

C-KIELI

JA

KÄYTÄNNÖN OHJELMOINTI

OSA 1



Lappeenrannan teknillinen yliopisto 2008

Jussi Kasurinen

ISBN 978-952-214-532-1 ISSN 1459-3092

Kannen kuva: Kevin Steele. Kuva julkaistu Creative Commons - Nimi mainittava-Ei kaupalliseen käyttöön - lisenssillä.

Tähän dokumenttiin sovelletaan "Creative Commons Nimi mainittava-Ei kaupalliseen käyttöön- Sama lisenssi 2.5" – lisenssiä. Opas on ei-kaupalliseen opetuskäyttöön suunnattu käsikirja.

Materiaali, esimerkit sekä taitto:

Jussi Kasurinen

Oikoluku, kommentointi:

Risto Westman  
Satu Alaoutinen

Lappeenrannan teknillinen yliopisto, Tietojenkäsittelytekniikan laboratorio.

Lappeenranta 14.1.2008

Tämä ohjelmointiopus 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 opas on neljäs Lappeenrannan teknillisen yliopiston tietojenkäsittelytekniikan laboratorion opas, ja se on tarkoitettu perehdyttämään lukija C-ohjelmointikielen perusteisiin. Oppaan tarkoituksena ei ole olla täydellinen käsikirja C-ohjelmointikielestä, vaan ensisijaisesti auttaa opiskelijoita siirtymään korkeamman tason ohjelmointikielistä kuten Pythonista laitteistoläheisempien ohjelmointikielten pariin.

Oppaassa olevat esimerkkikoodit on tehty ja testattu Windows XP-työasemalla käyttäen kääntäjänä DEV-C++-ohjelmointiympäristön versiota 4.9.9.2. Lisäksi oppaan alkupään esimerkeissä oletetaan, että käyttäjä on aiemmin perehtynyt Python-ohjelmointikieleen sekä osaa sen perusteet. Opasta on kuitenkin tarkoitus pystyä käyttämään myös itsenäisenä oppaana.

Mikäli ohjelmointi alana on sinulle uusi, niin on suositeltavampaa, että aloitat opiskelun aiemmin julkaistusta Python-ohjelmointioppaasta [1] (Tietojenkäsittelytekniikan käsikirjat 10, LTY 2007). Opas on ilmainen ja aloittaa matalammalta lähtötasolta ohjelmoinnin alkeista. Se on suositeltava alkeisopas ennen tämän kirjan aiheisiin tutustumista, sillä tämä opas menee nopeasti hyvin laitteistoläheiseen ja monessa mielessä vaikeampiin asioihin.

Oppaassa on joissain tapauksissa käytetty osia aiemmin kirjoitetusta Lyhyestä C-oppaasta [2], jonka on kirjoittanut Satu Alaoutinen (LTY). Kyseisestä oppaasta otetut kappaleet on merkitty tässä oppaassa otsikon vieressä marginaalissa olevalla **LCO**-merkillä. Kappaleita on saatettu muokata oppaan yhdenmukaisuuden parantamiseksi.

## SISÄLLYSLUETTELO

<b>VALMISTELUT</b> .....	<b>5</b>
<b>LUKU 1: C-SYNTAKSI, ENSIMMÄINEN OHJELMA</b> .....	<b>6</b>
C-SYNTAKSI LYHYESTI .....	6
SYÖTTEIDEN VASTAANOTTAMINEN .....	9
<b>LUKU 2: MUUTTUJAT, TIETOTYYPIT</b> .....	<b>11</b>
PERUSTEET .....	11
NUMEERISET MUUTTUJAT .....	12
MUUTTUJIEN OPERAATTORIT .....	14
OPERAATTORIEN SUORITUSJÄRJESTYS.....	16
MERKKIJONOT C-KIELESSÄ .....	17
<b>LUKU 3: VALINTARAKENTEET</b> .....	<b>21</b>
PERUSTEET .....	21
IF-ELSEIF-ELSE .....	21
SWITCH-CASE.....	24
GOTO-KÄSKYSTÄ .....	26
<b>LUKU 4: TOISTORAKENTEET, PERUSKIRJASTOISTA</b> .....	<b>27</b>
PERUSTEET .....	27
WHILE-RAKENNE .....	27
FOR-RAKENNE .....	28
DO-WHILE-RAKENNE .....	30
OHJAUSKÄSKYISTÄ CONTINUE, BREAK JA PASS .....	31
#include, VALMIIDEN FUNKTIOKIRJASTOJEN KÄYTTÄMINEN .....	33
#define, KIINTOARVOJEN MÄÄRÄÄMINEN .....	34
<b>LUKU 5: TIEDOSTOJEN LUKEMINEN JA KIRJOITTAMINEN</b> .....	<b>36</b>
PERUSTEET .....	36
TIEDOSTOJEN KÄSITTELYYN KÄYTETTÄVIÄ FUNKTIOITA .....	39
<b>LUKU 6: FUNKTIOT JA OSOITTIMET, KOMENTORIVIPARAMETRIT</b> .....	<b>41</b>
PERUSTEET .....	41
ALIFUNKTION TOTEUTTAMINEN .....	41
PARAMETRIEN VÄLITYS.....	43
NIMIAVARUUKSISTA.....	44
OSOITTIMET .....	46
KOMENTORIVIPARAMETRIEN KÄYTTÄMINEN .....	48
<b>LUKU 7: TIETUEET JA MUISTINKÄSITTELY</b> .....	<b>50</b>
TIETUEET.....	50
MUISTINVARAUS .....	52
<b>LOPPUSANAT</b> .....	<b>55</b>
HUOMIOITA .....	55
LISÄLUETTAVAA .....	55
<b>LÄHDELUETTELO</b> .....	<b>56</b>
<b>LIITE 1: FUNKTIOKIRJASTOT</b> .....	<b>57</b>

# Valmistelut

---

## Ennakovalmistelut C-ohjelmoinnin aloittamista varten

C-ohjelmointi aloittaaksesi joudut asentamaan tietokoneellesi DEV-C++-ohjelmointiympäristön, joka löytyy osoitteesta <http://www.bloodshed.net/devcpp.html>. Tämä ohjelmointiympäristö on ilmainen GPL-lisenssin alaisuudessa julkaistu työkalu, joka C-kielen lisäksi tukee C++-ohjelmointikieltä.

- Mikäli et ole varma kuinka ohjelma asennetaan, voit lukea asennus- ja virheenkorjausohjeet kurssin ”CT20A0210 Käytännön Ohjelmointi” verkkosivuilta.
- Lisäksi huomioi, että ohjelman käyttöönoton yhteydessä saatat joutua tekemään kääntäjään joitain asetuksia kytkeäksesi C-kääntäjän oikeaan työskentelytilaan. Myös nämä ohjeet löytyvät ko. sivuilta.
- Tämä opas siirtyy varsinaisessa asiasisällössään suoraan ohjelmoinnin käytännölliseen puoleen. Ohjeita siihen, miten DEV-C++-ohjelmointiympäristöä käytetään, kuinka käännettyä C-koodia ajetaan Windowsissa jne. löytyy nekin kurssin verkkosivuilta. Kuten sanottu, tässä oppaassa oletetaan, että ymmärrät työkalun peruskäytön eikä sen käyttämiseen erikseen kiinnitetä huomiota esimerkeissä.
- Linux-koneisiin työkalut asennetaan tavallisesti käyttöjärjestelmän mukana, joten ne löytyvät useimmista työasemista valmiina eikä sinun tarvitse itse asentaa mitään. Katso tarkemmat ohjeet käytettävistä työkaluista sekä niiden käyttöohjeista kurssisivuilta.

# Luku 1: C-syntaksi, ensimmäinen ohjelma

---

## C-syntaksi lyhyesti

C-ohjelmointikieli julkaistiin vuonna 1972 Yhdysvalloissa Unix-käyttöjärjestelmien ohjelmointikieleksi. Vaikka kieli alun perin suunniteltiin nimenomaisesti järjestelmäohjelmointiin, on se nyttemmin löytänyt paikkansa myös sovellusohjelmoinnin parissa. Erityisesti aloilla, joissa muistin tai suoritustehon määrä on rajoitettu, on C-kieli osoittanut olevansa vahva johtuen käännetyn ohjelman kompaktista koosta. C-kieli onkin hyvin voimakkaasti laitteistoläheinen ohjelmointikieli, ja sille tyypillistä onkin, että käyttäjä vastaa pitkälti monien uudempien ”korkeampitasoisten” kielten automatisoimista toiminnoista. Näitä toimintoja ovatkin mm. muistinvaraus, dynaamisten rakenteiden määrittely ja toteutus sekä omien toimintojen jälkeinen muistinhallinta.

Kuitenkaan perusohjelmointi C-kielen kanssa ei kovinkaan merkittävästi poikkea Python-ohjelmoinnista muuten kuin syntaksin tasolla. Jos otamme esimerkiksi tekstirivin tulostavan ohjelman, voimme nähdä siinä jotain selkeitä eroja:

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

Ensinnäkin, kaikki C-kielellä toteutettu toiminnallinen koodi tulee sijoittaa aina funktion sisälle, ja päätason ohjelmakoodi sijoittaa `main` -nimiseen funktioon. Lisäksi jokaisella funktiolla on oma tyyppinsä – tästä puhumme myöhemmin lisää mutta nyt riittää kun muistat 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 arvon `0 (return 0)`.

Jos taas tarkastelemme kielen merkkirakennetta, voimme huomata että C-kielessä on jotain ylimääräisiä rakennemerkkejä. Esimerkiksi jokainen koodiosio (funktion, toisto- tai ohjausrakenteen toiminnallinen osa), joka Pythonissa sijoitettaisi eri sisennystasolle, merkitään C-kielessä aaltosuluilla ”{” ja ”}”. Aukeavat aaltosulut ovat aina osion ensimmäinen merkki ja sulkeutuvat aaltosulut osion viimeinen merkki; sisennyksillä ei C-kielessä ole toiminnallista roolia, mutta ne helpottavat merkittävästi lähdekoodin ymmärtämistä. Juuri tämän vuoksi olisikin hyvä jatkaa Pythonista tuttua sisennysten käyttöä kaikesta huolimatta. Lisäksi jokainen toiminnallinen rivi päättyy puolipisteeseen ”;”.

## C-kielen perusmuistisäännöt lyhyesti

Ohessa vielä kerran lyhyesti C-kielen perussyntaksin erot Python-ohjelmointikieleen erillisenä listana:

- Kaikki koodi tulee sijoittaa funktion sisälle.
- Päätasen koodi tulee aina `main`-nimiseen funktioon, joka on tyyppiä `int`. Jokaisessa ohjelmassa on oltava yksi (ja vain yksi) `main`-niminen funktio.
- Funktiot, joilla on palautusarvo – myös `main` – loppuvat aina `return`-käskyyn, joka palauttaa funktion tyyppiin sopivan arvon. `int main`-funktio loppuu aina käskyyn `return 0`.
- Lähdekoodiosio alkaa aina aukeavalla aaltosululla `"{"` ja loppuu sulkeutuvaan aaltosulkuun `"}"`. Sisennys ei vaikuta lähdekoodin osiojakoon.
- Jokainen käskyrivi koodilohkossa päättyy puolipisteeseen `;"`.
- Käytännössä siis **jokainen koodirivi päättyy joko aaltosulkuun tai puolipisteeseen**. Jos rivi päättyy aukeavaan aaltosulkuun, aloita seuraava rivi yhtä sisennystasoa alemmaa, jos sulkeutuvaan aaltosulkuun, jatka yhtä sisennystasoa ylempänä.

Seuraavaksi tutustumme näihin sääntöihin käytännössä ohjelmakoodiesimerkin avulla. Mikäli et vielä ole tutustunut oppaan kanssa käytettävän DEV-C++-ohjelmointiympäristön käyttöohjeisiin, olisi tässä vaiheessa hyvä tehdä niin. Ohjelman käyttämiseen löydät apua kurssisivuilta, sekä ensimmäisen kurssiviikon harjoituksista.

