C-KIELI

JA

KÄYTÄNNÖN OHJELMOINTI

OSA 1, VERSIO 2

Lappeenrannan teknillinen yliopisto 2013
Jussi Kasurinen, Uolevi Nikula
ISBN 978-952-265-466-3 ISSN 2243-3392

Lappeenrannan teknillinen yliopisto
Lappeenranta University of Technology

Tuotantotalouden tiedekunta
LUT School of Industrial Engineering and Management

Software Engineering and Information Management

LUT Scientific and Expertise Publications

Oppimateriaalit – Lecture Notes 2

Jussi Kasurinen, Uolevi Nikula

C-kieli ja käytännön ohjelmointi osa 1, versio 2

ISBN 978-952-265-466-3
ISBN 978-952-265-467-0 (PDF)
ISSN-L 2243-3392
ISSN 2243-3392

Lappeenranta 2013

Lappeenrannan teknillinen yliopisto, Ohjelmistotuotannon ja tiedonhallinnan laitos

LUT Scientific and Expertise Publications, Oppimateriaalit – Lecture Notes 2

Lappeenranta 30.9.2013

ESIPUHE

Tämä opas on neljäs Lappeenrannan teknillisen yliopiston Tietotekniikan koulutusohjelman ohjelmoinnin perusopetukseen liittyvä opas. Tämä opas on tarkoitettu perehdyttämään lukija C-ohjelmointikielen perusteisiin. Oppaan tarkoituksena ei ole olla täydellinen käsikirja C-ohjelmointikielestä, vaan ensisijaisesti auttaa Python-kielen osaavia aloittamaan C-kielen opiskelu ja käyttö. Opas on laajennettu ja päivitetty versio vuonna 2011 julkaistusta oppaasta.

Oppaassa olevat esimerkkikoodit on tehty ja testattu Windows 7 ympäristössä toimivassa Linux-virtuaalikoneessa (Ubuntu, versio 12.04 LTS, gcc-kääntäjä). Oppaan alkupään esimerkeissä oletetaan, että käyttäjä on aiemmin perehtynyt Python-ohjelmointikieleen sekä osaa sen perusteet, mutta opasta voi käyttää myös itsenäisenä oppaana.

Mikäli sinulla ei ole aiempaa kokemusta ohjelmoinnista, kannattaa ohjelmoinnin opiskelu aloittaa aiemmin julkaistun Python-ohjelmointioppaan avulla (Vanhala ja Nikula 2011). Opas on ilmainen ja aloittaa matalammalta lähtötasolta ohjelmoinnin alkeista. Se on suositeltava alkeisopas ennen tämän oppaan aiheisiin tutustumista, sillä tämä opas etenee nopeammin ja edellyttää aiempaa ohjelmointikokemusta.

Oppaassa on käytetty osia aiemmin kirjoitetusta Satu Alaoutisen (2005) Lyhyestä C-oppaasta ja versiossa 2 on otettu mukaan osia oppaasta ”C-kieli ja käytännön ohjelmointi Osa 2” (Kyttälä ja Nikula 2012). Oppaan nykymuotoon ovat myötävaikuttaneet erityisesti Risto Westman, Satu Alaoutinen ja Erno Vanhala, joista viimeinen osallistui myös Unix-osuuden kirjoittamiseen. Lisäksi oppaan aiempiin versioihin on saatu kommentteja opiskelijoilta. Suuret kiitokset kaikille opasta kommentoineille ja lukeneille henkilöille – turhahan näitä oppaita olisi kirjoittaa, jos niitä ei kukaan lukisi.

SISÄLLYSLUETTELO

Valmistelu

Aloittaaksesi C-ohjelmoinnin tarvitset ohjelmointioppaan lisäksi tietokoneellesi ohjelmointiympäristön. Tämä ohjelmointiopas perustuu Linux-käyttöjärjestelmää, jossa ohjelmointityökalut ovat valmiina mukana. Kurssin ensimmäisellä luennolla puhutaan Linux-ympäristön asentamisesta Windows-ympäristöön virtuaalikoneena, jolloin voit käyttää Windows PC:täsi normaalisti ja siirtyä Linux-ympäristöön niin halutessasi käynnistämällä virtuaalikoneen. Mikäli et ole varma kuinka asennukset tehdään, käy läpi ensimmäisen luennon asiat sekä käy kysymässä apua assistentilta harjoituksissa. Asennus- ja virheenkorjausohjeita löytyy myös kurssin verkkosivuilta. Tämä opas keskittyy ohjelmoinnin käytännölliseen puoleen ja olettaa, että ymmärrät työkalun peruskäytön eikä siihen kiinnitetä erikseen huomiota ensimmäisen luvun ensimmäisen puoliskon jälkeen.

Luku 1: Unix-ympäristö ja C-kieli

Tässä luvussa tutustutaan Unix/Linux käyttöjärjestelmiin sekä käydään läpi Linux-ympäristön ja C-kielen perusasiat. Unix/Linux-osuuden tavoitteena on tarjota sinulle perustiedot näiden käyttöjärjestelmien historiasta, eroista ja yhtenäisyyksistä, sillä vaikka Windows on työpöytäympäristössä yleisin käyttöjärjestelmä, pitää Linux valtaa niin supertietokoneissa, älypuhelimissa, tableteissa, pelikonsoleissa kuin televisioissakin. Siksi Unix/Linux-historian ymmärtäminen on tärkeää ohjelmistoalan asiantuntijoille. Tämän luvun käytännöllinen tavoite on varmistaa, että ymmärrät Linuxin ja C-kielen lähtökohdat ja pystyt tekemään C-kielisiä ohjelmia. Tämä luvun Linux-osuus perustuu vahvasti lähteeseen (Moody 2001).

Tämä opas on tehty lähtien siitä oletuksesta, että tunnet Windows-ympäristön Linux-ympäristöjä paremmin. Windows käyttöjärjestelmä on työasemamarkkinoiden yleisin käyttöjärjestelmä 92 % markkinaosuudellaan (Keizer 2010) ja se edustaa käyttöjärjestelmänä puhtaasti kaupallista linjaa. Windowsin tuettuja työasemaversioita on käytössä tällä hetkellä neljä: Windows XP (tuki loppuu huhtikuussa 2014), Vista, Windows 7 ja Windows 8. Perinteisesti Windowsin työkaluohjelmistot ovat olleet kaupallisia, mutta käyttöjärjestelmään on saatavissa enenevässä määrin myös freeware ja open source -ohjelmia. Nykyään useimmat kehitysympäristöt tukevat Windows-käyttöjärjestelmää sen johtavan markkina-aseman tähden, mutta useat kehitysympäristöt tukevat myös GNU/Linuxia kuten esim. Eclipse, NetBeans ja CodeBlocks. Linux-ympäristössä vallitseva trendi on avoimuus ja vapaat lisenssit GNU-työkalujen ja Linuxin tapaan.

Unixin tausta

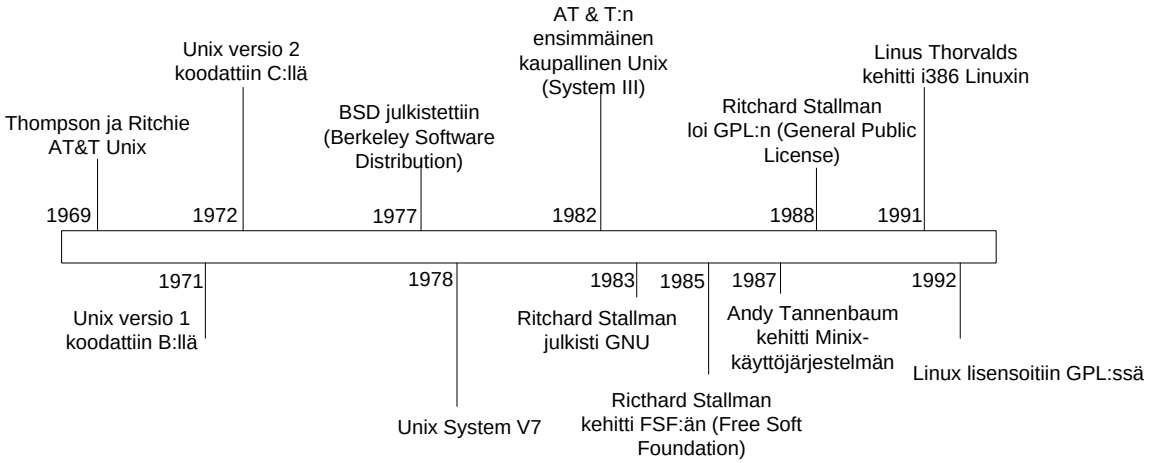
Tietotekniikan aamuhämärässä, kun tietokoneita ei ollut vielä jokaisen ihmisen taskussa vaan lähinnä armeijan ja yliopistojen käytössä, kehitettiin Yhdysvalloissa yleispätevä käyttöjärjestelmä, jonka nimeksi tuli Unix. Samassa yhteydessä aiemmin käytetystä B-ohjelmointikielestä jalostettiin C-kieli.

Unix oli kaupallinen käyttöjärjestelmä, mutta myöhemmin 80-luvulla tietokoneiden jo yleistyttyä Richard Stallman halusi tarjota ihmisille avoimen käyttöjärjestelmän, joka olisi kuitenkin yhtä hyvä kuin Unix ja yhteensopiva sen kanssa. Projekti sai nimekseen GNU (GNU’s not Unix) ja ihmiset työstivät GNU manifeston mukaisesti avoimia ohjelmistoja korvaamaan kaupallisia suljettuja tuotteita, kuten kääntäjiä, pitkin 80-lukua. Projektilta puuttui kuitenkin se kaikkein tärkein eli käyttöjärjestelmän ydin - kernel.

Myös kernelin rakentaminen aloitettiin Stallmanin ohjauksessa ja se valmistui 90-luvun puolivälissä. Projekti kesti kuitenkin niin pitkään, että muut ehtivät havaita saman ongelman ja mm. Linus Torvalds aloitti oman projektinsa Helsingin yliopiston suojissa. Torvalds oli ollut tekemisissä tietokoneiden kanssa lapsuudestaan lähtien ja yliopistoon päästyään hän keräsi rahaa ostaakseen Intelin 80386-arkkitehtuurilla (nykyään i386) varustetun PC:n. Koneen mukana tuli Microsoftin MS-DOS käyttöjärjestelmä, mutta se oli Torvaldsin mielestä aivan liian rajoittunut ja Torvalds käytti sitä vain Prince of Persia –pelin pelaamiseen. Torvalds tilasi koneeseensa Andrew Tanenbaumin luoman MINIX-käyttöjärjestelmän ja huomasi, että MINIXin ja GNUn välissä olisi tilaa uudelle kernelille. Kernelin tärkeimpiä lähtökohtia oli POSIX-yhteensopivuus, sillä POSIX-standardi määrittelee Unix-käyttöjärjestelmien perustoimintoja kuten tiedosto-operaatiot, signaalit ja putket ja näin ollen POSIX-yhteensopivia ohjelmistoja pystyisi käyttämään suoraan uudessa kernelissä. Torvaldsille selvisi kuitenkin pian, että standardit ovat maksullisia eikä hänen opintotukeen perustuva elämänvaihe mahdollistanut niiden ostamista. Torvalds ratkaisi tämän ongelman etsimällä käsiinsä yliopiston SunOS Unix-koneen, sen käsikirjan ja käytti näitä referenssinä luodessaan Linuxin eli käyttöjärjestelmän kernelin. Linux ei siis ole teknillisesti Unix,

mutta käytännössä Linux tekee kaiken mitä muutkin Unixit, sillä se pohjautuu BSD-Unixiin. Unix-järjestelmistä on saatavan myös kaupallisia versioita (esim. Solaris), mutta ne tuntuvat olevan väistymässä avoimen lähdekoodin Unix-järjestelmien tieltä. Näistä esimerkkinä erilaiset BSD:t: FreeBSD, NetBSD ja OpenBSD. On myös syytä huomata, että Mac OS X perustuu Unixiin.

Käsitteisiin liittyen on hyvä huomata, että Linux viittaa periaatteessa vain kerneliin kun taas GNU/Linux viittaa koko käyttöjärjestelmään. Puhekielessä ja tässä oppaassa Linuxilla viitataan koko käyttöjärjestelmään. Kuvassa 1 näkyy Unix-GNU/Linux –järjestelmien tärkeimmät kehitysvaiheet alkaen vuodesta 1969 päättyen Linuxin julkistamiseen.



Kuva 1. Unix GNU/Linux aikajana (Robot Wisdom 2011).

Linux

Ohjelmointi Linux-ympäristössä on luonnollista C-kielellä, sillä järjestelmän ydin, kernel, on kirjoitettu C-kielellä ja esim. perus C-kääntäjät tulevat asennuksen mukana tai ainakin ne ovat Linuxissa ilmaisia ja helppoja asentaa. Lisäksi kääntäjien käyttö on yleensä varsin ongelmatonta ja suoraviivaista. GNU/Linux-jakeluja on useita, joista useimmat perustuvat joko Debianiin, Red Hatiin, Slackwareeen ja SuSeen. Red Hat on vanhimpia Linux-jakeluita ja se tarjoaa kaupallisen käyttöjärjestelmän, johon samanniminen pörssiyhtiö tarjoaa mm. ylläpitopalveluja.

Vaikka Linux ja Unix aikoinaan rakentuivatkin komentorivikäytön varaan, on Linuxille nykyään tarjolla lukuisi työpöytäympäristöjä. Linux-maailma on tällä hetkellä siirtymässä vanhasta X11-ikkunointijärjestelmästä Wayland- ja Mir-järjestelmiin. Näiden järjestelmien päälle on kehitetty useita yhtenäisen käyttöliittymän antavia työpöytäympäristöjä. Käytetyimmät ovat Gnome (Wikipedia 2011), KDE (K Desktop Environment), Xfce ja LXDE (Lightweight X11 Desktop Environment). Työpöytäympäristöt tuovat mukanaan omia sovelluksiaan (KDE:n Kate vs. Gnomen gedit) tai käyttävät yhteisiä hyviksi havaittuja työkaluja (LibreOffice, Firefox, Chromium, Thunderbird). Luonnollisesti sovelluksia voi käyttää ristiin miten tahtoo ikkunointiympäristöstä riippumatta.

Sovelluskehitys Linuxissa hoituu niin komentorivityökaluilla (esim. gcc ja nano) tai sitten laajemmilla integroiduilla kehitysympäristöillä (IDE, Integrated Development Environment) kuten Eclipse tai Netbeans. Ubuntuun löytyy monia hyviä peruseditoreja kuten esim. perusasennukseen kuuluvat `vi, nano, gedit` sekä erikseen asennettava `emacs`. Emacs kehittyneempänä editorina tarjoaa monipuolisia mahdollisuuksia myös koodin kääntämiseen suoraan alasvetovalikon kautta.

Yleisesti ottaen Linuxissa ohjelmia ei etsitä Internetin syövereistä (kuten vaikkapa Winampia tai Firefoxia Windowsissa) tai osteta kaupasta (kuten Microsoft Office tai Adobe Photoshop), vaan ohjelmat asennetaan pakettienhallintatyökalulla. Tästä on se hyöty, että käyttöjärjestelmä päivittää itsensä lisäksi kaikki käytössä olevat ohjelmat, joten käytössä on aina uusin versio Internet-selaimesta ja pikaviestinohjelmasta. Esimerkkejä käytetyimmistä Gnomen mukana toimitettavista sovellusohjelmista ovat

- Evince – PDF- ja PostScript-dokumenttien katselin
- Evolution – sähköposti- ja kalenteriohjelma
- Empathy – pikaviestiohjelma
- Screenshot – kuvankaappausohjelma.

Ubuntu Linux

Ubuntu on avoimesta lähdekoodista koostuva Linux-käyttöjärjestelmä, joka pohjautuu Debian-jakeluun ja sisältää haluttaessa myös suljetun koodin ohjelmistoja kuten Skypen, Adobe Flashin tai suljetut näytönohjainajurit. Ubuntu sisältää kaikki peruskäyttöön tarvittavat ohjelmat, kuten esim. tekstinkäsittelyn, taulukkolaskennan, sähköpostiohjelman ja nettiselaimen. Ohjelmia on helppo asentaa lisää joko komentoriviltä tai pakettienhallintatyökalulla. Graafisen asennusohjelman avulla käyttöjärjestelmän asennus on ongelmatonta ja sujuu nopeasti. Ubuntu-projekti on sitoutunut noudattamaan vapaan ohjelmistokehityksen periaatteita.

Ubuntu on myös pelaajalle parhaiten sopiva Linux-jakelu, koska sen käyttäjäkunta on niin laaja, että ohjeita ja tukea löytyy. Esimerkiksi Valve tarjoaa Steam-palveluaan Ubuntulle.

Yliopistomme Linux-luokassa on käytössä Ubuntu-käyttöjärjestelmä ja kirjauduttaessa koneelle sisään käyttäjä voi valita työpöytäympäristökseen Unityn, KDE:n, Xfce:n tai LXDE:n.

Ubuntun asennus

Linux voidaan asentaa tyhjän koneen lisäksi Windowsin rinnalle nk. dual-boot:ina tai virtuaalikoneeseen. Virtuaalikone löytyy esim. linkistä: http://www.vmware.com/products/player/, samoin ohjeet sen asentamiseen. Dual-boot asennus hoituu sekin suoraviivaisesti asennusohjelman ohjeitten mukaisesti. Valittaessa asennustavaksi dual-boot, GRUP-loader ohjelma latautuu koneelle hoitaen jatkossa alkukäynnistyksen. Ubuntu Linux-käyttöjärjestelmä on ladattavissa osoitteesta http://www.ubuntu-fi.org/ iso- levykuvana.

Keskeisimpiä Unix-komentoja

Linux-ympäristö löytyy www.ubuntu-fi.org –osoitteesta, jossa on myös linkki suomenkieliseen *Ubuntu tutuksi* -oppaaseen (http://fi.wikibooks.org/wiki/Ubuntu_tutuksi).

Keskeisimpiä käskyjä Linux-komentoriviympäristössä tämän kurssin kannalta ovat mm. seuraavat käskyt. Katso tarkemmin http://linux.fi/index.php/Komentorivin_perusteet. Huomaa, että Unixissa ja C-kielessä isot ja pienet kirjaimet ovat eri merkkejä eikä niitä saa sekottaa.

- ls – list (listaa hakemiston sisältö)
- cd – change directory (vaihda hakemistoa)
- pwd – print working directory (tulosta työhakemiston polku)
- mkdir – make directory (luo hakemisto)
- rmdir – remove directory (poista hakemisto)
- touch – luo tyhjän tiedoston tai ”koskettaa” tiedostoa ja merkkaa sen muutetuksi
- rm – remove (tiedosto)
- man – manual (käsikirja-sivut)
- cp – copy (kopioi tiedosto)
- mv – move (siirrä tiedosto)
- gcc – GNU Compiler Collection (CC tarkoittaa tyypillisesti C Compiler eli C-kääntäjä). Tällä käännetään C-ohjelmia alla olevan ohjeen mukaisesti.

Ohjelman kääntäminen Unixissa

Ohjelma kirjoitetaan editointiohjelmalla (esim. gedit) ja tallennetaan haluttuun hakemistoon. Editorin voi käynnistää joko graafisen käyttöliittymän ikonista tai sitten komentoriviltä `gedit ohjelma.c` –käskyllä. Tämän jälkeen ohjelma käännetään komennolla:

```
gcc –o ohjelma ohjelma.c
```

Mikäli käännös onnistuu ilman virheitä, tallentuu hakemistoon suoritettava ohjelmatiedosto, joka on ajettavissa komennolla

```
./ohjelma
```

Huomaa, että ohjelmaa suoritettaessa suoritettavan tiedoston nimen eteen laitetaan piste ja jakoviiva (./), jonka perusteella käyttöjärjestelmä suorittaa nykyisessä hakemistossa olevan binaaritiedoston nimeltään ”ohjelma”.

Edellä olevan minimaalisen komennon sijaan kannattaa yleensä käyttää seuraavaa käännöskäskyä:

```
gcc –o ohjelma ohjelma.c -std=c99 –Wall
```

Molemmissa tapauksissa käännös tehdään gcc-kääntäjällä, jossa parametri `-o` (output) kertoo tehtävän tiedoston nimen (ohjelma) ja syötteenä käytetään `ohjelma.c` nimistä lähdekoodia. Parametri `-std=c99` kertoo kääntäjälle, että lähdekoodi tulee tarkastaa käännöksen yhteydessä ja verrata sitä ANSI C99 standardin mukaisiin vaatimuksiin. Vastaavasti `-Wall` ohjeistaa kääntäjää kertomaan kaikista varoituksista käännöksen yhteydessä (warning all).

Linkkejä Unixin/Linuxin käyttöön liittyen

Ohessa on muutamia hyödyllisiä linkkejä Unixiin liittyen:

- Ohje gcc-kääntäjästä: http://www.network-theory.co.uk/gcc/intro/gccintro-sample.pdf
- Unix-harjoituksia aloittelijoille: http://www.ee.surrey.ac.uk/Teaching/Unix/index.html
- Suomenkielinen *Ubuntu tutuksi* –opas: http://fi.wikibooks.org/wiki/Ubuntu_tutuksi
- Linux-komentoriviympäristö: http://linux.fi/index.php/Komentorivin_perusteet
- Ohjeita työpöytien asennukseen: http://www.psychocats.net/ubuntu/kde.

Ubuntun asennus:

- Virtuaalikone mahdollistaa Linuxin käytön Windows-ympäristössä, esim. luennoilla käytettävä

  http://www.vmware.com/products/player/

  - tai

    https://www.virtualbox.org/
- Ubuntu: http://www.ubuntu-fi.org/
- Linux-ympäristö: www.ubuntu-fi.org

C-kielen syntaksi lyhyesti

C-ohjelmointikieli otettiin käyttöön vuonna 1972 Yhdysvalloissa Unix-käyttöjärjestelmien ohjelmointikielenä. Vaikka kieli alun perin suunniteltiinkin nimenomaisesti järjestelmäohjelmointiin, on se sittemmin löytänyt paikkansa myös sovellusohjelmoinnin parissa. Erityisesti tehtävissä, joissa muistin määrä on rajoitettu tai ohjelman suorituskyvylle asetetaan kovia vaatimuksia, on C-kieli osoittanut olevansa vahva vaihtoehto käännetyn ohjelman kompaktin koon ansiosta. C-kieli sopii hyvin laitteistoläheiseen ohjelmointiin ja sille on tyypillistä, että käyttäjä vastaa monista uudempien kielten automatisoimista toiminnoista. Näitä toimintoja ovat mm. muistinvaraus, dynaamisten rakenteiden määrittely ja toteutus sekä omien toimintojen jälkeinen muistinhallinta. Esimerkiksi Python-kielen laajennukset tehdään usein C-kielellä.

C-kielellä kirjoitetut ohjelmat poikkeavat Python-ohjelmista monilla tavoin alkaen niiden syntaksista. Jos otamme esimerkiksi tekstirivin tulostavan C-ohjelman, poikkeaa se selvästi esimerkiksi Python-ohjelmasta:

```
#include <stdio.h>

int main(void) {
    printf("Minä tein tämän!\n");
    return 0;
}
```

Ensinnäkin, kaikki C-kielellä toteutettu toiminnallinen koodi tulee sijoittaa aina funktion sisälle, ja päätason ohjelmakoodi tulee sijoittaa `main` -nimiseen funktioon. Lisäksi jokaisella funktiolla on oma tyyppinsä – tästä puhumme myöhemmin lisää, joten nyt riittää muistaa, että `main`-funktion tyyppi on integer (`int`). Tämän lisäksi jokainen funktio palauttaa aina toiminnan lopettaessaan jonkin funktion tyyppiin sopivan arvon: esimerkiksi `main`-funktio palauttaa lopettaessaan tyypillisesti arvon 0 (`return 0`). Lisäksi ennen pääohjelmaa jouduimme sisällyttämään `stdio.h` –kirjaston ohjelmaan `include`-käskyllä.

Tarkasteltaessa C-kielen rakennetta merkkitasolla huomaamme, että C-kielessä on Pythoniin nähden ylimääräisiä rakennemerkkejä. Esimerkiksi jokainen koodiosio/lohko (funktio tai toisto- tai ohjausrakenteen toiminnallinen osa), joka Pythonissa sijoitettaisi eri sisennystasolle, merkitään C-kielessä aaltosuluilla ”{” ja ”}”. Aukeavat aaltosulut ovat aina osion ensimmäinen merkki ja sulkeutuvat aaltosulut osion viimeinen merkki; sisennyksillä ei C-kielessä ole toiminnallista roolia, mutta ne helpottavat merkittävästi lähdekoodin ymmärtämistä. Juuri tämän vuoksi on hyvä jatkaa Pythonista tuttua sisennysten käyttöä. Lisäksi jokainen toiminnallinen rivi päättyy puolipisteeseen ”;”.

