Sormet C:hen

ITKA203 Käyttöjärjestelmät -kurssin Demo 2 kesällä 2007. Paavo Nieminen / Jyväskylän yliopiston Tietotekniikan laitos.

"Lähespikaintro C-kielellä ohjelmointiin kun esitietona on Java ja olio-ohjelmointi"

1 Mistä tässä harjoitteessa on kyse

Materiaali syntyi Käyttöjärjestelmät -kesäkurssilla 2007. Kiitän kurssilaisia palautteesta. Tästä piti tulla Superpikaintro, mutta tulikin vain Lähespikaintro. Kesäkurssin opiskelijoilla tämän tekemiseen kului käsittääkseni enimmillään noin kymmenisen tuntia. Tässä on aika paljon luettavaa ja hahmotettavaa. C-ohjelmointi on tällä hetkellä jotakin, jota Käyttöjärjestelmät -kurssin seuraaminen nykymuodossaan vaatii mutta jota ei vaadita esitietona. Implikaationa C-ohjelmointia on käytävä läpi osana tätä kurssia. Materiaalina on vähimmillään tämä lehdykkä.

Toinen huomio, joka on ikävä kyllä tehtävä, on että rakenteisen ohjelmoinnin edellyttämä imperatiivisten algoritmien kehittelykyky ei monellakaan ole vielä kohdillaan, vaikka muodollisesti asiat sisältävä Ohjelmointi 1 -kurssi olisi suoritettu kohtuullisilla tai jopa hyvillä arvosanoilla. On siis otettava huomioon, että tämän materiaalin läpikäynti voi kestää opiskelijasta riippuen tunnista (*läpiluku, tutuksi asiaksi toteaminen*) useisiin päiviin tai viikkoihin (*ohjelmointitaidon "kertaaminen", käsitteiden oppiminen, kattavamman oppikirjallisuuden etsiminen ja lukeminen, tukiopetustuokiot kavereiden kanssa tmv.*). Hyvänä puolena toivottavasti Käyttöjärjestelmät -sisällön lopputulema tältä osin olisi myös opiskelijoiden ohjelmointitaidollisten ja -ymmärryksellisten tasoerojen pieneneminen.

Huomioi seuraavaa:

- Käännökset ja kokeilut sekä pakollinen palautus tehdään shell-pääteyhteydellä THK:n koneessa `jalava.cc.jyu.fi`, joten tämä rakentuu suoraan demo 1:ssä opittujen perustaitojen päälle ja lisää niitä entisestään (komentojen ja argumenttien anto; ohjelmien tulosteiden tulkitseminen päätteeltä ym.).
- Tekstieditoinnin voi tehdä joko jollain Windowsin tekstieditorilla (sellaisella jossa on koodin väritys ym. eli esim. ConTEXT) tai suoraan THK:n koneessa `nano` -editorilla (elleipä hurjana halua opetella esim. emacsia tai vimiä käyttämään). Nanon tai muun käyttöä varten voi ottaa rinnakkaisen pääteyhteyden, niin koodi on aina eri ikkunassa kuin komentorivikäännökset ym.

- Ajankäytöllisesti osan tästä voi laskea kuuluvan harjoitustyöhön, jossa pitää toteuttaa yksinkertainen C-ohjelma, kääntää se assemblerille ja osoittaa ymmärtävänsä ohjelman toiminta konekielitasolla.

Toivon, että tämä materiaali olisi hyödyllinen myös tulevaisuudessa Käyttöjärjestelmät-kurssilla tarvittavan C-ohjelmoinnin ymmärtämisen apuna (ellei kurssista sitten poisteta C:n ymmärtämistä vaativia osioita, jolloin en tiedä mitä siihen jäisi jäljelle). Käyttöjärjestelmät tehdään monasti C-kielellä, ja jos niiden toteutusta tai edes käyttöä käyttöjärjestelmäkutsujen tasolla pitäisi osata lukea, kai vähän pitäisi tietää C-ohjelmoinnista.

Harjoituksen tavoitteet:

- Osaat siirtää aiemmin oppimiasi ohjelmoinnin perusrakenteita Java-kielestä C-kieleen, erityisesti:
  - Muuttujat
  - Ehtolauseet
  - Silmukat eli toistorakenteet
  - Aliohjelmakutsut (vastaa jotakuinkin luokkametodin kutsua)
- Tiedät, mitä ovat taulukot C-kielessä
- Tiedät, mitä ovat merkkijonot C-kielessä
- Tiedät, mitä ovat datakokoelmat eli "struktuurit" C-kielessä
- Tiedät, mitä ovat osoittimet ja miten niitä käytetään
- Ymmärrät näkökulmaeron: C on puhtaasti imperatiivinen kieli ja lisäksi laitteistoläheinen, eli toisin sanoen:
  - Olioita ei samassa mielessä ole (eli ei instanssimetodeja, ei perintää, ei sisäisen tilan piilotusta, ei rajapintoja samassa mielessä, ei poikkeuksia, ei roskienkeruuta, eikä muitakaan elämää helpottavia abstraktioita...)
  - Datakokoelmien rakenteet ja niitä käsittelevät aliohjelmat ovat erillisiä kokonaisuuksia eivätkä olioluokiksi paketoituja
- Ymmärrät eron aliohjelmakutsussa ja instanssimetodikutsussa

Rajauksia: Tässä opetellaan ISO-standardin C90 mukaista C-kieltä. Tämä on nykyinen "pienin yhteinen nimittäjä" eri C-kääntäjien välillä. Tässä siis **ei** tutustuta C99-standardiin, joka helpottaa ohjelmointia jonkin verran, mutta rajaa vanhoja ja eksoottisia kääntäjiä siirrettävyyden ulkopuolelle. Ja todellakin, kieli on C eikä siis missään tapauksessa olio-ohjelmointiin soveltuva **C++**! C++ on oma paljon laajempi kielensä, vaikka C++ -kääntäjät osaavat kääntää standardin mukaiset C-ohjelmat. C-kieli on siis C++:n osajoukko (joka osaltaan aiheuttaa C++:aan tietyn, hankalan, historiallisen taakan).

Kaikkia ominaisuuksia edes C-kielestä ei käydä läpi, vaan koetetaan nopealla *hands-on* -kokemuksella saada hahmottumaan, miten lyhyitä C-koodin pätkiä ymmärretään tai tehdään. Esitieto-oletus on olio-ohjelmoinnin perustaito erityisesti Java-kielellä, johon C:tä tässä kautta linjan vertaillaan.

Sisältö

2 Ohjeita

Viime demoissa teit hakemistot kaikille Käyttöjärjestelmät-kurssin harjoitteille. Ota yhteys THK:n koneeseen `jalava.cc.jyu.fi` ja aseta työhakemistoksesi kakkosdemon hakemisto, nimeltään esim. `~/kj07kesa/demo2/`. Muistele tarvittaessa viime kertaa: miten hakemisto vaihdettiin, ja jos oli epävarmuutta siitä missä ollaan, niin miten nykyinen työhakemisto tarkistettiin jne. ...

Kun olet demo 2:n hakemistossa, hae esimerkkikoodit kurssin nettisivulta, eli komenna shellissä:

```
wget http://www.cc.jyu.fi/~nieminen/kj07kesa/demo2.zip
```

ja pura samaan paikkaan:

```
unzip demo2.zip
```

Sait kasan koodeja. Näiden demojen tekeminen perustuu esimerkkien ja demomateriaalin lukemiseen sekä puoliksi tehtyjen C-ohjelmien täydentämiseen opitun perusteella. Palautustehtävänä lähetät tietyn muotoisena sähköpostina kaikki ne täydentämäsi/korjaamasi koodit. Automaattinen tarkistus pyrkii testaamaan, että koodit on oikealla tavoin täydennetty/korjattu. Tällä kertaa en täydellisesti pysty toteamaan, että olet itse tehnyt tehtävät, eikä kaveri -- jokainen on oman onnensa nojassa, ja "teknisesti sallittu" laiskuus on jokaisen oma **häpeä**. Näillä sanoin työn touhuun...

