Videregående programmering i C

Innhold

- Funksjoner
- Pekere
- Arrays / tabeller
- Strenghåndtering
- Filhåndtering
- Kommandolinjeargumenter
- Strukturer
- Systemprogrammering i C under Linux

Funksjoner

Funksjon: “Byggeklossen” i C

- C har ikke klasser og objekter som i Java
- I stedet deles programmet opp i funksjoner:
  - En samling kode som hører sammen / gjenbrukes
  - Kan kalles fra andre funksjoner
  - Kan returnere en verdi av en bestemt datatype
  - Kan ha parametre av ulike typer
  - Tilsvarer statiske metoder i Java

Funksjonsprototyper *

- Før en funksjon kan brukes, må enten:
  - Hele funksjonen være beskrevet, eller:
  - Funksjonens *prototype* være beskrevet
- Funksjonsprototypen angir:
  - Funksjonens navn
  - Datatypen for verdien som returneres
  - Parametre: Antall, rekkefølge, datatyper
- Eksempel: `sum.c`

*: Tilsvarer en metodes *signatur* i Java

Standardbibliotekene i C

- C tilbyr mange ferdig programmerte funksjoner
- Funksjonene er er organisert i *bibliotek* for bl.a.
  - Input/output
  - Strenghåndtering
  - Matematiske beregninger
  - Systemprogrammering
- Hvert bibliotek har en *headerfil* med prototyper
- Oversikt: C Standard Library header files

Bruk av standardbibliotekene

- For å benytte en biblioteksfunksjon, må tilhørende headerfil inkluderes i toppen av programmet med preprosessordirektivet `#include`
- For å inkludere headerfiler fra standarbibliotekene brukes `<` og `>` rundt navnet til headerfilen
- For å inkludere egendefinerte headerfiler benyttes anførselstegn rundt headerfilens navn
- Eksempel: `andregradsligning.c` *

*: Må kompileres med opsjonen `-lm` for å linke inn ferdig kompilerte matematikkfunksjoner

Programmer satt sammen av flere filer

- Vanlig å legge funksjoner som hører sammen i en egen fil – en modul – som kan gjenbrukes
- En modul består av en *headerfil* med prototypene, og en *kodefil* med fullstendig kode for alle funksjonene
- Headerfilen beskriver *grensesnittet* til en modul
- Moduler kan separatkompileres og senere lenkes sammen med annen kode til fullstendige programmer
- Kun én av filene som utgjør et helt program kan inneholde funksjonen `main`

Eksempel: Geometriske figurer

- Modulen `figurer.c` er et “tegnebibliotek” med funksjoner som tegner enkle geometriske figurer
- Funksjonsprototypene ligger på headerfilen `figurer.h`
- Headerfilen inkluderes både i modulens kildefil og i hovedprogrammet `tegne.c` som bruker biblioteket
- Merk at preprosessordirektivet `#ifndef` brukes i `figurer.h` for å unngå at samme headerfil inkluderes flere ganger i samme sammensatte kode
- Kompilering: `cc -o tegne figurer.c tegne.c`

Parameteroverføring og returverdier

- Vanlige parametre til C-funksjoner *verdioverføres*:
  - Det opprettes lokale variable i funksjonen
  - Får samme verdi som variablene i funksjonskallet
  - Eksempel: sum.c
- En funksjon kan *returnere* bare én verdi
- For å returnere flere verdier / endre på variable:
  - Funksjonen må kjenne minneadressen til parameter
  - Gjøres i C ved å bruke parametre som er *pekere*

Pekere

Pointers are a very powerful, but primitive facility contained in the C language.

Pointers are a throwback to the days of low-level assembly language programming and as a result they are sometimes difficult to understand and subject to subtle and difficult-to-find errors.

Still it has to be admitted that pointers are one of the great attractions of the C language and there will be many an experienced C programmer spluttering and fuming at the idea that we would dare to refer to pointers as 'primitive'!

In an ideal world we would avoid telling you about pointers until the very last minute, but without them many of the simpler aspects of C just don't make any sense at all.

So, with apologies, let's get on with pointers.

