

En 2⁵-sidors första titt på ganska mycket

Denna uppgift är egentligen en samling av diskreta ”småuppgifter” och syftar till att ge en överblicksbild av många av de verktyg som kursen omfattar så tidigt som möjligt, och samtidigt orientera i C. Lejonparten av denna uppgift består av instruktioner som skall följas. Successivt får uppgiften karaktär av att du själv måste tänka och utföra något. Inför varje moment finns det pekare till material, t.ex. screencasts, som du bör titta på innan du går vidare. Tabellen nedan visar de moment som ingår i uppgiften.

Verktyg	Introduktion till C	Programmeringsmetodik
Emacs	Minneshantering/Malloc	Läsbarhet
GDB	I/O	Kodgranskning
L ^A T _E X	Rekursion vs iteration	Enhetstestning
Make	Arrayer och strängar	Parprogrammering
Terminal	Kompilering	Versionshantering
Git	Debuggning	Testdriven utveckling
Valgrind		

Notera att ingenting i det här dokumentet behöver redovisas.


A. Förberedande steg

- 1.) Hitta någon att jobba med för den här uppgiften (sprint 0, fas 0)¹. Senare i kursen måste din labbpartner vara någon i din grupp, men för den första veckan kan du alltså jobba med vem du vill.
- 2.) Logga in på en lämplig Unixmaskin.
- 3.) Starta ett terminalfönster.

B. Emacs

Lämplig förberedelse är att titta på screencasts Emacs 1 och 2.

Nu skall du få bekanta dig med Emacs, som är en texteditor för bland annat programmering. Emacs är skrivet i C och Emacs Lisp (Elisp) och består av en ”liten” kärna runt vilken tusentals paket av Elisp är virade för att tillhandahålla utökad funktionalitet, som t.ex. färgläggning av nyckelord, stöd för att navigera mellan kodfiler som refererar till varandra, kompilering, felsökning och mycket annat. I texten nedan kommer vi att använda Emacs syntax för kortkommandon. Så här tolkar du dem:

M-x Håll ned meta-tangenten (I Sunray-maskinerna i datorsalarna på Polacksbacken har dessa symbolen , på andra tangentbord motsvarar meta ofta **Alt** eller **Cmd**) och tryck x

C-x + **b** Håll ned kontrolltangeten (**ctrl**) och tryck x, släpp, tryck b

C-x + **C-f** Håll ned kontrolltangeten och tryck x, släpp, håll ned kontrolltangeten igen och tryck f

Nu börjar vi!

- 1.) Starta Emacs från terminalen: `emacs &`
- 2.) Titta runt litet i menyerna – vad finns det för alternativ?
- 3.) Nu skall vi skapa en ny fil. Det gör man genom att öppna en fil och använda ett namn som inte finns. Öppna en fil med **C-x** + **C-f**. Döp filen till något godtyckligt som slutar med `.c`, till exempel `foo.c` och lägg den i din hemkatalog.
- 4.) Notera att Emacs nu har gått in i ”c-mode” och förväntar sig att du skall skriva C-kod. Prova att skriva några C-nyckelord (t.ex. **int**, **void**, **return**, **for**, **while**, **case**) och se hur de färgläggs.
- 5.) Skriv av följande program (utan radnumreringen):

¹Det är alltid kaos när en kurs startar. Många av delarna av denna uppgift går i värsta fall att göra på egen hand, men kvaliteten på inläringen blir i regel bättre när man är två och diskuterar allt tillsammans.

```

1  #include <stdio.h>
2  int main(void) {
3      puts("Hello, world\n");
4      return 0;
5  }

```

Tryck **C-x**+**C-s** för att spara.

- 6.) För att kunna köra programmet måste vi kompilera det. Låt oss pröva att göra det inifrån Emacs. Vi måste nu utföra ett så kallat utökad kommando: **M-x** compile **↵**. Dessa symboler betyder tryck på meta-tangenten och skriv sedan compile och tryck på enter. Notera att vid **M-x** så flyttas fokus ned till den sk "minibuffern" längst ned på skärmen.

Emacs föreslår nu att kompilera filen med kommandot `make -k`. Det fungerar inte ännu på grund av att det inte finns en s.k. Makefil, vi skall bygga en sådan senare. Ersätt kommandot med `gcc foo.c` (där `foo.c` alltså är vad du döpte filen till). Tryck sedan enter.

När du gjort detta bör programmet ha kompilerats och Emacs öppnar en ny buffer som meddelar detta eller visar eventuella felmeddelanden.

Notera att Emacs har stöd för automatisk komplettering av kommandon med **→** (tabtangenter). Om du skriver `compi` och trycker **→****→** så kommer Emacs att visa alla utökade kommandon som börjar med `compi`. Om det bara finns ett alternativ räcker det med att trycka **→** en gång så skriver Emacs resten av kommandot själv. Du kan pröva detta med t.ex. **M-x** auto-f **→** som bör expandera till `auto-fill-mode`². Tryck sedan **↵** för att utföra kommandot.

- 7.) Lås oss köra programmet. Det kan vi göra genom att göra **M-!** `./a.out` **↵**. Nu bör texten "Hello, world" visas i minibuffern.
- 8.) Ett annat sätt att köra programmet är att starta en terminal inuti Emacs. Kommandot **M-x** eshell **↵** gör detta och du hamnar i den aktuella katalogen. Nu kan du skriva `./a.out` för att köra programmet och se resultatet utskrivet precis som på en vanlig terminal.
- 9.) Fortfarande i eshell, pröva att skriva `grep puts foo.c`. Kommandot `grep` listar alla rader i en fil som innehåller ett visst ord (i detta fall "puts") men när du kör det i eshell visas output i en särskild Emacs-buffert där varje rad är klickbar och tar dig till rätt rad i filen i fråga. Navigera på detta sätt till raden där "Hello, world" skrivs ut i `foo.c` och ändra till ett annat meddelande. Kompilera om enligt ovan. För att köra programmet igen, tryck **C-x**+**b** för att byta till en annan buffert. Emacs föreslår den senaste aktiva buffern, men skriv `*eshell*` (minns att du kan använda **→**) och tryck enter för att komma dit igen. Du kan använda **↑** för att navigera tillbaka till kommandot `./a.out` och köra det igen och se det nya meddelandet.
- 10.) Nu skall vi pröva några enkla kommandon:

- a.) Pröva att trycka mellanslag framför return i din .c-fil och tryck **→**. Vad händer?
- b.) Skapa en ny buffert **C-x**+**b** temp **↵**. Skriv följande (observera användandet av små och stora bokstäver):

```

hej Hopp sköna min
Vi skall dansa i det gröna

```

- c.) Placera markören på h i hej och tryck **M-c**. Vad händer?
- d.) Tryck sedan **M-l** och sedan **M-u**. Vad händer?
- e.) Tryck sedan **M-t**. Vad händer?
- f.) Om vill ångra någon ändring, pröva **M-x** undo **↵**.
- g.) Gå till början av denna rad med **C-a** (**C-e** går till slutet av raden).
- h.) Gå till nästa rad genom att trycka **↓** eller **C-n**.
- i.) Byt plats på raderna genom att trycka **C-x**+**C-t**.
- j.) Om du vill veta mer om vad ett kommando gör, pröva **C-h**+**k** och tryck sedan kortkommandot så visas en förklaring, samt hur man kan köra kommandot med hjälp av **M-x**.
- k.) Det omvända fungerar också, ta reda på kortkommandot för undo genom att trycka **C-h**+**f** undo **↵**.

²Med detta läge påslaget radbryter Emacs i textfiler. Detta är inte så vettigt i programkod så slå av det igen med **M-x** auto-fill-mode **↵** om du har slagit på det!

- 11.) Emacs kan betydligt mer spännande textmanipulering. Titta t.ex. här (YouTube-länkar):
 - 1.) Multiple Cursors Mode
 - 2.) Yasnippets
- 12.) Låt oss så till sist göra några inställningar i Emacs som är persistenta mellan körningar. Nu vill vi skapa filen `.emacs.d/init.el`. Det kan vi göra t.ex. så här:

```
C-x + C-f ~/.emacs.d/init.el ↵
M-x make-directory ↵ ↵
```

I den tomma filen, skriv sedan:

```
1 (menu-bar-mode -1)
2 (tool-bar-mode -1)
3 (scroll-bar-mode -1)
```

Detta kommer att *stänga av* scrollbar (aka rullningslistorna), verktygsfältet och menyraden i Emacs så att det ser ut som i kursens screencasts.

