SLI 280

KOMPENDIUM I PROGRAMMERING — DEL 1

Programmeringsspråkene

C og C++
av

Asbjørn Brændeland

Januar 2001
INNHOLD

1 Programmeringsspråkene C og C++ .................................................................................................................... 1
   1.1 C ........................................................................................................................................................................... 1
   1.2 C++ ..................................................................................................................................................................... 2
   1.3 Hvorfor lære C og C++? ......................................................................................................................................... 2
   1.4 C, C++, Scheme, Pascal, Simula og andre språk .................................................................................................. 3
   1.5 C's kompakte særregenhet .................................................................................................................................. 3
   1.6 Objekter, venstreverdier, deklarasjoner, definisjoner, uttrykk og setninger ......................................................... 4
2 Programmeringsomgivelsene DrScheme og Visual C++ ............................................................................................ 5
3 I/O i C og i C++ ......................................................................................................................................................... 7
4 Kildefiler for C og C++ .............................................................................................................................................. 9
   4.1 Kildefilnavn ........................................................................................................................................................ 9
   4.2 Definisjonsfiler (Header-filer) og implementasjonsfiler ....................................................................................... 9
   4.3 Kildekode og objektkode ..................................................................................................................................... 10
5 Språkregler og konvensjoner ................................................................................................................................... 11
   5.1 Deklarasjoner og definisjoner av variabler ....................................................................................................... 11
   5.2 Identifikatorer ..................................................................................................................................................... 11
   5.3 Språklike konvensjoner ...................................................................................................................................... 11
   5.4 Norske oversettelser av engelsk programmeringsterminologi ....................................................................... 14
6 Posisjonelle tallsystemer ......................................................................................................................................... 15
   6.1 Ikke-desimale tallsystemer .................................................................................................................................. 15
   6.2 Det binære tallsystemet ....................................................................................................................................... 15
   6.3 Heksadesimale og oktale tall .......................................................................................................................... 16
   6.4 Tall med fortegn .................................................................................................................................................. 16
7 Den syntaktiske formen til et C-program .................................................................................................................. 19
   7.1 Minimumsprogrammet Hello_World ................................................................................................................. 19
   7.2 Hovedprogrammet ............................................................................................................................................ 19
   7.3 Funksjoner ......................................................................................................................................................... 20
   7.4 Deklarasjon av variabler ................................................................................................................................... 20
   7.5 Kommentarer .................................................................................................................................................... 21
8 Datatyper ................................................................................................................................................................. 23
   8.1 Tall .................................................................................................................................................................... 23
   8.2 Booleske ........................................................................................................................................................... 23
   8.3 Tegn ................................................................................................................................................................... 24
   8.4 Integrale og flytende typer .............................................................................................................................. 24
9 I/O ............................................................................................................................................................................ 27
   9.1 Skrivning til skjermen og lesing fra tastaturet i C ............................................................................................... 27
   9.4 Tegnvis og linevis skrivning og lesing i C .......................................................................................................... 28
   9.1 Skrivning til skjermen og lesing fra tastaturet i C++ ........................................................................................... 28
   9.2 Tegnvis og strengvis skrivning og lesing i C++ .................................................................................................... 29
   9.5 Et eksempel på bruker-I/O ............................................................................................................................... 29
   9.6 Et eksempel på fil-I/O ....................................................................................................................................... 31
16 Referanser .................................................................................................................................................. 67
16.1 Deklarering og initialisering av referanser ........................................................................................... 67
16.2 Verdi- og referanseoverføring av parametre .................................................................................... 67
17 Utvidede funksjonsdomener .................................................................................................................... 69
17.1 Funksjoner som tar ulike typer enkle argumenter ........................................................................... 69
17.2 Funksjoner som tar både enkle og sammensatte typer som argumenter ........................................... 69
17.3 Stiltiende argumenter — Intil et gitt antall argumenter .................................................................. 70
17.4 Funksjoner som tar et vilkårlig antall argumenter ........................................................................... 70
18 Inlining .................................................................................................................................................... 73
19 Klasser og objekter ................................................................................................................................. 75
19.1 Definering av en klasse ......................................................................................................................... 75
19.2 Aksessering av metoder og data ......................................................................................................... 76
19.3 Offentlige, beskyttede og private metoder ......................................................................................... 76
19.4 ”Inlining” av klasse metoder ............................................................................................................. 76
19.5 Konstruktører og destruktører .......................................................................................................... 77
19.7 Superklasser og subklasser — arv og polymorfi ............................................................................. 79
19.8 Flerfoldig arv ..................................................................................................................................... 83
19.9 Kompilatorgenererte defaultkonstruktører ..................................................................................... 85
19.10 Klassevenner .................................................................................................................................. 86
19.11 Utvidelse av operatoromener ............................................................................................................ 87
20 Filbehandling ........................................................................................................................................ 89
20.1 Et lite eksempel på lese fra og skrive til fil implementert i C og C++ .................................................... 89
20.3 Noen filbehandlingsrutiner og -algoritmer i C++ ............................................................................ 90
20.4 Direktefiler i C++ ............................................................................................................................... 93
20.5 Behandling av I/O-feil i C++ ............................................................................................................ 94
20.6 Klassehierarkiet for filbehandling i C++ ............................................................................................ 95
20.6 De mest brukte I/O-funksjoner, operatorer og manipulatorer i C++ .................................................. 96
20.7 Innlesing av filnavn fra bruker ......................................................................................................... 97
21 Mer om tegnstrønger ............................................................................................................................ 101
21.1 Strenger i Scheme, Pascal og C ........................................................................................................ 101
21.2 Standard strengoperasjoner ............................................................................................................. 101
21.3 Egendefinerte strengoperasjoner .................................................................................................... 102
22 Cs preprosessor ..................................................................................................................................... 103
Forord

Del 1 av kompendiet i SLI 180, heretter kalt K1, er ment å skulle dekke syntaksen for C og C++ — bortsett fra templates og feilhåndtering vha. try og catch i C++.

Målet med K1 er at den skal kunne fungere som referansebok alene, men heftet må nok utprøves minst ett semester før det målet kan sies å være nådd (og dessuten trenger den en indeks).

I tillegg inneholder K1 en rekke sammneligninger mellom C/C++ og både Scheme og Pascal. Leseren forutsettes å ha gjennomført et innføringskurs i programmering, og for det fleste vil dette være SLI 130 der språket Scheme brukes. Pascal er tatt med delvis fordi dette var det prosedyrale språket som ble brukt tidligere ved ILF, og delvis fordi Pascal vel må sies å være et ”alminnelig” språk, der C og C++ har sine særegenheter.¹

K1 har en forholdsvis kondensert form med korte eksempler som bare er ment å illustrere de ulike syntaktiske mekanismer — til forskjell fra resten av kompendiet (K2 og K3) som har lengre og mer sammensatte eksempler, men som ikke dekker all syntaksen like godt.

Asbjørn Brændeland

¹ Hva som er alminnelig eller vanlig i denne sammenhengen, er selvsagt relativt. På 1960- og 70-tallet var FORTRAN det suverent mest brukt språket, mens det på 80- og 90-tallet kanskje var C. Det vi snakker om her, er imidlertid hvilket språk som har minst særegenheter i forhold til snittet av de mest brukte programmeringsspråkene, og da er det nok mer nærliggende å vise til Pascal enn til C eller C++. 
1 Programmeringsspråkene C og C++

1.1 C

Om C skriver opphavsmannen Dennis M. Ritchie sammen med Brian W. Kernighan:

“C is a general purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C is not a "very high level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie [ved AT&Ts laboratorium i Palo Alto 1960–75, mitt innsk.]. […]”


Med modern control flow siktes det til bruk av valg- og løkkemekanismer.

Med modern data structures siktes det til gruppering av data av ulike typer innenfor brukerdefinerte (i motsetning til forhåndsdetifinerte) sammensatte datatyper.

Navnet C er et produkt av språkets forhistorie

CPL (Combined Programming Language — en meget kompleks forløper for BCPL)
→ BCPL (Basic Combined Programming Language — en typeløs forenkling av CPL utviklet av engelskmannen Martin Richards under hans besøk ved MIT på 60-tallet)
→ B (Basic (men ikke Basic) eller Bonnie (Thompsons kone) — en, fremdeles typeløs, videreutvikling av BCPL av Ken Thompson ved AT&Ts lab. rundt 1970)
→ C (Åpent for fortolkning (neppe Combined) — Ritchies typede viderutvikling av B)

C utmerker seg fremfor alt ved

• mulighet for fleksibel strukturering og tolkning av data kombinert med
• maskinnsikt, som ikke minst gjør det mulig å skrive raske programmer, og
• syntaktiske mekanismer for kompaktivering av kode (se kodeeksempel under).

Disse trekkene må imidlertid utnyttes med årvåkenhet og ansvar idet

• den frihet programmereren tilbys, kan bli for stor (kompliatoren oppdager ikke nok feil),
• utnyttelse av maskinnsikkerheten kan føre til liten portabilitet, og
• utnyttelse av kompakteringsmulighetene kan gi kryptisk kode.

### Scheme

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(set! i (+ i 1))</td>
<td>i := i + 1;</td>
<td>v[+i] += 16;</td>
</tr>
<tr>
<td>(vector-set! v i) := v[i] + 16;</td>
<td>v i (+ (vector-ref i) 16)</td>
<td></td>
</tr>
</tbody>
</table>

Flere av de eksemplene på Scheme-kode som vises, vil være lite typiske, idet de viser destruktiv programmering der rent funksjonell programmering ville ha vært mer naturlig. Den første setningen i Scheme-eksemplet over er tatt med for å sikre den semantiske likheten mellom de tre variantene. Den andre Scheme-setningen ville ellers mer typisk innå i en rekursiv funksjon der den første setningen ikke hørte hjemme, idet i å bli incrementert ved det rekursive kallet.
1.2 C++

Om C++ skriver opphavsmannen Bjarne Stroustrup:

“Clearly, C++ owes much to C [Kernighan 1978]. C is retained as a subset. I also retained C’s emphasis on facilities that are low-level enough to cope with the most demanding systems programming tasks. […] The other main source of inspiration for C++ was Simula67 [Dahl, 1970] [Dahl, 1972]; the class concept with derived classes and virtual functions was derived from it. C++’s facility for overloading operators and the freedom to place a declaration wherever a statement can occur resembles Algol68 [Woodward, 1974].”

[Stroustrup, The C++ Programming Language, 1997, s.10]

“[…] In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types […] that closely match the concepts of [an] application. […] objects of user-defined types can be used conveniently and safely in contexts where their type cannot be determined at compile time. Programs using such objects are often called object based. […] The key concept in C++ is class. Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user defined types, dynamic typing, user controlled memory management and mechanisms for overloading operators. C++ provides much better facilities for type checking […] than C does. It also contains improvements […] including […] default function arguments, overloaded functions […] and a reference type. […]”


Her brukes mange vanskelige ord. Inntil videre skal vi bare merke oss noen nøkkelord:

(a) mekanismer for lav-nivå-programmering (low-level prograrming facilities — som i C)
(b) typesjekking (type checking — ved parameteroverføring og verditilordning / -binding)
(c) referansetype (reference type — for referenceoverføring av parametre)
(d) stilltiende funksjonsargumenter (default function arguments — ved utelatelse av aktuelle param.)
(e) utvidede funksjonsdomener / funksjonsoverlessing (function overloading — bruk av samme funksjon for ulike typer og/eller antall argumenter)
(f) klasser, objekter, dynamisk typebestemmelse, avledede klasser og virtuelle funksjoner (object based programs — generelt: objekt-orientert programmering)
(g) utvidede operatordomener / operatoroverlessing (operator overloading — bruk av samme operator for ulike typer operander)

(a)—(e) vil bli tatt opp i løpet av kurses første halvdel. Av disse gjelder (b)—(e) sider ved C++ som i og for seg er nyttige og gode, men ikke i seg selv viktige nok til å velge C++ fremfor C.

(f) og (g) hører hjemme i andre halvdel av kurset, dvs. (g) er egentlig et underpunkt til (f), idet operator overloading er eksklusivt knyttet til klassebegrepet — og det er også klasse-begrepet som interesserer oss mest ved C++ i forhold til C.

NB! Det som i det følgende sies om C, gjelder også, om ikke annet er sagt, for C++.

1.3 Hvorfor lære C og C++?

• C er kanske det mest utbredte språket i dag — med sterk konkurranse fra C++, og disse to språkene til sammen brukes i flertallet av større programmeringsprosjekter.
• Programmering i C befordrer en forståelse av datamaskinens virkemåte, og gir et supplement til den mer «abstrakte» Scheme- og Prolog-programmering man ellers beskjøfter seg med i SLI.
• C++ er et gjenomført objektorientert språk med all de egenskaper som er beskrevet av Dahl & Nygaard — opphavsmennene til objektorientert programmering (OOP — hvis fortrinn vi skal komme tilbake til).
Mer underordnet er det at man med C kan man skrive raskere programmer i mer kompakt kode enn med de fleste andre språk (dette hadde vært rimelig uinteressant, hvis ingen brukte språket). Dette er allikevel av betydning for oss, siden SLI 280 skal legge grunnlaget for bl.a. sortering av og søking i store datamengder.

Endelig kommer for Cs vedkommende, at dette er sterkt knyttet til operativsystemet UNIX, for dem som måtte være interessert i det. UNIX (og beslektede operativsystemer som LINUX) er imidlertid ikke tema i SLI 280.

1.4 C, C++, Scheme, Pascal, Simula og andre språk.

Særlig til å begynne med i kurset, vil vi sammenligne C++ med andre språk.
- De fleste studentene vil bare kjenne Scheme, og bl.a. derfor bruker vi det som det ene sammenligningsspåket.
- Noen vil kanskje bare kjenne Simula eller Pascal. Vi har valgt Pascal som andre sammenligningsspråk, siden det er det vi har brukt ved IIF tidligere.
- De som måtte kjenne Java, vil alt kjenne mye av den grunnleggende syntaksen i C++.

De viktigste prinsipptuelle forskjellene finner vi mellom det funksjonelle språket Scheme på den ene siden og de prosedyrale språkene Simula, Pascal, C, C++, Java, etc. på den andre siden. Riktignok gir både Pascal og Simula muligheten for å definere funksjoner (prosedyrer som returnerer verdier), og i C og C++ finnes det bare funksjoner, men ikke desto mindre er det de destruktive operasjonene, tilstandsendringene, sideeffektene og instruksjonsekvensen som spiller hovedrollene i programmer skrevet i disse språkene.

Noe av det første man lærer i et innføringskurs i et prosedyralt språk, er verditilordning, der vi tilordner en verdi til en variabel som forutsettes å være opprettet på forhånd. Dette er også mulig i Scheme, men i et innføringskurs vil temaet knapt berøres — eller det vil komme langt ut i kurset.

Noe tilsvarende gjelder I/O (Input/Output — lesing og skriving av kommandoer og data): Mens man i et Scheme-kurs i første omgang overlater dette til Read-Eval-Print-løkken, må studentene i et C- eller Pascal-kurs lære seg I/O fra første stund, for overhodet å få data inn i og ut av programmene.

En annen viktig prinsipiell forskjell mellom Scheme på den ene siden og C, C++, Pascal og Simula på den andre siden består i at Scheme er utypet, mens de andre er typet. Det er imidlertid ingen nødvendig sammenheng mellom utypet og funksjonell eller typet og prosedyral. — BCPL og B er begge utypede prosedyrale språk.

1.5 C’s kompakte særegenhet

Det er også noen viktige forskjell mellom C/C++ på dene ene siden og andre prosedyrale språk som Pascal og Simula på den andre siden. Dette gjelder bl.a.
- bruken av unære operatorer (vi kan f.eks. i C forkorte \( i = i + 1 \) til \( i++ \), mens vi i Pascal og Simula bare har binære aritmetiske operatorer og må skrive \( i := i + 1 \));
- behandling av pekere (pekere er i bunn og grunn tall, og vi kan i C utføre aritmetiske operasjoner på disse),
- behandling av vektorer (en vektorvariabel evlerer til en peker til vektores første element),
- bruken av eksplositve setninger (instruksjoner) i kontrollsetningene i løkker, og
- samtlig bruk av tilordninger som setninger og uttrykk med returverdi (og dermed som førsteklasses elementer).

Alle disse egenskapene bidrar til muligheten for å skrive svært kompakt kode.

I ett henseender er C mer lik Simula enn Pascal, idet typen angis foran navnet i en variabel-deklarasjon i C og Simula, mens rekkefølgen er omvendt i Pascal. Dette er imidlertid mindre viktig.
1.6 Objekter, venstreverdier, deklarasjoner, definisjoner, uttrykk og setninger

K&R definerer et objekt som et nøyværende område i hukommelsen.

I samme forbindelse definerer de det de kaller en *venstreverdi* (*lvalue*) som et *uttrykk som refererer til et objekt*.

Typiske venstreverdier er *variabelnavn*. Navnet tilsier å disse *kan* forekomme på venstre side i en tilordningssetning, dvs. de *kan få sine verdier endret*. Semantikken i en tilordning består i at den verdien høyresiden i uttrykket evaluerer til, lagres i det objektet venstresiden i uttrykket refererer til.

Det K&R kaller objekter tilfredsstiller A&S’ kriterier for førsteklasses elementer:
- De kan navngis som variabler.
- De kan sendes som argumenter til prosedyrer.
- De kan returneres som resultater av prosedyreutførelser.
- De kan innlemmes i datastrukturer.

Abelson & Sussman [1997]

Prosedyrer er *ikke* førsteklasses elementer i C (men vi kan operere med dem som *quasi* førsteklasses elementer vha. pekere til prosedyrer).

Det K&R kaller *objekter* et noe annet enn det Stroustrup og mange andre kaller *objekter*. Stroustrup knytter objektbegrepet ekslusivt til klassebegrepet og definerer *objekt* som *instant av klasse*. Et typisk objekt er ikke bare et område i hukommelsen, men har også *funktionselle egenskaper*.

Det er dette som ligger til grunn for betegnelsen *objektorientert programmering* (OOP omtales senere i kompendiet).

En *deklarasjon* angir hvordan et navn skal fortas, dvs. hvilken type en navngitte variabel har.

En *definisjon* er en deklarasjon som *knytter/binder et navn til et objekt*, dvs. som *setter av plass i hukommelsen* for en navngitt variabels verdi. En deklarasjon er ikke nødvendigvis også en definisjon, og er den det ikke, må det i deklarasjonen angis *at* variabelen er definert et annet sted (men ikke hvor den er definert).

Et *uttrykk* (*expression*) er kjennetegnet ved at det evaluerer til en verdi. Typiske uttrykk er *variabler*, *operasjoner* (f.eks. \(a + b\) og \(x < y\)) og *funksjonskall*.

I et Scheme-program er alt uttrykk..

En *setning* er noe som *utføres*.

*Setninger utføres i den rekkefølgen de står*, bortsett fra i den grad de er underlagt kontroll i valg, løkker og hopp.

En setning som sådan gir ingen verdi og utføres mht. sin effekt.

De vanligste setning er uttrykksetninger som består av et uttrykk som også er en setning, fulgt av et semikolon.

Typiske uttrykksetninger er tilordninger og funksjonskall — f.eks. \(i = 5\); (gi tallvariablen \(i\) verdien 5, og returner denne verdien) og \(strcpy(s1, s2)\); (kopier innholdet i strengen \(s2\) til strengen \(s1\), og returner adressen til \(s1\)).

Flere setninger kan settes sammen til en *sammensatt setning* (*compound statement*) — også kalt *blokk*.

Andre typer setninger er
- *valgsetninger* (velg hva som skal gjøres på grunnlag av utfallet av en test),
- *løkkesetninger* (gjenta en setning (evt. blokk) så lenge en gitt betingelse holder) og
- *hoppsetninger* (fortsett programutførelsen fra et gitt punkt, eller hopp ut av en gitt kontekst).

Ingen av disse er uttrykk (og dermed returnerer ingen av dem noen verdi).
2 Programmeringsomgivelsene DrScheme og Visual C++

Den viktigste praktiske forskjellen angår de omgivelsen vi programmerer i. Vi skal her bare gå inn på de omgivelsene som finnes På ILFs øvingsmaskiner finnes bl.a. DrScheme som brukes i SLI 130 for programmering i Scheme, og er det Microsoft Visual Studio som bl.a. omfatter Visual C++ og brukes i SLI 280 for programmering i C / C++.

DrScheme er kanskje de mest utviklede og omfattende omgivelsene for Scheme-programmering i dag (1999), men grensesnittet for kjøring av Scheme-programmer er allikevel ganske enkelt. Det omfatter
- et vindu for redigering av programfiler (definisjonsfiler),
- et vindu for interaksjon med Scheme, samt
- noen få menyer og dialoger for interaksjon med DrScheme.

Det som fremfor alt gjør Scheme-omgivelser generelt svært enkle, er imidlertid
- språket Scheme i seg selv, og
- den umiddelbar evalueringen av all input fra programmereren i Read-Eval-Print-løkken.

Alt i Scheme er funksjoner, og vi eksekverer et Scheme-program ved å kalle en funksjon, som typisk starter en kjede av funksjonskall som til slutt gir oss tilbake et resultat gjennom en kjede av returverdier.

Vi kan dermed få gjort ting i Scheme
- svært enkelt — ved å ved å taste inn et kall på en forhåndstildefinert funksjon,
- relativt enkelt — ved å definere en funksjon, få definisjonen utført ved et museklikk, og deretter få funksjonen utført umiddelbart ved å taste inn et kall på den,
- fremdeles noksa enkelt — ved å skrive inn et program bestående av definisjoner av funksjoner som kaller hverandre, få definisjonene utført ved et museklikk, og deretter få programmet utført ved å kalle den funksjonen som sette i gang den ønskede kallsekvensen.

For språk som Pascal og C++ omfatter programmeringsomgivelsene typisk en editor, en prosjektorganisator, en kompilator og en linker (og dessuten gjerne en andre komponenter for redigering av ressurser (dialoger, menyer, etc.) debugging, profilering, etc.). For å få utført noe som helst i må vi
- opprette et programprosjekt for samordning av kildefiler og andre filer,
- skrive programmet til en eller flere kildefiler,
- legge kildefilen(e) inn i prosjektet,
- få programmet kompilert til objektfiler
- få objektfilene lenket sammen til en eksekverbar fil, og
- starte den eksekverbare filen.

Dessuten må vi sørge for at programmet selv mer eller mindre eksplicit håndterer brukergrensesnittet, som kommer i tillegg til grensesnittet mellom programmereren og programmeringsomgivelsene.

En meget kort innføring i bruk av Visual C++ er gitt i Kompendiets del 0.
3 I/O i C og i C++

Den umiddelbart iøyenfallende forskjellen mellom typiske C og C++ programmer gjelder I/O (lesing og skriving). Ingen av språkene har mekanismer direkte innebygget for å håndtere dette. I stedet har hvert språk sitt standard-bibliotek for I/O.

- Cs ene I/O-bibliotek omfatter bl.a. definisjoner av
  - filtypen FILE,
  - to globale variabler av typen FILE for hhv. lesing fra tastaturet og skriving til skjermen,
  - rutiner som stilltiende (uten parameteroverføring) bruker de globale variablene for lesing fra tastaturet og skriving til skjermen,
  - rutiner for å åpne fysiske filer for lesing og skriving og knytte disse til FILE-variabler, og
  - rutiner som tar FILE-objekter som argumenter, for lesing fra, skriving til og lukking av filer.

- C++s gruppe av I/O-biblioteker omfatter bl.a. definisjoner av
  - et hierarki av klasser — herunder
    - ios med formateringmekanismer, diverse I/O-relevante konstanter, etc.,
    - istream for lesing generelt,
      - ifstream for lesing fra fil,
    - ostream for skriving generelt,
      - ofstream for skriving til fil,
  - metodene til istream og ostream, for lesing og skriving generelt,
  - metodene til ifstream og ofstream, for åpning og lukking av og lesing fra og skriving til filer, og
  - de to globale objektene cin av typen istream og cout av typen ostream for hhv. lesing fra tastaturet og skriving til skjermen.

Cs I/O-bibliotek kan brukes i C++, siden C er et subset av C++, men C++s eget bibliotek vil normalt bli foretrukket — også av oss, siden det alt vesentlig representerer en forbedring i forhold til Cs. Vi skal allikevel gi en kort forklaring av Cs I/O-bibliotek senere, siden littertauren inneholder en rekke eksempler på dets bruk.

I siste del av kurset skal vi definere våre egne klasser for lesing fra og skriving til filer. Vi definerer da alt selv, bortsett fra de grunnleggende rutinene for direkte kommunikasjon med operativsystemet (API-er — Application Programming Interfaces).
4 Kildefilen for C og C++

4.1 Kildefilnavn

Visual C++ forventer at kildefilene til C- og C++-programmer hhv. har navnetilleggene c og cpp. Skulle vi av en eller annen grunn ønske å få et program kompilert i henhold til Cs syntaks, må vi altså gi kildefilene navntillegget c; men vi holder oss stort sett til C++-syntaks og bruker cpp-filer. (I prosjekter med flere kildefiler, er det navnet på den første filen som bestemmer syntaksen.)

Forøvrig er Cs syntaks i Visual C++ (og andre omgivelser) utvidet til å omfatte visse C++-elementer (bl.a. kommentartegnet //, som gjør resten av den linjen der det står, til en kommentar, i tillegg de kommentartegnene /* og */ som plasseres hhv. før og etter en kommentar).

4.2 Definisjonsfiler (Header-filer) og implementasjonsfiler

I C bruker man ofte filer med ferdig definerte funksjoner (standard libraries), og man må da hente inn de nødvendige definisjonene. Dette gjør med direktivet #include.

Merk at # må stå helt til venstre på linjen, og at linjen ikke skal avsluttes med ; (semikolon).

Vi har sett direktivet #include <iostream.h>, som angir at vi vil ha med filen der bl.a. de to I/O-objektene cin og cout er definert.

NB! Så å si alle programeksplomplene inneholder minst én utskriftsetning. I flere av disse er imidlertid direktivet #include <iostream.h> utelatt av plasshensyn. Husk å ta dette med, dersom du ønsker å kjøre eksemplet.

Vi skal siden benytte andre pakker — både standardpakker og egendefinerte pakker.

De to mest brukte pakkene i C er:
- stdlib («standard library») definerer de vanligste C-funksjonene - f.eks. exit.
- stdio («standard I/O») definerer alt som har med lesing og skriving å gjøre.


I C++ erstattes som regel stdio av pakken iostream, men de to pakkene kan brukes side om side.

De to mest brukte pakkene i C++ er:
- stdlib («standard library») definerer de vanligste C-funksjonene - f.eks. exit.
- stdio («standard I/O») definerer alt som har med lesing og skriving å gjøre.


I C++ erstattes som regel stdio av pakken iostream, men de to pakkene kan brukes side om side.

I store programmer er det vanlig å fordele kildekoden på flere filer — bl.a. for å grupperer variabler og funksjoner som hører sammen, og generelt for å lette administreringen av programmet. I den forbindelsen fordeler man prototypene og funksjonsdefinisjonene på hver sine typer filer. Den første typen kalles header-filer (og har så å si alltid filnavntillegget h). Den andre typen kan vi kalle implementasjonsfiler (selv om dette ikke er den vanlige betegnelsen i C-litteraturen). Når vi skal innlemme en fil i et program, er det header-filen vi angir i #include-direktivet.
4.3 Kildekode og objektkode

Den kompilerte koden, som maskinen leser som binær kode (nuller og ettall) kaller vi objektkoden. Objektkoden inneholder imidlertid selv en god del rent symbolsk informasjon som ikke blir aktualisert før all objektkoden er lenket sammen til en integrert fysisk representasjon i maskinenes hukommelse under kjøring. Den symbolske informasjonen angir bl.a. at programmet på et punkt skal hoppe til et annet punkt som er navngitt, men som ennå ikke har noen fysisk adresse. Dette innebærer at kompilatoren kan generere objektkode med kall på funksjoner som er angitt i header-filene, uten å sjekke om det også er kompilert objektkode for disse funksjonene. En slik sjekk er under enhver omstendighet ikke hensiktsmessig, i og med at C tillater sirkulære kallsekvenser på tvers av kildekodefilene, og dermed kan ikke kompilatoren baseres på at det foreligger objektkode for alle funksjoner som kalles.

At det faktisk foreligger kompilert objektkode for alle de funksjoner som kalles i hele programmet, er det linkeren som sjekker idet den (forsøksvis) lenker alle de kompilerte programdelene sammen til ett program. Linkeren sjekker også bl.a. at variabler hvis lagerplass er spesifisert som extern, overhodet er definert ett eller annet sted.

Kompileringen foregår i to runder, dvs. kompilatoren kaller et eget program for såkalt preprossering av kildekodeen. Dette medfører bl.a. at alle filer som er angitt med et #include, innlemmes i den koden som gjøres til gjenstands for den endelige kompileringen.
Preprosessoren foretar dessuten
- substitusjoner av makroer gitt ved #define, dvs. substitusjon av identifikatorene ved de C-uttrykk de er definert ved, og
- betinget kompilering gitt ved bl.a. #if, #ifdef og #ifndef, dvs. fjerning av kode som ikke skal kompileres, avhengig av gitte betingelser, f.eks. om et globalt navn er definert eller ikke, eller hvordan det eventuelt er definert.
— Mer om dette siden.
5 Språkregler og konvensjoner

5.1 Deklarasjoner og definisjoner av variabler

Det råder en viss uklarhet i programmeringslitteraturen i bruken av begrepene *deklarasjon* og *definisjon*. Vi skal holde oss til Kernighan & Ritchie som bl.a. sier følgende (i min redigering):

- En deklarasjon spesifiserer en variabels *type*.
- En variabel må være deklarert før den brukes — i funksjonen, eller ved en forutgående global definisjon.
- En variabel som bare er deklarert lokalt i en funksjon, opprettes idet funksjonen kalles, og eksisterer bare under utførelsen av kallet. Slike variabler kalles *automatiske*, pga. måten de opprettes på ved kallet.
- En deklarasjon som, i tillegg til å spesifisere en type, *setter av plass i hukommelsen* for den deklarerte variabelen, kalles en *deklarasjon*. Dette gjelder lokale og globale deklarasjoner som ikke er spesifisert som *extern*, dvs. som ikke forutsettes definert et annet sted.
- En variabel kan definieres én gang for alle, før eller mellom funksjonsdefinisjonene. Den er da globalt tilgjengelig — *implisitt* i alle etterfølgende funksjoner, eller ved at variabelen spesifiseres *eksplicit*, som en *extern* variabel, i en *funksjon*.

