C-ohjelmointi
Viikko 3: Osoittimet

Sisältö

- Muistin rakenteesta
- Operaatiot ja void-tyyppi
- Muistinhallinta – varaus, vapautus
- Osoitinaritmetiikka ja muistilohkon käsittely
- Osoittimet ja funktiot
- Osoitinlohko
- Merkkijonojen kopiointi (jos ehditään)

Muistinhallinta

Java vs C

- Luokka on viittaustyypin määritelmä ja olio on viittaustyypin ilmentymä
- Muistinhallinta on implisiittistä. Java suoritusympäristö varaa muistia uusille olioille aina oliota luotaessa
- Roskienkeruu vapauttaa muistia eli poistaa muistista ne oliot, joihin ei enää ole viittauksia
- Mikä sitten on olioviite???? (No se on oikeastaan osoitin)

- Ei luokkia, mutta kuitenkin tietorakenteita ja osoittimia
- Eksplisiittinen muistinhallinta. Ohjelman on varattava muistia uusille tietorakenteille.
- Eksplisiittinen vapautus. Ohjelman on vapautettava tarpeeton muisti.
- Osoittimet ovat oikeastaan vain viittauksia muistissa oleviin tietoalkioihin

Prosessin rakenne

KJ-1

- PCB
- Ohjelma-alue
- Data-alue (keko)
- Pino alueen lopussa

- Suhteellisia viittauksia prosessin omalla muistialueella

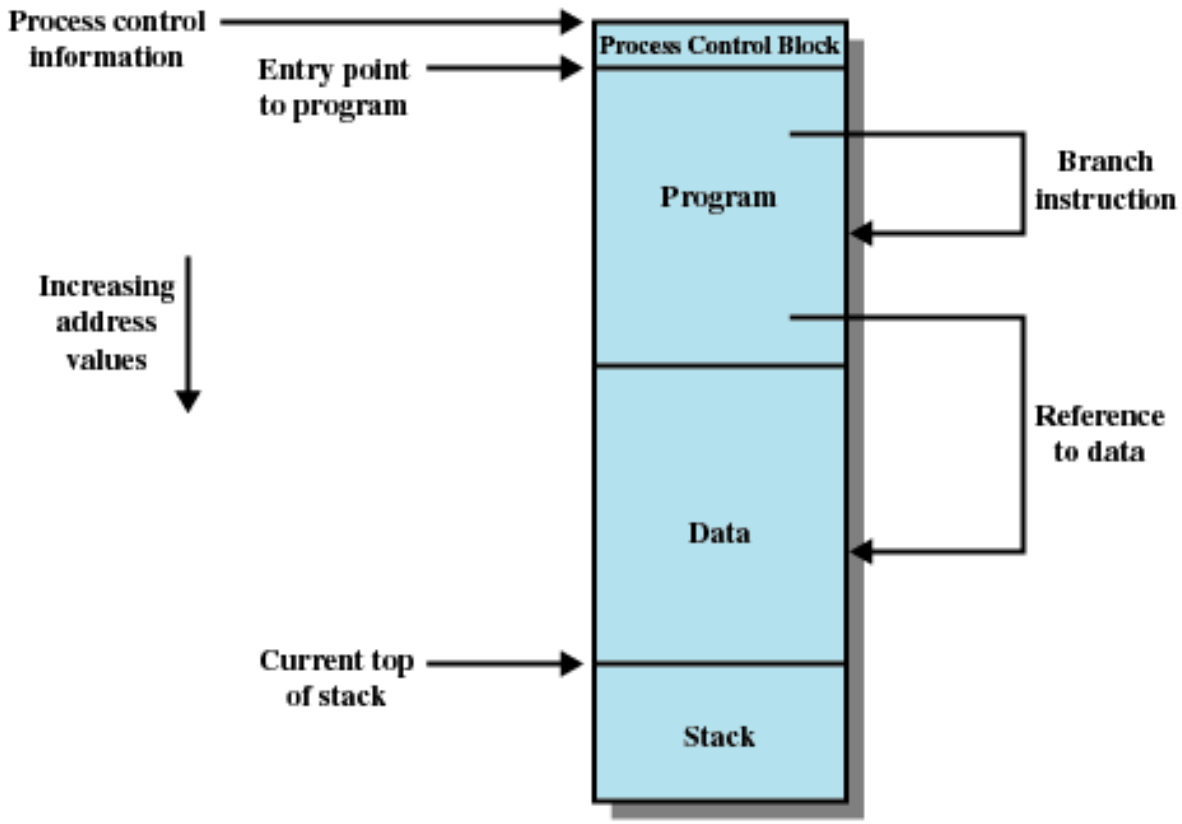
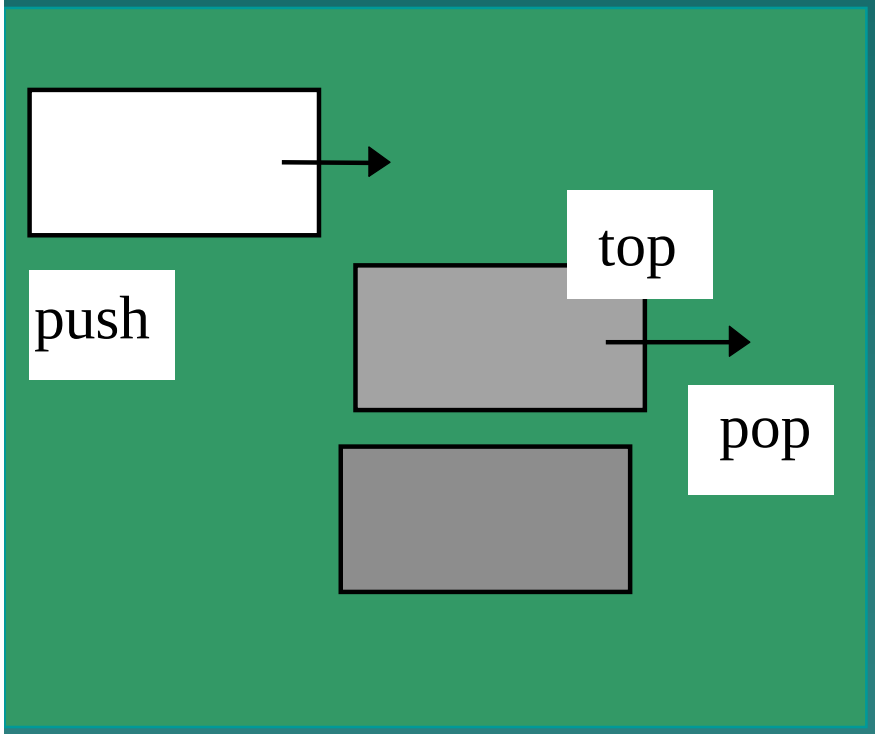


