27. Tekststrenge i C

Denne lektionen er om tekststrenge, som både har berøringspunkter med arrays og pointers. Programmering med tekststrenge er vigtig i enhver form for tekstbehandling. Lektionen udspænder kapitel 27 - kapitel 31.

27.1. Strenge og tekststrenge

Lektion 7 - slide 2

Vi starter forholdsvis generelt med at definere strenge og tekststrenge.

En *streng* er en sekvens af data af samme type

En *tekststreng* er en streng af datatypen tegn

I de fleste programmeringssprog noteres tekststrenge ved brug af dobbelte anførselstegn. Dermed adskiller strenge sig fra tegn (se afsnit 16.3) der i mange sprog noteres ved enkelte anførselstegn. Se endvidere afsnit 27.6.

- Almen notation for tekststrenge:
  - **`"En tekststreng"`**
- Den tomme streng
  - Strengen der ikke indeholder data
  - Strengen af længde *0* - den kortest mulige streng
  - Den tomme tekststreng noteres naturligt som `""`

27.2. Tekststrenge og arrays

Lektion 7 - slide 4

Efter den korte, generelle introduktion til tekststrenge i afsnit 27.1 ser vi nu konkret på tekststrenge i C. Den vigtigste detalje er konventionen om, at der skal bruges et særligt tegn, nultegnet, for at afslutte en tekststreng. Nultegnet er det tegn i ASCII alfabetet (afsnit 16.1 og program 16.2) som har talværdien 0. Det er med andre ord det første tegn i tabellen.

En tekststreng i C er et *nulafsluttet* array med elementtypen **`char`**.

I figur 27.1 herunder ser vi tekststrengen `"Aalborg"`, som det repræsenteres i et array. Læg mærke til tegnet i 7'ende position: nul tegnet. Uden dette tegn ved vi ikke hvor tekststrengen afsluttes i lageret.

0	1	2	3	4	5	6	7
'A'	'a'	'l'	'b'	'o'	'r'	'g'	'\0'

Figur 27.1 *En illustration af en nulafsluttet tekststreng i C*

- Det afsluttende nultegn kaldes en *sentinel*
  - Oversættes direkte som en 'vagt'
  - Eksplicit markering af afslutningen af tekststrengen
  - Som et alternativt kunne vi holde styr på den aktuelle længde af tekststrengen

Herunder, i figur 27.2 viser vi tekststrengen `"Aalborg"` placeret midt i et array. Med en `char` pointer som peger på celle 2 i tabellen har vi stadig fat i `"Aalborg"`.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
		'A'	'a'	'l'	'b'	'o'	'r'	'g'	'\0'				

Figur 27.2 *En tekststreng som er placeret 'i midten' af et array of char*

En tekststreng i C skal ikke nødvendigvis udfylde hele det omkringliggende array

27.3. Initialisering af tekststrenge

Lektion 7 - slide 5

Der er flere forskellige måder at initialisere en tekststreng. Herunder, i program 27.1 viser vi essentielt tre forskellige. Den røde måde benytter manifest array notation, som blev diskuteret i program 21.3 i afsnit 21.2.

Den blå og den lilla måde er overfladisk set sammenfaldende. Her benytter vi en strengkonstant i dobbeltquotes, som diskuteret i afsnit 27.1. Den blå måde bruger en array variabel, og den lilla en pointervariabel. I afsnit 27.5 diskuterer vi nogle vigtige forskelle mellem den blå og lilla form, hvad angår mulighederne for mutation af tegn i strengene.

Med den brune måde indsætter vi enkelttegn i et array pr. assignment.

Læg mærke til at den røde og brune måde kræver eksplicit indsættelse af nultegnet. Nultegnet skal ikke indsættes eller noteres eksplicit, når vi bruger strengkonstanter.

```
char str_1[] = {'A', 'a', 'l', 'b', 'o', 'r', 'g', '\0'};

char str_2[] = "Aalborg";

char *str_3 = "Aalborg";

char str_4[8];
str_4[0] = 'A';  str_4[1] = 'a';  str_4[2] = 'l';
str_4[3] = 'b';  str_4[4] = 'o';  str_4[5] = 'r';
str_4[6] = 'g';  str_4[7] = '\0';
```

Program 27.1 *Forskellige initialiseringer af tekststrenge.*

Ved initialisering via tekstkonstanter tilføjes nultegnet automatisk af compileren

27.4. Tekststrenge og pointere

Lektion 7 - slide 6

Vi skal nu se på forholdet mellem tekststrenge og pointere. I bund og grund er dette forhold afledt af at en tekststreng er et array (afsnit 27.2), og at et array er en pointer til det første element i arrayet (afsnit 24.1).

