17. Typer

I lektionen, som starter med dette kapitel, studerer vi typer i C. Vi har allerede ved flere lejligheder set på typer. I denne lektion vil vi dog være mere systematiske end vi hidtil har været. Vi vil også møde flere nye ting, som har med typer at gøre i C.

17.1. Typer

Lektion 5 - slide 2

Følgende definition fortæller i bund og grund, at en type er en mængde af værdier. Så simpelt er det faktisk.

En *type* er en mængde af værdier med fælles egenskaber

Der er følgende tre gode årsager til at skrive programmer i programmeringssprog, som understøtter typer.

- Typer forbedrer programmets læsbarhed og forståelighed
- Typer gør det muligt for compileren at generere bedre og mere effektiv kode
- Typer gør det muligt at opdage fejl i programmet - typisk under oversættelsen

Nedenstående program er tænkt som illustration af det første og det sidste af disse punkter.

```
#include <stdlib.h>

double f (double p1, int p2){
  if (p2 == 0)
    return p1 * 2.0;
  else
    return p1 - p2;
}

int main(void){
  int i = 1, j = 5;
  double d = 2.0, e = 6.0;

  printf("The first result is %lf\n", f(e-d, i+j));
  printf("The second result is %lf\n", f(d-e, 0));

  return 0;
}
```

Program 17.1 *Et C program med typer.*

Først læsbarhed: Vi kan se, at funktionen `f` altid vil få overført to tal, nemlig først en double (`p1`) og en `int` (`p2`). Vi kan også se at funktionen altid vil returnere en værdi i typen `double`. Denne information er nyttig, når vi skal læse og forstå funktionen `f`.

Alle *udtryk* har en type, som kan bestemmes allerede på det tidspunkt at programmet oversættes. I program program 17.1 er typen af `e-d` `double`, `i+j` er `int`, og typen af `f(e-d, i+j)` er `double`. Hvis ikke disse typer passer ind, det sted, hvor udtrykket er placeret, vil compileren udskrive en fejlbesked.

Hvis vi kalder `f` som `f(i+j, e-d)` har vi principielt en fejlsituation, idet typen af første parameter er et heltal (`int`), og typen af den anden er et reelt tal (`double`). Det er imidlertid således, at C compileren implicit konverterer `i+j` til en `double`, og `e-d` til en `int`. Disse implicitte konverteringer vil vi diskutere i afsnit 19.1.

Det skulle være let at verificere, at programmet ovenfor giver output, som det fremgår herunder.

```
The first result is -2.000000
The second result is -8.000000
```

Program 17.2 *Output fra ovenstående program.*

> En type udgør en klassificering af data som fortæller compilereren hvordan programmøren ønsker at *fortolke data*

Ud over den fortolkning, som typerne bidrager med, finder der yderlig en type fortolkning sted i C. I de to `printf` kald i program 17.1 fortæller konverterings tegnene **`%lf`** at værdien skal fortolkes som en 'long float', hvilket vil sige en `double`. Hvis vi i stedet skrev **`%i`** vil vi få udskrevet et helt andet resultat, som fortolker bitmønstret i de udskrevne udtryks værdier som heltal.

17.2. Typer i C

Lektion 5 - slide 3

Vi opdager ikke så mange typefejl på 'compile time' som vi kunne have ønsket os, når vi programmerer i C. C er meget villig til at omfortolke værdier fra én type til værdier i en anden type. Vi tænker på dette som *løshed* i typesystemet i C.

> ANSI C har en relativ løs skelnen mellem værdier i forskellige typer

- Eksempler på løshed i C's typesystem:
  - Typen boolean er indlejret i de numeriske typer
  - Typen **`char`** er reelt et heltalsinterval
  - Enumeration typer er reelt heltalstyper
    - Vi møder enumeration typer senere i denne lektion

Det traditionelle C sprog har en endnu mere løs omgang med typer end ANSI C

18. Fundamentale C datatyper

I dette kapitel ser vi primært på de fundamentale taltyper. Vi vil også se hvordan taltyperne passer ind i en lidt større sammenhæng. Endvidere vil vi også studere enumeration typer i dette kapitel.

18.1. Oversigt over typer i C

Lektion 5 - slide 5

Herunder giver vi et bud på en hierarkisk klassificering af typerne i C.

- Typer i C
  - Typen **`void`** svarende til den tomme mængde uden værdier.
  - Skalar typer
    - Aritmetiske typer
      - Heltalstyper (integral types)
        - **`short`**, **`int`**, **`long`**, **`char`**
        - enumeration typer
      - Floating-point typer
        - **`float`**, **`double`**, **`long double`**
    - Pointer typer
  - Sammensatte typer (aggregerede typer).
    - Array typer
    - Record typer (**`struct`**ure types)

På det øverste niveau skelner vi mellem skalartyper og sammensatte typer. Endvidere har vi også typen `void` på dette niveau. Typen `void` benyttes ofte når vi vil signalere 'ingen værdi', f.eks. når en funktion ikke tager parametre eller ikke returnerer nogen værdi.