I tillegg snakker vi om definisjoner av *prosedyrer* og *typ*er, men i disse tilfellene gjelder ikke definisjonene *när*, eller *om*, definens instansieres.

5.2 Identifikatorer

En *identifikator* (et navn på en type, variabel, funksjon eller annet) kan bestå av store og små bokstaver og sifre samt tegnet _ (\_*_\_\_), men kan ikke begynne med et siffer.

Lovlige identifikatorer

|  loenn  | sum1   | sum_2 |

Lovlig, men ikke anbefalt, identifikator:

|  _sum_ | __SUMTOT |

Ulovlige identifikatorer

|  lønn  | sum_3   | 4de_sum |

Det er forskjell på store og små bokstaver i C, og f.eks. SUM og sum er tre ulike navn. Identifikatorer som alt er definert i omgivelsene, og som angår systemforhold har gjerne én eller to innledende, og kanskje avsluttende underscores, og vi bør derfor avstå fra å bruke slike identifikatorer.

#333

Lovlig, men ikke anbefalt

- **loenn:** inneholder en ikke-engelsk bokstav.
- **sum-3:** inneholder et minustegn og kan forveksles med uttrykket `sum - 3`.
- **4de_sum:** inneholder et siffer og kan forveksles med `4 de_sum`.

5.3 Språklig konvensjoner

I tillegg til det rent syntaktiske er mange C-programmerere opptatt av hvordan navn for ulike formål bør gis ulike former. Lervik & Ljosland har sine egne konvensjoner. Disse er beskrevet i Vedlegg 9 [L&L 1994].

Under er utdrag fra et par andre tekster som tar for seg dette. (Utdragene inneholder en rekke termer som vil bli forklart etterhvert. De som ønsker å følge f.eks. *Indian Hill C Style*, kan vende tilbake hit, underveis.)

5.3.1 Indian Hill C Style Naming Conventions

- Names with leading and trailing underscores are reserved for system purposes and should not be used for any user-created names. Most systems use them for names that the user should not have to know. If you must have your own private identifiers, begin them with a letter or two identifying the package to which they belong.
  - **#define** constants should be in all **CAPS**.
  - **enum** constants are **Capitalized** or in all **CAPS**
  - Function, **typedef**, and variable names, as well as **struct**, **union**, and **enum** tag names should be in lower case.
  - Many macro “functions” are in all **CAPS**. Some macros (such as `getchar` and `putchar`) are in lower case since they may also exist as functions. Lower-case macro names are only acceptable if the macros behave like a function call, that is, they evaluate their
parameters exactly once and do not assign values to named parameters. Sometimes it is impossible to write a macro that behaves like a function even though the arguments are evaluated exactly once.

- Avoid names that differ only in case, like foo and Foo. Similarly, avoidfoobar and foo_bar. The potential for confusion is considerable.

- Similarly, avoid names that look like each other. On many terminals and printers, 'l', '1' and 'I' look quite similar. A variable named 'l' is particularly bad because it looks so much like the constant '1'.

In general, global names (including enums) should have a common prefix identifying the module that they belong with. Globals may alternatively be grouped in a global structure. Typedefed names often have "_t" appended to their name.

Avoid names that might conflict with various standard library names. Some systems will include more library code than you want. Also, your program may be extended someday.

Fra


Henry Spencer, Zoology Computer Systems, University of Toronto

David Keppel, EECS, UC Berkeley; CS&E, University of Washington

Mark Brader, SofQuad Incorporated, Toronto

Recommended C Style and Coding Standards 1990. [... an updated version of the brook Hill C Style and Coding Standards paper, with modifications by the last three authors.]

http://www.apocalypse.org/pub/u/paul/docs/cstyle/cstyle.htm#contents

5.3.2 Hungarian Notation

The names of pointer types such as PWORD and LPWORD are examples of a style of naming convention that is followed throughout the Windows programming interface. In this naming style, the names of data types, variables, structure members, and function parameters commonly begin with a prefix of anywhere from one to four letters identifying the general type of data they represent, followed by the main descriptive part of the name. For example, whereas a handle to a menu record in the Macintosh Toolbox is called a MenuHandle, in Windows it's an HMENU ("handle to menu").

This style of naming is called Hungarian notation, and it was actually invented by a real, live Hungarian [...] [Microsoft employee*] Charles Simonyi [...].

Common Hungarian prefixes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Meaning</th>
<th>Prefix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>by</td>
<td>Byte (8 bits)</td>
<td>sz</td>
<td>String (zero-terminated)</td>
</tr>
<tr>
<td>w</td>
<td>Word (16 bits, unsigned)</td>
<td>pt</td>
<td>Point</td>
</tr>
<tr>
<td>dw</td>
<td>Double word (32 bits, unsigned)</td>
<td>x</td>
<td>Horizontal coordinate (16 bits, signed)</td>
</tr>
<tr>
<td>i</td>
<td>Integer (signed)</td>
<td>y</td>
<td>Vertical coordinate (16 bits, signed)</td>
</tr>
<tr>
<td>n</td>
<td>Short integer (16 bits, signed)</td>
<td>rgb</td>
<td>RGB color</td>
</tr>
<tr>
<td>l</td>
<td>Long integer (32 bits, signed)</td>
<td>fn</td>
<td>Function</td>
</tr>
<tr>
<td>c</td>
<td>Count</td>
<td>p</td>
<td>Pointer</td>
</tr>
<tr>
<td>f</td>
<td>Flag (boolean)</td>
<td>lp</td>
<td>Long pointer</td>
</tr>
<tr>
<td>ch</td>
<td>Character</td>
<td>h</td>
<td>Handle</td>
</tr>
<tr>
<td>s</td>
<td>String (unterminated)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


*Hungarian notation brukes fremfor alt av Microsofts programmerere.
5.3.3 Brændelands konvensjoner

For å gjøre bildet komplett, bruker vi i dette kompendiet konvensjoner som mer eller mindre bryter med alle de ovennevnte.

- **Navn generelt** hentes bekymringsfritt fra engelsk såvel som fra norsk og kan gjerne utgjøre en blanding av de to språkene.
- **Navn** angi verdi til innleddende store bokstaver (bortsett fra i makronavn).
- **Makroer** skrives med bare store bokstaver, og man kan, om ønskelig, bruke underscores mellom navnedele — f.eks. slik:
  ```c
  #define BIG_ARR_SIZE 65536 // number of elements in a large array
  ```
- **Navn på enum-verdier** skrives som makroer.
- **Typenavn** angis også som makroer.
- **Variablenavn** inneledes med liten forbokstav — f.eks. slik:
  ```c
  int theNumber;
  ```
- **Funksjonsnavn** inneledes med stor forbokstav — f.eks. slik:
  ```c
  void CopyArray(pInt dest, const pInt source, const int lng) {...}
  ```
- **Klassenavn** inneledes med stor C.
- **Navn på klassesattributter forrige** inneledes med liten m (bokstaven svarer til den etterhånden utberedte praksis å betegne klassers attributter i felt som members (dvs. lemma, ikke medlemmer) og klassens metoder som member functions) — f.eks. slik:
  ```c
  class CBankAccount
  {
      ... 
      void Deposit(double amount);
      void Withdraw(double amount);
      ... 
      int mAccountNumber;
      ... 
      double mBalance;
  };
  ```

I tillegg er det verdi å merke seg følgende:

C og C++ inneholder en rekke syntaksene som angis med overlekkede enkeltegn, dvs. samme tegn brukes for ulike formål. Brukt riktig, innenfor en kontekst, vil de enkelte tegnene alltid være formelt entydige (selvfølgelig, ellers kunne vi ikke ha brukt disse språkene), men overlessingen kan i høyeste grad virke forvirrende på den meneskelige programmer.

Et slikt tegn er stjerne (asterisk, '*') som brukes i C både som

- **multiplikasjonssperator,**
- **spefisifikasjon** i en deklarasjon/definisjon av en variabel eller en definisjon av en type, og
- **operator** for såkalt *enfente* av en peker (for å angi det en peker peker på).

Dette kan illustreres med følgende lille eksempel:

```c
int* a; *a = 5 * 2;
```

Her definerer vi først `a` som en peker til en `int*(int*)`, og deretter tilordner vi verdien av produktet `5 * 2` til det `a` peker på.


At jeg konsekvent plasserer stjernen rett etter typenavnet, vil kanskje irritere noen av de som er kjent med C fra før, men jeg mener å ha mine grunner for dette. For det første vil den naturlig-språklige beskrivelsen av en type spesifisert med en stjerne, være f.eks. `peker til int`, og det burde være all grunn til å la denne begrepsdannelsen reflekteres i skrivemåten, så lengde den er syntaktisk lovlig. For det andre er det faktisk slik vi allikevel skriver det, når vi definerer en peker til en type — f.eks. slik `typedef int* PINT;` (Her er forståvidt ikke poenget at stjernen står etter typenavn, men at den ikke står foran noen variablenavn.) Det som kanskje kunne sies å tale imot min praksis (foruten den praksis K&R har institusjonalisert) er det faktum at vi i en definisjon av flere pekervariabler i samme setning må gjenta pekerspesifikasjonen for hver enkelt peker. Dette anser jeg imidlertid mest som en uheldig konsekvens av en flertydighetskonflikt.
### 5.4 Norske oversettelser av engelsk programmeringsterminologi


<table>
<thead>
<tr>
<th>L&amp;L engelsk</th>
<th>kommentar</th>
</tr>
</thead>
<tbody>
<tr>
<td>blokk</td>
<td>block-statement/compund statement</td>
</tr>
<tr>
<td>datacelle</td>
<td>—</td>
</tr>
<tr>
<td>etikett</td>
<td>label</td>
</tr>
<tr>
<td>erklæring</td>
<td>declaration</td>
</tr>
<tr>
<td>forditdefinert</td>
<td>predefined</td>
</tr>
<tr>
<td>finn</td>
<td>get</td>
</tr>
<tr>
<td>forekomst</td>
<td>instance/object</td>
</tr>
<tr>
<td>forktoret beregning</td>
<td>lazy evaluation</td>
</tr>
<tr>
<td>frie lageret</td>
<td>heap</td>
</tr>
<tr>
<td>skift</td>
<td>shift</td>
</tr>
<tr>
<td>medlem</td>
<td>member</td>
</tr>
<tr>
<td>lager</td>
<td>memory</td>
</tr>
<tr>
<td>rekkevidde</td>
<td>scope</td>
</tr>
<tr>
<td>standard</td>
<td>default</td>
</tr>
<tr>
<td>tekststreng</td>
<td>string/character string</td>
</tr>
<tr>
<td>utvelgning</td>
<td>access</td>
</tr>
<tr>
<td>utvidet anv. omr. for funksjoner/operatorer</td>
<td>overloading</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>par/tgn</th>
<th>funksjon i C++</th>
<th>norsk</th>
<th>engelsk</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>rundt arg.liste, gruppering av deluttrykk</td>
<td>parentes/rund parentes</td>
<td>parenthesis</td>
</tr>
<tr>
<td>[ ]</td>
<td>rundt array størrelse og indekser</td>
<td>skarpe klammer / firkanteparentes</td>
<td>square brackets</td>
</tr>
<tr>
<td>[ ]</td>
<td>rundt sammensatte setninger</td>
<td>klammeparentes</td>
<td>curly braces</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>rundt header-filnavn, rundt typen</td>
<td>spissparentes / vinkelparentes</td>
<td>angle brackets</td>
</tr>
<tr>
<td>#</td>
<td>innleder et preprosessordirektiv</td>
<td>dobbeltkryss</td>
<td>number-sign / hash</td>
</tr>
<tr>
<td>@</td>
<td>(ingen formell funksjon i C/C++)</td>
<td>krylalifa</td>
<td>at-sign</td>
</tr>
<tr>
<td>&amp;</td>
<td>adresseoperator,</td>
<td>et / og-tegn</td>
<td>ampersand</td>
</tr>
</tbody>
</table>

*Oversettelsen er for så vidt grei nok, men assosierses kanskje ikke umiddelbart til den engelske kildeterminen. Det hersker også en smule forvirring rundt bruk av norske navn på parenteser og en del andre tegn som er vanlige i programmeringslitteraturen. Her er et lite utvalg (se førsvrig Operatører).*
6 Posisjonelle tallsystemer

I posisjonelle tallsystemer angir hvert siffers posisjon hvilken vekt det skal ha:

\[ 107 = 1 \times 100 + 0 \times 10 + 7 \times 1 \]
\[ = 1 \times 10^2 + 0 \times 10^1 + 7 \times 10^0 \]

I dette tallsystemet, som vi kaller det *dekadiske* eller *desimale*, er grunntallet 10 og sifrene 0, 1, 2, 3, 4, 5, 6, 7, 8 og 9, men vi kan bruke andre grunntall og andre antall sifre.

### 6.1 Ikke-desimale tallsystemer

#### Tid angis i et 60-tallsystem:

Klokken 10.27.49

Det brukes også for vinkler

\[ \triangle ABC \] er \[ 82° 14' 51'' \]

På dansk og fransk lever ennå

«ni-og-halv-fems»: \[ 9 + 4/2 \times 20 = 99 \]

restene av et 20-tallsystem:

«quattro-vingt-dix-neuf»: \[ 4 \times 20 + 10 + 9 = 99 \]

### 6.2 Det binære tallsystemet

#### Bit

Siden det er spesielt enkelt å lagre bit (*binary digit*) benyttes det binære tallsystemet i datamaskiner.

I dette tallsystemet er grunntallet 2 og sifrene 0 og 1.

<table>
<thead>
<tr>
<th>sifre</th>
<th>0</th>
<th>1</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0\text{\textsubscript{b}}</td>
<td>0\text{\textsubscript{b}}</td>
<td></td>
<td>1\text{\textsubscript{b}}</td>
<td>9\text{\textsubscript{b}}</td>
<td>10\text{\textsubscript{b}}</td>
<td>11\text{\textsubscript{b}}</td>
<td>12\text{\textsubscript{b}}</td>
<td>13\text{\textsubscript{b}}</td>
<td>14\text{\textsubscript{b}}</td>
<td>15\text{\textsubscript{b}}</td>
</tr>
<tr>
<td>1\text{\textsubscript{b}}</td>
<td>1\text{\textsubscript{b}}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Byte**

Slik en typisk datamaskin er organisert, er det vanskelig å operere med enkelt-bit, og det ville dessuten for de fleste formål være uhensiktsmessig. I stedet slås bit-ene oftest sammen i på 8-bits-grupper kalt en **byte** (*a whole bit*, som er mer enn *just a little bit*). Her kan man lagre tall fra 0 til 255:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0 \text{\textsubscript{b}}</th>
<th>0 \text{\textsubscript{b}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1 \text{\textsubscript{b}}</td>
<td>1 \text{\textsubscript{b}}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 \text{\textsubscript{b}}</td>
<td>254 \text{\textsubscript{b}}</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1 \text{\textsubscript{b}}</td>
<td>255 \text{\textsubscript{b}}</td>
</tr>
</tbody>
</table>
6.3 Heksadesimale og oktale tall

Binære tall tar stor plass i utskrift, og derfor benyttes ofte heksadesimale tall (hex-tall) som har grunntall 16. Man slår da sammen 4 og 4 bit. 4 bit gir 16 verdier (0–15) som angis med enkeltvise hex-sifre: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Eksempel: 10100111010 = 0100,1001,1101 = 49D hex

Tidligere brukte man ofte oktale tall som har grunntall 8. Her slår man sammen 3 og 3 bit. I C skriver man hex-tall og oktale tall med hhv. 0x og 0 først.

Eksempel: 0x1A0 = 1A0 hex
          01765 = 1765 oct

6.4 Tall med fortegn

6.4.1 Negative tall

Man kan også lagre negative tall i det binære tallsystemet. Det vanligste formatet for dette heter toer-komplement (twos complement).

For å negere et tall, må man: 1. snu alle bit-ene i tallet (0 <-> 1), og 2. legge til 1.

For eksempel slik:

\[13_{16} = 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1_{16}\]
\[-13_{16} = 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1_{16}\]

Det finnes også en negeringsmåte, kalt ener-komplement (ones complement), der man nøyere seg med snu alle bits.

Verdiene rundt 0 ved hhv. ener- og toer-komplement

\[
\begin{align*}
000...00011 &= +3 \\
000...00010 &= +2 \\
000...00001 &= +1 \\
000...00000 &= +0 \\
111...11111 &= -0 \\
111...11110 &= -1 \\
111...11101 &= -2 \\
111...11100 &= -3 \\
111...11010 &= -4 \\
\end{align*}
\]

Som man ser gir ener-komplement to ulike bit-representasjoner av 0, eller om man vil de to nullene +0 og −0.
Med en-bytes tall får vi 00000000 for +0 og 11111111 for -0, noe som fører til det eiendommelige resultat at

\[-0 \quad \begin{align*} + 1 & \quad 11111111 & +00000001 \quad +0 \quad 00000000 \end{align*}\]

Poengset er ganske enkelt at \(1_+ + 1_- = 0\). Når vi starter addisjonen fra høyre får vi dermed hele tiden 0 i den posisjonen vi er, mens vi dra med oss \(1\) til neste posisjon. I siste posisjon får vi også 0, mens 1 forsvinner ut til venstre.

Ener-komplementering er generelt raskere enn toer-komplementering, men siden vi må sikre oss mot ovennevnte resultat, må vi i praksis utføre den relevante testen for alle aritmetiske operasjoner. Toer-komplement synes alt i alt å være å foretrekke, siden den, én gang for alle, sparer oss for den forlegenhet ovennevnte resultat ville ha satt oss i.

### 6.4.2 Tallområde

Når man lagrer tall med fortegn, må man bruke ett bit til dette, og får dermed færre bits til representasjon av selve tallverdien (absoluttverdi) og en mindre maskimal tallverdi — f.eks slik at mens et byte uten fortegn kan ha 256 verdier i intervallet 0..255, kan et byte med fortegn ha 255 verdier i intervallet -128..0..+127 —

\[11111111..01111111\] (merk at første bit fra venstre både er fortegnsbit og et siffer i tallet).

<table>
<thead>
<tr>
<th>med fortegn</th>
<th>utan fortegn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1</td>
<td>127, hex = (7F)</td>
</tr>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>126, hex = (7E)</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>-1, hex = (-01)</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>0, hex = (00)</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>-127, hex = (-7F)</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>-128, hex = (-80)</td>
</tr>
</tbody>
</table>

### 6.4.3 Fortegns-bit

Man kan teste om tall lagret som toer-komplement er negative ved å se på det venstre bit-et. Det er alltid 0 for positive tall (tall >= 0), og 1 for negative tall. Dette bit-et kalles derfor *fortegns-bitet*.

### 6.4.4 Tall med og uten fortegns-bit

Datamaskinen vet ikke uten videre om den har et stort positivt tall uten fortegns-bit, eller et negativt tall med fortegns-bit. I mange språk forutsettes det riktigognk at alle tall har fortegn, slik at dette er lagt inn i språket, men i C operes det eksplisitt med tall både med og uten fortegn (*signed og unsigned*), og vi må selv holde rede på hva som er hva.

Dette er ellers mest typisk for programmering på maskinennivå:

- **Maskinen lager og opererer på bit-mønstre.**
- **Alt ansvaret for fortolkning av bit-mønstrene ligger på programmereren.**
- **Av og til er det nyttig å kunne tolke et bit-mønster på ulike måter.**
7 Den syntaktiske formen til et C-program

7.1 Minimumsprogrammet Hello_World

Så å si alle lærebøker i programmering har med et kort eksempelprogram (i tråd med Kernighan & Ritchie's foregangseksempel — K&R [1978]) som skriver ut "Hello, World!" til skjermen:

Scheme:
(display "Hello, World!")
(begin)

Pascal:
program Hello_World;
begin
  WriteLn("Hello, World!");
end.

C:
#include <stdio.h>
int main(void)
{
  printf("Hello, World \n");
  return 0;
}

C++:
#include <iostream.h>
int main(void)
{
  cout << "Hello, World!" << endl;
}

Studer disse eksemplene nøye og sammenlign C og C++ variantene med den varianten du kjenner best fra før. Alle er komplette programmer som gir utskriften Hello World! til det vinduet der interaksjonen med bruker foregår.

7.2 Hovedprogrammet

Et C-program er, i likhet med et Scheme-program, essensielt en liste av funksjonsdefinisjoner:

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define (main ...)</td>
<td></td>
<td>int main()</td>
</tr>
<tr>
<td>begin</td>
<td>begin</td>
<td>{</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>return 0;</td>
</tr>
<tr>
<td>...)</td>
<td>end.</td>
<td>}</td>
</tr>
</tbody>
</table>

NB! Hovedprogrammet i C er selv en funksjon ved navn main:

Når et C-program lenkes, identifiserer lenkeren funksjonen med navnet main, og lær kallet på denne bli startpunktet i det ferdig lenkede programmet. Finner ikke lenkeren noen slik funksjon, kan den heller ikke fullføre lenkingen. C, i motsetning til C++, krever at main er en int-funksjon.
7.3 Funksjoner

En C-funksjon består alltid av fire deler:
1 typen til returverdien, eller void, hvis det ikke skal gis returverdi.
2 navnet på funksjonen.
3 argumentlisten i parentes med komma mellom og typeangivelse av hvert argument,
   eller en tom parentes (eller (void)), hvis funksjonen ikke tar argumenter.
4 funksjonskroppen — omsluttet av { og }.

Eventuelt returverdie angis med en return-setning.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define (sum a b)</td>
<td>function sum(a, b: real): real;</td>
<td>float sum (int a, int b)</td>
</tr>
<tr>
<td>(+ a b))</td>
<td>begin</td>
<td>{</td>
</tr>
<tr>
<td>sum := a + b;</td>
<td>end;</td>
<td>return a + b;</td>
</tr>
</tbody>
</table>

7.4 Deklarasjon av variabler

Variabler i C kan, som nevnt over, deklareres i eller mellom funksjonene. En deklarasjon i en funksjon er lokal, mens en deklarasjon mellom funksjoner er global. En global variabel er tilgjengelig i alle etterfølgende funksjoner, og kan gjøres tilgjengelig i en forutgående ved at den deklareres som extern der.

I C må lokale deklarasjoner plasseres først i funksjonen, mens de i C++ kan plasseres foran en hvilken som helst setning, så lenge en variabel er deklarert før den brukes.

En deklarasjon består av to deler pluss en siste valgfri del.
1 Variabelens type.
2 Variabelens navn. Flere variable, adskilt med komma, kan deklareres i samme setning.
3 Tilordning av en initialverdi.

Verdien til en variabel uten angitt initialverdi er ubestemt (vilkårlig).

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define a 0)</td>
<td>a: integer;</td>
<td>int a;</td>
</tr>
<tr>
<td>(define b 0)</td>
<td>b: integer;</td>
<td>int b := 0;</td>
</tr>
<tr>
<td>(let ((i 0) (j 0))...)</td>
<td>i, j: integer;</td>
<td>int i, j;</td>
</tr>
<tr>
<td>(let ((x 3.14) (y 0)) ...)</td>
<td>x, y: real;</td>
<td>float x = 3.14, y = 0;</td>
</tr>
<tr>
<td></td>
<td>x := 3.14; y := 0;</td>
<td></td>
</tr>
</tbody>
</table>
Det er visse semantiske forskjeller mellom de tre språkene her. Dvs. forskjellen gjelder både programmeringsspråkene som sådan, og den måten vi beskriver disse språkene på.

Variabler i Scheme, er det samme som variabler i matematiske funksjoner — altså formelle argumenter. Medmindre vi utfører destruktive operasjoner på en variabel, vil dens verdi være konstant. Hvorvidt vi kan utføre destruktive operasjoner på en variabel, slik at dens verdi kan variere, har ingenting å gjøre med at den er en variabel. Variabiliten til en variabel i Scheme består snarere i at funksjonen der den er argument, kan kalles med ulike aktuelle parametre.

I Scheme kan vi også sette navn på verdier som gjelder globalt, vha. (define navn verdi), eller er begrenset til en program-blokk vha. bl.a. (let ((navn verdi)) ...). Ønsker vi å skille terminologisk mellom variabler og denne typen bindinger, må vi kalle de siste nettopp navn. Variabler i Pascal, C og andre prosedyrale språk er navngitte lokasjoner med verdier som kan, og normalt vil, variere. Deklarasjonen av en variabel behøver ikke å knytte denne til en verdi — og hvis den gjør det, sier vi at verdien er initial, nettopp fordi den normalt vil variere. Ønsker vi å sette navn på verdier som ikke skal variere — altså på konstanter —, har både Pascal og C syntaktiske mekanismer for dette.

Formelle argumenter i Pascal og C opptrer som lokale variabler (i "prosedyral" betydning) i prosedyren, og får sine initielle verdier satt til de aktuelle parametrene når prosedyren kalles.

Variabelens type kan i C være gitt ganske enkelt f.eks. en int, eller den kan være nærmere spesifisert og/eller kvalifisert.

Spesifikasjonen kan omfatte
- en nærmere typebestemmelse: signed el. unsigned for char og int, short for int og long for int og double, og
- en angivelse av avhengig av (eller et forslag om) lagringsplass (se under).

Kvalifikasjonen kan være const, som angir at variabelen ikke kan endres, og som dermed bare gir mening for variabler som også initialiseres (og volatile, som er uten praktisk betydning).

Angivelse av lagringsplass kan være én av følgende:
- auto(matik) er den stilltiende angivelsen for lokale variable i funksjoner. For disse settes det av plass på stakken når funksjonen kalles, og plassen frigjøres når funksjonen er ferdig. En eventuell initiiering (f.eks. int i = 0;) utføres hver gang variabelen oppstår. static angis for lokale variable i funksjoner som det skal settes av plass til idet programmet starter. De ligger altid på samme sted og bevarer verdien fra ett kall til det neste. En eventuell initiiering foretas dermed kun én gang. (static er den stillitiende spesifikasjonen for globale variabler, og det gir liten mening å angi disse som auto, i og med at disse allikevel instansieres en gang for alle og hele tiden ligger på samme plass).
- extern angir at variabelen eller funksjonen er definert et annet sted. Bindingen til den definerte enheten skjer under linkingen.
- register angir at vi, om mulig, vil ha variabelen i et register for raskere aksess. Dette kan være aktuelt for mye brukte variabler.

7.5 Kommentarer

Kommentarer kan plasseres hvor som helst mellom /* og */, eller etter og på samme linje som //.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>; en kluge /* En kluge <em>/ /</em> En kluge */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>; og en til { og en til } // og en til</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Kommentaravgrenserne /* og */ er greie å bruke når man enten ønsker å legge kommentarer og kode om hverandre på én og samme linje, eller når man skal legge inn større kommentaravsnitt som går over flere linjer.

Avgrenseren // er nyttig for løpende kommentarer til sekvenser av programsetninger.
8 Datatyper

8.1 Tall

Egentlig har C kun de 4 basistypene char, int, float og double som alle er talltyper. De øvrige i tabellen under er bare ulike spesifikasjoner av disse.

<table>
<thead>
<tr>
<th>navn</th>
<th>kortnavn</th>
<th>bytes</th>
<th>spenn</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>char</td>
<td>1</td>
<td>-128..127</td>
</tr>
<tr>
<td>unsigned char</td>
<td></td>
<td>1</td>
<td>0 .. 255</td>
</tr>
<tr>
<td>signed short int</td>
<td>short</td>
<td>2</td>
<td>-32768..32767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td></td>
<td>2</td>
<td>0..65535</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td></td>
<td>systemavh. (typisk 2 el 4 bytes)</td>
</tr>
<tr>
<td>unsigned int</td>
<td></td>
<td>4</td>
<td>-2147483648..2147483648</td>
</tr>
<tr>
<td>signed long int</td>
<td>long</td>
<td>4</td>
<td>-2147483648..2147483648</td>
</tr>
<tr>
<td>unsigned long int</td>
<td></td>
<td>4</td>
<td>0..4294967295</td>
</tr>
<tr>
<td>enum</td>
<td></td>
<td></td>
<td>som for int</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>4</td>
<td>-3.4E38..3.4E38, 7-8 desimaler. °</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td>8</td>
<td>-1.7E308..1.7E308, 15-16 desim.</td>
</tr>
<tr>
<td>long double</td>
<td></td>
<td>10</td>
<td>-1.2E4932..1.2E4932, 19-20 des.</td>
</tr>
</tbody>
</table>

En enumerasjon (oppregning) er en liste med navngitte konstanter — f.eks. slik:

```c
enum dager {MAN, TIR, ONS, TOR, FRE, LOR, SON};  // 0, 1, ..., 6
```

Hvis ikke annet er angitt, får første konstant i listen verdien 0, mens de øvrige inkrementeres med 1.

Man kan spesifisere en eller flere konstanter. En eller flere uspesifiserte konstanter som følger etter en spesifisert, inkrementeres med 1 i forhold til den spesifiserte.

```c
enum maaneder {JAN=1,FEB,MAR,APR,MAI,JUN,JUL,AUG,SEP,OKT,NOV,DES};  // 1, 2, ..., 12
enum effekttegn {BS = '', TAB = '	', NL = '
', CR = ''};
```

° 3.4E38 betyr 3.4 \times 10^{38} = 3400000000000000000000000000000000000000000.

Hvordan kan et 39-sifret tall representeres med bare 4 bytes? Svaret er at tallet deles opp, med 3 bytes til mantissen (de interessante sifrene, her 3.4) og ett byte til eksponenten (her tier-potensen 38).

8.2 Booleske

C har ingen type for å lagre logiske verdier (som boolean i Pascal).