C-kielen perussyntaksi

C-kielen perussyntaksin erot Python-ohjelmointikieleen ovat seuraavat:

- Kaikki koodi tulee sijoittaa funktion sisälle.
- Päätason koodi tulee aina `main`-nimiseen funktioon eli pääohjelmaan, joka on tyyppiä `int`. Jokaisessa ohjelmassa on oltava yksi (ja vain yksi) `main`-niminen funktio eli pääohjelma, jonka lisäksi samassa ohjelmassa voi olla muita aliohjelmia.

- Funktiot loppuvat aina `return`-käskyyn, joka palauttaa funktion tyyppiin sopivan arvon. Pääohjelma loppuu tyypillisesti käskyyn `return 0`. Mikäli aliohjelma ei palauta arvoa, on se tyypiltään `void`.
- Koodilohko alkaa aina aukeavalla aaltosululla ”{” ja loppuu sulkeutuvaan aaltosulkuun ”}”. Sisennys ei vaikuta lähdekoodin osiojakoon, mutta se vaikuttaa merkittävästi koodin luettavuuteen. Näin ollen systemaattinen sisentäminen on hyvän ohjelmointityylin perusta.
- Jokainen käskyrivi koodilohkossa päättyy puolipisteeseen ”;”.
- Käytännössä **jokainen koodirivi päättyy joko aaltosulkuun tai puolipisteeseen.** Jos rivi päättyy aukeavaan aaltosulkuun, sisennä seuraavaa riviä yksi sisennystaso enemmän, ja jos taas sulkeutuvaan aaltosulkuun, sisennä seuraavaa riviä yksi sisennystaso vähemmän.

Seuraavaksi tutustumme näihin sääntöihin käytännössä ohjelmakoodiesimerkin avulla. Mikäli et vielä ole tutustunut ohjelmointiympäristöön, tässä vaiheessa on hyvä tehdä niin. Ohjelman käyttämiseen löydät apua kurssisivuilta sekä ensimmäisen kurssiviikon harjoituksista.

Esimerkki 1.1: Tekstin tulostus ruudulle

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    /* Tämä on kommentti */
    printf("Minä tein tämän!\n");
    printf("Huomaa, että teksti jatkuu aina edellisen ");
    printf("perään,\n ellei tulostus pääty rivinvaihtomerkkiin!");

    /* C-kielen kommenttimerkit
      ovat monirivisiä */
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin käskyllä ’`gcc –o ohjelma ohjelma.c`’ ja ajamme sen kotihakemistossa olevassa un-hakemistossa, saamme seuraavanlaisen tuloksen (vrt. Ohjelman kääntäminen Unixissa):

```
un@linux:~/un$ ./ohjelma
Minä tein tämän!
Huomaa, että teksti jatkuu aina edellisen perään,
 ellei tulostus pääty rivinvaihtomerkkiin!un@linux:~/un$
```

Kuinka koodi toimii

Tässä vaiheessa on hyvä sanoa muutama sana C-kääntäjän käytöstä. Ensinnäkin, koska C on kääntäjäpohjainen ohjelmointikieli, tarkoittaa se sitä, että virheettömästä lähdekoodista tehdään itse asiassa itsenäisesti ajettavissa oleva tiedosto. Tämä on merkittävä ero tulkattaviin kieleen kuten esimerkiksi Python-ohjelmointikieleen, jossa Python-lähdekoodi piti erikseen paketoida erillisellä moduulilla, jos lähdekoodia halusi käyttää ilman ajoympäristöä. C-kielessä taas kääntäjä tekee lähdekoodista sellaisen ohjelmapaketin, jota käytetty käyttöjärjestelmä – tässä tapauksessa siis

Linux – ymmärtää ilman muiden ohjelmien apua. Tähän liittyy kuitenkin muutama rajoite, joihin palaamme hetken päästä tarkemmin.

Ohjelmaesimerkki alkaa kirjaston sisällyttämisellä. Sisällytämme koodiin `stdio.h` –kirjaston, joka sisältää tavallisimpia tietojen vastaanottamiseen ja tulostamiseen käytettyjä funktioita. C-kielessä on tavallista, että tarvitsemme käyttöömme jo pelkkiä ”perustoimintoja” varten kirjastoja. Kirjastojen käytöstä puhumme lisää myöhemmin luvussa 4. Toistaiseksi riittää kun ymmärrät, että käskyllä `#include <kirjastonnimi>` voimme ottaa käyttöön kirjastoja ja että kirjasto `stdio.h` pitää ottaa käyttöön aina kun ohjelmassa on syöttö- tai tulostuslauseita eli I/O:ta (Input/Output).

Mitä itse ohjelma tekee? Jos katsomme pääohjelman määrittelevää käskyä `int main(void)`, niin näemme, että siinä on funktion parametrien sijaan sana `void`. `void` on C-kielen varattu sana ja tarkoittaa tyhjää, olematonta, ”ei-olemassaolevaa”. Kerromme siis itse asiassa, että `main`-funktio ei vastaanota mitään parametreja. Tämän jälkeen avaamme aaltosululla pääohjelman koodilohkon.

Pääohjelmaa tarkasteltaessa huomaamme, että funktiossa on ensimmäisenä kauttaviivan ja tähtimerkin erottama kommenttimerkki. C-kielessä tällä tavoin merkitään lähdekoodin seassa olevia kommenttimerkkejä eli laittamalla teksti merkkiparien ”/*” ja ”*/” sisään. Kommenttimerkit C-kielessä ovat aina oletuksena monirivisiä, joten kääntäjä odottaa aina saavansa merkkiparille ”/*” parin ”*/”. Avoimeksi jätetyn kommenttirivin jälkeisen koodin kääntäjä katsoo edelleen kommenteiksi niin kauan kunnes vastaan tulee lähdekooditiedoston loppu tai sulkeva kommenttimerkki. Vuonna 1999 hyväksytty C-standardi (C99) sisältää myös yhden rivin kommentit eli ”//” (kaksi jakoviivaa) aloittavat kommenttikentän, joka päättyy seuraavaan rivinvaihtomerkkiin.

Seuraavaksi koodissa on kolme tulostuslausetta. Tulostuksen siirtyminen uudelle riville ei tapahdu automaattisesti, joten jos tulostettava teksti ei pääty rivinvaihtomerkkiin, jatkuu seuraava tulostus aina edellisen rivin perään. Muista, että `printf`-lauseet eivät aloita uutta koodiosiota, joten ne on lopetettava puolipisteellä.

Tämän jälkeen koodissa on toinen kommenttiteksti sekä funktion lopetuksen osoittava `return`-käsky. Tämä käsky käytännössä määrää, että `main`-funktio lopettaa toimintansa lopetusarvolla 0. Tämä tarkoittaa sitä, että lopetusarvo palautetaan ohjelman käynnistäneelle ohjelmalle/ käyttöjärjestelmälle. Paluuarvo voi olla virhekoodi, mutta tyypillisesti käyttöjärjestelmälle palautetaan lopetusarvo 0, joka kertoo lopetuksen tapahtuneen hallitusti ja odotusten mukaisesti. Lopuksi vielä suljemme auki olleen `main`-funktion koodiosion sulkevalla aaltosulkeella.

Huomioita kääntäjistä ja käyttöjärjestelmistä

Aiemmin mainitsimme, että kääntäjien ja käyttöjärjestelmien kanssa on joitain huomionarvoisia seikkoja. Tällä viittasimme siihen, että jokainen kääntäjä tuottaa ohjelmia omaan kohdeympäristöönsä. Yleensä tuo kohdeympäristö on sama kuin missä kääntäjä toimii (esim. Linux tai Windows), mutta erityisesti sulautettujen järjestelmien ohjelmat voidaan kääntää kehitysympäristössä ja siirtää sitten varsinaiseen kohdeympäristöönsä suoritettavaksi (esim. askelmittari, kännykkä, tms.). Normaalisti Linux-ympäristössä käännetty ohjelma ei toimi Windows-ympäristössä ja päinvastoin.

Ongelma eri kääntäjien välillä on siinä, että ne saattavat tulkita samaa koodia hieman eri tavoin tehdessään suoraan käyttöjärjestelmän päällä toimivaa ohjelmaa. Tämän lisäksi eri

käyttöjärjestelmillä saattaa olla erilainen politiikka esimerkiksi muistinhallinnan tai muiden resurssien käytön suhteen.

Usein nämä ongelmat tulevat huomionarvoisiksi vasta myöhemmin eivätkä oikeastaan vaikuta mitenkään perusrakenteilla ohjelmointiin. Siitäkin huolimatta kannattaa muistaa, että C-kielistä ohjelmaa ei välttämättä pysty kääntämään sellaisenaan muussa käyttöjärjestelmässä kuin missä se on tehty. Lisäksi eri kääntäjät saattavat tulkita asioita eri tavoin, jolloin yhdellä kääntäjällä toimiva lähdekoodi saattaa vaatia muuntelua toimiakseen toisella. Tämän vuoksi on hyvä huomata, että mikäli työskentelet jollain muulla kääntäjällä kuin oppaassa mainitulla kääntäjällä, voivat jotkin yksityiskohdat poiketa toisistaan, esim. muuttujien ylä- ja alarajat sekä varoitus- ja huomautuspolitiikka. Myös kääntäjän optioilla voidaan vaikuttaa merkittävästi käännöksen kulkuun ja käännettyjen ohjelmien toimintaan.

Syötteiden vastaanottaminen

Otetaan vielä toinen esimerkki perusohjelmoinnista. Teemme nyt ohjelman, joka kysyy käyttäjältä lukua, tallentaa luvun muuttujaan ja tulostaa annetun luvun käyttäjälle.

Esimerkki 1.2: Syötteen ottaminen käyttäjältä

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    int luku;
    printf("Anna kokonaisluku: ");
    scanf("%d", &luku);
    printf("Annoit luvun %d.\n", luku);
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa hakemistossa `un`, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./1-2
Anna kokonaisluku: 34
Annoit luvun 34.
un@linux:~/un$
```

Kuinka koodi toimii

Ohjelma alkaa pääohjelmalla eli `main`-funktion koodilohkolla, jossa ensitöiksemme määrittelemme muuttujan `luku`. C-kielessä muuttujat pitää esitellä funktion tai koodilohkon alussa eikä niitä voi käyttää ennen määrittelyä. Myös tästä puhumme lisää seuraavassa luvussa.

Seuraavana meillä on koodin varsinainen toiminnallisuus: syötteen pyytäminen ja tallentaminen. C-kielessä syötteen pyytäminen tehdään tyypillisesti kahdella lauseella:

```
printf("Anna kokonaisluku: ");
scanf("%d",&luku);
```

Ensin annamme käyttäjälle toimintaohjeen `printf`-lauseella, joka tulostaa ruudulle ohjeen antaa kokonaisluku. Koska C-kielen `printf`-funktio ei lisää rivinvaihtomerkkiä merkkijonon perään, pysyy kursori samalla rivillä ohjeiden kanssa muodostaen syötekentän. Seuraavalla rivillä olevalla käskyllä `scanf` luemme tähän kohtaan käyttäjän antaman syötteen.

Huomionarvoista `scanf`-käskyssä on sen käyttö. Ensin kerromme minkälaista tietoa otamme vastaan määrittelemällä, että käyttäjältä odotetaan yhtä kokonaislukua (`printf`-tyylinen formaatti eli `"%d"`). Tämän jälkeen kerromme, että haluamme sijoittaa kyseisen käyttäjältä luetun arvon muuttujaan `luku`. Tämä arvon sijoittaminen muuttujaan edellyttää C-kielessä muuttujan nimen eteen &-merkkiä. Palamme tähän &-merkkiin eli muuttujan osoitteeseen myöhemmin.

Saatuamme luettua käyttäjän arvon tulostamme sen vielä `printf`-lauseen avulla, jotta varmistumme lukemisen onnistumisesta. Lopuksi päätämme funktion palauttamalla arvon 0 ja viimeistelemme ohjelmakoodin sulkemalla `main`-funktion koodiosion.

Tämän esimerkin avulla näimme lyhyesti, kuinka perustason lukuarvojen pyytäminen ja vastaanottaminen toimii. Esimerkki jätti luultavasti vielä paljon kysymyksiä, mutta niihin pyrimme vastaamaan seuraavissa luvuissa, joissa tutustumme C-kieleen tarkemmin.

Osaamistavoitteet

Tämän ensimmäisen luvun jälkeen sinun tulee tietää ja ymmärtää Unixin ja C-kielen perusasiat siinä määrin, että voit aloittaa niiden omatoimisen käytön. Sinun tulee myös tuntea Unixin käskyjä siinä määrin, että pystyt liikkumaan Unix-pohjaisen käyttöjärjestelmän hakemistoissa, pystyt luomaan uusia tiedostoja ja hakemistoja sekä poistamaan niitä. C-kielen osalta sinun tulee pystyä luomaan yksinkertainen tulostamista ja kokonaislukusyötteitä sisältävä C-kielinen ohjelma ja kääntää sekä suorittaa se Unix-pohjaisessa järjestelmässä.

Luku 2: Tiedon käsittely ja tulostus C-kielessä

Tässä luvussa käymme läpi tarkemmin tiedon käsittelyyn tarvittavat muuttujat sekä niihin liittyvät tietotyypit, operaattorit, syöttö- ja tulostusoperaatiot sekä kaksi Python-kielestä merkittävästi poikkeavaa asiaa eli osoittimet ja merkkijonojen käsittelyn.

Muuttujat ja tietotyypit

C-kielen tavoitteena on ollut kevyt ja nopea kieli, joten ohjelmointikieli itse ei automatisoi juuri mitään tehtäviä vaan käyttäjän on tehtävä kaikki itse, jos pitää sitä aiheellisena. Myöhemmin kehitetyt kielet automatisoivat monia asioita, jotka C-kielessä pitää tehdä itse. Useimmissa ohjelmointikielissä on kuitenkin tiettyjä yhteisiä piirteitä kuten esimerkiksi muuttujat ja muuttujien tietotyypit.

Python-ohjelmointikielessä muuttujilla oli käytännössä kolme erilaista tyyppiä, numeroarvo (int/float), merkkijono (str) taikka rakenne kuten lista tai tuple. Lisäksi muuttujan tyyppiä pystyttiin vaihtamaan lennosta sijoittamalla merkkijono numeroarvoja sisältäneeseen muuttujaan tai toisinpäin. C-kielessä muuttujien käyttö ei varsinaisesti poikkea paljoakaan Pythonista, mutta muuttujat pitää aina määritellä ennen käyttöä ja samassa yhteydessä määritetään sen tietotyyppi, jota ei voi vaihtaa ohjelman suorituksen aikana.

Esimerkki 2.1: Muuttujan määrittely

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    /* Määritellään muuttujat */

    int luku;
    float liukuluku;

    /* Nyt meillä on joukko muuttujia,
     * käytetään niitä koodissa... */

    luku = 5;
    liukuluku = 8.234234645;
    printf("Luku-muuttuja on %d.\n", luku);
    printf("%d %f", luku, liukuluku);

    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./2-1
Luku-muuttuja on 5.
5 8.234235
un@linux:~/un$
```

Kuinka koodi toimii

Ohjelma alkaa kuten ensimmäisen luvun esimerkkikin. Teemme ensin `main`-funktion, ja avaamme sille koodiosion kaarisuluilla. Tämän jälkeen meillä on kommenttirivi, jota siis ei lasketa varsinaisesti koodiriviksi, ja tämän jälkeen törmäämmekin muuttujien esittelyyn.

Kuten huomaat, ensimmäinen main-funktiossa tehtävä asia on muuttujien esittely. Jokainen ohjelman esittelyn jälkeisessä koodissa käytettävä muuttuja tulee määritellä tässä kohtaa kertomalla sen nimi ja tyyppi. Riveillä `int luku;` ja `float liukuluku;` kerromme C-kääntäjälle, että aiomme käyttää kahta muuttujaa nimiltään `luku` ja `liukuluku`. Muuttuja `luku` on tyyppiä `int`, eli kokonaisluku ja `liukuluku` tyyppiä `float`, eli liukuluku.

Kun olemme esitelleet muuttujat, voimme käyttää niitä koodissa aivan kuten Python-muuttujiakin. Voimme sijoittaa muuttujiin arvoja, tulostaa niitä sekä laskea eri muuttujia yhteen tai vähentää mielemme mukaisesti. Huomaa kuitenkin, että C-kieli ei ymmärtäisi liukuluvun tallentamista kokonaislukuun, vaan automaattisesti katkaisisi desimaaliosan pois tallentaen ainoastaan kokonaislukuosan muuttujaan. Tämän vuoksi on hyvin tärkeää etukäteen suunnitella ohjelmaansa sen verran eteenpäin, että ei joudu tilanteeseen jossa kokonaislukumuuttujaan pitäisi tallentaa vaikkapa liukuluku.

Numeeriset muuttujat

Huomasit varmaan edellisessä esimerkissä, että kokonaislukua `int` ja liukulukua `float` käsiteltiin siinä mielessä eritavoin, että jopa niiden tulostuskäskyn sijoitusmerkit (`%d` ja `%f`) olivat erilaisia. Tämä liittyy siihen, että C-kielen muistinhallinta ei ole automatisoitu kuin hyvin alhaisella tasolla. Koska erilaiset numeromuuttujat tarvitsevat muistiin sijoittamista varten erilaisen määrän muistia, käsitellään niitä oikeasti erilaisina muuttujatyyppeinä. Tämän vuoksi erityyppisten numeeristen muuttujien välisten laskutoimitusten yhteydessä tulee olla tarkkana, sillä tietotyyppien välisissä operaatioissa on tunnettava niiden yhteensopivuussäännöt virheiden välttämiseksi.

C-kielen tiukat tietotyyppimäärittelyt mahdollistavat muistin käytön optimoinnin erikseen jokaiseen tarpeeseen. Koska Pythonissa dynaamisuus huolehti erilaisten numeroarvojen välisestä yhteensopivuudesta, ei käytännössä tarvittu kuin kaksi numeromuuttujaa, kokonaisluku ja liukuluku. Automatisoidun muistinkäytön vuoksi numeerisilla muuttujilla ei myöskään ollut varsinasta ylä- eikä alarajaa. Tämä lähestymistapa tekee ohjelmoinnista helpompaa, mutta samalla ohjelmoija menettää muistinkäytön kontrollointimahdollisuudet. C-kieli jättää vastuun muistin käytöstä ohjelmoijalla, jonka seurauksena C-ohjelmien muuttujat voivat joutua ns. ylivuoto-tilanteeseen, jossa ohjelma ei enää toimi oikein muuttujan arvon ylittäessä ylä- tai alarajan. Usein termeillä yli- ja alivuoto vielä erotellaan mentiinkö alarajan ali vai ylärajan yli.

Taulukko 1: C-kielen kokonaislukutyyppien minimikoot

Muuttujan nimi	Tyyppi	Koko (tavua)	Alaraja	Yläraja
etumerkitön arvo /merkki	unsigned char	1	0	255
etumerkillinen arvo/merkki	signer char	1	-128	127
Merkki	char	1	-128	127
etumerkitön lyhyt kokonaisluku	unsigned short int	2	0	65535
lyhyt kokonaisluku	short int	2	-32768	32767
etumerkitön kokonaisluku	unsigned int	2	0	65535
kokonaisluku	int	2	-32768	32767
etumerkitön pitkä kokonaisluku	unsigned long int	4	0	4294967295
pitkä kokonaisluku	long int	4	-2147483648	2147483647

Taulukkoa tutkiessa kannattaa huomata, että annetut ala- ja ylärajat eivät ole vakioita vaan arvoja, joita niiden on oltava vähintään käytetystä kääntäjästä ja ajoympäristöstä riippumatta. Lisäksi esimerkiksi `long int` ei nimestään huolimatta välttämättä tue kovinkaan suuria numeroarvoja, vaikka tavallisesti raja onkin paljon korkeampi. Vastaavasti pelkkä `int` on kokoluokaltaan erityisen vaihteleva eri ajoympäristöjen välillä: joissain järjestelmissä sen koko vastaa `short int`-tyyppiä, joissain `long int`:ia. Joka tapauksessa ympäristöstä riippumatta se on vähintään samankokoinen kuin `short int`, joten sitä tulisi käsitellä kuten `short int`-muuttujaa. Lisäksi merkkimuuttuja `char` on listattu tähän listalle, koska C ei varsinaisesti käsittele yksittäistä merkkiä kirjaimena, vaan ASCII-taulukon merkkiä vastaavana numeroarvona. Käytännössä siis C-kieli tallentaa lukuarvon, mutta esittää tulostettaessa sitä vastaavan ASCII-merkin.

Taulukko 2: C-kielen liukuluvut

Muuttujan nimi	Tyyppi	Koko (tavua)	Alaraja	Yläraja
Liukuluku	float	4	1.17549e-38	3.40282e+38
Kaksinkertaisen tarkkuuden liukuluku	double	8	2.22507e-308	1.79769e+308

Liukulukuja käsiteltäessä tulee pitää mielessä, että ohjelmointikielet käsittelevät desimaalilukuja bittiarvoina. Tämän seurauksena desimaaliluvut esitetään approksimaatioina. Erityisesti päättymättömien desimaalilukujen kohdalla joudutaan käyttämään parhaalle tarkkuudelle tehtyjä pyöristyksiä johtuen siitä, ettei binaarilukujärjestelmä tue tällaisia lukuarvoja. Tästä ei kuitenkaan kannata olla huolissaan peruslaskutoimitusten yhteydessä, sillä esimerkiksi normaalin liukuluvun `float`:in tarkkuus riittää kuvaamaan $2.3*10^{-9}$ –kokoluokan (0.00000000023) eroja. Ja mikäli tämä ei riitä, voidaan tarkkuus kaksinkertaistaa siirtymällä `double`-liukulukutyyppiin.

Operaattorit

Numeeristen muuttujien käyttäminen olisi merkityksetöntä, mikäli niitä ei voitaisi käyttää laskutoimituksissa. C-kieli tarjoaa monia laskentaoperaattoreita, loogisia ja muita operaattoreita kuten bittioperaattoreita. Monet näistä ovat samoja kuin Python-ohjelmoinnissa, mutta huomaa ettei C-kieli ei käytä avainsanoja True ja False vaan niiden numeerisia esitysmuotoja 0 (False) ja 1 (True) väittämien yhteydessä.

Taulukko 3. Laskentaoperaattorit

Operaattori	Nimi	Selite	Esimerkki
=	Sijoitus	Sijoittaa annetun arvon kohdemuuttujalle	`luku = 5` sijoittaa muuttujalle luku arvon 5. Operaattori toimii ainoastaan mikäli kohteena on muuttuja.
+	Plus	Laskee yhteen kaksi operandia, esim. 2 lukua	`3 + 5` antaa arvon `8`.
[muuttuja]++	Lisäys	Lisää muuttujan arvoa yhdellä.	Jos `luku = 5;` niin `luku++;` tuottaa muuttujalle `luku` arvon `6`.
-	Miinus	Palauttaa joko negatiivisen arvon tai vähentää kaksi operandia toisistaan	`-5` tarkoittaa negatiivista numeroarvoa. `50 - 24` antaa arvon `26`.
[muuttuja]--	Vähennys	Vähentää muuttujan arvoa yhdellä.	Jos `luku = 5;` niin `luku--;` tuottaa muuttujalle `luku` arvon `4`.
*	Tulo	Palauttaa kahden operandin tulon	`2 * 3` antaa arvon `6`.
/	Jako	Jakaa x:n y:llä	`4/3` antaa arvon `1` (kokonaislukujen jako palauttaa kokonaisluvun). `4.0/3.0` antaa arvon `1.3333333333333333`. Muista muuttujien tyypit!
%	Jakojäännös	Palauttaa x:n jakojäännöksen y:stä.	`8%3` antaa 2. `-25.5%2.25` antaa `1.5`.

Taulukko 4. Loogiset- eli vertailuoperaattorit

Operaattori	Nimi	Selite	**Esimerkki**
<	Pienempi kuin	Palauttaa tiedon siitä onko x vähemmän kuin y. Vertailu palauttaa arvon 0 tai 1.	`5 < 3` palauttaa arvon `0` ja `3 < 5` palauttaa arvon `1`.
>	Suurempi kuin	Palauttaa tiedon onko x enemmän kuin y.	`5 > 3` palauttaa arvon 1.
<=	Vähemmän, tai yhtä suuri	Palauttaa tiedon onko x pienempi tai yhtä suuri kuin y.	`x = 3; y = 6; x <= y` palauttaa arvon 1.
>=	Suurempi, tai yhtä suuri	Palauttaa tiedon onko x suurempi tai yhtä suuri kuin y.	`x = 4; y = 3; x >= 3` palauttaa arvon 1.
==	Yhtä suuri kuin	Testaa ovatko operandit yhtä suuria.	`x = 2; y = 2; x == y` palauttaa arvon 1.
!=	Erisuuri kuin	Testaa ovatko operandit erisuuria.	`x = 2; y = 3; x != y` palauttaa arvon 1.

