21. Introduktion til arrays

I dette kapitel vil vi introducere arrays. Vi vil se på de væsentligste overordnede egenskaber af arrays, og vi vil se på nogle forhold, som er specifikke for C.

21.1. Arrays

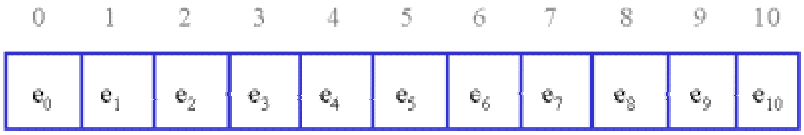
Lektion 6 - slide 2

Et array er en tabel med effektiv tilgang til elementerne via heltallige indeks numre

Vi viser herunder et array med 11 elementer. Disse kaldes her e_0, e_1, ..., e_{10}.

Med *array elementerne* mener vi det som befinder sig inden i kasserne i figur 21.1. Eksempelvis er e_0 et element i tabellen. I et konkret program kunne e_0 være et heltal. Elementerne er det lagrede indhold af tabellen. Alle elementerne i et array skal være af samme type.

Med *array indekserne* mener vi de forskellige pladser i et array. Inspireret af gængs matematisk notation kaldes indekser også ofte for *subscripts*. Indekserne i arrays i C starter altid med 0. Hvis der i alt er 11 elementer i tabellen er det højeste indeks således 10. Indekserne lagres ikke i tabellen. Det er kun en pladsbetegnelse. Da alle kasserne i tabellen har samme størrelse kan vi let slå direkte ned på et element med et bestemt indeks nummer.



Figur 21.1 *Et array med 11 elementer indiceret fra 0 til 10*

Herunder beskriver vi de vigtigste egenskaber af et array.

- Alle elementer er af samme type
- Index angiver en plads i tabellen
- Elementerne i et array `a` tilgås med *subscripting* notation: `a[i]`
- Når et array er skabt har det faste nedre og øvre indeksgrænser
  - Den nedre indeksgrænse er altid *0* i C.
- Elementerne lagres *konsekutivt* (umiddelbart efter hinanden).
  - Dermed er det simpelt at beregne hvor et element med et bestemt index er lagret
  - Effektiv tilgang via index

Lad os understrege to ting her. For det første at et array er lagret i ét sammenhængende lagerområde. For det andet at grænserne for, og dermed antallet af elementer i et array, ikke kan ændres når først arrayet er skabt.

21.2. Arrays i C

Lektion 6 - slide 3

Vi træder nu et skridt tilbage, og søger en motivation for at have arrays i vore programmer. Det gør vi i program 21.1 ved at erklære, initialisere og udskrive 11 variable, navngivet `tabel0`, ..., `tabel10`.

Det fremgår tydeligt, at program 21.1 er "træls". Det er uholdbart eksplicit at skulle erklære alle disse variable enkeltvis, at initialisere disse enkeltvis (på trods af at initialiseringen er systematisk), og at udskrive disse enkeltvis. Vi ønsker en bedre løsning, hvor vi i ét slag kan erklære `table0`, ..., `table10`. Endvidere ønsker vi at kunne initialisere og udskrive variable i en for-løkke.

```
#include <stdio.h>

int main(void) {

  double table0, table1, table2, table3, table4,
         table5, table6, table7, table8, table9, table10;

  table0 = 0.0;
  table1 = 2 * 1.0;
  table2 = 2 * 2.0;
  table3 = 2 * 3.0;
  table4 = 2 * 4.0;
  table5 = 2 * 5.0;
  table6 = 2 * 6.0;
  table7 = 2 * 7.0;
  table8 = 2 * 8.0;
  table9 = 2 * 9.0;
  table10 = 2 * 10.0;

  printf("%f, %f, %f, %f, %f, %f, %f, %f, %f, %f, %f\n",
         table0, table1, table2, table3, table4,
         table5, table6, table7, table8, table9, table10);

  return 0;
}
```

Program 21.1 *Motivation for arrays - et stort antal simple variable.*

I program 21.2 viser vi et program som svarer til program 21.1. I programmet herunder erklærer vi på det røde sted 11 doubles, som benævnes `table[0]`, ... `table[10]`. Variablen `table` er erklæret som et array. På de to blå steder initialiserer og udskriver vi elementer i `table`.

```c
#include <stdio.h>

int main(void) {

  double table[11];
  int i;

  for(i = 0; i < 11; i++)
    table[i] = (double)(2 * i);

  for(i = 0; i < 11; i++)
    printf("%f ", table[i]);

  printf("\n");

  return 0;
}
```

Program 21.2 *Introduktion af arrays.*

Vi viser herunder en variant af program 21.2 hvor vi på det røde sted viser en direkte, manifest repræsentation af tabellen. Dette er nyttigt hvis ikke vi på en systematisk måde kan initialisere tabellen i en løkke.

```c
#include <stdio.h>

int main(void) {

  double table[11] =
    {0.0, 5.0, 8.0, 9.9, 10.2, 8.5, 99.9, 1.0, -5.2, 7.5, 9.4};

  int i;

  for(i = 0; i < 11; i++)
    printf("%f ", table[i]);

  printf("\n");

  return 0;
}
```

Program 21.3 *Eksplicit array initialisering.*

Med disse eksempler og beskrivelserne har vi introduceret de vigtigste egenskaber ved arrays i C. I de efterfølgende kapitler ser vi på arrays i forbindelse med pointere. I kapitel 27 og efterfølgende kapiteler ser vi på arrays i forbindelse med tekststrenge.