Du behöver dock inte stänga av Emacs och starta om det för att ladda dessa inställningar. Istället kan du köra:

```
M-x eval-buffer ↵
```

Voila!

Att ta med sig:

1. Emacs är din vän
2. Man kan göra oerhört kraftfulla texttransformationer med Emacs
3. Var rädd om dina handleder – undvik att hoppa mellan tangentbord och mus, och försök att hålla dina fingrar på "home row" så mycket som möjligt

C. Versionshantering & Git

För en video-introduktion, see även screencast VC 1.


Versionshantering är en grundläggande aspekt av all programutveckling. Enkelt uttryckt handlar versionshantering om stöd för hantering av kataloger med filer som förändras över tid. Förändringar loggas i ett så kallat repository³, löpande, manuellt av programmeraren, med förklarande loggmeddelanden. Vid behov kan man återskapa gamla versioner av filer (t.ex. för att återuppliva en bit kod som man trodde att man kunde ta bort, eller gå tillbaka till en fungerande version av en funktion). Eftersom versionhanteringssystemet kommer ihåg allt kan man hålla rent och snyggt i sitt program – vid behov kan det återskapas. Loggfilen är viktig för att kunna spåra förändringar och för att navigera bakåt i historiken. Här är t.ex. loggarna för den L^AT_EX-fil som jag just nu skriver dessa ord i (nyast högst upp):

```
First draft of VC assignment done.
----
First draft of Emacs assignment 1.
----
Finished with some kind of first run-through of all the steps.
Decided on the content on about the first 3rd.
----
Sketches for a few of the first tasks
----
[Initial] A collection of smaller assignments to start the course.
```

³En plats där skillnaderna mellan alla versioner av alla filer sparas så att man kan gå fram och tillbaka i förändringshistoriken. Repo:t kan vara en katalog på din egen hårddisk, ligga på en separat dator eller en kombination.

Viktigt är att versionhanteringssystem tillåter programmerare att ha varsin lokal kopia av alla filer i ett program och få automatiskt stöd för att slå ihop förändringar som gjorts parallellt i olika kopior av samma fil, etc. Ett sätt att samarbeta distribuerat utan behov av att det finns en speciell "masterkopia".

I denna enkla instruktion till Git kommer vi att skapa två repositorys med inställningar för Emacs och länka samman dessa repositorys så att man enkelt kan kopiera information mellan dem och på så sätt hålla dem synkroniserade.

- 1.) I ett terminalfönster (t.ex. eshell i Emacs), placera dig i din hemkatalog genom att köra kommandot `cd`
- 2.) Gå in i katalogen `.emacs.d` med kommandot `cd .emacs.d`
- 3.) Initiera denna katalog som ett tomt repo med `git init`
- 4.) Markera att `init.el` som du tidigare skapat skall versionshanteras med `git add init.el`. Filer som inte explicit markeras för versionshantering kommer att ignoreras av systemet. *Notera att du måste markera `init.el` för versionshantering varje gång du vill checka in en förändring till denna fil!*
- 5.) Checka in den nuvarande `init.el` i repot med `git commit -m "Initial commit"`. Nu är filen incheckad och sparad.
- 6.) Editera nu `init.el` lägg till en ny textrad "; Kortlivad". Nu skall vi undertrycka denna ändring genom att återskapa filen från repot med `git checkout init.el`. Verifiera att texten "; Kortlivad" har försvunnit. OBS! Om filen är öppen i Emacs måste du göra `M-x revert-buffer`  för att läsa in filen på nytt från disk eftersom Emacs håller filens innehåll i minnet.
- 7.) Titta på loggen för `init.el` med `git log init.el` eller `git log` som visar hela loggen (ekvivalent eftersom det bara finns en fil för närvarande).
- 8.) Gissningsvis är ni två som gör detta (*Bonnie och Clyde*), men alla ändringar och filer finns nu bara hos en person (*C*). I detta steg skall vi försöka råda bot på detta med hjälp av Git.

Öppna ett nytt terminalfönster och byt aktuell användare för detta fönster till *B* genom att logga in på någon dator vid IT med hjälp av SSH, t.ex. `ssh abcd1234@celsius.it.uu.se` där `abcd1234` ersätts med *B*s användarnamn. Hädanefter refererar vi till det fönster där *B* är inloggad som "*B*" och det fönster där *C* är inloggad som "*C*". Ställ båda dessa fönster i katalogen `.emacs.d` (t.ex. med `cd; cd .emacs.d`).

I *C*, kör kommandot `pwd` som skriver ut den absoluta sökvägen till den aktuella katalogen.⁴

I *B*, kör kommandot `git clone dir` där `dir` är sökvägen ni nyss fick ut. Detta klonar (kopierar) all information om repot från *C* till *B*. Så skapas alltså en katalog `.emacs.d` även hos *B*. Gå in i denna katalog med `cd .emacs.d` och lista filerna med `ls -l` för att kontrollera att de är där.

Kör `emacs -nw` för att starta upp en icke-grafisk instans av Emacs i *B*. Om rullningslistor, meny och verktygsrad inte syns har inställningarna lästs in som avsett. Öppna `init.el` på nytt och lägg till följande:

```
1 ;; Turn off annoying splash screen
2 (setq inhibit-splash-screen t)
3 ;; Turn on line number and column number modes
4 (column-number-mode t)
5 ;; Turn on parenthesis mode
6 (show-paren-mode t)
```

Kör nu `git add init.el` för att markera att du kommer vilja checka in ändringarna i filen, och utför sedan incheckningen med `git commit -m "Some additional essential settings"`. Nu är den nya revisionen incheckad hos *B* men inte hos *C*. För att göra denna ändring synlig hos *C*, har vi två möjligheter.

Alt 1.) *B* "knuffar" över ändringar med `git push`. Detta alternativ är du kanske bekant med om du använt t.ex. Subversion eller CVS förut, men detta kräver att man initierar sitt git-repo på ett speciellt sätt som inte är lika enkelt. **Därför väljer vi istället alternativ 2.**

Alt 2.) *C* "drar" över ändringar med `git pull dir` där `dir` är sökvägen till *B*'s repo.

⁴**OBS!** För att denna övning skall fungera krävs att *B* har läsrättigheter till katalogen som innehåller *C*'s git-repo, och tvärtom. Om problem med filrättigheter uppstår, prova: `chmod 711 ~;` `chmod -R a+r ~/.emacs.d`. Om git-repot (i detta fall `.emacs.d`) inte ligger direkt under din hemkatalog (`~`), utan t.ex. i underkatalogerna `~/foo/bar/.emacs.d` måste du också göra `chmod a+x ~/foo` och `chmod a+x ~/foo/bar` i anslutning till föregående kommandon.

- 9.) Varför behövs ett `dir`-argument om *C* ska dra med `pull` men inte om *B* ska knuffa med `push`? Gör `cat .git/config` i *B* – där finns informationen om var repot är klonat från och det är denna information som läses när man gör `push` (eller `pull`). Motsvarande information saknas dock i *C*. För att enkelt kunna dra information från varandras repon utan att manuellt ange sökväg varje gång måste man lägga till en motsvarande fil hos *C*. Gör det med hjälp av Emacs och skriv av *B*s fil från grunden, men ändra sökvägen som står efter `url` så att den går till *B*s repo istället.
- 10.) Experimentera med att göra simultana förändringar i `init.el`, checka in och dra. Under vilka omständigheter kan Git räkna ut hur flera versioner av en fil enkelt kan slås samman, och under vilka omständigheter kommer den anse att filerna är ”i konflikt”?
Lös eventuella konflikter manuellt i Emacs, eller prova att köra `git mergetool --tool emerge` för att öppna Emacs i ett speciellt git-konflikt-läge. Läs också på nätet vid behov.
- 11.) Emacs innehåller ett fantastiskt stöd för Git i form av `magit`. Vidare läsning om `magit` finns på nätet, inklusive många bra video-tutorials.