3 Ensimmäinen C-ohjelma

Katso tekstieditorilla ohjelmaa `helloworld.c` -- siitä nähdään yksinkertaisen C-ohjelman muoto. Tällainenhan se on (rivinumerot lisätty tulosteeseen):

```
1  #include <stdio.h>
2  int main(int argc, char **argv){
3      printf("Hello World!\n");
4      printf("Tama on C-ohjelma!\n");
5      return 0;
6  }
```

Selitetään merkitys rivi riviltä, ja peilataan Javasta tuttuihin asioihin. Syntaksiin mennään tarkemmin myöhemmin; nyt mietitään vain rivien merkitystä ohjelman kannalta.

Rivi 1:

```
#include <stdio.h>
```

Tällä otetaan mukaan määrittelyt tiedostosta `stdio.h` (*Standard input/output*). Kyseessä on C-kielen apukirjasto, jonka avulla toteutetaan kirjoitus- ja lukuoperaatioita. Tässä ohjelmassa tulostetaan `printf()` -aliohjelmalla, jonka käyttö edellyttää nimenomaan kirjaston `stdio.h` ottamista mukaan. Vastine Javassa on `import` -avainsana, jolla lähdekooditiedoston alussa otetaan mukaan tarvittavat apukirjastot. Samoin kuin Java-lähdekoodien alussa on paljon `import` -rivejä, C-ohjelmien alussa on usein paljon `#include` -rivejä.

Rivi 2:

```
int main(int argc, char **argv){
```

Tästä alkaa C-ohjelman suoritus. On sovittu, että ohjelman ensimmäinen suoritettava käsky on ensimmäinen käsky sellaisesta aliohjelmasta, jossa:

- nimi on `main`
- paluuarvo on `int` eli kokonaisluku
- on kaksi parametria, joista ensimmäinen on `int`-tyyppinen ja toinen on `char **argv` eli osoitin merkkijonotaulukkoon (älä vielä huoli tietotyypeistä, ne ovat myöhemmän tehtävän asia).

Tämä on ihan samanlainen sopimusasia kuin se, että suoritettavan Java-luokan ensimmäinen käsky on seuraavanlaisen luokkametodin ensimmäinen käsky:

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

Huomataan, että C, kuten Javakin, on lohkorakenteinen, ja lohkojen syntaksikin on sama: lohko alkaa kiharasululla `{` ja päättyy vastaavaan käänteiseen sulkuun `}`. Muutenkin syntaksissa on yhtäläisyyttä: C:n aliohjelmat aloitetaan muodossa:

```
PALUUARVON_TYYPPI aliohjelmanNimi(TYYPITETTY_PARAMETRILISTA)
```

joka on täysin sama kuin Javan metodimäärittelyn syntaksi (varmasti ainakin sen takia, että Java on tarkoituksella tehty samannäköiseksi kuin C ja C++). Sekä C:ssä että javassa aliohjelmamäärityksen eteen laitetaan tarvittaessa lisämääreitä.

Rivit 3 ja 4:

```
printf("Hello World!\n");
printf("Tama on C-ohjelma!\n");
```

Tällainen on C:ssä aliohjelmakutsun syntaksi. Eikö näytäkin samalta kuin Javassa metodin käyttäminen? Paitsi että usein Javassa kutsutaan pistenotaatiota käyttämällä jonkun

olion instanssimetodia; esimerkiksi Java-kutsussa `System.out.println("juu")` kutsutaan `java.lang.System` -luokan luokka-attribuuttina löytyvän `out`-nimisen `PrintStream`-luokan instanssin metodia `println`.

(C:n tulostus on vähän lyhyempi selittää täsmällisesti kuin Javan luokkahässäkät: esimerkin rivi 3 siirtää ohjelman suorituksen aliohjelman nimeltä `printf` alkuun, välittäen sille parametriksi merkkijonovakion.)

Huomaa, että C:n lauseet tulee päättää puolipisteellä `;` ihan niinkuin Javassakin.

Rivi 5:

```
return 0;
```

Ohjelmat voivat kertoa operaation onnistumisesta tai epäonnistumisesta virhekoodilla. Se on kokonaisluku, jonka ohjelman käynnistäjä saa haltuunsa. C-kielessä koodi annetaan `main()` -funktion paluuarvona, eli tuon arvon asettaminen on viimeinen asia, minkä käyttäjän ohjelma suorittaa. Yleensä 0 tarkoittaa, että mitään virhettä ei tullut.

Rivi 6:

```
}
```

Kun aliohjelman sisällön määrittelevä lohko avataan ohjelman alussa, niin toki se pitää lopuksi sulkea. Ihan niinkuin Javassa.

Vastaavaa syntaktista yhtäläisyyttä tulet näkemään kautta linjan. Syy tosiaan on että Java on tarkoituksella C:n näköistä. Erot ovat siis enemmän käsitteellisiä ja toiminnallisia kuin kieliopillisia.

Käännä ja testaa ohjelmaa Jalavassa shell-yhteydellä. Komenna:

```
gcc helloworld.c
```

Jos kaikki meni hyvin, ohjelma kääntyi oletusnimelle `a.out`, jonka voit nyt suorittaa komennolla:

```
./a.out
```

Toivottavasti tulostui se, mitä odotitkin. Huomaa, että nykyisestä työhakemistosta ei koskaan etsitä ajettavia tiedostoja automaattisesti, koska se voisi olla vaarallista... Ajaaksesi jotakin työhakemistosta, pitää kertoa eksplisiittisesti, että haluat ajaa ohjelman *tästä* hakemistosta eli `./` eli piste ja kauttaviiva.

Tehtävä: Muuta ohjelma tulostamaan "`Hello ktunnus`", missä `ktunnus` on unix-käyttäjätunnuksesi. Tallenna samalle nimelle eli `helloworld.c`, käännä uudelleen ja testaa että toimii.

4 Komentoriviargumentit C:ssä ja Javassa

Jotta yliopittaisiin myös shell-komentojen argumenttien periaate, kokeillaan pääsyä argumentteihin C-kielessä ja Javassa. Samalla nähdään, miten tietyt syntaksit voivat olla identtisiä C:ssä ja Javassa.

Tutustu simppeliin Java-ohjelmaan `Argumentit.java`:

```
public class Argumentit{
    public static void main(String[] args){
        int i;
        System.out.printf("Ohjelman saamat komentoriviargumen-
tit ovat:\n");
        for(i=0; i<args.length; i++){
            System.out.printf("Argumentti %d: %s\n", i, args[i]);
        }
    }
}
```

ja sen C-vastineeseen `argumentit.c`:

```
#include <stdio.h>
int main(int argc, char **argv){
    int i;
    printf("Ohjelman saamat komentoriviargumentit ovat:\n");
    for (i=0; i<argc; i++){
        printf("Argumentti %d: %s\n", i, argv[i]);
    }
}
```

Käännä ohjelmat komentamalla:

```
javac Argumentit.java

gcc argumentit.c
```

Totea että syntyi käännetyt tiedostot `Argumentit.class` ja `a.out` Aja ne esim. komentamalla:

```
java Argumentit kissa koira

./a.out kissa koira
```

Tarkastele tulosteita ja tee pari kokeilua argumenteilla. Huomioita:

- Tavukoodiksi käännetyn Java-ohjelman ajamista varten käynnistetään itse asiassa `java` -niminen virtuaalikoneohjelma, jonka ensimmäisen argumentin pitää olla ajettavan luokan nimi (ilman tarkennetta `.class`); kaikki loput argumentit välittyvät suoritettavalle Java-ohjelmalle `main`-metodin taulukkoparametriksi.
- C-käännös on suoraan ajettavissa x86-64:ssä, joten se käynnistetään suoraan ja kaikki argumentit annetaan ohjelmalle.
- C-käännös itse asiassa saa ensimmäisenä argumenttina (indeksi 0) ohjelman nimen siten kuin käyttäjä sen kirjoitti. Javassahan tuli vain argumentit eikä ohjelman nimeä.
- Java-dokumentaatio sanoo, että tällainen komentoriviargumenttien syöttäminen "ei ole 100% Javaa", vaikka pääohjelman parametrilistan sopimisesta voisi niin päätellä. Kumma juttu...

