1. Variable og assignment

Dette er starten af det faglige indhold i første lektion af 'Programmering i C'. Før dette følger et antal mere praktiske slides, som vi ikke har medtaget i denne 'tematiske udgave' af materialet.

Vi starter med at se på variable og assignments. Dette er meget centrale elementer i enhver form for imperativ programmering. Vi slutter dette afsnit med et første kig på datatyper.

1.1. Assignment

Lektion 1 - slide 14

Tænk på variable som små kasser, hvori computeren kan placere forskellige værdier når programmet kører. Ændringer af kassernes værdier foregår gennem de såkaldte assignment kommandoer. På dansk vil vi ofte bruge betegnelsen *tildelinger* for assignments.

> Variable er pladser i lageret, hvis værdier kan ændres ved udførelse af assignment kommandoer.
>
> Alle variable skal erklæres før brug.

Her og i det følgende vil en mørkeblå boks med hvid skrift være præcise definitioner af faglige termer. Her definerer vi 'variabel' og 'assignment'.

> En *variabel* er en navngiven plads i computerens arbejdslager, som kan indeholde en værdi af en bestemt type.
>
> Et *assignment* er en kommando, som ændrer værdien af en variabel.

Tilsvarende vil gule bokse med sort skrift være syntaktiske definitioner. Syntaksen af et programmeringssprog beskriver lovlige strukturer af bestemte udtryksformer. Herunder ser vi at en type (et typenavn) kan efterfølges af et antal (her n) variable. Dette introducerer variablene, hvilket er nødvendigt inden disse kan anvendes til noget som helst. Det angiver også, hvilken slags (hvilken type) af værdier variablene kan antage.

```
type variable₁, variabel₂..., variableₙ;
variableᵢ = expression;
```

Syntaks 1.1

I sidste linie af syntaks boksen ser vi strukturen af et assignment: `var = udtryk`. Vi taler om en variabel på venstre side af = og et udtryk på højresiden. Udtrykket beregnes, og udtrykkets værdi placeres i variablen på venstresiden. C er speciel derved, at `var = udtryk` faktisk er et udtryk, som

har en værdi. I forhold til denne tankegang er = en operator, ligesom + og - . Mere om dette i afsnit 3.1.

I program 1.1 ser vi på et eksempel med variablene `course`, `classes`, `students` og `average_pr_class` alle af typen `int`. Dette er den røde del. Typen `int` betegner integers, altså heltal (negative såvel som positive). I den blå del ser vi et antal assignments til disse variable. De tre første er udprægede initialiseringer (se afsnit 1.2). I det sidste assignment tilskriver vi variablen `average_pr_class` kvotienten mellem `students` og `classes`. Bemærk her at I C er `students / classes` heltalsdivision: Indholdet af `students` (et heltal) heltalsdivideres med indholdet af `classes` (et andet tal). Som et eksempel på heltalsdivision har vi at 7 / 3 er 2. Mere om dette i kapitel 2. Tallet som er gemt i variablen `average_pr_class` udskrives i den grå del. `printf` kommandoer diskuteres i kapitel 4.

```
#include <stdio.h>

int main(void) {

  int course, classes, students, average_pr_class;

  course = 1;
  classes = 3;
  students = 301;

  average_pr_class = students / classes;

  printf("There are %d students pr. class\n", average_pr_class);

  return 0;
}
```

Program 1.1 *Et simplet program med variable og assignments.*

I program 1.2 viser vi en variant af programmet, hvor initialiseringerne af variablene er foretaget sammen med variabelerklæringerne. Sådanne erklæringer og initialiseringer er ofte bekvemme. De to programmer er iøvrigt ækvivalente.

```
#include <stdio.h>

int main(void) {

  int course = 1, classes = 3, students = 301, average_pr_class;

  average_pr_class = students / classes;

  printf("There are %d students pr. class\n", average_pr_class);

  return 0;
}
```

Program 1.2 *En variation af programmet.*

1.2. Initialisering

Lektion 1 - slide 15

Vi omtalte allerede initialiseringer i afsnit 1.1 Her vil vi gøre lidt mere ud af begrebet, og vi vil 'moralisere' over vigtigheden af at initialisere alle variable.