– fra innføringskurs i C, University of Leicester, England

Variabler og minneadresser

- En variabel er et *navngitt* område i minnet
- F.eks. setter vi av plass i minnet til et heltall med:

```
int x;
```

- Vi kan senere refere til dette heltallet med navnet `x`, og bl.a. lagre data i det reserverte minneområdet:

```
x = 10;
```

- “Bak kulissene” bruker systemet et “minnekart” der (den virtuelle) *minneadressen* til `x` ligger lagret

Pekervariable

- En peker i C er en *variabel* som kan lagre en *adresse* i (det virtuelle) minnet
- Alle pekere må ha en *datatype*, som forteller hvor mange bytes med data som ligger lagret på adressen
- En peker lagrer minneadressen til en variabel av samme type – den *peker* til variabelen
- Pekere deklareres med en `*` foran variabelnavnet:

```
int *p;      /* peker til heltall */
int *q, i;   /* peker og vanlig heltall */
```

Sette verdien til en pekervariabel

- En peker kan få verdi ved å bruke *adresseoperatoren* `&` :

```
int *p, i=0;
p = &i;       /* p tilordnes adressen til i */
```

- Operatoren `&` returnerer *minneadressen* til variabelen

- Pekere kan også få verdien til pekere av samme type:

```
int i = 0;
int *p1, *p2;
p1=&i;          /* p1 tilordnes adressen til i */
p2=p1;          /* p2 tilordnes verdien til p1 *
                 * (begge vil nå peke til i)   */
```

Dereferering av pekervariable

- Bruk av *dereferanseoperatoren* `*` :
  - Setter `*` foran pekeren for å *dereferere* en peker
  - Henter ut *verdien* til variabelen den peker på
- Eksempel:

```
int *p, i=0, j;
p = &i;
j = *p;
```

- De to tilordningene ovenfor er det samme som:

```
j = i;
```

- Eksempel: `peker_1.c`

Operatoren `*` i C

- Effekten av operatoren `*` er *kontekstavhengig*
- Multiplikasjonsoperator hvis det er operander både på høyre og venstre side:

```
int m, n = 4;
m = n * 3;
```

- I deklarasjon brukes `*` til å opprette pekere:

```
int *p;
```

- Med én operand brukes `*` til å dereferere peker:

```
i = *p;
```

Indirekte tilordning med pekere

- En peker kan brukes til å gjøre en *indirekte* tilordning til variabelen den peker på:

```
int i = 0;
int *p = &i;    /* p peker på i */
*p = 15;        /* Indirekte tilordning,
                   i får verdien 15 */
```

- Brukes i funksjoner som skal forandre på verdien av parametere (returnere flere enn én verdi)

Swap-funksjon som ikke virker

- Funksjon som skal bytte innholdet i to variable:

```
void swap(int x, int y)
{
    int tmp = x;
     x = y;
     y = tmp;
}
```

- Virker ikke fordi det opprettes lokale variable i funksjonen som bare er kopier av parametrene
- Testprogram: `swap_feil.c`

Swap-funksjon som virker

- Må i stedet bruke *pekere* til variablene som skal endres som parametere:

```
void swap(int *x, int *y)
{
   int tmp = *x;
   *x = *y;
   *y = tmp;
}
```

- Må bruke adresseoperator i kall: `swap(&a, &b);`
- Testprogram: `swap_riktig.c`

Arrays / tabeller

Deklarasjon og initialisering

- Deklarasjon av array:

```
int A[10];
```

  - Setter av plass til 10 heltall, som ligger i en sammenhengende del av (det virtuelle) minnet
  - Variabelen `A` blir en *peker* av typen `int *` til det *første* av disse 10 heltallene

- Deklarasjon med initialisering:

```
int A[] ={1,2,3,4,5,6,7,8,9,10};
```

- Eksempler: `array_1.c` `array_2.c`

Arrayer i C og Java

- Syntaksen er nesten identisk i C og Java
- Men det er flere viktige forskjeller:
  - Arraygrenser kontrolleres ikke i C
  - Indeksering utenfor array i C er “tillatt”, men kan gi feil og uforutsigbare resultater
  - En arrayvariabel i C er bare en peker til tabellens første element, ikke et objekt som i Java
  - En array i C kjenner ikke til sin egen lengde
- Eksempel: `indeks_feil.c`

Array som parameter til funksjon

- En arrayvariabel er bare en peker til første element i arrayen
- Når en funksjon har en array som parameter, mottar den bare denne pekeren til første element
- For å overføre array som parameter i et funksjonskall, brukes bare *navnet* til arrayen
- Eksempel: `array_3.c`

Arrayer og pekeraritmetikk

- Kan utføre *regneoperasjoner* på pekere i C:
  - Er tillatt å legge til eller trekke fra heltallsverdier
  - Kan bruke operatorene `++` og `--` på pekere
- Eksempel:

```
int *p;
p++;
```

  - Flytter pekeren “fremover” i minnet med det antall bytes som trengs for å lagre en `int`, til *neste* heltall i minnet
- Testprogram for pekeraritmetikk: `array_4.c`

