



# C-ohjelmointi, kevät 2006

---

## Taulukot

Yksiulotteiset taulukot

Moniulotteiset taulukot

Dynaamiset taulukot

## Binääritiedostot

Luento 8

21.3.2006



# Luennon sisältö

---

- n Taulukoiden käsittelyä
  - n Yksiulotteiset taulukot
    - n Määrittely
    - n Vertailu
    - n Alustus
    - n Vakiotaulukko
    - n Taulukko parametrina
  - n Moniulotteiset taulukot
  - n Dynaamiset taulukot
- n Binääritiedostot

# Taulukot (arrays) (Müldnerin kirjan luku 10)

## Yksiulotteiset taulukot

- n C:ssä taulukot ovat staattisia (koko tiedettävä ennen kääntämistä), mutta muuten samankaltaisia kuin Javassa.
- n Alkaa aina nollostaa
  - n määrittely: `type arrayName[size];`

```
int luvut[20];  
char *nimet[5*10+1];
```

```
#define SIZE 10  
int taulu1[SIZE];
```

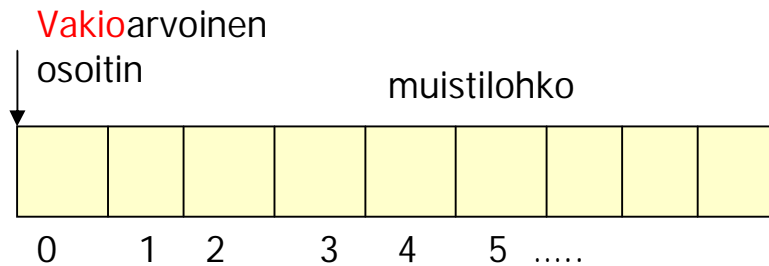
```
const int koko =20; /* vain funktioissa */  
/* int taulu2[koko]; tässä laiton */  
int foo(){  
int taulu2[koko];
```

### Siirrettävyys:

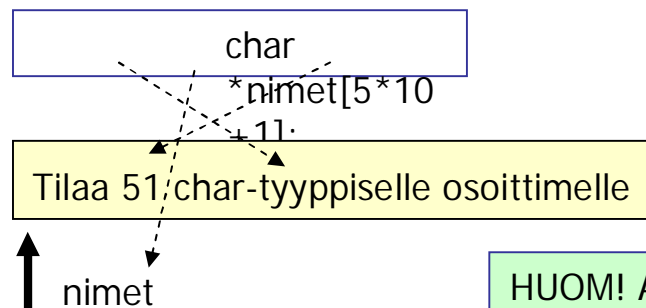
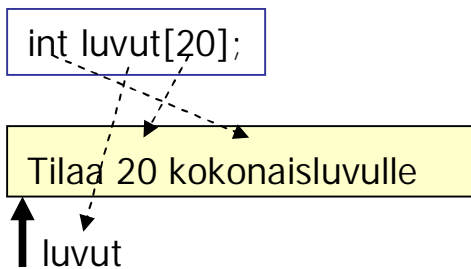
```
warning:  
ISO C90 forbids variable- size  
array 'taulu '  
(ISO C99 sallii käytön, kuten  
myös gcc)
```

# Mikä taulukko on?

- n Taulukko on osoitin
  - n Vakioarvoinen eli osoittaa aina samaan paikkaan
  - n Osoittaa muistilohkoon, josta on varattu tilaa taulukon alkiuille



Muistilohkossa on tilaa taulukon **koon** ilmoittamalle määrälle taulukon **tyypin** kokoisia olioita



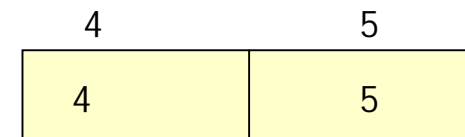
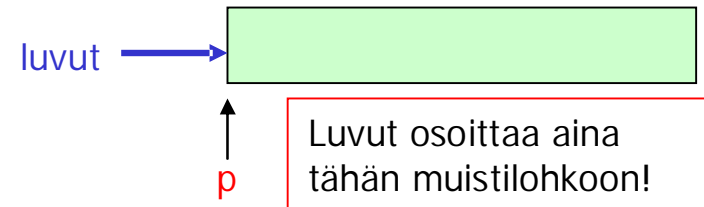
Viittaukset eri alkioihin: luvut[0], luvut[1], luvut[2] .... luvut [19]

HUOM! Ajoaikana järjestelmä ei tarkista indeksien oikeellisuutta. Viittaus luvut[20] ei välttämättä aiheuta suoritusvirhettä, vaan ohjelma vain toimii, miten sattuu toimimaan!

