**UMEÅ UNIVERSITET**
**Instutionen för Datavetenskap**
**Guide**

**17 januari 2023**

5DV149

# Datastrukturer och Algoritmer

Guide inför DoA:n

version 1.0

**Författare** Arvid Bergman Thörn
**Användarnamn** dv20arn

# Innehåll

# 1 Introduktion till dokumentet

I detta dokument är syftet att ni, studenter på kursen *Datastrukturer och Algoritmer*, ska få lite förberedande eller uppfriskande kunskaper inom programspråket `C`, verktyget `OverLeaf` och LaTeX samt hur man skriver en rapport då detta för många av er kommer vara den första kursen där ni ska skriva kod med en tillhörande rapport.

Dokumentet är indelat i tre delar:

1. *En genomgång av programspråket C* där vi går igenom lite saker som är bra att kunna. För er som går C så kommer ni känna igen stora delar från kursen *Imperativ Programmering (C)*, men det kan ändå vara nyttigt med lite repetition.
2. *En introduktion till LaTeX* där vi går igenom hur vi kan använda `OverLeaf` och LaTeX för att enkelt kunna skriva snygga och korrekta rapporter.
3. *En introduktion till rapportskrivande* behandlar ämnet att skriva rapporter om kod. På universitetet så ställs andra krav på rapporter än på gymnasiet och det är viktigt att bygga en bra grund för framtida skrivande. I detta kapitel tar vi upp vad som är viktigt att tänka på, och "*the do's and dont's*".
4. *En avslutande kommentar om kursen* där jag avslutar detta dokument med lite tankar om kursen.

Dokumentet är skrivet av Arvid Bergman (dv20arn), handledare på kursen och handledare på samma kurs förra året. Det är baserat på erfarenheter av svårigheter från tidigare års studenter. Förhoppningsvis så ska ni som läser och bearbetar detta dokument ha en bättre och mer '*smooth*' upplevelse av kursen.

Det är viktigt att komma ihåg att detta dokument inte är en absolut guide på exakt hur ni ska göra, utan en guide på hur ni kan göra.


Figur 1: meirl

# 2 En genomgång av programspråket C

I denna kurs kommer de praktiska datalaborationerna skrivas i programspråket `C`. Datalaborationerna kan komma att kännas svåra och knepiga men inom programmering är det oftast då vi lär oss mest. Ni som läser denna kurs går en utbildning som innefattar mycket programmering (eller så går ni en annan utbildning och vill lära er mer om programmering) och denna kurs är en av de viktigare kurserna att förstå för vidare studier inom datavetenskap.

## 2.1 Grunder i C

Vi kommer inte gå in särskilt mycket på grunderna av `C`, då det förväntas att ni har lärt er från era grundkurser inom det. Men, vi repeterar litegrann. Känner ni att ni inte kommer ihåg något av den tidigare `C` kursen så kan ni titta på följande länk (C Crash Course). Jag skulle rekommendera att inte börja med uppgifterna på denna kurs om ni inte behärskar grunderna från förra kursen.

`C` är ett imperativt språk, där ordningen av saker spelar roll. Vi styr programflödet genom logiska operander och sekvensblock. Det är viktigt att komma ihåg att `C` är *top-down*, det går alltså uppifrån nedåt.

### 2.1.1 Funktioner

Funktioner är en viktigt del av programspråket `C`. Funktioner kan ses som hjälpredor till ditt program, där du kan ha olika funktioner som utför specifika uppgifter. Du kan som utvecklare av ett program definiera egna funktioner men du kan också ta del av fördefinierade funktioner som återfinns genom att inkludera olika bibliotek.

För att skriva en funktioner behöver vi några saker. Vi behöver veta dess returtyp, dess namn och eventuella parametrar.

```c
void print_message(char *msg) {
    printf("%s\n", msg);
}
```