Att ta med sig:

1. Git är din vän
2. Versionshantering är ett kraftfullt verktyg för att samarbeta med text
3. Möjligheten att återskapa filer och ångra stora ändringar över många filer är en livräddare många gånger

Nu är det dags att hämta kursens repo. Den URL:en är gömd och hämtas fram i nästa uppgift.

D. Terminalhantering

När man är van vid en grafisk filhanterare (som Finder i Mac OSX eller Explorer i Windows) kan det verka omständigt att lära sig använda terminalen. Många känner sig tryggare när de kan se allting som ikoner och klicka med musen, och så länge det enda man gör är att öppna och kanske flytta enstaka filer så är lidandet begränsat. Men när man ska göra mer avancerade saker så är terminalen ett ovärderligt verktyg!

Tänk dig till exempel att du har en mapp med 100 mappar som *bland annat* innehåller mp3-filer och att du skulle vilja flytta alla dessa till en egen mapp. I ett grafiskt användargränssnitt skulle du behöva gå in i varje mapp separat, markera alla mp3-filer och dra över dem till den nya mappen. Innan du är klar med den första mappen så är någon som kan använda terminalen redan klar genom att ha skrivit⁵

```
foo> mv */*.mp3 music/
```

Innan du är klar med den andra mappen har terminalfantomen redan utmanat dig till att hitta hård-diskens alla textfiler som innehåller ordet ”choklad”. Du hinner knappt uppfatta frågan innan hen har skrivit

```
foo> find / -type f -name "*.txt" | xargs grep -l choklad
```

OBS! Oroa dig inte om du inte förstår de här kommandona än!

Terminalfantomen vill utmana dig igen, men du har fattat poängen och lovar dig själv att du ska lära dig att använda terminalen.

Att navigera och söka bland text och filer

Nu ska vi prova på att använda terminalen för att hitta en specifik rad text bland en massa filer som ligger i en enda röra. Texten vi letar efter är URLen för kursens repo!



- 1.) Börja med att öppna en terminal (om du inte redan har en öppen)
- 2.) Ta reda på vilken mapp du befinner dig i med kommandot `pwd` (som står för *print working directory*).

⁵Vi använder konsekvent `foo>` för att representera terminalprompten. Det är alltså ingenting du skall skriva själv.

```
foo> pwd
/home/abcd1234
```

- 3.) Med `cd` (*change directory*) byter du mapp till det som står efter kommandot. Om du inte skriver någonting efter `cd` hamnar du i din hem-mapp. Vi ska dock till följande mapp:

```
foo> cd /it/kurs/imperoopmet/labyrinth
```

Kom ihåg att du nästan alltid kan använda  för att komplettera något du har skrivit eller se vilka alternativ som finns. Till exempel behöver du inte skriva ut ordet "labyrinth", det räcker med att skriva "lab" och trycka .

Någonstans i den här labyrinten finns URLen vi letar efter. Med hjälp av terminalen ska vi hitta den!

- 4.) Kommandot `ls` listar alla filer i mappen du befinner dig i.

```
foo> ls
east          north          rK8Mmp          south          VEN0z1.txt  west
```

Man kan också lista alla filer i en eller flera andra mappar

```
foo> ls north
east          n6qlnh          south          Y24dr8
N53t2F.txt    north          west
foo> ls north south
north:
east          n6qlnh          south          Y24dr8
N53t2F.txt    north          west

south:
east          north          sZh7pv.txt    west
FPh8zv.txt    south          UM9BvA        XOUwPV.txt
```

- 5.) I alla mappar finns två namn som är speciella. Namnet `.` betyder "mappen jag står i" och `..` "mappen ovanför den jag står i". Gå till mappen `north/west` och lista innehållet i den mappen.

```
foo> cd north/west
foo> ls
9RlYdp.txt    laKnns.txt    P6roYu.txt    READMETHREE    south
east          north          README        README TOO     west
```

Använd `cat` för att visa innehållet i `README`. Använd sedan `cd ../..` för att komma tillbaka för att komma tillbaka till mappen `labyrinth`.

- 6.) Det finns det väldigt många filer under `labyrinth`. För att se precis hur många filer kan du prova kommandot `find .` som listar alla filer under sitt första argument (här `..`, alltså mappen vi står i).
- 7.) Som namnet antyder kan `find` användas för att hitta filer. Vi ska nu begränsa vilka filer `find` visar. Till att börja med vill vi ange att vi bara är intresserade av "vanliga filer" och inte till exempel mappar:

```
foo> find . -type f
...
```

Hade vi skrivit `find -type d` hade vi istället bara visat mappar (directories).

- 8.) Nu ska vi dessutom ange att vi bara är intresserade av filer som slutar på `".txt"`:

```
foo> find . -type f -name "*.txt"
...
```

Stjärnan är ett så kallat *wildcard* som betyder "vad som helst".

- 9.) Ett väldigt kraftfullt verktyg i terminalen är möjligheten att koppla ihop flera kommandon. Det gör man med symbolen `|`, som kallas "pipe". Kommandot `find` producerar en lista med filnamn. Prova att skicka den listan till kommandot `wc` (*word count*) för att se hur många rader listan består av:

```
foo> find . -type f -name "*.txt" | wc -l
519
```

Flaggan `-l` anger att vi bara är intresserade av antalet rader (och inte antalet ord och tecken, som `wc` också visar annars).

- 10.) Om vi istället vill ge varje rad i listan som `find` producerar som argument till `wc` använder vi kommandot `xargs`. `xargs` tar ett annat kommando som argument och väntar sedan på input att mata till kommandot. För att räkna antalet rader i varje enskild fil skriver man så här:

```
foo> find . -type f -name "*.txt" | xargs wc -l
...
```

För att skriva ut alla filer som `find` hittar använder vi `xargs` med `cat` som argument:

```
foo> find . -type f -name "*.txt" | xargs cat
...
```

Nu kan vi alltså läsa alla textfiler i labyrinten, men det är fortfarande väldigt mycket text att gå igenom för hand (lägg till `| wc -l` till ovanstående kommando för att se hur mycket!).

- 11.) Kommandot `grep` används för att söka efter en textsnitt i en fil. Skriver man `grep foo file.txt` så visas alla rader i filen `file.txt` som innehåller teckenföljden `foo`. Eftersom vi är intresserade av att hitta en URL så ber vi `grep` leta efter "http" i alla filer som `find` hittar:

```
foo> find . -type f -name "*.txt" | xargs grep http
...
```

- 12.) Nu är det "bara" 1014 rader att gå igenom (istället för 51960), men vi ska förfinas sökningen ytterligare en sista gång. Eftersom vi använder Git för kursens repo kan vi söka efter det ordet i vår resultatlista genom att använda `grep` igen. Vi lägger också till flaggan `-h` på den första förekomsten av `grep`, som gör så att `grep` inte skriver ut vilka filer resultaten kommer ifrån:

```
foo> find . -type f -name "*.txt" | xargs grep -h http | grep git
https://github.com/TobiasWrigstad/ioopm14
```

- 13.) Success! Gå nu tillbaka till din hemkatalog (kommandot `cd`) och checka ut kursens repo. Eftersom Gits egna protokoll⁶ är något snabbare kommer vi använda det istället för `https`:

```
foo> cd
foo> git clone git://github.com/TobiasWrigstad/ioopm14
```

Nu borde du ha en mapp som heter `ioopm14` i din hemkatalog. Ta och utforska den med dina nya terminalkunskaper! Om du stöter på en PDF som du vill läsa kan du öppna den med kommandot `acoread file.pdf &`.

Som du säkert förstår går det att göra mycket, mycket mer i terminalen (titta på man-flerna för de här få kommandona för att få en uppfattning av hur stora möjligheterna är, se avsnitt G). Var inte rädd att googla eller fråga om du kommer på något du skulle vilja göra i terminalen. Ju mer du håller på desto mer naturligt kommer det kännas!

⁶Se <http://git-scm.com/book/en/Git-on-the-Server-The-Protocols>

Don't fear the void!