Alustavia huomioita syntaksista:

- `for`-silmukan syntaksi on tässä tapauksessa täysin sama molemmissa kielissä
- Javassa on *jo niinkin aikaisin* kuin sen lähivuosien versiossa "Java 2" toteutettu `PrintStream`-luokkaan formatoidun tulostuksen hoitava metodi `printf()`, joka tekee samalla syntaksilla samat kätevät asiat kuin C-kielessä on saatu tehtyä standardilla `printf()`-aliohjelmalla 30 vuotta. Tämä on hauska huomio siitä, miten ympäri mennään ja yhteen tullaan, kun kieliä ja abstraktioita kehitetään (Javassa kompaktin tulostussyntaksin toteuttaminen vaati mm. ns. Autoboxing-ominaisuuden, jota ei ennen Java 2:ta ollut).
- Käytännön erot syntaksissa olivat siis tässä tapauksessa pieniä (eikä suuria ole odotettavissakaan, vaikka havaittaneen pian, että C on toisaalta rajoitetumpi ja toisaalta kuitenkin osin "kryptisempi" kuin Java.)

Tähän astisen tarkoitus oli saada mahdollinen "C-pelkokerroin" putoamaan välittömästi. Kyseessä on suurelta osin tuttu asia, jos osaat jonkin verran ohjelmoida Javalla. Seuraavaksi syvennytään joihinkin yksityiskohtiin, jotka ovat erilaisia.

5 Tyypit, muuttujat ja osoittimet, muistimalli

Tyypit, muuttujat, muistimalli, viite/osoitin, ... nämä ovat ohjelmoinnin yleisiä käsitteitä, jotka on ymmärrettävä abstraktisti, että pystyy ohjelmoimaan erilaisilla kielillä, joissa kukin käsite toteutuu nyansseiltaan hieman eri tavoin.

5.1 Kertaus: mikä olikaan muuttuja

Ohjelmoinnissa voi yleensä määritellä muuttujia. Niillä on nimet, joiden kautta niihin pääsee käsiksi, niihin voi tallentaa dataa operaatioiden suorittamista varten, ja niitä voi käyttää

lausekkeissa. Vahvasti tyypitetyissä kielissä, jollaisia sekä Java että C ovat, muuttujien käyttöön liittyy rajoituksia: mm. muuttujat pitää esitellä aina tietyn tyyppisiksi, eikä tyyppiä voi enää esittelyn jälkeen muuttaa. Muuttujien tyypit on tunnettava jo ohjelmaa käännettäessä joka tilanteessa.

Javassa muuttujat jakautuvat **primitiivityyppisiin** (`int`, `double`, `boolean` ja niin edelleen) ja **olioviitteisiin**, joiden tyyppiin kuuluu tieto siitä, minkä luokan (tai tästä perityn aliluokan) olioon viite vain voi osoittaa. Muuttujat eivät voi olla Javassa muunlaisia, vaan pelikenttä on siellä pohjimmiltaan näinkin yksinkertainen. Taulukot ja merkkijonot ovat olioita, joita käytetään vastaavantyyppisten viitteiden kautta. Myös C-kielessä on tietyt primitiivityyppit, sitten C:ssä on ohjelmoijan määrittelemiä **struktuurityyppejä** (kokoelmia muista tyypeistä, vähän niinkuin luokkia joilla on vain julkisia attribuutteja eikä yhtään metodia; struktuurityypeissä voi olla sisällä muita struktuureja, eli rakenne voi olla hierarkkinen), **taulukkotyyppi** kustakin primitiivityypistä (tai samantyyppisistä struktuureista tai taulukoista), ja lisäksi on **osoittimia** edellä mainittuihin.

Muistimalli tarkoittaa sitä, miten muuttujien ja muiden objektien (oliot tai tietorakenteet) ajatellaan sijoittuvan ohjelman suorituksen aikana. Esimerkiksi Javassa olioattribuutit sijoittuvat kekomuistiin omien olioinstanssiensa osana. Kekomuistista varataan tilaa attribuuteille aina kun olio luodaan eli instantoidaan. Metodien parametrit sekä niiden lokaalit muuttujat sen sijaan sijoittuvat pinomuistiin, jota tavukoodin virtuaalikonekäskyt käyttävät operaatioissa. Olioviitteet, olivatpa ne lokaaleja muuttujia tai attribuutteja, ovat olennaisesti jonkun kekomuistissa olevan olion osoite (Javan viite siinä mielessä on "osoitin") tai `null`, ts. ei viittaa mihinkään juuri tällä hetkellä.

C-kielen muistimalli on laitteistoläheinen. Muuttujilla ja osoittimilla on yksi-yhteen vastaavuus laitteistossa suoritettavan prosessin virtuaalimuistin kanssa: Lokaalit tietorakenteet (primitiivimuuttujat ja muunkin tyyppiset lokaalit muuttujat) sijaitsevat suorituspinossa kulloisenkin aliohjelman pinokehyksessä (ks. luennot ja materiaali joka kertoo suorituspinosta ja pinokehyksestä), globaalit rakenteet (käsittääkseni) data-alueella ja dynaamisesti varatut tietorakenteet ovat niille varatulla muistialueella.

C:n primitiivityyppejä käytetään kuten Javassa, eli niiden arvoja voidaan käyttää lausekkeissa. Kääntäjä osaa tuottaa ohjelmakoodin, joka sisältää lokaalien muuttujien oikeat muistiosoitteet suhteessa kullakin hetkellä suorituksessa olevan aliohjelma-aktivaation pinokehykseen (ks. luennot ja materiaali joka kertoo suorituspinosta). Tämän lisäksi mihin tahansa muuttujaan voidaan tarvittaessa tehdä osoitin, joka on konkreettisesti alla olevan prosessoriarkkitehtuurin virtuaalimuistiosoite. Tässä koodissa tehdään osoitin kokonaislukuun:

```
int* tyhma_aliohjelma(){
  /* (Vanhan standardin mukaisessa C:ssä kaikki muuttujat on
   * esiteltävä ennen aliohjelman muuta koodia.)
   */

  int primi;
  int *osoitin;

  /* Muuttujan nimeltä primi arvo sijaitsee pinokehyksestä sille
```

```
     * varatussa paikassa. Samoin muistiosoite nimeltä osoitin.
     * C-kielessä ei luvata kummallekaan mitään tiettyä alkuarvoa.
     * Tässä vaiheessa voivat olla siis mitä tahansa.
     */

    /* Sijoitetaan seuraavassa muuttujaan vakioluku 123;
     * käytännössä pinomuistin vastaavaan kohtaan
     * menee silloin lukuarvo 123:
     */

    primi = 123;

    /* Osoitinkin on lukuarvo pinossa, mutta sen merkitys on olla
     * jonkun toisen muuttujan muistiosoite; tässä tapauksessa
     * sijoitetaan osoittimen pinokehyspaikkaan primin
     * pinopaikan muistiosoite:
     */

    osoitin = &primi;

    /* Funtsi juttua, ja piirrä tarvittaessa muistin sisällöstä kuvia
     * paperille, ja kysy jos ei muuten hahmotu!
     * Harjoitustyömateriaalissa on tästä lisää tietoa.
     */

    /* Alla oleva on tässä tapauksessa JÄRKYTTÄVÄÄ JA VÄÄRIN
     * koska aliohjelmasta palautetaan muistiosoite, joka viittaa
     * nykyisen pinokehyksen sisään. Tämä kehys lakkaa olemasta
     * returnissa, mutta joku ehkä kuvittelee että palautettu
     * muistiosoite tarkoittaisi aliohjelma-aktivaation
     * päätyttyäkin jotain järkevää. Ei tarkoita. Se on
     * virtuaalimuistiosoite; se on numero; sen osoittaman
     * muistipaikan sovittu käyttötarkoitus vaihtelee, ja
     * sen käyttö paikallisen primi -nimisen muuttujan säilömiseen
     * loppuu heti tämän returnin suoritukseen:
     */

    return osoitin;
}
```

Eli C:ssä käsitellään asioita hyvin laiteläheisesti. On helppo tehdä erittäin pahoja ohjelmointivirheitä osoittimien kanssa. Kääntäjä ei osaa varoittaa loogisesti väärin käytetyistä osoittimista, ja ajonaikana varsinkaan ei ole laitteiston ja C-kielestä konekielelle käännetyn ohjelman välissä mitään tuplavarmistuksia kuten JVM Javassa tarjoaa.