En 'skalar' (engelsk: 'scalar') er en værdi karakteriseret udelukkende ved sin størrelse (having only magnitude). Skalartyperne består af taltyperne, som vi har meget at sige om i resten af dette kapitel. Pointertyper klassificeres også som skalartyper, og de introduceres i kapitel 22.

Vi har endnu ikke set på de sammensatte typer, arrays og records (structs). Arrays behandles mere dybdegående i kapitel 24. Records er et af de sene emner i dette kursus. Vi møder det først i kapitel 39.

18.2. Heltalstyper

Lektion 5 - slide 6

Vi har måske ikke så meget nyt at sige om heltal på dette tidspunkt. Vi vil dog pege på, at heltal findes i tre længder: **`int`**, **`long int`** og **`short int`**. Endvidere findes både signed og unsigned udgaver af disse. På jævn dansk betyder dette med og uden fortegn. Dette kan kombineres til seks

forskellige typer, som vi viser i tabel 18.1. Tabellen medtager også typen **`char`**, som vi allerede i kapitel 16 indså blot repræsenteres som heltal.

Type	Kort typenavn	Suffix	`printf` conv. tegn	`scanf` conv. tegn	Eksempel
`signed int`	`int`	*intet*	`%d` eller `%i`	`%d` eller `%i`	*-123*
`unsigned int`	`unsigned`	u eller U	`%u`	`%u`	*123U*
`signed long int`	`long`	l eller L	`%li`	`%li`	*-123456L*
`unsigned long int`	`unsigned long`	lu eller LU	`%lu`	`%lu`	*123456LU*
`signed short int`	`short`	*intet*	`%hi`	`%hi`	*-12*
`unsigned short int`	`unsigned short`	*intet*	`%hu`	`%hu`	*12U*
`char`	`char`	-	`%c`	`%c`	*'a'* eller *97*

Tabel 18.1

Tabellen ovenfor viser fuld og kort typenavn samt suffix tegn så vi kan typebestemme (de fleste) talkonstanter. Vi viser også hvilke konverteringstegn der skal bruges i printf og scanf for de forskellige typer. Bemærk illustrationen af suffix tegn i den højre eksempel kolonne.

Herunder viser vi typerne fra tabel 18.1 i et konkret C program.

```
#include <stdio.h>

int main(void) {
  int i = -123;
  unsigned ui = 123U;
  long l = -123456L;
  unsigned long ul = 123456LU;
  short s = -12;
  unsigned short us = 12U;
  char c = 97;

  printf("int: %i, unsigned: %u, long: %li, unsigned long: %lu\n",
         i, ui, l, ul);
  printf("short: %hi, unsigned short: %hu, char: %c\n",
         s, us, c);

  printf("Enter integer numbers: ");
  scanf(" %i %u %li %lu %hi %hu %c", &i, &ui, &l, &ul, &s,
        &us, &c);

  printf("int: %i, unsigned: %u, long: %li, unsigned long: %lu\n",
         i, ui, l, ul);
  printf("short: %hi, unsigned short: %hu, char: %c\n", s, us, c);

  return 0;
}
```

Program 18.1 *Et C program som illustrerer ovenstående.*

I program 18.2 viser vi hvordan man finder ud af bytestørrelsen af de forskellige typer fra tabel 18.1. Outputtet fra programmet kan ses i program 18.3. Da C ikke har standardiseret taltyperne kan man opleve andre output, hvis programmet kører på en anden maskine.

```
/* Compute the size of some fundamental types. */

#include <stdio.h>

int main(void)
{
   printf("\n");
   printf("Here are the sizes of some integral types:\n\n");

   printf("           int:%3d bytes\n", sizeof(int));
   printf("      unsigned:%3d bytes\n", sizeof(unsigned));
   printf("          long:%3d bytes\n", sizeof(long));
   printf(" unsigned long:%3d bytes\n", sizeof(unsigned long));
   printf("         short:%3d bytes\n", sizeof(short));
   printf("unsigned short:%3d bytes\n", sizeof(unsigned short));
   printf("          char:%3d byte \n", sizeof(char));

   printf("\n");
   return 0;
}
```

Program 18.2 *Et program der 'udregner' bytestørrelsen af heltalstyperne.*

```
Here are the sizes of some integral types:

           int:  4 bytes
      unsigned:  4 bytes
          long:  4 bytes
 unsigned long:  4 bytes
         short:  2 bytes
unsigned short:  2 bytes
          char:  1 byte
```