Figure 7.1 Addressing Requirements for a Process

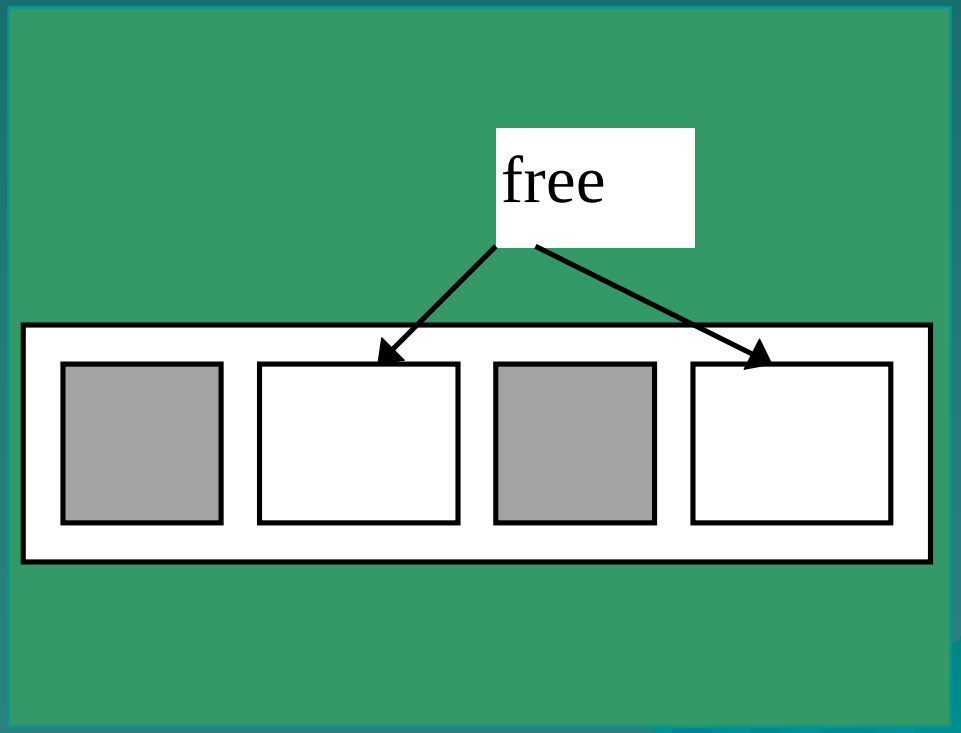
Pino (stack)



- *Implisiittinen käyttö* (paikalliset muuttujat ja funktion param.)
- Hyöty: Muistia ei tarvitse varata tai vapauttaa itse.
- Haitta: Pinon alkioita voi käyttää vain sen hetken kun ne ovat pinossa. (Tästä voi seurata virheitä, jos näihin viitataan osoittimilla.)
- Roikkuva viite (dangling reference)
  - Osoitin, joka vieläkin viittaa jo vapautettuun muistialueeseen

Keko (heap)

- *Eksplisiittinen käyttö*
- Ohjelmoija *vastaa* muistinhallinnasta C:ssä
- Muisti voi pirstoutua (fragmentatoitua)
- Sopimaton tai huolimaton käyttö voi johtaa *muistivuotoon*
- Muisti on resurssi (siinä kuin tiedostotkin) ja sitä pitää hallita ihan yhtälailla



Osoittimien määrittely ja viittaukset

- Osoittimen voi määritellä mille tahansa tyypille

```
int *p;     Osoitin kokonaislukuun
char *q;    Osoitin merkiin  (merkkiosoitin)
double **w;    Osoitin reaalilukuosoittimeen
```

- Tässä siis
  p osoittaa muistialueeseen, jonka koko on **sizeof(int)**
  q osoittaa muistialueeseen, jonka koko on **sizeof(char)**
  w osoittaa muistialueeseen, jonka koko on **sizeof(double*)**

- Tai tyyppimäärittelyllä:

```
typedef int* Pint;
Pint  p1, p2;   /*p1 ja p2 ovat osoittimia int-tyyppiin */
```

Operaatiot

- `p = &c` osoitteen otto
- `c = *p` osoitetun muuttujan arvo
- `c = **r` -”- (Nyt vain kaksi osoitinta peräkkäin)
- `p = q` sijoitus, kun samantyyppiset osoittimet

- `p+i` p osoitin muistialueelle, i sopivan kokoinen kok.luku
- `p-i`
- `p-q` saman muistialueen osoittimia ja q<p

- `*ip++` osoittimen arvo kasvaa yhdellä
- `(*ip)++` osoitetun muuttujan arvo kasvaa yhdellä!