(Täytyy muistaa, että koodin optimointi voi vähän sekavoittaa kuviota yllä kuvatusta; muuttujan arvoa voidaan nimittäin pitää prosessorin rekisterissä, jolloin se on paljon nopeammin

saatavilla laskutoimituksiin kuin pinomuistista).

Osoittimet ovat tavallaan ikään kuin viitteet Javassa, mutta olioitahan C-kielessä ei ole, vaan osoitin osoittaa muistipaikkaa. Osoitetussa muistipaikassa voi olla yksi tietyn tyyppinen primitiivialkio, tai yhtä hyvin siitä voi alkaa muistialue, joka sisältää vaikka miten monimutkaisen ja ison tietorakenteen.

Maistelepa huvikseen seuraavia sanontoja, joita voisi ihminen päästää suustaan:

- "Viitemuuttuja osoittaa olioinstanssiin"
- "Osoitinmuuttuja viittaa tietorakenteeseen"
- "Viite olioon"
- "Osoitin tietorakenteeseen".

5.2 Primitiivityypit

Javassa virtuaalikoneen standardi kertoo, minkä kokoisia (biteissä) mitkäkin primitiivityypit ovat. Viitemuuttujia ei Javassa voi käyttää laskemiseen, joten niiden sisäisellä toteutustavalla tai pituudella ei ole mitään väliä. C:ssä on toteutettu ns. **osoitinaritmetiikka** eli osoittimeen voidaan lisätä ja vähentää lukuja: Muistiosoitteethan ovat aina yhden tavun osoitteita, joten esim. jos taulukossa on 4 tavun mittaisia eli 32-bittisiä lukuja, niin aina seuraavan alkion osoittamiseksi pitäisi lisätä muistiosoitteeseen nelonen. Taas jos taulukossa on 64-bittisiä asioita, niin pitäisi lisätä kahdeksan. Taas jos taulukossa on 1234 tavun mittaisia merkkijonopuskureita, niin ilmeisesti pitää lisätä 1234 että osoitin päätyy osoittamaan seuraavaa alkiota. Koska C-kääntäjä tietää, minkä tyyppiseen asiaan osoitinmuuttuja osoittaa, niin se osaa tulkita esim. koodin `osoitin++` siten että osoitetaan seuraavan samantyyppisen alkion ensimmäistä tavua; ei siis lisätä ykköstä vaan tyypin koko tavuissa.

Seuraavassa luetellaan C-kielen primitiivityypit ja esimerkki, minkä kokoisia ne voisivat olla. C-kielen primitiivityyppien ja osoittimien koko riippuu prosessoriarkkitehtuurista, jolle kääntäjä on tehty! Standardi määrittelee vain minimipituudet. Eli mm. tätä pitää varoa, jos aikoo tehdä siirrettävän C-ohjelman! (Tämä kohta voi olla vähän hätäseen kirjoitettu. Tarkista varmuuden vuoksi itse vaikka C90 -standardista...)

Kokonaisluvut:

```
unsigned char             - etumerkitön 8-bittinen luku
signed char               - etumerkillinen 8-bittinen luku
char                      - 8-
bittinen luku (riippuu kääntäjästä onko
                            etumerkillinen vai ei; eli siirrettäväs-
sä
                            koodissa syytä sanoa eksplisiittisesti)

unsigned short int        - 16 bittinen luku, etumerkilliset ja -
merkittömät
```

```
signed short int
short int

       (voi kirjoittaa myös "unsigned short", "sig-
ned short", "short")

unsigned int              -
16 tai 32 bittinen tai joku muu sellainen luku,
signed int                  tarkistettava aina kääntäjän speksistä.
int                         Jalavan GCC:ssä ilmeisesti 32-bittisiä

       (voi kirjoittaa myös "unsigned", "signed")

unsigned long int         - 32 tai jotain bittinen tjsp.
signed long int             tarkistettava aina kääntäjän speksistä,
long int

       (nämäkin voi kirjoittaa ilman int'iä)

unsigned long long int    - 64 tai jotain bittinen tms,
signed long long int        tarkistettava aina kääntäjän speksistä,
long long int

       (nämäkin voi kirjoittaa ilman int'iä)
```

Liukuluvut:

```
float                      - single precision

double                     - double precision

long double                - extended double precision
```

Ei-mitään -tyyppi:

```
void                       - "ei mitään"; käytettävä aliohjelman
                             esittelyssä, jos paluuarvoa tai para-
metreja
                             ei ole. Voi myös tehdä void-tyyppisiä
                             osoittimia, millä voi kiertää osoitti-
men
                             vahvan tyypityksen; mihin tahansa ob-
jektiin
                             voi osoittaa void-osoittimella.
```

Huomioita:

- "Merkki" eli `char` on luku; sillä voi ajatella koodaavansa ASCII-merkin tai minkä tahansa muun asian, joka numeroituu välille 0..255 (jos `char` on 8 bittiä) Merkeillä voi laskea yhteen tai vähentää, ne kun on vaan vähäbittisiä lukuja...
- Javasta tuttua `boolean` -totuusarvotyyppiä ei ole C:ssä olemassa (paitsi uudemmassa standardissa). Lukuarvot (myös muistiosoitteet eli pointterit) tulkitaan olevan tosia, jos ne eivät ole nolla, ja epätosia, jos arvo on nolla. Vertailuoperaattoreiden käyttö loogisessa lausekkeessa koodaa epätoden kokonaisluvulla 0 ja toden kokonaisluvulla 1. Näillä mennään.

5.3 Tietuetyypit eli ohjelmoijan määrittelemät tyypit

Lokaaleille primitiivityypeille varataan tilaa pinosta tai data-alueelta sen verran kuin ne tarvitsevat. C-ohjelmoija voi koostaa primitiivityypeistä isompia ns. tietueita eli struktuureja, jotka ilmenevät muistitilana kaikille jäsenilleen. Esim. seuraavaa tyyppiä voisi käyttää pistemäisen massan kuvaamiseen:

```
struct pistemassa3d {
    double x, y, z;
    double massa;
    int idnumero;
}
```

Käyttöesim:

```
pistemassa3d kappaleA;    /* kappaleA olisi pistemassa3d -
tyyppinen*/

kappaleA.y = -3.0;        /* jäseniin pääsisi käsiksi näin*/
kappaleA.massa = 48.0;    /* jäseniin pääsisi käsiksi näin*/

...
```

Yllä sijoitus `kappaleA.y = -3.0` asettaisi liukuluvun -3.0 muistipaikkaan, joka on varattu kyseiselle tietuekentälle. Kenttä sijaitsee tietueelle varatun kokonaistilan sisällä. Kääntäjä pitää kirjaa siitä, mikä y-kentän osoite on suhteessa kokonaisuuteen. Ohjelmoijalle päin tämä näyttää periaatteessa samalta kuin Javassa luokka, jossa on vain julkisia attribuutteja eikä yhtään metodia. Kuitenkin käsitteellisiä eroja on: C-kielessä struktuuri syntyy lokaaliin pinoon eikä mihinkään eri paikkaan (Javassahan olioiden datat ovat kekomuistissa ja paikallisessa pinossa on vain viite). Ja yllä esimerkissä `kappaleA` ei ole millään tavoin viite tai pointteri, vaan se tarkoittaa koko datakönttää. Siis sijoitus:

```
pistemassa3d kappaleA, kappaleB;
```

```
...

kappaleB = kappaleA;
```

aiheuttaisi koko `kappaleA` -tietueen sisällön siirtämisen kenttä kentältä `kappaleB` -tietueen vastaaviin kenttiin (käytännössä tapahtuisi tavujen kopiointi muistissa paikasta toiseen). Jotta turhilta siirroilta vältyttäisiin, ja jotta voitaisiin käyttää dynaamista muistinvarausta C:ssä, tarvitaan osoittimia.