# Yleisiä virheitä määrittelyssä ja käytössä

- n Taulukon koko ilmoitettava vakiona
  - n `int lkm=5; int taulu[lkm]; /*virheellinen */`
- n Taulukko on vakioarvoinen osoitin, jonka arvoa ei saa muuttaa
  - n `int luvut[20];`
  - n `char *p;`
  - n .....
  - n `luvut = p; /*ei näin*/`
  - n `p=luvut; /*ihan OK! Nyt p:kin osoittaa samaan */`
- n Varo sivuvaikutuksia
  - n `a[i] = i++; /* toiminta riippuu toteutuksesta! */`

Kumpi tehdään ensin, kasvatus vai käyttö?  
ensin `a[i]` ja sitten vasta `i++`?  
vai ensin `i++` ja sitten `a[i]`?



Kumpi??



# Taulukko-osoitin ó tavallinen osoitin

n int luvut[20]

- n Vakio
- n Osoittaa aina samaan tietyn kokoiseen muistialueeseen
- n **sizeof(luvut)** on 20 kokonaisluvun tarvitsema tavumäärä eli koko taulukon koko

n int \*osoitin

- n Muuttuja, joka voi osoittaa eri paikkoihin
- n Mitään muistialuetta ei ole varattu
- n **sizeof(osoitin)** on osoittimen tarvitsema tavumäärä



# Kokojen tulostaminen

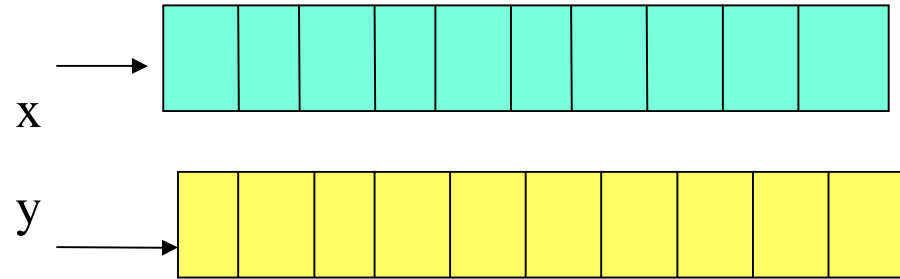
```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    int *taulu[20];
    int *s;
    printf ("Taulukon 'taulu' koko: %d\n", sizeof(taulu) );
    printf ("Osoittimen 's' koko: %d\n", sizeof(s));
    printf ("Taulukon alkion koko: %d\n", sizeof(taulu[0]));
    printf ("Taulukon alkioden lukumäärä: %d\n",
           sizeof(taulu)/sizeof(taulu[0]));
    return 0;
}
```

```
int *taulu[20]; int *s;

printf ("Taulukon 'taulu' koko: %d\n", sizeof(taulu) ); /*=> 80 */
printf ("Osoittimen 's' koko: %d\n", sizeof(s)          /*=> 4 */
printf ("Taulukon alkion koko: %d\n", sizeof(taulu[0])); /*=> 4 */
printf ("Taulukon alkioden lukumäärä: %d\n", sizeof(taulu)/sizeof(taulu[0]); /*=>20*/
```

# Taulukkojen vertailu: sama sisältö?

```
#define SIZE 10  
int x[SIZE];  
int y[SIZE];
```



```
int *px,*py;  
for (px=x, py = y; px<x+SIZE; px++, py++)  
if (*px!=*py) .....
```

Toimii vain jos  
saman kokoisia!  
Entä jos eivät ole?

```
for (px=x; px<x+SIZE; px++)  
if (*px!=*y[px-x]) .....
```

```
for (i=0; i<SIZE; i++ )  
if(x[i] != y[i]) ....
```

Entä vertailu X== Y??  
Mitä tässä verrataan?



# Taulukon alustaminen

- n Taulukko alustetaan antamalla sen alkioiden arvot muodossa

$\{a_1, a_2, \dots, a_n\}$

Taulukon nollaus:

```
int taulu[10000] = {0};
```

Esim.

```
int t[] = {1, 2, 3};
```

1	2	3
---	---	---

Alkioiden lukumäärä määrää taulukon koon!

```
int t[3] = {1, 2, 3};
```

1	2	3
---	---	---

```
int t[3] = {1, 2};
```

1	2	0
---	---	---

Täytetään järjestyksessä.  
Loput nolliä.

```
int t[3] = {1, 2, 3, 4};
```

1	2	3
---	---	---

Kolmen alkion taulukossa ei ole tilaa arvolle '4' => virhe!!