Det kan också vara bra att veta terminalprogram som inte är gjorda för att skriva ut saker oftast bara gör ljud (alltså text) ifrån sig när något har blivit fel. Om du till exempel ska kopiera tusentals filer och du får upp en ny prompt utan något meddelande så har det alltså förmodligen lyckats.

Vill man lära sig mer om terminalen finns det gott om resurser på Internet. Se (till exempel) http://linuxcommand.org/learning_the_shell.php för en mer utförlig genomgång.

Att ta med sig:

1. Terminalen är din vän
2. Ett interaktivt terminalgränssnitt är oerhört kraftfullt för interaktion med ett filsystem eller ett helt operativsystem
3. Försök att automatisera arbetsflöden genom att skriva sekvenser av kommandon i en fil, ett så-kallat "shell script"

E. Mer Emacs

- 1.) I ditt nyligen hämtade kursrepo finns ett antal lathundar, bland annat en för Emacs där man kan läsa om några av de vanligaste kommandona. Längst bak i det dokumentet finns ett referensblad som man kan skriva ut och ha liggandes bredvid datorn när man arbetar. Hitta den här filen, antingen manuellt med `cd` och `ls` eller genom att söka efter "emacs.pdf" med `find`. Öppna den med `acoread`.
- 2.) Det finns också en `init`-fil för Emacs som innehåller fler praktiska inställningar för Emacs. Du hittar den genom att söka efter "`init.el`" med `find`. Du kan kopiera de inställningar du vill ha till din egen `init`-fil som du skapade i steg B (eller ersätta hela din fil med den här). Filen är självdokumenterande och kommer att uppdateras under kursens gång.

Att ta med sig:

1. Varje gång du ska göra något i Emacs, om det så bara är att flytta markören, fundera på hur du kan göra det med så få knapptryckningar som möjligt
2. Du kan konfigurera Emacs precis hur du vill
3. Emacs innehåller stöd för i stort sätt allt som finns i t.ex. Netbeans eller Eclipse, plus en massa annat

Tips! Ett utmärkt sätt att lära sig hur kraftfullt Emacs faktiskt är, är att bestämma sig för att inte lämna det under en vecka. Att inte lämna Emacs betyder förstås att använda Emacs för allt inklusive skicka och läsa epost, surfa, filhantera, etc. Det är inte helt enkelt men lärorikt!

F. Kompilera ett C-program

Lämpliga förberedelser: titta på screencast C 1 a–c.

- 1.) Ett C-program består av en eller flera källkodsfiler. Filer med ändelsen `.c` innehåller körbar kod medan filer med ändelsen `.h` (header-filer) normalt endast innehåller deklarationer och metadata om programmet. Vi kommer att använda oss av båda dessa under kursen, men initialt fokuserar vi på `.c`-filerna.

Vid kompilering översätts en eller flera källkodsfiler till *objektkod* som är ett mellansteg på vägen till ett körbart program. Objektkodsfiler innehåller maskinspecifika instruktioner skapade från källkoden, men för att bli körbara måste dessa *länkas* samman med varandra och de maskinspecifika standardfunktioner som programmet använder. Kompilering och länkning är två olika steg, men vi kommer i regel att utföra dem direkt efter varandra så att de framstår som ett enda.

Under kompileringssteget försöker kompilatorn identifiera fel i källkoden (t.ex. du anropar funktionen *f* med för få argument). Under länkningen uppstår fel om inte alla beroenden är lösta (t.ex. du anropar en funktion *f* men det finns ingen sådan funktion). En anledning till att ett

beroende inte är löst är att man kanske stavat fel till *f* alternativt att man glömt att inkludera den fil som innehåller *f* i sin kompilering.

I denna uppgift skall vi steg för steg bygga upp en "trollformel" som med fördel kan användas för all kompilering under denna sprint. Programmet som vi skall kompilera finns i `ioopm14/uppgifter/fas0/sprint0/kod/` och heter `passbyval.c`. Börja att ställa dig i katalogen `ioopm14/uppgifter/fas0/sprint0/kod`.

- 2.) Kompilera programmet med hjälp av C-kompilatorn GCC:

```
foo> gcc passbyval.c
```

Du bör nu få ett felmeddelande "error: 'for' loop initial declaration used outside C99 mode". Detta felmeddelande får du för att koden använder sig av den C-standard som fastslogs 1999 (en ny C-standard fastslogs 2011⁷, men den har ännu inte stöd bland de flesta stora C-kompilatorer). Du måste ange explicit att du vill tillåta sådana "moderna features":

```
foo> gcc -std=c99 passbyval.c
```

Nu bör koden kompilera som den skall.

- 3.) Gör `ls -l` för att lista innehållet i den nuvarande katalogen. Lägg märke till filen `a.out` – det är det kompilerade programmet. Du kanske är van från OS X att körbara program har filändelsen ".app" eller från Windows, ".exe"? Om man inte explicit anger ett namn på det kompilerade programmet får det namnet `a.out`, och körs så här (kör inte ännu!):

```
foo> ./a.out
```

Notera att vi vill att du anger `./` vilket betyder "i denna katalog", vilket hindrar att du råkar köra ett annat program som heter `a.out` som ligger någonstans i din `path`⁸ och som därför inte nödvändigtvis gör det du vill.

- 4.) Kompilera om programmet och döp den exekverbara filen till `passbyval`:

```
foo> gcc -std=c99 -o passbyval passbyval.c
```

- 5.) Normalt när vi kompilerar vill vi att kompilatorn skall vara maximalt känsligt för eventuellt knas och vi vill därför slå på "alla varningar":

```
foo> gcc -std=c99 -Wall -o passbyval passbyval.c
```

- 6.) Vidare vill vi slå på maximalt stöd för debuggning, och eftersom vi använder gcc skall vi speciellt dra nytta av "gdb", GNUs debugger:

```
foo> gcc -std=c99 -Wall -ggdb -o passbyval passbyval.c
```

Nu har vi kommit fram till den "trollformel" som, med undantag för namnet "passbyval", är den ramsa du kommer att upprepa vid varje kompilering av ett enkelt program.

- 7.) Om ditt program består av flera källkodsfiler som skall byggas till en enda exekverbar fil kan du ange samtliga dessa filer direkt vid kompilering:

```
foo> gcc -std=c99 -Wall -ggdb -o passbyval passbyval.c more.c another.c etc.c
```

Lägg på din att-göra-lista att se på våra screencasts om separatkompilering och makefiler som tar kompilering till nästa nivå.

- 8.) Nu är det dags att köra programmet (det kanske du redan har gjort). Du kommer då att märka att det inte gör rätt. Det är avsiktligt – det leder oss nämligen in på nästa ämne: debuggning.

⁷http://en.wikipedia.org/wiki/C11_%28C_standard_revision%29

⁸Miljövariabeln `PATH` avgör var terminalen ska leta efter program. Kommandot `echo $PATH` skriver ut dess värde.

Att ta med sig:

1. Kompilatorn är din vän, och den hittar många dumma fel – men inte alla
2. Kompilera *ofta* för att minimera mängden felutskrifter vid varje givet tillfälle, och slå på maximalt med varningar för att fånga alla fel
3. Om du får varningar eller kompileringsfel – se till att alltid åtgärda dem innan du ”fortsätter”
4. Kompileringsfelen skall läsas uppifrån och ned!

G. Att läsa manual-sidor

”man-sidor” är smeknamn på ”manual-sidor” som är dokumentation och hjälpsidor för de verktyg och bibliotek vi använder i kursen. Om du vill läsa dokumentationen till programmet `passwd` för att söka efter hjälp skriver du:

```
foo> man passwd
```

Då öppnas en textfil som beskriver `passwd`. Filen visas i verktyget `less` eller `more`. Du kan navigera upp och ner genom tangenterna `[k]` respektive `[j]` eller med piltangenterna. Tryck `[q]` för att avsluta och återgå till terminalen.

Man-sidor är skrivna efter en mall. Detta gör det lättare att leta efter exempel, beskrivningar av argument till programmet etc. Några sådana avsnitt är: Namn (Name), sammanfattning (Synopsis), beskrivning (Description) och argument (Options).

