

LAPPEENRANNAN-LAHDEN TEKNILLINEN YLIOPISTO LUT
LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT

School of Engineering Science

Ohjelmistotuotanto

LUT Scientific and Expertise Publications

Oppimateriaalit – Lecture Notes

24

Uolevi Nikula

C-ohjelmointiopas versio 2.1



Lappeenrannan-Lahden teknillinen yliopisto LUT

School of Engineering Science, Ohjelmistotuotanto

LUT Scientific and Expertise Publications: Oppimateriaalit – Lecture Notes 24

Uolevi Nikula

C-ohjelmointiopus

versio 2.1

LUT-yliopisto
School of Engineering Science
Ohjelmistotuotanto
PL 20
53851 Lappeenranta

ISBN 978-952-335-684-9
ISBN 978-952-335-685-6 (PDF)
ISSN-L 2243-3392
ISSN 2243-3392

Lappeenranta 2021

Tähän dokumenttiin sovelletaan Creative Commons 4.0 Nimeä - Ei kaupallinen - Jaa samoin – lisenssiä. Opas on ei-kaupalliseen opetuskäyttöön suunnattu käsikirja.

Lappeenranta 16.6.2021

Tämä ohjelmointiopas on tarkoitettu ohjeeksi, jonka avulla lukija voi perehtyä C-ohjelmointikieleen sekä sen käyttämiseen ohjelmointiprojektien työvälineenä. Ohjeet sekä esimerkkitehtävät on suunniteltu siten, että niiden ei pitäisi aiheuttaa ei-toivottuja sivuvaikutuksia, mutta siitäkin huolimatta lopullinen vastuu harjoitusten suorittamisesta on käyttäjällä. Oppaan tekemiseen osallistuneet henkilöt taikka Lappeenrannan teknillinen yliopisto eivät vastaa käytöstä johtuneista suorista tai epäsuorista vahingoista, vioista, ongelmista, tappioista tai tuotannon menetyksistä.

ESIPUHE

Tämä on yksi LUT-yliopiston Tietotekniikan koulutusohjelman ohjelmoinnin perusopetuksen oppaista. Tämä ei ole täydellinen C-ohjelmointikielen käsikirja, vaan sen tarkoituksena on auttaa Python-kielen osaavia aloittamaan C-kielen opiskelu käymällä läpi perusrakenteet C-kielen syntaksilla ja esittelemällä keskeiset C-kielen erityispiirteet.

Oppaan esimerkeissä oletetaan, että käyttäjä osaa ohjelmoida Python-ohjelmointikielellä. Ohjelmointiympäristönä oppaassa on käytetty Win10-käyttöjärjestelmään asennettua Visual Studio Code -editoria ja esimerkit on käännetty ja testattu Linux-ympäristössä gcc-kääntäjällä. Opas ei edellytä varsinaista Linux-osaamista vaan käytettävät yksittäiset käskyt käydään läpi oppaassa ja ne oppii tehtäviä tekemällä.

Mikäli sinulla ei ole aiempaa kokemusta ohjelmoinnista, kannattaa ohjelmoinnin opiskelu aloittaa Python-ohjelmointioppaan avulla (Vanhala ja Nikula 2020). Opas on ladattavissa ilmaiseksi lutpub.lut.fi -osoitteesta ja siinä käydään läpi ohjelmointia perusteista alkaen. Se on suositeltava lähtökohta tämän oppaan aiheisiin, sillä tämä opas etenee nopeammin ja edellyttää aiempaa ohjelmointikokemusta.

Oppaan ensimmäisessä versiossa (Kasurinen ja Nikula 2011) käytettiin osia Satu Alaoutisen (2005) Lyhyestä C-oppaasta. Vuonna 2013 opasta laajennettiin lisäämällä osia oppaasta ”C-kieli ja käytännön ohjelmointi Osa 2” (Kyttälä ja Nikula 2012) ja opas julkaistiin nimellä ”C-kieli ja käytännön ohjelmointi osa 1, versio 2” (Kasurinen ja Nikula 2013). Tämän version muotoon myötävaikuttivat erityisesti Risto Westman, Satu Alaoutinen ja Erno Vanhala, joista viimeinen on osallistunut merkittävästi erityisesti Unix/Linux-osuuksien kirjoittamiseen. Lisäksi oppaan kaikkiin versioihin on saatu kommentteja opiskelijoilta.

Oppaan uusin versio on julkaistu 2021 nimellä ”C-ohjelmointiopas versio 2.1”. LUTin kurssin sisällöt ovat kehittyneet aikojen kuluessa ja aiemmin käytetystä Linux-ympäristöstä on tässä vaiheessa siirrytty Visual Studio Code -editoriin ja WSL-ympäristöön (Windows Subsystem for Linux). Tämä mahdollistaa ohjelmien tekemisen monille tutussa Windows-ympäristössä, mutta ohjelmat käännetään ja suoritetaan Linux-ympäristössä hyväksi koettujen työkalujen gcc, make ja Valgrind avulla.

Suuret kiitokset kaikille opasta kommentoineille ja lukeneille henkilöille – turhahan näitä oppaita olisi kirjoittaa, jos niitä ei kukaan lukisi.

SISÄLLYSLUETTELO

| | |
|---|-----------|
| VALMISTELU | 9 |
| LUKU 1. C-OHJELMOINNIN TAUSTAA JA PERUSTEET | 10 |
| UNIX, LINUX JA OHJELMOINTIYMPÄRISTÖT | 10 |
| C-KIELEN TAUSTAA | 12 |
| C-OHJELMAN KÄÄNTÄMINEN..... | 12 |
| <i>C-ohjelman kääntämisprosessi</i> | <i>12</i> |
| <i>Huomioita kääntäjistä ja käyttöjärjestelmästä.....</i> | <i>13</i> |
| C-OHJELMAN PERUSASIOITA | 14 |
| Syntaksi | 14 |
| Tekstin tulostaminen näytölle..... | 16 |
| Muuttujan määrittely..... | 17 |
| Tietojen lukeminen käyttäjältä | 19 |
| Merkkijonot, merkkitaulukon koko ja merkkijonon loppumerkki | 20 |
| Rivin lukeminen..... | 23 |
| Muotoilumerkit tiedon tulostuksessa ja lukemisessa | 25 |
| SÄÄNTÖJÄ JA TAULUKOITA..... | 26 |
| Kokonaisluvut ja liukuluvut..... | 27 |
| Operaattorit | 28 |
| Operaattorien suoritusjärjestys eli presedenssi | 30 |
| Muotoilumerkkijonon liput ja muunnosmerkit | 30 |
| C-KIELEN OMINAISPIIRTEITÄ | 31 |
| Tietotyyppimuunnokset ja kokonaislukujako..... | 32 |
| Tiivis koodi..... | 32 |
| PIENEN C-OHJELMAN TYYLIOHJEET | 33 |
| YHTEENVETO..... | 35 |
| Osaamistavoitteet..... | 35 |
| Käsitellyt asiat..... | 35 |
| Käytetyt työkalut: Win10, VSC, WSL1, Linux, gcc..... | 35 |
| Kokoava esimerkki | 36 |
| LUKU 2. VALINTA- JA TOISTORAKENTEET, ESIKÄSITTELIJÄ..... | 37 |
| VALINTARAKENTEET JA EHDOLLINEN SUORITTAMINEN..... | 37 |
| if-else..... | 37 |

| | |
|---|-----------|
| <i>switch-case</i> | 39 |
| <i>Ehdollinen lauseke</i> | 40 |
| TOISTORAKENTEET | 41 |
| <i>while-rakenne</i> | 41 |
| <i>for-rakenne</i> | 42 |
| <i>do-while -rakenne</i> | 43 |
| MUITA OHJAUSKÄSKYJÄ | 44 |
| <i>return, continue, break</i> | 44 |
| <i>goto-käskyn käyttö kielletty</i> | 46 |
| ESIKÄSITTELIJÄ | 46 |
| <i>Kirjastojen sisällyttäminen #include-direktiivillä</i> | 46 |
| <i>Vakioiden määrittely #define-direktiivillä vs. C-kielen const-määrittely</i> | 47 |
| <i>Ehdollinen koodilohko #if 0 ... #endif-direktiiveillä</i> | 49 |
| C-KIELEN OMINAISPIIRTEITÄ | 49 |
| <i>Kääntäjäoptiot apuna virheiden välttämässä ja etsimisessä</i> | 50 |
| <i>Linuxin man-sivut eli manuaali</i> | 51 |
| <i>Merkki, merkkijono, merkkitaulukko ja osoite</i> | 52 |
| <i>Osoittimen käyttö merkkijonon läpikäynnissä</i> | 53 |
| <i>Staattinen muistinvaraus</i> | 55 |
| PIENEN C-OHJELMAN TYYLIOHJEET | 55 |
| YHTEENVETO | 55 |
| <i>Osaamistavoitteet</i> | 56 |
| <i>Käsitellyt asiat</i> | 56 |
| <i>Kokoava esimerkki</i> | 56 |
| LUKU 3. TIEDOSTONKÄSITTELY JA ALIOHJELMAT | 58 |
| TIEDOSTONKÄSITTELY | 58 |
| <i>Tiedoston kirjoittaminen</i> | 58 |
| <i>Tiedoston lukeminen</i> | 59 |
| <i>Tiedostonkäsitteilyn virheentarkastus</i> | 60 |
| <i>Binaaritiedostojen käsittely</i> | 61 |
| <i>Tiedostonkäsitteilyn perusfunktioita</i> | 61 |
| ALIOHJELMAT | 62 |
| <i>Määrittely ja kutsuminen</i> | 63 |

C-ohjelmointiopas versio 2.1

| | |
|--|-----------|
| <i>Arvoparametrit ja paluuarvo</i> | <i>64</i> |
| <i>Muuttujaparametrit eli osoittimet aliohjelmissä</i> | <i>65</i> |
| <i>Tunnusten näkyvyys.....</i> | <i>68</i> |
| C-KIELEN OMINAISPIIRTEITÄ | 70 |
| <i>Tiedonvälitys aliohjelmaan ja takaisin kutsuvaan ohjelmaan.....</i> | <i>70</i> |
| <i>Tietovirrat ja tiedostot Unixissa.....</i> | <i>72</i> |
| <i>Tiedostoformaateista.....</i> | <i>73</i> |
| <i>Makrot.....</i> | <i>74</i> |
| <i>Kirjastofunktiot</i> | <i>75</i> |
| PIENEN C-OHJELMAN TYYLIOHJEET | 76 |
| YHTEENVETO..... | 76 |
| <i>Osaamistavoitteet.....</i> | <i>77</i> |
| <i>Käsitellyt asiat.....</i> | <i>77</i> |
| <i>Kokoava esimerkki</i> | <i>77</i> |
| LUKU 4. TIETORAKENTEITA JA ALGORITMEJA..... | 80 |
| TIETORAKENTEITA..... | 80 |
| <i>Taulukot</i> | <i>80</i> |
| <i>Tietue.....</i> | <i>82</i> |
| <i>Uuden tietotyypin määrittely.....</i> | <i>84</i> |
| <i>Komentoriviparametrit.....</i> | <i>85</i> |
| <i>Tietue-osoitin</i> | <i>87</i> |
| ALGORITMEJA | 88 |
| <i>Taulukon alkoiden käsittely.....</i> | <i>88</i> |
| <i>Rekursio</i> | <i>89</i> |
| C-KIELEN OMINAISPIIRTEITÄ | 91 |
| <i>Osoittimien käyttö</i> | <i>91</i> |
| <i>Tietue ja binaaritiedosto</i> | <i>92</i> |
| PIENEN C-OHJELMAN TYYLIOHJEET | 93 |
| YHTEENVETO..... | 93 |
| <i>Osaamistavoitteet.....</i> | <i>93</i> |
| <i>Käsitellyt asiat.....</i> | <i>93</i> |
| <i>Kokoava esimerkki</i> | <i>94</i> |
| LUKU 5. MUISTINHALLINTA, TIETOTYYPIT | 96 |

C-ohjelmointiopas versio 2.1

| | |
|--|------------|
| KÄYTTÄJÄN MÄÄRITTELEMÄT TIETOTYYPIT JA MUISTIN KÄYTÖSTÄ | 96 |
| DYNAAMINEN MUISTINHALLINTA..... | 98 |
| <i>Huomioita muistinvarauksesta</i> | 101 |
| <i>Dynaaminen muistinhallinta ja tietue</i> | 102 |
| C-KIELEN OMINAISPIIRTEITÄ | 103 |
| <i>Merkkijonoista</i> | 103 |
| <i>Tietorakenteet ja aliohjelmat</i> | 104 |
| PIENEN C-OHJELMAN TYYLIOHJEET | 106 |
| YHTEENVETO..... | 107 |
| <i>Osaamistavoitteet</i> | 107 |
| <i>Käsitellyt asiat</i> | 107 |
| <i>Kokoava esimerkki</i> | 107 |
| LUKU 6. LINKITETTY LISTA JA AIKA C-OHJELMISSA | 110 |
| LINKITETTY LISTA | 110 |
| <i>Taulukoista linkitettyihin tietorakenteisiin</i> | 110 |
| <i>Linkitetyn listan toimintaperiaate</i> | 112 |
| <i>Linkitetyn listan toteutus yhden ohjelman sisällä</i> | 113 |
| <i>Linkitetyn listan toteutus aliohjelmina</i> | 118 |
| AJAN KÄSITTELY C-OHJELMISSA | 121 |
| <i>Unix-järjestelmän aika, Epoch</i> | 121 |
| <i>Ajan perusoperaatiot</i> | 122 |
| VARATUN MUISTIN VAPAUTUKSEN TARKISTAMINEN | 125 |
| PIENEN C-OHJELMAN TYYLIOHJEET | 126 |
| YHTEENVETO..... | 127 |
| <i>Osaamistavoitteet</i> | 127 |
| <i>Käsitellyt asiat</i> | 127 |
| <i>Käytetyt työkalut: Valgrind</i> | 127 |
| LUKU 7. KASVAVIEN C-OHJELMIEN TOTEUTUS..... | 129 |
| MONESTA TIEDOSTOSTA MUODOSTUVA MINIMAALINEN C-OHJELMA | 129 |
| OTSIKKOTIEDOSTO .H | 131 |
| LAAJAN OHJELMAN KÄÄNTÄMINEN – MAKE JA MAKEFILE | 133 |
| YHTEENVETO..... | 134 |
| <i>Osaamistavoitteet</i> | 134 |

| | |
|---|------------|
| <i>Käsitellyt asiat.....</i> | <i>134</i> |
| <i>Käytetyt työkalut: make ja Makefile.....</i> | <i>135</i> |
| LOPPUSANAT | 136 |
| LÄHDELUETTELO | 137 |
| LIITTEET | 138 |

Valmistelu

Ohjelmointi edellyttää ohjelmointioppaan lisäksi tietokoneelle ohjelmointiympäristöä. Tämän ohjelmointioppaan liitteenä 1 on Asennusohje, jossa käydään läpi Visual Studio Code (VSC) -editoriin ja WSL:ään (Windows Subsystem for Linux) perustuvan ohjelmointiympäristön asennus. LUTin ohjelmointikurssien materiaaleihin kuuluu myös ohjelmointivideoita, joista ensimmäisessä käydään läpi em. ohjelmointiympäristön asennus. Asennusohjeen lopussa tehdään pieni C-ohjelma ja käännetään sekä suoritetaan se, jotta voit varmistua ohjelmointiympäristösi toimivuudesta ja keskittyä C-ohjelmoinnin opetteluun varsinaisen oppaan avulla. Tämä opas keskittyy ohjelmointiin ja olettaa, että pystyt kääntämään ja ajamaan ohjelmat itse omassa ohjelmointiympäristössäsi.

Luonnollisesti Linux-ympäristöä voi käyttää suoraan oppaan esimerkkien tekemiseen. Myös Apple/Mac tuotteet perustuvat siinä määrin Unixiin, että asennuksissa voi tyypillisesti noudattaa Linux-ohjeita. Sekä Linux että Mac-ympäristöihin on saatavilla VSC-editori, joten näihin kaikkiin ympäristöihin on saatavissa vastaavat ohjelmointiympäristöt ja niiden asennuksen jälkeen oppaan esimerkit toimivat kaikissa ympäristöissä samalla tavalla. Tämän oppaan liitteenä 2 on lyhyt VSC:n käyttöohje.

Luku 1. C-ohjelmoinnin taustaa ja perusteet

Tässä luvussa tutustumme Unix/Linux käyttöjärjestelmiin sekä käymme läpi Linux-ohjelmointiympäristön ja C-kielen perusasiat. Unix/Linux-osuus tarjoaa perustiedot näiden käyttöjärjestelmien historiasta ja eroista sekä C-kielen roolista Unixin yhteydessä. Vaikka Windows on työpöytäympäristössä yleisin käyttöjärjestelmä, on Linuxilla valta-asema supertietokoneissa, älypuhelimissa, tableteissa, pelikonsoleissa ja televisioissakin. Linuxin rooli on myös vahvistumassa, sillä ensin meillä oli yhdessä tietokoneessa aina yksi käyttöjärjestelmä, jonka jälkeen virtuaalikoneet mahdollistivat useiden eri käyttöjärjestelmien käytön yhdessä tietokoneessa. Tästä seuraavana vaiheena Microsoft on mahdollistanut Linuxin käytön Windows'ssa omana järjestelmänään eli Windows Subsystem for Linux (WSL), joka tekee Linuxin käytöstä Windows-koneissa entistä helpompaa. Näin ollen Unix/Linux-historian ymmärtäminen on tärkeää ohjelmistotalan asiantuntijoille. Tämän luvun käytännöllinen tavoite on varmistaa, että ymmärrät Linuxin ja C-kielen lähtökohdat ja pystyt kirjoittamaan sekä kääntämään C-kielisiä ohjelmia. Tämä luvun Linux-osuus perustuu vahvasti lähteeseen Moody (2001).

Tämä opas on tehty lähtien oletuksesta, että tunnet Windows-ympäristön Linux-ympäristöjä paremmin. Windows käyttöjärjestelmä on työasemamarkkinoiden yleisin käyttöjärjestelmä 75 % markkinaosuudellaan (Statcounter 2021) ja se edustaa käyttöjärjestelmänä puhtaasti kaupallista linjaa. Perinteisesti myös 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 takia, mutta useat kehitysympäristöt tukevat myös GNU/Linuxia kuten esim. Visual Studio Code, Eclipse, NetBeans ja CodeBlocks. Linux-ympäristössä vallitseva trendi on avoimuus ja vapaat lisenssit GNU-työkalujen ja Linuxin tapaan.

Unix, Linux ja ohjelmointiympäristöt

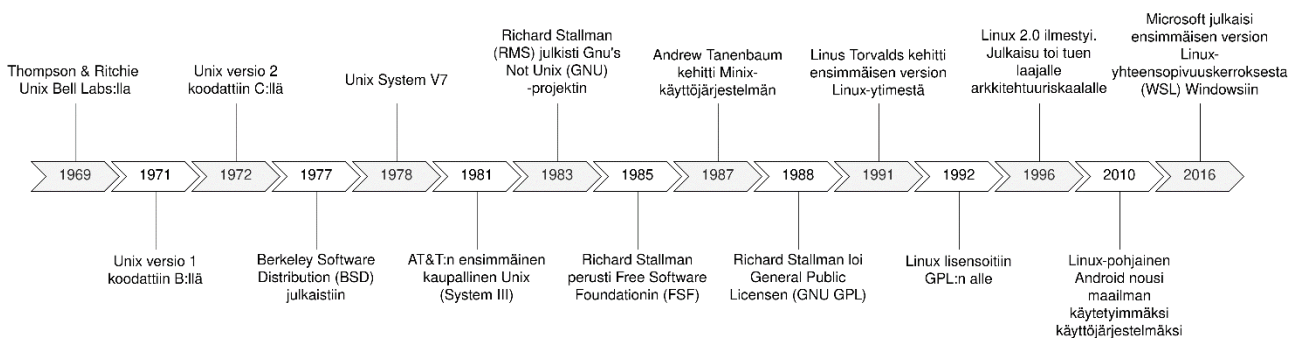
Tietotekniikan aamuhämärässä, kun tietokoneita ei ollut vielä jokaisen ihmisen taskussa vaan lähinnä armeijan ja yliopistojen käytössä, kehitettiin Yhdysvalloissa yleispätevä käyttöjärjestelmä, jonka nimeksi tuli Unix. Unix toteutettiin C-kielillä, joka muodostui tässä samassa yhteydessä aiemmin käytetystä B-ohjelmointikielestä.

Unix oli kaupallinen käyttöjärjestelmä, mutta myöhemmin 1980-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 suljettuja kaupallisia tuotteita, esim. kääntäjiä, pitkin 80-lukua. Projektilta puuttui kuitenkin se kaikkein tärkein eli käyttöjärjestelmän ydin eli kernel.

Myös kernelin rakentaminen aloitettiin Stallmanin ohjauksessa ja se valmistui 1990-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 GNU:n välissä oli tilaa uudelle kernelille. Kernelin tärkeimpiä lähtökohtia oli Portable Operating System Interface (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 saatavana myös kaupallisia versioita (esim. Solaris), mutta ne tuntuvat olevan väistymässä avoimen lähdekoodin Unix-järjestelmien tieltä kuten esimerkiksi 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. Puhemielessä ja tässä oppaassa Linuxilla viitataan koko käyttöjärjestelmään. Kuvassa 1.1 näkyy Unix-GNU/Linux –järjestelmien tärkeimmät virstanpylväät vuoden 1969 ensimmäisestä versiosta päättyen WSL:n julkistamiseen 2016.



Kuva 1.1. Unix GNU/Linux aikajana (Robot Wisdom 2011).

Linux-ympäristössä ohjelmointi on luonnollista C-kielellä, sillä järjestelmän ydin, kernel, on kirjoitettu C-kielellä ja esim. perus C-kääntäjät tulevat asennuksen mukana tai ainakin ne ovat ilmaisia ja helppoja asentaa. Lisäksi kääntäjien käyttö on yleensä varsin ongelmaton ja suoraviivaista. GNU/Linux-jakeluja on useita, joista useimmat perustuvat joko Debianiin, Red Hatiin / Fedoraan, Slackwareen tai SuSeen. Red Hat on vanhimpia Linux-jakeluita ja se tarjoaa kaupallisen käyttöjärjestelmän, johon samanniminen pörssiyhtiö tarjoaa mm. ylläpitopalveluja. Ubuntu on esimerkki avoimeen lähdekoodiin perustuvasta Linux-käyttöjärjestelmästä, joka pohjautuu Debian-jakeluun ja on sitoutunut noudattamaan vapaan ohjelmistokehityksen periaatteita. Ubuntu-käyttöjärjestelmä löytyy esimerkiksi LUTin Linux-luokasta.

Sovelluskehitys Linuxissa hoituu niin komentorivityökaluilla (esim. nano, gcc ja make) tai laajemmilla integroiduilla kehitysympäristöillä (IDE, Integrated Development Environment) kuten Eclipse tai Netbeans. Ubuntuun löytyy monia hyviä peruseditoreja kuten esim. perusasennukseen kuuluvat vi, nano ja gedit sekä erikseen asennettavat emacs ja Visual Studio Code (VSC). Emacs ja VSC ovat kehittyneitä editoreita ja mahdollistavat mm. koodin kääntämisen editorin sisällä.

Tässä oppaassa ohjelmien kääntäminen, suorittaminen ja testaaminen tehdään Linux-ympäristössä avoimilla, ilmaisilla ja hyväksi koetuilla työkaluilla gcc, make ja Valgrind. Kuten edellä totesimme, Unix ja C-kieli sopivat hyvin yhteen ja Linux on tällä hetkellä helposti saatava ja ilmainen Unix-ympäristö, joten Linux-kehitysympäristö on luonnollinen valinta C-ohjelmoijalle. VSC-editorin käyttö Windows'ssa madaltaa aloituskynnystä, koska näin sinun ei tarvitse opetella ensin uutta käyttöjärjestelmää voidaksesi opetella ohjelmoimaan. Koska VSC-editori on saatavilla Windows, Linux ja Mac -käyttöjärjestelmiin, saamme kaikkiin näihin käyttöjärjestelmiin hyvin samanlaisen ohjelmointiympäristön ja tätä opasta voi käyttää käyttöjärjestelmästä riippumatta. Kuten edellä todettiin, ohjelmistokehittäjät käyttävät usein integroituja kehitysympäristöjä eli IDE:tä, mutta

koska tämä opas keskittyy C-ohjelmointiin, pidättäydymme kevyessä ja ohjelmoijien suosimassa VSC-editorissa.

C-kielen taustaa

C-ohjelmointikieli otettiin siis käyttöön vuonna 1972 Yhdysvalloissa Unix-käyttöjärjestelmän ohjelmointikielenä. Kieli suunniteltiin alun perin järjestelmäohjelmointiin, mutta se on löytänyt paikkansa myös sovellusohjelmoinnin parissa. Erityisesti muistin määrän ollessa rajallinen tai kun ohjelman suorituskyvylle asetetaan kovia vaatimuksia, on C-kieli osoittanut olevansa varteenotettava vaihtoehto. C-kieli sopii hyvin laitteistoläheiseen ohjelmointiin ja sille on tyypillistä, että käyttäjä vastaa monista toiminnoista, jotka on automatisoitu uudemmissa kielissä. Näitä toimintoja ovat mm. muistinvaraus, dynaamisten rakenteiden määrittely ja toteutus sekä varatun muistin vapautus. Esimerkiksi Python-kielen laajennusten ydinosat tehdään usein C-kielillä, jonka jälkeen koodaus Pythonilla on luonnollinen valinta. Myös C++, C# ja Java pohjautuvat vahvasti C-kieleen, joka näkyy mm. käskyissä ja monissa ohjelmointikielten periaatteista.

C-kieli rupeaa olemaan vanha kieli, mutta 2000-luvulla se on vuorotellut Javan kanssa suosituimman ohjelmointikielen roolissa (Tiobe 2021). Suomessa C-kieli mainitaan harvoin työpaikkailmoituksissa, mutta erityisesti monet matalan tason ominaisuudet kuten muistinhallinta tekevät siitä kielen, jolla voi tehdä vaikka mitä. Automaation vähäisyyden vuoksi ohjelmoija joutuu usein kirjoittamaan paljon koodia itse, mutta toisaalta se antaa ohjelmoijalle mahdollisuuden hallita tarkasti tehtävää ohjelmaa ja erityisesti muistinkäyttöä. Siksi C-kieli tarjoaa ohjelmoijalle paljon mahdollisuuksia, mutta edellyttää myös tarkkuutta, jotta kontrolli säilyy ohjelmoijalla eikä ohjelmointikieli pääse päättämään ohjelman toiminnasta.

C-ohjelman kääntäminen

Ohjelmointi alkaa ohjelman kirjoittamisella editorissa. Tämän jälkeen tulkattavan ohjelman voi suorittaa heti tulkin sisällä, mutta käännettävät kielet edellyttävät lähdekoodin kääntämistä suoritettavaksi ohjelmaksi, joka voidaan suorittaa itsenäisenä ohjelman eli ilman erillistä suoritussympäristöä kuten tulkkia. Python on tulkattava kieli, kun taas C-kieli on käännettävä kieli. Tässä kappaleessa katsotaan, miten pieni C-ohjelma käännetään suoritettavaksi ohjelmaksi ja suoritetaan Linux-ympäristössä. Itse ohjelman rakenne ja toiminta käydään läpi seuraavissa luvuissa. Käännösprosessin läpikäynnin jälkeen on muutama huomio kääntäjien eroista, sillä käännettyä ohjelmaa ei voi tyypillisesti suorittaa erilaisissa käyttöjärjestelmissä ja samakin lähdekoodi voi toimia eri tavoin käännettynä eri kääntäjillä.

C-ohjelman kääntämisprosessi

Katso oppaan liitteenä olevat asennusohjeet ja asenna C-kehitysympäristö itsellesi. Sen jälkeen kirjoita alla oleva ohjelma VSC:llä ja tallenna se nimellä E1_1.c.

Esimerkki 1.1. Käännettävä C-ohjelma

```
#include <stdio.h>

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

Tämän jälkeen käännä ohjelma VSC:n terminaalissa seuraavalla käskyllä (avaa terminaali painamalla CTRL-Ö):

```
gcc E1_1.c -o E1_1
```

Mikäli käännös onnistuu ilman virheitä, tallentuu hakemistoon suoritettava ohjelmatiedosto `E1_1` ja voit suorittaa sen komennolla

```
./E1_1
```

Ohjelman suorittamisessa kannattaa huomata muutama yksityiskohta. Linux-ympäristössä isot ja pienet kirjaimet ovat merkitseviä eli `E1_1` ja `e1_1` ovat eri asioita ihan niin kuin C-ohjelmissakin. Toinen asia on, että ohjelmaa suoritettaessa Linux edellyttää polkua ajettavan ohjelman nimen lisäksi ja kirjoittamalla suoritettavan ohjelman `"E1_1"` nimen eteen `./` kerromme polun eli että tämä tiedosto löytyy tästä samasta hakemistosta. Tämä tuntuu aluksi oudolta, mutta sille on hyvät tietoturvaan liittyvät perusteet, jotka käydään läpi tarkemmin Linux-peruskurssilla. Tässä vaiheessa on vain muistettava, että ohjelma suoritetaan polun kanssa eli `./E_1`.

Edellä olevan minimaalisen käännös-komennon sijaan tulemme jatkossa määrittelemään tarkemmin tehtävän käännöksen eli ohjaaman käännöstä seuraavilla valinnoilla:

```
gcc E1_1.c -o E1_1 -std=c99 -Wall -pedantic
```

Molemmissa tapauksissa käännös tehdään `gcc`-kääntäjällä, jossa syötteenä on `E1_1.c` niminen lähdekooditiedosto ja parametri `-o` (output) kertoo tehtävän tiedoston nimen `E1_1`. Parametri `-std=c99` kertoo kääntäjälle, että lähdekoodi tulee tarkastaa käännöksen yhteydessä ja verrata sitä ANSI C99 standardin mukaisiin vaatimuksiin. Vastaavasti parametri `-Wall` ohjeistaa kääntäjää kertomaan kaikista varoituksista käännöksen yhteydessä (warning all) ja `-pedantic` kertoo kääntäjälle, että sen tulisi olla pikkutarkka käännöksessä eli raportoida kaikki potentiaalisetkin virheet käyttäjälle. Yhteenvetona edellisen ohjelman kääntäminen ja suorittaminen Linux-komentokehoteessa (VSC:n terminaalissa) tapahtuu seuraavilla käskyillä:

```
un@LUT8859:~/Opas$ gcc E1_1.c -o E1_1 -Wall -std=c99 -pedantic
un@LUT8859:~/Opas$ ./E1_1
Minä tein tämän!
un@LUT8859:~/Opas$
```

Huomioita kääntäjistä ja käyttöjärjestelmistä

Kääntämiseen liittyen on hyvä huomata, 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.). Lähtökohtaisesti Linux-ympäristössä käännetty ohjelma eli binäärikoodi ei toimi Windows-ympäristössä ja päinvastoin.

Lähdekoodi ei välttämättä toimi samalla tavoin käännettynä eri kääntäjillä. Eri kääntäjät on kehitetty eri tiimien toimesta ja niillä on voinut olla erilaisia tavoitteita, joten ne saattavat tehdä hieman erilaisia ratkaisuja käännöksen yhteydessä tehdessään suoritettavaa ohjelmaa. Tämän lisäksi eri käyttöjärjestelmillä saattaa olla erilainen politiikka esimerkiksi muistinhallinnan ja muiden resurssien käytön suhteen. Lähtökohtaisesti Windows-puolen PC-kääntäjät on usein suunniteltu henkilökohtaiseen käyttöön, kun taas Unix/Linux-kääntäjät on tehty palvelinkäyttöön. Näin ollen Windows-kääntäjät saattavat käyttää aikaa esim. muuttujien alustamiseen nollassa käytön helpottamisen nimissä ja Linux-kääntäjät taas jättävät ne alustamatta nopeuden nimissä.

Usein tällaiset ongelmat tulevat esille joskus myöhemmin eivätkä vaikuta perusrakenteilla ohjelmointiin. Siitäkin huolimatta kannattaa muistaa, että C-kielistä ohjelmaa ei välttämättä pysty kääntämään sellaisenaan muussa käyttöjärjestelmässä kuin missä se on tehty ja testattu. 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 minimi- ja maksimiarvot sekä varoitus- ja huomautuspolitiikka. Myös kääntäjän optioilla voidaan vaikuttaa käännöksen kulkuun ja käännettyjen ohjelmien toimintaan. Kääntämällä ohjelmat C-kielen standardin c99:n mukaisesti tekee niistä helpommin siirrettäviä eri ympäristöjen välillä, mutta edelleen moni yksityiskohta jää ohjelmoijan päätettäväksi ja siten ohjelmat ovat erilaisia. Tämä on yksi asia, jossa ohjelmoijat poikkeavat toisistaan eli joku saa ohjelman menemään yhdestä kääntäjästä läpi, kun taas toinen kirjoittaa ohjelmia, jotka menevät eri kääntäjistä läpi ja toimivat samalla tavoin eri ympäristöissä.

C-ohjelman perusasioita

Ohjelmointikielenä C-kieli noudattaa normaaleja ohjelmointikielten periaatteita. Käydään seuraavaksi läpi C-kielen perusasiat ja keskitytään eroihin Python-ohjelmiin verrattuna. Periaatteet ovat pitkälti vastaavia ja tuttuja muista ohjelmointikielistä, mutta varmuuden vuoksi kannattaa katsoa kaikki esimerkit läpi, ettei joku yksittäinen ero muodostu ongelmaksi.

Kääntämisen ja tulkkauksen erot käsiteltiin jo edellä, joten katsotaan seuraavaksi syntaksiin, tekstin tulostamiseen, muuttujan määrittelyyn, tietojen lukemiseen, merkkijonoihin, rivin lukemiseen ja muotoiltuun I/O:hon (Input/Output, syöttö ja tulostus) liittyvät asiat. Tärkeimmät erot Pythoniin ovat C-kielen edellyttämä muuttujien määrittely ennen käyttöä ja joustamattomuus tietotyyppien kanssa. Myös tietojen lukeminen käyttäjältä eroaa Pythonista ja merkkijonojen käsittely tuntuu perustellusti monesta työläältä. Kaikki nämä asiat pystyy kuitenkin tekemään, kunhan noudattaa kielen periaatteita ja syntaksia.

Syntaksi

C-kielillä kirjoitetut ohjelmat poikkeavat Python-ohjelmista monilla tavoin alkaen niiden syntaksista. Katsotaan kääntäjän testaamisessa käytettyä esimerkkiä 1.1 tarkemmin.

```
#include <stdio.h>

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

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 – palaamme tähän tarkemmin myöhemmin, mutta jo nyt on muistettava, 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 sisällytimme C-kielen mukana tulevan `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 ”}”. Aukeava aaltosulku on aina osion ensimmäinen merkki ja sulkeva aaltosulku osion viimeinen merkki. Sisennyksillä ei ole C-kielessä toiminnallista roolia, mutta ne helpottavat merkittävästi lähdekoodin ymmärtämistä ja siksi on hyvä jatkaa Pythonista tuttua sisennysten käyttöä ja yllä näkyvää esimerkin 1.1 tyyliä. Lisäksi jokainen toiminnallinen rivi päättyy puolipisteeseen ”;”.

Tällaiset ohjelman kirjoittamissäännöt ovat syntaksia ja yksi yleisimpiä virheitä C-ohjelmoinnin alkuvaiheessa on syntaksivirhe. Vaikka syntaksivirhe kuulostaa pahalta, on kyseessä itse asiassa selkeä – ja helppo – virhetyyppi. Kyseessä on siis kirjoitusvirhe eli ohjelma ei ole kirjoitettu kielen kieliopin mukaisesti. Kaikkiin kieliin liittyy kielioppi eli esimerkiksi ”suomen kielen lauseessa on oltava aina verbi”. Positiivisesti ajatellen kääntäjä huomauttaa syntaksivirheistä ja siten niiden korjaaminenkin helppoa. Tai olisi, jos virheilmoitus osuisi oikealle riville, mutta joka tapauksessa syntaksin oppii, kun kirjoittaa ohjelmia eli harjoittelee ja opettelee tulkitsemaan kääntäjän antamia virheilmoituksia.

Tiivistetysti C-ohjelmat poikkeavat Python-ohjelmista etenkin seuraavien asioiden osalta:

- 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 aliohjelmia.
- Funktiot loppuvat aina `return`-käskyyn, joka palauttaa funktion tyypin mukaisen arvon. Pääohjelman tyyppi on normaalisti `int` ja siten se 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 koodin luettavuuteen. Näin ollen systemaattiset sisennykset on hyvän ohjelmointityylin perusta. Aloittava { kannattaa laittaa rivin loppuun viimeiseksi merkiksi ja lopettava } tyhjälle riville yksinään.
- 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.

Tutustutaan seuraavaksi tarkemmin C-kielen peruskäskyihin esimerkkiohjelmien avulla.

Tekstin tulostaminen näytölle

Ohjelman perustoimintoihin kuuluu tekstin tulostaminen näytölle. Huomaa, että tulostuskäskyt edellyttävät kirjaston käyttämistä ja tässä esimerkissä on myös kommentteja.

Esimerkki 1.2. Tekstin tulostus näytölle

```
#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);
}
```

Ohjelman tuottama tulos

Tämä ohjelma tuottaa seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E1_2
Minä tein tämän!
Huomaa, että teksti jatkuu aina edellisen perään,
    ellei tulostus pääty rivinvaihtomerkkiin!un@LUT8859:~/Opas$
un@LUT8859:~/Opas$
```

Kuinka ohjelma toimii

Ohjelma alkaa kirjaston otsikkotiedoston sisällyttämisellä. Tiedosto `stdio.h` sisältää tavallisimmat tietojen lukemiseen ja tulostamiseen käytetyt funktiot eli **standard input/output**, joten tiedoston sisällyttäminen ohjelmaan mahdollistaa ko. kirjaston määrittelyiden käytön ohjelmassa. C-kielessä kirjastoja tarvitaan jo pelkkiä ”perustoimintoja” varten ja palaamme kirjastojen käyttöön laajemmin luvussa 3. Toistaiseksi riittää ymmärtää, 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”. Käytännössä kerromme, että `main`-funktio ei ota vastaan 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. Kommentit ovat C-kielessä oletuksena monirivisiä, joten kääntäjä odottaa aina löytävänsä merkkiparille `"/*` parin `*/`. Avoimeksi jätetyn kommenttirivin jälkeisen koodin kääntäjä katsoo edelleen kommentteiksi 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. Kuten koodista näkyy, kaikki lauseet, nyt kolme `printf`-lausetta ja `return`, päättyvät puolipisteeseen.

Ohjelman lopussa on toinen kommenttiteksti sekä funktion lopetuksen osoittava `return`-käsky. Tämä käsky määrää käytännössä, että `main`-funktio lopettaa toimintansa lopetusarvolla 0 ja tämä lopetusarvo palautetaan ohjelman käynnistäneelle ohjelmalle/käyttöjärjestelmälle. Paluarvo 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 aaltosululla.

Muuttujan määrittely

Python-kielessä muuttujilla oli käytännössä kolme erilaista tietotyyppiä eli numeroarvo (`int/float`), merkkijono (`str`) taikka rakenne kuten lista tai tuple. Muuttujan tyyppiä pystyttiin myös vaihtamaan lennosta esimerkiksi sijoittamalla merkkijono numeroarvoja sisältäneeseen muuttujaan tai toisinpäin ja kaikki toimi hienosti eli automaattisesti. C-kielessä muuttujien käyttö poikkeaa Pythonista ja tärkeimmät erot ovat seuraavat:

1. Muuttujat pitää määritellä aina ennen käyttöä. Määrittely sisältää kaksi asiaa: muuttujan nimen ja sen tietotyyppi.
2. Muuttujan tietotyyppiä ei voi vaihtaa ohjelman suorituksen aikana.

Muuttujan arvo voidaan sijoittaa toista tietotyyppiä olevaan muuttujaan, mutta usein tietotyyppin vaihto pitää koodata näkyviin ohjelmaan eikä automaattista tietotyyppin vaihtoa juurikaan tapahdu. Esimerkiksi kokonaisluvusta liukulukuun siirtyminen edellyttää uutta liukuluku-muuttujaa ja merkkijonoksi vaihtaminen on vielä tätä isompi projekti, johon palataan myöhemmin.

Esimerkki 1.3. Muuttujan määrittely

```
#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\n", luku, liukuluku);

    return(0);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E1_3
Luku-muuttuja on 5.
5 8.234235
un@LUT8859:~/Opas$
```

Kuinka ohjelma toimii

Ohjelma alkaa kuten aiemmatkin esimerkit `main`-funktiolla, jolle avaamme koodiosion aaltosululla. Tämän jälkeen meillä on kommenttirivi, jota ei lasketa varsinaisesti koodiriviksi, ja tämän jälkeen törmäämme muuttujien määrittelyyn.

Ensimmäinen `main`-funktiossa tehtävä asia on muuttujien määrittely. Jokainen ohjelmassa 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 määritelleet muuttujat, voimme käyttää niitä koodissa aivan kuten Python-muuttujiakin. Voimme sijoittaa muuttujiin arvoja, tulostaa niitä sekä laskea eri muuttujia yhteen tai vähentää mieleemme mukaisesti. Huomaa kuitenkin, että C-kieli antaa tallentaa liukuluvun kokonaisluku-muuttujaan, mutta katkaisee automaattisesti 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. Tässä on hyvä huomata, että ohjelma toimii etukäteen määritellyllä tavalla, mutta lopputulos ei välttämättä ole ohjelmoijan odottama, jos ei ole tarkkana.

Tietojen lukeminen käyttäjältä

C-ohjelmissa tietojen kysyminen käyttäjältä poikkeaa Pythonista. Kokonaislukujen osalta kysyminen on varsin selkeää, vaikka sekä logiikka että syntaksi ovatkin erilaisia. Tässä osiossa esiteltävä `scanf` sopii hyvin kokonaislukujen, liukulukujen ja yksittäisten merkkien lukemiseen käyttäjältä eli näppäimistöltä.

Esimerkki 1.4. Syötteiden lukeminen

```
#include <stdio.h>

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

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E1_4
Anna kokonaisluku: 34
Annoit luvun 34.
```

Kuinka ohjelma toimii

Ohjelma alkaa pääohjelmalla eli `main`-funktion koodilohkolla, jossa ensimmäiseksi määrittelemme kokonaislukumuuttujan `luku`. 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.

`scanf`-käsky alkaa kertomalla, minkälaista tietoa se ottaa vastaan. Nyt määrittelemme, että odotamme käyttäjältä yhtä kokonaislukua ja se tehdään samalla tavoin kuin `printf`-lauseissa eli antamalla formaatti `"%d"`. Huomaa, että tämä formatoitu-osuus näkyy näiden funktioiden nimissä – `printf` eli print formatted ja `scanf` eli scan formatted – ja formatointi noudattaa molemmissa tapauksissa samoja sääntöjä. 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 ja tässä vaiheessa riittää muistaa, että luettaessa kokonais- ja liukulukuja `scanf`-käskyllä tulee muuttujan eteen laittaa `&`-merkki.

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

Tässä esimerkissä näimme, kuinka kokonais- ja liukulukujen sekä yksittäisten merkkien pyytäminen ja vastaanottaminen toimii. Formaatin osalta kokonaisluvut luetaan muotoilumerkillä `“%d”`, liukuluvut `“%f”` ja yksittäiset merkit `“%c”`. Yksittäinen merkki tarkoittaa yhtä kirjainta, jota voi käyttää esim. kysyttäessä käyttäjältä ”haluatko jatkaa (k/e)?”. Esimerkki saattoi herättää paljon kysymyksiä ja palaamme näihin asioihin vielä monta kertaa tässä oppaassa. Mutta katsotaan seuraavaksi, miten merkkijonoja luetaan käyttäjältä.

Merkkijonot, merkkitaulukon koko ja merkkijonon loppumerkki

Edellisessä osiossa keskityimme numeerisiin muuttujiin ja totesimme, että yksittäiset merkit käsitellään samalla tavalla. Mutta kuinka merkkijonoja käsitellään C-kielessä? Tämä onkin yksi merkittävä ero C-kielen ja Pythonin välillä. Koska merkkijonon tarvitsema muistin määrä riippuu merkkijonon pituudesta ja C-kielessä ohjelmoija päättää muistin varaamisesta, ei merkkijonojen käsittely ole C-kielessä yhtä helppoa kuin Pythonissa.

C-kielessä merkkijonon, esim. etunimi ”Ville”, käsittely on laajennus yksittäisen merkin tapauksesta eli yhden merkin sijaan varataankin merkkijonolle tilaa monta peräkkäistä muistipaikkaa peräkkäisiä merkkejä varten. Yhden merkin kohdalla asia on helppo, sillä tiedämme, että meillä on yksi merkki käytössä. Usean merkin kohdalla asia on kuitenkin toinen ja herää kysymyksiä kuten ”kuinka monta merkkiä” ja ”mistä tiedän kuinka monta merkkiä”? Pythonissa nämä tekniset yksityiskohdat on piilotettu ja merkkijonon käsittely on tehty yksinkertaiseksi ohjelmoijalle. C-kielessä taas ohjelmoija joutuu miettimään nämä asiat merkki kerrallaan ja lähtökohta on, että merkkijonolle on varattava muistia riittävä määrä sekä toisaalta merkkijonon loppumisen tietää loppu-merkistä.

Merkkijonolle pitää varata muistia ”riittävä määrä”, mutta mikä on riittävä, on tietysti toinen asia. Käytännössä esimerkiksi nimien kohdalla Väestörekisterin mukaan suomalaiset etunimet ovat tyypillisesti 1-16 merkkiä pitkiä, sukunimet 2-25 merkkiä ja monissa muissa maissa nimet ovat merkittävästi pidempiä. Näin ollen suomalaisia nimiä käsiteltäessä 30 merkkiä on tyypillisesti riittävä maksimimäärä tämän oppaan kohdalla eli voimme käyttää sitä tällä kertaa varattavana muistimääränä. Ja laittamalla nimen perään aina loppu-merkin voimme käsitellä merkkijonoja oikean kokoisina eli lisätä suoraan nimen perään sopivan seuraavan merkin jne.

Käydään seuraavaksi läpi merkkijonojen käsittelyn tekniset yksityiskohdat ja aloitetaan katsomalla esimerkki. Tässä esimerkissä määritellään merkkijono-muuttujia ja katsotaan niiden peruskäyttöä. Esimerkin pääkohdat selitetään alla, sillä tässä on useita merkkijonojen määrittelyyn ja käyttöön liittyviä asioita, jotka pitää osata jatkossa.

Esimerkki 1.5. Merkkijonon käyttö

```
#include <stdio.h>

int main(void) {
    /* Määritellään muutama merkkijono ja apumuuttujia. */
    char nimi[30];
    char ammatti[] = "Palomies";
    char harrastus[30] = "autot";
    int koko;

    /* Nyt meillä on merkkijonoja, käytetään niitä vähän. */
    printf("Anna nimi (max. 29 merkkiä): ");
    scanf("%s", nimi);

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

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E1_5
Anna nimi (max. 29 merkkiä): Erkki
Erkki on Palomies.
Harrastuksenaan hänellä on autot.
Sanalle 'autot' on varattu 30 merkkiä.
un@LUT8859:~/Opas$
```

Kuinka ohjelma toimii

Ohjelma alkaa tavalliseen tapaan `stdio.h`-kirjaston käyttöönotolla ja `main`-funktiolla. 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 ilmoitamme kääntäjälle, kuinka monta peräkkäistä merkkiä meillä on, jotta kääntäjä osaa varata tilan etukäteen. Peräkkäiset merkit muodostavat käytännössä taulukon, jonka voi varata esim. seuraavalla tavalla:

```
char nimi[30];
```

Tällä käskyllä määrittelemme merkkitaulukon `nimi`, johon on varattu tilaa 30 merkkiä (taulukon paikat 0-29). Huomaa, että merkkitaulukon ensimmäinen paikka on numeroltaan 0. Tällöin 8 merkkiä pitkän merkkijonon viimeinen merkki tallennetaan paikkaan 7.

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

Nyt määrittelemme merkkitaulukon `ammatti`, johon on varattu tilaa juuri sen verran kuin merkkijono "Palomies" tarvitsee. Kääntäjä laskee merkkijonon pituuden, lisää siihen loppumerkin tarvitseman tilan ja varaa tarvittavan taulukon.