En tekststreng opfattes i C som en pointer til det første tegn

Dette følger direkte af den generelle sammenhæng mellem arrays og pointere

Herunder, i program 27.2 ser vi et eksempel på et program, der tilgår tekststrenge via pointere. Overordnet set kopierer programmet tegnene i `"Aalborg"` ind midt i strengen `str`.

```
#include <stdio.h>

int main(void) {
  char str_aal[] = "Aalborg";
  char str[14];

  char *str_1, *str_2;
  int i;

  /* fill str with '-' */
  for(i = 0; i < 14; i++)
    str[i] = '-';

  /* let str_1 and str_2 be running pointers */
  str_1 = str;  str_2 = str_aal;

  /* copy str_all into the middle of str */
  str_1 += 2;
  for( ; *str_2 != '\0'; str_1++, str_2++)
```

```
    *str_1 = *str_2;

  /* terminate str */
  *str_1 = '\0';

  printf("%s\n", str);

  return 0;
}
```

Program 27.2 *Et program der ved brug af pointere kopierer strengen "Aalborg" ind midt i en anden streng.*

Lad os forklare programmet i lidt større detalje. I starten af `main` ser vi to variable `str_all` og `str`, som er strenge. Vi initialiserer `str_all` til `"Aalborg"`. `str` initialiseres i den første forløkke til udelukkende at indeholde tegnene `'-'`.

Dernæst assignes `str_1` og `str_2` til pointere til de første tegn i hhv. `str` og `str_all`. Med rødt ser vi at `str_1` føres to positioner frem i arrayet med `'-'` tegn. Dernæst kommer en forløkke, som kopierer tegnene i `str_all` over i `str`. Dette sker gennem de pointere vi har sat op til at gennemløbe disse arrays. Før vi udskriver `str` med `printf` indsætter vi eksplicit et `'\0'` tegn i `str`.

Herunder, i program 27.3 ser vi outputtet fra program 27.2.

```
--Aalborg
```

Program 27.3 *Output fra programmet.*

27.5. Ændringer af tekststrenge

Lektion 7 - slide 7

Vi arbejder i dette afsnit videre på program 27.1. Det er vores mål at forklare under hvilke omstændigher vi kan ændre i de tekststrenge, som indgår i program 27.1.

> Tekststrenge, til hvilke der er allokeret plads i et array, kan ændres (muteres)
>
> Tekststrenge, der er angivet som en strengkonstant refereret af en pointer, kan ikke ændres

Mutation er et biologisk inspireret ord, som betyder *at ændre (i en del af) noget*. Ordet benyttes ofte når vi foretager ændringer i en del af en eksisterende datastruktur, eksempelvis et array.

Vores pointe bliver illustreret i program 27.4. Svarende til de farvede aspekter i program 27.1 foretager vi fire ændringer i strengen `"Aalborg"`.

Kun den lilla ændring giver problemer. `str_3` peger på en tekststreng, som ikke er allokeret som et array i det program vi har skrevet. `str_3` peger derimod på en konstant tekststreng, som er en del af programmet. Denne tekststreng er lagret sammen med programmet, på et sted vi ikke kan ændre i strengen.

```
char str_1[] = {'A', 'a', 'l', 'b', 'o', 'r', 'g', '\0'};
*(str_1 + 1) = 'A';

char str_2[8] = "Aalborg";
*(str_2 + 1) = 'A';

char *str_3 = "Aalborg";
*(str_3 + 1) = 'A';                         /* Ulovligt */

char str_4[8];
str_4[0] = 'A';  str_4[1] = 'a';  str_4[2] = 'l';
str_4[3] = 'b';  str_4[4] = 'o';  str_4[5] = 'r';
str_4[6] = 'g';  str_4[7] = '\0';
*(str_4 + 1) = 'A';
```

Program 27.4 *Et program der ændrer det andet tegn i Aalborg fra 'a' til 'A'.*

Moralen vi kan uddrage af ovenstående er at mutation af en tekststreng kræver at denne tekststreng er allokeret statisk (afsnit 26.1) eller dynamisk (afsnit 26.2) i programmet. Vi kan ikke ændre i konstante tekststrenge, til hvilke vi kun har en pointer, og som er angivet i selve programmet.

27.6. Tekststrenge i forhold til tegn

Lektion 7 - slide 8

Vi vil her påpege en væsentlig forskel mellem notationen for tegn og tekststrenge.