Om du läser man-sidan för `passwd` igen ser du att det står `PASSWD(1)` högst upp till vänster. Det finns flera man-sidor för kommandot `passwd` som beskriver andra delar av programmet, prova att skriva `man 5 passwd`, vad beskriver den sidan som inte `man 1 passwd` gör? Du ser om en man-sida har flera sidor genom att söka efter avsnittet ”SEE ALSO”. Öppna man-sidan för `printf`, tryck `[/]` skriv ”SEE ALSO” eller bara ”SEE” och trycker `[Enter]`. Vilken mer man-sida finns för sökordet ”printf”? En man-sida beskriver terminal-kommandot `printf`; det andra C-biblioteksfunktionen ”printf”. Titta på exemplen i man-sidan för C-biblioteksfunktionen ”printf” (Sök efter ”EXAMPLES” enligt ovan).

H. Debugging

När ditt program avslutas oväntat eller det betar sig oförutsägbart hittar du felet mycket fortare och effektivare om du vet *var* felet uppstår och även *varför* felet uppstår. För detta använder vi programmerare ett debugging- verktyg (svenska: avlusning). Vi ska använda `gdb` som finns förinstallerat de allra flesta Linux-distributioner (Ubuntu, Debian, etc.) och finns tillgängligt till OSX⁹.

Programmet nedan använder en funktion `plusEtt` som ska öka värdet i den variabel som skickas in med ett. Kör programmet och observera att värdet (enligt utskriften) inte ökas med ett. Vi ska nu använda `gdb` för att hitta var felet uppstod i `passbyval.c` (t.ex. är felet i utskriften eller är det fel i logiken):

```
1  #include <stdio.h> // för printf
2
3  // ökar parametern a med ett som sido-effekt
4  void plusEtt(int a) {
5      a = a + 1;
6  }
7
8  int main() {
9      int a = 0;
10     for(int i = 0; i < 3; i++) {
11         plusEtt(a);
12         printf("%d + 1 = %d\n", i, a);
13     }
14     return 0;
15 }
```

Se till att programmet i fråga är kompilerat med flaggan `-ggdb` och starta `gdb` genom att köra kommandot `gdb passbyval` (förutsatt att `passbyval` är namnet du gav till det kompilerade programmet).

⁹Sedan OSX 10.9 har `gdb` ersatts av `lldb`. Gränssnittet är dock mycket likt `gdb`s

Funktionen `plusEtt` är huvudmisstänkt. Vi ber gdb pausa programmet vid rad 11 och 12 så vi kan titta på värdet i variabeln `a` före och efter funktionsanropet. Ställ in pauserna (breakpoints) genom att skriva:

```
(gdb) break 11
(gdb) break 12
(gdb) run
```

Notera att "(gdb)" är prompten, och inget du ska skriva själv. Efter `run` så kör gdb programmet och stannar vid rad 11. Undersök värdet på variabeln `a` genom att skriva

```
(gdb) print a
```

Är värdet som det ska? Be gdb fortsätta:

```
(gdb) continue
```

Undersök värdet igen och konstatera att funktionen inte gör vad den var tänkt att göra. Rensa pauserna och be gdb att stanna i funktionen:

```
(gdb) delete
```

Tryck "y" för yes om gdb frågar om du verkligen vill ta bort alla pauser.

```
(gdb) break plusEtt
(gdb) run
```

Om gdb frågar om den ska starta om från början så svara "y" för yes. När programmet stannar kan du själv undersöka vad som händer på varje rad genom att stega igenom funktionen:

```
(gdb) step
(gdb) step
(gdb) step
...
```

För att fortsätta köra programmet till nästa paus (eller tills programmet avslutas) används kommandot `continue`. Givetvis behöver vi inte vara så noga att stava rätt till "continue" och "step" varje gång utan gdb tillåter kortkommandot "c" för continue och "s" för "step". För att avsluta gdb skriver du "quit" eller "q".

Koden tycks onekligen lite ogenomtänkt eftersom funktionen inte förändrar värdet på variabeln `a` som är deklarerad i `main`-funktionen, utan bara sin egen lokala kopia (kom ihåg att värden kopieras till argumentet i funktionen vid funktionsanrop). Ett sätt att få koden att göra rätt är att skicka adressen till `a` som argument istället, och göra så att `plusEtt` ändrar på värdet som ligger på den här adressen. Ändra rad 11 till "`plusEtt(&a)`", och sätt en asterisk (*) framför alla förekomster av `a` i funktionen `plusEtt`¹⁰. Kompilera programmet och kör igen!

Ett annat sätt att utröna var problem uppstår är att låta gdb skriva ut vilka funktioner som programmet hoppade igenom för att ta sig till raden där problemet uppenbarar sig. En sådan sekvens av funktionsanrop kallas för ett *backtrace*. gdb visar den här sekvensen om debuggningen stannar någonstans och vi ger gdb kommandot `backtrace` eller `bt`. Nedan har programmet stannats efter anropet till `plusEtt`. Kommandot `bt` ger då

```
(gdb) bt
#0 plusEtt (a=0) at passbyval.c:5
#1 0x000000000400534 in main () at passbyval.c:11
(gdb)
```

vilket ska tolkas som att exekveringen av programmet gick från `main` (indikeras med ett s.k. frame-nummer: #1) till `plusEtt` (frame-nummer #0).

Om man har ett program som kraschar och kör det utan breakpoints i gdb så kommer programmet att pausa precis i ögonblicket där kraschen sker, så att man kan se vilken rad kraschen skedde på, vilka funktioner som hade anropats och vilka värden alla variabler hade.

¹⁰Oroa dig inte om du inte förstår hur det här fungerar än. Det kommer att förklaras senare i kursen!

Att ta med sig:

1. Debuggern är din vän
2. Kompilera alltid med debugflaggorna påslagna när du utvecklar
3. Att bara köra ett kraschande program i gdb för att se utskriften är ofta nog för att lösa felet

I. Typsättning med L^AT_EX

När du skriver rapporter och dokument är du kanske van att använda Word eller OpenOffice. I den här kursen ska vi istället använda L^AT_EX för all dokumentframställning. Att använda L^AT_EX påminner till stor del om att programmera eftersom att du skriver text som vanligt och använder L^AT_EX-kommandon för att byta typsnitt, storlek, bakgrund, sidhuvud, titel etc. En stor poäng med L^AT_EX är att de kommandon du skriver har karaktären "metadata" och inte "så här skall texten formateras". Du anger "detta är en rubrik på nivå 2", "detta är en lista", "detta är en sökväg till en fil", etc. Hur rubriker, listor och sökvägar slutligen skall formateras beskrivs på *ett enda ställe* och är ofta enkelt att byta ut genom att bara ändra på en enda rad i dokumentet.

Se följande exempel som skapar en punktlista:

```
\begin{itemize}
  \item Mother of Dragons
  \item Jon Snow
  \item Hodor
\end{itemize}
```

Vilket med standardinställningar för formatering ger

- Mother of Dragons
- Jon Snow
- Hodor

Byter man ut `itemize` mot `enumerate` får man en numrerad lista istället:

1. Mother of Dragons
2. Jon Snow
3. Hodor

Latex är ett typsättningssystem, där typsättningen syftar på att vi i exemplet typsätter "Mother of Dragons", "Jon Snow" och "Hodor" till en lista. Precis som andra språk som HTML så beskriver man början och slut av typsättningen med start- och stoppkommandon.

Nu tar vi det från början och skapar ett minimalt dokument i Latex.

```
\documentclass[a4paper]{article}

\begin{document}
  Hello World!
\end{document}
```

Den första raden (`\documentclass`) avser grundinställningar, t.ex. vilken papperstorlek som skall användas. Samma källkodsfil kan användas för att göra allt ifrån stora affischer och presentationer till små visitkort. Parametern `article` ger sidorna i dokumentet ett utseende som passar för "vanliga dokument".

Det som står mellan `\begin{document}` och `\end{document}` är det som kommer synas i det färdiga dokumentet.