Dynamiske arrayer

- Statisk array:
  - Lengde av array konstant, endres ikke under kjøring
- Dynamisk array:
  - Opprettes under eksekvering av programmet
  - Lengde kan endres
- C-funksjoner for dynamisk minnehåndtering er definert i `stdlib.h` – se `man malloc`

Allokering av dynamisk array

- Systemfunksjonen `calloc`:
  - Setter av plass i minnet til array
  - Returnerer peker til første element
  - To parametre:
    - Antall elementer i arrayen
    - Størrelsen i bytes til hvert element
- Dynamisk allokert minne frigis etter bruk for å unngå minnelekkase, med funksjonen `free`
- Eksempel: `array_5.c`

Strenghåndtering

Tekststrenger

- En streng i C er en en array der hvert element er en verdi av typen `char`
- Siste tegn i en streng skal være *nulltegnet* `'\0'`
- Funksjonene for håndtering av strenger i C vil ignorere alle tegn i arrayen som kommer etter det terminerende nulltegnet
- Eksempler med initialisering og utskrift av strenger: `string_1.c`

Innlesing av tekststrenger

- Bruker array med tegn til å lagre innlest streng
- Nulltegnet legges automatisk til på slutten
- Bilbioteksfunksjoner for innlesing sjekker *ikke* om strengen som leses er lengre enn arrayen – fare for overskriving av arraygrense
- Vanlig å skrive *egne* innlesingsfunksjoner som leser ett og ett tegn, for sikrere lesing av tekst

Innlesing av tekst fra standard input

- Funksjonen `scanf` med formatspesifikasjon `%s`:
  - Leser forbi whitespace, deretter frem til første whitespace
  - Eller vi kan angi maks. antall tegn som skal leses
- Funksjonen `gets`:
  - Leser frem t.o.m. første linjeskift eller EOF
  - Regnes som usikker å bruke
- Eksempler: `string_2.c`

C-biblioteket for strenghåndtering

- Innebygde strengoperasjoner i C:
  - Ligger spesifisert i `string.h`
  - For full dokumentasjon, se `man string`
- Tilbyr operasjoner bl.a. for å:
  - Finne lengde på streng `strlen`
  - Kopiere strenger `strcpy` `strdup`
  - Sammenligne strenger `strcmp`
  - Slå sammen strenger `strcat`
- Eksempler: `string_3.c`

Filhåndtering

Strømmer og filer i C

- I/O til et program er en sekvens med data som kommer fra eller går til en ytre enhet som f.eks. skjerm, tastatur, fil på disk eller printer
- En *strøm* (stream) er *kommunikasjonskanalen* mellom C-programmet og den fysiske enheten
- En strøm utgjør et logisk grensesnitt mellom et C-program og I/O-enhetene
- Grensesnittet mot strømmen er uavhengig av hva slags ytre enhet det er – “alt er filer” på samme måte som i Linux

Forhåndsdefinerte strømmer

- Tre strømmer åpnes automatisk i et C-program:
  - `stdin` (tastatur), `stdout` (skjerm) og `stderr`
  - Alle tre standardstrømmer er definerte i `stdio.h`
  - Brukes av f.eks. `printf` og `scanf`
  - Samme som standard I/O-enheter i Linux shell
  - Kan redirigeres på samme måte som i Linux
- Andre strømmer kan opprettes i programmet og kobles til ytre enheter som f.eks. en fil på disk

Brukerdefinerte strømmer

- Strømmer deklareres som pekere av typen `FILE` :

```
FILE *filepointer;
```

- En fil åpnes med `fopen` , som har prototypen:

```
FILE *fopen(char *filename, char *mode);
```

`mode`: Forteller hvorledes filen skal brukes

- Filen lukkes med `fclose` etter bruk, prototypen er:

```
int fclose (FILE *filepointer);
```

Noen vanlig brukte modus for filer

```
FILE *fopen(char *fname, char *mode);
```

Mode	Meaning
`r`	Open a text file for reading
`w`	Create a text file for writing
`a`	Append to a text file
`rb`	Open a binary file for reading
`wb`	Open a binary file for writing
`ab`	Append to a binary file
`r+`	Open a text file for read/write
`w+`	Create a text file for read/write

Noen I/O-funksjoner fra `stdio.h`

`fgetc`	leser ett tegn fra tekstfil
`fputc`	skriver ett tegn til tekstfil
`fgets`	leser en streng
`fputs`	skriver en streng
`fscanf`	leser formattert
`fprintf`	skriver formattert
`fread`	leser blokk med data fra binær fil
`fwrite`	skriver blokk med data til binær fil