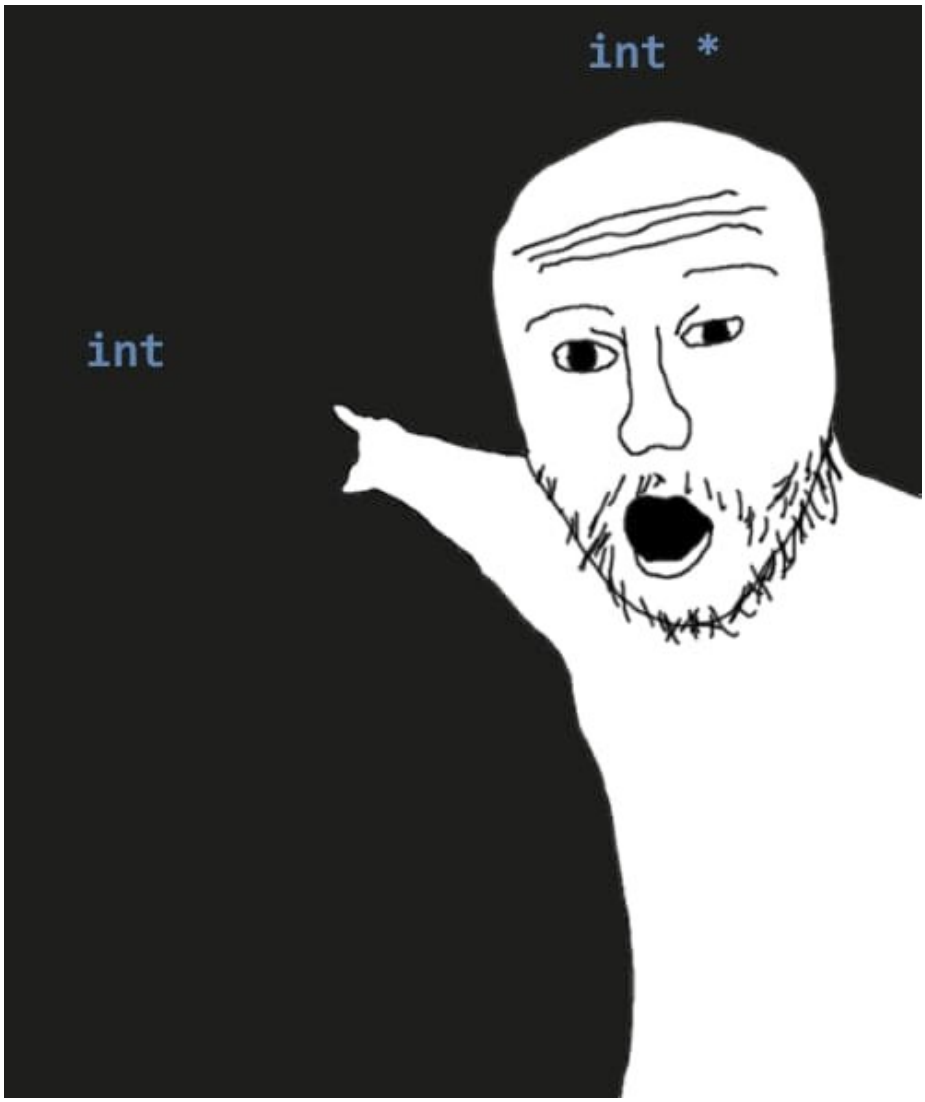
Listing 1: Exempel på en funktion i C.

I vår funktion ovan så kan vi betrakta dessa delar. Den första delen, `void`, är vad vår funktion returnerar. I detta fall, ingenting. Den andra delen, `print_message`, är namnet på vår funktion. `C` är skiftläge-känsligt, så en funktion som heter `print_Message` skulle inte bli igenkänd som den vi just definierade. Sista delen, `(char *msg)`, är vår formella parametrar. Vi kallar det en formell parameter och när vi anropar funktionen kallar vi det ett argument.

### 2.1.2 Pekare

Pekare är en grundsten inom `C`. Denna kurs kommer således pröva era kunskaper om pekare. Det är helt okej att tycka pekare är klurigt och svårt - för det är dem! Det är dock viktigt att komma ihåg att pekare är i grunden inte särskilt komplicerade utan det är oftast en själv som lägger krokben för en själv.

Följande video (länk) förklarar pekare på en rätt bra nivå för denna kurs (namnet på videon behöver ni inte lägga så mycket vikt på). Kolla på den och försäkra er om att ni förstår vad skaparen av videon förklarar.




Figur 2: Kolla, en pekare.

## 2.2 Header-filer

Vi har i tidigare kurser använt funktionsdeklarationer eller behövt hålla koll på ordningen vi definierar funktioner för att inte det ska bli problem vid kompilieringen av programmet. Genom använda så kallade *header*-filer så kommer vi undan den jobbiga processen av att kolla koll på vilken ordning funktionerna är definerade (kom ihåg att C är *top-down*).

En *header*-fil skapas genom att ge filen ändelsen `.h`. Ni har stött på *header*-filer tidigare - när ni inkluderar bibliotek! Att inkludera fördefinierade bibliotek är ungefär samma sak som att inkludera egna. Nedan kan vi se hur vi gör det.

```
1     #include <stdio.h>
2     #include "min_egna_header.h"
```

Listing 2: Exempel på att inkludera header-filer.

Från kodsnutten ovan kan en se att det som skiljer dem åt är att vi använder citattecken (") kring våra egen-definierade *header*-filer istället för krokodilmunnar. En *header*-fil (kan) se ut som följande:

```
1  #ifndef MIN_EGNA_HEADER_H
2  #define MIN_EGNA_HEADER_H
3      #include <stdio.h>
4
5      void print_message(char *msg);
6
7      int add_two_integers(int a, int b);
8
9  #endif /* MIN_EGNA_HEADER */
```

Listing 3: Exempel på en header-fil.

I *header*-filen deklarerar vi enbart funktionen, vi definierar den inte. Definitionen av funktionen sker i dess `.c`-fil. Vi kan nu skapa en fil, `min_egna_header.c` för att definiera funktionerna, se nedan.

```
1     #include "min_egna_header.h"
2
3     void print_message(char *msg) {
4         fprintf(stdout, "%s\n", msg);
5     }
6
7     int add_two_integers(int a, int b) {
8         sum_of_integers = a + b;
9         return sum_of_integers;
10    }
```

Listing 4: Exempel på att definiera header-fil.

När vi nu har definierat vår *header*-fil kan vi även inkludera den i andra filer för att få beteendet av `min_egna_header` i dem klasserna. Kom ihåg att även länka ihop filerna vid kompiliering av programmet!