Nu skall du kompilera ditt exempel:

```
foo> pdflatex doc.tex
```

Från en källkodfil spottar `pdflatex` ur sig en `.aux`-fil och en `.pdf`-fil. Ett bra tips är att ha `.pdf`-filen öppen samtidigt som du redigerar och kompilerar `.tex`-filen för att kunna se framstegen.

Typsätt nu rubriker av olika storlek:

```

\begin{document}

\section{Introduction}

It has begun!

\subsection{A subheading}

It goes on and on...

\subsection{Another subheading}

...and on and on.

\section{Ending}

It's all over!

\end{document}

```

Kompilera och titta på resultatet. Du kan lägga till ytterligare nivåer av rubriker med kommandona subsubsection och paragraph. Vad händer om du skriver `\section*` istället för `\section`?

Du kan kontrollera hur din text ska se ut på många olika sätt:

Some words are more `\emph{important}` than others.

Sometimes you might want a part of a sentence to `\textbf{stand out}`.

You can also write pretty math: $x^2 + 4x + 2 \leq \sqrt{\pi} \cdot |x|$

Koden som står mellan `\documentclass` och `\begin{document}` kallas för dokumentets preamble. Här kan du ställa in metainformation om dokumentet, importera paket och definiera egna kommandon. Prova att skriva följande kod i preambeln

```

\title{Baby's first steps in \LaTeX}
\author{Alice \and Bob}
\date{\today}

```

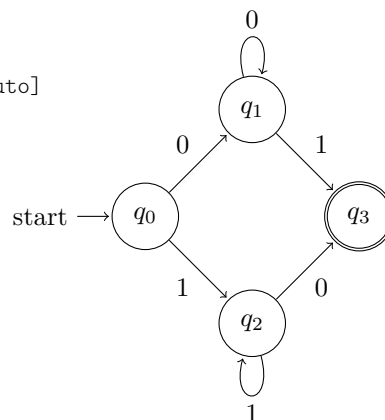
Lägg sedan in kommandot `\maketitle` direkt efter `\begin{document}`. Hur blev det? Du kan prova att byta ut `article` i `\documentclass{article}` till andra format som `t.ex. report` och `book`.

Även om inlärningskurvan bitvis kan vara lite brant så är \LaTeX ovärderligt när man ska typesätta dokument. Till exempel är dokumentet du läser nu skrivet helt i \LaTeX . För att vattna din tunga hoppar vi nu långt framåt och ger en försmak på möjligheterna! Här är ett exempel som använder stödpaketet Tikz för att rita en automat. (Resultatet till höger.)

```

\documentclass{article}
\usepackage{tikz}
\usetikzlibrary{automata,positioning}
\begin{document}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid,auto]
  \node[state,initial] (q_0) {$q_0$};
  \node[state] (q_1) [above right=of q_0] {$q_1$};
  \node[state] (q_2) [below right=of q_0] {$q_2$};
  \node[state,accepting] (q_3) [below right=of q_1] {$q_3$};
  \path[->]
    (q_0) edge node {0} (q_1)
    (q_0) edge node [swap] {1} (q_2)
    (q_1) edge node {1} (q_3)
    (q_1) edge [loop above] node {0} ()
    (q_2) edge node [swap] {0} (q_3)
    (q_2) edge [loop below] node {1} ();
\end{tikzpicture}
\end{document}

```



Fler otroligt tjustiga bilder kan du ögna igenom på <http://www.texample.net/tikz/examples/>

J. Grundläggande C-programmering

Lämpliga förberedelser: de flesta screencasts om C, inklusive enkla datatyper, loopar, variabler, funktioner, grundläggande I/O, arrayer. Om du inte vill se alla på en gång kan du t.ex. vänta med arrayer till just före det avsnittet, och likadant för strängar.

J.1. Iteration vs. rekursion

En rekursiv definition är rekursiv, det vill säga den refererar till sig själv. Ett exempel på en rekursiv definition är en definition av en lista som lyder ”en lista är antingen **tom** eller en *lista* plus ytterligare ett element”. För att en rekursiv definition skall vara sund behövs (minst) ett **basfall**, som inte refererar tillbaka till den term som man håller på att definiera. I vårt exempel säger vi litet slarvigt att ”den tomma listan är en lista”. Låt oss beteckna denna lista []. Nu kan vi använda den *rekursiva* delen av definitionen för att skapa en ny lista genom att lägga till ett element till den redan existerande, tomma, listan, t.ex. talet 42. Då får vi en lista [42]. Vi kan upprepa denna procedur och skapa en lista [42, 7, 5, 100] etc. En syntax som bättre fångar listans rekursiva natur är denna: (42 (7 (5 (100 ())))), där () avser en tom lista. Kan du se varför?

Definitionen av Fibonaccitalserien Fibonaccitalserien är ett klassiskt exempel på en rekursiv definition. Vilket/vilka är basfallet/basfallen? Vilket/vilka är det/de rekursiva steget/stegen?

Ett fibonaccital ingår i en sekvens av heltal, där varje tal är summan av de två föregående fibonaccitalen; de två första talen är 0 och 1. (Citatet taget från Wikipedia)

Från rekursiv definition till rekursiv implementation Nedanstående kod som också finns i kursrepot utgör en funktion skriven i C som räknar fram det n :te talet i Fibonaccitalserien. Implementationen är rekursiv. Studera hur koden förhåller sig till definitionen! Vilka delar av koden motsvarar basfall och vilka delar är rekursiva?

```

1  int fib(int n) {
2      if (n == 0) {
3          return 0;
4      } else {
5          if (n == 1) {
6              return 1;
7          } else {
8              return fib(n-1) + fib(n-2);
9          }
10     }
11 }
```

Ett enkelt sätt att beskriva en rekursiv implementation är i termer av mönstermatchning mot en rekursiv definition. Om vi återkopplar till exemplet med definitionen av fibonaccital kan vi se att vi för varje ”probleminstans” n först jämför n med våra två basfall (0 och 1) och i de fall de inte matchade går vidare till det rekursiva steget som kräver att vi räknar ut (eller känner till) de $n-1$:te och $n-2$:te fibonaccitalen. Notera hur ”problemet hela tiden minskar” – varje steg i rekursionen minskar fibonaccitalet som vi försöker räkna ut tills det blir 0 eller 1, alltså något av basfallen, för vilka svaren är givna från definitionen (”[...] de två första talen är 0 och 1.”). När vi reducerat problemet till något av basfallen kan vi konstruera ett svar. Den uträkning som det rekursiva programmet ”logiskt utför” skulle kunna sägas vara denna (för $n = 5$):

$$fib(5) = fib(4) + fib(3) \quad (1)$$

$$fib(4) = fib(3) + fib(2) \quad (2)$$

$$fib(3) = fib(2) + fib(1) \quad (3)$$

$$fib(2) = fib(1) + fib(0) \quad (4)$$

$$fib(1) = 1 \text{ enligt definitionen} \quad (5)$$

$$fib(0) = 0 \text{ enligt definitionen} \quad (6)$$

$$fib(2) = 1 + 0 \text{ enligt rad (5) och (6)} = 1 \quad (7)$$

$$fib(3) = 1 + 1 \text{ enligt rad (5) och (7)} = 2 \quad (8)$$

$$fib(4) = 2 + 1 \text{ enligt rad (7) och (8)} = 3 \quad (9)$$

$$fib(5) = 3 + 2 \text{ enligt rad (8) och (9)} = 5 \quad (10)$$

Vissa rekursiva funktioner kan implementeras med hjälp av iteration. Iterativa lösningar kan beskrivas som en loop som upprepar en *imperativ process*. En viktig anledning varför detta kan vara eftersträfvansvärt är att iteration oftast är mer effektivt (kortare körtid med mindre resurser) än rekursion.¹¹

Nedan följer en iterativ implementation av funktionen `fib`. Istället för att reducera problemet till ett basfall och sedan konstruera ett svar räknar denna implementation fram n :te fibonaccitalet genom att iterativt räkna fram `fib(2)` (med hjälp av basfallen), sedan `fib(3)` med hjälp av 2:a basfallet och uträkningen av `fib(2)`, etc.:

$$fib(0) = 0 \text{ enligt definitionen} \quad (1)$$

$$fib(1) = 1 \text{ enligt definitionen} \quad (2)$$

$$fib(2) = 0 + 1 \text{ enligt rad (1) och (2)} = 1 \quad (3)$$

$$fib(3) = 1 + 1 \text{ enligt rad (2) och (3)} = 2 \quad (4)$$

$$fib(4) = 1 + 2 \text{ enligt rad (3) och (4)} = 3 \quad (5)$$

$$fib(5) = 2 + 3 \text{ enligt rad (4) och (5)} = 5 \quad (6)$$

Denna uträkning ser trivial ut men implementationen följer inte samma struktur som den rekursiva definitionen av fibonaccitalserien och kan anses svårare att läsa:

```
1  int fib(int n) {
2      int a = 0;
3      int b = 1;
4      for (int i=1; i<=n; ++i) {
5          int fib_i = a + b;
6          a = b;
7          b = fib_i;
8      }
9      return a;
10 }
```

Iterativ lösning På papper, gå igenom vad som händer om man anropar `fib(5)` ovan. Utför varje rad för sig, ungefär som när du stegade igenom kod med gdb tidigare.