Program 18.3 *Output fra programmet.*

Endelig viser vi program 18.4 hvordan man kan få oplyst mindste og største værdier i forskellige heltalstyper. De røde konstanter i program 18.4 stammer fra header filen `limits.h`. Når man kører programmet på min maskine får man grænserne, som vises i program 18.5.

```c
#include <stdio.h>
#include <limits.h>
int main(void) {

  printf("Minimum int:            %12i  Maximum int:           %12i\n",
         INT_MIN, INT_MAX);

  printf("Minimum unsigned int:   %12u  Maximum unsigned int:  %12u\n",
         0, UINT_MAX);

  printf("Minimum long:           %12li  Maximum long:          %12li\n",
         LONG_MIN, LONG_MAX);

  printf("Minimum unsigned long:  %12lu  Maximum unsigned long: %12lu\n",
         0, ULONG_MAX);

  printf("Minimum short:          %12hi  Maximum short:         %12hi\n",
         SHRT_MIN, SHRT_MAX);

  printf("Minimum unsigned short: %12hu  Maximum short:         %12hu\n",
         0, USHRT_MAX);

  return 0;
}
```

Program 18.4 *Et program der tilgår konstanter i limits.h.*

```
Minimum int:             -2147483648  Maximum int:             2147483647
Minimum unsigned int:              0  Maximum unsigned int:    4294967295
Minimum long:            -2147483648  Maximum long:            2147483647
Minimum unsigned long:             0  Maximum unsigned long:   4294967295
Minimum short:                 -32768  Maximum short:               32767
Minimum unsigned short:            0  Maximum short:               65535
```

Program 18.5 *Output fra programmet.*

18.3. Enumeration types (1)

Lektion 5 - slide 7

Enumeration types kan anvendes i situationer, hvor vi har brug for et mindre antal navngivne heltal, som spiller bestemte og veldefinerede roller i et program.

En *enumeration type* er en endelig mængde af heltal som er knyttet til enumeration konstanter

En *enumeration konstant* (enumerator) er et navn, som på mange måder ligner en variabel

Som sædvanlig viser vi den syntaktiske komposition af nye sproglige elementer. Det viste er type erklæringer, hvor vi under betegnelsen `enum tag` henviser til en type med bestemte navne, der har heltalsværdier.

```
enum tag {name1, name2, ... namei}

enum tag {name1=expr1, name2=expr2, ... namei=expri}
```

Syntaks 18.1 *Syntaktisk definition af to mulige former af enumeration typer i C*

Lad os forstå enumeration typer gennem et eksempel, se program 18.6. Under betegnelsen `enum days` gemmer sig en type, som netop består af navnene `sunday`, `monday`, ... `saturday`. Værdien af navnet `sunday` er 0, værdien af `monday` er 1, ..., og værdien af `saturday` er 6.

Typen `enum days` bruges i program 18.6 i funktionen `next_day_of`. Denne funktion tager en dag som input, og returnerer den efterfølgende dag som output. Bemærk 'cyklen' der består i, at `next_day_of(saturday)` er lig med `sunday`.

```
enum days {sunday, monday, tuesday, wednesday, thursday,
           friday, saturday};

enum days  next_day_of(enum days  d){
  enum days next_day;
  switch (d){
    case sunday: next_day = monday;
      break;
    case monday: next_day = tuesday;
      break;
    case tuesday: next_day = wednesday;
      break;
    case wednesday: next_day = thursday;
      break;
    case thursday: next_day = friday;
      break;
    case friday: next_day = saturday;
      break;
    case saturday: next_day = sunday;
      break;
  }
  return next_day;
}
```

Program 18.6 *En enumeration type* ***enum days*** *og en funktionen* ***next_day_of***.

I program 18.7 vises en procedure (som en void C funktion), der givet en dag som input udskriver denne dag på standard output via `printf`.

Implementationen af `prnt_day` illustrerer, at vi ikke har direkte adgang til det symbolske navn af en enumeration konstant. Hvis vi blot udskriver værdien `sunday` vil vi se værdien 0, og ikke "sunday". Derfor må vi lave en `switch` kontrolstruktur, som på baggrund af en **`enum days`** værdi udskriver en passende tekst.

Lad mig også her nævne, at man ikke kan læse enumerators med `scanf`. Man skal indlæse heltal, og evt. internt typekonvertere disse en enumeration værdi. Typekonvertering (type casting) bliver diskuteret i afsnit 19.2. (Under opgaveregning ved øvelserne i 5. lektion 2004 så jeg en del studerende, som havde forventet at kunne indlæse enumerators via scanf. Men den går altså ikke).

```
void prnt_day(enum days d){
  switch (d) {
    case sunday: printf("Sunday");
       break;
    case monday: printf("Monday");
       break;
    case tuesday: printf("Tuesday");
       break;
    case wednesday: printf("Wednesday");
       break;
    case thursday: printf("Thursday");
       break;
    case friday: printf("Friday");
       break;
    case saturday: printf("Saturday");
       break;
  }
}
```

Program 18.7 *En funktion der udskriver det symbolske navn på en dag.*

Der findes en version af programmet, hvor de to funktioner fra program 18.6 og program 18.7 er sat sammen med et hovedprogram, `main`. Af pladshensyn er den ikke vist her. Se Det samlede program - inklusive main på den tilhørende slide.