Viktigt att tänka på att när vi använder oss av en *header*-fil så flyttar vi ut ansvaret för funktionsdokumentation till den. Alltså, en funktionskommentar ligger alltså i *header*-filen och inte i vår `.c`-fil!

## 2.3 Dynamisk Minneshantering

I `C` så måste vi allokera eget minne och frigöra det minnet själva. Andra språk som exempelvis `Java` använder sig av en s.k. `garbage-collector` men det kommer ni få lära er om nästa kurs, *Objektorienterad Programmering*.

I `C` så har vi fördefinierade funktioner för att hjälpa oss att allokera minne. Vi har `malloc`, `calloc` samt `realloc`.

De två första funktionerna `malloc` och `calloc` är rätt lika varandra. Den enda skillnaden är egentligen att `calloc` ger oss minne som är "nollat", d.v.s. att allt inom det minnet är satt till noll. Det gör att denna funktioner är lite långsammare än `malloc` som inte nollställer minnet. Ni kanske kommer ihåg att det var viktigt att explicit sätta en variabel man summerar till till noll. Detta är på grund av att det kan ligga kvar skräpvariabler i den minnesplatsen.

`Realloc` skiljer sig mot de två tidigare funktionerna. I denna funktion utökar vi redan allokerat minne! Det kan vara att vi exempelvis allokerar minne för 10 karaktärer men upptäcker sen att vi i själva verket behöver 20. Då kan vi m.h.a `realloc` allokera 10 nya platser till den variabeln.

I följande exempel (taget från TutorialsPoint) så ser vi både hur `malloc` samt `realloc` fungerar:

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

int main () {
   char *str;
   /* Initial memory allocation */
   str = (char *) malloc(15);
   strcpy(str, "tutorialspoint");
   printf("String = %s,  Address = %u\n", str, str);

   /* Reallocating memory */
   str = (char *) realloc(str, 25);
   strcat(str, ".com");
   printf("String = %s,  Address = %u\n", str, str);

   free(str);
   return(0);
}
```

Listing 5: Exempel på malloc och realloc.

Utdatan blir då följande:

```
String = tutorialspoint, Address = 355090448
String = tutorialspoint.com, Address = 355090448
```

Som ni kan se är adressen på variabeln fortfarande densamma, och det är för att vi inte skapade en ny variabel utan utökade minnet för den nuvarande.

Minne man allokerar måste man också frigöra. Detta görs genom funktionen `free`. I tidigare kodsnutt så kan vi se hur de frigör variabeln de allokerat minne åt genom anropet `free(str);`. Minneshantering kan vara svårt och ofta ge upphov till många errors samt krascher av programmet, och det är därför viktigt att man håller noga koll på vad man allokerar och när man ska frigöra det minnet.

### 2.3.1 Valgrind

`Valgrind` är ett verktyg som hjälper oss hitta minnesläckor. Ni kommer under kursen att bli mer bekanta med det. För att få bättre utdata från `Valgrind` kan ni kompilera ert program med flaggan `-g`.

För att köra `Valgrind` behöver det vara installerat på er dator. Sedan är det så enkelt som att skriva:

```
1    valgrind ./mitt_program
```

Listing 6: Exempel på körning med Valgrind.

En kan även lägga till flaggan `--leak-check=full` för mer detaljerad information. I figur 3 ser vi ett program som har gjort 1,149 allokeringar av minne och 1,149 frigivningar av minne - allt minne är därmed frigjort. För att era datalaborationer ska gå igenom `Labres` och våra tester får de inte läcka minne.


Figur 3: Ett program utan minnesläckor. Nice.

Ni kommer stöta på minnesläckor under datalaborationerna under denna kurs och det kommer vara frustrerande. Det råd jag har till er för att minska det är

att arbeta inte med tankesättet ”jag fixar det sen”, utan tänk direkt på vart minnet du allokerar sedan ska frigöras. Du behöver inte frigöra det direkt, men skriv en kommentar så du kommer ihåg! Det är också värt att tänka på vilket minne du frigjort - du kommer få problem om du försöker frigöra det minnet.

Det finns lite vanliga ”felmeddelanden” som uppstår med dynamisk minneshantering och minnesläckor. Nedan kommer några och vad de (oftast) beror på.

1. `Segmentation Fault`
   Kanske det vanligaste ’error’:et ni kommer stöta på under denna kurs. Det är (relativt) enkelt att lösa då det oftast beror på att ni försöker läsa eller skriva minne som ni inte har tillgång till.

2. `Invalid write of size x`
   Detta fel beror på att ni skriver (alltså, att ni exempelvis försöker tilldela något till något annat) mot en minnesplats ni inte har tillgång till. Det kan oftast följas av ett annat meddelande, ungefär ”*Address 0xabcdefg is 0 bytes after a block of size y alloc’d.* Detta betyder alltså att vi kanske försöker skriva y+x saker men vi allokerade bara ett block av y platser.

3. `Invalid read of size x`
   Denna är lite lik ovan, men istället för skriva så försöker ni läsa.

4. `Conditional jump or move depends on uninitialised value(s)`
   En `jump` är en datainstruktion som ’hoppar’ någonstans i minnet. Detta felmeddelande kan vara lite lurigt för ditt program kan fungera som vanligt men testerna på `Labres` kan misslyckas. Det beror (som meddelandet säger) på o-initialiserade värden.

5. `Invalid free() / delete / delete[] / realloc() at y`
   Detta fel beror oftast (nästan alltid) på att ni frigör något ni inte ska frigöra (som tidigare nämnt). Det kan vara att ni försöker frigöra något som inte har fått dynamiskt allokerat minne eller att ni försöker frigöra dynamiskt minne två (eller fler) gånger.

Punkt 1, 2, och 3 kan låta lite lika - och på ytan är de det. Just nu behöver ni inte förstå exakt vad som skiljer dem åt!

## 2.4 Dokumentation

Ni kommer under kursen få se och ta del av kodexempel. Dessa exempel är oftast dokumenterade i överkant - det är för **er** skull. Det är inte så man bör dokumentera. Man behöver inte kommentera exempelvis variabeldeklarationer eller funktionsanrop - det är överkant. Lägg hellre krutet på att göra välskrivna funktionskommentarer och filkommentarer. Välskriven och bra namngiven kod kan ofta göra mer för förståelsen av programmet än massa kommentarer.

I `C` så kan vi dokumentera enligt `Doxygen` standard. Det uppnår vi genom skriva dokumentera på följande sätt:

```
1    /**
2    * @brief Checks if parameter is leet.
3    * @param a The integer to check.
4    * @returns A boolean.
5    */
6    bool is_leet(int a) {
7        return a == 1337;
8    }
```

Listing 7: Exempel på en funktionskommentar.

Observera att detta enbart är ett exempel på `Doxygen`-standard. Det är alltså **inte** en mall på hur innehållsrika/fattiga kommentarer ska vara.

Ett exempel på för mycket dokumentation:

```
1    // entry-point function of program silly_prog.c
2    int main(int argc, char** argv) {
3        // creates a variable and sets it to 1.
4        int a = 1;
5        // creates a char array
6        char *my_name;
7        // allocates memory to variable my_name.
8        my_name = malloc(5 * sizeof(char));
9        // copies my name into variable my_name
10       strncpy(my_name, "arvid", 5);
11       // prints my name to stdout
12       fprintf(stdout, "%s\n", my_name);
13       // frees memory allocated to my_name
14       free(my_name);
15       // return from main
16       return 0;
17
18   }
```

Listing 8: Exempel på dokumentation gone wrong.

## 2.5 Argument till main

Ibland vill vi ta indata till ett program direkt från kommando-tolken. Vi kan exempelvis vilja öppna andra filer (exempelvis `.txt`-filer) i vårt program för att läsa datan i dem.

### 2.5.1 Argv och argc

En del av har kanske bara sett att man deklarerar en `main`-funktion med formella parametern `void`. Men, vi kan istället definiera den såhär:

```
1    int main(int argc, char** argv) {
2        ...
3    }
```

Listing 9: Exempel på en main-funktion

Dessa två variabler är rätt *straight-forward*. Den första (`argc`) är antalet argument vi skickar till programmet och `argv` är en array av argumentens namn.

Om vi skulle skicka följande till vårt program:

```
1    ./silly_prog Datastrukturer och Algoritmer
```

Listing 10: Exempel på argument till main

I tabell 1 ser vi resulterande array av argument.

Tabell 1: Hur vår array av argument (argv) ser ut efter ovanstående körning. Argc är 4.

| | |
|---|---|
| argv[0] | ./silly_prog |
| argv[1] | Datastrukturer |
| argv[2] | och |
| argv[3] | Algoritmer |

Vi kan alltså få tillgång till argument vi skickar till programmet från kommando-tolken genom:

```
1    ..
2    char* arg_1 = argv[1];
3    char* arg_2 = argv[2];
4    char* arg_3 = argv[3];
5    ..
```

Listing 11: Exempel på hur man tar argument från kommando-tolken.

### 2.5.2 Att skicka en fil till programmet

När vi jobbar med filer så använder vi oss av en filpekare som vi deklarerar med typen `FILE`. Att öppna en fil kan vi göra genom att använda oss av funktionen `fopen`.

Vi öppnar filen genom följande syntax:

```
1    fp = fopen("filename","mode");
```

Listing 12: Exempel på hur man tar argument från kommando-tolken.

`Mode` definerar hur vi vill öppna filen. Med `r` så kan vi läsa filen, med `w` kan vi skriva till den. Med `w+` kan vi skriva och läsa. Öppna den inte med `w+` 'bara för att'. När vi öppnat en fil och vi är klara med den är det viktigt att vi **stänger** filen. Genom `fclose` kan vi stänga den fil vi öppnat.

När man ska hantera filer är det viktigt att vi kontrollerar att filen öppnades korrekt och att vi har fått rätt antal parametrar till programmet. Om vi tar exemplet från tabell 1 hade det kunnat se ut såhär:

```
1
2    if(argc != 4) {
3        fprintf(stderr, "We're missing something.. :thinking:");
4    }
5
6    *in_fp = fopen(argv[1], "r");
7    if(*in_fp == NULL)
8    {
9        fprintf(stderr, "Could not open file: %s\n", argv[1]);
10       return EXIT_FAILURE;
11   }
```

Listing 13: Exempel på hur man verifierar antalet argument samt att fil gick att öppna.

## 2.6 Tips för enklare kod

Ett av våra mål när vi kodar är att skriva enkel och lättförstådd kod. Ett av sätten vi uppnår detta är genom bra dokumentation men kanske viktigare är att koden är väl utformad. Inom programmering finns det ett begrepp som kallas refaktorisering - det betyder ungefär att när du skrivit ett stycke kod så kan du i efterhand kolla på det och fundera om det finns sätt att göra koden mer simpel. Väl utformade och specifika funktioner är ett sätt att uppnå detta. Något jag alltid uppmanar till är att skriva en "fel-funktion".

