HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

C-ohjelmointi: Tietorakenteita ja tyyppejä

Liisa Marttinen & Tiina Niklander
22.3.2005

Rakenne

1. **Tyyppimuunnoksista**
2. Bittioperaatiot
3. Moduulirakenne
4. Näkyvyyssäännöt
5. Funktio-osoittimien käyttö
6. Assert makron käyttö
7. Kirjastorutiineista

Tyyppimuunnokset

- Laskutoimituksissa tehdään automaattisia tyyppimuunnoksia
- Tyyppimuunnoksen voi myös tehdä eksplisiittisesti

```
int i;
float f;
void *g;

…

i = i+ (int)f;
f = f*i;
g = &i;
*(int*)g=435;
```

Tyyppimuunnoksia aritmeettisten tyyppien välillä

Jos lausekkeen operandit ovat erityyppisiä, niin c:ssä tehdään automaattinen tyyppimuunnos laskutoimitusta varten aritmeettisten tyyppien välillä. Aina pienemmän tarkkuuden omaava tyyppi muunnetaan tarkemmaksi tyypiksi seuraavan järjestyksen mukaisesti:

- **`int`**
- **`unsigned`**
- **`long`**
- **`unsigned long`**
- **`float`**
- **`double`**
- **`long double`**

Bittioperaatiot

- & | ^ vertaavat alkioita bitti kerrallaan ja palauttavat tämän tuloksen
- << >> siirtävät bittejä sanan sisällä
- ~ yhden komplementti 0-> 1 ja 1-> 0

& bitwise and
| bitwise or
^ bitwise xor (exclusive or)
<< left shift
>> right shift
~ one's complement

low-order byte

& x	…		B
0xff	0…0	0…0	11111111
x & 0xff	0 … 0		B

`&` (bitti and)

- jos $(x)_i$ == 1 ja $(y)_i$ == 1 niin $(x \,\&\, y)_i$ == 1
- muuten $(x \,\&\, y)_i$ == 0
- Tätä käytetään usein bittien nollaukseen, esim:
- **`x & 0xff`** nollaa sanan kaikki muut bitit paitsi vähiten merkitsevän tavun (low-order byte):

low-order byte

& x	…		B
0xff	0…0	0…0	11111111
x & 0xff	0 … 0		B

| (bitti or)

- jos $(x)_i == 1$ *tai* $(y)_i == 1$ niin $(x \mid y)_i == 1$
- muuten $(x \mid y)_i == 0$
- Tällä usein asetetaan tiettyjä bittejä ykkösiksi.
- Asetetaan x:n alimman tavun ylin bitti ykköseksi: **`x | 0x80`**,

low-order byte

x	C	$B_1B_2B_3\ldots$
|		
0x80	0…0	10000000
x | 0x80	C	$1B_2B_3\ldots$

^ (bitti xor)

- jos $(x)_i == (y)_i$ niin $(x \wedge y)_i == 0$
- muuten $(x \wedge y)_i == 1$

- Bittioperaatio ^ (*xor*, poissulkeva or) nollaa ne bittipositiot, joiden arvo on sama ,
- ja asettaa ykköseksi ne bittipositiot, jotka olivat eriarvoiset.
- Yksinkertainen testi sanojen samuudelle: **`x ^ y`**
- palauttaa 0 jos **`x`** ja **`y`** samat (eli kaikki bitit vastasivat toisiaan).

~ (yhden komplementti)

- **Yhden komplementti**: **`~x`**

  $(\sim x)_i == 1 - (x)_i$

- Bitit vaihtuvat vastakkaisiksi.
- Esimerkiksi. Halutaan vain nollata sanan **`x`** 3 alinta bittiä ja jättää loput ennalleen: **`x & ~7`**
  (Niin miten tuo oikein toimii?)
- Vastaavasti, kun halutaan asettaa kaikki bitit ykkösiksi **`~0`**

<< Siirto vasemmalle

- **Left shift**: **`i << j`**

- Tuloksena on sana, jossa bitit ovat siirtyneet vasemmalle **`j`** paikkaa
- Sana on täydennetty oikealta tarvittavalla määrällä nollia.