Enhver variabel bør eksplicit tildeles en startværdi i programmet

Ved *initialisering* forstås tilskrivning af en startværdi til en variabel

Initialisering er altså tilskrivning af initielle (start) værdier til de variable, som vi introducerer (erklærer). En god måde at anskueliggøre vigtigheden af initialiseringer kan ses gennem et eksempel, hvor vi undlader at gøre det. I program 1.3 ser vi et sådant eksempel, hvor variablen `classes` er uinitialiseret. Bortset fra den uinitialiserede variabel er programmet næsten identisk med program 1.1. I program 1.4 viser vi programmets output. Værdien af `classes` er et meget stort tal, som vi ikke kan genkende. Forklaringen er, at indholdet er `classes` er det tilfældige lagerindholde, som befinder sig det sted i RAM lageret, hvor variablen `classes` er blevet placeret.

I stedet for at få adgang til tilfældig data i uinitialiserede variable ville det have været bedre at få en særlig udefineret værdi tilbage, en fejlværdi, eller slet og ret at stoppe programmet som følge af at der tilgås en uinitialiseret variabel. Det sker ikke i C, men andre programmeringssprog er mere hjælpsomme i forhold til programmøren, som naturligvis ønsker at opdage den slags fejl i programmet.

```
#include <stdio.h>

int main(void) {

  int course, classes, students, average_pr_class;

  course = 1;
  /* classes uninitialized */
  students = 301;


  average_pr_class = students / classes;

  printf("There are %d students pr. class\n", average_pr_class);
  printf("course: %d, classes: %d, students: %d\n",
         course, classes, students);

  return 0;
}
```

Program 1.3 *Et program med en uinitialiseret variabel.*

```
There are 0 students pr. class
course: 1, classes: 1107341000, students: 301
```

Program 1.4 *Program output.*

Hvis man undlader at initialisere variable risikerer man at der opstår tilfældige fejl, når man kører programmet

1.3. Assignments af formen i = i + 1

Lektion 1 - slide 16

Vi vil nu se på assignments, hvor variablen på venstresiden af assignment symbolet = også optræder i udtrykket på højre side. Mere specifikt forekommer dette i `i = i + 1`.

Der er ofte behov for at tælle en variabel op eller ned.

Dermed optræder variablen både på venstre og højre side af assignmentsymbolet

Herunder forklarer vi trinvis hvad der sker når assignmentet `i = i + 1` udføres i en situation, hvor `i` indledningsvis er 7.

- Antag at **`i`** har værdien 7
- Beregning af **`i = i + 1`**
  - Udtrykket **`i + 1`** på venstre side af assignmentsymbolet beregnes til 8
  - Værdien af variablen **`i`** ændres fra 7 til 8

I program 1.5 har vi et lille program med variablene `i`, `j` og `k` . I de tre farvede assignments tælles disse op og ned. Først tælles `i` op fra 3 til 4 Dernæst tælles `j` ned fra 7 til 3 (idet `i` jo er 4). Endelig gøres `k` 3 gange større (idet `j` jo netop har fået værdien 3). Dermed ender k med at være 27. Når man kører programmet får man output som kan ses i program 1.6.

```c
#include <stdio.h>

int main(void) {

  int i = 3, j = 7, k = 9;

  i = i + 1;
  j = j - i;
  k = k * j;

  printf("i: %d, j: %d, k: %d\n", i, j, k);

  return 0;
}
```

Program 1.5 *Et program med forskellige optællinger og nedtællinger. annotation*

```
i: 4, j: 3, k: 27
```

Program 1.6 *Output fra ovenstående program.*

1.4. Et første blik på datatyper i C

Lektion 1 - slide 17

En datatype er en mængde af værdier. Dette vil vi se meget mere til i en senere del af materialet. I tabel 1.1 ser vi en oversigt over forskellige typer af heltal, reelle tal, tegn og tekststrenge. Vi har allerede mødt typen int tidligere. Disse typer ses i spil i program 1.7 Læg lige mærke til endelserne (suffixes) ala L i 111L. Dette betyder 111 opfattet som en **L**ong.