```
1    void exit_on_error(char *msg) {
2        fprintf(stderr, "%s\n", msg);
3        exit(EXIT_FAILURE);
4    }
```

Listing 14: Exempel på en funktion för errors.

En sådan funktion är inte nödvändig - men den kan underlätta läsbarheten av koden. Pausa ibland och kolla på koden och fundera över vad du kan göra för att förenkla den och göra den mer läsbar.

Om ni har en funktion som heter `subtract_or_add`, så kan vi alltså refaktorisera detta till `subtract()` samt `add()`. Mer lättläst kod gör det lättare för dig att förstå koden för att exempelvis debugga den men också för oss som rättar koden.

## 2.7 Manualen för C

Det bästa sättet att ta reda på information om C är genom de s.k. `man`-sidorna. I din terminal, skriv helt enkelt in `man` (ev. följt utav vilken sida) följt utav det du vill kolla upp, ex:

```
man malloc
```

Listing 15: Exempel på manualen för malloc.

I figur 4 så kan vi se en skärmpdump utav en del av manualsidan för kommandot `malloc`. Här finns massvis med matnyttig information som kommer att hjälpa er under denna och framtida kurser.

```
MALLOC(3)                          Linux Programmer's Manual                          MALLOC(3)

NAME
       malloc, free, calloc, realloc, reallocarray - allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);
       void *reallocarray(void *ptr, size_t nmemb, size_t size);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       reallocarray():
           Since glibc 2.29:
               _DEFAULT_SOURCE
           Glibc 2.28 and earlier:
               _GNU_SOURCE

DESCRIPTION
       The  malloc()  function allocates size bytes and returns a pointer to the allocated memory.  The
       memory is not initialized.  If size is 0, then malloc() returns either NULL, or a unique pointer
       value that can later be successfully passed to free().

       The free() function frees the memory space pointed to by ptr, which must have been returned by a
       previous call to malloc(), calloc(), or realloc().  Otherwise, or if free(ptr) has already  been
       called before, undefined behavior occurs.  If ptr is NULL, no operation is performed.

       The calloc() function allocates memory for an array of nmemb elements of size bytes each and re-
       turns a pointer to the allocated memory.  The memory is set to zero.  If nmemb  or  size  is  0,
       then  calloc()  returns  either  NULL,  or a unique pointer value that can later be successfully
       passed to free().  If the multiplication of nmemb and size would  result  in  integer  overflow,
       then  calloc()  returns an error.  By contrast, an integer overflow would not be detected in the
       following call to malloc(), with the result that an incorrectly sized block of memory  would  be
       allocated:

           malloc(nmemb * size);

       The  realloc()  function  changes  the size of the memory block pointed to by ptr to size bytes.
       The contents will be unchanged in the range from the start of the region up to  the  minimum  of
       the  old  and new sizes.  If the new size is larger than the old size, the added memory will not
       be initialized.  If ptr is NULL, then the call is equivalent to malloc(size), for all values  of
       size;  if  size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr).
       Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc(), or  re-
       alloc().  If the area pointed to was moved, a free(ptr) is done.

       The  reallocarray()  function changes the size of the memory block pointed to by ptr to be large
       enough for an array of nmemb elements, each of which is size bytes.  It  is  equivalent  to  the
       call

               realloc(ptr, nmemb * size);

