T-106.3101 Ohjelmoinnin jatkokurssi T2 (6 op)
Luento 6: Lisää taulukoita. Komentoriviparametrien käsittely. Linkitetty lista. Funktio-osoittimia.

Sami Liedes

5. helmikuuta 2008

Tänään luvassa

- Kaksiulotteisia taulukoita
- ”Puolidynaamisia” taulukoita (`int *taulu[10]`)
- Dynaamisia kaksiulotteisia taulukoita
- Komentoriviparametrien käsittely
- Linkitetty lista
- Funktio-osoittimia

Kaksiulotteiset taulukot 1/2

- `int taulu[9][10]`
- Yhtenäinen muistialue, kuten 9×10 yksiulotteinen taulukko
- Muistissa ensin on `taulu[0]`:n 9 alkiota, niiden perässä `taulu[1]`:n 9 alkiota jne.
- Tyyppejä:

Lauseke	*Tyyppi*
`taulu`	`int (*)[10]`
`taulu[0]`	`int *`
`*taulu`	`int *`

- Sisäisesti kaikki nämä ovat osoittimia taulukon ensimmäiseen alkioon

Kaksiulotteiset taulukot 2/2

- Jotta voidaan antaa funktion parametriksi, on jälkimmäinen dimensio (sarakkeiden määrä) tunnettava käännösaikana
- Sarakkeiden määrää tarvitaan sen laskemiseen, monenteenko alkioon muistialueessa oikeastaan viitataan
- Parametrina funktiolle:

```
void fun(int (*taulu)[10], int rivien_maara);
```

tai yhtäpitävästi

```
void fun(int taulu[][10], int rivien_maara);
```

- Jos sarakkeiden määrä tunnetaan käännösaikana mutta rivien määrää ei, voidaan tällainen taulukko toki varata myös dynaamisesti:

```
int (*taulu)[10] = malloc(rivit*sizeof(int[10]));
```

"Puolidynaamiset" taulukot 1/3

- Jos taas rivien määrä tunnetaan käännösaikana, mutta sarakkeiden määrää ei, voidaan käyttää osoitintaulukkoja:

```
int *taulu[10]; /* Huom. eri kuin int (*taulu)[10] */
```

- Yksiulotteinen taulukko `int`-osoittimia
- Tyyppi `int **`
- Osoittimia ei ole vielä alustettu eikä niille ole varattu tilaa.
- `taulu[0]` ei osoita 2D-taulukon ensimmäiseen alkioon, vaan ensimmäiseen riviin (eli rivien taulukon ensimmäiseen alkioon)

"Puolidynaamiset" taulukot 2/3

- Muistin varaaminen:

```
1  int *taulu[10], i;
2  for (i=0; i<10; i++)
3      taulu[i] = malloc(rivin_koko*sizeof(int));
```

- Vapauttamiseen tarvitaan tietysti yhtä monta free()-kutsua (vastaavanlainen silmukka)
- Samassa taulukossa voi jopa olla eri mittaisia rivejä!

"Puolidynaamiset" taulukot 3/3

- Muisti voidaan toki varata myös yhtenä muistialueena:

```
1  int *taulu[10], i; /* 10 riviä */
2  /* rivin pituus = sarakkeiden määrä */
3  int *muistia = malloc(sizeof(int)*10*sarakkeita);
4  for (i=0; i<10; i++) {
5      /* taulu on tyyppiä int **, joten taulu[0] on int * -tyyppinen */
6      taulu[i] = muistia;
7      /* kasvatetaan 'muistia'-osoitinta rivin pituuden verran: */
8      muistia += sarakkeita;
9  }
10 ...
11 free(taulu[0]); /* taulu[0] = malloc()in palauttama osoitin */
```

- Tällöin tietysti yksi `free()`-kutsu riittää

Dynaamiset kaksiulotteiset taulukot 1/2

- `int **`, eli osoittimia int-osoittimiin
- Näitähän nähtiin jo edellä, eikä kyse nytkään ole mistään sen kummemmasta
- Ainoa ero edelliseen: Myös osoitintaulukko (edellä `int *[10]`) varataan dynaamisesti

```
int i, **taulu = malloc(sizeof(int *)*rivien_maara);
for (i=0; i<rivien_maara; i++)
    taulu[i] = malloc(sarakkeita*sizeof(int));
```

- Huomaa: `taulu` voidaan vapauttaa tietysti vasta sen sisältämien rivien vapauttamisen jälkeen