5.4 Osoitintyypit

Mihin tahansa muuttujaan voidaan viitata osoittimella, eli voidaan esitellä vastaavan tyyppinen osoitinmuuttuja, johon voidaan sijoittaa viitattavan muuttujan muistiosoite. Lisäksi, koska osoitinkin on olennaisesti muuttuja, voidaan siihen tehdä osoitin! Siis seuraavanlaiset muuttujat ovat C:ssä yleisiä:

```
double luku;           /* Primitiivityyppi, liukuluku nimeltä 'lu-
ku'*/
double *pluku;         /* osoitintyyppi! Liukuluvun muistiosoite. */
double **ppluku;       /* osoitintyyppi: Liukuluvun osoitteen osoi-
te */
double ***pppluku;     /* edelleen liukuluvun osoitteen osoit-
teen osoite */

/* jne... eli muistiosoituksen "epäsuoruuden" asteelle ei ole rajoi-
tusta.*/

char **argv;  /* Osoitin joka osoittaa osoittimeen joka osoit-
taa chariin */
char *argv[]; /* Itseas. sama asia! Osoitin joka osoittaa char-taulukkoon */
```

Huomaa, että tyypin syntaksissa osoitin merkitään tähdellä `*`, joka **edeltää välittömästi sen muuttujan nimeä, josta halutaan tehdä osoitin eikä primitiivi**. Huomaa, että vaikka kaikki osoittimet ovat samanlaisia (eli muistiosoitteita, osoiteväylän leveyden kokoisia bittijonoja), ne ovat C-kielessä tarkkaan tyypitettyjä: `double`-osoittimella ei voi osoittaa vahingossakaan esim. `int`-muuttujaan (Paitsi jos kirjoitetaan kyseinen muunnos tyypistä toiseen eksplisiittisesti eräällä syntaksilla! eli ohjelmoijalla on jälleen täysi kontrolli kaikesta ja vastuu oikeellisen ohjelman tekemisestä. Ei turvaverkkoja.)

5.5 Taulukon toteutus C:ssä

Javassa taulukot ovat käytön kannalta tavallisia olioita (`Object` -yliluokan rajapinta, metodit ja kaikki), vaikka niille on kääntäjän tasolla syntaksisokeria, ja kaiketi tehokas sisäinen

toteutustapa. Silloin on olio-ohjelmoinnin kaikki mukavuudet ja herkut käytössä, mm. taulukon pituuden saa attribuutista `taulukko.length` jne. C-kielessä mitään tällaista ei ole -- olioabstraktiota kun ei tueta kielen ominaisuuksilla.

C-kielessä taulukko on muistialue, joka sijaitsee peräkkäisissä, yhden alkion mittaisissa muistipaikoissa (varaustavasta riippuen prosessin data-alueella, dynaamisella alueella tai pinossa). Mitään muuta se ei ole. Viiden 32-bittisen luvun taulukko nimeltä `taul` olisi seuraavanlainen:

```
.                    +-------------------+
Muistipaikka  N+24   | Jotain ihan muuta!|
                     |                   |
Muistipaikka  N+20   |                   |
                     +-------------------+
Muistipaikka  N+16   | taul[4] (4 tavua) |
                     +-------------------+
Muistipaikka  N+12   | taul[3] (4 tavua) |
                     +-------------------+
Muistipaikka  N+8    | taul[2] (4 tavua) |
                     +-------------------+
Muistipaikka  N+4    | taul[1] (4 tavua) |
                     +-------------------+
Muistipaikka  N      | taul[0] (4 tavua) |
                     +-------------------+
Muistipaikka  N-4    | Jotain ihan muuta!|
                     |                   |
Muistipaikka  N-8    |                   |
                     +-------------------+

HUOM: Tässä muistipaikan numero N == taul == &taul[0]
      eli C:ssä taulukkomuuttuja ja osoitin ensimmäiseen
      alkioon ovat samaistettavissa.
```

Mitään muuta taulukko ei C:ssä ole kuin varattua muistia. Taulukkoon viitataan aina tavallaan muistiosoitteen avulla. Syntaksi vain näyttää kätevältä:

```
oso = taul;      /* oso saisi arvokseen alkion taul[0] muistiosoit-
teen! */

aa  = taul[2];   /* Kääntäjä laskisi valmiiksi alkion taul[2]
                    muistipaikan, ja aa saisi siellä sijaitsevan
                    lukuarvon */

paa = &taul[2]   /* Tässä kääntäjä laskisi alkion taul[2] muistipai-
kan,
                    ja nimenomaan se muistipaikka eli osoite
                    laitettaisiin muuttujaan paa, jonka pitäisi
```

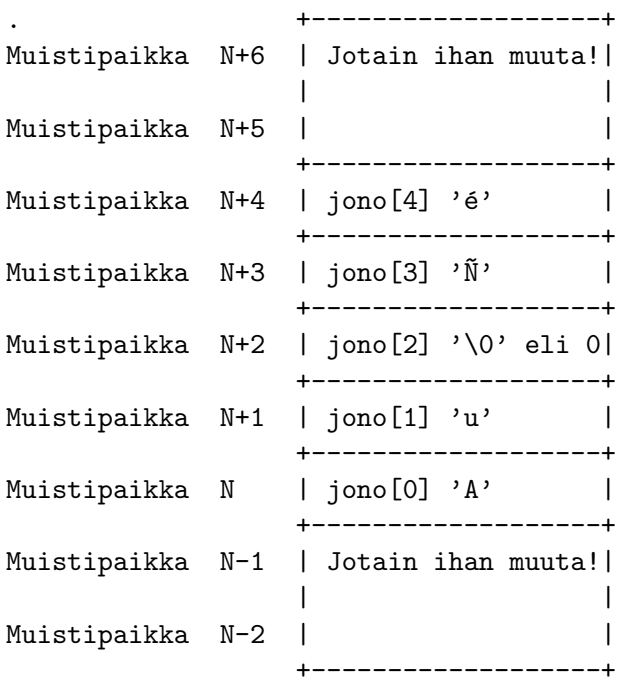
```
                    olla tyypiltään osoitin samantyyppiseen tietoon,
                    kuin mitä taulukko sisältää */
```

HUOMAA: Kukaan ei kerro ajonaikaisesti, minkä verran taulukolle on varattu muistista tilaa! Taulukon käyttö C-kielessä edellyttää aina sitä, että ohjelmoija pitää ihan itse kirjaa taulukoiden koosta jossain muuttujassa tai vakiossa. Esimerkiksi aina kun aliohjelmalle annetaan parametrina taulukko (eli ensimmäisen alkion muistiosoite), se tarvitsee tiedon myös siitä, mihin taulukko päättyy, eli missä on viimeinen käsiteltävä alkio.

5.6 Merkkijonon toteutus C:ssä

Javassa merkkijonot voidaan tehdä esim. `String` tai `StringBuffer` -luokkien instansseina. Tässäkin on olio-ohjelmoinnin mukavuudet ja herkut käytössä, mm. Java-merkkijonon pituuden saa tietää olion rajapinnan kautta, esim. `"kissa".length()`. Muuttuvaisella jonolla eli `StringBuffer` -luokan oliolla on mahdollisuus pidentyä ja lyhentyä dynaamisesti metodien suorituksen yhteydessä. C-kielessä mitään tällaista ei ole - ei ole olioita rajapintoineen.

C-kielessä merkkijono sijaitsee taulukossa, johon on varattu tilaa `char`-tyyppisille muuttujille vähintään merkkijonon merkkien verran plus yksi. Plus yksi sen takia, että C:ssä merkkijonon loppu pitää ilmoittaa "nollamerkillä", siis `char`-tyyppisellä luvulla 0. Tällainen voisi esimerkiksi olla merkkijonon `"Au"` sijoittuminen muistiin:



Eli jonon alku on jossain muistipaikassa ja jonolle on varattu tilaa viiden merkin verran, tässä tapauksessa merkin pituus on yksi tavu. Koska jono on "Au" eli siinä on merkit 'A' ja 'u', ne ovat

vastaavissa paikoissa taulukkoa. Niiden jälkeen on nolla, joka kertoo, että jono päättyy siihen. Muilla taulukon arvoilla ei ole merkitystä, koska niitä ei tulkita kuuluvaksi merkkijonoon.