Tekststrenge er sammensat af tegn
De to værdier `'a'` og `"a"` meget forskellige

Forskellen på notationen for tegn og tekststrenge kan eksemplificeres som følgende:

- `'a'`
  - Et enkelt tegn af typen char
  - Reelt heltallet *97*
- `"a"`
  - Et array med to elementer
  - Nulte element er tegnet `'a'` og første element er tegnet `'\0'`

27.7. Den tomme streng og NULL

Lektion 7 - slide 9

I lighed for forskellen på `'a'` og `"a"` vil vi nu pege på forskellen mellem `NULL` og den tomme tekststreng.

> Man skal kunne skelne mellem *den tomme tesktstreng* og en **`NULL`** pointer
>
> **`NULL`** og `""` er meget forskellige

NULL og den tomme streng kan karakteriseres som følger:

- **`NULL`**
  - **`NULL`** er en pointer værdi
  - **`NULL`** værdien bruges for en pointer, der ikke peger på en plads i lageret
  - Reelt heltallet 0
- Den tomme streng `""`
  - Den tomme string `""` er en streng værdi
  - `""` er et array med ét element, nemlig `'\0'` tegnet

28. Leksikografisk ordning

Dette kapitel handler om ordning og lighed mellem strenge.

28.1. Leksikografisk ordning af strenge

Lektion 7 - slide 11

En ordning af tekststrenge kan afledes af ordningen blandt de tegn, som tekststrengen er bygget op af. I dette afsnit diskuterer vi ordningen af tekststrenge på det generelle plan, uafhængig af C.

> Ordningen af tegn inducerer en ordning af tegnstrenge (tekststrenge)
>
> Den normale alfabetiske ordning af tekststrenge spiller en vigtig rolle ved opslag i leksika, telefonbøger, mv.
>
> Vi vil her definere hvad det betyder af strengen *s* ***er mindre end*** strengen *t*

Herunder definerer vi betydningen af at en streng *s* er mindre end en streng *t*.

- Lad e betegne den tomme streng `""` og lad s og t være to tekststrenge
- $s < t$ hvis og kun hvis der findes tegn **`c`** og **`d`** samt to kortere strenge u og v så
    - $s = e$
      $t =$ **`c`** u *eller*
    - $s =$ **`c`** u
      $t =$ **`d`** v
        - **`c`** $<$ **`d`**
        - **`c`** $=$ **`d`** og $u < v$

Der er to tilfælde i definitionen. Første tilfælde siger at den tomme streng er mindre end enhver ikke-tom streng.

Det andet tilfælde har to undertilfælde, som begge udtaler sig om to ikke-tomme strenge s og t. Det første undertilfælde siger at $s < t$ hvis første tegn i s er mindre end første tegn i t. Det andet undertilfælde siger at hvis første tegn i s er lig med første tegn i t, afgøres vurderingen $s < t$ rekursivt af $u < v$, hvor u er halen af s og v er halen af t.

Lad os se på nogle eksempler. De følgende fire udsagn er alle sande.

1. `"Andersen"` < `"Hansen"`
2. `"abe"` < `"kat"`
3. `"abehat"` < `"abekat"`
4. `"abe"` < `"abekat"`

Det første tilfælde forklares ved at `'A'` < `'H'`.

Det andet tilfælde argumenteres ganske tilsvarende, nemlig ved at `'a'` < `'k'`.

I det tredje tilfælde skræller vil tegnene `'a'`, `'b'`, og `'e'` af begge strenge og vurderer dermed `"hat"` i forhold til `"kat"`.

I det sidste tilfælde skræller vi også tegnene `'a'`, `'b'`, og `'e'` af begge strenge og vurderer den tomme streng `""` i forhold til `"kat"`.

28.2. Lighed af strenge (1)

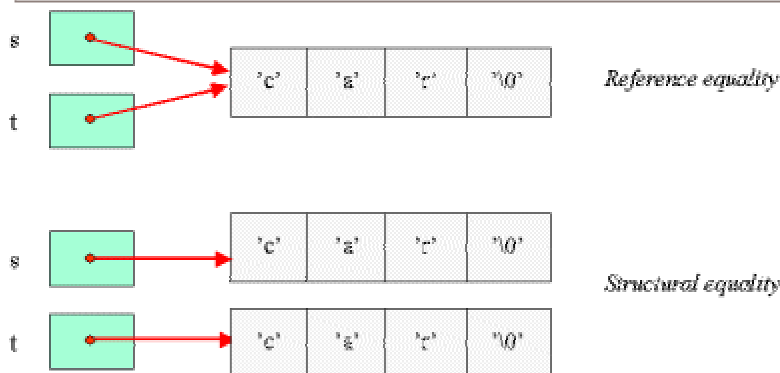
Lektion 7 - slide 12

I dette afsnit ser vi på et eksempel, som generelt illustrerer at lighed mellem to ting kan have flere forskellige betydninger.