22. Pointers

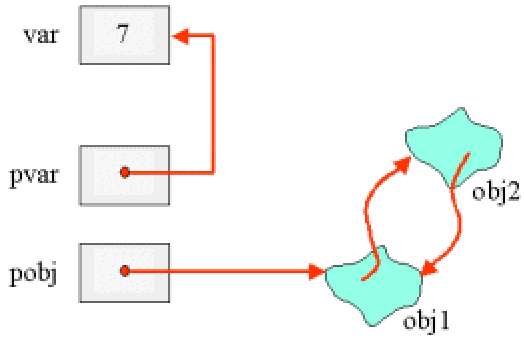
I dette kapitel diskuterer vi pointere i C.

22.1. Introduktion til pointere

Lektion 6 - slide 5

Herunder viser vi, at pointere kan bruges som alternative adgangsveje til variable. Vi viser også, at pointere kan bruges som adgangsvej til dataområder, som er placeret uden for variablene. Dette er pointerne til de grønne områder i figur 22.1. De grønne dataområder i figuren vil typisk være frembragt med dynamisk allokering med `malloc` eller `calloc`. Mere om dette i afsnit 26.2.

Pointere er uløseligt knyttet til programmering i C



Figur 22.1 *En illustration af to pointer variable.* ***pvar*** *er en pointer til en integer, som peger på adressen af var.* ***pobj*** *peger på en objekt obj1, som ikke er indeholdt i nogen variabel. obj1 peger på et andet objekt, som 'cyklisk' peger tilbage på obj1.*

Den normale måde at tilgå data er gennem navngivne variable.

Pointere udgør en alternativ og fleksibel tilgangsvej til data.

22.2. Pointerbegrebet i C

Lektion 6 - slide 6

Inden vi ser på eksempler beskriver vi aspekter af pointere i nedenstående punkter.

En *pointer* er en værdi som betegner en adresse på et lagret dataobjekt

- Værdier af variable kan tilgås via pointere til variablene
  - Adresseoperatoren **`&`** giver os adressen på variablen - hvilket er en pointer til indholdet i variablen
- Pointere til andre pladser i lageret returneres af de primitiver, der allokerer lager dynamisk
  - **`calloc`** og **`malloc`**
- Vi visualiserer ofte en pointer som en pil, som peger på et dataobjekt eller en variabel
  - Angivelse af lageradresser ville udgøre en mere maskinnær visualisering
- Pointere har typer, alt efter hvilke typer af værdier de peger på
  - Pointere til integers, pointere til chars, ...
  - Pointer til **`void`** er en *generisk pointer type* i C
- Dataobjekter, som ikke er indeholdt i nogen variabel, tilgås via pointere
- Pointere, som ikke peger på noget dataobjekt, skal have værdien **`NULL`**

22.3. Pointer variable

Lektion 6 - slide 7

Variable der indeholder pointere til dataobjekter af typen **`T`** skal erklæres af typen **`T *`**

I program 22.1 erklærer vi variablene `i`, `ptr_i`, `j`, `ptr_j`, `c` og `ptr_c`. Alle på nær `j` og `ptr_j` initialiseres sammen med erklæringerne.

Hvis der står en **`*`** foran en variabel i en erklæring betyder det at variablen skal indeholder en pointer. Konkret i program 22.1 skal `ptr_i` og `ptr_j` være pointers til heltal (`int`). `ptr_c` skal være en pointer til en char.

Assignmentet `ptr_j = &j` lægger adressen af `j` ind i `ptr_j`.

Assignmentet `ptr_i = ptr_j` lægger (kopierer) den pointer, som `ptr_j` indeholder, over i `ptr_i`. I figur 22.2 viser vi situationen efter at program 22.1 er kørt til ende.

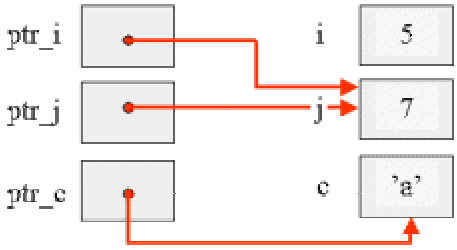
Bemærk at pointere visualiseres som pile, men reelt er en pointer en lagreradresse.

```
int i = 5, *ptr_i = &i, j = 7, *ptr_j;
char c = 'a', *ptr_c = &c;

ptr_j = &j;

ptr_i = ptr_j;
```

Program 22.1 *Erklæring og initialisering af et antal variable, hvoraf nogle er pointer variable.*



Figur 22.2 *Grafisk illustration af variablene i, j, c og deres tilsvarende pointervariable. Læg mærke til værdien af ptr_i, som er sat til at pege på samme værdi som ptr_j, altså variablen j.*