I stedet benyttes heltall:

<table>
<thead>
<tr>
<th>Verdi</th>
<th>Betydning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>usann (false)</td>
</tr>
<tr>
<td>≠0</td>
<td>sann (true)</td>
</tr>
</tbody>
</table>

Dette minner om Scheme der alt som ikke evaluerer til #f enten evaluerer til #t eller er ekvivalent med #t. Likheten er imidlertid ikke fullstendig, ettersom bare integrale verdier (tegn, heltall og enumererte) kan tolkes som boolean i C.
Dette betyr at testen i if- og while-setninger (valg- og løkke-setninger) gjelder om noe er 0 eller ≠0 — og det er derfor det samme om vi skriver if (a != 0) ... eller if (a) ...

C++ har typen bool med verdiene true og false. Alle tegn, heltall (inklusive pekere / adresser) og enumererte verdier er kompatible med booleske. Gitt heltallet x, evaluerer (bool)x til false for x lik 0, og til true for alle andre verdier av x (tolket som en boolesk). Motsatt evaluerer (int)false og (int)true til hhv. 0 og 1 (false og true tolket som heltall).

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>(let ((b #f))</td>
<td>b boolean;</td>
<td>int b = 0;</td>
<td>bool b = false;</td>
</tr>
<tr>
<td>(set! b #f)...</td>
<td>b := false;...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(set! b #t)...</td>
<td>b := true;...</td>
<td>b = 1;...</td>
<td>b = true;...</td>
</tr>
<tr>
<td>(set! b 'true)...</td>
<td></td>
<td>b = 255;...</td>
<td></td>
</tr>
<tr>
<td>(if b ...) ...</td>
<td>f b then ...;</td>
<td>if (b) ...;</td>
<td>if (b) ...;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (b!=0) ...;</td>
<td></td>
</tr>
</tbody>
</table>

8.3 Tegn

Også for representasjonen av tegn, brukes heltall, dvs. i C brukes tegn og heltall om hverandre. Riktig nok finnes det en egen type char for tegn, men dette er egentlig en ett-bytes heltallstyp. (Om en verdi fremtrer for bruker som et tegn eller ikke, er gitt ved den måten den skrives ut på.)

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define (char-func c)</td>
<td>c char;</td>
<td>char c; int i;</td>
</tr>
<tr>
<td>...)</td>
<td>i integer;</td>
<td>unsigned char uc;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned int ui;</td>
</tr>
</tbody>
</table>

Tegnkonstanter skrives i enkle anførselstegn — enten direkte, eller, som i Scheme, vha. såkalte escape sequences. En av de vanligste er \n som angir linjeskift.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>#\a #\newline</td>
<td>'a' char(13)</td>
<td>'a' '\n'</td>
</tr>
</tbody>
</table>

8.4 Integrale og flytende typer

Heltall, tegn og booleske er alle integrale typer (in-tenger — u-berørt, hel; fra lat. in og tangere) og syntaktisk utbyttbare.

char c = 'a'; unsigned char uc = 48; int i = 1024; long n = 4294967295;
c = 'a'; uc = c; i = c; c = n; i = n;
c = uc; uc = i; i =\'\n\'; n = 'x';

I praksis må man passe på at det er plass nok til de verdiene som tilordnes en variabel. Settes f.eks. en signed char lik 255, får variabelen verdien -1 (se Tall med fortegn).
Tilordningen $c = n$; over gir samme resultat

$255_{10} = 11111111_{2}$ og $4294967295_{10} = 11111111111111111111111111111111_{2}$,

mens resultatet av tilordningen $i = n$; er systemavhengig. En int tar enten to bytes, og $i = 4294967295 \rightarrow -1$,
eller fire bytes, og $i = 4294967295 \rightarrow 4294967295$. I Windows / Visual C++ tar en int 4 bytes.

De integrale typene står i motsetning til de flytende typene — typer hvis verdier ikke kan bestemmes fullstendig i en endelig representasjon. Også disse er syntaktisk utbyttbare, men eventuelle desimaler forsvinner når et flyttall tilordnes en integral variabel.

Ved en tilordning som kan føre til tap av signifikante bits, gir Visual C++-kompilatoren en advarsel.

I tillegg de integrale og flytende tpene har vi en spesialtype som har den ene verdien void.

8.5 Tegnstrenger

En lengstreng (eller bare streng) er en vektor med tegn (hvordan vi beskriver og opererer på vektorer i C, er viktige temata som vi skal komme tilbake til).

Tegnstrengkonstanter skrives i doble anførselstegn — f.eks slik: "tekst".

I C kan vi legge inn spesialtegn i teksten vha. escape sequences.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define (string-func s)...))</td>
<td>s string[31];</td>
<td>char s[32]</td>
</tr>
<tr>
<td>(string-func &quot;En tekst&quot;)</td>
<td>s := 'En tekst';</td>
<td>strcpy(s, &quot;En tekst\n&quot;)</td>
</tr>
<tr>
<td>(let ((t &quot;Hei!&quot;))...)</td>
<td>t := 'Hei!';</td>
<td>char t[] = &quot;Hei!&quot;;</td>
</tr>
</tbody>
</table>

I Scheme har en streng alltid samme lengde, og vi kan bare endre de enkelte tegnene innenfor strengen.

I Pascal angis makslengden til strengen i klamme etter typeangivelsen (string er kort for string[255]). Den faktiske lengden er til enhver tid gitt ved verdien i strengens 0-te byte.

I C angir vi makslengden til strengen + 1 i klamme etter variabelnavnet. Det ekstra byteet brukes til å terminere strengen med tegnet 0 (dvs. ‘\0’ — ikke ‘0’), og strengens faktiske lengde er til enhver tid gitt ved antall tegn fremt til terminasjonstegnet. Når vi initialiserer strengen i deklarasjonen, uten å angi noe tall i klamme etter strengnavnet, må vi gå ut fra at makslengden til strengen er lik lengden til den initielle strengen.

I C kan vi, som vist over legge ny-linje-tegnet (‘\n’) inn i en streng — og skrives strengen ut, skrives det også ut et linjeskift (dette får vi ikke til i Scheme og Pascal).

De 32 første tegnen i tabellen er såkalte "kontrollspreker", og det gjør de i SLI 280, må vi definere våre egne strengtyper med tilhørende funksjoner. — Mer om det siden.

De 32 første tegnen i tabellen er også kjent som "kontrolltegn", eller det vi også kunne kalle "effekt-tegn" — så som tabulator og linjeskift.

Siden tegntypen char stilltillende oppfattes som signed char, og tegnstrøgene er vektorer av char, kan vi strengt tatt ikke representere annet enn de 128 første tegnene i tabellen over i en C-streng. Plassene -127..-1 finnes ikke i tabellen. Ønsker vi å behandle tegnene 128..255, noe vi så absolutt gjør i SLI 280, må vi definere våre egne strengtyper med tilhørende funksjoner. — Mer om det siden.

![Tabellen for Latin-1](image-url)
9 I/O

9.1 Skriving til skjermen og lesende fra tastaturet i C

For å skrive til skjermen og lese fra tastaturet i et konsoll-program bruker vi i C implisitt de to objektene `stdout` og `stdin` som er definert i `stdio.h`. Assosiert til disse er bl.a. funksjonene `printf` og `scanf`.

**Skriving**

Første argument til `printf` er en *formateringsstreng*. Denne kan inneholde spesialtegn som linjeskift samt plassholdere for uttrykk (variabler eller beregninger) som skal evalueres, evt. konverteres til strenger, og skrives ut. En plassholder består av tegnet % fulgt av en bokstavkode for type.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(display &quot;Hei, &quot;)</td>
<td>Write(&quot;Hei, &quot;);</td>
<td>printf(&quot;Hei, &quot;);</td>
</tr>
<tr>
<td>(display &quot;dere!&quot;)</td>
<td>WriteLn(&quot;dere!&quot;);</td>
<td>printf(&quot;dere!\n&quot;);</td>
</tr>
<tr>
<td>(display a)</td>
<td>WriteLn(a,&quot; og &quot;,b);</td>
<td>printf(&quot;%d og %d\n&quot;, a, b);</td>
</tr>
<tr>
<td>(display &quot; og &quot;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(display b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(newline)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Her er a og b tallvariable, og koden d angir tall. Både i Pascal og C kan tall formateres slik at de får fast bredde og evt. et gitt antall desimaler.

De typekodene vi er interessert i, er  

d for heltall,  
f for flyttall,  
c for tegn,  
s for streng.

**Formatering**

I plassholderen kan man mellem tegnet % og bokstavkoden plassere et tall for å angi minste utskriftsbredde. Tar verdien mindre plass en utskriftsbredden, høyrejusteres utskriften, medmindre vi har angitt at den skal venstrejusteres. Vi angir venstrejustering ved å plassere et minustegn mellom % og breddetallet.

```c
printf("%d", 123);   printf("-%5s", "hei");
```

I tillegg kan vi angi antall desimaler for et flyttall ved å plassere et punktum fulgt av et tall foran bokstavkoden.

```c
printf("%5.2f", 1.234);   printf("%5.2f", "hei");
```


```c
printf("-%s", bredde, streng);   printf("%-*s", bredde, streng);   printf("%*.f", bredde, des, tall);   printf("%*d", s);   printf("%d", s);
```

**Lesing**

Også ved innlesning brukes en streng med plassholder som første argument (når vi leser fra tastaturet) til `scanf` — f.eks. %c for tegn, %d for heltall og %f for flyttall. som her:

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(set! a (read))</td>
<td>(Read a);</td>
<td>scanf(&quot;%d&quot;, &amp;a);</td>
</tr>
<tr>
<td>(set! x (read))</td>
<td>(Read x, y);</td>
<td>scanf(&quot;%f%f&quot;, &amp;x, &amp;y);</td>
</tr>
<tr>
<td>(set! y (read))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legg merke til & foran variabelnavnene i C-kolommen. Dette er adresseoperatoren som `mà` være der. Operatoren & gir oss adressen til den plassen variabelnavnet er bundet til, slik at `scanf` får et sted å legge den innlesste verdien.
Når vi minner oss selv om at lesing i Scheme ikke må være destruktiv, som her, illustrerer eksempel et viktig forskjell mellom Scheme, Pascal og C som angår parameteroverføring. I C brukes bare verdioverføring, mens vi i Pascal og C++ i tillegg har referanseoverføring. Verdioverføring innebærer at argumentet alltid evalueres ved kallet og at resultatet av evalueringen bindes til en lokal variabel i prosedyren. Referanseoverføring derimot innebærer at prosedyren arbeider direkte med den overførte variabelen, og dermed gjøres eventuelle endringer av variabelens verdi gjeldende etter at kallet er utført. Dette er hva vi ønsker, når vi leser, og i Pascal og C++ er dette sikret ved at alle argumentene til leserutinene referanseoverføres. I C derimot tyr vi til et triks, idet den verdiene vi overfører til scanf, er adressen til variabelen, i stedet for verdien til variabelen selv. Dermed kan scanf legge inn det den leser på de adressene den får. Plasholderkodene sikrer at det ikke legges inn flere byte enn det det er plass til på de ulike adresseene. Vi kan kalle dette adresseoverføring eller quasi referanseoverføring. Referansen, dvs. adressen, til en variabel får vi tak i vha. operatoren & som plasseres foran variabelnavnet.

Scheme er på sin side et funksjonelt språk, og også leseprosedyren er definert som en funksjon — en funksjon som returnerer det den leser (enten et tegn, om vi bruker read-char, eller et forutsettingvis velformet Scheme-uttrykk, om vi bruker read). En annen ting er at vi ved destruktive operasjoner — typisk på strenger og vektorer — kan oppnå tilsvarende effekter i Scheme som ved adresseoverføring i C.

De ulike formene for parameteroverføring er nærmere behandlet i avsnittet Verdi-, referanse- og navneoverføring av parametre.

9.4 Tegnvis og linjevis skriving og lesing i C

I tillegg til printf og scanf har vi følgende funksjoner for skriving til skjerm og lesing fra tastatur:
- putchar(c); skriver tegnt c til skjermen.
- puts(s); skriver strengen s til skjermen.
- getchar(); leser og returnerer et tegn fra tastaturet.
- gets(s); leser fra tastaturet frem t.o.m. linjeslutt (enter), plasserer de leste tegn i strengen s, men erstatter linjeslutt med strengterminatoren ‘\0’, og returnerer en peker til strengen.

9.1 Skriving til skjermen og lesing fra tastaturet i C++

For lesing fra tastatur og skrivning til skjerm i C++ bruker vi objektene cin og cout som er definert i iostream.h.

Operatorene << og >> gir hendige mekanismer for hhv. skrivning og lesing av ulike typer. << og >> er eksempler på overlappende operatorer, idet de brukes både for bitvis forskyvning og for I/O (bare i C++). Overlissing av operatorer er bare mulig i forbindelse med klasser. Operatorene << og >> er definert hhv. for klassene ostream og istream.

Gitt heltallet x med verdien 5, vil setningen

cout << "x = " << x << endl;

gi utskiften

x = 5 pluss linjeskift.

printf("Hei, ");
printf("dere!\n");
printf("%d og %d\n", a, b);

endl er en såkalt manipulator som << kan operere på. Den gir et linjeskift (og tømmer utskriftsbufferen — det vi kan betrakte som løpende linje).

Gitt tallvariablene x og y, vil følgende setning gi x og y verdiene til de to neste tallene (fulgt av enter) som evt. skrives på tastaturet:

```
cin >> x >> y;
```

scanf("%d", &x);
scanf("%f%f", &x, &y);
getchar();
9.2 Tegnvis og strengvis skriving og lesing i C++

I tillegg til operatorene << og >> er det definert diverse metoder for de klassene cout og cin er instanser av.

* cout.put(c) skriver tegnet c.

F.eks. vil cout.put('a'); gi utskriften a

* cout.write(p, n) tar en peker til en tegnstreng og et antall som parametre og skriver så mange tegn som det er angitt, eller som strengen inneholder til fil eller skjerm.

F.eks. vil cout.write("hallo\n", 6); gi utskriften hallo pluss linjeskift.

* cin.get() leser og returnerer neste tegn i innlesningsbufferen.

* cin.get(ps, n) tar (en peker til) en streng og et antall tegn som parametre, leser det gitte antall tegn, eller frem til, men ikke til og med, linjeslutt, og plasserer de leste tegnene, bortsett fra evt. linjeslutt i strengen.

* cin.getline(ps, n) tar (en peker til) en streng og et antall tegn som parametre, leser det gitte antall tegn, eller frem til og med linjeslutt, og plasserer de leste tegnene, bortsett fra evt. linjeslutt i strengen. (getline gjør altså det samme som get med argumenter — og leser i tillegg linjeslutt.)

* cin.ignore(n, s) tar et antall tegn og et stopptegn s som parametre og leser det gitte antall tegn eller t.o.m. s.

Vi kommer nærmere tilbake til disse og, noen flere senere — bl.a. flere varianter av put, write, get, getline og ignore med ulike typer og antall parametre.

9.5 Et eksempel på bruker-I/O

Regn om et gitt antall tommer til fot og tommer:
- Les første antall tommer.
- Gå deretter i løkke og beregn, skriv og les, inntil bruker svarer 0.

Scheme
(define (feet inches) (quotient inches 12))
(define (inch-rest inches) (remainder inches 12))
(define (read-loop inches)
 (if (> inches 0)
   (begin
     (if (and (> inches 0) (<= inches 999))
       (begin (display inches) (display " tommer = "))
       (display (feet inches)) (display " fot og ")
       (display (inch-rest inches)) (display " tommer.")))  
     (display "Programmet godtar kun verdiene 0-999!"))
   (newline)
     (display "Gi et mål i tommer (avslutt med 0): ")
     (read-loop (read))))
  
(define (top-level)
  (display "Gi et mål i tommer (avslutt med 0): ")
  (read-loop (read)))
Pascal

program Inches;

function Feet(inches: integer): integer;
begin
  Feet := inches div 12;
end;

function InchRest(inches: integer): integer;
begin
  InchRest := inches mod 12;
end;

var
  inches: integer;

begin
  Write('Gi et mål i tommer (avslutt med 0): ');
  ReadLn(inches);
  while (inches > 0) do
  begin
    if (inches > 0) and (inches <= 999) then
    begin
      WriteLn(inches, ' tommer = ', Feet(inches), ' fot og ', InchRest(inches), ' tommer.');
    end
    else
    begin
      WriteLn('Programmet godtar kun verdiene 0-999!');
      Write('Gi et mål i tommer (avslutt med 0): ');
    end
  end.
end.

C
#include <stdio.h>

int feet(int inches) { return inches / 12; }

int inch_rest(int inches) { return inches % 12; }

int main(void)
{
  int tmr;
  printf("Angi et mål i tommer: ");
  scanf("%d", &tmr);
  while (inches > 0)
  {
    if (tmr > 0 && tmr <= 999)
    {
      printf("%d tommer = %d fot og %d tommer\n", tmr, feet(tmr), inch_rest(tmr));
    }
    else
    {
      printf("Programmet godtar kun verdiene 0-999!\n");
      printf("Gi et nytt mål i tommer (0 for avslutning): ");
      scanf("%d", &tmr);
    }
  }
}
#include <iostream.h>

int Feet(int inches) { return inches / 12; }

int InchRest(int inches) { return inches % 12; }

int main(void)
{
    int inches;
    cout << "Angi et mål i inches: ";
    cin >> inches;
    while (inches)
    {
        if (inches > 0 && inches <= 999)
        {
            cout << inches << " tommer = " << Feet(inches)
            << " fot og " << InchRest(inches) << " tommer " << endl;
        }
        else
        {
            cout << "Programmet godtar kun verdiene 0-999!" << endl;
        }
        cout << "Gi et nytt mål i tommer (0 for avslutning): ";
        cin >> inches;
    }
}

9.6 Et eksempel på fil-I/O

I dette eksempelet leses det fra fil. Kort fortalt er variabelen fil i alle variantene av CountChars/count-chars et innlesingsobjekt som er knyttet til en fil som alt er åpen. I C-varianten kan vi sammenligne infile med objektet stdin som alltid er knyttet til tastaturet (når stdio.h er inkludert), og som er en slags stilltiende argument til alle de funksjonene vi bruker til å lese fra tastaturet. I C++-varianten kan vi sammenligne infile med objektet cin som spiller tilsvarende rolle som stdin (når iostream.h er inkludert), men som må angis eksplisitt når det brukes. — For øvrig gjennomgås filbehandling i sin helhet i kapittel 20 Filbehandling.

Scheme
(define (count-chars-helper infile n)
    (let ((c (read-char infile)))
        (if (eof-object? c)
            n
            (count-chars-helper infile (+ n 1)))))

(define (count-chars)
    (let* ((infile (open-input-port "test.txt"))
            (count (count-chars-helper infile 0)))
        (close-input-port infile)
        count))
Pascal

```pascal
procedure CountChars (var infile: text; var count: integer);
    var 
c: char;
begin 
count := 0;
Read(infile, c);
    while not EoF(infile) do 
begin 
count := count + 1;
Read(infile, c);
    end
end;
```

```pascal
var 
count: integer;
infile: text;
begin 
Reset(infile, "test.txt");
TellTegn(infile, count);
Close(infile)
end;
```

C

```c
#include <stdio.h>

int CountChars(FILE *infile)
{
    int count = 0;
    char c = fgetc(infile);
    while (!feof(infile))
    {
        count++;
        c = fgetc(infile);
    }
    return count;
}

int main()
{
    FILE *infile = fopen("test.txt", "r");
    int count = CountChars(infile);
    fclose(infile);
    return 0;
}
```

Konstruksjonen **init; while (test) {setn; progress;}** gir en løkke der setn gjentas inntil test er oppfylt, og der progress forutsettes å bidra til endringer som er relevante for testen. — Se forøvrig Setninger.
C++
#include <fstream.h>
int CountChars(ifstream& infile)
{
  int count;
  for (count=0;! infile.eof(); infile.get())
    count++;
  return count;
}
void main()
{
  ifstream infile("test.txt");
  int count = CountChars(infile);
}

Konstruksjonen for (init; test; progress) setn; er ekvivalent med
konstruksjonen init; while (test) {setn; progress;} — Se forovrig Setninger.

& etter typeangivelsen i parameterlisten til CountChars betyr at infile er en referanse til et ifstream-objekt — dvs. et lokalt alias for objektet under utførelsen av prosedyren. Eventuelle endringer gjelder det refererte objektet.

NB! Det er overhodet ingen sammenheng mellom & qua referansespesifikator, slik det er brukt her, og & qua adresseoperator.

ifstream er en klasse, og infile i main er en instans / objekt av denne klassen. Når programmet forlater den setningsblokken der et objekt er deklarert (her kroppen til main), kalles automatisk objektets dеструктор — en prosedyre som, hva et ifstream-objekt angår, bl.a. lukker den åpne filen. — Se Klasser og objekter.
10 Setninger
10.1 Setningstyper

Setninger er instruksjoner (imperativer) som utføres mht. deres effekter, og har ikke selv verdier. Setninger inneholder imidlertid uttrykk som har verdier — konstanter, variabler, operasjoner, og/eller funksjonskall.

Setninger utføres sekvensielt — én etter én.

Alle bieffekter av eventuelle operasjoner og funksjonskall i en setning, finner sted før neste setning utføres.

Noen typiske setninger:

<table>
<thead>
<tr>
<th>setning</th>
<th>kategori</th>
<th>effekt</th>
<th>bieffekt</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0;</td>
<td>tilordning</td>
<td>i får verdien 0.</td>
<td></td>
</tr>
<tr>
<td>i++;</td>
<td>tilordning ved</td>
<td>i’s verdi økes med 1 til i.</td>
<td></td>
</tr>
<tr>
<td>i += 2</td>
<td>tilordning ved</td>
<td>i’s verdi økes med 2 til 3.</td>
<td></td>
</tr>
<tr>
<td>(j = -2;</td>
<td>sammensatt: tilordning.</td>
<td>j får verdi -2, og i’s verdi</td>
<td></td>
</tr>
<tr>
<td>i--;</td>
<td></td>
<td>reduseres med 1 til 2.</td>
<td></td>
</tr>
<tr>
<td>i = abs(j)</td>
<td>funksjonskall,</td>
<td>i får verdien 2.</td>
<td></td>
</tr>
<tr>
<td>if (i &gt; 0)</td>
<td>test, tilordn.v.inkrem.,</td>
<td>j får verdi til 1 (2) før inkrem. av i.</td>
<td>i’s verdi økes med 1 til 3.</td>
</tr>
<tr>
<td></td>
<td>tilordning.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (i &gt; 0)</td>
<td>test, tilordn.v.inkrem.,</td>
<td>j får verdi til 1 (4) etter inkrem. av i.</td>
<td>i’s verdi økes med 1 til 4.</td>
</tr>
<tr>
<td></td>
<td>tilordning.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for (i = 0;</td>
<td>løkke: tilordning</td>
<td>j får verdiene 4, 5, 7.</td>
<td></td>
</tr>
<tr>
<td>i &lt; 4;</td>
<td>tilordning ved inkrem.,</td>
<td>i får verdiene 0, 1, 2, 3.</td>
<td></td>
</tr>
<tr>
<td>i++)</td>
<td>tilordning ved aritmetikk</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I C avsluttes alle setninger med semikolon ; bortsett fra sammensatte setninger som avsluttes med ).

Det er fremfor alt forekomsten av imperative setninger som skiller prosedyrale språk fra funksjonelle.

10.2 Tilordningsuttrykk

I prosedyral programmering er endring av variablers verdier en i høyeste grad vanlig operasjon, i motsetning til i funksjonell programmering, der dette bare gjøres unntaksvis — eller rett og slett ikke er tillatt, hvis det dreier seg om strengt funksjonell programmering. En endring av en variabels verdi kaller vi en verdi tilordning.

I Scheme skilles det mellom bindinger og tilordninger. En binding etableres mellom en variabel og en plass (location) i hukommelsen bl.a. for et formelt argument ved et funksjonskall og ved bruk av en bindingskonstruksjon som let. Ved en tilordning av en verdi til en variabel, endres innholdet i hukommelsen på den plassen variabelen er bundet til, til den gitte verdi.

Vi karakteriserte nettop en tilordning (helt presist) som en operasjon. Det kan for så vidt være mange grunner til å betrakte tilordninger som setninger, og det er da også derfor vi omtaler dem i dette kapitlet. Strengt tatt gjelder imidlertid følgende: En tilordning er et uttrykk som evaluerer til den verdien den aktuelle variabelen tilordnes.
I C er en tilordning en operasjon utført med operatoren =, mens en binding er en variabeldefinisjon som effektereres ved programstart hvis variabelen er `static` (implisitt, global eller eksplisitt, lokal) eller ved et prosedyreкалл hvis variabelen er `auto` (matic) (implisitt, formelt argument, lokal).

### 10.3 Sammensatte setninger

Setninger kan være enkle eller sammensatte. En sammensatt setning består av flere enkeltsetninger — slått sammen til én med klammer: `{ setning1; setning2; ...; }.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(set! a 5)</td>
<td><code>a := 5;</code></td>
<td><code>a = 5;</code></td>
</tr>
</tbody>
</table>

En `l-value` er et uttrykk som kan forekomme på venstre side av tilordningsoperatoren — typisk en variabel, i motsætning til en konstant. En `l-value` kan settes i parentes uten å miste sin status som `l-value`.

### 10.4 Test-setninger

Generelt vil en test-setning omfatte en test, som består av et boolesk uttrykk, og en konsekvent, som er den setning som utføres hvis testen evaluerer til true. I tillegg kan en test-setning inneholde et alternativ.

#### 10.4.1 if-setningen

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(if (&lt; a 0) a -a)</td>
<td><code>if a &lt; 0 then</code></td>
<td><code>if (a &lt; 0)</code></td>
</tr>
<tr>
<td></td>
<td><code>a := -a;</code></td>
<td><code>a = -a;</code></td>
</tr>
<tr>
<td>(set! b</td>
<td><code>if a &lt; 0 then</code></td>
<td><code>if (a &lt; 0)</code></td>
</tr>
<tr>
<td>(if (&lt; a 0) -1 1))</td>
<td><code>b := 1;</code></td>
<td><code>b = 1;</code></td>
</tr>
<tr>
<td></td>
<td><code>else</code></td>
<td><code>else</code></td>
</tr>
<tr>
<td></td>
<td><code>b := 1;</code></td>
<td><code>b = 1;</code></td>
</tr>
</tbody>
</table>

NB! I C skal det stå parenteser rundt testen, og det skal være `;` foran `else`.

```c
function Trinary(a: integer) returns integer
begin
    Trinary := -1
    if a < 0 then
        return -1;
    else if a > 0 then
        return 1;
    else
        Trinary := 0
        return 0;
end;
```
10.4.2 switch-setningen

Både Scheme, Pascal og C har sin spesialiserte test-setning med flere nestede teser og alternativer der de enkelte testene alle består i en likhetsoperasjon på en gitt variabel og ulike mulige verdier for denne:

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(case a)</td>
<td>case a of</td>
<td>switch (a)</td>
</tr>
<tr>
<td>(((1) sent1)</td>
<td>1 :</td>
<td>{ case 1: sent1;</td>
</tr>
<tr>
<td>((2) sent2)</td>
<td>2 :</td>
<td>case 2: sent2;</td>
</tr>
<tr>
<td>(else sent3))</td>
<td>otherwise :</td>
<td>default: sent3;</td>
</tr>
<tr>
<td></td>
<td>sent3;</td>
<td>break;</td>
</tr>
<tr>
<td>end;</td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Disse setningene kan hver for seg skrives om slik:

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cond ((= a 1) sent1)</td>
<td>if (a = 1) then</td>
<td>if (a = 1) sent1;</td>
</tr>
<tr>
<td>((= a 2) sent2)</td>
<td>else if (a = 2)</td>
<td>else if (a = 2) sent2;</td>
</tr>
<tr>
<td>(else sent3))</td>
<td>else</td>
<td>else sent3;</td>
</tr>
<tr>
<td></td>
<td>sent3;</td>
<td></td>
</tr>
</tbody>
</table>

I motsetning til testen i if-setningen, må uttrykket a etter switch evaluere til et heltall.

case-lablene er ulike mulige verdier av a, og programmet hopper til labelen med den aktuelle verdien til a eller til default-labelen, hvis a’s verdi ikke er angitt.

En label er et navngitt sted i et program. I C følges et label-navn av kolon. (Label kan bl.a. oversettes med merkelapp og etikett. Her, som ellers der en engelske term er rimelig godt innarbeidet i norsk programmeringssjargon, foretrekker jeg imidlertid denne.)

break fører til et uthopp fra switch-setningen, etter utførelsen av den label-kontrollerte setningen.

break kan sløyfes etter en label, og i så fall fortsetter eksekveringen gjennom alle de etterfølgende setningene i switch-setningen — eller til neste break.

Eks:

```scheme
switch (a)
{
  case 1:  sent1;  // Er a == 1 utføres sent1, sent2 og sent3,
  case 2:  sent2;  // er a == 2, utføres sent2 og sent3, og
  default: sent3; // ellers utføres bare sent3.
}
```

I Pascal kan man angi flere verdier for samme label. I C oppnår vi tilsvarende effekt ved å liste opp de aktuelle lablene etter hverandre uten setninger mellom. Under er en lett modifikasjon av et eksempel fra K&R (s 59) der det dreier seg om å telle opp ulike kategorier tegn i en tekst (vi forutsetter her å ha en åpen fil).