Lighed mellem skalartyper (specielt aritmetiske typer), jf. afsnit 18.1, volder sjældent problemer. Lighed mellem sammensatte typer, såsom arrays og strenge, kan have flere forskellige betydninger.

Den lære vi drager i dette afsnit for strenge kan generaliseres til andre sammensatte datastrukturer, f.eks. lister (afsnit 41.1) og records/structs (kapitel 39).

Der er to mulige fortolkninger af lighed af to strenge *s* og *t*



Figur 28.1 *En illustration af reference lighed og strukturel lighed mellem* **char *** *variable*

- Pointerværdierne indeholdt i de to variable **s** og **t** er ens
  - **s** og **t** peger på det samme array af tegn
  - Hvis vi ændrer på tegnene i **s** ændres tegnene i **t** automatisk
  - *Reference lighed*
- **s** og **t** udpeger forskellige arrays, med tegn der parvis er ens
  - Hvis vi ændrer på tegnene i **s** ændres intet i **t**
  - *Strukturel lighed*

Hvis to strenge er *reference ens* er de også *strukturelt ens*, men ikke nødvendigvis omvendt

Ovenstående observation følger naturligt ved at se på figur 28.1. Hvis to strenge `s` og `t` er reference ens vil det naturligvis være tilfældet at tegnene i `s` og `t` er parvis ens. Hvis derimod to strenge `s` og `t` er strukturelt ens, kan vi ikke sluttet at `s` og `t` udpeges af identiske pointere.

28.3. Funktionen strcmp fra string.h

Lektion 7 - slide 14

Funktionen `strcmp` fra standard biblioteket i C er nyttig i forbindelse med vurdering af leksikografisk ordning og strukturel lighed når vi programmerer i C.

Functionen **strcmp** fra **string.h** implementerer den leksikografiske ordning samt

strukturel lighed på tekststrenge i C

- Tre mulige output af **`strcmp(str1,str2)`**:
  - ***Positivt heltal***: **`str1`** > **`str2`**
  - ***Nul***: **`str1`** = **`str2`** (**`str1`** og **`str2`** er ens i strukturel forstand)
  - ***Negativt heltal***: **`str1`** < **`str2`**

Det vil ofte give bedre programmer hvis vi vurderer to strenge med en funktion, som giver et boolsk (true/false) resultat. Med sådanne kan vi skrive `string_leq(s,t)` i stedet for `strcmp(s,t) == -1`. Og i en selektiv kontrolstruktur kan man skrive

```
  if (string_equal(s,t))
    ...
  else
    ...
```

i stedet for

```
  if (strcmp(s,t) == 0)
    ...
  else
    ...
```

Hvis vi ønsker funktioner ala `string_leq` og `string_equal` må vi selv programmere dem, hvilket naturligvis er ganske enkelt.

Vi kender ikke de konkret returværdier af et kald af `strcmp`. Returværdien er altså implementationsafhængig. Dette illustreres klart i program 28.2.

Programmet herunder, program 28.1, viser eksempler på leksikografisk ordning målt med `strcmp`.

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

void pr(char*, char*, int);

int main(void) {

  int b1 = strcmp("book", "shelf");
  int b2 = strcmp("shelf", "book");
  int b3 = strcmp("","book");
  int b4 = strcmp("book","bookshelf");
  int b5 = strcmp("book", "book");
  int b6 = strcmp("7book", "book");
  int b7 = strcmp("BOOK", "book");

  pr("book", "shelf", b1);
```

```
  pr("shelf", "book", b2);
  pr("", "book", b3);
  pr("book", "bookshelf", b4);
  pr("book", "book", b5);
  pr("7book","book", b6);
  pr("BOOK", "book", b7);

  return 0;
}

void pr(char *s, char *t, int r){
 printf("strcmp(\"%s\",\"%s\") = %i\n", s,t,r);
}
```

Program 28.1 *Et program der illustrerer den leksikografiske ordning af tegn i C.*

Output fra program 28.1 kan ses i program 28.2.