Vi viser i program 22.2 et helt C program, som indeholder fragmentet fra program 22.1. Outputtet fra programmet ses i program 22.3. Vær sikker på at du kan relatere outputtet fra program 22.2 med situationen i figur 22.2.

```
#include <stdio.h>

int main(void) {
  int i = 5, *ptr_i = &i, j = 7, *ptr_j;
  char c = 'a', *ptr_c = &c;

  ptr_j = &j;

  ptr_i = ptr_j;

  printf("i=%d, ptr_i=%p, *ptr_i = %i\n", i, ptr_i,  *ptr_i);
  printf("j=%d, ptr_j=%p, *ptr_j = %i\n", j, ptr_j,  *ptr_j);
  printf("c=%c, ptr_c=%p, *ptr_c = %c\n", c, ptr_c,  *ptr_c);

  return 0;
}
```

Program 22.2 *Hele programmet inklusive udskrivning af variablenes værdier.*

```
i=5, ptr_i=ffbef534, *ptr_i = 7
j=7, ptr_j=ffbef534, *ptr_j = 7
c=a, ptr_c=ffbef52f, *ptr_c = a
```

Program 22.3 *Output fra programmet.*

Bemærk i udskriften ovenfor, at pointere, udskrevet med konverteringstegnet `%p` i program 22.2, er vist i hexadecimal notation. Vi studerende hexadecimale tal i afsnit 16.5.

22.4. Addresse og dereferencing operatorerne

Lektion 6 - slide 8

Vi sætter i dette afsnit fokus på to nye operatorer, som knytter sig til pointere, nemlig adresse operatoren **`&`** og dereferencing operatoren **`*`**.

Udtrykket `&var` beregner til *adressen af variablen* `var`.

Udtrykket `*ptrValue` beregner *den værdi, som* `ptrValue` *peger på.*

Operatorerne `*` og `&` er unære prefix operatorer med høj prioritet

Dereferencing operatoren følger pointeren, og tilgår dermed det som pilen peger på.

I programmet herunder, som er helt identisk med program 22.2, viser de blå steder anvendelser af dereferencing operatoren, og de brune steder anvendelser af adresse operatoren.

```
int i = 5, *ptr_i = &i, j = 7, *ptr_j;
char c = 'a', *ptr_c = &c;

ptr_j = &j;

ptr_i = ptr_j;

printf("i=%d, ptr_i=%p, *ptr_i = %i\n", i, ptr_i,  *ptr_i);
printf("j=%d, ptr_j=%p, *ptr_j = %i\n", j, ptr_j,  *ptr_j);
printf("c=%c, ptr_c=%p, *ptr_c = %c\n", c, ptr_c,  *ptr_c);
```

Program 22.4 *Fremhævelse af adresse og dereferencing udtryk programmet fra forrige side.*

Oparatoren `*` kaldes også for *indirection operatoren*

For alle variable `v` gælder at udtrykket `*&v` er ækvivalent med udtrykket `v`

23. Call by reference parametre

I dette korte, men vigtige kapitel, introducerer vi en ny måde at bruge parametre på; Nemlig ved at overføre pointere som parametre. Kort sagt er der tale om parametre, som kan bruges til at ændre værdien af eksisterende variable. Det er også tale om en parametermekanisme som ofte er meget økonomisk i både tid og plads.

23.1. Call by reference parametre

Lektion 6 - slide 10

Ved brug af pointere kan vi realisere *call by reference parametre* i C

Vi skriver først et program, som ikke udnytter pointere og adresser på variable. Dermed bliver vi motiveret for at bruge pointere som parametre. Vi kan også bruge program 23.1 til at erindre hvordan værdiparametre (call by value parametre), diskuteret i bl.a. afsnit 12.5, virker.

```c
#include <stdio.h>

void swap(int p, int q){
   int   tmp;

   tmp = p;
   p = q;
   q = tmp;
}


int main(void)
{
   int   a = 3, b = 7;

   printf("%d  %d\n", a, b);      /* 3  7 is printed */
   swap(a, b);
   printf("%d  %d\n", a, b);      /* 3  7 is printed */
   return 0;
}
```

Program 23.1 *Et forsøg på ombytning af to variables værdi uden pointere - virker ikke.*

I program 23.1 herover ser vi med blåt definitionen af en funktion `swap`, og med rødt kaldet af `swap`. Vi overfører de to aktuelle parametre `a` og `b`, som modsvarer de to formelle parametre `p` og `q` i `swap`. I `swap` ombytter vi de to formelle parametres værdier, men der sker ingen ændring af de to aktuelle parametres værdier. Med andre ord, værdierne af `a` og `b` ændres ikke i kaldet af `swap`.

```c
#include <stdio.h>

void swap(int *p, int *q){
   int   tmp;

   tmp = *p;
   *p = *q;
   *q = tmp;
}


int main(void)
{
   int   a = 3, b = 7;

   printf("%d  %d\n", a, b);      /* 3  7 is printed */
   swap(&a, &b);
   printf("%d  %d\n", a, b);      /* 7  3 is printed */
   return 0;
}
```

Program 23.2 *Funktionen swap og et hovedprogram, som kalder **swap** på to variable.*

Programmet herover, program 23.2, er ændret lidt i forhold til program 23.1. I den blå del - funktionsdefinitionen - har vi indført to pointer parametre `p` og `q`. Både `p` og `q` skal pege på heltal. I

den røde del, hvor vi kalder `swap`, overfører vi adressen på hhv. `a` og `b` som aktuelle parametre. Bemærk brugen af adresseoperatoren **`&`**.