I et C program bidrager anvendelse af enumeration typer primært til større læsbarhed - og lettere programforståelse

18.4. Enumeration types (2)

Lektion 5 - slide 8

I dette afsnit ser vi på et antal regler som knytter sig til enumeration typer i C. Reglerne knytter sig til syntaks 18.1. Og vi ser på endnu et eksempel.

- Regler om betydningen af enumeration typer og konstanter
  - Enumeration konstanter har samme status som variable og må som sådan kun defineres én gang i det samme scope
  - I det første tilfælde tildeles *name1* værdien *0*, *name2* værdien *1*, etc.
  - I det andet tilfælde bestemmer programmøren hvilke heltalsværdier de enkelte enumeration konstanter tildeles
    - Der er mulighed for at to eller flere konstanter i samme enumeration type har samme værdi

I eksemplet herunder anvender vi to enumeration typer, `enum grade_simple` og `enum grade_13`. Disse afspejler karaktererne på de to skalaer bestået/ikke bestået og den danske 13 skala.

Vi ser at enumeratorerne på 13 skalaen bliver afbildet på de heltal, som karaktererne naturligt står for. Eksempelvis skriver vi `nul_nul = 0` for at binde enumeratoren `nul_nul` til 0.

```
#include <stdio.h>

enum grade_simple {not_passed, passed};

enum grade_13 {nul_nul = 0, nul_tre = 3, fem = 5, seks,
               syv, otte, ni, ti, elleve, tretten = 13};

enum grade_simple convert_grade_13_to_simple_grade
                                        (enum grade_13 g){
  enum grade_simple result;

  if (g <= fem)
    result = not_passed;
  else
    result = passed;

  return result;
}

void prnt_grade_simple(enum grade_simple g){
  switch (g) {
    case not_passed: printf("Not passed");
      break;
    case passed: printf("Passed");
      break;
  }
}

int main(void){

  int grade_number;

  printf("Enter '13 skala' grade: ");
  scanf(" %d", &grade_number);

  prnt_grade_simple(
         convert_grade_13_to_simple_grade(grade_number));
  printf("\n");

  return 0;
}
```

Program 18.8 *Et eksempel på et program som bruger enumeration typer til karakterskalaer.*

I karaktereksemplet i program 18.8 kan man klart diskutere, om introduktionen af de symbolske navne for karakterværdierne på 13 skalaen gør en nævneværdig forskel. En konsekvent anvendelse beskytter os dog mod at bruge meningsløse karakterværdier, så som -1, 4, 12 og 14.

Anvendelsen af `not_passed` og `passed` (for ikke-bestået og bestået) i stedet for 0 og 1 bidrager klart til bedre læsbarhed og forståelighed. Vi kan sige at det bringer vores program lidt tættere på virkelighedens begrebsverden.

18.5. Enumeration typer i andre sprog

Lektion 5 - slide 9

Der findes programmeringssprog, der på en mere klar og ren måde end C understøtter enumeration typer. Der findes også nyere sprog, som slet ikke har særlig understøttelse af enumeration typer.

Enumeration typer i C er anderledes end tilsvarende typer i de fleste andre programmeringssprog

- Enumeration typer i Pascal-lignende sprog:
  - Værdien af enumeration konstanterne er nye, entydige værdier - ikke heltal
  - Ordningen af enumeration konstanterne er af betydning
    - *Efterfølger* og *forgænger* funktioner af enumeration konstanter giver mening

Java har ikke overtaget enumeration typerne fra C

18.6. Floating point typer (1)

Lektion 5 - slide 10

I dette afsnit giver vi en oversigt over floating point typer i C, efter samme mønster som vi diskuterede for heltalstyper i afsnit 18.2.

Type	Suffix	`printf` conv. tegn	`scanf` conv. tegn	Eksempel
`float`	f eller F	`%f`	`%f`	*5.4F*
`double`	*intet*	`%f`	`%lf`	*5.4*
`long double`	l eller L	`%Lf`	`%Lf`	*5.4L*

Tabel 18.2

Bemærk lige `scanf` konverterings- og modifier tegnene `%lf` for indlæsning af double. I forhold til det tilsvarende `printf` konverteringstegn `%f` kan brug af `%lf` (i betydningen `l`ong `f`loat) virke underlig.

Nedenstående program viser mulighederne i tabel 18.2 i et konkret C program.

```c
#include <stdio.h>

int main(void) {

  float f = 123.456F;
  double d = 123.456;
  long double ld = 123.456L;

  printf("float: %f, double: %f, long double: %Lf\n", f, d, ld);

  printf("Enter new values: ");
  scanf(" %f %lf %Lf",&f, &d, &ld);
  printf("float: %f, double: %f, long double: %Lf\n", f, d, ld);

  return 0;
}
```