       However, unlike that realloc() call, reallocarray() fails safely in the case where the multipli-
       cation would overflow.  If such an overflow occurs, reallocarray() returns NULL, sets  errno  to
       ENOMEM, and leaves the original block of memory unchanged.
```

Figur 4: En skärmdump av en sektion av manualsidan för malloc.

Använd man-sidorna regelbundet och bemästra dem. Att bli bekväm med att läsa dokumentation är viktigt!

# 3 En introduktion till LaTeX

Ett väldigt användbart verktyg för att skriva rapporter är hemsidan `OverLeaf`. Det är en online textredigerare och kompilator för verkyget *LaTeX*. *LaTeX* har en liten inlärningskurva, men du kommer tacka dig själv i framtida studier att du lärde dig det. I denna del ska jag försöka mjukna inlärningskurvan litegrann.

## 3.1 OverLeaf

Du behöver inte använda `OverLeaf` för att använda *LaTeX*, men det underlättar väldigt mycket. För använda `OverLeaf` behöver du skapa ett konto. På din användare sparas alla dina rapporter och du kan strukturera upp dina rapporter i olika mappar.

## 3.2 Att skapa en mall

*LaTeX* använder sig av mallar för att ge rapporten det utseende den har. Detta går utmärkt att göra från grunden själv - men det finns tillgängligt massa färdiggjorda mallar åt dig så du slipper besväret. Detta dokument är exempelvis skrivet i Umeå Universitets egna rapport mall, som ni kan hitta här. Genom att använda den mallen så minskar ni drastiskt att ”åka dit” på tråkiga fel såsom felaktigt sidnumrering (t.ex. att innehållsförteckningen ska vara numrerade som *i*).

## 3.3 Kommandon

I *LaTeX* använder man sig mycket utav kommandon. Om man är van i att skriva i `Microsoft Word` eller `Google Docs` så kan det känns mycket främmande. Vi går igenom några enkla kommandon för att komma igång.

1. **Uppdelning av rapport.**
   För att dela upp en rapport så använder man sig främst utav `\section`, `\subsection` och `\subsubsection`. För att namnge en sektion så skriver vi alltså: `\section{Lorem Ipsum}`.

1. **Fetmarkera eller kursivera.**
   För att skriva med fetmarkerad eller kursiverad text kan man använda kommandona `\textbf` eller `\textit`. Man kan också använda `ctrl-b` eller `ctrl-i`.

2. **Ny rad eller ny sida?**
   För att tvinga en ny rad kan en använda kommandot `\newline` (eller `\\`) och för att skapa en ny sida använder en kommandot `\newpage`.

3. **Att hänvisa till en figur eller sektion.**
   Det är viktigt att kunna hänvisa till figurer korrekt (mer om det senare). Du kan genom `\label{fig:my_figure}` skapa en referens, för att senare använda `\ref{fig:my_figure}` för att hänvisa till den figuren.

Det var lite grundläggande kommandon som kommer vara viktiga för er när ni skriver rapporter. Det finns lite andra nyckelord som man använder för att göra saker, exempelvis `\begin`.

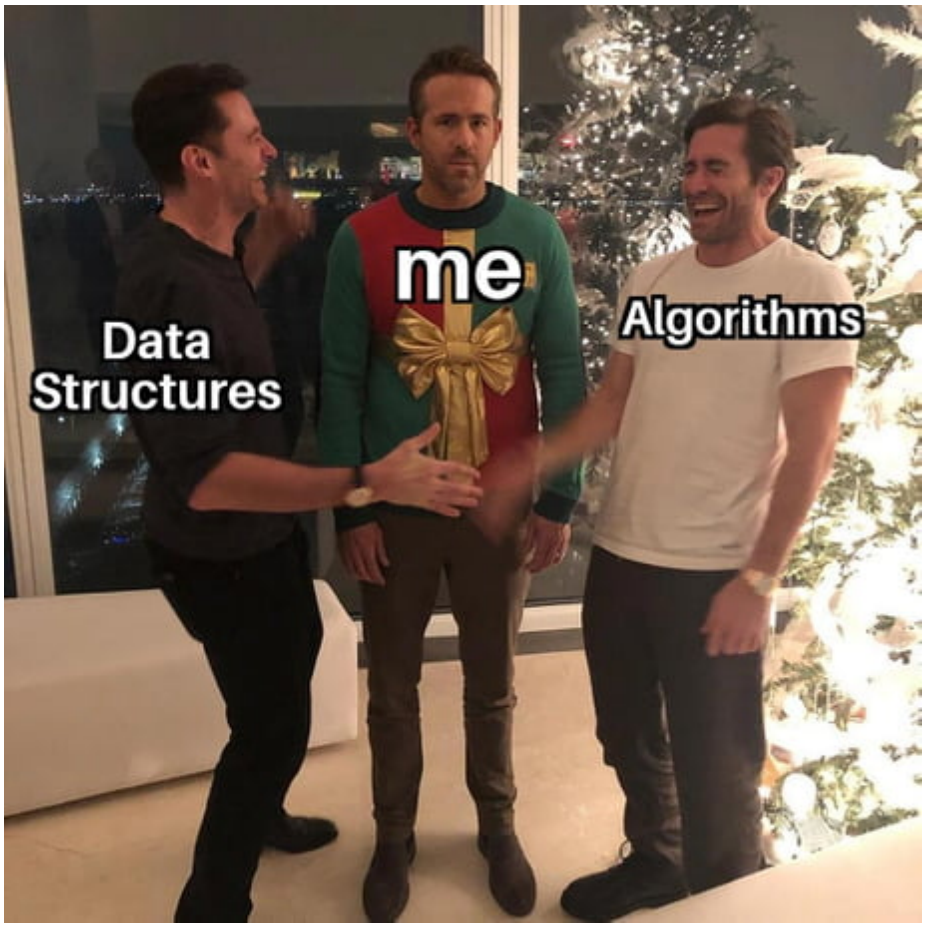
Exempelvis för att skapa listan ovan, där vi använde `\begin{enumerate}`. Det är viktigt att komma ihåg att stänga en `\begin` med en `\end`, så i fallet med listan stänger vi den med `\end{enumerate}`

Om vi vill infoga en bild så måste vi använda oss av det. Nedan följer ett exempel på hur en kan inkludera en bild.