```
char harrastus[30] = "autot";
```

Viimeisenä olemme luoneet merkkitaulukon `harrastus`, jossa on tilaa 30 merkille ja josta olemme ottaneet käyttöön merkkijonon ”autot” tarvitseman tilan. Lopuksi luomme vielä kokonaislukumuuttujan `koko`, johon selvitämme yhden merkkitaulukon varaaman tilan määrän.

Seuraavilla kahdella rivillä suoritamme merkkijonon lukemisen käyttäjältä. Ensimmäinen `printf`-lause ei poikkea aiemmasta käytöstämme ja nyt voimme myös lukea käyttäjän antaman merkkijonon samalla tavalla kuin luimme lukuja eli `scanf`-käskyllä käyttäen muotona ”%s” eli merkkijonoa. Huomaa, että nyt `scanf`-käskylle annetaan pelkästään merkkitaulukon nimi ilman perässä olevia hakasulkeita, jolloin saamme merkkitaulun osoitteen ja se vastaa lukujen vaatimaa &-merkkiä. `scanf`-käsky lukee merkkijonon välilyöntiin tai rivinvaihtomerkkiin asti, joten katsotaan useiden sanojen lukuun sopiva tapa seuraavassa kohdassa.

`sizeof`-operaattoria käytetään ensimmäisen kerran seuraavalla rivillä. `sizeof` palauttaa numeroarvona tiedon siitä, kuinka monta tavua muistia parametrina annettu rakenne varaa. Tässä tapauksessa mittautimme merkkitaulukon `harrastus`. Kuten tulostuksesta näkyy, merkkitaulukon `harrastus` koko on 30 tavua huolimatta siitä, että ainoastaan 5 ensimmäistä alkionota on käytössä. `sizeof`-operaattoria käytetään C-kielessä paljon, koska se mahdollistaa parametrin varaaman tilan selvittämisen ja tämä on lähtökohtaisesti tärkeä asia C-ohjelmissa.

Kuten edellä totesimme, merkkijonolla pitää olla loppumerkki. Taulukko 1 alla näyttää loppumerkin ”autot” merkkijonon perässä indeksillä 5 terminaattorimerkkinä `\0` (*null character, null terminator, NULL*). Tämä merkki kertoo ohjelmalle, missä kohdassa varsinainen tietosisältö loppuu ja tyhjä tila alkaa. Koska varaamme merkkitaulukon aiemmin käytössä olleelta muistialueelta, voi taulukon muistialueella olla tietoja samaa muistipaikkaa aiemmin käyttäneiltä ohjelmilta. Näillä ei kuitenkaan ole merkitystä hyvin tehdyssä ohjelmassa, sillä tallennamme muistiin omat tiedot ja merkkijonon päätteeksi laitetaan loppumerkki, jolloin muistialueella mahdollisesti olevat vanhat tiedot eivät häiritse meitä. Käytettäessä merkkijonojen käsittelyyn tehtyjä funktioita, huolehtivat nämä terminaattorimerkin systemaattisesta käytöstä ja merkin muistaminen on tarpeellista vain silloin, kun merkkijonoja käydään läpi tai käsitellään merkki kerrallaan.

Taulukko 1. Merkkitaulun ’harrastus’ sisältö.

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

Seuraavassa esimerkissä näkyy merkkijonon käsittely ja loppumerkki. Tässä esimerkissä käytetään `strcpy`-funktioita, joka kopioi merkkijonon (string copy) merkkitaulukkoon, mutta palaamme näihin merkkijonofunktioihin tarkemmin myöhemmin.

Esimerkki 1.6. Merkkijonon loppumerkki

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

int main(void) {
    char merkkijono[20] = "Kirsi Virtanen";
    printf("%s\n", merkkijono);

    strcpy(merkkijono, "Matti");
    printf("%s\n", merkkijono);

    merkkijono[5] = ' '; // Poistetaan Matti-sanasta perästä loppumerkki
    printf("%s\n", merkkijono);

    return(0);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Esimerkit$ ./E1_6
'Kirsi Virtanen'
'Matti'
'Matti Virtanen'
```

Kuinka ohjelma toimii

Ohjelma tulostaa nyt ensin annetun merkkijonon, Kirsi Virtanen, kuten aiemmissakin esimerkeissä. Seuraavaksi strcpy-funktio kopioi tuon merkkijonon päälle sanan ”Matti” ja lisää sen perään loppumerkin, joten seuraava tulostus tuottaa tulokseksi ”Matti” niin kuin pitääkin. Kolmantena vaiheena oleva merkkijonon yhden merkin korvaus toisella on normaalia C-ohjelmointia, vaikka äkkiseltään se tuntuukin oudolta. Tässä vaihdetaan Matti-sanasta perässä olevan loppumerkin tilalle välilyöntimerkki ja tämän seurauksena seuraavassa tulostuksessa näytölle tulee ”Matti Virtanen”. Tämä johtuu siitä, että kun kopioimme Matti-sanasta merkkijonoon, muutimme sen alkuosaa ja lisäsimme sen perään loppumerkin eli käytimme uudelleen varatun muistialueen alkuosaa muuttamatta loppuosaa mitenkään. Ja kun nyt viimeisessä vaiheessa poistimme Matti-sanasta perässä olleen loppumerkin, tulosti printf-käske merkkejä loppumerkkiin asti, joka löytyi nyt Virtanen-sanasta perästä.

Tyypillisesti merkkijonon perästä puuttuva loppu-merkki johtaa siihen, että näytölle tulee muutama epämääräinen merkki – esim. naurava naama – ennen kuin tulostus loppuu yllä olevan esimerkin mukaisesti. Huomaa, että loppu-merkin puute voi näkyä tai olla näkymättä riippuen siitä, minkälaisessa käytössä muistialue on ollut aiemmin. Ainoa tapa suojautua tältä ongelmalta on huolehtia siitä, että loppu-merkki (NULL, '\0') on aina sijoitettu oikealle paikalleen merkkijonon viimeiseksi merkiksi.

Rivin lukeminen

Tietojen lukeminen ja tulostaminen onnistuu scanf- ja printf-käskyillä edellä olevien esimerkkien mukaisesti. Käytännössä tietojen tulostaminen printf-käskyllä toimii hyvin ja sillä saa hoidettua useimmat tulostustarpeet, vaikka puts/fputs mahdollistavatkin pelkän

merkkijonon tulostamisen ilman formatointia näytölle tai tiedostoon, esim. `'fputs ("Moi\n", stdout);'`. Tiedon lukemisen yhteydessä tulee helpommin vastaan tilanteita, joissa `scanf:n` käyttö edellyttää tarkempaa muotoilua tai vaihtoehtoisia käskyjä. Esimerkiksi syötteen pituuden voi rajoittaa 10 merkkiin muotoilulla `'scanf ("%10s", str);'` ja vastaavasti hyväksyttävät merkit voi määritellä Unixin regular expression'lla, joka jää Unix-asian tämän kurssin ulkopuolelle. Lähtökohtaisesti on hyvä muistaa, että `scanf` ja `printf` perustuvat puskurointiin eli merkit kulkevat puskurin kautta ja esim. `scanf` lukee merkkejä puskurista tiettyjen sääntöjen mukaan. Koska luku-operaatiolle annetaan haluttu formaatti, noudattaa `scanf` formaattia ja huomioi siinä yhteydessä omien sääntöjensä mukaisesti myös ”valkoiset merkit” eli white space -merkit, joita ovat välilyönti, sarkain ja rivinvaihtomerkki. Kaiken tämän seurauksena yhden merkin lukeminen johtaa tyypillisesti siihen, että luettua merkkiä seuraava rivinvaihtomerkki jää puskurin ensimmäiseksi merkiksi, jolloin seuraava lukuoperaatio saa puskurista pelkän rivinvaihtomerkin ja lopettaa lukemisen siihen. Poistamalla tuo rivinvaihtomerkki puskurista esim. `getchar()` -käskyllä onnistuu seuraava luku normaalisti.

Vaihtoehto muotoillulle lukemiselle on lukea yksi rivi kerrallaan eli kaikki merkit rivinvaihtomerkkiin asti. Tämä voidaan toteuttaa `fgets`-käskyllä, jossa luettavien merkkien rajoittaminen estää ohjelman rikkoutumisen liian pitkän syötteen takia. Mikäli luettavia merkkejä on enemmän kuin voidaan lukea yhdellä kertaa, luetaan maksimäärä-1 merkkiä ja lisätään luettujen merkkien perään NULL-merkki. Näin ollen seuraava lukuoperaatio palauttaa puskuriin jääneet merkit. Lukemalla koko rivi `fgets`:llä ohjelmoija voi käsitellä merkkijonon itse haluamallaan tavalla ja tämä mahdollistaa myös välilyöntejä sisältävien syötteiden lukemisen. Huomaa, että `fgets'n` palauttamassa merkkijonossa on rivinvaihtomerkki, joten ohjelmoijan on itse päätettävä mitä sen kanssa tehdään.

Tiivistetysti `fgets`:llä luettaessa merkkitaulukossa tulee olla halutun merkkimäärän lisäksi tilaa 2 merkille, rivinvaihto- ja NULL-merkeille, jotka eivät näy normaalille käyttäjälle. Itse `fgets`-käskyssä luettava merkkimäärä on sama kuin varattu merkkitaulukon koko, jolloin `fgets` saa luettua näkymättömän rivinvaihtomerkin sekä lisättyä NULL-merkin loppuun.

Esimerkki 1.7. Rivin lukeminen fgets:llä

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

int main(void) {
    char c;
    char nimi[30];

    printf("Anna merkki: ");
    scanf("%c", &c);
    getchar();
    printf("Anna etunimesi (max 28 merkkiä): "); // nimi + \n + NULL
    fgets(nimi, 30, stdin); // luetaan nimi + \n ja lisätään NULL
    /* scanf("%s", nimi); */
    nimi[strlen(nimi)-1] = '\0';
    printf("Nimesi on '%s' ja merkki oli %c.\n", nimi, c);

    return(0);
}
```

Esimerkin tuottama tulos

```
un@LUT8859:~/Opas$ ./E1_7
Anna merkki: k
Anna etunimesi (max 28 merkkiä): Ville Kalle
Nimesi on 'Ville Kalle' ja merkki oli k.
```

Kuinka ohjelma toimii

Esimerkissä 1.7 näkyy tyypillinen ongelma-kohta eli ensin luetaan yksi merkki, jolloin rivinvaihtomerkki jää puskuriin. Se saadaan puskurista pois `getchar`-käskyllä. Tämän jälkeen `fgets`:llä luetaan koko rivi rivinvaihtomerkki mukaan lukien, jonka jälkeen rivinvaihtomerkki poistetaan merkkijonosta sijoittamalla sen kohdalle merkkijonoon `NULL`-merkki. Merkkijonon pituus saadaan selville `strlen`-funktiolla, joka löytyy `string.h`-kirjastosta. Tämän jälkeen ohjelma toimii esimerkkiajon mukaisesti. Voit kokeilla ohjelmaa kirjoittamalla sen itse ja kannattaa kokeilla `getchar`-käskyn kommentoimista pois, `fgets`-käskyn korvaamista `scanf`-käskyllä ja rivinvaihtomerkin ylikirjoittamisen jättämistä pois. Ohjelman tuottama tulos näyttää varsin erilaiselta näillä pienillä muutoksilla.

`fgets`-funktion ensimmäinen parametri on muuttuja, johon tietoa luetaan, tämän jälkeen on merkkimäärä, jonka korkeintaan lueimme ja viimeisenä kanava, josta tietoa luetaan. Jos kanava on `stdin`, luetaan merkit käyttäjän antamasta syötteestä. Staattista taulukkoa käytettäessä ohjelma lukee vain annetun määrän merkkejä, joten syötettä pyytäessä on hyvä kertoa syötteen maksimikoko. Luettavan merkkijonon pituudessa on huomioitava rivinvaihto- ja `NULL`-merkit; `fgets`-lisää `NULL`-merkin, joten merkkejä luetaan taulukon koko – 1, joista viimeinen on käyttäjän antama rivinvaihtomerkki ja näin ollen käyttäjän antamani nimi on korkeintaan 28 merkkiä pitkä, kun taulukossa on tilaa 30 merkille.

Puskurointi ei vaikuta `printf`-käskyn toimintaan normaalisti muuten kuin hidastamalla tietojen siirtymistä näytölle. Toinen asia on, jos ohjelma kaatuu odottamatta, sillä tällöin puskurissa olevat tiedot jäävät tulostumatta näytölle ja kaatumisen aiheuttava ohjelman kohta ei välttämättä vastaa näytölle tulleita tulosteita. Puskurointiin kuuluu luonnollisesti mahdollisuus puskurin tyhjentämiseen, joten asiasta kiinnostuneet voivat jatkaa asian tarkempaa selvittelyä tästä eteenpäin omatoimisesti.

Muotoilumerkit tiedon tulostuksessa ja lukemisessa

C-kielessä tulostus tehdään tyypillisesti `printf`-funktiolla, joka tulostaa saamansa parametrit halutussa muodossa näytölle. `printf`-funktion ensimmäisenä parametrina on lainausmerkkien sisällä oleva tulostettava merkkijono muotoilumerkeillä ja sen jälkeen tulostettavat arvot.

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

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

Jokaisen tulostettavan arvon muotoilu määritellään muotoilumerkkijonolla:

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

Muotoilumerkkijono alkaa aina %-merkillä, jota seuraa mahdolliset liput. 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. Esimerkiksi

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

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

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). Muotoilumerkkien liput ja yleisimmät muunnosmerkit on esitetty kootusti seuraavan alakohdan lopussa, ks. Taulukko 9 ja Taulukko 10.

scanf-funktiolla voidaan lukea käyttäjän syöttämää tietoa näppäimistöltä. Käsky noudattaa samaa muotoilu-idea 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 seuraavan esimerkin 1.8 mukaisesti.

Esimerkki 1.8. Muotoiltu tulostus ja lukeminen

```
#include <stdio.h>

int main(void) {
    int a;
    float b;

    printf("Anna kokonaisluku ja liukuluku välilyönnillä erotettuina: ");
    scanf("%d %f", &a, &b);
    printf("Annoit luvut %d ja %5.2f.\n", a, b);
    return(0);
}
```

Esimerkin tuottama tulos

```
un@LUT8859:~/Opas$ ./E1_8
Anna kokonaisluku ja liukuluku välilyönnillä erotettuina: 2 3.456
Annoit luvut 2 ja 3.46.
```

Huomaa tulosteessa, että ja-sanalla ja numeron 3 välissä on 2 välilyöntiä, jotta luku 3.46 käyttää yhteensä määritellyt 5 merkkiä. Yleisimmät muotoiluissa käytettävät liput ja muunnosmerkit on esitetty taulukoina seuraavan alakohdan lopussa (Taulukko 9 ja Taulukko 10).

Sääntöjä ja taulukoita

Edelle kävimme läpi yksinkertainen C-ohjelman syöttö- ja tulostuslauseet. C-kieli on kuitenkin joustava ja erityisesti muistinkäyttöä voidaan optimoida tarpeen mukaan. Näin ollen esimerkiksi ”kokonaisluku” voi tarkoittaa lyhyttä tai pitkää lukua, tyypillisesti 2 ja 4 tavua, tai etumerkillistä tai

etumerkitöntä lukua, ts. esimerkiksi arvoja -32768...32767 tai 0...65535. Tässä ei tietenkään ole kaikki mahdolliset vaihtoehdot vaan C-kielen standardi määrittelee kaikille yhteiset miniarvot yms. ja ne määritellään jokaisessa järjestelmässä tapauskohtaisesti. Tutustutaan seuraavaksi näihin raja-arvoihin ja niiden määrittelyyn käytettäviin taulukoihin. Näistä on hyvä olla tietoinen, jotta näihin voi palata, kun kysymyksiä tulee – ja jos ohjelmoi, niin kysymyksiä tulee vastaan aina ajoittain.

Kokonaisluvut ja liukuluvut

Huomasit varmaan edellisessä esimerkissä, että kokonaislukua `int` ja liukulukua `float` käsiteltiin eritavoin ja niiden tulostuskäskyn sijoitusmerkit (`%d` ja `%f`) olivat erilaisia. Tämä liittyy siihen, että C-kielen muistinhallintaa on automatisoitu vain hyvin vähän. Koska erityyppiset numeromuuttujat tarvitsevat 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 Python huolehti erilaisten numeroarvojen välisestä yhteensopivuudesta, ei käytännössä tarvittu kuin kaksi numeromuuttujaa, kokonaisluku ja liukuluku. Automatisoidun muistinkäytön vuoksi numeerisilla muuttujilla ei myöskään ole varsinasta ylä- eikä alarajaa. Tämä lähestymistapa tekee ohjelmoinnista helpompaa, mutta samalla ohjelmoija menettää muistinkäytön kontrollointimahdollisuudet. C-kielessä vastuu muistin käytöstä on ohjelmoijalla, jonka seurauksena C-ohjelmien muuttujat voivat joutua ns. ylivuoto-tilanteeseen, jossa ohjelma ei toimi oikein muuttujan arvon ylittäessä ylä- tai alarajan. Usein termeillä yli- ja alivuoto vielä erotellaan se, mentiinkö alarajan ali vai ylärajan yli.

Taulukko 2 listaa on C-kielen kokonaislukujen raja-arvoja. Sitä tutkiessa kannattaa huomata, että annetut ala- ja ylärajat eivät ole vakioita vaan standardin edellyttämiä minimejä ko. arvoille. 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 toisissa `long int`:a. Joka tapauksessa ympäristöstä riippumatta `int` on vähintään samankokoinen kuin `short int`, joten sitä tulisi käsitellä kuten `short int`-tyyppiä. Lisäksi merkkimuuttuja `char` on listattu tähän listalle, koska C ei käsittele yksittäistä merkkiä kirjaimena vaan ASCII-aulukon merkkiä vastaavana numeroarvona eli indeksinä. Käytännössä C-kieli tallentaa lukuarvon, mutta esittää tulostettaessa sitä vastaavan ASCII-aulukon merkin.

Taulukko 2: C-kielen kokonaislukutyyppien raja-arvoja, koko tavuina

| Muuttujan nimi | Tyyppi | Koko | Alaraja | Yläaraja |
|--------------------------------|--------------------|------|-------------|------------|
| etumerkitön arvo /merkki | unsigned char | 1 | 0 | 255 |
| etumerkillinen arvo/merkki | signed 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 |

Taulukko 3 tiivistää liukulukujen tietoja ja on hyvä muistaa, että ohjelmointikielet käsittelevät desimaalilukuja bittiarvoina ja siten desimaaliluvut ovat approksimaatioita. 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 $2.3 \cdot 10^{-9}$ –kokoluokan (0.00000000023) eroja. Ja mikäli tämä ei riitä, voidaan tarkkuus kaksinkertaistaa siirtymällä `double`-liukulukutyyppiin.

Taulukko 3: C-kielen liukulukujen raja-arvoja, koko tavuina

| Muuttujan nimi | Tyyppi | Koko | Alaraja | Yläraja |
|--------------------------------------|---------------------|------|--------------|--------------|
| Liukuluku | <code>float</code> | 4 | 1.17549e-38 | 3.40282e+38 |
| Kaksinkertaisen tarkkuuden liukuluku | <code>double</code> | 8 | 2.22507e-308 | 1.79769e+308 |

Operaattorit

Numeeristen muuttujien mielekäs käyttö edellyttää vastaavia laskuoperaattoreita. C-kielessä on monia operaattoreita ml. lasku- (Taulukko 4), loogiset- (Taulukko 4), Boolean- (Taulukko 5) ja bittiopeattorit (Taulukko 6). Monet näistä ovat samoja kuin Python-ohjelmoinnissa, vaikka C-kielessä ei ole avainsanoja `True` ja `False` vaan niiden sijaan käytetään tyypillisesti numeerisia esitysmuotoja 0 (`False`) ja 1 (`True`) väittämien yhteydessä.

Taulukko 4. Laskuoperaattorit

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

Taulukko 5. 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 6. Boolean-operaattorit

| Operaattori | Nimi | Selite | Esimerkki |
|-------------|-------------|---|---|
| ! | Boolean NOT | !x: Jos x on 1, palauttaa arvon 0. Jos x on 0, palauttaa arvon 1. | x = 1; !x palauttaa arvon 0. |
| | Boolean OR | x y: Jos x on 1, palauttaa arvon 1; x:n arvolla 0 palauttaa y:n arvon. | x = 1; y = 0; x y palauttaa arvon 1. (Merkki saadaan näppäimillä AltGr - ”<”). |
| && | Boolean AND | x && y: Jos x on 0, palauttaa arvon 0; x:n arvolla 1 palauttaa y:n arvon | x = 0; y = 1; x && y palauttaa arvon 0 koska x on 0. |

Taulukko 7. 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 bittijonona 10, josta siirryttäessä vasemmalle 2 bittiä antaa bittijonon 1000, joka on desimaalilukuna 8. |
| >> | Siirto oikeaan | Siirtää luvun bittejä annetun numeroarvon verran oikealle. | 11 >> 1 palauttaa 5. 11 on bittijonona 1011, josta siirryttäessä oikealle 1 bitti antaa bittijonon 101, joka on desimaalilukuna 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 eli presedenssi

Jos sinulla on vaikkapa lauseke $2 + 3 * 4$, niin missä järjestyksessä laskutoimitukset tehdään? Matematiikassa on omat laskusäännöt eri operaattoreille ja niiden suoritusjärjestyksille ja tilanne oli sama myös Python-ohjelmoinnissa. Laajemmin ottaen ohjelmointikieliin on määriteltä suoritusjärjestys operaattoreille ja näistä on hyvä olla tietoinen. C-kielessä operaattoreiden suoritusjärjestys noudattaa Taulukko 8 järjestystä.

Taulukko 8. C-kielen operaattorien suoritusjärjestys

| Ryhmä | Operaattori | Suoritussuunta | |
|--------------------|------------------|---------------------|--------------|
| Sijoitukset, sulut | () [] -> . ++ -- | vasemmalta oikealle | Korkea |
| 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 | |
| Looginen OR | | vasemmalta oikealle | |
| Ehtolause | ?: | oikealta vasemmalle | prioriteetti |
| Sijoitus | = += -= *= /= %= | oikealta vasemmalle | |
| Pilkku | , | vasemmalta oikealle | |

Suoritusjärjestys on 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.

Muotoilumerkkijonon liput ja muunnosmerkit

Muotoiltu tulostus ja lukeminen `printf`- ja `scanf`-funktiolla edellyttää erilaisia muotoilumerkkejä, joten ne esitetty tiivistetysti alla, Taulukko 9 ja Taulukko 10. Merkkien käyttö käytiin läpi tarkemmin aiemmin tässä luvussa.

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 |

C-kielen ominaispiirteitä

Monet C-kielen ominaispiirteistä eivät tunnu aloittelevasta C-ohjelmoijasta loogisilta vaan ensimmäinen ajatus on, että ”kääntäjässä on virhe”. Usein, tai siis aina, nämä ratkaisut ovat kuitenkin täysin loogisia ja luonnollisia kun asiaa ajattelee 1970-luvun vaihteen ohjelmoijan ja matemaatikon näkökulmasta. Alla katsotaan muutamia tyypillisiä ongelmia aiheuttavia asioita vähän tarkemmin ja käydään läpi mistä niissä on kysymys. Vastaavia asioita tulee vastaan myös myöhemmin, joten jos tavoitteenasi on kehittyä ohjelmoijana, kannattaa miettiä miten tällaisia ominaispiirteitä tunnistaa ja miten ne hoitaa tehokkaasti. Jos taas tavoitteena on ”vain tämän kurssin läpipääsy”, ei näistä useimmista tarvitse välittää ja tarvittaessa yksittäisiä asioita voi opetella ulkoa.

C-kielisiä ohjelmia tehdään mitä erilaisimpiin laiteympäristöihin, mutta itse C-kieli on standardoitu, jotta ohjelmien kehitysympäristöt eivät olisi sidoksissa niiden suoritusympäristöön. Kuten tässä vaiheessa on jo varmaan selvää, ohjelmia kehitetään mm. Windows, Linux ja Apple -ympäristöissä ja keskeisimmistä työkaluista on näissä kaikissa ympäristöissä toimivia versioita. C-kieleen liittyviä

standardeja on julkaistu vuodesta 1989 ja uusin versio C18 on ISO/IEC 9899:2018 (ISO/IEC 2018) Vuodelle 2021 suunniteltu versio on edelleen työn alla ja sen luonnoksessa oli syksyllä 2019 641 sivua (ISO/IEC 2019). Standardi määrittelee varsin tarkasti C-kielen ominaisuudet, mutta ohjelmoijan kannalta oleellista on ymmärtää edellisessä aliluvussa esiteltyt taulukot ja niiden idea. Standardeja noudattamalla laitekehittäjät voivat valita omiin tarpeisiinsa optimaaliset tietotyypit jne. ja siitä huolimatta kehittää ohjelmat esim. Windows-ympäristössä VSC-editorilla ja GNU-kääntäjällä. Usein pienissä sulautetuissa laitteissa pitää pystyä optimoimaan muistin käyttöä ja standardien avulla se onnistuu, kun esimerkiksi kokonaisluvusta löytyy monta erilaista vaihtoehtoa ml. lyhyt ja pitkä, etumerkillinen ja etumerkitön sekä tarpeen tullen lyhyenkin kokonaisluvun muistitarve voidaan puolittaa käyttämällä muistia yksi tavu eli byte.

C-kielen keskeisiä piirteitä muistin käytön hallinnan lisäksi on pienten käskyjen eli funktioiden käyttö. Tästä on hyvä esimerkki mahdollisuus yksittäisten merkkien hakuun `std::in` syöttökanavasta `getchar()`-funktioilla ja se, että tietoa kysyttäessä annetaan ensin ohjeet käyttäjälle `printf`-käskyllä ja vasta sitten luetaan tieto `scanf`-käskyllä. Tällä luvussa esitellään muutamia muita C-ohjelmille tyypillisiä asioita, jotka aiheuttavat usein hämmennystä aloittelevien ohjelmoijien parissa. Nämä ovat luonteeltaan vastaavia kuin edellisessä luvussa olleet taulukot eli nämä kannattaa katsoa läpi, jotta niistä jää jälki takaraivoon ja jos joskus ohjelmoidessa törmää tällaiseen juttuun, voi näitä katsoa tarkemmin täältä ja yrittää päästä siten eteenpäin.

Tietotyyppimuunnokset ja kokonaislukujako

Jakolaskun $2 / 3$ tulos on monien mielestä 0.6667 ja matemaatikkojen piirissä keskustellaan lähinnä tarkkuudesta ja esitysasusta, sillä $2/3$ on tarkka arvo ja joidenkin mielestä murtoluku näyttää paremmalta kuin desimaaliluku. Python-ohjelmoijien kannalta oleellinen kysymys on se, pyöristyykö viimeinen desimaali ylös- vai alaspäin ja monta desimaalia pitäisi olla. Mutta aloitteleva C-ohjelmoija pysähtyy tyypillisesti ihmettelemään, miten tulos voi olla 0. Matemaattisesti tämä tulos on epäuskottava, mutta kun muistetaan, että C-kielen kannalta kyseessä on kokonaislukujakolasku ja siten kysymys voidaan esittää muodossa ”kuinka monta kertaa luku 3 menee lukuun 2”, niin tällöin ainoa oikea tulos on 0. C-kieli tekee harvoin automaattisesti tietotyypin muutoksia ja lähtökohtana on, että käyttäjä on tarkoittanut asian niin kuin se on koodiin kirjoitettu eli tässä yhteydessä kokonaislukujakona.

C-kielessä $2 / 3$ jakolasku tuottaa tulokseksi 0.6667 muuttamalla toinen operandeista liukuluvuksi (float)-käskyllä, esim. `“(float)2 / 3”`. Kun yksi laskutoimitukseen osallistuvista operandeista on liukuluku, suoritetaan lasku liukuluvuilla ja tulos on liukuluku. Eli varsinaisen laskuoperaation lisäksi ohjelmoijan tulee miettiä laskutoimitus myös käytettävien tietotyyppien kannalta virheiden välttämiseksi. Python-tyyliset automaattiset tietotyyppimuunnokset helpottavat ohjelmoijan elämää ja useimmissa tapauksissa lopputulos on se, mitä ohjelmoija haluaakin, mutta C-kieli on kehitetty vuosia ennen Pythonia, joten se toimii eri tavalla.

Esimerkki C-kielen automaattisesta tyyppimuunnoksesta on liukuluvun sijoittaminen kokonaislukumuuttujaan. Tässä yhteydessä desimaaliosa leikataan pois, joten tulos ei välttämättä ole ohjelmoijan kannalta toivottu. C-kielessä automaattiset toiminnot on minimoitu, jotta ohjelmoija voi optimoida operaatioita siellä missä haluaa – jos haluaa. Ja ohjelma suoritetaan niin kuin ohjelmoija on sen kirjoittanut eli vastuu ohjelmasta on yksikäsitteisesti ohjelmoijalla.

Tiivis koodi

Pythonissa yksi looginen käsky, lause, vastasi lähtökohtaisesti yhtä fyysistä riviä. Myös C-kielessä kannattaa noudattaa tätä periaatetta, vaikka lauseiden päätyminen puolipisteeseen mahdollistaakin

useiden lauseiden ja ääritapauksessa koko ohjelman kirjoittamisen yhdelle riville. C-kieli mahdollistaa tiiviin koodin kirjoittamisen ja monet sen syntaksin ominaisuudet tukevat sitä. Näin ollen ohjelmoijan vastuulle jää usein huolehtia siitä, että ohjelma on ymmärrettävää ja ylläpidettävää. Tässä luvussa on muutama esimerkki siitä, miten tiivis koodi voi kääntyä itseään vastaan ja johtaa virheisiin – joskus jo kirjoitusvaiheessa ja usein viimeistään ylläpitovaiheessa, kun toinen ohjelmoija lukee koodia ja yrittää ymmärtää sitä sekä tehdä siihen muutoksia.

Kokonaislukujaon yhteydessä käsitelty tyyppimuunnos $(float)2 / 3$ on hyvä kohta miettiä myös operaattoreiden presedenssejä. Tulostamalla em. laskutoimituksen tulos C-ohjelmasta näkyy, että kaikki menee oikein. Tässä lausekkeessa on kuitenkin myös toinen ihmetyksen paikka, sillä se sisältää kaksi operaattoria ja niiden suoritusjärjestyksellä on merkitystä. Eli muuttamalla 2 ensin liukuluvuksi, $(float)$ operaattori, ja tekemällä sen jälkeen jakolasku, $/$ -operaattori, menee kaikki oikein. Ja C-kieli toimii näin, koska operaattoreiden presedenssit näin määräävät taulukon 7 mukaisesti. Mikäli tuo presedenssitaulukko ei ole vielä painunut täydellisesti muistiin, kannattaa käyttää sulkuja aina tällaisten lausekkeiden ympärillä eli varmuuden vuoksi voisi kirjoittaa $((float)2) / 3$. Tämä näyttää jo liioittelulta, mutta tiivistetysti kannattaa käyttää mieluummin liian paljon sulkuja kuin liian vähän, sillä puuttuvat sulut tuottavat väärän tuloksen, kun taas liian monet sulut tekevät lukemisesta haastavampaa.

Toinen tyyppillinen esimerkki tiivistä koodista on alla oleva lause:

```
x = a == b;
```

Python ei hyväksy tätä lausetta ollenkaan, koska siinä on ilmeinen riski virheeseen. C-kääntäjä tulkitsee lauseen kuitenkin ilman ongelmia eli suorittaa $a:n$ ja $b:n$ vertailun ja sijoittaa vertailun tuloksen $x:n$ arvoksi. Eli näin asia menee presedenssitaulukon mukaan, jossa vertailun presedenssi on sijoitusta korkeampi ja se tehdään ensin. Äkkiseltään lausetta lukeva voisi kuitenkin päätyä siihen, että $x:ään$ sijoitetaan $a:n$ arvo, tai sitten ohjelmoijalle on käynyt kirjoitusvirhe ja tarkoitus onkin sijoittaa $a:lle$ arvoksi b , ja sen jälkeen $x:lle$ a – siis presedenssitaulukon mukaan. Tässä vaiheessa Python-tulkin kieltäytyminen yllä olevan lauseen hyväksymistä kuulostaa järkevältä ja C-ohjelmoijan kannattaa tehdä samoin eli kirjoittaa selkeää koodia ja käyttää aina tarvittaessa sulkuja selventämään ohjelmaa ja sen tarkoitusta.

Kaikki tiivistä koodia tuottavat C-ohjelmointikäytännöt eivät ole huonoja. Esimerkiksi $++$ ja $--$ operaattorit ovat käteviä monessakin kohtaa kuten erityisesti osoittimien kohdalla tullaan huomaamaan. Jo tässä vaiheessa on hyvä huomata, että $++i$ ja $i++$ -operaatioilla on eroa eli $++i$ kasvattaa $i:n$ arvoa yhdelle *ennen* käyttöä ja $i++$ kasvattaa arvoa vasta käytön *jälkeen*. Tiivistetysti tiivis koodi ei ole huono asia niin kauan kuin se ei muodostu itseisarvoksi ja haittaa ohjelman ymmärrettävyyttä ja ylläpidettävyyttä.

Pienen C-ohjelman tyyliohjeet

Tässä luvussa teimme pieniä C-ohjelmia, jotka olivat yhdessä tiedostossa ja joissa oli vain pääohjelma mutta useita muuttujia. Jo näiden ohjelmien tekemisessä on hyvä omaksua selkeät vakiokäytännöt, joita noudattaa tämän oppaan yhteydessä. Isompia ohjelmia tehdessä nämä ohjeet eivät välttämättä riitä, vaan silloin kannattaa tehdä itse projektiin sopivat tyyliohjeet. Monissa yrityksissä on omat tyyliohjeet, esim. nimellä programming standards, joita kaikkien tulee noudattaa ohjelmien ymmärrettävyyden ja ylläpitämisen parantamiseksi. Alla on muutamia keskeisiä asioita, jotka kannattaa pitää mielessä jo alusta alkaen.

Muuttujien nimeäminen. Muuttujille kannattaa antaa kuvaavat nimet, jotta ohjelman laajentuessa muuttujan nimestä näkee sen tarkoituksen. Muuttujien määrän kasvaessa ne voi ryhmitellä käyttötarkoituksen mukaan esimerkiksi niin, että nimi muodostuu useasta osasta samalla logiikalla

ja tarkentavia osia lisätään aina tarpeen mukaan. Mikäli samantyyppisiä muuttujia on muutama, voi ne erotella numeroimalla. Käsitlemme monesta samantyyppisestä muuttujasta muodostuvat tietorakenteet myöhemmin, jonka jälkeen kannattaa tyypillisesti siirtyä niihin. Tässä vaiheessa tyypillisiä muuttujanimiä ovat seuraavat:

1. Useita päivämääriä, joita vertaillaan tms.: paiva1, paiva2
2. Useita opiskelijoita ja paljon tietoja näistä:
 - a. opiskelija1NimiEtu
 - b. opiskelija1NimiSuku
 - c. opiskelija1PuhelinTyo
 - d. opiskelija1PuhelinMobiili
 - e. Vastaavasti opiskelija2 ja opiskelija3 jne.

Sisennykset. Tässä vaiheessa olemme tutustuneet pääohjelmaan, joka kannattaa sisentää systemaattisesti seuraavalla tavalla:

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

int main(void) {    // koodilohkon avaava { main-rivillä
    printf("Minä tein tämän!\n"); // koodi sisennetty
    return(0);      // aina samaan tasoon
}                  // koodilohkon sulkeva } omalla rivillä
```

Tiedostorakenne. Pienten C-ohjelmien tiedostot kannattaa ryhmitellä seuraaviin osiin:

1. Ohjelman alkukommentti
 - a. Kuka tehnyt, milloin, tiedostonimi; esim.: /* 20210520 un E1_1.c */
2. Kirjastojen sisällytykset
 - a. Esim. #include <stdio.h>
3. Pääohjelma
 - a. Muuttujien määrittelyt ja alustukset
 - b. Tietojen kysyminen käyttäjältä
 - c. Tietojenkäsittely, esim. laskenta
 - d. Tulosten tulostaminen käyttäjälle

Tietotyytit. C-kielen tarkkuus tietotyyppien suhteen voi tuottaa ongelmia Pythonin jälkeen. Pythonissa tietotyypeistä ei tarvitse välittää kovin paljoa, mutta C-kielessä ne on syytä muistaa koko ajan. Yksi tätä asiaa helpottava tekniikka on Unkarilainen notaatio (Wikipedia 2021), jossa muuttujan tietotyyppi merkitään näkyville muuttujan nimeen. Unkarilaisen notaation käytössä kannattaa katsoa käytettävät tietotyypit ja muodostaa niistä itselle sopiva nimeämisohje, mutta tässä oppaassa tullaan käyttämään jatkossa seuraavia merkintöjä helpottamaan tietotyyppien tunnistamista:

- Oppaassa tähän mennessä käytetyt tietotyypit ja niiden tunnukset muuttujan nimessä:
 - kokonaisluku: int – i
 - pitkä kokonaisluku: long – l
 - liukuluku: float – f
 - kaksoistarkkuuden liukuluku: double – d
 - merkki, 1 kpl: char – c
 - merkkitaulukko, monta merkkiä eli array: char [] – a
- Jatkossa muuttujat tullaan nimeämään seuraavalla tavalla:
 - int iLuku; long lLuku; float fLuku; double dLuku;
 - char cMerkki; char aNimiEtu[30];

Yhteenveto

Tässä luvussa käytiin läpi monia asioita, jotta pääsimme tekemään ohjelmia. Suurin osa käsitellyistä asioista on tuttuja ohjelmointia osaaville ja siksi katsoimme kaikista keskeisistä asioista C-kielen kannalta oleelliset taustiedot ja miten asiat tehdään C-kielillä. Näin ollen useimmat asiat olivat entuudestaan tuttuja, kuten esim. tietojen lukeminen näppäimistöltä, mutta C-kielen toteutus on omanlaisensa erillisen tulostuksen ja useiden tietojen lukufunktioiden takia.

Kerrataan aluksi luvun osaamistavoitteet, sitten tiivistetysti luvussa läpikäydyt asiat ja katsotaan lopuksi luvun keskeiset asiat kokoava ohjelmointiesimerkki.

Osaamistavoitteet

Tämän ensimmäisen luvun osaamistavoitteet ovat kahtiajakoiset. Ensinnäkin sinun tulisi nyt ymmärtää C-kielen historia ja sen suhde Unixiin ja Linuxiin. Samoin C-kielen syntaksin keskeisten konseptien kuten tietotyypit, operaattorit, raja-arvot ja presedenssit pitäisi nyt tuntua tutuilta asioilta. Kun käsitteet ovat tuttuja, voit palata ja tutustua niihin tarkemmin tarpeen vaatiessa.

Sinun pitäisi nyt osata tehdä C-ohjelmia eli sinulla pitäisi olla toimiva C-kehitysympäristö käytössäsi ja olet kirjoittanut, kääntänyt ja suorittanut C-ohjelmia tässä ympäristössä. Itse C-ohjelmointiin liittyen ymmärrät pääohjelman, koodilohkon, kirjastojen, muuttujien, tietotyyppien ja operaattorien perusideat ja pystyt käyttämään niitä ohjelmissasi. Tiedon lukemisen ja tulostamisen peruskäskyt ovat sinulle tuttuja ja pystyt muotoilemaan tietoa tarpeen mukaan. Muuttujien määrittely ja merkkijonojen käyttö voi tuntua vielä vieraalta Pythonin jälkeen, mutta olet päässyt alkuun niidenkin kanssa. Tiivistetysti pystyt nyt tekemään pieniä C-ohjelmia sekä kääntämään ja suorittamaan niitä Visual Studio Code -editorin ja gcc-kääntäjän avulla VSC:n terminaalissa.

Käsiteltyt asiat

Tässä luvussa kävimme läpi C-ohjelmointiin liittyen seuraavat asiat:

- C-ohjelmointiympäristön asennus: Liite 1. C-ohjelmointiympäristön asennusohje
- VSC-editorin käyttö, ks. Liite 2. VSC-editorin käyttöohje ja ohjelmointivideot
- C-ohjelman kääntäminen: Esimerkki 1.1
- Tulostus: Esimerkki 1.2 ja 1.9
- Muuttujat ja tietotyypit: Esimerkki 1.3, *Pienen C-ohjelman tyyliohjeet*
- Syötteen lukeminen näppäimistöltä: Esimerkki 1.4 ja 1.9
- Merkkijonojen määrittely, luku ja tulostus: Esimerkki 1.5, 1.6 ja 1.7
- Muotoilumerkit: Esimerkki 1.8, Taulukko 9 ja 10
- C-kielen määrittelyt: kokonaisluvut Taulukko 1, liukuluvut Taulukko 2, laskuoperaattorit Taulukko 3, Loogiset- eli vertailuoperaattorit Taulukko 4, Boolean-operaattorit Taulukko 5, Bittioperaattorit Taulukko 6, Presedenssit Taulukko 7

Käytetyt työkalut: Win10, VSC, WSL1, Linux, gcc

Tässä oppaassa käytettävä ja suositeltu ohjelmointiympäristö muodostuu alla olevista työkaluista. Myöhemmin oppaan lopussa otetaan käyttöön lisää työkaluja. Näiden työkalujen asennusohjeet löytyvät liitteestä 1.

- Win10 käyttöjärjestelmä

- Visual Studio Code -editori
- WSL1 eli Windows Subsystem for Linux versio 1
- Ubuntu-Linux 20.04 LTS
- gcc kääntäjä

Kokoava esimerkki

Seuraavaan esimerkkiin on koottu tämän luvun keskeiset asiat ohjelmoinnin kannalta eli kirjastojen sisällytys ohjelmaan, main-funktion käyttö, muuttujien määrittely, erityyppisten tietojen lukeminen käyttäjältä sekä tietojen tulostus näytölle. Ohjelmassa ei ole alustettu muuttujia, koska näihin muuttujiin luetaan arvot käyttäjiltä. Normaalisti C-kielessä on syytä alustaa muuttujat ohjelman alussa, sillä kaikki kääntäjät eivät alusta muuttujia vaan se on ohjelmoijan vastuulla.

Esimerkki 1.9. Luvun 1 kokoava esimerkki

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

int main(void) {
    /* Muuttujien määrittelyt ja alustukset */
    int iLuku;
    float fLuku;
    char cMerkki;
    char aSana[30];
    char aRivi[30];

    /* Tietojen kysyminen käyttäjältä */
    /* Kokonaisluku ja liukuluku */
    printf("Anna kokonaisluku: ");
    scanf("%d", &iLuku);
    printf("Anna liukuluku: ");
    scanf("%f", &fLuku);
    /* Yksi merkki */
    getchar();
    printf("Anna yksi merkki: ");
    scanf("%c", &cMerkki);
    getchar();
    /* Yksi sana */
    printf("Anna yksi sana: ");
    scanf("%s", aSana);
    getchar();
    /* Koko rivi rivinvaihtomerkki mukaan lukien */
    printf("Anna yksi rivi (max 28 merkkiä): ");
    fgets(aRivi, 30, stdin);
    aRivi[strlen(aRivi)-1] = '\0';

    /* Tietojen tulostaminen käyttäjälle */
    printf("Kokonaisluku on '%d', liukuluku on '%5.2f' ja merkki on '%c'.\n",
           iLuku, fLuku, cMerkki);
    printf("Sana on '%s' ja rivi on '%s'.\n", aSana, aRivi);

    return(0);
}
/* eof */
```

Luku 2. Valinta- ja toistorakenteet, esikäsittelijä

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

Valintarakenteet ja ehdollinen suorittaminen

Valintarakenteen käyttö on suoraviivaista Python-ohjelmointikielessä, koska kielessä ei ole muita valintarakenteita kuin `if-elif-else`-rakenne. C-kieli tukee oletuksena kolmea erilaista valintarakennetta, jotka ovat Pythonista tuttu `if-else`, suoraan valintaan perustuva `switch-case` sekä ehdollinen lauseke eli operaattori `?:`. Tutustumme näihin rakenteisiin yksinkertaisten esimerkkien avulla.

if-else

C-kielen `if`-rakenne toimii kuten Pythonin vastaava rakenne. Ainoa varsinainen 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 tarkastella näiden käytännön eroja esimerkiksi puuttumatta perusrakenteeseen.

Esimerkki 2.1. if-else -rakenne

```
#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);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E2_1
Anna kokonaisluku: 25
Antamasi luku on pienempi kuin 100.
Lopetetaan.
un@LUT8859:~/Opas$ ./E2_1
Anna kokonaisluku: 250
Antamasi luku on 100 ja 1000 välillä.
Lopetetaan.
un@LUT8859:~/Opas$ ./E2_1
Anna kokonaisluku: 1001
Antamasi luku on suurempi kuin 1000.
Lopetetaan.
```

Kuinka koodi toimii

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

Kuten esimerkistä näemme, on mekanismi hyvin samankaltainen Python-ohjelmointikielen vastaavan rakenteen 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. C-kielessä sisennyksellä ei merkitystä, mutta luettavuuden kannalta se edelleen keskeinen käytäntö.

Esimerkki 2.2. Koodilohkojen sisentäminen

```
#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);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E2_2
Sana voisi olla kolmipyörä.
Lopetetaan.
```

Huomioita koodista

Esimerkki havainnollistaa sisennyksen ja koodin huolellisen muotoilun merkitystä. Koska jokainen alkava koodiosio avaa uuden aaltosulun, ei useamman alkaneen sisennyksen jälkeen enää pysty silmäääräisesti erottamaan, onko sulkeita oikea määrä. Jos sulkeutuva aaltosulku on väärässä paikassa – tai sellainen puuttuu kokonaan – ei ohjelma enää käänny. Itse asiassa se, että ohjelma kääntyy väärillä koodilohkoilla, on huono asia, sillä tällöin ohjelma toimii tyypillisesti väärin ja ongelman löytäminen on työlästä. Yksittäisen sulkumerkin paikan selvittäminen 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 voi kuvata valikkona, johon ohjelmoidaan kaikki valintamuuttujan vaihtoehdot ja 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 2.3. switch-case -rakenne

```
#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("Tuntematon valinta.\n");
            break;
    }

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


Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E2_3
Tee valinta (1-3): 1
Valitsit 1.
Lopetetaan.
un@LUT8859:~/Opas$ ./E2_3
Tee valinta (1-3): 5
Tuntematon valinta.
Lopetetaan.
```

Kuinka koodi toimii

`switch`-rakenteen syntaksi poikkeaa hieman `if-else` -rakenteesta. Ensinnäkin `switch` itsessään muodostaa yhden koodiosion, joten aaltosulkuja tarvitaan vähemmän. Toiseksi `case`-valinta vertaa muuttujan arvoa valinta-vaihtoehtoihin, joita tässä tapauksessa ovat numeroarvot 1, 2 ja 3. `switch`-lauseessa vertaillaan *vakioita*, joten vertailun voi tehdä kokonaisluvuilla tai ASCII-merkeillä, jolloin vertailu tehdään ASCII-taulukon kokonaislukuindekseillä.

`case`-valinta muodostaa 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 tyypillisesti `break`-käskyyn. `break`-käsky toimii C-kielessä samalla tavoin kuin Pythonin vastaava käsky eli se 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 toteudu. `default` -valinta on tavallisesti viimeisenä, mutta `default` voi esiintyä missä kohdin tahansa rakennetta, esimerkiksi rakenteen alussa. Siksi `default`-osio on syytä päättää aina `break`-käskyyn, jotta rakenne toimii oikein myös koodin muokkaamisen jälkeen.

Ehdollinen lauseke

C-kieli tarjoaa useita mahdollisuuksia kirjoittaa tiivistä koodia. Yksi tällainen on ehdollinen lauseke, joka mahdollistaa valintarakenteen toteuttamisen lyhyesti. Alla olevassa esimerkissä vertaillaan lukuja 1 ja 2, jolloin vertailun tulos on totuusarvo tosi tai epätosi. Tämän totuusarvon perusteella valitaan kysymysmerkin jälkeen kysymysmerkin ja kaksoispisteen välinen arvo, jos vertailulauseke on tosi, ja kaksoispisteen jälkeinen lause, jos se on epätosi. Tiivistä ja yksinkertaista – ainakin ammattimaisen C-ohjelmoijan mielestä. Tätä saa siis käyttää, muttei tarvitse, ja satunnaiselle C-ohjelmoijalle riittää ihan hyvin `if`-valintarakenne.

Esimerkki 2.4. Ehdollinen lauseke

```
#include <stdio.h>

int main(void) {
    int luku1 = 2, luku2 = 13, suurempi;
    suurempi = (luku1 > luku2) ? luku1 : luku2;
    printf("Luvuista suurempi on %d.\n", suurempi);
    return(0);
}
```

Toistorakenteet

Myös toistorakenteissa C-kieli tarjoaa enemmän vaihtoehtoja kuin Python. Pythonista tuttujen alkuehtoisen `for`- ja avoimen `while`-rakenteen lisäksi kielestä löytyy loppuehtoinen `do-while`. C-kieli tukee samoja toisto- ja ohjausrakenteiden ohjauskäskyjä kuten `continue` ja `break`.

while-rakenne

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

Esimerkki 2.5. while -rakenne

```
#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);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E2_5
Olemme kierroksella 0!
Olemme kierroksella 1!
Olemme kierroksella 2!
Olemme kierroksella 3!
Olemme kierroksella 4!
Lopetetaan tähän.
```

Kuinka koodi toimii

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, joten tutkitaan sitä hieman tarkemmin.

C-kielen `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, ja muussa tapauksessa lopetetaan sekä siirrytään `while`-rakennetta seuraavalle koodiriville. Tässä rakenteessa joudumme muuttamaan käsin toistorakenteen katkaisua ohjaavaan `kierros`-muuttujaan arvoja, jotta ohjelmamme toimii oikein. Huomaa, että Python-kielestä poiketen `while`-rakenteen loppuun ei voi liittää `else`-osiota.

for-rakenne

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

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

Käytännössä ensin kerromme mitä muuttujaa käytämme kierroslaskurina (perinteisesti `i`, `j` ja `k`) ja minkä arvon ko. muuttuja saa alussa, minkä ehdon tulee täytyä toiston jatkamiseksi ja miten kierroslaskuria muutetaan jokaisella kierroksella. Toistoehdosta tulee muistaa, että toistoa jatketaan niin kauan kuin ehto on tosi ja katkaisu tapahtuu kierroksella, jolloin ehto muuttuu epätodeksi.

Esimerkki 2.6. for -rakenne

```
#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);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E2_6
for-rakenne laskee itse kierroksensa:
0 1 2 3 4 5 6 7 8 9 10
Lopetetaan tähän.
```

Kuinka koodi toimii

Koodi ei sisällä yllätyksiä. Käyttämämme `for`-rakenne alustaa kierrosmuuttujan `i` nollassa. Tämän jälkeen ilmoitamme, että `for`-rakenne jatkuu niin kauan kuin `i` on pienempi tai yhtä suuri kuin 10 ja lopuksi kerromme, että joka kierroksella `i`:n arvo kasvaa yhdellä (`i++`). Muilta osin koodin rakenne on vastaava kuin aiemmassa `while`-rakenteen esimerkissä ja myöskään `for`-rakenteen loppuun ei voi lisätä `else`-osiota. Tulostukseen liittyen kannattaa huomata, että silmukassa ei tulosteta rivinvaihtomerkkejä, jonka vastapainona loppu-tulosteessa on rivinvaihto sekä tekstin alussa että lopussa. Näin loppu-kommentti tulostuu omalle rivilleen.

do-while -rakenne

`do-while` on toistorakenne, jota ei löydy Python-kielestä. Se poikkeaa muista toistorakenteista, sillä se on loppuehtoinen toistorakenne. Tämä tarkoittaa käytännössä sitä, että rakenteen **do-osio suoritetaan aina kerran riippumatta siitä, mitä lopetusehdot ovat**. `do-while` -rakenne toimii muuten samalla tavoin kuin normaali `while`-rakenne.

Esimerkki 2.7. do-while -rakenne

```
#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 do-osion perään ja päättyy puolipisteeseen. */
    /* do {} while (ehto); */

    return(0);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E2_7
do-osio toteutuu ainakin kerran.
Lopetus on valmiiksi pienempi kuin aloitus:
lopetus: 5 ja aloitus: 10
```

Kuinka koodi toimii

Toisin kuin muut toistorakenteet, `do-while` tarkistaa lopetusehdon vasta kun ensimmäinen kierros on tehty kokonaan. Esimerkissä toistoa oli tarkoitus jatkaa niin kauan kuin `aloitus` on pienempi kuin `lopetus`, mutta nyt ensimmäinen kierros suoritettiin, vaikka jo toiston alkaessa luku oli kaksinkertainen. `do-while` -rakenne onkin parhaimmillaan kun suoritetaan toistoja, joissa ensimmäinen kierros periaatteessa testaa onko jotain käytettävissä ja jatkaa toistoa loppuun asti. Tällaisia tilanteita ovat esimerkiksi tiettyjen tietorakenteiden läpikäynti, merkkien lukeminen tiedostosta ja dynaamisten rakenteiden selaaminen.

Muita ohjauskäskyjä

C-kielen ohjauskäskyihin lukeutuu tuttujen `return`, `continue` ja `break`'n lisäksi myös `goto`-lause. Käydään nämä tutut läpi ensin ja katsotaan sitten, miksi `goto`-lausetta ei saa käyttää tässä oppaassa.

return, continue, break

C-kielen ohjauskäskyt `return`, `continue` ja `break` 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-kielessä ei ole.

Esimerkki 2.8. return, continue ja break -ohjausrakenteet

```
#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);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E2_8
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!
```

Kuinka koodi toimii

Tässä esimerkissä näkyy tyypillinen do-while -rakenteen käyttö. Ohjelma selvittää, onko muuttujan alku arvo jossain vaiheessa 42. Ensin 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 taas erotus on 0, 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.

goto-käskyn käyttö kielletty

C-kielessä on myös ohjausrakenne `goto`. Sen idea on tehdä ohjelmakoodiin nimetty kohta ja ohjelman suoritus siirtyy siihen hyppäämällä `goto nimi` -käskyllä. Koska ratkaisu johtaa tyypillisesti täysin hallitsemattomaan koodisekamelskaan, on käskyn käyttö yleisesti ottaen huonoa ohjelmointityyliä eikä sitä käsitellä tämän enempää tässä oppaassa vaan tämän oppaan yhteydessä sen **käyttö on kielletty**. Jos jostain syystä haluat tietää käskystä enemmän esim. joutuessasi ylläpitämään ko. rakenteita sisältävää koodia, löytyy käskystä lisäinformaatiota normaaleissa lähdemateriaaleissa kuten Kernighan ja Ritchie (1988).

Esikäsittelijä

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

Kirjastojen sisällyttäminen `#include`-direktiivillä

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 esimerkeissä 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), tekisimme sen komennolla

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

Huomaa, että C-kielessä meidän ei tarvitse erikseen kertoa, mistä kirjastosta funktio haetaan. Esimerkiksi merkkijonojen lukeminen käyttäjältä voidaan toteuttaa käskyllä `scanf`, joka on `stdio`-kirjaston funktio. Python-analogialla käyttäisimme siis komentoa `stdio.scanf`, mutta C-kielessä riittää pelkkä funktion nimi. Luonnollisesti tämä tarkoittaa sitä, ettei saman nimisiä funktioita saa olla ja peruskirjastot onkin suunniteltu niin, ettei ongelmia synny. Taulukko 11 nimeää muutamia hyödyllisiä funktiokirjastoja.

Taulukko 11. Hyödyllisiä funktiokirjastoja

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

Tulemme tutustumaan näihin kirjastoihin ja niistä löytyviin funktioihin sitä mukaa kun tarvitsemme niitä. Keskeisimmät kirjastofunktiot esitellään lyhyesti liitteestä 3. Jos haluat lisätietoa kirjastoista, voit tutustua C-kielen perusfunktioihin mm. Eric Hussin kirjoittaman C-referenssioppaan avulla (Huss 1997). Opas on englanninkielinen. Omien funktiokirjastojen tekemiseen palaamme tämän oppaan lopussa.

Vakioiden määrittely #define-direktiivillä vs. C-kielen const-määrittely

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 esikäsittelijä korvaa annetut merkkijonot (vakiot) niiden arvoilla. Käytännössä tämä tarkoittaa sitä, että annamme lähdekoodin alussa käskyn

```
#define nimi arvo
```

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

Esimerkki 2.9. #define -määrittely

```
#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 määrittely toimii myös ");
        printf("numeroarvon kanssa.\n");
    }

    return(0);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E2_9
99999
Toimii!
#define-käskyn määrittely toimii myös numeroarvon kanssa.
```

Kuinka koodi toimii

Tässä esimerkissä olemme luoneet vakiot TRUE, FALSE sekä NUMEROARVO, ja antaneet niille arvot 1, 0 sekä 99999. Kuten näemme, vakiot toimivat käytännössä siten, että ne ovat eräänlaisia muuttujia, joiden arvoa ei pystytä ajonaikaisesti muuttamaan. Myös kääntäjä ymmärtää vakiot numeroina ja estää niiden muuttamisen ohjelman suorituksen aikana.

Erityisen hyödyllisiä vakiot ovat nimenomaan ”asetusarvoina” esimerkiksi taulukon koolle tai toistorakenteen alku- ja loppuehtoina. Kun määrittelemme arvon yhdessä paikassa lähdekoodin alussa, voimme asetuksien muuttamista 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 kaikkien muutettavien kohtien huomaaminen on tyypillinen ongelma.

C-kielen uudemmissa versioissa merkkijonoliteraalien rinnalle on tullut myös aidot vakio-
muuttujat, joita ei siis korvata esikäsittelyvaiheessa vaan ne menevät kääntäjälle asti ja niille voidaan suorittaa esimerkiksi tyyppitarkistuksia. Nämä vakio-
muuttujat 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;
```

Kun C-ohjelmassa tarvitaan kiintoarvoa eli vakiota, kannattaa käyttää C-kielen `const`-määrettä muuttujan määrittelyssä. Tämä on suositeltava ratkaisu, sillä se mahdollistaa tyyppien tarkastamisen ja muuttujien nimien käsittelyn debuggerissa. Myös esikäsittelijän `define`-direktiivi toimii täysin ja sitä voi myös käyttää vapaasti. Palaamme esikäsittelijän `define`-direktiiviin vähän myöhemmin tässä oppaassa, sillä sitä voidaan käyttää myös makrojen määrittelyyn.