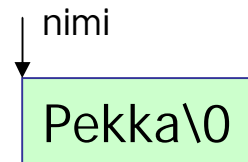
# Vakioksi määritelty taulukko

n `const int days[] = {1, 2, 3, 4, 5, 6, 7}`

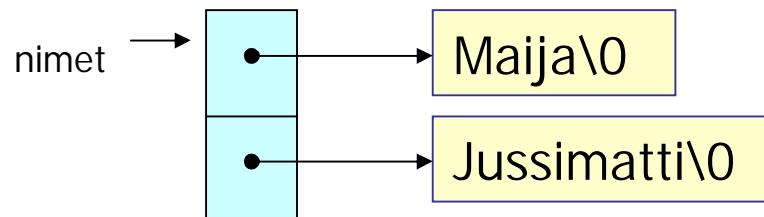
osoittimen days tyyppi: `const int * const`

n Merkkijonovakiot:

`char nimi[] = "Pekka";`



`char *nimet[] = {"Maija", "Jussimatti"};`





# Taulukon kopiointi

```
n for (i=0; i<SIZE; i++) x[i] = y[i];
```

Oltava saman kokoisia!  
Entä jos eivät ole?

```
kokox=sizeof(x);  
kokoy = sizeof(y);  
if (kokox < kokoy) koko = kokox  
else koko = kokoy;  
for (i=0; i<koko; i++) x[i] = y[i];
```

```
koko = kokox < kokoy ? kokox: kokoy;
```



# Taulukko parametrina

---

- n funktion määrittelyssä voi käyttää:

```
int miniT(double arr[], int size);  
int miniP(double *arr, int size);
```

- n Funktion sisällä taulukkoparametreja käsitellään aina osoittimina

=> Taulukon kokoa ei voi kysyä näin!

```
sizeof( arr) (= osoitinmuuttujan koko)
```

# Esimerkki:

`maxmin` palauttaa taulukon maksimi ja minimiarvon

```
int maxmin (double arr[], int size, double *max,  
            double*min) {  
    double *p  
    if (arr==NULL || size <= 0) return 0;  
    for (*max=*min = arr[0], p = arr+1; p<arr + size; p++) {  
        if(*max <*p) *max= *p;  
        if (*min >*p) *min = *p;  
    }  
    return 1;  
}
```

Kutsu: `maxmin(x, SIZE, &max, &min);`

Mitä tekee seuraava kutsu?  
`maxmin(x+3, 5, &max, &min);`

# Paikalliset static-taulukot

```
static char* opnimi(int n) {  
    static char* operaattori [] =  
        {"lvalue", "rvalue", "push", "+", "-"};  
    return operaattori[n];  
}
```

static => funktion  
yksityinen ja  
pysyvä taulukko

Varo  
roikkumaan  
jääviä  
osoittimia!

```
char *setName (int i){  
    char name1[] = "Maija";  
    char name2[] = "Jussi";  
    if (i=0) return name1;  
    return name2;  
}
```

Paikalliselle muuttujalle varataan  
tilaa **pinosta** joka kutsukerralla  
erikseen.

```
char *p = setName(1);
```

Osoitin p jää osoittamaan siihen  
kohtaan pinoa, josta tällä kertaa  
varattiin tilat, mutta joka  
funktion suorituksen jälkeen  
vapautettiin.



# Mielivaltaisen pitkän rivin lukeminen

---

- n Varataan tarpeeksi suuri muistilohko ja toivotaan sen riittävän.
- n Varataan esim. 80 merkin muistilohko ja aina tarvittaessa kasvatetaan riville varatun muistilohkon pituutta (dynaaminen taulukko)
- n Käytetään **rekursiivista funktiota**: aina kun varattu (esim. 80 merkin) muistialue on täynnä ja rivi yhä jatkuu, niin funktio kutsuu itse itseään. Edellisen kutsun muistialue jää talteen pinoon ja uusi suorituskerta varaa uuden muistialueen pinosta.