Program 18.9 *Et C program som illustrerer ovenstående.*

Ligesom i program 18.2 viser vi herunder hvordan vi kan bestemme bytestørrelsen af typerne `float`, `double` og `long double`. Det selvforklarende output fra programmet vises i program 18.11. Som omtalt tidligere er det ikke sikkert, at du får det samme output hvis du kører programmet på din computer.

```c
/* Compute the size of some fundamental types. */

#include <stdio.h>

int main(void)
{
   printf("\n");
   printf("Here are the sizes of some floating types:\n\n");

   printf("      float:%3d bytes\n", sizeof(float));
   printf("     double:%3d bytes\n", sizeof(double));
   printf("long double:%3d bytes\n", sizeof(long double));

   printf("\n");
   return 0;
}
```

Program 18.10 *Et program der 'udregner' bytestørrelsen af float typerne.*

```
Here are the sizes of some floating types:

      float:  4 bytes
     double:  8 bytes
long double: 12 bytes
```

Program 18.11 *Output fra programmet.*

Ligesom i program 18.4 viser vi herunder grænserne for værdierne i typerne `float`, `double` og `long double`. Bemærk at der er tale om mindste og største *positive* tal i typerne. Output af programmet program 18.12 ses i program 18.13.

```c
#include <stdio.h>
#include <float.h>
int main(void) {

  printf("Minimum float:        %20.16e  \nMaximum float:        %20.16e\n\n",
         FLT_MIN, FLT_MAX);

  printf("Minimum double:       %20.16le \nMaximum double:       %20.16le\n\n",
         DBL_MIN, DBL_MAX);

  printf("Minimum long double:  %26.20Le \nMaximum long double:  %26.20Le\n\n",
         LDBL_MIN, LDBL_MAX);


  return 0;
}
```

Program 18.12 *Et program der tilgår konstanter i floats.h.*

```
Minimum float:         1.175494e-38
Maximum float:         3.402823e+38

Minimum double:        2.2250738585072014e-308
Maximum double:        1.7976931348623157e+308

Minimum long double:   3.36210314311209350626e-4932
Maximum long double:   1.18973149535723176502e+4932
```

Program 18.13 *Output fra programmet.*

I outputtet fra program 18.12, som er vist herover, kan vi notere os at vi i `printf` kaldene har taget hensyn til den *præcision*, som understøttes af hhv. `float`, `double` og `long double`. Præcision er et af de emner vi ser på i næste afsnit, afsnit 18.7.

18.7. Floating point typer (2)

Lektion 5 - slide 11

Efter nu at have brugt tid og kræfter på at forstå de tre floating point typer `float`, `double` og `long double` vil vi nu se på *notation* for flydende tal, *præcisionen* af værdierne i de tre typer, *intervallet* der dækkes af typerne.

- *Notation*
  - Decimal: 12345.6789
  - Eksponential: 0.1234567e-89
  - En flydende tal konstant skal enten indeholde *decimal punktet* eller *eksponential delen* (eller begge)
- *Præcision*: Antallet af signifikante cifre i tallet
  - **`float`**: typisk 6 cifre
  - **`double`**: typisk 15 cifre
  - **`long double`**: typisk 20 cifre

- *Interval* (range): Største og mindste mulige *positive* værdier
  - **`float`**: Typisk 10^{-38} til 10^{38}
  - **`double`**: Typisk 10^{-308} til 10^{308}
  - **`long double`**: Typisk 10^{-4932} til 10^{4932}

Dette afslutter vores dækning af de fundamentale heltals og floating point typer i C. Relativ til oversigten i afsnit 18.1 mangler vi at dække pointertyper og sammensatte typer. Pointertyper tager vi hul på i kapitel 22. Arrays behandles i kapitel 24, kapitel 25 og kapitel 40. Structs kommer på banen i kapitel 39.

19. Typekonvertering og typedef

Der er undertiden behov for at konvertere en værdi i én type til en værdi i en anden type. I nogle situationer er det programmøren, som ønsker dette. I så fald taler vi om en *eksplicit typekonvertering*, eller i teknisk jargon for en *cast*. I andre tilfælde er det C compileren der føler trang til at foretage en typekonvertering. Disse omtales som *implicitte typekonverteringer*. I afsnit 19.1 ser vi først på de implicitte typerkonverteringer.

Som et andet emne i dette kapitel ser vi på brug af `typedef` til omnavngivning af eksisterende typer.

19.1. Implicit typekonvertering

Lektion 5 - slide 13

Vi giver en ganske kortfattet oversigt over tre former for implicit typekonvertering herunder.

C foretager en række typekonverteringer 'bag ryggen på programmøren'

- *"Integral promotion"*
  - Udtryk af heltalstype hvori der indgår **`short`** eller **`char`** værdier konverteres til en værdi i typen **`int`**.
  - Eksempel: Hvis x og y er af typen **`short`** er værdien af x + y **`int`**.
- *"Usual arithmetic conversions" - widening*
  - Konvertering af en mindre præcis værdi til en tilsvarende mere præcis værdi således at begge operander får samme type
  - Der mistes ikke information og præcision
- *Narrowing - demotion*
  - Konvertering af mere præcis værdi til en mindre præcis værdi
  - Der mistes information

Integral promotion benyttes bl.a. til at få beregninger på `shorts` og `chars` til at forgå inden for værdier i typen int.