```
strcmp("book","shelf") = -17
strcmp("shelf","book") = 17
strcmp("","book") = -98
strcmp("book","bookshelf") = -115
strcmp("book","book") = 0
strcmp("7book","book") = -43
strcmp("BOOK","book") = -32
```

Program 28.2 *Output fra programmet.*

29. Tidligere eksempler

I dette kapitel vil vi genprogrammere et par eksempler fra hhv. afsnit 16.10 og afsnit 18.3.

Vores ærinde er at illustrere hvordan vi kan drage fordel af at overføre tekststrenge som parametre og af at returnere en tekststreng som resultatet af en funktion. Dette vil ofte give mere generelle og mere genbrugbare funktioner.

29.1. Konvertering mellem talsystemer

Lektion 7 - slide 16

I en tidligere lektion har vi programmeret funktioner der konverterer tal mellem forskellige talsystemer

De involverede funktioner læste/skrev input/output med **`getchar`** og **`putchar`**

Det er mere alsidigt at skrive funktioner, der modtager eller returnerer tekststrenge

Herunder, i program 29.1 viser vi en funktion, som konverterer et tal `n` i talsystemet med basis `base` til et decimalt tal. Da tal i f.eks. 16-talsystemet kan gøre brug af alfabetiske cifre ala `'a'` og `'f'` kan vi ikke overføre et heltal som parameter til `to_decimal_number`. Det er derfor bekvemt at overføre en tekststreng, som opfattes som et tal i `n`-talsystemet. Den oprindelige version af funktionen kan ses i program 16.8, `read_in_base`.

```
/* Convert the string n to a decimal number in base and return it.
   Assume that input string is without errors */
int to_decimal_number(char *n, int base){
  int ciffer_number, res = 0;
  char *ciffer_ptr = &n[0], ciffer = *ciffer_ptr;

  do {
    if (ciffer >= '0' && ciffer <= '9')
      ciffer_number = ciffer - '0';
    else if (ciffer >= 'a' && ciffer <= 'z')
      ciffer_number = ciffer - 'a' + 10;
    else ciffer_number = -1;    /* error */

    if (ciffer_number >= 0 && ciffer_number < base)
      res = res * base + ciffer_number;

    ciffer_ptr++; ciffer = *ciffer_ptr;
  }
  while (ciffer != '\0');

  return res;
}
```

Program 29.1 *En funktion der konverterer et tal `n` i `base` talsystemet (en streng) til et decimalt tal.*

I program 29.1 ser vi på det røde sted at `n` er en tekststreng `(char *)`. På det blå sted initialiserer vi en pointer, `ciffer_ptr`, der skal bruges til at løbe igennem strengen `n`. Vi dereferencer straks `ciffer_ptr`, og lægger tegnet over i `char` variablen `ciffer`. Vi bruger `ciffer` i en formel (udtryk), som svarer til formlen i program 16.8. Den brune del opdaterer `ciffer_ptr` og `ciffer`. Den lilla del holder udkig efter nulafslutningen i `n`.

I program 29.2 ses funktionen `to_decimal_number` i et komplet C program. Læg specielt mærke til hvordan vi kalder `to_decimal_number` i `printf`.

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

int read_in_base(int);

int main(void) {
  int i, n, base;
  char *the_number[20];

  for (i = 1; i <= 5; i++){
    printf("Enter number base (a decimal number)"
           "and a number in that base: ");
    scanf("%d %s", &base, the_number);
```

```c
    printf("The decimal number is: %d\n",
            to_decimal_number(the_number, base));
  }

  return 0;
}

/* Convert the string n to a decimal number in base and return it.
   Assume that input string is without errors */
int to_decimal_number(char *n, int base){
  int ciffer_number, res = 0;
  char *ciffer_ptr = &n[0], ciffer = *ciffer_ptr;

  do {
    if (ciffer >= '0' && ciffer <= '9')
      ciffer_number = ciffer - '0';
    else if (ciffer >= 'a' && ciffer <= 'z')
      ciffer_number = ciffer - 'a' + 10;
    else ciffer_number = -1;   /* error */

    if (ciffer_number >= 0 && ciffer_number < base)
      res = res * base + ciffer_number;

    ciffer_ptr++; ciffer = *ciffer_ptr;
  }
  while (ciffer != '\0');

  return res;
}
```

Program 29.2 *Hele programmet.*

29.2. Udskrivning af enumeration konstanter

Lektion 7 - slide 17

I proceduren `prnt_day`, program 18.7, så vi hvordan vi kan lave og udskrive værdierne af enumerators. Herunder, i program 29.3, ser vi hvordan vi kan få hånd på *print værdien* af en enumerator via funktionen `print_name_of_day`. Læg mærke til at `print_name_of_day` returnerer en streng, hvorimod `prnt_day` fra program 18.7 returnerer `void`.

