyvän taulukoita läpi rivijärjestyksessä, niin silloin olisi välimuistin toiminnan kannalta parempi, jos taulukot on myös talletettu muistiin riveittäin.

Moniulotteisten taulukoiden talletus muistiin

Talletus rivi kerrallaan

- C, Pascal, Java

Talletus sarake kerrallaan

- Fortran

Talletus riveittäin rivitaulukkoon

- vain rivit yhtenäisenä lohkona

Entä 3-, 4-, tai useampiulotteiset taulukot?

- samalla tavalla...

T	0	1	2
0			
1			
2			
3			

$$X = T[1][2]$$

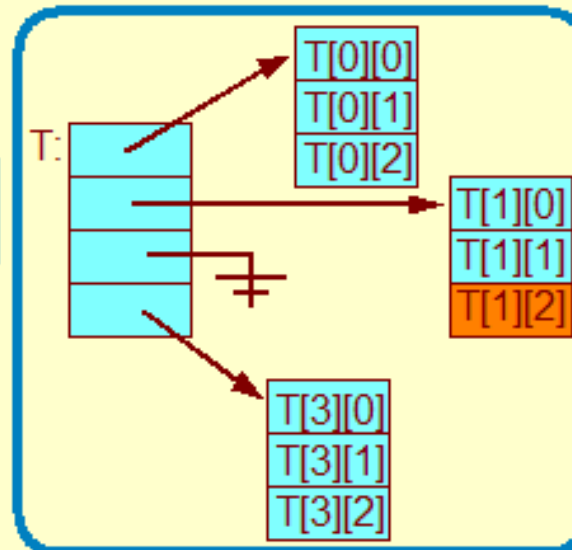
Diagram illustrating a sequence of nodes in a 4x4 grid, labeled T. The nodes are arranged in two columns, each with 4 rows. The nodes are labeled T[i][j], where i is the row index and j is the column index. The nodes are connected by arrows, forming a path from the top-left node to the bottom-right node. The path is highlighted in orange.

Left Column (T):

- T[0][0]
- T[0][1]
- T[0][2]
- T[1][0]
- T[1][1]
- T[1][2]
- T[2][0]
- T[2][1]
- T[2][2]
- T[3][0]
- T[3][1]
- T[3][2]

Right Column (T):

- T[0][0]
- T[1][0]
- T[2][0]
- T[3][0]
- T[0][1]
- T[1][1]
- T[2][1]
- T[3][1]
- T[0][2]
- T[1][2]
- T[2][2]
- T[3][2]



```
LOAD R1, i
LOAD R1, T(R1)
JZER AllokoiTrivi
ADD R1, j
LOAD R2, @R1
STORE R2, X
```

Copyright Teemu Kerola 2004

Kolmas vaihtoehto 2-ulotteisten taulukoiden tallettamiseen onkin aivan erilainen. Nyt ainoastaan kukin rivi talletetaan yhtenäiselle muistialueelle. Taulukon ensimmäinen taso on nyt ainoastaan osoitintaulukko, josta löytyy kunkin rivin talletusalue. Hyvinä puolina tässä menetelmässä on se, että taulukkoa varten ei tarvita yhtenäistä suurta muistialuetta. Toisena etuna on, että muistitilaa voidaan helposti säästää siten, että alustamattomille riveille ei edes varata muistitilaa. Niihin osoittava osoitin rivitaulukossa on tällöin esimerkiksi nolla, mikä tietenkin pitää tarkistaa jokaisen taulukkoviitteen yhteydessä.