Ehdollinen koodilohko `#if 0 ... #endif`-direktiiveillä

Esimerkiksi tämän oppaan ohjelmointiesimerkkien videoinnin yhteydessä halutaan usein poistaa käytöstä monta riviä koodia, jotta uutta asiaa voidaan tutkia tarvitsematta huomioida muuta 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. Esikäsittelijän `if-end`-direktiivit mahdollistavat haluttujen rivien mukanaolon lähdekoodissa, mutta ko. rivien poistamisen esikäsittelijän suorittamisen yhteydessä, jolloin ne eivät vaikuta 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
```

Edellisen ehdollisen koodilohkon lisäksi esikäsittelijä voi määritellä koodia ehdollisesti.

```
#ifndef xx
#define xx
/*
 */
#endif
```

Tätä `ifndef`-lohkoa (if not defined) tullaan käyttämään oppaan lopussa kun ohjelmat muodostuvat useista tiedostoista. Tähän palataan siis myöhemmin, mutta lähtökohtaisesti se on yksi variaatio `if`-direktiivistä vastaavalla tavoin kuin `#ifdef` ja `#define` jne.

C-kielen ominaispiirteitä

Edellä tuli vastaan C-kielen ominaispiirteitä, jotka tuntuvat usein vanhanaikaisilta ja vaikeilta tavoilta tehdä asioita. Katsotaan seuraavaksi näitä asioita vähän lähemmin eli mistä niissä on kysymys ja miten niiden kanssa kannattaa toimia. Ensinnäkin kääntäjälle voi antaa monenlaisia ohjeita suorittavaan käännökseen liittyen. Toiseksi ohjelmoidessa muiden kirjoittamat funktiot ovat keskeisessä roolissa helpottamassa uusien ohjelmien tekemistä, joten tutustutaan lyhyesti C-ohjelmoijan perustietolähteeseen eli man-online-ohjelmointioppaaseen. Kolmanneksi merkkijonojen käsittely C-kielessä on usein haastavaa, joten kerrataan merkkijonon rakenne sekä

katsotaan osoitteen ja osoittimen periaatteita merkkijonojen yhteydessä. Lopuksi kerrataan tähän asti käyttämämme staattisen muistinvarauksen periaatteet.

Kääntäjäoptiot apuna virheiden välttämässä ja etsimisessä

C-kielen rakenne mahdollistaa hyvin erilaisten ohjelmien kirjoittamisen, vaikka alussa kielestä voi tulla pikkutarkkaan määritellyn maku tiettyjen asioiden kuten tietotyyppien takia. Tietotyyppijä on määritelty paljon, jotta kaikkiin tarpeisiin löytyy sopiva tietotyyppi, mutta yksityiskohtaisella tasolla C-kieli mahdollistaa todellisuudessa hyvin erilaiset ratkaisut. Eksoottiset ratkaisut eivät ole hyvästä ylläpidon kannalta, joten kääntäjässä on paljon erilaisia optioita, joilla ohjelmointikäytäntöjä voidaan standardoida. Näistä oli puhetta jo aiemmin ja kun ohjelmien tekoprosessin pitäisi nyt olla tuttu, kerrataan ymmärrettävän ja ylläpidettävän ohjelmointityylin perusteet. Ohjelmia kääntäessä kannattaa ohjeistaa kääntäjää tarkistamaan ohjelmat C-kielen vuoden 1999 standardin mukaisesti (`-std=c99`), ilmoittamaan kaikista varoituksista (`-Wall`) ja ilmoittamaan kaikki mahdollisetkin virheet eli olemaan pikkutarkka (`-pedantic`). Siksi ohjelmat kannattaa kääntää seuraavilla käännösoptioilla:

```
un@LUT8859:~/Opas$ gcc E2_9.c -o E2_9 -std=c99 -Wall -pedantic
```

Kääntäjää voi käyttää muutenkin tarkistamaan käännösprosessin etenemistä. Esimerkiksi kääntäjää voi pyytää tekemään vain esikäsittelyn ja katsoa miltä tiedosto näyttää sen jälkeen. Tämä onnistuu kääntäjän `-E` -optiolla ja alla oleva käsky ohjaa kääntäjän kirjoittamaan esikäsittelyn tiedoston uuteen tiedostoon `esikäsitelty.c`, jota voi tarkastella editorilla.

```
un@LUT8859:~/Opas$ gcc lahdekoodi.c -E > esikäsitelty.c
```

Alla on esimerkkinä lähdekoodi ja sen jälkeen esikäsittely koodi. Huomaa, että kommentit ovat hävinneet ja `#define`-määritellyt vakiot on korvattu numeroilla, mutta `const`-määritelty muuttuja on säilynyt koodissa muuttumattomana. Tämän esikäsittelyn tiedoston alusta puuttuu myös `#include <stdio.h>` -määrittely, joka on korvattu reilulla 700 rivillä koodia kuten jälkikäteen lisätty kommentti kertoo.

Esimerkki 2.10a. Esikäsitlemätön lähdekoodi E2_10a.c

```
#include <stdio.h>
#define MAX 100
#define PII 3.14

int main(void) {
    const int nMax = 50;
    printf("MAX on %d, PII on %f, nMax on %d.\n", MAX, PII, nMax);
    return(0);
}
```

Suoritetaan ohjelman esikäsittely seuraavalla käskyllä:

```
un@LUT8859:~/Opas$ gcc E2_10a.c -E > E2_10b.c
```

Esimerkki 2.10b. Esikäsitelty lähdekoodi E2_10b.c

```
// Tässä tiedoston alussa on reilu 700 riviä koodia #include:n tilalla
# 5 "E2_10a.c"
int main(void) {
    const int nMax = 50;
    printf("MAX on %d, PII on %f, nMax on %d.\n", 100, 3.14, nMax);
    return(0);
}
```

Usein kääntäjäoptiot lisäävät kääntäjän antamien varoitusten määrää. Kääntäjän palautteita kannattaa kuitenkin opetella lukemaan ja etsimään niistä perusongelmat, jotka poistavat usein monia muita virheilmoituksia sekä vähentävät ongelmia esim. siirrettäessä koodia Windows- ja Linux-ympäristöjen välillä. Ei nimittäin ole ollenkaan poikkeuksellista, että Windows-ympäristössä näennäisesti virheettä toimiva ohjelma ei käänny eikä toimi virheettä Linux-ympäristössä.

Tyypillisesti editorit auttavat luettavan ohjelman kirjoittamisessa sisentämällä koodia automaattisesti. Toinen tyypillinen editorin apu on muuttujien ja funktioiden nimien ehdottaminen ja valinta, jolloin vältetään niiden kirjoitusvirheitä. Myös tooltips on yksi tällainen apuväline, jolla on tyypillisesti helppo tarkistaa funktion parametrien määrä ja tietotyypit.

Valmiin koodin tarkastamiseen sisennysten ja muiden hyvien ohjelmointitapojen kannalta on olemassa pretty printer -työkaluja, jotka tarkastavat koodin ja muotoilevat sen haluttaessa ”kauniiksi”. Esimerkkejä tällaisista työkaluista ovat Linuxin lint-ohjelma ja VSC:n C/C++ -laajennos, jossa on ”format document” -toiminnallisuus. Palaamme myöhemmin muihin työkaluihin, joilla C-koodien ymmärrettävyyttä voidaan arvioida ja parantaa. Erityisesti dynaamisen muistinhallinnan yhteydessä kannattaa käyttää työkaluja, jotka tarkistavat muistin vapauttamisen sen varaamisen jälkeen.

Linuxin man-sivut eli manuaali

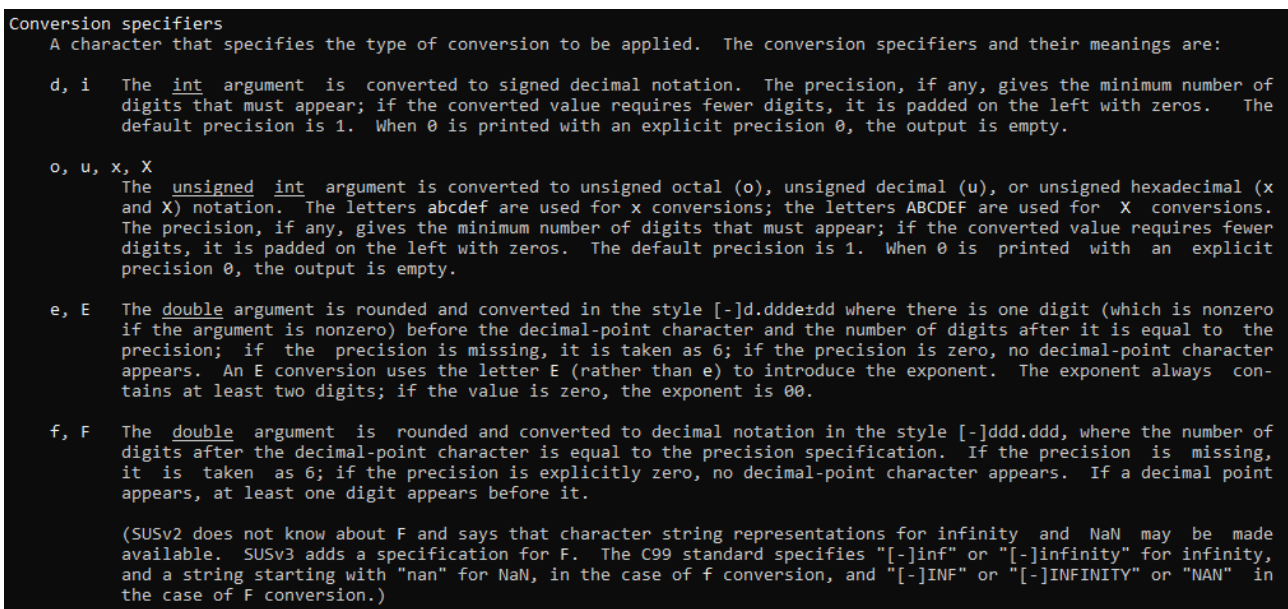
C-kielelle on tyypillistä, että saman asian voi tehdä monilla vähän erilaisilla tavoilla. Olemme tähän mennessä tulostaneet tietoja näytölle `printf` -funktioilla, mutta tästä funktiosta on itse asiassa monta variaatiota, joiden kaikkien tehtävä on ”formatted output conversion”: `fprintf`, `dprintf`, `sprintf`, `snprintf`, `vprintf`, `vfprintf`, `vdprintf`, `vsprintf` ja `vsnprintf`. Vaikka ensimmäinen ajatus on, että kuka noita kaikkia tarvitsee, niin lähtökohtaisesti näille kaikille on sopiva käyttökohde ja sen käyttäminen tekee elämästä helpompaa. Näin ollen kysymys on, että miten ja mistä näistä funktioista, niiden toiminnasta ja käytöstä löytää tietoa?

Unixin perusajatuksiin on alusta alkaen kuulunut, että järjestelmän ohjekirjat, manuaalit, löytyvät itse järjestelmästä. Siksi Linuxissakin komentoriville voi kirjoittaa man-käskyn haluamansa funktion nimen kanssa ja saa näkyville ko. funktion kattavan dokumentaation. Esimerkiksi `printf`-funktioista saa perustiedot esiin man 3 `printf`-käskyllä kuvan 2.1 mukaisesti, jossa man -käsky käynnistää manuaali-ohjelman ja se etsii osiosta 3 tiedot `printf`-käskylle. Manuaalissa on useita osia, mutta ohjelmoijan referenssi on osio 3 eli se kannattaa laittaa käskyyn mukaan tarvittaessa. Esimerkiksi `printf`-käsky löytyy myös osiosta 1, mutta tämä on Linux-käyttäjän opas ja kertoo, miten `printf`-käsky toimii Linuxin komentorivillä. Siksi ohjelmointikäskyä etsiessä tulee joidenkin käskyjen kohdalla nimetä haluttu osia (Kerrisk 2021b).



Kuva 2.1. Unixin man-käsky, ohjelmoijan referenssiopas löytyy osasta 3 (Kerrisk 2021b)

Ohjelmoija löytää man-sivuilta mm. käskyn tarvitseman kirjaston nimen, parametrit ja paluuarvot sekä esim. `printf`-käskyn kohdalla tietotyyppiin liittyvät tunnukset kuvan 2.2 mukaisesti. Tiiviin asiasisällön näkökulmasta näiden man-sivujen kanssa on vaikea kilpailla, sillä esim. Kernighan ja Ritchie (1988) kirja ”The C Programming Language” lähestyy asiaa vähän eri näkökulmasta ja on siten man-sivuja täydentävä lähde.



Kuva 2.2. man-sivuilta löytyy käskyn oleelliset tiedot, esim. `printf`-käskyn formatointikoodit

Linux-käyttäjä voi avata man-sivut terminaalista tai VSC:n terminaalista. Windows:ssa ei ole valmiina man-sivuja, mutta ne löytyvät myös Internetistä, esim. (Kerrisk 2021a) ja ohjelmoijan opas eli osio 3 osoitteessa (Kerrisk 2021b). Muita vastaavia sivustoja on olemassa, joten jokainen voi etsiä itselleen sopivan tietolähteen.

Merkki, merkkijono, merkkitaulukko ja osoite

Merkkijono ja sen käsittely tuntuu usein haastavalta C-kielessä etenkin Pythonin jälkeen. Tiivistetysti merkkijono perustuu C-kielessä merkkeihin ja perusasiat ovat seuraavat:

- **Merkki** vie yhden tavun muistia, byte, joka on tietokoneen muistin perusyksikkö ja siten sitä vastaa yksi muistiosoite. Käytännössä C-kielessä merkin sijaan muistipaikassa on kokonaisluku ja kun merkki tulostetaan, se haetaan ASCII taulukosta käyttäen em. kokonaislukua taulukon indeksinä. Laajennetussa ASCII-taulukossa on 256 merkkiä, joten indeksin kooksi riittää 8 bittiä, sillä 2^8 on 256 ja siten sillä voidaan esittää indeksit 0-255.
- **Merkkijonossa** on monta merkkiä peräkkäin ja sen loppuminen tunnistetaan loppumerkistä. Merkkijono sisältää siis kaikki näkyvät merkit ja sen lisäksi yhden näkymättömän merkin, loppumerkin, joka on tyypillisesti `'\0'` eli NULL-merkki.

- **Merkkitaulukko** varataan merkkijonon tallettamista varten. Esimerkiksi kysyttäessä käyttäjältä nimeä emme voi tietää, kuinka monta merkkiä annettava nimi sisältää ja siksi sille varataan tyypillisesti ”riittävän monta” merkkiä sisältävä merkkitaulukko. Taulukon kokea miettiessä loppumerkkiä varten pitää varata yksi merkki ja jos esim. käyttää `fgets`-funktia merkkijonon lukemiseen, pitää myös rinvaihtomerkillä varata yksi merkki. Suomalaiset nimet mahtuvat tyypillisesti 30 merkin taulukkoon. Taulukon alkiot ovat aina peräkkäisissä muistipaikoissa ja merkistä seuraavaan siirtyminen tapahtuu siirtymällä muistissa yksi tavu eteenpäin.
- **Merkkijonofunktiot** kuten `strlen` löytyvät kirjastosta, jonka saa käyttöön otsikkotiedostolla `string.h`, Nämä funktiot huolehtivat aina loppumerkistä asianmukaisesti.

Seuraava esimerkki käy läpi merkkijonon merkit ja niiden osoitteet yllä olevien periaatteiden mukaisesti.

Esimerkki 2.11. Merkkijonon läpikäynti ja merkkien osoitteet

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

int main(void) {
    int i;
    char nimi[10] = "Ville";

    // Merkkitaulukon alkioden tulostus merkki kerrallaan
    // Ensin merkki ja sitten ko. merkin muistipaikan osoite
    for (i=0; i < strlen(nimi); i++)
        printf("%c - %p\n", nimi[i], &nimi[i]);
    printf("\n");

    return(0);
}
```

Esimerkin tuottama tulos

```
un@LUT8859:~/Opas$ ./E2_11
V - 0x7ffffdfbcb5e
i - 0x7ffffdfbcb5f
l - 0x7ffffdfbcb60
l - 0x7ffffdfbcb61
e - 0x7ffffdfbcb62
```

Kuten yllä olevasta tulosteesta näkyy, merkit tulostuvat aina omille riveilleen ja niiden perässä on jokaisen merkin muistipaikan osoite. Osoitteet ovat peräkkäisten muistipaikkojen osoitteita. Jos ajat tämän ohjelman omalla koneellasi, muistipaikkojen osoitteet ovat jotain muuta, mutta samat periaatteet pätevät ja osoitteet ovat peräkkäisten muistipaikkojen osoitteita.

Osoittimen käyttö merkkijonon läpikäynnissä

Osoittimet ovat monille aloitteleville C-ohjelmoijille kokonaan uusi konsepti ja siksi niitä käsitellään tässä oppaassa useaan otteeseen aina vähän eri näkökulmasta. Tähän asti olemme käyttäneet osoitetta tiedon lukemiseen käyttäjältä `scanf`-käskyllä, jolloin lukutyyppisen-muuttujan osoite saatiin `&`-merkillä ja merkkijonomuuttujan ensimmäisen muistipaikan osoite saatiin muuttujan nimellä *ilman* hakasulkuja tai `&`-merkkiä. Osoitteen lisäksi konseptiin kuuluu *osoitin*,

joka tarkoittaa sitä, että itse tieto ja mistä tuo tieto löytyy, erotetaan toisistaan eli ne ovat kaksi eri asiaa. Ihan niin kuin sinä voit asua eri paikoissa (esim. vanhempien osoite ja opiskeluosoite), mutta Maistraatti pitää yllä sinuun liittyvää osoitetietoa ja sen avulla sinulle lähetetyt kirjeet löytävät perille, kunhan sinä pidät osoitetiedon ajan tasalla.

Edellä kerrattiin merkkijonon rakenne, miten se muodostuu merkeistä ja miten jokaisella merkillä on oma muistipaikka eli osoite. Merkkijono eli merkkitaulukko voidaan käydä läpi taulukon indeksejä hyväksi käyttäen esimerkin 2.11 mukaisesti. Toinen vaihtoehto merkkijonon läpikäyntiin on tehdä osoitin, laittaa se osoittamaan merkkijonon ensimmäiseen alkioon, siirtää osoitin merkistä seuraavaan ja lopettaa tämä sitten kun merkki on loppumerkki. Edellä olevan mukaan ensimmäisen merkin osoite saadaan muuttujan nimestä, kaikki merkit ovat samankokoisissa muistipaikoissa eli 1 tavu kukin ja loppumerkki on NULL. Koska merkkejä tulostetaan nyt niin kauan kunnes merkki on loppumerkki, on tämä selvästi while-silmukka asia ja itse koodi näkyy alla olevassa esimerkissä. Mikäli ohjelmassa on vain yksi osoitin, on sen nimi usein `ptr`, joka on lyhenne englanninkielisestä sanasta *pointer*.

Esimerkki 2.12. Merkkijonon läpikäynti osoittimen avulla

```
#include <stdio.h>
int main(void) {
    /* Merkkijono, loppumerkki ja toistorakenne */
    char str[] = "C-kieli on kivaa.";
    char *ptr;
    int iPituus = 0;

    ptr = str;
    while (*ptr != '\0') {
        printf("%c - %d\n", *ptr, iPituus);
        iPituus++;
        ptr++;
    }
    printf("\nLause on '%s' ja sen pituus on %d merkkiä.\n", str, iPituus);
    return(0);
}
```

Esimerkin tuottama tulos

```
un@LUT8859:~/Opas$ ./E2_12
C - 0
- - 1
k - 2
i - 3
e - 4
l - 5
i - 6
- 7
o - 8
n - 9
- 10
k - 11
i - 12
v - 13
a - 14
a - 15
. - 16
```

Lause on 'C-kieli on kivaa.' ja sen pituus on 17 merkkiä.

Edellä oleva merkkijonon läpikäynti osoittimella on tyypillistä C-koodia. Lähtökohtaisesti tiedot on sijoitettava johonkin muistiin eli tiedoille on varattava niiden tarvitsema tila muistia, mutta usein tietojen läpikäyntiin käytetään osoitinta, koska tällöin itse tietoon ei tarvitse puuttua vaan riittää, kun katsotaan tiedosta kiinnostava osa sopivan osoitin avulla. Osoittimet tekevät mahdolliseksi tiedon sijainnin eri paikoissa muistia eli ne liittyvät läheisesti muistinhallintaan ja kun siirrymme dynaamiseen muistinhallintaan, osoittimet ovat keskeisessä roolissa. Palaamme osoittimien käyttöön jatkossa usein, joten niiden käyttöä kannattaa harjoitella aina tilaisuuden tullen.

Staattinen muistinvaraus

C-kielessä kääntäjä varaa muistin määrittelyvaiheessa, kun muuttujatyypin tarvitsema muistimäärä on tiedossa (esim. `int`, `float`, `char` tai `char [30]`). Samalla kertaa määritellään tuohon muistialueeseen osoittava muuttuja (esim. `iLuku`, `fLuku`, `cMerkki`, `aNimi`). Ohjelman suorituksen aikana käyttöjärjestelmä laittaa muuttujien nimet osoittamaan ohjelmalle varattujen muistialueiden alkuun, jonka jälkeen muuttujia voidaan käyttää ohjelmassa. Muuttujien määrittelyn perusteella käyttöjärjestelmä tietää, kuinka monta tavua kukin muuttuja käyttää, esim. `cMerkki` 1 tavu, `iLuku` 4 tavua ja `aNimi` 30 tavua. Tätä toimintamallia kutsutaan staattiseksi muistivarakseksi ja se tarkoittaa sitä, että muuttuja ja niiden tarvitsema muistimäärä määritellään ohjelmassa eikä niitä voi muuttaa ajon aikana. Tämä tekee ohjelmista kankeita ja C-kielessä onkin mahdollisuus muistin varaamiseen myös ohjelman suorituksen aikana. Tällöin kyseessä on dynaaminen eli ajonaikainen muistinvaraus ja siihen palataan myöhemmin oppaassa.

Pienen C-ohjelman tyyliohjeet

Tämän luvun ohjelmointiosuus keskittyi valinta- ja toistorakenteisiin sekä yksittäisiin käskyihin. Näiden osalta ohjelmoidessa kannattaa noudattaa esimerkeissä näkyviä sisennyksiä jne.

goto:n käyttö kielletty. Rakenteisen ohjelmoinnin periaatteiden mukaisesti hyppiminen ohjelmariviltä toiselle on huonoa ohjelmointityyliä ja siksi **goto:n** käyttö on kielletty tässä oppaassa hyvän ohjelmointityylin nimissä.

Kommentit. C-kielessä on rivikommentit (`//`), normaalit kommentit (`/* ... */`) ja näiden lisäksi koodia voidaan kommentoida pois ehdollisesti esikäsittelijän `#if 0 ... #endif`-lohkona. Näitä kaikki kannattaa käyttää tarpeen mukaan.

Osoittimen nimeäminen. Osoittimen nimenä käytetään usein `ptr`-lyhennettä englanninkielisestä sanasta pointer. Mikäli osoittimia on useita, käytetään tässä oppaassa jatkossa osoittimen tunnistimenä `p`-kirjainta, esim. `pNimi`.

Vakiot. Vakioita voi määritellä sekä esikäsittelijän `define`-käskyllä että C-kielen `const`-määritteellä. Koska `const`-määritellyt vakiot säilyttävät tietotyyppinsä sekä nimensä, helpottaa se vakioiden käyttämistä esim. debuggerissa ja on näin ollen suositeltava ohjelmointityyli ohjelmien kasvaessa.

Yhteenveto

Kerrataan luvun osaamistavoitteet ja esimerkkien aiheet sekä katsotaan keskeiset asiat kokoava ohjelmointiesimerkki.

Osaamistavoitteet

Tämä luvun jälkeen sinun tulee ymmärtää C-kielen valinta- ja toistorakenteet sekä keskeiset esikäsittelijän määritteet. Pystyt määrittelemään vakioita, ottamaan käyttöön olemassa olevia kirjastoja ja kirjoittamaan ohjelmia, joissa on sekä valinta- että toistorakenteita. Läpikäydyt valintarakenteet käsittivät `if` ja `switch` -rakenteet sekä ehdollisen lausekkeen kun taas toistorakenteina käytiin läpi `for`, `while` ja `do-while` -rakenteet. Esikäsittelijän käskyistä tutustuimme `include`, `define` ja `if-end` -käskyihin sekä vakioiden määrittelyyn C-kielen `const` -käskyllä. Lopuksi kävimme vielä läpi kääntäjäoptioita, man-sivut, merkkijonot ja osoite-osoitin konseptit sekä staattisen muistivarauksen. Näiden asioiden hallitseminen mahdollistaa ohjelmien toiminnallisuuden määrän nopean kasvun tästä eteenpäin.

Käsitellyt asiat

Tässä luvussa kävimme läpi C-kielellä perusrakenteiden käyttöä:

- Valintarakenteet: Esimerkki 2.1, 2.2, 2.3, 2.4, 2.13
- Toistorakenteet: Esimerkki 2.5, 2.6, 2.7, 2.13
- Ohjauskäskyt: Esimerkki 2.8, 2.13
- Kirjastofunktioita: Taulukko 11, 2.13
- Vakioiden määrittely: Esimerkki 2.9
- Koodin esikäsittely, preprocessing: Esimerkki 2.10a, 2.10b
- Merkkijonon läpikäynti: Esimerkki 2.11, 2.12, 2.13

Kokoava esimerkki

Kokoava esimerkki on valikkopohjainen ohjelma, joka kysyy käyttäjältä merkkijonon ja tulostaa sen etu- tai takaperin käyttäjän toiveen mukaisesti. Huomaa, että tämä ohjelma toimii perus-ASCII-aulukon arvoilla eli kun ei käytetä laajennettuun ASCII-aulukkoon kuuluvia skandinaavisia merkkejä. UTF-8 -koodaus käsittelee skandinaavisia merkkejä kahdella tavulla, jonka vuoksi takaperin kirjoitettaessa tavut menevät sekaisin eikä tulos ole toivottu. Tämä asia menee tämän kurssin fokuksen ohi, joten emme puutu siihen tämän enempää. Asiasta kiinnostuneet voivat todentaa tilanteen tulostamalla alla olevassa ohjelmassa merkkien (`%c`) sijaan lukuja (`%d`), joka ovat siis lähtökohtaisesti indeksejä ASCII-aulukkoon, mutta UTF-8 koodaus muuttaa tilannetta ja takaperin kirjoitettaessa tulostus menee sekaisin.

Esimerkki 2.13. Valikkopohjainen ohjelma

```

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

int main(void) {
    int i, iValinta;
    char aRivi[30];

    do {
        printf("Valitse alla olevista valinnoista\n");
        printf("1) Syötä rivi\n");
        printf("2) Tulosta rivi etuperin\n");
        printf("3) Tulosta rivi takaperin\n");
        printf("0) Lopeta\n");
        printf("Anna valintasi: ");
        scanf("%d", &iValinta);
        getchar();

        switch (iValinta) {
            case (1):
                printf("Anna rivi (max 28 merkkiä): ");
                fgets(aRivi, 30, stdin);
                aRivi[strlen(aRivi)-1] = '\0';
                break;
            case (2):
                for (i=0; i < strlen(aRivi); i++) {
                    printf("%c", aRivi[i]);
                }
                printf("\n");
                break;
            case (3):
                for (i=strlen(aRivi)-1; i >= 0; i--) {
                    printf("%c", aRivi[i]);
                }
                printf("\n");
                break;
            case (0):
                printf("Kiitos ohjelman käytöstä.\n");
                break;
            default:
                printf("Annoit tuntemattoman valinnan.\n");
                break;
        }
    } while (iValinta != 0);
    return(0);
}
/* eof */

```

Luku 3. Tiedostonkäsittely ja aliohjelmat

Tiedostonkäsittely

Ulkoisia tiedostoja voidaan käyttää ohjelmoinnissa moninaisiin tarkoituksiin, mutta tavallisimmin niitä käytetään tallettamaan ohjelman käsittelemää tietoa tai sille annettuja asetuksia pysyvästi. 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 ja virheenkäsittely.

Tiedostojen käsittely C-kielellä on suoraviivaisempaa kuin Pythonilla. Normaalien ASCII-merkkien lisäksi tiedostoja voidaan helposti käsitellä binaarimuodossa, eikä C-kieli itse asiassa tee merkittävää eroa näiden kahden lähestymistavan välille. Eri käyttöjärjestelmissä on omia erityispiirteitä, mutta toimimalla systemaattisesti ja huolellisesti sekä ASCII- että binaaritiedostot ovat tehokkaita ja ongelmattomia työkaluja.

Tiedoston kirjoittaminen

Tutustutaan ensin tekstitiedostojen kirjoittamiseen. Tavoitteena on luoda tiedosto ja kirjoittaa sinne rivi tekstiä, joten avataan tiedosto kirjoittamista varten, kirjoitetaan sinne merkkijono ja suljetaan tiedosto esimerkin 3.1 mukaisesti.

Esimerkki 3.1. Tiedoston kirjoittaminen

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

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

    tiedosto = fopen("tiedosto.txt", "w");
    printf("Mitä haluat kirjoittaa tiedostoon (max 48 merkkiä)?\n");
    fgets(rivi, 50, stdin);
    rivi[strlen(rivi)-1] = '\0';

    fprintf(tiedosto, "%s\n", rivi);
    fclose(tiedosto);
    return(0);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E3_1
Mitä haluat kirjoittaa tiedostoon (max 48 merkkiä)?
Robottikana
```

Kuinka koodi toimii

Edellä olevan mukaisesti avataan tiedosto kirjoittamista varten, tiedostoon kirjoitetaan yksi rivi tekstiä ja se suljetaan. Tiedoston avaaminen edellyttää tiedoston nimen tietämistä ja meidän tulee kertoa, haluammeko kirjoittaa, write, vai lukea, read, tiedostoa. Itse tiedostoon kirjoittaminen tapahtuu funktiolla `fprintf` (file-printf), joka toimii kuten normaali ruudulle tulostuksen toteuttava `printf`-funktio. Lopuksi tiedosto tulee sulkea `fclose`-käskyllä. Tiedostoon kirjoitettaessa `fprintf`-funktiolle pitää antaa parametrina kirjoitettavan tiedoston tiedostokahva. Nyt käytetty kirjoitustila, "w", luo automaattisesti uuden tiedoston tai tuhoaa aikaisemman samannimisen tiedoston `fopen:n` yhteydessä. `fprintf`-kirjoitti tiedostoon annetun merkkijonon "Robottikana\n" eli em. sanan rivinvaihtomerkin kanssa muttei tiedoston loppumerkkiä tms. (esim. EOF, end of file). Huomaa, että tästä esimerkistä puuttuu tiedoston aukeamisen onnistumisen tarkastus, joka pitää normaalisti aina tehdä. Palataan virheenkäsittelyyn tiedoston lukemisen jälkeen.

Tiedoston lukeminen

Tiedoston kirjoittamisen jälkeen voimme lukea saman tiedoston. Seuraavassa esimerkissä avaaamme edellä luodun tiedoston, luemme sen sisällön ja tulostamme sen ruudulle. Tiedostossa on yhdellä rivillä korkeintaan 48 näkyvää merkkiä ja rivinvaihtomerkki, viimeinen rivi on tyhjä.

Esimerkki 3.2. Tiedoston lukeminen

```
#include <stdio.h>

int main(void) {
    char rivi[50];
    /* Luodaan tiedostokahva eli osoitin FILE-tyyppiseen muuttujaan. */
    FILE *tiedosto;

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

    /* Tiedoston kaikkien rivien luku ja tulostus. */
    while (fgets(rivi, 50, tiedosto) != NULL) {
        printf("%s", rivi);
    }

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

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E3_2
Tiedoston sisältö:
Robottikana
```

Kuinka koodi toimii

Tämä koodi ei poikkea paljoakaan aiemmasta tiedostonkirjoitus-esimerkistä. Ohjelma alkaa määrittelemällä 50 merkin kokoinen merkkitaulukko nimeltä `rivi` sekä tiedostokahva `tiedosto`. Tämän jälkeen avamme haluamamme tiedoston `tiedosto.txt` lukumoodiin `"r"` eli tilaan, josta voimme ainoastaan lukea tiedoston sisältämää tietoa. Tämän jälkeen suoritamme itse tiedoston lukemisen `while`-toistorakenteella. Tiedoston avaaminen voi epäonnistui useista syistä, joten tiedoston avaaminen onnistuminen on aina tarkistettava – katsomme tämän asian seuraavaksi.

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 48 näkyvään merkkiin, jotta se mahtuu `rivi`-muuttujaan rivinvaihtomerkin ja `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 näistä esimerkeistä huomasimme, on C-kielen tapa käsitellä tiedostoja hyvin samankaltainen kuin Pythonissa. Tiedosto avataan tarpeita vastaavalla moodilla tiedostokahvaan, tiedostoon kohdistetaan kirjoitus-/luku-operaatiot, ja kun tiedostoa ei enää tarvita, se suljetaan. 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 edellisen esimerkin mukaisesti.

Tiedostonkäsittelyn virheentarkastus

C-kielessä tyypillinen virheenkäsittely on tarkistaa funktion palauttama arvo ja mikäli se on virhekoodi, käsitellään virhetilanne. Funktioiden palauttavat virhekoodit löytyvät manuaalisivuilta, mutta osoittimen tapauksessa virhekoodina toimii tyypillisesti osoittimen arvo `NULL`.

Tiedostojen käsittelyn yhteydessä tässä oppaassa riittää tarkistaa tiedoston avaamisen onnistuminen. Tämä on tyypillisin virhekohta ja siksi sen onnistuminen on tarkistettava aina. Samoin tässä oppaassa virheen tapahtumisesta kerrotaan käyttäjälle ja lopetetaan ohjelman suoritus sen jälkeen. Virheestä voi kertoa käyttäjälle `printf`-käskyllä, mutta käyttämällä `perror`-funktiota, `print-error`, voidaan virheilmoitus muodostaa omasta virheilmoituksesta ja sitä seuraavasta käyttöjärjestelmän virheilmoituksesta, joten sitä kannattaa opetella käyttämään.

Tässä oppaassa ei käydä läpi kokonaisvaltaista virneenhallintaa vaan lähtökohta on tunnistaa virhetilanne, kertoa käyttäjälle ongelma ja lopettaa ohjelman suoritus hallitusti `exit`-käskyllä. Käsky löytyy `stdlib.h` otsikkotiedostosta ja alla olevasta esimerkistä näkyy tässä oppaassa suositeltava tiedoston avaamiseen liittyvä virheenkäsittely. `exit`-käskyn parametri tulee päättää kokonaisvaltaisen virheenkäsittelyn näkökulmasta, mutta tällä kurssilla se on 0 selkeyden vuoksi.

Esimerkki 3.3. Tiedoston avaamisen virheenkäsittely

```
if ((tiedosto = fopen("tiedosto.txt", "w")) == NULL) {
    perror("Tiedoston avaaminen epäonnistui");
    exit(0);
}
```

Binaaritiedostojen käsittely

Binaaritiedostojen käsittely noudattaa samoja periaatteita kuin tekstitiedostojen käsittely eli tiedosto on avattava ennen käsittelyä, suljettava käsittelyn jälkeen ja tiedoston avaamisen yhteydessä on varmistettava 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). Esimerkissä 3.4 on tiedon kirjoittamiseen ja lukemiseen käytettävät `fwrite`- ja `fread`-funktiot `tiedosto`-nimisen `FILE*` -muuttujan kanssa. Ensimmäinen toistorakenne kirjoittaa tiedostoon luvut 0-9 binaarimuodossa ja seuraava lukee binaaritiedostosta 10 kokonaislukua sekä tulostaa ne näytölle. Molemmissa funktiokutsuissa kolmas parametri on 1, joka kertoo käsiteltävien alkiodien määrän eli jokainen funktiokutsu käsittelee yhtä alkiota kerrallaan.

Esimerkki 3.4. Binaaritiedoston kirjoitus ja luku

```
int i, luku;

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);
}
```

Tiedostonkäsittelyn perusfunktioita

Tässä luvussa esitellään tärkeimmät C-kielen tiedostonkäsittelyyn liittyvät funktiot.

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" tai "wb"). Haluttaessa lukea ja kirjoittaa samanaikaisesti, lisätään + ("r+"). Lisäysmoodia käyttävän on syytä tietää mitä tekee, ettei tiedosto mene sekaisin.
- Huomaa, että avattaessa tiedosto + -moodissa **r:n** ja **w:n** merkitys muuttuu. Käytettäessä **r:ää** avataan olemassa oleva tiedosto tai luodaan uusi. **w:llä** luodaan aina uusi tiedosto ja jos tiedosto oli olemassa, se tuhoutui.

fclose(tietovirta);

- Sulkee tiedoston *tietovirta*.

fscanf(tietovirta, formaattimerkkijono, muuttujalista);

fprintf(tietovirta, formaattimerkkijono, muuttujalista);

- `fscanf` lukee tiedostoa *tietovirta* ja `fprintf` kirjoittaa sinne. Muuten näiden funktioiden toiminta on vastaava kuin `printf` ja `scanf`.

fgets(merkkitaulukko, merkkien enimmäismäärä, tietovirta);

fputs(merkkitaulukko, tietovirta);

- `fgets` lukee tietovirrasta rivin tai merkkien enimmäismäärän verran merkkejä – 1 ja sijoittaa ne merkkitaulukoon. `fputs` kirjoittaa merkkitaulukon tietovirtaan.
- Luettaessa käyttäjältä syötteitä `fgets`'llä pitää merkkitaulukossa olla tilaa merkkijonon loppumerkille sekä rivinvaihtomerkille käyttäjän antamien merkkien lisäksi.

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 muistipaikkaan *osoitin*.
 - `fwrite` kirjoittaa *tietovirtaan* muistipaikassa *osoitin* olevan tiedon, jota on *koko*-muuttujan kokoisissa lohkoissa *määrä*-kappaletta.

Alla on muutama muu hyödyllinen tiedostonkäsittelyfunktio. Niihin tarkempia tietoja ja käyttöohjeita löytyy man-sivuilta.

- `fgetc(tietovirta);`
- `fputc(merkki, tietovirta);`
- `rewind(tietovirta);`
- `ftell(tietovirta);`
- `fseek(tietovirta, sijainti, alku);`
- `feof(tietovirta);`
- `remove(tiedosto);`
- `rename(nimi, uusi_nimi);`

Aliohjelmat

Aina ohjelmia tehtäessä joudumme väistämättä miettimään ohjelman toimintaa. Useimmiten hyötyisimme mahdollisuudesta käyttää uudelleen aiempaa koodia useaan kertaan toistuvien tehtävien suorittamiseen ja toisinaan tehtävien jakaminen loogisiin kokonaisuuksiin helpottaisi sekä 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 osoittimia. Käydään seuraavaksi nämä asiat läpi C-kielen kontekstissa.

Määrittely ja kutsuminen

Kuten tähän mennessä on tullut selväksi, 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.

C-kielessä funktion paluuarvon tyyppi pitää päättää etukäteen. Jos haluamme ohjelman palauttavan kokonaisluvun, on funktion tyyppi `int`, ja jos palautamme liukuluvun niin tyyppi on `float`. Jos taas haluamme palauttaa merkin, käytämme tyyppiä `char`. Jos taas emme aio palauttaa mitään arvoja, on funktion tyyppiksi laitettava tyhjä eli `void`. Muista siis, että `void`-tyyppinen aliohjelma ei voi palauttaa mitään kutsuvaan ohjelmaan ja funktion, jolla on jokin ei-tyhjä tyyppi, pitää palauttaa jotain. Myös `main`-funktion tyyppiksi voi laittaa `void` ja tällöin sekään ei palauta minkäänlaista arvoa käyttöjärjestelmälle ohjelman lopettaessa toimintansa. Näin ei kuitenkaan kannata tehdä, sillä käyttöjärjestelmä odottaa `main`-funktiolta paluuarvoa. Ohjelman loppuminen osoitetaan `return`-käskyllä. Mikäli ohjelma ei palauta mitään, käytetään `return;` -muotoa ilman sulkuja ja muussa tapauksessa käytetään `return (paluuarvo);` -muotoa, jossa parametri `paluuarvo` palautetaan kutsuvaan ohjelmaan tyypillisesti sijoitettavaksi muuttuun.

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

Esimerkki 3.5. Aliohjelman määrittely, esittely ja kutsu

```
#include <stdio.h>

void demo(void); /* Aliohjelman esittely */

int main(void) {
    printf("Tämä tulee pääohjelmasta!\n");
    demo(); /* Aliohjelman kutsu */
    return(0);
}

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

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E3_5
Tämä tulee pääohjelmasta!
Tämä tulee aliohjelmasta!
```


Kuinka koodi toimii

Ohjelmakoodi alkaa aiemmin mainitulla aliohjelman esittelyllä, joka tarkoittaa käytännössä sitä, että kääntäjälle kerrotaan ohjelman alussa aliohjelman paluuarvon tietotyyppi, aliohjelman nimi ja parametrien tietotyypit. Käytännössä aliohjelman esittely vastaa sen määrittelyn ensimmäistä riviä puolipistettä lukuun ottamatta.

Koodin lopussa on itse aliohjelman määrittely. Aliohjelma on rakenteellisesti samanlainen kuin pääohjelma, eli se toteutetaan ohjelman nimen ja parametrien jälkeen tulevien aaltosulkeiden sisällä. Tässä koodissa olemme määritelleet aliohjelman nimeltä `demo`, joka ei palauta arvoa (ensimmäinen `void`) eikä vastaanota parametreja (sulkeissa oleva toinen `void`). Aliohjelma loppuu `return`-käskyyn ja koska siinä ei ole parametria, tämä aliohjelma ei palauta mitään kutsuvaan ohjelmaan. 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 aliohjelmalle välitetään.

Arvoparametrit ja paluuarvo

Aliohjelmalle voidaan välittää tietoa kuten yhteenlaskettavia lukuja tai tutkittavia merkkejä parametreina. Aliohjelman vastaanottamat parametrit tulee nimetä 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 luku1, int luku2);
```

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

Huomaa, että tällä tavoin määriteltyjen parametrien arvoihin tehdyt muutokset ovat paikallisia aliohjelmassa eivät välity takaisin kutsuvaan ohjelmaan ja siksi niitä kutsutaan *arvoparametreiksi*. Käytännössä arvoparametri tarkoittaa sitä, että aliohjelmassa käsitellään alkuperäisen argumentin kopioita. C-kielessä on olemassa tapa myös parametrien arvojen muuttamisen aliohjelmassa ja palaamme siihen kohta *muuttujaparametrien* yhteydessä.

Paluuarvot ovat tyypillinen tapa palauttaa tietoa funktiosta. C-kielessä paluuarvoja voi olla vain yksi ja aliohjelman määrittely kertoo sen tietotyypin. Yllä oleva `laskukone`-esimerkki näytti tyypillisen tavan käyttää arvoparametreja tiedon viemiseen aliohjelmaan sekä tiedon palauttamiseen aliohjelmasta paluuarvolla kutsuvaan ohjelmaan. Usein tieto on yksinkertainen luku, tai esim. virhekoodi, tai sitten osoite johonkin muistipaikkaan. Tyypillinen esimerkki tästä on edellä käsitelty tiedoston avaaminen, joka palautti osoittimen `FILE`-rakenteeseen. Osoittimien palauttamiseen palaamme myöhemmissä luvuissa, mutta kaikki tiedot palautetaan `return`-käskyllä samalla tavoin kuin `main`-ohjelmassa. Ainoa ero on, että pääohjelman rajapintana on käyttöjärjestelmä, joten se on ohjelman ulkoinen rajapinta eikä sitä voi muuttaa. Jos palautettua arvoa haluaa käyttää myöhemmin, pitää se sijoittaa muuttujan arvoksi kutsun yhteydessä.