### Esimerkki 1.1: Tekstintulostus ruudulle

#### Esimerkkikoodi

```
int main(void){  
  
    /*Tämä on kommenttimerkki*/  
  
    printf("Minä tein tämän!\n");  
    printf("Huomaa että teksti jatkuu aina edellisen ");  
    printf("perään,\n ellei tulostus pääty rivinvaihtomerkkiin!");  
  
    /*C-kielen kommenttimerkit  
       ovat luonnollisesti monirivisiä */  
  
    return 0;  
}
```

## Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa `X:\C-kurssi\esimerkkikoodit`>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>>1-1.exe
Minä tein tämän!
Huomaa että teksti jatkuu aina edellisen perään,
  ellei tulostus pääty rivinvaihtomerkkiin!
X:\C-kurssi\esimerkkikoodit>>
```

## Kuinka koodi toimii

Tässä vaiheessa on alkuun hyvä sanoa muutama sana C-kääntäjän käytöstä. Ensinnäkin, koska C on kääntäjäpohjainen ohjelmointikieli, tarkoittaa se sitä, että virheettömästä lähdekoodista tehdään itse asiassa itsenäisesti ajettavissa oleva tiedosto. Tämä on merkittävä ero esimerkiksi Python-ohjelmointikieleen, jossa Python-lähdekoodi piti erikseen paketoita esimerkiksi py2exe-moduulilla, jotta lähdekoodia pystyttäisi käyttämään ilman ajoympäristöä. C-kielessä taas kääntäjä tekee lähdekoodista sellaisen ohjelmapaketin, jota itse käyttöjärjestelmä – tässä tapauksessa siis Windows XP – ymmärtää ilman muiden ohjelmien apua. Tähän liittyy kuitenkin muutama ongelma mutta niistä puhumme hetken päästä tarkemmin.

Mitä ohjelma siis itse asiassa tekee? Jos katsomme pääfunktion määrittelevää käskyä `int main(void)`, niin näemme että siihen on ilmestynyt sana `void` paikalle, jossa normaalisti olisivat funktion vastaanottamat parametrit. `void` on C-kielen varattu sana ja tarkoittaa tyhjää, olematonta, ”ei-olemassaolevaa”. Kerromme siis itse asiassa, että `main`-funktio ei vastaanota mitään parametreja. Tämän jälkeen avaaamme kaarisululla pääfunktion koodiosion.

Kun tarkastelemme pääfunktioita, niin huomaamme että funktiossa on ensimmäisenä kenoviivan ja tähtimerkin erottama kommenttimerkki. C-kielessä tällä tavoin merkitään lähdekoodin seassa olevia kommenttimerkkejä, eli laittamalla teksti merkkiparien `”/*”` ja `”*/”` sisään. Kommenttimerkit C-kielessä ovat aina oletuksena monirivisiä, joten kääntäjä odottaa aina saavansa merkkiparille `”/*”` parin `”*/”`. Avoimeksi jätetyn kommenttirivin jälkeisen koodin kääntäjä katsoo edelleen kommentteiksi niin kauan kunnes vastaan tulee lähdekooditiedoston loppu tai sulkeva kommenttimerkki.

Seuraavaksi koodissa on kolme tulostuskäskyä, jotka toiminnaltaan vastaavat Pythonin `print`-käskyä. Huomioi kuitenkin, että C-kielen `printf`-käsky ottaa tulostettavan merkkirivin parametrina, joten merkkijono on funktiokutsujen tapaan suljettava sulkumerkkien sisään. Myöskään uudelle riville aloittaminen ei enää tapahdu automaattisesti: jos tulostettava teksti ei pääty rivinvaihtomerkkiin, jatkuu seuraava tulostus aina edellisen rivin perään. Lisäksi muista, että koska `print`-komennot eivät aloita uutta koodiosiota, on ne lopetettava puolipisteellä.

Tämän jälkeen koodissa on toinen kommenttiteksti, sekä funktion lopetuksen osoittava `return`-käsky. Tämä käsky käytännössä määrää, että `main`-funktio lopettaa toimintansa lopetusarvolla 0.

Tämä käytännössä tarkoittaa, että käyttöjärjestelmälle kerrotaan lopetuksen tapahtuneen hallitusti ja odotusten mukaisesti. Lopuksi vielä suljemme auki olleen `main`-funktion koodiosion kiinnimenevällä kaarisulkeella.

## Huomioita kääntäjistä ja käyttöjärjestelmistä

Aiemmin mainitsimme, että kääntäjien ja käyttöjärjestelmien kanssa on joitain huomionarvoisia seikkoja. Tällä viittasimme siihen, että esimerkiksi tämän oppaan tapauksessa DEV-C++-kääntäjä tekee meille automaattisesti ohjelmia, jotka toimivat suoraan Windows-käyttöjärjestelmässä ilman erillisiä ajoympäristöjä. Koska DEV-C++ on nimenomaan Windows-ympäristöön ohjelmia luova kääntäjä, ei sen tuottama ohjelma luonnollisesti toimi esimerkiksi Linux-ympäristöissä. Toisaalta taas, Linux-ympäristön kääntäjällä (vaikkapa `cc`) käännetty ohjelma ei toimi Windows-puolella.

Ongelma eri kääntäjien välillä on siinä, että ne saattavat tulkita samaa koodia hieman eri tavoin tehdessään suoraan käyttöjärjestelmän päällä toimivaa ohjelmaa. Tämän lisäksi eri käyttöjärjestelmillä saattaa olla erilainen politiikka esimerkiksi muistinhallinnan tai muiden resurssien käytön suhteen.

Usein nämä ongelmat tulevat huomionarvoisiksi vasta myöhemmin eivätkä oikeastaan vaikuta mitenkään perusrakenteilla ohjelmointiin. Siitäkin huolimatta kannattaa muistaa, että C-kieltä ei laitteistoläheisyytensä takia pystytä välttämättä kääntämään suoraan sellaisenaan jos se siirretään toiseen käyttöjärjestelmään kuin missä se on tehty. Lisäksi eri kääntäjät saattavat tulkita asioita eri tavoin, jolloin yhdellä kääntäjällä toimiva lähdekoodi saattaa vaatia muuntelua toisella. Tämän vuoksi onkin hyvä huomata, että mikäli työskentelet jollain muulla kääntäjällä kuin oppaassa mainittulla DEV-C++ version 4.9.9.2 -kääntäjällä, voi jotkin yksityiskohdat (muuttujien ylä- ja alarajat, varoitus- ja huomatuspolitiikka) hieman poiketa toisistaan.

## Syötteiden vastaanottaminen

Jotta pääsemme varsinaisesti tarkastelemaan Pythonin ja C-kielen eroja, otamme vielä ensimmäisen luvun loppuun toisen esimerkin perusohjelmoinnista. Teemme nyt ohjelman, joka kysyy käyttäjältä lukua, tallentaa luvun muuttuunaan ja tulostaa annetun luvun käyttäjälle.

### Esimerkki 1.2: Syötteen ottaminen käyttäjältä

#### Esimerkkikoodi

```
#include <stdio.h>

int main(void){

    int luku;

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

    printf("Annoit luvun %d.", luku);
    return 0;
}
```

## Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa `X:\C-kurssi\esimerkkikoodit`, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>1-2
Anna kokonaisluku: 34
Annoit luvun 34.
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

Tällä kertaa ohjelmaesimerkkimme alkaa myös Python-ohjelmoinnista tutulla kirjastomodulin sisällyttämisellä. Sisällytämme koodiin `stdio.h` -kirjaston, joka sisältää tavallisimpia tietojen vastaanottamiseen ja tulostamiseen käytettyjä funktioita. C-kielessä onkin tavallista, että tarvitsemme käyttöömmme jo pelkkiä ”perustoimintoja” varten moduulikirjastoja. Kirjastomodulien käytöstä puhumme lisää myöhemmin luvussa 4. Toistaiseksi riittää kun ymmärrät, että käskyllä `#include <kirjastonnimi>` voimme käyttöönottaa erilaisia lisäkirjastoja, ja että otat kirjaston `stdio.h` käyttöön aina esimerkkien testaamisen yhteydessä.

Tämän jälkeen aloitamme `main`-funktion koodiosion, jossa ensitöiksemme määrittelemme muuttujan `luku`. C-kielessä muuttujat esitellään aina funktion alussa, toisin kuin Pythonissa jossa muuttujia voidaan käyttöönottaa sitä mukaa kun niitä tarvitaan. Myös tästä puhumme lisää heti seuraavassa luvussa.

Seuraavana meillä onkin koodin varsinainen toiminnallisuus: syötteen pyytäminen ja tallentaminen. Aiemmin Python-ohjelmointikielessä tämä pystyttiin toteuttamaan helposti `input`- ja `raw_input`-funktioilla, mutta nyt joudumme käyttämään saman toteuttamiseen hieman enemmän vaivaa.

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

Ensin annamme käyttäjälle toimintaohjeet `printf`-lausekkeella, joka tulostaa ruudulle ohjeet antaa kokonaisluku. Koska C-kielessä `print`-käskyjä ei automaattisesti rivitetä eri riveille, pysyy kursori samalla rivillä ohjeiden kanssa muodostaen syötekentän. Seuraavalla rivillä olevalla komennolla `scanf` luemme tähän kohtaan käyttäjän antaman syötteen.

Huomionarvoista `scanf`-käskyssä on se, miten sitä käytetään: Ensinnäkin `scanf`-käsky toimii kuten funktio. Me ensin kerromme minkälaista tietoa otamme vastaan määrittelemällä, että käyttäjältä odotetaan yhtä kokonaislukua (`%d`). Tämän jälkeen kerromme, että haluamme sijoittaa kyseisen käyttäjältä luetun arvon muuttujaan `luku`. Huomaa, että tässä kohdassa joudumme antamaan muuttujan nimen lisäksi etuliitteen `&`, joka kertoo että haluamme sijoittaa luetun arvon muuttujan uudeksi arvoksi.

Lopuksi päätämme funktion palauttamalla arvon 0, ja viimeistelemme ohjelmakoodin sulkemalla `main`-funktion koodiosion.

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

## Luku 2: Muuttujat, tietotyypit

---

### Perusteet

C-kielen ja Pythonin keskeisin ero lähtökohtaisesti on siinä, että mikäli Pythonissa jotain asiaa ei ole automatisoitu työskentelyn helpottamiseksi, on se poikkeuksellista. Vastaavasti mikäli C-kielessä jokin asia on automatisoitu siten että sisäisiä tapahtumia ei tarvitse tai edes pysty muuttamaan, on se yhtä poikkeuksellista. Kuitenkin näillä kielillä, kuten kaikilla ohjelmointikielillä yleisesti, on joitain yhteisiä piirteitä kuten esimerkiksi muuttujat ja muuttujien tietotyypit.

Python-ohjelmointikielessä muuttujilla oli käytännössä kolme erilaista tyyppiä, numeroarvo (int/float), merkkijono (str) taikka rakenne kuten lista tai tuple. Lisäksi muuttujan tyyppiä pystyttiin vaihtamaan lennosta sijoittamalla merkkijono numeroarvoja sisältäneeseen muuttujaan tai toisinpäin. C-kielessä muuttujien käyttö ei varsinaisesti poikkea paljoakaan Pythonista, mutta muuttujat pitää määritellä funktion alussa, sekä samassa yhteydessä valita tietotyyppi jota muuttujassa käytetään. Tätä tarkastellaan seuraavassa esimerkissä.

### Esimerkki 2.1: Muuttujan määrittely

#### Esimerkkikoodi

```
#include <stdio.h>

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

    int luku;
    float desimaaliluku;