I kroppen af definitionen af `swap` (den blå del) lægger vi `*p` over i `temp`. Dette indebærer at vi følger pointeren `p` for at tilgå den værdi, som `p` peger på. I det konkrete kald er det værdien af `a`, altså 3 som assignes til `tmp`. I det andet assignment bliver `*q`, som konkret er værdien af `b`, lagt over i `temp`. I det tredje assignment bliver `*q`, hvilket er `a`, assignet til værdien af `temp`. Altså lægger vi 3 i `b`. Du kan se outputtet i program 23.3.

Vi ser at vi i denne udgave af programmet opnår en ombytning af værdierne i `a` og `b`. Vi har altså skrevet en funktion, som gennem parametrene er i stand til at ændre værdier af variable, som er uden for selve funktionen. Dette er essensen af *call by reference parametre*.

```
3   7
7   3
```

Program 23.3 *Output fra programmet.*

Call by reference parametre opnås når pointere overføres som *call by value* parametre.

Vi har allerede mange gange gjort brug af *call by reference parametre* i **`scanf`**.

Lad os påpege én ekstra ting. I f.eks. assignmentet `*p = *q` i program 23.2 skelner vi mellem *venstre værdien* og *højre værdien* af et udtryk. I den engelske litteratur omtales dette ofte som l-values og r-values. `*p` optræder som venstreside værdi i assignmentet. Ergo får vi med `*p` fat i den *plads*, som pointeren `p` peger på. `*q` optræder på højresiden af assignmentet. Her får vi med `*q` fat i den værdi, som *q peger på.

24. Pointers og arrays

Der er et tæt samspil mellem pointere og arrays i C. Dette er et udpræget kendetegn ved C. Samspillet gælder ikke generelt for andre programmeringssprog.

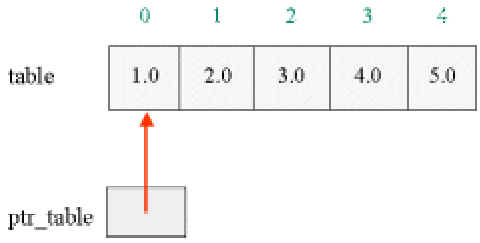
24.1. Pointers og arrays (1)

Lektion 6 - slide 12

Et array identificeres i C som en pointer til det første element.

Enhver operation der kan udføres med array subscripting kan også udføres med pointere.

I figur 24.1 illustrerer vi at `table` identificeres af en pointer til det første element i arrayet. For at illustrere dette klart har vi her lavet en ekstra variabel, `ptr_table`. Denne variabel er ikke nødvendigt idet udtrykket `&table[0]`, giver os den ønskede pointer. Husk her, at udtrykket `&table[0]` skal læses som adressen på plads 0 i `table`.



Figur 24.1 *Et array med fem elementer og en pointer til første element*

Hvis du skifter til 'slide view' af materiale (eller hvis du blot følger dette link) kommer du til en trinvis udvikling af figur 24.1. Følg de enkelte trin, og vær sikker på du forstår pointerens bevægelse hen over tabellen. I program 24.1 viser vi programmet for de trin, som gennemføres på figur 24.1 i den trinvise udvikling.

```
#include <stdio.h>

int main(void) {

  int i;
  double table[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  double *ptr_table;

  ptr_table = &table[0];

  *ptr_table = 7.0;

  ptr_table = ptr_table + 1;
  *ptr_table = *(ptr_table-1) * 5;

  ptr_table += 3;
  (*ptr_table)++;

  printf("Distance: %i\n", ptr_table - &table[0]);
  for(i=0; i<5; i++) printf("%f ", table[i]);
  printf("\n");

  return 0;
}
```

Program 24.1 *Et komplet C program, som illustrerer opdateringerne af table.*

24.2. Pointers og arrays (2)

Lektion 6 - slide 13

Vi vil her se flere eksempler på elementær pointertilgang til elementer i et array.