## Esimerkki: mielivaltaisen pituisen rivin lukeminen rekursiivisesti

```
#define KOKO 80
int luerivi(File *in, char **tulos){
    char puskuri[KOKO];
    int c, i, base; /*luettu merkki, indeksimuuttuja, apumuuttuja */
    static int luettu = 0; /* tämän rivipätkän koko */
    for (i=0; i< KOKO; i++) {
        c = fgetc(in);
        if (c==EOF) { ... } /*virhetilanne */
        if (c== '\n') break; /*rivi loppui*/
        puskuri[i] = c;
        luettu += i;
    }
    if (c!='EOF' && c!='\n'){ /*vielä luettavaa */
        if (luerivi(in, tulos) == 0) { luettu=0; return 0; }
    } else { /* kaikki luettu */
        if ((*tulos = malloc((luettu+1) *sizeof(char))) == NULL) { luettu= 0; return 0;}
        (*tulos)[luettu]='\0'; /* NULL-merkki viimeiseksi */
    }
    base = luettu -i;
    memcpy(*tulos +base, puskuri, i)
    luettu -=i;
    return 1;
}
```

Luetaan korkeintaan 80 merkkiä puskuriin. Joka kutsukerralla eri puskuri, joka jää talteen pinoon.

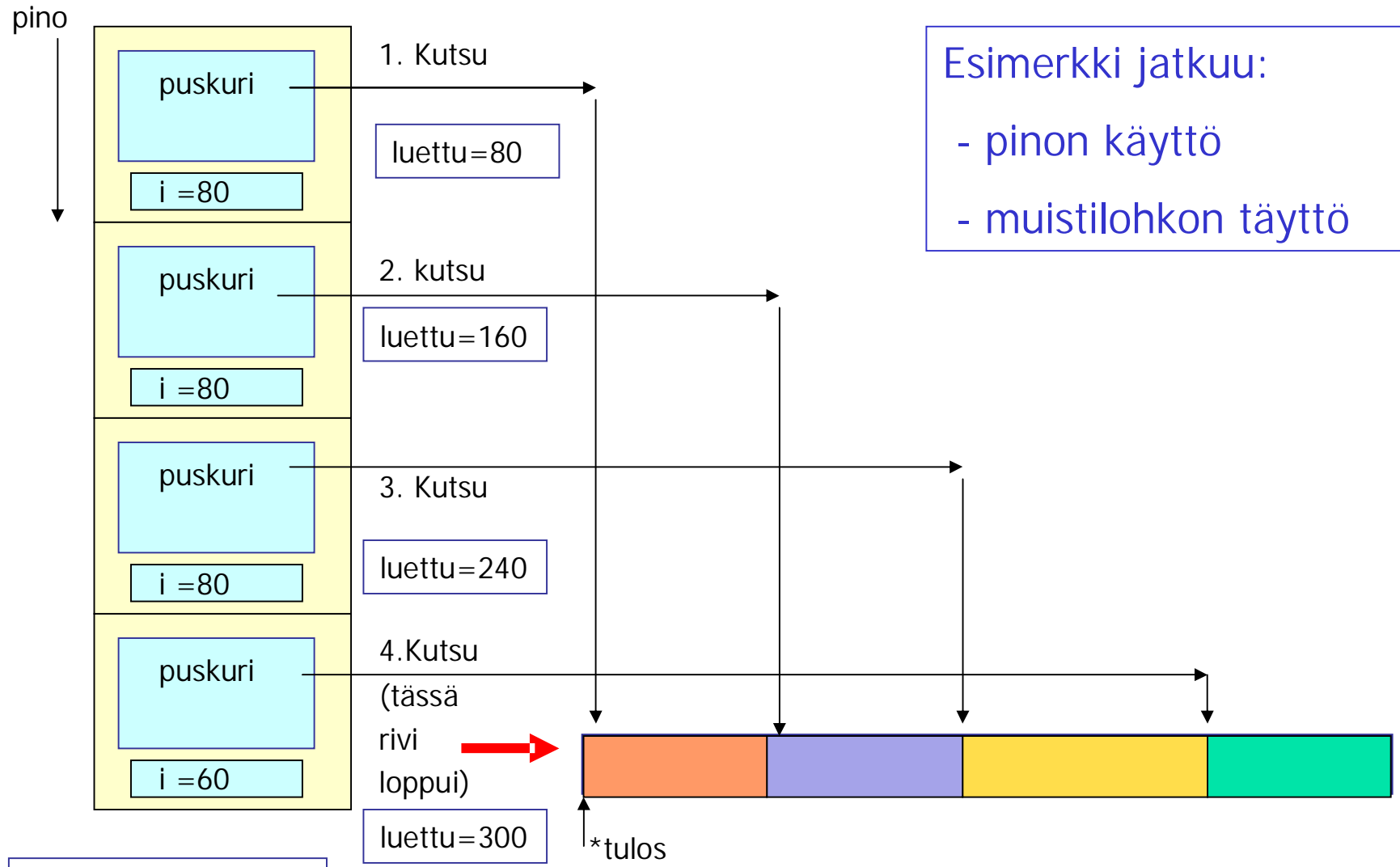
Rekursiivinen kutsu

Kukin kutsukerta kopioi lukemansa puskurin oikeaan paikkaan yhteistä muistitilaa.