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

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

    return 0;
}
```

## Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>2-1
Luku-muuttuja on 5.
5 8.234235
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

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

Kuten huomaat, ensimmäinen asia, mitä main-funktiossa tehdään, on muuttujien esittely. Jokaiselle muuttujalle, mitä aiomme esittelynjälkeisessä koodissa käyttää, tulee tässä kohtaa kertoa nimi ja tyyppi. Riveillä `int luku;` ja `float desimaaliluku;` kerromme C-kääntäjälle, että aiomme käyttää kahta muuttujaa nimiltään `luku` ja `desimaaliluku`. Muuttuja `luku` on tyyppiä `int`, eli kokonaisluku ja `desimaaliluku` tyyppiä `float`, eli liukuluku.

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

## Numeeriset muuttujat

Huomasit varmaan edellisessä esimerkissä, että kokonaislukua `int` ja liukulukua `float` käsiteltiin siinä mielessä eritavoin, että jopa niiden tulostuskäskyn sijoitusmerkit (`%d` ja `%f`) olivat erilaisia. Tähän liittyy nimenomaan se C-kielen tilanne, että muistinhallinta ei varsinaisesti ole automatisoitu kuin hyvin alhaisella tasolla. Koska erilaiset numeromuuttujat tarvitsevat muistiin sijoittamista varten erilaisen määrän muistia, käsitellään niitä oikeasti erilaisina muuttujatyyppeinä. Tämän vuoksi erityyppisten numeeristen muuttujien välisiä laskutoimituksia tulisi välttää, vaikka osalla tyypeistä on jonkin verran yhteensopivuutta keskenään.

Koska numeeriset muuttujat eivät ole keskenään täysin yhteensopivia, on C-kielessä ongelma pyritty ratkaisemaan lisäämällä numeeristen muuttujien tyyppiä jonkin verran. Koska Pythonissa dynaamisuus huolehti erilaisten numeroarvojen välisestä yhteensopivuudesta, ei käytännössä tarvittu kuin kaksi numeromuuttujaa, kokonaisluku ja desimaaliluku. Lisäksi, koska muistinkäyttö oli automatisoitua, ei numeerisilla muuttujilla ollut varsinaista ylä- eikä alarajaa. C-kielessä myös

## C-kieli ja käytännön ohjelmointi

### LTY

numeeristen muuttujien koko tulee ottaa siinä mielessä huomioon, että alitettaessa alaraja tai ylitettäessä yläraja päädytään aina ns. ylivuoto-tilanteeseen, jossa ohjelma ei enää toimi oikein. Usein termeillä yli- ja alivuoto vielä erotellaan mentiinkö alarajan ali vai ylärajan yli.

Muuttujan nimi	tyyppinimi	koko (tavua)	alaraja	yläraja
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 1: C-kielen kokonaislukutyypien minimikoot**

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

Muuttujan nimi	tyyppinimi	koko (tavua)
Liukuluku	float	32
Kaksinkertaisen tarkkuuden liukuluku	double	64

**Taulukko 2: C-kielen desimaaliluvut.**

Liukulukuja käsiteltäessä tulee pitää mielessä, että ohjelmointikielien käsittelevät desimaalilukuja bittiarvoina. Tästä seuraa se, että desimaaliluvut esitetään approksimaatioina. Erityisesti päättymättömien desimaalilukujen kohdalla joudutaan käyttämään parhaalle tarkkuudelle tehtyjä pyöristyksiä johtuen siitä, ettei binäärilukujärjestelmä tue tällaisia lukuarvoja. Tästä ei kuitenkaan kannata peruslaskutoimitusten yhteydessä olla huolissaan: esimerkiksi normaalin liukuluvun float:in tarkkuus riittää kuvaamaan  $2.3 \cdot 10^{-9}$  -kokoluokan (0.0000000023) eroja. Ja mikäli tämä

ei riitä, voidaan tarkkuus kaksinkertaistaa `double`-liukulukutyypillä.

## Muuttujien operaattorit

Tietenkin numeeristen muuttujien käyttäminen olisi merkityksetöntä, mikäli niitä ei voitaisi käyttää laskutoimituksissa. Myös C-kielessä meillä on käytössä joukko laskentaoperaattoreita, loogisia operaattoreita sekä muita tyyppisiä, kuten bittiopeattoreita. Monet näistä ovat jopa samoja, joita olemme aikaisemmin käyttäneet Python-ohjelmoinnissa. Huomaa kuitenkin, että C-kieli ei käytä avainsanoja `True` ja `False`, vaan niiden numeerisia esitysmuotoja 0 (`False`) ja 1 (`True`) väittämien yhteydessä.

**Taulukko 2.1** Laskentaoperaattorit

Operaattori	Nimi	Selite	Esimerkki
=	Sijoitus	Sijoittaa annetun arvon kohdemuuttujalle	<code>luku = 5</code> sijoittaa muuttujalle <code>luku</code> arvon 5. Operaattori toimii ainoastaan mikäli kohteena on muuttuja.
+	Plus	Laskee yhteen kaksi operandia	<code>3 + 5</code> antaa arvon 8.
[muuttuja]++	Lisäys	Lisää muuttujan arvoa yhdellä.	jos <code>luku = 5</code> ; niin <code>luku++</code> ; tuottaa muuttujalle <code>luku</code> arvon 6.
-	Miinus	Palauttaa joko negatiivisen arvon tai vähentää kaksi operandia toisistaan	-5 tarkoittaa negatiivista numeroarvoa. <code>50 - 24</code> antaa arvon 26.
[muuttuja]--	Vähennys	Vähentää muuttujan arvoa yhdellä.	jos <code>luku = 5</code> ; niin <code>luku--</code> ; tuottaa muuttujalle <code>luku</code> arvon 4.
*	Tulo	Palauttaa kahden operandin tulon tai toistaa merkkijonon operandin kertaa	<code>2 * 3</code> antaa arvon 6.
/	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!
%	Jakojäännös	Palauttaa x:n jakojäännöksen y:stä.	<code>8%3</code> antaa 2. <code>-25.5%2.25</code> antaa 1.5.

**Taulukko 2.2** Loogiset- eli vertailuoperaattorit

Operaattori	Nimi	Selite	Esimerkki
<	Pienempi kuin	Palauttaa tiedon siitä onko x vähemmän kuin y. Vertailu palauttaa arvon 0 tai 1.	<code>5 &lt; 3</code> palauttaa arvon 0 ja <code>3 &lt; 5</code> palauttaa arvon 1.
>	Suurempi kuin	Palauttaa tiedon onko x enemmän kuin y.	<code>5 &gt; 3</code> palauttaa arvon 1.
<=	Vähemmän, tai yhtä suuri	Palauttaa tiedon onko x pienempi tai yhtä suuri kuin y.	<code>x = 3; y = 6; x &lt;= y</code> palauttaa arvon 1.

C-kieli ja käytännön ohjelmointi  
LTY

>=	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 onko operandit yhtä suuria.	x = 2; y = 2; x == y palauttaa arvon 1.
!=	Erisuuri kuin	Testaa onko operandit erisuuria.	x = 2; y = 3; x != y palauttaa arvon 1.

### Taulukko 2.3 Boolean-operaattorit

Operaattori	Nimi	Selite	Esimerkki
!	Boolean NOT	Jos x on 1, palautuu arvo 0. Jos x on 0, palautuu arvo 1.	x = 1; !x palauttaa arvon 0.
&&	Boolean AND	x and y palauttaa arvon 0 jos x on 0, muulloin se palauttaa y:n arvon	x = 0; y = 1; x && y palauttaa arvon 0 koska x on 0.
	Boolean OR	Jos x on 1, palautuu arvo 1, Muussa tapauksessa se palauttaa y:n arvon.	x = 1; y = 0; x    y palauttaa arvon 1. Merkki   saadaan näppäimillä AltGr - ”<”.

### Taulukko 2.4 Bittioperaattorit

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

## Operaattorien suoritusjärjestys

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

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

Taulukko 3: C-kielen operandion suoritusjärjestys.

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

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

Käytännössä kieli toimii loogisesti samalla tavalla lasku- ja logiikkaoperaattorien kanssa kuin Python, joten tässä asiassa ei tulisi olla paljoakaan uutta opeteltavaa.

## Merkkijonot C-kielessä

Huomasit varmaan, että aikaisemmassa osiossa puhuimme ainoastaan numeerisista muuttujista sivuten ohimennen yksittäisten merkkien tapausta. Kuinka sitten merkkijonoja käsitellään C-kielessä? Tässä kohtaa törmäämme ensimmäiseen merkittävään eroon C-kielen ja Pythonin välillä. Koska merkkijono on luonteeltaan dynaaminen rakenne johtuen siitä, että sen pituus voi olla mitä tahansa, emme pystykään C-kielessä käyttämään niitä aivan yhtä suoraviivaisesti.

Periaatteessa meillä on muutamia vaihtoehtoja merkkijonon käsittelyyn. Ensinnäkin, voimme luoda yksittäisistä merkki-muuttujista määrätynmittaisen taulukon ja tallentaa tiedon siihen. Tämä on muuten näppärä idea, mutta jos teemme liian lyhyen merkkitaulukon, ei käyttäjän syöte mahdu kokonaan talteen ja jos taas teemme taulukosta liian suuren, tuhlaamme muistia tyhjän tilan tallentamiseen. Toinen vaihtoehto onkin varata käsin muistia riittävästi aina kun haluamme tallentaa merkkijonon. Koska kuitenkin muistinhallinta on työlästä ja tarkkuutta vaativaa työtä, puhumme siitä vasta myöhemmin ja tässä vaiheessa tutustumme merkkijonoihin staattisten taulukon avulla.

### Esimerkki 2.2: Merkkijonon määrittely ja käyttäminen

#### Esimerkkikoodi

```
#include <stdio.h>

int main(void)
{
    /*määritellään muutama merkkijono ja apumuuttujia*/

    char nimi[20];
    char ammatti[] = "Palomies";
    char harrastus[20] = "autot";
    int koko;

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

    koko = sizeof(harrastus);

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

## Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa `X:\C-kurssi\esimerkkikoodit`, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>2-2
Anna nimi (max. 20 merkkiä): Erkki
Erkki on Palomies.
Harrastuksenaan hänellä on autot.
Sanassa autot on 20 merkkiä.
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

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

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

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

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

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

Nyt taas määrittelemme merkkijonotaulukon `ammatti`, johon on varattu tilaa juuri riittävästi että merkkijono "Palomies" mahtuu taulukkoon.

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

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

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

syöte saa suurimmillaan olla. Yleisesti ohjelman onkin parempi kestää liian suuren syöteen antaminen kaatumatta, kuin vaatia käyttäjältä erityistä tarkkuutta.

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

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

Taulukko 4: merkkitaulun `harrastus` sisältö

Merkkitaulussa on myös toinen huomioita asia nimeltään terminaattorimerkki `\0` (*null character, null terminator*). Tämä merkki käytännössä kertoo ohjelmassa missä kohdassa varsinainen sisältö loppuu ja tyhjä tila alkaa. Koska luomme staattisen merkkitaulukon aina olemassa olevasta muistista, voi taulukkoon jäädä jotain arvoja aiemmin sitä muistipaikkaa käyttäneiltä ohjelmilta. Jos emme alusta merkkitaulukkoa tyhjäksi (tai kääntäjän luoma ohjelma ei tee sitä meidän puolestamme), on merkkijonojen loppua tarpeellista merkata terminaattorimerkillä. Lisäksi, koska merkki on uniikki niin voimme myöhemmin käyttää sitä vaikka taulukon lukemisessa lopetusehtona. Useimmiten merkkijonoja taulukkoon syötettäessä terminaattorimerkki lisätään taulukkoon automaattisesti, ja merkin muistaminen pakottaa toimenpiteisiin usein vasta kun merkkijonoja läpikäydään tai muunnellaan merkki kerrallaan:

## Esimerkki 2.4: Null-merkin poistamisen vaikutus merkkijonoon

### Esimerkkikoodi

```
#include <stdio.h>

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

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

    return 0;
}
```

## Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>2-4  
testi %äH$=
```

```
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

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

Merkkijonoille on myös olemassa joitain perusmetodeja, joista kuitenkin puhumme enemmän joissain myöhemmissä esimerkeissä sekä liitteessä 1.

## Luku 3: Valintarakenteet

---

### Perusteet

Python-ohjelmointikielessä valintarakenteiden valitseminen oli suoraviivaista koska kieli ei varsinaisesti tukenut kuin if-elif-else-rakenteita. Vaikka kielellisesti tämä ratkaisu olikin hyvin yksinkertainen mutta kattava, ei C-kielessä valintarakenteissa ole tyydytty näin tasapäistävään yleistapaukseen.