Operaatiot (jatkuu)

- `p < q` p ja q saman muistialueen osoittimia
- `p == q`

- Osoitinaritmetiikka toimii viitattavan tyypin koosta riippumatta:
  - `pa+1` viittaa seuraavaan samantyyppiseen alkioon.
  - `*(pa+1)` on kyseisen alkion arvo samoin kuin `pa[1]`

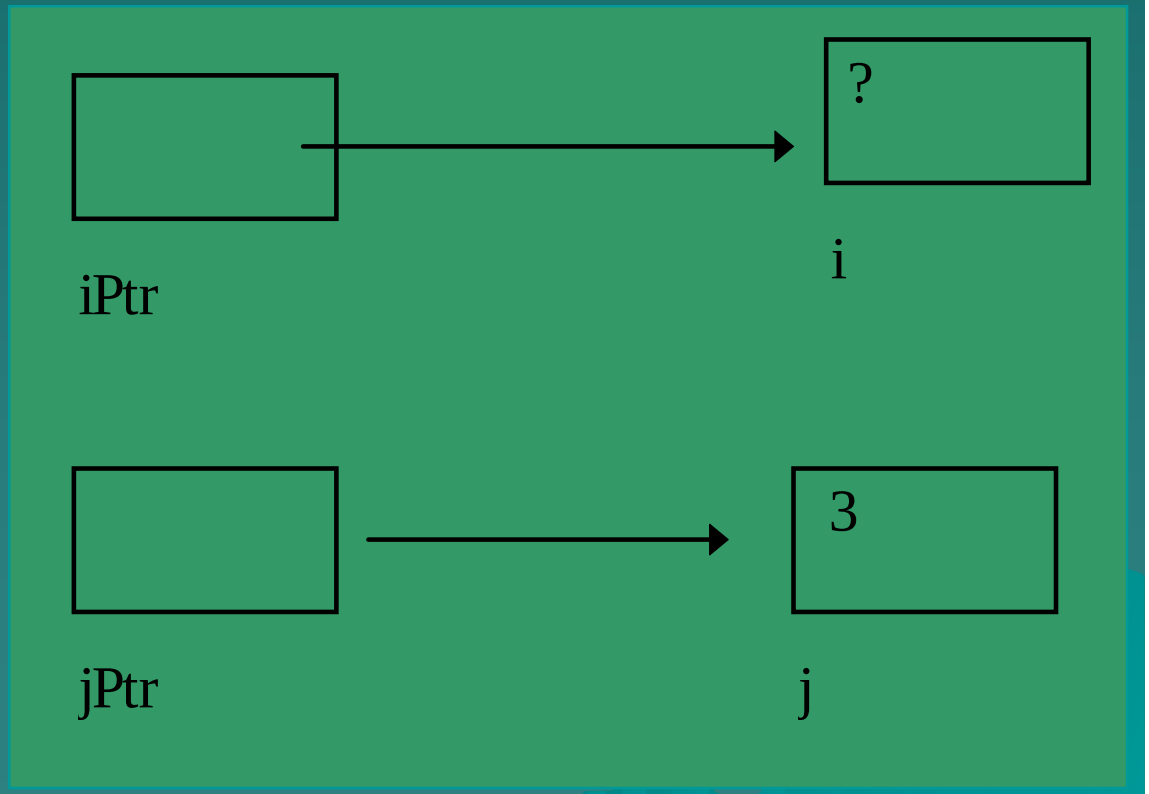
- Osoittimen arvo NULL ei osoita mihinkään
- NULL-arvon voi sijoittaa mille tahansa osoitinmuuttujalle tyypistä riippumatta.

Osoittimet ja sijoitus

```
int i;
int *iPtr = &i;
int j = 3;
int *jPtr = &j;
```

- Entäpä

```
*jPtr = *iPtr;
i = 4;
*jPtr = i;
iPtr = j;
```



Geneeriset osoittimet (void *p)

- **void *p** määrittelee geneerisen, tyypittömän osoittimen **p**.
- Osoitinta voidaan tyyppimuunnoksen ***(T*)p** avulla käyttää käsiteltäessä tyypin **T** muuttujaa

```
void *p = NULL;
int i = 2;
int *ip = &i;

p = ip;
printf("%d", *p);  /* VÄÄRIN?*/
printf("%d", *((int*)p ) );
```

HUOM:
Tyyppimuunnos!

Osoittimet ja **`const`** -määre

- **`const int *p;`**
  - **`p`**:n arvo on osoitin kokonaislukuvakioon
  - **`p`** voi muuttua, mutta **`*p`** ei
- **`int *const p;`**
  - vakio-osoitin kokonaislukuun **`*p`**
  - **`p`** ei voi muuttua, mutta **`*p`** voi
- **`const int *const p;`**
  - Vakio-osoitin kokonaislukuvakioon.

Sisältö

- Muistin rakenteesta
- Operaatiot ja void-tyyppi
- Muistinhallinta – varaus, vapautus
- Osoitinaritmetiikka ja muistilohkon käsittely
- Osoittimet ja funktiot
- Osoitinlohko
- Merkkijonojen kopiointi (jos ehditään)

malloc, calloc ja free

void * on yleinen osoitin, joka voi osoittaa millaiseen rakentee-seen tahansa.