Taulukko 5. Boolean-operaattorit

Operaattori	Nimi	Selite	**Esimerkki**
!	Boolean NOT	Jos x on `1`, palautuu arvo `0`. Jos x on `0`, palautuu arvo `1`.	`x = 1; !x` palauttaa arvon 0.
&&	Boolean AND	`x && y` palauttaa arvon `0` jos x on `0`, muulloin se palauttaa y:n arvon	`x = 0; y = 1; x && y` palauttaa arvon `0` koska x on 0.
||	Boolean OR	Jos x on `1`, palautuu arvo 1, muussa tapauksessa se palauttaa y:n arvon.	`x = 1; y = 0; x || y` palauttaa arvon 1. (Merkki | saadaan näppäimillä AltGr – ”<”.)

Taulukko 6. Bittioperaattorit

Operaattori	Nimi	Selite	**Esimerkki**
<<	Siirto vasempaan	Siirtää luvun bittejä annetun numeroarvon verran vasemmalle. (Jokainen luku on ilmaistu muistissa bitteinä.)	`2 << 2` palauttaa `8`. `2` on ilmaistu käyttäen arvoa `10`. Vasemmalle siirtyminen 2 bitin verran antaa rivin `1000`, joka taas on arvona 8.
>>	Siirto oikeaan	Siirtää luvun bittejä oikealle numeroarvon verran.	`11 >> 1` palauttaa `5`. `11` on bitteinä `1011`, josta siirryttäessä 1 bitti oikeaan tulee `101` joka taas lukuina on `5`.
&	Bittijonon AND	Bittijonon AND yksittäisille biteille.	`5 & 3` palauttaa arvon `1`.
|	Bittijonon OR	Bittijonon OR yksittäisille biteille.	`5 | 3` palauttaa arvon `7`.
^	Bittijonon XOR	XOR (valikoiva OR) yksittäisille biteille.	`5 ^ 3` antaa arvon `6`.

Operaattorien suoritusjärjestys

Jos sinulla on vaikkapa lauseke `2 + 3 * 4`, niin missä järjestyksessä laskutoimitukset tehdään? Jo Python-kurssilla opimme, että ohjelmointikieliin on sisäänrakennettu suoritusjärjestys operaattoreille ja tämä pitää paikkansa myös C-kielen tapauksessa. C-kielessä operaattoreiden suoritusjärjestys on seuraavanlainen:

Taulukko 7. C-kielen operaattorien suoritusjärjestys

Ryhmä	Operaattori	Suoritussuunta	
Sijoitukset, sulut	() [] -> . ++ --	vasemmalta oikealle	Korkea prioriteetti
Viittaukset	(tyyppi) * &	oikealta vasemmalle	
Kertoma	* / %	vasemmalta oikealle	
Lisäys	+ -	vasemmalta oikealle	
Siirto	<< >>	vasemmalta oikealle	
Vertailu	< <= > >=	vasemmalta oikealle	
Yhtäsuuruus	== !=	vasemmalta oikealle	
Bitti AND	&	vasemmalta oikealle	
Bitti XOR	^	vasemmalta oikealle	
Bitti OR	|	vasemmalta oikealle	
Looginen AND	&&	vasemmalta oikealle	Matala prioriteetti
Looginen OR	||	vasemmalta oikealle	
Ehtolause	?:	oikealta vasemmalle	
Sijoitus	= += -= *= /= %=	oikealta vasemmalle	
Pilkku	,	vasemmalta oikealle	

Kuten huomaamme, on suoritusjärjestys muutamaa poikkeusta lukuun ottamatta aina vasemmalta oikealle. Voimme tietenkin muuttaa operaattorien suoritusjärjestystä käyttämällä sulkuja. Esimerkiksi

`x = 7 + 3 * 2` tuottaa tuloksen 13, kun taas
`x = (7 + 3) * 2` tuottaa tuloksen 20.

Käytännössä C-kieli toimii loogisesti samalla tavalla lasku- ja logiikkaoperaattorien kanssa kuin muutkin ohjelmointikielet, joten laskentaan liittyvien suoritusjärjestysten eli presedenssien ei pitäisi tuottaa ongelmia. Epäselvissä tapauksissa kannattaa selkiyttää asiaa suluilla.

Merkkijonot

Huomasit varmaan, että aikaisemmassa osiossa puhuimme ainoastaan numeerisista muuttujista sivuten ohimennen yksittäisten merkkien tapausta. Kuinka sitten merkkijonoja käsitellään C-kielessä? Tämä on ensimmäinen merkittävä ero C-kielen ja Pythonin välillä. Koska merkkijonon tarvitsema muistin määrä riippuu merkkijonon pituudesta, emme pystykään C-kielessä käyttämään niitä aivan yhtä suoraviivaisesti.

Periaatteessa meillä on muutamia vaihtoehtoja merkkijonojen käsittelyyn. Ensinnäkin, voimme luoda yksittäisistä merkki-muuttujista määrätynmittaisen taulukon ja tallentaa tiedon siihen. Tämä on muuten näppärä idea, mutta jos teemme liian lyhyen merkkitaulukon, ei käyttäjän syöte mahdu kokonaan varattuun tilaan ja jos taas teemme taulukosta liian suuren, tuhlaamme muistia tyhjän

tilan tallentamiseen. Toinen vaihtoehto onkin varata käsin muistia riittävästi aina kun haluamme tallentaa merkkijonon. Palaamme muistinhallintaan oppaan myöhemmissä luvuissa ja tässä vaiheessa tutustumme merkkijonoihin staattisten taulukoiden avulla.

Esimerkki 2.2: Merkkijonon määrittely ja käyttäminen

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    /* Määritellään muutama merkkijono ja apumuuttujia. */
    char nimi[20];
    char ammatti[] = "Palomies";
    char harrastus[20] = "autot";
    int koko;

    /* Nyt meillä on merkkijonoja, käytetään niitä vähän. */
    printf("Anna nimi (max. 20 merkkiä): ");
    fgets(nimi, 20, stdin);

    koko = sizeof (harrastus);

    printf("%s on %s.\n", nimi, ammatti);
    printf("Harrastuksenaan hänellä on %s.\n", harrastus);
    printf("Sanassa %s on %d merkkiä.", harrastus, koko);
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./2-2
Anna nimi (max. 20 merkkiä): Erkki
Erkki on Palomies.
Harrastuksenaan hänellä on autot.
Sanassa autot on 20 merkkiä.
un@linux:~/un$
```

Kuinka koodi toimii

Ohjelma alkaa tavalliseen tapaan ensin `stdio.h`-kirjaston käyttöönotolla ja `main`-funktion aloittamisella. Tämän jälkeen määrittelemme käyttämämme merkkitaulukot `nimi`, `ammatti` ja `harrastus`.

Ensinnäkin, koska tiedämme käsittelevämme merkkejä, käytämme tässä tapauksessa muuttujien tyyppinä `char` - eli ’yksittäinen merkki’ -tyyppiä. Käytännössä me ilmoitamme kääntäjälle, kuinka monta peräkkäistä merkkiä me tarvitsemme, jotta kääntäjä osaa varata tilan etukäteen. Tämän voimme toteuttaa useammalla eri tavalla:

```
char nimi[20];
```

Tällä käskyllä määrittelemme merkkijonotaulukon `nimi`, johon on varattu tilaa 20 merkkiä (taulukon paikat 0-19). Huomaa, että merkkitaulukon ensimmäinen paikka on numeroltaan 0. Tällöin 8 merkkiä pitkän merkkijonon viimeinen merkki tallennetaan paikkaan 7.

```
char ammatti[] = "Palomies";
```

Nyt taas määrittelemme merkkijonotaulukon `ammatti`, johon on varattu tilaa juuri riittävästi että merkkijono ”Palomies” mahtuu taulukkoon. Kääntäjä laskee merkkijonon pituuden sekä lisää siihen loppumerkin tarvitseman tilan ja varaa tarvittavan taulukon.

```
char harrastus[20] = "autot";
```

Viimeisenä olemme luoneet merkkitaulukon `harrastus`, jossa on tilaa kahdellekymmenelle merkille, mutta josta olemme ottaneet käyttöön ainoastaan sen verran tilaa, että merkkijono ”autot” mahtuu taulukkoon. Lopuksi luomme vielä kokonaislukumuuttujan `koko`.

Seuraavilla kahdella rivillä suoritamme merkkijonon lukemisen käyttäjältä. Ensimmäinen `printf`-lause ei poikkea mitenkään erityisesti aiemmin käyttämistämme, mutta sen sijaan käsky, jolla luimme merkkijonon, vaatii hieman tarkastelua. `fgets`-funktio toimii siten, että määrittelemme ensin muuttujan johon tietoa luetaan, tämän jälkeen merkkimäärä, jonka korkeintaan luemme ja viimeisenä paikan, josta tietoa luetaan. Jos määräämme lukupaikaksi `stdin`, lukee kääntäjä merkkejä käyttäjän antamasta syötteestä. Huomaa lisäksi, että staattisia taulukoita käytettäessä ohjelma ei pysty tallentamaan kuin 20 merkkiä, joten esim. syötettä pyytäessä on hyvä kertoa syötteen maksimikoko. Yleisesti ottaen ohjelma ei saisi kaatua liian suuren syötteen antamiseen, jotta käyttäjältä ei vaadittaisi liikaa tarkkuutta ohjelman käytössä.

Seuraavalla rivillä käytämme ensimmäisen kerran `sizeof`-operaattoria. `sizeof` palauttaa numeroarvona tiedon siitä, kuinka monta tavua muistia parametrina annettu rakenne on. Tässä tapauksessa mittautimme merkkitaulukon `harrastus`. Kuten alempana olevasta tulostuksesta huomamme, on merkkitaulukon harrastus koko 20 tavua huolimatta siitä, että ainoastaan sen 5 ensimmäistä taulukkoalkiota on käytössä. Huomaa myös, että merkkijonon paikkamerkki tulostuskäskyssä on `%s`, kun yksittäisillä merkeillä se taas on `%c`.

Taulukko 8. Merkkitaulun ’harrastus’ sisältö.

Merkki	a	u	t	o	t	\0	-	.	.
alkionumero	0	1	2	3	4	5	6	7	8

Merkkitaulussa on myös toinen huomionarvoinen asia eli terminaattorimerkki `\0` (*null character, null terminator*). Tämä merkki kertoo käytännössä ohjelmalle, missä kohdassa varsinainen tietosisältö loppuu ja tyhjä tila alkaa. Koska luomme staattisen merkkitaulukon aina olemassa olevasta muistista, voi taulukkoon jäädä jotain arvoja aiemmin sitä muistipaikkaa käyttäneiltä ohjelmilta. Jos emme alusta merkkitaulukkoa tyhjäksi (tai kääntäjän luoma ohjelma ei tee sitä meidän puolestamme), on merkkijonojen loppu merkittävä terminaattorimerkillä. Koska loppumerkki on uniikki, voimme myöhemmin käyttää sitä taulukon lukemisessa lopetusehtona. Useimmiten merkkijonoja taulukkoon syötettäessä terminaattorimerkki lisätään automaattisesti ja merkin muistaminen pakottaa toimenpiteisiin usein vasta kun merkkijonoja läpikäydään tai muunnellaan merkki kerrallaan. Tästä tarkemmin seuraavassa esimerkissä.

Esimerkki 2.3: NULL-merkin poistamisen vaikutus merkkijonoon

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    char merkkijono[6] = "testi";

    /* Poistetaan automaattisesti sijoitettu
     * terminaattorimerkki lopusta. */
    merkkijono[5] = ' ';
    printf("%s\n", merkkijono);

    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./2-3
testi ¾äH$=

un@linux:~/un$
```

Kuinka koodi toimii

Tässä esimerkissä huomaamme ongelman. Koska merkkijonosta puuttuu terminaattorimerkki, aiheuttaa se ohjelmassa epätavallista käytöstä, koska varattuun taulukkoon on jäänyt tietoa aiemmin samaa muistialuetta käyttäneiltä ohjelmilta. Huomaa, että ongelma voi tosiaan näkyä tai olla näkymättä riippuen siitä, minkälaisessa käytössä olleelta alueelta muistia varataan ja siten tulosteen muotoa on mahdoton tietää. Ainoa tapa suojautua tältä on huolehtia siitä, että terminaattori (NULL)-merkki on aina sijoitettu oikealle paikalleen merkkijonon viimeiseksi merkiksi.

Merkkijonoille on myös olemassa perusfunktioita, joihin palaamme myöhemmin.

Tulostaminen ja sen muotoilu

C-kielessä tulostus tehdään tyypillisesti `printf`-funktiolla, joka tulostaa saamansa parametrit halutussa muodossa näytölle. `printf` on funkio, jonka ensimmäisenä parametrina on lainausmerkkien sisällä oleva tulostettava merkkijono.

```
float a=1, b=2;
printf("a=%d,\tb=%+6.3lf\n", a, b);
```

Tämä merkkijono sisältää tulostuvat merkit sekä tulostettavien arvojen tilalla niiden muotoilun; merkkijonon jälkeen on tulostettavat arvot (muuttujat) omina parametreinaan. Tiedon lukemiseen voidaan käyttää `scanf`-funktiota, joka toimii samalla idealla kuin `printf`-käsky ja käyttää myhös samoja muotoilumerkkejä kuin `printf`-käsky.

Jokaisen tulostettavan arvon muotoilu määritellään muotoilumerkkijonolla:

```
%liput leveys[.tarkkuus]tyyppi
```

Muotoilumerkkijono alkaa aina %-merkillä, jota seuraa mahdolliset liput (Taulukko 9). Lipuista tärkeimpiä ovat 0 (käytetään etunollia) ja - eli miinusmerkki (tasaus vasemmalle). Leveys-tieto kertoo tulostettavan kentän koko leveyden ja vaihtoehtoinen ”.tarkkuus” –kertoo tulostettavien desimaalien määrän. Muotoilumerkkijono päättyy tulostettavan arvon tyyppiin, joista yleisimpiä ovat d tai i kokonaisluvulle, f liukuluvulle ja s merkkijonolle (Taulukko 10). Esimerkiksi

```
float a = 1.12345;
printf(”%5.2f\n”, a);
```

tulostaa liukuluvun 2 desimaalilla 5 merkkiä leveään tilaan ja rivinvaihtomerkin sen perään.

Taulukko 9. Muotoilumerkkijonon liput

Lippu	**Tarkoitus**
#	Tulostetaan luku vaihtoehtoisessa muodossa
0	Käytetään etunollia
-	Tasaus vasemmalle (oletuksena oikealle)
+	Etumerkin tulostus positiivistenkin lukujen kanssa
välilyönti	Jos luku ei ala etumerkillä, niin välilyönti eteen

Taulukko 10. Yleisimmät muunnosmerkit

Muunnos	**Tarkoitus**
d, i	Etumerkillinen kokonaisluku desimaalimuodossa
f	Kaksoistarkkuuden liukuluku desimaalimuodossa
g	Kaksoistarkkuuden liukuluku eksponentti tai desimaalimuodossa
c	Etumerkitön merkki
s	Merkkijono
p	Tyypitön osoitin heksadesimaalimuodossa
o	Etumerkitön kokonaisluku oktaalimuodossa
u	Etumerkitön kokonaisluku desimaalimuodossa
x	Etumerkitön kokonaisluku heksadesimaalimuodossa
e	Kaksoistarkkuuden liukuluku eksponenttimuodossa
%	%-merkki
h	Seuraava kokonaisluku on tyyppiä short
l	Seuraava kokonaisluku on tyyppiä long

Muunnosmerkit kuten %d (kokonaisluku), %f (liukuluku) ja %s (merkkijono) määräävät käytännössä miten parametrina oleva tieto ymmärretään eli kuinka monta muistitavua kukin

parametri käyttää ja miten ko. muistialue on tulkittava – esim. kokonaislukuna (2 tavua), liukulukuna (32 tavua) vai merkkijonona (päättyy NULL-merkkiin).

Osoittimet

Osoitin määritellään normaalin muuttujan tapaan, mutta osoitin ei sisällä varsinaista tietoa kuten normaali muuttuja vaan muistiosoitteen, johon osoittimella viitataan. Osoitinmuuttujaa määriteltäessä on kerrottava, että se on osoitin ja se tehdään nimen eteen asetettavalla *-merkillä eli ”`tyyppi *muuttuja;`” tai esimerkkinä:

```
int *luku;
```

Myös tavallisen muuttujan muistipaikan osoitetta voidaan käyttää osoittimen arvona. Jos muuttujan nimen edessä käytetään etumerkkiä &, tarkoittaa tämä sitä, että muuttujan itsensä sijaan viitataan siihen muistipaikkaan, jossa muuttujan arvo sijaitsee.

```
int *luku;
int a=5;

luku=&a;   /* Nyt osoitinmuuttuja ’luku’ osoittaa muuttujan a
            * muistipaikkaan, jossa on arvo 5. */
```

Tämä viittaus toimii myös toisinpäin

```
int *luku;
int a, b;

luku=&b;  /* luku-osoitin laitetaan osoittamaan b:llä
           * varattuun muistipaikkaan. */
*luku=5;  /* luku-osoittimen osoittamaan muistipaikkaan
           * sijoitetaan arvo 5. */
a=*luku;  /* a saa arvokseen osoittimen luku osoittaman
           * muistipaikan sisällön, eli 5. */
```

Muuttujien ja osoitteiden kanssa toimiessa on muistettava, että osoittimen määrittely varaa muistia vain osoittimelle, ei itse muuttujalle eli datalle. Osoitin varaa muistia vain muutaman tavun verran kun taas muuttujan määrittelyn yhteydessä varataan muistia itse datalle tarvittava määrä. Kun kyseessä on suuret tietoalkiot (esim. valokuva jpeg-muodossa on helposti 1-10 megatavua), vie tiedon siirtäminen ja kopioiminen paljon aikaa ja tilaa. Jos taas tietoalkiot talletetaan vain yhteen paikkaan ja operoidaan sen jälkeen alkuperäiseen dataan osoittavilla osoittimilla, on tiedonkäsittely usein paljon nopeampaa ja säästetään muistia.

Osoittimien määrittelyn yhteydessä tulee olla tarkkana. Sijoitettaessa useamman osoittimen määrittely samalle riville pitää jokaisen osoittimen nimen yhteydessä olla *-merkki, sillä muutoin kääntäjä ei tee osoitinmuuttujaa vaan tavallisen muuttujan.

```
int *luku1, *luku2; /* määritellään kaksi osoitinmuuttujaa */
int *luku3, luku4; /* määritellään 1 osoitin ja 1 int muuttuja */
```

Tiedon luku scanf:lla

`scanf`-funktiolla voidaan lukea käyttäjän syöttämää tietoa näppäimistöltä. Käsky noudattaa samaa muotoilu-ideaa kuin `printf`-funktio (ks. yllä), mutta luettaessa tietoa `scanf`-funktio tarvitsee

tietoonsa niiden muistipaikkojen osoitteet, mihin luettavat arvot sijoitetaan muistissa. Näin ollen `scanf`-funktiolle on annettava parametrina muistipaikkojen osoitteet, ei muuttujia:

```
int a;
float b;
printf(”Anna kokonaisluku ja liukuluku välilyönnillä erotettuina: ”);
scanf(”%d  %f”, &a, &b);
printf(”Annoit luvut %d ja %f.\n”, a, b);
```

Osaamistavoitteet

Tämän toisen luvun jälkeen sinun tulee ymmärtää muuttujien ja tietotyyppien perusasiat sekä pystyä käyttämään niitä ohjelmissasi syötteiden vastaanottamiseen, tulostamiseen ja laskentaan. Perinteisesti totuttelua vaativia kohtia C-kielessä ovat merkkijonojen eli merkkitaulukoiden käsittely loppumerkin kanssa sekä tietotyyppien erilaiset toimintatavat, jotka korostuvat kokonaislukujaossa ja tulosteiden muotoilussa. Osoittimiin palataan tarkemmin myöhemmin, mutta keskeisyytensä takia niihin on syytä tutustua jo heti alussa.

Luku 3: Valinta- ja toistorakenteet, esikääntäjä

Tässä luvussa käymme läpi C-kielen ohjausrakenteet ja tähän mennessä esille tulleet esikääntäjään liittyvät asiat. Ohjausrakenteita ovat valinta- ja toistorakenteet sekä muutama yksittäinen käsky, joilla voidaan myös ohjata ohjelman suoritusta. Esikääntäjään liittyen käymme läpi muutaman keskeisen käskyn, direktiivin, joilla voimme ohjata ennen varsinaista ohjelman käännöstä tapahtuvaa esikäännösvaihetta.

Valintarakenteista

Python-ohjelmointikielessä valintarakenteiden käyttö on suoraviivaista, koska kieli ei tunne muita valintarakenteita kuin if-elif-else-rakenteen. C-kieli tukee oletuksena kolmea erilaista valintarakennetta, jotka ovat Pythonista tuttu `if-else`, suoraan valintaan perustuva `switch-case` sekä ehdoton hyppykäsky `goto`. Tutustumme näihin rakenteisiin yksinkertaisten esimerkkien avulla.

if-else

C-kielen `if`-rakenne toimii kuten Pythonin vastaava rakenne. Ainoa varsinaisesti muistettava ero on syntaksissa: kaikki valintaehdot tulee sijoittaa sulkeiden sisälle ja `else if`-osiossa käytetään nimenomaan `else if` –muotoa eikä lyhennettyä elif-muotoa. Koska syntaksi on hyvin samanlainen, voimme suoraan ottaa esimerkin ja tarkastella käytännön eroja puuttumatta sen tarkemmin itse perusrakenteeseen.

Esimerkki 3.1: if-else -rakenne

Esimerkkikoodi

```
#include <stdio.h>
int main(void) {
    int luku;
    int testi_1 = 100;
    int testi_2 = 1000;

    printf("Anna kokonaisluku: ");
    scanf("%d", &luku);

    if (luku < testi_1) {
        printf("Antamasi luku on pienempi kuin 100.\n");
    } else if ((luku >= testi_1) && (luku <= testi_2)) {
        printf("Antamasi luku on 100 ja 1000 välillä.\n");
    } else {
        printf("Antamasi luku on suurempi kuin 1000.\n");
    }

    printf("Lopetetaan.\n");
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./3-1
Anna kokonaisluku: 25
Antamasi luku on pienempi kuin 100.
Lopetetaan.

un@linux:~/un$ ./3-1
Anna kokonaisluku: 250
Antamasi luku on 100 ja 1000 välillä.
Lopetetaan.

un@linux:~/un$ ./3-1
Anna kokonaisluku: 1001
Antamasi luku on suurempi kuin 1000.
Lopetetaan.
```

Kuinka koodi toimii

Ohjelmaesimerkki aloitetaan tavalliseen tapaan `stdio.h` -kirjaston käyttöönotolla, `main`-funktion avaamisella sekä muuttujien esittelyllä. Lisäksi käytämme ohjelmassa jo aiemmasta esimerkistä tuttua kokonaisluvun pyytämistä käyttäjältä. Tämän jälkeen määrittelemme `if-else if-else`-rakenteen.

Kuten esimerkistä näemme, on mekanismi hyvin samankaltainen Python-ohjelmointikielen vastaavan esimerkin kanssa. Jokainen `if`-, `else if`- sekä `else`-rakenne avaa uuden koodiosion, jota merkitään C-kielessä aaltosuluilla. Lisäksi koodiosion valintaa koskevat ehdot sijoitetaan aina vähintään yksien normaalien kaarisulkeiden ”( )” sisäpuolelle. Huomaa, että C-kielessä sisennys ei ole merkitsevä tekijä, mutta hyvän ohjelmointityylin ja luettavuuden kannalta se edelleen erittäin hyödyllinen apuväline.

Esimerkki 3.2: Sisennyksien käytöstä

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    char sana[] = "kolmipyörä";

    if (sana[0] == 'k') {
        if (sana[1] == 'o') {
            if (sana[2] == 'l') {
                if (sana[3] == 'm') {
                    printf("Sana voisi olla %s.\n", sana);
                }
            }
        }
    }

/* Jos sulkujen kanssa ei ole tarkkana, voi tässä vahingossa
 * sulkea koko main-funktion väärässä paikassa tai jättää
 * if-lauseen auki! */

    printf("Lopetetaan.\n");
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./3-2
Sana voisi olla kolmipyörä.
Lopetetaan.
un@linux:~/un$
```

Huomioita koodista

Esimerkki havainnollistaa sisennyksen ja koodin huolellisen muotoilun merkitystä. Koska jokainen alkava koodiosio avaa uuden aaltosulun, ei useamman alkaneen sisennyksen jälkeen enää pysty silmämääräisesti erottamaan, onko sulkeita oikea määrä. Jos sulkeutuva aaltosulku on väärässä paikassa – tai sellainen puuttuu kokonaan – ei ohjelma enää käänny. Ja vaikka kääntyisikin, toimii se todennäköisesti väärin. Yksittäisen sulkumerkin paikan etsiminen pitkästä ja huonosti muotoillusta koodista on turhin mahdollinen tapa hukata aikaa ja nähdä vaivaa, kun koko ongelma voidaan välttää noudattamalla hyvää ohjelmointityyliä.

switch-case

Toinen C-kielen tyypillinen tapa suorittaa valinta on käyttää `switch-case` -rakennetta. Tätä rakennetta on helpointa kuvata valikkona, johon ohjelmoidaan kaikki valintamuuttujan vaihtoehdot,

joista sitten valitaan oikea haara. `switch-case` –rakennetta voi pitää redundanttina `if-else`-rakenteen kanssa, mutta oikein käytettynä se on hyvin näppärä ja nopea työkalu.

Esimerkki 3.3: switch-case -rakenne

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    int valinta;
    printf("Tee valinta (1-3): ");
    scanf("%d", &valinta);

    switch (valinta) {
        case 1:
            printf("Valitsit 1.\n");
            break;
        case 2:
            printf("Valitsit 2.\n");
            break;
        case 3:
            printf("Valitsit 3.\n");
            break;
        default:
            printf("En ymmärtänyt valintaa.\n");
            break;
    }

    printf("Lopetetaan.\n");
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./3-3
Tee valinta (1-3): 1
Valitsit 1.
Lopetetaan.

un@linux:~/un$ ./3-3
Tee valinta (1-3): 2
Valitsit 2.
Lopetetaan.

un@linux:~/un$ ./3-3
Tee valinta (1-3): 5
En ymmärtänyt valintaa.
Lopetetaan.

un@linux:~/un$
```

Kuinka koodi toimii

`switch`-rakenne toimii hieman erilaisella syntaksilla kuin `if-else` -rakenne. Ensinnäkin, `switch` itsessään ei muodosta kuin yhden varsinaisen koodiosion, joten aaltosulkujen sekamelskalta ei synny. Tämän lisäksi `case`-valinta vertaa ainoastaan muuttujan saamaa arvoa sekä annettua valintaa, joka tässä tapauksessa on numeroarvo 1, 2 tai 3. Vertailtavat tapaukset voivatkin olla numeroarvoja, yksittäisiä merkkejä taikka molempien yhdistelmiä. Erikoismerkit (!, ?, \ jne.) eivät `case`-valinnassa toimi.