<table>
<thead>
<tr>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure CountChars(var fil:text;</td>
<td>void CountChars(ifstream&amp; fil,</td>
</tr>
<tr>
<td>var whites,digits,others:integer);</td>
<td>int&amp; whites, int&amp; digits, int&amp; others)</td>
</tr>
<tr>
<td>const ht = Chr(9);</td>
<td>int ht = Chr(9);</td>
</tr>
<tr>
<td></td>
<td>if = Chr(10);</td>
</tr>
<tr>
<td></td>
<td>cr = Chr(13);</td>
</tr>
<tr>
<td></td>
<td>sp = ‘ ’;</td>
</tr>
<tr>
<td>var c: char;</td>
<td></td>
</tr>
</tbody>
</table>
begin /* kropp */    { /* kropp */
whites:=0; digits:=0; others:=0;
whites= 0; digits = 0; others = 0;
Read(fil, c);
while not eof(fil) do
begin
    case c of
    switch (c)
'0','1','2','3','4',
'5','6','7','8','9',
digits := digits + 1;
ht, lf, cr, sp:
whites := whites + 1;
otherwise
others := others + 1;
end;
Read(fil, c)
end;
end;
I noen Pascal-implementasjoner tillates den forkortede versjonen case c of '0'..'9':...
for sekvenser av verdier til samme label.
I Standard Pascal leses alle tegn som i filen er <= mellomrom ('  ' ) inn som mellomrom. I noen implementasjoner leses imidlertid alle tegn inn som de er — slik vi har forutsatt her.

10.4.3 Betingelses-uttrykk
C har en type uttrykk som har fellestrekk med en test-setning (men det er altså strengt tatt et *uttrykk og ikke en setning*). Uttrykket har formen test ? konsekvent : alternativ; og evaluerer til konsekvent hvis test evaluerer til true, og ellers til alternativ.

(C)

(set! b (if (a < 0 -1 1)))
if a < 0 then b := -1
elseif b := 1;

C-varianten her er ekvivalent med C-varianten av den andre *if*-setningen over.
Betingelses-uttrykket er, i motsetning til Cs *if*-setning, semantisk lik Schemes *if*-konstruksjon.

10.5 Løkker
Så å si alle realistiske (ikke-didaktiske) programmer inneholder en eller annen form for løkker der samme operasjon gjentas inntil et basistilfelle (kanhende ett av flere mulige) inntreffer. Dette vil ofte bestå i at en eller annen teller-variabel har fått eller passert en gitt grenseverdi.
I strengt funksjonelle språk utføres gjenitgarler formelt ved rekursjon. Dette er i ett henseende mindre effektivt enn iterasjon, idet programmet må trekke seg ut av rekursjonskjeden kall for kall, selv når det ikke er algoritmisk nødvendig. I Scheme kan vi oppnå den effektivitetsmessige gevisten ved iterasjon vha. halerekursjon, og vi har også en mekanisme for ekspisit iterasjon (*do* — se under). I begge tilfeller vil vi imidlertid ved hver gjenitgallage få nye bindinger mellom den eller de variablene som sjekkes mht. basistilfellelt (en *do*-setning per se medfører dermed ikke noen destruktive operasjon).
I C og andre prosedyrale språk utføres gjentagelser alt overveiende ved eksplisitt iterasjon, og det eller de basistilfeller det testes for, vil som regel involvere variabler som får sine verdier endret i løkken ved tilordning.

### 10.5.1 while-løkken

En while-løkke består av en test, typisk av verdien til en variabel i forhold til en grenseverdi, og en setning som skal utføres så lenge testen holder. Som regel vil dessuten konteksten forut for løkken inneholde en eller annen form for initialisering av den eller de variablene det testes for, mens setningen i løkken vil omfatte en eller annen form for endring, typisk inkrementering, av verdien til av den samme variabelen.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(do ((a a (- a 1)) (b b (* b a))) ((= a 0) b) effect)</td>
<td>while a &gt; 0 do begin effect; b := b * a; a := a - 1; end;</td>
<td>while (a &gt; 0) while (a &gt; 0)</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
 a &= 10; \quad a &= 10; \\
 b &= b * a; \quad b &= b * a; \\
 a &= a - 1; \quad a &= a - 1;
\end{align*}
\]

### 10.5.2 for-løkken

I for-løkken er det satt av plass for både initialiseringen og inkrementeringen i løkkens kontroll-setning (der testen allerede ligger i while-setningen) — for (initialiser; test; inkrementer)...;

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;som over&gt;</td>
<td>for a := 1 to 8 do</td>
<td>for (a = 1; a &lt;= 8; a = a + 1)</td>
</tr>
<tr>
<td></td>
<td>b := b * a;</td>
<td>b := b * a;</td>
</tr>
<tr>
<td></td>
<td>for a := 8 downto 1 do</td>
<td>for (a = 8; a &gt; 0; a = a - 1)</td>
</tr>
<tr>
<td></td>
<td>b := b * a;</td>
<td>b := b * a;</td>
</tr>
</tbody>
</table>

Progresjon er et vel så godt ord som inkrementering for det som gjøres i kontrollsetningens tredje ledd i Cs for-løkke. I Pascal's for-løkke ligger det i ordet to/downto mellom angivelsen av start- og sluttpunktet for iterasjonen, at den forutsetningsvis integrale kontrollvariabelen inkrementeres/dekrementeres med 1.

Cs for-løkke gir betydelig større frihet både mht. initialisering, test og inkrementering.

- Kontrollvariabelen behøver ikke være integral (max er et tall):
  
  for (float f = 0; f < max; f = f + .1)...;


  long stop = time(NULL) + period;
  StartSound(sound);
  for (long t = time(NULL); t < stop; t = time(NULL));
  StopSound(sound);
• Første led kan initialisere mer enn én variabel — f.eks. slik:
  
  ```
  StartSound(sound);
  for (long t = time(NULL), stop = t + period; t < stop; t = time(NULL));
  StopSound(sound);
  ```

• Det kan være mer enn én kontrollvariabel. Anta vi skal gå inntil et visst antall skritt = maxSteps eller en viss lengde = maxLength, og at skrittlengden = steplength.
  
  ```
  for (int step = 0; lng = 0;
       step < maxSteps && lng < maxLength;
       step = step + 1, lng = lng + Length)
  cout << "Skritt: " << step << "  Lengde: " << lng << endl;
  ```

• Progresjonen kan være implisitt i utførelsen av testen. Anta vi skal lese en streng tegnvis, til det er slutt på inndata eller på linja, eller det ikke er plass til flere tegn i strengen, og at antall plasser i strengen, terminasjonstegnet medregnet, er maxStr.
  
  ```
  for (int i = 0; c = getchar() != EOF && c != \n && i < maxStr;
       i = i + 1)
  s[i] = c;
  ```

• Alt kan utføres i kontrollsetningen, slik at løkkens innhold kan sløyfes.
  
  ```
  for (int i = 0; c = getchar() != EOF && i < maxStr; s[i] = c, i = i + 1)
  ;
  ```

• Også testen såvel som inkrementeringen kan sløyfes. Anta vi skal lese forbi alle tegn frem til et gitt merke.
  
  ```
  for ( ; c = getchar() != merke && c != EOF; )
  ;
  ```

• Faktisk kan i prinsippet alle ledd i kontrollsetningen sløyfes — noe som gir en nokså triviell evig løkke.
  
  ```
  for ( ; ; )
  ...;
  ```

Inkrementering, dekrementering eller annen trinnvis endring i kontrollvariabelens verdi, er den mest valige formen for progresjon i en for-løkke, og for dette formålet er det laget egne operatorer som vi skal komme tilbake til i neste kapittel. Kort fortalt gjør disse det mulig å erstatte setninger som

  ```
  i = i + 1;
  lng = lng + step;
  ```

med

  ```
  i++;
  lng += step;
  ```

for-løkken egentlig bare er en omskrivning av den mer generelle while-løkken, hvis vi definerer initialiseringen og progresjonen som en del av løkke-begrepet.

Følgende to programbiter er både praktisk og formelt ekvivalent:

```
init;
while (test) for (init; test; progr)
{ sentence;
  progr;
}
```
Den praktiske ekvivalensen innebærer bl.a. at prosessen utføres etter testen og innholdet i løkken.

Den formelle ekvivalensen innebærer bl.a. at vi kan deklarere de aktuelle variablene i selve for-løkken — f.eks. slik:

```c
for (int max = 10, i = 0; i < max; i++) ...
```

C-varianten av while-løkken og de to for-løkkene på forrige side kunne ha vært skrevet slik:

```c
while (a)
{
    b *= a;
    a--;
}
for (a = 1; a <= 8; a++)
    b *= a;
for (a = 8; a--; a--)
    b *= a;
```

Når vi kan erstatte \(a > 0\) med \(a\), så er det, som vi har sett, fordi \(0\) qua boolean er false.

Operatorene ++, -- og *= er forklart i neste avsnitt.

### 10.5.3 do-løkken

Mens testen i while- og for-løkkene utføres for hvert gjennomløp, utføres de etter hvert gjennomløp i følgende konstruksjoner:

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>repeat</strong></td>
<td>do</td>
<td><strong>do</strong></td>
</tr>
<tr>
<td><strong>effect</strong>;</td>
<td><strong>{ effect</strong>;</td>
<td><strong>effect</strong>;</td>
</tr>
<tr>
<td><strong>b := b * a</strong>;</td>
<td><strong>b = b * a</strong>;</td>
<td><strong>b = b * a</strong>;</td>
</tr>
<tr>
<td><strong>a := a - 1</strong></td>
<td><strong>a = a - 1</strong>;</td>
<td><strong>a = a - 1</strong>;</td>
</tr>
<tr>
<td><strong>until a &lt;= 0</strong>;</td>
<td><strong>while (a &gt; 0)</strong></td>
<td><strong>while (a &gt; 0)</strong></td>
</tr>
</tbody>
</table>

Legg merke til at testen i Pascal's repeat-setning gjelder negasjonen av basistilfellet — så lenge basistilfellet ikke har inntruffet.

### 10.6 Hopp-instruksjoner

Etter at det i løpet av 60-årene ble utviklet en rekke språk med syntaktiske mekanismer for valg og løkker, utviklet det seg nærmest et tabu mot bruk av eksplicitte hopp-instruksjoner (som alle FORTRAN-programmer var fulle av), selv om alle disse språkene også hadde syntaktiske mekanismer for dette.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
<th>Betydning</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/cc*</td>
<td>goto label;</td>
<td>goto label;</td>
<td>hopp til label (angitt med kolon etter)</td>
</tr>
<tr>
<td></td>
<td>cycle;</td>
<td>continue;</td>
<td>fortsatt fra toppen av løkken</td>
</tr>
<tr>
<td></td>
<td>leave;</td>
<td>break;</td>
<td>hopp ut av løkken / switch-en</td>
</tr>
<tr>
<td>call/cc</td>
<td>—</td>
<td>return expr;</td>
<td>returner verdien til uttrykket</td>
</tr>
<tr>
<td>call/cc</td>
<td>—</td>
<td>exit(prosedyre)</td>
<td>return; rhopp ut av funksjonen</td>
</tr>
</tbody>
</table>

I C brukes imidlertid hopp-instruksjoner regelmessig i switch-setninger og ved terminering av funksjonskall (selv om hopp-instruksjonen oftest er siste setning i funksjonen). Forøvrig er det heller ikke uvanlig at man under marginal betingelse hopper ut av en løkke eller til løkkens første setning — kanskje primært for å selve løkke-testen skal vise det normale gjennomløpet i løkka.

* `call/cc` er et alias for Scheme-funksjonen `call-with-current-continuation` som kort fortalt gør det mulig å hoppe ut av prosedyrer — men da med større eleganse og sikkerhet, hva ivaretagelsen av data og prosessinformasjon angår, enn det en enkel `goto` ofte vil gi.
11 Operatorer

11.1 Operatoroversikt

Her gis en kortfattet oversikt over operatorene i C og C++. — Detaljene gis i de følgende avsnittene.

11.1.1 Cs utgaver av de vanligste operatorene i prosedyrale språk

<table>
<thead>
<tr>
<th>Aritmetikk</th>
<th>Logikk</th>
<th>Relasjon</th>
<th>Likhet</th>
<th>Tilordning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (plus)</td>
<td>&amp; &amp; (og)</td>
<td>&lt; (mindre enn)</td>
<td>== (lik)</td>
<td>= (sett lik!)</td>
</tr>
<tr>
<td>- (minus)</td>
<td></td>
<td></td>
<td>(eller)</td>
<td>&lt;= (m.e. el.lik)</td>
</tr>
<tr>
<td>* (ganger)</td>
<td>! (ikke)</td>
<td>&gt; (større enn)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ (delt på)</td>
<td></td>
<td>&gt;= (s.e. el. lik)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>% (rest)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11.1.2 Spesielle operatorer i C (i forhold til andre prosedyrale språk)

- bitvis & (AND — et), | (OR — vel), ^ (XOR — aut),
- << (forskyv mot venstre), >> (forskyv mot høyre), ~ (ener-komplement)
- inkrementtering prefix ++, prefix --, postfix ++, postfix --
- tilordning *=, /=, %=, +=, -=, <<=, >>=, &=>, ^=, |=
- rekkfølge ,
- betingelse test ? konsekvens : alternativ
- størrelse sizeof(variabel) sizeof(type)
- typeomtolkning (type)uttrykk
- dereferanse *uttrykk
- adresse &variabel
- felttakssess struct_variabel.felt
- dynamisk felttakssess struct_peker->felt ≡ (*struct_peker).felt

(De to ekvivalente uttrykkene leses begge slik: Aksesser felt i det struct_peker peker på.)

11.1.3 Spesielle operatorer i C++

- tilhørighet, rekkevidde ::
- skriv fra uttrykk til fil <<
- les fra fil til variabel >>
- felttakssess objekt_variabel.felt
- dynamisk felttakssess objekt_peker->felt ≡ (*objekt_peker).felt
- allokert dynamisk new type eller new type(arg)
- slett dynamisk allokert delete variabel
- allokert vektor dyn. new type[maxant]
- slett dynamisk vektor delete vektor[]

43

11.2 Tilordningsoperatoren

I C brukes = for vanlig tilordning.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(set! a 5)</td>
<td>a := 5;</td>
<td>a = 5;</td>
</tr>
</tbody>
</table>

I tillegg kan man i C koble tilordningsoperatoren til de aritmetiske operatorene (også de bitvise) slik at

a operator b = a = a operator b

f.eks: a += 5 = a = a + 5
       a *= 2 = a = a * 2

Se under.

11.3 De logiske operatorene

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
<th>Symbol</th>
<th>Betydning</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>not</td>
<td>!</td>
<td>¬</td>
<td>ikke</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>&amp; &amp;</td>
<td>∧</td>
<td>og</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td></td>
<td></td>
<td>eller</td>
</tr>
</tbody>
</table>

I evalueringen av a & & b evalueres først a. Gir dette 0, returneres 0, og b evalueres ikke. Evaluerer a til 1, evalueres b, og resultatet av dette returneres.

I evalueringen av a || b evalueres først a. Gir dette 1, returneres 1, og b evalueres ikke. Evaluerer a til 0, evalueres b, og resultat av dette returneres.

11.4 Likhetsoperatoren og de relasjonelle operatorene

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
<th>Symbol</th>
<th>Betydning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>=</td>
<td>=</td>
<td>=</td>
<td>lik</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>!=</td>
<td>≠</td>
<td>ulik / forskjellig fra</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>mindre enn</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;=</td>
<td>≤</td>
<td>mindre enn eller lik</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>større enn</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&gt;=</td>
<td>≥</td>
<td>større enn eller lik</td>
<td></td>
</tr>
</tbody>
</table>

I C, som i Scheme, kan sammenligningsoperatorene i utgangspunktet bare brukes for tall (inklusive tegn).

Mens man i Pascal også kan bruke disse for strenger, må man i C bruke standardfunksjoner eller egendefinerte funksjoner for å sammenligne strenger.

I C++ kan man i en klasse utvide domeninet til en operator til å omfatte instanser av klassen.
11.5 De aritmetiske operatorene

<table>
<thead>
<tr>
<th>Scheme*</th>
<th>Pascal</th>
<th>C</th>
<th>Symbol</th>
<th>Betydning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>adder</td>
</tr>
<tr>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>subtraher</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>×</td>
<td>multipliser</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>÷</td>
<td>divider</td>
<td></td>
</tr>
<tr>
<td>quotient</td>
<td>div</td>
<td>/</td>
<td>divider hele tall</td>
<td></td>
</tr>
<tr>
<td>remainder</td>
<td>mod</td>
<td>%</td>
<td>beregn rest etter divisjon av hele tall</td>
<td></td>
</tr>
</tbody>
</table>

I Scheme og Pascal finnes det egne operatorer for heltallsdivisjon (hhv. quotient og div). I C vil divisjon gi et flyttall hvis én av eller begge operandene er flyttall. Men hvis begge operandene er heltall blir resultatet alltid et heltall — og dermed tregner vi ikke noen egen operator for heltallsdivisjon. Ønsker vi at divisjonen av ett heltall med et annet skal evaluere til et flyttall, kan vi omtolke typen til dividend eller divisor (eller for den saks skyld begge) f.eks. slik:

\[(\texttt{float}) i / j\] eller slik:

\[i / (\texttt{float}) j\]

*I Scheme finnes det strengt sett bare funksjoner. Det finnes standardfunksjoner i C tilsvarende de fleste aritmetiske funksjonene i Scheme.

11.6 De bitvise operatorene — booleske vektorer, mengder og flagg

En lottokupong med 32 plasser kan realiseres bl.a. som en mengde — som på sin side kan realiseres som en boolesk vektor. I C kan vi representere dette med et 32-bits, 4-bytes, heltall der hvert bit representerer et tall i mengden / en plass i vektoren. Er f.eks. bit 5, 7, 13, 18, 25, 28 og 30 satt, har vi mengden \(\{6, 8, 14, 19, 26, 29, 31\}\) (bitspennet er \(0 .. 31\), mens mengdens domene er \(1 .. 32\)).

Dette gir tallet \(01010010 \ 00000100 \ 00100000 \ 10100000_\ll{i six} = 1376002208\) ti.

NB! En mengde / boolesk vektor kan ikke ha flere bits enn det vi får plass til i en long int — dvs. 32 bits.

For å finne ut om et gitt bit er satt, bruker vi operatoren \& for bitvis AND.

For å sette bits, bruker vi operatoren | for bitvis OR.

<table>
<thead>
<tr>
<th>binært</th>
<th>heksadesimalt</th>
<th>desimalt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000111 &amp; 00000100 =&gt; 00000100</td>
<td>0x0F &amp; 0x04 =&gt; 0x04</td>
<td>15 &amp; 4 =&gt; 4</td>
</tr>
<tr>
<td>00000111 &amp; 01000000 =&gt; 00000000</td>
<td>0x0F &amp; 0x40 =&gt; 0x00</td>
<td>15 &amp; 64 =&gt; 0</td>
</tr>
<tr>
<td>00000001</td>
<td>0x01</td>
<td>0x04</td>
</tr>
<tr>
<td>00100000</td>
<td>01000000 =&gt; 01010101</td>
<td>0x10</td>
</tr>
<tr>
<td>00001101 &amp; 01010101 =&gt; 00000101</td>
<td>0x0F &amp; 0x55 =&gt; 0x05</td>
<td>15 &amp; 85 =&gt; 5</td>
</tr>
</tbody>
</table>

Vi kan sette navn på de tallene der de enkelt bit er satt (her har vi nøyd oss med de første 8):

```c
enum bits {BIT0=1,BIT1=2,BIT2=4,BIT3=8,BIT4=16,BIT5=32,BIT6=64,BIT7=128};
```

Gitt en mengde \(m\) representert ved et byte, vil følgende teste om bit 0, 3, 5 eller 7 er satt:

```c
if (m & (BIT0 | BIT3 | BIT5 | BIT7)) ...;
```

Når vi tester for bits på denne måten, kaller vi ofte det tallet som gir bit-et for et flagg.

Et flagg vil alltid være en potens av 2, der potensen gir bit-nummeret \((2^x = 1)\).

Bit-ene nummereres tradisjonelt fra 0. 0-te bit er bit-et lengst til høyre (dette kalles LSB for Least Significant Bit — i motsetning til MSB, for Most Significant Bit, som er bit \(n - 1\) i en rekke med \(n\) bits).
Det finnes flere bitvise operatorer enn disse, men i SLI 280 bruker vi bare bitvis OR og AND — for hhv. å sette opp og sjekke flagg.

De samlede bitvise operatorer er

Bitvis AND

Bitvis OR (vel, dvs. inkluderende)

Bitvis XOR (nutt, dvs. ekskluderende)

Bitvis forskyvning mot venstre

Bitvis forskyvning mot høyre

ener-komplement (unær)

Alle, bortsett fra en's komplement er binære (tar to operander).

Bitvis AND setter de bit der begge opperandenes bit har verdien 1, og nuller de øvrige.

Bitvis OR setter de bit der ene eller den andre av opperandenes bit har verdien 1, og nuller de øvrige.

Bitvis XOR setter de bit der opperandenes bit har motsatte verdier (0 og 1 eller 1 og 0) og nuller de bit der verdiene er like (0 og 0 eller 1 og 1).

Bitvis forskyvning (shift) forskyer bitene i første operand så mange plasse mot venstre eller høyre som andre operand tilsier, og etterfyller med 0-er.

Eksempler på tap av bits ved forskyvning:

01010000 << 2 → 01000000; 0x50 << 2 → 0x0A
01010000 >> 3 → 00001010; 0x50 >> 3 → 0x0A

Legg merke til at dette er det samme som hhv. å multiplisere og dividere med $2^3$; og bortsett fra når bits skyves ut av den første operanden, har vi $n << b = n \times 2^b$ og $n >> b = n / 2^b$.

Eksempler på tap av bits ved forskyvning:

01010000, << 3 → 01000000; 0x0A, << 3 → 0x50
01010000, >> 3 → 00001010; 0x50, >> 3 → 0x0A

Énner-komplement av et tall n er det tallet der alle bits i n er snudd — f.eks. slik:

$\sim 01010101 → 10101010; \sim 0x55 → 0xAA$; $\sim 85 → 170$;

11.7 Inkrementerings og dekrementeringsoperatorene

Det å inkrementere og dekrementere, dvs. hhv. øke og redusere verdien til variabler med 1, er svært vanlig i sekvensiell programmering (se Setninger over)— faktisk så vanlig at man i C har innført egne operatorer for dette:

```c
++a
a++
--a
a--
```

Disse er unære, idet de tar én operand — i motsetning til + og - som tar to operander og dermed er binære.

Står ++ eller -- foran variabelen, endres variabelen før vi får dens verdi.

Står ++ eller -- etter variabelen, endres variabelen etter at vi får dens verdi.

Følgende program

```c
void main()
{
    int a = 5;
    cout << "a er " << a;
    cout << ", ++a er " << ++a;
    cout << ", og a++ er " << a++;  
    cout << "men etter a++ er a " << a << endl;
}
```

Resultatet er a er 5, ++a er 6 og a++ er 6, men etter a++ er a 7.
Erstatter vi de 4 utskriftsetningene med én — slik

```cpp
cout << " a er "               << a
<< ", ++a er "           << ++a
<< " og a++ er "          << a++
<< " men etter a++ er a " << a << endl;
```

Får vi resultatet

\[ a \text{ er } 7, \quad ++a \text{ er } 7 \quad \text{og} \quad a++ \text{ er } 5, \quad \text{men etter } a++ \text{ er } a \quad 7 \]

Grunnen er at alle utskriftsleddene evalueres før hele linjen skrives ut — etter følgende regler (slid det ser ut):

1: evaluer de ledd som må regnes ut først, og
2: evaluerer forøyvfr fra venstre mot høyre.

Først evalueres \(+a+\) til 5, men evalueringssessessen i seg selv fører til at a får verdien 6.

Deretter evalueres \(+a+\) til 7 samtittig som evalueringssessessen i seg selv gir a verdien 7.

Siste og første a evalueres deretter til seg selv, dvs. 7.

### 11.8 Tilordningsoperatoren koblet med de aritmetiske operatorene

C har også en mer generell kompakt notasjon for endring av integrale variabler verdier. K&R kategoriserer dette under *tilordning* (*assignment*). Poenget er at vi kobler en aritmetisk operator til tilordningsoperatoren. De aritmetiske operatorene omfatter i denne sammenhengen også de bitvise.

\( a \text{ operator=} \langle uttrykk \rangle \) gjør det samme som \( a = a \text{ operator} \langle uttrykk \rangle \).

Eksempler

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(set! a (+ a 2))</td>
<td>a := a + 2; a += 2; a &lt;&lt;= 2;</td>
<td></td>
</tr>
<tr>
<td>(set! a (- a 2))</td>
<td>a := a - 2; a -= 2; a &gt;&gt;= 2;</td>
<td></td>
</tr>
<tr>
<td>(set! a (* a 2))</td>
<td>a := a * 2; a *= 2; a &amp;= 2;</td>
<td></td>
</tr>
<tr>
<td>(set! a (/ a 2))</td>
<td>a := a / 2; a /= 2; a ^= 2;</td>
<td></td>
</tr>
<tr>
<td>(set! a (quotient a 2))</td>
<td>a := a div 2; a /= 2; a</td>
<td>== 2;</td>
</tr>
<tr>
<td>(set! a (remainder a 2))</td>
<td>a := a mod 2; a %= 2;</td>
<td></td>
</tr>
</tbody>
</table>

\* I C (som i Scheme) utføres divisjon slik at hvis begge operandene er heltall, blir resultatet også et heltall, mens i alle andre tilfeller blir resultatet et flyttall —

f.eks. slik at

\[ 5 \div 2 \quad \rightarrow \quad 2, \]

må

\[ 5.0 \div 2.0 \quad \rightarrow \quad 2.5, \]

\[ 5.0 \div 2.0 \quad \rightarrow \quad 2.5. \]

### 11.9 Sekvensoperatoren

Operatoren , (av K&R kalt *comma operator*) brukes mellom uttrykk som skal evalueres i en gitt rekkefølge, som om de var imperativer skilt med semikolon, men fortrinnsvis i en syntaktisk sammenheng der vi ikke kan bruke semikolon. Uttrykkene evalueres fra venstre mot høyre, og resultatet av den siste evalueringen returneres.

Operatoren må ikke forveksles med komma brukt ved oppramsing — som skille mellom argumentene i en argumentliste, mellom variablene i en deklarasjon eller mellom elementene eller feltene i initialiseringen av en vektor eller et struct.
Sekvensoperatoren kan f.eks. brukes slik

\[ f(a, (b=2, b+3), c) \]

Funksjonen \( f \) får her 3 aktuelle parametre hvorav det andre har verdien 5.

\[ \text{if} ((a *= b, b)) \ldots; \]

Sekvens-uttrykket evaluerer til \text{false} (0) eller \text{true} (1) avhengig av om b evaluerer til 0 eller noe annet.

Merk at vi i begge disse tilfellene må sette sekvensen i parentes, slik at komma ikke kan forveksles med oppramsingskomma, slik det bl.a. brukes mellom argumentene i en argumentliste (hva \text{if} angår, kunne vi litt upresist si at \text{if} bare skal ha ett argument).

Mest interessant er sekvensoperatoren i \text{for}-løkker,

Eks: Vi skal sjekke om en streng er et palindrom, dvs. om den er samme streng forlengs som baklengs.

Vi lar to indekser løpe fra hver sin ende av strengen inntil de eller møtes eller peker ut to ulike bokstaver.

\[
\text{char} \ s[256]; \ \text{int} \ i, j; \\
cout \ll "Skriv en streng: \"; \\
cin.getline(s, 256); \\
\text{for} \ (i = 0, j = strlen(s) - 1; // løkkeinitiering \\
i < j && s[i] == s[j]; \quad \text{// løkke-test} \\
i++, --j) \quad \text{// løkke-inkrementering e.l.} \\
; \quad \text{// tom løkke-kropp} \\
cout \ll \ll (i >= j \ ? \ " er " : \ " er ikke ") \ll \ll \ "et palindrom.\" \ll \ll \ \text{endl};
\]

Initierings- og inkrementeringssetningen er begge sekvenser, mens testen ikke er det. Hadde vi skrevet testen som sekvensen \( i < j, s[i] == s[j] \), ville den for et palindrom evaluert til \text{true} også for \( i = j \), inntil de to indeksene hadde havnet utenfor s på motsatt side av der de startet. Poenget er at, uansett hva første ledd evaluerer til, evaluerer hele sekvensen til siste ledd.

Eks: Vi skal lese en serie tall fra tastaturet, summere dem, og til slutt skrive ut summen og snittet.

\[
\text{cout} \ll \ll \ "Skriv en serie tall og avslutt med \ -1: \"; \\
\text{for} \ (\text{int} \ \text{tall}, \ \text{sum} = 0, \ \text{ant} = 0; // løkkeinitiering \\
\text{cin} \gg \ \text{tall}, \ \text{tall} >= 0; \quad \text{// løkke-test} \\
\text{sum} += \ \text{tall}, \ \text{ant}++) \quad \text{// løkke-inkrementering e.l.} \\
; \quad \text{// tom løkke-kropp} \\
cout \ll \ll \ "\text{Sum: } \ll \ll \ \text{sum} \ll \ll \ ". \ \text{Snitt: } \ll \ll (\text{float})\ \text{sum} \ / \ \text{ant} \ll \ll \ \text{endl};
\]

Her bruker vi sekvenser både i initieringen, testen og inkrementeringen. Innlesingen bidrar ikke til resultatet av testen, men ved å legge innlesingen i testen i stedet for i iterasjonen, sørger vi for at den blir utført på toppen i stedet for i bunnen av løkken. Returen fra \text{cin} er uten interesse her, og uavhengig av den, evaluerer sekvensen til sitt siste ledd.