I program 24.2 herunder har vi et array med fem reelle tal (doubles). I den blå del etablerer vi en pointer til det første element i arrayet. I den røde del bruger vi denne pointer til at ændre de første to elementer i tabellen. Udskriften i program 24.3 afslører de ændringer vi har lavet via pointeren `pa`.

```c
#include <stdio.h>

int main(void) {

  int i;
  double a[5] = {5.0, 7.0, 9.0, 3.0, 1.0}, *pa = &a[0];

  /* Print the elements in a */
  for(i = 0; i < 5 ; i++) printf("%f ", a[i]);  printf("\n");

  /* Modify elements in a via the pointer pa */
  *pa = 13.0;
  pa++;
  *pa = 19.5;

  /* Print the elements in a again */
  for(i = 0; i < 5 ; i++) printf("%f ", a[i]);  printf("\n");

  return 0;
}
```

Program 24.2 *Hele det ovenstående program.*

```
5.000000 7.000000 9.000000 3.000000 1.000000
13.000000 19.500000 9.000000 3.000000 1.000000
```

Program 24.3 *Output fra programmet.*

I vores motivation for arrays så vi på et simpelt program, som erklærer, initialiser og udskriver et array. Det var program 21.2. Herunder, i program 24.4 viser vi en alternativ version af program 21.2 som bruger pointere til gennemløb (initialisering og udskrivning) af elementerne. Outputtet af programmmet vises i program 24.5.

```c
#include <stdio.h>

#define TABLE_SIZE 11

int main(void) {

  double table[TABLE_SIZE], *table_ptr = &table[0], *tp;
  int i;

  for(tp = table_ptr, i=0; tp < &table[TABLE_SIZE]; tp++, i++)
    *tp = (double)(2 * i);

  for(tp = table_ptr; tp < &table[TABLE_SIZE]; tp++)
    printf("%f\n",  *tp);

  printf("\n");

  return 0;
}
```

Program 24.4 *Et program hvor tabellen `table` tilgås via en pointer. Med rødt og blåt har vi fremhævet de aspekter, som svarer til detaljerne med samme farver i det oprindelige array program: De røde erklæringer af tabellen, og de blå tilgange til array elementerne. Med brunt ser vi et udtryk som udgør betingelsen for afslutningen af forløkkerne. Med lilla ser vi en 'optælling' af pointeren.*

```
0.000000
2.000000
4.000000
6.000000
8.000000
10.000000
12.000000
14.000000
16.000000
18.000000
20.000000
```

Program 24.5 *Output fra programmet.*

24.3. Pointers og arrays (3)

Lektion 6 - slide 14

Inden vi ser på et 'rigtigt eksempel', bubblesort i afsnit 24.4, vil vi her fastslå en ækvivalens mellem array indeks notation og pointer (aritmetik) notation. Vi vender tilbage til pointeraritmetik i afsnit 24.5.

Lad ligesom i program 24.2 `a` være et array, og lad `pa` være en ekstra pointer variabel som peger på det første element i `a`:

- `*(pa+i)` og `a[i]` har samme værdi, nemlig tabellens `i`'te element
- `pa`, `a` og `&a[0]` har samme værdi, nemlig en pointer til det første element i arrayet

Når et array overføres som parameter til en funktion er det reelt en pointer til arrayet som overføres

Bubble sort programmet på næste side giver flere eksempler på dette.

24.4. Eksempel: Bubble sort

Lektion 6 - slide 15

Foreløbig slut med de helt små kunstige eksempler. Vi ser her på et program der kan sortere et array af heltal. Vi bruger bubblesort algoritmen, som er én af de forholdsvis velkendte sorteringsalgoritmer.

Boblesortering er en klassisk, men ikke særlig effektiv sorteringsalgoritme

Kernen i sorteringsalgoritmen vises herunder. `j` løber fra bagenden af vores array tilbage til `i`. `i` bliver gradvis større i takt med at den ydre forløkke gør fremskridt. Billedlig talt bobler de lette elementer (de mindste elementer) op mod forenden af tabellen.

```
void bubble(int a[] , int n)     /* n is the size of a[] */
{
   int   i, j;

   for (i = 0; i < n - 1; ++i){
     for (j = n - 1; j > i; --j)
       if (a[j-1] > a[j])
          swap(&a[j-1], &a[j]);
   }
}
```

Program 24.6 *Function* ***bubble*** *som laver 'bubble sort' på arrayet a som har n heltalselementer.*

Herunder, i program 24.7 ser vi et komplet program, hvori `bubble` funktionen fra program 24.6 er indsat næsten direkte. (Den brune del, kaldet af `prn_array`, er dog tilføjet).

I main ser vi med blåt et udtryk der måler størrelsen (antallet af elementer) i tabellen `a`. Læg godt mærke til hvordan dette sker. `sizeof` er en operator, som giver størrelsen af en variabel eller størrelsen af en type. Størrelsen læggers over i variablen `n`.

Med rødt viser vi kaldet af `bubble` på `a` og `n`. Tabellen `a` overføres som en reference til arrayet. Dette er fordi `a` i bund og grund blot er en pointer til tabellen, som omtalt i afsnit 24.1.

