C-ohjelmointi, kevät 2006

Modulaarinen ohjelmointi
Kertaus osoittimista

Luento 9
28.3.2006

Sisältö

- Modulaarisuus C:ssä
- Käännösyksikkö ja otsaketiedosto
- Makefile
- Kertaus osoittimista ja funktioista
- Kokeesta

Ison ohjelman toteuttaminen

- Kokonaisuus on jaettava hallittaviin osiin
  - Toiminnallisia kokonaisuuksia (Java: luokat)
    - Syöttö ja tulostus
    - Virheiden käsittely
    - Tietyn toiminnan tai rajatun tehtävän toteuttaminen
  - Osien väliset rajapinnat eli parametrit ja paluuarvot
    - Selkeästi määriteltyjä
    - Vähän riippuvuuksia eri osien välillä

Moduularisuus on keino hallita monimutkaisuutta!

Moduulit toteuttavat abstraktion, kapseloinnin ja informaation piiliottamisen.

Modulaarisuuden hyödyt

- Jako osiin + selkeät ja yksinkertaiset rajapinnat => koko ohjelman rakenne selkeämmäksi
- Ohjelmisto voidaan toteuttaa projektityönä: eri henkilöt ohjelmoivat eri osat
- Testaus voidaan aloittaa hyvin varhaisessa vaiheessa
- Ylläpito helpottuu: muutokset vain tarvittaviin kohtiin eli joihinkin funktioihin
- Koodin uudelleenkäyttö: standardikirjastot, omat kirjastot
  - Kerran ratkaistu ja koodattu, voidaan käyttää monta kertaa

Modulaarinen ohjelmointi ja C-kieli

- C ei varsinaisesti tue modulaarista ohjelmointia (vertaa esim. Modula)
- C:ssa on piirteitä, joiden avulla ohjelman modulaarisuus voidaan toteuttaa
  - funktiot ja niiden protyypit
  - otsaketiedostot (header files)
- Näitä piirteitä sopivasti käyttämällä saadaan C:ssä toteutettua modulaarinen ohjelma tehokkaasti
  - Toiminnon toteutus ó toiminnon käyttö

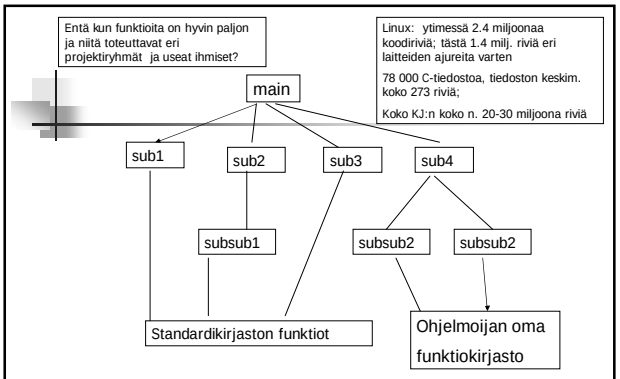
Piilotetaan käyttäjältä tarkat ohjeet käyttämiseen

C:n piirteitä modulaariseen ohjelmointiin

- funktiot
  - Ohjelma jaetaan useaksi pienehköksi funktioksi, joista kukin suorittaa tietyn toimenpiteen
    - ~ noin 20 riviä kommentteineen on sopiva funktion koko, sillä näkyy helposti kokonaan näytöllä
    - Valmiit kirjastofunktiot: standardikirjaston funktiot tai omat kirjastofunktiot käytettävissä
  - Pääohjelma main lähinnä koostuu funktiokutsuista
  - Lisähierarkiaa: funktiot voivat kutsua toisia funktioita
- funktioiden esittelyt eli prototyypit
  - Funktio on määriteltävä tai esiteltävä ennen käyttöä
  - Kääntäjä voi tarkistaa oikean käytön
- otsaketiedostot
  - Sisältävät tietoja, joita ohjelmat tarvitsevat voidakseen käyttää muualla määriteltyjä funktioita
    - #include <stdio.h>