`case`-valinta muodostaa teoriassa loogisen osion, mutta sen kanssa tulee muistaa kaksi asiaa. Ensinnäkin, kyseessä on yksi harvoista C-kielen lausekkeista, joka ei pääty puolipisteeseen tai aaltosulkuun vaan kaksoispisteeseen ja toiseksi ohjelmoijan tulee päättää `case`-osio `break`-käskyyn. `break`-käsky toimii C-kielessä samalla tavoin kuin Pythonin vastaava käsky, eli lopettaa käynnissä olevan rakenteen ja siirtyy rakennetta seuraavaan loogiseen lausekkeeseen. Mikäli `break`-käsky jätetään laittamatta, suorittaa `switch`-rakenne kaikki valittua `case`-valintaa seuraavat `case`-osiot seuraavaan `break`-käskyyn tai `switch`-rakenteen loppuun asti. Tyypillisesti tämä ei ole tarkoitus, joten ohjelmoijan on syytä varmistua siitä, että koodi toimii tässä kohdin halutulla tavalla. Lisäksi `switch-case` tukee `default`-valintaa, joka suoritetaan, mikäli yksikään määritelty `case`-valinta ei pidä paikkaansa. `default` –valinta on tavallisesti viimeisenä ja hyvän ohjelmointitavan mukaisesti sekin tulisi päättää `break`-käskyyn. Kannattaa muistaa, että `default` voi esiintyä myös muissakin väleissä kuten esimerkiksi rakenteen alussa. Tällöin myös `default`-osiossa on syytä tarkistaa `break`-käskyn käyttö, jotta rakenne toimii oikein.

goto-käskystä

C-kielessä on myös ohjausrakenne nimeltään `goto`. Tämä käsky toimii siten, että ohjelmakoodissa asetetaan nimettyjä lohkoja, joihin koodi pakotetaan hyppäämään `goto` *`lohkonnimi`* -käskyllä. Koska ratkaisu johtaa yhdeksässä tapauksessa kymmenestä täysin hallitsemattomaan koodisekamelskaan sekä on yleisesti ottaen huonoa ohjelmointityyliä, ei sitä käsitellä tämän enempää tässä oppaassa.

Jos jostain syystä haluat tietää käskystä enemmän esim. joutuessasi ylläpitämään ko. rakenteita sisältävää koodia, löytyy käskystä lisäinformaatiota normaaleissa lähdemateriaaleissa, esim. ((Alaoutinen 2005) ja (Kernighan ja Ritchie 1988)).

Toistorakenteista

Myös toistorakenteissa C-kieli tarjoaa enemmän vaihtoehtoja kuin Python. Pythonista tuttujen alkuehtoisen `for`- ja avoimen `while`-rakenteen lisäksi kielestä löytyy myös loppuehtoinen `do-while`. Lisäksi kieli tukee samoja toisto- ja ohjausrakenteiden ohjauskäskyjä kuten `continue` tai `break`. Seuraavaksi käymme läpi lyhyesti C-kielen toistorakenteet.

while-rakenne

`while`-toistorakenteen idea C-kielessä on sama kuin muissakin kielissä eli kierrosmäärää ei tarvitse määritellä etukäteen ja käyttäjä vastaa toistorakenteen lopettamisesta antamalla sopivan syötteen. Käytännössä `while`-rakenne on parhaimmillaan suoritettaessa toistoa, jonka kierrosmäärää ei etukäteen tiedetä.

Esimerkki 3.4: while-rakenne käytössä

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    int kierrosmaara = 5, kierros = 0;

    while (kierrosmaara > kierros) {
        printf("Olemme kierroksella %d!\n", kierros);

        /* Kasvatetaan kierroslukumittaria */
        kierros++;
    }

    printf("Lopetetaan tähän.\n");
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./3-4
Olemme kierroksella 0!
Olemme kierroksella 1!
Olemme kierroksella 2!
Olemme kierroksella 3!
Olemme kierroksella 4!
Lopetetaan tähän.
un@linux:~/un$
```

Kuinka koodi toimii

Ensinnäkin normaalien aloitustoimien lisäksi olemme tässä ensimmäistä kertaa käyttäneet monen muuttujan yhtäaikaista määrittelyä. Rivillä int kierrosmaara = 5, kierros = 0; määrittelemme kaksi int-muuttujaa, kierrosmaara ja kierros, joille lisäksi asetamme alkuarvot 5 ja 0. C-kielessä voimme määritellä monta samantyyppistä muuttujaa yhdellä rivillä erottelemalla ne pilkuilla toisistaan. Lisäksi olemme luonnollisesti käyttäneet while-rakennetta ohjelmakoodissa; tutkitaan sitä hieman tarkemmin.

`while`-rakenne ei poikkea merkittävästi Pythonin vastaavasta rakenteesta. `while`-käskylle annetaan toistoehto, tässä tapauksessa ”`kierrosmaara > kierros`”, jota ohjelma testaa aina toistorakenteen alussa. Mikäli ehto on tosi, jatketaan toistoa, muussa tapauksessa lopetetaan ja siirrytään `while`-rakennetta seuraavalle koodiriville. Lisäksi joudumme käsin lisäämään toistorakenteen katkaisua ohjaavaan `kierros`-muuttujaan arvoja, jotta ohjelmamme toimii oikein. Huomioi kuitenkin, että toisin kuin Python-kielessä, `while`-rakenteeseen ei voida enää liittää `else`-osiota.

for-rakenne

`for`-lauseen syntaksi poikkeaa C-kielessä jonkin verran Pythonista, vaikka käyttöperiaate onkin aivan sama. Käytännössä `for`-rakenne toimii näin:

```
for ([laskurin alustus]; [toistoehto]; [laskurin siirtymäväli])
```

eli periaatteessa siten, että ensin kerromme mitä muuttujaa käytämme kierroslaskurina (perinteisesti `i, j` ja `k`) ja minkä arvon ko. muuttuja saa alussa, minkä ehdon tulee täyttyä toiston jatkamiseksi ja minkä verran kierroslaskuri siirtyy yhden kierroksen aikana. Toistoehdosta tulee vielä muistaa, että se toimii siten, että toistoa jatketaan niin kauan kuin ehto on tosi ja katkaisu tapahtuu kierroksella, jolloin ehto muuttuu epätodeksi.

Esimerkki 3.5: for-rakenne käytössä

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    int i = 0;
    printf("for-rakenne laskee itse kierroksensa:\n");

    for (i = 0; i <= 10; i++) {
        /* Alussa i on 0, niin kauan kun i <= 10,
         * lisätään i:n arvoa yhdellä. */
        printf("%d ", i);
    }

    printf("\nLopetetaan tähän.\n");
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./3-5
for-rakenne laskee itse kierroksensa:
0 1 2 3 4 5 6 7 8 9 10
Lopetetaan tähän.
un@linux:~/un$
```

Kuinka koodi toimii

Koodi itsessään ei sisällä paljoakaan yllätyksiä. Käyttämämme `for`-rakenne määrittelee kierrosmuuttujan `i` sekä alustaa sen nollaksi. Tämän jälkeen ilmoitamme, että `for`-rakenne jatkuu niin kauan, kun `i` on pienempi tai yhtä suuri kuin 10 ja kerromme, että joka kierroksella `i`:n arvo kasvaa yhdellä (`i++`). Muilta osin koodin rakenne on vastaava kuin aiemmassa `while`-rakenteen esimerkissä. Myöskään `for`-rakenteeseen ei voida liittää `else`-osiota.

do-while-rakenne

`do-while` on toistorakenne, jota ei löydy Python-kielestä. `do-while` -toistorakenteesta hieman tavallisesta poikkeavan tekeekin se, että se on loppuehtoinen toistorakenne. Tämä tarkoittaa käytännössä sitä, että rakenteen **do-osio suoritetaan ainakin kerran riippumatta siitä, mitkä lopetusehdot ovat.** `do-while` -rakenne toimii muuten melko lailla samalla tavoin kuin normaali `while`-rakenne.

Esimerkki 3.6: do-while-rakenne käytännössä

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    int lopetus = 5, aloitus = 10;
    printf("do-osio toteutuu ainakin kerran.\n");

    do {
        if (lopetus < aloitus) {
            printf("Lopetus on valmiiksi pienempi kuin aloitus:\n");
        }
        printf("lopetus: %d ja aloitus: %d\n", lopetus, aloitus);
        aloitus++;
    } while (aloitus < lopetus);
    /* while-käsky tulee HETI do-osion perään
     * ja päättyy puolipisteeseen. */

    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./3-6
do-osio toteutuu ainakin kerran.
Lopetus on valmiiksi pienempi kuin aloitus:
lopetus: 5 ja aloitus: 10
un@linux:~/un$
```

Kuinka koodi toimii

Toisin kuin muut toistorakenteet, `do-while` ei ennen ensimmäisen kierroksen loppua tarkasta lainkaan lopetusehdon toteutumista. Esimerkkikoodissa toistoa oli tarkoitus jatkaa niin kauan kun `aloitus` on pienempi kuin `lopetus`, mutta vaikka jo toiston alkaessa luku oli kaksinkertainen, suoritettiin ensimmäinen kierros. `do-while` -rakenne onkin parhaimmillaan kun suoritetaan toistoja, joissa ensimmäinen kierros periaatteessa testaa onko jotain käytettävissä ja jatkaa toistoa loppuun asti. Tällaisia tilanteita esimerkiksi ovat puurakenteiden läpikäynti, tiedostosta merkkien lukeminen sekä dynaamisten rakenteiden selaaminen.

Muita ohjauskäskyjä

C-kielessä muita ohjauskäskyjä ovat `return`, `continue` ja `break`, jotka toimivat samalla tavoin kuin Pythonissa. `return`-lause lopettaa (ali)ohjelman suorituksen ja palauttaa kontrollin kutsuvaan ohjelmaan eikä `return`-lauseen jälkeen mahdollisesti olevia lauseita suoriteta koskaan. `continue`-käsky toistorakenteen sisällä lopettaa käynnissä olevan kierroksen ja siirtyy seuraavalle kierrokselle; `break` lopettaa sisimmän suoritettavana olleen toistorakenteen ja jatkaa loogisesti seuraavalta riviltä. Pythonin `pass`-käskyä C-kieli ei tunne.

Esimerkki 3.7: Ohjauskäskyt

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    int alku = 30, loppu = 100;

    do {
        alku++;
        if (alku % 2 != 0) {
            printf("Pariton luku, jatketaan suoraan uudelle ");
            printf("kierrokselle.\n");
            continue;
        }
        if (alku - 42 != 0) {
            printf("Erotus on %d\n", alku - 42);
        } else {
            printf("42 löytyi!\n");
            break;
        }
    } while (alku < loppu);
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./3-7
Pariton luku, jatketaan suoraan uudelle kierrokselle.
Erotus on -10
Pariton luku, jatketaan suoraan uudelle kierrokselle.
Erotus on -8
Pariton luku, jatketaan suoraan uudelle kierrokselle.
Erotus on -6
Pariton luku, jatketaan suoraan uudelle kierrokselle.
Erotus on -4
Pariton luku, jatketaan suoraan uudelle kierrokselle.
Erotus on -2
Pariton luku, jatketaan suoraan uudelle kierrokselle.
42 Löytyi!
```

```
un@linux:~/un$
```

Kuinka koodi toimii

Tässä esimerkkikoodissa otamme samalla toisen – tällä kertaa useamman kierroksen suorittavan – `do-while`–rakenteen. Ohjelman tehtävänä onkin testata onko muuttujan `alku` arvo jossain vaiheessa 42. Ensiksi testaamme onko luku parillinen: mikäli löydämme parittoman luvun, jatkamme suoraan seuraavalle kierrokselle `continue`-käskyllä. Mikäli meillä on parillinen luku, vähennämme siitä arvon 42 ja katsomme onko erotus 0. Mikäli näin ei ole, tulostamme erotuksen. Jos kuitenkin erotus on 0 ja luku on parillinen, voimme ilmoittaa käyttäjälle löytäneemme luvun 42 ja lopettaa toistorakenteen `break`-käskyllä. Muussa tapauksessa jatkaisimme toistorakennetta siihen asti kunnes muuttujan `alku` arvo olisi sama tai enemmän kuin muuttujan `loppu` arvo.

Esikääntäjästä

C-kieli on käännettävä kieli toisin kuin Python, joka on tulkattava kieli. Tämä tarkoittaa sitä, että ohjelman kirjoittamisen jälkeen lähdekoodi tulee kääntää suoritettavaksi ohjelmasi, joka voidaan suorittaa erillisellä käskyllä. Käännöksen aikana kääntäjä käy lähdekoodin läpi useita kertoja ja tekee tiettyjä toimenpiteitä kullakin kierroksella. Näillä toimenpiteillä säädetään lopputuloksen eli suoritettavan ohjelman ominaisuuksiin eli esim. ohjelman tarvitsemaa muistimäärää voidaan minimoida, suoritusnopeutta voidaan maksimoida, käännökseen voidaan ottaa mukaan virheiden etsimistä helpottavia debuggaustietoja yms. Käännösprosessin ensimmäinen vaihe on esikääntäjän läpikäynti, joka tehdään ennen varsinaista käännöstä ja jonka aikana suoritetaan esikääntäjälle suunnatut toiminnot. Yleisimpiä esikääntäjän toimintaa ohjaavia käskyjä ovat `include`, `define` ja `if-endif` –käskyt (direktiivit), joita edeltää risuaita-merkki (#, esim. `#include`).

#include

Monipuolisempien ohjelmien toteuttaminen vaatii tavallisesti ulkopuolisten funktiokirjastojen käyttämistä. Esimerkiksi Python-ohjelmointikielessä funktiokirjastojen avulla pystyimme luomaan graafisia käyttöliittymiä, suorittamaan tieteellistä laskentaa ja jopa käyttämään tietokoneen verkkoyhteyttä ilman että jouduimme itse kirjoittamaan merkittävästi uutta koodia.

C-kieli on pidetty tarkoituksellisesti mahdollisimman suppeana ja vain keskeisimmät toiminnot on sisällytetty itse C-kieleen. Näin ollen C-kielisissä ohjelmissa käytetään varsin paljon kirjastoja aina sen mukaan, minkälaisia laajennuksia eli lisäominaisuuksia tarvitaan. Esimerkiksi aiemmissa esimerkkitehtävissä olemme käyttäneet toistuvasti kirjastoa nimeltä `stdio.h`, joka sisältää normaalit luku- ja kirjoitusfunktiot näytölle sekä tiedostoihin (stdio eli standard input/output). Tämä tehdään siis käskyllä

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

Vastaavasti, jos haluaisimme ottaa käyttöön esimerkiksi kirjaston `stdlib.h`, joka sisältää laskentaa, muistinkäsittelyä, tyyppimuunnoksia sekä muita hyödyllisiä toimintoja (standard library), voisimme tehdä sen komennolla

```
#include <stdlib.h>
```

Huomaa, että C-kielessä meidän ei tarvitse erikseen kertoa, minkä kirjaston funktiota aiomme käyttää. Esimerkiksi merkkijonojen lukeminen käyttäjältä voidaan toteuttaa käskyllä `scanf`, joka on `stdio.h`-kirjaston funktio. Jos käyttäisimme Pythonia, joutuisimme viittaamaan tähän komennolla `stdio.scanf`, mutta C-kielessä riittää pelkkä funktion nimi. Luonnollisesti tämä tarkoittaa, että funktioilla ei saa olla samoja nimiä kuin jo toisessa käytössä olevassa kirjastossa on.

Peruskirjasto onkin suunniteltu siten, ettei näin pääse tapahtumaan. Taulukko 11 nimeää vielä muutamia hyödyllisiä funktiokirjastoja.

Taulukko 11. Muutamia hyödyllisiä funktiokirjastoja

Nimi	Sisältö lyhyesti
stdio.h	Luku- ja kirjoitusfunktioita, tiedostonkäsittely.
stdlib.h	Tyyppimuunnokset, muistinkäsittely, järjestelmäkomentoja; yleishyödyllisiä funktioita.
string.h	Merkkijonojen käsittelyyn tarkoitettuja funktioita.
time.h	Kello- ja kalenterifunktiot.
math.h	Matemaattisia funktioita.

Emme tässä osiossa käy kirjastoja tai niiden sisältöä sen tarkemmin läpi vaan tutustumme niistä löytyviin funktioihin sitä mukaa kun tarvitsemme niitä. Kattavammin kirjastojen sisältö on esitelty liitteessä 1. Jos haluat lisätietoa kirjastoista, voit tutustua C-kielen perusfunktioihin mm. Eric Hussin kirjoittaman C-referenssioppaan avulla (Huss 1997). Opas on englanninkielinen.

Omien funktiokirjastojen tekemisestä puhumme lisää oppaan toisessa osassa.

#define ja kiintoarvot

C-kieli tarjoaa mahdollisuuden vakioiden määrittelyyn, joita käyttäjä ei voi muuttaa ohjelman suorituksen aikana. Vakioita eli kiintoarvoja voidaan määritellä `#define`-määrittelyllä. `define`:llä tehdyt vakiot ovat merkkijonoliteraaleja eli esikääntäjä korvaa annetut merkkijonot (vakiot) niiden arvoilla. Käytännössä tämä tarkoittaa sitä, että annamme lähdekoodin alussa käskyn

```
#define nimi arvo
```

jossa `nimi` tullaan aina jatkossa ymmärtämään `arvo:`na. Tällä tavoin voimme esimerkiksi määritellä arvot TRUE ja FALSE määräämällä nimelle TRUE arvon 1 ja FALSE:lle arvon 0. Helpointa tämä on demonstroida esimerkin avulla:

Esimerkki 3.8: #define-määrittely

Esimerkkikoodi

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define NUMEROARVO 99999

int main(void) {
    int toista = TRUE;
    int testi = 0;

    printf("%d\n", NUMEROARVO);
```

```
    if (toista == TRUE) {
        printf("Toimii!\n");
    }
    if (testi == FALSE) {
        printf("#define-käskyn määritys toimii myös ");
        printf("numeroarvon kanssa.\n");
    }

    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./3-8
99999
Toimii!
#define-käskyn määritys toimii myös numeroarvon kanssa.
un@linux:~/un$
```

Kuinka koodi toimii

Tässä esimerkissä olemme luoneet kiintoarvot TRUE, FALSE sekä NUMEROARVO, ja antaneet niille arvot 1, 0 sekä 99999. Kuten näemme, kiintoarvot toimivat käytännössä siten, että ne ovat eräänlaisia muuttujia, joiden arvoa ei pystytä ajonaikaisesti muuttamaan. Parasta näissä onkin se, että myös kääntäjä ymmärtää kiintoarvot numeroina, joten voimme esimerkiksi luoda merkkitaulukon komennolla `char taulu[NUMEROARVO]`, joka normaaleilla muuttujilla ei onnistu.

Erityisen hyödyllisiä kiintoarvot ovat nimenomaan ”asetusarvoina” esimerkiksi taulukon koolle tai toistorakenteen alku- ja loppuehtoina. Kun määrittelemme arvon yhdessä paikassa lähdekoodin alussa, voimme asetuksien muutamista varten muuttaa pelkkää `#define` -käskyn arvoa sen sijaan, että joutuisimme käsin muuttamaan jokaisen kohdan missä ko. arvoa on käytetty. Tämä vähentää myös virheiden todennäköisyyttä, koska pidemmässä lähdekoodissa yhden muutettavan kohdan huomaamatta jättäminen muuttuu hyvinkin todennäköiseksi ongelmaksi.

C-kielen uudemmissa versioissa merkkijonoliteraalien rinnalle on tullut myös aidot vakiomuuttujat, joita ei siis korvata esikäännösvaiheessa vaan ne menevät kääntäjälle asti ja niille voidaan suorittaa esimerkiksi tyyppitarkistuksia. Nämä vakiomuuttujat määritellään samalla tavoin kuin muuttujat, mutta niiden eteen lisätään määre `const` eli vakio. Vakioita voi määritellä myös `enum`-käskyllä, josta tarkemmin myöhemmin.

```
const int lahtoarvo = 1;
```

#if 0 … #endif

Luentoesimerkkien kirjoittamisen yhteydessä halutaan usein poistaa käytöstä monta riviä koodia, jotta jotain uutta asiaa voidaan tutkia ikään kuin ilman aiemmin kirjoitettua koodia. C-kielessä kommentit ovat monirivisiä, joten periaatteessa yhdellä kommentilla voidaan poistaa monta koodiriviä. Ongelmia tulee kuitenkin siinä tapauksessa, että poistettava koodi sisältää kommenttien loppumerkin. Esikääntäjän `if-end` –direktiivit mahdollistavat haluttujen rivien mukanaolon

lähdekoodissa, mutta ko. rivien poistamisen esikääntäjän suorittamisen yhteydessä, jolloin ne eivät vaikuta enää ajettavassa ohjelmassa.

```
#if 0
/* Nämä rivit näkyvät lähdekoodissa.
 * Esikäsittelijä poistaa kuitenkin kaikki if 0 ja endif –määritteiden
 * väliset merkit.
 * Vaihtamalla 0:n tilalle esim. 1:n, menevät rivit myös kääntäjälle.
 */
#endif
```

Osaamistavoitteet

Tämä luvun jälkeen sinun tulee ymmärtää C-kielen valinta- ja toistorakenteet sekä esikääntäjän määritteet. Eli nyt pystyt määrittelemään vakioita, ottamaan käyttöön olemassa olevia kirjastoja ja kirjoittamaan ohjelmia, joissa on sekä valinta- että toistorakenteita. Läpikäydyt valintarakenteet käsittivät `if` ja `switch` –rakenteet kun taas toistorakenteina käytiin läpi `for`, `while` ja `do-while` –rakenteet. Esikääntäjän käskyistä tutustuimme `include`, `define` ja `if-end` –käskyihin sekä vakioiden määrittelyyn C-kielen `const` –käskyllä. Ohjelmien toiminnallisuuden määrä kasvaa nyt nopeasti.

Luku 4: Tiedostot ja aliohjelmat

Tiedostojen käsittely: lukeminen ja kirjoittaminen

Ulkoisia tiedostoja voidaan käyttää ohjelmoinnissa moninaisiin tarkoituksiin, mutta tavallisimmin ne toimivat pysyväismuistina, josta luetaan ohjelmalle annettuja asetuksia tai aiemmin käsiteltyä tietoa. C-kielessä tiedostoja voidaan käsitellä luku- ja kirjoitusoperaatioilla lähes yhtä helposti kuin tietoa tulostetaan näytölle ja luetaan näppäimistöltä ohjelman käyttöliittymän avulla. Käytännössä ulkoisen tiedoston käyttäminen on samanlaista kuin Pythonissa: tiedosto avataan haluttua käyttötarkoitusta varten sopivaan tilaan, tiedostoa luetaan tai kirjoitetaan ja käytön lopuksi tiedosto suljetaan. Lisäksi C-kielestä löytyy monet muutkin Python-tiedostonkäsittelystä tutut elementit, kuten esimerkiksi tiedosto-osoittimet (ns. *kirjanmerkki*).

Tiedostojen käsittely C-kielellä on monipuolisempaa kuin esimerkiksi Pythonilla. Normaalien ASCII-merkkien lisäksi tiedostoja voidaan helposti käsitellä binaarimuodossa, eikä C-kieli itse asiassa tee merkittävää eroa näiden kahden lähestymistavan välille. Sen sijaan käyttöjärjestelmien välillä eroa jonkin verran esiintyy: Windows- ja Linux-ympäristöt käsittelevät joitain erikoismerkkejä eri tavoilla. Windows esimerkiksi olettaa ASCII-tiedoston aina loppuvat EOF-merkkiin, kun taas Linux-järjestelmissä tällaista oletusta ei ole. Tavallisesti normaaleilla tekstitiedostoilla näihin poikkeamiin ei kuitenkaan törmätä ja binaarimoodissa työskennellessä ongelmia ei myöskään tavallisesti esiinny.

Ensimmäisenä voimme tutustua tiedostojen lukemiseen. Olemme ennen esimerkkejä tehneet samaan hakemistoon lähdekooditiedoston kanssa tekstitiedoston nimeltä *tiedosto.txt*, johon olemme tallentaneet muutaman merkkirivin sekä numeroja. Seuraavassa esimerkissä avaamme tiedoston, luemme sen sisällön ja tulostamme sen ruudulle. Tiedostossa on yhdellä rivillä korkeintaan 10 näkyvää merkkiä ja rivinvaihtomerkki, viimeinen rivi on tyhjä.

Esimerkki 4.1: Tiedostosta lukeminen

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    char muisti[12];
    /* Luodaan tiedostokahva eli osoitin FILE-tyyppiseen muuttujaan. */
    FILE *tiedosto;

    tiedosto = fopen("tiedosto.txt", "r"); /* Avataan tiedosto. */
    printf("Tiedoston sisältö:\n"); /* Virheentarkastus puuttuu! */

    /* Tiedoston kaikkien rivien luku ja tulostus. */
    while (fgets(muisti, 11, tiedosto) != NULL) {
        printf("%s\n", muisti);
    }

    fclose(tiedosto);
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./4-1
Tiedoston sisältö:
merkkijono
1234567890
karhennettu
turtana
0123456789
un@linux:~/un$
```

Kuinka koodi toimii

Ohjelma alkaa määrittelemällä 12 merkin kokoisen lukumuistin nimeltä `muisti` sekä tiedostokahvan `tiedosto` käskyllä `FILE *tiedosto;`. Tämän jälkeen avaamme haluamamme tiedoston *tiedosto.txt* lukumoodiin ”r”, eli tilaan josta voimme ainoastaan lukea tiedoston sisältämää tietoa. Tämän jälkeen suoritamme itse tiedoston lukemisen `while`-toistorakenteella. Huomaa, että tiedoston avaaminen voi epäonnistui useista syistä, joten normaalisti tiedoston avaaminen onnistuminen on aina tarkistettava – palaamme tähän asiaan vähän myöhemmin.