C understøtter forskellige fundamentale datatyper

	Konkrete typer	**Eksempler på værdier**
Heltal	int short long	10 5 111L
Reelle tal	double float long double	10.5 5.5F 111.75L
Tegn	char	'A' 'a'
Tekststrenge	char *	"Computer" "C"

Tabel 1.1

```
#include <stdio.h>

int main(void) {

  int i = 10;
  short j = 5;
  long k = 111L;

  double x = 10.5;
  float y = 5.5F;
  long double z = 111.75L;

  char c = 'A';
  char d = 'a';

  char *s = "Computer", *t = "C";

  return 0;
}
```

Program 1.7 *Illustration af konkrete typer i et C program.*

Hvis man indleder et tal med et 0 har man bedt om oktal notation. Tilsvarende, hvis man indleder et tal med 0x har man bedt om hexadecimal notation. Vi kommer stærkt tilbage til hvad dette betyder senere i materialet (se afsnit 16.5).

> Foruden de forskellige typer af tal understøtter C også heltal i oktal og hexadecimal notation (eksempelvis `0123` og `0x1bc`).

2. Udtryk og operatorer

Et imperativt program er bygget op af erklæringer, kommandoer og udtryk. Et funktionsorienteret program (ala et ML program) er har ingen kommandoer. I dette afsnit ser vi på udtryk i C.

2.1. Udtryk

Lektion 1 - slide 19

Et udtryk er et stykke program, som vi kan beregne med henblik på at producere en værdi. Denne værdi kan vi 'se på', gemme i en variabel, eller lade indgå i beregning af andre udtryk.

Udtryk i C er dog lidt mere komplicerede end ovenstående udlægning lader ane. C udtryk og C kommandoer lapper ganske kraftigt ind over hinanden. Denne problematik vil blive diskuteret i afsnit 3.1.

> Udtryk er sammen med kommandoer de vigtigste 'byggesten' i ethvert program

Et *udtryk* er et programfragment der beregnes med henblik på returnering af en værdi, som tilhører en bestemt type.

Et udtryk opbygges typisk af operatorer og operander.

Med efterfølgende antagelser vil vi i tabel 2.1 beregne en række udtryk (i venstre søjle). Udtrykkenes værdier vises i højre søjle. Typerne af udtrykkene vises i midten.

Antag at værdien af **`x`** er **`14`** og **`y`** er **`3`**:

Udtryk	Type	Værdi
`7`	Heltal	7
`x`	Heltal	14
`x + 6`	Heltal	20
`x / 3`	Heltal	4
`x % 3`	Heltal	2
`x <= y`	Boolean	*false*
`x + y * 3 >= y % x - 5`	Boolean	*true*

Tabel 2.1 *Forskellige udtryk med udtrykkenes typer og værdier.*

Typerne af et udtryk, som det fremgår af tabellen, udledes fra de erklærede typer af indgående variable, fra typerne af de indgående konstanter, og ud fra de erklærede returtyper af operatorer og funktioner.

På de følgende sider vil vi studere hvordan vi beregner udtryk, som er sammensat af flere deludtryk

2.2. Beregning af sammensatte udtryk

Lektion 1 - slide 20

Sammensatte udtryk er opbygget med brug af to eller flere operatorer. Det betyder, at det sammensatte udtryk kan beregnes på flere forskellige måder.

Problemstilling: Hvordan fortolkes udtryk som er sammensat af adskillige operatorer og operander?

Herunder giver vi to forskellige måder at håndtere beregningsrækkefølgen af sammensatte, og ofte komplicerede udtryk.

