

C-ohjelmointi, kevät 2006: *Funktiot*

Luento 4, ti 7.2.2006
Tiina Niklander

Sisältö

- Yleistä
- Parametrit
- Paluarvo ja sen käyttö
- Ohjelmointityylistä: modulaarisuus
 - Funktion esittelyt otsikkotiedostoon
 - Monen ohjelmatiedoston käyttö
 - Näkyvyysäännöt
- Osoitintyyppi
- Viiteparametrit
- Funktioparametrit

Yleistä

- Funktiot ovat kaikki irrallisia ja samalla tasolla (ei sisäkkäisiä funktioita)
- Ohjelmassa on aina tasan yksi **main**-funktio
 - Suoritus alkaa tästä funktiosta
- Näkyvyyttä voi rajoittaa vain tiedostoittain ja kirjoitusjärjestyksellä
- Funktioilla on vain arvoparametreja
 - Viiteparametrit on toteutettava osoitintyyppin avulla
 - Funktio voi saada parametrinaan myös toisen funktion osoitteen
- Funktioiden syntaksi kuten Javan metodit

TITO

Aliohjelma- esimerkki

```

int fA (int x, y)
{
    int z = 5;

    z = x * z + y;
    return (z);
}
...
T = fA (200, R);
                
```

aliohjelman toteutus:

```

retfA EQU -4
parX EQU -3
parY EQU -2
locZ EQU 1

fA    PUSH SP, =0 ; alloc Z
      PUSH SP, R1 ; save R1

      LOAD R1, =5; init Z
      STORE R1, locZ (FP)

      LOAD R1, parX (FP)
      MUL  R1, locZ (FP)
      ADD  R1, parY (FP)
      STORE R1, locZ (FP)
      STORE R1, retfA (FP)
      POP  SP, R1; recover R1
      SUB  SP, =1 ; free Z
      EXIT SP, =2 ; 2 param.
                
```

Kaikki viitteet näihin tehdään suhteessa FP:hen

paluarvo
param x
param y
vanha PC
vanha FP
paik. z
vanha R1

FP →
SP →

ks. fA.k91

prolog

epilog

Esimerkki: potenssiin korotus

```
#include <stdio.h>
int power (int m, int n);
int main()
{
    int i;
    for (i=0; i < 20; ++i)
        printf ("%d %d %d \n", i,
            power(2,i), power(-3,i));
    return 0;
}
int power (int base, int n)
{
    int i, p=1;
    for (i=1; i<=n; ++i)
        p = p*base;
    return p;
}
```

- Funktion esittely
 - Tai: int power (int, int);
 - Oltava ennen käyttöä
- Funktion kutsu
 - Parametreille arvot
- Funktion määrittely
 - Voi olla esittelyn yhteydessä tai erikseen
 - HUOM: ei puolipistettä otsikossa

Parametrit

- Aina arvoparametreja (kuten Javan metodeilla)
- Arvo kopioidaan pinon
- Käytetään funktion sisällä parametrin nimellä
- Funktion sisällä parametrit käyttäytyvät kuin paikalliset muuttujat
- Tarvittaessa viiteparametreja on välitettävä muuttujan osoite.
- Erityisellä tyyppillä **void** voi kertoa, jos funktiolla ei ole parametreja

void ja tyyppimuunnokset

- Esittely (declaration):
`int f()` on sama kuin `int f(void)`
- Kutsu (call):
`f();` on sama kuin `(void)f();`
- Parametrien arvot muunnetaan kutsussa tarvittaessa (kunhan tyypit tiedossa!).
- Vastaava muunnos tehdään paluuarvolle.
`int f(int);`
`double x = f(1.2);`

Paluuarvo ja sen käyttö

- Funktion paluuarvo on aina määriteltävä
 - Yksinkertainen tyyppi (ei voi olla rakenne kuten Javassa)
 - Voi olla myös void (eli ei paluuarvoa)
- Kutsuja päättää mitä paluuarvolla tekee
 - Kutsu lausekkeessa -> arvo suoraan käyttöön,
 - sijoitetaan muuttujan arvoksi tai
 - jopa jätetään hyödyntämättä (valitettavasti sallittua!)