1. Hur förändras värdena på variablerna `a` och `b`?
2. Hur många gånger går programmet igenom loopen på rad 4–8?
3. Variablerna `a` och `b` har dåliga namn – finns det bättre namn på dem som skulle göra koden enklare att läsa? (Jämför t.ex. med variabeln `fib_i`) som avser det i :te fibonaccitalet.)

Notera: Du kan kontrollera att du har tänkt rätt genom att stega igenom programmet med gdb och inspektera värdena på variablerna mellan varje steg.

Modifierad iterativ lösning Öppna `ioopm14/uppgifter/fas0/sprint0/kod/fib-rec.c` ur kursens repo. Detta program skriver ut resultatet på följande format:

```
foo> ./fib-rec 5
Fib 5 = 5
foo>
```

Modifiera programmet så att samtliga beräknade fibonaccital skrivs ut. När programmet körs skall utskriften se ut så här:

```
foo> ./fib-rec 5
Fib 0 = 0
Fib 1 = 1
```

¹¹Avancerade kompilatorer försöker ofta översätta rekursiv kod till iterativ kod dolt för programmeraren av just detta skäl.

```
Fib 2 = 1
Fib 3 = 2
Fib 4 = 3
Fib 5 = 5
foo>
```

Ledning: Satsen som skriver ut text i terminalen heter `printf`. Du kan använda/flytta den rad som skriver ut det slutgiltiga resultatet i det ursprungliga programmet. Vi skall experimentera mer med utskrift senare i detta häfte.

Hur fungerar retursatsen? Nedanstående rekursiva funktion är ekvivalent med den första rekursiva fib-funktionen ovan. Studera skillnaderna och förklara varför funktionerna är lika!

```
1  int fib(int n) {
2    if (n == 0) {
3      return 0;
4    }
5    if (n == 1) {
6      return 1;
7    }
8    return fib(n-1) + fib(n-2);
9  }
```

J.2. Arrayer

En array i C kan ses som en ordnad samling värden av en viss typ. De påminner om listor (som i ML) men är olika i flera avseenden. Till skillnad från listor så ligger arrayer i ett enda sammanhängande stycke i minnet, vilket gör att man kan indexera (läsa och skriva element) på konstant tid. Det innebär också att arrayer i allmänhet inte har dynamisk storlek. När man skapar en array så anger man dess storlek, och den storleken ändras aldrig. Man kan alltså inte "lägga till" och "ta bort" element på samma sätt som med en lista.

Nu ska vi skriva ett program som använder arrayer:

- 1.) Öppna en ny fil i Emacs (`C-x C-f`) och döp den till `array.c`
- 2.) Skriv in följande programskelett:

```
1  int main(){
2    int a[5];
3    return 0;
4  }
```

- 3.) Raden `int a[5];` deklarerar en array som heter "a" som har plats för fem heltalselement. Klammarna efter namnet visar att a ska vara en array av intar och inte bara en int. Femman mellan klammarna anger hur många element arrayen ska ha plats för.
- 4.) Klammernotationen används också för att indexera i arrayer. Följande rader skriver in några värden i arrayen. Skriv in dem mellan deklarationen av a och return-satsen. Notera att arrayer indexeras från 0.

```
a[0] = 42;
a[1] = 10;
a[2] = 10 - (5 + 6);
a[3] = a[0] + a[1];
a[4] = 999;
```

- 5.) När man arbetar med arrayer använder man ofta loopar för iterera över samtliga element. Här är en loop som skriver ut alla element i a. Skriv in den innan return-satsen. Du behöver inte förstå hur `printf` funkar än.

```
for (int i = 0; i < 5; ++i){
    printf("a[%d] = %d\n", i, a[i]);
}
```


För att `printf` ska gå att använda så måste man skriva in `#include <stdio.h>` högst upp i filen. Raden berättar för kompilatorn att den ska börja med att läsa in filen `stdio.h`, som innehåller information om funktioner som används för in- och utmatning.

Loopen följer ett mycket vanligt mönster. Loopvariabeln `i` antar alla värden mellan 0 och arrayens högsta index (som i det här fallet är 4), och i varje iteration av loopen görs något med värdet `a[i]`.

- 6.) Kompilera och kör programmet. Verkar rätt värden skrivas ut?
- 7.) Du ska nu skriva en loop som följer samma mönster. Istället för att skriva om allting kan du kopiera loopen (`[M-w]` kopierar, `[C-y]` klistrar in. Markera text genom att hålla ned `[↑]` samtidigt som du flyttar markören, alternativt tryck `[C-space]` för att starta markering och flytta runt markören). Lägg den nya loopen innan den första och byt ut `printf`-anropet mot raden `a[i] = i`. Vilka värden ligger nu i arrayen? Kompilera och prova!
- 8.) Hur kan man ändra på loopen ni skrev i det förra steget så att arrayen istället innehåller kvadraten av motsvarande index (alltså $\{0, 1, 4, 9, 16\}$)?

J.2.1. Mer arrayer

- 1.) Öppna filen `rooms.c`.
- 2.) Det här är ett mycket enkelt program som skriver ut en lista av universitetsanställda och i vilket rum på Polacksbacken de sitter. Namnen lagras i arrayen `names` och rumsnumren i arrayen `rooms`. Här är arrayerna initierade med värden direkt istället för att de skrivits in ett och ett. Notera också att ingen längd har angivits mellan klammarna. Eftersom vi berättar vilka element som ska finnas i arrayen från början så kan kompilatorn räkna ut det själv.

Loopen är i princip likadan som den i `array.c`, förutom att vi har flyttat ut längden till en variabel `length`. Vad borde programmet skriva ut? Kompilera och kör!

- 3.) Prova nu att lägga till Stephan Brandauer i "databasen" (han sitter också i rum 1357). Vilka variabler behöver man ändra på för att loopen ska skriva ut hela databasen?
- 4.) Vad händer om man sätter variabeln `length` till något större tal än antalet element i arrayerna? Kompilera och kör!

Förmodligen hände en av två saker:

- Programmet krashade och skrev ut meddelandet `Segmentation fault`. Detta är felet som uppstår när ett program försöker läsa eller skriva minne som den inte har tillgång till. I det här fallet försökte programmet läsa ett värde utanför arrayens gränser. Prova att köra programmet i `gdb` (utan breakpoints) för att se vilken rad programmet krashar på!
 - Programmet körde men skrev ut obegripliga namn och rumsnummer efter den vanliga utskriften. I det här fallet hade programmet tillgång till minnet utanför arrayerna, men eftersom programmet inte skrev något till de minnesadresserna så kan det ligga vilket skräp som helst där.
- 5.) Vad händer om man anger en felaktig längd mellan klammarna på någon av arrayerna? Vad säger kompilatorn? Prova att ange både ett för högt och ett för lågt tal.

J.2.2. Tillämpning med memoisering

I denna uppgift skall vi använda arrayerna för att förbättra de tidigare fibonacciprogrammen. Det krävs relativt få ändringar, men de kan vara lite kluriga att tänka ut.