- Indsæt parenteser som bestemmer beregningsrækkefølgen
  - `x + y * 3 >= y % x - 5`
  - `(x + (y * 3)) >= ((y % x) - 5)`
- Benyt regler som definerer hvilke deludtryk (operander) der beregnes før andre
  - Multiplikation, division og modulus beregnes før addition og subtraktion
  - Addition og subtraktion beregnes før sammenligning

C prioriterer operatorer i 15 forskellige niveauer. Men det er også muligt - og ofte nødvendigt - at sætte parenteser omkring deludtryk

Det er altid muligt at sætte så mange parenteser, at beregningsrækkefølgen fastlægges. Som bekendt for de fleste beregnes udtryk med parenteser 'indefra og ud'. Sideordnede udtryk beregnes som regel fra venstre mod højre. Der findes programmeringssprog som beror på konsekvent,eksplicit parentessætning, nemlig sprog i Lisp familien. Det er dog en almindelig observation, at mange programmører ikke ønsker at sætte flere parenteser i udtryk end strengt nødvendigt. Dermed er scenen sat for prioritering af operatorer og associeringsregler. Dette ser vi på i det næste afsnit.

2.3. Prioritering af operatorer

Lektion 1 - slide 21

Udtryk som involverer operatorer med høj prioritet beregnes før udtryk med operatorer, der har lav prioritet. I tabel 2.2 viser vi en simplificeret tabel med nogle af de mest almindelige operatorer i C. Lidt senere, i tabel 2.3 gengiver vi den fulde tabel.

Niveau	Operatorer
14	+ unary - unary
13	* / %
12	+ -
10	< <= > >=
9	== !=
5	&&
4	||

Tabel 2.2 *En operatorprioriteringstabel for udvalgte operatorer.*

Af tabellen fremgår for eksempel at udtryk med multiplikative operatorer (altså *, /, og %) beregnes før udtryk med additive operatorer (+ og -).

2.4. Associering af operatorer

Lektion 1 - slide 22

Problemstilling: Prioriteringstabellen fortæller ikke hvordan vi beregner udtryk med operatorer fra samme niveau

Vi ser nu på udtryk som anvender to eller flere operatorer med samme prioritet. Et simpelt eksempel på et sådant udtryk er 1-2-3. Der er to mulige fortolkninger: (1-2)-3 som giver -4 og 1-(2-3) som giver 2. Den første svarer til venstre associativitet, den sidste til højre associativitet. I C og de fleste andre programmeringssprog er - venstre associativ. Det betyder at 1-2-3 er lig med (1-2)-3 som igen er lig med -4.

Lad os nu se på et andet tilsvarende eksempel:

- `10 - 1 - 9 / 3 / 2`
  - Operator prioriteringen fortæller at divisionerne foretages før subtraktionerne
  - Associeringsreglerne fortæller at der sættes parenteser fra venstre mod højre
  - `((10 - 1) - ((9 / 3) / 2))`
  - Resultatet er 8

Vi skriver et simpelt C program, program 2.1, som indeholder ovenstående udtryk. Vi viser også programmets output i program 2.2. Som forventet er `res1` og `res2` ens. `res3` er resultatet af den højre associative fortolkning, og er som sådan forskellige fra `res1` og `res2`.

```c
#include <stdio.h>

int main(void) {

  int res1, res2, res3;

  res1 =  10 - 1 - 9 / 3 / 2;

  res2 =  ((10 - 1) - ((9 / 3) / 2));
  /* left to right associativity */

  res3 =  (10 - (1 - (9 / (3 / 2))));
  /* right to left associativity */

  printf("res1 = %d, res2 = %d, res3 = %d\n", res1, res2, res3);


  return 0;
}
```

Program 2.1 *Et C program med udtrykket* ***`10 - 1 - 9 / 3 / 2`***.

```
res1 = 8, res2 = 8, res3 = 18
```

Program 2.2 *Output fra ovenstående program.*

De fleste operatorer i C associerer fra venstre mod højre

2.5. Operator tabel for C

Lektion 1 - slide 23

Niveau	Operatorer	Associativitet
16	`()` `[]` `->` `.` `++`*postfix* `--`*postfix*	*left to right*
15	`++`*prefix* `--`*prefix* `!` `~` `sizeof(type)` `+`*unary* `-`*unary* `&`*unary,prefix* `*`*unary,prefix*	*right to left*
14	`(type name)` *Cast*	*right to left*
13	`*` `/` `%`	*left to right*
12	`+` `-`	*left to right*
11	`<<` `>>`	*left to right*
10	`<` `<=` `>` `>=`	*left to right*
9	`==` `!=`	*left to right*
8	`&`	*left to right*
7	`^`	*left to right*
6	`|`	*left to right*
5	`&&`	*left to right*
4	`||`	*left to right*
3	`?:`	*right to left*
2	`=` `+=` `-=` `*=` `/=` `>>=` `<<=` `&=` `^=` `|=`	*right to left*
1	`,`	*left to right*