Brug af widening er bekvem, når vi skriver udtryk der involverer værdier i to forskellige typer. Eksempelvis `d + i`, hvor `d` er en `double` og `i` er en `int`. Maskinen understøtter sandsynligvis ikke en addition af netop disse to typer, så derfor konverteres den mest snævre operand, her `i`, til typen af den mest brede operand. Altså konverteres værdien af `i` til en `double`, hvorefter der kan udføres en addition af to `double` værdier.

Narrowing udføres f.eks. når en `double` værdi assignes til en integer variable.

De tre forskellige former for typekonverteringer er illustreret i program 19.1.

```
#include <stdio.h>

int main(void) {

  short s = 12; char c = 'a';
  double d = 123456.7; float f = 4322.1;
  int i;

  printf("c - s = %i is converted to int\n", c - s);
  /* The type of c - s is promoted to int */

  printf("d + f = %f is converted to a double\n", d + f);
  /* f is converted to double before adding the numbers */

  i = d;
  printf("In assigning d to i %f is demoted to the int %i\n", d, i);
  /* d is converted to an int - information is lost */

  return 0;
}
```

Program 19.1 *Eksempler på implicitte typekonverteringer.*

19.2. Eksplicit typekonvertering

Lektion 5 - slide 14

Som programmører har vi en gang imellem brug for at gennemtvinge en konvertering af en værdi fra en type til en anden. Ofte er konverteringen dog "spil for galleriet", i den forstand at vi blot garanterer over for kompileren at værdien kan opfattes som værende af en bestemt type. I disse situationer finder der ingen reel forandring sted som følge af en typekonvertering.

I C er det muligt for programmøren at konvertere værdien af et udtryk til en anden type ved brug af en såkaldt *cast operator*

En typecast noteres ved at foranstille det udtryk, hvis værdi ønskes konverteret, i en typecase operator. En typecast operator er et typenavn i parentes.

At en typecast faktisk er en operator kan man overbevise sig om ved at studere niveau 14 i tabel 2.3.

```
(typeName) expression
```

Syntaks 19.1 *Syntaktisk definition af casting - eksplicit typekonvertering i C*

I program 19.2 viser vi hvordan værdien af `'A'+10` kan konverteres til en `long`. Vi ser også hvordan værdien af `long` variablen `y` kan konverteres til en `int`. Bemærk at `y` konverteres før additionen, på grund af den indbyrdes placering af typecast og addition i tabel 2.3. Endelig ser vi hvordan værdien af udtrykket `x=77`, altså heltallet 77, kan konverteres til en `double` før assignmentet til `z`.

```
#include <stdio.h>

int main(void) {

  long y;
  float x;
  double z;

  y = (long) ('A' + 10);
  x = (float) ((int) y + 1);
  z = (double) (x = 77);

  printf("y: %li, x: %f, z: %f", y, x, z);
  return 0;
}
```

Program 19.2 *Et program med eksempler på casts.*

I det næste eksempel, program 19.3, ser vi hvordan funktionen `next_day_of`, som oprindelig blev vist i program 18.6, kan omskrives. I den omskrevne, og meget kortere udgave af `next_day_of`, benytter vi to casts. Den blå konvertering transformerer en `enum days` værdi til en `int`. Til dette heltal adderer vi 1 og vi uddrager resten ved division med 7. Tænk lige over hvorfor... Den røde konvertering bringer det resulterende heltal tilbage som en `enum days` værdi.

```
enum days next_day_of(enum days d){
  return (enum days) (((int) d + 1) % 7);
}
```

Program 19.3 *Funktionen* ***`next_day_of`*** *omskrevet med brug af casts.*

19.3. Navngivne typer med typedef

Lektion 5 - slide 15

I nogle situationer kan vi have lyst til at give en eksisterende type et nyt navn. Dette gør sig specielt gældende for typer med ubekvemme betegnelser, så som `enum days`, `enum grade_13`, og `enum simple_grade`.

I C er det muligt at tildele en type et bestemt navn

I tilfælde af komplicerede typer kan dette øge læsbarheden af et program

Syntaksen for typedef er enkel. Blot skal man lige bemærke, at det nye typenavn angives til sidst.

```
typedef oldType newType
```