Pienehkön ohjelman modulaarinen rakenne

- #include-määrittelyt kirjastofunktioiden liittämiseksi
  - voivat sisältää toisia includeja
- vakiomäärittelyt, tyyppiesittelyt
  - helpompi hallita: löytää ja muuttaa
- Ohjelman funktioiden prototyypit eli niiden esittely
  - Näin määritellään ennen käyttöä
- Pääohjelma main (joka ohjelmassa ainakin yksi)
  - ja sen funktiokutsut
- Ohjelman funktioiden määrittelyt



Modulaarisuus funktioiden tasolla

Isohko ohjelma tai projekti

- Yhteenkuuluvat funktiot omiksi tiedostoiksi eli käännösyksiköiksi => moduuli
  - Linuxin koodi
- Kukin käännösyksikkö voidaan kääntää ja testata erikseen
  - => objektimoduuli
- Ajettava ohjelma saadaan, kun erikseen käännetyt objektimoduulit linkitetään yhteen
- Erikseen kunkin moduulin lähdekooditiedosto ja otsaketiedosto => joustavuutta kääntämisessä
  - Ei tarvitse kääntää turhaan objektimoduulin koodia
  - Otsaketietojen liittäminen riittää

Kooditiedostot muodostavat C:n moduulin

- C:n funktiot ovat globaaleja, kaikkialta ohjelmasta kutsuttavissa
  - Eivät sellaisenaan yhdessä tiedostossa tarjoa
    - riittävää kapselointia
    - toteuttamisen, muuttamisen, kääntämisen ja ylläpidon joustavuutta
- Toteutuksen ja käytön erottava moduuli syntyy, kun
  - kootaan joukko yhteenliittyviä funktioita samaan kooditiedostoon (jokunimi .c) = moduulin toteutus
  - ja näiden funktioiden prototyypit ja muu muiden moduulien käyttöön tarkoitettu data otsaketiedostoon (jokunimi.h) = moduulin rajapinta

Kapselointi: static -

- Määrittelee talletusluokkaa ja näkyvyyttä (scope)
  - Paikalliset muuttujat
    - Funktion paikallisille muuttujille varataan tilat pinosta. Ne ovat käytettävissä vain funktion suoritusajan.
    - static-määrittely => vain funktion sisällä käytettävissä, mutta säilyttävät arvonsa käyttökerrasta toiseen (esim.käyttökertalaskuri)
  - Globaalit muuttujat ja funktiot
    - static-määrittely rajaa näkyvyyden siihen tiedostoon, jossa ne ovat määritelty
    - Mahdollistaa datan piilottamisen

Kapselointi:
”estä tiedon vuotaminen ulkopuolelle”

~ Javan private

- Kapseloidaan moduulin muuttujia ja funktioita
  - static => eivät ole käytettävissä moduulin ulkopuolella
    - Hyvä käytäntö jotenkin erottaa nimestä tällaiset muuttujat ja funktiot
  - funktiota edeltää static-määre
    - static void e_append_(); /* kuuluu editor.h-tiedostoon */
  - Globaalia muuttujaa edeltää static-määre
    - static int e_flag_ = 0; /* tämä samoin*/
  - static muuttaa sen, kuinka linkittäjä käsittelee muuttujaa
- Huom! C:ssä kapselointiyksikkönä on tiedosto! Javassa objekti.

extern-määrittely

- Funktioiden esittelyssä oletusarvona => voidaan jättää pois
- Yhteiskäyttöiset muuttujat
  - Harvoin todella tarpeen ja hyödyllisiä
  - extern int myErrorNo
    - Linkittäjä: kaikki viittaukset kohdistuvat samaan muuttujaan, vaikka olisivat eri tiedostoissa
    - Globaalin muuttujan static-määrittely =>vain samassa tiedostossa kaikki viittaukset kohdistuvat samaan muuttujaan. Toisen tiedoston viittaukset kohdistuvat toiseen muuttujaan.