> I en tidligere lektion har vi programmeret funktioner der udskriver navnene på enumeration konstanter
>
> De involverede funktioner skrev output med **`printf`**
>
> Det er mere alsidigt at skrive funktioner, returnerer tekststrenge

```c
/* Return the symbolic name of day d */
char *print_name_of_day(days  d){
  char *result;
  switch (d) {
    case sunday: result = "Sunday";
       break;
    case monday: result = "Monday";
       break;
    case tuesday: result = "Tuesday";
       break;
    case wednesday: result = "Wednesday";
       break;
    case thursday: result = "Thursday";
       break;
    case friday: result = "Friday";
       break;
    case saturday: result = "Saturday";
       break;
  }
  return result;
}
```

Program 29.3 *En funktion som returnerer en symbolsk ugedag (en streng) givet dagens nummer.*

30. Biblioteket string.h

I standard biblioteket findes en række funktioner, der arbejder på tekststrenge. Knap 20 ialt. Du skal inkludere `string.h` for at få adgang til disse. Læs om streng funktionerne i appendix A i *C by Dissection*

30.1. Sammensætning af strenge

Lektion 7 - slide 19

Dette afsnit handler om `strcat` og `strcpy`. Fælles for disse og tilsvarende funktioner er den koncise, korte, og lidt kryptiske navngivning, som er en reminicens fra både C og Unix traditionen.

Funktionen **`strcat`** indsætter en streng i bagenden af en anden streng forudsat der er plads efter nultegnet

Vi viser herunder, i program 30.1, et program, som benytter `strcat` til at sammensætte strengene `"Aalborg"` og `"University"` til `"Aalborg University"`. Læg mærke til at vi bliver nødt til at sammesætte de to bestanddele med et eksplit mellemrum: `" "`.

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

int main(void) {

  char s[25] = "Aalborg";
  char t[] = "University";
  char *res;

  res = strcat(strcat(s," "),t);
  printf("%s: %i chars\n", res, strlen(s));

  return 0;
}
```

Program 30.1 *Et program der illustrerer **strcat** og **strlen**.*

Strengen `s` indeholder god plads til `"Aalborg University"`: 25 tegn ialt. Med to indlejrede kald af `strcat`, på det røde sted, sammensættes strengene til det ønskede resultat. På det blå sted ses et kald af `strlen`, som tæller antallet af tegn i en streng (ikke medregnet nultegnet).

Herunder observerer vi hvordan `strcpy` virker i forhold til `strcat`.

> Funktionen **strcpy** ligner **strcat**, blot indsætter den en streng i begyndelsen af en anden streng

30.2. En alternativ funktion til strcpy

Lektion 7 - slide 20

Som beskrevet ganske kort i afsnit 30.1 kopierer `strcpy` tegnene fra den anden parameter ind i strengen, som overføres i den første parameter.

> Som de fleste andre funktioner i **string.h** allokerers der ikke lager i **strcpy**
>
> Her vil vi programmere en streng-kopierings funktion der allokerer plads til en ny kopi

Vort udgangspunkt er således en observation om, at `strcpy` arbejde inden for allerede allokeret lager. Dette er et generelt kendetegn for mange funktioner i C standard library.

Herunder, i program 30.2 programmerer vi en streng kopieringsfunktion, som allokerer ny og frisk plads til den streng, som overføres som parameter. Den eksisterende streng kopieres over i det nyallokerede lager, således at den eksisterende streng og den nye streng er strukturelt ens (jf. afsnit 28.2).

```
/* Copy s to a fresh allocated string and return it */
char *string_copy(const char *s){

 static char *new_str;

 new_str = (char *)malloc(strlen(s)+1);
 strcpy(new_str,s);