Esimerkki 3.6. Parametrit ja paluuarvot

```
#include <stdio.h>

int summaa(int luku1, int luku2); /* Funktion esittely */

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);
}

int summaa(int luku1, int luku2) { /* Funktion määrittely */
    int tulos;
    tulos = luku1 + luku2;
    return tulos;
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E3_6
Parametrien 1 ja 2 summa on 3.
```

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 lasketaan yhteen ja paluuarvo palautetaan takaisin pääohjelmalle. Lopputuloksena saamme muuttujaan `vastaus` laskutoimituksen tuloksen. Huomaa, että tällä kertaa funktion tyyppiä 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 samassa järjestyksessä, kun niitä kutsutaan ohjelmassa. Ohjelmien monimutkaistuessa ja koon kasvaessa tämä ei aina onnistu, mutta tämä rakenne tarjoaa loogisen lähtökohdan koko ohjelman rakenteen ymmärtämiselle.

Muuttujaparametrit eli osoittimet aliohjelmissa

C-kielessä aliohjelman parametrit voivat olla joko *arvoparametrejä* tai *muuttujaparametrejä*. Olemme tähän asti puhuneet arvoparametreistä, jotka ovat kopioita alkuperäisistä parametreistä ja

joihin aliohjelmassa tehdyt muutokset eivät välity takaisin kutsuvaan ohjelmaan. Arvoparametrit sopivat hyvin esim. tulostettaessa arvoja `printf`-funktiolla. Aina tämä ei kuitenkaan riitä vaan parametrien arvoja on pystyttävä muuttamaan aliohjelmassa niin kuin esimerkiksi `scanf`-funktio tekee. Välittämällä aliohjelmaan muuttujaparametri eli alkuperäisen muuttujan osoite, saadaan muuttujan uusi arvo sijoitettua kutsuvassa ohjelmassa varattuun muistiin ja näin muuttujan arvo on tallessa myös aliohjelman jälkeen. Siksi `printf`-aliohjelmalle voidaan välittää esim. muuttuja luku, mutta kun luetaan uusi arvo muuttujalle `scanf`-funktiossa, pitää sinne välittää muuttujan osoite eli `&luku`.

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, kun taas osoittimet ovat 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.

Osoittimen tunnistaa *-merkistä muuttujan nimen edessä. Tämä kertoo kääntäjälle, ettei kyseessä ole normaali muuttuja vaan osoitin muuttujaan. Yleisesti ottaen *-etumerkki voidaan 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 katsotaan esimerkki osoittimen käytöstä.

Esimerkki 3.7. Osoittimen määrittely ja käyttö

```

#include <stdio.h>

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

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

int main(void) {
    int iLuku1=1, iLuku2=2; /* Määritellään 2 kokonaislukumuuttujaa */

    /*Luodaan kaksi kokonaislukuosoitinta eli osoitinta kokonaislukuun*/
    int *pLuku1;
    int *pLuku2;
    /* Asetetaan osoittimet muuttujiin iLuku1 ja iLuku2 */
    pLuku1 = &iLuku1;
    pLuku2 = &iLuku2;

    printf("Osoitin 1 viittaa muistipaikkaan %p, jossa on arvo %d.\n",
           pLuku1, *pLuku1);
    printf("Muuttujan iLuku2 arvo on alussa %d.\n", iLuku2);

    /* Kutsutaan funktiota ei_muuta. */
    ei_muuta(iLuku2);
    printf("Funktion ei_muuta jälkeen iLuku2 arvo on %d.\n", iLuku2);

    /* Kutsutaan funktiota muuttaa. */
    muuttaa(&iLuku2);
    printf("Funktion muuttaa jälkeen iLuku2 arvo %d.\n", *pLuku2);

    return(0);
}

```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```

un@LUT8859:~/Esimerkit$ ./E3_7
Osoitin 1 viittaa muistipaikkaan 0x7fffc581a260, jossa on arvo 1.
Muuttujan iLuku2 arvo on alussa 2.
Funktion ei_muuta jälkeen iLuku2 arvo on 2.
Funktion muuttaa jälkeen iLuku2 arvo 1000.

```

Kuinka koodi toimii

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

Itse ohjelmakoodissa luomme kaksi osoitinta, `pLuku1` ja `pLuku2`, ja asetamme ne osoittamaan muuttujien `iLuku1` ja `iLuku2` käyttämiin muistiosoitteisiin, joissa on meitä kiinnostavat muuttujien arvot. Tämän jälkeen testaamme osoittimien toimintaa. Ensinnäkin tulostamme hieman tietoja osoittimesta: näemme, että `pLuku1` osoittaa muistipaikkaan numero `0x7ffc581a260`, johon on tallennettuna arvo `1`. Huomaa erityisesti, että osoitinta voidaan käyttää tiedon tulostamiseen normaalisti muuttujan tavoin. Lisäksi tulostamme muuttujan `iLuku2` arvon alussa eli `2` näytölle.

Seuraavaksi kokeilemme oikeasti osoittimien eroja. Ensin kutsumme proseduuria `ei_muuta`, joka toimii kuten aiemmassa esimerkissä huomasiimmekin eli parametrin arvo ei muuttunut. Kuinka ohjelma voidaan muuttaa siten, että aliohjelmassa muutettu arvo saadaan käyttöön kutsuvassa ohjelmassa? Ensinnäkin määritellään muuttaa-proseduurin parametrin tyypiksi osoitin. Tämän takia kutsuessamme proseduuria, emme annakaan muuttujan `iLuku2` arvoa kuten aiemmin, vaan lähetämme `&`-merkin avulla muuttujan muistipaikan osoitteen, eli käytännössä siis osoittimen `iLuku2`-muuttujan käyttämään muistipaikkaan. Tämän seurauksena aliohjelma muuttaa sen muistipaikan arvoa, johon pääohjelman `iLuku2` on tallennettu, ja tämän vuoksi muutos jää voimaan.

Pääohjelmassa meillä on vielä viimeinen tulostuskäsky. Tulostuskäskyssä on taas muoto `*pLuku2`, joka tarkoittaa osoitteessa `pLuku2` olevan muuttujan arvoa. Koska muutimme muistipaikassa ollutta arvoa edellisessä aliohjelmassa, löytyy osoittimen `pLuku2` ilmoittamasta paikasta nyt arvo `1000`.

Aliohjelmissa voidaan siis muuttaa muuttujaparametrien arvoja. Käytännössä tällöin parametrinä lähetetään arvon sijasta muuttujan muistipaikan osoite, jotta tietoa voidaan muuttaa samalla tavoin kuin esim. `scanf`-funktio tekee. Ja jos parametrien ei haluta muuttuvan, käytetään arvoparametrejä `printf`-funktion tyyliin, jolloin aliohjelmassa käytetään alkuperäisten arvojen kopioita.

Tunnusten näkyvyys

Pythonin yhteydessä oli puhetta nimiavaruudesta ja nyt katsomme, miten vastaava asia on toteutettu C-kielillä. Tunnusten näkyvyys tarkoittaa sitä, että ohjelman tunnukset (muuttujat, aliohjelmat, vakiot) 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 vaikuta pääohjelman samannimisiin muuttujiin. Katsotaan ensin esimerkki ja käydään asia läpi tarkemmin sen jälkeen.

Esimerkki 3.8. Esimerkki tunnusten näkyvyydestä

```
#include <stdio.h>

int testi(int syote) { /* Huomaa: muuttujaa syote ei käytetä! */
    int luku;
    luku = 100;
    printf("Aliohjelmassa 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);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E3_8
Alussa muuttujan 'luku' arvo on 1.
Aliohjelmassa luku-muuttuja on 100.
Aliohjelma ei vaikuttanut pääohjelman muuttujaan.
Pääohjelman muuttujan 'luku' arvo on edelleen 1.
```

Kuinka koodi toimii

Tällä kertaa loimme sekä pää- että aliohjelmaan muuttujan nimeltä `luku`. Annamme 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, numeroarvon 1, emme muuttujaa `luku`, tai mitään muuta siihen liittyvää. Tämä arvo 1 on käytettävissä aliohjelmassa parametrinä olevan `syote`-muuttujan 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 mihinkään, 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);
```

Olisimme myös voineet käyttää globaaleja muuttujia (ks. lisätietoa Alaoutinen (2005) tai Kernighan ja Ritchie (1988)) tai käyttää osoittimia eli muuttujaparametreja, joista puhuimme edellisessä 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 siksi niiden käyttö on tyypillisesti kiellettyä. Myös tämän oppaan kanta on sama eli globaaleja muuttujia ei tule käyttää.

Globaalit *muuttujat* ovat huonoa ohjelmointityyliä, mutta *vakioiden* toteuttaminen globaaleina on hyvää ohjelmointityyliä. 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.

C-kielen ominaispiirteitä

Kerrataan lopuksi muutamia keskeisiä asioita niin C-ohjelmoinnin kuin Unixinkin kannalta. Katsotaan ensin tiedon välitystä aliohjelmiin ja sieltä takaisin, mietitään Unixin tietovirta-konseptia vähän tarkemmin, käydään läpi tyypillisiä ongelmia tiedostoformaatteihin liittyen, tutustutaan C-kielen makroiin ja kerrataan kirjastofunktioiden avainkohdat.

Tiedonvälitys aliohjelmaan ja takaisin kutsuvaan ohjelmaan

Tiedonvälitys aliohjelmiin voi tuntua haastavalta, mutta kaikessa yksinkertaisuudessaan tieto aliohjelmiin siirretään aina parametrien välityksellä. Aliohjelmista tieto kutsuvaan ohjelmaan siirretään paluuarvona ja toinen vaihtoehto on käyttää muuttujaparametreja.

Perustietotyypit kuten `int`, `float` ja `char` ovat selkeitä käyttää arvo- ja muuttujaparametreinä sekä paluuarvoina, koska näiden kohdalla käsitellään aina yhtä arvoa tai yhtä muistipaikkaa. Arvoparametri on alkuperäisen muuttujan kopio ja myös paluuarvo on näiden tietotyyppien kohdalla tyypillisesti itse muuttuja eli sen arvo. Siksi kyseessä on siis *arvoparametri*. Muuttujaparametrin kohdalla käsitellään muuttujan arvon sijasta sen muistipaikan osoitetta ja aliohjelmaan lähetään muuttujan `luku` sijasta sen osoite eli `&luku`. Aliohjelman kannalta parametri on osoitinmuuttuja, esim. `int *osoitin`. Muuttujan osoite toimii myös paluuarvona, mutta perustietotyyppien kanssa se on tyypillisesti tarpeetonta ja siksi niiden kohdalla palautetaan tyypillisesti arvo.

Rakenteinen tietotyyppi tarkoittaa useita tietoalkioita sisältävää tietorakennetta kuten esimerkiksi useista merkeistä muodostuvaa merkkitaulukkoa `char nimi[30]`. C-kielessä rakenteisista tietotyypeistä aliohjelmaan välitetään aina rakenteen ensimmäisen alkion osoite, joten kyseessä on aina muuttujaparametri. Merkkitaulukon yhtä alkioita eli merkkiä voidaan käsitellä `nimi[0]` -merkinnällä, jolloin voimme tulostaa ko. *merkin* näytölle tai asettaa sen arvon, esim. `nimi[0] =`

'K'. Jos lähetämme merkkitaulukon aliohjelmaan merkkijonon lukemista varten, välitetään aliohjelmaan taulukon osoite, joka on taulukon ensimmäisen merkin osoite ja parametrinä on tällöin muuttujan nimi, esim. `nimi`. Aliohjelman kannalta parametri on osoitin merkkijonoon, esim. `char *pNimi`, eikä aliohjelma tiedä tai pysty selvittämään varatun merkkitaulukon kokoa. Siksi koko pitää tarvittaessa välittää aliohjelmalle parametrinä samalla tavoin kuin esim. `fgets`-funktiossa. `fgets`-funktion parametrit ovat osoitin merkkijonoon, merkkijonon pituus ja luettava tietovirta, esim. `fgets(rivi, 80, stdin)`. Muuttujaparametrinä aliohjelmaan välitettyyn merkkitaulukkoon voidaan lukea merkkijono ja käyttää sitä sen jälkeen kutsuvassa ohjelmassa.

Paluuarvon kohdalla on muistettava sen sijoitus muuttujan arvoksi. Perustietotyyppien kohdalla asia on selkeä, sillä aliohjelmassa on tyypillisesti määritelty palautettava muuttuja, tai palautettava arvo lasketaan `return`-lauseessa. Tällöin kääntäjä huolehtii siitä, että paluuarvo saadaan sijoitettua muuttujan arvoksi ennen kuin aliohjelman käyttämä muisti vapautetaan ja samalla sen paikallisten muuttujien arvot häviävät, esim. `arvo = funktio(x, y)`.

Paluuarvon ollessa rakenteinen muuttuja kuten merkkitaulukko, palautetaan aliohjelmasta tyypillisesti rakenteisen muuttujan osoite eli esim. merkkijonon osoite yllä olevan mukaisesti. Staattisessa muistinvarauksessa palautettavien arvojen muisti on varattava kutsuvassa ohjelmassa ja välitettävä aliohjelmaan muuttujaparametrinä samalla tavoin kuin `fgets`- ja `scanf`-funktioiden kanssa. Myös merkkijonon kopiointifunktio `strcpy` toimii samoin eli se saa parametrina kahden merkkitaulukon osoitteet, esim. `char s1[30], s2[30]`. `strcpy` kopioi merkkijonon taulukosta toiseen ja palauttaa uuden merkkijonon osoitteen paluuarvona, joka mahdollistaa uuden merkkijonon helpon ja nopean jatkokäsittelyn aliohjelman jälkeen. Molempien taulukoiden muistitila on kuitenkin varattava kutsuvan ohjelman puolella. Kuten myöhemmin näemme, dynaaminen muistinhallinta tarjoaa tälle toisen vaihtoehdon, mutta tässä vaiheessa staattisen muistinvarauksen kanssa tälle menettelylle ei ole vaihtoehtoja.

Esimerkissä 3.9 näkyy tyypilliset tapaukset arvo- ja muuttujaparametreista sekä paluuarvoista.

Esimerkki 3.9. Parametrit ja paluuarvot

```
#include <stdio.h>
#include <string.h>
#define LEN 80

// Perustietotyyppi int, float, char arvoparametrinä ja paluuarvona
int summaa(int iLuku1, int iLuku2) {
    int iSumma = iLuku1 + iLuku2;
    return(iSumma);
}

// Merkkijono parametrinä ja perustietotyyppi paluuarvona
int kysyLuku(char *pPrompti) {
    int iNumero;
    printf("%s: ", pPrompti);
    scanf("%d", &iNumero);
    getchar(); // Ei jätetä rivinvaihtoa väijymään puskuriin
    return(iNumero);
}

// Merkkijono muuttujaparametrinä ja osoite paluuarvona, rakenteinen tietorakenne
char *kysyNimi(char *pPrompti, char *pNimi) {
    printf("%s: ", pPrompti);
    scanf("%s", pNimi);
    getchar(); // Ei jätetä rivinvaihtoa väijymään puskuriin
    return(pNimi);
}
```


C-ohjelmointiopas versio 2.1

```
// Merkkijono muuttujaparametrina ja osoite paluuarvona, rakenteinen tietorakenne
char *kysyRivi(char *pPrompti, char *pRivi) {
    printf("%s: ", pPrompti);
    fgets(pRivi, LEN, stdin);
    pRivi[strlen(pRivi)-1] = '\0';
    return(pRivi);
}

// Merkkijonon kopiointi, tila varattu kutsuvassa ohjelmassa, paluuarvona osoitin
char *kopioi(char p1[], char p2[]) {
    int i = 0;
    do {
        p1[i] = p2[i];
        i++;
    } while (p1[i-1] != '\0');
    return(p1);
}

int main(void) {
    int iIka, iPituus, iNro1 = 1, iNro2 = 2;
    char aNimi1[30], aNimi2[30], aRivi[LEN];
    printf("Lukujen %d ja %d summa on %d.\n", iNro1, iNro2, summaa(iNro1, iNro2));
    iIka = kysyLuku("Anna ikä");
    printf("Ikä on %d.\n", iIka);
    iPituus = kysyLuku("Anna pituus");
    printf("Pituus on %d.\n", iPituus);
    kysyNimi("Anna etunimi", aNimi1);
    kysyNimi("Anna sukunimi", aNimi2);
    printf("Etunimi on %s ja sukunimi on %s.\n", aNimi1, aNimi2);
    kopioi(aNimi1, aNimi2);
    printf("Kopioinnin jälkeen nimi 1 on '%s' ja 2 '%s'.\n", aNimi1, aNimi2);
    kysyRivi("Anna rivi", aRivi);
    printf("Annoit rivin '%s'.\n", aRivi);
    return(0);
}
```

Tietovirrat ja tiedostot Unixissa

Unix-järjestelmä on kehitetty ennen graafisia käyttöliittymiä ja siksi siinä on useita komentorivikäyttöliittymän ja tekstitiedostojen ominaispiirteitä. Esimerkiksi ohjelmat kirjoittavat tulosteensa lähtökohtaisesti näytölle ja käyttöjärjestelmä tarjoaa mahdollisuuden näytölle kirjoitettavien tietojen ohjaamiseen tiedostoon yksinkertaisesti >-merkillä. Näin ohjelman tulosteet saadaan tiedostoon kirjoittamalla komentoriville suoritettavan ohjelman nimi ja tulosten ohjaus tiedostoon, esim. ohjelma > tulokset.txt. Tulosteiden tiedostoon-ohjauksen käänteisoperaationa voidaan pitää syötteiden lukemista tiedostosta ja tämä on toinen Unixin perustoiminto, esim. käsky ohjelma < syöte.txt saa ohjelman lukemaan syötteet tiedostosta näppäimistön sijasta. Näiden jälkeen seuraava looginen askel on ohjata yhden ohjelman tulosteet suoraan toisen ohjelman syötteiksi, joten ei ole yllättävää, että tämä on Unixissa *putki*-niminen perusominaisuus ja kirjoittamalla komentoriville ohjelma1 | ohjelma2 päätyvät ohjelma1:n tulosteet ohjelma2:n syötteiksi.

Unix-järjestelmässä tiedostot on ajateltu laajemmin *tietovirta*-konseptina, jossa näppäimistöltä tulevat syötteet, näytölle menevät tulosteet ja tiedoston luku/kirjoitus kaikki ajatellaan tietovirtana eli englanniksi *stream*. Käytännössä näppäimistöltä lähtee ASCII-merkkejä tietokoneelle, tietokoneelta lähtee ASCII-merkkejä näytölle ja tiedostot sisältävät ASCII-merkkejä, joten nämä kaikki voidaan ajatella teknisesti samanlaisina tietovirtoina. Unixissa on kolme perustietovirtaa: *stdin* eli syöttö, *stdout* eli tuloste ja *stderr* eli virhetietovirta. Olemme käyttäneet tietovirtoja

jo tämän oppaan alusta alkaen, sillä luimme tietoja jo luvussa 1 `fgets`-funktioilla, jonka parametrit olivat merkkitaulukko, luettavien merkkien määrä ja tietovirta. Tällöin saimme luettua tiedot näppäimistöltä käyttämällä `stdin`-tietovirtaa. Nämä Unix-maailman perusominaisuudet liittyvät läheisesti C-ohjelmointiin, joten niihin kannattaa tutustua tarkemmin tarpeen ja mielenkiinnon mukaan Unix-lähteiden avulla `man`-sivuja unohtamatta.

Tiedostoformaateista

C-kielillä on helppo kirjoittaa ja lukea sekä teksti- että binaaritiedostoja. Vaikka itse toiminnot ovat selkeitä, on tiedostojen kanssa ajoittain erilaisia ongelmia. Siksi katsotaan muutamia yleisimpiä binaari- ja tekstitiedostojen ongelmia, jotta voit lähteä selvittämään niitä tarpeen tullen.

C-kielessä binaaritiedostoon kirjoitetaan yksinkertaisesti muuttujien käyttämä muisti bittikuviona eli arvoilla 0 ja 1. Näin ollen C-ohjelmalla kirjoitetun tiedoston lukemisen lähtökohta on tietää, mitä tiedostoon on kirjoitettu ja lukea kirjoitetut bitit identtisiin muuttujiin. Toisin sanoen, jos kirjoitit tiedostoon kolme tietoaikiota, 1 tavun merkin, 4 tavun kokonaisluvun, 8 tavun desimaaliluvun sekä 1 tavun merkin, niin nämä 14 tavua voidaan lukea kerralla muistiin, mutta ne on sijoitettava samankokoisiin ja tyyppisiin muuttujiin, mikäli halutaan saada selville mitä tiedostoon oli kirjoitettu. Ja jos itse kirjoitetun binaaritiedoston luku ei onnistu, voi lähinnä varmistua siitä, että tiedostoon on kirjoitettu samat tiedot samassa järjestyksessä kuin mitä yritetään lukea. Usein toisella ohjelmalla tai ohjelmointikielellä kirjoitetun binaaritiedoston lukeminen ei onnistu, sillä binaaritiedostojen toteutuksissa on merkittäviä eroja eri ohjelmointikielien välillä samoin kuin tietoaikioiden tietotyypeissä on myös helposti eroja Taulukon 1 mukaisesti.

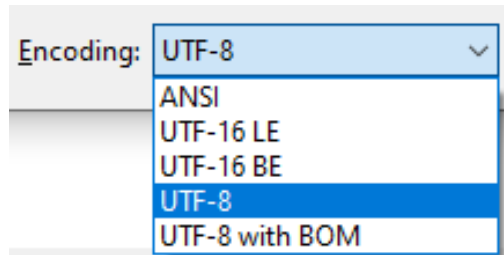
Tekstitiedostojen peruseriaatteet ovat samantyyllisiä kaikissa ohjelmointikielissä, sillä niitä pitää pystyä lukemaan ja kirjoittamaan ristiin eri kielillä. Tiedostomuotona CSV eli Comma Separated Values on tyypillinen datan siirtoon käytetty tekstitiedostoformaatti ja lähtökohtaisesti kaikilla ohjelmointikielillä pystyy lukemaan ja kirjoittamaan niitä. Kyseessä on tekstitiedosto, jossa tietoaikiot erotetaan toisistaan vakio-merkillä, tyypillisesti pilkulla. Mikäli pilkkua käytetään desimaalierottimena Suomen tapaan, on puolipiste tyypillinen vaihtoehto sille.

Tekstitiedostoissa on usein käyttöjärjestelmäkohtaisia eroja ja erityisesti erikoismerkkien toteutukset voivat poiketa eri järjestelmissä. Ääkköset luetaan tässä kohdin erikoismerkeiksi ja ne voivat siis edelleen aiheuttaa erilaisia ongelmia erityisesti käyttöjärjestelmätasolla. Windows'ssa ääkköset näyttävät toimivan hyvin ja Pythonkin lupaa, että ääkkösiä voi käyttää ohjelmien tunnuksissa, mutta usein tämä johtaa kuitenkin ongelmiin aina joissain kohdin ja siksi tätä ei voi suositella, jos tavoitteena on ongelmien välttäminen. Windows'ssa tekstitiedostot päättyvät aina EOF-merkkiin, End Of File, mutta Unixissa tällaista käytäntöä ei ole.

Unix-Windows -yhteiskäytössä törmää ajoittain rivinvaihtomerkki-ongelmiin, sillä sekä Unix että Windows ympäristöissä on rivinvaihtomerkki `'\n'`, mutta se toimii eri tavoin eri ympäristöissä. Unixissa rivi vaihtuu seuraavalle riville ja tulostuskohta palautuu rivin ensimmäisen merkin kohdalle yhdellä `'\n'`-merkillä. Windows-ympäristössä nämä kaksi operaatiota on pidetty erillään eli rivinvaihtomerkki koostuu kahdesta osasta – rivin vaihtaminen, new line, `\n`, sekä cursorin palautus rivin alkuun, carriage return, `\r`. Tämän toimintatapaeron vuoksi ohjelmien ja tekstitiedostojen siirtäminen ympäristöstä toiseen voi johtaa ongelmiin, vaikka kaikki toimisi hyvin yhdessä ympäristössä. Rivinvaihtomerkkiin liittyvät ongelmat ovat vähentyneet aikojen kuluessa, mutta niihin saattaa edelleen törmätä ja siksi tämä mahdollisuus on hyvä muistaa, jotta voi tarpeen tullessa tarkistaa ja korjata tilanteen sopivalla tavalla.

Yksi tyypillinen ongelma tekstitiedostojen kanssa on niiden koodaus, encoding. Kuten Pythonin yhteydessä oli puhetta, tiedostot on koodattu ja siihen on useita vaihtoehtoja. Nykyään tyypillinen koodaus on UTF-8, mutta monen ohjelman oletus on edelleen ANSI-koodaus. Näiden lisäksi jotkin

ohjelmat laittavat tiedoston ensimmäiseksi tavuksi BOM-merkin eli Byte Order Mark'n (U+FEFF). Tämä merkki luonnollisesti häiritsee tiedoston käyttöä, jos lukeva ohjelma ei osaa käsitellä sitä. Alla on kuva Win10 -käyttöjärjestelmän Notepad -ohjelman tiedoston tallennusoptioista ja erityisesti sen tukemista koodausvaihtoehdoista.



Kuva 3.1. Notepad'n tukemat tiedoston koodausvaihtoehdot

Tyypillisesti tiedostojen lukuongelmat johtuvat näkymättömistä merkeistä, joten niiden ratkominen on usein turhauttavaa ja aikaa vievää. Siksi ongelmatilanteessa kannattaa pyrkiä selvittämään asiaa sen verran, että voi etsiä apua esim. Internetistä tai kysyä muilta. Monet muutkin ohjelmoijat ovat törmänneet näihin samoihin ongelmiin eli kokeneilla ohjelmoijilla on usein erilaisia keinoja näiden ongelmien selvittämiseen ja ratkaisemiseen.

Makrot

Tutustuimme edellisessä luvussa esikäsittelijän `define`-käskyyn ja määrittelimme sillä vakioita merkkijonoliteraalien avulla. Esimerkiksi taulukon koon määrittely onnistui `#define MAX 10` -käskyllä, jolloin kaikkiin taulukon käsittelyyn käytettyihin silmukoihin saadaan sama raja-arvo.

Merkkijonoliteraalien avulla voidaan määritellä myös makroja, jotka ovat funktioiden tyylisiä uudelleen käytettäviä koodilohkoja. Todellisuudessa makro korvataan annetulla koodilla ja tästä seuraa merkittäviä eroja makron ja funktion toimintaan. Makrojen keskeinen etu on se, että ne näyttävät funktioilta eli yksinkertaistavat koodia, mutta välttävät funktiokutsun aiheuttaman aliohjelman liittyvän tietojen siirtämisen pinoon ja ohjelman suorituksen hidastumisen. Toisaalta esikäsittelijän korvatta makron koodilla, kääntäjä ei voi esim. tarkistaa ”parametrien” tietotyyppejä samalla tavoin kuin funktiokutsuissa. Lisäksi laskennan tulos ei välttämättä ole haluttu vaan esikäsittelijän tekemät etsi-korvaa -muutokset saattavat johtaa ongelmiin, jos parametrit sisältävät laskutoimituksia ja korvaus-operaation jälkeen operaattoreiden presedenssit muuttavat laskujärjestystä sulkujen puute vuoksi. Esimerkki 3.10 demonstroi tällaista tilannetta, jossa kaikki toimii ensimmäisellä kerralla normaalisti – ja oikein –, mutta toisella parametrillä tulos ei ole haluttu ja näin ollen voidaan väitellä siitä, toimiiko ohjelma oikein vai väärin. Ohjelma toimii kuitenkin juuri niin kuin se on toteutettu ja vastuu ohjelman toiminnasta on ohjelmoijalla.

Esimerkki 3.10. Makrot

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

int main(void) {
    printf("Makron square(3) tulos on %d.\n", square(3));
    printf("Makron square(3+1) tulos on %d.\n", square(3+1));
    return(0);
}
```

Esimerkkikoodin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E3_10
Makron square(3) tulos on 9.
Makron square(3+1) tulos on 7.
```

Kuinka koodi toimii

Ohjelman esikäsittelijä vaihtaa `square(3)` merkkien tilalle merkkijonon `3*3` ja kääntämisen 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 johtaa useimmille odottamattomaan tulokseen 7, kun makron tilalle vaihtuu esikäsittelyssä merkkijono `'3+1*3+1'`. Siksi on syytä muistaa, että makrot toimivat eri tavoin kuin funktiot ja niiden kanssa pitää olla tarkkana.

Kirjastofunktiot

C-kielen toteutuksessa on pyritty minimaalisen ytimen luomiseen. Näin ohjelmat voidaan räätälöidä sisältämään vain pakolliset ominaisuudet ja kaikki tarpeeton voidaan jättää pois. 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. Siksi tiedon syöttö- ja tulostuskäskyt on sijoitettu tähän tarkoitukseen tehtyyn kirjastoon (standard input/output –kirjastoon ja otsikkotiedostoon `stdio.h`). Kun näitä toimintoja tarvitaan, otetaan tarvittava kirjasto käyttöön sisällyttämällä halutun kirjaston otsikkotiedosto lähdekooditiedostoon `include`-käskyllä, esim.:

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

Liitteessä 1 on nimetty yleisimpiä C-kielen otsikkotiedostoja ja niiden sisältämiä funktioita. Otsikkotiedostot sisältävät myös vakioiden määrittelyjä, joita tyypillisesti tarvitaan näiden funktioiden ja toimintojen yhteydessä. Esimerkissä 3.11 näkyy satunnaisluku-funktion `rand()` käyttö pienessä ohjelmassa eli tällä tavoin voidaan hyödyntää olemassa olevia funktioita, kuten olemme tehneet jo usein aiemmin, esim. `string.h`-otsikkotiedoston ja `strlen()` sekä `strcmp()`-funktioiden kanssa. Näistä funktioista löytyy tarkempia tietoja man-sivuilta ja oppaan lopussa palaamme omien kirjastojen tekemiseen. Satunnaislukujen kohdalla tulee muistaa, että ne ovat pseudosatunnaislukuja ja generaattorin alustus esimerkin mukaisesti tuottaa aina samat tulokset.

Esimerkki 3.11. Kirjaston käyttö, satunnaisluku

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

int main(void) {
    int i;
    srand(3); // Alustetaan satunnaislukugeneraattori
    for (i=0; i < 10; i++)
        printf("Satunnaisluku '%d'.\n", rand());

    return(0);
}
```

Pienen C-ohjelman tyyliohjeet

Tiedostorakenne. Aliohjelmia sisältävien pienten C-ohjelmien tiedostoissa kannattaa noudattaa seuraavaa rakennetta:

1. Ohjelman alkukommentti: Kuka tehnyt, milloin, tiedostonimi
2. Kirjastojen sisällytykset: Esim. `#include <stdio.h>`
3. Aliohjelmien esittelyt samassa järjestyksessä kuin niitä käytetään ohjelmassa
 - a. Esim. `int summa(int iLuku1, int iLuku2);`
4. Pääohjelma
 - a. Muuttujien määrittelyt ja alustukset
 - b. Tietojen kysyminen käyttäjältä
 - c. Tietojenkäsittely, esim. laskenta
 - d. Tulosten tulostaminen käyttäjälle
5. Aliohjelmien määrittelyt samassa järjestyksessä kuin ne on esitelty tiedoston alussa

Aliohjelmarakenne. Aliohjelmat kannattaa suunnitella niin, että yhdessä aliohjelmassa tapahtuu vain yksi asia samaan tyyliin kuin `printf`-funktiota käytetään tiedon tulostamiseen ja `scanf` tai `fgets`-funktiota käytetään tiedon lukemiseen käyttäjältä tai tiedostosta. Näin ollen siis esim. tiedoston lukeminen ja kirjoittaminen ovat kaksi erillistä aliohjelmia eikä niissä tehdä periaatteessa muuta kuin luku tai kirjoitus ja esim. analyysi tai laskenta ovat erillinen/erillisiä ohjelmia tiedostonkäsittelyohjelmien rinnalla.

Aliohjelmien nimeäminen. Aliohjelmien nimet kannattaa tämän oppaan laajuudessa aloittaa pienellä kirjaimella ja muodostaa ne useista sanoista siten, että seuraava osa alkaa aina isolla kirjaimella. Esimerkiksi `lueTiedosto`, `kirjoitaTiedosto`, `kysyNimi` jne. ovat selkeitä aliohjelmien nimiä, joista arvaa helpolla, mitä niissä tehdään.

Makrot. Makrot ovat tyypillistä C-ohjelmointia, mutta ne toimivat eri tavoin kuin funktiot ja siksi niitä kannattaa käyttää vasta sitten, kun osaa käyttää kunnolla funktioita ja ymmärtää näiden kahden erot. Useissa kääntäjissä on lisäksi erilaisia ratkaisuja, joissa esimerkiksi pinon käsittely on toteutettu niin, etteivät ne ole makroja hitaampia. Esimerkki tällaisesta ratkaisusta on funktion `inline`-määrite, joten näihin kannattaa tutustua tarkemmin, jos nopeus on tärkeää.

Virheenkäsittely. Tässä oppaassa virheenkäsittelyyn liittyy kolme tehtävää: (1) virheen tunnistaminen, (2) virhetilanteen raportointi käyttäjälle ja (3) ohjelman suorituksen lopettaminen. Virhe tunnistetaan tyypillisesti paluuarvosta eli funktio palauttaa virhekoodin. Käyttäjälle voi kertoa virheestä `printf`:llä tai `perror`:lla, joista jälkimmäinen on suositeltu tapa, koska se lisää käyttöjärjestelmän virhe-tiedon ilmoitukseen ohjelmoijan tietojen lisäksi. Ohjelman suoritus lopetetaan `exit(0)`-käskyllä, jotta virhekohta on mahdollisimman helppo löytää koodista ja korjata.

Yhteenveto

Kerrataan luvun osaamistavoitteet ja esimerkkien aiheet sekä katsotaan keskeiset asiat kokoava ohjelmointiesimerkki.

Osaamistavoitteet

Tämä luvun jälkeen sinun tulee pystyä käsittelemään tekstitiedostoja sekä käyttämään aliohjelmaa ohjelman rakenteen hallitsemiseen. Luvun esimerkit keskittyivät tekstitiedostojen lukemiseen ja kirjoittamiseen, mutta binaaritiedostojen käsittely poikkeaa näistä lähinnä käytettävien funktioiden osalta. Tämän oppaan suositus on käyttää virheenkäsittelyä aina tiedostoja avattaessa ongelmien välttämiseksi. 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 paljon kirjastoja, joista löytyy paljon valmiita aliohjelmaa eri käyttötarkoituksiin. Itse C-kieli on pidetty pienenä, jotta käännettyjen ohjelmien koko saadaan pidettyä mahdollisimman pienenä.

Käsitellyt asiat

Tässä luvussa kävimme läpi C-ohjelmointiin liittyen seuraavat asiat:

- Tiedoston kirjoittaminen: Esimerkki 3.1, 3.3 ja 3.12
- Tiedoston lukeminen: Esimerkki 3.2, 3.3 ja 3.12
- Binaaritiedoston käsittely: Esimerkki 3.4
- Aliohjelman määrittely, esittely ja kutsu: Esimerkki 3.5
- Parametrit ja paluuarvot: Esimerkki 3.6, 3.9
- Muuttujaparametrit, osoitinmuuttuja: Esimerkki 3.7, 3.9
- Tunnusten näkyvyys: Esimerkki 3.8
- Makrot: Esimerkki 3.10
- Kirjastojen käyttö: Esimerkki 3.11

Kokoava esimerkki

Kokoavassa esimerkissä on valikkopohjainen ohjelma tiedostojen lukemiseen ja kirjoittamiseen. Pääohjelmassa on toistorakenne, josta kutsutaan valikon tulostavaa aliohjelmaa sekä tiedoston lukua ja kirjoitusta käyttäjän valintojen mukaan.

Esimerkki 3.12. Valikkopohjainen ohjelma tiedostojen käsittelyyn

```

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

int valikko() {
    int iValinta;
    printf("Valitse alla olevista valinnoista\n");
    printf("1) Kirjoita tiedosto\n");
    printf("2) Lue tiedosto\n");
    printf("0) Lopeta\n");
    printf("Anna valintasi: ");
    scanf("%d", &iValinta);
    getchar();
    return iValinta;
}

void kirjoitaTiedosto(char *pNimi) {
    int i;
    FILE *tiedosto;

    if ((tiedosto = fopen(pNimi, "w")) == NULL) {
        perror("Tiedoston avaaminen epäonnistui");
        exit(1);
    }
    for (i=0; i < 20; i++)
        fprintf(tiedosto, "%d\n", i);
    fclose(tiedosto);
    printf("Tiedosto '%s' kirjoitettu.\n", pNimi);
    return;
}

void lueTiedosto(char *pNimi) {
    char aRivi[52];
    FILE *tiedosto;
    printf("Tiedostosta löytyy seuraavat rivit:\n");

    if ((tiedosto = fopen(pNimi, "r")) == NULL) {
        perror("Tiedoston avaaminen epäonnistui");
        exit(1);
    }
    while (fgets(aRivi, 52, tiedosto) != NULL) {
        printf("%s", aRivi);
    }
    fclose(tiedosto);
    printf("Tiedosto '%s' luettu.\n", pNimi);
    return;
}

```

```
int main(void) {
    int iValinta;
    char aNimi[] = "tiedosto.txt";

    do {
        iValinta = valikko();
        switch (iValinta) {
            case (1):
                kirjoitaTiedosto(aNimi);
                break;
            case (2):
                lueTiedosto(aNimi);
                break;
            case (0):
                printf("Kiitos ohjelman käytöstä.\n");
                break;
            default:
                printf("Annoit tuntemattoman valinnan.\n");
                break;
        }
    } while (iValinta != 0);
    return(0);
}
/* eof */
```


Luku 4. Tietorakenteita ja algoritmeja

Tässä luvussa perehdymme tarkemmin muutamiin uusiin rakenteisiin tietorakenteisiin ja niihin liittyviin algoritmeihin. Lopuksi katsomme C-kielen ominaispiirteitä ennen luvun osaamistavoitteiden listaamista ja kokoavaa esimerkkiä.

Tietorakenteita

Olemme jo aiemmin käyttäneet merkkitaulukkoa ja C-kielen muut taulukot perustuvat samoihin periaatteisiin ulottuvuuksien määrästä riippumatta. Tietue on taulukkoa täydentävä tietorakenne, johon voidaan tallettaa erityyppisiä tietoalkioita ja siten siihen liittyy uusien tietotyyppien määrittely. Yhteistä taulukolle ja tietueelle on, että ne molemmat sisältävät useita eri tietoalkioita yhden muuttujan sisällä eli ne ovat rakenteisia tietorakenteita. Lopuksi tutustumme komentoriviparametreihin ja katsomme, miten osoittimia käytetään tietueiden kanssa.

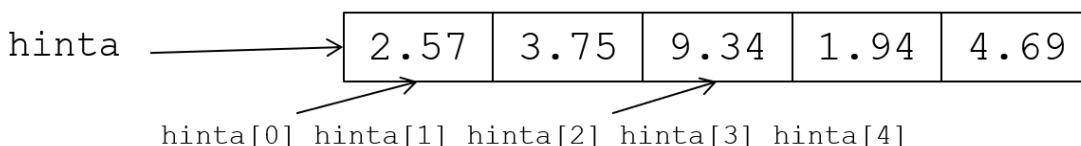
Taulukot

Olemme käyttäneet merkkitaulukoita oppaan alusta alkaen. Merkkitaulukko tarkoittaa useita peräkkäisiin muistipaikkoihin talletettuja merkkejä ja taulukko ilmoittaa, kuinka monta peräkkäistä yhden merkin kokoista muistipaikkaa varataan. Tämä tehdään muuttujan nimen perässä olevilla hakasuilla tyyliin `char nimi[30]`, joka varaisi muistia 30 tavua ASCII-merkkejä varten ja näistä muistipaikoista ensimmäinen löytyy muuttujan nimellä `nimi`.

C-kielen muut taulukot perustuvat samaan ideaan eli 30 kokonaisluvulle varattaisiin taulukko määrittelyllä `int taulukko[30]`. Merkkitaulukko on ainoa taulukko, joka voi sisältää merkkijonon loppumerkin, `'\0'` eli NULL-merkin. Loppumerkin avulla maksimikokoa pienemmille merkkijonoille on luonnollinen esitystapa, mutta muissa taulukoissa, esim. kokonaislukutaulukoissa, ei ole loppumerkkiä. Siksi C-ohjelmissa on pidettävä tarkkaa kirjaa taulukoiden koosta ja varmistuttava, että kaikki alkiot tulevat käytyä läpi eikä ohjelmissa tapahdu ”ylivuotoja” eli jatketa taulukon alkioden läpikäyntiä niiden loputtua.

Kuvassa 4.1 näkyy liukulukutaulukon määrittely ja sen alkioden indeksit. Tältä osin taulukko vastaa merkkitaulukkoa. C-kielen tietotyyppien selkeyden takia on syytä muistaa, että taulukon kaikkien alkioden tulee olla samaa tyyppiä, tässä tapauksessa liukulukuja. Vain merkkitaulukoiden käsittelyyn on olemassa valmiita operaatioita string-kirjastossa ja muut taulukot on käsiteltävä aina alkio kerrallaan ohjelmoijan toimesta. Käytännössä esimerkiksi taulukon kopiointi tarkoittaa toisen vähintään yhtä suuren taulukon varaamista ja tietojen kopioimista alkio kerrallaan ensimmäisestä taulukosta toiseen. Staattisesti varattujen taulukoiden kokoa ei pysty muuttamaan ohjelman suorituksen aikana, joten taulukon tulee olla alusta alkaen riittävän iso ja kun alkioden määrää ei pysty selvittämään jälkikäteen, joten ohjelmoijan tulee itse pitää kirjaa niistä sopivalla tavalla.

```
float hinta[5];
```



Kuva 4.1. Liukulukutaulukon `hinta` määrittely 5 luvulle ja alkioden indeksit

Esimerkissä 4.1 näkyy kokonaislukutaulukon määrittely, arvojen sijoittaminen taulukkoon sekä taulukossa olevien arvojen tulostaminen näytölle. Taulukon määrittelyssä tulee käydä ilmi taulukon koko, mikä tehdään tyypillisesti vakion avulla esimerkin mukaisesti. Määrittelemällä taulukon koko vakiolla, samaa vakiota voidaan käyttää aina taulukon läpikäynnin yhteydessä. Kaikki taulukon alkiot ovat aina samaa tietotyyppiä ja kun taulukon koko on tiedossa etukäteen, on `for`-lause luonnollinen ratkaisu taulukon alkioiden läpikäyntiin niiden indeksien perusteella.

Esimerkki 4.1. Numeerinen taulukko

```
#include <stdio.h>
#define KOKO 10

int main(void) {
    int i;
    int taulukko[KOKO];

    for (i=0; i < KOKO; i++)
        taulukko[i] = i;

    for (i=0; i < KOKO; i++)
        printf("%d ", taulukko[i]);
    printf("\n");

    return(0);
}
```

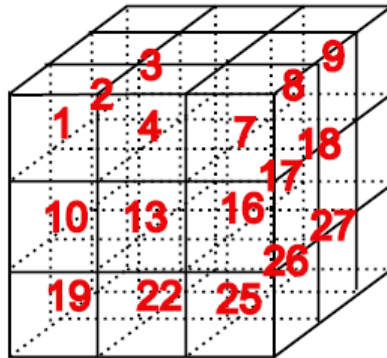
Taulukoiden alustus noudattaa merkkitaulukoiden yhteydessä esiteltyjä periaatteita. Esimerkissä 4.1 varataan taulukko, mutta sen alkioita ei alusteta. Näin ollen C-kääntäjä varaa halutun muistialueen, muttei alusta sitä vaan taulukon arvoina voi olla mitä tahansa. Numeeristen taulukoiden alustus voidaan tehdä aaltosuluilla alla olevaan tyyliin:

```
int taulukko2[] = {6, 2, 3};
```

Tällä tavoin taulukon alkioiksi sijoitettaisiin arvot 6, 2 ja 3 kääntäjän toimesta ja taulukko olisi vastaavasti kolmen kokonaisluvun kokoinen. Vaikka tämä onkin nopea tapa päästä alkuun ohjelman kanssa, ei tässä ratkaisussa taulukon alkioiden määrää tule kirjattua ja alkioiden läpikäynti toistorakenteissa voi muodostua hankalaksi ohjelman kasvaessa.

Moniulotteiset taulukot perustuvat samaan ideaan kuin yksiulotteiset taulukot eli varataan muistia useita samanlaisia tietoalkioita varten haluttu määrä. Näin ollen muistialue yhtä tietoalkiota varten varataan muuttujalla, esim. `int numero`, ja jos tarvitaan useita alkioita, ilmoitetaan samanlaisten tietoalkioiden määrä nimen perässä hakasuluilla, esim. 5 alkioinen taulukko `int numero[5]`. Jos tietoalkiota on 36, voidaan varata muistia `int numero[36]` -määrittelyllä, tai jos tietoja on 6 alkioita ja niitä on 6 rivillä, voidaan muistia varata `int numero[6][6]` -määrittelyllä eli kaksiulotteisena taulukkona. Myös kolmiulotteinen taulukko muodostuu samalla logiikalla eli lisätään määrittelyyn yksi dimensio lisää uudella hakasulkuparilla `int numero[6][6][6]`. Tältä pohjalta ei ole yllättävää, että C-kielessä voidaan käsitellä haluttu määrä ulottuvuuksia laittamalla sopiva määrä hakasulkuja taulukkomuuttujan määrittelyyn. Teknisesti useat ulottuvuudet eivät ole ongelma, mutta tyypillisesti riittää yksiulotteinen taulukko merkkijonoille ja kaksiulotteinen taulukko sopii hyvin tietojen esittämiseen taulukkolaskennan tyyliin riveinä ja sarakkeita. 3D mallinnuksen, ja pelien, yleistyessä joudutaan tietoja käsittelemään myös kolmessa ulottuvuudessa, mutta harva pystyy hahmottamaan ja hyödyntämään tätä useampia ulottuvuuksia.

Kuvassa 4.2 näkyy kolmiulotteisen taulukon visualisointi ottamatta kantaa siihen, missä origo on ja miten alkiot tulisi käydä läpi jne.



Kuva 4.2. Kolmiulotteisen taulukon visuaalinen esitys. Origo ja akselit määräytyvät sovellusalueen mukaan eikä yhtä ainuttakaan oikeaa ratkaisua ole olemassa.

C-koodissa ulottuvuudet näkyvät siis hakasulkujen määränä. Alla on määritelty 2-ulotteinen taulukko, matriisi, hyödyntäen vakioita RIVI ja SARAKE, ja matriisi on alustettu arvoilla sen määrittelyn yhteydessä.