Dynaamiset kaksiulotteiset taulukot 2/2

- Tämäkin voidaan tietysti tehdä yhtenäisellä muistialueella:

```
1  int i, **taulu = malloc(sizeof(int *)*rivien_maara);
2  int *muistia = malloc(sizeof(int)*rivien_maara*sarakkeita);
3  for (i=0; i<rivien_maara; i++) {
4      taulu[i] = muistia;
5      muistia += sarakkeita;
6  }
7  ...
8  free(taulu[0]); free(taulu); /* huom. järjestys */
```

- `malloc()`in parametrien kanssa on syytä olla tarkkana
  - Esim. rivillä 1 olisi helppo sanoa vahingossa `malloc(sizeof(int)*rivien_maara)`
  - Tämä toimii jos `sizeof(int)` $\geq$ `sizeof(int *)`
  - x86-alustalla tämä yleensä pätee, mutta esim. 64-bittisillä Inteleillä tyypillisesti ei $\longrightarrow$ aikapommibugi

Vaihtoehto: Makrot tai apufunktiot

- Usein on helpompaa käyttää yksiulotteista taulukkoa ja määritellä makrot tai apufunktiot sen käsittelyyn:

```
int taulu_priv[riveja*sarakkeita];
#define TAULU(x,y) (taulu_priv[(x)+(y)*sarakkeita])
int alkio(int x, int y);
void aseta_alkio(int x, int y, int uusi_arvo);
```

- Apufunktiossa voi halutessaan myös tarkistaa x:n ja y:n arvojen laillisuuden
- Lisäksi näin piilotetaan itse toteutus muulta koodilta: Toteutustapaa voi muuttaa muuttamatta taulukkoa käyttävää koodia

Komentoriviparametrit

- `main()`-funktio voi ottaa joko 0 tai 2 parametria
- `int main(int argc, char **argv);`
- Nimet `argc` ja `argv` ovat konventio
- `argc` = parametrien määrä (ml. ohjelman nimi)
- `argv` = itse parametrit

`argv[0]`	=	ohjelman nimi (jos `argc` > 0)
`argv[1]`	=	ensimmäinen parametri
	⋮	
`argv[argc-1]`	=	viimeinen parametri
`argv[argc]`	=	`NULL`

- `argv` on taulukko jonka alkiot ovat merkkijonoja

Linkitetty lista

- Sovellus osoittimille
- Käytetään hyväksi sitä, että tietueet voivat sisältää osoittimia itsensä tyyppisiin tietueisiin
  - Ei sinänsä eroa Javasta tässä suhteessa
- Yhteen suuntaan linkitetty lista:

```
1 typedef struct solmu {
2     int arvo;
3     struct solmu *next;
4 } Solmu;
5 Solmu *lista;
```

- Viimeisen solmun `next` on `NULL`, jotta tiedetään listan loppuneen
- Huom. tyhjä linkitetty lista on siis pelkkä `NULL`

Uuden solmun varaaminen ja vapauttaminen

```
1  typedef struct solmu {
2      int arvo;
3      struct solmu *next;
4  } Solmu;
5
6  Solmu *uusi_solmu(int arvo) {
7      Solmu *s = malloc(sizeof(Solmu));
8      s->arvo = arvo;
9      return s;
10 }
11
12 void vapauta_solmu(Solmu *s) {
13     free(s);
14 }
```

Solmun lisääminen ja poistaminen 1/2

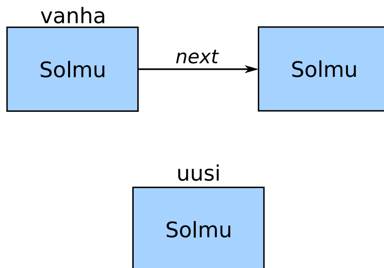
- Solmun lisääminen:

```
1 /* Lisää uuden solmun listaan solmun 'vanha' jälkeen */
2 void lisaa_jalkeen(Solmu *uusi, Solmu *vanha) {
3     /* uuden solmun seuraaja on vanhan vanha seuraaja */
4     uusi->next = vanha->next;
5     vanha->next = uusi;
6 }
```

- Solmun poistaminen: Vastaava idea
  - Solmujen osoittimet kuntoon
  - Mieti erikoistapaukset: Tyhjä lista, listan alku, listan loppu
  - Jos ei muuten aukene: Solmu on laatikko, osoitin seuraavaan (edelliseen) on nuoli. Piirrä paperille ja mieti.
- Kahteen suuntaan linkitetyt listat vastaavasti