- Tämä on näppärä kahden kertolaskuissa:

- **`x <<= 1`** on sama kuin **`x *= 2`**

- **`x <<= 3`** on sama kuin **`x *= 8`**.

>> Siirto oikealle

- **Right shift**: **`i >> j`**

- Tuloksena on sana, jossa bitit ovat siirtyneet **`j`** paikkaa oikealle.
- Etumerkittömillä tyypeillä (ja positiivisilla luvuilla) oikealta täydennetään aina nollilla.
- Etumerkillisillä ja negatiivisilla luvuilla täydennyksenä voi olla 0 tai 1 toteutuksesta riippuen. (MIKSI?)

- **`x >>= 1`** is equivalent to **`x /= 2`**

- **`x >>= 2`** is equivalent to **`x /= 4`**.

Esimerkki:

```
#include <stdio.h>
/* Bittipeliä*/
int main(void)
{
 enum {LL = 011 };
 int i, j;

 i = 0;
 j = i | LL;
 printf("i: %d, LL (okt):%o, i|LL: %d, oktaalina %o\n",
          i, LL, j, j);

 printf("1 & 6: %d, 1 && 6: %d\n",
          1 & 6, 1 && 6);

 printf("1<<3: %d, 8>>3: %d\n",
          1<<3, 8>>3);
return 0;
}
```

Tulostaa:
i: 0, LL (okt):11, i|LL: 9, oktaalina 11
1 & 6: 0, 1 && 6: 1
1<<3: 8, 8>>3: 1

Rakenne

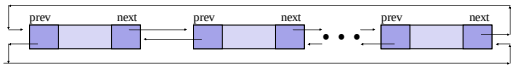
1. Tyyppimuunnoksista
2. Bittioperaatiot
3. **Moduulirakenne**
4. Näkyvyyssäännöt
5. Funktio-osoittimien käyttö
6. Assert makron käyttö
7. Kirjastorutiineista

Moduulit

- C ei suoraan tue modulaarista (tai rakenteista) ohjelmointitapaa
  - ei sisäkkäisiä funktioita
  - globaalit muuttujat näkyvät saman tiedoston funktioille
- C:ssä on **ohjelmoijan** kuitenkin mahdollista käsitellä tiedostoja siten, että
  - yhteen kuuluvat elementit sijoitetaan samaan tiedostoon
  - yhden abstraktin tietotyypin toteutus yhteen tiedostoon
- ”Moduulia voi jopa ajatella luokan kaltaisena elementtinä”
- Esim. tiedostot lista.h ja lista.c
  - lista.h –tietorakenteiden kuvaus ja funktioiden prototyypit
  - lista.c –funktioiden toteutukset ja omien muuttujien esittelyt

Esimerkki: LINUXin prosessilista

- Tehokkuuden vuoksi prosessien kuvaajien tiedot tallennetaan kahteen suuntaan linkitettyyn listaan



- Listalle on myös määritelty omat käsittelyrutiinit (funktioita ja makroja)
  - list_add(n,p), list_add_tail(n,h)
  - list_del(p)
  - list_empty(p), list_entry(p,t,f)
  - list_for_each(p,h)
- Prosessilistan omat makrot: SET_LINKS REMOVE_LINKS (kts. www-sivu http://lxr.linux.no/source/include/linux/list.h)

Rakenne

1. Tyyppimuunnoksista
2. Bittioperaatiot
3. Moduulirakenne
4. **Näkyvyyssäännöt**
5. Funktio-osoittimien käyttö
6. Assert makron käyttö
7. Kirjastorutiineista

Muuttujien määrittelyt ja näkyvyyssäännöt

```
#include <stdio.h>
/* globaalit muuttujat tänne*/
double tulos;
int main(int argc, char** argv)
{
 /* funktion muuttujat tänne */
 int i;
 for (i=0; i < 20; i++) {
  /* Lohkon muuttujia */
  int j;
  j = i*4;
 }
return 0;
}
```