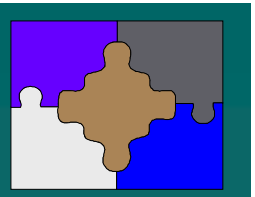
- void *malloc (size_t size);
  - varaa muistia size tavua ja palauttaa osoittimen varatun muistin alkuun. malloc palauttaa NULL, jos muistin varaus ei onnistu.
- void *calloc (size_t nobj, size_t size);
  - * calloc on kuten malloc, mutta varattavan muistin määrä on nobj*size tavua ja muisti nollataan.
- void *realloc (void *p, size_t size);
  - * realloc muuttaa parametrina annetun osoittimen p osoittaman varatun muistialueen kokoa.
- void free (void *p);
  - * free vapauttaa dynaamisesti varatun muistin takaisin käyttöjärjestelmälle.

Paluuarvon tarkistus!

- Hyvään ohjelmointitapaan kuuluu, että aina tarkistetaan funktion palauttama arvo ja toimitaan sen mukaan.
- Erityisen tärkeää tämä on mm. muistin varauksen yhteydessä.

```
void *malloc(size_t requestedSize);
void *calloc(size_t requestedCount,
             size_t requestedSize);
T *p;
if ((p=malloc(n*sizeof(T))) == NULL)
 error;
if ((p=calloc(n, sizeof(T))) == NULL)
 error;
```

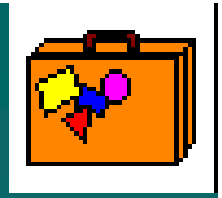
Idioms



Dynaaminen muistinvaraus: n kokonaislukua

```
int* p;
  if((p = malloc(n*sizeof(int))) == NULL)
      error

/* tai makrolla */
#define MALLOC(p, type, n)    \
   (((p) = malloc((n)*sizeof(type))) == NULL)
/* makron käyttö */
  if MALLOC(p, int, n) {error}
```

Portability



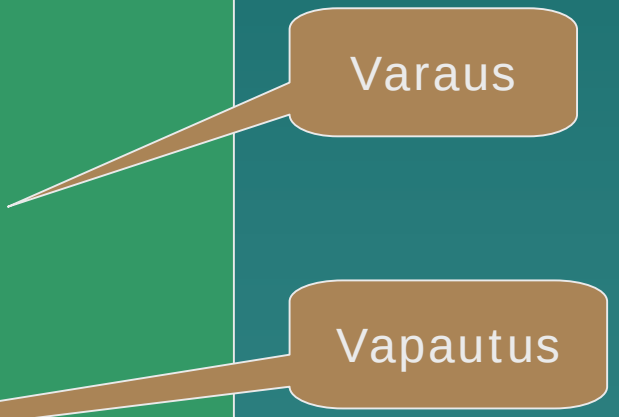
malloc

- Käytä aina absoluuttisen numeerisen arvon sijasta alkion kokona **`sizeof(type)`** varausfunktion kutsussa.
- Esimerkiksi
  `malloc(sizeof(int))`
  on parempi kuin
  `malloc(2)`
- Tyypin koko voi vaihtua ympäristöstä toiseen

Muistin vapauttaminen

- Eksplisiittisesti keosta varattu muisti pitää myös vapauttaa eksplisiittisesti, kun sitä ei enää tarvita.

```
int *p;

if(MALLOC(p, int, 1))
        exit(EXIT_FAILURE);
*p = 12;
    ...
free(p);
p = NULL;
/* Älä enää käytä osoitinta *p */
```



Errors

Muistinhallinta

- Älä sekoita implisiittisesti (pinosta) ja eksplisiittisesti (keosta) varattua muistia.

```
int i;
int *p;
&i = malloc(sizeof(int));  /*VÄÄRIN*/
```

- Vapauta ainoastaan aiemmin eksplisiittisesti malloc tai calloc –kutsuilla varattua muistia. Vapauta vain kerran!

```
p = &i;
free(p);  /* VÄÄRIN */
```

Errors

Muistinhallinta

- Osoittimen arvon sijoittaminen **`q = p`** EI kopioi osoitettua muistia. Sijoituksen jälkeen sekä **`p`** että **`q`** osoittamaan SAMAAN muistialueeseen.

- Näin ollen kutsu **`free(p)`** vapauttaa myös **`q`**:n osoittaman muistialueen. Älä siis käytä q:ta tai vapauta samaan muistialuetta toistamiseen.

Osoitin muistilohkoon

- Osoittimen voidaan katsoa osoittavan yhtenäisen muistialueen (muistilohkon) ensimmäiseen alkioon
- Lohkon sisällä voidaan tiettyyn alkioon i viitata lisäämällä tuo i osoitinmuuttujan arvoon: pa+i
- Alkion arvo saadaan lausekkeella *(pa+i)tai pa[i]
- Älä kadota dynaamisesti varatun lohkon alkua.
  - Pidä siis ainakin yksi osoitin koko ajan lohkon ensimmäisessä alkiossa
- Lohkon läpikäynti