 return new_str;
}
```

Program 30.2 *Et funktioner der allokerer plads til og kopierer en streng.*

På det røde sted ser vi selve lagerallokeringen. Vi bruger dynamisk lagerallokering med `malloc`, jf. afsnit 26.2. Vi assigner pointeren `new_str` til det nye lagerområde. Dernæst foretages kopieringen af `s` over i `new_str` med `strcpy`. Vi erklærer `new_str` som `static` (det lilla sted), idet `new_str` skal returneres som resultat fra funktionen `string_copy`. Dette er ikke strengt nødvendigt, men dog en god ide når vi returnerer en array fra en funktion.

Herunder ser vi på en variant af `str_copy` fra program 30.2. I denne variant er det essentielt at vi på det røde sted erklærer `new_str` som `static`. Årsagen er, at vi i denne variant af programmet statisk har allokeret lokalt lager på køretidsstakken, som ønskes 'med ud fra funktionen' pr. returværdi. I program 30.2 allokerede vi dynamisk lager i det område, vi plejer at benævne som *heapen*.

Hvis vi glemmer 'static' i erklæringen af `new_str` i program 30.3 vil vi få en *dangling reference*, jf. afsnit 26.3. Compileren vil muligvis advare dig, hvis du glemmer `static`.

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


/* Copy s to a locally allocated string and return it */
char *string_copy(const char *s){

 static char new_str[8];    /* static is essential */
 strncpy(new_str,s,7);

 return new_str;
}

int main(void) {

  char s[] = "Aalborg University", *t;
  t = string_copy(s);
  strcpy(s,"---");       /* destroy s */

  printf("The original is: %s.\nThe copy is: %s\n", s, t);

  return 0;
}
```

Program 30.3 *En variant af string_copy der opbygger kopien i lokalt lager. Vi har valgt kun at kopiere 7 tegn.*

Som en mindre detalje i program 30.3 overskriver vi på det sorte sted delvist `s`, som lige er kopieret over i `t`. Dermed kan vi erkende, at `s` og `t` ikke er lig med hinanden i reference forstand, jf. afsnit 28.2.

30.3. Substring ved brug af strncpy

Lektion 7 - slide 21

Vi viser i dette afsnit en typisk anvendelse af `strncpy`, som udtrækker en delstreng af en tekststreng. Bemærk at det ikke er `strcpy`, men `strncpy` vi diskuterer.

> Der er ofte behov for at udtrække en substreng af en anden streng
>
> Dette kan gøres med **`strncpy`**

Strengen vi betragter er `str`, som indeholder `"The Aalborg University basis year"`. På det sorte sted kopierer vi et antal `'\0'` tegn ind i strengen `target`. Dette sikrer i den sidste ende nulafslutningen af `target`. Det røde sted er central for dette eksempel. Bemærk først udtrykket `str+4`, som udpeger `"Aalborg University basis year"`. Den sidste parameter, 18, i kaldet af `strncpy` kopierer netop 18 tegn. Samlet set kopieres `"Aalborg University"` over i `target`.

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

#define LEN 25

int main(void) {

  char str[] = "The Aalborg University basis year";
  char target[25];
  int i;

  for(i = 0; i < LEN; i++) target[i] = '\0';

  strncpy(target, str+4, 18);

  printf("The substring is: %s\nLength: %i\n",
         target, strlen(target));

  return 0;
}
```

Program 30.4 *Et program der udtrækker en substreng af en streng. Det er essentielt at target initialiseres med nultegn, idet det viser sig, at strncpy ikke overfører et nultegn.*

31. Andre emner om tekststrenge

I dette kapitel diskuterer vi fortrinsvis arrays af strenge, herunder det array af strenge som kan overføres fra operativsystemet til `main` i et C program, når vi starter programmet fra en command prompt (shell). Vi tangerer også input/output af string i `printf`/`scanf`.

31.1. Arrays af tekststrenge

Lektion 7 - slide 23

> Et array af tekststrenge kan enten forstås som en to dimensionel **`char`** tabel eller som en pointer til en **`char`** pointer

Herunder, i program 31.1, skal vi først lægge mærke til erklæringen af `numbers`. Vi ser at `numbers` er et array af strenge. `numbers` kan også opfattes som en to-dimensionelt char array. `numbers` initialiseres i samme åndedrag det erklæres, med brug af den notation vi så første gang i program 21.3 i afsnit 21.2.

```
char *numbers[] = {"one", "two", "three"};
char ch1, ch2, ch3, ch4;