11.10 Operatorers prioritet

Med et uttrykk i et C program forstår vi et funksjonskall, en operator og dens operand(er) eller et enkeltstående tall, variabelnavn e.l.. I Scheme er uttrykk alltid funksjonskall avgrenset ved parenteser, mens alt annet er atomer. I C har vi de uttrykk som er gitt i tabellen under, og blant disse finner vi de primære uttrykk som tilsvarer atomene i Scheme. Siden de enkelte operatorer ikke krever parenteser, må vi kjenne reglene for hvordan de grupperer seg uten parenteser, og sette inn parenteser der vi ønsker en annen gruppering.
Prioritetsordenen i C er som følger:

1. Automatisk genererte pekere til vektorer og funksjoner
2. Primære uttrykk, dvs. navn og konstanter (herunder strengkonstanter)
   Uttrykk i parenteser (som evaluerer til primære uttrykk).
3. Postfix-uttrykk
   vektornavn med elementangivielse — v[i], w[i, j, ...]
   funksjonsnavn med parameterliste — f(a1, a2, ...)
   struct-navn med feltaksessor — s.f1
   struct-pekere med feltaksessor — ps->f1
   heltall med in-/decrement-operator — i++ j--
4. Uttr. med unære oper.
   prefix in-/decrement-operator — ++i --j
   adresse-operatoren — &a
dereferanse-operatoren — *p
   fortegn — +i -j
   ener-komplement — ~n
   negasjonsoperatoren — !b
   størrelseseoperatoren — sizeof(var) sizeof(type)
5. Typeomtolkningsuttr. — (type)uttrykk
6. Multiplikative uttrykk — a * b a / b a % b
7. Additive uttrykk — a + b a - b
8. Bitvise forskyvningsut.
   a << b a >> b
9. Relasjonelle uttrykk — a < b a <= b a >= b a > b
10. Likhetsuttrykk — a == b a != b
11. Bitvise AND-uttrykk — a & b
12. Bitvise OR-uttrykk — a | b
13. Bitvise XOR-uttrykk — a ^ b
14. Logiske AND-uttrykk — a && b
15. Logiske OR-uttrykk — a || b
17. Tilordningsuttrykk — a = b; a *= b; a /= b; a %= b; a+= b; a -= b;
    a <<= b; a >>= b; a &= b; a ^= b; a |= b
18. Sekvenser — if ((a *= b, b))...;

Vi kan lese dette slik at det som står foran noe på listen kan være argument / operand, uten parentes, til det som står etter — f.eks. slik at operander til + kan være multiplikative uttrykk, uttrykk med unære operatorer, postfix-uttrykk, parenteser, primære uttrykk og automatisk genererte pekere til vektorer og funksjoner, mens additive uttrykk, osv. ikke kan være operander til multiplikative uttrykk.

Dette kan illustreres vha. parenteser slik at hvis o1 og o2 er operatorer, og o3 har høyest prioritet, så kan

x o3 y o2 z o1 s o3 p, men ikke til x o1 (y o3 z).
11.11 Gruppering av enkeltuttrykk i sammensatte uttrykk

I tillegg til operatorenes prioritetsorden, må vi være oppmerksomme på hvordan enkeltuttrykk med operatorer som har samme prioritet, grupperer seg innenfor et sammensatt uttrykk.

Generelt grupperer slike uttrykk seg fra venstre mot høyre.

\[
\begin{array}{ll}
\text{slik} & (2 + (5 - (2 + 3))) \\
2 + 5 - 2 + 3 & \Rightarrow 8 & (2 + (5 - (2 + 3))) & \Rightarrow 2 \\
2 \times 5 / 2 \% 3 & \Rightarrow 2 & (2 \times (5 / (2 \% 3))) & \Rightarrow 4 \\
\end{array}
\]

Generelt grupperer slike uttrykk seg fra venstre mot høyre.
12 Aggregater

Et aggregat er en samling av data under ett navn. Det kan dreie seg om vektorer, dvs. en serie med elementer av samme type eller struct-er en gruppering av elementer av ulike typer.

12.1 Vektorer

I C deklarerer man vektorer og matriser ved å angi antall elementer i skarpe klammer etter variabelnavnet.

### Scheme

```scheme
(let ((v    v: array[10..19]of integer;     int v[10];  
      (make-vektor 10))...)
```

### Pascal

```pascal
var  
  v: array[10] of integer;  
  v[1] := 17;  
```

### C

```c
int v[10];  
int v[1] = 17;
```

**NB! I C som i Scheme har alltid første vektorelement indeks 0, og det er opp til programmereren å foreta omregninger fra en reell indeksverdi til en vektorindeks, der det måtte være nødvendig.**

Bruk av 0 som startindeks er ofte hensiktmessig, i det indeksen på den måten alltid gir oss avstanden (offset) fra begynnelsen av vektoren til det indekserte elementet. Dette forenkler bl.a. omregning fra absolutt til relativt offset i en flerdimensjonal vektor.

En vektor i en argumentliste kan angis med en tom skarp klamme etter argumentnavnet.

```c
void ZeroAll(int v[], int n) {  
  int i = 0;  
  while(i < n)  
    v[i] = 0;  
    i = i + 1;  
}
```

*Square Brackets* oversettes gjerne med *halve parenteser* eller skarpe klammer. Siden vi i programmeringssammenheng bruker fire ulike parenteser eller klammer — (), [], {} og <>, trenger vi en terminologi som skiller disse fra hverandre. Mulig navn er *parentes*, *vandl parentes*, *firkantparentes*, *klammer* og *vinkelparentes*.

Deklarasjonen av en vektor kan, som en hvilken som helst annen deklarasjon, inneholde et initialiseringsledd. For vektorer er dette en klamme med de initielle elementverdiene, skilt fra hverandre med komma:

```c
int v[8] = {1, 2, 3, 4, 5, 6, 7, 8};
```

Er initialiseringsleddet med, kan vi utelate angivelsen av vektorens lengde, og den får da automatisk den lengden initialiseringen krever.

```c
int v[] = {1, 2, 3, 4, 5, 6, 7, 8};  
```

Tar vi med lengdeangivelsen, må denne gi plass for en eventuell initiell verdi, dvs. vi kan angi flere, men ikke færre plasser enn det initialiseringen krever.

Lovlig:

```c
int v[12] = {1, 2, 3, 4, 5, 6, 7, 8};
```

Ulovlig:

```c
int v[4] = {1, 2, 3, 4, 5, 6, 7, 8};
```

For strenger, som er vektorer med tegn, bruker vi gjerne syntaksen for strengkonstanter ved initialisering:

```c
char s1[] = "Hallo", s2[4] = "Hei";
```

Her legges strengterminatoren ’\0’ automatisk inn sist i tegnvektoren.

Bruker vi klamme, må vi selv sørge for å få med strengterminatoren:

```c
char s[] = {'H', 'e', 'i', '\0'};
```

Det er et spesielt forhold mellom vektorer og pekere i C som vi skal komme tilbake til. I korthet går det ut på at navnet på en vektorvariabel evaluerer til en peker, dvs. til adressen til første element i vekten.
Dette har bl.a. følgende konsekvenser
- det gir oss alternative måter å referere til vektorelementer på, og
- vektorer qua parametre overføres alltid i praksis ved referanse.
Det første innebærer bl.a. at uttrykket type var[] er ekvivalent med uttrykket type * var. En peker angis i en deklarasjon ved en åfkmek. F.eks. int * pi; betyr at pi deklareres som en peker til en int. En vektor i en argumentliste kan dermed alternativt angis med en stjerne foran navnet. — Se forøvrig Pekere og Vektorer.
Det siste innebærer at alle de endringer vi måtte utføre i en funksjon på elementene i en parameteroverført vektor, vil være synlig i de omgivelsene der funksjonen kalles. Dette er viktig å merke seg, ettersom det ikke gir oss noe valg, samtidig som det skiller seg fra hva som gjelder for andre prosedyrale språk som Pascal (på den andre siden tilsvarer det den praksis som benyttes i Scheme). —

12.2 Struct-er
I C, som i Pascal, har man syntaks for å sette sammen flere datatyper til én:

**Pascal**

```
pascal
record
  integer a, b;
  char c, d;
end;
r: record
```

**C**

```
c
struct
  int a, b, c;
  unsigned char c, d;

r, s: TRec;
```

I begge språk kan man også gå via en typedefinisjon — noe som er en forutsetning, når man opererer med flere variabler av samme sammensatte type:

**Pascal**

```
pascal
type TRec = record
  integer a, b;
  char c, d;
end;
```

**C**

```
c
struct TStr
  int a, b;
  unsigned char c, d;

```

Den definerende type er her klammen [...]. Denne definerer navnet tstr. Vi kan også ha eksplisitte typedefinisjoner som innledes med det reservede ordet `typedef`. — Se Basistyper og egendefinerte typer.

NB! Merk at vi tar med semikolon etter `struct`-definisjonen — også når denne ender med `}`. I Pascal skiller det mellom definisjoner av konstanter og typer og deklarasjoner av variabler. En eller flere definisjoner eller deklarasjoner innledes alltid med ett av de reservede ordene `const` (for konstanter), `type` (for typer) eller `var` (for variabler).

I C kan vi velge å angi en såkalte structure tag (som `tstr` over), om vi ønsker å bruke definisjonen for mer enn én variabel, og vi kan velge om vi vil deklarere én eller flere variabler sammen med `struct`-definisjonen.

Når vi skal snakke om de enkelte navngitte elementene i et struct, bruker vi, både i Pascal og C, såkalt prikk- eller dot-aksess — f.eks. slik

**Pascal**

```
pascal
r.b := s.a + 2;
if r.a < 0 then r.c = ‘?’;
```

**C**

```
c
r.b = s.a + 2;
if (r.a < 0) r.c = ‘?’;
```

I Pascal-litteraturen kalles gjerne et element i en record for et felt (field), mens man i C-litteraturen kaller et `struct`-element for et member (elem).

Vi skal bruke alle disse betegnelsene, inklusive element, men foretrekker felt på norsk. — Og mens K&R kaller prikken (dot-en) mellom struct-variabelnavnet og feltnavnet `the member operator`, skal vi kalle denne feltoperatoren eller feltaksesoren.

Eksempel: En brøk (fraction) består av en teller og en nevner (numerator og denominator):

```
c
struct fraction {int numerator, denominator};
```
```c
struct fraction MakeFrac(int n, int d)
{
    struct fraction tmp;
    tmp.numerator = n;
    tmp.denominator = d;
    return tmp;
}

struct fraction AddFrac(struct broek b1, struct broek b2)
{
    struct fraction res;
    res.numerator = b1.numerator * b2.denominator
            + b2.numerator * b1.denominator ;
    res.denominator = b1.numerator * b2.denominator;
    return res;
}

struct fraction MulFrac(struct broek b1, struct broek b2)
{
    struct fraction res;
    res.numerator = b1.numerator * b2.numerator ;
    res.denominator = b1.denominator * b2.denominator;
    return res;
}

void WriteFrac(struct fraction b)
{
    cout << b.numerator << "/" << b.denominator;
}

int main(void)
{
    struct fraction a, b;
    a = MakeFrac(1, 3);
    b = MakeFrac(2, 7);
    WriteFrac(a);   cout << "+";
    WriteFrac(b);   cout << "=";
    WriteFrac(AddFrac(a,b)); cout << endl;
    WriteFrac(a);   cout << "+";
    WriteFrac(b);   cout << "=";
    WriteFrac(MulFrac(a,b)); cout << endl;
}
```

Kjøring av dette programmet gir

\[ \frac{2}{3} + \frac{5}{7} = \frac{29}{21} \]
\[ \frac{2}{3} \times \frac{5}{7} = \frac{10}{21} \]
12.3 Unioner

Med en union kan vi lagre data i en variabel som kan opptre med ulike typer etter behov.

Unioner har samme syntaks som struct-er.

```c
union uni { int i; double d; char c; } u;
```

En union inneholder imidlertid bare ett felt, og tar så mye plass som det største feltet — her d som tar 8 bytes.

Programmet

```c
void main()
{
    union { long l; char s[4]; } u;
    strcpy(u.s, "HEI");
    cout << u.s << " " << u.l << endl;
}
```

Gir utskriften

```
HEI  4801864
```
13 Egen definerte typer

Vi har alt nevnt de predefinerte typene i C (se Predefinerte typer i C).

Vi kan definere nye typer slik:

```c
typedef unsigned char uc;
typedef struct { int x, y; } point;
```

Med definisjonen `typedef struct tpt { int x, y; } point;` får vi faktisk to navn for samme type, `tpt` og `point`, som kan brukes om hverandre.

Typedefinisjoner brukes til
- å unngå lange navn:
  ```c
  uc i stedet for unsigned char
  point i stedet for struct point
  ```
- definisjoner som kan variere fra ett system til et annet f.eks:
  ```c
  typedef unsigned int size_t;
  ```

Typen `size_t` er definert i `stdlib.h` og brukes som type for en størrelse (pragmatikken her er at en `int` skal være tilstrekkelig til å romme størrelsen på en hukommelsesstruktur i antall bytes).
14 Typekonvertering

14.1 Implisitt typekonvertering

C har (som de fleste andre språk) automatisk konvertering ved tilordning:

```c
unsigned int  u = 0;
int           i = u;
float         f = i;
unsigned char u = f;
```

Dette forutsetter at den finnes en fornuftig konverteringsalgoritme, for eksempel:

- float → int benytter en egen omregningsfunksjon.
- unsigned char → unsigned short og
- unsigned short → unsigned long fyller inn 0-bit
- signed char → signed short og
- signed short → signed long fyller inn 0-bit for positive og 1-bit for negative
- unsigned short → unsigned char, signed long → unsigned signed char, og
- unsigned long → unsigned signed char fjerner overflødige bit.
- signed xxx → unsigned xxx endrer ingenting.

14.2 Eksplicit typekonvertering (type casting / type coercion)

Noen ganger må vi angi en eksplitt typekonvertering. Dette gjør vi ved å sette typebetegnelsen i parantes foran det uttrykket (konstantverdi, variabelen, utregningen eller funksjonskallet) som skal konverteres: `(type)uttrykk`

Hvis uttrykket er sammensatt, setter vi en parentes rundt det også.

Følgende program runder av et beløp gitt som kroner med desimaler til nærmeste 50-øring:

```c
double les_kroner()
{
    double kr;
    cout << "Gi et beløp: "; cin >> kr;
    return kr;
}

double avrund(double kroner)
{
    return (int)(kroner * 2 + 0.5) / 2.0;
}

void main()
{
    double kroner = les_kroner();
    while (kroner >= 0.0)
    {
        cout << kroner << " avrundes til " << avrund(kroner) << endl;
        kroner = les_kroner();
    }
}
```
Eksempel på en kjøring: Bakenfortiggende utregninger

Gi et beløp: 2
2 avrundes til 2

Gi et beløp: 2.1
2.1 avrundes til 2

Gi et beløp: 2.24
2.24 avrundes til 2

Gi et beløp: 2.25
2.25 avrundes til 2.5

Gi et beløp: 2.45
2.45 avrundes til 2.5

Gi et beløp: 2.75
2.75 avrundes til 3

Gi et beløp: 2.00 + 0.5  \rightarrow  4.50 \hspace{1em} \text{int} \rightarrow 4 \hspace{1em} \frac{4}{2.0} \rightarrow 2.0

Gi et beløp: 2.10 + 0.5  \rightarrow  4.60 \hspace{1em} \text{int} \rightarrow 4 \hspace{1em} \frac{4}{2.0} \rightarrow 2.0

Gi et beløp: 2.24 + 0.5  \rightarrow  4.98 \hspace{1em} \text{int} \rightarrow 4 \hspace{1em} \frac{4}{2.0} \rightarrow 2.0

Gi et beløp: 2.25 + 0.5  \rightarrow  5.00 \hspace{1em} \text{int} \rightarrow 5 \hspace{1em} \frac{5}{2.0} \rightarrow 2.5

Gi et beløp: 2.49 + 0.5  \rightarrow  5.48 \hspace{1em} \text{int} \rightarrow 5 \hspace{1em} \frac{5}{2.0} \rightarrow 2.5

Gi et beløp: 2.75 + 0.5  \rightarrow  6.00 \hspace{1em} \text{int} \rightarrow 6 \hspace{1em} \frac{6}{2.0} \rightarrow 3.0

Gi et beløp: 2.1  \times 2.55
2.1 avrundes til 2

Gi et beløp: 2.24  \times 2.0
2.24 avrundes til 2

Gi et beløp: 2.25  \times 2.5
2.25 avrundes til 2.5

Gi et beløp: 2.45  \times 2.5
2.45 avrundes til 2.5

Gi et beløp: 2.75  \times 3.0
2.75 avrundes til 3

I Scheme og Pascal finnes det egne operatorer for heltallsdivisjon (hhv. quotien og div). I C vil divisjon gi et flyttall hvis én av eller begge operandene er flyttall. Men hvis begge operandene er heltall blir resultatet alltid et heltall — og dermed tregner vi ikke noen egen operator for heltallsdivisjon.

Programmet

void main()
{
    cout.flags(ios::fixed); // gir fast format for flyttall
    double a = 5.1 / 2; cout << a << endl; // flyttallsdivisjon
    int    b = 5 / 2;    cout << b << endl; // heltallsdivisjon
    double c = 5 / 2;    cout << c << endl; // implisitt konvertering fra heltallsuttrykk
                                            // /5/2, som evaluerer til 2, til flyttallet 2.00000
    int    d = 5.1 / 2; cout << d << endl; // implisitt konvertering fra flyttallsuttrykk
                                            // /5.1/2, som evaluerer til 2.55, til heltallet 2.
}

gir utskriften
2.55000
2
2.00000
2

* ios er en klasse over bl.a. ostream (klassen som cout er en instans av) i hierarkiet av I/O-klasser.
flags er en ios-funksjon som bl.a. kan brukes for å formattere utskrift.
fixed er en ios-dest, et såkalt flagg (se Flagg), som gir fast format (5 desimaler) for flyttall.

Ønsker vi at divisjonen av ett heltall med et annet skal evaluere til et flyttall, kan vi omtolke typen til dividend eller divisor (eller for den saksk skyld begge)

f.eks slik  \hspace{1em} \text{(float)} i / j
eller slik  \hspace{1em} i / \text{(float)} j
## 15 Pekere

### 15.1 Henvisninger til plasser i hukommelsen

Som nesten alle programmeringsspråk har C pekere, dvs. verdier som *henviser* til data, men som ikke selv *er* data. Har vi en variabel og en peker til denne, kan vi operere direkte på variabelens verdi ved å angi variabelens navn, eller vi kan operere indirekte på verdien via pekeren.

En peker i C angis i definisjoner og deklarasjoner med en *stjerne* * mellom typenavnet og pekernavnet.


<table>
<thead>
<tr>
<th>Scheme</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip ^integer;</td>
<td>int ip;</td>
<td></td>
</tr>
</tbody>
</table>

I Scheme snakkes det ikke eksplicit om pekere. I praksis vil imidlertid alle aktuelle parametre som ikke evaluerer til enkle verdier (tall, tegn eller booleske), evaluere til pekere.

Gitt deklarasjonene

```plaintext
int i;
int * ip;
```

kan vi sette *ip til å peke på *i via operatoren & slik:

```plaintext
ip = &i;
```

Når vi skal snakke om verdien til det heltallet pekeren peker på, bruker vi også stjernen.

Etter at siste setning er utført, vil *i ha verdi 0.*

Deklarasjonen av pekeren *ip kan leses slik slik: *Vi deklarerer det *ip peker på som en int.*

K&R kaller den unære operatoren * the *indirection* eller dereferencing operator.

Vi skal, i mangel av noe bedre, kalle *indirekte- eller dereferanseoperatoren, og* 

*adresseoperatoren.*

Datamaskinens hukommelsen (RAM) er en samling byte indeksert med tall som vi kaller *adresser.*

En peker er rett og slett en adresse i RAM, og operatoren & gir oss en slik adresse.

#### RAM

<table>
<thead>
<tr>
<th>Adresse</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>0x00000002</td>
<td></td>
</tr>
<tr>
<td>0x00000003</td>
<td></td>
</tr>
<tr>
<td>0x00000004</td>
<td></td>
</tr>
<tr>
<td>0x00000005</td>
<td></td>
</tr>
<tr>
<td>0x00000006</td>
<td></td>
</tr>
<tr>
<td>0x00000007</td>
<td></td>
</tr>
</tbody>
</table>

Adressene er her gitt i hexadesimale tall som har sifrene 0..9, A..F, og som i C skrives med 0x først. — Se forøvrig **Posisjonelle addressystemer.**


---

59
Forholdet mellom verdier og adresser illustreseres av følgende tre eksempler som alle gir samme resultat:

```c
int g = 1;    // global variabel
void main()
{
    int l = 2;    // lokal variabel
    cout << "g har adresse " << &g " og innhold " << g << end;
    cout << "l har adresse " << &l " og innhold " << l << end;
}
```

```c
int g = 1, *pg = &g;
void main()
{
    int l = 2, *pl = &l;
    cout << "g har adresse " << pg " og innhold " << g << end;
    cout << "l har adresse " << pl " og innhold " << l << end;
}
```

```c
int g = 1, *pg = &g;
void main()
{
    int l = 2, *pl = &l;
    cout << "g har adresse " << pg " og innhold " << *pg << end;
    cout << "l har adresse " << pl " og innhold " << *pl << end;
}
```

Resultatet av kjøringen vil selvfølgelig variere fra én maskin til en annen, ett mulig resultat er:

```
g har adresse 0x10001050 og innhold 1.
l har adresse 0x7fff2eec og innhold 2.
```

### 15.2 Aksess via pekere

Det er det samme om vi aksesserer en variabel direkte eller indirekte via pekere:

Programmet gir utskriften:

```c
int main(void)
{
    int v, *p = &v;
    v = 5;
    cout << "v er " << v << endl;        v er 5.
    ++v;
    cout << "v er " << v << endl;        v er 6.
    *p = 11;
    cout << "v er " << v << endl;        v er 11.
    ++*p;
    cout << "v er " << v << endl;        v er 12.
}
```
15.3 Pekere til struct-er

Ved hjelp av pekernotasjon, kan vi definere følgende funksjon i programeksemplet i avsnittet struct-er:

```c
void SetFrac(struct fraction * b, int n, int d)
{
    (*b).numerator = n;
    (*b).denominator = d;
}
```

Parentesen rundt den derefererte fraction-pekeren er nødvendig for å gjøre uttrykket entydig (uten parentesen kunne uttrykket tolkes som enten det b peker på, sin numerator, eller det b sin numerator peker på).

Når vi ofte foretrekker funksjonen av denne typen fremfor funksjoner som MakeFrac (avsnitt struct-er), er det fordi datautvekslingen mellom funksjonen og kallstedet blir mindre omfattende. Antar vi at en int tar 4 byte, vil en fraction ta 8 byte. Da vil vi ved et kall på lag_MakeFrac sende og motta 8 + 8 = 16 byte med data, mens vi ved et kall på SetFrac kan nøyde oss med å sende sende 12 byte — en 4-bytes peker + to 4-bytes heltall. Med større struct-er enn dette vil forskjellen i omkostningene ved datatransport mellom de to typene funksjoner øke.

15.4 Dynamisk allokering og deallokering av hukommelse til struct-er

Hvis et program skal arbeide med mer enn en instans av et struct, men antall instanser ikke kan bestemmes på forhånd, vil det lønne seg å sette av plass til de struct-etc man trenger, under kjøringen, etter hvert som behovet melder seg. Prøver man å sikre seg ved å deklarere det antall struct-er man tror man vil trene, kan man risikere at dette allikevel ikke var nok, og/eller at struct-deklarasjonene tar opp den plassen man trengte til andre data.

Det å sette av plass til data etter behov under programkjøringen, kaller vi dynamisk plassallokering. I Scheme foregår dette hele tiden bak kulissene, samtidig som garbage collector-en hele tiden prøver å frigjøre allokkert hukommelse som vi ikke lenger har bruk for (som det ikke lenger er noen referanse til i programmet). I Pascal og C derimot tar vi selv hånd om både allokeringen og frigjøringen av hukommelsen.

I C++ kan plassallokeringen bl.a. knyttes til opprettelsen av instanser av klasser, og den fører da med seg ting som vi ikke skal gå inn på foreløpig.

Standardfunksjonene for allokering og frigjøring av hukommelse i C er hhv. malloc og free — definert i <stdlib.h>

**Pascal**

```pascal
type
    rec = record
        a, b: char;
        c, d: integer
    end;

var
    pr: ^prec;
    ps: struct str *;
   ...
    Dispose(pr);
```

**C**

```c
struct
    str { char a, b;
        int c, d;
    };

struct str * ps;

New(pr);
...
Dispose(pr);
```

Parameter til malloc er antall byte den skal opprette; operatoren sizeof gir oss dette.

NULL (definert som 0 i stdio.h) er en pekerverdi som i C brukes som en peker til «ingenting» — bl.a. i forbindelse med dynamisk allokkerte objekter — f.eks. når vi skal sjekke om noe er siste objekt (med ingenting etter seg) i en liste med sammenkjedede objekter.

**Pascal**

```pascal
nil
```

**C**

```c
NULL
```
15.5 Dynamisk allokering og deallokering i C++
C++ har egne operatorer (innebygget i språket) for dynamisk allokering og deallokering av hukommelse.

`new` er en unær prefiksoperator som tar en hvilken som helst type som argument og returnerer en peker til en nyallokert hukommelsesblokk hvis størrelse er bestemt av type-argumentet.

```
int * ip = new int;  gir en peker til et heltall med ubestemt verdi.

Vi kan initialisere det allokerte objektet ved å angi en parentes med en verdi etter typeangivelsen etter `new`.

int * jp = new int(0);  gir en peker til et heltall med verdien 5.
```

Gitt typen `str` (et `struct`) i forrige avsnitt gir følgende setninger følgende resultat

```
str *sp = new str;  gir en peker til et `struct` med ubestemt innhold.

str s = {'A', 'B', 5, 8};  gir et `struct` med tegnene 'A' og 'B' og tallene 5 og 8.

str *pt = new str(s);  gir en peker til et `struct` med tegnene 'A' og 'B' og tallene 5 og 8.
```

Merk at vi ikke kan bruke konstante verdier for sammensatte typer som initialiseringsparameter.

```
Vi kan allokere en vektor ved å angi antall elementer i skarpe klammer etter typeangivelsen etter `new`.

int * iv = new int[8];  gir en peker til en heltallsvektor med 8 elementer.
```

`delete` er en unær prefiksoperator som tar som argument en peker til en hukommelsesblokk som ble allokert med `new`, og frigjør blokken.

**NB! `delete` endrer ikke pekerens verdi, og er denne av interesse andre steder, må vi selv sørge for at den settes til `NULL`.

15.6 Aksessor-operatoren for pekervariable
For å forenkle adgangen til feltene i et `struct` som bare er gitt ved en peker, har man innført operatøren `->`.

```
Pascal

pr^.a := 3;    ps->a = 3;
pr^.c := Chr(pr^.a);    ps->c = ps->a;
with pr^ do b = 0;    —

Det finnes ingen ekvivalent til Pascals with-setning i `C`.

`void SetFrac(struct broek * b, int n, int d)`
{  b->numerator = n;
  b->denominator = d;
}
```

15.7 Pekere som parametre
I `C`, som i Scheme, benyttes bare verdioverføring av parametre. Det er allikevel subtile forskjeller mellom `C` og Scheme hva paramateroverføring angår — noe vi skal komme tilbake til (Se Verdi- og referanseoverføring av parametre).

Hva `C` angår skal vi nøye oss med å konstatere at en aktuell parameter evalueres før det sendes til den kalte funksjonen.

Er den aktuelle parameteren en variabel, sends det på sett og vis en kopi av variabelen til funksjonen, og eventuelle endringer av parameteren, som opptrer som lokal variabel i funksjonen, vil ikke være synlige utenfor funksjonen.
Dette er vel og bra for enkle typer som tegn og tall, og for struct-er, men når en peker overføres som parameter, er det særdeles viktig å være oppmerksom på at en peker forblir den samme pekeren selv om den representeres ved to forskjellige variabler — den som under utførelsen av et funksjonskall bare er synlig inne i funksjonen, og den som også er synlig utenfor, og at det det pekes på også forblir det samme. Gjør man endringer i det en peker-paramater peker på, vil disse være synlige også utenfor funksjonen.

Under følger to eksempler, hvorav det første viser det bortkastede ved ganske enkelt å regne på parametre, mens det andre viser hvordan regneinnsatsen kan tas vare på hva pekere:

```c
void gi_igjen(int tot, int k20, int k10, int k5, int k1, int o50)
{
    k20 = tot / 2000;
    k10 = tot % 2000 / 1000;
    k5  = tot % 1000 / 500;
    k1  = tot % 500 / 100;
    o50 = tot % 100 / 50;
}

int main(void)
{
    int kr20, kr10, kr5, kr1, oere50;
    gi_igjen(14750, kr20, kr10, kr5, kr1, oere50);
    cout<<"Kr 147,50 blir "<<kr20<<" 20kr, "<<kr10<<" 10kr"<<endl;
    cout<<kr5<<" 5kr, "<< kr1<<" 1kr og "<<oere50<<" 50ører."<<endl;
}
```

Kjøring gir et svar som dette (der variablene i main skrives ut med sitt initielle grums — upåvirket av de beregningene som ble utført av gi_igjen):

Kr 147,50 blir 263616052 20kr, 0 10kr, 262893064 5kr, 2147430336 1kr og 0 50ører.

Løsningen er å overføre pekere til de aktuelle variablene:

```c
void gi_igjen(int tot, int* k20, int* k10, int* k5, int* k1, int* o50)
{
    *k20 = tot / 2000;
    *k10 = tot % 2000 / 1000;
    *k5  = tot % 1000 / 500;
    *k1  = tot % 500 / 100;
    *o50 = tot % 100 / 50;
}

int main(void)
{
    int kr20, kr10, kr5, kr1, oere50;
    gi_igjen(14750, &kr20, &kr10, &kr5, &kr1, &oere50);
    cout<<"Kr 147,50 blir "<<kr20<<" 20kr, "<<kr10<<" 10kr"<<endl;
    cout<<kr5<<" 5kr, "<< kr1<<" 1kr og "<<oere50<<" 50ører."<<endl;
}
```

Nå blir svaret riktig: Kr 147,50 blir 7 20kr, 0 10kr, 1 5kr, 2 1kr og 1 50ører.
15.8 Pekere og vektorer

I C gjelder to litt uvanlige konvensjoner:

• Et vektornavn evaluerer til en peker til vektorens 0’te element — slik at $v = \&v[0]$ — og når en vektor overføres som aktuell parameter, er det en peker til vektorens første element som overføres.

• Aksess av vektorelementer kan også uttrykkes med pekere: $v[i] = *(v + i)$

Det er altså det samme om vi skriver $v[3]$ eller $*(v + 3)$.