Moniulotteisten taulukoiden talletus muistiin

Talletus rivi kerrallaan

- C, Pascal, Java

Talletus sarake kerrallaan

- Fortran

Talletus riveittäin rivitaulukkoon

- vain rivit yhtenäisenä lohkona

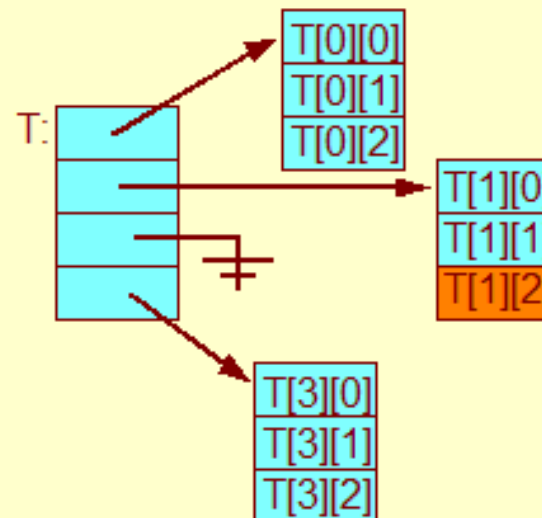
Entä 3-, 4-, tai useampi-ulotteiset taulukot?

- samalla tavalla...

T	0	1	2
0			
1			
2			
3			

$X = T[1][2]$

T:	T[0][0]	T[0][1]	T[0][2]	T[1][0]	T[1][1]	T[1][2]	T[2][0]	T[2][1]	T[2][2]	T[3][0]	T[3][1]	T[3][2]
T:	T[0][0]	T[1][0]	T[2][0]	T[3][0]	T[0][1]	T[1][1]	T[2][1]	T[3][1]	T[0][2]	T[1][2]	T[2][2]	T[3][2]



Copyright Teemu Kerola 2004

Entä jos taulukon ulottuvaisuuksia on enemmän? Ei siinä ole mitään erikoista. Laajennetaan vain tässä esitettyjä taulukoiden muistiinsijoitusmenetelmiä useampaan ulottuvuuteen. Tottakai taulukon alkion sijainnin paikallistava koodi on nyt vähän pidempi, mutta ei oikeastaan loogisesti yhtään monimutkaisempi.

Monimutkainen tietorakenne

Taulukko T, jonka jokainen alkio on 4-kenttäinen tietue

- pituus
- ikä
- taulukko, jossa palkka viimevuodelta joka kuukaudelta
- taulukko, jossa töissäolopäivien lkm viime vuodelta joka kuukaudelta

Talletustapa

- riveittäin, sarakettain, linkitetty rakenne

Viitteet

- laske talletustavan perusteella

Tarkistukset

- kaikki tarkistetaan, paljon koodia

T:

pituus
ikä
pal[1]
...
pal[12]
pv[1]
...
pv[12]
pituus
ikä
pal[1]
...
pal[12]
pv[1]
...
pv[12]
...

Copyright Teemu Kerola 2004

Entäpä, jos kyseessä on vielä paljon monimutkaisempi tietorakenne, sisältäen taulukoita ja tietueita sisäkkäin? Ei tämäkään tuota mitään erikoista ongelmaa, pikkasen vain mekaanista laskentaa lisää. Esimerkkinä on henkilörekisterin osataulukko T, jossa kukin alkio sisältää henkilön pituuden, iän, palkan ja töissäolopäivät viime vuoden kuukausilta. Tottakai rekisteriin kuuluu myös paljon muita tietoja, mutta katsellaan nyt tätä taulukkoa.

Monimutkainen tietorakenne

Taulukko T, jonka jokainen alkio on 4-kenttäinen tietue

- pituus
- ikä
- taulukko, jossa palkka viimevuodelta joka kuukaudelta
- taulukko, jossa töissäolopäivien lkm viime vuodelta joka kuukaudelta

Talletustapa

- riveittäin, sarakettain, linkitetty rakenne

Viitteet

- laske talletustavan perusteella

Tarkistukset

- kaikki tarkistetaan, paljon koodia

```
rec EQU 26
pit EQU 0
ikä EQU 1
pal EQU 1
pv EQU 13
```

T:

pituus

ikä

pal[1]

...

pal[12]

pv[1]

...

pv[12]

pituus

ikä

pal[1]

...

pal[12]

pv[1]

...

pv[12]

...