exit -funktio

- Ohjelman suorituksen voi missä tahansa ja koska tahansa lopettaa kutsumalla exit-funktiota. `exit(int code);`

```
double f(double x) {
    if(x < 0) {
        fprintf(stderr, "negative x in %s\n",
                __FILE__);
        exit(EXIT_FAILURE); /* no return ... */
    }
    return sqrt(x);
}
```

- Pa
EXIT_FAILURE

programming Guidelines



Funktion dokumentointi

- Esittelyn tai määrittelyn (tai molempien) on syytä kuvata funktion käyttäytyminen:
 - **Function:** nimi
 - **Purpose:** Yleiskuvaus toiminnasta (tyypillisesti kuvaa, miten funktion *oletetaan* toimivan)
 - **Inputs:** Luettelo parametreista ja käytetyistä globaaleista muutt.
 - **Returns:** Paluuarvo
 - **Modifies:** Luettelo parametreista ja globaaleista muuttujista joiden arvo muuttuu toiminnan seurauksena (myös: **sivuvaikutukset!!**)
 - **Error checking:** Funktion tekemät parametrien virhetarkistukset
 - **Sample call:** kutsuesimerkki

Esimerkki

```
/* Funktio:   maxi
 * Tehtävä:   Etsii suuremman
 *             parametriarvon
 * Parametrit: kaksi kokonaislukua
 * Palauttaa: suuremman arvon
 * Muuttaa:   ei mitään
 * Virhetark.: ei ole
 * Kutsuesimerkki: i = maxi(k, 3)
 */
int maxi(int i, int j) {
    return i > j ? i : j;
}
```

Esimerkki: sarjan summa

```
/* Function:  oneOverNseries
 * Purpose:   compute the sum of 1/N series
 * Inputs:    n (parameter)
 * Returns:   the sum of first n elements of
 *            1+ 1/2 + 1/3 + ... 1/n
 * Modifies:  nothing
 * Error checking: returns 0 if n negative
 * Sample call: i = oneOverNseries(100);
 */
double oneOverNseries(int);
```

```
double oneOverNseries(int n) {
    double x;
    int i;

    /* Check boundary conditions */
    if(n <= 0) return 0;

    for(x = 1, i = 1; i < n; i++)
        x += 1/((double)i);

    return x;
}
```

Moduulit ja modulaarinen ohjelmointi

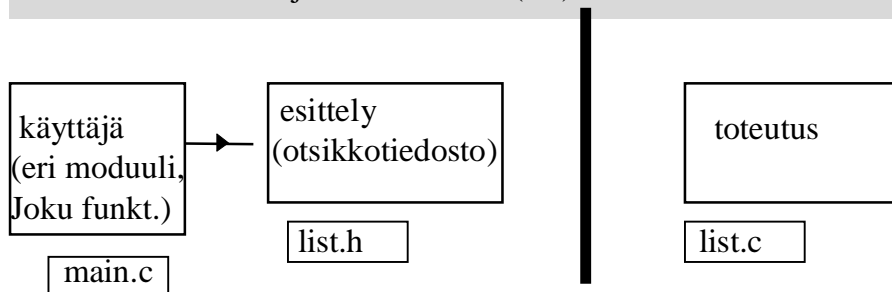
- C ei suoraan tue modulaarista (tai rakenteista) ohjelmointitapaa
 - ei sisäkkäisiä funktioita
 - globaalit muuttujat näkyvät saman tiedoston niille funktioille, joiden esittely tai määrittely on vasta muuttujan määrittelyn jälkeen

Moduulit

- 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 esittelyt
 - lista.c –funktioiden toteutukset ja omien muuttujien esittelyt

Moduulit: esittely vs. toteutus

Moduulilla on aina esittely otsikkotiedostossa (*.h) ja toteutus erillisessä ohjelmatiedostossa (*.c)



Esimerkki: LINUXin prosessilista



- Linuxissa on abstrakti tietorakenne kahteen suuntaan linkitetty lista (käytetään mm. prosessin kuvaajille)
- Listalle on 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>)