- Lohkon sisäiset muuttujat
  - määritellään lohkon alussa
  - Funktiokin on lohko
  - Lohko on mikä tahansa aaltosulkujen { } kattama ohjelman osa
  - Eivät näy lohkon ulkopuolelle
  - Arvo ei säily kutsukerrasta toiseen
- Globaalit muuttujat
  - Määritellään funktioiden ulkopuolella
  - Näkyvät myöhemmin määritellyille funktioille
  - Arvo säilyy suorituksen ajan

Poikkeamat näkyvyyssäännöistä: Viittaaminen muualla määriteltyyn – extern

```
int i;
static int si;
extern int ei;
void f() {
  int fi;
  static int sif;
  extern int eif;  /* vältä tätä! */
}
void g();
static void h();
int eif; /* määritelty vasta täällä */
```

- Määrittely on tällöin
  - tyypillisesti jossain toisessa tiedostossa tai kirjastossa
  - Samassa tiedostossa, mutta myöhemmin
- Vältä tämän käyttöä!!
- Pyri määrittelemään muuttujat mahdollisimman paikallisesti, ja ainakin yhden tiedoston sisällä

Poikkeamat säilyvyyssäännöistä: Funktion sisäinen muuttuja pysyväksi - static

```
int i;
static int si;
extern int ei;
void f() {
  int fi;
  static int sif;
  extern int eif;  /* vältä tätä! */
}
void g();
static void h();
int eif; /* määritelty vasta täällä */
```

- Määrittelemällä sisäinen muuttuja 'static' määreellä muuttujan arvo säily suorituskerrasta toiseen.
- Toisaalta 'static' määre globaalin muuttujan tai funktion nimen edessä rajoittaa kyseisen tunnuksen näkyvyyttä. Sitä ei voi tämän jälkeen käyttää muista tiedostoista käsin. (Vrt. javan private)

Muuttujien sijoittelu muistiin

- **Extern** määre kertoo kääntäjälle, että tässä kohtaa muuttujalle ei tarvitse varata tilaa, koska muuttuja on määritelty muualla
- **Static** määre kertoo kääntäjälle, että paikalliselle muuttujalle on varattava tilaa pinon ulkopuolelta, koska arvon pitää säilyä
- **Register** määre kertoo kääntäjälle, että muuttujaa käytetään niin paljon, että sille kannattaisi varata oma rekisteri prosessorilta tässä lohkossa
- Muuttujan esittely **ilman määrettä**, ns. automaattinen tilanvaraus
  - globaaleille muuttujille tilanvaraus käännösaikana
  - paikallisille muuttujille tilanvaraus pinosta suoritusaikana

Rakenne

1. Tyyppimuunnoksista
2. Bittioperaatiot
3. Moduulirakenne
4. Näkyvyyssäännöt
5. **Funktio-osoittimien käyttö**
6. Assert makron käyttö
7. Kirjastorutiineista

Funktio-osoittimien käyttö yleisissä moduuleissa

- Funktioita voi kutsua useita kertoja
- Funktion paluuarvo ja parametrilista on tyypitetty
- Geneerisissä tietorakenteissa ja niiden käsittelyrutiineissa käytetään usein tyypittömiä parametreja ja paluuarvoja:
  - Yleiskäyttöisempiä rutiineja, mutta
  - Paljon enemmän virhemahdollisuuksia
- Usein alkion sisältöä käsittelevä funktio välitetään funktioparametrina

Etsintä: funktio search

```
/* Search a block of double values */
int search( const double *block, size_t size,
           double value) {
  double *p;

  if(block == NULL)
    return 0;

  for(p = block; p < block+size; p++)
    if(*p == value)
      return 1;

  return 0;
}
```

Osoitin viiteparametrina

Tietorakenteen läpikäynti

Yleistetään tuo etsintäfunktio search

- C ei salli polymorfismia, mutta voimme simuloida sitä käyttämällä geneerisiä osoittimia (i.e. `void*`).
- Funktion prototyyppi voi määritellä että paluuarvo ja parametrit ovatkin määräämätöntä tyyppiä `void`

```
int searchGen(const void *block,
              size_t size, void *value);
  /* Määrittämätön tyyppi – ei riitä*/
  /* Vaan tarvitaan lisää parametreja */

int searchGen(const void *block,
  size_t size, void *value, size_t elSize
  int (*compare)(const void *, const void *));
```