Tämä suoritetaan vain kerran: kun viimeinen pätkä riviä on luettu!

Vertaa rekursiivinen kertoman laskeminen tito-kurssilla!





Esimerkki jatkuu:  
- pinon käyttö  
- muistilohkon täyttö

Kun rivin pituus on 300 merkkiä

Kaikilla kutsukerroilla:  
base = luettu - i;  
memcpy(\*tulos + base, puskuri, i)

# Moniulotteinen taulukko

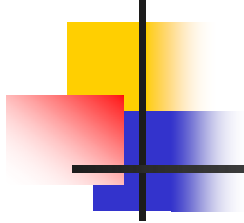
C:n moniulotteiset taulukot ovat yksiulotteisia taulukoita, joiden alkiot ovat taulukoita

```
int t[3][2] = { {1,2}, {11,12}, {21,22} };
```

	0	1
0	1	2
1	11	12
2	21	22

```
for (i=0; i<3; i++) {  
    for (j = 0; j<2; j++)  
        printf ("t[%d][%d] = %d\n",i,j t[i][j]);  
    putchar('\n');
```

```
t[0][0] = 1    t[0][1] = 2  
t[1][0] = 11   t[1][1] = 12  
t[2][0] = 21   t[2][1] = 22
```



```
static char paivat [2][13] = {  
    {0, 31, 28, 31, 30, 31,30,31, 30, 31,30, 31},  
    {0, 31, 29, 31, 30, 31,30,31, 30, 31,30, 31}  
};
```

```
lkm = paivat[karkaus][2];
```

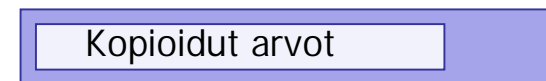
Kun karkaus ==0, niin lkm = 28,  
kun karkaus ==1, niin lkm = 29

# Dynaaminen taulukko

- n Käytetään dynaamista muistin varaamista (`malloc`, `calloc`)
- n Kun taulukolle varattu muistilohko on täynnä eikä siihen enää mahdu alkioita,
  - n Yritetään kasvattaa jo varattua muistilohkoa varaamalla lisää muistia heti sen sen perästä.
  - n Jos tämä ei onnistu, niin varataan jostain muualta muistista suurempi muistilohko, jonne kopioidaan pieneksi käyneen muistilohkon tiedot, ja vapautetaan tämän varaama muistitila.
  - n Voidaan käyttää funktiota `realloc` tai laatia itse funktio, joka varaa tarvittaessa muistia (`malloc`, `calloc`), suorittaa kopioinnin ja vapauttaa (`free`) turhaksi käyneen muistilohkon.

```
void* malloc (size_t koko);  
void* calloc (size_t koko);  
void free( void* pt);  
void* realloc(void* pt, size_t koko);
```

`void` on geneerinen osoitintyyppi  
"a pointer to something, but we don't know yet to what kind of thing"

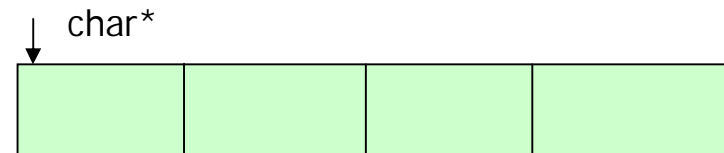


malloc-funktio (void\* malloc (size\_t sz);)

calloc-funktio (void\* calloc(size\_t n, size\_t sz);)

n (char\*) malloc(4\* sizeof(char));

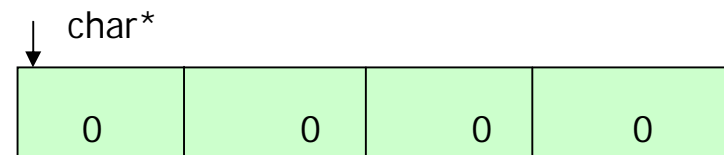
```
if((p = malloc(4* sizeof(char))) == NULL) ..
```



Muistia ei ole alustettu, se sisältää mitä, siihen on sattunut jäämään.

n calloc(4, sizeof(char));

```
if((p = calloc(4, sizeof(char))) == NULL) ..
```



Muisti on nollattu.