Anta vi har `char s[6];`

og at $s$ har adressen `0x03A50000`.  
Da er denne adressen verdien til $s$,
og adressen til $s[3]$ er $s+3$

\[ \&s[3] = 0x03A50005. \]

Her er det direkte samsvar mellom adresse- og pekeraritmetikken. Slikt samsvar blir det imidlertid bare fordi vektorelementene tar opp ett byte hver, og adressene gjelder de enkelte byte i RAM.

Anta vi har `short v[6];`

og at $v$ fikk adressen `0x03A50002`.  
Da er denne adressen verdien til $v$,
og adressen til $v[3]$ er $v+3$

\[ \&v[3] = 0x03A50008. \]

Here er det ikke samsvar mellom adresse- og pekeraritmetikken. Vektorlementene tar opp 2 byte hver, og pekerverdien inkrementeres med 2.

$v + 3$ betyr dermed

\[ \&v[0] + 3 * \text{sizeof}(short) = \&v[0] + 3 * 2 \]

Har vi `struct str {long a, b} sv[6];`,
vil $sv + 3$ bety

\[ \&sv[0] + 3 * \text{sizeof}(str) = \&sv[0] + 3 * 8 \]

Vi kan bruke pekeraritmetikk til å løpe igjennom en vektor. Følgende løkker gjennom strengen $s$ ved hjelp av tegnpikeren $p$ inntil nul påtreffes og konverterer små engelske bokstaver til store.

```c
for (p = s; *p; p++)
    if (\'a\' <= *p && *p <= \'z\')
        *p = 32;
```
15.9 Pekere til funksjoner

I Scheme er funksjoner førsteklasses objekter og kan uten videre opptre som funksjonsargumenter og liste og vektorelementer. C tilbyr noe som tar vare på noe av det samme, nemlig pekere til funksjoner. Det er imidlertid en viktig forskjell mellom Schemes funksjonsobjekter og Cs funksjonspekere, idet de første kan genereres dynamisk vha. lambda, mens de siste bare kan peke på funksjoner som er definert, kompilert og linket før programkjøringen starter.

I følgende definisjon er f en peker til en funksjon som tar ett \texttt{int}-argument og returnerer en \texttt{int}.

\begin{verbatim}
int funk_sum(int (*f)(int x), int n)    // Beregn summen av f(x) for x=1,...,n.
{
    int sum = 0, i;
    for (i = 1; i <= n; i++)
        sum += f(i);
    return sum;
}
\end{verbatim}

Vi kan definere typer av funksjonspekere.

\begin{verbatim}
typedef int (*funk)(int);
typedef int (*komb)(int, int);
\end{verbatim}

Her defineres typen \texttt{funk} som gir en peker til en funksjon som som tar ett \texttt{int}-argument, og typen \texttt{komb} som gir en peker til en funksjon som som tar to \texttt{int}-argumenter.

\texttt{funk} kan nå settes inn i funksjonshodet over:

\begin{verbatim}
int funk_sum(funk f, int n)
\end{verbatim}

15.10 Bruk av funksjonspekere

Bruk av et funksjonsnavn uten parenteser bak, gir en peker til funksjonen. (Dette implementeres som adressen til første instruksjon i funksjonen.)

\begin{verbatim}
int Odd(int x)    { return x % 2;  }
int Even(int x)   { return !(x % 2); }
int DivisibleBy3(int x) { return !(x % 3); }
int Square(int v) { return v * v; }
\end{verbatim}

\begin{verbatim}
void WriteFuncSum(const char *funk_navn, funk f, int n)
{
    cout << "sum " << funk_navn << "(" << n << ") " << funk_sum(f, n) << endl;
}
\end{verbatim}

\begin{verbatim}
void main()
{
    WriteFuncSum("odde", Odd, 10);
    WriteFuncSum("like", Even, 10);
    WriteFuncSum("delelig_med_3", DivisibleBy3, 10);
    WriteFuncSum("kvadrat", Square, 10);
}
\end{verbatim}

Kjøring av programmet gir følgende:

\begin{verbatim}
sum odde(10) 5
sum like(10) 5
sum delelig_med_3 3
sum kvadrat 385
\end{verbatim}

Når man ser på funksjonene like og delelig_med_3 kunne man kanskje begynne å lure på om det her var mulig å skrive en funksjon delelig som tok ett \texttt{int}-argument (en divisor) og returerte en annen funksjon som tok ett \texttt{int}-argument (dividend). Dette kunne vi ha gjort i Scheme slik:

\begin{verbatim}
(define (delelig divisor) (lambda (x) (if (zero? (remainder x divisor)) 1 0)))
\end{verbatim}
I C har vi imidlertid ingen lambda-mekanisme, og alle funksjoner må være kompilert og på plass i den lenkede filen, når denne eksekveres (dette gjelder for så vidt i praksis i Scheme også, men lambda-syntaksen får det i alle fall til å se ut som om funksjonene genereres dynamisk).

15.11 Pekere til pekere

Noen ganger trenger man en peker til en pekervariabel (f. eks.l fordi den skal overføres som parameter og endres).

En vanlige pekere deklareres slik

En peker til en peker deklareres slik

Peker-pekere får adressen til en pekervariabel som verdi

Vi kan i prinsippet også ha pekere til pekere til pekere, pekere til pekere til pekere til pekere, osv. — med så mange ledd vi måtte ønske.

15.12 Vanlige feil ved bruk av pekere

Det er noen feil som går igjen i programmering med peker — ikke minst når disse peker (eller skulle ha pekt) på dynamisk allokerede objekter:

- Vi glemmer å initialisere pekeren:

- Vi glemmer å frigjøre objektet:

Etter siste setning vil det allokerede objektet være utilgjengelig for oss, men det vil allikevel «flyte rundt» og oppta plass så lenge programmet kjører. Dette kalles en hukommelseslekkasje.

- Vi lar en global peker peke på en lokal variabel

Etter kallet på `f`, peker `p` på en variabel som ikke lenger finnes. Stedet på stakken der `x` lå, kan være tatt i bruk av andre funksjoner.

- Vi lar en peker peke på et resirkulert objekt!

Etter kallet på `free`, peker `q` på et objekt som er frigjort og hvis tidligere plass i hukommelsen kanskje er tatt i bruk gjennom nye kall på `malloc`.
16 Referanser

16.1 Deklarasjon og initialisering av referanser

En referanse er et alternativt navn, et alias, for et objekt (dvs. et dataobjekt i vid forstand).

Notasjonen \(\tau\) betyr referanse til et objekt av typen \(\tau\).

Eksempel:

```plaintext
int i = 4;
int* r = i; // i og r er nå to navn på variabel,
int x = r; // x er nå 4.
```

For å sikre at en referanse faktisk refererer til noe (er et genuint navn) må en referanse alltid initialiseres (med mindre den deklareres som `extern`, og må forutsettes initialisert et annet sted). Initialiseringen av en referanse er noe ganske spesielt, og skiller seg klart fra den — som f.eks. `i`. Når \(r\) har fått sin betydning etter initieringen over, vil den deretter i hele sin levetid fastholde denne betydningen (å være et alias for `i`), uansett hvilken verdi vi senere tilordner `r` (eller `i`).

Dette betyr at ingen operator kan ta en referanse som sådan som operand. Effekten av setningen `r++;` er at verdien til `i` økes med `1`, mens `r` qua referanse, beholder sin betydning (men vi kan vel tillate oss, når konteksten gjør det entydig, å si, noe upresist, at verdien til `r` økes med `1`).

Merk at heller ikke adresseoperatoren `*` kan opererer på en referanse som sådan. Setningen `int* p = &r`; fører til at `p`'s verdi blir `i`'s adresse — altså ikke `x`'s adresse, men adressen til det `x` refererer til.

16.2 Verdi- og referanseoverføring av parametre

Vi bruker referanser alt vesentlig for å spesifisere argumenter og returverdier fra funksjoner (herunder også returverdier fra overløsede operatorer — se Klasser og objekter).

Det aller viktigste for oss i denne forbindelsen er følgende:

Når et argument er spesifisert som en referanse, opptrer det under utførelsen av funksjonen som et alias for den aktuelle parameteren, og alle endringer av argumentet gjøres umiddelbart gjeldende for den variabelen det refererer til.

Eksempel:

```plaintext
void Add(int* addend, int addor) { addend += addor; }
```

Uansett når dette går ut på at den aktuelle parameteren tilordnes den formelle referanse-parameteren ved kallet.

Når vi i programmeringssammenheng generelt snakker om ulike former for paramateroverføring, angår dette først og fremst hvordan aktuelle parametre evalueres (reduseres), og de effektive endringer av parametrenes verdier i en funksjon har på omgivelserne for kallet.

I Scheme og C brukes bare verdiğerføring av parametre, mens Pascal og C++ som nevnt både har verdi- og referanseoverføring.

Ved verdiğerføring evalueres argumentene før prosedyren utføres, og resultatene av evalueringen sendes prosedyren på de respektive parameterplassene. I prosedyren opptrer de formelle argumentene som lokale variabler, og eventuelle endringer av disse verdier, får ingen effekt utenfor prosedyren.

Evalueringsformen sies å være strong (streng), idet den ikke tillater funksjonen å returnere før dens argumenter er ferdig evalueret.

67
Ved referanseoverføring sendes alternative navn på de aktuelle parametrene til prosedyren. Dermed kan bare variabler, og ikke konstanter, overføres på denne måten. I prosedyren opptar de formelle argumentnavnene som aliaser for de overførte parametrene, og eventuelle endringer i disse verdi gjøres dermed gjeldende utenfor prosedyren — også etter at kallet er utført.

Hva omgivelser og bieffekter angår, synes verdioverføring å være å foretrekke, dersom man ønsker å bruke det formelle argumentet fritt, som en lokal variabel, under utførelsen av prosedyren, uten at dette har effekter utenfor prosedyren. Dette holder imidlertid ikke dersom den overførte parameteren er en peker, og man gjør endringer i de data denne peker på, inne i prosedyren. Faktisk kan man ved å overføre pekere, i praksis realisere referanseoverføring i et språk som C, der man formelt sett bare har verdioverføring.

Dette er ikke tilsvarende likteil i Scheme. I og med at Scheme er typeødeløst, blir det her mer snakk om pragmatikk enn semantikk og syntaks. Ikke desto mindre er det en realitet i Scheme at lister, strenger og vektorer overføres vha. pekere, slik at hvis man gjør en `set-cdr!`, `string-set!` eller `vector-set!` på en lokal variabel (en parameter) i en Scheme-funksjon, så vil endringen gjøres gjeldende på den aktuelle listen, strengen eller vektoeren i de omgivelsene der funksjonen ble kalt.

Hva kostnadene ved selve overføringen angår, er referanseoverføring å foretrekke — først og fremst fordi det ikke innebærer noen dataoverføring. For mange Pascalprogrammerere gjelder forhåpentligvis verdier fra lokal parameteret i en lokal kopi i prosedyren. Av den grunn foretrekker man ofte referanseoverføring (var-overføring) av vektorer, selv om det ikke er logisk nødvendig mht. effektene i kallomgivelsene. I C er dette problemet løst for vektorer ved at man kan overføre en koderer verb til en vektor-variabel rett og slett og slette adresse til vektoeren som er gitt som parameter. Dette kan man gjøre i Pascal ved å overføre en peker til det aktuelle struct-et, fremfor struct-et selv.

Det at referanseoverføring ikke innebærer noen `evaluating`, er ikke kostnadsmessig relevant. Det har bare betydning når den aktuelle parameteren er et uttrykk bestående av regnestykker og/eller funksjonskall, og i så fall gir ikke referanseoverføring noen mening.

Hva klarheten angår, er meningene delte.

Overføring av parametre ved referanse gjøres i Pascal tilsvarende slik det gjøres i C++ (men Pascal har ingen syntaks for referanser i og for seg). For mange Pascalprogrammerere gjelder følgende konvensjon: Hvis en prosedyre endrer én verdi i sine omgivelser, skal den deklareres som en funktion som returnerer verdien, men hvis en prosedyre endrer flere verdier, skal den deklareres med tilsvarende mange referanseoverførte parametre.


Hva referanseoverføring ikke innebærer noen `evaluating`, er ikke kostnadsmessig relevant. Det har bare betydning når den aktuelle parameteren er et uttrykk bestående av regnestykker og/eller funksjonskall, og i så fall går ikke referanseoverføring noen mening.

I henhold til Stroutstups anbefalinger og konsensuell Pascal-programmering burde vi skrive overføringen slik:

```c
int Add(int addend, int addor) { return addend + addor; }  
int main() { int a = 10; a = Add(5); }  
```

Det finnes andre overføringsstrategier enn de to som er nevnt over. Viktigst er kanskje den som kalles `nameoverføring`. Ved navneoverføring sendes det aktuelle argument-uttrykket til prosedyren uten noen forutgående evaluering; dvs. i praksis sendes det gjerne en peker til en kodesekvens `Add()` som vil returere argument-uttrykks verdi, bundet sammen med uttrykkets omgivelser der verdiene til uttrykkets frie variabler er gitt.
17 Utvidede funksjonsdomener

Bruk av utvidede funksjonsdomener kalles på engelsk *function overloading*, og av L&L utvidelse av bruksområde for *funksjoner* (vi kaller det dessuten *overissing av funksjoner*). Strengt tatt er alle termene litt misvisende. Det det dreier seg om er at *funksjoner identifiseres ved navn + argumentliste*. Dette er nokså unikt for C++ — i alle fall i forhold til andre prosedyrale språk.

17.1 Funksjoner som tar ulike typer enkle argumenter

Som nevnt finnes det ingen språklige mekanismer i C / C++ for direkte sammenligning av strenger, men standardbiblioteket *string* i C har funksjonen `strcmp` som tar to strengarumenter og returnerer −1 hvis første streng er mindre enn andre (etter en tegn-for-tegn sammenligning), 0 hvis første streng er lik andre, og 1 hvis første streng er større enn andre. Vi ønsker å definere en generell funksjon for sammenligning av verdier — i første omgang tall og strenger.

Dette gjører vi ved å definere to funksjoner som har samme navn, men tar ulike typer argumenter:

```c
int Compare(const double d1, const double d2)
{
    return d1 < d2 ? -1 : d1 == d2 ? 0 : 1;
}

int Compare(const char * s1, const char * s2)
{
    return strcmp(s1, s2);
}
```

I prinsippet har vi her to ulike funksjoner med samme navn, men i praksis har vi én funksjon som kan ta to ulike typer argumenter. Hvilken funksjon som faktisk kalles i det ene eller det andre tilfellet, bestemmes av hvilke type argument funksjonen mottar.

Merk at når vi har definert `Compare` med to `double`-argumenter, kan vi ikke også definere følgende:

```c
int Compare(const char c1, const char c2)
int Compare(const long l1, const long l2)
```

Dvs. vi får lov til å definere disse funksjonene, men prøver vi å kalle `Compare` med en eller annen form for tall (*bool, char, int, long, float, double*) får vi melding om at kallet er tvetydig (*ambiguous call to overloaded function*). Poenget er at kompilatoren ikke klarer å skille ulike typer tall fra hverandre i denne sammenhengen — og dette gir da også mening, forsåvidt som alle typer tall skal være kompatible.

17.2 Funksjoner som tar både enkle og sammensatte typer som argumenter

La oss utvide eksempellet med en `struct`-type. Vi ønsker å kunne sammenligne ulike instanser av denne typen med hemblikk på en innbyrdes ordning av dem, og lar ett felt i `struct`-et være sorteringskriterium.

```c
struct strct {
    char * key;
    long data;
};
```

Vi kan nå utvidet domenen for `compare` til også å omfatte `strct`.

```c
int Compare(const strct s1, const strct s2)
{
    return Compare(s1.key, s2.key);
}
```

Vi kan imidlertid ikke sende noen konstantverdi (av den typen vi kan angi som initiell verdi i en deklarasjon av en `struct`-variabel) på `struct`-plassen til en slik funksjon.
17.3 Stilltiende argumenter — Inntil et gitt antall argumenter

Vi kan også utvide domenet for en funksjon ved å variere antall argumenter. I biblioteket string finnes funksjonen `strcat` som tar to strengargumenter og skjøter andre streng til første. Vi ønsker å definere en funksjon `Concat` som tar inntil 4 strengargumenter for tilskjøtning til den første strengen.

```c
void Concat(char * s1, const char * s2,
            const char * s3 = NULL, const char * s4 = NULL, const char * s5 = NULL)
{
    strcat(s1, s2);
    if (s3)
        Concat(s1, s4);
    if (s4)
        Concat(s1, s4);
    if (s5)
        Concat(s1, s5);
}
```

`Concat` tar minst 2 og inntil 5 argumenter. De argumentene som utelates, får den stilltiende verdien (default-verdien) som er angitt i definisjonen. I dette tilfellet vil det si at med 2 aktuelle argumenter får vi i tillegg stilltiende 3 argumenter med verdien `NULL`, og med 3 aktuelle argumenter, får vi stilltiende 2 `NULL`-argumenter, osv..

Argumenter som det angis stilltiende verdier for, må stå sist i argumentlisten, og er det mer enn ett slikt argument, må alle ha kompatible typer.

17.4 Funksjoner som tar et vilkårlig antall argumenter

Det er også mulig å definere funksjoner som tar vilkårlig mange argumenter på (ikke bare inntil et gitt antall som over). Mens utvidelse av funksjonsdomener ellers bare er mulig i C++, er bruk av et vilkårlig antall argumenter mulig i C (se K&R 7.3). Til dette må vi bruke standardbiblioteket `<stdarg.h>`. Her finner vi bl.a.

```c
typedef pointer_to_argument_list va_list;
void va_start(va_list pa, void * ppa);
type va_arg(va_list pa, type arg_type);
void va_end(va_list pa);
```

`pa` skal peke på første argument i den vilkårlig langde delen av argumentlisten (VA), mens `ppa` skal peke på siste argument `foran` denne delen — siste ordinære argument.

`type` er dynamisk og bestemmes av det aktuelle argumentet, når `va_arg` kalles suksessivt for hvert argument i listen.

Bruken av disse funksjonene er enklast forklaart vha et eksempel. Anta vi ønsker å arbeide med mengder av tegn. Vi definerer en tegnmengdetype som en boolesk vektor med 256 plasser.

```c
typedef bool CHARSET[256];
typedef bool * PCHARSET;
```
Alle elementene kan initieres til \textit{false} slik:

\begin{verbatim}
void SetEmptyCharSet (PCHARSET s)
{
    for (unsigned int i = 256; i; s[--i] = false);
}
\end{verbatim}

Elementene tilsvarende de tegn som legges inn i mengden, kan så settes til \textit{true} slik.

\begin{verbatim}
void AddCharsToSet (PCHARSET s, ...)                   // (a)
{
    va_list addChars;                                    // (b)
    va_start(addChars, s);                               // (c)
    for (; ;)
    {
        unsigned char c = va_arg(addChars, unsigned char); // (d)
        if (c == 0) break;                                 // (e)
        s[c] = true;                                       // (f)
    }
    va_end(addChars);                                    // (g)
}
\end{verbatim}

(a) Siste del av argumentlisten ..., etter \textit{s}, er vilkårlig lang.
(b) Deklarer en peker til VA av typen \textit{va_list}.
(c) Kall \textit{va_start} for å identifisere starten på VA — bl.a. vha. siste argument foran denne.
(d) Kall \textit{va_arg} for å dentifisere første argument i den gjenværende delen VA, og flytte argumentpekeren til neste argument.
(e) Vi har kommet til terminasjonsverdien for VA, så hopp ut av løkken.
(f) Sett mengde-element for neste argument i VA.
(g) Kall \textit{va_end} for å avslutt behandlingen av VA.

I Scheme kan vi få til dette mye enklere vha. en prikket argumentliste — \textit{(define (funk første-arg . resten)...)}. 

\__________________________

C++ gir også mulighet for å utvide domenen for \textit{operators}, men dette er knyttet eksklusivt til klasser.
18 Inlining

*Inline* eller *unfold* vil i programmeringssammenheng si å erstatte et funksjonskall med selve koden til funksjons-kroppen — vanligvis under kompileringen.

Poenget med inlining er å få programmet til å gå fortere. Funksjoner som inlines er typisk enkle funksjoner (med én eller noen få programlinjer) som kalles ofte.

Inlining av en rekursiv funksjon er ulovlig og vil i prinsippet kunne føre kompilatoren inn i en uendelig regress, hvis den ikke er smart nok til selv å oppdage ulovligheten.

*Overvåg inlining* vil i prinsippet kunne medføre eksponentiell vekst av koden — f.eks. hvis \( f \) kaller \( g \) to ganger, og \( g \) kaller \( h \) to ganger og \( h \) er inlinet i \( g \), og \( g \) er inlinet i \( f \), vil vi få fire kopier av kroppen til \( h \) i \( f \).

I praksis vil de fleste C++-kompilatorer være smarte nok til å unngå ovennevnte problemer.

(Inlining er ikke tillatt i C.)

I C++ gjøres inlining enten eksplisitt ved spesifikasjonen `inline`, eller implisitt i definisjonen av en klasse.

Opererer vi med en header-fil (eller flere), kan vi plassere fulle definisjoner av `inline`-spesifiserte funksjoner der.

Både i et slikt tilfelle, og når en funksjon inlines implisitt i en klasse, erstatter vi prototypen med den fulle funksjons-definisjonen.

Eksempel på inlining av fem funksjoner for sammenligning av strenger:

```c++
inline bool StrEQ(char*s1, char*s2){return !strcmp(s1,s2);} // strcmp ret. 0 for ==
inline bool StrLT(char*s1, char*s2){return strcmp(s1,s2) < 0;} // strcmp ret. -1 for <
inline bool StrGT(char*s1, char*s2){return strcmp(s1,s2) > 0;} // strcmp ret. 1 for >
inline bool StrLE(char*s1, char*s2){return strcmp(s1,s2) <= 0;} //
inline bool StrGE(char*s1, char*s2){return strcmp(s1,s2) >= 0;} //
```

Det er visse likheter mellom inlining og macro-definisjoner (merk at vi bl.a. snakker om *unfolding* i forb.m. inlining og *expansion* i forb.m. makroer). Følgende macro-definisjoner gir noenlunde samme effekt som ovenstående inline-definisjoner, forutsatt at kompilatoren faktisk foretar inliningen.

```c++
#define STR_EQ(s1, s2) (!strcmp((s1), (s2)))
#define STR_LT(s1, s2) ( strcmp((s1), (s2))< 0)
#define STR_GT(s1, s2) ( strcmp((s1), (s2))> 0)
#define STR_LE(s1, s2) ( strcmp((s1), (s2))<=0)
#define STR_GE(s1, s2) ( strcmp((s1), (s2))>=0)
```

Forskjellene fremgår delvis av den notasjonen som er brukt. I en macro-definisjon sies det ingenting om typer. Definisjonen gir rett og slett en anvisning for tekstsubstitusjon, slik at f.eks. uttrykket

```c++
bool equal = STR_EQ(a, b);
```

erstattes av preprosessoren med uttrykket

```c++
bool equal = (!strcmp((a), (b)));
```

og det vil da være opp til kompilatoren å sjekke at \( a \) og \( b \) har rett type.

Forøvrig består forskjellen i at mens `#define` er et direktiv som preprosessoren *må* følge, er `inline` bare en oppfordring til kompilatoren om å inline den etterfølgende definisjonen, hvis den finner det mulig og forsvarlig.

73
19 Klasser og objekter

En klassetype er en datatype vi definerer selv (egendefinert, *user-defined*, i mots.t. forhåndsdefinert, *predefined*). En klasse har visse begrensete likheter med et *struct*, for så vidt som begge kan angi en samling av ulike typer innenfor samme struktur. Til en klasse er det imidlertid også direkte knyttet egne funksjoner — klasse*metoder*, til forskjell fra funksjoner som tar *struct*-er eller *struct*-pekere som parametre og/eller returnerer *struct*-er eller *struct*-pekere. En klasse instansieres i objekter, og (de ikke-statistiske) metodene knyttes i en viss forstand direkte til de enkelte objektene, for så vidt som en metode opererer på de data som finnes i det objektet den tilhører. Den kompilerede koden for en metode vil imidlertid være gitt én gang for alle, og brukes av objektet, bl.a. vha. en del datatransport bak kulissene.

19.1 Definering av en klasse

En klasse kan f.eks. defineres slik:

class CAccount
{
    public:
        CAccount();                                // Konstruktor
        void    SetBalance  (double);             // Sett saldo
        void    SetCredit   (double);             // Sett kreditt
        double  GetBalance  () const;            // Hent saldo
        double  GetCredit   () const;            // Hent kreditt
        double  GetAvailable() const;             // Hent tilgjengelig saldo
        void    Deposit     (double);             // Innlevere
        bool    Withdraw    (double);             // Hente
    
    private:
        double mBalance;                         // Saldo
        double mCredit;                          // Kreditt
};

Merk at vi avslutter klasse definisjonen med semikolon — slik vi avslutter en *struct*-definisjon.

Under følger implementasjonen av klassens metoder — utenfor selve klassedefinisjonen som her bare inneholder det vi kaller metodenes *prototyper* i tillegg til deklarasjonene av data-attributtene.

Siden implementasjonen av klassens metoder ligger utenfor klassedefinisjonen bruker vi klassenavnet sammen med operatoren :: for å angi tilhørigheit mellom klassen og metodeimplementasjonene.

CAccount::CAccount(): mBalance(0), mCredit(0) {}                 // Konstruktor
void    CAccount::SetBalance  (double) { mBalance = balance; }
void    CAccount::SetCredit   (double) { mCredit = limit; }
double  CAccount::GetBalance  () const { return mBalance; }
double  CAccount::GetCredit   () const { return mCredit; }
double  CAccount::GetAvailable() const { return mBalance + mCredit; }
void    CAccount::Deposit     (double amount) { mBalance += amount; }
bool    CAccount::Withdraw    (double amount) { if (GetAvailable() < amount) return false; mBalance -= amount; return true; }
19.2 Aksessering av metoder og data

I en klassens metode kan vi aksessere klassens metoder og data direkte.
Alternativt kan vi bruke den globale variabelen **this** som alltid refererer til det objektet hvis metode vi til enhver tid befinner oss i. Dette er en konvensjon som etablerte seg i deler av Pascal-litteraturen for noen år siden. Poenget er at bruken av **this** (eller **self** som det heter i Object Pascal) gjør det helt entydig hva som er objekt-attributter og hva som er lokale, eller globale variabler. Et annet poeng på Macintosh, der man kan risikere å operativsystemet flyttet det aktuelle objekt fra et sted i hukommelsen til et annet under utførelsen av en metode, var å sikre gyldigheten av referansen til objektets attributter. Uavhengig av hva operativsystemet måtte finne på på egen hånd, er dette ikke noe problem i 
++.

Utenfor klassedefinisjonen og metodeimplementasjonene må vi enten bruke
- tilhørighetsoptatorenen :: sammen med klassens navn, for å aksessere typer, konstanter og numererte verdier i klassen, og statiske (se under) metoder.
- prikk-aksessoren . for globalt deklarerte eller automatiske (lokalt deklarerte) objekter, eller
- pil-aksessoren -> for dynamisk allokerte objekter, når vi refererer til en klassens offentlige data og metoder.

19.3 Offentlige, beskyttede og private metoder

**public** angir at metoder og felt er tilgjengelige for omverdenen.
**private** angir at metoder og felt bare er tilgjengelige for klassens egne metoder.
**protected** (ikke vist over) angir at metoder og felt bare er tilgjengelige for metodene til klassen og dens subklasser.

I prinsippet er det mulig å definere offentlige data-attributter, men gjør vi det, tillater vi hvem som helst å endre et objekts data — noe vi normalt ikke anser som ønskelig.
På den andre siden kan vi godt definere private metoder. Dette gjør vi når vi ønsker metoder for å endre et objekts data, som hjelp i andre metoder.

**const** sist i et metodehode angir at metoden ikke endrer dataene i det aktuelle objektet.

Merk at mens **public** og **private** er termer som brukes eksklusivt i definisjoner av klasser, brukes **const** i ulike sammenhenger for å angi at nøe ikke skal endres — en variabel eller en parameter.

19.4 "Inlining" av klasse metoder

Metoder hvis implementasjon omfatter bare én eller svært få setninger, og som brukes ofte, kan vi implementere direkte i klassedefinisjonen — f.eks. slik:

```cpp
class CAccount
{
    public:    CAccount(): mBalance(0), mCredit(0) {}        // Konstruktor
    void SetBalance (double b) { mBalance = b;  }
    void SetCredit (double c) { mCredit = c;       }
    double GetBalance () const { return mBalance; }
    double GetCredit  () const { return mCredit;  }
    double GetAvailable() const { return mBalance + mCredit; }
    void Deposit     (double d) { mBalance += d;          }
    void Withdraw    (double);  // Implementert utenfor klassedef.

};
```

Her er alle metodene bortsett fra Withdraw inlinet ved at deres fulle definisjoner er gitt direkte i klassen. Disse inneholder alle typisk én setning hver. Vi lar Withdraw stå utenfor klassen, siden den er mer omfattende enn de øvrige (men den er ikke større enn at også den godt kunne ha vært inlinet). (Se forøvrig Inlining).
19.5 Konstruktører og destruktører

En klasse' metoder vil alltid omfatte — implisitt eller eksplisitt — en konstruktør og en destruktør.
- Konstruktoren kalles **automatisk** når vi deklarerer et objekt, eller allokerer et objekt dynamisk med operatoren `new`.
- Destruktoren kalles automatisk når vi **forlater dens scope** (den blokken der objektet er deklarert), eller **allokerer et dynamisk alloket objekt med operatoren delete**.

NB! vi skal altså aldri kalle en konstruktør eller en destruktør selv.
Hverken konstruktøren eller konstruktøren har returtype, og **void skal ikke anges foran -struktornavnet**.

19.5.1 Konstruktører

En konstruktør skal ha samme navn som klassen.
Vi anger konstruktøren eksplisitt, bl.a. når vi ønsker å sikre at objektenes data har bestemte initielle verdier, og eller at visse operasjoner utføres idet en instans av klassen opprettes (f.eks. at det allokeres hukommelse på heapen for objektets data, at en fil åpnes, e.l.).
Har konstruktøren argumenter, angis disse i parentes etter objektNAVnet når vi deklarere eller allokerer et objekt — på samme måte som aktuelle parametre angis ved et funksjonskall.