ch1 = **numbers;
ch2 = numbers[0][0];
ch3 = *(*(numbers+1) + 1);
ch4 = numbers[2][2];
```

Program 31.1 *Et program der allokerer og tilgår et array af tre tekststrenge.*

Når vi skal følge de fire assignments efter erklæringen af `numbers` i program 31.1 er det meget nyttigt at se på figur 31.1. Figuren viser `numbers`, som referer til et array med tre pointere til strengene `"one"`, `"two"`og `"three"`.

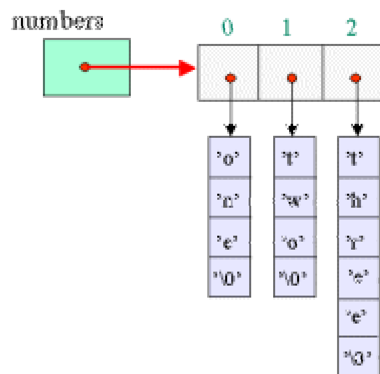
`*numbers` følger den røde pointer i figur 31.1. `**numbers` følger den første sorte pointer til tegnet `'o'`.

`numbers[0]` er cellen som udpeges af den røde pointer. Dette er strengen "one". `numbers[0][0]` er således tegnet 'o'.

`numbers+1` peger på det andet element i det vandrette array. `*(numbers+1)` peger på strengen "two". `*(numbers+1) + 1` peger på strengen "wo". `*(*(numbers+1) + 1)` peger således på tegnet 'w'.

`numbers[2]` peger på strengen "three". `numbers[2][2]` peger på tegnet 'r'.

Det komplette C program i program 31.2 bekræfter gennem outputtet program 31.3 at vore ræsonnementer ovenfor er korrekte.



Figur 31.1 *En illustration af variablen* **`numbers`** *fra ovenstående program*

```
#include <stdio.h>

int main(void) {

  char *numbers[] = {"one", "two", "three"};
  char ch1, ch2, ch3, ch4;

  ch1 = **numbers;
  ch2 = numbers[0][0];
  ch3 = *(*(numbers+1) + 1);
  ch4 = numbers[2][2];

  printf("ch1 = %c, ch2 = %c, ch3 = %c, ch4 = %c\n", ch1, ch2, ch3, ch4);

  return 0;
}
```

Program 31.2 *Hele programmet.*

```
ch1 = o, ch2 = o, ch3 = w, ch4 = r
```

Program 31.3 *Output fra programmet.*

I boksen herunder viser vi hvordan vi kunne have erklæret `numbers` som et to-dimensonelt array.

> Variablen **`numbers`** kunne alternativt erklæres og initialiseres som **`char numbers[][6] = {"one", "two", "three"}`**
>
> Den anden af dagens opgaver handler om et to dimensionelt array af tekststrenge

31.2. Input og output af tekststrenge

Lektion 7 - slide 24

Pointen i dette afsnit er reglerne for, hvordan `"%s"` virker i `printf` og `scanf`. Vi bemærker, at I `scanf` springer vi over white space (mellemrum, eksempelvis). Dernæst læser og opsamler `scanf` tegn indtil der igen mødes white space.

Leg gerne med program 31.4 og bliv dermed fortrolig med `scanf` på strenge.

Input af teststrenge med **`scanf`** har specielle regler

Output af tekststrenge med **`printf`** virker som forventet

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

int main(void) {
  char input[100];

  do {
    printf("Enter a string: ");
    scanf("%s", input);
    printf("You entered \"%s\"\n", input);
  }  while (strcmp(input, "exit"));

  return 0;
}
```

Program 31.4 *Et program der indlæser og udskriver en tekststreng med **`scanf`** og **`printf`**.*

31.3. Programparametre

Lektion 7 - slide 25

Vi slutter lektionen om tekststrenge af med en kig på, hvordan vi kan overføre parametre til `main` fra operativsystemets command prompt eller shell.

Operativsystemet overfører et array af strenge til `main` funktionen med information om, hvordan programmet er kaldt

Dette giver muligheder for at overføre programparametre som tekststrenge

På det røde sted i program 31.5 ser vi parametrene til main. `argc` vil indeholde antallet af programparametre. `argv` vil indeholde et array af tekststrenge, ala strukturen i figur 31.1.

I program 31.6 ser vi med fed sort skrift en aktivering af **`a.out`** (som er det oversatte program) på tre programparametre. Outputtet fra program 31.5 er vist med rødt i program 31.6.

```c
/* Echo the command line arguments. */

#include <stdio.h>

int main(int argc, char *argv[])
{
   int   i;

   printf("\n  argc = %d\n\n", argc);
   for (i = 0; i < argc; ++i)
      printf("   argv[%d] = %s\n", i, argv[i]);
   putchar('\n');
   return 0;
}
```

Program 31.5 *Et program der udskriver de overførte programparametre.*

```
[normark@localhost strings]$ a.out programming in C

  argc = 4

   argv[0] = a.out
   argv[1] = programming
   argv[2] = in
   argv[3] = C
```

Program 31.6 *Input til (fed) og output (rød) fra programmet.*