Syntaks 19.2 *Syntaktisk definition af casting - eksplicit typekonvertering i C*

Vi viser herunder hvordan program 18.6 og 18.7 kan omskrives til program 19.4. Ændringerne er ikke revolutionerende, men læg alligevel mærke til de blå og røde steder i forhold til program 19.4.

```
#include <stdio.h>

enum days {sunday, monday, tuesday, wednesday, thursday,
           friday, saturday};
typedef enum days days;

days  next_day_of(days  d){
  return ( days ) (((int) d + 1) % 7);
}

void prnt_day(days  d){
  switch (d) {
    case sunday: printf("Sunday");
       break;
    case monday: printf("Monday");
       break;
    case tuesday: printf("Tuesday");
       break;
    case wednesday: printf("Wednesday");
       break;
    case thursday: printf("Thursday");
       break;
    case friday: printf("Friday");
       break;
    case saturday: printf("Saturday");
       break;
  }
}

int main(void){
```

```
  days  day1 = saturday,  another_day;
  int i;

  printf("Day1 is %d\n", day1);

  printf("Day1 is also "); prnt_day(day1); printf("\n");

  another_day = day1;
  for(i = 1; i <= 3; i++)
    another_day = next_day_of(another_day);

  printf("Three days after day1: ");  prnt_day(another_day);
  printf("\n");

  return 0;
}
```

Program 19.4 *En omskrivning af ugedags programmet som benytter typedef.*

Det kan måske forvirre lidt, at vi bruger `days` både som navnet på den nye type og som en del af navnet på den eksisterende type. Teknisk set er *tag navnet* `days` i `enum days` og det nye typenavn `days` i to forskellige navnerum. Derfor generer de ikke hinanden.

20. Scope og storage classes

Scope begrebet hjælper os til at forstå, hvor et bestemt navn i program er gyldigt og brugbart. Storage classes er en sondring i C, som bruges til at skelne mellem forskellig lagringer og forskellige synligheder af variable, funktioner, mv.

20.1. Scope

Lektion 5 - slide 17

Scope reglerne bruges til at afgøre i hvilke programdele et navn er gyldigt.

På dansk kan vi bruge ordet *'virkefelt'* i stedet for *'scope'*.

I store programmer bruges det samme navn ofte i forskellige scopes med forskellige betydninger.

Scope af et navn er de dele af en programtekst hvor navnet er kendt og tilgængeligt

Følgende beskriver den vigtigste af alle scoperegler af alle i C.

- De mest basale scoperegler i C - storage class **`auto`**
  - Scopet af et navn er den blok hvori den er erklæret

- Introducerer *huller i scope*

Den bedste måde at forstå reglerne, som blev omtalt herover, er at bringe dem i spil på et godt eksempel. Et sådant er vist i program 20.1. Nogle vil nok genkende dette eksempel fra afsnit 7.1 (program 7.1) hvor vi diskuterede sammensatte kommandoer.

```
#include <stdio.h>

int main(void) {

  int a = 5, b = 7, c;

  c = a + b;

  {
    int b = 10, c;

    {
      int a = 3, c;
      c = a + b;
      printf("Inner:  a + b = %d + %d = %d\n", a, b, c);
    }

    c = a + b;
    printf("Middle: a + b = %d + %d = %d\n", a, b, c);
  }

  c = a + b;
  printf("Outer:  a + b = %d + %d = %d\n", a, b, c);

  return 0;
}
```

Program 20.1 *Illustration af scope i tre indlejrede blokke.*

Der vises tre blokke i `main`, som indlejres i hinanden. I den yderste blok har vi variablene `a`, `b` og `c`. I den mellemste blok har vi `b` og `c`, som skygger for `b` og `c` fra den yderste blok. I den inderste blok har vi `a` og `c`, som igen skygger for hhv. `a` (fra yderste blok) og `c` (fra den mellemste blok). Skygningen medfører det vi kalder et "hul i scopet".

Bemærk at vi kan se fra indre blokke ud i ydre blokke, men aldrig omvendt.

Du skal kunne forklare det output, som kommer fra program 20.1. Hvis du kan, har du givetvis styr på blokke og storage class auto. Vi har mere at sige om sidstnævnte - automatiske variable i C - i næste afsnit.

```
Inner:  a + b = 3 + 10 = 13
Middle: a + b = 5 + 10 = 15
Outer:  a + b = 5 + 7 = 12
```

Program 20.2 *Output fra programmet.*

20.2. Storage class auto

Lektion 5 - slide 18

Der findes et lille antal forskellige former for såkaldte storage classes i C, som er vigtige for fastlægningen af scope af de forskellige navngivne enheder i programmet.

Storage class af variable og funktioner medvirker til bestemmelse af disses scope

Storage classes angives som en modifier før selve typenavnet i en erklæring:

```
storageClass type var1, var2, ..., vari
```

Syntaks 20.1 *Syntaktisk definition af storage classes i variabel erklæringer*

Variable med storage class *auto* kaldes automatiske variable

En *automatisk variabel* er lokal i en blok, dvs den skabes når blokken aktiveres og nedlægges når blokken forlades

Essentielt set illustrerede vi allerede storage class auto i program 20.1 i afsnittet ovenfor.

- Storage class **`auto`**
  - Default storage class i blokke, herunder kroppe af funktioner
    - Hvis man ikke angiver storage class af variable i blokke er den således **`auto`**
  - Man kan angive storage class **`auto`** eksplicit med brug af nøgleordet **`auto`**

`register` storage class er en variant af **`auto`** storage class

Sammenfattende kan vi sige, at storage class auto sammenfatter lokale variable, som oprettes ved indgang i en blok og nedlægges ved udgang af en blok.