Lesing og skriving av tegn

- `int fgetc(FILE * filpeker);`
  - Leser og returnerer neste tegn fra en åpen tekstfil
  - Returnerer `int` , men kan leses inn i en `char`
  - Returnerer EOF hvis fil slutt eller feil
- `int fputc(int ch, FILE * filpeker);`
  - Skriver tegnet `ch` til åpen tekstfil
  - Returnerer tegnet som er skrevet ut, EOF hvis feil
- Eksempel: `file_1.c`

Lesing og skriving av strenger

- `char *fgets(char *s, int n, FILE *fp);`
  - Leser høyst `n-1` tegn, stopp etter linjeskift/EOF
  - Lagrer innleste tegn, *inkludert* linjeskift i strengen `s`
  - Nullterminеringstegnet legges inn til slutt
  - Returnerer peker til lest streng, eller NULL hvis feil
- `int fputs(char *s, FILE *fp);`
  - Skriver strengen `s` til fil, returnerer NULL hvis feil
  - Tar ikke med nulltegnet, legger ikke på linjeskift
- Eksempel: `file_2.c`

Formatert lesing og skriving av filer

```
int fprintf(FILE *fp, char *ctrl_str, ...);

int fscanf(FILE *fp, char *ctrl_str, ...);
```

- Virker på samme måte som `printf` og `scanf` , men leser fra fil i stedet for fra standard input:
- Eksempler: `file_3.c` `file_4.c`

Kommandolinjeargumenter

Kommandolinjeargumenter

- C-program kan ta argumenter fra kommandolinjen
- Tilgjengelige i parametrene til hovedprogrammet:

```
int main(int argc, char *argv[])
```

  - `argc`: Antall kommandolinjeargumenter
  - `argv`: Array med argumentene som strenger§
- Eksempel: kommandolinje.c

§: `argv[0]` er programmets navn, `argv[1]` inneholder første argument, `argv[2]` andre argument osv.

Strukturer

Strukturer: Sammensatte datatyper

- Strukturer i C brukes til å definere egne, sammensatte datatyper
- En struktur definerer en datatype som er sammensatt av flere variable som kalles strukturens *medlemmer*
- Medlemmene i en struktur kan være av ulik type
- Strukturer er det nærmeste C kommer "objektorientering", men det er store forskjeller mellom strukturer i C og f.eks. klasser i Java

Syntaks for å *definere* en struktur

```
typedef struct
{
    <Deklarasjon av medlemmer>
} navn_t;
```

- `typedef` – Definisjon av en ny datatype
- `struct` – Datatypen som defineres er en struktur
- Strukturens variabler/medlemmer angis mellom `{}`
- Etter `{}` angis strukturdatatypens navn
- Vanlig å avslutte egendefinerte typenavn i C med `_t`

Eksempel: Struktur for brøker

```
typedef struct
{
  int teller;
  int nevner;
} brok_t;

main()
{
  brok_t b;
  b.teller = 1;
  b.nevner = 2;
}
```

- Deklarerer en variabel `b` av typen `brok_t` i `main`
- `b` kan lagre data om én brøk
- Trenger ikke gjøre *new* e.l. som i Java for å opprette "objektet" `b`
- For å aksessere et medlem i strukturen, brukes et punktum og medlemmets variabelnavn

Strukturer i C vs. klasser i Java

- Klasseobjekter i Java kan ha både data (variable) og oppførsel (metoder). Strukturer i C har bare data.
- Java-klasser kan arve hverandre. Arv finnes ikke i C.
- Variable i en Java-klasse kan ha ulik tilgang “fra utsiden” (private, public, protected). Finnes ikke i C.
- En klasse har en konstruktør som initierer objektenes variable. En struktur inneholder ikke funksjoner og har derfor ingen konstruktør.

Kopiering av innholdet i en strukturvariabel

- La `b1` og `b2` begge være strukturvariable av samme type, f.eks. av typen `brok_t`
- Setningen:

```
b2 = b1;
```

  fører til at innholdet i *hvert av feltene* i strukturvariabelen `b1` *kopieres* over til tilsvarende felt i strukturvariabelen `b2` .