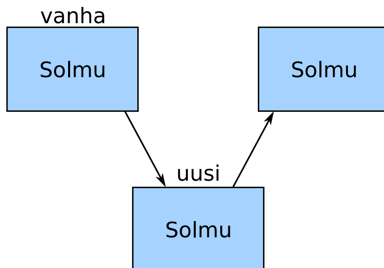
Solmun lisääminen ja poistaminen 2/2

```
1  /* Lisää uuden solmun listaan solmun 'vanha' jälkeen */
2  void lisaa_jalkeen(Solmu *uusi, Solmu *vanha) {
3      /* uuden solmun seuraaja on vanhan vanha seuraaja */
4      uusi->next = vanha->next;
5      vanha->next = uusi;
6  }
```



Solmun lisääminen ja poistaminen 2/2

```
1  /* Lisää uuden solmun listaan solmun 'vanha' jälkeen */
2  void lisaa_jalkeen(Solmu *uusi, Solmu *vanha) {
3      /* uuden solmun seuraaja on vanhan vanha seuraaja */
4      uusi->next = vanha->next;
5      vanha->next = uusi;
6  }
```



Pino linkitettynä listana

- Linkitetyn listan avulla on helppoa toteuttaa mm. pino
  - Yhteen suuntaan linkitetty lista, jonka ensimmäinen alkio on pinossa päällimmäisenä oleva alkio
  - push() lisää listan alkuun alkion
  - pop() poistaa ja palauttaa päällimmäisen alkion
  - Muista vapauttaa poistettujen solmujen muisti
- Myöskään monimutkaisempia rakenteita (puut, verkot ym.) ei ole paljon sen vaikeampi toteuttaa
- Vaatii kuitenkin vähän harjoittelua

Funktio-osoittimia

- Joskus on tarpeen kutsua funktiota joka tiedetään vasta ajonaikana
- Esimerkki: Standardikirjaston lajittelurutiini

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

- Lajittelee taulukon, joka alkaa osoitteesta `base` ja jossa on `nmemb` alkiota joiden kunkin koko on `size`
- `compar` on osoitin vertailufunktioon, joka palauttaa negatiivisen luvun, nollan tai positiivisen luvun sen mukaan, kumpi annetusta kahdesta alkiosta on suurempi
- Koska `qsort()` ei tiedä alkioiden tyyppiä, `compar` saa osoittimet niihin `void *` -tyyppisinä
  → yleensä vertailuun tarvitaan tyyppimuunnosta

Syntaksi

- Funktio-osoittimien määrittelyn syntaksi:

```
int (*ptr)(char param1, long *param2);
```

- Tässä `int` on funktion paluuarvo ja `paramN` ovat parametreja.
- Muuttujaan `ptr` voidaan sijoittaa minkä tahansa sellaisen funktion osoite, jonka prototyyppi on muotoa

```
int funktio(char a, long *p);
```

- Funktion nimi on samalla funktion osoite:

```
ptr = funktio; /* sijoitetaan em. funktion osoite osoittimeen ptr */
```

- Funktio-osoittimen läpi kutsutaan aivan kuin se olisi itse funktio:

```
ptr('z', NULL);
```

Käyttötarkoituksia

- Kun halutaan toteuttaa geneeristä toiminnallisuutta
- Olio-ohjelmointi C:llä
  - Esim. siten, että jokainen olio (käytännössä tietue) tietää, minkä (ali)luokan edustaja on
  - Metodien ylikuormitus voidaan toteuttaa funktio-osoittimilla
  - Mahdollista, mutta C ei ainakaan auta siinä yhtään
  - Toisinaan silti hyvä ratkaisu, jos esim. jostain syystä joudutaan toteuttamaan käyttöliittymäkoodia C:llä
- `qsort()` ja vastaavat tapaukset

Loppukurssin aikatauluista

- Seuraava luento tiistaina 12.2.
  - Unix-työkalujen käytöstä (Riku Saikkonen)
- Torstai 21.2.
  - Yksittäisten bittien käsittelyä
  - Projektityön esittely (Kenneth Oksanen)
- Torstai 28.2.
  - Ryhmätyön Best Practices ja versionhallinta (Jari Vanhanen)
- Harjoitustehtäväkierrokset
- Projektityö
- Tarkempaa tietoa kurssin kotisivuilta