```
\begin{figure}[h!]
    \centering
    \includegraphics[width=\textwidth]{images/3_dudes.jpg}
    \caption{Ibland blir omformateringen av bilder inte super..}
    \label{fig:kekw}
\end{figure}
```

Från kodsnutten ovan så kan ni se hur en använder `\begin` för att instansera en figur. De hårda klamrarna med *h!* i är ett direktiv till kompilatorn om placering av bilden. Man kan även se hur bredden på bilden definieras till att vara lika bred som texten, och då sköter Overleaf omformateringen av bilden *automagiskt*.

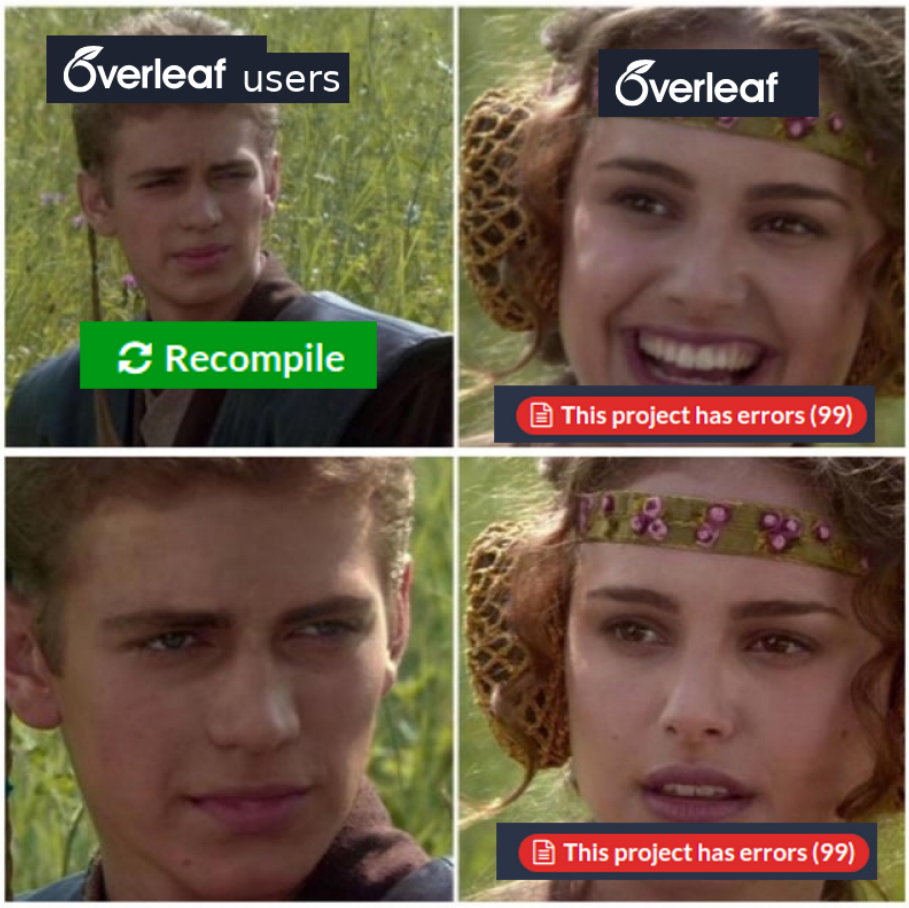
Resultatet blir figur 5, sedd nedan:




Figur 5: Ibland är omformateringen av bilder inte super..

Det finns lite saker som kan vara förvirrande med LaTeX också. Exempelvis, att många tecken inte går att skriva i vanlig text utan måste anges via kommandon. Denna sida (länk) kan vara bra att kika på. Om man ska använda citattecken behöver man skriva $”$. För att kommentera ut text så markerar du texten och trycker %.

Att lära sig Overleaf och LaTeX är inte trivialt. Det kan kännas drygt, tråkigt och onödigt. Att istället skriva i tidigare kände texteditorer är oftast den lättaste vägen att ta, men jag kan lova er att om ni lär er detta nu kommer ni tacka er själva i framtida studier. Ni får självfallet fråga oss som jobbar på kursen om frågor kopplade till LaTeX också.



# 4 En introduktion till rapportskrivande

Detta är för de flesta av er den första kursen där ni skriver en rapport till en kurs inom datavetenskap. Det finns en del krav på rapporter som kan tyckas vara petiga men kom ihåg att ni är här för att lära er och om ni inte gör det nu får man (oftast) bita det sura äpplet senare. Nedan listar jag några vanligt förekommande misstag,

1. **Framsidan på rapporten.**
   Framsidan av er rapport ska vara korrekt. Det ska framgå vad det är för kurs, vad det är för uppgift, vem som skrivit (namn och cs-id), institution, lärare och handledare.

2. **Innehållsförteckningen.**
   Det ska finnas en innehållsförteckning över innehållet av rapporten.

3. **Sidnumrering.**
   Era rapporter ska vara sidnumrerade. Framsidan av rapporten ska inte vara sidnumrerad och innehållsförteckningen vara numrerade som *i*. Efter de börjar sida ett.

4. **Rubriker är inte numrerade.**
   Rubriker måste vara numrerade. Vi vill inte se en rubrik som inte har korrekt numrering. Det gäller även underrubriker.

5. **Figurer och tabeller refereras inte korrekt.**
   En av de vanligaste misstagen är att man inte korrekt refererar till sina figurer eller tabeller, samt att de inte är korrekt formatterade. De ska alltid refereras till innan de presenteras och inte presenteras som ".. i figuren nedan." utan istället presenterar vi dem i stilen " i figur x presenteras .. ". Figurer har figurtext under figuren, tabeller har tabelltext ovan.

6. **Språket är inte korrekt.**
   Ni ska skriva med korrekt språk. Idag har alla textredigerare en nog sofistikerad stavningskontroll att ni inte bör ha stavfel i era rapporter. Ni får skriva på svenska eller engelska, men inte båda. Blanda inte ihop språken. Om ni vill använda ett engelskt ord, gör det på korrekt sätt. Använd inte talspråk i rapporten. Använd korrekt meningsuppbyggnad (håll reda på huvudsats och bisats. Undvik satsradning). Använd indrag eller blankrad för dela upp paragrafer, inte båda. Använd **inte** "jag".

7. **Tomma rubriker.**
   En rubrik ska alltid följas av mellanliggande text, inte följas direkt av en underrubrik!

8. **Ta inte skärmdumpar av kod.**
   Vi vill inte se skärmdumpar av kod. Använd hellre (i LaTeX) s.k. `lstlisting`, se www.overleaf.com för mer information. Se tidigare kodsnuttar för exempel på hur en kan presentera kod.

9. **Använd appendix.**