Tabel 2.3 *Den fulde operator prioriteringstabel for C.*

C by Dissection side 611

3. Assignment operatorer i C

Ifølge vore forklaringer i kapitel 1 dækker udtryk og kommandoer to forskellige begreber. Kommandoer (ala assignments) udføres for at ændre programmets tilstand (variablenes værdier). Udtryk beregnes for at frembringe en værdi.

I C er assignments operatorer, og dermed kan assignments indgå i udtryk.

3.1. Blandede udtryk og assignments i C

Lektion 1 - slide 25

> Udtryk i C er urene i forhold til vores hidtidige forklaring af udtryk
>
> Årsagen er at et assigment opfattes som et udtryk i C

Vi illustrerer problemstillingen konkret herunder.

- Antag at `i` har værdien 7
- Programfragmentet `a = i + 5` er et udtryk i C
  - Udtrykket beregnes som `a = (i + 5)`
    - Operatoren `+` har højere prioritet end operatoren `=`
  - Som en *sideeffekt* ved beregningen af udtrykket tildeles variablen `a` værdien 12
  - Værdien af udtrykket `a = i + 5` er 'højresidens værdi', altså 12.

I nedenstående program viser vi 'brug og misbrug' af assignments i C. Det røde assignment udtryk initialiserer variablene `a`, `b`, `c` og `d` til 7. Dette kan anses for en "fiks" forkortelse for 4 assignment kommandoer i sekvens, adskilt af semicolons.

Det blå fragment er mere suspekt. Reelt assignes a til værdien af `2 + 3 - 1` (altså 4). I forbifarten assignes `b` til 2, `c` til 3, og `d` til 1.

```
#include <stdio.h>

int main(void) {

  int a, b, c, d;

  a = b = c = d = 7;  /* a = (b = (c = (d = 7)))  */
  printf("a: %d, b: %d, c: %d, d: %d\n", a,b,c,d);

  a = (b = 2) + (c = 3) - (d = 1);
  printf("a: %d, b: %d, c: %d, d: %d\n", a,b,c,d);

  return 0;
}
```

Program 3.1 *Brug og misbrug af assignments i udtryk.*

Outputtet af program 3.1 vises i program 3.2.

```
a: 7, b: 7, c: 7, d: 7
a: 4, b: 2, c: 3, d: 1
```

Program 3.2 *Output fra ovenstående program.*

Assignment operatorerne har meget lav prioritering, hvilket indebærer at tildelningen af variablen altid sker efter beregningen af 'højresiden'.

Assignment operatorerne associerer fra højre mod venstre.

3.2. Increment og decrement operatorerne

Lektion 1 - slide 26

C understøtter flere forskellige specialiserede assignment operatorer. Principielt er disse ikke nødvendige. Vi kan altid begå os med de former for assignments vi introducerede i kapitel 1 og specielt i afsnit 1.3.

Vi starter med at se på increment og decrement operatorerne. Disse er højt prioriterede, hvilket betyder at de udføres før de fleste andre udtryk (med andre operatorer).

C understøtter særlige operatorer til optælling og nedtælling af variable.

Man kan også bruge ordinære assignments ala `i = i + 1` og `i = i - 1` til dette.

Vi viser herunder ++ i både prefiks og postfix form. Læg mærke til at vi diskuterer både værdien af udtryk ala `++i` og `i++`, og effekten på variablen `i`.