Koska emme tiedä, kuinka pitkä tiedosto on, määrittelemme lukemisen lopetusehdoksi tilanteen, jossa tiedosto on luettu loppuun. Tämä onnistuu käyttämällä `fgets`-funktiota, joka lukee ja palauttaa yhden rivin tiedostosta. Rivin pituus on rajoitettu 11 merkkiin, jotta se mahtuu `muisti`-muuttujaan NULL-merkin kanssa; mikäli `fgets`-funktio ei pysty lukemaan tiedostosta riviä eli tiedosto-osoitin on tiedoston lopussa, palauttaa se NULL-merkin tiedoston loppumisen merkiksi. Näin ollen `while`-rakenteessa ei tarvita muuta kuin varsinainen toiminnallisuus eli tässä tapauksessa rivin tulostus näytölle. Lopuksi vielä suljemme tiedoston ja lopetamme ohjelman.

Kuten tästä esimerkistä huomasimme, on C-kielen tapa käsitellä tiedostoja hyvin samankaltainen kuin Pythonissa. Tiedosto avataan tarpeita vastaavalla moodilla tiedostokahvaan, tiedostosta luetaan tietoa muuttujaan ja lopuksi, kun tiedostoa ei enää tarvita, se suljetaan. Varsinaisesti tärkein ero onkin siinä, että C-kielessä pystymme kertomaan, millaista tietoa tiedostosta luemme. Tiedostojen lukemiseen on myös tarjolla useita eri funktioita kuten esimerkiksi scanf-funktiota vastaava fscanf eli file-scanf. Edellä on kuitenkin käytetty fgets-funktiota, koska merkkirivien lukeminen on sillä suoraviivaista edellä olevan esimerkin mukaisesti. Seuraavaksi tutustumme tiedostojen kirjoittamiseen toisella lyhyellä esimerkillä.

Esimerkki 4.2: Tiedostoon kirjoittaminen

Esimerkkikoodi

```
#include <stdio.h>

int main(void) {
    char muisti[50];
    FILE *tiedosto;
```

```
    tiedosto = fopen("tuloste.txt", "w");/*Virheentarkastus puuttuu!*/
    printf("Mitä haluat tiedostoon kirjoittaa?\n");
    scanf("%s", &muisti);

    fprintf(tiedosto, "%s", muisti);
    fclose(tiedosto);
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./4-2
Mitä haluat tiedostoon kirjoittaa?
Robottikana
un@linux:~/un$
```

Kuinka koodi toimii

Tämä koodi ei itsessään poikkea paljoakaan aiemmasta tiedostonluku-esimerkistä. Itse tiedostoon kirjoittaminen tapahtuu funktiolla `fprintf` (file-printf), joka toimii aivan kuten normaalin ruudulle tulostuksen toteuttava `printf`-funktio. Tosin nyt joudumme antamaan tiedostokahvan, johon tietoa kirjoitetaan. Huomaa lisäksi, että tällä kertaa käytimme kirjoitustilaa ”`w`”, joka automaattisesti luo uuden tai tuhoaa aikaisemman samannimisen tiedoston `fopen`:n yhteydessä eikä tiedostossa ole muuta kuin tuo yksi sana ”Robottikana” – `fprintf`-kirjoitti tiedostoon vain tuon sanan, ei rivinvaihtomerkkiä tai tiedoston loppumerkkiä (esim. EOF, end of file). Tästäkin esimerkistä puuttui vielä tiedoston aukeamisen onnistumisen tarkastus, joka pitää normaalisti aina tehdä.

Binaaritiedostojen käsittely

Binaaritiedostojen käsittely noudattaa samoja periaatteita kuin tekstitiedostojen käsittely eli tiedosto on avattava ennen käsittelyä ja suljettava sen jälkeen sekä tiedoston avaamisen yhteydessä on varmistuttava sen onnistumisesta. Binaaritiedoston luku ja kirjoittaminen tapahtuu omilla funktioilla, jotka käsittelevät tietoa binaaridatana.

Tekstitiedostoa käsittelevästä ohjelmasta saadaan binaaridataa käsittelevä ohjelma vaihtamalla kirjoittamisen yhteydessä moodiksi ”`wb`” (write binary) ja lukemisen yhteydessä moodiksi ”`rb`” (read binary). Alla on esimerkit tiedon kirjoittamiseen ja lukemiseen käytettävistä `fwrite`- ja `fread`-funktioista `tiedosto`-nimisen FILE* -muuttujan kanssa. Ensimmäinen esimerkki kirjoittaa tiedostoon luvut 0-9 binaarimuodossa ja vastaavasti alempi lukee binaaritiedostosta 10 kokonaislukua sekä tulostaa ne näytölle. Molemmissa funktiokutsuissa on parametri 1, joka kertoo käsiteltävien alkioiden määrän – nyt jokainen funktiokutsu käsittelee yhtä alkiota kerrallaan.

```
        for (i=0; i < 10; i++)
                fwrite(&i, sizeof(i), 1, tiedosto);


        for (i=0; i < 10; i++) {
```

```
        fread(&luku, sizeof(luku), 1, tiedosto);
        printf("%d ", luku);
}
```

Tiedostojen käsittelyyn käytettäviä funktioita

Tässä luvussa käydään läpi kootusti C-kielen tiedostonkäsittelyyn liittyvät funktiot, joita esittelemme jatkossa lisää aina sitä mukaa kun niitä tarvitsemme.

tietovirta=fopen(tiedosto, moodi);

- Avaa tiedoston *tiedosto* käyttötilaan *moodi*.
- *tietovirta* on `FILE *` -tyyppinen osoitinmuuttuja, jolla avaamisen jälkeen viitataan tiedostoon. Mikäli tiedoston avaaminen epäonnistuu, palauttaa `fopen` NULL-merkin.
- *tiedosto* on avattavan tiedoston nimi, joko merkkijonomuuttuja tai merkkijono.
- *moodi* kertoo mihin tarkoitukseen tiedosto avataan. **"r"** tarkoittaa lukemista, **"w"** kirjoittamista, **"a"** kirjoituksen jatkamista. Binaaritiedostoja käsiteltäessä moodiin liitetään **b** ("rb"). Haluttaessa sekä lukea että kirjoittaa samanaikaisesti, lisätään vielä **+** ("r+").
- Avattaessa tiedosto **+** -moodissa **r**:n ja **w**:n merkitys hieman muuttuu. Käytettäessä **r**:ää avataan olemassa oleva tiedosto tai luodaan uusi. **w**:llä luodaan aina uusi tiedosto. Jos tiedosto oli olemassa, se tuhoutuu.

fclose(tietovirta);

- Sulkee tiedoston *tietovirta*.

fscanf(tietovirta, formaattimerkkijono, muuttujalista);

fprintf(tietovirta, formaattimerkkijono, muuttujalista);

- `fscanf` lukee tiedostoa *tietovirta* ja `fprintf` kirjoittaa sinne. Muuten vastaavat `printf`:ää ja `scanf`:ää.

merkki=fgetc(tietovirta);

fputc(merkki, tietovirta);

- `fgetc` lukee yhden merkin tietovirrasta ja `fputc` kirjoittaa sinne.

fgets(merkkitaulukko, merkkien enimmäismäärä, tietovirta);

fputs(merkkitaulukko, tietovirta);

- `fgets` lukee tietovirrasta rivin tai merkkien enimmäismäärän verran merkkejä ja sijoittaa ne merkkitaulukkoon. `fputs` kirjoittaa merkkitaulukon tietovirtaan.

rewind(tietovirta);

- Siirtää tiedosto-osoittimen tiedoston alkuun.

sijainti=ftell(tietovirta);

- *sijainti* on pitkä kokonaisluku (long int), johon tallentuu tiedoston senhetkinen käsittelykohta.

fseek(tietovirta, sijainti, alku);

- Tietovirran käsittelykohta siirtyy sijainnin osoittamaan kohtaan. alku kertoo mistä sijaintia aletaan laskea. SEEK_SET tarkoittaa tiedoston alkua, SEEK_CUR nykyistä kohtaa ja SEEK_END tiedoston loppua.

feof(tietovirta);

- Jos tietovirtaa luettaessa on tullut vastaan tiedoston loppu, funktio palauttaa nollasta poikkeavan arvon, muulloin nollan.

fread(osoitin, koko, määrä, tietovirta);

fwrite(osoitin, koko, määrä, tietovirta);

- Nämä funktiot ovat binaaritiedostojen käsittelyyn:
    - `fread` lukee tietovirrasta koko-muuttujan kokoisia lohkoja määrä kappaletta ja sijoittaa ne osoittimen osoittamaan muistipaikkaan.
    - `fwrite` kirjoittaa tietovirtaan osoittimen osoittamassa muistipaikassa olevan tiedon, joka on koko-muuttujan kokoisissa lohkoissa määrä kappaletta.

remove(tiedosto);

- Hävittää tiedoston.

rename(nimi, uusi_nimi);

- Vaihtaa tiedoston nimen.

Aliohjelmat, tiedonvälitys ja osoittimet

Aina kun ryhdymme rakentamaan suurempia ohjelmia, tulemme väistämättä tilanteeseen, jossa joudumme miettimään ohjelman toimintaa. Useimmiten hyötyisimme mahdollisuudesta käyttää uudelleen aiempaa koodia useaan kertaan toistuvien tehtävien suorittamiseen, toisinaan myös eri tehtävien jakaminen loogisiin kokonaisuuksiin helpottaisi ohjelman toteuttamista sekä ylläpitoa. Tämä kaikki on mahdollista aliohjelmien avulla. C-kielessä aliohjelmat palauttavat yleensä jotain kutsuvaan ohjelmaan eli ne ovat funktioita. Aliohjelmien käyttö edellyttää tiedonvälitystä aliohjelmien ja kutsuvien ohjelmien välillä, jossa tarvitaan C-kielessä usein myös osoittimia.

Aliohjelman toteuttaminen

Kuten olet jo varmaan huomannut, on C-ohjelmassa aina vähintään yksi funktio, `main`. Tämä funktio on pääohjelma, jota kutsutaan ohjelman käynnistyessä ja jonka lopettaminen sammuttaa ohjelman. C-kielessä aliohjelmien toteuttaminen vaatii hieman enemmän tarkkuutta kuin esimerkiksi Pythonissa. Ensinnäkin, me joudumme päättämään funktion paluuarvon tyypin etukäteen; jos haluamme ohjelman palauttavan kokonaislukuvastauksen, on funktion tyyppi yleensä `int`, jos liukuluvun niin tyyppi on yleensä `float`. Jos taas haluamme merkkejä, käytämme tyyppiä `char`. Lisäksi, mikäli emme aio palauttaa mitään arvoja, on funktion tyypiksi laitettava tyhjä eli `void`. Muista siis että aliohjelma, jonka tyyppi on `void`, ei voi palauttaa minkäänlaista arvoa ja että funktion, jolla on jokin ei-tyhjä tyyppi, olisi hyvä palauttaa jotain. Huomaa myös, että `main`-funktion tyyppi voi olla `void` ja tällöin sekään ei palauta minkäänlaista arvoa käyttöjärjestelmälle ohjelman lopettaessa toimintansa. Kääntäjä antaa tästä varoituksen.

Toinen vaatimus aliohjelmien käytölle on se, että aliohjelma on esiteltävä ennen sen käyttöä. Kääntäjälle kerrotaan siis etukäteen, minkä nimisiä aliohjelmia aiomme käyttää, mitä ne palauttavat ja millaisilla parametreilla niitä kutsutaan eli ne ”esitellään”. Huomaa, että esittelyä ei tarvita, mikäli aliohjelmaa käytetään koodissa vasta sen määrittelyn eli sen toiminnan määrittelevän koodin jälkeen, mutta on turvallisempaa esitellä kaikki aliohjelmat lähdekoodin alussa, jolloin niiden kirjoitusjärjestyksellä ei ole merkitystä. Tutustumme aliohjelmien tekemiseen ja käyttämiseen esimerkkien avulla; ensiksi tarkastelemme pelkästään aliohjelmien määrittelyä, esittelyä sekä kutsumista. Toisessa esimerkissä keskitymme paluuarvoihin ja parametreihin.

Esimerkki 4.3: Aliohjelman määrittely ja kutsuminen

Esimerkkikoodi

```
#include <stdio.h>

/* demo-aliohjelman esittely */
void demo(void);

int main(void) {
    printf("Tämä tulee pääohjelmasta!\n");
    /* Kutsutaan aliohjelmaa */
    demo();

    return 0;
}
```

```
/* Kirjoitetaan aliohjelman koodi eli sen määrittely */
void demo(void) {
    printf("Tämä tulee aliohjelmasta!\n");
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./4-3
Tämä tulee pääohjelmasta!
Tämä tulee aliohjelmasta!
un@linux:~/un$
```

Kuinka koodi toimii

Ohjelmakoodi alkaa aiemmin mainitulla aliohjelman esittelyllä, joka siis käytännössä tarkoittaa, että kääntäjälle kerrotaan heti ohjelman alussa aliohjelman paluuarvo, nimi sekä vastaanotettavat parametrit. Käytännössä aliohjelman esittely vastaakin täysin sen määrittelyn ensimmäistä riviä puolipistettä lukuun ottamatta.

Koodin lopussa on itse aliohjelman määrittely. Aliohjelma on rakenteellisesti samanlainen kuin pääohjelma, eli se toteutetaan aliohjelman nimen ja parametrien jälkeen tulevien aaltosulkeiden sisällä. Mikäli aliohjelma palauttaa jotain, päättyy se `return`-käskyyn aivan kuin `main`-funktiokin. Tässä koodissa olemmekin määritelleet aliohjelman nimeltä `demo`, joka ei anna palautusarvoa (ensimmäinen `void`) eikä vastaanota parametreja (sulkeissa oleva toinen `void`). Kun olemme määritelleet aliohjelman, voimme kutsua sitä kirjoittamalla aliohjelman nimen ja sulkeet, eli tässä tapauksessa demo();. Tässä tapauksessa aliohjelmalla ei ole parametreja, mutta jos niitä olisi annettu, olisi ne laitettu kutsussa sulkeiden sisään siinä järjestyksessä kun ne halutaan aliohjelmalle välittää.

Parametrien välitys ja paluuarvo

Aliohjelmalle voidaan välittää tietoa kuten yhteenlaskettavia lukuja tai tutkittavia merkkejä parametreina. Aliohjelman vastaanottamat parametrit tulee olla mukana sekä aliohjelman esittelyssä että sen määrittelyssä. Käytännössä tämä tarkoittaakin sitä, että aliohjelman esittelyn – ja samalla myös kutsun – syntaksi on seuraavanlainen:

```
palautustyyppi nimi(parametrin 1 tyyppi ja nimi, parametrin 2
tyyppi ja nimi,…);
```

Eli vaikkapa

```
int laskukone(int luku_1, int luku_2);
```

joka tarkoittaisi, että meillä on `laskukone`-niminen funktio, joka vastaanottaa kaksi integer-kokonaislukua ja palauttaa integer-arvon. Tällöin esimerkiksi kutsu

```
tulos = laskukone(numero1, numero2);
```

missä `tulos`, `numero1` ja `numero2` on integer-muuttujia, lähettäisi muuttujien `numero1` ja `numero2` arvot laskettavaksi funktioon `laskukone` ja tallentaisi paluuarvon muuttujaan `tulos`.

Huomaa, että tässä yhteydessä esiteltävien parametrien arvoja ei voi muuttaa aliohjelmassa. Tästä johtuen tällä tavoin määriteltyjä parametreja kutsutaan arvoparametreiksi. C-kielessä on olemassa myös keino parametrien arvojen muuttamisen mahdollistamiseksi ja palaamme siihen kohta muuttujaparametrien yhteydessä.

Esimerkki 4.4: Parametrit ja paluuarvot

Esimerkkikoodi

```
#include <stdio.h>

/* Esitellään funktio */
int summaa(int luku1, int luku2);

int main(void) {
    int arvo1 = 1, arvo2 = 2;
    int vastaus;

    /* Kutsutaan funktiota ja tallennetaan paluuarvo muuttujaan */
    vastaus = summaa(arvo1, arvo2);
    printf("Parametrien 1 ja 2 summa on %d.\n", vastaus);

    return 0;
}

/* Kirjoitetaan funktion koodi eli sen määrittely */
int summaa(int luku1, int luku2) {
    int tulos;
    tulos = luku1 + luku2;
    return tulos;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./4-4
Parametrien 1 ja 2 summa on 3.
un@linux:~/un$
```

Kuinka koodi toimii

Tällä kertaa olemme antaneet funktiokutsulle parametreina muuttujien `arvo1` ja `arvo2` lukuarvot ja tallennamme funktion paluuarvon muuttujaan `vastaus` komennolla

```
vastaus = summaa(arvo1, arvo2);
```

Annamme funktiolle parametrit, jotka sitten lasketaan yhteen ja paluuarvo palautetaan takaisin pääohjelmalle. Lopputuloksena saamme muuttujaan `vastaus` laskutoimituksen tuloksen. Huomaa, että tällä kertaa funktion tyypiksi valitsimme `integer`-tyypin, koska funktion tulee palauttaa kutsuvalle ohjelmalle kokonaislukuarvo.

Tyypillisesti kaikki aliohjelmat esitellään tiedoston alussa ja sen jälkeen tulee pääohjelma. Pääohjelman jälkeen aliohjelmat määritellään yleensä siinä järjestyksessä kun niitä kutsutaan ohjelmassa. Ohjelmien monimutkaistuessa ja koon kasvaessa tämä jää usein tavoitteeksi, mutta tämä rakenne tarjoaa kuitenkin ennakoitavan lähtökohdan koko ohjelman rakenteen ymmärtämiselle.

Tunnusten näkyvyys

Tässä vaiheessa on syytä palauttaa mieliin Pythonissa vastaan tullut nimiavaruus-asia ja katsoa sitä C-kielen näkökulmasta eli tunnusten näkyvyyden kannalta. Tunnusten näkyvyys tarkoittaa sitä, että ohjelman tunnukset (muuttujat ja aliohjelmat) näkyvät vain koodilohkon – tyypillisesti pääohjelman tai aliohjelman – määrittelevien aaltosulkujen sisällä. Esimerkiksi jokaisella C-ohjelmalla voi olla samanniminen muuttuja, esimerkiksi `luku1` tai `tulos`, ilman että ohjelman toiminta häiriintyy siitä millään tavalla. Tämä tietenkin tarkoittaa myös sitä, että nämä muuttujat eivät samasta nimestään huolimatta ole missään tekemisissä keskenään, eivätkä aliohjelmissa tehdyt muutokset pääsääntöisesti vaikuta pääohjelman samannimisiin muuttujiin. Otetaan tästä ensin esimerkin ja palataan asiaan tarkemmin sen jälkeen.

Esimerkki 4.5: Esimerkki tunnusten näkyvyydestä

Esimerkkikoodi

```
#include <stdio.h>

int testi(int syote) { /* Huomaa: muuttujaa syote ei käytetä! */
    int luku;
    luku = 100;
    printf("Aliohjelmanssa luku-muuttuja on %d.\n", luku);
    return luku;
}

int main(void) {
    int luku = 1;
    printf("Alussa muuttujan 'luku' arvo on %d.\n", luku);

    /* Kutsutaan aliohjelmaa ja annetaan luku parametrina */
    testi(luku); /* Huomaa: paluuarvoa ei käytetä eikä talleteta. */

    printf("Aliohjelma ei vaikuttanut pääohjelman muuttujaan.\n");
    printf("Pääohjelman muuttujan 'luku' arvo on edelleen %d.\n",
           luku);

    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./4-5
```

```
Alussa muuttujan ’luku’ arvo on 1.
Aliohjelman muuttujan luku-muuttuja on 100.
Aliohjelma ei vaikuttanut pääohjelman muuttujaan.
Pääohjelman muuttujan ’luku’ arvo on edelleen 1.
un@linux:~/un$
```

Kuinka koodi toimii

Tällä kertaa loimme niin pää- kuin aliohjelmaankin muuttujan nimeltä `luku`. Määrittelemme ensin pääohjelmassa muuttujalle arvon 1 ja tämän jälkeen annamme muuttujan arvon aliohjelmakutsussa parametrina.

Aliohjelmassa paikallisen `luku`-muuttujan arvoksi laitetaan 100, mikä todennetaan tulostamalla se ruudulle. Lisäksi tämän muuttujan arvo palautetaan pääohjelmaan, mutta siitäkin huolimatta pääohjelman `luku`-muuttujan arvo on edelleen sama 1 kuin ennen aliohjelmaa. Miksi näin käy?

Ensinnäkin, lähetämme aliohjelmaan ainoastaan muuttujan arvon, eli numeroarvon 1, emme muuttujaa `luku`, tai mitään muuta siihen liittyvää. Tämä arvo 1 on käytettävissä aliohjelmassa argumenttina olevan `syote`:en arvona. Aliohjelma luo omaan nimiavaruuteensa oman `luku`-muuttujan ja laittaa sen arvoksi 100 sekä palauttaa sen sisällön – numeroarvon 100 – takaisin pääohjelmaan. Pääohjelmassa meidän tulee seuraavaksi kiinnittää huomio aliohjelmakutsuun:

```
testi(luku);
```

Tämä kutsu ei ota paluuarvoa kiinni eli ei tallenna sitä minnekään, joten pääohjelman muuttujan `luku` arvo 1 ei muutu. Tämän vuoksi muuttujan arvo on edelleen sama ohjelman lopussa. Jos olisimme halunneet muuttaa pääohjelman `luku`-muuttujaa, olisi siihen meillä kolme vaihtoehtoa. Ensinnäkin, olisimme voineet ottaa aliohjelman paluuarvon talteen muuttamalla kutsun muotoon

```
luku = testi(luku);
```

tai käyttämällä globaaleja muuttujia (ks. lisätietoa (Alaoutinen 2005) tai (Kernighan ja Ritchie 1988)) tai käyttämällä osoittimia eli muuttujaparametreja, joista puhumme seuraavassa osiossa. Vain globaaleja muuttujia käyttämällä olisimme viitanneet koko ajan samaan muuttujaan, mutta koska globaalit muuttujat rikkovat ohjelmien perussääntöjä näkyvyyden osalta, ei niiden käyttö ole hyvää ohjelmointityyliä ja niitä ei kannata käyttää ollenkaan.