# Muistin vapauttaminen: free

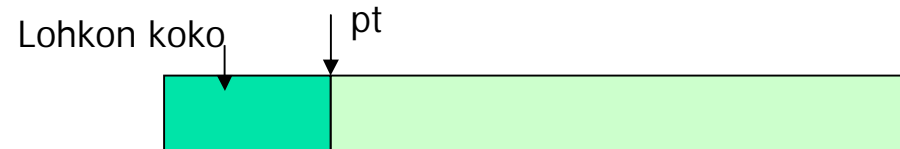
```
(void free( void* pt); )
```

- n Vapauttaa osoittimen osoittaman muistilohkon

- n Mistä se tietää lohkon koon?

- n Lohkoa varattaessa (malloc, calloc) sen koko on talletettu muistiin juuri ennen lohkon alkua

'dangling pointer':  
Jää osoittamaan poistettuun muistialueeseen.



- n Vain calloc:lla ja malloc:lla varattujen alueiden vapauttamisen

- n Vapauta kukin alue vain yhden kerran!

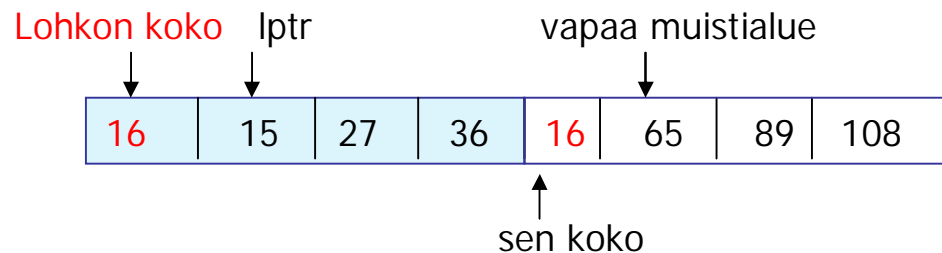
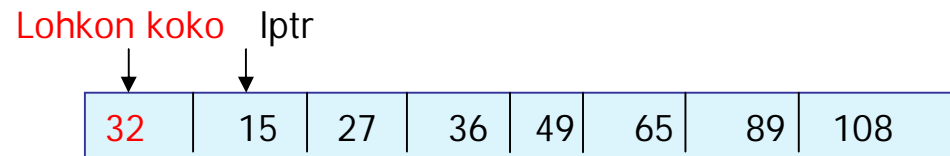
# Taulukon koon muuttaminen:

`realloc` (`void* realloc(void* pt, size_t sz);`)

- n malloc- tai calloc-funktiolla varatun muistilohkon, esim. taulukon, kokoa voidaan sekä suurentaa että pienentää.

Pienentäminen:

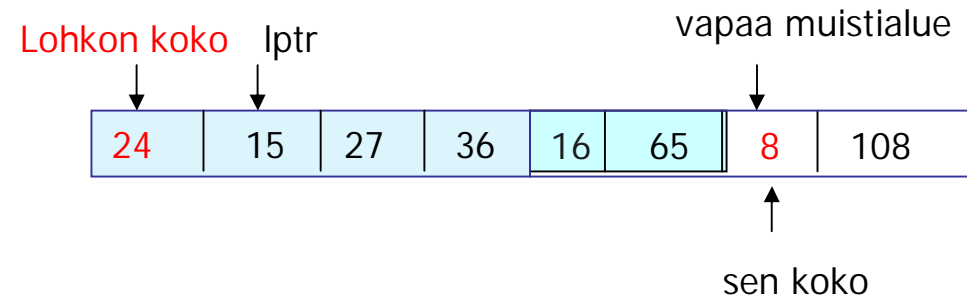
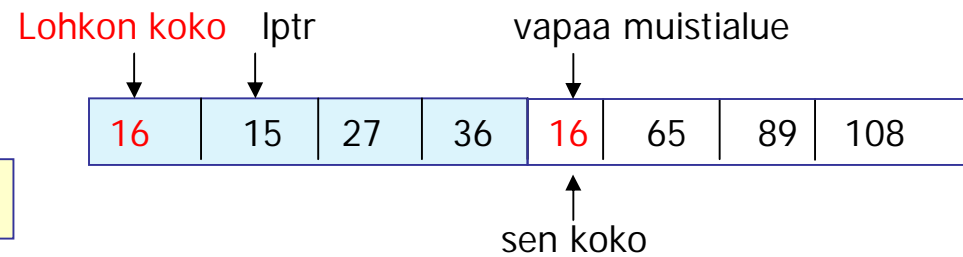
```
lptr = realloc(lptr, 3*sizeof(long));
```