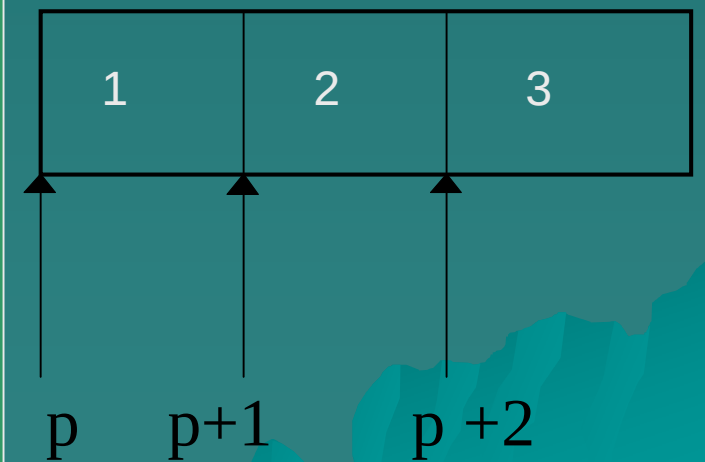
```
for (pi = p;  pi < p+SIZE; pi++) { }
```

Viittaus tiettyyn alkioon

- Kokonaisluvun lisääminen siirtää osoitinta kokonaisen osoitetun alkion verran eteenpäin

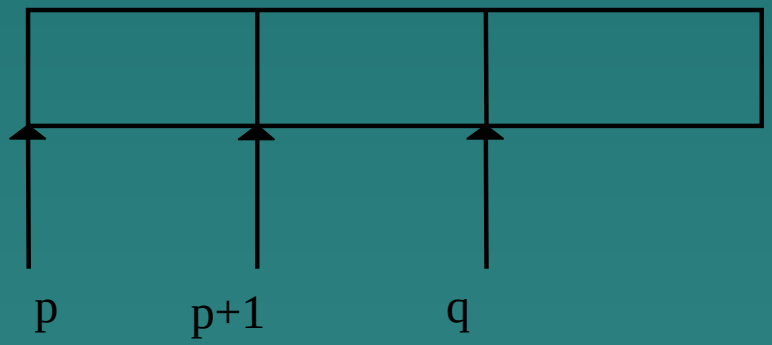
```
#define SIZE 3
 double *p;

 if(MALLOC(p, double, SIZE))
    exit(EXIT_FAILURE);



 *p = 1;
 *(p + 1) = 2; /* tai p[1]*/
 *(p + 2) = 3;
```

Vähennyslasku
Vastaavasti!



Osoittimien välinen erotus

- Osoittimen vähentäminen toisesta **q–p** tuottaa tulokseksi osoittimien etäisyyden muistialueen alkioina. HUOM: oltava **q>=p**




```
int x;

x = q-p;
x = (p+1) –p;
```

Esimerkki: muistilohkon läpikäynti

```
#define SIZE 15
double *p, *pi;
if(MALLOC(p, double, SIZE))
      exit(EXIT_FAILURE);
/* alkioiden sijoittelu muistilohkoon */

for(i = 0, pi = p, sum = 0; i < SIZE; i++, pi++)
      sum = sum + (*pi);
for(pi = p, product = 1; pi < p+SIZE; pi++)
   product *= *pi;
```

Muistilohkon kopiointi alkio kerrallaan

- Kopioidaan osoittimen p osoittama alue osoittimen q osoittamalle alueelle. Alueiden koko on SIZE alkiota

```
/* p:lle on jo aiemmin varattu tila ja
asetettu arvot muistilohkon alkioille */

double *pi, *qi;

 if(MALLOC(q, double, SIZE))
      exit(EXIT_FAILURE);

 for(qi = q, pi = p; qi < q+SIZE; qi++, pi++)
    *qi = *pi;
```

Muistilohkon kopiointi: memcpy ja memmove

- Otsikkotiedostossa **`string.h`** on joidenkin muistilohkoja käsittelevien standardifunktioiden määrittelyjä.
- Esimerkiksi

```
void *memcpy(void *dest, const void *src,
             size_t len);
```

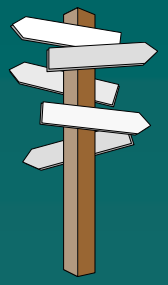
- Kopioi **`len`**-mittaisen muistilohkon osoitteesta **`src`** alkaen lohkoon, joka alkaa osoitteesta **`dest`**
- Jos lohkot ovat osittain päällekkäin on käytettävä funktiota

```
void *memmove(void *dest, const void *src,
              size_t len);
```

Sisältö

- Muistin rakenteesta
- Operaatiot ja void-tyyppi
- Muistinhallinta – varaus, vapautus
- Osoitinaritmetiikka ja muistilohkon käsittely
- Osoittimet ja funktiot
- Osoitinlohko
- Merkkijonojen kopiointi (jos ehditään)

Osoittimet ja funktiot



- Kaikki funktioiden tekemät dynaamiset muistinvaraukset täytyy dokumentoida huolellisesti
- Kutsujan täytyy tietää kenen vastuulla muistin vapauttaminen myöhemmin on

Osoitin paluuarvona

- Funktio varaa muistialueen ja palauttaa arvonaan osoittimen tähän alueeseen
- Kutsuja voi käyttää aluetta ja sen vapauttaminen jää kutsujan vastuulle!!

```
/* funktio: getBlock
 *gets a block of memory
 *to store int values
*/
int* getBlock
     (size_t size) {
  return malloc
   (size*sizeof(int));
}
```