Monimutkainen tietorakenne

Taulukko T, jonka jokainen alkio on 4-kenttäinen tietue

- pituus
- ikä
- taulukko, jossa palkka viimevuodelta joka kuukaudelta
- taulukko, jossa töissäolopäivien lkm viime vuodelta joka kuukaudelta

Talletustapa

- riveittäin, sarakettain, linkitetty rakenne

Viitteet

$X = T[i].pal[j];$

- laske talletustavan perusteella

Tarkistukset

- kaikki tarkistetaan, paljon koodia

```
rec EQU 26
pit EQU 0
ikä EQU 1
pal EQU 1
pv EQU 13
```

```
LOAD R1, i
MUL R1, =rec
ADD R1, =T
ADD R1, =pal
ADD R1, j
LOAD R2, @R1
STORE R2, X
```

T:	pituus
	ikä
	pal[1]
	...
	pal[12]
	pv[1]
	...
	pv[12]
	pituus
	ikä
	pal[1]
	...
	pal[12]
	pv[1]
	...
	pv[12]
	...

Monimutkainen tietorakenne

Taulukko T, jonka jokainen alkio on 4-kenttäinen tietue

- pituus
- ikä
- taulukko, jossa palkka viimevuodelta joka kuukaudelta
- taulukko, jossa töissäolopäivien lkm viime vuodelta joka kuukaudelta

Talletustapa

- riveittäin, sarakettain, linkitetty rakenne

Viitteet

- laske talletustavan perusteella

Tarkistukset

- kaikki tarkistetaan, paljon koodia

pituus rajoissa?
ikä rajoissa?
molemmat kk-indeksit ok?

T:

pituus
ikä
pal[1]
...
pal[12]
pv[1]
...
pv[12]
pituus
ikä
pal[1]
...
pal[12]
pv[1]
...
pv[12]
...

Copyright Teemu Kerola 2004

Edelläolevasta puuttui tietenkin kaikki tarkistukset. Kääntäjän pitää jokaisen muistiviitteen yhteydessä varmistaa, että viite kohdistuu lailliseen muistipaikkaan. Tämän voi tehdä joko päättelemällä turvallisuus etukäteen käännösaikana, tai sitten generoimalla koodia, joka tarkistaa turvallisuuden suoritusaikana. Tarkistusten lisääminen lisää suoritettavan koodin määrää, mutta loogisesti on tietenkin aika yksinkertaista asiaa. Jotkut ohjelmointikielet (tai oikeastaan niiden kääntäjät) ovat parempia tarkistusten kanssa kuin jotkut toiset. Esimerkiksi, muuttujan 'ikä' arvoalue voisi olla 0..120, eikä vain kokonaisluku.

Konekielinen ohjelmointi

Muistitilan käyttö ja tiedon sijainti suoritusaikana

Muuttujat

Tietorakenteet: yksi- ja moniulotteiset taulukot, tietueet

Kontrolli: valinta, toisto

Koodin optimointi

Tarkistukset

Copyright Teemu Kerola 2004

Olemme nyt käyneet läpi konekielisen ohjelmoinnin perusteet. Älkään luulkokaan, että teillä olisi tämän jälkeen heti valmiuksia kirjoittaa todellisia sovelluksia konekielellä. Samalla tavalla kuin ohjelmointi korkean tason kielillä, myös konekielinen ohjelmointi vaatii käytännön harjoittelua. Tarkoituksena on pikkuhiljaa toteuttaa tällä luennolla opittuja asioita esimerkikoneemme konekielellä ja suorittaa harjoitustyöohjelmat ttk-91 simulaattorissa. Konekielisen ohjelmoinnin perusteista jäi vielä puuttumaan aliohjelmien toteutus, mikä käydään läpi seuraavalla luennolla.