Vaikutus linkittäjän toimintaan

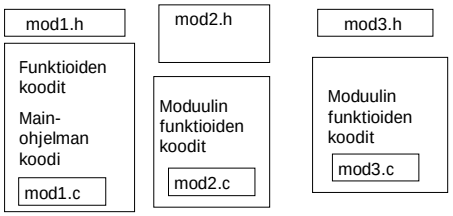
- Ohjelmassa voi olla useita samannimisiä muuttujia (esim. i,j) ja linkittäjän täytyy tietää, milloin kyseessä on viite yhteen ja samaan muuttujaan ja milloin taas eri muuttujiin
- muuttujien linkkiytymistapa (linkage)
  - ulkoinen (external): ulommalla tasolla määritellyt
    - Yhteiskäyttöisiä, usean tiedoston sisällä viittaavat aina yhteen ja samaan olioon
  - ei mitään: funktion sisällä määritellyt
    - Linkittäjän kannalta aina eri olioita
  - sisäinen (internal): const-määritellyt muuttujat, rakenteiset tyypit (struct, union, enum)
    - Yhden tiedoston sisällä viittavat aina samaan olioon, eri tiedostoissa eri olioihin
- static ja extern muuttavat linkkiytymistapaa
  - static muuttaa ulkoisen sisäiseksi
  - extern muuttaa ulkoiseksi

Globaalit muuttujat

Javassa ei ole globaaleja muuttujia!

- Ohjelmassa funktion ulkopuolella määritellyt muuttujat ovat globaaleja
  - koko ohjelmassa käytettävissä määrittelynsä jälkeen
  - elinikä sama kuin koko ohjelmalla
  - alustetaan nollaksi
- C:ssä tulee käyttää hyvin harkiten ja pyrkiä välttämään käyttämistä
  - Funktiokutsut eivät saa muuttaa globaaleja muuttujia
    - Seuraa vaikeuksia: testaus, virheiden jäljitys, ylläpito
    - Dokumentoitava huolelisesti
- Määriteltävä yhdessä käännösyksikössä ja liitettävä .h –tiedostoon (extern int globalvar)

Käännösmoduulit



Kooditiedosto (.c)

- Sisältää yhteenkuuluvien, jollakin tavoin samaan kokonaisuuteen liittyvien funktioiden koodin
  - Kukin funktio suorittaa jonkun tietyn toiminnon esim.
    - syöttötietojen käsittelyn
    - tulostukset
    - varsinaisen tietojen käsittelyn ja muokkaamisen:
      - laskenta, lajittelu, jne
    - virheidenkäsittelyn
    - Jne
- Yhdessä kooditiedostossa main
- Muiden moduulien käyttöön tarkoitettu informaatio kootaan otsaketiedostoksi
  - Pyritään pitämään suhteellisen pienenä
- Eri moduulit suhteellisen riippumattomiksi toisistaan

Vastaa Javan rajapintaa (interface)

Otsaketiedosto

Header files usually ONLY contain definitions of data types, function prototypes and C preprocessor commands.

- Sisältää:
  - funktioiden esittelyt (prototyypit)
  - Makrot
  - vakioiden määrittelyt (const)
  - Dokumentoinnin: kaikki mitä käyttäjän tarvitsee tietää osatakseen käyttää
    - Ei mitään pelkästään toteutukseen liittyvää
- esikääntäjä liittää otsaketiedoston moduuliin
  - #include <stdio.h>
    - standardikirjaston funktion prototyypit moduuliin
  - #include "oma.h"
    - Samassa hakemistossa olevan oman otsaketiedoston liittäminen

makefile

- Ohjelmassa on useita moduuleja. Kukin moduuli eli käännösyksikkö omassa tiedostossaan, kirjastofunktiot omassa tiedostossaan
- Käännösyksiköt käännetään erikseen ja linkitetään sitten yhteen
  - gcc –c main.c tekee objektitiedoston main.o
    gcc –c mod1.c => mod1.o
    gcc –c mod2.c => mod2.0
  - gcc –o ohjelma main.o mod1.o mod2.o
    linkittää objektitiedostot suoritettavaksi ohjelmaksi


```
/* main.c */
#include <stdio.h>
#include  "mod1.h"
#include "mod2.h"
int main(void) {
   mod1();
   mod2();
   return 1;
}
```

```
/* mod1.c */
#include <stdio.h>
#include "mod1.h"
void mod1(void){
   .........
  puts("moduuli yksi");
  ..........
}
```

```
/* mod2.c */
#include <stdio.h>
#include "mod2.h"
void mod2(void){
   .........
  puts("moduuli kaksi");
  ..........
}
```

```
/*mod1.h */
void  mod1(void);
```

```
/*mod2.h */
void  mod2(void);
```

```
gcc –c main.c
gcc –c mod1.c
gcc –c mod2.c
gcc –o ohjelma main.o mod1.o mod2.o
```