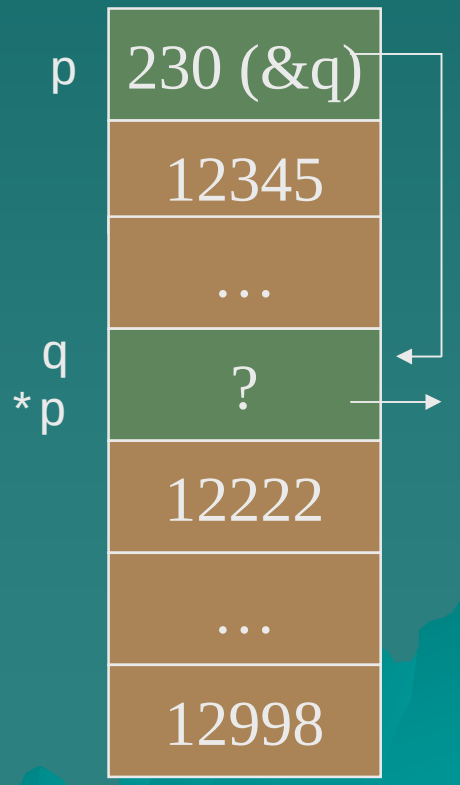
```
int *p;
if((p = getBlock(10))
    == NULL)
      error

/* vapautus joskus
myöhemmin */
free(p);
p = NULL;
```

Osoitinmuuttuja viiteparametrina

- Jos funktion täytyy muuttaa osoitinmuuttujan arvoa, on funktiolle välitettävä osoitinmuuttujan osoite

Osoittimen osoite (osoitinmuuttujan oma osoite)

```
int getBlockRef(int **p, unsigned n) {
 if((*p = (int*)malloc(n*sizeof(int)))
     == NULL)
      return 0;
 return 1;
}



   int *q;
   if(getBlockRef( &q , 10) == 1)
      success
```



Muistilohko parametrina

- Funktiolle parametrina
  - lohkon alun osoite sekä
  - lohkon alkioiden lukumäärä

```
void *funktio(void *ptr, size_t len);
```

- Varattu muistilohko on vain varattu muistilohko
  - Se ei sisällä tietoa koosta tai alkioiden tyypistä
  - Kokotieto täytyy säilyttää ja välittää erikseen
  - Myös tieto tyypistä on erillään lohkosta itsestään

Etsintäfunktio 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;
}
```

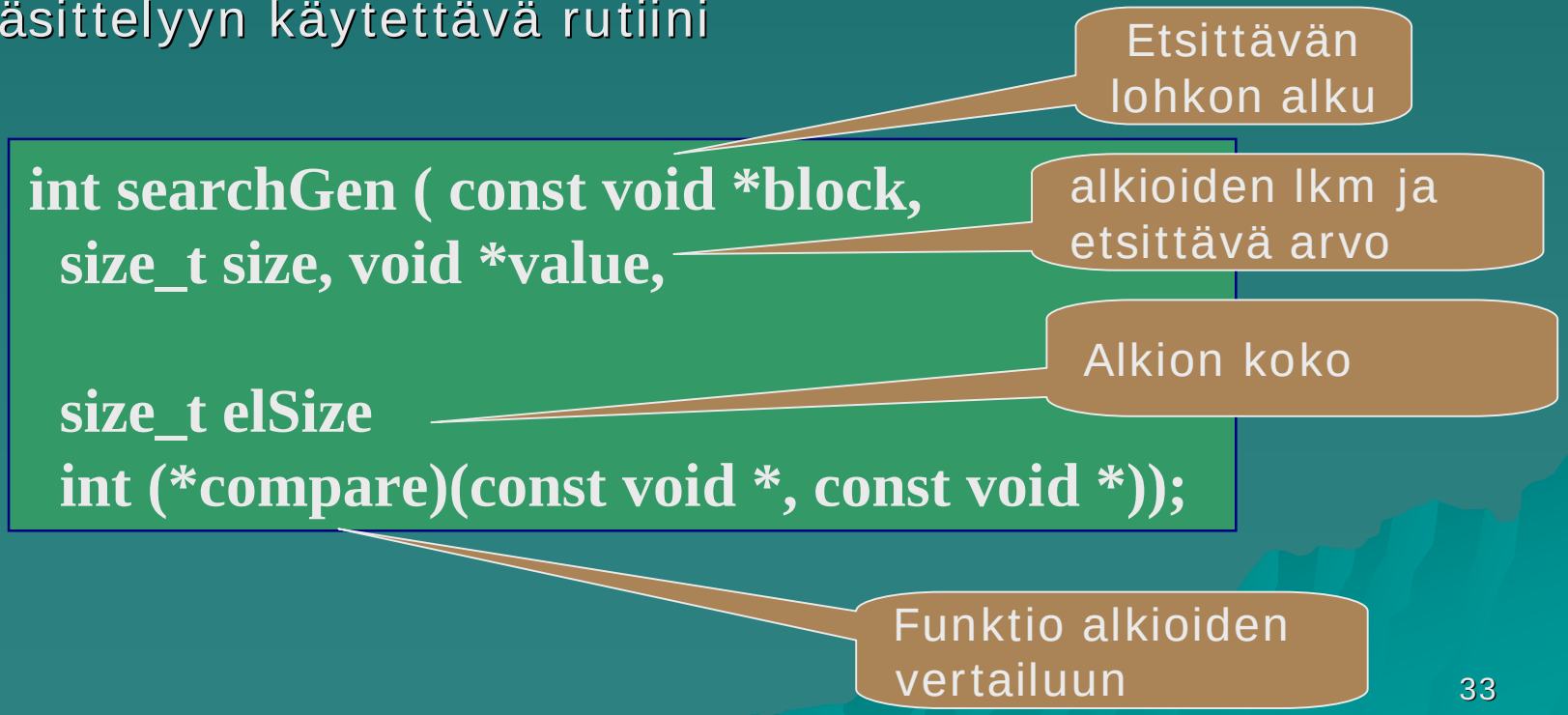
Alkioiden lkm

Osoitin viiteparametrina, jota ei saa muuttaa!

Lohkon läpikäynti

Yleistetään etsintäfunktio search

- C ei salli polymorfismia, mutta voimme simuloida sitä käyttämällä geneerisiä osoittimia (i.e. **`void*`**).
- Funktion esittelyssä voi määritellä, että paluuarvo ja parametrit ovatkin määräämätöntä tyyppiä **`void`**
- Tällöin tosin on kerrottava myös alkioiden koko ja niiden käsittelyyn käytettävä rutiini



Funktion kutsujan toimenpiteet

- Funktion kutsujan täytyy määritellä vertailufunktio, jota voidaan kutsua etsintäfunktiosta (ns. Call back -rutiini)
- 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;
}
```