Prototyyppi:

alkioiden lkm

tietorak.

alkion koko

vertailufunktio

Funktion kutsujan toimenpiteet

- Funktion kutsujan täytyy määritellä vertailufunktio, jota voidaan kutsua etsintäfunktiosta (ns. **Call back**)
- Tyyppien kanssa määrittely voisi olla:

```
int comp(const double *x, const double *y) {
  return *x == *y;
}
```

- Tyypittömien parametrien avulla funktio täytyykin määritellä seuraavasti

```
int comp(const void *x, const void *y) {
  return *(double*)x == *(double*)y;
}
```

Esimerkki geneerisen etsinnän käytöstä

```
/* Application of a generic search */
#define SIZE 10
double *b;
double v = 123.6;
int i;
int main (void) {
 if(MALLOC(b, double, SIZE))
   exit(EXIT_FAILURE);
 for(i = 0; i < SIZE; i++) /* initialize */
  if(scanf("%lf", &b[i]) != 1) {
      free(b);
      exit(EXIT_FAILURE);
  }
 printf("%f was %s one of the values\n",
   v, searchGen(b, SIZE, &v, sizeof(double), comp)
          == 1 ? "" : "not");
return 0;  /* tai exit(EXIT_SUCCESS); */
}
```

Geneerisen funktion search toteutus

```
int searchGen(const void *block,
  size_t size, void *value, size_t elSize,
  int (*compare)(const void *, const void *)) {
  void *p;
  if(block == NULL)
    return 0;
  for(p = (void*)block; p< block+size*elsize;
        p = p+elsize)
    if(compare(p, value))
      return 1;
  return 0;
}
```

HUOM: Osoittimen siirrossa on nyt otettava myös alkion koko huomioon!

Rakenne

1. Tyyppimuunnoksista
2. Bittioperaatiot
3. Moduulirakenne
4. Näkyvyyssäännöt
5. Funktio-osoittimien käyttö
6. **Assert makron käyttö**
7. Kirjastorutiineista

Virheiden käsittely ja havaitseminen: assert

- **Ennakkoehto (precondition)** on välttämätön ehto, jonka täytyy olla voimassa toimenpidettä suoritettaessa
- Esimerkiksi taulukolle `x[size]`
  `0 <= i < size` on ennakkoehto viittaukselle `x[i]`
- Standardikirjasto **`assert.h`** sisältää makron
  `assert(int e):` `assert(0 <= i && i < size)`
- Toiminnan määrää makron **`NDEBUG`** määrittely. (no debug)
- *oletusarvoisesti* `assert()` on käytössä ja valvoo ohjelman toimintaa
- Määrittele makro **`NDEBUG`** poistaaksesi nuo tarkistukset ohjelmasta. (joko #define tai kääntäjän parametrina)

Ennakkoehtojen tarkastaminen

- Kutsutaan `assert(i):`
- Vaihtoehto 1: **`NDEBUG`** on määrittelemättä **ja** **`i`**:n arvo on 0:
  - Tulostetaan virheilmoitus (kertoo tiedoston ja rivin) ja ohjelman suoritus keskeytetään kutsumalla **`abort()`**. Virheilmoituksessa on annettu ennakkoehto ja makrojen `__FILE__` ja `__LINE__` arvot.
- Vaihtoehto 2: **`NDEBUG`** on määrittelemättä **ja** **`i`**:n arvo ei ole 0, tai **`NDEBUG`** on määritelty:
  - makron **`assert()`** kutsu ei vaikuta suoritukseen millään tavalla.