Muuttujien määrittelyt ja näkyvyysäännöt

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

Poikkeamat näkyvyysäännöistä: 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 käyttöä!!
- Pyri määrittelemään muuttujat mahdollisimman paikallisesti, ja ainakin yhden tiedoston sisällä

Poikkeamat säilyvyysäännöistä: Funktion sisäinen muuttuja pysyväksi ja piiloon - 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 */
```

- 'static' määreellä muuttujan arvo säilyy suorituskerrasta toiseen.
- Toisaalta 'static' määre rajoittaa kyseisen muuttujan tai funktion (eli 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 lohkoissa
- Muuttujan esittely **ilman määrettä**, ns. automaattinen tilanvaraus
 - globaaleille muuttujille tilanvaraus käännösaikana
 - paikallisille muuttujille tilanvaraus pinosta suoritusaikana

Rekursio

- Rekursiivinen funktio kirjoitetaan c:ssä ihan niin kuin muissakin kielissä
- Rekursion pysäyttämiseksi on rekursiivisen kutsun oltava jollakin tavalla ehdollinen.

```
int digits ( int n) {  
    if ( n <= 0)  
        return 0;  
    if (n/10 == 0)  
        return 1;  
    return 1 + digits (n/10);  
}
```

```
int digits ( int n) {  
    int count =0;  
    if ( n <= 0)  
        return 0;  
    do { count ++;  
        n /= 10;  
    } while ( n!=0);  
    return count;  
}
```

Tauko

10 minuuttia

Osoittimet (pointers)

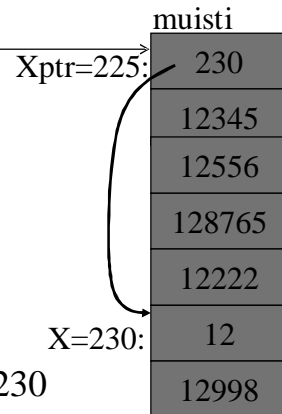
- Osoitin on muuttuja, jonka arvona on toisen muuttujan osoite.
- Osoittimien käyttö perustuu siihen, että useimpien tietokoneiden muisti voidaan kuvata ikään kuin valtavana yksiulotteisena taulukkona. Kaikki ohjelmat ja data sijaitsevat tässä taulukossa.
- Osoitin on oikeastaan indeksi johonkin kohtaan muistia.
- Osoitinmuuttuja on eräänlainen viitta, jonka avulla päästään käsiksi toiseen muuttujaan.
- Vrt. Javan olioviite (joka oikeastaan on osoitin)

Tieto ja sen osoite ⁽³⁾

```

Xptr DC 0
X    DC 12
LOAD R1, =X    ; R1 ← 230
STORE R1, Xptr ; R1 ← 230
LOAD R2, X     ; R2 ← 12
LOAD R3, @Xptr ; R3 ← 12

```



- Muuttujan X osoite on 230
- Muuttujan X arvo on 12
- Osoitinmuuttujan (pointterin) Xptr osoite on 225
- Osoitinmuuttujan Xptr arvo on 230
– jonkun tiedon osoite (nyt X:n osoite)
- Osoitinmuuttujan Xptr osoittaman kokonaisluvun arvo on 12

C-kieli: `Y = *ptrX /* ei ptrX:n arvo, mutta ptrX:n osoittaman muuttujan arvo */`

KOODI ESIMERKKI:

```

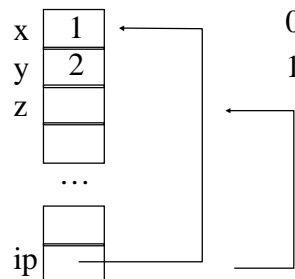
int main(int argc, char**
argv)
{
int x=1, y=2, z[10];
int *ip;
int *p, q;
int *r, *s;

ip = &x;
y = *ip; /* y = x = 1 */
*ip = 0; /* x = 0 */
ip = &z[0];
}

double atof(char *string);

```

HUOM: p on osoitin ja q tavallinen kokonaisluku



Funktion parametrina osoitin on erittäin tavallinen

Viiteparametrit

- Välitetään muuttujan osoite (eli viite), jos funktion pitää muuttaa muuttujan arvoa
- Funktion esittelyssä muodollinen parametri on osoitin
 - void lue_muuttujaan (int *luku);
- Funktion kutsussa todellisena parametrina välitetään muuttujan osoite
 - lue_muuttujaan (&x); (kun int x;)
 - lue_muuttujaan (ptr); (kun int *ptr;)

Viiteparametrin käyttö

- Esimerkkejä: funktion oma paluuarvo on tieto tehtävän onnistumisesta, tulos parametrissa