C-kieli tukee oletuksena kolmea erilaista valintarakennetta, jotka ovat Pythonista tuttu If-elseif-else, suoraan valintaan perustuva switch-case sekä ehdoton hyppykäsky goto. Tämän luvun ensimmäisessä osassa tutustumme näihin rakenteisiin yksinkertaisten esimerkkien avulla.

### If-elseif-else

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

### Esimerkki 3.1: If-else if-else-rakenne

#### Esimerkkikoodi

```
#include <stdio.h>

int main(void)
{
    int luku;
    int testi_1 = 100;
    int testi_2 = 1000;

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

    if (luku < testi_1){
        printf("Antamasi luku on pienempi kuin 100.\n");
    }

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

    else {
        printf("Antamasi luku oli suurempi kuin 1000.\n");
    }

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

#### Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>3-1
Anna kokonaisluku: 25
Antamasi luku on pienempi kuin 100.
Lopetetaan.
```

```
X:\C-kurssi\esimerkkikoodit>3-1
Anna kokonaisluku: 250
Antamasi luku on 100 ja 1000 välillä.
Lopetetaan.
```

```
X:\C-kurssi\esimerkkikoodit>3-1
Anna kokonaisluku: 1001
Antamasi luku oli suurempi kuin 1000.
Lopetetaan.
```

## Kuinka koodi toimii

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

Kuten esimerkistä näemme, on mekanismi hyvin samankaltainen Python-ohjelmointikielen vastaavan esimerkin kanssa. Jokainen `if`-, `else if`- sekä `else`-rakenne avaa uuden koodiosion, jota merkitään C-kielessä aaltosuluilla. Lisäksi koodiosion valintaa koskevat ehdot sijoitetaan aina vähintään yksien normaalien kaarisulkeiden `"( )"` sisäpuolelle. Huomaa myös, että tällä kertaa sisennys ei ole merkitsevä tekijä, mutta kuten seuraava esimerkki osoittaa, on se edelleen hyvin hyödyllinen apuväline.

## Esimerkki 3.2: Sisennyksien käytöstä

### Esimerkkikoodi

```
#include <stdio.h>

int main(void){

    char sana[] = "kolmipyörä";

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

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

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

### Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa `X:\C-kurssi\esimerkkikoodit`, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>3-2
Sana voisi olla kolmipyörä.
Lopetetaan.

X:\C-kurssi\esimerkkikoodit>
```

## Huomioita koodista

Esimerkissä voimme havainnollistaa miksi sisennys ja koodin huolellinen muotoilu on merkityksellistä. Koska jokainen alkava koodiosio avaa uuden aaltosulun, emme enää useamman alkaneen sisennyksen jälkeen silmämääräisesti erota onko sulkeita tarvittava määrä. Jos meillä on väärässä paikassa sulkeutuvat aaltosulut – tai unohtamme sijoittaa koodiin sellaiset – ei ohjelmamme enää käänny tai vaikka kääntyykin, toimii se väärin. Yksittäisen sulkumerkin paikan etsiminen pitkistä ja huonosti muotoillusta koodista on turhin mahdollinen tapa hukata aikaa ja nähdä vaivaa, kun koko ongelmaa ei synny jos koodin kanssa noudattaa hyvää ohjelmointityyliä.

## Switch-case

Toinen C-kielen tyypillinen tapa suorittaa valinta on käyttää switch-case-rakennetta. Tätä rakennetta on helpointa kuvata valikkona, johon ohjelmoidaan kaikki valintamuuttujan vaihtoehdot joista sitten valitaan oikea etenemistapa. Switch-case on jossain mielessä ehkä redundanttia verrattaessa if-else-rakenteeseen, mutta käyttötarkoituksessaan se voi olla hyvinkin näppärä työkalu:

### Esimerkki 3.3: Switch-case-rakenne

#### Esimerkkikoodi

```
#include <stdio.h>

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

    switch (valinta){

        case 1:
            printf("Valitsit 1.\n");
            break;

        case 2:
            printf("Valitsit 2.\n");
            break;

        case 3:
            printf("Valitsit 3.\n");
            break;

        default:
            printf("En ymmärtänyt valintaa.\n");
            break;
    }

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

## Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>3-3
Tee valinta (1-3): 1
Valitsit 1.
Lopetetaan.
```

```
X:\C-kurssi\esimerkkikoodit>3-3
Tee valinta (1-3): 2
Valitsit 2.
Lopetetaan.
```

```
X:\C-kurssi\esimerkkikoodit>3-3
Tee valinta (1-3): 5
En ymmärtänyt valintaa.
Lopetetaan.
```

```
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

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

`Case`-valinta muodostaa teoriassa loogisen osion, mutta sen kanssa tulee muistaa kaksi asiaa: Ensinnäkin, kyseessä on yksi harvoista C-kielen lausekkeista, joka ei pääty puolipisteeseen tai aaltosulkuun vaan kaksoispisteeseen, sekä se, että `case`-osio päättyy aina `break`-käskyyn. `break`-käsky taas toimii C-kielessä samalla tavoin kuin Pythonin vastaava käsky, eli lopettaa käynnissä olevan rakenteen ja siirtyy rakennetta seuraavaan loogiseen lausekkeeseen.

Mikäli `break`-käsky jätetään laittamatta, suorittaa `switch`-rakenne kaikki valittua `case`-valintaa seuraavat `case`-osiot, eikä koodi tällöin toimi oikein. Lisäksi `switch-case` tukee `default`-valintaa, joka suoritetaan, mikäli yksikään määritelty `case`-valinta ei pidä paikkaansa. `default`-valinnan paikka on tavallisesti viimeisenä, ja hyvän ohjelmointitavan mukaisesti sekin tulisi päättää `break`-käskyyn. Kannattaa kuitenkin muistaa, että `default` voi myös esiintyä muissakin väleissä, kuten esimerkiksi rakenteen alussa. Tällöin myös `default`-osio vaatii ehdottomasti `break`-käskyn, jotta rakenne toimii oikein.

## Goto-käskystä

C-kielessä on myös olemassa kolmas ohjausrakenne nimeltään `goto`. Tämä käsky toimii siten, että ohjelmakoodissa asetetaan nimettyjä lohkoja, joihin koodi pakotetaan hyppäämään `goto lohkonimi`-käskyllä. Koska ratkaisu johtaa yhdeksässä tapauksessa kymmenestä täysin hallitsemattomaan koodisekamelskaan sekä on yleisesti pidetty huonona ohjelmointityylinä, ei sitä opeteta tässä oppaassa.

Jos kuitenkin jostain syystä olet ehdottoman varma että haluat tietää käskystä lisää tai joudut ylläpitämään koodia, joka sisältää kyseisiä rakenteita, on käskystä olemassa käyttöohjeet esimerkiksi lähdemateriaalissa. [2,3]

# Luku 4: Toistorakenteet, peruskirjastoista

---

## Perusteet

Myös toistorakenteissa C-kieli tarjoaa enemmän vaihtoehtoja kuin Python. Pythonista tuttujen alkuehtoisen for- ja avoimen while-rakenteen lisäksi kielestä löytyy myös lopetusehtoinen do-while. Lisäksi kieli tukee samoja toisto- ja ohjausrakenteiden ohjauskäskyjä kuten continue tai break. Tässä luvussa käymmekin lyhyesti läpi kuinka C-kielen toistorakenteet toimivat.

## While-rakenne

While-toistorakenteen idea C-kielessä on säilynyt samana kuin aiemmin. While-toistossa kierrosmäärää ei etukäteen tarvitse määritellä, ja käyttäjä saakin itse halutessaan vastata toistorakenteen lopettamisesta ajallaan. Käytännössä while-rakenne on parhaimmillaan, kun suoritetaan toistoa jonka kierrosmäärää ei etukäteen voida laskea.

### Esimerkki 4.1: While-rakenne käytössä

#### Esimerkkikoodi

```
#include <stdio.h>

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

    while (kierrosmaara > kierros){

        printf("Olemme kierroksella %d!\n",kierros);

        /*kasvatetaan kierroslukumittaria */
        kierros++;
    }
    printf("Lopetetaan tähän.\n");
    return 0;
}
```

## Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa `X:\C-kurssi\esimerkkikoodit`>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>4-1
Olemme kierroksella 0!
Olemme kierroksella 1!
Olemme kierroksella 2!
Olemme kierroksella 3!
Olemme kierroksella 4!
Lopetetaan tähän.
```

```
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

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

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

## For-rakenne

`For`-lauseen käyttäminen poikkeaa C-kielessä jonkin verran Pythonin vastaavasta rakenteesta, vaikka sen käyttöperiaate onkin aivan sama. Käytännössä `for`-rakenne toimii näin:

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

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

## Esimerkki 4.2: For-rakenne käytössä

### Esimerkkikoodi

```
#include <stdio.h>

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

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

        printf("%d ",i);

    }

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

### Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>4-2
For-rakenne laskee itse kierroksensa:
0 1 2 3 4 5 6 7 8 9 10
Lopetetaan tähän.
```

```
X:\C-kurssi\esimerkkikoodit>
```

### Kuinka koodi toimii

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

## Do-While-rakenne

Do-while on toistorakenne, joka C-kielestä löytyy mutta esimerkiksi Python-kielestä ei. Do-while-toistorakenteesta hieman tavallisesta poikkeavan tekeekin se, että se on lopetusehtoinen toistorakenne. Tämä käytännössä tarkoittaa sitä, että rakenteen **do-osio suoritetaan ainakin kerran riippumatta siitä mitkä katkaisuehdot ovat**. Do-while-rakenne toimii muuten melko lailla samallatavoin kuin normaali while-rakenne.

### Esimerkki 4.3: Do-While-rakenne käytännössä

#### Esimerkkikoodi

```
#include <stdio.h>

int main(void)
{
    int lopetus=5, aloitus = 10;

    printf("Do-osio toteutuu ainakin kerran.\n");

    do{

        if (lopetus < aloitus){

            printf("Lopetus on valmiiksi pienempi kuin aloitus:\n");
        }
        printf("lopetus: %d ja aloitus: %d ",lopetus, aloitus);
        aloitus++;

    }
    while (aloitus < lopetus);
    /* While-käskey tulee HETI do-osion perään ja päättyy puolipisteeseen*/

    return 0;
}
```

#### Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>4-3
Do-osio toteutuu ainakin kerran.
Lopetus on valmiiksi pienempi kuin aloitus:
lopetus: 5 ja aloitus: 10
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

Ohjelma demonstroi hyvin kuinka do-while-rakenne toimii. Toisin kuin muut toistorakenteet, do-while ei ennen ensimmäisen kierroksen loppua tarkasta lainkaan onko katkaisuehto jo toteutunut. Esimerkkikoodissa rakennetta olisi ollut tarkoitus toistaa niin kauan kun aloitus on pienempi kuin lopetus, ja vaikka jo toiston alkaessa luku oli puolet isompi, suoritettiin ensimmäinen kierros silti. Do-while-rakenne onkin parhaimmillaan kun suoritetaan toistoja, joissa ensimmäinen kierros periaatteessa testaa onko jotain käytettävissä ja jatkaa toistoa loppuun asti. Tällaisia tilanteita esimerkiksi ovat puurakenteiden läpikäynti, tiedostosta merkkien lukeminen sekä dynaamisten rakenteiden selaaminen.

## Ohjauskäskyistä continue, break ja pass

Valinta- ja toistorakenteisiin liittyi Python-kielessä hyvin tavallisesti ohjauskäskyt continue, break sekä pass. Myös C-kielessä nämä käskyt ovat pass-käskyä lukuun ottamatta käytettävissä ja toimivat täysin samalla tavoin kuin Pythonissa. Continue-käsky toistorakenteen sisällä lopettaa käynnissä olevan kierroksen ja aloittaa kierroksen alusta; break lopettaa sisimmän suoritettavana olleen toistorakenteen ja jatkaa loogisesti seuraavalta riviltä. Ja kuten aiemmin mainittu, pass-käskyä C-kieli ei tällä erää tunne.

### Esimerkki 4.4: Ohjauskäskyt

#### Esimerkkikoodi

```
#include <stdio.h>

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

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

## Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>4-4
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!
```

```
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

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

## #include, valmiiden funktiokirjastojen käyttäminen

Monipuolisempien ohjelmien toteuttaminen vaatii tavallisesti ulkopuolisten funktiokirjastojen käyttämistä. Esimerkiksi Python-ohjelmointikielessä funktiokirjastojen avulla pystyimme luomaan nopeita graafisia käyttöliittymiä, suorittamaan tieteellistä laskentaa ja jopa käyttämään tietokoneen verkkoyhteyttä.