geneerisen etsinnän käyttö

```
/* 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

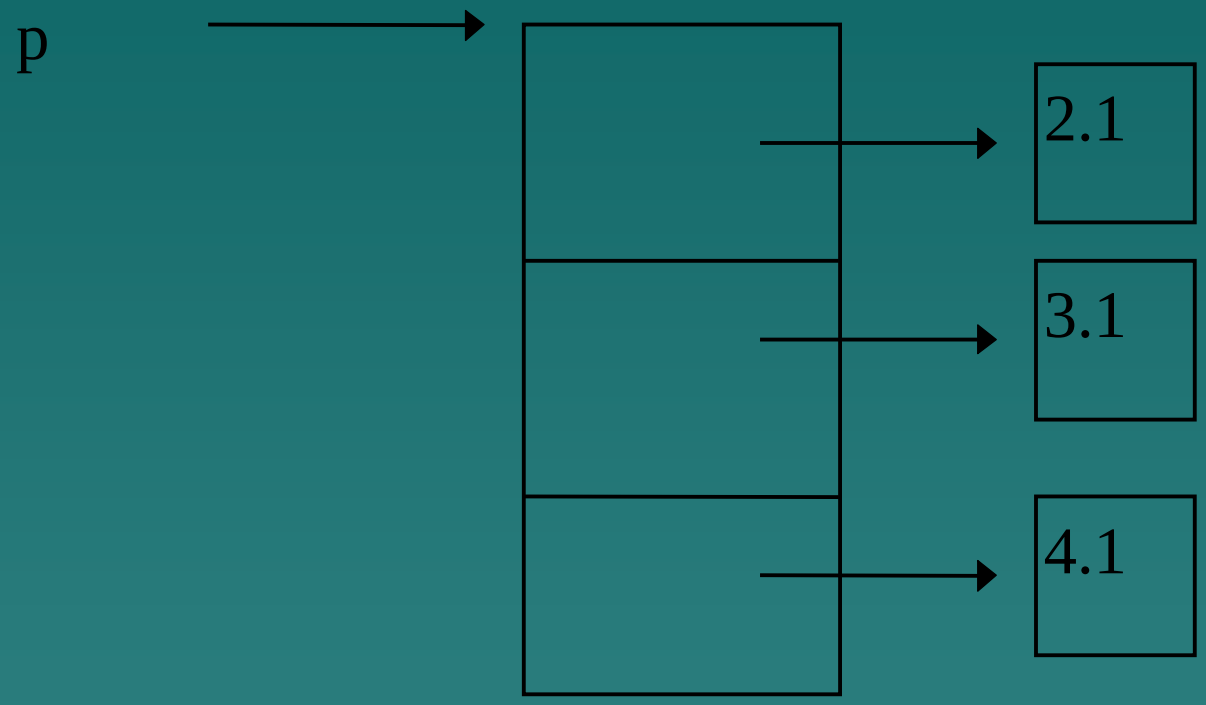
```
#define VOID(targ,size) ((void *) (((char *)(targ)) + \
(size)))
int searchGen(const void *block,
   size_t size, void *value, size_t elSize,
   int (*compare)(const void *, const void *)) {
   void *p;
   if(block == NULL)
      return 0;
   for(p = (void*)block; p< VOID(block,size*elsize);
   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

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

Sisältö

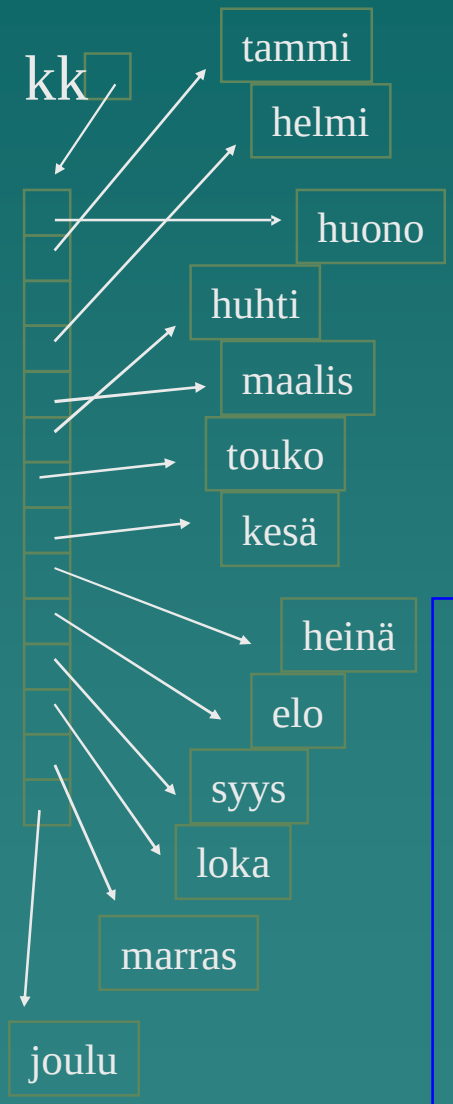
- Muistin rakenteesta
- Operaatiot ja void-tyyppi
- Muistinhallinta – varaus, vapautus
- Osoitinaritmetiikka ja muistilohkon käsittely
- Osoittimet ja funktiot
- → Osoitinlohko
- Merkkijonojen kopiointi (jos ehditään)