Globaalit muuttujat ovat huonoa ohjelmointityyliä, mutta vakiot ovat lähtökohtaisesti globaaleja. Koska vakioita ei voi muuttaa, ei niiden näkyvyyttä ole tarvetta rajoittaa. Muuttujien osalta tilanne on toinen, sillä väärin toimivan ohjelman syynä on usein jonkin (muuttujan) väärä arvo ja jos arvo voi muuttua missä kohdassa tahansa laajaa ohjelmaa, voi virheen löytäminen vaatia koko ohjelman tarkkaa läpikäyntiä. Tämän ongelman välttämiseksi muuttujat kannattaa pitää aina paikallisina eli halutun koodilohkon määrittelevien aaltosulkujen sisäpuolella; mikäli muuttuja määritellään koodilohkojen ulkopuolella eli samalla tasolla kuin pääohjelma, tulee muuttujista globaaleja.

Muuttujaparametrit eli osoittimet aliohjelmissa

Kävimme edellä läpi arvoparametrit, joiden arvoa ei voinut muuttaa aliohjelmassa ja lupasimme tällöin palata takaisin muuttamisen salliviin muuttujaparametreihin. C-kielessä parametrien muuttaminen on mahdollista käyttämällä osoittimia eli lähettämällä aliohjelmaan kiinnostavan tiedon sisältävän muistipaikan osoite itse arvon sijasta. Meillä oli puhetta muuttujien osoitteista eli osoittimista jo luvussa 2 `scanf`-funktion yhteydessä, jossa halusimme saada sijoitettua muuttujaan käyttäjän antaman arvon. Osoittimet ovat C-kielelle ominaisia ”muistiosoitinmuuttujia”. Tämä tarkoittaa sitä, että osoittimet ovat muuttujia, jotka sisältävät varsinaisen tiedon sijaan tiedon siitä,

mistä muistiosoitteesta tieto löytyy. Osoittimia voi ajatella samoin kuin internet-osoitteita: Normaali muuttuja voidaan ajatella itse verkkosivuna, joka sisältää kaiken sivuilla olevan tiedon. Osoittimet taas ovat kuin verkko-osoitteita (esimerkiksi *http://fi.wikipedia.org*), eli ne eivät sisällä sivun tietoja (tässä tapauksessa Suomen Wikipedian etusivun artikkeleita), vaan ainoastaan tiedon siitä, mistä nämä artikkelit löytyvät.

Osoittimien tärkein hyöty tulee niiden joustavuudesta tiedonkäsittelyn suhteen. Kuten myöhemmin tulemme näkemään, osoittimien avulla voidaan hallita monimutkaisia tietorakenteita sekä yhdistellä tietueita dynaamisiksi muistirakenteiksi. Tässä vaiheessa keskitymme osoittimien perusasioihin.

Ensinnäkin, osoittimen merkkinä käytetään *-merkkiä muuttujan nimen edessä. Tämä kertoo kääntäjälle, ettei kyseessä oli normaali muuttuja vaan osoitin muuttujaan. Yleisesti ottaen *-etumerkki tuleekin lukea ”arvo, joka sijaitsee tämän nimisessä osoitteessa”. Lisäksi osoittimien eli pointtereiden (eng. pointer) yhteydessä käytetään joskus muuttujan edessä &-merkkiä, joka tarkoittaa, että emme viittaa osoitettavan muistipaikan sisältöön, vaan muuttujan fyysiseen osoitteeseen keskusmuistissa, joka periaatteessa on suurehko numeroarvo: ”Osoite, jota tämänniminen muuttuja käyttää”. Yleensä &-merkkiä käytetään silloin kun asetamme osoittimen osoittamaan jonkin muuttujan sisältöön. Tämä kaikki voi ensi alkuun vaikuttaa hieman sekavalta, mutta seuraavassa esimerkissä kokeilemme osoittimen käyttöä käytännössä.

Esimerkki 4.6: Osoittimen määrittely ja käyttö

Esimerkkikoodi

```
#include <stdio.h>

void muuttaa(int *arvo) {
    *arvo = 1000;
}

void ei_muuta(int arvo) {
    arvo = 1000;
}

int main(void) {
    int luku = 1, luku_2 = 2;

    /*Luodaan kaksi kokonaislukuosoitinta eli osoitinta kokonaislukuun*/
    int *osoitin_1;
    int *osoitin_2;
    /* Asetetaan osoittimet muuttujiin luku ja luku_2 */
    osoitin_1 = &luku;
    osoitin_2 = &luku_2;

    printf("Osoitin 1 viittaa muistipaikkaan %p, jossa on arvo %d.\n",
            osoitin_1, *osoitin_1);
    printf("Muuttujan luku_2 arvo on alussa %d.\n", luku_2);

    /* Kutsutaan funktiota ei_muuta. */
    ei_muuta(luku_2);
    printf("Funktion ei_muuta jälkeen luku_2 arvo on %d.\n", luku_2);
```

```
    /* Kutsutaan funktiota muuttaa. */
    muuttaa(&luku_2);
    printf("Funktion muuttaa jälkeen luku_2 arvo %d.\n", *osoitin_2);

    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./4-6
Osoitin 1 viittaa muistipaikkaan 0x7fff47c2b098, jossa on arvo 1.
Muuttujan luku_2 arvo on alussa 2.
Funktion ei_muuta jälkeen luku_2 arvo on 2.
Funktion muuttaa jälkeen luku_2 arvo 1000.
un@linux:~/un$
```

Kuinka koodi toimii

Ohjelmassa on kaksi proseduuria eli aliohjelmaa, jotka eivät palauta arvoja. Huomaa, että tällä kertaa meidän ei tarvitse erikseen esitellä aliohjelmia, koska ne on määritelty ennen käyttöä. Ensimmäinen aliohjelma, `muuttaa`, ottaa vastaan osoitinmuuttujaan `arvo` kokonaisluvun osoitteen kun taas toinen aliohjelma, `ei_muuta`, ottaa vastaan kopion kokonaisluvusta.

Itse ohjelmakoodissa luomme kaksi osoitinta, `osoitin_1` ja `osoitin_2`, ja asetamme ne osoittamaan muuttujien `luku_1` ja `luku_2` käyttämiin muistiosoitteisiin, joissa on siis meitä kiinnostavat muuttujien arvot. Tämän jälkeen testaamme osoittimien toimintaa. Ensinnäkin tulostamme hieman tietoja osoittimesta: näemme, että `osoitin_1` osoittaa muistipaikkaan numero 0x7fff47c2b098, johon on tallennettuna arvo 1. Huomaa erityisesti, että osoitinta voidaan käyttää tiedon tulostamiseen normaalisti muuttujan tavoin. Lisäksi tulostamme muuttujan `luku_2` arvon alussa eli 2 näytölle.

Seuraavaksi kokeilemme oikeasti osoittimien eroja. Ensin kutsumme proseduuria `ei_muuta`, joka toimii kuten aiemmassa esimerkissä huomasimmekin eli parametrin arvo ei muuttunut. Kuinka tällainen ohjelma voidaan muuttaa siten, että aliohjelma saa siirrettyä muutetun arvon takaisin kutsuvaan ohjelmaan? Ensinnäkin, kuten varmaan huomaat, määritellään `muuttaa`-proseduurin parametrin tyypiksi osoitin. Tästä johtuen kutsuessamme proseduuria emme annakaan muuttujan `luku_2` arvoa kuten aiemmin, vaan lähetämme &-merkin avulla muuttujan muistipaikan osoitteen, eli käytännössä siis osoittimen `luku_2`-muuttujan käyttämään muistipaikkaan. Tämän seurauksena aliohjelma muuttaa sen muistipaikan arvoa, johon pääohjelman `luku_2` on tallennettu, ja tämän vuoksi muutos jää voimaan. Huomaa lisäksi, että osoittimen viittaamaa arvoa pystyttiin muuttamaan normaalilla sijoitusoperaatiolla.

Pääohjelmassa meillä on vielä viimeinen tulostuskäsky. Huomaa, että olemme jälleen käyttäneet tulostuskäskyssä muotoa `*osoitin_2`, joka tarkoittaa osoitteessa `osoitin_2` olevan muuttujan arvoa. Koska muutimme muistipaikassa ollutta arvoa edellisessä aliohjelmassa, löytyy osoittimen `osoitin_2` ilmoittamasta paikasta nyt arvo 1000.

Kirjastofunktiot

C-kielen ytimen toteutuksessa on pyritty vain pakolliset toiminnot sisältävän minimaalisen ytimen luomiseen. Tämä tarkoittaa sitä, että ohjelmat voidaan räätälöidä mahdollisimman tarkasti tehtävän ympäristön tarpeisiin eikä niissä ole mitään ylimääräistä. Esimerkiksi tehtäessä pieniä sulautettuja järjestelmiä, esim. kaukosäädin TV:lle tai CD-soittimelle, ei tarvita tiedon syöttö- ja tulostuskomentoja (esim. `printf` tai `scanf`) ja näitä ei siis tarvitse ottaa mukaan tällaisiin järjestelmiin. Näin ollen tiedon syöttö- ja tulostuskäskyt on sijoitettu tähän tarkoitukseen tehtyyn kirjastoon (standard input output –kirjastoon ja otsikkotiedostoon eli header'iin, `stdio.h`). Kun näitä toimintoja tarvitaan, otetaan tarvittava kirjasto käyttöön sisällyttämällä halutun kirjaston otsikkotiedosto lähdekooditiedostoon `include`-käskyllä, esim.:

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

Liitteessä 1 on lueteltu yleisimpien C-kielen kirjastojen otsikkotiedostot ja niiden sisältämät funktiot. Otsikkotiedostot sisältävät myös vakioiden määrittelyjä, joita tyypillisesti tarvitaan näiden funktioiden ja toimintojen yhteydessä.

Osaamistavoitteet

Tämä luvun jälkeen sinun tulee pystyä käsittelemään tekstitiedostoja sekä käyttämään aliohjelmia ohjelman rakenteen hallitsemiseen. Luvun esimerkit keskittyivät tekstitiedostojen lukemiseen ja kirjoittamiseen, mutta binaaritiedostojen käsittely poikkeaa näistä pääasiassa käytettävien funktioiden osalta. Aliohjelmat ovat C-kielessä yleensä funktioita, joilla on siis paluuarvo. Yleisesti ottaen aliohjelmiin välitetään tietoa parametreilla, joita on kahta tyyppiä – arvo- ja muuttujaparametreja. Arvoparametreja käytettäessä aliohjelmaan välittyy alkuperäisten parametrien kopiot, joten arvoparametrien arvojen muutokset eivät välity takaisin kutsuvaan ohjelmaan. Muuttujaparametrien kohdalla aliohjelmaan välitetään muistipaikan osoite, joka mahdollistaa muuttujan arvon muutosten näkymisen myös ko. aliohjelman ulkopuolella. Muuttujien näkyvyys C-kielessä perustuu koodilohkoon eli aaltosulkujen sisällä määritellyt muuttujat näkyvät ja ovat voimassa vain ko. koodilohkossa. Näin ollen yhdessä aliohjelmassa määritelty muuttuja ei näy toiseen aliohjelmaan. C-kielessä on olemassa monia kirjastoja, joista löytyy paljon valmiita aliohjelmia eri käyttötarkoituksiin. Itse C-kieli on pidetty pienenä, jotta käännettyjen ohjelmien koko saadaan pidettyä mahdollisimman pienenä.

Luku 5: Aliohjelmien täydennyksiä ja tietueet

Edellinen luku käsitteli aliohjelmien käyttöä ja tässä luvussa katsotaan muutamia aliohjelmiin liittyviä täydennyksiä sekä tutustutaan tietueisiin. Aliohjelmien opiskelua jatketaan tutustumalla makroihin, rekursioon ja komentoriviparametreihin. Tietueet liittyvät tietorakenteisiin ja tekevät mahdolliseksi omien rakenteisten tietotyyppien määrittelyn, jotka kokoavat useita perustietotyyppejä yhteen uuteen tietotyyppiin.

Merkkijonoliteraalista makroon

Esittelimme merkkijonoliteraalit luvussa 3 esikäsittelijän `define`-käskyn kanssa ja käytimme niitä vakioina. Esimerkiksi taulukon määrittely tehtiin `#define MAX 10` –käskyllä, jolloin kaikissa taulukon käsittelyyn käytetyissä silmukoissa on varmasti sama raja-arvo.

Merkkijonoliteraalien avulla voidaan määritellä myös makroja, jotka ovat funktioiden tyylisiä uudelleen käytettäviä koodilohkoja. Mutta kuten merkkijonoliteraali termi antaa ymmärtää, korvataan makron määritelmä suoraan annetulla koodilla ja tästä seuraa merkittäviä eroja makron ja funktion toimintaan käytännössä. Makrojen keskeinen etu on se, että ne näyttävät funktioilta eli yksinkertaistavat koodia, mutta välttävät funktiokutsun aiheuttaman aliohjelmaan liittyvän tietojen siirtämisen pinoon ja ohjelman suorituksen hidastumisen. Kääntöpuolena on se, että kun esikääntäjä korvaa makron koodilla, ei ”parametrien” arvoja lasketa kuten funktiokutsuissa, joten mahdolliset operaattorit parametreissa noudattavat niiden normaaleja prioriteetteja ja voivat tuottaa yllättäviä tuloksia huolimattomassa käytössä.

Esimerkki 5.1: Makro

Esimerkkikoodi

```
#include <stdio.h>
#define square(x) x*x

int main(void) {
    int luku = 3;
    printf("Makro square() parametrilla %d tuottaa %d.\n", luku,
            square(luku));
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./5-1
Makro square() parametrilla 3 tuottaa 9.
un@linux:~/un$
```

Kuinka koodi toimii

Ohjelman käännöksen ensimmäisessä vaiheessa esikäsittelijä vaihtaa `square(luku)` merkkien tilalle merkkijonon `luku*luku` ja normaalin käännöksen jälkeen ohjelma toimii esimerkkiajon mukaisesti niin kuin pitääkin. Makron käyttö tekee koodia lukevalle henkilölle selväksi, että nyt halutaan tulostaa luvun neliö ja tämä tekee koodista yleisesti ottaen helpomman ylläpitää. Kääntöpuolena on riski, että ohjelmoija kirjoittaa sulkuihin laskutoimituksen kuten `square(3+1)`, joka johtaakin useimmille odottamattomaan tulokseen 7.

Toistorakenteesta rekursioon

Luvussa 3 käytiin läpi toistorakenteet `for`, `while` ja `do-while`, jotka ovat normaali tapa toistaa sama koodilohko useita kertoja. Toistettava koodilohko voidaan toteuttaa myös aliohjelmassa ja laittaa aliohjelma kutsumaan itseään eli tehdä aliohjelmasta rekursiivinen. Rekursion hyviä puolia ovat sen helppo toteutus sekä hyvä sopivuus tiettyihin tehtäviin ja tietorakenteisiin kun taas kääntöpuolena on aliohjelmakutsuihin perustuvaan toteutukseen liittyvä hitaus ja muistin tarve.

Rekursiivisen aliohjelman idea on yksinkertainen – aliohjelma jakautuu kahteen haaraan, joista toinen kutsuu samaa aliohjelmaa uudella parametrin arvolla ja toinen haara palauttaa lopetusehdon täyttyessä lopetusarvon.

```
int aliohjelma(parametri) {

    Jos parametri == lopetusehto, esim. parametri on 1 tai 0.

        Lopeta, palauta sopiva arvo, esim. 1

    Muutoin

        Kutsu itseä uudella parametrin arvolla, esim. (parametri –
1)

}
```

Rekursio sopii hyvin esim. kertoman laskuun, jossa luku kerrotaan kaikilla itseään pienemmillä positiivisilla kokonaisluvuilla. Alla olevasta esimerkistä käy ilmi, miten kertoma voidaan laskea helposti rekursiivisella aliohjelmalla.

Esimerkki 5.2: Rekursio

Esimerkkikoodi

```
#include <stdio.h>

int kertoma(int x) {
    if (x == 0) /* Lopetusehto */
        return 1;
    else /* Kutsuu itseään, eri arvolla !! */
        return (x * kertoma(x - 1));
}

int main(void) {
    int luku = 6;
```

```
    printf("Luvun %d kertoma on %d.\n", luku, kertoma(luku));
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./5-2
Luvun 6 kertoma on 720.
un@linux:~/un$
```

Kuinka koodi toimii

Esimerkkiohjelma kutsuu `kertoma`-aliohjelmaa parametrina luku 6. Koska 6 ei ole 0, siirtyy suoritus valintarakenteen `else`-haaraan, jossa on uusi kutsu `kertoma`-aliohjelmaan arvolla 6-1 eli 5. Tämä sama toistuu kunnes `kertoma`-aliohjelmaan tullaan parametrin arvolla 0, joka johtaa `if`-haaran suorittamiseen ja aliohjelma palauttaa arvon 1 kutsuvalle ohjelmalle. Arvon palauttaminen kutsuvalle ohjelmalle johtaa pinossa olevan edellisen `kertoma`-kutsun suorituksen jatkumiseen ja `kertoma(1-1)`:n tilalle tulee paluuarvo 1, joka kerrotaan vielä `x`:n arvolla 1 ja funktio palauttaa kutsuvaan ohjelmaan 1:n. Tämä johtaa edellisen aliohjelmakutsun palauttamiseen pinosta ja `kertoma(2-1)`:n tilalle tulee nyt 1 ja ohjelma palauttaa kutsuneelle ohjelmalle arvon 2*1 eli 2. Koska ensimmäinen kutsu tehtiin arvolla 6, suorittaa `kertoma`-aliohjelman `else`-haara käytännössä laskun 6*5*4*3*2*1*1 eli siis 6! kuten alkuperäinen tehtävä oli.

Parametreista komentoriviparametreihin

Komentoriviparametrit ovat C-ohjelman pääohjelman, `main`-funktion, argumentteja. Ne toimivat vastaavalla tavalla kuin minkä tahansa aliohjelman parametrit, mutta koska komentoriviparametrit saavat arvokseen käyttäjän komentorivillä antamat tiedot, on niiden toiminnassa muutama huomioitava asia.

C-kielessä komentoriviparametrien käyttöönotto on yksinkertaista. Olemme aiemmin kirjoittaneet `main`-funktion muotoon

```
int main(void)
```

ja jatkaneet työskentelyä sen suuremmin itse koodia ajattelematta. Mikäli haluamme, että käyttäjä voi syöttää jo ohjelman käynnistyksen yhteydessä jotain tietoa, kuten kohdetiedoston nimen tai muuta vastaavaa, tarvitsemme keinon siirtää käyttäjän antamat syötteet `main`-funktion parametreiksi. Tämä on itse asiassa helppoa, sillä meidän tarvitsee vain muuttaa `main`-funktion parametrimäärittely seuraavaan muotoon:

```
int main(int <lukumäärämuuttujan nimi>, char *<parametrilistan nimi>[])
```

eli vaikkapa

```
int main(int args, char *argv[]) tai
int main(int maara, char *argumenttilista[])
```

Tämän jälkeen kääntäjä hoitaa loput. Näillä muutoksilla näemme annettujen parametrien määrän suoraan ensimmäisestä muuttujasta ja itse parametrit toisesta muuttujasta. Yksittäisiin parametreihin voimme viitata notaatiolla `argv[i]` (tai `argumenttilista[i]`), jossa i on

parametrin järjestysnumero. Huomaa, että parametrit välitetään aina merkkijonoina ja niiden käyttämistä varten saatetaan joutua tekemään tyyppimuunnoksia (kts. liite 1, `ctype`, `stdlib`). Lisäksi parametri 0 on ajettavan tiedoston nimi ja täten ensimmäinen varsinainen parametri on paikalla 1. Lisäksi tämä tarkoittaa sitä, että parametreja annetaan aina vähintään yksi. Katsotaan vielä esimerkki komentoriviparametrien toiminnasta:

Esimerkki 5.3: Komentoriviparametrit

Esimerkkikoodi

```
#include <stdio.h>

/* Lisäämme main-funktioon tuen komentoriviparametreille:
 * Ensin integer-muuttuja argc, johon tulee tieto parametrien määrästä
 * ja tämän jälkeen merkkitaulu argv (argument vector), johon tulee
 * itse parametrit. */

int main(int argc, char *argv[]) {
    int i;

    printf("Annoit %d komentoriviparametria, jotka olivat:\n", argc);
    for (i = 0; I i < argc; i++) {
        printf("%d. parametri oli %s\n", i, argv[i]);
    }
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./5-3 yksi kaksi 313
Annoit 4 komentoriviparametria, jotka olivat:
0. parametri oli 5-3
1. parametri oli yksi
2. parametri oli kaksi
3. parametri oli 313
un@linux:~/un$
```

Kuinka koodi toimii

Koodi ei näytä ohjelmoijalle kovinkaan yllättävältä. Otimme käyttöön komentoriviparametrit lisäämällä `main`-funktion määrittelyyn parametrit `int argc` ja `char *argv[],` jotta voimme vastaanottaa parametreja. Tämän jälkeen tulostamme saadut parametrit `argv`-taulukosta `for`-lauseen avulla. Huomaa edelleen, että ”0. parametri” oli suoritettavan tiedoston nimi (nyt siis `5-3`) ja että parametrien yhteismäärä `argc`-muuttujassa oli 4, joka on siis käyttöjärjestelmän tälle ohjelmalle välittämien parametrien lukumäärä.

Viimeiseen parametriin liittyy vielä yksi keskeinen asia eli tiedon käyttäminen numeroarvona. Koska parametrit on nyt tallennettu merkkeinä, tarkoittaa tämä sitä, että emme voi käyttää arvoa 313 suoraan numeroarvona tyypittämättä sitä ensin kokonaisluvuksi. Mikäli joskus törmäät

tällaiseen tilanteeseen, löydät lisäohjeita liitteestä 1 ja sieltä löytyvistä tyyppimuunnosfunktioiden käyttöohjeista (esim. `atoi(argv[3])`).

Yksinkertaisista tietotyypeistä tietueisiin

Monet sovellukset käsittelevät henkilötietoja ja niitä voi olla useita kuten esimerkiksi syntymäpäivä, etunimi, sukunimi, kotiosoite jne. Koska tällaisia useista tiedoista koostuvia tietokokonaisuuksia tarvitaan yhtään isommissa ohjelmissa paljon, tarjoavat useimmat ohjelmointikielet keinoja yhdistää useita muuttujia yhteen rakenteisiksi tietueiksi. C-kielessä tietue voidaan luoda `struct`-avainsanalla. Sen määrittely ja käyttäminen vastaavat pitkälti Pythonin luokkarakennetta ilman jäsenfunktioita ja määrittely tehdään seuraavalla syntaksilla:

```
struct tietueen_nimi{
     tietueen jäsenet, eli tyyppi ja nimi;
};
```

Luotuamme tietueen voimme tehdä tietueessa määriteltyä muotoa sisältäviä muuttujia käskyllä ’`struct tietueen_nimi muuttuja;`’. Seuraavaksi tarkastelemme tietueiden käyttöä esimerkin avulla.