Sanotaan, että varattu muistitila on **merkkijonopuskuri** (*string buffer*), johon mahtuu nollamerkkikoodauksen takia korkeintaan "puskurin koko - 1" merkkiä pitkä jono. Jos vahingossa puskuriin sijoitettaisiin esimerkiksi lukuoperaatiolla enemmän merkkejä kuin sinne mahtuu, olisi kyseessä "puskurin ohikirjoitus" eli **Buffer overrun/overflow**, joka on historiallisesti erittäin suosittu tapa murtautua tietojärjestelmiin -- jos kirjoitetaan merkkejä (eli tavuja) sopivasti yli puskurialueen, saatetaan päästä kirjoittamaan niitä jopa muistiosoitteeseen, jossa olisi suoritettavaa koodia. Sinne voisi kirjoittaa merkeillä mitä tahansa konekieltä, ja saada tietokone tekemään ihan mitä itse haluaa. Ihan vain vastaamalla ohjelman kysymykseen "Who are you?", jos sen tekijä ei osannut ohjelmoida C:llä turvallisesti! Onneksi prosessorien muistinsuojaus nykyään jonkun verran auttaa... ajettavan koodialueen virtuaaliosoitteet voivat olla kirjoitussuojattuja, jolloin ohjelma kaatuu ns. suojausvirheeseen, jos joku konekielikäsky yrittää sijoittaa koodialueelle. Toista oli ennenvanhaan, kun suojaus prosessori- ja käyttöjärjestelmäteknologian puolesta oli alkeellisempaa.

Mutta varovaisuudesta ei saa ikinä tinkiä: C-ohjelman, joka lukee merkkejä yhtään mistään, TÄYTYY olla toteutettu siten, että varattu puskurialue ei missään nimessä ylity! Mitä tämä taas edellyttää? Sitä, että ohjelmoija pitää kirjaa puskurille varatusta tilasta, vaikkapa jossakin muuttujassa, ja käyttää sellaisia algoritmeja, jotka hyödyntävät tuon tiedon. Sama asia siis kuin muidenkin taulukoiden yhteydessä.

5.7 Muita tyyppejä

C:ssä on pari muutakin eksoottista tyyppiä, jotka jätetään opiskeltavaksi tarkemmin muusta lähteestä:

- `enum` eli lueteltu tyyppi:

  ```
  enum {APPELSIINI, OMENA, PAARYNA};
  ```

  määrittelisi lukuarvot APPELSIINI==1, OMENA==2, PAARYNA==3; voi käyttää tilojen koodaamiseen.
- `union` tyyppi, joka itse asiassa vaihtaa tyyppiä sijoituksen mukaan; varmaan aiheuttanut monta sekaannusta ja vaikeaselkoisuutta ajan mittaan, veikkaan.

6 Dynaaminen muistinvaraus

Javassa aina kun syntyy olio `new` -operaattorin toimesta, olion tilatiedoille varataan tilaa kekomuistista. Tila vapautetaan automaattisesti roskien keruun yhteydessä sitten, kun mistään ei ole enää viitettä olioinstanssiin.

C:ssäkin on mahdollista varata tilaa tietorakenteille dynaamisesti eli aina tarvittaessa. Toisin sanoen on erittäin hyvin mahdollista tehdä dynaamisesti kasvavia ja pieneneviä tietorakenteita,

ihan niinkuin esim. Javan säilöluokat toimivat. Tilanvaraus pitää tehdä muistinvarauskutsulla `malloc()`. Arvatenkin, kuten C:ssä yleensä, ohjelmoija saa käyttöönsä osoittimen varatun tilan alkuun, ja kukaan muu ei pidä kirjaa muistialueiden vapauttamisesta. Ei ole roskienkeruuta eikä kirjanpitoa osoittimien viittauksista. C-ohjelmoinnissa on helppo saada aikaan **muistivuoto** eli mystinen ongelma, jossa ohjelma varaa koko ajan lisää ja lisää keskusmuistia eikä koskaan vapauta sitä. Muistin täyttyminen monen käyttäjän järjestelmässä on sen verran ikävä ilmiö, että jätettäköön `malloc` -harjoittelu itsenäisesti kotikoneella opeteltavaksi.

7 Hosoittaminen minne sattuu

Muistivuodon lisäksi C:ssä on helppo saada aikaan **irrallinen osoitin** (*dangling pointer*), joka on joskus ollut olennainen muistiosoite, mutta jonka osoittama data on aikaa sitten lakannut olemasta. Kyseessä on aina ohjelmoijan huolimattomuus -- hän ei ole pitänyt kirjaa osoituksistaan vaan niistä on tullut hosoituksia. Helposti tämä käy joko dynaamisesti varattavien ja poistettavien alueiden kanssa tai esim. “lasten virheellä”, jossa varataan lokaali taulukko ja kuvitellaan että sen voisi palauttaa aliohjelmasta. Jos pinokehyksen käsite ja aliohjelman suoritusperiaate konekielitasolla on selvää, tiedät, mikä tässä on väärin:

```
int *tee_taulu(){
    int tmp[100];
    return tmp;     /* -Ups- */
}
```

Spoilerina voin kertoa, että taulu `tmp` varataan pinokehyksestä, joka aina rysäytetään aliohjelman päättyessä olemattomiin. Mikään lokaali muuttuja ei elä aliohjelman lopun jälkeen; siksi niiden nimi on lokaali eli paikallinen... Taulu häviää, mutta sen muinoinen muistiosoite palautetaan kutsuvalle aliohjelmalle. Toiminnallista tulosta ei voi ennustaa. Oikeasti ilmeisesti oli tarkoitus tehdä dynaaminen varaus ja palauttaa osoitin dynaamisesti varattuun, uuteen ja dynaamisessa muistialueessa sijaitsevaan tilaan.

Virhe voi tulla helposti Javaan tottuneelle, koska taulukot, merkkijonot ja kaikki ovat siellä aina olioita, jotka luodaan ei-lokaalisti kekoon, ja lokaali viite voidaan kyllä palauttaa returnilla, eikä olio muutu roskaksi, mikäli viite menee kutsujalla talteen.

8 Kontrollirakenteet C:ssä

Kontrollirakenteet ovat C:ssä hyvin samanlaisia kuin Javassa, esimerkkejä alla. Jos se toimii Javassa jollain tavoin, se varmaan toimii C:ssä hyvin samankaltaisesti ja toisin päin.

Ehtolause:

```
if (ehto) {
    ... jotain ...
```

```
} else if (ehto2) {
    ... jotain muuta ...
} else {
    ... vielä jotain ...
}
```

Silmukoita:

```
for (alkuasetus; loppuehto; päivitystapa) {
  ... jotain ...
}
```

```
while (ehto) {... jotain ...}
```

Switch-lause:

```
switch(merkki)
   {
      case 'A' :
        printf("Aaa");
        break;            /* tärkeä! Muuten jatkaa suoritusta */
      case 'B' :
        printf("Bee");
        break;
      case 'C' :
        printf("See");
        break;
      case 'D' :
        printf("katellaas..." );
      case 'E' :
        printf("Dee tai Eee");
        break;
      default  :
        printf( "Ei ollu ABCDE");
   }
```

Aliohjelmakutsu (vrt. metodikutsu):

```
tulos = aliohjelma(param1, param2, param3);
```

Olihan niitä rakenteita varmaan muitakin... yleensä samat toimivat C:ssä kuin Javassa.

9 Mitä ohjelmointi on, kun ei ole olioita

C:ssä ei siis ole olioita. Eli mitä tämä tarkoittaa:

- tietorakenteita ja niitä käsitteleviä algoritmeja ei voi yhdistää samaan pakettiin, jota luokaksi sanottaisiin. On vain muuttujia, tietueita, taulukoita ym. ja sitten aliohjelmia, joille voi antaa dataa käsiteltäväksi. Useimmiten soveltuvinta on antaa datat osoittimina, jolloin kutsuttavat aliohjelmat voivat muuttaa kutsuvan aliohjelman dataa. Tämähän vastaa sitä, että annetaan Javassa olio(viite) jollekin metodille, joka voi käsitellä parametrina saadun olion tilaa, tosin vain olion rajapinnan tarjoamilla tavoilla, mikä on turvallisempaa (hyysäävämpää?) kuin C:ssä, jossa välittyy tieto muokattavista muistipaikoista...
- aiemmin tehtyjä tietorakenteita/algoritmeja ei voi laajentaa perimällä
- tietorakenteiden sisäistä toteutusta ei voi pakotetusti piilottaa soveltajalta; kaikkeen on mahdollista päästä sorkkimaan rajapinnan ohi.
- (ja ei ole poikkeuksia jne...)