```
#define RIVI 2
#define SARAKE 2
int matriisi[RIVI][SARAKE] = {{1,2}, {3,4}};
```

Matriisin läpikäynti on ohjelmoinnin perusoperaatioita, joka käydään läpi tämän luvun toisessa osassa algoritmien yhteydessä.

Tietue

Monet sovellukset käsittelevät henkilötietoja ja tällöin joudutaan käsittelemään useita erityyppisiä tietoja kuten esimerkiksi etunimi, sukunimi, syntymäaika, osoite jne. Tällaisia useista tiedoista koostuvia tietokokonaisuuksia tarvitaan enenevässä määrin ja siksi useimmat ohjelmointikielet tarjoavat keinoja yhdistää useita muuttujia yhteen tietorakenteeseen. C-kielessä tällaista rakennetta kutsutaan tietueeksi ja se luodaan `struct`-avainsanalla. Sen määrittely ja käyttö muistuttavat Pythonin luokkarakennetta, kun ei huomioida jäsenfunktioita, ja määrittely tehdään seuraavalla syntaksilla:

```
struct tietueen_nimi{
    tietueen_jäsenet, tietotyyppi ja nimi;
};
```

Kun tietue on luotu, voimme tehdä tietueen rakenteen mukaisia muuttujia käskyllä `'struct tietueen_nimi muuttuja;'` esimerkin 4.2 mukaisesti.

Esimerkki 4.2. Tietueen määrittely ja käyttö

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

struct henkilo {
    char aNimi[30];
    int iIka;
    char aOsoite[30];
};

int main(void) {
    struct henkilo hlo;

    hlo.iIka = 17;
    strcpy(hlo.aOsoite, "sihijuomakatu");
    /* strcpy eli string copy - kopioi merkkijono */
    printf("Anna henkilön nimi:\n");
    scanf("%s", hlo.aNimi);

    if (hlo.iIka < 18) {
        printf("%s on alaikäinen.\n", hlo.aNimi);
    }
    return(0);
}
```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E4_2
Anna henkilön nimi:
Hjördis
Hjördis on alaikäinen.
```

Kuinka ohjelma toimii

Jotta tietueita voisi käyttää, pitää lähdekoodin alussa kertoa kääntäjälle, mitä tietueita lähdekoodissa käytetään. Tämä tapahtuu kirjoittamalla tietueiden määrittelyt lähdekoodiin ennen niiden käyttöä eli ennen funktioesittelyitä ja toiminnallista koodia. Seuraava koodi luo tietueen `henkilo`:

```
struct henkilo {
    char aNimi[30];
    int iIka;
    char aOsoite[30];
};
```

Tietueen jäsenmuuttujina ovat merkkitaulukko `aNimi`, kokonaislukumuuttuja `iIka` sekä merkkitaulukko `aOsoite`. Kun olemme luoneet tietueen, voimme käyttää sitä koodissa. Tässä vaiheessa olemme kertoneen kääntäjälle, että lähdekoodissa voi esiintyä tällainen tietorakenne. Tämän vuoksi `main`-funktion sisällä on rivi

```
struct henkilo hlo;
```

jolla luomme tietueen tyypin muuttujan. Rivi siis kertoo, että luomme tietueen `henkilo` mukaisen muuttujan `hlo`. Tästä eteenpäin voimme käyttää `hlo:n` jäsenmuuttujia kuten normaaleja muuttujia. Esimerkin muilla riveillä näkyy mm. merkkijonojen kopiointi jäsenmuuttujaan, jäsenmuuttujan käyttö if-rakenteen ehtona sekä jäsenmuuttujien tulostus. 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 `hlo.aOsoite`.

Uuden tietotyypin määrittely

Omien tietueiden lisäksi C-kielessä voi määrittellä omia tietotyyppiejä tarpeen mukaan. Alla esimerkissä 4.3 näkyy tietueen `'struct henkilo'` määrittely ja miten siitä voidaan määrittellä oma tietotyyppi kahdella eri tavalla.

Esimerkki 4.3. Tietueen määrittely uudeksi tietotyyppiksi

```

/*****
// typedef vaihtoehto 1:
struct henkilo {                                // Tietueen määrittely
    char aNimi[30];
    int iIka;
};
struct henkilo muuttuja1;                        // Muuttujan määrittely
typedef struct henkilo HENKILO;                  // Tietotyypin määrittely
HENKILO muuttuja2;                               // Muuttujan määrittely
HENKILO *pHlo1;                                  // Osoittimen määrittely

/*****
// typedef vaihtoehto 2:
typedef struct henkilo { // Tietueen ja tietotyypin määrittely
    char aNimi[30];
    int iIka;
} HENKILO;
HENKILO muuttuja3;
HENKILO *pHlo2;

*****/

```

Vaihtoehto 1 noudattaa samaa kaavaa kuin Esimerkki 4.2, jossa määriteltiin ensin tietue `struct henkilo` ja sen jälkeen siihen perustuen muuttuja `struct henkilo muuttuja1`; Uuden tietotyypin määrittely perustuu samaan tietueeseen, `struct henkilo`, jota edeltää avainsana `typedef` ja jota seuraa uuden tietotyypin nimi `HENKILO` eli `typedef struct henkilo HENKILO`; . Tämän jälkeen uutta tietotyyppiä voidaan käyttää muuttujien ja osoittimien määrittelyyn samalla tavalla kuin muita tietotyyppiejä kuten `int`. Tässä oppaassa itse määritellyt tietotyypit kirjoitetaan SUURAAKKOSILLA selkeyden vuoksi, eikä se ole pakollista.

Vaihtoehto 2 määrittelee uuden tietotyypin samalla tavoin kuin vaihtoehto 1, mutta tiivistää tietueen ja tietotyypin määrittelyt yhteen lauseeseen. Käytännössä vaihtoehto 2 noudattaa vaihtoehdon 1 `typedef`-lausetta, mutta sisältää tietueen jäsenmuuttujien määrittelyt samassa lauseessa eli

`typedef struct henkilo {} HENKILO;`, kun aaltosulkuihin lisätään jäsenmuuttujat yllä olevan esimerkin 4.3 mukaisesti.

Tämän luvun lopussa on esimerkki 4.8 toimivasta ohjelmasta, jossa määritellään uusi tietotyyppi, muodostetaan siitä taulukko ja lopuksi tallennetaan sekä luetaan taulukko binaarimuodossa.

Komentoriviparametrit

Komentoriviparametrit ovat C-ohjelman pääohjelman, `main`-funktion, parametreja. 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 seuraavalla tavalla:

```
int main(void)
```

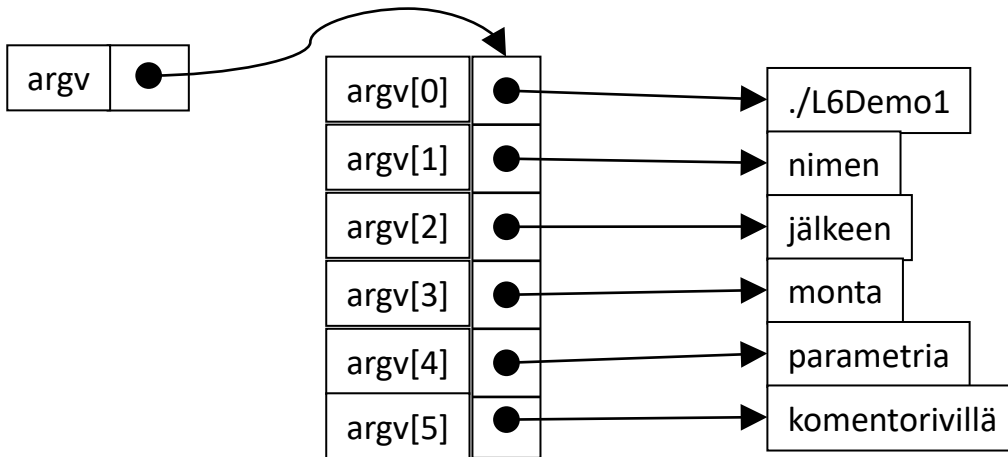
Mikäli haluamme, että käyttäjä voi syöttää jo ohjelman käynnistyksen yhteydessä jotain tietoa, kuten kohdetiedoston nimen tai muita toimintaa ohjaavia parametreja, tarvitsemme keinon siirtää käyttäjän antamat syötteet `main`-funktiolle. Tämä on itse asiassa helppoa, sillä meidän tarvitsee vain muuttaa `main`-funktion parametrimäärittely seuraavaan muotoon:

```
int main(int <lukumäärämuuttujan_nimi>, char *<parametrilistan_nimi>[])
eli tyypillisesti
```

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

Tämän jälkeen kääntäjä hoitaa loput. Näillä muutoksilla näemme annettujen parametrien määrän kokonaislukuna ensimmäisestä muuttujasta ja itse parametrit toisesta muuttujasta merkkijonoina. Yksittäisiin parametreihin voimme viitata notaatiolla `argv[i]`, jossa `i` on parametrin järjestysnumero. Parametri 0 on ajettavan tiedoston nimi ja täten ensimmäinen varsinainen parametri on indeksillä 1. `main`-ohjelma saa aina vähintään yhden parametrin eli ohjelman nimen. Kuva 4.3 visualisoi komentoriville kirjoitettua tekstiä sekä miltä se näyttää `main`-ohjelmassa `argv`-parametrin arvoina.

Komentorivi: `./L6Demo1` nimen jälkeen monta parametria komentorivillä



Kuva 4.3. Komentorivin siirtyminen main-ohjelmaan argv-rakenteen avulla

Komentoriviparametrien käyttö C-ohjelmassa näkyy esimerkissä 4.4. Kaikki komentoriviparametrejä käsittelevä koodi pitää kirjoittaa itse.

Esimerkki 4.4. Komentoriviparametrit

```
#include <stdio.h>

/* main-funktion parametrit:
 * argc - integer-muuttuja, johon tulee tieto parametrien määrästä
 * argv - argument vector eli merkkitaulu, jossa on itse parametrit
 */

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

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

Esimerkin tuottama tulos

Kun käänämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E4_4 yksi kaksi 313
Annoit 4 komentoriviparametria, jotka olivat:
0. parametri oli ./E4_4
1. parametri oli yksi
2. parametri oli kaksi
3. parametri oli 313
```

Kuinka ohjelma 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 `./E4_4`) ja että parametrien yhteismäärä `argc`-muuttujassa oli 4, joka on käyttöjärjestelmän tälle ohjelmalle välittämien parametrien lukumäärä.

Viimeinen parametri oli tällä kertaa numero merkkijonona, joten katsotaan sitä tarkemmin. Koska `argv:n` parametrit ovat merkkijonoja, emme voi käyttää arvoa ”313” muuttamatta sitä ensin luvuksi. C-kielessä merkkijonon muutos numeroksi tehdään esim. funktiolla `atoi()`, joka muuttaa merkkijonon kokonaisluvuksi (ASCII-to-integer). C-kielelle tyypillisesti funktiot tekevät vain pyydetyn muunnoksen ja muunnosfunktioita on useita, esim. `atoi()`, `atol()` ja `atof()`, jotka ottavat parametrina merkkijonon ja palauttavat halutun numeron (`int`, `long`, `double`). Lisäksi on muita samantyyllisiä funktioita kuten `strtol()`, `strtof()` ja `strtod()`, joten näitä kannattaa katsoa, jos `int atoi(char *ptr)/double atof(char *ptr)` ei vastaa tarvetta. Tämän oppaan osalta näillä kahdella pääsee pitkälle, mutta tarpeita on erilaisia ja siksi myös erilaisia vaihtoehtoja muunnosten toteutukseen on olemassa. Tyypimuunnosfunktioita löytyy lisää liitteestä 3 ja funktioiden tarkat tiedot löytyvät `man`-sivuilta.

Tietue-osoitin

C-kieli tarjoaa tyypillisesti tehtävien tekemiseen useita vaihtoehtoja, joilla on pieni mutta merkitsevä ero. Muuttuja ja osoitinmuuttuja ovat hyviä esimerkkejä tästä ja siksi niistä on puhuttu käytännössä kaikissa tämän oppaan luvuissa. Tietue-tietotyypin esittelyn vuoksi kerrataan muuttujan ja osoittimen perusasiat ja katsotaan sitten, miten tietueita käytetään osoittimien kanssa.

Alla on tyypillinen ohjelmakoodi, jossa määritellään muuttuja ja osoitin samantyyppiseen muuttujaan. Muuttujan määrittely varaa muistia ko. muuttujalle, jotta siihen voidaan sijoittaa arvo ja arvo on sen jälkeen käytettävissä ohjelmassa muuttujan nimellä. Osoittimen määrittely varaa muistia vain osoittimelle, joten osoitin ei pysty tallettamaan varsinaista tietoa eli arvoja vaan osoitin voi vain osoittaa muistipaikkaan ja käyttää siellä olevaa arvoa. Näin ollen arvo tulee sijoittaa muuttujaan, ja kun osoittimeen laitetaan muuttujan osoite, päästään arvoon käsiksi sekä muuttujan että osoittimen kautta alla olevan esimerkin mukaisesti.

```
// Määrittelyt
int luku; // Muuttuja: varaa muistia arvolle
int* pLuku; // Osoitin: osoittaa lukuun muttei varaa muistia arvolle

// Käyttö
luku = 2;
pLuku = &luku;
printf("%d, %d", luku, *pLuku);
```

Tietueiden kohdalla on pidettävä erillään (1) tietueen määrittely, (2) tietotyypin määrittely, (3) muuttujan määrittely ja (4) osoittimen määrittely. Esimerkissä 4.5 määritellään ensin tietue `struct henkilo`, sitten siihen perustuva tietotyyppi `HENKILO` ja lopuksi määritellään tähän tyyppiin perustuva muuttuja `hlo` ja osoitin `pHlo`. Tässä esimerkissä osoitin laitetaan osoittamaan muuttujalle varattuun muistialueeseen ja jäsenmuuttujien arvojen asettamisen voi katsoa esimerkistä

4.2, sillä oleellista on nyt huomata, miten jäsenmuuttujiin viitataan muuttujalla ja osoittimella. Esimerkin 4.2 mukaisesti muuttujan jäsenmuuttujiin viitataan pistenotaatiolla (esim. `hlo.iIka`), ja näin ollen osoitinmuuttuja mahdollistaa myös jäsenmuuttujiin viittaamisen pistenotaatiolla (esim. `(*pHlo).iIka`). Tätä viittaustyyliä käytetään kuitenkin harvoin ja tyypillisesti osoittimien yhteydessä käytetään nk. nuoli-notaatiota eli esim. `pHlo->iIka`.

Esimerkki 4.5. Tietueen jäsenmuuttujat ja osoitin

```
struct henkilo {                                // Uuden tietue-tietorakenteen
    char aNimi[30];                             // määrittely
    int iIka;
};

typedef struct henkilo HENKILO; // Uuden tietotyypin määrittely
HENKILO hlo;                     // Uuden tietue-muuttujan määrittely
HENKILO *pHlo;                  // Uuden tietue-osoitin määrittely
pHlo = &hlo;                    // Osoittimen arvoksi muuttujan osoite

... // Jäsenmuuttujien arvojen asettaminen, ks. Esimerkki 4.2

printf("Ikä on %d, nimi on %s.\n", hlo.iIka, hlo.aNimi);
printf("Ikä on %d, nimi on %s.\n", (*pHlo).iIka, (*pHlo).aNimi);
printf("Ikä on %d, nimi on %s.\n", pHlo->iIka, pHlo->aNimi);
```

Nuoli-notaation käyttöön tietue-osoitimissa palataan jatkossa useaan otteeseen. Se on tyypillinen C-ohjelmoijien käyttämä esitystapa, joten se on normaalia C-ohjelmointityyliä. Jokainen voi valita itselleen sopivan ohjelmointityylin erityisesti tämän oppaan laajuudessa.

Algoritmeja

Tutustuimme edellä muutamiin uusiin tietorakenteisiin, joista algoritmien kannalta kiinnostavin on moniulotteinen taulukko ja matriisi. Tämän jälkeen tutustumme rekursiivisen ohjelman toimintaan.

Taulukon alkioden käsittely

Moniulotteisten taulukoiden määrittely ja rakenne käytiin läpi aiemmin tässä luvussa, mutta jätimme niiden läpikäynnin tänne algoritmipuolelle. Taulukoiden keskeinen ominaisuus on alkioden lukumäärän määrittely etukäteen ja siten taulukoita käydään tyypillisesti läpi `for`-silmukalla, joka tarjoaa selkeän askeltajan ja samalla indeksin taulukon alkioihin kuten näimme merkkitaulukoiden kohdalla. Moniulotteisen taulukon kohdalla kaikki ulottuvuudet käydään läpi vastaavasti eli kaikkien indeksien kaikki arvot tulee käydä läpi. Katsotaan seuraavaksi matriisin eli kaksi ulotteisen taulukon läpikäynti, koska se on tyypillinen taulukkolaskennan rivi-sarake -rakenne. Mikäli ulottuvuuksia on enemmän, kuten esim. 3D-mallinnuksessa ja grafiikassa, lisätään taulukkoon kolmas ulottuvuus ja se käydään läpi vastaavalla tavalla kuin matriisi eli 2D-malli.

Esimerkissä 4.6 näkyy 2x2 matriisin määrittely, alustus ja tulostus näytölle. Matriisin rivien ja sarakkeiden määrät ovat vakioita, jotta kääntäjä pystyy alustamaan ohjelman käyttämän muistin oikein. Määrittelemällä ulottuvuuksien maksimi-arvot vakioilla, on vakioita helppo käyttää myös silmukoiden maksimi-arvoina ja kaikki taulukon alkiot tulee käytyä läpi. Rivien ja sarakkeiden järjestyksessä oleellista on yhteisymmärrys ohjelmaa käyttävien kesken, mutta näytölle

tulostettaessa on syytä tulostaa ensin kaikki yhden rivin alkiot eli käydä sarakkeet läpi ja sen jälkeen siirtyä seuraavalle rivillä ja tulostaa kaikki toiselle riville kuuluvat alkiot siihen jne. C-ohjelmassa on syytä huomata esimerkin aaltosulut. Kaikki yhdelle riville tulevat tietoalkiot saadaan tulostumaan oikein ilman aaltosulkuja, mutta rivin loppuun tuleva rivinvaihtomerkki pakottaa tekemään rivin tulostuksesta koodilohkon, johon kuuluu kaikkien tietoalkioiden tulostus ja sen lisäksi rivinvaihto.

Esimerkki 4.6. 2-ulotteisen taulukon eli matriisin läpikäynti

```
#include <stdio.h>
#include <stdlib.h>
#define RIVI 2
#define SARAKE 2

int main(void) {
    int i, j;
    int matriisi[RIVI][SARAKE] = {{1,2}, {3,4}};

    for (i=0; i < RIVI; i++) {
        for (j=0; j < SARAKE; j++)
            printf("%d ", matriisi[i][j]);
        printf("\n");
    }
    return(0);
}
```

Taulukoiden läpikäynti on luontevaa toistorakenteilla ja indekseillä, sillä tieto alkioden määrästä tekee tästä lähestymistavasta luonnollisen. Samaan lopputulokseen päästään myös osoittimilla, mutta tässä tapauksessa osoitin ei tuo lisäarvoa eikä helpota ratkaisua. Näin ollen taulukot käydään tyypillisesti läpi indekseillä osoittimien sijasta.

Moniulotteisten taulukoiden läpikäynnissä oleellista on systemaattinen kaikkien ulottuvuuksien läpikäynti kaikkien arvojen osalta. Missä järjestyksessä ulottuvuuden käydään läpi, on syytä sopia koodia käyttävien tahojen kanssa, ettei asiasta muodostu ongelmaa. On hyvä muistaa, että peruskoulussa koordinaatiston origo oli ”vasemmassa alakulmassa” ja näin X ja Y arvojen kasvaessa käyrä siirtyi ”oikealle yläkulmaan”. Tietokoneen näytöllä 0,0-piste on kuitenkin tyypillisesti vasemmassa ylänurkassa ja Y-koordinaattien kasvu vie näytöllä alaspäin. Siksi koordinaatistoista on syytä sopia etukäteen, ettei tule ongelmia integroitaessa eri henkilöiden toteuttamia moduuleja.

Rekursio

Luvussa 2 tutustuttiin toistorakenteisiin `for`, `while` ja `do-while`, jotka ovat normaali tapa käydä läpi sama koodilohko useita kertoja. Toistettava koodilohko voidaan toteuttaa myös aliohjelmana ja laittaa se kutsumaan itseään eli tehdä rekursiivinen aliohjelma. Rekursion hyviä puolia ovat sen tyypillisesti tiivis 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 yksinkertaisella rekursiivisella aliohjelmalla.

Esimerkki 4.7. Rekursio

```
#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);
}
```

Esimerkin tuottama tulos

Kun käänämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E4_7
Luvun 6 kertoma on 720.
```

Kuinka ohjelma toimii

Esimerkki kutsuu `kertoma`-aliohjelmaa parametrina luku 6. Koska 6 ei ole 0, siirtyy suoritus valintarakenteen `else`-haaraan, jossa on 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.

C-kielen ominaispiirteitä

C-kielen tyypillisiä piirteitä on osoittimien aktiivinen käyttö. Katsotaan asiaa seuraavaksi vähän laajemmin eli miksi osoittimia käytetään ja mikä on tämän kurssin osalta keskeinen osoittimen käyttötapa. Samoin C-kielessä määritellään usein uusia tietueita, joten katsotaan niiden käyttöä taulukoiden ja binaaritiedostojen kanssa. Tämä osuus demonstroi hyvin, miten erilaisilla rakenteilla pystyy tuottamaan tiivistä koodia, vaikka erityisesti binaaritiedostojen kanssa tulee olla varovainen siirrettävyysohjelmien välttämiseksi.

Osoittimien käyttö

C-kielessä käytetään tyypillisesti paljon osoittimia ja tähän on useita syitä. Ensinnäkin käsiteltäessä suuria tietomääriä tiedon kopiointi on hidas operaatio ja vaihtoehtona oleva osoittimen arvon välittäminen on nopea tapa saada tieto esimerkiksi aliohjelmaan. Tarpeeton isojen tiedostojen, esim. 10MB valokuvien, kopiointi voi myös johtaa muistin loppumiseen etenkin siirreltäessä suuria tietomääriä kuten tuhansia valokuvia. Toinen tyypillinen syy osoittimien käyttöön on se, että palauttamalla aliohjelmasta esimerkiksi kopioidun merkkijonon ensimmäisen merkin osoite, voidaan sitä mahdollisesti hyödyntää heti seuraavassa käskyssä eli nopeuttaa ohjelman toimintaa. Kolmanneksi osoittimien käyttö tarjoaa joustavuutta, sillä osoitin-konsepti toimii vastaavalla tavalla niin muuttujien kuin funktioidenkin kohdalla. Yhdellä muuttuja-osoitimella voi osoittaa eri tietoalkioihin ja vastaavasti funktio-osoitimella voi osoittaa eri funktioihin ja tehdä eri toimintoja tarpeen mukaan.

Tässä oppaassa keskitymme osoittimien käyttöön muuttujien yhteydessä ja niiden tarjoamaan joustavuuteen. Tyypillinen osoitinmuuttujan rooli on askeltaja, joka käy läpi esimerkiksi merkkijonon kaikki merkit aiemman esimerkin 2.12 mukaisesti. Tämän esimerkin oleelliset rivit näkyvät alla eli ohjelmassa määritellään ensin merkkitaulukko, jossa on kiinnostava tieto. Tämän jälkeen määritellään merkkiosoitin, jolla käydään läpi kaikki merkkijonon merkit yksitellen ja tulostetaan ne näytölle. Osoitin toimii liukurina, joka liukuu tietoalkiosta toiseen ja mahdollistaa tietoalkion käsittelyn halutulla tavalla. Tulemme palaamaan tähän asiaan myöhemmin muiden tietorakenteiden yhteydessä, joten alla olevaan yksinkertaiseen esimerkkiin kannattaa tutustua kunnolla, jotta liukuri-idea pystyy laajentamaan myöhemmin muihin tietorakenteisiin.

```
char str[] = "C-kieli on kivaa.";
char *ptr = str;

while (*ptr != '\0') {
    printf("%c\n", *ptr);
    ptr++;
}
```

Komentoriviparametrien `argv`-osuus aiheuttaa ajoittain hämmennystä ja aina ei ole selvää, miten se pitäisi määritellä. Alla on kolme eri vaihtoehtoa, joista periaatteessa kaikkia voisi käyttää, mutta kääntäjä ei hyväksy viimeistä vaihtoehtoa, jossa `argv` on kaksiulotteinen taulukko. Näin ollen `**argv` sekä `*argv[]` ovat kääntäjän kannalta hyväksyttäviä vaihtoehtoja ja molemmat myös toimivat ongelmitta. Näistä ”oikeampi” määrittely on `*argv[]`, jonka mukaan `argv` on taulukko osoittimia merkkitaulukoihin, joissa on kaikki komentorivin parametrit ohjelman käytettävissä.

```
int main(int argc, char *argv[]) // Oikea määrittely, käytä tätä
int main(int argc, char **argv)
int main(int argc, char argv[][])
```

Kannattaa muistaa, että `*` tarkoittaa osoitinta ja `[]` on taulukon tunnus eli osoitin haluttuun taulukon alkioon. Näin ollen `*argv` ja `argv[]` osoittavat molemmat yhteen merkkiin, `char`, ja kun tästä lähdetään etenemään merkkijonon loppua eli `NULL`-merkkiä kohti, molemmat merkinnät toimivat samalla tavoin. Käytännössä ohjelmoijan kannattaa pyrkiä mahdollisimman oikeisiin merkintöihin, jotta kääntäjä ja muut aputyökalut voivat varoittaa mahdollisista koodin ongelmista ja riskeistä. Näin ohjelmoijalla on paremmat mahdollisuudet eliminoida ongelmat ennen kuin ohjelma lähtee asiakkaalle ja tuotantoon, mikä jälkeen korjauskustannukset nousevat merkittävästi.

Tietue ja binaaritiedosto

Tämän luvun alussa käsitelimme taulukoita ja totesimme, että taulukon alkioina voi olla mitä tahansa tietotyyppisiä. Kun tämän jälkeen esittelimme tietueen, heräsi luonnollisesti kysymys siitä, voiko myös tietuetta käyttää taulukon alkiona. Jottei tästä asiasta jää epäselvyyttä, määritellään esimerkissä 4.8 uusi tietue, luodaan sen avulla taulukko ja talletetaan taulukon tiedot tiedostoon. Käytämme tällä kertaa binaaritiedostoa, sillä C-kielen toteutuksesta johtuen luotu taulukko voidaan tallettaa yhdellä käskyllä.

Esimerkki 4.8. Tietuetaulukko ja binaaritiedoston käsittely

```
#include <stdio.h>
#include <stdlib.h>
#define LKM 3

typedef struct henkilo {
    char aNimi[30];
    int iIka;
} HENKILO;

int main(void) {
    HENKILO hlot[LKM] = {{ "Ville", 5 }, { "Kalle", 6 }, { "Erkki", 56 } }; // 1
    HENKILO *pHlo = hlot;

    FILE *tiedosto;
    printf("Kirjoitetaan binaaritiedosto.\n"); // 2
    if ((tiedosto = fopen("Hlo.bin", "wb")) == NULL) { // 3
        perror("Tiedoston avaaminen epäonnistui");
        exit(1);
    }
    fwrite(hlot, sizeof(hlot), 1, tiedosto); // 4
    fclose(tiedosto);

    printf("Nimi '%s' ja ikä %d.\n", pHlo->aNimi, pHlo->iIka); pHlo++;
    printf("Nimi '%s' ja ikä %d.\n", pHlo->aNimi, pHlo->iIka); pHlo++;
    printf("Nimi '%s' ja ikä %d.\n", pHlo->aNimi, pHlo->iIka);

    return(0);
}
```

Edellä olevassa esimerkissä *kirjoitettiin* taulukko binaaritiedostoon, mutta sen *lukeminen* on yhtä helppoa. Esimerkin oikeassa reunassa on kommenteilla merkattu rivit, joita muokkaamalla alla olevalla tavalla saadaan binaaritiedoston kirjoittavasta ohjelmasta tehtyä binaaritiedoston lukeva ohjelma. Rivikohtaiset muutokset on merkattu molempiin koodeihin keltaisella, jotka ovat (1)

luettava tietorakenne voidaan jättää määrittelyvaiheessa tyhjäksi, sillä lukeminen täyttää taulukon tiedoilla; (2) käyttäjälle kerrotaan, että nyt tiedosto *luetaan*; (3) tiedosto tulee avata lukemista varten ja (4) lukeminen tapahtuu *fread*-käskyllä.

```
HENKILO hlot[LKM]; // 1
printf("Luetaan binaaritiedosto.\n"); // 2
if ((tiedosto = fopen("Hlo.bin", "rb")) == NULL) { // 3
    fread(hlot, sizeof(hlot), 1, tiedosto); // 4
}
```

Ohjelmassa käytetään taulukon yhteydessä osoitinta demonstroimaan osoittimen käyttöä tietueen kanssa. Tämä ratkaisu ei ole optimaalinen, sillä esimerkiksi taulukon alkioiden tietojen tulostus toistorakenteella olisi johtanut lyhyempään ohjelmaan.

Pienen C-ohjelman tyyliohjeet

Taulukon varaaminen. Taulukon varaaminen edellyttää tietoa taulukon koosta. Siksi taulukkoa varatessa pitää sen koko ilmoittaa numerolla eikä sitä voi laskea ohjelman suorituksen aikana – tämä on dynaamista muistin varaamista, johon palataan seuraavassa luvussa.

Tiedostorakenne. Tietueet tulee määritellä tämän oppaan yhteydessä globaaleiksi rakenteiksi ja laittaa ne tiedostoon `include`-osuuden jälkeen. Tietueita välitetään tyyppillisesti parametrinä aliohjelmien välillä, joten siksi ne on hyvä määritellä ennen aliohjelmiä.

Tietueen nimeäminen. Tietueet kannattaa nimetä käyttäen pieniä kirjaimia, joten tietueesta tehdylle tietotyyppille voi antaa saman nimen isoilla kirjaimilla. Uusi tietotyyppi kannattaa tehdä tietueesta, jotta sitä on helppo käyttää jatkossa sekä uusien muuttujien että osoittimien määrittelyyn.

Yhteenveto

Kerrataan luvun osaamistavoitteet ja esimerkkien aiheet sekä katsotaan keskeiset asiat kokoava ohjelmointiesimerkki.

Osaamistavoitteet

Tässä luvussa tutustuimme uusiin tietorakenteisiin ja algoritmeihin. Tietorakenteista taulukot, tietueet ja komentoriviparametrit on nyt käyty läpi ja sen lisäksi katsottu, miten uusia tietotyypppejä määritellään sekä miten tietueeseen viitataan osoittimella. Algoritmeissa perehdyimme rekursioon ja matriisin läpikäyntiin kahdella toistorakenteella sekä kertosilmän käytön liukurina tietorakenteen alkioihin.

Käsitellyt asiat

Tässä luvussa kävimme läpi C-ohjelmointiin liittyen seuraavat asiat:

- Numeerinen taulukko: Esimerkki 4.1
- Tietueen määrittely ja käyttö: Esimerkki 4.2, 4.9
- Tietotyyppin määrittely tietueesta: Esimerkki 4.3, 4.9
- Komentoriviparametrit ja merkkijono numeroksi: Esimerkki 4.4, 4.9

- Tietue-osoitin: Esimerkki 4.5, 4.8, 4.9
- Matriisin läpikäynti: Esimerkki 4.6
- Rekursio: Esimerkki 4.7
- Tietuetaulukko ja binaaritiedoston käsittely: Esimerkki 4.8
- Tietue arvo- ja muuttujaparametrinä: Esimerkki 4.9

Kokoava esimerkki

Kokoava esimerkki hyödyntää komentoriviparametrejä välittämällä komentorivillä ohjelman nimen jälkeen olevan luvun ohjelmalle muutettavien tietojen määräksi. Ohjelmassa määritellään tietue ja tietotyyppi sekä taulukko tietueeseen perustuen. Sen jälkeen ohjelma kysyy käyttäjältä tiedot taulukon tietueisiin ja tulostaa ne eri aliohjelmassa. Tässä kannattaa kiinnittää huomiota em. määrittelyiden lisäksi siihen, miten `muuta`-aliohjelmaan välitetään tietueen osoite muuttujaparametrinä tietojen lukemista varten samalla tavoin kuin `scanf`-funktiolle. Vastaavasti tietojen tulostus tehdään `printf`-tyyliin lähettämällä `tulosta`-aliohjelmaan tietue arvoparametrinä. Näin ollen `tulosta`-aliohjelmassa tietueen tietoja voi muuttaa, mutta muutokset eivät välity takaisin pääohjelmaan. Tämä kannattaa kokeilla vaihtamalla ikää tulostusaliohjelmassa ja kutsumalla sitä uudestaan. Komentoriviparametrien osalta kannattaa huomata, että ensin varmistutaan oikeasta parametrien määrästä ja lopetetaan ohjelma, jos parametrien määrä on väärä.

Esimerkki 4.9. Tietue arvo- ja muuttujaparametrinä

```

#include <stdio.h>
#include <stdlib.h>
#define MAX 10

typedef struct henkilo {
    char aNimi[30];
    int iIka;
} HENKILO;

void muuta(HENKILO *pHlo) {
    printf("Anna nimi: ");
    scanf("%s", pHlo->aNimi);
    getchar();
    printf("Anna ikä: ");
    scanf("%d", &pHlo->iIka);
    getchar();
    return;
}

void tulosta(HENKILO hlo) {
    printf("Nimi on '%s' ja ikä on %d.\n", hlo.aNimi, hlo.iIka);
    return;
}

int main(int argc, char *argv[]) {
    HENKILO aHlot[MAX];
    int iLkm, i;

    if (argc != 2) {
        printf("Ohjelman käyttö: ./E4_9 <muutettavien tietojen määrä>\n");
        printf("Komentoriviparametrinä tulee antaa ohjelman nimen\n");
        printf("lisäksi muutettavien henkilötietojen määrä, lopetetaan.\n");
        exit(0);
    }
    iLkm=atoi(argv[1]);

    for (i=0; i < iLkm; i++)
        muuta(&aHlot[i]);

    i = 0;
    while (i < iLkm)
        tulosta(aHlot[i++]);

    return(0);
}

/* eof */

```


Luku 5. Muistinhallinta, tietotyypit

Tässä luvussa tutustumme dynaamiseen muistinhallintaan ja tutustumme laajemmin itse määriteltyihin tietotyypeihin. Aloitetaan tietotyypeistä, sillä ne ovat muistinhallinnan ydin.

Käyttäjän määrittelemät tietotyypit ja muistin käytöstä

Tähän asti olemme käyttäneet pääasiassa C-kielen perustietotyyppejä (mm. `int` ja `float`), mutta luvussa 4 tutustuimme rakenteisiin tietorakenteisiin eli taulukkoon ja tietueeseen. C-kielen taulukoissa on syytä muistaa, että kaikki taulukon alkiot ovat samaa tietotyyppiä, kun taas tietue mahdollistaa erilaisten tietotyyppien paketoinnin yhteen avainsanalla `struct`. Tietue on keskeinen tietorakenne C-kielessä ja siten siitä on olemassa myös toinen variantti nimeltään *yhdiste*. Yhdiste määritellään `union`-avainsanalla, jolloin sille varataan eniten tilaa vaativan muuttujan tarvitse muistialue, mutta käyttäjä voi käyttää yhdistettä tallentamaan minkä tahansa nimetyistä muuttujista. Tämä mahdollisuus ei ole nykyaikana kriittinen ominaisuus muistin määrän jatkuvasti kasvaessa, mutta tarjoaa ohjelmoijalle mahdollisuuden minimoida muistin käyttöä tarpeen tullen. Alla on lyhyt esimerkki yhdisteestä ja asiasta kiinnostuneet voivat paneutua siihen tarkemmin omatoimisesti.

Esimerkki 5.1. Yhdiste

```
union lukuja {
    int iLuku;
    double dLuku;
    char aNimi[30];
} yhdiste;

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

Yhdiste määritellään tietuetta vastaavalla tavalla eli `'struct tietue {};`' ja `'union yhdiste {};`'. Määrittelyn jälkeen yhdisteen alkioita voidaan käyttää pistenotaatiolla ja oleellista on muistaa, että **kun yhdisteen muuttujaan sijoitetaan arvo, on sieltä luettava saman muuttujan arvo**. Muutoin tulokset voivat yllättää ja sotkea ohjelman toiminnan.

Käsittelimme luvussa 2 esikäsittelijän `define`-käskyllä määritelty merkkijonovakiot, jolla voitiin määritellä vakioita aina yksi vakio käskyä ja riviä kohden. Mikäli ohjelmassa tarvitaan useita vakioita, esim. kuukausien tai viikonpäivien nimet, voi määrittely käydä työlääksi. Tilannetta helpottaa *lueteltu tyyppi* eli `enum`-käsky, joka antaa luetelluille merkkijoille arvoksi kokonaislukuja kasvattaen arvoa aina yhdellä. Ensimmäisen tunnuksen oletusarvo on 0.

Esimerkki 5.2. Lueteltu tyyppi

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

Esimerkissä 5.2 EI-vakion arvo on 0 ja KYLLA-vakion arvo 1, kun RISTI-vakion arvo on 4 ja RUUTU-vakion arvo 5 jne. Lueteltu tyyppi määritellään samalla rakenteella kuin tietue eli 'enum lueteltutyyppi {};'.

Rakenteisten tietotyyppien määrittelyn yhteydessä on käytettävä varattuja sanoja (struct, union, enum) sekä halutun tietotyypin yksilöivää tunnistetta. Jottei näitä varattuja sanoja tarvitse toistaa jatkuvasti, voidaan määritellä oma tietotyyppi typedef-käskyllä samalla tavoin kuin luvussa 4 tehtiin tietueiden kanssa. Seuraavassa esimerkissä 5.3 on pieni ohjelma, jossa käytetään yhdistettä ja lueteltuja tyypejä.

Esimerkki 5.3. Yhdiste ja lueteltu tyyppi ohjelmassa

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

// Tietorakenteen määrittely päätasolla eli globaalina rakenteena
union lukuja {
    int iLuku;
    double dLuku;
    char aNimi[30];
};

int main(void) {
    union lukuja yhdiste; // Yhdiste eli union
    int iLuku;
    // typedef union lukuja LUKUJA; // Vaihtoehtoinen määrittely typedef'llä
    // LUKUJA yhdiste2;

    yhdiste.iLuku = 4;
    iLuku = yhdiste.iLuku;
    strcpy(yhdiste.aNimi, "C-kielen yhdiste.");
    printf("Arvot: iLuku='%d', yhdiste.iLuku='%d', yhdiste.dLuku='%lf', nimi='%s'.\n",
        iLuku, yhdiste.iLuku, yhdiste.dLuku, yhdiste.aNimi);
    printf("Koot: int=%ld, double=%ld, merkkitaulukko=%ld, union = %ld.\n",
        sizeof(int), sizeof(double), sizeof(yhdiste.aNimi), sizeof(yhdiste));

    enum tosi { EI, KYLLA }; // Lueteltu tyyppi, kokeile myös esim. EI=5
    printf("%d, %d\n", EI, KYLLA);

    printf("Kiitos ohjelman käytöstä.\n");
    return(0);
}
```

Esimerkin tuottama tulos

```
un@LUT8859:~/Esimerkit$ ./E5_3
Arvot: iLuku='4', yhdiste.iLuku='1768631619',
yhdiste.dLuku='61952045015647896425696701969406695443709542720078914624288499348
53533773750784048962233106312560784752050697357962983291875652664346984641166684
2527754609170751312466368839355353814443452519808804792535092372343122508644352.
000000', nimi='C-kielen yhdiste.'.
Koot: int=4, double=8, merkkitaulukko=30, union = 32.
0, 1
Kiitos ohjelman käytöstä.
```

Äkkiseltään ohjelman tulosteessa ei ole järkeä, mutta se oli odotettuakin. Ensimmäisessä vaiheessa sijoitimme yhdisteen kokonaisluvulle arvon, sijoitimme sen seuraavaksi omaan muuttujaan ja tulostimme sen näytölle. Tämä osuus meni hyvin eli ohjelma tulosti numeron 4 näytölle. Tämän jälkeen sijoitimme yhdisteen merkkitaulukkoon merkkijonon, joka tulostui `printf:n` viimeisenä arvona eli sekin toimi halutulla tavalla. Näiden kahden välissä tulostetut kokonaisluku ja kaksoistarkkuuden liukuluku eivät kuitenkaan tuottaneet järkevää tulostetta, mutta rehellisyyden nimissä tämä oli odotettua kahdestakin syystä. Ensinnäkin liukuluvulle ei asetettu arvoa eli se oli alustamaton muuttuja, jolloin tulosta ei lähtökohtaisesti voi tietää etukäteen. Toinen ja tässä tapauksessa määräävä tekijä on se, että merkkijono `aNimi` on yhdisteen suurin jäsenmuuttuja, joten se käyttää koko yhdisteen varaaman muistialueen. Näin ollen tulostettaessa `iLuku` ja `dLuku` muuttujat `printf`-katsoo kokonaisluvun ja kaksoistarkkuuden liukuluvun varaamaan muistialueen ja tulkitsee tällä alueella olevat bitit kokonaislukuna ja liukulukuna. Kuten tulosteesta näkyy, tässä ei ollut nyt mitään järkeä, mutta ohjelma toimi niin kuin ohjelmoija sen koodasi. Yhteenvetona yhdisteestä tulee muistaa, että sen toiminta perustuu vain yhden muuttujan käyttämiseen kerrallaan. Todellisuudessa yhdistettä tarvitsee nykyään harvoin ja jos sen kanssa joutuu toimimaan, kannattaa varata aikaa aiheeseen tutustumiseen ongelmien välttämiseksi.

Yhdisteen lisäksi edellisessä esimerkissä näkyi luetellun tyyppin käyttö. Lueteltu tyyppi sopii hyvin tiettyihin tehtäviin ja jos nimetyille vakioille on käyttöä, kannattaa tähän tutustua tarkemmin.

Tarkkasilmäinen ohjelmoija huomasi, että kaiken edellä olevan mukaan unionin olisi pitänyt olla 30 tavun kokoinen, mutta sille olikin varattu 32 tavua. Tämä johtuu siitä, että ohjelman tekemisessä käytetyssä ohjelmointiympäristössä muistin varausyksikkö on 8 tavua, joten muistia varatessa se tulee aina 8 tavun ryhmissä. Näin ollen yhdisteen koko voisi olla 0, 8, 16, 24, 32, 40 jne. tavua. Tämän asian voi tarkistaa vaihtamalla `aNimi`-taulukon koon esim. 23, 24 ja 25 merkin kokoiseksi ja katsomalla sille varatun muistin määrä suorittamalla ohjelma. C-kieli toimii siis varsin lähellä käyttöjärjestelmätasoa ja siksi sen tason ratkaisut saattavat vaikuttaa ohjelman toimintaan kuten tässä näkyy.

Dynaaminen muistinhallinta

Puhuimme muistinvarauksesta merkkijonojen yhteydessä ja totesimme, että helpointa on käyttää staattisia taulukoita, vaikka niiden kanssa on kaksi rajoitetta muistin käyttöön liittyen. Ensinnäkin voimme varata suuren merkkitaulukon, mutta lyhyen merkkijonon kohdalla tuhlaamme tilaa. Toiseksi varaamalla liian vähän muistia merkkijonolle ohjelma ei toimi oikein. Näistä kumpikaan vaihtoehto ei houkuttele, joten tarvitaan parempi tapa hoitaa asia ja nyt on tullut aika katsoa sitä. Tutustutaan siis dynaamiseen muistinhallintaan, joka tarkoittaa tarvittavan muistimäärän varaamista ajon aikana.

C-kielessä ajonaikainen muistinvaraus toteutetaan `stdlib`-kirjaston `malloc`-funktiolla. Funktio saa parametrina tarvittavan muistimäärän, joten voimme varata juuri oikean määrän muistia tarpeen mukaan. Tästä seuraa muna-kana -ongelma, eli miten voimme varata muistia merkkijonoa varten, jos meidän pitäisi ensin tietää sen pituus ja vasta tämän jälkeen voimme varata tarvittavan määrän muistia?

Tarvittava muistimäärä voidaan selvittää lukupuskurilla, joka on käytännössä yksi riittävän suuri staattisesti varattu merkkitaulukko standardisyötevirran lukuun. Lukupuskurin tulee olla niin iso, että siihen mahtuu kaikki käyttäjän antamat syötteet, muttei sen suurempi. Esimerkiksi 255 tavua eli merkkiä on useimmiten riittävä puskuri käyttäjän antaman syötevirran lukemiseen.

Tekstiedostojen yhteydessä voimme käyttää vastaava puskuria tai laskea ensin rivillä olevien merkkien määrän ja lukea ne vasta sen jälkeen, jolloin puskuria ei tarvita.

`malloc`-funktio palauttaa osoittimen varaamansa muistialueen alkuun. Luvussa 2 käsiteltiin muuttujan ja osoittimen eroa, mutta tiivistetysti muuttuja varaa muistia tietoa varten ja muuttujan arvoon päästään käsiksi muuttujan nimen sisältämän osoitteen avulla. Muuttujan osoite voidaan asettaa myös osoittimen arvoksi, jolloin tietoon pääsee käsiksi muuttujan tai osoittimen avulla. Kun muistia varataan `malloc`-funktiolla, annetaan funktiolle parametrina tarvittava muistin määrä ja `malloc` palauttaa varatun muistialueen osoitteen, joka on sijoitettava osoitinmuuttujan arvoksi. `malloc`'n palauttama osoite on aina tyyppiä `(void *)`, joten se tulee muuttaa oikean tyyppiseksi kirjoittamalla haluttu tietotyyppi `malloc`-sanan eteen eli tehdä tyyppimuunnos, `type-cast`, esim. `(char *)`. Lisäksi halutun kokoisen muistialueen vapaana olo ei ole itsestään selvää, joten muistinvarauksen onnistuminen on tarkistettava aina.

Esimerkissä 5.4 näkyy dynaamisen muistinvarauksen oleelliset asiat: määritellään osoitin haluttuun tietotyyppiin, selvitetään tarvittava muistimäärä, varataan muisti `malloc`'lla ja sijoitetaan saatu muistiosoite osoittimen arvoksi tyyppimuunnoksen jälkeen. Merkkijonon yhteydessä tulee aina muistaa varata yksi merkki loppumerkkiä eli `NULL`'ia varten ja dynaamisesti varattu muisti pitää vapauttaa aina ennen ohjelman loppumista. Esimerkissä näkyy myös testimerkkijono `aNimi` sekä sen sisältämän merkkijonon kopiointi `malloc`'lla varattuun muistialueeseen `strcpy`-käskyllä. C-kielessä merkkijonoa ei voi "sijoittaa" uuteen paikkaan muistissa vaan jokainen merkkijonon merkki on kopioitava uuteen muistialueeseen yksitellen ml. loppumerkki `NULL`.

Esimerkki 5.4. Dynaaminen muistinvaraus

```
char aNimi[30] = "Ville";
char* pMuistilohko = NULL;
int muistin_maara = (strlen(aNimi)+1) * sizeof(char);
pMuistilohko = (char *)malloc(muistin_maara);
strcpy(pMuistilohko, aNimi);
free(pMuistilohko);
```

Dynaaminen muistin varaus voi epäonnistua, joten varauksen onnistuminen on aina tarkistettava seuraavan esimerkin mukaisesti. `malloc` palauttaa `NULL`'n, jos varaus epäonnistuu, joten siinä tapauksessa tulee suorittaa virheenkäsittely. Tämän oppaan yhteydessä virheenkäsittely tarkoittaa virheilmoituksen antamista käyttäjälle `perror`-käskyllä ja ohjelman lopettamista `exit`-käskyllä. Sekä `exit` että `perror` käskyt löytyvät `stdlib`-kirjastosta.

Esimerkki 5.5. Dynaamisen muistinvarauksen virheenkäsittely

```

if ((pMuistilohko = (char *)malloc(muistin_maara)) == NULL) {
    perror("Muistinvaraus epäonnistui");
    exit(0);
}

```

Esimerkissä 5.6 on pieni ohjelma dynaamisella muistinkäsittelyllä. Tämä ohjelma toteuttaa aiemmin puhutun käyttäjän syötteenä antaman merkkijonon käsittelyn eli luetaan käyttäjän antama merkkijono ensin puskuriin, selvitetään saadun merkkijonon tarvitsema muistimäärä, varataan tarvittava muisti dynaamisesti ohjelman suorituksen aikana sekä siirretään saatu merkkijono siihen. Lopuksi ohjelma tulostaa muistinkäyttöstatistiikkaa ennen kuin se vapauttaa varatun muistin ja suoritus päättyy.

Esimerkki 5.6. Dynaaminen muistinvaraus ohjelmassa

```

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

int main(void) {
    char lukuPuskuri[255];
    char *pMuistilohko;
    int muistin_maara;

    printf("Anna merkkijono, max. 253 merkkiä:\n");
    fgets(lukuPuskuri, 255, stdin); // Huom. Rivinvaihto ja NULL -merkit

    lukuPuskuri[strlen(lukuPuskuri)-1] = '\0';
    muistin_maara = (strlen(lukuPuskuri)+1) * sizeof(char);
    if ((pMuistilohko = (char*)malloc(muistin_maara)) == NULL) {
        perror("Muistin varaus epäonnistui");
        exit(1);
    }
    strcpy(pMuistilohko, lukuPuskuri);

    printf("Merkkijonon lukemista varten varattiin muistia %ld tavua.\n",
        sizeof(lukuPuskuri));
    printf("Annoit merkkijonon '%s', joka käyttää muistia %d tavua.\n",
        pMuistilohko, muistin_maara);
    printf("Merkkijonon osoitin tarvitsee lisäksi muistia %ld tavua.\n",
        sizeof(pMuistilohko));

    free(pMuistilohko); // Dynaamisesti varattu muisti pitää aina vapauttaa
    return(0);
}

```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```

un@LUT8859:~/Opas$ ./E5_6
Anna merkkijono, max. 253 merkkiä:
Apumiehensijaisenvaramiespalveluvastaava
Merkkijonon lukemista varten varattiin muistia 255 tavua.
Annoit merkkijonon 'Apumiehensijaisenvaramiespalveluvastaava',
joka käyttää muistia 41 tavua.
Merkkijonon osoitin tarvitsee lisäksi muistia 8 tavua.

```

Kuinka koodi toimii

Ohjelman alussa luomme kolme muuttujaa, staattisesti varatun merkkitaulukon `lukuPuskuri`, merkki-osoittimen `pMuistilohko` sekä kokonaislukumuuttujan `muistin_maara`, joiden avulla siirrämme käyttäjän syöttämän merkkijonon dynaamisesti varattuun muistiin. Idea on käyttää yhtä staattisesti varattua merkkijonotaulukkoa lukemaan useita käyttäjän antamia tietoja ja tämä taulukko niin suuri, että siihen mahtuu suurin käyttäjän kerralla antama tietomäärä. Kun käyttäjä on antanut 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 otamme käyttäjältä merkkijonon, tallennamme sen muuttujaan `lukuPuskuri` ja poistamme siitä `fgets`'n lukeman rivinvaihtomerkin. Tämän jälkeen laskemme merkkijonon pituuden funktiolla `strlen`, joka laskee annetun merkkijonon pituuden päättävään NULL-merkkiin asti, joten lisäämme siihen NULL-merkin tilan sekä kerromme merkkien määrän yhden merkin tarvitsemalla muistin määrällä.