20.3. Storage class static af lokale variable

Lektion 5 - slide 19

En automatisk variabel ophører med at eksistere når den omkringliggende blok ophører med at eksistere. Dette kan f.eks. ske når en funktion returnerer.

Hvis vi ønsker at en lokal variabel overlever blok ophør eller funktionsreturnering kan vi gøre den global (ekstern, se afsnit 20.4). Men vi kan også lave en lokal variabel af storage class static.

I blokke er storage class **`static`** et alternativ til storage class **`auto`**

En *statisk variabel* i en blok beholder sin værdi fra en aktivering af blokken til den næste

- Storage class static - lokale variable i en blok
  - Skal angives med **`static`** som storageClass
  - Variabel initialiseringen udføres ved første aktivering

Lad os illustrere statiske lokale variable med følgende eksempel. Vi ser en funktion `accumulating_f`, som kaldes 10 gange fra `main`. I alle kald overføres parameteren 3 til `accumulating_f`.

I funktionen `accumulating_f` ser vi med rødt den statiske variabel `previous_result`, hvori vi gemmer den forrige returværdi fra kaldet af `accumulating_f`. Kun når funktionen kaldes første gang initialiseres `previous_result` til 1. (Dette er en vigtig detalje i eksemplet). Hvis ikke `previous_result` er 1 multipliceres `previous_result` med funktionens input, og denne størrelse returneres.

Output fra funktionen vises i program 20.4. Hvis vi sletter ordet **`static`** fra program 20.3 ser vi let, at alle de 10 kald returnerer tallet 1.

```
#include <stdio.h>

int accumulating_f (int input){
  int result;
  static int previous_result = 1;

  if (previous_result == 1)
    result = input;
  else
    result = previous_result * input;

  previous_result = result;
  return result;
}

int main(void) {
  int i;

  for (i = 0; i < 10; i++)
    printf("accumulating_f(%d) = %d\n", 3, accumulating_f(3));

  return 0;
}
```

Program 20.3 *Illustration af statiske lokale variable - en funktion der husker forrige returværdi.*

```
accumulating_f(3) = 3
accumulating_f(3) = 9
accumulating_f(3) = 27
accumulating_f(3) = 81
accumulating_f(3) = 243
accumulating_f(3) = 729
accumulating_f(3) = 2187
accumulating_f(3) = 6561
accumulating_f(3) = 19683
accumulating_f(3) = 59049
```

Program 20.4 *Output fra programmet.*

Statiske lokale variable er et alternativ til brug af globale variable.

En statisk lokal variabel bevarer sin værdi fra kald til kald - men er usynlig uden for funktionen.

Man kunne få den tanke at en variabel som er statisk ikke kan ændres. Dette vil dog være underligt, idet betegnelsen 'variabel' jo næsten inviterer til ændringer. Men 'static' i C har ikke denne konsekvens. Hvis vi i C ønsker at gardere os mod ændringer af en variable skal vi bruge en modifier som hedder `const`.

20.4. Storage class extern

Lektion 5 - slide 20

Nogle variable og funktioner ønskes at være tilgængelige i hele programmet. For variable opnås dette ved at placere disse uden for funktionerne, og uden for **`main`**. Sådanne variable, og funktioner, siges at have external linkage, og de er tilgængelige i hele programmet.

Dog gælder stadig reglen, at en variabel eller funktion ikke kan bruges før det sted i programmet, hvor den er introduceret og defineret. For funktioner bruger vi ofte de såkaldte prototyper til at annoncere en funktion, som vi endnu ikke har skrevet.

Variable og funktioner med storage class *extern* kaldes eksterne variable

En *ekstern variabel* eller funktion er global tilgængelig i hele programmet

- Storage class **`extern`**
  - Default storage class af variable som erklæres uden for funktioner
  - Default storage class af alle funktioner er også **`extern`**
  - *"External linkage"*

20.5. Storage class static af eksterne variable

Lektion 5 - slide 21

Den sidste storage class vi vil introducere hedder også static, men den virker på eksterne variable. Statiske eksterne variable kan kun ses i den kildefil, hvori de er erklæret.

Statiske eksterne variable har ikke noget at gøre med statiske lokale variable ala afsnit 20.3.

For globale variable virker storage class **`static`** som en scope begrænsning i forhold til storage class **`extern`**.

Statiske globale variable er private i den aktuelle kildefil.

En *statisk variabel* på globalt niveau er kun synlig i den aktuelle kildefil

- Storage class static - globale variable
  - Skal angives med **`static`** som storage class
  - Synlighed i den aktuelle kompileringsenhed - kildefil
  - Kun synlig fra det punkt i filen hvor variablen er erklæret
  - *"Internal linkage"*

Funktioner kan også være statiske, og dermed private i en kildefil.

Synlighedskontrol er vigtig i store programmer.

Synlighedskontrol er meget vigtig i objekt-orienteret programmering.