```
/* Lue korkeintaan n merkkiä
 * Palauta merkin c esiintymisten lukumäärä
 * Paluuarvona onnistumistieto!
 */
int readIt(int n, char c, int *occurrences);

/* Palauta lukujen n ja m summa ja tulo */
int compute(int n, int m,
            int *sum, int *product);
```

Kirjoita
funktio

Vaihteleva parametrimäärä (vrt. printf, scanf)

- Yksi nimetty ja tyypitetty parametri, loput kolme pistettä
- Parametrien käsittely (va_list, va_start, va_arg)

```
/* return a product of double arguments */  
double product(int number, ...) {  
    va_list list;  
    double p;  
    int i;  
    va_start(list, number);  
    for(i = 0, p = 1.0; i < number; i++)  
        p *= va_arg(list, double);  
    va_end(list);  
    return p;  
}
```

Parametrilistan tyyppi

AINA: Haetaan nimetty

Haetaan seuraava

HUOM: Ei autom.
tyyppimuunnoksia

Vaihteleva parametrimäärä

- **OIKEIN:** `product(2, 2.0, 3.0) * product(1, 4.0, 5.0)`

- **VÄÄRIN:** `product(2, 3, 4)`

Miksi?

- Vaihtelevassa parametrilistassa EI voi tehdä automaattista tyyppimuunnosta, kun tyyppi ei ole tiedossa käännoäsaikana!

Funktio-osoittimet

- Vaikka funktiot eivät ole muuttujia, niin niilläkin on osoite muistissa (kts. esim. TTK-91 konekieli). Tähän osoitteeseen voi viitata funktio-osoittimen avulla.
- Osoitin määritellään seuraavasti
paluutyypin (*nimi) (parametrilista);
`int (*lfptr) (char[], int);`
`lfptr = getline;`
/* kun int getline(char s[], int len); */
- Funktio-osoittimen avulla, voidaan eri kutsukerroilla kutsua eri funktiota. Näin voidaan rakentaa yleisiä aliohjelmia. (Esim. `stdlib.h`:ssa on funktio `qsort`, joka saa parametrina järjestysfunktion)

Funktio-osoittimien käyttö parametrina

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

Esimerkki funktiotietue (kts. include/linux/quota.h)

```
231 /* Operations which must be implemented by each quota format */
232 struct quota_format_ops {
233     int (*check_quota_file)(struct super_block *sb, int type);
234     /* Detect whether file is in our format */
235     int (*read_file_info)(struct super_block *sb, int type);
236     /* Read main info about file - called on quotaon() */
237     int (*write_file_info)(struct super_block *sb, int type);
238     /* Write main info about file */
239     int (*free_file_info)(struct super_block *sb, int type);
240     /* Called on quotaoff() */
241     int (*read_dqblk)(struct dqquot *dqquot);
242     /* Read structure for one user */
243     int (*commit_dqblk)(struct dqquot *dqquot);
244     /* Write structure for one user */
245     int (*release_dqblk)(struct dqquot *dqquot);
246     /* Called when last reference to dqquot is being dropped */
247 };
```

Ohjelmointiesimerkki

- Suunnittele ohjelma, joka lukee käyttäjän syöttämän rivin. Rivin ensimmäinen merkki kertoo, mitä rivillä oleville kokonaisluvuille pitää tehdä.
 - S - rivin luvuista pitää etsiä suurin
 - P – etsitään pienin
 - + - lasketaan lukujen summa
 - K – lasketaan keskiarvo

Millainen rakenne

- Luonnollisesti
 - Yhden luvun lukeminen omaan funktioon
 - Pääohjelmaan tuo valinta
- Mutta entä yhden rivin käsittely?
 - Kullekin tyypille oma rivinkäsittelyfunktio ja valinta toimenpiteen mukaan?
 - Geneerinen käsittelyfunktio, joka saa parametrina laskennan?

Millainen rakenne? Verrataan!

- Kullekin oma:

Lue merkki

Switch merkki

'P': pienin();

S: suurin();

+: summa();

K: keskiarvo();

Tulosta arvo;

- Geneerinen toisto:

Lue merkki

Switch merkki

'P' : toistofunktio(pienin)

S: toistofunktio(suurin)

+: toistofunktio(summa)

K : toistofunktio(keskiarvo)

Tulosta arvo;

Tehdään ensin: kullekin oma

- Funktiot: lue_luku, main, pienin(), ...