Esimerkki 5.4: Tietueen määrittely ja käyttö

Esimerkkikoodi

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

struct ihminen {
    char nimi[30];
    int ika;
    char osoite[30];
};

int main(void) {
    struct ihminen henkilo;

    henkilo.ika = 17;
    strcpy(henkilo.osoite, "sihijuomakatu");
    /* strcpy eli string copy - kopioi merkkijono */

    printf("Anna henkilön nimi:\n");
    scanf("%s", &henkilo.nimi);

    if (henkilo.ika < 18) {
        printf("%s on alaikäinen.", henkilo.nimi);
    }
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./5-4
```

```
Anna henkilön nimi:
Hjördis
Hjördis on alaikäinen.
un@linux:~/un$
```

Kuinka koodi toimii

Jotta tietueita voisi käyttää, pitää lähdekoodi aloittaa kertomalla kääntäjälle, millaisia tietueita lähdekoodissa käytetään. Tämä tapahtuu kirjoittamalla tietueiden esittelyt lähdekoodiin ennen niiden käyttöä eli ennen funktioesittelyitä ja toiminnallista koodia. Seuraava koodi luo tietueen `ihminen`:

```
struct ihminen {
    char nimi[30];
    int ika;
    char osoite[30];
};
```

Tietueen jäsenmuuttujina ovat merkkijonotaulukko nimi, kokonaislukumuuttuja ika sekä merkkijonotaulukko osoite. Kun olemme luoneet tietueen, voimme käyttää sitä itse koodissa. Tässä vaiheessa olemme kertoneen kääntäjälle, että lähdekoodissa voi esiintyä tällainen tietorakenne. Tämän vuoksi main-funktion sisällä on rivi

```
struct ihminen henkilo;
```

jolla luomme tietueen tyyppisen muuttujan. Rivi siis kertoo, että luomme tietuetyypin ihminen mukaisen (struct ihminen) muuttujan henkilo. Tästä eteenpäin voimme käyttää henkilo:n jäsenmuuttujia kuten normaaleja muuttujia. Esimerkin muilla riveillä on esitelty mm. merkkijonojen kopiointia jäsenmuuttujaan, jäsenmuuttujan käyttöä if-rakenteen ehtona sekä jäsenmuuttujien tulostamista. Huomaa, että tietueen jäsenmuuttujiin viitataan pistenotaatiolla Pythonin tapaan eli ensin on muuttujan nimi, sitten piste ja lopuksi käsiteltävän tietorakenteen sisäisen muuttujan eli jäsenmuuttujan nimi, esimerkiksi henkilo.osoite.

Osaamistavoitteet

Tämä luvun jälkeen sinun tulee pystyä luomaan ja käyttämään C-kielen rakenteisia tietorakenteita eli `struct`:eja sekä komentoriviparametreja. Sinun tulee myös pystyä luomaan yksinkertaisia makroja ja rekursiivisia aliohjelmia luvussa käsiteltyjen esimerkkien pohjalta.

Luku 6: Muistinhallinta, tietotyypit ja versionhallinta

Tässä luvussa tutustumme dynaamiseen muistinhallintaan ja täydennämme aiempaa käsittelyä omiin tietotyyppeihin liittyen. Lisäksi tutustumme versionhallintaan SVN-järjestelmän avulla.

Dynaaminen muistinvaraus

Aiemmin puhuimme muistinvarauksesta merkkijonojen käsittelyn yhteydessä. Silloin totesimme, että on helpointa käyttää staattisia taulukoita vaikka niiden käyttöön liittyi kaksi ongelmaa: ensinnäkin jos teimme suuren merkkitaulukon johon tallensimme ainoastaan vähän tietoa, tuhlasimme tilaa ja jos taas teimme liian pienen taulukon, ohjelma ei toiminut oikein. Mainitsimmekin jo silloin, että toinen vaihtoehto on tallentaa tieto varaamalla muistia käsin, eli toiminto, johon tutustumme nyt seuraavaksi.

C-kielessä muistinvaraus toteutetaan `stdlib`-kirjastossa olevalla `malloc`-funktiolla. `malloc`-funktio saa syötteenä itselleen tietotyypin, jota haluamme käyttää tallentamiseen sekä määrän, jonka haluamme varata. Tässä on kuitenkin ongelma – kuinka voimme varata muistia merkkijonoa varten, jos meidän ensin tulee laskea sen pituus ja vasta tämän jälkeen tiedämme, minkä verran muistia tarvitaan.

Tähän ongelmaan käytämme ratkaisuna lukupuskuria, joka käytännössä on yksi staattinen merkkitaulu standardisyötevirran lukuun. Lukupuskurin on hyvä olla riittävän suuri, jotta siitä on mitään hyötyä, mutta kohtuullisen kokoinen, että sen käyttö olisi mielekästä. Esimerkiksi 255 tavua on useimmiten riittävä puskuri syötevirran lukemiseen. Lisäksi tiedostojen yhteydessä voimme ensin laskea montako merkkiä esimerkiksi rivillä on ja vasta tämän jälkeen lukea, jolloin tarve puskurille periaatteessa poistuu.

`malloc`-funktio palauttaa osoittimen varaamaansa muistialueeseen. Luvussa 2 käsiteltiin muuttujan ja osoittimen eroa tarkemmin, mutta idea on se, että tietoa säilytetään muistissa ja varatulla muistialueella on osoite, josta se löytyy muistiavaruudessa. Kun muistia varataan `malloc`-funktiolla, annetaan funktiolle parametrina tarvittava muistin määrä ja `malloc` palauttaa sopivaa tyyppiä olevan osoittimen arvon, joka on otettava talteen osoitinmuuttujaan. Koska halutun kokoisen muistialueen vapaana ole ei ole itsestään selvää, on hyvän ohjelmointitavan mukaista tarkistaa aina muistinvarauksen onnistuminen. Seuraavassa esimerkissä kokeilemme merkkijonon muistinvarausta sekä lukupuskurin käyttämistä.

Esimerkki 6.1: Dynaaminen muistinvaraus

Esimerkkikoodi

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

int main(void) {
    char lukumuisti[255];
    char *muistissa;
    int pituus;

    printf("Kirjoita jotain (max.255 merkkiä):\n");
    fgets(lukumuisti, 255, stdin);

    /* Lasketaan luetun merkkijonon pituus. */
    /* Otetaan pituudesta pois rivinvaihtomerkki, sillä se toimii
     * loppumerkkinä eikä siten ole osa käyttäjän antamaa
     * syötettä. */
    pituus = strlen(lukumuisti) - 1;

    /* Varataan muistia luetun sanan pituuden verran ottaen
     * huomioon myös loppumerkin tarvitsema tila:
     * ((pituus+1)* 1 merkin koko). */
    muistissa = (char *)malloc((pituus + 1) * sizeof(char));
    /* HUOM: virheentarkistus on seuraavassa esimerkissä. */

    /* Kopioidaan merkkijono ilman rivinvaihtomerkkiä strncpy:llä. */
    strncpy(muistissa, lukumuisti, pituus);
    printf("Kirjoitit merkkijonon '%s', joka oli %d merkkiä
pitkä.\n", muistissa, pituus);
    printf("Staattisessa muistinhallinnassa merkkijono vie tilaa
%ld tavua.\n", sizeof (lukumuisti));
    printf("Dynaamisessa muistinhallinnassa merkkijonon pointteri
osoitin vie tilaa %ld tavua.\n", sizeof (muistissa));

    /* Lopuksi vapautetaan varattu muisti. */
    free(muistissa);

    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./6-1
Kirjoita jotain (max.255 merkkiä):
Apumiehensijaisenvaramiespalveluvastaava
```

```
Kirjoitit merkkijonon ’Apumiehensijaisenvaramiespalveluvastaava’,
joka oli 40 merkkiä pitkä.
Staattisessa muistinhallinnassa merkkijono vie tilaa 255 tavua.
Dynaamisessa muistinhallinnassa merkkijonon pointteri vie tilaa 8
tavua.
un@linux:~/un$
```

Kuinka koodi toimii

Ohjelman alussa luomme kolme muuttujaa, staattisen merkkijonotaulukon `lukumuisti` ja merkki-osoittimen `muistissa` sekä kokonaislukumuuttujan `pituus`, joiden avulla tallennamme käyttäjän syöttämän merkkijonon dynaamisesti varattuun muistiin. Idea on siis käyttää yhtä staattisesti varattua merkkijonotaulukkoa lukemaan useita käyttäjän antamia tietoja ja tehdään tästä taulukosta niin suuri, että siihen mahtuu suurin käyttäjän kerralla antama tietomäärä. Kun käyttäjä on antanut yhden tiedon, voidaan sen tarvitsema muistin määrä laskea ja varata täsmälleen oikea määrä muistia ko. tietomäärälle. Näin voimme ottaa vastaan käyttäjältä paljon erilaista tietoa ja minimoida tarvittavan muistin määrä käyttäen dynaamista muistivarausta.

Ensimmäisenä tietenkin otamme käyttäjältä merkkijonon ja tallennamme sen muuttujaan `lukumuisti`. Tämän jälkeen laskemme merkkijonon pituuden funktiolla `strlen`, joka laskee annetusta merkkitaulukosta ei-tyhjien merkkien muodostaman merkkijonon pituuden. Kun nyt olemme ottaneet merkkijonon talteen ja laskeneet sen pituuden, voimme käskyllä

```
muistissa = (char *)malloc((pituus+1)*sizeof(char));
```

varata muistin. Vaikka kyseinen käsky näyttää hieman sekavalta, kannattaa sitä ennemmin ajatella näin:

```
tallennuspaikka = (tallennettava tyyppi)malloc(varattava tila)
```

eli periaatteessa ilmoitamme muistinvarausfunktiolle, että tallennamme `(pituus+1)` kappaletta merkin kokoista tilaa `sizeof(char)`, haluamme tämän varatun tilan merkkitaulukko-osoittimena `(char *)` ja se tallennetaan paikkaan `muistissa`. Tällä käskyllä saamme siis osoittimen `muistissa` osoittamaan paikkaan vapaata tilaa `(pituus+1)` merkkiä.

Nyt kun olemme varanneet muistin, voimme kopioida `lukumuisti`-muuttujan ei-tyhjät merkit dynaamisesti varattuun muistiin `strncpy`-funktiolla. Kuten vielä seuraavista `printf`-käskyjen esimerkeistä näemme, voimme nyt käyttää `muistissa`-muuttujan sisältämää tietoa normaalin muuttujan tavoin, ja voisimme vapauttaa `lukumuisti`-merkkitaulukon muuhun käyttöön. Aivan viimeisenä asiana vielä vapautamme varaamamme muistin, jotta käyttöjärjestelmä tietää jatkossa, että tätä tilaa saa taas käyttää.

Tässä esimerkissä merkkijonon pituus vaati erityistä tarkkuutta. Tämä johtuu siitä, että `fgets`-funktio otti käyttäjän antaman rivinvaihtomerkin (’`\n`’) mukaan merkkijonoon, vaikka se tässä yhteydessä oli syötteen lopetusmerkki. Näin ollen tämä merkki otettiin pois rivin pituudesta ja sen kopioituminen uuteen merkkijonoon estettiin `strncpy`-funktion käytöllä. Toisaalta `strlen`-funktio laskee merkkijonon pituuden loppumerkkiin asti (’`\0`’) sitä huomioimatta. Koska loppumerkin on kuitenkin oltava mukana merkkijonossa, varattiin sitä varten ylimääräinen merkki `malloc`:n yhteydessä `(pituus+1)`:llä.

Huomioita muistinvarauksesta

Tämä esimerkki osoitti lyhyesti sen, kuinka muistia yleisesti ottaen käytetään, mutta muistinvaraukseen liittyy vielä muutama huomio:

- Vapauta aina käyttämäsi muisti `free(varattu alue)`-komennolla. Ohjelma, joka ei vapauta käyttämäänsä muistia, toimii virheellisesti eli ns. vuotaa muistia, joka taas voi haitata muiden ohjelmien toimintaa.
- Jo varatun alueen kokoa voidaan muuttaa `realloc`-funktiolla, mikäli joskus tulee tarve vaihtaa muistialueelle tallennettua tietoa.
- `malloc` (ja `realloc`) palauttavat NULL-osoittimen mikäli eivät onnistu varaamaan halutunkokoista muistialuetta. C-ohjelmoinnin hyvien tapojen mukaista onkin aina muistinvarauksen yhteydessä testata, ettei näin pääse käymään. Helpointa tämä on sitomalla muistinvarauskäsky `if`-rakenteen ehdoksi:

```
if (muistissa = (char *)malloc((pituus+1)*sizeof(char)) == NULL){
    printf("Muistinvaraus epäonnistui!\n");
    exit(0);
}
```

`exit()` on `stdlib`-kirjastossa määritelty funktio, joka lopettaa ohjelman suorituksen.

Laajennoksia omiin tietotyyppeihin

Tähän asti olemme käyttäneet pääasiassa C-kielen perustietotyyppejä (esim. `int` ja `float`), mutta luvussa 5 tutustuimme myös rakenteiseen tietorakenteeseen eli tietueeseen, joka määritellään C-kielessä avainsanalla `struct`. Tietue on keskeinen rakenteinen tietorakenne C-kielessä, muttei kuitenkaan ainoa. `union` –avainsanalla voidaan määritellä **yhdiste**-tyyppisiä rakenteisia tietorakenteita, jolle varataan määreeseen kuuluvista tietotyypeistä enemmän tilaa vaativan muuttujan tarvitsema tila, mutta käyttäjä voi käyttää samaa muistialuetta myös vaihtoehtoisella tietotyypille. Tämä mahdollisuus ei ole nykyaikana kriittinen ominaisuus muistin määrän jatkuvasti kasvaessa, mutta tarjoaa kuitenkin ohjelmoijalle joustavuutta tarkasti määriteltyihin tietotyyppeihin tarpeen tullen. Alla on lyhyt esimerkki yhdisteestä ja asiasta kiinnostuneet voivat paneutua siihen tarkemmin omatoimisesti.

```
union luku_t {
    int kokonaisluku;
    float liukuluku;
} yhdiste;

int iLuku;
yhdiste.kokonaisluku = 4;
iLuku = yhdiste.kokonaisluku;
```

Luvussa 3 ja 5 oli puhetta merkkijonovakioista, joita määritellään esikäsittelijän `define`-käskyllä. Tällä käskyllä voidaan määritellä esim. vakioita aina yksi vakio käskyä ja riviä kohden. Mikäli ohjelmassa tarvitaan useita vakioita, esim. kuukausien tai viikonpäivien nimet, on määrittely

kuitenkin työlästä. Tilannetta helpottaa **lueteltu tyyppi** eli `enum` –käsky, joka antaa luetelluille nimille arvoksi kokonaislukuja kasvattaen arvoa aina yhdellä ellei käyttäjä aseta vakioille tiettyjä arvoja sijoituslausekkeella. Ensimmäisen tunnuksen oletusarvo on 0.

```
enum tosi { EI, KYLLA };
enum pakan_maat { RISTI = 4, RUUTU, HERTTA, PATA };
struct kortti {
    int numero;
    enum pakan_maat maa;
};
```

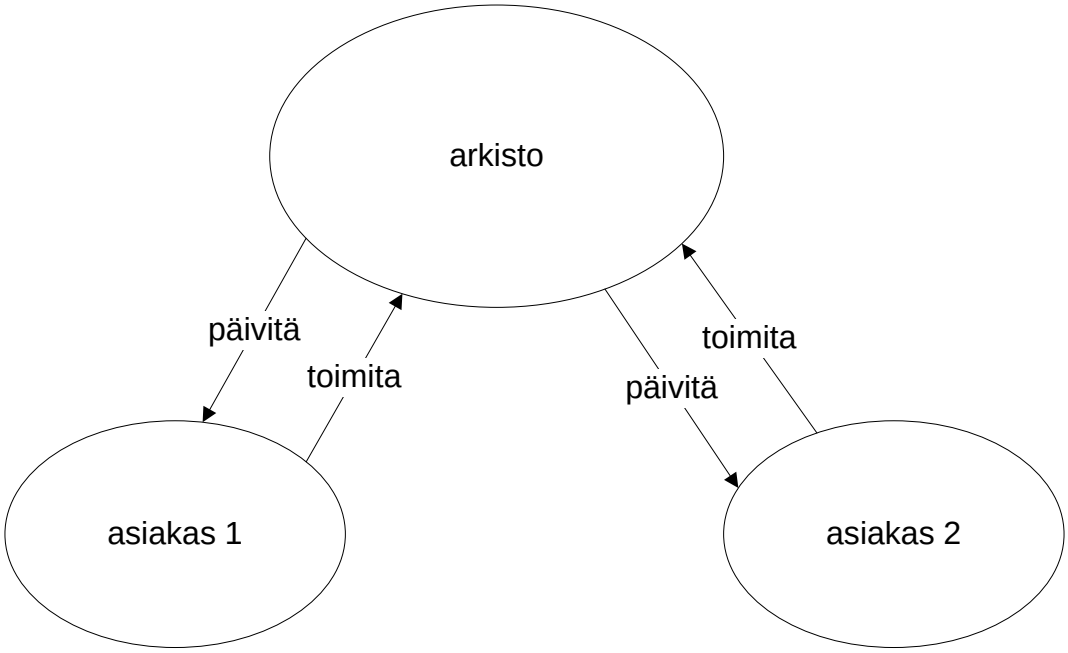
Rakenteisten tietotyyppien määrittely yhteydessä on siis käytettävä varattuja sanoja (`struct`, `union`, `enum`) sekä halutun tietotyypin yksilöivää tunnistetta. Jottei näitä varattuja sanoja tarvitse toistaa jatkuvasti, voidaan määritellä **oma tietotyyppi** `typedef`-käskyllä alla olevan esimerkin mukaisesti.

```
struct kortti {
    int numero;
    char maa[7];
};

typedef struct kortti Kortti;
Kortti pakka[52];

enum pakan_maat { RISTI = 4, RUUTU, HERTTA, PATA };
typedef enum pakan_maat laji;

struct kortti {
    int numero;
    laji maa;
};
```

Versionhallinta

Ohjelmistoprojektien keskeisimpiä tehtäviä on tuottaa ohjelmakoodia. Koska koodi usein kehittyy ja muuttuu ajan myötä, nousee toiseksi keskeiseksi tehtäväksi tämä koodin hallinta, jota yleisesti ottaen kutsutaan versionhallinnaksi. Versionhallintajärjestelmät perustuvat asiakas-palvelin periaatteen (Kuva 2). Tällä rakenteella varmistetaan se, että ohjelmamoduulista säilyy aina alkuperäinen kappale arkistossa (repository) ja käyttäjät muokkaavat eli editoivat vain kopioita alkuperäisestä koodista.

Versionhallintajärjestelmällä on useita tehtäviä. Ensinnäkin versionhallinta takaa stabiilin työympäristön varmistamalla alkuperäisten moduulien säilymisen, vaikka useampi kehittäjä tekisi muutoksia samaan moduuliin samaan aikaan. Toiseksi useimmat versionhallintajärjestelmät tarjoavat myös tukea tällaisten muutosten tunnistamiseen ja moduulien yhdistämiseen, jotta kaikki muutokset saadaan hallitusti mukaan seuraavaan versioon. Kolmanneksi muutettujen moduulien toimittaminen arkistoon tarjoaa luonnollisen kohdan kommentoida tehtyjä muutoksia, jotta myöhemmin tehtyjen muutosten paikallistaminen olisi mahdollisimman helppoa (esim. ”Lisätty järjestä ja tulosta –toiminnot.”). Neljänneksi versionhallintajärjestelmä pitää kirjaa jokaisen moduulin jokaisesta versiosta, mikä mahdollistaa minkä tahansa aiemman version palauttamisen tarpeen tullen. Viidenneksi järjestelmä kirjaa kaikista muutoksista normaalit hallinnolliset tiedot, jotka sisältävät tyypillisesti muutosajankohdan sekä tekijän käyttäjätunnuksen.



Kuva 2. SVN-versionhallintajärjestelmän yksinkertaistettu asiakas-palvelin periaate.

SVN (SubVersioN)

Tigris SVN (http://subversion.tigris.org/) on avoimen lähdekoodin versionhallintaohjelmisto GNU/Linux- ympäristöön. Versionhallinnan arkistoon voidaan tallettaa kaikkien ohjelmamoduulien kaikki versiot, joista muodostuu ohjelman versiohistoria. Päivitä-komennolla haetaan työversio (working copy) arkistosta. Jos työversio olisi editoitavana aina vain yhdellä henkilöllä kerrallaan, ei versionhallintaa tarvittaisi ollenkaan. Näin ollen versionhallinnan idea on tarjota toimintoja, joiden avulla usean kehittäjän työversioon tekemiä muutoksia voidaan hallita, esim. SVN:ssä kopioi/muuta/yhdistä -malli.

Toimita-komennolla tallennetaan editoitu tiedosto ja siihen liittyvät kommentit arkistoon. Versionhallintatyökalun avulla projektissa useamman henkilön on mahdollista tehdä hallitusti muutoksia ohjelmistoon.

SVN:n tärkeimmät komennot

svn checkout [polku] [svn_hakemisto] : noutaa työkopion arkistosta työpisteelle

[polku] esim. `https://www2.it.lut.fi/svn/courses/CT60A0210`

svn update *:* päivittää työkopion eli hakee arkistosta muuttuneet tiedostot työpisteelle

svn ci -m *:* tallentaa työpisteellä olevan työkopion arkistoon kommentoituna

esim. ***svn ci –m*** ”Tuotu tiedosto nimi foo.c”

svn add foo.c *:* lisää työhakemistosta projektiin tiedoston/tiedostoja

svn status *:* tarkistaa tiedostojen tilanteen edelliseen työkopion päivitykseen (update)

tai

svn diff **:** tarkistaa tiedostojen tilanteen edelliseen työkopion päivitykseen

svn revert *:* peruuttaa tehdyt muutokset

svn help *:* näyttää komentolistauksen

SVN-arkistoon lähetettävissä tiedostonimissä ei tule käyttää skandinaavisia merkkejä (ä, ö) tai välilyöntejä, sillä eri käyttöjärjestelmien välillä operoitaessa nämä aiheuttavat usein ongelmia. Samoin isot ja pienet kirjaimet erotellaan toisistaan unixin tapaan. Ongelmien välttämiseksi eri käyttöjärjestelmissä ei siis kannata erotella tiedostojen nimiä vain kirjaimien koolla vaan tiedostoille kannattaa antaa oikeasti erilaiset nimet.

Lyhyt SVN ohje

Seuraavassa on lyhyt esimerkki SVN-versionhallinnan käyttöönotosta Linux-työasemalla. Huomaa, että tässä esimerkissä `www2.it.lut.fi` -palvelimella on olemassa projekti, josta otetaan paikallinen kopio omalle työasemalle ja jatketaan projektin työstämistä siitä. `www2.it.lut.fi` – palvelin on laitoksen laboratoriomestarin hallinnoima palvelin ja hän perustaa projekteja sinne

pyyntöjen perusteella. Omalle palvelimelle projektien perustamiseen on useita vaihtoehtoja, joihin ohjeita löytyy laajemmista SVN-oppaista.

Aluksi siirrytään kotihakemistoon, luodaan projektihakemisto ja siirrytään sinne:

```
cd ~
mkdir CT60A0210
cd CT60A0210
```

Suoritetaan SVN `checkout`, joka luo paikallisen version SVN-palvelimen tämän hetkisestä sisällöstä:

```
:~/CT60A0210$ svn checkout
https://www2.it.lut.fi/svn/courses/CT60A0210
```

(Syötetään tarvittaessa käyttäjätunnus ja salasana.)

Lisätään hakemistossa oleva `testi.c` –tiedosto versionhallinnan piiriin:

```
:~/CT60A0210$ svn add testi.c
```

Päivitetään tehty muutos palvelimelle `commit`-komennolla:

```
:~/CT60A0210$ svn ci -m "Lisättiin uusi tiedosto testi.c."
```

Tarkistetaan, onko palvelimelle tullut muita muutoksia sitten `checkout`:in.

```
:~/CT60A0210$ svn update
```

(Paikallinen sisältö päivittyy tarvittaessa. Käytetään `svn update`:a aina tästä eteenpäin tuomaan päivitetty tilanne palvelimelta, `checkout`:ia ei enää tarvitse tehdä.)

Tehdään muutoksia jo versionhallinnan piirissä oleviin tiedostoihin:

```
:~/CT60A0210$ pico tiedosto.c
```

Voidaan halutessa tarkistaa tiedostojen tilanne suhteessa edelliseen päivitykseen (`update`).

```
:~/CT60A0210$ svn status
```

tai

```
:~/CT60A0210$ svn diff
```

Kun tiedostoa on editoitu, voidaan tehdä taas uusi `commit`. SVN `add`-komentoa ei enää tarvita uudestaan tälle tiedostolle, sillä SVN pitää jo kirjaa sen muutoksista.

Tortoise SVN

Tortoise SVN on avoimen lähdekoodin asiakasohjelma Windows-käyttöjärjestelmiin, joka integroituu resurssienhallintaan (http://tortoisesvn.net/). Sen voi linkittää samaan laitoksen svn-palvelimeen kuin Linux-kone.

Lisää TortoiseSVN:än käytöstä linkissä: http://tortoisesvn.net/support.html, josta löytyy myös suomenkielinen käyttöopas.

Versionhallinnan linkkejä

Ohessa on muutamia hyödyllisiä linkkejä liittyen versionhallintaan/Subversioniin:

Linux:

http://linux.fi/wiki/Ohjelmien_asentaminen

SVN:

http://linux.fi/wiki/Subversion

http://svnbook.red-bean.com/

http://subversion.tigris.org/

TortoiseSVN:

http://tortoisesvn.net/support.html (myös suomenkielinen käyttöopas)

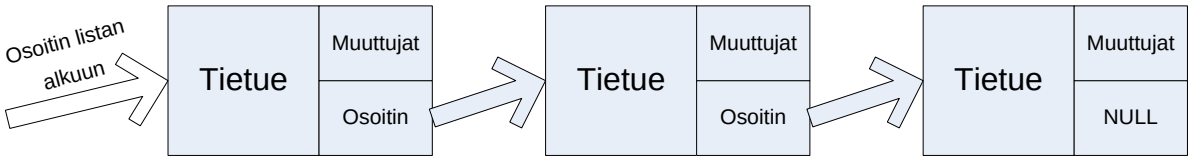
Osaamistavoitteet

Tämän luvun keskeisin asia on dynaamisen muistinvarauksen oikeaoppinen käyttö eli muistin varaaminen, varauksen onnistumisen tarkistaminen sekä muistin vapauttaminen käytön lopuksi. Omien tietotyyppien osalta tutustuimme kolmeen uuteen asiaan: yhdiste, lueteltu tyyppi ja uuden tietotyypin määrittely. Nämä kaikki asiat ovat C-kielen perusasioita, joista ehdottomasti tärkein on dynaaminen muistinvaraus, mutta myös tietotyyppien määrittely on C-kielen perustaitoja.

Toinen keskeinen asia tässä luvussa on versionhallinta. Käytännön ohjelmoinnissa versionhallinnan käyttö on keskeisessä roolissa, sillä ohjelmakoodi on ohjelmistoyrityksen tärkein omaisuus ja siten sen asianmukainen hallinta on erittäin tärkeää. Tässä yhteydessä keskityttiin yhteen versionhallintajärjestelmään, SVN:ään, ja se toimii hyvänä esimerkkinä tyypillisestä versionhallintajärjestelmästä. Käytännössä jokainen yritys valitsee itselleen sopivan järjestelmän ja ohjelmoijat käyttävät annettua järjestelmää, mutta perusperiaatteet tuntemalla siirtymien uusiin järjestelmiin ei yleensä vaadi suuria ponnisteluita.