Osoitin lohkoon, jossa osoittimia



- Lohko, jossa on kolme osoitinta double-tyyppisiin alkioihin
- Viittaamineen alkioon asti vaatii kaksi viittauksenpurkua **p

Osoitinlohko, jonka alkiot merkkijono-osoittimia



- Tällaista rakennetta käytetään komentoriviparametrien käsittelyssä (muuttuja argv).
- Tässä esimerkkinä funktio, joka palauttaa arvonaan osoittimen kuukauden numeroa vastaavaan nimeen.

```
char *kk_nimi(int k)
{
  static char *kk[] = {”huono”, ”tammi”, ”helmi”,
maalis”, ”huhti”, ”touko”, ”kesä”, ”heinä”, ”elo”,
”syys”, ”loka”, ”marras”, ”joulu”};
return ( (k<1 || k > 12)  ? kk[0] : kk[k] );
}
```

Osoitinlohkon käyttö – dynaaminen muistinvaraus lohkolle ja viitattaville alkioille (tai uusille lohkoille)

```
double **block;
#define SIZE 3
if((block=calloc(SIZE, sizeof(double*)))==NULL)
      error;

for(i = 0; i < SIZE; i++)
  if((block[i]=calloc(1, sizeof(double)))==NULL)
          error;

*(*block) = 2.1;
block[0][0] = 2.1;
```

Varaus itse osoitinlohkolle

Varaus yhdelle alkiolle kerrallaan

Osoitinlohkon käyttö – viittaaminen alkioihin ja vapautus

- Lohkon viittaamien alkioiden alustaminen

```
for(i = 0; i < SIZE; i++)
        block[i][0] = 2.1 + i;
```

- Muistinvapauttaminen: viitatut alkiot ja osoitinlohko

```
for(i = 0; i < SIZE; i++)
        free(block[i]);
    free(block);
    block = NULL;
```

Ensin alkiot

Ja sitten lohko

Esimerkki: merkkijonon kopiointi (merkkijonon lopussa on merkki ’\0’)

```
#include <stdio.h>

void kopioi( char *s, char *t)
{
   int i =0;
  while (  (s[i] = t[i]) != ’\0’ )
     i++;
}

int main (void)
{
  char taalta [] =”Tämä kopioidaan.”,
        tanne[50];
  kopioi ( tanne, taalta);    printf(”%s\n”, tanne);
  return 0;
}
```

Merkkijono parametrina. Oikeasti osoittimia muistilohkojen alkuihin.

Viitataan merkkeihin taulukkomaisesti yksi kerrallaan

Esimerkki: merkkijonon kopiointi – muita tapoja kirjoittaa viittaus

Versio 1:

```
void kopioi( char *s, char *t)
{
  while (  (*s = *t) != '\0' )
   { s++; t++; }
}
```

Versio 2:

```
void kopioi( char *s, char *t)
{
  while (  (*s++ = *t++) != '\0' )
    ;
}
```

Versio 3:

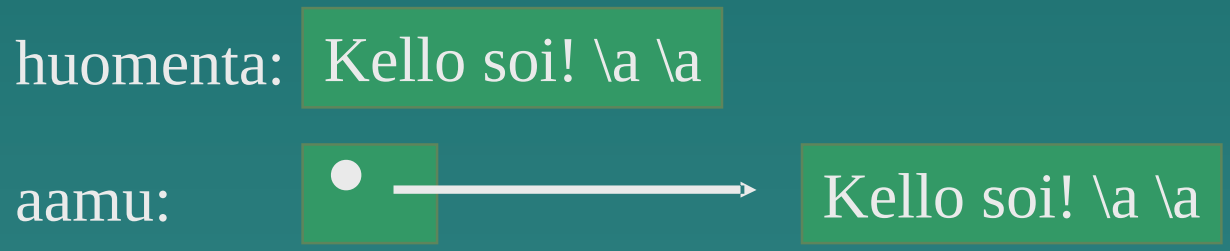
```
void kopioi( char *s, char *t)
{
  while  ( *s++  =  *t++ )   ;
}
```

Funktion otsikko on identtinen edellisen kalvon kanssa, vain viittaaminen alkioihin erinäköinen.

Minimalistisen selkeä!

Merkkijonovakiot ja osoittimet

- Merkkijonovakio on yhtenäinen muistialue, jonka päättää lopetusmerkki ’\0’.
- char *aamu = ”Kello soi! \a \a”
- char huomenta[] = ”Kello soi! \a \a”



- Muistialueen (huomenta) yksittäisiin alkioihin voidaan viitata
- Osoitin aamu voidaan asettaa muualle, mutta osoitetun merkkijonon muutosyritysten tulos on ’epämääräinen’

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.