Forneden i program 24.7 ser vi funktionen `swap`, som vi udviklede i afsnit 23.1. `swap` laver de primitive skridt i sorteringsarbejdet, nemlig naboombytningerne i boble processen.

```
#include <stdio.h>

void   bubble(int a[], int n);
void   prn_array(char* s, int a[], int n);
void   swap(int *p, int *q);

int main(void)
{
   int   a[] = {7, 3, 66, 3, -5, 22, -77, 2};
   int   n;

   n = sizeof(a) / sizeof(int);
   prn_array("Before", a, n);
   bubble(a, n);                 /* bubble sort */
   prn_array(" After", a, n);
   putchar('\n');
   return 0;
}

void bubble(int a[] , int n)      /* n is the size of a[] */
{
   int   i, j;
```

```
   for (i = 0; i < n - 1; ++i){
     for (j = n - 1; j > i; --j)
       if (a[j-1] > a[j])
          swap(&a[j-1], &a[j]);
     prn_array("During", a, n);
   }
}

void prn_array(char* s , int a[] , int n)
{
   int   i;

   printf("\n%s%s%s", "   ", s, " sorting:");
   for (i = 0; i < n; ++i)
      printf("%5d", a[i]);
   putchar('\n');
}

void swap(int *p, int *q)
{
   int   tmp;

   tmp = *p;
   *p = *q;
   *q = tmp;
}
```

Program 24.7 *Hele 'bubble sort' programmet - med løbende udskrift af arrayet.*

På den slide, som er tilknyttet dette afsnit, har vi illustreret de enkelte skridt i boblesorteringen.

Det er let at set at boblesorteringsprogrammet fortager op til n^2 sammenligninger og ombytninger.

De gode sorteringsalgoritmer kan nøjes med i størrelsesordenen $n\ log(n)$ sammenligninger og ombytninger.

24.5. Pointeraritmetik

Lektion 6 - slide 16

Vi skal her se, at man kan bruge de aritmetiske operatorer i udtryk hvor der indgår pointere.

```
T *p, *q;
int i;
```

Program 24.8 *Erklæring af to pointere p og q samt en int i.*

Variablene `p`, `q` og `i` vist herover sætter scenen for punkterne herefter.

- Følgende former for pointeraritmetik er lovlige i C:
  - Assignment af pointere af samme type: `p = q`
  - Assignment af pointer til `NULL` eller 0: `p = 0` eller `p = NULL`
  - Addition og subtraktion af integer til pointer: `p + i` og `p - i`
  - Subtraktion af to pointere: `p - q`
  - Sammenligning af to pointere: `p < q`
  - Sammenligning af pointere med *0* eller `NULL`:
    `p == 0`, `p == NULL`, `p != NULL` eller `p > NULL`

Når vi i f.eks. `p + i` adderer et heltal til en pointer bliver pointeren optalt i logisk enheder, ikke i bytes. Med andre ord er C smart nok til at addere et passende antal bytes til `p`, som svarer til størrelsen af typen `T` i erklæringerne i program 24.8.

Vi viser et praktisk eksempel på pointeraritmetik i et program vi har lånt fra bogen 'The C Programming Language'. I funktionen `my_strlen` benytter det røde og det blå udtryk pointeraritmetik. Funktionen beregner antallet af tegn i en C tekststreng. Vi har ikke endnu diskuteret tekststrenge. Det vi ske i kapitel 27.

```
int my_strlen(char *s){
  char *p = s;

  while (*p != '\0')
    p++;

  return p-s;
}
```

Program 24.9 *En funktion som tæller antallet af tegn i en streng.*

I program 24.10 viser vi funktionen i konteksten af et komplet C program.

```c
#include <stdio.h>

/* Program from 'The C programming language' by
   Kernighan and Ritchie. */

#define STR1 "monkey"
#define STR2 "programming in C"

int my_strlen(char *s){
  char *p = s;

  while (*p != '\0')
    p++;

  return p-s;
}

int main(void) {

  printf("Length of \"%s\" = %i\n", STR1, my_strlen(STR1));
  printf("Length of \"%s\" = %i\n", STR2, my_strlen(STR2));

  return 0;
}
```

Program 24.10 *Hele programmet.*

24.6. Index out of bounds

Lektion 6 - slide 17

Når man programmerer med arrays er der altid fare for at man kan adressere uden for arraygrænserne. I de fleste sprog vil man få en fejlbesked fra det kørende program, hvis dette sker. I C skal man ikke regne med at få en sådan advarsel. Typisk vil man blot tilgå (læse eller skrive) data uden for arrayet. Dette kan have alvorlige konsekvenser.

> For et array **`a[N]`** er det programmørens ansvar at sikre, at indexes forbliver i intervallet **`[0..N-1]`**

Vi ser nedenfor på et typisk eksempel, hvor lidt uforsigtighed i en forløkke (det røde sted) betyder at vi adresserer `a[5]`. Husk på, at det kun giver mening at referere `a[0]`, ..., `a[4]`.

```c
double table[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

for(i = 0; i <= 5; i++)     /* index out of bounds for i = 5 */
{
  table[i] += 5.0;
  printf("Element %i is: %f\n", i, table[i]);
}
```

Program 24.11 *Et eksempel på indicering uden for indeksgrænserne i et array.*

Når man kører programmmet kan der ske lidt af hvert. Typisk vil man få fat i et bitmønster i `a[5]` (otte bytes), som fortolkes som et double. Dette kan give 'sjove' resultater.

> Det kørende C program opdager ikke nødvendigvis at indekset løber over den øvre grænse.
>
> Programmets opførsel er *udefineret* i sådanne tilfælde.