C-kielessä ei oletusarvoisesti ole yhtä laajaa tai monitaitoista peruskirjastoa, mutta siitäkin huolimatta C-kieli tukee ulkoisten funktiokirjastojen käyttämistä. Itse asiassa C-kielessä jopa tarvitsemme tavallista useammin ulkoisia kirjastoja kuin Pythonissa, johtuen siitä että monet kehittyneemmät toiminnot on siirretty oletuskäskykannasta ulkoisiin kirjastoihin.

Olemme aiemmissa esimerkkitehtävissä toistuvasti aloittaneet aina käyttöönottamalla kirjaston nimeltä `stdio.h`, joka sisältää kehittyneemmät luku- ja kirjoitusfunktiot näytölle sekä tiedostoihin. Tämä luonnollisesti tehdään siis käskyllä

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

Vastaavasti, jos haluaisimme käyttöönottaa esimerkiksi kirjaston `stdlib.h`, joka sisältää laskentaa, muistinkäsittelyä, tyyppimuunnoksia sekä muita hyödyllisiä toimintoja, voisimme tehdä sen komennolla

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

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

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

Emme tässä osiossa käy kirjastoja tai niiden sisältöä sen tarkemmin läpi, vaan tutustumme niistä löytyviin funktioihin sitä mukaa kun tarvitsemme niitä. Kattavammin kirjastojen sisältö on esitelty liitteessä 1. Jos haluat lisätietoa kirjastoista, voit tutustua C-kielen perusfunktioihin mm. Eric Hussin kirjoittaman C-referenssioppaan [2] avulla, joka löytyy osoitteesta [http://www.acm.uiuc.edu/webmonkeys/book/c\\_guide/](http://www.acm.uiuc.edu/webmonkeys/book/c_guide/). Opas on englanninkielinen.

Omien funktiokirjastojen tekemisestä puhumme lisää oppaan toisessa osassa.

## #define, kiintoarvojen määrittäminen

C-kieleen liittyen on myös tavallista, että ohjelmakoodiin joskus määritellään pysyviä vakioarvoja, joita ohjelmassa käytetään toistuvasti. Tällöin puhumme kiintoarvon määrittämisestä, joka toteutetaan C-kielessä #define-määrittelyllä.

Käytännössä tämä tarkoittaa sitä, että annamme lähdekoodin alussa käskyn

```
#define nimi arvo
```

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

### Esimerkki 4.5: #define-määrittely

#### Esimerkkikoodi

```
#include <stdio.h>

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

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

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

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

    return 0;
}
```

#### Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehoteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>4-6
99999
Toimii!
#define-käskyn määrittely toimii myös numeroarvon kanssa.
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

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

Erityisen hyödyllisiä kiintoarvot ovatkin nimenomaan ”asetusarvoina” esimerkiksi taulukon koolle tai vastaaville toiminnoille. Kun määrittelemme arvon yhdessä paikassa lähdekoodin alussa, voimme asetusten muuttamista varten muuttaa pelkkää `#define`-käskyn arvoa sen sijaan, että joutuisimme muuttamaan jokaista kohta missä ko. arvoa on käytetty käsin. Tämä myös parantaa virhevarmuutta, koska pidemmässä lähdekoodissa yhden muutettavan kohdan huomaamatta jättäminen muuttuu hyvinkin todennäköiseksi ongelmaksi.

# Luku 5: Tiedostojen lukeminen ja kirjoittaminen

---

## Perusteet

Ulkoisia tiedostoja voidaan ohjelmointikielessä käyttää moninaiisiin tarkoituksiin, mutta tavallisimmin ne toimivat pysyväismuistina, josta luetaan ohjelmalle annettuja asetuksia tai aiemmin käsiteltyä tietoa. C-kielessä tiedostoja voidaan käsitellä luku- ja kirjoitusoperaatioissa lähes yhtä helposti kuin normaalia käyttäjältä tiedon lukemista. Käytännössä ulkoisen tiedoston käyttäminen on samanlaista kuin Pythonissa: tiedosto avataan haluttua käyttötarkoitusta varten sopivaan tilaan, tiedostoa luetaan tai kirjoitetaan ja käytön lopuksi tiedosto vapautetaan ja suljetaan. Lisäksi C-kielestä löytyy samalla tavoin monet muutkin Python-tiedostonkäsittelystä tutut elementit, kuten esimerkiksi tiedosto-osoittimet (ns. *kirjanmerkki*).

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

Ensimmäisenä voimme tutustua tiedostojen lukemiseen. Olemme ennen esimerkkejä tehneet samaan kansioon lähdekooditiedoston kanssa tekstitiedoston nimeltä *tiedosto.txt*, johon olemme tallentaneet muutaman merkkirivin sekä numeroja. Seuraavassa esimerkissä avaamme tiedoston, luemme sen sisällön ja tulostamme sen ruudulle.

## Esimerkki 5.1: Tiedostosta lukeminen

### Esimerkkikoodi

```
#include <stdio.h>

int main(void)
{
    char muisti[10];
    /*Luodaan tiedostokahva*/
    FILE *tiedosto;

    tiedosto = fopen("tiedosto.txt","r");

    printf("Tiedoston sisältö:\n");

    while (feof(tiedosto) == 0){
        fscanf(tiedosto, "%s", muisti);
        printf("%s\n",muisti);
    }

    fclose(tiedosto);

    return 0;
}
```

### Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>5-1
Tiedoston sisältö:
merkkijono
1234567890
karhennettu
turtana
0123456789

X:\C-kurssi\esimerkkikoodit>
```

### Kuinka koodi toimii

Koodissa meillä tällä kertaa aloitetaan sillä, että määrittelemme itsellemme kymmenen merkin kokoisen lukumuistin nimeltä `muisti`, sekä tiedostokahvan `tiedosto` käskyllä `FILE *tiedosto;`. Tämän jälkeen avaaamme haluamamme tiedoston `tiedosto.txt` lukumoodiin `"r"`, eli tilaan josta voimme ainoastaan lukea tiedoston sisältämää tietoa. Tämän jälkeen suoritamme itse tiedoston lukemisen `while`-toistorakenteella.

Koska emme tiedä, kuinka pitkä tiedosto on, määrittelemme lukemisen lopetusehdoksi tilanteen, jossa tiedosto on luettu loppuun. Tämä taas toteutetaan `while`-rakenteen ehdolla `feof(tiedosto) == 0`. `feof`-funktio toimii näet siten, että se palauttaa arvon nolla niin kauan, kun tiedostossa on vielä lukematonta tietoa, eli tiedosto-osoitin ei ole tiedoston lopussa.

## C-kieli ja käytännön ohjelmointi

### LTU

Seuraavalla rivillä suoritamme itse tiedostosta luvun. Komennolla `fscanf(tiedosto, "%s", muisti);` ilmoitamme, että luemme tiedostokahvasta tiedosto yhden merkkijonon, joka tallennetaan muuttujan muisti. Tässä yhteydessä tämä siis tarkoittaa sitä, että luemme tiedostosta merkkejä yhden merkkijonon, tai korkeintaan muisti-muuttujan koon verran. tämän jälkeen tulostamme luetun merkkijonon ruudulle. Huomaa, että `fscanf` toimii pääsääntöisesti aivan kuten käyttäjän syötteitä lukeva `scanf`, ainoan varsinaisen eron ollessa siinä, että ensimmäisenä parametrina määrittelemme, mistä tiedostokahvasta syöte luetaan. Lopuksi vielä suljemme tiedoston ja lopetamme ohjelman.

Kuten tästä esimerkistä huomasimme, on C-kielen tapa käsitellä tiedostoja hyvin samankaltainen kuin Pythonissa. Tiedosto avataan tarpeita vastaavalla moodilla tiedostokahvaan, tiedostosta luetaan tietoa muuttujaan ja lopuksi, kun tiedostoa ei enää tarvita, se suljetaan. Varsinaisesti tärkein ero onkin siinä, että C-kielessä pystymme kertomaan, millaista tietoa tiedostosta luemme. Seuraavaksi tutustumme tiedostojen kirjoittamiseen toisella lyhyellä esimerkillä.

### Esimerkki 5.2: Tiedostoon kirjoittaminen

#### Esimerkkikoodi

```
#include <stdio.h>

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

    tiedosto = fopen("tuloste.txt", "w");
    printf("Mitä haluat tiedostoon kirjoittaa?:\n");
    scanf("%s", &muisti);

    fprintf(tiedosto, "%s", muisti);

    fclose(tiedosto);