   Om ni ska presentera en stor mängd data, använd en appendix där ni bifogar den datan. Håll rapporten mer stilren. Håll även koll hur numrering av titlar ser ut för ett appendix. Se A för exempel av appendix.

# 5 Några avslutande kommentarer om kursen

Denna kurs är för många ett stort hopp från tidigare programmeringskurs. Att känna sig stressad över att hinna med eller att man inte kommer klara det hör till - men kom ihåg hur många som klarat den tidigare. Man måste inte få godkänt direkt, utan det får ta flera försök om det så krävs. Ni är här för att lära er, se till att lära er så mycket ni kan så kommer det lösa sig. Att ta till med olika hjälpmedel för att klara kursen kan kännas lockande - men kom ihåg att det är bara er själva ni lurar.

Ta del av handledningen. Vi handledare (och lärare) på kursen är på den här kursen för er skull. Vi hjälper er gärna, men vi kommer inte lösa uppgifterna åt er.

Datastrukturer och Algoritmer är en kurs som anses som en grundsten för datavetenskapliga utbildningar - över hela världen. Det innebär också att det finns otroligt mycket material på ex. `YouTube` för er att ta del av och lära er utav. Att lära sig grunderna i DoA underlättar vidare studier avsevärt. Ni bör dock inte ersätta kurslitteratur eller föreläsningar med `YouTube`, då vi kan gå in på mer specifika detaljer relevanta för denna kurs.

Slutligen, ha kul och kom ihåg att ibland kan det bästa man kan göra är att ta en paus från koden (bara ni kommer tillbaka sen!).

# A En fin text

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Hac habitasse platea dictumst vestibulum rhoncus est pellentesque. Metus vulputate eu scelerisque felis imperdiet proin. A cras semper auctor neque vitae tempus quam. Feugiat in fermentum posuere urna nec tincidunt praesent. Dolor sit amet consectetur adipiscing elit pellentesque habitant. Tincidunt id aliquet risus feugiat in ante. Et tortor consequat id porta. Ridiculus mus mauris vitae ultricies leo integer malesuada nunc vel. Convallis tellus id interdum velit laoreet id. Consequat id porta nibh venenatis cras sed. Diam quis enim lobortis scelerisque fermentum dui faucibus in ornare. In pellentesque massa placerat duis ultricies lacus sed.

Fames ac turpis egestas integer eget. At volutpat diam ut venenatis tellus in. Morbi tempus iaculis urna id. Eleifend donec pretium vulputate sapien nec sagittis aliquam malesuada. Posuere ac ut consequat semper. Eget sit amet tellus cras adipiscing enim eu turpis. Id volutpat lacus laoreet non. Mattis rhoncus urna neque viverra justo. Faucibus scelerisque eleifend donec pretium vulputate sapien nec. Eget nunc lobortis mattis aliquam faucibus purus. Mattis vulputate enim nulla aliquet porttitor lacus. Malesuada bibendum arcu vitae elementum curabitur vitae. Sed viverra tellus in hac habitasse. Dis parturient montes nascetur ridiculus. Facilisis sed odio morbi quis. Netus et malesuada fames ac turpis egestas. Aliquet eget sit amet tellus cras adipiscing. Purus in massa tempor nec feugiat nisl pretium fusce.

Porttitor massa id neque aliquam vestibulum morbi. Pharetra et ultrices neque ornare aenean. Ac orci phasellus egestas tellus rutrum tellus pellentesque. Id consectetur purus ut faucibus pulvinar elementum. Tincidunt ornare massa eget egestas purus viverra accumsan in nisl. Ut eu sem integer vitae justo eget magna fermentum. Non tellus orci ac auctor augue mauris augue neque. Morbi leo urna molestie at elementum eu facilisis sed. Morbi tristique senectus et netus et malesuada fames ac turpis. Faucibus nisl tincidunt eget nullam. Ac tortor dignissim convallis aenean. Tincidunt dui ut ornare lectus sit amet est. Odio pellentesque diam volutpat commodo sed egestas egestas fringilla phasellus.