25. Arrays af flere dimensioner

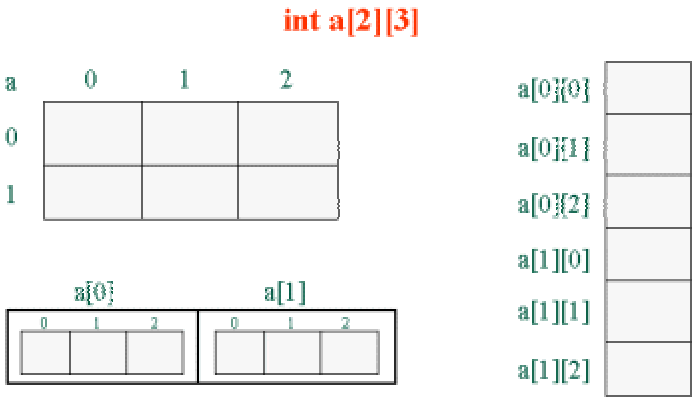
Efter at vi har set på arrays med elementer af simple typer vil vi nu studere arrays hvor elementerne er arrays. Sådanne arrays kan forstås som arrays af to eller flere dimensioner.

25.1. To dimensionelle arrays

Lektion 6 - slide 19

> Arrays af to dimensioner er et array hvor elementtypen er en anden arraytype

I figur 25.1 ses tre forskellige fortolkninger af `a[2][3]`. Øverst til venstre vises vores mentale model af en table med to dimensioner. Nederst til venstre vises *arrays i arrays* forståelsen, som er den korrekte begrebsmæssige forståelse i forhold til C. Til højre vises hvordan arrayet håndteres i lageret; Vi kan sige at arrayet er lineariseret. I forhold til den viste tabel af to dimensioner sker der en rækkevis lagring.



Figur 25.1 *Illustrationer af et to dimensionelt array. Øverst til venstre illustreres forståelse af de to dimensioner, med to rækker og tre søjler. Vi kunne naturligvis lige så godt have byttet om på rækker og søjler. Nederst til venstre ses en forståelse der understreger at a er et array med to elementer, som begge er arrays af tre elementer. Længst til højre ses den maskinnære forståelse, 'row major' ordningen.*

Vi tænker normalt på `a` som en to-dimensionel tabel med to rækker og tre søjler.

Reelt er elementerne lagret lineært og konsekutivt, i *row major* orden.

Elementer som kun afviger med én i det sidste indeks er naboelementer.

25.2. Processering af arrays med to dimensioner

Lektion 6 - slide 20

Næste emne er helt naturligt hvordan vi gennemløber et array af flere dimensioner.

Det er naturligt at bruge to indlejrede `for` løkker når man skal processere to dimensionelle arrays

Alternativt er det muligt at gennemløbe den samlede tabel lineært, med udgangspunkt i en pointer til det første element.

De to gennemløb omtalt ovenfor er illustreret i program 25.1.

```
double a[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}};

for (i = 0; i < 2; i++)
  for (j = 0; j < 3; j++)
    sum1 += a[i][j];

for (k = 0; k < 2 * 3; k++)
  sum2 += *(&a[0][0] + k);
```

Program 25.1 *Erklæring, initialisering og to identiske gennemløb af et to dimensionelt array.*

For god ordens skyld viser vi program 25.1 som et komplet C program i program 25.2.

```c
#include <stdio.h>

int main(void) {

  int i, j, k;
  double sum1 = 0.0, sum2 = 0.0;
  double a[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}};

  for (i = 0; i < 2; i++)
    for (j = 0; j < 3; j++)
      sum1 += a[i][j];

  for (k = 0; k < 2 * 3; k++)
    sum2 += *(&a[0][0] + k);

  printf("First method:  the sum is: %f\n", sum1);
  printf("Second method: The sum is: %f\n", sum2);

  return 0;
}
```

Program 25.2 *Det samlede program.*

C understøtter arrays af vilkårlig mange dimensioner

Tre- og flerdimensionelle arrays er umiddelbare generaliseringer af to dimensionelle arrays

26. Statisk og dynamisk lagerallokering

Variable kræver lagerplads når programmet kører. Vi siger, at der *allokeres lager* til vore vore variable. Når vi i kildeprogrammet har erklæret en variabel vil køretidssystemet klare lagerallokeringen af variablen uden yderligere programøranstrengelser. Dette er *statisk lagerallokering*.

I C er det også muligt, når programmet kører, at bede om lager, som ikke er indeholdt i nogen variable. Når dette sker vil vi få overdraget en pointer til begyndelsen af det pågældende lagerområde. Dette kaldes *dynamisk lagerallokering*.

26.1. Statisk lagerallokering

Lektion 6 - slide 22

I kurset indtil nu har vi benyttet *statisk lagerallokering* for alle data

Med *statisk lagerallokering* afsættes lagerplads implicit når den omkringliggende blok aktiveres. Lagerplads inddrages implicit når blokken forlades.