# Varatun muistilohkon koon kasvattaminen realloc-funktiolla (1)

- n Käyttöjärjestelmä pitää kirjaa malloc:lla ja calloc:lla varattujen alueiden koosta ja tietää myös, mitkä alueet ovat vapaana

```
lptr=realloc(lptr, 2*sizeof(long));
```

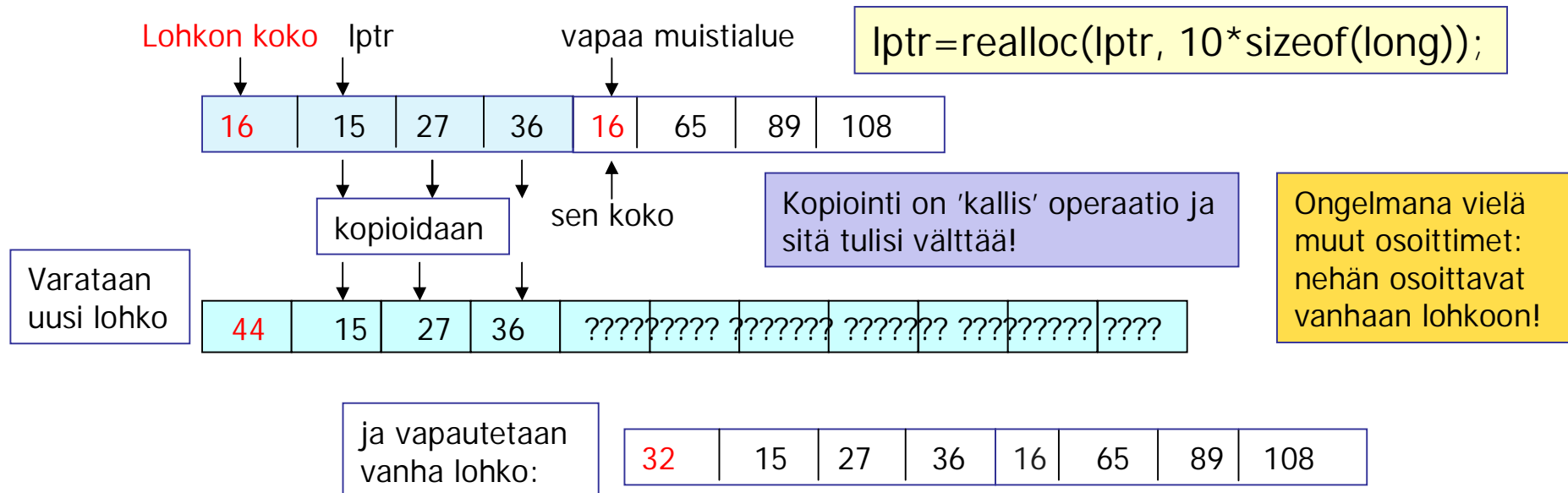




# Varatun muistilohkon koon kasvattaminen realloc-funktiolla (1)

realloc onnistuu aina, kun vain jossain on vapaata muistia tarvittava määrä!

Jos kasvatettavan lohkon perässä ei ole riittävästi vapaata tilaa, niin varataan muistista riittävän suuri vapaa tila, kopioidaan sinne taulukon alkiot ja vapautetaan taulukolle alun perin varattu tila.



# Dynaamisen taulukon käyttö

## n Lajittelussa

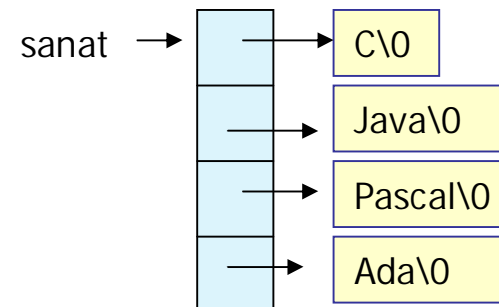
```
printf(" Montako alkiota lajitellaan?\n");
scanf("%i", &n);
if ((alkiot = malloc(n * sizeof(float))) ==
    NULL) mallocerror();
read_file(alkiot, n); /*luetaan alkiot*/
sort_data(alkiot, n);
```

alkiot



## n Eri pituisten merkkijonojen (sanojen tai nimien) tallettaminen taulukkoon

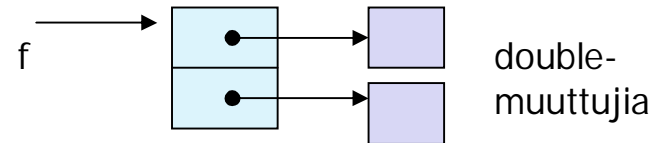
```
scanf("%20", buf); /*sanan lukeminen*/
len = strlen(buf); /*sanan pituus */
if ((sanat[i] = malloc((len+1) *
    sizeof(char))) == NULL) mallocerror();;
strcpy (sanat[i], buf);
```