    return 0;
}
```

#### Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa `X:\C-kurssi\esimerkkikoodit`, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>5-2
Mitä haluat tiedostoon kirjoittaa?:
Robottikana

X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

Tämä koodi ei itsessään poikkea paljoakaan aiemmasta tiedostonluku-esimerkistä. Itse tiedostoon kirjoittaminen tapahtuu funktiolla `fprintf`, joka itse asiassa toimii aivan kuten normaalin ruudulle tulostuksen toteuttava `printf`-funktio. Tosin jälleen kerran joudumme tietenkin antamaan myös tiedostokahvan, johon tietoa kirjoitetaan. Huomaa lisäksi, että tällä kertaa käytimme kirjoituskahvaa ”w”, joka automaattisesti luo uuden tai tuhoaa aikaisemman samannimisen tiedoston, kun tiedosto otetaan käyttöön.

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

## LCO Tiedostojen käsittelyyn käytettäviä funktioita

**tietovirta=fopen(tiedosto, moodi);**

-Avaa tiedoston *tiedosto* käyttötilaan *moodi*.

-*tietovirta* on FILE \* -tyyppinen osoitinmuuttuja, jolla avaamisen jälkeen viitataan tiedostoon.

-*tiedosto* on avattavan tiedoston nimi, joko merkkijonomuuttuja tai merkkijono.

-*moodi* kertoo mihin tarkoitukseen tiedosto avataan. ”r” tarkoittaa lukemista, ”w” kirjoittamista, ”a” kirjoituksen jatkamista. Binääritiedostoja käsiteltäessä moodiin liitetään **b** (”rb”). Haluttaessa sekä lukea että kirjoittaa samanaikaisesti, lisätään vielä + (”r+”).

-Avattaessa tiedosto + -moodissa **r:n** ja **w:n** merkitys hieman muuttuu. Käytettäessä **r:ää** avataan olemassa oleva tiedosto tai luodaan uusi. **w:llä** luodaan aina uusi tiedosto. Jos tiedosto oli olemassa, se tuhoutuu.

**fclose(tietovirta);**

Sulkee tiedoston *tietovirta*.

**fscanf(tietovirta, formaattimerkkijono, muuttujalista);**

**fprintf(tietovirta, formaattimerkkijono, muuttujalista);**

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

**merkki=fgetc(tietovirta);**

**fputc(merkki, tietovirta);**

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

**fgets(merkkitaulukko, merkkien enimmäismäärä, tietovirta);**

**fputs(merkkitaulukko, tietovirta);**

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

**rewind(tietovirta) ;**

Siirtää tiedosto-osoittimen tiedoston alkuun.

**sijainti=ftell(tietovirta);**

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

**fseek(tietovirta, sijainti, alku);**

Tietovirran käsittelykohta siirtyy sijainnin osoittamaan kohtaan. alku kertoo mistä sijaintia aletaan laskea. SEEK\_SET tarkoittaa tiedoston alkua, SEEK\_CUR nykyistä kohtaa ja SEEK\_END tiedoston loppua.

**feof(tietovirta)**

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

**fread(osoitin, koko, määrä, tietovirta);**

**fwrite(osoitin, koko, määrä, tietovirta);**

*Nämä funktiot ovat binääritiedostojen käsittelyyn:*

fread lukee tietovirrasta koko-muuttujan kokoisia lohkoja määrä kappaletta ja sijoittaa ne osoittimen osoittamaan muistipaikkaan.

fwrite kirjoittaa tietovirtaan osoittimen osoittamassa muistipaikassa olevan tiedon, joka on koko-muuttujan kokoisissa lohkoissa määrä kappaletta.

**remove(tiedosto);**

Hävittää tiedoston.

**rename(nimi, uusi\_nimi);**

Vaihtaa tiedoston nimen.

## Luku 6: Funktiot ja osoittimet, komentoriviparametrit

---

### Perusteet

Aina kun ryhdymme rakentamaan suurempia ohjelmia, tulemme väistämättä tilanteeseen, jossa joudumme miettimään ohjelman toimintaa. Useimmiten hyötyisimme mahdollisuudesta käyttää uudelleen aiempaa koodia useaan kertaan toistuvien tehtävien suorittamiseen, toisinaan myös eri tehtävien jakaminen loogisiin kokonaisuuksiin helpottaisi ohjelman toteuttamista sekä ylläpitoa. Tämä kaikki on mahdollista alifunktioiden avulla.

### Alifunktion toteuttaminen

Kuten olet jo varmaan huomannut, on C-lähdekoodissa aina vähintään yksi funktio, `main`. Tämä funktio on se, jota kutsutaan ohjelman käynnistyessä ja jonka lopettaminen sammuttaa ohjelman. C-kielessä alifunktioiden toteuttaminen on hieman enemmän tarkkuutta vaativaa työtä kuin esimerkiksi Pythonissa. Ensinnäkin, me joudumme päättämään funktion palautusarvon tyyppin etukäteen; jos haluamme ohjelman palauttavan numerovastauksen, on funktion tyyppiksi annettava `int`, jos liukuluvun niin `float`. Jos taas haluamme merkkejä, käytämme tietenkin tyyppiä `char`. Lisäksi, mikäli emme aio palauttaa mitään arvoja, voimme laittaa funktion tyyppiksi myös tyhjä, eli `void`. Muista siis, että alifunktio, jonka muoto on `void`, ei voi palauttaa minkäänlaista arvoa, ja että funktio jolla on jokin ei-tyhjä muoto, on palautettava aina jotain. Huomaa myös, että `main`-funktion muoto voi olla `void`, ja tällöin sekään ei palauta minkäänlaista arvoa käyttöjärjestelmälle ohjelman lopettaessaan toimintansa.

Toinen vaatimus alifunktioiden käytölle on lisäksi se, että alifunktio on esiteltävä ohjelmakoodin alussa. Tämä käytännössä tarkoittaa sitä, että kääntäjälle kerrotaan etukäteen minkä nimisiä funktioita aiomme käyttää, mitä ne palauttaa ja millaisilla parametreilla niitä kutsutaan. Huomaa kuitenkin, että periaatteessa kutsua ei tarvita, mikäli funktiota käytetään koodissa vasta sen määrittelyn jälkeen, mutta on turvallisempaa esitellä kaikki alifunktiot lähdekoodin alussa, jolloin niiden kirjoitusjärjestyksellä ei enää ole mitään merkitystä. Tutustumme alifunktioiden tekemiseen ja käyttämiseen esimerkkien avulla; ensiksi tarkastelemme pelkästään alifunktion määrittelyä, esittelyä sekä kutsumista. Toisessa esimerkissä keskitymme palautusarvoihin ja parametreihin.

## Esimerkki 6.1: Alifunktion määrittely ja kutsuminen

### Esimerkkikoodi

```
#include <stdio.h>

/*Esitellään funktio*/
void alifunktio (void);

int main(void)
{
    printf("Tämä tulee pääfunktioista!\n");
    /*kutsutaan alifunktiota*/
    alifunktio();

    return 0;
}

/*Kirjoitetaan funktion koodi*/
void alifunktio(void){
    printf("Tämä tulee alifunktioista!\n");
}
```

### Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa `X:\C-kurssi\esimerkkikoodit`, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>6-1
Tämä tulee pääfunktioista!
Tämä tulee alifunktioista!

X:\C-kurssi\esimerkkikoodit>
```

### Kuinka koodi toimii

Ohjelmakoodi alkaa aiemmin mainitulla alifunktion esittelyllä, joka siis käytännössä tarkoittaa, että kääntäjälle kerrotaan heti ohjelman alussa funktion palautusarvo, nimi sekä vastaanotettavat parametrit. Käytännössä funktion esittely vastaakin täysin funktion määrittelyn ensimmäistä riviä.

Lisäksi koodin lopussa on itse funktion määrittely. Alifunktio on rakenteellisesti samanlainen kuin main-funktio, eli se toteutetaan pääfunktion kaarisulkeiden ulkopuolelle omaksi kokonaisuudekseen ja se päättyy kaarisulkeisiin, joka sulkee viimeisen auki olevan kaarisulkeen. Lisäksi, mikäli aliohjelma palauttaa jotain, päättyy se silloin `return`-käskyyn aivan kuin main-funktiokin. Tässä koodissa olemmekin määritelleet funktion nimeltä `alifunktio`, joka ei anna palautusarvoa (1. `void`) eikä vastaanota parametreja (sulkeissa oleva 2. `void`). Kun olemme määritelleet alifunktion, voimme kutsua sitä pääfunktiossa kirjoittamalla alifunktion nimen ja sulkeet, eli tässä tapauksessa `alifunktio()`; Tässä tapauksessahan funktiollamme ei ole parametreja, mutta jos niitä olisi annettu, olisi ne laitettu kutsussa sulkeiden sisään siinä järjestyksessä, missä ne on tarkoitus parametreina antaa.

## Parametrien välitys

Jos taas haluamme antaa funktiolle jotain parametreja, kuten esimerkiksi yhteenlaskettavia lukuja tai tutkittavia merkkejä, voimme tehdä tämän parametrien avulla. Alifunktion vastaanottamat parametrit tulee määritellä kääntäjälle etukäteen funktion esittelyn sekä funktion koodin yhteydessä. Käytännössä tämä tarkoittaaakin sitä, että funktion esittelyn -ja samalla myös kutsun- syntaksi on seuraavanlainen:

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

Eli periaatteessa vaikkapa

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

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

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

missä tulos, numero1 ja numero2 on integermuuttujia, lähettäisi muuttujien numero1 ja numero2 arvot laskettavaksi ja tallentaisi paluuarvon muuttujaan tulos.

### Esimerkki 6.2: Parametrit ja palautusarvot

#### Esimerkkikoodi

```
#include <stdio.h>

/*Esitellään funktio*/
int alifunktio (int lukul, int luku2);

/*Kirjoitetaan funktion koodi*/
int alifunktio(int lukul, int luku2){
    int tulos;
    tulos = lukul+luku2;
    return tulos;
}

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

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

    return 0;
}
```

## Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>5-1  
Parametrien 1 ja 2 summa on 3.
```

```
X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

Tällä kertaa olemme antaneet alifunktiokutsulle parametreina muuttujien arvo1 ja arvo2 lukuarvot ja tallennamme alifunktion paluuarvon muuttujaan vastaus komennolla

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

Annamme alifunktiolle parametrin, jotka sitten lasketaan yhteen ja paluuarvo annetaan takaisin pääohjelmalle. Lopputuloksena saamme siis muuttujaan vastaus laskutoimituksen tuloksen. Huomaa, että tällä kertaa alifunktion tyyppiä valitsimme integer-tyypin, koska alifunktio joutuu palauttamaan pääfunktiolle kokonaislukuarvon.

## Nimiavaruuksista

Tässä vaiheessa joudumme myös mainitsemaan myös muutaman sanan nimiavaruuksista. Nimiavaruushan tarkoittaa sitä, että ohjelman jokaisella funktiolla on oma nimiavaruutensa, eli periaatteessa omat muuttujansa. Helpommin sanottuna, jokaisella C-koodin funktiolla voi olla samanniminen muuttuja, kuten 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, eikä alifunktioissa tehdyt muutokset pääsääntöisesti vaikuta pääfunktion samannimisiin muuttujiin. Otamme tästä ensin esimerkin ja puhumme asiasta vielä hieman lisää:

## Esimerkki 6.3: Esimerkki nimiavaruuksista

### Esimerkkikoodi

```
#include <stdio.h>  
  
int alifunktio(int arvo){  
    arvo = 100;  
    printf("Alifunktiossa muuttuja arvo on %d.\n",arvo);  
    return arvo;  
}
```

## C-kieli ja käytännön ohjelmointi LTY

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

    /*kutsutaan funktiota ja annetaan arvo parametrina*/
    alifunktio(arvo);

    printf("alifunktion sijoitus ei vaikuttanut pääfunktion muuttujaan:\n");
    printf("Muuttuja arvo on edelleen %d.\n",arvo);

    return 0;
}
```

### Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>6-3
Alussa arvo on 1.
Alifunktiossa muuttuja arvo on 100.
alifunktion sijoitus ei vaikuttanut pääfunktion muuttujaan:
Muuttuja arvo on edelleen 1.
```

```
X:\C-kurssi\esimerkkikoodit>
```

### Kuinka koodi toimii

Tällä kertaa loimme niin pää- kuin alifunktioon muuttujan nimeltä arvo. Määrittelemme ensin pääfunktiossa muuttujalle arvon 1, ja tämän jälkeen annamme muuttujan arvon funktiokutsussa parametrina.

Alifunktiossa muuttujan arvoksi muutetaan 100, mikä todistetaan tulostamalla se ruudulle. Lisäksi tämä arvo vielä palautetaan pääfunktioon, mutta siitäkin huolimatta pääfunktion muuttuja arvo on edelleen sama 1 kuin ennen funktiokutsua. Miksi näin käy?

Ensinnäkin, lähetämme alifunktioon tosiaan ainoastaan muuttujan arvon, eli numeroarvon 1, emme muuttujaa arvo, tai mitään muuta siihen liittyvää. Alifunktio luokin omaan nimiavaruuteensa samannimisen muuttujan, ja muuttaa itsensä luoman arvo-nimisen muuttujan arvoksi 100, sekä palauttaa oman muuttujansa sisällön – numeroarvon 100 – takaisin pääfunktioon. Pääfunktiossa meidän tulee kiinnittää huomiomme funktiokutsuun:

```
alifunktio(arvo);
```

Tämä kutsu ei ota paluarvoa kiinni – eli ei tallenna sitä minnekään - joten pääfunktion muuttujan arvo sisältämä luku 1 ei muutu. Tämän vuoksi muuttujan sisältämä luku on edelleen sama ohjelman lopussa. Jos olisimme halunneet muuttaa pääfunktion muuttujaa arvo, olisi siihen meillä kolme vaihtoehtoa. Ensinnäkin, olisimme voineet ottaa alifunktion paluarvon talteen muuttamalla kutsun muotoon

```
arvo = alifunktio(arvo);
```

tai käyttämällä globaaleja muuttujia (lisätietoa [2,3]) tai käyttämällä osoittimia, joista puhumme seuraavassa osiossa lisää.

## Osoittimet

Osoittimet ovat C-kielelle sekä muutamalle muulle laitteistotason ohjelmointikielelle ominaisia ”muistiosoitinmuuttujia”. Lyhyesti ilmaistuna, osoittimet ovat eräänlaisia muuttujia, jotka sen sijaan, että sisältäisivät tiedon mikä muuttujan arvo on, sisältävät tiedon siitä mistä muistiosoitteesta arvo löytyy. Osoittimia kannattaa ajatella hieman kuten internet-osoitteita: Normaali muuttuja on periaatteessa kuin itse varsinainen verkkosivu, joka sisältää kaiken sivuilla olevan tiedon. Osoittimet taas ovat kuin verkko-osoitteita (esimerkiksi <http://fi.wikipedia.org>), eli ne eivät sisällä sivun tietoja (tässä tapauksessa Suomen wikipedian etusivun artikkeleita), vaan ainoastaan tiedon siitä, mistä nämä artikkelit on mahdollista löytää.