```
muistin_maara = (strlen(lukuPuskuri)+1) * sizeof(char);
```

Kun tarvittava muistimäärä on tiedossa, voimme varata muistin `malloc`-käskyllä

```
pMuistilohko = (char *)malloc(muistin_maara)
```

Tämä käsky koostuu itse asiassa seuraavista osista:

```
tallennuspaikka = (tallennettava_tyyppi)malloc(varattava_tila)
```

Ilmoitamme siis muistinvarausfunktiolle, että tarvitsemme muistia `muistin_maara` yksikköä, muutamme saatavan muistialueen osoitteen tyyppin merkkiosoittimeksi (`char *`) ja sijoitamme sen osoitteen `pMuistilohko`'n osoittamaan paikkaan muistissa. Tällä käskyllä saamme osoittimen `pMuistilohko`'n osoittamaan paikkaan, jossa on nyt meille varattua tilaa `muistin_maara` merkkiä.

Nyt kun olemme varanneet muistin, voimme kopioida `lukuPuskuri`-muuttujassa olevat merkit loppumerkkiin asti dynaamisesti varattuun muistiin `strcpy`-funktiolla. Kuten seuraavista `printf`-käskyistä näemme, voimme nyt käyttää `pMuistilohko`-muuttujan sisältämää tietoa normaalin muuttujan tavoin ja voisimme vapauttaa `lukuPuskuri`-merkkitaulukon muuhun käyttöön. Aivan viimeisenä asiana ohjelmassa vapautamme varaamamme muistin, jotta käyttöjärjestelmä tietää, että se voi käyttää tätä tilaa taas vapaasti.

Tässä esimerkissä merkkijonon pituus vaati erityistä tarkkuutta. Tämä johtuu siitä, että `fgets`-funktio otti käyttäjän antaman rivinvaihtomerkin (`'\n'`) mukaan merkkijonoon, sillä se oli tässä yhteydessä syötteen lopetusmerkki. Tämä merkki poistettiin merkkijonosta ylikirjoittamalla se NULL-merkillä. Lisäksi kannattaa muistaa, että `strlen`-funktio laskee merkkijonon pituuden loppumerkkiin asti (`'\0'`) sitä huomioimatta. Koska loppumerkille on kuitenkin varattava tilaa merkkijonossa, lisättiin merkkijonon pituuteen 1 merkki ennen tarvittavan muistimäärän laskua.

Huomioita muistinvarauksesta

Edellä oleva esimerkki demonstroi lyhyesti dynaamisen muistinhallinnan tehtävät. Muistinvaraukseen liittyy muutama muistettava asia:

- Vapauta aina käyttämäsi muisti `free(varattu_alue_osoitin)`-komennolla. Ohjelma, joka ei vapauta käyttämäänsä muistia, toimii virheellisesti eli ns. vuotaa muistia ja se voi haitata tietokoneen normaalia toimintaa.
- Jo varatun alueen kokoa voidaan muuttaa `realloc`-funktiolla, mikäli joskus tulee tarve vaihtaa muistialueelle tallennettua tietoa.
- `malloc` (ja `realloc`) palauttavat `NULL`-osoittimen, mikäli ne eivät onnistu varaamaan halutun kokoista muistialuetta. C-ohjelmoinnin hyvään tyyliin kuuluu, että aina muistinvarauksen yhteydessä tarkistetaan varauksen onnistuminen. Siksi muistinvarauksen yhteydessä on aina oltava virheen käsittely.

Dynaaminen muistinhallinta ja tietue

Dynaaminen muistinhallinta on normaalia toimintaa merkkijonojen kanssa ja toinen keskeinen dynaamisesti käytettävä tietorakenne on tietue. Siksi alla on lyhyt esimerkki oman tietueen ja tietotyypin määrittelystä, ko. rakenteen varaamisesta dynaamisesti, käytöstä osoittimen kanssa sekä varatun muistialueen vapauttamisesta. Nämä kaikki asiat on käsitelty jo aiemmin, mutta tässä ne näkyvät yhtenä kokonaisuutena.

Tässä ohjelmassa olisi luonnollisempaa varata tietue staattisesti, mutta tämä esimerkki valmistelee meitä tulevaa varten. Tulemme nimittäin jatkossa tarvitsemaan dynaamista muistin varausta, kun tarvittavien tietueiden määrä selviää vasta ajon aikana eikä staattinen muistin varaus pysty vastaamaan tähän tarpeeseen.

Esimerkki 5.7. Tietueen dynaaminen käyttö ajon aikana

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

struct henkilo {
    char aNimi[30];
    int iIka;
};
typedef struct henkilo HENKILO;

int main(void) {
    HENKILO *pHlo;

    // muistin dynaaminen varaaminen
    if ((pHlo = (HENKILO*)malloc(sizeof(HENKILO))) == NULL ){
        perror("Muistin varaus epäonnistui");
        exit(1);
    }

    strcpy(pHlo-&gtaNimi, "Ville");
    pHlo->iIka = 4;
    printf("Nimi on '%s' ja ika on '%d'.\n", pHlo->aNimi, pHlo->iIka);

    free(pHlo);
    printf("Kiitos ohjelman käytöstä.\n");
    return(0);
}
```

C-kielen ominaispiirteitä

Olemme käsitelleet merkkijonoja usein aiemmin eri näkökulmista, joten tehdään seuraavaksi yhteenveto merkkijonoista ja katsotaan samalla yksi tapa pilkkoa merkkijono sanoiksi. Sen jälkeen palataan taas muistinhallintaan ja katsotaan, miten aliohjelmassa voi varata muistia niin että sinne sijoitetut tiedot ovat käytettävissä myös kutsuvassa ohjelmassa.

Merkkijonoista

Merkkijonojen käsittely C-kielillä voi tuntua varsin työläältä, ja koska merkkijonot käsitellään periaatteessa merkki ja loppumerkki kerrallaan, ei työläyttä voi kieltää. Asiaa voi kuitenkin lähestyä toisesta näkökulmasta ja miettiä, voisiko merkkijonoja käsitellä jotenkin systemaattisesti tilanteesta riippuen. C-ohjelmoijan onkin hyvä huomata, että merkkijonot voidaan käytännössä luokitella kolmeen erilaiseen tapaukseen, joihin kaikkiin sopii hieman erilaiset perustekniikat.

Merkkijonojen yksinkertaisin tapaus on yksittäinen sana kuten nimi, esim. ”Ville”. Näiden käsittely on selkeää `scanf`- ja `printf`-käskyillä `%s`-formaattilla, kunhan muistaa varata merkkijonolle riittävän ison taulukon ja huomioida loppumerkin. Muita tyypillisiä ongelmia on kaksi. Ensinnäkin `scanf`-lukee merkkijonon whitespace’iin asti (välilyönti, sarkainmerkki, rivinvaihto) ja toiseksi se jättää rivinvaihtomerkin lukupuskuriin. Toisaalta sanan erottimena on tyypillisesti välilyönti, joten sen ei pitäisi yllättää ja rivinvaihtomerkin saa pois puskurista esim. `getchar`-käskyllä.

Usein merkkijono koostuu useista sanoista ja luontevaksi loppumerkiksi käyttäjäsyötteelle tulee tällöin rivinvaihtomerkki. Tässä tapauksessa olemme lukeneet merkkijonon `scanf`’n sijaan `fgets`:llä. Myös `scanf`-käskyn hyväksymät merkit voidaan määritellä formaatissa, esim. `scanf("%255[A-Za-z0-9 \\t]", merkkijono)`. Aloittelevalla ohjelmoijalla tämä voi näyttää vaikealta, mutta Unix-osaaja tunnistaa *regular expression*’n ja siten esitystapa on tuttu ja turvallinen. `fgets` on käytännöllinen ja turvallinen funktio, sillä se toimii samalla tavalla sekä näppäimistösyötteelle eli `stdin`-tietovirralla kuin tekstitiedostoillekin. Turvallisuus taas tulee siitä, että `fgets`’lle ilmoitetaan luettavien merkkien maksimimäärä eikä näin ollen ylivuoto-ongelmia pääse syntymään. Siksi rivejä luetaan tyypillisesti `fgets`’llä ja oleellista on muistaa, että se poistaa tietovirrasta myös rivinvaihtomerkin ja ottaa sen mukaan luettuun merkkijonoon. Näin ollen se pitää tyypillisesti poistaa merkkijonosta, joka onnistuu esim. kirjoittamalla sen päälle loppumerkki.

Tekstitiedostoja käytetään usein tiedon jakamiseen ja tällöin ne tallennetaan tyypillisesti CSV- eli Comma Separated Values -muodossa. Käytännössä yhdellä rivillä on useita tietoalkioita, jotka erotetaan toisistaan erotinmerkillä, ja kaikilla riveillä on vastaavia tietoja. Tämä tiedon esitysmuoto vastaa taulukkolaskentaa, jossa tiedot esitetään riveinä ja sarakkeina eli yhden rivin sarakkeet muodostavat CSV-tiedoston yhdellä rivillä olevat erotinmerkillä erotetut tietoalkiot.

Englanninkielisen nimen mukaan kyseessä on *pilkulla erotetut arvot* -rakenne, mutta koska Suomen kielessä käytetään desimaalipilkkua, ei pilkku ole sopiva erotin tässä ympäristössä. Sen sijaan erottimena käytetään tyypillisesti puolipistettä `;`, sarkainmerkkiä `\t` tai muuta merkkiä, jota tiedoissa ei muutoin esiinny. Näin ollen C-ohjelmien CSV-tiedostojen käsittely perustuu tyypillisesti tiedoston lukemiseen `fgets`-funktioilla ja rivillä olevien tietojen erotteluun `strtok`-funktioilla.

`strtok`-funktioita ei ole käsitelty aiemmin tässä oppaassa, joten seuraavassa esimerkissä näkyy sen käyttö yksinkertaisen CSV-tiedoston lukemiseen. Funktio saa 2 parametriä, joista ensimmäinen on käsiteltävän merkkijonon ensimmäisen muistipaikan osoite ja toinen etsittävä lopetusmerkki. Funktio palauttaa osoittimen ensimmäisen merkkijonon alkuun eikä loppumerkkiin päättyvä

merkkijono sisällä erotinmerkkiä. Huomaa, että toisella ja sitä seuraavilla kerroilla samaa merkkijonoa käsiteltäessä `strtok:n` ensimmäisen parametrin tulee olla `NULL`, jotta funktio tietää jatkaa saman merkkijonon käsittelyä. Esimerkissä toinen merkkijono loppuu rivinvaihtomerkkiin yksinkertaisen tiedoston takia. Tiedoston rakenne on seuraava: tiedostossa on yhdellä rivillä kaksi tietoalkiota, jotka on erotettu puolipisteellä, rivi loppuun rivinvaihtomerkkiin ja tiedosto lopussa on tyhjä rivi. Alla näkyy tiedoston alusta kaksi riviä, jotka ohjelma erottelee ja tulostaa.

```
Ville;3
```

```
Kalle;8
```

Esimerkki 5.8. Merkkijonon pilkkominen tietoalkioiksi

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

#define LEN 80

int main(void) {
    FILE *tiedosto;
    char aRivi[LEN], *p1, *p2;

    printf("Lue tiedosto tst.csv:\n");
    if ((tiedosto = fopen("tst.csv", "r")) == NULL) {
        perror("Tiedoston avaaminen epäonnistui");
        exit(1);
    }

    while (fgets(aRivi, LEN, tiedosto) != NULL) {
        printf("%s", aRivi);
        if ((p1 = strtok(aRivi, ";")) == NULL) {
            printf("Merkkijonon '%s' pilkkominen epäonnistui.\n", aRivi);
            exit(1);
        }
        if ((p2 = strtok(NULL, "\n")) == NULL) {
            printf("Merkkijonon '%s' pilkkominen epäonnistui.\n", aRivi);
            exit(1);
        }
        printf("p1 '%s' ja p2 '%s'\n", p1, p2);
    }

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

Huomaa `strtok`'iin liittyvä virheenkäsittely. Tässä oppaassa `strtok'n` yhteydessä ei ole välttämätöntä käyttää virheenkäsittelyä, mutta se on kuitenkin suositeltavaa. Virheenkäsittely helpottaa ongelman paikallistamisessa sen syntykohtaan sen sijaan, että epäonnistunut arvon tunnistus paljastuu vasta myöhemmin sitä käytettäessä. Laajempi esimerkki CSV-tiedoston lukemisesta ja käsittelystä `strtok`-funktioilla on tämän luvun kokoavassa esimerkissä.

Tietorakenteet ja aliohjelmat

C-ohjelmointi pyörii paljolti muistinkäsittelyn ympärillä. Muistia voidaan varata staattisesti tai dynaamisesti ja se, miten ja missä muisti on varattu vaikuttaa sen käytössä olemiseen. Näin ollen

aloittelevan ohjelmoijan on joskus vaikea hahmottaa, miten esim. aliohjelmien kanssa muistia tulee varata ja käyttää.

Pienissä ohjelmissa tyypillisin tapa muistin varaamisen kannalta on muistin varaus pääohjelmassa staattisesti ja tiedon välittäminen aliohjelmiin muuttuja- tai arvoparametreina. Tästä hyviä esimerkkejä ovat `printf`- ja `scanf`-funktiot, joista `printf` käyttää arvoparametreja ja `scanf` muuttujaparametreja. Arvoparametrien kohdalla aliohjelmassa käsitellään alkuperäisten muuttujien kopioita eli arvoja, ja siten niiden muutoksia eivät palaudu kutsuvaan ohjelmaan. Muuttujaparametrien kohdalla aliohjelmaan välitetään alkuperäisen muuttujan osoite muistissa ja näin ollen tehdyt muutokset siirtyvät takaisin kutsuvaan ohjelmaan. Nämä periaatteet pätevät kaikissa ohjelmakutsuissa eli välittämällä muistipaikan osoite muuttujaparametrinä aliohjelmaan, tietoa voidaan muuttaa ja se palautuu takaisin pääohjelmaan. Välittämällä muuttujan arvo aliohjelmaan arvoparametrinä, eivät mahdolliset muutokset palaudu kutsuvaan ohjelmaan.

Kutsuva ohjelma voi varata muistin muuttujille ja välittää muuttujat aliohjelmiin parametreinä niin kuin luvun 4 kokoavassa esimerkissä tehtiin. Tässä ohjelmassa varattiin 10-alkion tietuetaulukko pääohjelmassa, jonka jälkeen muisti tiedoille oli varattu. Kun taulukon tietueiden arvoja haluttiin muuttaa `muuta`-aliohjelmassa, välitettiin aliohjelmalle yhden taulukon alkion osoite eli kutsu oli muotoa `muuta(&hlot[0])` ja tietoja saatiin muutettua. Tulostettaessa käytimme arvoparametreja ja muotoa `tulosta(hlot[0])` ja jos tällöin olisimme muuttaneet tietoja aliohjelmassa, eivät muuttuneet tiedot olisi olleet käytössä enää kutsuvassa ohjelmassa.

Seuraavassa esimerkissä on toinen vaihtoehto saada tietoja aliohjelmasta kutsuvaan ohjelmaan dynaamisen muistinvarauksen avulla. Ohjelma varaa pääohjelmassa vain osoittimen haluttuun tietorakenteeseen ja kutsuttava aliohjelma `kysyTiedot()` varaa tarvittavan muistin dynaamisesti `malloc`'lla, täyttää tietorakenteen tiedot ja palauttaa varatun muistialueen osoitteen kutsuvaan ohjelmaan. Näin kutsuvassa ohjelmassa voidaan käyttää aliohjelmassa kysyttyjä tietoja ja vapauttaa varattu muistialue, kun sitä ei enää tarvita.

Esimerkki 5.9. Tietojen kysyminen ja muistin varaus aliohjelmassa

```

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

typedef struct henkilo {
    char aNimi[30];
    int iIka;
} HENKILO;

HENKILO *kysyTiedot() {
    HENKILO *pHlo;

    // Muistin varaaminen tietoja varten dynaamisesti
    if ((pHlo = (HENKILO*)malloc(sizeof(HENKILO))) == NULL ) {
        perror("Muistin varaus epäonnistui");
        exit(1);
    }

    // Tietojen kysyminen ja tallettaminen tietueeseen
    printf("Anna nimi: ");
    scanf("%s", pHlo->aNimi);
    getchar();
    printf("Anna ikä: ");
    scanf("%d", &pHlo->iIka);

    return pHlo; // Tietueen osoite palautetaan kutsuvaan ohjelmaan
}

int main(void) {
    HENKILO *ptr;

    // Tietojen kysyminen aliohjelmassa ja tietueen osoitteen talteen otto
    ptr = kysyTiedot();
    printf("Nimi on '%s' ja ikä on %d.\n", ptr->aNimi, ptr->iIka);

    free(ptr); // Dynaamisesti varattu muisti pitää vapauttaa ohjelman lopuksi
    return(0);
}

```

Pienen C-ohjelman tyyliohjeet

Dynaaminen muistinhallinta. Dynaamisen muistinhallinnan kannalta oleellista on aina tarkistaa muistivarauksen onnistuminen ja varatun muistin vapauttaminen ohjelman lopuksi. Normaalin virheen käsittelyn mukaisesti (1) muistinvarauksen onnistuminen pitää tarkistaa ja (2) epäonnistumisesta tulee kertoa käyttäjälle sekä (3) ohjelman lopetetaan `exit`-käskyllä. Kaikki `m/re/calloc`-funktioilla varatut muistialueet tulee vapauttaa ohjelman lopuksi `free`-käskyllä, jottei ohjelma ala vuotamaan muistia.

Merkkijonon pilkkominen. Merkkijono voidaan jakaa alimerkkijonoihin `strtok`-käskyllä. Ohjelman ongelmien selvittämisen kannalta jokaisen `strtok`-käskyn onnistuminen kannattaa tarkistaa normaalilla virheenkäsittelyllä, sillä muutoin isossa tiedostossa olevan yksittäisen virheellisen alkion aiheuttamat ongelmat voivat olla työläitä paikallistaa.

Yhteenveto

Kerrataan luvun osaamistavoitteet ja esimerkkien aiheet sekä katsotaan keskeiset asiat kokoava ohjelmointiesimerkki.

Osaamistavoitteet

Tämän luvun keskeisin asia on dynaaminen muistinhallinta eli muistin varaaminen, varauksen onnistumisen tarkistaminen ja muistin vapauttaminen käytön jälkeen. Omien tietotyyppien osalta tutustuimme yhdisteeseen ja lueteltuun tyyppiin.

Tässä luvussa vedettiin lisäksi yhteen merkkijonojen käsittelyä sekä tiedon välittämistä aliohjelmien välillä tietorakenteita hyödyntäen. Näitä asioita on käsitelty jo aiemmin, mutta asioiden vaiheittainen käsittely on saattanut jättää kokonaiskuvan yksityiskohtien varjoon. Merkkijonojen kannalta on oleellista tunnistaa käsiteltävän merkkijonon rakenne ja mitä sille on tarkoitus tehdä, sillä sanat, rivit ja tietoalkioista muodostuvat rivit käsitellään C-ohjelmissa usein eri funktioilla. Tietojen siirtäminen ohjelmien välillä edellyttää toimivaa muistinhallintaratkaisua ottaen huomioon staattisen ja dynaamisen muistinhallinnan, arvo- ja muuttujaparametrit sekä rakenteiden osoitteiden ja osoittimien sopivan käytön.

Käsitellyt asiat

Tässä luvussa kävimme läpi C-ohjelmointiin liittyen seuraavat asiat:

- Yhdiste, lueteltu tyyppi ja muistinkäyttö: Esimerkki 5.1, 5.2, 5.3
- Dynaaminen muistinhallinta: Esimerkki 5.4, 5.5, 5.6, 5.10
- Tietueen dynaaminen käyttö: Esimerkki 5.7, 5.10
- Merkkijonon pilkkominen strtok’lla: Esimerkki 5.8, 5.10
- Tietojen kysyminen ja muistin varaus aliohjelmassa: Esimerkki 5.9
- Tekstitiedoston lukeminen dynaamisesti varattuun tietuetaulukkoon: Esimerkki 5.10

Kokoava esimerkki

Kokoava esimerkki lukee CSV-tiedoston ja sijoittaa kiinnostavat tiedot tietueen jäsenmuuttujiin. Tietueet ovat dynaamisesti varattavassa taulukossa, jonka kokoa kasvatetaan aina kun uusi rivi luetaan. Muistin varaus suoritetaan aliohjelmassa, joka palauttaa taulukon osoitteen kutsuvaan ohjelmaan. Tässä ohjelmassa kannattaa kiinnittää huomio taulukon osoitteeseen, sillä muistia varatessa oleellista on varattavan muistin määrän lisäksi varatun alueen sijainti muistissa eli osoite. Toisaalta ohjelma käsittelee tietoja tietuetaulukkona eli tietueisiin viitataan indekseillä. Tämä on mahdollista, sillä taulukko koostuu samankokoisista tietueista ja siirtyminen seuraavaan taulukon alkioon voidaan tehdä muistipaikkojen osoitteilla eli osoittimilla tai taulukon alkioden indeksejä käyttäen. Tämä ero näkyy myös aliohjelmien parametreissa eli osoitinta käyttävässä aliohjelmassa taulukko on määritelty `*pTbl:ksi` kun taas indeksejä käyttävässä aliohjelmassa se on määritelty `aTbl[]:ksi`.

Ohjelman lukeman tiedoston yksi rivi on alla. Tietueen määrittelyssä näkyy tässä ohjelmassa kiinnostavat tietoalkiot eli CSV-tiedoston kolme saraketta.

01-01-2019;2859;10587;7.82;57;577;256;44;47;1497

Esimerkki 5.10. Tekstitiedoston luku dynaamisesti varattuun tietuetaulukkoon

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define LEN 80

typedef struct tiedot {
    char aPvm[11]; // CSV:ssä sarake 1
    int iAskelia; // CSV:ssä sarake 3
    double dMatka; // CSV:ssä sarake 4
} TIEDOT;

TIEDOT *lisaaTaulukkoon(TIEDOT *pTbl, int iLkm, char *pRivi) {
    char *p1, *p2, *p3;

    // Dynaaminen muistin varaus taulun kokoa muuttamalla/kasvattamalla
    if ((pTbl = (TIEDOT*)realloc(pTbl, iLkm*sizeof(TIEDOT))) == NULL ){
        perror("Muistin varaus epäonnistui");
        exit(1);
    }

    // Tietojen irroittaminen merkkijonosta
    if ((p1 = strtok(pRivi, ";")) == NULL) {
        printf("Merkkijonon '%s' pilkkominen epäonnistui.\n", pRivi);
        exit(1);
    }
    if ((p2 = strtok(NULL, ";")) == NULL) { // Tätä ei käytetä, otetaan pois
        printf("Merkkijonon '%s' pilkkominen epäonnistui.\n", pRivi);
        exit(1);
    }
    if ((p2 = strtok(NULL, ";")) == NULL) {
        printf("Merkkijonon '%s' pilkkominen epäonnistui.\n", pRivi);
        exit(1);
    }
    if ((p3 = strtok(NULL, ";")) == NULL) {
        printf("Merkkijonon '%s' pilkkominen epäonnistui.\n", pRivi);
        exit(1);
    }

    // Tietojen tallettaminen taulukon uusimpaan alkioon
    strcpy(pTbl[iLkm-1].aPvm, p1);
    pTbl[iLkm-1].iAskelia = atoi(p2);
    pTbl[iLkm-1].dMatka = atof(p3);

    return pTbl; // Uuden taulukon osoite palautetaan kutsuvaan ohjelmaan
}

void tulostaTaulukko(TIEDOT aTbl[], int iLkm) {
    int i, iAskeliaYht=0;
    double dMatkaYht=0;

    for (i=0; i < iLkm; i++) {
        printf("%s liikuttiin %d askelta ja %.2f km.\n",
            aTbl[i].aPvm, aTbl[i].iAskelia, aTbl[i].dMatka);
        iAskeliaYht += aTbl[i].iAskelia;
        dMatkaYht += aTbl[i].dMatka;
    }
    printf("Yhteensä liikuttiin %d askelta ja %.2f km.\n", iAskeliaYht, dMatkaYht);

    return;
}

```

C-ohjelmointiopas versio 2.1

```
int main(void) {
    FILE* tiedosto;
    TIEDOT* pTaulukko=NULL;
    char aRivi[LEN], aNimi[] = "L5Data2.csv";
    int iLkm=0;

    printf("Luetaan tiedosto '%s'.\n", aNimi);
    if ((tiedosto = fopen(aNimi, "r")) == NULL) {
        perror("Tiedoston avaaminen epäonnistui");
        exit(1);
    }
    while (fgets(aRivi, LEN, tiedosto) != NULL) {
        iLkm++;
        pTaulukko = lisaaTaulukkoon(pTaulukko, iLkm, aRivi);
    }
    fclose(tiedosto);

    printf("Tulostetaan taulukon alkiot:\n");
    tulostaTaulukko(pTaulukko, iLkm);
    free(pTaulukko);

    return(0);
}
/* eof */
```

Luku 6. Linkitetty lista ja aika C-ohjelmissa

Tässä luvussa perehdymme linkitettyyn listaan ja sen toteutukseen C-kielillä sekä katsomme ajan käsittelyn perusasiat. Luku päättyy kokoavaan yhteenvetoon.

Linkitetty lista

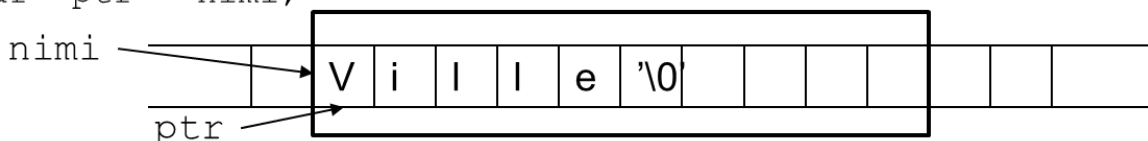
Aloitetaan linkitettyyn listaan tutustuminen kertaamalla taulukon toteutus ja vertaamalla sitä linkitettyyn listaan sekä muihin linkitettyihin rakenteisiin. Koska linkitetty lista on yksinkertaisin ratkaisu toisiinsa linkitetystä tiedusta, tutustutaan seuraavaksi tarkemmin sen toimintaperiaatteeseen. Lopuksi käydään läpi linkitetyn listan operaatiot eli luonti, läpikäynti ja vapautus toteutettuna yhden ohjelman sisällä sekä jaettuna useisiin aliohjelmiin.

Taulukoista linkitettyihin tietorakenteisiin

Otimme merkkitaulukon käyttöön jo tämän oppaan luvussa 1 ja totesimme tällöin, että merkkitaulukko tarkoittaa useille merkeille varattua yhtenäistä muistialuetta. Esimerkiksi 10 merkin merkkitaulukko saa käyttöjärjestelmältä käyttöön 10 tavun mittaisen muistialueen, koska yksi merkki käyttää aina yhden tavun muistia. Koska kyseessä on taulukko, voidaan taulukon alkioihin viitata indeksillä ja voimme tällä tavoin käsitellä haluamaamme taulukon merkkiä. Kuvassa 6.1 on varattu `nimi`-taulukko 10 merkille ja kun taulukkoon sijoitetaan merkkijono ”Ville”, voimme viitata indeksillä 4, ts. `nimi[4]`, taulukossa olevaan ’e’-merkkiin.

Merkkijonon joustavan käytön vuoksi se päättyy aina NULL merkkiin, ’\0’. Idea on varata riittävän iso taulukko merkkejä varten, käyttää tarvittava määrä merkkejä taulukon alusta, merkitä merkkijonon päättymisen NULL-merkillä ja näin saamme aina tarpeeseen sopivan merkkijonon. Kaikki C-kielen merkkijonofunktiot hyödyntävät NULL-merkkiä ja se mahdollistaa osoittimen tehokkaan käytön merkkijonon kanssa. Voimme nimittäin määritellä merkki-osoittimen, laittaa sen osoittamaan merkkijonon ensimmäiseen merkkiin ja käydä kaikki merkkijonon merkit läpi yksitellen siirtämällä osoitinta aina yhden merkin verran eteenpäin. Koska kaikki merkit ovat samankokoisia, yksi tavu, voidaan osoitinta siirtää eteenpäin inkrementti-operaattorin verran eli kuvan 6.1 `ptr`-osoitin siirtyy yhden merkin eteenpäin seuraavaan merkkiin `ptr++`-käskyllä. Koska merkkijono loppuu NULL-merkkiin, voimme käydä osoittimella läpi kaikki merkkijonon merkit lopetusehdolla `while (*ptr != '\0')` esimerkin 2.12 mukaisesti. Tiivistetysti merkkitaulukko on yhtenäinen muistialue, joka mahdollistaa useiden merkkien tallettamisen peräkkäin muistiin ja ne voidaan käydä läpi indeksillä tai osoittimella.

```
char nimi[10] = "Ville";
char *ptr = nimi;
```

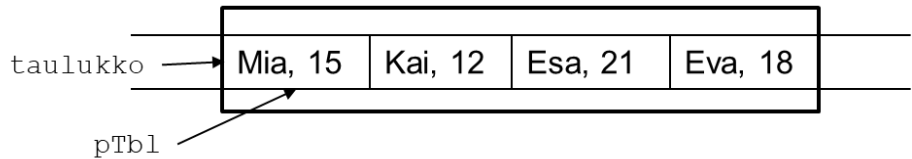


Kuva 6.1. Merkkitaulukon ja osoittimen määrittely sekä sijoittuminen muistiin

Kaikki C-kielen taulukko-rakenteet on toteutettu samoilla periaatteilla kuin merkkitaulukko loppumerkkiä lukuun ottamatta. Taulukon alkion koko on aina sama, oli kyseessä sitten merkki-, kokonaisluku-, liukuluku- tai tietue-aulukko. Alkiot voidaan käydä läpi indeksillä tai osoittimella, sillä osoitin siirtyy eteenpäin aina tietotyypin tarvitseman muistin verran. Merkkitaulukko poikkeaa muista taulukoista siinä, että taulukkoon sijoitetulla merkkijonolla on loppumerkki, josta tietää sen loppuneen. Itse taulukossa ei ole tietoa sen koosta vaan ohjelmoijan on ylläpidettävä koko-tietoa muuttujissa tai vakioissa. Taulukko voi olla yksi tai moniulotteinen ja jokainen ulottuvuus näkyy taulukossa omana indeksinään eli hakasulkuina. Tietue-aulukon alkion koko määräytyy tietueen tarvitseman tilan perusteella, joten se muuttuu tietueen määrittelyn muuttuessa.

Kuvassa 6.2 näkyy tietueen, tietue-aulukon ja tietue-osoittimen määrittelyt sekä miten ne sijoittuvat loogisesti tietokoneen muistissa. Taulukon alkiot ovat toistensa perässä yhtenäisessä muistialueessa, joten näille pätee samat periaatteet kuin merkkitaulukolle kunhan muistaa, että vain merkkijonolla on loppumerkki ja ohjelmoijan on tiedettävä muiden taulukoiden alkioden määrä.

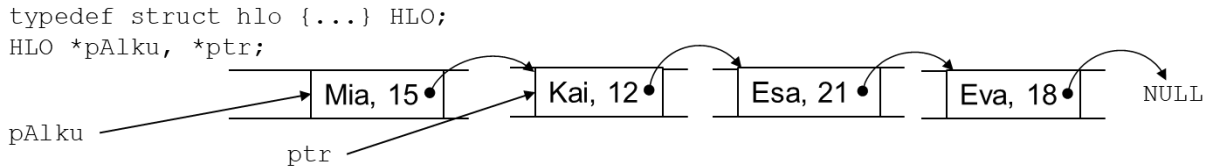
```
typedef struct hlo {char nimi[30];int ika;} HLO;
HLO taulukko[4];
HLO *pTbl = taulukko;
```



Kuva 6.2. Tietueen, tietue-aulukon ja osoittimen määrittely sekä sijoittuminen muistiin

Linkitetty lista on tietueista muodostuva tietorakenne, jossa tietueiden ei tarvitse olla peräkkäisissä muistipaikoissa samalla tavoin kuin tietue-aulukossa. Koska seuraava tietue ei löydy edellisen tietueen perästä muistissa, on sen osoite tallennettava edelliseen tietueeseen. Tämä on linkitetyn listan perusidea eli jokaisessa tietueessa on linkki seuraavaan tietueeseen, ts. sen osoite, ja näin listan alkioista muodostuu *linkitetty lista*. Linkitetyn listan alkiot varataan dynaamisesti tarpeen mukaan `malloc`'lla, joten listan ensimmäisen alkion eli solmun osoitetta varten pitää varata yksi osoitin, alku-osoitin. Kun listan ensimmäinen alkio luodaan, asetetaan alku-osoitin osoittamaan siihen ja sen jälkeen muut listan alkiot löytyvät aina edellisen alkion perusteella. Listan läpikäyntiä varten on luotava toinen osoitin, joka toimii listan liukuriina eli osoittaa aina haluttuun listan alkioon. Yhteen suuntaan linkitettyssä listassa liukuri aloittaa läpikäynnin aina alku-osoittimen ensimmäisestä alkioista ja etenee loppuun. Linkitetyn listan lopun tunnistaa siitä, että listan viimeinen alkio ei osoita uuteen alkioon vaan sen arvo on NULL listan lopun merkiksi.

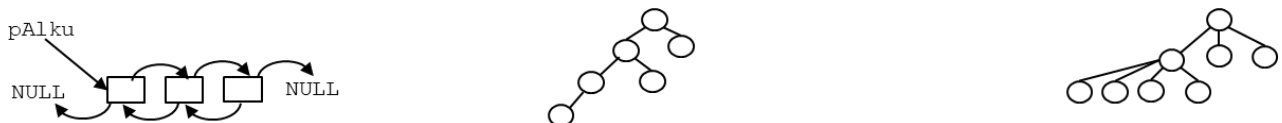
Linkitetty lista ja taulukko tarvitsevat yhtä paljon muistia talletettavalle tiedolle, mutta taulukko on määritelmän mukaan yksi iso muistialue, kun taas linkitettyssä listassa tietueet voivat sijaita missä tahansa muistissa ja ne on linkitetty toisiinsa. Esimerkiksi käsiteltäessä valokuvia tiedostojen koot ovat nykyään helposti 1-5MB ja kun aktiivisella kuvaajalla voi olla esim. 1 000, tai 10 000, digikuvaa tallennettuna tietokoneelle, on vapaan muistin löytäminen helpompaa, kun tarvittavan muistialueen koko on 5MB eikä 50GB. Kuvassa 6.3 näkyy tietueen ja osoittimien määrittelyt sekä listan alkioden ja osoittimien toiminnan logiikka tietokoneen muistissa. Listan alkioita kutsutaan useilla erilaisilla nimillä kuten solmu, tietue, alkio ja node, mutta kaikki ovat osa helminauhaa muistuttavaa rakennetta, jossa solmujen välissä on linkit vastaavasti kuin helmien välissä on lanka.



Kuva 6.3. Tietueen ja listaosoittimien määrittelyt sekä linkitetyn listan periaatekuva

Linkitetty lista on yksinkertaisin linkitetyistä tietorakenteista. Linkitettyjen tietorakenteiden perusidea on yhdistää tietorakenteita toisiinsa ja ylläpitää linkitettyjen tietorakenteiden osoitteita jäsenmuuttujissa. Linkitettyssä listassa ylläpidetään vain yhtä osoitetta eli seuraavan alkion osoitetta. Vastaavasti kahteen suuntaan linkitettyssä listassa on kaksi jäsenmuuttujaa listan osoitteita varten – yksi seuraavan alkion osoitetta varten ja toinen edellisen alkion osoitetta varten. Kaksisuuntainen lista mahdollistaa luonnollisesti liikkumisen listassa kahteen suuntaan ja siten se on joustavampi käytön kannalta. Helminauhaa imitoivan lista-rakenteen lisäksi linkeillä voidaan muodostaa erilaisia tietorakenteita kuten esim. puu, jossa on useita haaroja. Kuvassa 6.4. näkyy viimeisenä ”yleinen puu”, jossa yhden solmun alla voi olla rajaton määrä uusia solmuja eli osoitteita toisiin solmuihin. Kuvassa näkyy myös binaaripuu, jossa yhden solmun alla voi olla 0, 1 tai 2 solmua eli kaikki mahdolliset haarojen lukumäärät voidaan esittää binaariluvulla. Ja kuvan 6.4 vasemmassa reunassa näkyy kahteen suuntaan linkitetyn listan periaatekuva.

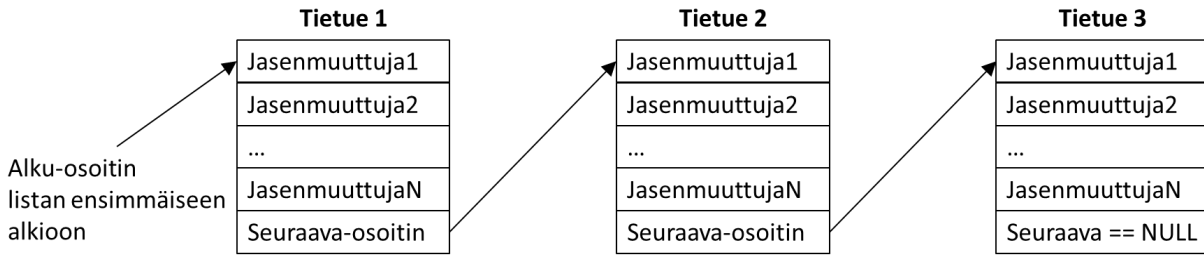
Tässä oppaassa keskitytään yhteen suuntaan linkitettyyn listaan ja tavoitteena on osata tehdä siihen kaikki perusoperaatiot eli listan luominen, läpikäynti ja muistin vapautus. Muita linkitettyjä tietorakenteita käsitellään laajemmin Tietorakenteet ja algoritmit -kursseilla ja kuten nimikin kertoo, käydään siellä tarkemmin läpi myös erilaisille rakenteille sopivia algoritmeja ja niiden tehokkuutta.



Kuva 6.4. Linkitettyjä tietorakenteita: kahteen suuntaan linkitetty lista, binaaripuu ja yleinen puu

Linkitetyn listan toimintaperiaate

Linkitetty lista tarkoittaa joukkoa tietueita eli alkioita, jotka on kytketty toisiinsa tallentamalla jokaiseen alkioon seuraavan alkion osoite. Siksi linkitetyn listan tietueessa on oltava varsinaisten tietoa sisältävien jäsenmuuttujien lisäksi yksi jäsenmuuttuja seuraavan alkion osoitetta varten. Alkiot muodostavat ketjun linkitettyjä alkioita eli linkitetyn listan ja listan lopun tunnistaa siitä, että alkion seuraava-osoittimen arvo on NULL. Listan alkioiden käsittely edellyttää osoitinmuuttujaa, joka avulla voimme siirtyä haluttuun alkioon ja käsitellä sen tietoja. Luonnollisesti tarvitsemme osoittimen listan ensimmäiseen alkioon eli sen alkuun, alku-osoittimen. Kuva 6.5 havainnollistaa linkitetyn listan periaatetta visuaalisesti.



Kuva 6.5. Yhteen suuntaan linkitetty lista, jossa on 3 tietuetta. Ensimmäiseen tietueeseen eli alkioon osoittaa erillinen osoitinmuuttuja eli alku-osoitin

Linkitetty lista on dynaaminen tietorakenne, joka tarkoittaa sitä, että siihen voi lisätä ja siitä voi poistaa alkioita tarpeen mukaan ohjelman suorituksen aikana. Listan käsittely perustuu aiemmin käsiteltyyn dynaamiseen muistinhallintaan (luku 5) ja osoittimiin (luvut 2 ja 5), joten nämä asiat kannattaa kerrata tarvittaessa.

Ohjelman suorituksen alkaessa lista on tyhjä ja sen merkkinä alku-osoittimen arvo on `NULL`. Lista luodaan uusia alkioita `malloc`-käskyllä ja ensimmäisen alkion osoite laitetaan listan alku-osoittimen arvoksi. Ensimmäisen alkion seuraava-osoittimen arvoksi tulee `NULL`, koska nyt uusi alkio on myös listan viimeinen alkio. Jatkossa listaan lisätään uusia alkioita tyypillisesti viimeiseksi alkioiksi, koska tällöin uuden alkion seuraava-osoittimen arvoksi voidaan laittaa `NULL` ja uuden alkion osoite voidaan sijoittaa aiemmin viimeisenä olleen alkion seuraava-osoitteeksi. Kun listan alkioita poistetaan, tulee poistettavaa alkioita edeltävän alkion seuraava-osoitin päivittää ja vapauttaa varattu muisti `free`-funktioilla. Näin listaan voi lisätä uusia ja poistaa tarpeettomia alkioita. Listan käytön lähtökohta on, että ensimmäiseen alkioon osoittava alku-osoitin on aina kunnossa –joko tyhjän listan kohdalla `NULL` tai ei-tyhjän listan kohdalla ensimmäisen alkion osoite.

Listan alkioden käsittely perustuu liukuri-osoitimeen, joka käy listaa läpi kunnes haluttu alkio löytyy ja alkioita voidaan käsitellä halutulla tavalla. Listan läpikäynti alkaa asettamalla liukuri-osoitin osoittamaan samaan alkioon kuin alku-osoitin ja sen jälkeen siirytään listan seuraavaan alkioon seuraava-osoittimen avulla kunnes haluttu alkio löytyy. Myös listan tyhjennys perustuu liukuri-osoitimeen, jonka avulla käydään läpi listan kaikki alkiot ja vapautetaan niiden varaama muisti. Listan alkioden vapautus edellyttää toista apumuuttujaa, kuten kohta nähdään.

Linkitetty lista koetaan usein hankalana asiana. Tämä johtuu siitä, että linkitettyssä listassa on monta huomiotavaa asiaa ja jos yhdessäkin niistä on virhe, ei ohjelma toimi oikein. Siksi on tärkeää olla huolellinen linkitettyä listaa tehdessä 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. Nämä kaikki asiat on käsitelty tämän oppaan aiemmissa luvuissa ja nyt niitä käytetään linkitetyn listan toteutukseen.

Linkitetyn listan toteutus yhden ohjelman sisällä

Edellä esitelty linkitetty lista perustuu tietueisiin ja siten ohjelman teko alkaa tietueen määrittelyllä. Esimerkissä 6.1 on määritelty `asiakas`-tietue, joka sisältää varsinaisena tietona yksinkertaisuuden vuoksi vain asiakkaan numeron `iNumero`. Tämän lisäksi tietueessa on jäsenmuuttuja seuraavan alkion osoitetta varten eli osoitin `pSeuraava`. Tietueen määrittelyn yhteydessä on nyt määritelty myös uusi tietotyyppi `ASIAKAS`, sillä käytämme tätä tietuetta sen verran usein, että uuden tietotyypin määrittely on paikallaan. Tietueen kannalta on oleellista huomata `pSeuraava`-osoittimen tietotyyppi, `struct asiakas*`, eli se on osoitin toiseen samanlaiseen tietueeseen.

Esimerkissä 6.1 on neljä osoitinta ASIAKAS-tietorakenteeseen. Listan ensimmäinen alkio on laitettava varmaan talteen alku-osoittimeen ja `pAlku` hoitaa tämän tehtävän. `pLoppu`-osoitin osoittaa aina listan viimeiseen alkioon ja sitä käytetään uusien alkioden lisäämiseen listaan. Tämä on apumuuttuja eikä välttämätön linkitetyn listan toteutuksen kannalta. Kolmantena osoittimena on `pUusi`, johon sijoitetaan `malloc`'n varaaman alkion osoite siihen asti, että se lisätään listaan. Neljäntenä osoittimena on määritelty liukuri osoitin `ptr`, jota käytetään listan läpikäyntiin tai halutun alkion etsimiseen listasta. `pAlku` ja `pLoppu`-osoittimet on alustettu `NULL`-arvolla, sillä ne osoittavat listan tiettyihin alkioihin ja kun listaa ei ole, on niiden arvo aina `NULL`.

Esimerkki 6.1. Linkitetyn listan määrittelyt

```
typedef struct asiakas {
    int iNumero;
    struct asiakas *pSeuraava;
} ASIAKAS;

ASIAKAS *pAlku = NULL, *pLoppu = NULL;
ASIAKAS *pUusi, *ptr;
```

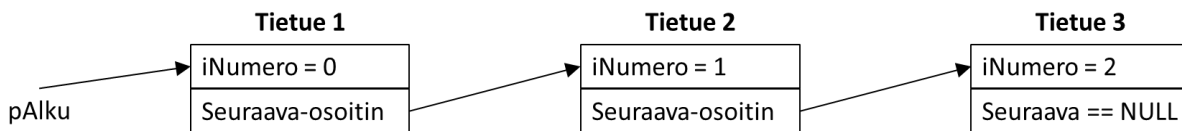
Määrittelyjen jälkeen alkioille on varattava muistia ja se on alustettava. Esimerkissä 6.2 on ensin muistin varaus aiemmin opetellulla tavalla virheenkäsittelyn kanssa ja onnistuneen muistinvarauksen jälkeen asetetaan listan uuden alkion jäsenmuuttujille arvot. Tässä esimerkissä `iNumero` saa arvoksi silmukkamuuttujan askeltajan `i:n` arvo, jotta kaikissa alkioissa on eri arvot, ja `pSeuraava` saa arvoksi `NULL`, koska tämä alkio laitetaan listan viimeiseksi alkioiksi ja siten seuraavaa alkioita ei ole.

Viimeisenä kohtana esimerkissä 6.2 on uuden alkion lisäys listaan. Tällöin meillä on kaksi vaihtoehtoa, joista ensimmäinen on tyhjän listan tapaus. Tyhjän listan `pAlku`-osoittimen arvo on `NULL`, joten se pitää laittaa osoittamaan uuteen alkioon. Tässä esimerkissä ylläpidämme myös listan viimeisen alkion osoitetta `pLoppu`-osoittimessa, joten myös sen arvoksi tulee uuden alkion osoite. Jos taas lista ei ole tyhjä vaan siinä on jo alkioita, lisäämme uuden alkion viimeisen alkion perään `pLoppu`-osoittimen avulla ja päivitämme `pLoppu`-osoittimen osoittamaan uuteen alkioon eli laajennetun listan viimeiseen alkioon. Toistamalla esimerkin 6.2 toimenpiteet jokaisen uuden alkion kohdalla saamme tehtyä listan eli kaikki listaan tulevat alkiot luodaan ja ne linkitetään toisiinsa.

Esimerkki 6.2. Muistin varaus, jäsenmuuttujien ja solmun lisäys listaan

```
// Muistin varaus
if ((pUusi = (ASIAKAS*)malloc(sizeof(ASIAKAS))) == NULL ) {
    perror("Muistin varaus epäonnistui");
    exit(1);
}
// Uuden alkion jäsenmuuttujien arvojen asettaminen
pUusi->iNumero = i; // i on ympärillä olevan silmukan askeltajan arvo
pUusi->pSeuraava = NULL;
// Uuden alkion lisääminen listaan viimeiseksi alkioksi
if (pAlku == NULL) { // lista on tyhjä, joten tehdään ensimmäinen alkio
    pAlku = pUusi;
    pLoppu = pUusi;
} else { // lista ei ole tyhjä, joten lisätään loppuun
    pLoppu->pSeuraava = pUusi;
    pLoppu = pUusi;
}
```

Esimerkin 6.2 listan luomisen tuloksena meillä on kuvassa 6.6 näkyvä linkitetty lista, kun listaan lisätään kolme alkioa.



Kuva 6.6. Esimerkin 6.2 luoma linkitetty lista

Listan luomisen jälkeen voimme käydä sen läpi esimerkin 6.3 toistorakenteella. Läpikäynti alkaa laittamalla liukuriosoittimen `ptr` arvoksi listan ensimmäisen alkion osoite. Tämän jälkeen jokaisen listan alkion sisältämä arvo, `iNumero`, tulostetaan näytölle ja `ptr`-liukurin arvoksi tulee seuraavan listan alkion osoite. Kun osoite on `NULL`, loppuu silmukan läpikäynti ja jokaisen listassa olevan alkion sisältämä arvo on tulostettu näytölle.

Esimerkki 6.3. Linkitetyn listan läpikäynti

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

Dynaamisen muistinvarauksen periaatteiden mukaisesti kaikki varattu muisti pitää vapauttaa. Käytännössä lista pitää käydä läpi toistorakenteella ja vapauttaa jokaiselle alkioille varattu muisti. Tämä edellyttää kahta osoitinta ja tarkkuutta, sillä listan alkion poistaminen väärällä hetkellä estää muiden solmujen läpikäynnin. Listan vapauttaminen alkaa samalla tavalla kuin listan läpikäynti eli liukuri `ptr` laitetaan osoittamaan listan ensimmäiseen alkioon ja sen arvoksi tulee `pAlku`-arvo. Ennen alkion vapauttamista `pAlku` -osoittimen arvoksi laitetaan *sitä seuraavan alkion osoite*. Huomaa, että ensimmäisen alkion poistamisen jälkeen tämä on jäljellä olevan listan ensimmäinen alkio. Nyt kun meillä on osoitin jäljelle jäävään listaan, voimme vapauttaa liukurin osoittaman

ensimmäisen alkion `free`-käskyllä. `ptr`-liukuri on nyt vapaana ja voimme laittaa sen osoittamaan samaan alkioon `pAlku`-osoittimen kanssa eli jäljellä olevan listan ensimmäiseen alkioon. Tässä vaiheessa huomaamme, että olemme saaneet vapautettua listan ensimmäisen alkion ja olemme lähtötilanteessa eli `pAlku` ja `liukuri-ptr` -osoittimet osoittavat listan ensimmäiseen alkioon. Toistamalla samat toimenpiteet kaikkien listan alkioden kohdalla saamme vapautettua kaikki listan alkiot ja lopetamme toistorakenteen kun `liukuri`-osoittimen arvo on `NULL`.

Esimerkki 6.4. Linkitettylle listalle varatun muistin vapauttaminen

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

Käsittelimme edellä linkitettyyn listaan liittyvät operaatiot loogisina kokonaisuuksina yksi kerrallaan, jotta pystyimme paremmin keskittymään aina kulloinkin oleellisiin kohtiin. Esimerkissä 6.5 näkyy ohjelma, jossa nämä koodit ovat yhdessä toimivana kokonaisuutena. Kannattaa tutustua siihen, jotta ymmärtää sen toiminnan ja tarpeen mukaan tarkistaa yksityiskohtia edellä olevista selityksistä. Ohjelman jälkeen näkyy sen testiajon tulokset, jotta voit vertailla niitä oman ohjelmasi toimintaan, jos haluat testata alla olevaa ohjelmaa ja kirjoittaa sen itse.

Esimerkki 6.5. Toimiva ohjelma yhteen suuntaan linkitetulle listalle

```

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

// Minimaalinen tietue listalle
typedef struct asiakas {
    int iNumero;
    struct asiakas *pSeuraava;
} ASIAKAS;

int main(void) {
    ASIAKAS *pAlku = NULL, *pLoppu = NULL; // Osoittimia listan käyttöön
    ASIAKAS *pUusi, *ptr; // Apuosoitin muistin varaukseen ja liukuri
    int i;