- Antag at variablen `i` har værdien `7`
- Prefix formen: `++i`
  - Tæller først variablen `i` op fra `7` til `8`
  - Returnerer dernæst værdien `8` som resultat af udtrykket
- Postfix formen: `i++`
  - Returnerer først værdien `7` som resultatet af udtrykket
  - Tæller dernæst variablen `i` op fra `7` til `8`

3.3. Eksempel: increment og decrement operatorerne

Lektion 1 - slide 27

I fortsættelse af diskussionen i forrige afsnit viser vi her et konkret eksempel på brug af increment og decrement operatorerne i et C program.

I den blå del tælles `i` op med 1, `j` ned med 1, og `k` op med 1. Det er principielt ligegyldigt om det sker med prefix eller postfix increment operatorer.

I den røde del indgår increment operatorerne i andre assignment operatorer, og det er derfor vigtigt at kunne afgøre værdierne af `--i`, `j++`, og `--k`.

Først tælles `i` ned til 3, som så assignes til `res1`. Dernæst assignes værdien af `j` (som på dette sted i programmet er 3 til) `res2`, hvorefter `j` tælles op til 4. Med dette ændres værdien af variable `j` altså fra 3 til 4. Endelig tælles `k` ned til 5, og den nedtalte værdi assignes til `res3`.

```
#include <stdio.h>

int main(void) {

  int i = 3, j = 4, k = 5;
  int res1, res2, res3;

  i++; j--; ++k;
  /* i is 4, j is 3, k is 6 */

  printf("i: %d, j: %d, k: %d\n", i,j,k);

  res1 = --i;
  res2 = j++;
  res3 = --k;

  printf("res1: %d, res2: %d, res3: %d\n", res1, res2, res3);
  printf("i: %d, j: %d, k: %d\n", i,j,k);

  return 0;
}
```

Program 3.3 *Illustration af increment og decrement operatorerne.*

Outputtet af programmet vises herunder. Vær sikker på at du kan følge ændringerne af de forskellige variable.

```
i: 4, j: 3, k: 6
res1: 3, res2: 3, res3: 5
i: 3, j: 4, k: 5
```

Program 3.4 *Output fra ovenstående program.*

Herefter opsummerer vi vores forståelse af increment og decrement operatorerne.

Sondringen mellem prefix og postfix relaterer sig til tidspunktet for op/nedtælling af variablen relativ til fastlæggelsen af returværdien.

Hvis man bruger increment- og decrementoperatorerne som kommandoer, er det ligegyldigt om der anvendes prefix eller postfix form.

3.4. Andre assignment operatorer

Lektion 1 - slide 28

Foruden increment og decrement operatorerne fra afsnit 3.3 findes der også et antal akkumulerende assignment operatorer, som vi nu vil se på. At akkumulere betyder at 'hobe sig op'. I forhold til variable og assignments hentyder betegnelsen til at vil oparbejder en værdi i variablen, som er beregnet ud fra den forrige værdi ved enten at tælle det op eller ned.

Ligesom increment og decrement operatorerne er de akkumulerende assignment operatorer i bund og grund overflødige. Vi kan sagtens klare os med den basale assignment operator, som beskrevet i kapitel 1. Der kan dog være en lille effektivitetsgevinst ved at bruge increment, decrement, og de akkumulerende assignment operatorer. Mere væsentligt er det dog, at mange programmører bruger disse; og vi skal naturligvis være i stand til at forstå programmer skrevet af sådanne programmører.

C understøtter *akkumulerende assignment operatorer* for +, -, *, / and flere andre binære operatorer

Herunder vises den generelle syntaks af de nye akkumulerende assignment operatorer. *op* kan f.eks. være + eller −.

```
variable op= expressions
```

Syntaks 3.1

I syntaks 3.2 viser vi den basale assignment operator, som beskrevet i afsnit 1.1, i en situation som er ækvivalent med syntaks 3.1.

```
variable = variable op (expression)
```

Syntaks 3.2

Vi illustrerer igen med et konkret C program. I det røde assignment tælles `i` én op. I det blå assignment `j` ned med værdien af i. I det lilla assignment ganges `k` med værdien af `j`. I alle tre tilfælde udfører vi altså et assignment til hhv. `i`, `j` og `k`.

```c
#include <stdio.h>

int main(void) {

  int i = 3, j = 7, k = 9;

  i += 1;       /*  i = i + 1; */
  j -= i;       /*  j = j - i; */
  k *= j;       /*  k = k * j; */

  printf("i: %d, j: %d, k: %d\n", i, j, k);

  return 0;
}
```