Luku 7: Linkitetty lista

Kaikki tässä oppaassa käsitellyt C-kieleen liittyvät asiat voidaan esittää kootusti linkitetyn listan yhteydessä. Linkitetty lista tarkoittaa käytännössä joukkoa tietueita, jotka ovat kytketty toisiinsa tallentamalla jokaiseen tietueeseen tieto siitä, missä seuraava tietue on. Siksi linkitetyn listan tietueessa on oltava varsinaisten tietoa sisältävien muuttujien lisäksi yksi ylimääräinen muuttuja, joka on osoitin seuraavaan tietueeseen. Näin tietueet muodostavat ketjun linkitettyjä tietueita eli linkitetyn listan. Kun viimeisessä tietueessa on seuraavan tietueen osoitteen paikalla `NULL`-osoitin, tiedämme listan loppuvan siihen. Jotta pystymme käsittelemään haluttua listan alkioita, tarvitsemme lisäksi osoitinmuuttujan, joka avulla voimme siirtyä haluttuun alkioon ja käsitellä sen tietoja sekä tietysti osoittimen listan alkuun eli ensimmäiseen tietueeseen. Alla Kuva **3** näyttää linkitetyn listan periaatekuvan visuaalisesti.



Kuva 3: Yhteen suuntaan linkitetty lista, jossa on 3 tietuetta. Ensimmäiseen tietueeseen osoittaa erillinen osoitinmuuttuja.

Linkitetty lista on lähtökohtaisesti dynaaminen rakenne eli siihen lisätään ja siitä poistetaan tietueita tarpeen mukaan ohjelman suorituksen aikana. Listan käsittely perustuu aiemmin käsiteltyyn dynaamiseen muistinhallintaa (luku 6) ja osoittimiin (luvut 2 ja 6), joten nämä asiat on hyvä kerrata tässä vaiheessa.

Ohjelman suorituksen alkaessa lista muodostuu yleensä vain listan alkuun osoittavasta osoittimesta, jonka arvo on `NULL` tyhjän listan tapauksessa. Listaan luodaan uusia tietueita `malloc`-käskyllä ja ensimmäisen tietueen osoite laitetaan listan alkuun osoittavan osoittimen arvoksi; tietueessa olevan seuraavaan tietueeseen osoittavan osoittimen arvoksi tulee taas `NULL`, koska uusi tietue on lisätty listan viimeiseksi tietueeksi. Jatkossa listaan lisätään uusia tietueita tyypillisesti aina viimeiseksi tietueeksi, koska tällöin uuden tietueen seuraavaan tietueeseen osoittavan osoittimen arvoksi voidaan laittaa `NULL` ja uuden tietueen osoite voidaan sijoittaa aiemmin viimeisenä olleen tietueen seuraavaksi osoitteeksi. Kun listan tietueita poistetaan, tulee poistettavan tietueen edeltäjä-tietueen osoitekenttä päivittää sopivasti ja vapauttaa varattu tietue `free`-funktiolla. Näin listaan voidaan lisätä uusia ja poistaa tarpeettomia tietueita. Listan toimivuuden kannalta on tärkeää, että ensimmäiseen tietueeseen osoittava osoitin on aina kunnossa – siis joko tyhjän listan kohdalla `NULL` tai ei-tyhjän listan kohdalla ensimmäisen tietueen osoite.

Listan yhteydessä käytetään ”liukuri” osoitinta, joka osoittaa aina siihen tietueeseen, jota halutaan käsitellä. Käytäessä kaikki listan alkiot läpi osoittaa tämä liukuri-osoitin vuorotellen kaikkia listan alkioita. Samoin listaa tyhjennettäessä tarvitaan tällaista liukuri-tyyppistä apuosoitinta.

Linkitetty lista koetaan usein hankalana asiana. Tämä johtuu siitä, että linkitetyssä listassa on monta asiaa ja jos yhdessäkin niistä on virhe, ei ohjelma toimi oikein. Siksi on tärkeää, että linkitettyä listaa tehdessä on huolellinen ja varmistaa, että kaikki yksittäiset asiat tulee tehtyä oikein: osoittimet, tietueen määrittely, muistin varaaminen, osoitteiden ylläpitäminen mukaan lukien `NULL`-osoittimet, toistorakenteet listan läpikäynnissä, muistin vapauttaminen, jne. Näistä kaikista on puhuttu tämän oppaan aiemmissa luvuissa ja nyt ne vain esitetään yhdessä linkitetyn listan muodossa. Asian selventämiseksi katsotaan seuraavaksi esimerkki linkitetystä listasta. Esimerkin pituuden kasvaessa kommentit on lihavoitu auttamaan ohjelman rakenteen hahmottamisessa.

Esimerkki 7.1: Yhteen suuntaan linkitetty lista

Esimerkkikoodi

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

/* Minimaalinen tietue listalle */
struct asiakas {
    int numero;
    struct asiakas *seuraava;
};

int main(void) {
    /* Osoittimet listan ensimmäiseen, viimeiseen ja käsiteltävään solmuun */
    struct asiakas *pAlku = NULL, *pLoppu = NULL, *ptr;
    int i;

    /* Listan luominen, 3 alkiota malliksi */
    for (i = 0; i < 3; i++) {
        /* Muistin varaus */
        if ((ptr = (struct asiakas*)malloc(sizeof(struct asiakas)))==NULL) {
            perror("Muistin varaus epäonnistui");
            exit(1);
        }
        /* Uuden alkion arvojen määrittely */
        ptr->numero = i;
        ptr->seuraava = NULL;
        /* Uuden alkion lisääminen listaan viimeiseksi alkioksi */
        if (pAlku == NULL) { /* Tyhjä lista */
            pAlku = ptr;
            pLoppu = ptr;
        } else { /* Lisätään listan loppuun */
            pLoppu->seuraava = ptr;
            pLoppu = ptr;
        }
    }

    /* Listan läpikäynti */
    ptr = pAlku;
    while (ptr != NULL) {
        printf("Nyt palvelemme numeroa %d.\n", ptr->numero);
        ptr = ptr->seuraava;
    }

    /* Muistin vapauttaminen */
    ptr = pAlku;
    while (ptr != NULL) {
        pAlku = ptr->seuraava;
        free(ptr);
        ptr = pAlku;
    }

    printf("Viimeisimmän asiakkaan jälkeen ei ollut enää ketään.\n");
    return 0;
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen, saamme seuraavanlaisen tuloksen:

```
un@linux:~/un$ ./7-1
Nyt palvelemme numeroa 0.
Nyt palvelemme numeroa 1.
Nyt palvelemme numeroa 2.
Viimeisimmän asiakkaan jälkeen ei ollut enää ketään.
un@linux:~/un$
```

Kuinka koodi toimii

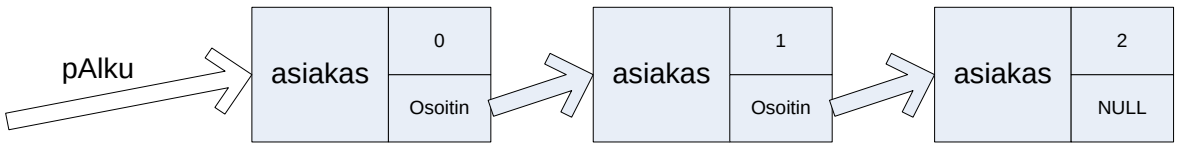
Koodin alussa luodaan uusi `asiakas`-tietue, joka sisältää varsinaisena tietona yksinkertaisuuden vuoksi vain asiakkaan numeron. Tämän lisäksi tietueessa on `seuraava`-osoitin, johon tallennetaan seuraavan tietueen sijainti.

Varsinainen ohjelma alkaa luomalla kolme osoitinta `asiakas`-tietueeseen. `pAlku`-osoitin osoittaa aina listan ensimmäiseen alkioon, koska emme saa millään muulla tavalla selville listan ensimmäisen tietueen sijaintia. `pLoppu`-osoitin osoittaa aina listan viimeiseen tietueeseen ja sitä käytetään vain listan luomisen helpottamiseksi eli uuden tietueen osoite korvaa aina tämän tietueen `seuraava`-osoittimen `NULL`-arvon. Tämä loppu-osoitin ei ole välttämätön, mutta se helpottaa listan luomista. Kolmantena osoittimena on liukuriosoitin `ptr`, jota käytetään kun lista pitää käydä läpi tai halutaan löytää tietty tietue listasta.

Seuraavaksi koodissa luodaan dynaamisesti `malloc`-käskyllä uusia tietueita ja jokainen uusi tietue laitetaan listan viimeiseksi alkioksi alla olevalla koodilla. Huomaa, että tyhjä lista pitää käsitellä erikseen, sillä tällöin uuden tietueen osoite laitetaan `pAlku`-osoittimen arvoksi kun muutoin se laitetaan `asiakas`–tietueen `seuraava`-osoittimen arvoksi. Mikäli meillä ei ole `pLoppu`-osoitinta käytössä, pitää listan viimeinen tietue etsiä listan alkuosoitteen perusteella joka kerta erikseen.

```
/* Uuden alkion lisääminen listaan viimeiseksi alkioksi */
if (pAlku == NULL) { /* Tyhjä lista */
        pAlku = ptr;
        pLoppu = ptr;
} else { /* Lisätään listan loppuun */
        pLoppu->seuraava = ptr;
        pLoppu = ptr;
}
```

Listan luomisen tuloksena meillä on seuraavanlainen linkitetty lista:



Kuva 2: Esimerkkikoodin luoma linkitetty lista

Tässä vaiheessa meillä on normaali linkitetty lista. Listan läpikäynti on nyt helppoa: (1) ensimmäinen tietue löytyy `pAlku`-osoittimen avulla, (2) seuraava tietue löytyy aina `seuraava`-osoittimen avulla ja (3) listan loppuminen näkyy `seuraava`-osoittimesta sen arvon ollessa `NULL`.

```
ptr = pAlku;
while (ptr != NULL) {
     printf("Nyt palvelemme numeroa %d.\n", ptr->numero);
     ptr = ptr->seuraava;
}
```

Ohjelman lopuksi sen varaama muisti pitää vapauttaa. Tämä perustuu yllä olevaan listan läpikäyntirutiinin, mutta nyt jokainen läpikäyty tietue vapautetaan `free`-käskyllä. Huomaa, että tässä tarvitaan kaksi osoitinta ja niiden siirtojärjestys sekä `free`-käskyn oikea sijoitus ovat erittäin tärkeitä ohjelman oikean toiminnan kannalta.

```
ptr = pAlku;
while (ptr != NULL) {
       pAlku = ptr->seuraava;
       free(ptr);
       ptr = pAlku;
}
```

Periaatteessa tässä oli linkitetty lista kaikessa yksinkertaisuudessaan. Toki listan käsittely on haastavampaa kun ohjelman koko kasvaa ja toimintoja siirretään aliohjelmiin, mutta linkitetyn listan perustoiminta ja keskeisimmät asiat käyvät ilmi edellä käsitellystä esimerkistä.

Osaamistavoitteet

Tässä luvussa keskityttiin linkitettyyn listaan eli siihen, miten yksittäiset tietueet voivat sisältää seuraavan tietueen osoitteen ja muodostaa linkitetyn listan. Linkitetty lista on lähtökohtaisesti dynaaminen rakenne eli siihen kuuluvia tietueita varataan ja vapautetaan tarpeen mukaan ohjelman suorituksen aikana. Käsitelty esimerkki osoittaa, miten yhdestä tietueesta päästään toiseen tietueeseen ja miten liukuri-osoittimen avulla voidaan käydä läpi kaikki linkitetyn listan alkiot sekä niiden sisältämät tiedot. Tässä vaiheessa sinun tulee siis pystyä toteuttamaan yhteen suuntaan linkitetty lista, jossa tietueiden määrää voidaan muuttaa dynaamisesti ohjelman suorituksen aikana.

Loppusanat

Olemme tässä oppaassa käyneet läpi C-ohjelmoinnin alkeet lyhyiden esimerkkien ja ohjetekstien avulla sekä esitelleet lyhyesti Linux-käyttöympäristöä ja SVN-versionhallintaa. Tässä oppaassa ei käsitelty laajemmin ”käytännön ohjelmointia”, joka pitää sisällään mm. ohjelmien suunnittelua ja laadunvarmistusta. Näihin asioihin palataan kurssin oppaan toisessa osassa.

C-ohjelmointiin liittyen on olemassa paljon hyviä kirjoja. Paras lähde C-kielen ymmärtämiseen on Kerninghanin ja Ritchien kirjoittama The C Programming Language (Kernighan ja Ritchie 1988) kun taas C-kielen käyttämisen kannalta hyvä opas on aiemminkin mainittuun Eric Hussin C-referenssikirjasto (Huss 1997). C-kieleen on saatavilla myös muita suomenkielisiä oppaita, mutta pääosa kirjallisuudesta on englanninkielistä. Käytännön ohjelmointia käsitellään laajemmin Kernighanin ja Piken (2005) kirjassa The Practice of Programming.

Mikäli kaipaat suomenkielistä lisämateriaalia, voit turvautua Internetin hakukoneisiin kuten Google tai www.fi. Molemmissa hakukoneissa on optio rajoittaa hakua ainoastaan suomenkielisiin sivustoihin. Älä myöskään unohda tämän oppaan liitettä 1, johon on koottu referenssikirjastosta tavallisimpien käskyjen lisäksi myös muita käsittelemättä jääneitä asioita.

Lähdeluettelo

Alaoutinen, Satu. 2005. Lyhyt C-opas. Lappeenrannan teknillinen yliopisto, Tietotekniikan osasto. Saatavilla osoitteesta http://www.it.lut.fi/kurssit/05-06/Ti5210210/opas/c_opas/c_opas.html.

Huss, Eric. 1997. The C Library Reference Guide. Saatavilla osoitteesta http://www.acm.uiuc.edu/webmonkeys/book/c_guide/, viitattu 18.12.2007.

Keizer, G. 2010, *Windows market share slide resumes*, Computerworld, [viitattu 22.03.2011], <URL:http://www.computerworld.com/s/article/9142978/Windows_market_share_slide_resumes >.

Kernighan, Brian W., Ritchie, Dennis M. 1988. The C Programming Language, Second edition. Prentice Hall Software Series.

Kernighan, Brian W., Pike, Rob. 2005. The Practice of Programming, 13th edition. Addison-Wesley Professional Computing Series.

Kyttälä, Lauri ja Nikula, Uolevi. 2012. C-kieli ja käytännön ohjelmointi – Osa 2, Versio 1.0. Lappeenrannan teknillinen yliopisto, Tietotekniikan käsikirjat 15.

Glen Moody. 2001. Kapinakoodi, Tammi.

Robot Wisdom. 2011. [viitattu 10.5.2011], <URL:http://www.robotwisdom.com/linux/timeline.html>.

Vanhala, Erno ja Nikula, Uolevi. 2011. Python 3 -ohjelmointiopas, versio 1.1. Lappeenrannan teknillinen yliopisto, Tietotekniikan käsikirjat 14.

Wikipedia. 2011. X Window System [viitattu 01.04.2011], <URL:http://fi.wikipedia.org/wiki/X_Window_System>.

Liite 1: Kirjastot ja otsikkotiedostot

Tässä liitteessä käymme läpi kootusti keskeisiä referenssikirjaston käskyjä ja vakioita.

float.h

DBL_MIN	Pienin positiivinen kaksoistarkkuuden luku
DBL_MAX	Suurin kaksoistarkkuuden luku

limits.h

CHAR_MIN	Merkkityypin pienin arvo
CHAR_MAX	Merkkityypin suurin arvo
INT_MIN	Pienin kokonaisluku
INT_MAX	Suurin kokonaisluku

ctype.h

isalnum(x)	Testaa, onko x alfanumeerinen merkki
isalpha(x)	Testaa, onko x kirjain
isdigit(x)	Testaa, onko x numeromerkki
islower(x)	Testaa, onko x pieni kirjain
isspace(x)	Testaa, onko x tyhjä merkki (välilyönti, rivinvaihto,...)
isupper(x)	Testaa, onko x iso kirjain
tolower(x)	Muuntaa x:n vastaavaksi pieneksi kirjaimeksi
toupper(x)	Muuntaa x:n vastaavaksi isoksi kirjaimeksi

math.h

Haluttaessa käyttää math.h:n funktioita, on käännettäessä lisättävä rivin perään -lm (esim. gcc testi.c -o testi -lm)

abs(x) - Kokonaisluvun itseisarvo

acos(x) - Arkuskosini x

asin(x) - Arkussini x

atan(x) - Arkustangentti x

ceil(x) - Palauttaa pienimmän kokonaisluvun, joka on suurempi kuin x

cos(x) - Kosini x

cosh(x) - Hyperbolinen kosini x

exp(x) - e potenssiin x

fabs(x) - Reaaliluvun itseisarvo

floor(x) - Palauttaa suurimman kokonaisluvun, joka on pienempi kuin x

log(x) - ln x

log10(x) - log10 x

pow(x,y) - x potenssiin y

sin(x) - Sini x

sinh(x) - Hyperbolinen sini x

sqrt(x) - Neliöjuuri x

tan(x) - Tangentti x

tanh(x) - Hyperbolinen tangentti x

stdio.h

Sisältää kaikki syöttö-, tulostus- ja tiedostojenkäsittelyfunktiot, tyyppien määrittelyjä ja vakioita.

EOF - (-1), tiedoston loppu

NULL - (0), osoittimen arvo 0

stdin - standardi syöttötiedosto, normaalisti näppäimistö

stdout - standardi tulostustiedosto, normaalisti näyttö

stderr - standardi virhetiedosto, normaalisti näyttö

SEEK_SET - (0), tiedoston alku

SEEK_CUR - (1), nykyinen tiedostosijainti

SEEK_END - (2), tiedoston loppu

FILE - rakenteinen tyyppi tiedostojen käsittelyyn

fpos_t - tyyppi, jota käytetään haluttaessa määrittää yksikäsitteisesti sijainti tiedoston sisällä

fopen(tiedostonimi, avausmuoto) - avaa tiedoston `tiedostonimi` `avausmuoto` kertomaan käyttötarkoitukseen

fclose(tiedosto) - sulkee `tiedosto` tiedoston

fflush(tiedosto) - tyhjentää tiedoston `tiedosto` puskurin

fseek(tiedosto, etäisyys, paikka) - siirtyy tiedostossa `tiedosto` `etäisyys` tavua kohdasta `paikka` laskien. `paikka` on joko SEEK_SET, SEEK_CUR tai SEEK_END

ftell(tiedosto) - kertoo sijainnin tiedostossa `tiedosto`

rewind(tiedosto) - siirtyy tiedoston alkuun.

fgetpos(tiedosto, sijainti) - sijoittaa parametrin sijainti arvoksi nykyisen kohdan tiedostossa

fsetpos(tiedosto, sijainti) - siirtyy tiedostossa kohtaan `sijainti`

feof(tiedosto) - palauttaa nollasta poikkeavan arvon, jos ollaan tiedoston lopussa

perror(merkkijono) - tulostaa käyttäjän oman virheilmoituksen `merkkijono` yhdistettynä järjestelmän virheilmoitukseen standardi virhevirtaan stderr

getc(tiedosto) - lukee seuraavan merkin tiedostosta. Makro

getchar() - lukee merkin standardisyöttövirrasta (näppäimistöltä). Makro

gets(merkkijono) - lukee merkkijonon standardisyöttövirrasta (näppäimistöltä) ja sijoittaa sen parametrin `merkkijono` arvoksi. Makro

fgetc(tiedosto) - lukee seuraavan merkin tiedostosta

fgets(rivi, määrä, tiedosto) - lukee enintään määrä-1 merkkiä tiedostosta ja sijoittaa ne merkkijonoon rivi. Rivin loppu tai tiedoston loppu lopettavat myös lukemisen

fputc(merkki, tiedosto) - kirjoittaa merkin tiedostoon

fputs(s, tiedosto) - kirjoittaa merkkijonon `s` tiedostoon

putc(merkki, tiedosto) - kirjoittaa merkin tiedostoon. Makro

putchar(merkki) - tulostaa merkin standarditulostusvirtaan (näytölle). Makro

puts(merkkijono) - tulostaa merkkijonon standarditulostusvirtaan (näytölle). Makro

fprintf(tiedosto, formaatti, muuttujalista) - kirjoittaa muotoiltua tekstiä tiedostoon

printf(formaatti, muuttujalista) - kirjoittaa muotoiltua tekstiä standarditulostusvirtaan (näytölle)

sprintf(merkkijono, formaatti, muuttujalista) - kirjoittaa muotoiltua tekstiä merkkijonoon

fscanf(tiedosto, formaatti, muuttujalista) - lukee tekstiä tiedostosta ja sijoittaa sen muuttujalistan muuttujiin

scanf(formaatti, muuttujalista) - lukee tekstiä standardisyöttövirrasta (näppäimistöltä) ja sijoittaa sen muuttujalistan muuttujiin

sscanf(merkkijono, formaatti, muuttujalista) - lukee tekstiä merkijonosta ja sijoittaa sen muuttujalistan muuttujiin

fread(rakenne, koko, määrä, tiedosto) - binaaritiedoston käsittelyyn. Lukee tiedostosta enintään `määrä * koko` tavua ja sijoittaa sen muuttujan `rakenne` osoittamaan paikkaan

fwrite(rakenne, koko, määrä, tiedosto) - binaaritiedoston käsittelyyn. Lukee muuttujasta `rakenne` `määrä * koko` tavua ja kirjoittaa sen tiedostoon

remove(tiedosto) - poistaa tiedoston

rename(mjono1, mjono2) - muuttaa tiedoston nimen `mjono1`:stä `mjono2`:ksi

stdlib.h

atoi(x) Muuttaa merkkijonon x kokonaisluvuksi

atof(x) Muuttaa merkkijonon x liukuluvuksi

rand() Antaa satunnaisluvun väliltä 0..RAND_MAX

srand(x) Alustaa satunnaislukugeneraattorin siemenluvulla x

malloc(x) Varaa muistia x:n kokoisen alueen. Palauttaa osoittimen alueen alkuun

free(x) Vapauttaa x:n osoittaman muistitilan

system(x) Suorituttaa käyttöjärjestelmällä x:n osoittaman komennon. `system("rmtiedot.txt");`

string.h

Sisältää merkkijonojen käsittelyyn tarvittavat funktiot. str-alkuisia käytetään loppumerkillisten (\0) merkkijonojen käsittelyyn ja mem-alkuisia loppumerkittömien käsittelyyn. memxxx -funktiossa on annettava merkkijonon pituus. strnxxx-funktioissa merkkijonossa oletetaan olevan loppumerkki mutta annetaan enimmäispituus.

memchr(s1, c, n)

etsii merkkiä c osoitteesta s1 alkavasta merkkijonosta. Käy läpi enintään n merkkiä.

memcmp(s1, s2, n)

vertailee kahta enintään n:n pituista merkkijonoa. Funktio palauttaa negatiivisen arvon, jos s1 on aakkosissa ennen s2:ta, positiivisen jos s1 on s2:n jälkeen ja nollan, jos s1 ja s2 ovat samat

memcpy(s1, s2, n)

kopioi s2:n osoittamasta paikasta enintään n merkkiä s1:n osoittamaan paikkaan

memset(s1, c, n)

asettaa s1:n osoittaman, enintään n:n pituisen merkkijonon kaikki merkit c:ksi

strcat(s1, s2)

kopioi s2:n osoittaman merkkijonon s1:n osoittaman merkkijonon perään

strchr(s1, c)

etsii merkkijonosta s1 merkkiä c ja palauttaa osoittimen löydettyyn merkkiin

strcmp(s1, s2)

vertaa s1:n osoittamaa merkkijonoa s2:n osoittamaan merkkijonoon. Jos s1 < s2, tulos on <0, jos s1==s2, tulos on 0 ja jos s1>s2, tulos on >0

strcpy(s1, s2)

kopioi s2:n osoittaman merkkijonon s1:n osoittamaan paikkaan

strlen(s)

antaa merkkijonon s pituuden

strstr(s1, s2)

etsii merkkijonosta s1 merkkijonoa s2

strtok(s1, s2)

käyttäen s2:ssa olevia merkkejä erottimina jakaa merkkijonoa s1 osiin

strncat(s1, s2, n)

kopioi s2:n osoittaman, enintään n:n pituisen merkkijonon, s1:n osoittaman merkkijonon perään

strncmp(s1,s2, n)

vertaa s1:n osoittamaa merkkijonoa s2:n osoittamaan merkkijonoon. Funktio ottaa huomioon enintään n merkkiä kummastakin. Jos s1 < s2, tulos on <0, jos s1==s2, tulos on 0 ja jos s1>s2, tulos on >0

strncpy(s1,s2, n)

kopioi s2:n osoittamasta merkkijonosta n merkkiä tai loppumerkkiin \0 asti s1:n osoittamaan paikkaan