    // Listan luominen, 3 alkiota malliksi
    for (i=0; i < 3; i++) {
        // Muistin varaus
        if ((pUusi = (ASIAKAS*)malloc(sizeof(ASIAKAS))) == NULL ){
            perror("Muistin varaus epäonnistui");
            exit(1);
        }
        // Uuden alkion jäsenmuuttujien arvojen asettaminen
        pUusi->iNumero = i;
        pUusi->pSeuraava = NULL;
        // Uuden alkion lisääminen listaan viimeiseksi alkioksi
        if (pAlku == NULL) { // lista on tyhjä, joten
            pAlku = pUusi;    // tehdään ensimmäinen alkio
            pLoppu = pUusi;
        } else {             // lista ei ole tyhjä, joten lisätään loppuun
            pLoppu->pSeuraava = pUusi;
            pLoppu = pUusi;
        }
    }

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

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

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

```

Esimerkin tuottama tulos

Kun käännämme ja ajamme ohjelman, saamme seuraavanlaisen tuloksen:

```
un@LUT8859:~/Opas$ ./E6_5
Nyt palvelemme numeroa 0.
Nyt palvelemme numeroa 1.
Nyt palvelemme numeroa 2.
Viimeisen asiakkaan jälkeen ei ollut ketään.
```

Tämä esimerkki kävi läpi linkitetyn listan oleelliset rakenneosat ja toiminnot. Lista koostuu tietueesta ja osoittimista, joiden avulla voidaan varata listan tarvitsema muisti ja luoda linkitetty lista, käydä se läpi ja vapauttaa sen varaama muisti. Tässä esimerkissä kaikki toiminnot oli toteutettu yhden ohjelman sisällä, joka sopii pieneen ohjelmaan. Usein ohjelman koon kasvaessa modulaarinen toteutus aliohjelmien avulla on järkevää, joten katsotaan sitä seuraavaksi.

Linkitetyn listan toteutus aliohjelmina

Edellisessä kohdassa kävimme läpi linkitetyn listan perusoperaatiot yhdessä ohjelmassa ja kokonaisuus oli tällöin varsin selkeä. Ohjelmien toiminnallisuuden määrän kasvaessa toimintoja sijoitetaan aliohjelmiin rakeisen ohjelmoinnin nimissä, jotta osat muodostuvat pienemmiksi ja niiden toteutus pysyy selkeänä. Aliohjelmat mahdollistavat myös uudelleenkäytön eli tekemällä selkeitä pieniä aliohjelmia, voidaan niitä hyödyntää muualla ja siten vähentää tarvetta uuden koodin kirjoittamiseen. Linkitetty lista ei ole tässä suhteessa poikkeus ja ohjelmia tehdessä kannattaa pyrkiä muodostamaan selkeitä ja pieniä aliohjelmia, jotta mahdollistetaan niiden uudelleenkäyttö.

Esimerkin 6.5 pohjalta on selvää, että linkitetyn listan peruskäsittelyssä on kolme operaatiota, joista voidaan muodostaa omat aliohjelmat: uuden alkion lisäys listaan, listan läpikäynti ja listan vapautus. Koska seuraavassa ohjelmassa käytetään samoja tietorakenteita kuin edellä, ei niitä käydä enää läpi vaan ne voi katsoa edellisistä esimerkeistä.

Uuden alkion lisäys listaan on selkeä kokonaisuus, jonka sisältö ei poikkea yhden ohjelman toteutuksesta vaan käsittää edelleen muistin varaamisen, alkion jäsenmuuttujien arvojen asettamisen ja alkion lisäämisen listaan. Aliohjelmatoteutuksen vuoksi minimoidaan tiedonsiirtoa ja siksi aliohjelmaan välitetään parametreina listan alku-osoitin sekä alkioon lisättävät tiedot eli tässä tapauksessa vain yksi kokonaisluku.

Aliohjelmaan tulee parametrinä sen ensimmäisen alkion osoite ja aliohjelmasta tulee palauttaa aina ensimmäisen alkion osoite, koska se muuttuu, kun tyhjiin listaan lisätään ensimmäinen alkio. Tässä esimerkissä aliohjelmaan välitetään ja sieltä palautetaan alkion osoite, joten ne molemmat ovat `ASIAKAS *`-tyyppisiä tietoalkioita. Vaihtoehtoisesti listan ensimmäisen alkion osoite voidaan välittää muuttujaparametrina, jolloin tietoa ei tarvitse palauttaa paluuarvona. Tällöin meidän tulisi välittää ensimmäisen solmun osoitteen sisältävä osoite, jolloin kyseessä olisi osoitteen osoite ja `ASIAKAS **`-tyyppinen muuttuja. Tällaisen rakenteen käyttö ei ole mielekäästä tämän oppaan kannalta, vaan tässä oppaassa suositellaan käytettäväksi esimerkin 6.6 mukaisia parametreja ja paluuarvoja eli `ASIAKAS *`-osoitteita.

Esimerkissä 6.6 solmun lisäys aliohjelmaan tulee vain listan ensimmäisen alkion osoite, joten lisättäessä listaan uusi alkio, pitää listan viimeinen alkio etsiä `while`-rakenteen avulla. Tämän `while`-rakenteen tilalla yhden ohjelman sisällä meillä oli esimerkissä 6.5 kaksi osoitinta – osoittimet listan ensimmäiseen ja viimeiseen alkioon, `pAlku` ja `pLoppu`. Nämä ovat vaihtoehtoisia ratkaisuja ja aliohjelman kanssa listan viimeisen alkion etsiminen on luonnollinen

toimintatapa, sillä se vähentää tiedonsiirron tarvetta. Tyypillisestä listan läpikäynnistä poiketen nyt listan läpikäynnin lopetusehtona on `ptr->pSeuraava != NULL` eikä `ptr != NULL`. Huomaa myös, että aliohjelma palauttaa joko parametrinä tulleen alkion osoitteen tai `malloc`'lla varatun ensimmäisen alkion osoitteen.

Esimerkki 6.6. Solmun lisäys linkitettyyn listaan aliohjelmassa

```

ASIAKAS *lisaaSolmu(ASIAKAS *pA, int x) {
    ASIAKAS *pUusi, *ptr;

    // Muistin varaus
    if ((pUusi = (ASIAKAS*)malloc(sizeof(ASIAKAS))) == NULL ) {
        perror("Muistin varaus epäonnistui");
        exit(1);
    }
    // Uuden alkion arvojen määrittäminen
    pUusi->iNumero = x;
    pUusi->pSeuraava = NULL;
    // Uuden alkion jäsenmuuttujien arvojen asettaminen
    if (pA == NULL) { // lista on tyhjä -> eka alkio
        pA = pUusi;
    } else { // lista ei tyhjä -> viimeinen alkio
        ptr = pA;
        while (ptr->pSeuraava != NULL)
            ptr = ptr->pSeuraava;
        ptr->pSeuraava = pUusi;
    }
    return(pA);
}

```

Listan läpikäynti ja sen alkioden arvojen tulostus ei poikkea aiemmasta ratkaisusta. Liukuri-osoitin asetetaan osoittamaan ensimmäistä listan alkioita ja sen jälkeen käydään listan alkioita läpi, kunnes saavutetaan loppumerkki eli `NULL`. Aliohjelmasta ei tarvitse palauttaa mitään, joten aliohjelman tyyppi voi olla `void`.

Esimerkki 6.7. Linkitetyn listan tulostus aliohjelmassa

```

void tulosta(ASIAKAS *pA) {
    ASIAKAS *ptr = pA;
    while (ptr != NULL) {
        printf("Nyt palvelemme numeroa %d.\n", ptr->iNumero);
        ptr = ptr->pSeuraava;
    }
    return;
}

```

Myös listan tyhjentäminen tapahtuu samalla tavalla kuin yhden ohjelman sisäisessä ratkaisussa. Molemmissa tapauksissa kannattaa laittaa listan alku-osoitin tyhjäksi eli `NULL`:ksi, jolloin aliohjelmatoteutuksessa paluuarvo on `NULL` ja se tulee sijoittaa alku-osoittimen uudeksi arvoksi. Näin vältetään ongelmilta, kun ohjelma laajenee ja listan käyttö jatkuu tyhjennyksen jälkeen.

Esimerkki 6.8. Linkitetyn listan tyhjentäminen aliohjelmassa

```

ASIAKAS *tyhjenna(ASIAKAS *pA) {
    ASIAKAS *ptr = pA;
    while (ptr != NULL) {
        pA = ptr->pSeuraava;
        free(ptr);
        ptr = pA;
    }
    return(pA);
}

```

Esimerkissä 6.9 näkyy vielä koko aliohjelmista koostuva ratkaisu tietue- ja osoitinmäärittelyineen. Huomaa, miten toiminnallisuuden siirtäminen loogisesti nimettyihin aliohjelmiin pienentää pääohjelmaa ja tekee siitä helpon ymmärtää. Nyt pääohjelma sisältää (1) listan osoitinmäärittelyn ja aliohjelmatkutsut, joilla (2) listaan lisätään solmujen, (3) lista tulostetaan ja (4) lista tyhjennetään. Jokaista aliohjelmaa voidaan tarkastella erikseen tarpeen mukaan ja kun ne ovat lyhyitä ja selkeitä, tukee järkevä aliohjelmarakenne ohjelman yleistä ymmärrettävyyttä ja ylläpidettävyyttä.

Esimerkki 6.9. Ohjelma linkitetyn listan toteutukseen aliohjelmillä

```

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

typedef struct asiakas {
    int iNumero;
    struct asiakas *pSeuraava;
} ASIAKAS;

ASIAKAS *lisaaSolmu(ASIAKAS *pA, int x) {
    ASIAKAS *pUusi, *ptr;

    // Muistin varaus
    if ((pUusi = (ASIAKAS*)malloc(sizeof(ASIAKAS))) == NULL ){
        perror("Muistin varaus epäonnistui");
        exit(1);
    }
    // Uuden alkion arvojen määrittäminen
    pUusi->iNumero = x;
    pUusi->pSeuraava = NULL;
    // Uuden alkion jäsenmuuttujien arvojen asettaminen
    if (pA == NULL) { // lista on tyhjä -> eka alkio
        pA = pUusi;
    } else { // lista ei tyhjä -> viimeinen alkio
        ptr = pA;
        while (ptr->pSeuraava != NULL)
            ptr = ptr->pSeuraava;
        ptr->pSeuraava = pUusi;
    }
    return(pA);
}

```

```

ASIAKAS *tulosta(ASIAKAS *pA) {
    ASIAKAS *ptr = pA;
    while (ptr != NULL) {
        printf("Nyt palvelemme numeroa %d.\n", ptr->iNumero);
        ptr = ptr->pSeuraava;
    }
    return(pA);
}

ASIAKAS *tyhjenna(ASIAKAS *pA) {
    ASIAKAS *ptr = pA;
    while (ptr != NULL) {
        pA = ptr->pSeuraava;
        free(ptr);
        ptr = pA;
    }
    return(pA);
}

int main(void) {
    ASIAKAS *pAlku = NULL;
    int i;

    // Listan luominen, 3 alkiota kokeeksi
    for (i=0; i < 3; i++ ) {
        pAlku = lisaaSolmu(pAlku, i);
    }
    // Listan läpikäynti
    tulosta(pAlku);
    // Muistin vapauttaminen
    pAlku = tyhjenna(pAlku);

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

```

Ajan käsittely C-ohjelmissa

Päivämäärien ja kellonaikojen käsittely on tietokoneen perustoimintoja, joten tutustutaan C-kielen perustoimintoihin ajan suhteen. Aloitetaan ajan määrittelystä Unix-järjestelmissä ja sen jälkeen katsotaan esimerkkejä aika-tiedon käsittelystä ja miten aikatiedolla voidaan laskea.

Huomaa, että aika-tiedon käsittely C-kielessä perustuu useisiin tietorakenteisiin ja funktioihin, joilla tyypillisesti tehdään yksi asia kullakin. Hyvin samantyyppisiä funktioita ja rakenteita on useita, joita käytetään hieman eri tarkoituksiin niin kuin on tyypillistä C-kielessä.

Unix-järjestelmän aika, Epoch

Yksi käyttöjärjestelmien perusominaisuuksia on ajan käsittely ja Unix-puolella ”ajanlasku” alkaa 1.1.1970 klo 00:00:00 +0000 (UTC). Käytännössä tämä tarkoittaa sitä, että aika-muuttujan arvo on tällä ajanhetkellä 0, jonka jälkeen se on positiivinen kasvava luku kun taas ennen sitä arvo on negatiivinen pienenevä luku normaalin aikajanana mukaisesti. Unix-järjestelmässä aika on sisäisesti luku, jonka kokonaisosa esittää sekunteja. Ajan sisäinen esitys lukuna tekee aika-arvojen vertailusta

helppoa, sillä normaali päivämäärä-kellonaika -rakenne tiivistyy yhteen lukuarvoon. Aika on siis selkeä asia tietokoneen sisällä, mutta ajan esittäminen ihmisten ymmärtämässä muodossa päivinä, kuukausina ja vuosina ottaen huomioon karkausvuodet jne. vaatii tarkkuutta ja siksi C-kieli tarjoaa useita funktiota ajan muuttamiseen ihmisten ymmärtämän ja sisäisen esitysmuodon välille.

Ajan sisäinen esitysmuoto on aina luku, jonka tietotyyppi on tyypillisesti `time_t`. Eri järjestelmissä aika voidaan toteuttaa eri tavoin esim. 32 tai 64 bittisenä kokonaislukuna, etumerkillä tai ilman, tai liukulukuna. Siksi kannattaa käyttää aina ajan käsittelyyn tehtyjä valmiita funktioita, sillä ne toimivat ympäristöstä riippumatta oikein.

Tutustutaan seuraavaksi tarkemmin ajan käsittelyyn C-kielellä ja tästä eteenpäin on hyvä muistaa, että aikaleimat ovat sisäisesti lukuja, joita voi vertailla ja joilla voi laskea peruslaskutoimituksia. Tietovisailuja varten kannattaa muistaa, että Unixin ajanlasku alkoi 1.1.1970 klo 00:00:00, jolloin Unixin sisäisessä esitysmuodossa aika oli 0 ja tätä hetkeä kutsutaan Unixin Epoch'ksi.

Ajan perusoperaatiot

Aikaan liittyvät perusoperaatiot voidaan jakaa näljään perustyyppiin. Ensimmäinen on tietokoneen järjestelmässä olevan ajan selvittäminen ja sen muuttaminen paikalliseksi ajaksi. Toinen on ajan esittäminen ihmisen ymmärtämässä muodossa esim. merkkijonona näytöllä tai tallettaminen tiedostoon. Kolmas on tiedostosta luetun tai käyttäjältä saadun aikaa esittävän merkkijonon muuttaminen tietokoneen ymmärtämäksi aika-tiedoksi, jota voidaan hyödyntää ohjelmissa esim. laitettaessa tietoja aikajärjestykseen. Ja neljäs on päivämäärillä laskeminen, esim. kahden päivämäärän välissä olevien päivien, tai vuosien, laskeminen.

Tietokoneen ajan selvittäminen onnistuu `time`-funktiolla esimerkin 6.10 mukaisesti. Funktio hakee ajanlaskun alusta kuluneiden sekuntien määrän ja se voidaan tulostaa lukuna. Antamalla saatu aika `localtime`-funktiolle, saadaan selville vastaava ajanhetki eli päivämäärä ja kellonaika. Helpoiten tämän saa tulostettua `asctime`-funktiolla, jolloin formaatti on etukäteen päätetty ml. lopussa olevan rivinvaihtomerkki. Mutta näillä saa nopeasti aikatietoja selville, kun muistaa sisällyttää ohjelmaan otsikkotiedoston `time.h`.

Esimerkki 6.10. Epoch-aika ja tietokoneen aika

```
#include <time.h>

time_t result;
result = time(NULL);
printf("Unixin ajanlaskun alusta on %ld sekuntia.\n", result);
printf("Tänään on %s", asctime(localtime(&result)));
```

`time`-funktiolle voi antaa myös parametrinä `time_t` -rakenteen osoitteen esimerkin 6.11 mukaisesti, jolloin ajan voi välittää `localtime`-funktiolle muokattavaksi ja paluuarvona tulee osoitin tietueeseen `struct tm`. `tm`-tietueen voi tulostaa itse määritellyssä muodossa `strftime`-funktiolla merkkitaulukkoon ja käyttää sopivalla tavalla, esim. tulostaa näytölle. Päivämäärän ja kellonajan muotoilumerkit löytyvät funktion `man`-sivuilta ja keskeisimmät niistä näkyvät esimerkissä 6.11.

Esimerkki 6.11. Tietokoneen ajan muotoiltu tulostus

```
char buf[255];
time_t result;
struct tm *tm;
time(&result);
tm = localtime(&result);
strftime(buf, sizeof(buf), "%d.%m.%Y %H:%M\n", tm);
puts(buf);
```

Epoch-aika saadaan selville time-funktiolla, Epoch-ajan saa muokattua paikalliseen aikaan localtime-funktiolla tm-tietueeksi, joka taas saadaan muokattua ihmisille tutuksi päivämäärä/-kellonaika -merkkijonoksi strftime-funktiolla. Näiden käänteisoperaatio on siirtyä merkkijonona esitetystä päivämäärästä tm-tietueeseen, joka onnistuu strptime-funktiolla esimerkin 6.12 mukaisesti. Huomaa, että strptime-funktio edellyttää ohjelman alkuun määrittelyn `_XOPEN_SOURCE`. strptime-funktio hoitaa asian kerralla kuntoon, mutta tm-tietueen jäsenmuuttujia voi käsitellä myös tarpeen mukaan yksitellen alla olevan esimerkin 6.15 mukaisesti.

Esimerkki 6.12. Merkkijono aika-tietueeksi Linuxissa

```
#define _XOPEN_SOURCE // strptime-edellyttää tätä määrittelyä

char buf[255];
struct tm tm;
strptime("2001-11-12 18:31:01", "%Y-%m-%d %H:%M:%S", &tm);
strftime(buf, sizeof(buf), "%d.%m.%Y %H:%M\n", &tm);
puts(buf);
```

Erilaiset ajan esitysmuodot sopivat erilaisiin tarkoituksiin ja siksi niitä on useita. Epoch-aika helpottaa aikaleimojen vertailua, sillä liukulukujen vertailu on tuttua samoin kuin niillä laskenta. Merkkijono-esitysmuoto on tuttu ihmisille, vaikka merkkijonoissa on kulttuuri- ja henkilökohtaisia eroja. Lisäksi ajan käsittelyyn kuuluu aikavyöhykkeiden huomiointi, jos aikavyöhyke muuttuu joko maantieteellisten siirtymien tai kesä-talviaikamuutosten takia. Esimerkissä 6.13 käydään läpi struct tm:n ja Epoch-ajan käsittely laskentaa varten ja sekuntien muuttaminen päiviksi. tm-tietue voidaan muuttaa Epoch-aikaan mktime-funktiolla tämän esimerkin mukaisesti. Huomaa aika-tietueen nollaaminen memset-funktiolla, jottei muistialueella olevat aiemmat tiedot sotke laskentaa. Oleellista on, että kaikki rakenteen jäsenmuuttujat nollataan, sillä muutoin lopputulos tahtoo yllättää ohjelmoijan.

Esimerkki 6.13. Päivämäärillä laskeminen

```
#define _XOPEN_SOURCE // strptime-edellyttää tätä määrittelyä

struct tm eka, toka;
time_t ekaEpoch, tokaEpoch;
double paivia;

memset(&eka, 0, sizeof(eka));
memset(&toka, 0, sizeof(toka));
strptime("2020-02-01", "%Y-%m-%d", &eka);
strptime("2020-03-01", "%Y-%m-%d", &toka);

ekaEpoch = mktime(&eka);
tokaEpoch = mktime(&toka);
paivia = (tokaEpoch - ekaEpoch) / (60*60*24); // Sekunneista päiviksi
printf("%.0lf paivaa\n", paivia);
```

Aika-tietue `tm`:n rakenne näkyy esimerkissä 6.14. Rakenne on määritelty otsikkotiedostossa `time.h`, ja useimmat tietueen arvot ovat loogisia – esim. sekunnit ovat 0-60, minuutit 0-59 ja tunnit 0-23. Rakenteen kanssa kannattaa kuitenkin olla tarkkana, sillä kuukausi on arvoina 0-11 ja vuosiluku pitää laskea halutun vuoden ja vuoden 1900 erotuksena. Näin ollen haluttaessa esim. vuosi 2021, pitää `tm`-rakenteeseen vuoden kohdalle sijoittaa kokonaisluku 121 eli (2021-1900).

Esimerkki 6.14. Aika-tietueen struct tm rakenne time.h:ssa

```
struct tm {
    int tm_sec;    /* Seconds (0-60) */
    int tm_min;    /* Minutes (0-59) */
    int tm_hour;   /* Hours (0-23) */
    int tm_mday;   /* Day of the month (1-31) */
    int tm_mon;    /* Month (0-11) */
    int tm_year;   /* Year - 1900 */
    int tm_wday;   /* Day of the week (0-6, Sunday = 0) */
    int tm_yday;   /* Day in the year (0-365, 1 Jan = 0) */
    int tm_isdst;  /* Daylight saving time */
};
```

Aika-tietueen `struct tm` käyttö on helppointa `strptime` ja `strftime`-funktiolla, mutta sitä voi käyttää myös käsin asettamalla jäsenmuuttujien arvot halutuiksi esimerkin 6.15 mukaisesti. Niin kuin edellä oli puhetta, `struct tm`-rakenne kannattaa nollata ennen käyttöä `memset`:llä, koska käyttöjärjestelmä ei sitä välttämättä tee.

Esimerkki 6.15. Aika-tietueen käyttö ja muotoiltu tulostus

```
#include <string.h>

char buf[255];
struct tm tm;
memset(&tm, 0, sizeof(tm)); // Tietorakenteen nollaus
printf("Tulostetaan aika 15.2.2021 12:13 strftime-funktiolla.\n");
tm.tm_hour = 12;           // Ajan asetus jäsenmuuttuja kerrallaan
tm.tm_min = 13;
tm.tm_mday = 15;
tm.tm_mon = 1;             // Huom. 0-11 !!
tm.tm_year = (2021-1900);  // Huom. year - 1900 !!
strftime(buf, sizeof(buf), "%d.%m.%Y %H:%M\n", &tm);
puts(buf);
```

Kuten edellä oli puhetta, merkkijonona olevan päivämäärän voi muuttaa aika-tiedoksi `strptime`-funktiolla. Tämä on luonteva tapa asian tekemiseen, koska käänteisoperaatiota varten on olemassa `strftime`-funktio. Mikäli `strptime`-funktiolle kaippaa vaihtoehtoja, voi käyttää `scanf`-funktio, jolla saadaan luettua kaikki `tm`-tietueen tiedot yhdellä käskyllä. Voimme nimittäin pyytää käyttäjältä päivämäärän, antaa esitysmuoto kuten `pp.mm.vvvv`, ja sen jälkeen lukea `struct tm`-tyyppiseen `pvm`-muuttujaan annettu päivämäärä yhdellä käskyllä:

```
scanf("%d.%d.%d", &pvm.tm_day, &pvm.tm_mon, &pvm.tm_year);
```

Tällöin pitää muistaa tehdä korjaukset `struct tm`:n vuosi- ja kuukausimuuttujiin esimerkin 6.15 mukaisesti. Toinen huomioitava asia on koko tietueen tyhjennys `memset`-funktiolla ennen sen käyttöä, sillä C-kielessä muistin alustus on ohjelmoijan vastuulla ja usein alustamaton muisti estää ohjelman normaalin toiminnan.

Kolmas vaihtoehto päivämäärän käsittelylle on käsitellä merkkijono paloittain `strtok`-funktiolla esimerkin 5.8 mukaisesti. Tämäkin tapa edellyttää kuukausi- ja vuosilukujen päivittämistä `struct tm`:n odottamaan muotoon. `struct tm`:n jäsenmuuttujat ovat kokonaislukuja, mikä helpottaa niiden kanssa operointia.

Yksi keskeinen rajoite edellä esitetylle ajankäsittelyosuudelle on, ettemme huomioineet siinä kesä- ja talviaikaa. Näillä tiedoilla pääsee kuitenkin alkuun ajan käsittelyssä ja asioita pitää selvittää lisää aina tarpeen mukaan.

Varatun muistin vapautuksen tarkistaminen

Muistin varaamiseen jälkeen se on myös vapautettava, jottei muistia rupea vuotamaan. C-kielessä muistinvaraus on ohjelmoijan vastuulla samoin kuin muistin vapauttaminen. Muistin vapautus tapahtuu yksinkertaisesti `free`-käskyllä antamalla funktiolle parametrinä dynaamisesti varatun muistialueen osoite. Yksittäisen muistialueen kohdalla vapautus on selkeää, sillä silloin riittää lisätä jokaista `malloc`-käskyä vastaava `free`-käsky. Siirryttäessä linkitettyyn listaan muistin vapauttaminen tulee entistä tärkeämmäksi, sillä listassa voi olla useita alkioita ja siten varatun muistin määrä voi olla merkittävästi yksittäistä tapausta suurempi. Linkitetyn listan läpikäynnissä on lisäksi käsiteltävä kaikki listan alkiot, ettei joku alkio jää vapauttamatta. Tyypillisiä ongelmakohtia ovat ensimmäinen ja viimeinen alkio, mutta oikein koodatulla toistorakenteella nekin hoituvat ongelmitta.

Linuxin mukana tulee Valgrind-työkalu muistialueen käytön tarkistamiseen. Suorittamalla ohjelman Valgrindilla saa suorituksen päätteeksi raportin, josta näkyy varatut ja vapautetut muistialueet eli jäikö joku muistialue vapauttamatta. Tämä on tyypillinen tapa varmistua ohjelman oikeasta toiminnasta ja vastaavia ohjelmia löytyy myös muihin ympäristöihin sekä tarkastuksiin.

Valgrind asennetaan seuraavalla käskyllä Linuxin komentorivillä (esim. VSC-terminaalissa):

```
sudo apt install valgrind
```

Tämän jälkeen tarkistettava ohjelma tulee kääntää optiolla `-g` eli sisällytetään debug-tietoja suoritettavaan ohjelmaan:

```
gcc E6_9.c -o E6_9 -Wall -std=c99 -pedantic -g
```

Esimerkissä 6.16 näkyy Valgrindin suorituskäsky ja tulosteet, kun ohjelmassa on muistivuoto. Esimerkissä näkyy ohjelman tulosteet keltaisella ja muistivuotoa koskevat virheilmoitukset on merkattu punaisella. Tulosteen tarkastelun voi aloittaa lopusta, sillä ERROR SUMMARY kertoo, että ohjelmassa on virheitä ja LEAK SUMMARY tarjoaa tarkempia tietoja niistä. Vuoto-yhteenvedon yläpuolella näkyy ongelmien tausta eli `main`-ohjelmasta on kutsuttu `lisaaSolmu`-aliohjelmaa, joka on kutsunut `malloc`'ia eikä näin varattua muistia ole vapautettu ohjelmassa. Tässä tapauksessa virhe oli `tyhjenna`-aliohjelmassa, jossa listan viimeisen solmu jäi vapauttamatta. Vaikka Valgrind ei osannutkaan kertoa virheen kohtaa tarkasti, se tunnisti virheen ja mahdollisti näin sen korjaamisen.

Esimerkki 6.16. Valgrind’n raportin oleelliset osat muistia vuotavalle ohjelmalle

```

un@LUT8859:~/Esimerkit$ valgrind --leak-check=full ./E6_9
...
Nyt palvelemme numeroa 0.
Nyt palvelemme numeroa 1.
Nyt palvelemme numeroa 2.
Viimeisen asiakkaan jälkeen ei ollut ketään.
==444==
==444== HEAP SUMMARY:
==444==    in use at exit: 16 bytes in 1 blocks
==444==   total heap usage: 4 allocs, 3 frees, 1,072 bytes allocated
==444==
==444== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==444==    at 0x483B7F3: malloc (in ...)
==444==    by 0x109205: lisaaSolmu (E6_9.c:13)
==444==    by 0x109345: main (E6_9.c:57)
==444==
==444== LEAK SUMMARY:
==444==    definitely lost: 16 bytes in 1 blocks
==444==    indirectly lost: 0 bytes in 0 blocks
==444==    possibly lost: 0 bytes in 0 blocks
==444==    still reachable: 0 bytes in 0 blocks
==444==    suppressed: 0 bytes in 0 blocks
==444==
==444== For lists of detected and suppressed errors, rerun with: -s
==444== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

```

Vertailun vuoksi esimerkissä 6.17 näkyy Valgrind’n tulosteet oikein toimivalle ohjelmalle eli ”ERROR SUMMARY: 0 errors” ja ”All heap blocks were freed -- no leaks are possible”. Näin ollen ohjelma vapauttaa varaamansa muistin ja toimii siltä osin oikein.

Esimerkki 6.17. Valgrind’n raportin oleelliset osat muistia vuotamattomalle ohjelmalle

```

==450== HEAP SUMMARY:
==450==    in use at exit: 0 bytes in 0 blocks
==450==   total heap usage: 4 allocs, 4 frees, 1,072 bytes allocated
==450==
==450== All heap blocks were freed -- no leaks are possible
==450==
==450== For lists of detected and suppressed errors, rerun with: -s
==450== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Valgrindistä löytyy tarkempaa tietoa sen kotisivuilta (Valgrind 2021a) ja edellä käsitelty muistivuodon tarkistus löytyy sen manuaalin osasta 4, memcheck (Valgrind 2021b). Komentoriviltä saa myös perusohjeet ohjelman käyttöön käskyllä `valgrind --help`.

Pienen C-ohjelman tyyliohjeet

Ajan käsittely. Käsiteltäessä aikaa C-kielellä tulee käyttää systeemifunktioita, sillä ne mahdollistavat ajan esitysmuotojen muuttamisen, vertailut ja laskentaoperaatiot. Muokatessa käsin `struct tm:n` alkioita pitää varmistua, että koko tietorakenne tyhjennetään ennen sen käyttöä ja numerot noudattavat rakenteen määrittelyitä eli esim. vuosiluvusta vähennetään 1900 ennen sen tallennusta.

Linkitetyn listan toteutus. Linkitetyn listan operaatiot kuten listan luonti, läpikäynti ja tyhjennys kannattaa toteuttaa erillisinä aliohjelmina rakenteisen ohjelmoinnin mukaisesti. Tämä selkiyttää koko ohjelman rakennetta ja mahdollistaa toiminnallisuuksien tarkastelun yksi kerrallaan tarpeen mukaan erityisesti ohjelmaa muokatessa ja korjatessa.

Lista luonti. Lisättäessä listaan alkioita aliohjelmassa, kannattaa parametrina ja paluuarvona käyttää tietue-osoittimia eli arvoparametri ja paluuarvo -paria.

Yhteenveto

Kerrataan luvun osaamistavoitteet, esimerkkien aiheet ja käytetyt työkalut. Linkitetyn listan ja ajan käsittelyn esimerkit ovat sen verran kattavia, ettei tässä luvussa ole kokoavaa esimerkkiä.

Osaamistavoitteet

Tässä luvussa keskityttiin kahteen asiaan, jotka olivat linkitetty lista ja aika-tiedon käsittely C-kielellä. Linkitettyssä listassa yksittäiset alkiot sisältävät osoitteen seuraavaan alkioon ja muodostavat linkitetyn listan. Linkitetty lista on dynaaminen rakenne eli siihen kuuluvien alkioiden tarvitsema muisti varataan ja vapautetaan ohjelman suorituksen aikana. Linkitetyn listan kannalta keskeiset operaatiot ovat sen luonti eli uusien tietueiden varaaminen dynaamisesti tarpeen mukaan ja lisääminen listaan, läpikäynti ja alkioiden tietojen käsittely liukuri-osoittimen avulla sekä linkitetyn listan varaaman muistin vapauttaminen. Tässä vaiheessa sinun tulee siis pystyä toteuttamaan yhteen suuntaan linkitetty lista, jossa alkioiden määrä voi muuttua ohjelman suorituksen aikana.

Aika-tiedon osalta kävimme läpi keskeiset tietorakenteet ja funktiot, joiden avulla aika-tietoa pystyy käyttämään C-ohjelmissa. Lähtökohta on ajan esittäminen tietokoneen sisällä lukuna, joka mahdollistaa tietojen vertailun ja laskemisen Unixin Epoch-aikaan perustuen eli 1.1.1970 klo 00:00:00. Koska tietokoneen sisäinen esitystapa poikkeaa ihmisten normaalisti käyttämästä merkkijono-esityksestä, esitysmuotoja voi muuttaa perusfunktioilla aikavyöhykkeet huomioon ottaen. Sisäistä esitysmuotoa käyttämällä päivämäärillä voi myös laskea normaalisti.

Käsitellyt asiat

Tässä luvussa kävimme läpi C-ohjelmointiin liittyen seuraavat asiat:

- Linkitetyn listan määrittelyt: Esimerkki 6.1, 6.5, 6.9
- Linkitetyn listan luominen: Esimerkki 6.2, 6.5, 6.6, 6.9
- Linkitetyn listan läpikäynti: Esimerkki 6.3, 6.5, 6.7, 6.9
- Linkitetyn listan alkioiden vapauttaminen: Esimerkki 6.4, 6.5, 6.8, 6.9
- Tietokoneen ajan selvittäminen: Esimerkki 6.10, 6.11
- Merkkijonon muuttaminen aika-tiedoksi: Esimerkki 6.12, 6.14
- Päivämäärillä laskeminen: Esimerkki 6.13
- Aika-tiedon muuttaminen merkkijonoksi: Esimerkki 6.15

Käytetyt työkalut: Valgrind

Dynaamiseen muistinkäsittelyyn liittyy varatun muistin vapautus. Muistin vapautuksen puuttumien ei kuitenkaan näy tyypillisesti mitenkään ohjelman suorituksessa, joten ongelman huomaaminen

edellyttää tyypillisesti muistinkäyttöä seuraavien apuohjelmien käyttöä. Yksi tällainen perustyökalu Linux-ympäristössä on Valgrind ja tutustuimme sen käyttöön C-ohjelmien muistinkäytön tarkistamisessa. Ohjelmaa kannattaa käyttää aina dynaamisen muistinhallinnan yhteydessä ongelmien välttämiseksi.

Luku 7. Kasvavien C-ohjelmien toteutus

Tähän asti tässä oppaassa tehdyt ohjelmat ovat kaikki rajoittuneet yhteen itse tehtyyn C-lähdekooditiedostoon. Luonnollisestikaan tämä ei ole C-kielen rajoite vaan usein ohjelmat on järkevää toteuttaa useissa tiedostoissa, joten tutustutaan tällaisten C-ohjelmien periaatteisiin.

Katsotaan ensin minimaalinen useista tiedostoista muodostuva ohjelma ja tutustutaan sen jälkeen tekniikoihin, jotka auttavat useista tiedostoista muodostuvien ohjelmien hallinnassa. Viimeisenä asiana käydään läpi useiden tiedostojen kääntämisen automatisointi `make`-työkalun avulla.

Monesta tiedostosta muodostuva minimaalinen C-ohjelma

Laajempia ohjelmia tehtäessä lähtökohta on, että C-ohjelmat sisältävät usein otsikkotiedostoja `stdio.h:n` tapaan tarpeen mukaan. Sisällyttämällä ohjelman alkuun otsikkotiedoston saamme käytössä olevien ulkoisten funktioiden esittelyt mukaan ohjelman alkuun ja pystymme käyttämään niitä ohjelmassa. Itse C-kieli on pyritty pitämään pienenä ja siksi ilman kirjastoja ohjelmissa on käytössä vain kuvassa 7.1 näkyvät avainsanat. Avainsanojen joukossa on mm. perustietotyypit `int`, `float` ja `char`, valinta- ja toistorakenteet `if-switch-for-while-do` ja muutama muu tuttu käsky kuten tietue eli `struct`, tietotyyppin määrittely `typedef` ja varatun muistialueen koon selvittämisen `sizeof`. Toisaalta tietojen syöttö- ja tulostuskäskyt kuten `printf` ja `scanf`, muistinvarauskäskyt kuten `malloc` ja `free` jne. ovat kaikki sijoitettu kirjastoihin, jotta ne voidaan ottaa mukaan ohjelmaan tarvittaessa ohjelmoijan niin erikseen pyytäessä. Tämä mahdollisuus minimoida ohjelman kokoa on yksi C-kielen keskeinen ominaisuus, joka tukee pienten ohjelmien toteuttamista.

| | | | |
|-----------------------|-----------------------|-----------------------|-----------------------------|
| <code>auto</code> | <code>float</code> | <code>sizeof</code> | <code>_Atomic</code> |
| <code>break</code> | <code>for</code> | <code>static</code> | <code>_Bool</code> |
| <code>case</code> | <code>goto</code> | <code>struct</code> | <code>_Complex</code> |
| <code>char</code> | <code>if</code> | <code>switch</code> | <code>_Decimal128</code> |
| <code>const</code> | <code>inline</code> | <code>typedef</code> | <code>_Decimal32</code> |
| <code>continue</code> | <code>int</code> | <code>union</code> | <code>_Decimal64</code> |
| <code>default</code> | <code>long</code> | <code>unsigned</code> | <code>_Generic</code> |
| <code>do</code> | <code>register</code> | <code>void</code> | <code>_Imaginary</code> |
| <code>double</code> | <code>restrict</code> | <code>volatile</code> | <code>_Noreturn</code> |
| <code>else</code> | <code>return</code> | <code>while</code> | <code>_Static_assert</code> |
| <code>enum</code> | <code>short</code> | <code>_Alignas</code> | <code>_Thread_local</code> |
| <code>extern</code> | <code>signed</code> | <code>_Alignof</code> | |

Kuva 7.1. C-kielen avainsanat standardiluonnos 202x:n mukaisesti (ISO/IEC 2019, 48).

Ohjelma jaetaan useisiin tiedostoihin selkeiden peruseriaatteiden mukaan. C-ohjelma sisältää aina `main`-ohjelman, joka toimii rajanpintana käyttöjärjestelmään ja esim. komentoriviparametrit saadaan ohjelmaan `main:n` parametreina. Siksi pääohjelma sijoitetaan tyypillisesti omaan tiedostoon. Ohjelman muu koodi jaetaan muihin tiedostoihin loogisiin kokonaisuuksiin perustuen kuten tiedostonkäsittely, listankäsittely, käyttöliittymä jne. Loogisen tiedostojen lisäksi tiedostot jaetaan kahteen tiedostotyyppiin, jotka ovat lähdekoodi- ja otsikkotiedostot eli `.c` ja `.h`-tiedostot. Lähdekooditiedostoihin sijoitetaan käännettävä toiminnallinen koodi eli tyypillisesti aliohjelmat, kun taas otsikkotiedostoihin laitetaan lähdekooditiedostossa olevien aliohjelmien esittelyt sekä uusien tietorakenteiden, tietotyyppien ja vakioiden määrittelyt.

Esimerkissä 7.1 näkyy minimaalinen useaan tiedostoon jaettu C-ohjelma. Tiedosto `E7_1.c` sisältää pääohjelman, joka kutsuu itse tehtyä funktiota `summa()`. Tämä funktio on sijoitettu tiedostoon `E7_1_kirjasto.c`, jossa ei ole mitään muuta tämän funktion lisäksi yksinkertaisuuden nimissä. Esimerkin kolmas tiedosto on otsikkotiedosto `E7_1_kirjasto.h`, joka sisältää `E7_1_kirjasto.c`-tiedostossa olevan aliohjelman esittelyn. Sisällyttämällä tämä otsikkotiedosto pääohjelmätiedostoon kääntäjää tietää tiedoston käännösvaiheessa funktion parametrit ja niiden tyypit sekä paluuarvon tyypin. Molemmat lähdekooditiedostot `E7_1_kirjasto.c` ja `E7_1.c` käännetään normaalisti, mutta aiemmasta poiketen nyt käännös ja linkitys tehdään kahdessa vaiheessa. Ensimmäisessä vaiheessa kaikki lähdekooditiedostot käännetään yksitellen objektitiedostoiksi ja toisessa vaiheessa kaikki objektitiedostot linkitetään kerralla yhdeksi suoritettavaksi ohjelmaksi. Pelkkä käännös tehdään antamalla kääntäjälle optio `-c`, jonka seurauksena kääntäjä tuottaa käännetyn objektitiedoston suoritettavan ohjelman sijaan. Objektitiedostot tunnistaa `.o`-tarkenteesta, ja kun kaikki C-tiedostot on käännetty objektitiedostoiksi, voidaan ne linkittää yhteen suoritettavaksi ohjelmaksi. Esimerkin 7.1 kahden lähdekooditiedoston käännösprosessi muodostuu nyt seuraavista kolmesta käskystä:

```
gcc E7_1_kirjasto.c -c -std=c99 -pedantic -Wall
gcc E7_1.c -c -std=c99 -pedantic -Wall
gcc E7_1.o E7_1_kirjasto.o -o E7_1
```

Esimerkki 7.1. Minimaalinen kolmeen tiedostoon jaettu C-ohjelma

```
// E7_1.c - pääohjelman sisältävä tiedosto
#include <stdio.h>
#include "E7_1_kirjasto.h"

int main(void) {
    printf("Summa(2,2) on %d.\n", summa(2, 2));
    return(0);
}

////////////////////////////////////
// E7_1_kirjasto.c - itse tehdyn aliohjelman sisältävä tiedosto
int summa(int l1, int l2) {
    return (l1 + l2);
}

////////////////////////////////////
// E7_1_kirjasto.h - itse tehdyn C-koodin otsikkotiedosto
int summa(int l1, int l2);
```

Pääohjelma-tiedostossa olevat kaksi `include`-käskyä poikkeavat toisistaan. C-kielen asennukseen kuuluva `stdio.h`-otsikkotiedosto sisällytetään `<ja>`-merkeillä, kun taas itse tehty kirjasto, nyt `E7_1_kirjasto.h`, sisällytetään lainausmerkeillä `" ja "`. Käyttämällä `<ja>`-merkkejä kääntäjä etsii otsikkotiedostoa kääntäjäasetusten mukaisesta normaalista sisällytettävien tiedostojen hakemistosta, kun taas lainausmerkkien `"` sisään sijoitettua kirjastoa etsitään samasta hakemistosta käännettävän tiedoston kanssa. Siksi itse tehdyt otsikkotiedostot sisällytetään lähtökohtaisesti lainausmerkeillä lähdekooditiedostoihin.

Edellä oleva minimaalinen esimerkki osoittaa käytännössä, miten useista tiedostoista voi tehdä toimivia C-ohjelmia. Ohjelman sisältämien tiedostojen, tietorakenteiden, vakiodien jne. määrän

kasvaessa useista tiedostoista muodostuvien ohjelmien hallinta voi olla haastavaa, joten seuraavissa luvuissa tutustutaan muutamiin laajempien ohjelmien tekemisessä käytettäviin tekniikoihin.

Otsikkotiedosto .h

Edellä olevassa esimerkissä 7.1 tiedostot jaettiin lähdekoodi- ja otsikkotiedostoihin.

Lähdekooditiedostojen käsittely on selkeää, sillä ne ovat .c -tiedostoja ja ne käännetään objektikoodiksi, jotka linkitetään suoritettavaksi tiedostoksi. Mikäli lähdekooditiedosto sisältää vakioita, omia tietueita tai muita määrittelyitä, joita käytetään vain saman tiedoston sisällä, voidaan ne sisällyttää tiedostoon ennen aliohjelmia. Usein kuitenkin lähdekooditiedoston määrittelyitä tarvitaan myös kutsuvan ohjelman puolella, jolloin määrittelyt ovat yhteisiä useille tiedostoille ja ne sijoitetaan sopivaan otsikkotiedostoon.

Aliohjelmien yhteydessä luvussa 3 totesimme, että aliohjelma on esiteltävä ennen sen käyttöä ja tähän on kaksi tapaa. Ensinnäkin voimme sijoittaa aliohjelman tiedoston alkuun ennen sen kutsun sisältävää pää- tai aliohjelmaa, jolloin kääntäjä löytää aliohjelman määrittelyn ennen sen käyttöä. Toinen vaihtoehto oli sijoittaa tiedoston alkuun aliohjelman esittely, josta käy ilmi aliohjelman parametrien määrä ja niiden tietotyypit sekä aliohjelman tyyppi eli palauttaako se jotain ja jos palauttaa niin mitä. Tämä jälkimmäinen ratkaisu mahdollisti aliohjelman määrittelyn eli suoritettavan koodin sijoittamisen tiedoston loppuun esittelyiden jälkeen ja se sopii hyvin myös otsikkotiedostojen kanssa. Tyypillisesti lähdekooditiedostossa olevien aliohjelmien esittelyt laitetaan otsikkotiedostoon ja kun otsikkotiedosto sisällytetään lähdekooditiedoston alkuun esikäsitteilyvaiheessa, löytyy tiedostosta aliohjelmien esittelyt ja ohjelma saadaan käännettyä.

Otsikkotiedosto sisältää yhden moduulin tietotyyppien ja vakioiden määrittelyt sekä aliohjelmien esittelyt. Moduuli tarkoittaa yhtä loogista kokonaisuutta, joka voi vaihdella yhdestä aliohjelmasta useista tiedostoista muodostuvaan kirjastoon. Tässä yhteydessä moduuli tarkoittaa selkeyden vuoksi yhtä C-lähdekooditiedostoa. Kun kaikki moduulin ulkopuolelle näkyvät määrittelyt ja esittelyt laitetaan yhteen otsikkotiedostoon, voidaan moduulia hyödyntää muissa tiedostoissa sisällyttämällä otsikkotiedosto niihin.

Useista tiedostoista muodostuvan ohjelman lähdekoodi- ja otsikkotiedostot kannattaa nimetä systemaattisesti ja siten, että yhteenkuuluvat tiedostot on helppo tunnistaa. Tämä onnistuu esimerkiksi antamalla tiedostoille sama nimi ja eri tarkanteet kuten `tiedosto.c` ja `tiedosto.h`. Tällöin `tiedosto.c/h` tiedostot sisältäisivät todennäköisesti tiedostonkäsitteilyyn liittyvän koodin, kun taas `lista.c/h` tiedostot sisältäisivät linkitettyyn listaan liittyvän koodin ja `paa.c/h` pääohjelmaan liittyvän koodin. Yleisemmän uudelleenkäytön kannalta saatetaan luonnollisesti tarvita otsikkotiedosto, joka edustaa monista tiedostoista muodostuvaa uudelleenkäytettävää moduulia samalla tavoin kuin esimerkiksi `stdio.h` ja `string.h` -tiedostot tekevät. Aina tiedostojen suora linkitys ei onnistu, mutta usein sillä pääsee hyvin alkuun.

Tiedostojen määrän kasvaessa otsikkotietojen sisällytys lähdekooditiedostoihin voi muodostua ongelmaksi, sillä vaikka tunnukset voi esitellä monta kertaa, saa ne määritellä vain yhden kerran. Esimerkiksi linkitettyä listaa käsittelevässä ohjelmassa pitää tyypillisesti määritellä listan alku-osoitin pääohjelmassa, varsinaiset listankäsittelyoperaatiot ovat omassa lista-moduulissa ja näiden lisäksi tiedostokäsittelymoduulin pitää myös tietää listan alkiodien käyttämän tietueen rakenne. Näin ollen lista-otsikkotiedosto tulee sisällyttää kaikkiin kolmeen lähdekooditiedostoon, koska se sisältää käytettävän tietueen määrittelyn. Jos ohjelmaan lisätään uusi moduuli tietojen analysointia varten, pitää se tyypillisesti sisällyttää myös kaikkiin muihin moduuleihin. Ja kun sisällytettävien tiedostojen lukumäärä kasvaa, pitää niiden riippuvuuksien kanssa olla tarkkana, sillä tunnukset pitää olla esitelty tai määritelty ennen niiden käyttöä, mutta niitä ei saa määritellä montaa kertaa samassa

ohjelmassa. Seuraavassa esimerkissä saman tiedoston useista sisällytyksistä aiheutuvat ongelmat on estetty otsikkotiedoston ehdollisella sisällytyksellä lähdekooditiedostoon.

Esimerkissä 7.2 on otsikkotiedosto, jossa on määritelty vakio, tietue, uusi tietotyyppi ja esitelty aliohjelmat. Kaikki tiedoston sisältö on laitettu `#ifndef` (if not defined) -ehdollisen määrittelyn sisälle, jolloin ensimmäisen sisällytyskäsken yhteydessä tiedoston sisältö tulee näkyville ja samalla määritellään vakio `lista_h`, jonka määrittely estää tiedoston sisällyttämisen useita kertoja. Tämän esimerkin koodi on oppaan esimerkki 6.9 jaettuna kolmeen tiedostoon `E7_2_lista.h`, `E7_2_lista.c` ja `E7_2.c`, joten koko ohjelman koodi on nähtävillä edellisessä luvussa. Tässä ohjelmassa tiedostojen sisällöt on suunniteltu niin, ettei ehdollista sisällytystä tarvita, joten tässä esimerkissä näkyy ehdollisen määrittelyn idea ja toteutus ilman tarvetta sille.

Esimerkki 7.2. Otsikkotiedoston sisältö ja sisällytys `#ifndef`-direktiivillä

// E7_2_lista.h - Otsikkotiedosto ehdollisella sisällytyksellä

```
#ifndef lista_h
#define lista_h
#define LKM 3

typedef struct asiakas {
    int iNumero;
    struct asiakas *pSeuraava;
} ASIAKAS;

ASIAKAS *lisaaSolmu(ASIAKAS *pA, int x);
void tulosta(ASIAKAS *pA);
ASIAKAS *tyhjenna(ASIAKAS *pA);

#endif
```

// E7_2_lista.c - Tiedosto sisältää funktioiden määrittelyt

```
#include <stdio.h>
#include <stdlib.h>
#include "E7_2_lista.h"

ASIAKAS *lisaaSolmu(ASIAKAS *pA, int x) { ... }
void tulosta(ASIAKAS *pA) { ... }
ASIAKAS *tyhjenna(ASIAKAS *pA) { ... }
```

// E7_2.c - Tiedosto sisältää main-ohjelma määrittelyn

```
#include <stdio.h>
#include <stdlib.h>
#include "E7_2_lista.h"

int main(void) {
    ...
    pAlku = lisaaSolmu(pAlku, i);
    ...
    tulosta(pAlku);
    ...
    pAlku = tyhjenna(pAlku);
    ...
    return(0);
}
```

Otsikkotiedostoja kannattaa sisällyttää vain lähdekooditiedostoihin. Tällöin jokaisen lähdekooditiedoston tarvitsemat otsikkotiedostot pitää käydä läpi, mikä auttaa välttämään tiedostojen turhia ja/tai useita sisällytyksiä. Siksi otsikkotiedostojen sisällytys vain lähdekooditiedostoihin johtaa usein selkeään ohjelmarakenteeseen ja on tässä oppaassa suositeltava tapa. Teknisesti otsikkotiedostoja voi sisällyttää myös otsikkotiedostoon, jolloin tarvittavat otsikkotiedostot voi saada kirjattua yhteen otsikkotiedostoon. Tämä lähestymistapa johtaa helpolla samojen otsikkotiedostojen useisiin sisällytyksiin ja edellä oleva ehdollinen sisällytys tulee aktiiviseen käyttöön.