Utenfor klassedefinisjonen har konstruktøren formen

```
klassenavn::klassenavn (argumentliste) : initialiseringsliste {kropp}
```

Initialiseringslisten hører til implementasjonen, sammen med kroppen, og prototypen i klassen har formen:

```
klassenavn (argumentliste);
```

Initialiseringslisten, som er valgfri, har formen

```
felt(initialverdi), felt2(initialverdi), ...
```

Utelates listen, skal også kolon etter argumentlisten utelates.

En konstruktør som kan kalles uten argumenter (som er definert uten argumenter eller bare har stilltiende argumenter) blir defaultkonstruktøren (den stilltiende konstruktøren). Defultkonstruktøren til `CAccount` nøyer seg med å nullstille objektets felt i initialiseringslisten, og har ellers en tom kropp

Det er nokså vanlig å overlesse konstrukturen med ulike argumentlister. En enkel variant kunne ha en argumentliste med feltenes initialverdier, og en initialieringsliste der disse bl ble brukt — f.eks. slik:

```
CAccount::CAccount(double b, double c): mBalance(b), mCredit(c) {}
```

Dersom et objekt i utgangspunktet trenger plass på heapen for sine data (f.eks. til en vektor), må klassen til objektet ha en konstruktør som allokerer denne plassen.

Dersom et objekt med data på heapen skal kunne kopieres, må det dessuten ha en kopikonstruktor som tar en referanse til originalobjektet som argument.

```
CAccount::CAccount(CAccount& orig){...};
```
Kroppen til kopikonstruktoren vil normalt inneholde like mange setninger som det er felt —
enten for direkte kopiering slik
felt = orig.felt;
eller for allokering og kopiering slik:
pData = new (datatype);
KopierData(pData, orig.pData);

En kopikonstruktør kan ta flere argumenter, etter originalen, men i så fall må det for alle disse være angitt stilltiende (default) aktuelle argumenter.

For konstruktører som bare tar ett argument er følgende to skrivemåter ekvivalente:
CSomeClass object(arg);
og
CSomeClass object = arg;

Vær imidlertid oppmerksom på at denne formen for initialisering ikke gir samme resultat som en senere tilordning, medmindre vi har definert en tilordningsoperator for klassen som sikrer dette (se Utvidelse av operatordemener).

19.5.2 Destruktører

En destruktør skal ha samme navn som klassen, prefikset med tegnet ~ — f.eks. slik:
CAccount::~CAccount(){...;}

Vi angir destruktøren eksplisitt når det er behov for å rydde opp i data, før et objekts eksistens opphører.
Vi har ikke så ofte bruk for en eksplisitt destruktør som for en eksplisitt konstruktør, men dersom et objekt kan ha data på heappen idet det dreieres, må klassen til objektet ha en destruktør som deallocerer denne plassen.
Destructøren kalles automatisk når vi forlater den programblokken der det aktuelle objektet er deklarert, eller når vi deallocerer et objekt dynamisk med operatoren delete.

19.6 Static klassevariabler

I noen tilfeller ønsker vi at et datum skal være gitt for alle objekter samtidig — typisk et konstant tall eller en konstant streng. For eksempelets skyld kan vi tenke oss en klasse for en familie de alle medlemmer har samme etternavn. For å slippe å definere dette vha. #define, eller som en global const-variabel, kan vi angi det inne i klassen som static.

class Familiedlem
{
static char etternavn[32];
char fornavn[32];
long fdato;
static char setEtternavn(char* navn) { strcpy(Familiedlem::etternavn, "Hansen"); } 
...
}

Siden etternavnet tilhører klassen, kan det ikke settes i noen konstruktør. Det kan i det hele tatt ikke settes for noen enkelt objekt, dvs. vi må betakte klassen selv som et objekt og angi denne som etternavnets eier.
strcpy(Familiedlem::etternavn, "Hansen");

Vi foretrekker imidlertid å gjøre dette via en modifikator:
Familiedlen::setEtternavn("Hansen");

Merk at en funksjon som opererer på et static felt, selv må være spesifisert som static. Metoder som tilhører de enkelte objektene kan aksessere et static felt, men ikke endre det.
19.7 Superklasser og subklasser — arv og polymorfri


Maskinen holder rede på hvilken versjon som skal kalles for det aktuelle objektet vha. en egen tabell "bak kulissene" (dvs. som det ikke refereres til i programkoden).

I eksemplet under definerer vi superklassen CPerson som har felt og metoder som er felles for alle slags personer, og under denne de to arvingene CCensusPerson og CChurchPerson for personer fra hhv. folketellinger og kirkebøker.

```cpp
// Følgende typedefinisjoner er innført av plasshensyn
typedef const char CCHR;
typedef const char * PSTR;
typedef const long CLNG;

class CPerson
{    private: char mLName[64];
    char mMName[64];
    char mFName[64];
    long mBirthDate;
    char mSex;
    ...
    public: CPerson();
    CPerson(PSTR l, PSTR m, PSTR f, CLNG b, CCHR s);
           virtual void WriteData() const;
           ...
};
virtual betyr "kan omdefineres i en subklasse".
cost etter funksjonshodet betyr, som nevnt, at funksjonen ikke endrer det aktuelle objektets data.

CPerson har en default konstruktør som ganske enkelt nullstiller alle feltene.

CPerson::CPerson() : mBirthDate(0),mSex(0)
{    *mLName    = 0; // Sett 0'te tegn i mLName til '\0'
    *mMName    = 0; //        "         mMName    
    *mFName    = 0; //        "         mFName    
}

I tillegg har klassen en konstruktør som tar ett argument for hvert felt.

CPerson::CPerson(PSTR l,PSTR m,PSTR f,CLNG b,CCHR s) : mBirthDate(b),mSex(s)
{    strcpy(mLName, l);
    strcpy(mMName, m);
    strcpy(mFName, f);
}

Legg merke til at strengfeltene ikke kan få sine verdier satt i initialiseringsalisten. Dette er fordi en streng, eller for den saks skyld en hvilken som helst vektor, ikke kan få sin verdi satt i en tilordningssetning (bortsett fra i initialiseringsleddet i en deklarasjon). En grei måte å gi én streng verdien til en annen på, er å kalle strcpy — som her.

Vi definerer CCensusPerson som en subklasse av CPerson ved å føye : public CPerson til klassenavnet.
CCensusPerson arver dermed alle feltene og metodene til CPerson.

79
class CCensusPerson: public CPerson
{
    private: char mSocialStatus[8];
             char mSivileStatus[8];
             char mOccupation[32];
...
    public: CCensusPerson();
           CCensusPerson(const CCensusPerson& orig);
           CCensusPerson(PSTR l, PSTR m, PSTR f, CLNG b, CCHR s,
                          PSTR sos, PSTR sis, PSTR occ);
    void WriteData() const;
...
};

Klassen har, i likhet med sitt opphav, en defaultkonstruktør og en feltargumentkonstruktør. I tillegg har den en kopi-konstruktør — kjennetegnet som sådan ved at den tar en referanse til en instans av klassen som eneste argument (se Konstruktører over).

CCensusPerson::CCensusPerson() : CPerson()
{
    *mSocialStatus = 0;
    *mSivileStatus = 0;
    *mOccupation   = 0;
}

CCensusPerson::CCensusPerson(const CCensusPerson& orig)
: CPerson(orig.mLName, orig.mMName, orig.mFName, orig.mBirthDate, orig.mSex)
{
    strcpy(mSocialStatus, orig.mSocialStatus);
    strcpy(mSivileStatus, orig.mSivileStatus);
    strcpy(mOccupation,   orig.mOccupation);
}

CCensusPerson::CCensusPerson(PSTR l, PSTR m, PSTR f, CLNG b, CCHR s,
                             PSTR sos, PSTR sis, PSTR occ)
: CPerson(l, m, f, b, s)
{
    strcpy(mSocialStatus, sos);
    strcpy(mSivileStatus, sis);
    strcpy(mOccupation,   occ);
}

Legg merke til at opphavet CPerson initialiseres i initialiseringslisten på samme måte som den ville ha blitt initialisert i en deklarasjon.

Igjen er vi henvist til å bruke strcpy i kroppen til konstruktoren for å få initialisert strengfeltene.

Endelig definerer vi også CChurchPerson som en subklasse av CPerson.
class CChurchPerson: public CPerson
{
    private:
    const CChurchPerson* mFather;
    const CChurchPerson* mMother;
    long mBaptDate;
...

    public:
    CChurchPerson ()
        : CPerson(), mFather(NULL), mMother(NULL), mBaptDate(0)
    {}
    CChurchPerson (const CChurchPerson& orig)
        :CPerson(orig.mLName, orig.mMName, orig.mFName,
                orig.mBirthDate, orig.mSex),
                mFather(orig.mFather), mMother(orig.mMother),
                mBaptDate(orig.mBaptDate)
    {}
    CChurchPerson (PSTR l, PSTR m, PSTR f, CLNG b, CCHR s,
                const CChurchPerson* dad, const CChurchPerson* mom,
                CLNG bpt)
        :CPerson(l, m, f, b, s), mFather(dad), mMother(mom), mBaptDate(bpt)
    {}
    void WriteData() const;
...};

Klassen har samme typen konstruktører som CCensusPerson, men vi kan nå implementere alle disse fullstendig i initialiseringslisten — såvel superklassen som de tre feltene, siden de alle er tall — to pekere og et stort heltall. — Og i og med at kroppene i konstruktorene blir tomme, inliner vi like godt alle sammen.

Vi implementerer de tre utgavene av SkrivPersondata slik:

void CPerson::WriteData() const
{ cout << "NAme: " << mLName << " ";
    if (*mMName) cout << mMName << " ";
    cout << mFName << endl;
    cout << "Born: " << mBirthDate << "Sex: " << mSex << endl;
}

void CCensusPerson::WriteData() const
{ CPerson::WriteData();
    cout << "Soc.Status: " << mSosialStatus
         << "Siv.Status: " << mSivileStatus
         << "Occupation: " << mOccupation << endl;
}

void CChurchPerson::WriteData() const
{ cout << "Father: " << endl;
  mFather->WriteData();
  cout << "Father: " << endl;
  mFather->WriteData();
  CPerson::WriteData();
  cout << "Baptised: " << mBaptDate << "Sex: " << mSex << endl;
Vi kaller superklassens metode (ved å angi klassens navn og tilhørighetsoperatoren i kallet) før vi utfører det som er spesielt for subklassen. Kallet på superklassens metode kan stå først, mellom andre setninger eller sist.

Vi kan også overskrive superklassens metode fullstendig ved å la være å kalle den. Nyttet av dette fremgår ikke av eksemplet over. I eksempelet under derimot, der flyten i utførelsen av en del av programmet organiseres på øverste klassenivå ved at én metode kaller en serie med andre polymorft definerte metoder, må vi ha prototyper av alle metodene allerede på øverste nivå, mens metodene implementeres på ett eller flere av de underliggende nivåene. Hvilke aktuelle versjoner av metodene som utføres, bestemmes av hvilken klasse det aktuelle objektet er en instans av.

Vi har klasser for å håndtere to typer vinduer — tekst- og tegnevinduer. Over disse er det definert en generisk vindusklasse (en generisk eller abstrakt klasse er en klasse som aldri instansieres ved noe objekt):

```cpp
class CWindow
{
private:
    HINSTANCE mhInst;
    HWND mhWind;
    char msFileName[256];
    ...
public:
    long OnMessage(UINT, WPARAM, LPARAM);
    virtual long OnCommand(WPARAM, LPARAM) = 0;
    virtual long OnSize(WPARAM, LPARAM) = 0;
    long OnSetFocus();
    long Close();
    ...
}
```

virtual betyr som nevnt kan omdeneres i en subklasse.

= 0 etter funksjonshodet betyr skal ikke implementeres her, og må defineres i en subklasse.


OnCommand kan ta seg av noen kommandoer på dette nivået, og returner 1 eller 0 avhengig av om den aktuelle kommandoen ble behandlet eller ikke.

OnSize kan ikke gjøre noe på dette nivået, og all behandling overlates til subklassenes metoder.

OnSetFocus og Close kan gjøre seg ferdig på dette nivået, men behandlingen kan innebære kall på metoder som bare er implementert i subklassene (og som ikke er tatt med her).

```cpp
long CWindow::OnMessage(UINT msgCode, WPARAM wParam, LPARAM lParam)
{
    switch(msgCode)
    {
    case WM_SIZE:
        return OnSize(wParam, lParam);
    case WM_SETFOCUS:
        return OnSetFocus();
    case WM_COMMAND:
        return OnCommand(wParam, lParam);
    case WM_CLOSE:
        return !Close();
    default:
        return 0;
    }
}
```

CTextView og CDrawingWindow trenger sine egne versjoner av OnCommand og OnSize (her viser vi bare klasse-definisjonene, siden implementasjonene av disse er uten interesse for det vi skal illustrere).
class CTextWindow: public CWIndow
{
private:
    ...
public:
    long OnCommand(WPARAM, LPARAM);
    long OnSize(WPARAM, LPARAM);
    ...
}
class CDrawingWindow: public CWIndow
{
private: ...
public:
    long OnCommand(WPARAM, LPARAM);
    long OnSize(WPARAM, LPARAM);
    ...
}

19.8 Flerfoldig arv

Én klasse kan ha mer enn én superklasse, og får i så fall en flerfoldig arv (multiple inheritance).
Vi tenker oss et program for produksjon ulike ordlister for en gitt tekstfil, samt statistikk om ord- og setningslengder.
Til dette kan vi bruke et hierarki av klasser for produksjon av de ulike ordlistetypene.

CPrefs                              CFile            ————————————————————— | ———————————————                ————————————— | ———————————            |                        |              |                |                                    |
Diverse dialogklasser

CIFile          COFile                                    |              :................    ...                                   |                                       |                                       |

CTextStat

De spesifikke ordlisteklassene tar hånd om det som er spesielt for produksjonen av de ulike typene ordlister (frekvensordliste, indeks, konkordans, etc.).
CTextStat tar hånd om lesing av teksten og beregning og skriving av statistikk.

Av diagrammet ser vi at CWordList har både CPrefs og CWordListIO som superklasser.
Både CWordList og CTextStat trenger egenskapene til, og er derfor subklasser av, CWordListIO.
CWordlist trenger dessuten (i mots. CTextStat) egenskapene til CPrefs, og der derfor også en subklasse av denne.

De stiplede linjene fra CFile og COFile til CWordListIO betyr at den siste inneholder en instans av hver av de to første (egentlig inneholder CWordListIO pekere til et CFile- og et COFile-objekt, men det er uvesentlig her).

Vi kan illustrere flertydighet med en utvidelse av personklassehierarkiet. Anta at vi ønsker å innføre en klasse for personer som antas og være registrert både i en folketelling og en kirkebok, uten at de data som er felles i de to kildene behøver å være helt identiske. Vi kaller klassen `CDoublePerson` og lar den være en subklasse av både `CCensusPerson` og `CChurchPerson`. Vi kan f.eks. tenke oss at det har vært brukt ulike skrivemåter for navnet, og/eller at fødselsdatoen er upresis angitt i den ene eller den andre, eller for den saks skyld i begge kildene. Hvilke kriterier vi måtte ha brukt for å avgjøre om to poster i hhv. en kirkebok og en folketelling faktisk representerer samme individ, er uten betydning her.

```c
class CDoublePerson: public CCensusPerson, public CChurchPerson
{
    private:
        bool SameLName(){return !strcmp( CCensusPerson::mLName, CChurchPerson::mLName);}  
        bool SameMName(){return !strcmp( CCensusPerson::mMName, CChurchPerson::mMName);}  
        bool Same FName(){return !strcmp( CCensusPerson::mFName, CChurchPerson::mFName);}  
        bool SameName() {return SameLName() && SameMName() && Same FName();}
    ...

    public:
        CDoublePerson(PSTR lcs, PSTR mcs, PSTR fcs, CLNG bcs, CCHR scs,  
                      PSTR sos, PSTR sis, PSTR occ,  
                      PSTR lch, PSTR mch, PSTR fch, CLNG bch, CCHR sch,  
                      const CChurchPerson* dad, const CChurchPerson* mom,  
                      CLNG bpt)  
            :CCensusPerson(lcs, mcs, fcs, bcs, scs, sos, sis, occ),  
                      CChurchPerson(lch, mch, fch, bch, sch, dad, mom, bpt)  
            {}  
        CDoublePerson (const CCensusPerson & ccp, const CChurchPerson & chp)  
            :CCensusPerson(ccp), CChurchPerson(chp)  
            {}  
        void WriteData() const  
            { CCensusPerson::WriteData();  
                    CChurchPerson::WriteData();  
            }  
    ...
};
```

Som det fremgår av de inlinede metodene i `CDoublePerson` for å sjekke om parvise navndeler er like, unngår vi flertydighet ved bruk av de respektive superklassenavnene sammen med tilhørighetsoperatoren `::`. 

84
19.9 Kompilatorgenererte defaultkonstruktorer

Legg merke til at CDoublePerson ikke har noen eksplisitt definert defaultkonstruktør. Som nevnt genereres denne automatisk av kompilatoren, dersom den ikke er eksplisitt definert. En kompilatorgenererte konstruktør kaller defaultkonstruktorene til alle eventuelle superklasser til objektets klasse.

En eksplisering av en slik konstruktoren for CDoublePerson kunne ha sett slik ut:

```cpp
CDoublePerson::CDoublePerson () : CCensusPerson(), CChurchPerson() {}
```

I tillegg kaller den kompilatorgenererte konstruktoren defaultkonstruktorene til alle eventuelle instanser av klasser i det aktuelle objektet. Anta at vi hadde omdøpt CDoublePerson slik at den hadde to felt tilsvarende de to superklassene.

```cpp
class CDoublePerson
{
    private:
    CCensusPerson mCsP;
    CChurchPerson mChP;
    ...
};
```

I så fall ville vi, med noen få endringer, ha kunnet beholde begge de to eksplisitte konstruktorene, og den kompilatorgenererte defaultkonstruktoren ville også utføre det samme som før.

Endringene fremgår av den mer fullstendige definisjonen.

```cpp
class CDoublePerson
{
    private:
    CCensusPerson mCsP;
    CChurchPerson mChP;
    bool SameLName(){ return !strcmp(mCsP.mLName, mChP.mLName);}
    bool SameMName(){ return !strcmp(mCsP.mMName, mChP.mMName);}
    bool SameFName(){ return !strcmp(mCsP.mFName, mChP.mFName);}
    bool SameName() { return SameLName() && SameMName() && SameFName();}
    ...
    public:
    CDoublePerson() : mCsP{}, mChP{}
    {}
    CDoublePerson(PSTR lcs, PSTR mcs, PSTR fcs, CLNG bcs, CCHR scs,
    PSTR sos, PSTR sis, PSTR occ,
    PSTR lch, PSTR mch, PSTR fch, CLNG bch, CCHR sch, 
    const CChurchPerson* dad, const CChurchPerson* mom,
    CLNG bpt)
    :mCsP(lcs, mcs, fcs, bcs, scs, sos, sis, occ),
    mChP(lch, mch, fch, bch, sch, dad, mom, bpt)
    {}
    CDoublePerson(const CCensusPerson& ccp, const CChurchPerson& chp)
    :mCsP(ccp), mChP(chp)
    {}
    void WriteData() const {mCsP.WriteData();mChP.WriteData();}
    ...
};
```

Som man ser, har vi byttet ut alle forekomster av navnene på de tidligere superklassene kominert med tilhørighetsoperatoren ::, med feltnavnene kombinert med feltselectorene.

En eksplisering av den kompilatorgenererte defaultkonstruktoren kunne ha sett slik ut:

```cpp
CDoublePerson::CDoublePerson () : mCsP(), mChP() {}
```
19.10 Klassevenner

Etter ovennevnte endring, er ikke CDoublePerson lenger en subklasse av CPerson. Denne klassen inneholder bl.a. de private feltene mName, mMName og mFName, som refererer direkte i metodene til CDoublePerson, og kompilatoren vil ikke lenger godkjenne dette. Det er flere løsninger på dette. Vi kan deklarere de aktuelle feltene i CPerson som offentlige (dette vil ikke tru klasses integritet, og anbefales ikke), vi kunne definere de nødvendige offentlige feltselektorene i CPerson og bruke disse i (en anbefalelsesverdig god stil), eller vi kunne deklarere CDoublePerson som en venn av CPerson — slik:

```cpp
class CPerson
{
    private: friend class CDoublePerson;
    ...
};
```

Det spiller ingen rolle om venneerklæringen gjøres privat eller offentlig.

NB! Vennskap er hverken gjensidig eller arvelig. At CDoublePerson er venn av CPerson, innebærer ikke at CPerson er venn av CDoublePerson; og heller ikke at CDoublePerson er venn av CCensusPerson eller CChurchPerson; og hadde CDoublePerson hatt subklasser, ville disse ha vært venner av CPerson.

En venneerklæring kan også gjøres gjeldende for en funksjon som ikke tilhører noen klasse. Dette er mest aktuelt, dersom vi ønsker å operere på to eller flere klasser samtidig, uten å blottlegge de involverte klassene for hverandre — f.eks. slik:

```cpp
class CA; class CB; class CC;

class CA
{
    int mA;
    friend int Add(const CA&, const CB&, const CC&);
};

class CB
{
    int mB;
    friend int Add(const CA&, const CB&, const CC&);
};

class CC
{
    int mC;
    friend int Add(const CA&, const CB&, const CC&);
};

int Add(const CA& a, const CB& b, const CC& c)
{
    return a.mA + b.mB + c.mC;
}
```

Den første linjen inneholder såkalte forward deklarasjoner av de tre klassene (vi kunne også si at de definerer prototypes for klassene). Klassene må være kjent før de bruke, f.eks. som når CB og CC nevnes i CA før de er definert.

Endelig kan en venneerklæringer begrenses til én eller flere av en annen klasses funksjoner. Dette innebærer imidlertid en så vidt komplisert form for sirkularitet, at vi ikke lar det ligge.
19.11 Utvidelse av operatordomener

Vi har tidligere sett på funksjonsoverloading (function overloading — utvidelse av domenent til en funksjon). Dette er noe C++ tillater uavhengig av om vi bruker klasser eller ikke.

Legg forårsig merke til at funksjonsoverloading et noe annet enn polymorf. Ved funksjonsoverloading har vi egentlig å gjøre med et overleset navn og flere ulike funksjoner som deler dette navn, men som har ulike argumentlister. Hvilken funksjon av den gruppen funksjoner som deler et overleset navn som faktisk kalles av i det ene eller andre tilfellet, bestemmes under kompileringen (statisk), på grunnlag av typene til og/eller antallet av de aktuelle parametrene. Ved polymorfi har vi derimot å gjøre med samme funksjon i ulike utgaver, og hvilken utgave som skal utføres, avgjøres først under kjøringen (dynamisk) på grunnlag av hvilken klasse det aktuelle objektet tilhører.

Overlessing av operatorer derimot, er noe vi bare kan få til i forbindelse med klasser. Selve operasjonen defineres på samme måte som en funksjon, men vi bruker en særegen notasjon i funksjons hodet. Anta vi ønsker å sammenligne to personer. Vi ønsker å vite om to objekter representerer samme person (muligens for å utelukke duplikater fra en personobjektleiste), og vi ønsker å kunne ordne personobjekter alfabetisk. Vi kan da innføre følgende overlessede operatorer i klassen CPerson.

```cpp
class CPerson {
private: ...

public: ...

  bool operator== (CPerson& sml) { return !Sammenlign(sml); }
  bool operator!= (CPerson& sml) { return Sammenlign(sml); }
  bool operator<  (CPerson& sml) { return Sammenlign(sml) < 0; }
  bool operator>  (CPerson& sml) { return Sammenlign(sml) > 0; }
  bool operator<= (CPerson& sml) { return Sammenlign(sml) <= 0; }
  bool operator>= (CPerson& sml) { return Sammenlign(sml) >= 0; }
... }
```

Som man ser har vi her definert en hjelpemetode som de ulike operator-funksjonene kan bruke. Siden disse hver for seg bare omfatter et kall på hjelpemetoden, kan vi "inline" dem i klassen.

```cpp
int CPerson::Sammenlign(CPerson& sml)
{
  int res = strcmp(mENavn, sml.mENavn); // Sammenlign etternavnene.
  if (!res)                             // Etternavnene var like, så
    res = strcmp(mMNavn, sml.mMNavn);   // sammenlign også mellomnavnene.
  if (!res)                           // Like e.navnene og m.navnene,
    res = strcmp(mMNavn, sml.mMNavn); // så sammenlign også fornavnene.
  return res;
}
```

Gitt deklarasjonene

```cpp
CPerson a("Hansen", "Østbye", "Per"); b("Nilsen", "Sørholt", "Ole");
```

kan vi sammenligne a og b vha sammenligningsoperatorene for CPerson:

```cpp
a = b  ⇒  0  a < b  ⇒  1  a > b  ⇒  0
```

Vi kan også definere operatorfunksjoner uavhengig av klassen. Dvs. en operatorfunksjon behøver ikke å være en klassemetode, men typen(e) til funksjonens argument(er) må være en definert klasse.
Her har vi flyttet sammenligningsfunksjonene ut av CPerson (og vi har her inlinet dem eksplisitt av samme grunn som vi inlinet dem implisitt i klassedefinisjonen):

```cpp
inline bool operator==(CPerson& a, CPerson& b) { return !a.Sammenling(sml); }
inline bool operator!=(CPerson& a, CPerson& b) { return a.Sammenling(sml); }
inline bool operator<(CPerson& a, CPerson& b) { return a.Sammenling(sml) < 0; }
inline bool operator>(CPerson& a, CPerson& b) { return a.Sammenling(sml) > 0; }
inline bool operator<=(CPerson& a, CPerson& b) { return a.Sammenling(sml) <= 0; }
inline bool operator>=(CPerson& a, CPerson& b) { return a.Sammenling(sml) >= 0; }
```

NB! Merk at siden CPerson::Sammenlign er private, er ingen av disse definisjonene lovlige. For å gjøre dem lovlige, må vi først deklarere metoden som public. Dette poenget illustreres enda bedre av eksemplet under.

Vi definerer sammenligningsoperatorene == og < for CAccount utenfor klassen slik:

```cpp
bool operator==(CAccount& a, CAccount& b) { return a.GetBalance() == b.GetBalance(); }
bool operator<(CAccount& a, CAccount& b) { return a.GetBalance() < b.GetBalance(); }
```

Siden CAccount::mBalance er private, må vi bruke aksessorene for disse i funksjonsdefinisjonene.

Hadde vi plassert funksjonene i klassedefinisjonen, kunne vi ha aksessert feltene direkte.

Merk at siden et kall på en inlinet aksessor forutsettingsvis skal være erstattet av kompilatoren med aksessorens kode, vil det ikke koste noe mer enn en direkte aksess.
20 Filbehandling

20.1 Et lite eksempel på lesering fra og skriving til fil implementert i C og C++

Funksjonen CopyFile i eksemplet under utfører følgende:
- åpner en fil med et gitt navn for lesering,
- åpner en fil med gitt navn for skriving,
- går i løkke så lengde det er tegn igjen å lese, og
  - leser innfilen tegn for tegn, og
  - skriver de samme tegn til utfilen
- lukker utfilen
- lukker innfilen.

La oss først se på C-utgaven av eksemplet (etter dette, lar vi Cs I/O-rutiner være):

```c
#include <stdio.h>
...
void CopyFile (const char *infilename, const char *outfilename)
{
  FILE  *infile  = fopen(infilename,  "r"),
        *outfile = fopen(outfilename, "w");
  char c;
  while ((c = fgetc(infile)) != EOF)
    fputc(c, outfile);
  fclose(outfile );
  fclose(infile );
}
```

- `fopen` tar to argumenter: et filnavn og en modus (en streng) som bl.a. kan være "r" for lesering og "w" for skriving.
- Angis "w", vil det enten opprettes en fil med det gitte navnet, hvis filen ikke fantes fra før, eller innholdet i den eksisterende filen vil bli slettet.


- `fclose` tar en peker til en `FILE` som argument og lukker den tilsvarende filen.
- `fgetc` tar en peker til en `FILE` som argument og enten
  - leser, om mulig å lese neste tegn i filen, i forhold til plasseringen av lesehodet, flytter lesehodet ett tegn fremover, og
  - returnerer det leste tegnet,
- eller returnerer verdien `EOF` (et forhåndsdefinert heltall), hvis det ikke var flere tegn igjen å lese.
- `fputc` tar et tegn og en peker til en `FILE` som argument, skriver, om mulig, det gitte tegnet til den gitte filen og
  - returnerer det skrevne tegnet eller `EOF`, hvis skrivoperasjonen ikke lyktes.

I tillegg til `fgetc` og `fputc`, finnes det funksjoner for å lese og skrive strenger, og for å lese og skrive formaterte data (tilsvarende `scanf` og `printf`).
I C++ ser eksemplet over slik ut:

```cpp
#include <fstream.h>
...
void CopyFile (const char *ifilename, const char *ofilename)
{
    ifstream ifile (ifilename);
    ofstream ofile (ofilename);
    char c;
    while (ifile.get(c), !ifile.eof())
        ofile.put(c);
}
```

*ifstream* og *ofstream* er klasser, og får konstruktorene til disse et filnavn (som her), vil de forsøke å åpne filen med det gitte navnet. *ofstream*-konstrukturen vil dessuten, hvis filen ikke finnes, først forsøke å opprette en ny fil med det gitte navnet, eller, hvis den alt finnes, å slette dens innhold.

Hvis åpningen av en fil ikke lyktes, ville et kall på *ios::fail()* ha returnert en feilkode ≠ 0 (både *ifstream* og *ofstream* er arvinger til klassen *ios*, slik at vi f.eks. kan kalle *ifil.fail()*).

Defensiv I/O-programmering, der vi hele tiden tar høyde for mulige I/O-feil, skal vi komme tilbake til.

*Den overlessede funksjonen* `ifstream::get` kan bl.a. ta et tegn som referanseoverført parameter; og får den det, leser den neste tegn i inffilen, og tilordner den refererte variabelen dette.

*Funksjonen* `ifstream::eof` returnerer 1 eller 0 avhengig av om det er lest til filslutt eller ikke.

*Funksjonen* `ofstream::put` tar et tegn som parameter; og skriver dette til filen.

*Når programmet forlater den blokken der filobjektene er deklarert (i dette tilfellet funksjonen *CopyFile*), kalles automatisk objektene desstruktører som bl.a. sørger for å lukke de åpne filene.

`CopyFile` kan kalles slik.