Program 3.5 *Et program med forskellige optællinger og nedtællinger - modsvarer tidligere vist program.*

Outputtet af programmet vises herunder.

```
i: 4, j: 3, k: 27
```

Program 3.6 *Output fra ovenstående program.*

Du kan få en oversigt over alle akkumulerende assignment operatorer i C i tabel 2.3, nærmere betegnet i prioritetsklasse 2 (anden række regnet fra bunden af tabellen).

4. Udskrivning og indlæsning

Udskrivning af tegn på skærmen og på filer er vigtig for mange af de programmer vi skriver. Tilsvarende er det vigtigt at kunne indlæse data via tegn fra tastaturet, eller fra tekstfiler. Indlæsning er mere udfordrende end udskrivning, idet vi skal fortolke tegnene som indlæses. Undertiden vil vi sige at disse tegn skal *parses*. Ved udskrivning skal vi formatere data, f.eks. tal, som et antal tegn.

Undervejs i kurset vender vi flere gange tilbage til problematikken. (Den mest grundige behandling gives i kapitel 45). Her og nu introducerer vi det mest nødvendige, så vi kan foretage nødvendig input og output i de programmer, vi skriver i den første del af kurset.

4.1. Udskrivning med printf

Lektion 1 - slide 30

Kommandoen printf udskriver tal, tegn, og tekststrenge i et format, som er foreskrevet af en kontrolstreng. Deraf nok også navnet, print**f** - '**f**' for formattering. For hver procent-tegn, konverteringstegn, i kontrolstrengen skal der være en parameter efter kontrolstrengen. Der er således en en-til-en korrespondance mellem konverteringstegn i kontrolstrengen og parametre, som overføres efter kontrolstrengen. (Vi kommer tilbage til parametre til funktioner i afsnit 12.7).

Kommandoen **`printf`** kan udskrive data af forskellige fundamentale typer.

Formateringen af output styres af en *kontrolstreng*, som altid er første parameter.

I program 4.1 ser vi straks på et eksempel, som er lavet i forlængelse af program 1.7. Værdien af hver af de forskellige variable i program 1.7 skrives ud med `printf`.

I den røde del ser vi at alle heltal kan udskrives med konverteringstegnet d. Vi ser endvidere at værdier i typen short kan angives med hd, hvor h kaldes en 'size modifier' i forhold til konverteringstegnet. Værdien af `k`, som er af typen long, udskrives med en 'size modifier' l til konverteringstegnet d.

I den blå del udskriver vi forskellige reelle tal. Værdien af `x` er af typen double, som udskrives med konverteringstegnet f. Floats kan udskrives med det sammen konverteringstegn. For `z` bruger vi konverteringstegnet e. Med dette beder vi om 'scientific notation'. Bemærk at det er nødvendigt at bruge size modifier L foran e, idet `z` er erklæret som en long double.

I den brune del udskriver vi tegn med konverteringstegnet c, og i den lilla del strenge med s. Dette er uden de store komplikationer.

```
#include <stdio.h>

int main(void) {

  int i = 10;
  short j = 5;
  long k = 111L;
  printf("i: %d, j: %3hd, k: %10ld\n", i, j, k);

  double x = 10.5;
  float y = 5.5F;
  long double z = 111.75L;
  printf("x: %f, y: %5.2f, z: %Le\n", x, y, z);

  char c = 'A';
  char d = 'a';
  printf("c: %c, d: %d\n", c, d);

  char *s = "Computer", *t = "C";
  printf("s: %s,\nt: %s\n", s, t);

  return 0;
}
```

Program 4.1 *Udskrivning af variable af forskellige typer med printf.*

Det er vigtigt at typen af den dataværdi, vi udskriver, svarer nøje til det anvendte konveteringstegn, som er tegnet efter %-tegnet. Output fra ovenstående program kan ses herunder.

```
i: 10, j:   5, k:         111
x: 10.500000, y:  5.50, z: 1.117500e+02
c: A, d: 97
s: Computer,
t: C
```

Program 4.2 *Output fra ovenstående program.*

Vi opsummerer her, ganske overordnet, egenskaberne og mulighederne i printf. Vi har mere at sige om dette i afsnit 18.2, afsnit 18.6 og afsnit 45.2.