Osoittimien tärkein hyöty tulee niiden joustavuudesta tiedonkäsittelyn suhteen. Vaikka se vielä tässä vaiheessa ei olekaan keskeistä tietoa, niin mainittakoon että osoittimien avulla voidaan suhteellisen helposti hallita monimutkaisia tietorakenteita sekä periaatteessa aivan vapaasti yhdistellä tietueita dynaamisiksi muistirakenteiksi. Tässä vaiheessa emme kuitenkaan mene vielä niin pitkälle, vaan keskitymme ensin osoittimien perusasioihin.

Ensinnäkin, osoittimen merkinä käytetään \*-merkkiä muuttujan nimen edessä. Tämä kertoo kääntäjälle, että aiomme luoda muuttujasta osoittimen emmekä normaalia muuttujaa. Yleisesti ottaen \*-etumerkki tulee lukea ”arvo, joka sijaitsee tämännimisessä osoitteessa”. Lisäksi pointtereiden yhteydessä käytetään joskus &-merkkiä, joka tarkoittaa, että emme viittaa osoitettavan muistipaikan sisältöön, vaan osoitinmuuttujan fyysiseen osoitteeseen, joka periaatteessa on suurehko numeroarvo, ja se viittaa suoraan muistipaikkaan tietokoneen keskusmuistissa; ”Osoite, jota tämänniminen muuttuja käyttää”. Yleisintä &-merkin käyttö on kun asetamme pointterin osoittamaan jonkin muuttujan sisältöön. Tämä kaikki voi ensi alkuun vaikuttaa hieman sekavalta, mutta seuraavassa esimerkissä kokeilemmekin osoittimen käyttöä käytännössä.

### Esimerkki 6.4: Osoittimen tekeminen ja käyttäminen

#### Esimerkkikoodi

```
#include <stdio.h>

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

void ei_kasvata(int arvo){
    arvo = 1000;
}
```

## C-kieli ja käytännön ohjelmointi LTY

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

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

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

    /*kutsutaan alifunktiota ei_kasvata*/

    ei_kasvata(luku_2);

    printf("Alifunktion ei_kasvata jälkeen luku_2 arvo on %d.\n", luku_2);

    /*kutsutaan alifunktiota kasvata*/
    kasvattaa(&luku_2);
    printf("Alifunktion kasvattaa jälkeen luku_2 arvo %d.\n", *osoitin_2);

    return 0;
}
```

### Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>6-3
Osoitin 1 viittaa muistipaikkaan 2293612 jossa on arvo 1.
Alifunktion ei_kasvata jälkeen luku_2 arvo on 2.
Alifunktion kasvattaa jälkeen luku_2 arvo 1000.
```

```
X:\C-kurssi\esimerkkikoodit>
```

### Kuinka koodi toimii

Koodissa on ensinnäkin kaksi alifunktiota; huomaa, että tällä kertaa meidän ei tarvitse esitellä alifunktiota, koska ne on määritelty ennen ensimmäistä käyttökutsua. Ensimmäinen alifunktio, kasvattaa, saa parametrina yhden osoittimen nimeltään arvo, joka viittaa kokonaislukuun, ja toinen alifunktio, ei\_kasvata, saa normaalin kokonaislukuparametrin. Muuten funktiot ovat identtisiä.

Itse ohjelmakoodissa luomme kaksi osoitinta, osoitin\_1 ja osoitin\_2, ja asetamme ne osoittamaan muuttujien luku\_1 ja luku\_2 arvoihin. Tämän jälkeen testamme osoittimien toimintaa. Ensinnäkin tulostamme hieman tietoja osoittimesta: näemme, että osoitin\_1 osoittaa muistipaikkaan numero 2293612, johon on tallennettuna arvo 1. Huomaa erityisesti, että osoitinta voidaan käyttää tiedon tulostamiseen normaalisti muuttujan tavoin.

Seuraavaksi kokeilemme oikeasti osoittimien eroja. Ensin kutsumme alifunktiota ei\_kasvata, joka toimii kuten aiemmassa esimerkissä huomasimmekin. Kuinka tämä voidaan siis toteuttaa osoittimilla? Ensinnäkin, kuten varmaan huomaat, määritellään kasvattaa-funktion parametreissa

se, että parametri arvo on oltava osoitin. Tästä johtuen kutsuessamme alifunktiota emme annakaan muuttujan `luku_2` arvoa kuten tavallisesti, vaan lähetämme `&`-merkin avulla arvon fyysisen osoitteen, eli käytännössä siis osoittimen `luku_2`-muuttujan käyttämään muistipaikkaan. Tämän seurauksena alifunktio muuttaa sen muistipaikan arvoa, johon pääfunktion `luku_2` on tallennettu, ja tämän vuoksi muutos jää voimaan. Huomaa lisäksi, että osoittimen viittaamaa arvoa pystyttiin muuttamaan normaalilla sijoitusoperaatiolla.

Pääohjelmassa meillä on vielä viimeinen tulostuskäske. Huomaa, että olemme jälleen käyttäneet tulostuskäskeyssä muotoa `*osoitin_2`, joka edelleen viittaa muuttujan `luku_2` arvon fyysiseen sijaintiin. Koska muutimme itse muistipaikan arvoa edellisessä alifunktiossa, löytyy osoittimen `osoitin_2` ilmoittamasta paikasta arvo 1000.

## Komentoriviparametrien käyttäminen

Luvun viimeinen uusi asia on komentoriviparametrit. Aiemmissa kohdissa puhuimme mm. alifunktioiden parametreista, joten onkin luonnollista, että tutustumme näin lopuksi vielä main-funktion parametreihin.

Aivan kuten Python-ohjelmointikielessä, on C-kielessäkin komentoriviparametrien käyttöönotto varsin yksinkertaista. Olemme aiemmin kirjoittaneet main-funktion muotoon

```
int main(void)
```

ja jatkaneet työskentelyä sen suuremmin koodia itseään ajattelematta. Kuitenkin, jos haluaisimme, että käyttäjä syöttäisi jo ohjelman käynnistyksen yhteydessä joitain tietoja, kuten kohdetiedoston nimen tai muuta vastaavaa, tarvitsisimme mahdollisuuden tallentaa käyttäjän antamat syötteet main-funktion parametreina. Tämä on itseasiassa varsin helppoa, meidän tarvitsee vain muuttaa main-funktioon parametrinärittely

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

eli vaikkapa

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

ja kääntäjä hoitaa loput. Tämän jälkeen näemme suoraan annettujen parametrien määrän ensimmäisestä muuttujasta, ja itse parametrit toisesta. Yksittäisiin parametreihin voimme viitata notaatiolla `argv[i]` (tai `parametrilista[i]`), jossa `i` on parametrin järjestysnumero. Huomaa kuitenkin, että parametrit ovat aina tallennettu merkkijonoina, ja niiden käyttämistä varten saatetaan joutua tekemään tyyppimuunnoksia (kts. liite 1, `cctype`, `stdlib`). Lisäksi muista, että parametri 0 on - kuten Pythonissakin - ajettavan tiedoston nimi ja täten ensimmäinen varsinainen parametri on paikalla 1. Lisäksi tämä tarkoittaa sitä, että parametreja annetaan aina käytännössä vähintään yksi. Lopuksi katsotaan vielä esimerkki komentoriviparametrien toiminnasta:

## Esimerkki 6.5: Komentoriviparametrien käyttäminen

### Esimerkkikoodi

```
#include <stdio.h>

/* Lisäämme main-funktioon tuen komentoriviparametreille:
/* Ensin interger-muuttuja argc, johon tulee tieto parametrien määrästä
   tämän jälkeen merkkitaulu argv, johon tulee 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;
}
```

### Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>6-4 yksi kaksi 313
Annoit 4 komentoriviparametria, jotka olivat:
0. parametri oli 6-4
1. parametri oli yksi
2. parametri oli kaksi
3. parametri oli 313

X:\C-kurssi\esimerkkikoodit>
```

### Kuinka koodi toimii

Koodi ei Python-ohjelmoijalle ole kovinkaan epätavallisen näköinen. Käyttöön otimme komentoriviparametrit lisäämällä main-funktion määrittelyyn parametrit `int args` ja `char *argv[]`, jotta voimme vastaanottaa parametreja. Tämän jälkeen tulostamme saadut parametrit `argv`-taulukosta `for`-lauseen avulla. Huomaa edelleen, että ”0. parametri” oli käynnistettävän tiedoston nimi ja että parametrien yhteismäärä `args`-muuttujassa sanottiin olevan 4 vaikka käyttäjä periaatteessa ei syöttänyt kuin 3 parametria.

Viimeiseen parametriin liittyy kuitenkin vielä yksi huomio: numeroarvona käyttäminen. koska parametrit on nyt tallennettu merkkeinä, tarkoittaa tämä sitä että emme voi käyttää arvoa 313 suoraan numeroarvona tyypittämättä sitä ensin kokonaisluvuksi. Mikäli joskus törmäät tällaiseen tilanteeseen, löydät lisäohjeita liitteestä 1 ja sieltä löytyvistä tyyppimuunnosfunktioiden käyttöohjeista.

## Luku 7: Tietueet ja muistinkäsittely

---

Tässä luvussa käymme läpi C-kielen esitystavan rakenteiselle tietomuodolle, eli tilanteille, jossa haluamme muodostaa useammasta perustyyppin muuttujasta koostuvan tietorakenteen. Tähän törmäämme esimerkiksi silloin, kun haluamme tallentaa vaikkapa henkilötietoja. Koska jokaisella meistä on syntymäpäivä, etunimi, sukunimi, kotiosoite jne, eikö olisikin helpointa jos meillä olisi tietomuoto, joka olisi suunniteltu tällaisten asioiden tallentamiseen?

### Tietueet

C-kielessä voimmekin itse määritellä oman tietomuotomme – tietueen – `struct`-määrittelyillä. Määrittely ja käyttäminen vastaa pitkälti Pythonin luokkarakennetta ilman jäsenfunktioita, ja niitä muodostetaan syntaksilla

```
struct tietueen_nimi{
    tietueen_jäsenet, eli tyyppi ja nimi
};
```

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

### Esimerkki 7.1: Tietueen määrittely ja käyttäminen

#### Esimerkkikoodi

```
#include <stdio.h>

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

## C-kieli ja käytännön ohjelmointi

### LTY

```
int main(void)
{
    struct ihminen henkilo;

    henkilo.ika = 17;
    strcpy(henkilo.osoite, "sihijuomakatu");

    printf("Mikä laitetaan henkilön nimeksi?:\n");
    scanf("%s",&henkilo.nimi);

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

### Esimerkkikoodin tuottama tulos

Kun käännämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>7-1
Mikä laitetaan henkilön nimeksi?:
Hjördis
Hjördis on alaikäinen.
X:\C-kurssi\esimerkkikoodit>
```

### Kuinka koodi toimii

Jos aiomme käyttää tietueita, joudumme aloittamaan lähdekoodin ensiksi kertomalla kääntäjälle millaisia tietueita lähdekoodissa käytetään. Tämä tietenkin tapahtuu kirjoittamalla tietueiden esittelyt lähdekoodiin ennen funktioesittelyitä ja toiminnallista koodia. Tällä kertaa luomme koodilla

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

tietueen nimeltä ihminen, jonka jäsenmuuttujina ovat merkkijonotaulu nimi, kokonaislukumuuttuja ika sekä merkkijonotaulu osoite. Kun olemme luoneet tietueen, voimme käyttää sitä itse koodin sisällä. Vielä tässä vaiheessa emme kuitenkaan ole tehneet muuta kuin kertoneet kääntäjälle millaisia rakenteita lähdekoodissa esiintyy. Tämän vuoksi main-funktion sisällä meillä on rivi

```
struct ihminen henkilo;
```

jolla varsinaisesti käyttöönotamme tietueen. Itseasiassa, rivillä ilmoitetaan, että luomme tietuetyypistä ihminen (struct ihminen) muuttujan henkilo. Tästä eteenpäin voimme käyttää henkilön jäsenmuuttujia kuten normaaleja muuttujia. Esimerkin muilla riveillä onkin esitelty mm. merkkijonojen kopiointia jäsenmuuttujaan, jäsenmuuttujan käyttöä if-rakenteen ehtona sekä jäsenmuuttujien tulostamista.