Edellä kuvatut periaatteet toimivat tyypillisesti tämän oppaan käsittelemissä pienissä ohjelmissa, jotka rupeavat kasvamaan. Laajat C-ohjelmat voivat kuitenkin koostua miljoonista koodiriveistä, jolloin tiedostoja voi olla satoja tai tuhansia ja siksi niiden hallinta pitää suunnitella tarkasti ja valita tapauskohtaisesti sopivat käytännöt. Lähtökohtaisesti laajoissa projekteissa on omat projektiin räätälöidyt käytännöt tiedostojen nimeämiseen samoin kuin moneen muuhun asiaan. Hyvä esimerkki laajan ohjelmiston tyyliohjeesta löytyy esim. GNU-projektista (Stallman et al. 2021).

Laajan ohjelman kääntäminen – make ja Makefile

Kuten esimerkin 7.1 yhteydessä kävi ilmi, laajan ohjelman kääntäminen tarkoittaa jokaisen lähdekooditiedoston kääntämistä ensin erikseen ja sen jälkeen kaikkien objektikoodien linkittämistä suoritettavaksi ohjelmaksi. Laajassa projektissa satojen tiedostojen kanssa tämä ei ole houkutteleva tehtävä ja siksi se onkin automatisoitavissa työkalujen avulla. Unix-puolen perinteinen työkalu tähän tehtävään on `make`, joka lukee `Makefile` -tiedostossa olevat käännöskäskyt ja muodostaa niiden avulla suoritettavan ohjelman. `Makefile` nimenä on samat käännöskäskyt, joilla ohjelma voidaan muodostaa yksitellen kääntämällä komentorivillä. Jokaista käännöskäskyä edeltää ”hallinnollinen rivi”, jossa näkyy ensimmäisenä tavoitteena oleva tiedosto ja sen jälkeen tiedostot, joista tavoitetiedosto tehdään. `make` käy kaikki nämä käskyt läpi ja kun tiedosto on oikein rakennettu, saadaan lopputuloksena suoritettava ohjelma.

Esimerkissä 7.3 näkyy `Makefile`, jolla saadaan käännettyä esimerkin 7.2 useista tiedostoista muodostuva ohjelma. Ensimmäinen rivi alkaa #-merkillä ja se on kommenttirivi. Tämän jälkeen toisella rivillä kerrotaan, että tiedosto `E7_2` muodostetaan tiedostoista `E7_2.o` ja `E7_2_lista.o`. Näin ollen `make` katsoo näiden kolmen tiedoston luontiajat ja jos tavoitteena oleva tiedosto on luotu muiden jälkeen, ei sitä tarvitse päivittää, koska se on aikaleima-analyysin perusteella tehty muiden tiedostojen pohjalta.

```
E7_2: E7_2.o E7_2_lista.o
```

Mikäli objektitiedosto tai molemmat ovat tavoitetiedostoa uudempia, linkitetään nämä tiedostot uudestaan tavoitetiedoston uuden version tuottamiseksi alla olevalla käskyllä. Huomaa, että rivi alkaa sarkainmerkillä (tabulaattori, `\t`) ja tätä erotinmerkkiä ei voi korvata muilla merkeillä, esim. yhdellä tai usealla välilyönnillä.

```
gcc E7_2.o E7_2_lista.o -o E7_2
```

`make` käy näin läpi kaikki `Makefile`:ssä olevat rivit ja päivittää ohjelmatiedostot tarpeen mukaan. Objektitiedostojen kääntämisen tarve perustuu luonnollisesti itse lähdekooditiedostoon, mutta sen lisäksi kaikki ko. tiedostoon sisällytetyt otsikkotiedostot tulee tarkistaa muutosten varalta. Siksi riippuvuussuhteissa nämä kaikki otsikkotiedostot mainitaan erikseen.

```
L7lista.o: E7_2_lista.c E7_2_lista.h
```

Esimerkki 7.3. Makefile. Huomaa sarkainmerkit (nuoli) ja välilyönnit (piste merkkien välissä)

```
# Makefile E7_2 20210222 un -- Tämä on kommenttirivi
E7_2: E7_2.o E7_2_lista.o
    → gcc E7_2.o E7_2_lista.o -o E7_2
E7_2.o: E7_2.c E7_2_lista.h
    → gcc E7_2.c -c -std=c99 -pedantic -Wall
E7_2_lista.o: E7_2_lista.c E7_2_lista.h
    → gcc E7_2_lista.c -c -std=c99 -pedantic -Wall
.....

```

`make` ja `Makefile` ovat Unixin perustyökaluja, joilla voidaan automatisoida kääntäminen. Kaikissa uusissa IDE-työkaluissa on vastaavat työkalut, vaikka ne onkin tyypillisesti piilotettu projektitiedosto-konseptin taakse. Projektitiedosto hoitaa projektin kääntämisen graafisen käyttöliittymän kautta annetuilla käännösoptioilla eli vastaava konsepti hieman erilaisesta toteutuksesta huolimatta. `make` on erittäin monipuolinen työkalu ja sitä käsitteleviä kirjoja on paljon, mutta erityisen helppo asiaan perehtymistä on jatkaa ilmaisen kirjan avulla Mecklenburg (2005).

Ohjelmien kääntämisen lisäksi nykyään on erilaisia työkaluja helpottamaan ja automatisoimaan ohjelmien julkaisua. Myös julkaisua edeltää tyypillisesti tietyt rutiinitoimenpiteet ja niiden automatisoinnilla varmistetaan kaikkien toimenpiteiden teko oikeassa järjestyksessä sekä nopeutetaan niiden tekemistä. Ohjelmoimalla voi siis automatisoida myös ohjelmoijan tehtäviä, joten omaa työtä ja toimintaa kannattaa kehittää aina tarpeen ja tilaisuuden tullen.

Yhteenveto

Kerrataan luvun osaamistavoitteet, esimerkkien aiheet sekä käytetyt työkalut.

Osaamistavoitteet

Tässä luvussa kävimme läpi yhdestä lähdekooditiedostosta muodostuvan C-ohjelman jakamisen useisiin tiedostoihin ja kääntämisen automatisoinnin. Yksinkertaisessa tapauksessa riittää jakaa lähdekooditiedosto useisiin lähdekoodi- ja otsikkotiedostoihin, jonka jälkeen lähdekooditiedostot käännetään erikseen ja linkitetään suoritettavaksi ohjelmaksi. Usein otsikkotiedostojen sisältöjä ja sisällytystä lähdekooditiedostoihin joutuu harkitsemaan tarkasti, jotta kaikki tunnukset, tietorakenteet ja aliohjelmat tulevat esiteltyä ja määriteltyä oikein ennen käyttöä. Lisäksi tutustuimme `make`:en ja `Makefile`:en, joten tämän jälkeen pystyt suunnittelemaan useista tiedostoista muodostuvan C-ohjelman rakenteen sekä automatisoimaan sen kääntämisen.

Käsitellyt asiat

Tässä luvussa kävimme läpi C-ohjelmointiin liittyen seuraavat asiat:

- Useista tiedostoista muodostuva C-ohjelma: Esimerkki 7.1
- Otsikkotiedoston ehdollinen sisällytys: Esimerkki 7.2
- Makefile: Esimerkki 7.3

Käytetyt työkalut: `make` ja `Makefile`

`make` on Linuxin perustyökaluja ohjelmoinnissa ja se tulee `gcc`-paketin mukana. Siksi se tuli asennettua jo `gcc`:n asennuksen yhteydessä. IDE-työkaluissa on vastaavat työkalut integroituna graafisen käyttöliittymän taakse, joten työkalu tallentaa projektitiedostoon käyttäjän tekemät asetukset ja näitä tiedostoja ei tyypillisesti pysty editoimaan itse. Mutta taustalla olevat `make` ja `Makefile` on hyvä ymmärtää, sillä aina joskus asioita joutuu tekemään ja säätämään käsin, jolloin perusteiden ymmärrys on hyvä lähtökohta.

Loppusanat

Olemme tässä oppaassa käyneet läpi C-ohjelmoinnin perusteet esimerkkien avulla ja tutustuneet ohjelmien tekemiseen Windows'n WSL-ympäristössä Linux-työkaluilla ja VSC-editorilla. Perusteiden oppimisen jälkeen ohjelmointi opettaa ohjelmoijaa eli kannattaa jatkaa ohjelmointia ja miettiä, mitä on oikeasti tekemässä ja sen ymmärtämisen pohjalta toteuttaa sitten halutut toiminnot ja ominaisuudet. Koodaamisen lisäksi kannattaa katsoa muiden kirjoittamia ohjelmia avoimin mielin, sillä kuluneen 50 vuoden aikana C-koodia on kirjoitettu paljon ja useimpiin haastaviinkin ongelmiin löytyy hyvin toimivia malleja.

C-ohjelmointia käsitellään monissa kirjoissa, mutta edelleen paras lähde C-kielen ymmärtämiseen on Kernighan ja Ritchie (1988) eli The C Programming Language -kirja. C-kielen käytön kannalta hyvä opas on Eric Hussin C-referenssikirjasto (Huss 1997) ja C-kieltä laajemmin käytännön ohjelmointia käsitellään Kernighan ja Pike (2005) -kirjassa The Practice of Programming. C-kieleen on saatavilla myös suomenkielisiä oppaita, joita kannattaa etsiä Internetin hakukoneilla. Hakuja voi rajata suomenkielisiin sivustoihin, mutta luonnollisesti englanninkielistä materiaalia on tarjolla paljon enemmän. Kannattaa myös katsoa tämän oppaan liite 3, johon on koottu keskeisten kirjastojen yleisimpiä funktioita.

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, viitattu 28.5.2021.
- Huss, Eric. 1997. The C Library Reference Guide. Saatavilla osoitteesta http://www.acm.uiuc.edu/webmonkeys/book/c_guide/, viitattu 18.12.2007.
- ISO/IEC. 2018. Programming languages - C, N2454 working draft 9899:2018.
- ISO/IEC. 2019. Programming languages - C, N2454 working draft 9899:202x (E), November 18.
- 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.
- Kerrisk, Michael. 2021a. The man-pages project. <https://man7.org/linux/man-pages/index.html>, viitattu 28.5.2021.
- Kerrisk, Michael. 2021b. Linux- programmer's manual introduction. <https://man7.org/linux/man-pages/man3/intro.3.html>, viitattu 28.5.2021.
- Kyttälä, Lauri ja Nikula, Uolevi. 2012. C-kieli ja käytännön ohjelmointi – Osa 2, Versio 1.0. Lappeenrannan teknillinen yliopisto, Tietotekniikan käsikirjat 15.
- Mecklenburg, Robert. 2004. Managing Projects with GNU Make, 3rd edition, <http://oreilly.com/catalog/make3/book/index.csp>, viitattu 28.5.2021.
- Moody, Glen. 2001. Kapinakoodi, Tammi.
- Stallman, Richard et al. 2021. GNU Coding Standards. <https://www.gnu.org/prep/standards/standards.html>, viitattu 28.5.2021.
- Statcounter. 2021. Desktop operating system market share worldwide 2021, <https://gs.statcounter.com/os-market-share/desktop/worldwide/#yearly-2021-2021-bar>, viitattu 16.6.2021.
- Robot Wisdom. 2011. <http://www.robotwisdom.com/linux/timeline.html>, viitattu 10.5.2011.
- Tiobe 2021. TIOBE Index for May 2021, <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>, viitattu 21.5.2021.
- Valgrind Developers. 2021a. Valgrind: an instrumentation framework for building dynamic analysis tools. <https://www.valgrind.org/>, viitattu 28.5.2021.
- Valgrind Developers. 2021b. Valgrind User Manual Chapter 4. Memcheck: a memory error detector. <https://www.valgrind.org/docs/manual/mc-manual.html>, viitattu 28.5.2021.
- Vanhala, Erno ja Nikula, Uolevi. 2020. Python 3 – ohjelmointiopas versio 1.2.1. LUT-yliopisto. <http://urn.fi/URN:ISBN:978-952-335-622-1>, viitattu 28.5.2021
- Wikipedia. 2021. Hungarian notation. https://en.wikipedia.org/wiki/Hungarian_notation, viitattu 28.5.2021.

Liitteet

Tähän oppaaseen kuuluu seuraavat liitteet:

1. C-ohjelmointiympäristön asennusohje
2. Visual Studio Code -editorin käyttöohje
3. Keskeisten C-kirjastojen funktioita

Liite 1. C-ohjelmointiympäristön asennusohje

| | |
|--|---|
| 1. Johdanto | 1 |
| 1.1. Suositeltava kehitysympäristö | 1 |
| 1.2. Muut kehitysympäristöt | 1 |
| 2. Suositusympäristön asennusohjeet..... | 2 |
| 2.1. WSL eli Windows Subsystem for Linux | 2 |
| 2.2. Linux-kehitysympäristö | 2 |
| 2.3. Visual Studio Code | 3 |
| 2.4. WSL:n etäkäyttö VSC:stä | 3 |
| 2.5. Linux-terminaalin käyttö VSC:ssä..... | 4 |
| 2.6. Tiedostojen siirto Windows'n ja WSL:n välillä | 4 |
| 2.7. Huomioita | 5 |
| 3. Linux ja Apple -ympäristöt..... | 5 |
| 3.1. Linux virtuaalikone | 6 |

1. Johdanto

Tämä on LUTin C-ohjelmointikurssin asennusohje. Kehitysympäristö on valittu siten, että ohjelmat tehdään Windows-sovelluksessa ja näin ollen käyttöliittymä on monille tuttu. Toisaalta ohjelmien kääntäminen ja suorittaminen tehdään Linux-ympäristössä hyväksi koetuilla työkaluilla gcc, make ja Valgrind. Linux-ympäristö sopii hyvin myös ohjelmien suorittamiseen, sillä se auttaa havaitsemaan ohjelmointivirheet ilmoittamalla monista virheistä, esim. segmentation fault, ja tekemällä core dumped'n. Tämä auttaa ohjelmien sisältämien virheiden havaitsemista ja helpottaa samalla niiden poistamista ohjelmista.

1.1. Suositeltava kehitysympäristö

Tämän oppaan yhteydessä **suositeltava kehitysympäristö** on seuraava:

- Tietokoneen käyttöjärjestelmä: Win10, uusin/päivitetty versio
- Koodieditori: Visual Studio Code, VSC, uusin versio
- Kääntämistä ja suorittamista varten: WSL1 ja Ubuntu 20.04 LTS (Linux)
- GNU-kehitystyökalut WSL:ssä: gcc, make ja Valgrind

1.2. Muut kehitysympäristöt

Myös muita kehitysympäristöjä voi käyttää, mutta niiden osalta jokainen vastaa itse ympäristön toimivuudesta. Ohjelmien ”oikea toiminta” tarkoittaa sen toimintaa em. suositusympäristössä.

- Mac ja Linux tietokone/virtuaalikone toimivat lähtökohtaisesti vastaavalla tavalla, kunhan asennus tehdään samoilla periaatteilla ja ohjelmistoversioilla
- LUT tarjoaa Linux-ympäristöjä käyttöön seuraavilla tavoilla:
 - Etäyhteys <https://one.lut.fi>, valitse Apps -kohdasta ”LUT virtual Desktop” ja avaa ”Desktop 3 – Ubuntu Linux”. Tämä vastaa EXAM-tentissä käytettävää Linux-ympäristöä
 - Linux-luokka 6218

2. Suositusympäristön asennusohjeet

Tässä luvussa on ohjelmien asennusohjeet. Ohjelmat kannattaa asentaa samassa järjestyksessä kuin ne ovat tässä ohjeessa. Periaatteessa nämä ohjeet riittävät normaaliin asennukseen, mutta tarvittaessa tarkempia ohjeita löytyy jokaisen kohdan lopussa olevan linkin takaa.

2.1. WSL eli Windows Subsystem for Linux

- Tämän oppaan tehtävien tekemiseen sopii WSL1
 1. Salli WSL:n käyttö tietokoneessasi
 - Kirjoita Windows’n Etsi-ikkunaan ”Turn Windows feature on or off” / ”Ota Windowsin ominaisuuksia käyttöön tai poista niitä käytöstä” ja käynnistä tämä sovellus, edellyttää ylläpitäjän oikeuksia
 - Laita rasti kohtaan ”Windows Subsystem for Linux” / ”Windows alijärjestelmä Linuxille”
 - Valinnan hyväksymisen jälkeen Windows pitää käynnistää uudestaan
 2. Asenna Ubuntu Linux 20.04 LTS Microsoft-kaupasta
<https://www.microsoft.com/fi-fi/p/ubuntu-2004-lts/9n6svws3rx71>
 - Luo itsellesi käyttäjätunnus Linuxiin asennuksen jälkeen. Älä käytä LUTin tunnuksia tai salasanaa vaan esim. omaa nimeäsi kuten *ville*
- Nyt sinulla on WSL1 eli Ubuntu Linux käytössäsi Win10-käyttöjärjestelmän rinnalla
 - Tässä oppaassa riittää WSL1 ja koska sen asennusohje on helpompi, käytämme sitä. Myös WSL2:ta voi käyttää niin halutessaan
- Tarvittaessa katso tarkemmat asennusohjeet seuraavasta osoitteesta
<https://docs.microsoft.com/en-us/windows/wsl/install-win10>
 - Yllä on WSL1 asennusohjeet, mutta linkin takana on laajemmat asennusohjeet WSL 1 ja 2:lle, rekisteröitymistä edellyttävä ”Simplified install” ja yllä olevaa laajemmat 6 vaihetta sisältävät ”Manual install” ohjeet. Täältä löytyy myös vinkkejä ongelmiin eli *Troubleshooting installation* -kohta

2.2. Linux-kehitysympäristö

- Käynnistä edellä asennettu Ubuntu, päivitä se ja asenna seuraavat ohjelmat kirjoittamalla alla olevat käskyt terminaaliin:
 - Päivitysten haku koneelle: `sudo apt update`
 - Päivitysten asennus koneelle: `sudo apt upgrade`
 - GNU-työkalut: `sudo apt install build-essential gdb`
 - Valgrid: `sudo apt install valgrind`
- Tarkista asennuksen onnistuminen testaamalla, että gcc ja make ovat käytössäsi eli että ne löytyvät Linux-terminaalista seuraavilla käskyillä
 - `gcc --version`
 - `make --version`

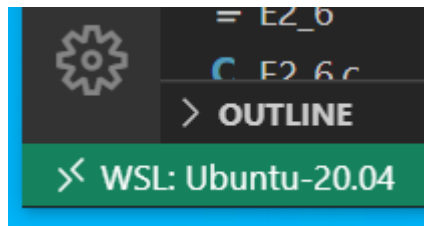
- `valgrind --version`
- Versionumeroilla ei ole nyt merkitystä vaan oleellista on varmistua, että em. ohjelmat ovat käytettävissä. Toisin sanoen, jos em. käsky palauttaa tekstin "Command 'xx' not found, but can be installed with:", ei asennus onnistunut vaan joudut yrittämään uudestaan suorittamalla edellisen ilmoituksen jatkona olevan käskyn
- Nyt sinulla on Linuxissa tarvittavat C-kehitystyökalut asennettuna. Alla olevan linkin takaa löytyy tarkempia ohjeita C++ kehitysympäristön konfigurointiin
 - <https://code.visualstudio.com/docs/cpp/config-wsl>
- Muista, että Linuxissa ylläpitäjä hoitaa käyttöjärjestelmän päivitykset ml. tunnettujen tietoturva-aukkojen korjaukset. Päivitykset kannattaa tarkistaa vähintään kuukausittain
 - Ohjelmistojen uusimpien versioiden haku: `sudo apt update`
 - Ohjelmistojen päivitys uusimpiin versioihin: `sudo apt upgrade`

2.3. Visual Studio Code

- Mene Windows'n selaimella seuraavaan osoitteeseen:
<https://code.visualstudio.com/Download>
 - Valitse omalle koneellesi sopiva asennusohjelma, lähtökohtaisesti Win10, User Installer, 64 bit (tai 32 bit tai ARM)
 - Voit tarkistaa käyttämäsi prosessorityypin painamalla Windows- ja Pause-näppäimiä samanaikaisesti ja katso dialogista Järjestelmätyyppi/System type -kohdasta
 - Asennuksen yhteydessä kannattaa tehdä seuraavat asetukset
 - Add "Open with Code" action to Windows Explorer file context menu
 - Add "Open with Code" action to Windows Explorer directory context menu
 - Register Code as an editor for supported file types
 - Add to PATH (requires shell restart)
- Asennuksen jälkeen käynnistä VSC ja asenna siihen seuraavat laajennokset eli extensions. Laajennokset kannattaa asentaa wsl:ään (remote wsl) eikä paikallisesti (local)
 - Remote – WSL (Microsoft)
 - C/C++ (Microsoft)
 - CMake Tools (Microsoft)
 - VS Code Printing Free (PD Consulting; jos haluat tulostaa paperille)
- Nyt sinulla on VSC editori asennettuna Windows'ssa, muttei vielä yhdistettynä WSL-kehitystyökaluihin
- Katso tarvittaessa asennusohjeet osoitteesta <https://code.visualstudio.com/docs>

2.4. WSL:n etäkäyttö VSC:stä

- Viimeisenä vaiheena on yhdistää WSL:n työkalut VSC:n terminaaliin
- Käynnistä VSC ja asenna Remote WSL laajennos. Tässä yhteydessä VSC tekee WSL serverin asennuksen ja sen päätteeksi VSC:n vasempaan alanurkkaan tulee vihreä laatikko, jossa lukee WSL: Ubuntu-20.04
 - Asennuksen aikana VSC:ssä näkyy ikkuna, jossa lukee "Starting VS Code in WSL (Ubuntu-20.04): Downloading Server..."



- Asennukset on nyt tehty ja voimme siirtyä ohjelmoimaan

2.5. Linux-terminaalin käyttö VSC:ssä

- Varmistetaan ohjelmointiympäristön toimivuus tekemällä pieni C-ohjelma
- Luo VSC:ssä uusi tiedosto CTRL-N -käskyllä ja kopioi seuraava C-koodi siihen

```
#include <stdio.h>
int main(void) {
    printf("Heippa maailma!\n");
    return(0);
}
```

- Tallenna tiedosto CTRL-S -käskyllä VSC:n ehdottamaan hakemistoon nimellä oma.c
- Siirry VSC:n terminaaliin CTRL-Ö -käskyllä. Terminaaliin tulee nyt sama Linux-promptti kuin edellä GNU-työkalujen asennuksen yhteydessä. Tee ohjelman käännös ja suoritus
 - Käännös: gcc oma.c -o oma
 - Suoritus: ./oma
- Ohjelma tulostaa "Heippa maailma!" -tekstin ja rivinvaihdon
- Kehitysympäristön asennus on nyt valmis ja toimiva
- Katso tarvittaessa tarkemmat asennusohjeet osoitteesta <https://code.visualstudio.com/docs/cpp/config-wsl>

2.6. Tiedostojen siirto Windows'n ja WSL:n välillä

VSC-ohjelman käyttöliittymä ajetaan Windows-käyttöjärjestelmässä ja ohjelmien kääntäminen sekä suoritus tehdään Linux-käyttöjärjestelmässä, ts. kehitystä tehdään samaan aikaan kahdessa eri käyttöjärjestelmässä. Käännettävät tiedostot voivat olla kummassa tahansa järjestelmässä, mutta paremman suorituskyvyn vuoksi tiedostot kannattaa pitää Linuxin puolella WSL:ssä. Mikäli WSL:ssä olevia tiedostoja haluaa kopioida tiedostoina johonkin Windows'n resurssienhallinnalla, ovat suoritettava ohjelma ja levyjärjestelmä taas eri järjestelmissä ja tiedostoihin ei pääse käsiksi yhtä helpolla kuin toimittaessa vain yhdessä järjestelmässä. Tämä kuitenkin onnistuu esimerkiksi seuraavilla tavoilla

- **Tiedostojen sisällöt voi siirtää leikepöydän kautta.** Tämä toimii samalla tavalla kuin palautettaessa ohjelmakoodi Viopien. Voit esim. valita koko tiedoston sisällön VSC:n ikkunassa (CTRL-A), kopioida sen leikepöydälle (CTRL-C) ja liittää sen toisen järjestelmän ikkunaan (CTRL-V)
- **Tiedostot voi kopioida Windows'n Explorer'illa.** WSL'n käyttämä hakemisto löytyy esim. käynnistämällä ensin WSL-terminaali ja sitten Explorer työhakemistossa seuraavalla käskyllä (ohjelman nimi + välilyönti + piste):
 - explorer.exe .

- **Explorerin osoiteriville voi kirjoittaa halutun hakemiston osoitteen**, jonka alkuosa on seuraava
 - \\wsl\$\Ubuntu-20.04\home\un
 - Edellä olevassa polussa on wsl\$ on käyttöjärjestelmän ylläpitämä muuttuja, joka viittaa wsl:n juurihakemistoon. Sen jälkeen on asennetun Linux-version nimi (Ubuntu-20.04) ja Linux-hakemistoista mennään kotihakemistoon ja sitten luodun käyttäjätunnuksen juurihakemistoon (un)
- **Tiedostoja voi käsitellä myös WSL:n puolelta.** Windows'n levyasemat löytyvät mountattuina WSL:ään /mnt-hakemistosta, joten VSC:ssä voi valita normaalisti halutun hakemiston Windows'n hakemistosta

Tällä kurssilla käsiteltävät tiedostot ja niiden lukumäärät ovat sen verran pieniä, että tiedostoja voi säilyttää WSL:n tai Windows'n puolella eikä se vaikuta merkittävästi suorituskyykyyn tms. Laajemmissa projekteissa kaikki tiedostot kannattaa laittaa samaan hakemistoon WSL:ään kääntämisen nopeuden vuoksi.

2.7. Huomioita

- **Ylläpitäjän tehtäviä.** Käyttöjärjestelmän ml. WSL:n asennus ei ole jokapäiväisiä toimenpiteitä, joten ne edellyttävät usein ylläpitäjän oikeuksia eli ne pitää suorittaa PowerShell'ssä Ylläpitäjänä. Omalla koneella tämä tyypillisesti onnistuu, mutta esim. työnantajan koneella ei, joten tällaisissa tapauksissa pitää harkita saako asennuksen tehtyä esim. tietohallinnon kautta vai pitääkö keksiä muita ratkaisuja kuten esim. siirtyä käyttämään ohjeen alussa mainittua etäyhteyttä tai Linux-luokkaa
- **Linuxin päivitys.** Linuxin asennuksen jälkeen se on syytä päivittää ajan tasalle. Lähtökohtaisesti oppaan läpikäynnin aikana eli muutaman kuukauden aikana ei pitäisi tarvita järjestelmän päivittämistä. Tämän liitteen kohdassa 2.2 oli kuitenkin Linuxin päivitysohjeet, joita kannattaa noudattaa, jos Linux jättää omalle koneelle pidemmäksi aikaa. Tämä jää käyttäjän omalle vastuulle.

3. Linux ja Apple -ympäristöt

Applen taustalla on Unix-ideat, joten Applen tuotteet toimivat komentorivillä pitkälti samalla tavalla kuin Linux-tuotteet. Siksi oppaan tehtävien tekemiseen käytettävät ympäristöt saa asennettua niihin suhteellisen vähällä työllä ja samoilla ideoilla.

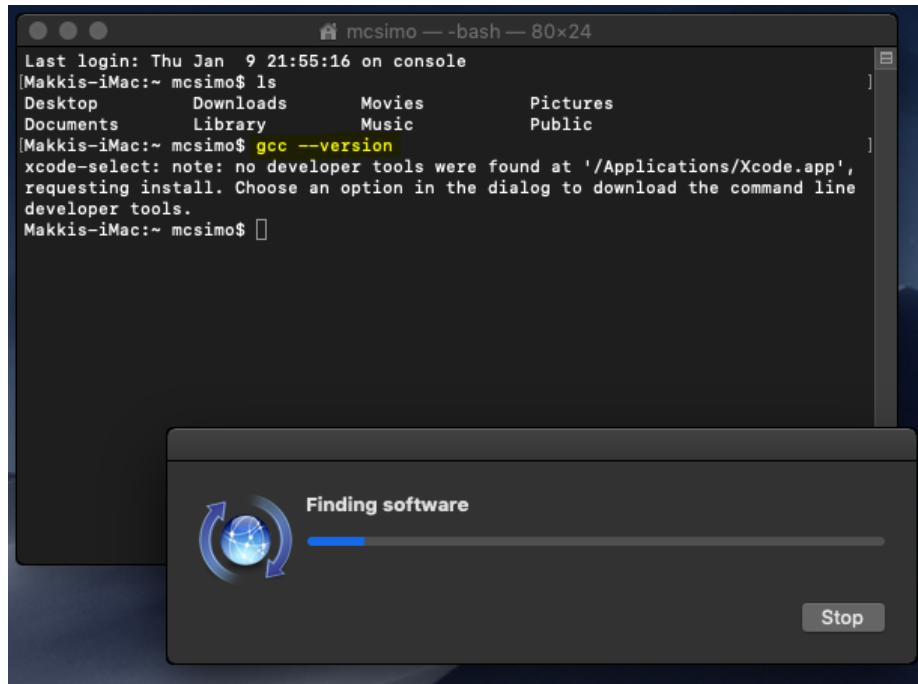
Asenna Visual Studio Code –editori alla olevan linkin kautta:

- <https://code.visualstudio.com/download>

Asenna editoriin seuraavat laajennukset/extensions:

- C/C++, Microsoft
- CMake Tools, Microsoft
- VS Code Printing Free, PD Consulting (jos haluat tulostaa paperille)

Kääntäjä eli gcc löytyy järjestelmistä esiasennettuna ja sitä voi käyttää terminaalista samalla tavoin Windows'ssa käytetään WSL:n gcc:tä. Voit varmistua kääntäjän olemassaolosta kirjoittamalla terminaaliiin komennon `gcc --version`. Jos gcc-kääntäjä on asennettu niin terminaaliiin pitäisi tulostua sen versio. Muussa tapauksessa järjestelmän pitäisi kysyä, että haluatko asentaa *Command Line Developer Tools*, johon kannattaa vastata myöntävästi. Alla olevassa kuvassa näkyvän asennusprosessin pitäisi alkaa.



Kääntäjän asentaminen Applen järjestelmään

3.1. Linux virtuaalikone

Mikäli haluat asentaa itsellesi Linux-virtuaalikoneen, sellainen löytyy asennusohjeiden kanssa esim. osoitteesta <http://www.vmware.com/products/player/>

Virtuaalikone tarvitsee kaverikseen käyttöjärjestelmän ja Ubuntu-Linux käyttöjärjestelmä löytyy asennusohjeiden kanssa esim. osoitteesta <http://www.ubuntu-fi.org/>

Liite 2. Visual Studio Code -editorin käyttöohje

| | |
|--|---|
| 1. Visual Studio Code -koodieditori | 1 |
| 1.1. VSC:n keskeiset ikkunat | 1 |
| 1.2. Explorer eli kansion kanssa työskentely | 1 |
| 1.3. Editori eli tiedostojen kanssa työskentely | 2 |
| 1.4. Terminaali eli ohjelman kääntäminen ja suorittaminen..... | 3 |
| 1.5. Konfigurointi..... | 4 |

1. Visual Studio Code -koodieditori

Tässä ohjeessa on kerrottu VSC:n perusteet, joilla pääset tekemään ohjelmia. Kyseessä on editori eli muokkausohjelma, joka toimii pitkälle samalla tavalla kuin kaikki Windows-editorit. Tekstieditorilla kuten MS-Word ja koodieditorilla on kuitenkin eri käyttötarkoitukset ihan niin kuin urheilu- ja kuorma-autolla, joten kannattaa käyttää omaan tehtävään sopivaa editoria. Nyt koodataan, joten nyt käytetään VSC-koodieditoria.

1.1. VSC:n keskeiset ikkunat

VSC:ssä kannattaa pitää auki kolme ikkunaa:

1. Explorer, jossa näkyy valitussa kansiossa olevat tiedostot
2. Editori, jossa koodia editoidaan eli muokataan
3. Terminaali, jossa ohjelma käännetään ja suoritetaan

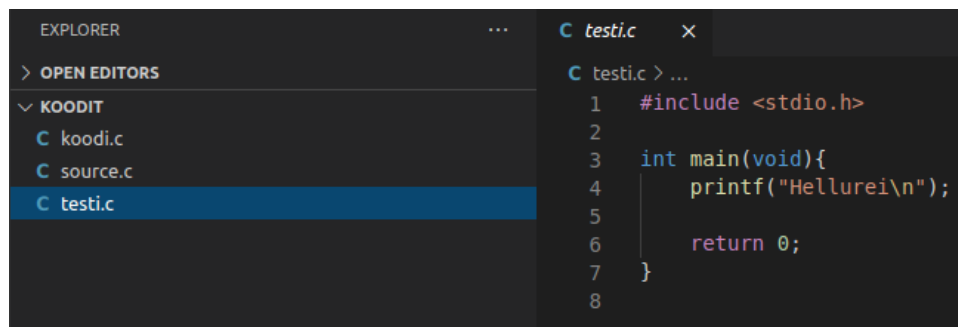
1.2. Explorer eli kansion kanssa työskentely

Kaikki yhteen projektiin kuuluvat tiedostot kannattaa tallentaa samaan kansioon, jolloin niitä on helppo käsitellä yhtä aikaa. Tämän oppaan kannalta kaikki oppaan esimerkit muodostavat yhden kansion. Oppaan lopussa ohjelmat muodostuvat useista tiedostoista ja tiedostojen kääntämiseen käytetään `make`'a, jolloin on luontevaa luoda jokaista isompaa ohjelmaa kohden oma projekti eli jokaista `Makefile`'ä kohti tehdään aina yksi kansio.

Kansion avaaminen

Ohjelmatiedostokansion voi avata VSC:n kautta tai tiedostonhallinnasta. Tiedostojen avaaminen tiedostonhallinnasta onnistuu, jos asennuksen yhteydessä valitsit "Open with Code"-vaihtoehdon. Luonnollisesti C-ohjelmatiedostot voi yhdistää myös VSC:hen jälkikäteen Windows'n asetuksista. Halutun kansion avaaminen onnistuu seuraavilla tavoilla:

- VSC
 - File → Open Folder → Valitse koodikansio → Ok
 - Valitset CTRL-K ja CTRL-O peräkkäin näppäimistöltä
- Tiedostonhallinta: Klikkaa hiiren oikeaa painiketta kansiossa -> Open with Code

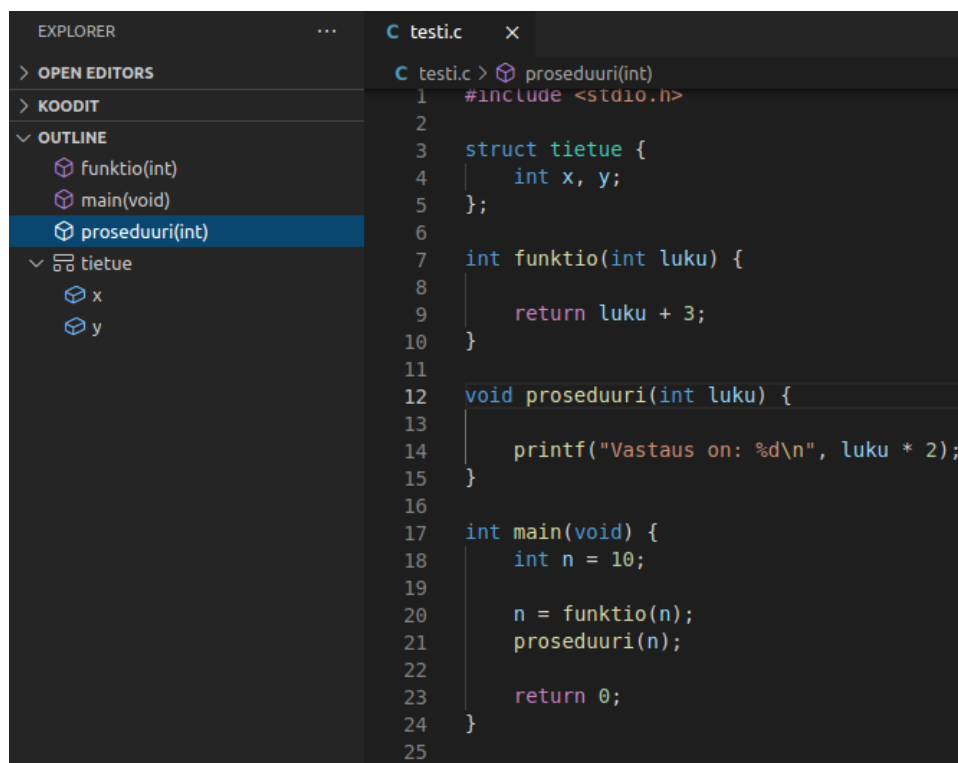


Kuva 1. Avattu kansionäkymä, kansiota “Koodit”.

Koodikansion auki pitäminen helpottaa työskentelyä useiden tiedostojen kanssa, kun kaikki kansiossa olevat tiedostot näkyvät yhtä aikaa vasemmalla. Lisäksi terminaaali avautuu aina suoraan koodikansioon, joten näin välttyy tarpeelta siirtä terminaalissa kansiota toiseen.

Koodin yleiskuva (Outline navigaatio)

Kooditiedostojen kasvaessa ja niiden sisältäessä useita eri aliohjelmia ja tietorakenteita, voi koodissa navigaatiota helpottaa VSC:n Outline-näkymällä. Outline-näkymän saa auki klikkaamalla editorin vasemmalla puolella olevaa Outline-kohtaa. Tästä näkymästä voi helposti nähdä avatun kooditiedoston rakenteet ja rakennetta klikkaamalla editoriin aukeaa koodin haluttu kohta kuten esimerkiksi kiinnostava aliohjelma tai tietue.



Kuva 2. Outline-näkymä tiedostosta testi.c.

1.3. Editori eli tiedostojen kanssa työskentely

Explorerissa näkyvät tiedostot ja ne saa auki editoriin klikkaamalla tiedostonimeä. Tämän jälkeen tiedostoa voi muokata editorissa. Koodieditorissa on tiettyjä erityispiirteitä, jotka tekevät editorista sopivan juuri ohjelmakoodin muokkaamiseen. Alkuun pääsee ihan

normaaleilla Windows-ohjelmien käskyillä kuten talleta (CTRL-S), kopioi (CTRL-C) jne. sekä käyttämällä näppäimistöä normaalisti. Kun koodaa enemmän, tulee vastaan tiettyä usein toistuvia asioita ja siksi niitä varten on olemassa omia pikanäppäimiä, jotka löytyvät esim. seuraavista lähteistä:

- Windows: <https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf>
- Linux: <https://code.visualstudio.com/shortcuts/keyboard-shortcuts-linux.pdf>
- macOS: <https://code.visualstudio.com/shortcuts/keyboard-shortcuts-macos.pdf>

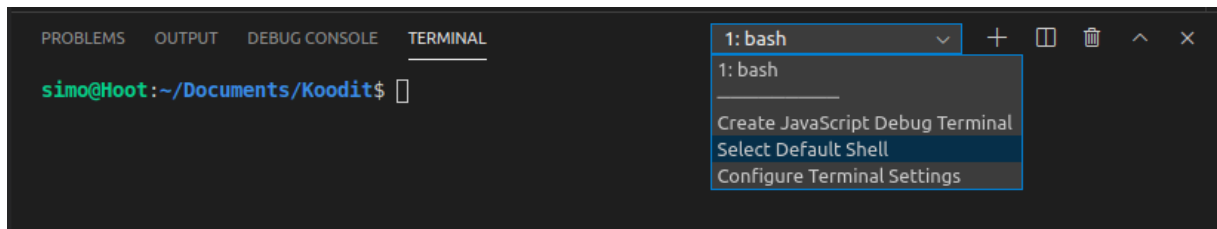
Yleisimpiä näppäinyhdistelmiä ovat seuraavat:

- Pikanäppäinasetukset. Paina Ctrl+K ja sen jälkeen Ctrl+S. Täältä voi etsiä tai muokata VSC-pikanäppäimiä.
- Extensions -valikko. Ctrl+Shift+X
- Format document. Koodin saa automaattisesti formatoitua “nättiin muotoon”
 - Windows: Shift+Alt+F
 - Linux: Ctrl+Shift+I
- Kaikki saman sanan esiintymiset koodissa voidaan muokata kerralla maalaamalla teksti ja painamalla F2
- Painamalla Alt ja nuolinäppäintä (ylös/alas) voidaan siirtää koodiriviä helposti riviltä toiselle
- Monelle eri riville voidaan kirjoittaa samanaikaisesti käyttämällä ns. Multikursoria. Voit asettaa kursoria painamalla Alt-painiketta ja klikkaamalla haluttua riviä
- Rivikommentin voi lisätä painamalla
 - Linux ja Windows oletus: Ctrl+Shift+/'
 - Windowsissa voi olla Ctrl + ` (heittomerkki)
- Painamalla oikeasta yläkulmasta laatikkoikonkia tai raahaamalla tiedoston nimen haluttuun kohtaan näkymään saa näkymän jaettua useaan osaan (tämän voi myös asettaa omaksi pikanäppäimeksi)
- Uuden tiedoston avaus. Ctrl+N
- Koodin sisentämisen lisääminen ja vähentäminen sarkain ja Shift-sarkain (sarkain = tabulaattori eli TAB).
- Käyttöliittymää voi suurentaa eli zoomata painamalla Ctrl+ + sekä pienentää painamalla Ctrl+ -
- Zoomauksen voi resetoida painamalla Ctrl + NumPad0
- Uuden terminaalin saa auki painamalla Ctrl+ Ö (tai englantilaisella näppäimistöllä Ctrl+')
- Kaiken koodin voi kopioida painamalla Ctrl + A ja painamalla sen jälkeen Ctrl + C kuten kaikissa Windows-sovelluksissa

Editorilla kirjoitetaan siis ohjelma eli editoidaan koodia. Kun ohjelma on tehty, C-ohjelma pitää kääntää ja suorittaa.

1.4. Terminaali eli ohjelman kääntäminen ja suorittaminen

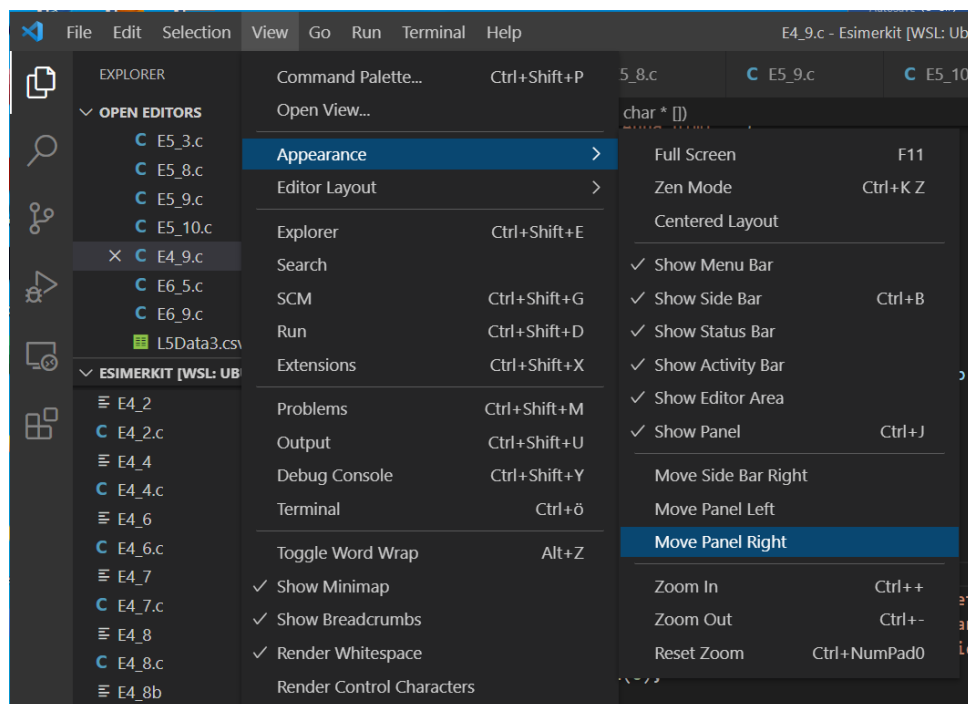
Jos editorin alareunassa ei näy terminaalia, voi sen avata ylävalikosta Terminal -> New Terminal. Terminaaliin avautuu järjestelmän oletuskomentotulkki, joka WSL-asennuksessa on bash eli Linux-komentorivi. Oletustulkin voi vaihtaa klikkaamalla terminaalissa olevaa pudotusvalikkoa ja valitsemalla “Select Default Shell” ja valitsemalla ylävalikon vaihtoehdoista mieluisen.



Kuva 3. Terminaali ja oletuskomentorivitulkki bash

Komentotulkki ei vaihdu vielä samaan terminaali-istuntoon, vaan terminaali pitää käynnistää uudelleen tai avata uusi terminaali. Käyttötarkoitukseemme riittää yksi terminaali, joten sulje ensin vanha painamalla terminaalin pudotusvalikon vieressä olevaa roskakori-ikonia (Kill Terminal) ja avaa ylävalikosta uusi terminaali.

Terminaali on normaali Linuxin terminaali eli siinä voi kääntää ja suorittaa ohjelmia sekä katsoa esim. manuaali-sivuja. Mikäli suoritettava ohjelma, tai manuaalisivu, on niin pitkä, ettei sen käyttö ole luontevaa editorin alapuolella, voi sen siirtää myös editorin viereen esim. oikealla (kuva 4) tai avata oman Ubuntu-Linux-terminaalin Windows'sta sitä varten.



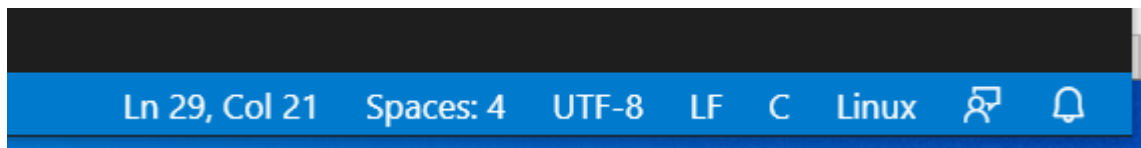
Kuva 4. VSC:n asetukset, terminaalin siirto editorin viereen oikealle puolelle (Move Panel Right)

1.5. Konfigurointi

Koodieditoria voi muokata paljon eli ohjelmoija voi tehdä työkalustaan itsensä näköisen. Alla on yksi tällainen konfigurointivinkki eli sarkain-näppäimen asetusten muuttaminen. Makefile'ssä on pakko käyttää sarkain-merkkiä/tabulaattoria, vaikka koodissa sisennys välilyönneillä on toimiva käytäntö.

Sarkain eli tabulaattori, \t

- VSC korvaa oletuksena sarkaimen 4 välilyönneillä.
- Asetuksen voi vaihtaa editorin alareunassa näkyvästä "Tab Size: 4" -kohdasta.



Kuva 5. VSC:n asetukset, joista voi vaihtaa mm. välilyönti- ja sarkainasetuksia

Liite 3. Keskeisten C-kirjastojen funktioita

| | |
|--|---|
| 1. C-kielen kirjastoja ja otsikkotiedostoja..... | 1 |
| 1.1. ctype.h..... | 1 |
| 1.2. float.h..... | 1 |
| 1.3. limits.h..... | 1 |
| 1.4. math.h..... | 2 |
| 1.5. stdio.h..... | 2 |
| 1.6. stdlib.h..... | 3 |
| 1.7. string.h..... | 3 |

1. C-kielen kirjastoja ja otsikkotiedostoja

C-kielessä käytetään paljon kirjastoja, jotka saadaan käyttöön sisällyttämällä otsikkotiedosto ohjelmaan. Tähän liitteeseen on koottu keskeisiä otsikkotiedostoja sekä niihin liittyviä kirjastojen funktioita ja vakioita.

Jotkut kirjastot pitää myös linkittää mukaan tehtävään ohjelmaan. Tämä onnistuu `-l` -optiolla sekä halutun kirjaston tunnisteella kuten `m` math-kirjaston kohdalla eli optiolla `-lm`

```
gcc testi.c -o testi -lm
```

1.1. 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

1.2. float.h

- **DBL_MIN.** Pienin positiivinen kaksoistarkkuuden luku
- **DBL_MAX.** Suurin kaksoistarkkuuden luku

1.3. limits.h

- **CHAR_MIN.** Merkkityypin pienin arvo
- **CHAR_MAX.** Merkkityypin suurin arvo
- **INT_MIN.** Pienin kokonaisluku
- **INT_MAX.** Suurin kokonaisluku

1.4. math.h

- **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

1.5. stdio.h

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

- **EOF**. (-1), tiedoston loppu
- **NULL**. (0), osoittimen arvo 0
- **stdin**. Standardi syöttötietovirta, normaalisti näppäimistö
- **stdout**. Standardi tulostustietovirta, normaalisti näyttö
- **stderr**. Standardi virhetietovirta, 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 merkkijonosta 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

1.6. 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).** Suorittaa käyttäjärjestelmällä x:n osoittaman komennon, esim. `system("rm tiedot.txt");`

1.7. 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 ja 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

ISBN 978-952-335-684-9

ISBN 978-952-335-685-6 (PDF)

ISSN-L 2243-3392

ISSN 2243-3392

Lappeenranta 2021