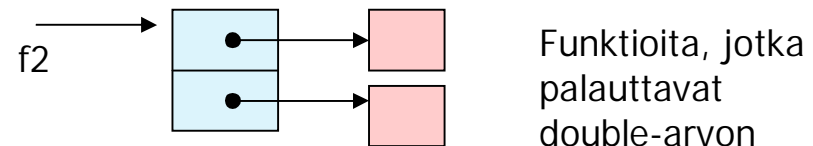
# Monimutkaisia määritelmiä?

n [] -merkeillä korkeampi presedenssi kuin \*-merkillä

`double *f[2];`



`double (*f2[2])()`



`double (*f3())[]`

f3 on funktio, joka palauttaa osoittimen double-taulukkoon.

`double *(f4[])()`

VIRHE! Ei voi olla funktiotaulukkoa!  
Vain osoitteita funktioihin.



# Binääritiedostot

---

- n Tiedon tiiviiseen tallettamiseen
- n Ei rivirakennetta => ei voida käsitellä standardeilla välineillä
- n Eivät suoraan ihmisen luettavissa
- n Usein eivät ole siirrettäviä koneesta toiseen

```
fopen("tied1", "wb");  
fopen("tied2", "rb");
```

# Operaatioita binääritiedostoille

- n hajasaantitiedostoja (random access)
- n Kun tiedosto on avattu, kahva osoittaa sen hetkiseen käsittelykohtaan
- n Operaatioita
  - n `long ftell(FILE *f)` palauttaa käsittelykohdan
  - n `int fseek(FILE *f, long offset, int mode)`
    - siirtää käsittelykohtaa siirtymän (offset) verran
    - mode kertoo mistä kohtaa siirto alkaa:
      - SEEK\_SET tiedoston alusta
      - SEEK\_CUR nykykohdasta
      - SEEK\_END lopusta
  - n `rewind(FILE *f)` kelaat tiedoston alkuun



# Esimerkki:

## Annetun tiedoston koon selvittäminen

---

```
long fileSize (const char *filename) {  
    FILE *f;  
    long size;  
    if ((f = fopen(filename, "rb")) == NULL) return -1L;  
    if (fseek(f, 0L, SEEK_END) == 0) { /* OK!*/  
        size = ftell(f);  
        if (fclose(f) == EOF) return -1L;  
        return size;  
    }  
    fclose(f);  
    return -1L;  
}
```

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

# Binäärinen lukeminen ja kirjoittaminen

= oliolohkojen kirjoittaminen ja lukeminen

n `size_t fread (void *buf, size_t elsize, size_t count, FILE *in);`  
lukee tiedostosta `in` `count`:n ilmoittaman määrän oliolohkoja muistilohkoon, johon `buf` osoittaa. Oliolohkon koon ilmoittaa `elsize`.

Palauttaa luettujen objektien määrän. Virhetilanteessa palauttaa nollan.

n `size_t fwrite (void *buf, size_t elsize, size_t count, FILE *out);`  
Kirjoittaa tiedostoon `out` `count`:n ilmoittaman määrän oliolohkoja `buf`:n osoittamasta muistilohkosta. Oliolohkon koon ilmoittaa `elsize`.

n Olettavat, että tiedosto on jo avattu oikeaa toimintaa (lukemista tai kirjoittamista) varten.



# Tekstiedostosta binääritiedostoon ja binääritiedostosta tekstiedostoon

---

n Teksti => binääri

```
n while (fscanf(in, "%lf", &d) == 1) if (fwrite (&d, sizeof(double), 1,
out) != 1)
```

n Binääri => teksti

```
n while(fread(&d, sizeof(double), 1, in) == 1) {
    i++;
    if (i == Max) {
        putchar('\n');
        i = 0;
    }
    fprintf(out, "%f\t", d);
    .....
}
```





# Ensi kerralla

---

- n Modulaarinen ohjelmointi
- n Kirjastofunktioiden käyttö
- n Kertausta
  - n Mitä asioita tarpeen kerrata?
    - n osoittimia