Muuta virheiden käsittelystä

- errno – perinteinen tapa raportoida virhetilanteesta
  - muuttujan arvo kertoo virheen tyypin
- stderr – oletustiedosto virheiden tulostusta varten
  - tulee ruudulle, vaikka varsinainen stdout olisikin putkitettu tiedostoon
  - siis virheilmoitukset kannattaa tulostaa
    fprintf(stderr, …)
    jolloin varsinainen tulostus ei sotkeennu näistä!
- EXIT_SUCCESS ja EXIT_FAILURE
  - Ohjelman suorituksen voi päättää mistä tahansa exit-funktiolla, jolle voi antaa parametriksi suorituksen onnistumista kuvaavan arvon

Virheitä ja siirrettävyyttä

- *Ylivuodon* testaus:
  - Älä koskaan tee näin: `i + j > INT_MAX` **(EI!)**
  - Vaan näin: `i > INT_MAX - j`
- Tee bittisiirtoja vain etumerkittömille kokonaisluvuille. Erityisesti vältä negatiivisia lukuja: `x << -2` **(EI!)**

Rakenne

1. Tyyppimuunnoksista
2. Bittioperaatiot
3. Moduulirakenne
4. Näkyvyyssäännöt
5. Funktio-osoittimien käyttö
6. Assert makron käyttö
7. **Kirjastorutiineista**

Kirjastorutiinit

- Kirjastorutiinien otsikkotiedostot ovat usein UNIX-ympäristöissä saatavilla hakemistosta **/usr/include**
- Kirjastorutiinien käyttö edellyttää tuon otsikkotiedoston liittämistä ohjelmaan #include
- Jotkut kirjastot, kuten math.h, pitää ottaa vielä käännösaikana erikseen mukaan sopivalla ohjauksella, matematiikkakirjastolle ohjaus on –lm

Standardinmukaiset otsikkotiedostot (ainakin nämä on kaikissa)

	assert.h	diagnostiikkamakro
➡	ctype.h	erilaisia merkkitekstejä
	errno.h	järjestelmän virhetilanteet, muuttuja errno
	float.h	liukulukujen ominaisuuksia
	limits.h	kokonaislukujen ominaisuuksia
	locale.h	paikallisen merkistön ominaisuuksia
	math.h	matematiikkafunktioita
	setjmp.h	pikahypyt eri kohtiin ohjelmassa
	signal.h	signaalien käsittelyyn tarvittavia funktioita
	stdarg.h	vaihtuvamittaisten parametrijonojen käsittely
	stddef.h	tyyppimäärittelyjä, ei yleensä tarvita erikseen
➡	stdio.h	syöttö ja tulostus
➡	stdlib.h	sekalaista, mm. muunnos ja muistinhallinta
➡	string.h	merkkijonojen käsittely
	time.h	päiväyksen ja kellonajan käsittely

Ohjelman toteutuksen vaiheet

- Määrittely – kuvaa toteutettavan ohjelman toiminnan
- Suunnittelu
  - Moduulit eli tietorakenteet ja niiden käsittelyrutiinit
  - Tiedostorakenne
    - ainakin kukin moduuli omaan tiedostoonsa
- Toteutus
  - Moduuli kerrallaan?
- Testaus (ja virheenjäljitys)
  - Erilaiset testityökalut
  - aputulostukset?
- Käyttöohje
  - Tämän voisi tehdä jo suoraan määrittelystä!!!

Tietorakenteista

- Puut, jonot, verkot, pinot - suurimmalta osin käsitelty TIRAn kurssilla
- Ei kannata keksiä pyörää uudelleen joka kerta, vaan käyttää aiemmin tutkittuja yleisiä rakenteita niille sopivissa paikoissa

Tietorakenteiden suunnitteluperiaatteita:

- Toimivat vaihtoehdot: taulukko, hajautustaulu, linkitetty tietorakenne
- Tiedetäänkö koko ennalta?
  - Staattinen – taulukko on usein toimiva
  - Dynaaminen – linkitetty tietorakenne tai hajautus
- Alkiot käsiteltävä järjestyksessä?
  - Järjestämätön – kaikki käy
  - Järjestetty – taulukko, linkitetty tietorakenne
- Mitä tietoja tarvitaan?
  - Tietue, yksittäisiä arvoja?
- Aiheuttaako sovellusalue muita rajoituksia?
  - Tiedostoja? Teksti vai binääri?