Mutta tosiaan ei ole mitään tehtävää, jonka oliokielellä voisi tehdä ja C:llä ei. Kummallakin voi ratkaista minkä tahansa tehtävän, jonka tietokoneella ylipäätään voi. Kyse on vain toteutuksen helppoudesta. Loppujen lopuksi kaikki palautuu siihen, että prosessori suorittaa prosessin konekielistä käskyjonoa yksi käsky kerrallaan nouto-suoritus -syklinsä mukaisesti, sanottiinpa tuota käskysarjaa sitten aliohjelmaksi tahi metodiksi. Tämä asia toivottavasti on yksi, joka iskostuu mieleen Käyttöjärjestelmät -kurssilta.

10 Pakollinen palautustehtävä

Edellä on esitelty C:n toimintaa. Tehtäväpaketin mukana on ohjelmakoodit `koktaulu.c`, `mjono.c` ja `tietue.c`, joiden kommenteissa pyydetään tekemään tietyt täydennykset.

Tehtävä: Tee kommenteissa pyydetyt täydennykset tekstieditorilla, käännä, kokeile ja muokkaa ohjelmia tarpeen mukaan, kunnes ne mielestäsi toimivat oikein.

Kun olet tehnyt muutokset kaikkiin kolmeen tiedostoon, komenna `demo2/` -hakemistossasi:

```
./palautus.sh
```

Tämä ajaa skriptin, joka lähettää mulle meiliä; katso läpi mitä skripti tekee, ja **mielellään käytä sitä vain kerran**, etten huku teidän spämmiin! Jos ei tule virheilmoitusta, viesti todennäköisesti saapuu mulle. Pyrin tekemään automaattisen tarkistimen pian (mikä siinä on, kun ei saa alettua koodaamaan moista pikku juttua; lyökää mua...)

Pari juttua:

- `tietue.c` tarvitsee aliohjelmaa `sqrt()` joka on määritelty `libm.a` -kirjastossa. Käännä siis optiolla `-lm`
- jos ohjelmaan tuli ikuinen silmukka tai muuta jumia, pystyt luultavasti lopettamaan sen painamalla `Ctrl-C`

11 Liite: C-kääntäjän sielunelämä

Katsellaan ensinnäkin, mitä pitää toteutua, kun C-ohjelma käännetään. Verrataan jatkuvasti Javaan, joka oletetaan tutuksi.

11.1 Lähdekoodien ja moduulien organisointi

Java-käännös on selkeää: Jokainen `.java` -päätteinen tiedosto sisältää yhden luokan, joka käännetään erillisesti yhdellä kääntäjäohjelmalla `.class` -päätteiseksi tavukoodiluokaksi. Paketit (eli isommat ohjelmamoduulit, kuten luokkakirjastot) rakennetaan sijoittelemalla lähdekoodit hakemistoihin. Hakemistojen nimet vastaavat paketin nimen osia. Eli esim. luokka `javax.sound.sampled.ReverbType` sijaitsisi hakemistossa `javax/sound/sampled/` josta paketin tunniste `javax.sound.sampled` muodostuu. Luokat ladataan Java-ohjelman ajon aikana tarvittaessa; niitä etsitään sovituista hakemistoista (ns. *classpath*) ja jos niitä ei löydy, heittyy poikkeus `ClassNotFound` tai vastaavaa... Isompia luokkakokoelmia voi paketoida `.jar` -paketeiksi. *Simple as that.*

C:ssä ei ole mitään kiinnitettyä hakemistorakennetta moduuleille, kunhan kääntäjä tietää, mistä sen pitää etsiä `.h` ja `.c` -tiedostoja. Valmiiden kirjastojen include-tiedostot kuten `stdlib.h` ovat usein hakemistossa `/usr/include` (voit huvikseen listata hakemiston, tietenkin... lakkaan nyt selostamasta, että voit huvikseen kokeilla asioita ja olla niistä kiinnostunut). Includeissa on vain ns. "otsikot" eli aliohjelmien ja tietorakenteiden esittelyt. Varsinaiset toteutukset eli valmiiksi käännetyt aliohjelmakirjastot ovat usein hakemistossa `/lib` (peruskirjastot) ja `/usr/lib/` (tarpeen mukaan asennettuja lisäkirjastoja sovellusohjelmien kääntämistä ja ajamista varten). Kääntäjä voitaisiin kuitenkin ohjata etsimään kirjastoja ja otsikoita muualtakin; sijainteja ei ole pakotettu.

C-ohjelman perusyksikkö on `.c`-tiedosto, jossa voi olla yksi tai useampia aliohjelmia. Ei ole mitään standardia ohjaamassa, mitä yhteen tiedostoon laitetaan; siellä voi olla koko 10 000 rivin ohjelma kaikkine määrityksineen ja aliohjelmineen, tai siellä voi olla yksi 30 rivin aliohjelma. Sovelluksen C-lähdekoodi koostuu yhdestä tai useammasta (siis vaikka kuinka monesta) `.c` -ohjelmasta ja `.h` -otsikkotiedostosta (joiden merkitykseen kohta tutustutaan).

C-ohjelman sijoittelussa tiedostoihin ja hakemistoihin on siis rajaton vapaus. Ohjelmoijalla on vastuu siitä, että rakenne on selkeä: hyvä, ylläpidettävä koodi luultavasti on sellainen, joka on jaettu toiminnallisuuden mukaisesti järkevänkokoisiin, samantyyppisiä asioita tekeviä tai käsitteleviä aliohjelmia sisältäviin `.c`-tiedostoihin ja vastaavasti `.h`-tiedostoihin, jotka julkaisevat `.c`-tiedostojen rajapinnat muiden moduulien käytettäväksi. Isompi C-ohjelma on syytä jakaa hakemistoihin, ihan niin kuin Java-ohjelmakin paketteihin. Esimerkiksi Linux Kernel on hyvä C-ohjelmisto, josta voinee ottaa mallia, niin ei mene kovin pahasti pieleen (jos ottaa mallia oikealla tavoin).

11.2 Ohjelman luonti: moduulien erillinen kääntö, yhdistäminen linkittämällä

C-ohjelmien lähdekoodi on siis `.c` -päätteisiä tekstitiedostoja, ja C-koodin suorittaminen on pääasiassa sitä, kun aliohjelmat kutsuvat toisia aliohjelmia jossakin järjestyksessä. Ajettavassa ohjelmassa on löydettävä jokainen tarvittava aliohjelma, joita voi olla monessa paikassa, mm.:

- käännettävässä sovelluksessa, mahdollisesti eri lähdekoodihakemistoissa
- C:n standardiapukirjastoissa
- sovelluskohtaisissa apukirjastoissa

Isot, monia lähdekooditiedostoja sisältävät, C-ohjelmat käännetään ensiksi erillisesti ns. objektitiedostoiksi, joiden pääte on `.o` tai `.obj`. Kussakin objektitiedostossa on vastaavassa lähdekoodissa (`.c` -tiedostossa) olleiden aliohjelmien käännetyt konekielikoodit.

Objektitiedostoja voidaan yhdistää toisiinsa ns. kirjastoiksi (*archive* eli `.a` -tiedostot), joissa on siis kaikissa mukaan otetuissa objektitiedostoissa olevien aliohjelmien konekielikoodi. Apukirjastot ovat yleensä tällaisina `.a` -kirjastoina valmiina liitettäväksi niitä käyttäviin sovelluksiin. Käännösprosessin lopussa objektit ja kirjastot ns. **linkitetään** toisiinsa **linkkerillä**, joka on tätä varten tehty ohjelma. Jonkun objektitiedoston pitää sisältää oikeanlainen `main()` -aliohjelma. Tällä tavoin käännetty lopullinen tiedosto sisältää siis `main`-aliohjelman ja paljon muiden aliohjelmien koodia. Tällaista käännettyä ohjelmaa kutsutaan **staattisesti linkitetyksi**.