## Muistinvaraus

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

C-kielessä muistinvaraus toteutetaan `stdlib`-kirjastossa olevalla `malloc`-funktioilla. `malloc`-funktio saa syötteenä itselleen perustyyppin (muuttujatyyppin), jota haluamme käyttää tallentamiseen sekä määrän, jonka tyyppiä aiomme varata. Tässä on kuitenkin eräs ongelma; kuinka me voimme varata muistia merkkijonoa varten, jos meidän ensin tulee laskea sen pituus ja vasta tämän jälkeen osaisimme kertoa minkä verran muistia tarvitaan.

Tähän ongelmaan käytämme ratkaisuna lukupuskuria, joka käytännössä on yksi staattinen merkkitaulu joka lukee standardisyötevirtaa. Lukupuskurin on hyvä olla riittävän suuri jotta siitä on mitään hyötyä, mutta kohtuullisen kokoinen jotta sen käyttö olisi mielekästä. Esimerkiksi 256 tavua on useimmiten riittävä puskuri syötevirran lukemiseen. Lisäksi tiedostojen yhteydessä voimme ensin laskea montako merkkiä esimerkiksi rivillä on ja vasta tämän jälkeen lukea, jolloin tarve puskurille periaatteessa poistuu.

Kun varaamme muistia, tulee `malloc`-funktioille tyyppin ja määrän lisäksi antaa pointteri, jonka `malloc` asettaa varaamansa muistin alkuun. Tämän pointterin arvon `malloc` myös palauttaa. Lisäksi on C-kielen hyvän ohjelmointitavan mukaista suorittaa aina muistinvarauksessa testaus siitä onnistuiko muistin varaaminen. Seuraavassa esimerkissä kokeilemme merkkijonon muistinvarausta sekä lukupuskurin käyttämistä.

## Esimerkki 7.2: Manuaalinen muistinvaraus

### Esimerkkikoodi

```
in #include <stdio.h>

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

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

    /*Lasketaan luetun merkkijonon pituus. */
    pituus = strlen(lukumuisti);

    /*Varataan muistia luetun sanan pituuden verran (pituus* 1 merkin koko). */
    muistissa = (char *) malloc(pituus*sizeof(char));

    /*Kopioidaan merkkijono muistiin*/
    strcpy(muistissa, lukumuisti);

    printf("Kirjoitit merkkijonon %s, joka oli %d merkkiä pitkä.\n", muistissa,
    pituus);

    printf("Staattisessa muistinhallinnassa merkkijono vie tilaa %d tavua.\n",
    sizeof(lukumuisti));

    printf("Dynaamisessa muistinhallinnassa merkkijonon pointteri vie tilaa %d
    tavua.\n", sizeof(muistissa));

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

    return 0;
}
```

### Esimerkkikoodin tuottama tulos

Kun käänämme esimerkkikoodin ja ajamme sen komentokehotteessa kansiossa X:\C-kurssi\esimerkkikoodit>, saamme seuraavanlaisen tuloksen:

```
X:\C-kurssi\esimerkkikoodit>7-2
Kirjoita jotain (max.255 merkkiä):
Apumiehensijaisenvaramiespalveluvastaava
Kirjoitit merkkijonon Apumiehensijaisenvaramiespalveluvastaava, joka oli 40
merkkiä pitkä.
Staattisessa muistinhallinnassa merkkijono vie tilaa 255 tavua.
Dynaamisessa muistinhallinnassa merkkijonon pointteri vie tilaa 4 tavua.

X:\C-kurssi\esimerkkikoodit>
```

## Kuinka koodi toimii

Ohjelman alussa luomme kolme muuttujaa, staattisen merkkijonotaulukon lukumuisti, merkki-osoittimen muistissa sekä integer-muuttujan pituus, joiden avulla tallennamme käyttäjän syöttämän merkkijonon dynaamisesti varattuun muistiin.

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

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

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

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

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

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

## Huomioita muistinvarauksesta

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

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

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

`exit()` on funktio joka lopettaa ohjelman suorituksen; vastaava käsky kuin Pythonin `sys.exit()`.

## Loppusanat

---

### Huomioita

Tässä oppaan ensimmäisessä osassa käydään lyhyesti läpi C-ohjelmoinnin alkeet lyhyiden esimerkkien ja ohjetekstien avulla. Kuten varmaan huomasit, emme vielä tähän mennessä ole oikeastaan käsitelleet ”käytännön ohjelmointia”, eli ohjelmointiprojekteja tai suunnittelulähtöistä ohjelmointia, mutta palaamme siihen oppaan toisessa osassa. Vaikka tämä johdanto ei ehkä ole kovinkaan täydellinen ja jättää monta asiaa ohjelmoijan itsensä opeteltavaksi, toivon että tästä alustuksesta kuitenkin on jotain hyötyä jatkos kannalta.

### Lisäluettavaa

Mikäli haluat lisää ohjeita C-kielen käyttämiseen, on suositeltavaa, että tutustut jo aiemminkin mainittuun Eric Hussin C-referenssikirjastoon. Toisaalta taas, mikäli haluat tutustua kirjamuodossa olevaan tekstiin, on Kerninghanin ja Piken *The Practice of Programming* [4] ja Kerninghanin ja Ritchien *The C Programming Language* [5] hyviä kirjoja lisätiedon hankkimiseen. Molemmat kirjat tosin ovat englanninkielisiä.

Mikäli taas haluat enemmän suomenkielistä lisämateriaalia, voit turvautua Internetin hakukoneisiin kuten Google tai [www.fi](http://www.fi). Molemmissa hakukoneissa on optio rajoittaa hakua ainoastaan suomenkielisiin sivustoihin. Älä myöskään unohda vilkaista tämän oppaan liitettä 1, johon on koottu referenssikirjastosta tavallisimpien käskyjen lisäksi myös käsittelemättä jääneitä asioita.

## Lähdeluettelo

---

- [1] Kasurinen, Jussi (2006). Python-ohjelmointiopas, versio 1. Lappeenrannan teknillinen yliopisto, Tietotekniikan käsikirjat 7. ISBN 952-214-286-7
- [2] 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](http://www.it.lut.fi/kurssit/05-06/Ti5210210/opas/c_opas/c_opas.html).
- [3] Huss, Eric (1997). The C Library Reference Guide. saatavilla osoitteesta [http://www.acm.uiuc.edu/webmonkeys/book/c\\_guide/](http://www.acm.uiuc.edu/webmonkeys/book/c_guide/), viitattu 18.12.2007.
- [4] Kernighan, Brian W., Pike, Rob. (2005) The Partice of Programming, 13<sup>th</sup> edition. Addison-Wesley Professional Computing Series.
- [5] Kernighan, Brian W., Ritchie, Dennis M. (1988) The C Programming Language, Second edition. Prentice Hall Software Series.

## Liite 1: Funktiokirjastot

---

Tässä liitteessä käymme läpi kootusti referenssikirjaston käskyt. Jo mainittujen käskyjen lisäksi liitteessä on mukana myös ne käskyt, joita oppaan esimerkeissä ei erikseen mainittu tai käytetty missään yhteydessä.

Liitteen teksti on pitkälti Alaoutisen lyhyestä C-oppaasta. Tekstiä on saatettu muokata ulkoasun yhtenäisyyden parantamiseksi

### **LGO float.h**

#### **DBL\_MIN**

Pienin positiivinen kaksoistarkkuuden luku

#### **DBL\_MAX**

Suurin kaksoistarkkuuden luku

### **limits.h**

#### **CHAR\_MIN**

Merkkityypin pienin arvo

#### **CHAR\_MAX**

Merkkityypin suurin arvo

#### **INT\_MIN**

Pienin kokonaisluku

#### **INT\_MAX**

Suurin kokonaisluku

## ctype.h

### isalnum(x)

Testaa, onko x alfanumeerinen merkki

### isalpha(x)

Testaa, onko x kirjain

### isdigit(x)

Testaa, onko x numeromerkki

### islower(x)

Testaa, onko x pieni kirjain

### isspace(x)

Testaa, onko x tyhjä merkki (välilyönti, rivinvaihto,...)

### isupper(x)

Testaa, onko x iso kirjain

### tolower(x)

Muuntaa x:n vastaavaksi pieneksi kirjaimeksi

### toupper(x)

Muuntaa x:n vastaavaksi isoksi kirjaimeksi

## math.h

haluttaessa käyttää math.h:n funktioita, on käännettäessä lisättävä rivin perään -lm ( <hepo:> cc -Aa koodi.c -o koodi -lm )

### abs(x)

Kokonaisluvun itseisarvo

### acos(x)

Arkuskosini x

### asin(x)

Arkussini x

### atan(x)

Arkustangentti x

### ceil(x)

Palauttaa pienimmän kokonaisluvun, joka on suurempi kuin x

### cos(x)

Kosini x

### cosh(x)

Hyperbolinen kosini x

### exp(x)

e potenssiin x

### fabs(x)

Reaaliluvun itseisarvo

### floor(x)

Palauttaa suurimman kokonaisluvun, joka on pienempi kuin x

### log(x)

ln x

### log10(x)

log<sub>10</sub> x

### pow(x,y)

x potenssiin y

### sin(x)

Sini x

### sinh(x)

Hyperbolinen sini x

### sqrt(x)

Neliöjuuri x

### tan(x)

Tangentti x

### tanh(x)

Hyperbolinen tangentti x

## stdio.h

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

### EOF

(-1), tiedoston loppu

### NULL

(0), osoittimen arvo 0

### stdin

standardi syöttötiedosto, normaalisti näppäimistö

### stdout

standardi tulostustiedosto, normaalisti näyttö

### stderr

standardi virhetiedosto, normaalisti näyttö

### SEEK\_SET

(0), tiedoston alku

### SEEK\_CUR

(1), nykyinen tiedostosijainti

### SEEK\_END

(2), tiedoston loppu

### FILE

rakenteinen tyyppi tiedostojen käsittelyyn

### fpos\_t

tyyppi, jota käytetään haluttaessa määrittää yksikäsitteisesti sijainti tiedoston sisällä

### fopen(tiedostonimi, avausmuoto)

avaa tiedoston *tiedostonimi* avausmuoto kertomaan käyttötarkoitukseen

### fclose(tiedosto)

sulkee *tiedosto* tiedoston

### fflush(tiedosto)

tyhjentää tiedoston *tiedosto* puskurin

### fseek(tiedosto, etäisyys, paikka)

siirtyy tiedostossa *tiedosto* etäisyys tavua kohdasta *paikka* laskien. *paikka* on joko SEEK\_SET, SEEK\_CUR tai SEEK\_END

### ftell(tiedosto)

kertoo sijainnin tiedostossa *tiedosto*

### rewind(tiedosto)

siirtyy tiedoston alkuun.

### fgetpos(tiedosto, sijainti)

sijoittaa parametrin *sijainti* arvoksi nykyisen kohdan tiedostossa

### fsetpos(tiedosto, sijainti)

siirtyy tiedostossa kohtaan *sijainti*

### feof(tiedosto)

palauttaa nollasta poikkeavan arvon, jos ollaan tiedoston lopussa

### ferror(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)**

binääritiedoston käsittelyyn. Lukee  
tiedostosta enintään määrä \* koko  
tavua ja sijoittaa sen muuttujan  
rakenne osoittamaan paikkaan

**fwrite(rakenne, koko, määrä, tiedosto)**

binääritiedoston käsittelyyn. Lukee  
muuttujasta rakenne määrä \* koko  
tavua ja kirjoittaa sen tiedostoon

**remove(tiedosto)**

poistaa tiedoston

**rename(mjono1, mjono2)**

muuttaa tiedoston nimen mjono1:stä  
mjono2:ksi

## stdlib.h

**atoi(x)**

Muuttaa merkkijonon x kokonaisluvuksi

**atof(x)**

Muuttaa merkkijonon x reaaliluvuksi

**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 käyttöjärjestelmällä x:n osoittaman komennon. `system("rm tiedot.txt");`

## string.h

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

### **memchr(s1, c, n)**

etsii merkkiä c osoitteesta p 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