- Egenskaber af **`printf`** kommandoen:
  - Indlejring af formaterede parametre i en fast tekst.
  - Én til én korrespondance mellem konverteringstegn og parametre efter kontrolstrengen.
  - Mulighed for at bestemme antal tegn i output, antal cifre før og efter 'komma', venstre/højrestilling, og output talsystem.
  - Antallet af udlæste tegn returneres

4.2. Indlæsning med scanf

Lektion 1 - slide 31

Vi ser her kort og relativt overfladisk på indlæsning med scanf.

Kommandoen **`scanf`** fortolker tegn med henblik på indlæsning af værdier af de fundamentale datatyper

Fortolkning og styring af input sker med en *kontrolstreng*, analogt til kontrolstrengen i **`printf`**.

Her følger et eksempel. Vi indlæser en float i variablen `x`. Dette gøres gentagne gange, i en while løkke. Indlæsningen stopper når der ikke længere er indlæst netop ét tal. I praksis vil løkken nok blive afbrudt, hvis man slet ikke indtaster data, hvilket vil medføre at scanf returnerer 0.

Læg endvidere mærke til hvorledes værdierne af heltallet `cnt` og det reelle tal `sum` udskrives med printf i eksemplet.

```c
/* Sums are computed. */

#include <stdio.h>

int main(void)
{
   int     cnt = 0;
   float   sum = 0.0, x;

   printf("The sum of your numbers will be computed\n\n");
   printf("Input some numbers:  ");
   while (scanf("%f", &x) == 1) {
      cnt = cnt + 1;
      sum = sum + x;
   }
   printf("\n%s%5d\n%s%12f\n\n",
      "Count:", cnt,
      "  Sum:", sum);
   return 0;
}
```

Program 4.3 *Indlæsning og summering af tal.*

- Egenskaber af **`scanf`** kommando:
  - De indlæste data indlæses i angivne lageraddresser
  - Antallet af gennemførte indlæsninger/konverteringer returneres

4.3. Et input output eksempel

Lektion 1 - slide 32

Vi afslutter dette afsnit med et eksempel, som stammer fra C by Dissection. Læg først og fremmest mærke til at vi indlæser `radius` med lf i kontrolstrengen i scanf. Dette svarer altså ikke helt til det tilsvarende konverteringstegn for doubles i printf, som jo kun er f. Vi viser en udgave hvor vi har fremhævet de enkelte bestanddele med farver, så det forhåbentlig bliver lidt lettere at se, hvordan det hele hænger sammen.

```c
#include <stdio.h>

#define PI 3.141592653589793

int main(void)
{
   double radius;

   printf("\n%s\n\n%s",
      "This program computes the area of a circle.",
      "Input the radius:  ");
   scanf("%lf", &radius);
   printf("\n%s\n%s%.2f%s%.2f%s%.2f\n%s%.5f\n\n",
      "Area = PI * radius * radius",
      "     = ",  PI, " * ",  radius, " * ",  radius,
      "     = ", PI * radius * radius );
   return 0;
}
```

Program 4.4 *Samme program med farvede fremhævninger af kontrolstreng og parametre til printf.*

Her ses output fra programmet:

```
This program computes the area of a circle.

Input the radius:  5.3

Area = PI * radius * radius
     = 3.14 * 5.30 * 5.30
     = 88.24734
```

Program 4.5 *Input til og output fra programmet.*

Flere detaljer om scanf følger senere i materialet. I afsnit 18.2 giver vi en nyttig oversigt over heltalstyper og scanf konverteringstegn. Tilsvarende oversigt bliver givet for flydende tal i afsnit 18.6. Endelig diskuterer vi nogle flere scanf detaljer i afsnit 45.3.