Olisi hyvä, jos apukirjaston kaikkia aliohjelmia ei tarvitsisi laittaa mukaan jokaisen niitä käyttävän ohjelman käännökseen. Onhan tilanhukkaa, jos vaikka megatavun kokoinen aliohjelmakirjasto laitetaan kymmeneen eri sovellukseen samanlaisena. Tätä varten voidaan tehdä ns. **dynaaminen linkitys**. Eli varsinkin laajat sovelluskohtaiset apukirjastot (kuten grafiikkakoneistot ym.) kannattaa kääntää ns. **paikkariippumattomaksi koodiksi** (*position independent code*). Silloin niiden kaikki muistiosoiteviitteet (hyppykäskyt tai datan osoitukset) tapahtuvat suhteellisesti IP-rekisterin arvoon (hyppykäskyn omaan osoitteeseen) tai pinokehykseen. Tällainen koodi voidaan linkittää dynaamisesti, eli varsinaisen apukirjastoa käyttävän ohjelman käynnistyksen yhteydessä käyttöjärjestelmän dynaaminen linkitysjärjestelmä etsii apukirjaston tiedostonimen perusteella ja liittää sen ohjelmaan. Tämä on mahdollista tehdä myös ohjelman omasta pyynnöstä sen suorituksen aikana. Eli apukirjaston lähdekoodi käännettiin erityisesti tätä tarkoitusta varten paikkariippumattomiksi objektitiedostoiksi, jotka koottiin dynaamisesti linkitettäväksi kirjastoksi (Windows-maailmassa nimeltään *dynamically linked library*, `.DLL`, Unix-maailmassa *shared object*, `.so`). Järjestelystä on etua ainakin seuraavin tavoin:

- Levytilaa kuluu vähemmän, kun useille ohjelmille käy sama yhdessä paikassa sijaitseva dynaaminen kirjasto.
- Useat ohjelmat voivat käyttää samaa fyysisessä keskusmuistissa olevaa kirjastoa, koska niiden sijainti virtuaalimuistialueella voi olla mikä tahansa. Näin ohjelmien lataamiseen kuluva aika ja keskusmuistin tarve pienenevät.

- Ohjelmista saadaan modulaarisia, esim. jos rajapinta pysyy samana, bugikorjaukset tai parannukset ohjelmiin voidaan tehdä päivittämällä vain kirjasto (joka on pienempi kuin kokonaissofta).

Javan virtuaalikoneen suorituksessa tavallaan ohjelmat aina "linkittyvät dynaamisesti" koska luokkia (jotka tavallaan vastaavat C:n objektitiedostoja) ladataan muistiin tarpeen mukaan.

Esimerkiksi jalavan `/usr/lib/` -hakemisto (alihakemistoineen!) sisältää melkoisen paljon kirjastoja, joita sovellusohjelmiin voisi linkittää dynaamisesti ("`.so`") tai staattisesti ("`.a`", ja ilmeisesti käsittääkseni "`.la`" -tiedostot ovat myös staattisia versioita eli eivät ole riippumattomia sijoittelusta muistialueelle).

Myöhemmässä vaiheessa (viimeistään harjoitustyössä) rakennetaan käsipelillä pieni C-aliohjelmien "kirjasto" ja sitä käyttävä ohjelma.

11.3 Esikäännös: ylimääräinen makrokieli

Ennen kuin kääntäjä alkaa tehdä konekieltä C-lähdekoodista, se ajaa lähdekoodin ns. **esikääntäjän** (*pre-compiler*) läpi. Eli koko C-ohjelman tuottamisen ruljanssi on seuraavanlainen:

- Kaikille `.c` -tiedostoille:
  - alkuperäinen lähdekoodi esikääntäjän läpi
  - esikäännöksen tuottama uusi lähdekoodi varsinaisen C-kääntäjän läpi
- syntyneet objektitiedostot lopulta yhteen ajettavaksi ohjelmaksi tai monikäyttöiseksi apukirjastoksi linkkeriohjelmalla.

Kun C-lähdekoodissa on rivi, joka alkaa risuaitamerkillä `#`, kyseessä on niin sanottu esikääntäjädirektiivi. Esimerkiksi ohjelman alussa olevat `#include` -rivit ovat tällaisia. Siinä kohtaa esikääntäjä etsii includessa mainitun `.h` -tiedoston ja liittää sen sisällön siihen kohtaan lähdekoodia. Jos `.h`-tiedostossa on lisää `#include` tai muita direktiivejä, ne kaikki käsitellään, eli prosessointi on rekursiivinen. Periaatteessa varsinainen kääntäjä saattaa saada aika paljon pidemmän lähdekoodin pureskeltavakseen kuin alkuperäisestä `.c`:stä arvasikaan. Yhtä hyvin `#include` voi ladata minkä tahansa tiedoston lähdekoodin sekaan, mutta siitä voi seurata hankalasti ylläpidettäviä ohjelmia... normaalikäyttö on otsikkotiedostojen sisällyttäminen.

Muita usein käytettyjä direktiivejä ovat seuraavat:

- `#define VAKIO 3.14159` määrittelee makron. Missä kohtaa lähdekoodia tuleekaan vastaan sana `VAKIO`, se korvataan tekstillä `3.14159` esikääntäjän tulosteeseen. Näin voi määritellä vakioita, joita voi käyttää lausekkeissa, esim. `ympari=2*VAKIO*r`. Makrojen nimet kirjoitetaan yleensä isoilla kirjaimilla.
- Pätkä:

```
#ifdef OTA_KOODI_MUKAAN
   mulla = koodia * tassa + jotakin;
   printf("mutta en aina halua sitä mukaan");
#endif
```

tarkoittaa esikääntäjälle sitä, että `#ifdef` ja `#endif` -direktiivien välinen koodi tulee ottaa mukaan vain, jos makro nimeltä `OTA_KOODI_MUKAAN` on asetettu. Yleensä otsikkotiedostot ympäröidään `#ifndef` eli "*if not defined*" menettelyllä:

```
#ifndef TAMA_ON_JO_LISATTY
#define TAMA_ON_JO_LISATTY
... varsinainen otsikkotiedosto ...
#endif
```

joka varmistaa, että jokainen tietorakenne ja aliohjelma esitellään vain kerran -- C ei nimittäin salli uudelleenesittelyä, ja toisekseen muuten voisi tulla ikuinen rekursio esikäännökseen, jos `joku.h` sisällyttää `toinen.h`:n, joka puolestaan sisällyttää `joku.h`:n.

Tällä tavoin voidaan myös toteuttaa siirrettävää C-koodia, jossa sama lähdekoodi käy eri alustoille:

```
#ifdef KAANNETAAN_WINDOWSILLE
   ... Windows-riippuvaista koodia ...
#elif KAANNETAAN_LINUXILLE
   ... Linux-riippuvaista koodia ...
#else
#error "Ei ole toteutettu sinun käyttöjärjestelmällesi.."
#endif
```

Riippuen makrojen asetuksesta kääntäjä jättää jäljelle vain tietyn osan koodia. Linux Kernelissä näkyy näitä paljon; niillä otetaan tai jätetään koodia, joka valitaan konfigurointivaiheessa ennen kääntämistä.

- Pätkä:

```
#if 0

   ... koodia ...

#endif
```

olennaisesti deletoi välissä olevan koodin ennen varsinaista käännöstä, koska `#if 0` ei ole ehtona koskaan tosi. Tällä tavoin voi kätevästi poistaa lähdekoodin osan käytöstä väliaikaisesti kokeilumielessä. Sen voi palauttaa helposti muuttamalla esikääntäjädirektiivit.

Siinäpä se; jotain varmaan unohtuikin. C:n kääntämisen ja lähdekoodien hallinta on vähän monimutkaista. Onneksi uudemmissa kielissä, kuten Javassa, on ymmärretty tehdä järkevämpiä ratkaisuja. (C++:n kun haluttiin olevan täysin yhteensopiva C:n kanssa, niin siinä tämä esikäännös ja objektitiedostojen käyttö on ihan samanlaista).

Tämä siis pohjustuksena C-ohjelmien kääntämisestä, eli työkalujen käytöstä.