Moduulien kääntäminen – make

- Kääntämiskomennot ja ohjeet kirjoitetaan säännöiksi makefile tai Makefile nimiseen tiedostoon
  - kohde: tarvittavat tiedostot
    komento1
    komento2
    ....
    komentoz
  - Huom. Komentojen sisennykseen käytetään tabulaattorimerkkiä (tabulointia), ei välilyöntejä.

makefile:n laatiminen

Eräs skriptikieli, kuten mm
-Job Control Languages
-Unix Shell
-QuakeC, AWK, perl, ...

```
gcc –c main.c
gcc –c mod1.c
gcc –c mod2.c
gcc –o ohjelma main.o mod1.o mod2.o
```

Makefile kirjoitetaan vain kerran, mutta sitä voidaan käyttää monta kertaa komennolla make

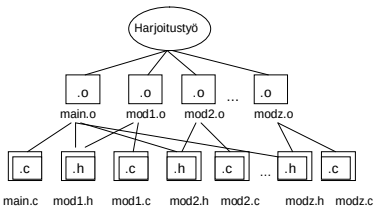
- käännetään vain ne tiedostot, joita on muutettu
- linkitetään uudelleen vain, jos jokin linkitettävistä objektitiedostoista on muuttunut

kommenttirivi

```
#makefile
CC = gcc –ansi –pedantic –Wall
ohjelma: main.o mod1.o mod2.o
        $(CC) –o ohjelma main. mod1.o mod2.o
mod1.o: mod1.c mod1.h
        $(CC) –c mod1.c
mod2.o: mod2.c mod2.h
        $(CC) –c mod2.c
main.o: main.c mod1.h mod2.h
        $(CC) –c main.c
```

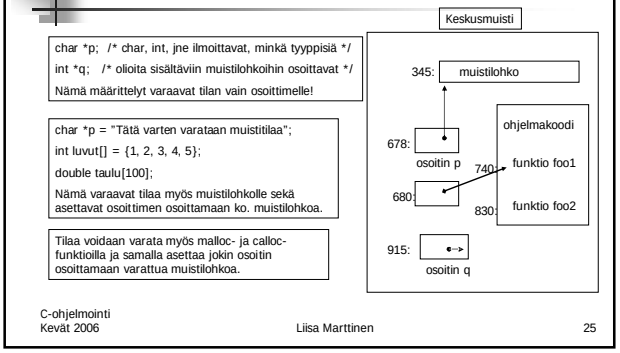
http://www.cs.helsinki.fi/group/sqltr/makefile_ohje.html
http://www.eng.hawaii.edu/Tutor/Make/index.html
http://vertigo.hsrl.rutgers.edu/ug/make_help.html

riippuvuusgraafit



- ” Modular programming is a life saver, you can save an immense amount of time, if you do it right”

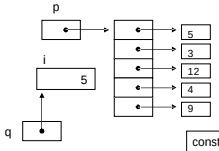
Lyhyt kertaus osoittimista



Osoittimien käyttöä

- int **p;
  int *q,r;
  int i;

  i= **p



q = &i; /* i:n osoite q:n arvoksi
i = *q+1; i = ++*q; /* i=6*/
i = *q++; /* ???? */
r = q; *r = 3; /* i=3 */

const int *p;
int const *p;
const int const *p;

void *p; i= *(int*) p;

char viesti [] = "On aika"; viesti: On aika\0
char *pv ="On aika"; On aika\0
pv

Osoitin parametrina

- C:ssä vain arvoparametreja => funktio käyttää vain kopioita eikö voi mitenkään muuttaa saamiensa parametrien arvoja:

```
void swap(int x, int y) {
  int apu;
  apu=x;
  x=y;
  y= apu;
}
```

kopioita

x:n osoite, y:n osoite

```
void swap(int *x, int *y) {
    int apu;
    apu=*x;
    *x=*y;
    *y= apu;
}
```