- Eksempel: `struct_1.c`

Struktur med medlemmer av ulik type

- Medlemmer kan være både enkle variable og arrayer:

```
#define MAX 80

typedef struct
{
  char etternavn[MAX];
  char fornavn[MAX];
  int alder
  char snittkarakter;
} student_t;
```

- Eksempel: `struct_2.c`

Arrayer med strukturvariable

- C støtter også arrayer av egendefinerte datatyper
- Her settes det av plass til 10 student-variable som legges rett etter hverandre i minnet:

```
student_t studenter[10];
```

- Når et medlem i en strukturvariabel i en array skal aksesseres, plasseres operatoren `.` mellom indeks og medlemmets variabelnavn:

```
int a = studenter[4].alder;
```

- Eksempel: `struct_3.c`

Sammenligning av strukturvariable

- Operatorene for sammenligning av enkle variable i C (`==` `!=` `<` `>` etc.) kan *ikke* anvendes direkte på sammensatte datatyper
- I stedet må hvert enkelt *medlem* av en sammensatt datatype sammenlignes separat, for de to struktur-variablene som skal sammenlignes
- Det er vanlig å skrive *egne* funksjoner for å sammenligne innholdet i egendefinerte datatyper
- Eksempel: `struct_4.c`

Strukturer som funksjonsparametre

- Når vi overfører en strukturvariabel som parameter til en funksjon, kopieres *hele innholdet* av strukturen over i en lokal strukturvariabel i funksjonen
- Innholdet i hele *arrayer* i strukturvariabelen kopieres også – parameteroverføring kan være tidkrevende
- Funksjonen kan *ikke* endre på innholdet i en vanlig strukturvariabel som brukes som parameter
- Eksempel: `struct_5.c`
- For å endre verdier på medlemmene i en struktur, må vi i stedet overføre en *peker* til strukturvariabelen

Pekere til strukturer

```
#include <stdio.h>

typedef struct
{
  int teller;
  int nevner;
} brok_t;

main ()
{
  brok_t b, *b_p;
  b_p = &b;
  b_p->teller = 1;
  b_p->nevner = 2;
}
```

- Deklarerer en variabel `b` av typen `brok_t` , og en variabel `b_p` av typen *peker til* `brok_t`
- `b` kan lagre data om en brøk, `b_p` lagrer minneadressen til en brøk
- `b_p` kan senere endres til å peke på andre variable av type `brok_t`
- For å aksessere et medlem til en strukturvariabel gjennom en peker, brukes “pil-operatoren” `->`

Peker til struktur som funksjonsparameter

- For at en funksjon skal kunne endre verdiene på medlemmene i en struktur, overfører vi en *peker* til strukturvariabelen som parameter
- Funksjonen bruker da strukturvariabelens adresse som utgangspunkt, og jobber direkte mot innholdet i strukturen
- Raskere å overføre en peker enn en strukturvariabel
- Eksempel: `struct_6.c`

Struktur som returverdi fra funksjoner

- En funksjon kan returnere en verdi som er en variabel av selvdefinert struktur-datatype
- Funksjonen returnerer da *ikke* en peker til struktur-variabelen, men *verdiene* til alle medlemmene
- Funksjoner *kan* også returnere pekere til strukturvariable, raskere enn å returnere hele inneholdet av strukturen
- Eksempel: `struct_7.c`

Systemprogrammering i C under Linux

Linux systemkall i C-programmer*

- Linux-kjernen tilbyr et sett med *systemkall*:
  - C-funksjoner som kan brukes for å få operativsystemet til å utføre (lavnivå) tjenester
  - Grensesnittet mellom C-applikasjoner og OS
- Systemkall brukes bl.a. for å:
  - Kontrollere prosesser og tråder
  - Håndtere device
  - Styre filsystemet
- Oversikt over C-systemkall: `man syscalls`

*: Vi skal bare se enkle eksempler på prosesshåndtering

Starte nye prosesser fra C-program

- Systemkallet `fork` :
  - Lager ny prosess som er kopi av forelderprosessen
  - Returnerer PID til den nye prosessen
  - De to prosessene vil eksekvere samme C-kode fra det punktet der `fork` ble kalt
- Systemkallet `execvp` :
  - Terminerer nåværende prosess, starter en helt ny prosess i stedet
- Eksempler: `fork.c` `execvp.c` `fork_execvp.c`

POSIX-tråder i C-program: `pthreads`

- Ny tråd startes med systemkallet `pthread_create` :
  - Parameter angir funksjonen som tråden skal starte i
  - Tråd eksekveres *parallelt* med hovedprogrammet
  - Tråder terminerer ved slutten på funksjonen de starter i
  - Kan også terminere med systemkallet `pthread_exit`
- C-programmer med tråder må inkludere `pthread.h` og kompileres med opsjonen `-pthread`
- Eksempel : `threads.c`