```
/* tehdään erillinen lukufunktio,
jota voidaan myöhemmin
parantaa */
int lue_luku(int *luku) {
    return scanf("%d", luku);
}
```

```
int pienin() {
    int luku, pieni = INT_MAX;
    while (lue_luku(&luku) != 1) {
        if luku < pieni
            pieni = luku;
    }
    return pieni;
}
```

```
int main () {
    char merkki; int tulos;
    printf("Syötä tietoa yksi rivi\n");
    while ( (merkki=getchar()) != EOF) {
        switch merkki {
            case 'P': tulos = pienin (); break;
            case 'S': tulos = suurin (); break;
            case '+': tulos = summa(); break;
            case 'K': tulos = ka (); break;
            default : break;
        }
        printf ("%c: %d\n", merkki, tulos);
        return 1;
    }
}
```

Tehdään: geneerinen toisto

- Funktiot: lue_luku, main, pienin(), ...

```
int toisto(int (*fun) (int luku)) {
    int luku, tulos;
    while (lue_luku(&luku) != 1)
        tulos = fun (luku);
    return tulos;
}
```

```
int pienin( int luku) {
    static int pieni = INT_MAX;
    if luku < pieni
        pieni = luku;
    return pienin;
}
```

```
int main () {
    char merkki; int tul;
    printf("Syötä tietoa yksi rivi\n");
    while ( (merkki=getchar()) != EOF) {
        switch merkki {
            case 'P': tul = toisto(pienin); break;
            case 'S': tul = toisto(suurin); break;
            case '+': tul = toisto(summa); break;
            case 'K': tul = toisto(ka); break;
            default : break;
        }
        printf ("%c: %d\n", merkki, tul);
        return 1;
    }
}
```

Muutetaan tehtävää

- Entäpä jos pitäisikin lukea useita rivejä?
 - Kumpi ratkaisu soveltuisi tähän paremmin?
 - Miksi?
- Geneerisen toiston saa soveltumaan, kun
 - Static muuttujat pois funktioista (globaaleiksi)
 - Kullekin muuttujalle alustusfunktio lisäksi
 - Yleiskäyttöisyys lisääntyy, mutta
 - Tämä on raskas ratkaisu näin yksinkertaiseen ongelmaan

Muutettu geneerinen toisto

Oma tiedosto

```
/* pieni.h */  
void alustaPieni (int a);  
int Pienin( int luku);
```

```
static int __pieni;  
void alustaPieni (int a) {  
    __pieni = a;  
    return;  
}  
int Pienin( int luku) {  
    if luku < __pieni  
        __pieni = luku;  
    return pienin;  
}
```

```
int main () {  
    char merkki; int tul;  
    /* rivi toisto tähän ympärille*/  
    alustaPieni(-1); alustaSuurin(-INT_MAX);  
    printf("Syötä tietoa yksi rivi\n");  
    while ( (merkki=getchar()) != EOF) {  
        switch merkki {  
            case 'P': tul = toisto(Pienin); break;  
            case 'S': tul = toisto(suurin); break;  
            case '+': tul = toisto(summa); break;  
            case 'K': tul = toisto(ka); break;  
            default : break;  
        }  
        printf ("%c: %d\n", merkki, tul);  
        return 1;  
    }  
}
```

C assumes that programmer is intelligent enough to use all of its constructs wisely, and so few things are forbidden.

C can be a very useful and elegant tool. People often dismiss C, claiming that it is responsible for a "bad coding style". The bad coding style is not the fault of the language, but is controlled (and so caused) by the programmer.