Kutsu: swap (&x, &y);

x 3
y 4

double product (const double block, int size);

Varmistaa, ettei funktio muuta viiteparametrina saamaansa lohkoa

Java: "overriding"

C: funktio-osoittimet

- Esimerkki: funktio voi suorituksen aikana vaihtaa käyttämäänsä lajittelualgoritmia alkioiden lukumäärän perusteella

int (*fp) (void);
Osoitin int-tyypin palauttavaan funktiooon

int *fp()
Funktio, joka palauttaa int-tyyppisen osoittimen!

```
int fname(); /* kunhan funktio on oikean tyyppinen */
fp = fname; /* fp() merkitsee nyt samaa kuin fname() */
```

- Funktion parametrina funktio

```
void qsort(*line[], int left, int right, int (*comp)(void *, void*));
```



```
void main (void) {
int choice;  double x, fx;
funcptr fp;
.....
funcprt function[7] = {NULL, sin, cos, tan, log , log_2, exp}; /*määriteltyjä funktioita*/
/*  funktiomenun tulostus: käyttäjää valitsee   haluamansa vaihtoehdon  */
 ....
scanf ("%i", &choice);
/* lisäksi tarkistetaan, että valinta on sallittu arvo */
...
if (choice ==0) break;
printf("Enter x: "); scanf/("%lg", &x);
fp = function[choice];
fx = fp(x);
printf("\n  (%g) = %g\n", x, fx);
}
}
```

typedef double (*funcptr) (double);

function [0] [1] [2] [3] [4] [5] [6]
NULL sin cos tan log log_2 exp

Kurssikokeesta

- 2.5. klo 16-19
  - Ylimääräinen koetilaisuus 8.5. klo 9-12 niille, jotka hyvästä syystä ovat estyneet osallistumasta 2.5. pidettävään kokeeseen
    - Ilmoita Liisalle (liisa.marttinen@cs.helsinki.fi), jos olet tulossa tähän kokeeseen!
  - Näihin kokeisiin saa osallistua vain, jos on jättänyt harjoitustyönsä tarkastettavaksi 24.4. mennessä
- Kokeessa saa olla mukana A4:n kokoinen lunttilappu

Vähän kokeesta: mitä pitää osata

- Aiempien kurssin asioista perusohjelmointi, algoritmien kirjoittaminen, tietorakenteiden käyttö(taulukko, lista, pino) ja niiden tavanomaiset käsittelyrutiinit (lisääminen, poisto ja järjestäminen)
- C-kielen syntaksi ja semantiikka. Kirjastorutiinien käyttöä ei sinänsä edellytetä, mutta erimerkiksi merkkijonojen ja tiedostojen käsittelyyn käytettävät tavanomaiset funktiot on syytä hallita.
- Kielen rakenteista on hyvä hallita ainakin:
  - Funktioiden ja niiden parametrien käyttö
  - Taulukko
  - Tekstitiedosto
  - Linkitetty tietorakenne ja osoittimet
  - Komentoriviparametrit
  - Merkkijonot

Lisää kokeesta

- Näistä osattavista asioista muodostetaan sitten kokeessa erilaisia yhdistelmiä eri tehtävissä.
- Esimerkiksi toukokuun 2005 kokeessa oli
  - tehtävässä 1:
    - funktion käyttöä ja tietojen lukemista käyttäjältä (eli tiedostosta stdin),
  - tehtävässä 2:
    - funktio, osoittimia ja linkitetyn listan kopiointi ja järjestäminen
  - tehtävässä 3:
    - Tekstitiedosto, komentoriviparametri ja taulukko
- Koska koeaikaa on vain 2,5 tuntia, niin kolmeen tehtävään vastaaminen edellyttää jonkinlaista rutiinia C-ohjelmien kirjoittamisessa
  - Tätä rutiinia on saatu jo Viopen harjoituksissa, mutta etenkin kurssin harjoitustehtäviä tehdessä!
  - Lunttilappu helpottaa asioiden muistamista!