```cpp
CopyFile("test_in.txt", "test_out.txt");
```

Verdiene til filnavnene kan også leses inn fra bruker— noe vi skal komme tilbake til.

### 20.3 Noen filbehandlingsrutiner og -algoritmer i C++

Eksemplet over gir en oppskrift på lesing og skrivning som i og for seg er tilstrekkelig til å lese og skrive alle slags filer, uansett innhold og format. I den grad vi ønsker å kontrollere innndata mht. type og format, må vi sjekke de enkelte relevante tegnene etter hvert som de leses, og i den grad vi ønsker å formattere utdata, skriver vi ut blanke, tabulatører og linjeskift på passende steder. De grunnleggende I/O-operasjonene forblir imidlertid de samme:

- åpne filen, evt. etter at den er opprettet, implisitt i deklarasjonen av filobjektet, eller eksplicitt (se under)
- les og/eller skriv tegn for tegn, mens vi evt. sjekker for filslutt
- lukk filen, implisitt ved destruksjon av filobjektet, eller eksplicitt (se under).

Det finnes imidlertid rutiner som tar seg av typiske operasjoner som er mer komplekse enn dette. Her følger noen eksempler:
Her åpner og lukker vi filene eksplisitt. Motivasjonen for å gjøre dette er ikke tilstede i dette korte eksemplet, men det kunne f.eks. være aktuelt om vi bruker samme filobjekt for å skrive eller lese flere filer etter hverandre.

```cpp
#include <fstream.h>
...
void CopyFile (char *ifilename, char *ofilename)
{
    ifstream ifile;
    ofstream ofile;
    ifile.open(ifilename);
    ofile.open(ofilename);
    ...
    ifile.close();
    ofile.close();
}
```

Her leser og skriver vi linje for linje.

- `ifstream::getline` tar følgende 3 parametre:
  - strengen det skal leses til,
  - antall plasser i strengen, terminasjonsbytet medregnet, og
  - tegnet for linjeslutt som stilltiende settes til `\n` (newline).
- `ofstream::write` tar følgende 2 parametre:
  - strengen som skal skrives, og
  - lengden til denne strengen.
- `ofstream::put` kalles for å få skrevet ut linjeskift.

```cpp
#include <string.h>  // vi bruker strlen
#include <fstream.h>
...
const short maxLine = 1024;
void WriteLine(ofstream& ofile, char * line)
{
    ofile.write(line, strlen(line));
    ofile.put('\n');
}
void CopyFile (const char *ifilename, const char *ofilename)
{
    ifstream ifile(ifilename);
    ofstream ofile(ofilename);
    char line[maxLine];
    while (!ifile.eof())
    {
        ifile.getline(line, maxLine);
        WriteLine(ofile, line);
    }
}
```
Nå kunne vi utmerket godt ha brukt utskriftsoperatoren `<<`. Denne virker like godt for disk-filer som for `cout`:

```c++
while (!ifile.eof())
{
    ifile.getline(line, maxLine);
    ofile << line << endl;
}
```

Vi kunne imidlertid ikke ha brukt innlesingsoperatoren `>>` med tilsvarende hell.

Anta vi har en fil med følgende innhold:

Dette er en testfil med data fordelt på noen få linjer.
Dette er tredje linje, og dette er fjerde og siste linje.

Prøver vi å bruke innlesingsoperatoren — f.eks. slik:

```c++
while (!ifile.eof())
{
    ifile >> line;
    ifile.get();
    ofile << line << endl;
}
```

får vi følgende utfil:

Dette er en testfil med data fordelt på noen få linjer.
Dette er tredje linje, og dette er fjerde og siste linje.

Generelt leser `>>` en eller annen `enhet`, frem til første "hvite" tegn (linjeskift, tabulator eller mellomrom).

Er den maskimale linjelengden (her gitt ved konstanten `maxLine`) mindre enn lengden til den lengste linjen i filen, deles linjene opp, men vi mister ingen data. I eksempelteksten har den lengste linjen 31 tegn, og for å få med terminasjonbytet for denne, må den maksimale linjelengden være 32. Setter vi denne til f.eks. 24, får vi følgende utfil:

Dette er en testfil med data fordelt på noen få linjer.
Dette er tredje linje, og dette er fjerde og siste linje.

En maks linjelengde på f.eks. 1024 dekker mange, men ikke nødvendigvis alle linjene i f.eks. en tekst med harde linjeskift som avsnitsskiler og ellers med myke lineskift. Vi kan gjerne øke linjelengden, men ikke ubegrenset, siden linjebufferen må få plass på stacken. Alternativt kunne vi ha allokert plass til bufferen på heapen.

Ønsker vi uansett å begrense linjelengden, og å trunkere linjer som er for lange, kan vi bruke følgende algoritme:
```c
void ReadLine(ifstream& ifile, char * line)
{
    ifile.get(line, maxLine);
    char c;
    while (c = ifile.get(), c != '\n' && c != EOF)
    {
    }
}
```

Som nevnt er ifstream::get overlesset. Én variant tar de samme argumentene som ifstream::getline (se over) — men mens getline leser til of med linjeskift (eller det gitte stopptegnet) leser get bare frem til linjeskift.

For å oppnå samme effekt som med getline, kunne vi ha nøyd oss med følgende:

I while-løkken leses det minst ett tegn, eller så mange som nødvendig til linjeslutt eller filslutt er lest.

Løkken kan være vanskelig å lese. Den inneholder en saksens (med operatoren ,) der første ledd er en tilordning, og andre ledd er en test. I tilordningen får c verdien til det tegnet get() leser og returnerer. I testen kontrolleres det at c hvertken er linjeslutt (\'\n\') eller filslutt (EOF). Siden lesingen utføres i kontrollsettningen, er selve utførelsessetningen i løkken tom. Følgende ekvivalente konstruksjon er kanskje lettere å lese.

```c
    c = ifile.get();
    while (c != '\n' && c != EOF)
    {
        c = ifile.get();
    }
```

Prøver vi denne konstruksjonen med linjelengde 24, får vi følgende utfil:

Dette er en testfil med fordelt på noen få linj
Dette er tredje linje,
rette er fjerde og sist

20.4 Direktefiler i C++
Så langt har vi bare sett på sekvensielle filoperasjoner. Det er imidlertid å lese eller skrive direkte til gitte posisjoner i en fil, uten å gå gjennom sekvensielle posisjoner. Dette kalles direkte eller vilkårlig (random) filaksess.

Vi snakker i den forbindelse om den løpende filposisjonen.

- Første tegn i filen har posisjon 0, og siste tegn i en fil med n tegn har posisjonen n - 1.
- For en innfil kan vi si at et kall på get() vil returnere tegnet i denne posisjonen, hvis posisjonen er innenfor filen.
- For en utfil kan vi si at et kall på put(c) vil skrive tegnet c til den løpende posisjonen i filen.

```c
istream::tellg() returnerer den løpende filposisjonen i den aktuelle innfilen.
istream::seekg(pos) setter den løpende filposisjonen til det gitte heltallet (pos) i den aktuelle innfilen.
ostream::tellp() returnerer den løpende filposisjonen i den aktuelle utfilen.
ostream::seekp(pos) setter den løpende filposisjonen i den aktuelle utfilen til det gitte heltallet (pos).
```
20.5 Behandling av I/O-feil i C++

Det er ikke mulig å utføre I/O-operasjoner på en realistisk måte uten å ta høyde for feil.

- Vi har allerede sett eksempler på én uangåelig, men også helt akseptabel feil som vi må forholde oss til mens vi leser en fil — nemlig at det kan være slutt på filen.
- Anta vi leser fra en fjern disk, og forbindelsen til denne brytes, eller at vi skriver til en diskett, og denne blir full. Dette er også feil vi bør kunne forholde oss til uten å la programmet bryte sammen. Slike feil er selvsagt ødeleggende for den pågående I/O-operasjonen, men ikke nødvendigvis for programkjøringen i sin helhet.
- Anta vi prøver å åpne en fil for lesing som ikke finnes, eller prøver å åpne for skriving, eller omdøpe eller slette en eksisterende fil som er beskyttet mot endring. I slike tilfeller må vi la programmet enten gi beskjed om at operasjonen mislyktes, eller foreta de nødvendige nedgraderinger for at operasjonen skal kunne lykkes.
- Anta at vi forventer at data i den filen vi leser fra, har bestemte formatter, er av bestemte typer, og/eller ligger innenfor bestemte grenseverdier. Får vi inn data som ikke tilfredsstiller disse forventningene, bør programmet kunne håndtere dette og gi passende feilmeldinger — evt. til en logg.

Alt i alt må skikkelig I/O-programmering være defensiv.

ios er den øverste filklassen og forgjenger til alle andre filklasser. Den har bl.a. følgende feilbehandlingsfunksjoner:

- **bad** ret. *true* hvis en terminal feil har oppstått
- **fail** ret. *true* hvis en terminal feil har oppstått, eller en akseptabel feil så som at filen ikke ble funnet.
- **good** ret. *true* hvis ingen feil har oppstått, og filpos. er innenfor filen (dvs. at det ikke er slutt på filen).
- **eof** ret. *true* hvis filposisjonen er bak siste tegn i filen, dvs. hvis det er slutt på filen.
- **rdstate** ret. feilstatusinfo. om filen — en *int* som kan testes i forhold til gitte flagg.

I tillegg kan vi teste verdien til det aktuelle filobjektet direkte —

- slik at
  
  ```
  if (!file) ...;    er ekvivalent med
  if (file.fail())...;
  ```
- og
  
  ```
  if (file) ...;    er ekvivalent med
  if (file.good())...;
  ```

Vi kan nå bruke dette i kopieringseksemplet

```cpp
bool CopyFile (const char *ifilename, const har *ofilename)  
{    ifstream ifile(ifilename);
    ofstream ofile(ofilename);
    char c;
    while (ofile && ifile && ifile.get(c), !ifile.eof())
        ofile.put(c);
    return ifile && ofile;
}
```

Hvis en av filene ikke ble skikkelig initialisert, får vi ingen runder i løkken

Hvis det oppstår en feil under lesingen eller skrivingen, avbrytes løkken før innfilen er lest ferdig, og funksjonen returnerer **false**.

Ellers returnerer funksjonen **true**.

Returnerer funksjonen **false**, kan vi sjekke nærermere hva feilen består i og rapportere dette.
20.6 Klassehierarkiet for filbehandling i C++

Begrenser vi oss til fil-I/O, kan vi forenkle bildet:

ios

<table>
<thead>
<tr>
<th>istream</th>
<th>ostream</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifstream</td>
<td>istream_withassign</td>
</tr>
<tr>
<td>iostream</td>
<td></td>
</tr>
<tr>
<td>fstream</td>
<td>stdiostream</td>
</tr>
<tr>
<td>streambuf</td>
<td></td>
</tr>
<tr>
<td>filebuf</td>
<td>stdiobuf</td>
</tr>
<tr>
<td>iostream_init</td>
<td></td>
</tr>
</tbody>
</table>

- ios har metoder for bl.a. formattering av input og output og for kontroll av filens status under lesing/skriving, og dessuten en rekke typer og konstante verdier som brukes av de andre klassene.
- istream og ostream har metoder for hhv. å lese og skrive data.
- ifstream og ofstream har metoder for bl.a. å åpne og lukke filer.
- iostream utgjør først og fremst bindeleddet mellom det øverste klassehierarkiet og streambuf-klassen.
- fstream utvider funksjonaliteten til metodene for lesing, skriving, åpning og lukking til å omfatte data-bufring.
- streambuf holder orden på databufferen.
- filebuf utvider funksjonaliteten til metodene til streambuf til å omfatte disk-fil-I/O.

De øvrige klassene i det fulle hierarkiet brukes for
- lesing fra tastaturet og skriving til skjermen: istream_withassign, ostream_withassign, stdiostream, stdiobuf, iostream_init, istrstream, ostrstream, strstream, strstreambuf.
- og lesing fra og skriving til strenger:

objektene cin og cout er definert i headerfilen <iostream.h>. Første gang programmet finner en referanse til et av disse objektene, kaller det konstruktoren til iostream_init
20.6 De mest brukte I/O-funksjoner, operatorer og manipulatorer i C++

NB! Denne oversikten er ikke fullstendig.

**istream**

int peek()

Retunerer neste tegn klar til å leses.

int get();

Leler et tegn og returnerer dette.

**istream& get(char& rChr)**

Leser et tegn og sender det tilbake i det tegnet rChr refererer til.

**istream& get(char* pStr, int count, char delim)**

Leser inntil count tegn eller til stopptegnet delim og plasserer de leste tegnene i den strengen pStr peker på. Stopp-tegnet leses ikke, selv om det påtreffes før count tegn er lest.

**istream& getline(char* pStr, int count, char delim)**

Gjør det samme som siste variant av get og leser dessuten stopptegnet, hvis det påtreffes.

**istream& read(char* pStr, int count, char delim)**

Leser count og plasserer de leste tegnene i den strengen pStr peker på.

**istream& ignore(int count, char delim)**

Gjør det samme som getline, bortsett fra at de leste tegnene ikke lagres.

void eatwhite()

Leler frem til neste ikke-hvite tegn (hvite tegn er linjeskift, tabualtor og mellomrom).

int gcount() const

Retunerer antall tegn lest av siste kall på get, getline, read, ignore eller eatwhite.

**istream& seekg(long offset, ios::seek_dir rel_to = ios::beg)**

Setter den løpende filposisjonen relativt til rel_to som er enten ios::beg (default), ios::cur eller ios::end — filstart, løpende posisjon eller filslutt.

long tellg()

Retunerer løpende filposisjon.

**istream& operator>>(char* pStr)**

Innlesingsoperator for strenger. Skipper evt. hvite tegn foran strengen og leser strengen til neste hvite tegn.

**istream& operator>>(num_type& ref)**

Innlesingsoperator for ulike talltyper. Skipper evt. hvite tegn foran tallet og leser tallet til neste hvite tegn.

**ostream**

**ostream& put(char c)**

Skriver ut tegnet c.

**ostream& write(const char* pStr, int count)**

Skriver ut count tegn fra og med det tegnet pStr peker på (mer at vi må selv holde rede på strenglengden, hvis det er en streng som skal skrives ut).
osstream& flush()  
Tømmer filbufferen (merk at først etter et kall på flush, kan vi være sikre på at alle tegn som er skrevet ut vha. put og/eller write, blir skrevet til selve disken.

osstream& seekp(long offset, ios::seek_dir rel_to = ios::beg)  
Tilsvarer seekg (g og p står for hhv. get og put).

long tellp()  
Tilsvarer tellg.

osstream& operator<<(char* pStr)  
Utskriftsoperator for nullterminerte strenger.

osstream& operator<<(num_type val)  
Utskriftsoperator for ulike talltyper.

**manupilatorer** for plassering etter utskriftsoperatoren (for å forenkle utskriften)

osstream& endl  
Skriver ut linjeskift eller tømmer bufferen. Tilsvarer kallene ofile.put(‘\n’);ofile.flush();.

osstream& ends  
Skriver ut strengtermineringstegnet nul. Tilsvarer kallet ofile.put(‘\0’);.

osstream& flush  
Tømmer buferen.Tilsvarer kallet ofile.flush();.

**fstream**

void open(char* pName, int mode, int prot = ios::openprot)  
Åpner en fysisk fil og knytter den til filobjektet i programmet.

pName forutsettes å peke på en streng med et gyldig filnavn.

mode er i den nedarvede utgaven til ifstream og ofstream stilltiende hhv. ios::in og ios::out.

prot angir beskyttelsen (vi bruker alltid deafult-verdien).

void close()  
Lukker filen, om den er åpen, etter evt. å ha tømt utskriftsbufferen, hvis filen er en utfil.

### 20.7 Innlesing av filnavn fra bruker

Under utviklingen av et program vil vi ofte operere med konstante filnavn skrevet rett inni koden, og holde kontroll på at innfilnavn svarer til eksisterende filer, og at utfilnavn ikke svarer til filer vi ikke ønsker å slette.

Ønsker vi å lese filnavn fra bruker, kan dette gjøres i konsollvinduet, på samme måte som vi leser inn verdier til andre variabler, men vi har da ingen kontroll på om bruker faktisk oppgir en eksisterende fil (som enten skal leses eller kan-ske ikke skal slettes). Vi har heller ingen kontroll på hvor filen er plassert, og må overlate til bruker å oppgi veien frem til filen.

Det er langt å foretrekke å bruke en av de såkalte *Common Dialogs* i Windows — GetOpenFileName for å lese inn navn på en fil som skal åpnes, og GetSaveFileName for å lese inn navnet på en ny fil som skal arkiveres.
Følgende lille program viser hvorledes dialogene kan brukes.

```c
#include <windows.h>
#define MAXPATH 256
#define MAXTITLE 64

bool FileNameDialog(char* path, char* title)
{
    OPENFILENAME ofn;       // Parameters for file dialog
    ZeroMemory(&ofn, sizeof(ofn)); // Set all fields to zero/NULL
    ofn.lStructSize = sizeof(ofn); // Size of data structure
    ofn.Flags = OFN_PATHMUSTEXIST | // Verify path and filename on open
               OFN_FILEMUSTEXIST |
               OFN_OVERWRITEPROMPT; // Prompt if file already exists on save
    ofn.lpstrFilter =
        "Alle\0 *.\00Tekst\0*.TXT\0"; // A string of 4 zero-term.strings
    ofn.nFilterIndex = 1; // Select first filter in string
    ofn.lpstrFile = path; // Pointer to path buffer
    ofn.nMaxFile = MAXPATH; // Length of path buffer
    ofn.lpstrFileTitle = title; // Pointer to path buffer
    ofn.nMaxFileTitle = MAXTITLE; // Length of path buffer
    return *title ? GetSaveFileName(&ofn) : GetOpenFileName(&ofn);
}

void main()
{
    char path [MAXPATH] = "", title[MAXTITLE] = "";
    FileNameDialog(path, title);
    FileNameDialog(path, title);
}
```

Headerfilen <windows.h> må inkluderes for å få med definisjonene av `struct`-et `OPENFILENAME` og prototypene til `ZeroMemory`, `GetOpenFileName` og `GetSaveFileName`.

`OPENFILENAME` kan beskrives som en `parameterblokk` — en samling av de parametrene filnavnrutinene trenger. Langt de fleste er uten interesse i vår endefremme bruk av dialogene.

`ZeroMemory` nuller ut et gitt antall bytes i hukommelsen f.o.m. en gitt adresse — i dette tilfellet nuller funksjonen ut den hukommelsen parameterblokken `ofn` opptrer.

- Alle `struct`-er definert i Windows starter med et felt som skal initialiseres med størrelsen til `struct`-et.
- `Flags` er et heltall der man kan sette flagg som modifiserer dialogens gang.
  - Er flagget `OFN_PATHMUSTEXIST` satt, sjekker `GetOpenFileName` om en eventuell filvei eksisterer.
  - Er flagget `OFN_FILEMUSTEXIST` satt, sjekker `GetOpenFileName` om et eventuelt gitt filnavn eksisterer.
  - Er flagget `OFN_OVERWRITEPROMPT` satt, sjekker `GetSaveFileName` om et eventuelt gitt filnavn eksisterer, og spør i så fall bruker om den eksisterende filen skal overskrives.
- `lpstrFilter` skal være en streng med parvise strenger. Første ledd i et par skal gi en beskrivelse av en filtype somvises bruker, mens andre ledd skal inneholde det tilsvarende navnmønstret (der * står for "hva som helst").
  `GetOpenFileName` bruker disse strengene til å filtrere de filene som vises i dialogens filvindu. Her angis "Alle" med mønstret "*." for "Alle typer filer" og "Tekst" med mønstret "*.TXT" for tekstfiler.
• nFilterIndex, som starter på 1, angir hvilken filtype som velges i utgangspunktet (i dette tilfelle "Alle").
• lpstrFile skal peke på en streng som den kallende funksjonen må tilveiebringe (her er denne deklarert i main, og sendes derfra til FileNameDialog. Når fildialogen avsluttes med OK, plasseres det fulle filnavnet (vei + navn) der.
• nMaxFile skal angi max-lengden til den fulle filnavnstrengen — termineringsbytet inkludert.
• lpstrFileTitle skal peke på en streng som den kallende funksjonen må tilveiebringe (her er denne deklarert i main, og sendes derfra til FileNameDialog. Når fildialogen avsluttes med OK, plasseres filnavnet alene der.
• nMaxFileTitle skal angi max-lengden til filnavnstrengen — termineringsbytet inkludert.
De to siste feltene er ikke strengt nødvendige. Vi bruker dem her for å kunne kontrollere om det skal spørres etter navnet på en fil som skal lagres eller på en fil som skal åpnes. I første fall, sender vi med et forslag til filnavn i strengen title, slik at denn ikke er tom — og dette er også kriteriet for at GetSaveFileName skal kalles. I siste fall lar vi title være tom, med det resultat at GetOpenFileName kalles.
• Kjører vi dette programmet, vil vi først få opp en dialog for valg av eksisterende fil. Velger vi en fil, og klikker OK, vil vi dernest få opp en dialog for angivelse av navn på fil som skal lagres. (Programmet kan prøves helt uforpliktene. Det verken åpner, eller sletter filer, selv om dialogen skulle gi det inntrykket.)
21 Mer om tegnstrenger

21.1 Strenger i Scheme, Pascal og C

Som nevnt, er en tegnstreng i C en *vektor med tegn* — f.eks.: `char str[200];`
Strengelementene burde vært *unsigned char*, men dessevere er det ikke slik. Dette betyr at vi må passe ekstra godt på når vi arbeider med strenger som kan inneholde tegn fra den øvre delen av ASCII-tabellen — i spørringen 128..255.

I *Scheme* har en streng mange likhetstrekk med en vektor av tegn, men det er definert egne primitiver for operasjoner på strenger. En streng allokeres med en fast lengde, og skulle vi ønske å håndtere én og samme streng mht. at den *kunne* inneholde et variert antall signifikante tegn, måtte vi definere egne selektorer og konstruktøer.

I *Pascal* er strengbegrep en del av selve språket. En streng er en *pakket vektor med tegn* indeksert fra 0 til maksimum 255, der 0-te tegn angir strengens lengde. Uavhengig av hvor mange tegn det faktisk er satt av plass til, er spørringen for lengden til en streng $0.2^7 - 1 = 0..255$ (så mange verdier som kan representeres med etpakket tegn, dvs. ett byte).

21.2 Standard strengoperasjoner

I standardbiblioteket `<string.h>` er det definert en rekke funksjoner som opererer på strenger ut fra den forutsetning at en streng alltid avsluttes med tegnet *NUL* (`0`, `'\0'`). Dette betyr:
- En C-streng kan ikke inneholde *NUL* (men selvsagt sifferet `'0'` som har representasjonen `48`).
- Det er ingen øvre grense på hvor lang en C-streng kan være (bortsett fra hvor mange tegn det er satt av plass til i definisjonen av strengen, minus den plassen terminasjons-bytet 0 skal ha).

Initielt inneholder en nydeklarert tekst bare grums.

*<string.h>* inneholder bl.a. følgende funksjoner:

```c
char *strcpy(char *dest, const char *source) // («string copy»)
kopierer innholdet i *source til *dest.

char *strcat(char *dest, const char *source) // («string concatenate»)
legg innholdet i *source inn i *dest etter *dest's opprinnelige innhold.

int strcmp(const char *s1, const char *s2) // («string compare»)
sammenligner *s1 med *s2, og returnerer -1, hvis *s1 er alfabetisk mindre enn *s2, 0 hvis *s1 er alfabetisk lik *s2, og 1 hvis *s1 er alfabetisk større enn *s2,

char *strchr(const char *s, const char *c) // («string char»)
returnerer en peker til den første forekomsten av *c i s, eller NULL, hvis *c ikke finnes i s.

char *strrchr(const char *s, const char *c) // («string rear char»)
returnerer en peker til den siste forekomsten av *c i s, eller NULL, hvis *c ikke finnes i s.

char *strstr(const char *s1, const char *s2) // («string stringr»)
returnerer en peker til den første forekomsten av *s2 i *s1, eller NULL, hvis *s2 ikke finnes i *s1.

char *strlen(const char *s) // («string length»)
returnerer lengden til den gitte strengen.
```
For konstante strengargumenter (spesifisert som **const**) kan vi sende både strengvariabler og strengkonstanter som aktuelle parametre.

Eksempel:
```cpp
#include <iostream.h>
#include <string.h>

void main() {
    char s1[20], s2[200];
    strcpy(s1, "Hallo!");
    cout << "Teksten " << s1 << " er på " << strlen(s1) << " tegn" << endl;
    strcpy(s2, s1);
    cout << "Teksten " << s2 << " er på " << strlen(s2) << " tegn" << endl;
}
```
Programmet skriver følgende:
Teksten "Hallo!" er på 6 tegn.
Teksten "Hallo!" er på 6 tegn.

### 21.3 Egendefinerte strengoperasjoner

Siden tekster er representert som vektorer kan vi enkelt lage våre egne tekstoparasjoner.

```cpp
#include <iostream.h>
#include <string.h>

char forstetegn(const char s[]) {
    return s[0];
}

char sistetegn(const char s[]) {
    return s[strlen(s)-1];
}

void kun_store(char s[]) {
    int i = 0;
    for (i = 0; s[i]; i++)
        if (\'a\' <= s[i] && s[i] <= \'z\' ||
            \'À\' <= s[i] && s[i] <='ÿ')
            s[i] -= 32;
}

void main() {
    char s1[200], s2[200];
    strcpy(s1, "Hallo!");
    cout<<"Første tegn i \""<<s1<<\"\" er '"<<forstetegn(s1)<<\"'."<<endl;
    cout<<"Siste tegn i \""<<s1<<\"\" er '"<<sistetegn(s1)<<\"'."<<endl;
    strcpy(s2, "Med ærbødig hilsen,"); kun_store(s2);
    cout<<"Kun store bokstaver i \""<<s2<<\"\"."<<endl;
}
```
Programmet skriver følgende:
Første tegn i "Hallo!" er 'H'.
Siste tegn i "Hallo!" er 'l'.
Kun store bokstaver i "MED ÆRBØDIG HILSEN,".

Legg merke til at vi i angivelsen av en vektor-parameter enten kan plassere en stjerne foran eller en tom skarp klamme etter parameteravinet. —

Eks: (**char s[]**) = (**char *s**). — Se forøvrig [Pekere og vektorer](https://example.com).

102
22 Cs preprosessor

Preprosessordirektiver

Før selve kompileringen går C-kompiatoren gjennom koden med en preprosessor som bl.a.

- henter inn filer angitt med `#include`-direktivet,
- ekspanderer makro-definisjoner angitt med `#define`-direktivet, og
- utelater eller inkluderer kode som er underlagt betinget kompilering angitt med et av direktivene `#if`, `#ifdef`, `#ifndef` eller `#else` fulgt av `#endif`.

Et preprosessordirektiv skal alltid stå først på linjen.

Inkludering av filer

Navnet på en fil som skal inkluderes settes enten i spisse klammer eller i doble anførselstegn.

```c
#include "incl.h"
#include <stdio.h>
```

Er navnet gitt i spisse klammer, leter preprosessoren etter filen blant standardbibliotekene.

Ekspanding av makroer

En makrodefinisjon omfatter alltid *definiendum*, dvs. det makronavnet som defineres, og kan inneholde *definiens*, dvs. det uttrykket som definerer makroen.

En definisjon der definiendum er utelatt, er strengt tatt ingen definisjon, og angr heller ingen makro. Kategoriseringen av dette som en makrodefinisjon er en ren syntaktisk bekvemmelighet.

```c
#define MAX_PASCAL 256 // max length of a pascal string - 0’th byte included
```

Konnvensjonelt skrives makronavn med store bokstaver.

Benytter man makroer med parametre, bør disse settes i parenteser. Likeledes, hvis definiens er et uttrykk med flere symboler, bør det stå parenteser rundt hele uttrykket.

```c
#define MIN(x, y) ((x) < (y) ? (x) : (y))
```

Ønsker vi f.eks. å signalisere om programmet skal inneholde kode for debugging eller ikke, kan vi definere termen `DEBUG` til 1 eller 0 — slik: `#define DEBUG 1` eller `#define DEBUG 0`, og vi spør `#if DEBUG`. Eller vi kan rett og slett ta med definisjonen av `DEBUG` — slik: `#define DEBUG` eller utelukke den, og vi spør `#ifdef DEBUG`.

Betinget kompilering

<table>
<thead>
<tr>
<th>direktiv</th>
<th>betydning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#if</code> uttr</td>
<td>Er uttr ≠ 0, tas etterfølgende linjer frem til <code>#else</code> eller <code>#endif</code> med.</td>
</tr>
<tr>
<td><code>#ifdef</code> sym</td>
<td>Er sym definert, tas etterfølgende linjer frem til <code>#else</code> eller <code>#endif</code> med.</td>
</tr>
<tr>
<td><code>#ifndef</code> sym</td>
<td>Er ikke sym definert, tas etterfølgende linjer frem til <code>#else</code> eller <code>#endif</code> med.</td>
</tr>
<tr>
<td><code>#else</code></td>
<td>Alternativt til <code>#if / #ifdef / #ifndef</code></td>
</tr>
<tr>
<td><code>#endif</code></td>
<td>Avslutter <code>#if / #ifdef / #ifndef</code></td>
</tr>
</tbody>
</table>

Eksempel:

```c
#define DEBUG
...
#define DEBUG
    cout << "Crashed at " << ... << endl; DisplayData(...);
#endif
```