I programmet herunder allokeres `tab1` og `tab2` statisk i `main`. Til `tab1` afsættes plads til `N` doubles. `N` er en symbolsk konstant, som er defineret til 500. Til `tab2` afsættes plads til `N*K` doubles, altså konkret 500 doubles (eller typisk 4000 bytes). Der allokeres naturligvis også lager til `i`.

```
#include <stdio.h>
#define N  500
#define K   10

int main(void) {
  int i;
  double tab1[N];       /* allocation of N doubles */
  double tab2[N*K];     /* allocation of N*K doubles */

  /* do something with tab1 and tab2 */
  return 0;
}
```

Program 26.1 *Illustration af statisk lagerallokering i C.*

Når `main` er kørt færdig frigives lagerpladsen, som er afsat til `tab1`, `tab2` og `i`.

Statisk lagerallokering er knyttet til erklæringen af variable.

Udtryk som indgår i indeksgrænser for et array skal være *statiske udtryk*

26.2. Dynamisk lagerallokering (1)

Lektion 6 - slide 23

I mange tilfælde ved vi først på programmets køretid hvor meget lager vi har brug for. Dette kan f.eks. afhænge af et tal, som er indlæst tidligere i programkørslen. Derfor er dynamisk lagerallokering ikke let at komme uden om.

Der er ofte behov for en mere *fleksibel* og *dynamisk* form for lagerallokering, hvor det kørende program eksplicit allokerer lager.

Med *dynamisk lagerallokering* afsættes lagerplads eksplicit, når programmet har behov for det.

I program 26.2 vises et forsøg på at at allokere passende plads til `a` i forhold til `n`, som er et indlæst heltal.

Ideen med at indlæse `n` i en ydre blok i forhold til den blok, hvor arrayet allokeres er god. Men vi skal ikke regne med, at den virker i C. ANSI C kræver, at `n` er et konstant udtryk, hvilket vil sige at der kun må indgå konstanter i udtrykket. `sizeof` operatoren kan også indgå.

```
#include <stdio.h>

int main(void) {

  int n;
  printf("Enter an integer size:\n");
  scanf("%d",&n);

  { int i;
    double a[n];

    for (i = 1; i < n; i++)
      a[i] = (double)i;

    for (i = 1; i < n; i++)
      printf("%f ", a[i]);
    printf("\n");

  }

  return 0;
}
```

Program 26.2 *Et kreativt forsøg på dynamisk allokering - ikke ANSI C.*

I stedet for at bruge ideen i program 26.2 skal vi anvende enten `malloc` eller `calloc`, som eksemplificeret i program 26.3. På det røde sted allokerer vi plads til `n` heltal (`int`s). Vi får en pointer til lagerområdet. Læg mærke til at `a` er en pointer til en `int`. Det nyallokerede lagerområde bruges i de to efterfølgende forløkker. På det blå sted deallokerer vi igen lagerområdet. Det betyder at vi overgiver lageret til C køretidssystemet, som så efterfølgende kan bruge det til andre formål. F.eks. kan lageret genbruges hvis vi lidt efter igen beder om nyt laver via `calloc`.

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

int main(void)
{
   int   *a, i, n, sum = 0;

   printf("\n%s",
      "An array will be created dynamically.\n\n"
      "Input an array size n followed by n integers:  ");
   scanf("%d", &n);
   a = calloc(n, sizeof(int));  /* get space for n ints */
   for (i = 0; i < n; ++i)
      scanf("%d", &a[i]);
   for (i = 0; i < n; ++i)
      sum += a[i];
   free(a);                     /* free the space */
   printf("\n%s%7d\n%s%7d\n\n",
      " Number of elements:", n,
      "Sum of the elements:", sum);
   return 0;
}
```

Program 26.3 *Et program der foretager dynamisk allokering af et array.*

Vi viser afslutningsvis et program, som dynamisk allokerer plads til det array, som vi illustrerede i figur 25.1. Allokeringen sker på det røde sted. I dette eksempel frigiver vi ikke lagerpladsen, idet lageret bruges helt til programafslutningen.

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

#define N 2
#define M 3

int main(void) {
  int *pint, i, j;

  pint = malloc(N*M*sizeof(int));

  for (i=0; i<N; i++)
   for (j=0; j<M; j++)
     *(pint + M*i + j) = (i+1) * (j+1);

  for (i = 0; i < M * N; i++){
    printf("%4i ", *(pint + i));
    if ((i+1) % M == 0) printf("\n");
  }

  return 0;
}
```

Program 26.4 *Et program der dynamisk allokerer et to-dimensionelt array.*

26.3. Dynamisk lagerallokering (2)

Lektion 6 - slide 24

Herunder opsummerer vi dynamisk lagerallokering.

- Dynamisk lagerallokering
  - Eksplicit *allokering* med **`calloc`** eller **`malloc`** fra fra **`stdlib.h`**
  - Eksplicit *deallokering* med funktionen **`free`**
  - Risiko for *dangling references*
    - Hvis en pointer til et dynamisk allokeret objekt følges efter at objektet er deallokeret med **`free`**

I mange moderne sprog styres lager deallokering af en *garbage collector*, som frigiver lagerplads, når det ikke længere kan tilgås af programmet.

Understøttelse af lageradministration med en garbage collector er et meget spændende emne. En diskussion af dette falder dog uden for rammerne af disse noter.