Inom matematiken (och den rena funktionella programmeringen) är en funktion f "stabil" i det avseendet att om $f(x) = y$ vid någon tid t så gäller att $f(x)$ alltid ger y . I de flesta programspråk behöver detta dock inte vara fallet – om f betyder "ge mig första elementet" och x är en lista *vars innehåll kan förändras* är det enkelt att se att två förekomster av $f(x)$ inte behöver ge samma resultat om x kan ha ändrats. Inom matematiken och i ren funktionell programmering kan x inte förändras (eller om x förändras så anser vi det vara en annan lista som inte längre är x , analogt med Herakleitos klassiska uttalande om floder).

Om vi för ett ögonblick glömmer *sidoeffekter* är det enkelt att inse att en funktion som är stabil i bemärkelsen ovan (ofta kallar man detta för en "ren funktion") aldrig behöver beräknas mer än en gång. En av anledningarna till att den rekursiva Fibonacci-implementationen ovan är så mycket långsammare än den iterativa är att den kommer att beräkna samma fibonaccital flera gånger (varför!?).

I denna uppgift skall vi återbesöka den rekursiva Fibonacci-implementationen. Först skall vi ta reda på hur mycket långsammare den är än den iterativa implementationen, sedan hur många gånger varje fibonaccital räknas ut vid en given fråga, och sist använda oss av en enkel optimeringsteknik för att förbättra dess prestanda avsevärt.

Prestandamätning med time Gå till kursens repo och kompilera en rekursiv och en iterativ implementation av vårt program. Följ instruktionerna nedan och justera för eventuella ändringar i katalognamn som du gjort efter eget huvud.

```
foo> cd ~/ioopm14/uppgifter/fas0/sprint0/kod
foo> make fib
```

Nu skall programmen vara kompilerade och heta `fib-rec` respektive `fib-iter`¹². Det lilla programmet `time` låter oss på ett enkelt sätt jämföra körtiden hos program. Du kan läsa mer om hur du skall tolka utdatat med hjälp av `man time`. Nedan ser du hur du kan mäta körtiden för det rekursiva programmet för `fib(5)`.

```
foo> time ./fib-rec 5
... resultatet skrivs ut
```

Generellt räcker det inte med att bara göra en mätning för att avgöra om eller hur mycket ett program är långsammare än ett annat. I de flesta fall är körtiden en funktion av indatats storlek, och man kan vaska fram ett förhållande genom att göra flera mätningar och t.ex. rita en graf. Gör därför 10 mätningar för `fib(n)` där n går från 1 till 50 och visualisera resultatet i ett diagram där y-axeln är körtid i millisekunder och x-axeln avser storleken på n . Vad visar den? Går det att få fram svar för samtliga uträkningar?

Hur många gånger räknas varje tal ut? Vi skall nu modifiera `fib-rec` för att hålla reda på hur många "onödiga" uträkningar som görs för en viss fråga. Ett enkelt sätt är att skapa en "tabell" där den i :te kolumnen håller reda på hur många gånger `fib(i)` har räknats ut. Detta kan enkelt åstadkommas med hjälp av en array av heltal. En sådan kan deklarerars på detta vis (ovanför `fib`-funktionen):

```
int calculations[128];
```

Deklarationen `int calculations[128]` skapar en global variabel¹³ `calculations` som håller en array av 128 heltal vars värden är noll. Notera att samma deklaration inuti en funktion hade gett upphov till en array med mer eller mindre slumpartade värden¹⁴. Då hade vi varit tvungna att manuellt sätta alla värden till noll, till exempel med en loop.

Nu vill vi för varje anrop till `fib(i)`, på den i :te positionen i `calculations`, öka det värde som finns där (initialt 0) med 1. Detta skulle kunna se ut så här (där i bör ersättas med något lämpligare):

```
calculations[i] += 1;
```

Stoppa in denna rad (eller gör efter eget huvud) på ett lämpligt ställe i programmet.

Slutligen vill vi, sist i `main`-funktionen, men före `return`-satsen, skriva ut resultatet. Det kan göras så här:

```
1  for (int i=0; i<128; ++i) {
2      printf("fib(%d) was calculated %d times\n", i, calculations[i]);
3  }
```

När programmet körs för `fib(6)` bör denna utdata inkludera följande:

```
fib(0) was calculated 5 times
fib(1) was calculated 8 times
fib(2) was calculated 5 times
fib(3) was calculated 3 times
fib(4) was calculated 2 times
fib(5) was calculated 1 times
```

¹²Programmet `make` används för att köra så kallade byggsript. I filen `Makefile` står byggsriptet som kördes med kommandot ovan. Du kommer få lära dig använda `make` senare i kursen.

¹³Observera att det anses som dålig programmering att använda globala variabler – framledes skall vi se hur vi kan undvika detta.

¹⁴Dessa värden uppstår av att det kan ligga bitar i minnet på den plats där arrayen skapas som då tolkas som siffror. Mer om detta senare.

```

fib(6) was calculated 1 times
fib(7) was calculated 0 times
...
fib(127) was calculated 0 times

```

Fundera över vad som behöver ändras i föregående kodlistning för att bara `fib(0)` till `fib(6)` skall skrivas ut så att man slipper (i detta fall fler än 100) onödiga utskrifter för tal som man inte försökt räkna ut. Lösningen skall vara generell och fungera för `fib(i)` där i är mindre än längden på calculations. (Fundera över varför det är så!)

Memoisering Nu vill vi optimera programmet så att första gången som `fib(i)` räknas ut sparas resultatet – alla andra gånger använder man det sparade resultatet av uträkningen. Denna teknik kallas memoisering och används i rena funktionella programspråk (d.v.s. inte i C).

Spara ditt modifierade program från föregående uppgift under ett nytt filnamn med `C-x C-w` och ersätt uträkningsräknaren för `fib(i)` på plats i med värdet av `fib(i)`. Byt även namn på calculations till `memoised_fib_values` som är ett bättre namn på variabeln, och ta bort sammanställningen av antalet anrop på slutet. Försök att göra detta effektivt i Emacs, t.ex. med `M-x` `query-replace` `C-j` som söker och ersätter, och `C-k` som tar bort en hel rad från markören.

Det skulle kunna se ut så här (modifiering av rad 8 i övning 2):

```

1  int fib_n = fib(n-1) + fib(n-2);
2  memoised_fib_values[n] = fib_n;
3  return fib_n;

```

Notera att C tillåter att vi skriver ovanstående utan den temporära variabeln:

```

return memoised_fib_values[n] = fib(n-1) + fib(n-2);

```

Detta anses dock svårsläsligt och därmed felbenäget. När varje uträknat fibonaccial sparats i arrayen kvarstår att också använda dessa memoiserade (sparade) resultat. För basfallen 0 och 1 behöver vi inte bry oss om memoiserade resultat eftersom svaren då är givna, men för $n > 1$ bör vi först titta i `memoised_fib_values[n]` för att se om svaret redan är uträknat. Så är fallet om det värde som finns sparad där är > 0 (varför?!), och då kan detta värde direkt returneras. Annars skall värdet räknas ut och sparas.

Skriv klart detta program själv och jämför med "facit", `fib-memo.c` som finns i samma katalog som `fib-iter` etc. När du är klar, använd `time` och undersök prestandaförbättringen jämfört med `fib-rec`.

Tänkvärt

1. C bygger på några få enkla koncept – dessa går fort att lära sig, men att lära sig hur de kan kombineras till felfria program är inte en parkpromenad! Läs andras kod och skriv egen kod så mycket du kan!
2. Om du är van vid ML, Ruby, Python, Java etc. så tycker du kanske att C bjuder på få funktioner – vad C ger i gengäld är hög prestanda och låg overhead och enorm kontroll över exakt vad som pågår i ett program.

J.3. Strängar och tecken

J.3.1. Tecken

I C är en variabel som deklarerats med typen `char` inget annat än ett (något förklätt) heltal (jämför med `Char` i till exempel ML, som är en helt egen datatyp). Teckenkonstanter som `'a'` och `'?'` är *precis samma sak* som motsvarande ASCII-värde (97 respektive 63). Att skriva `'a' + 3` är alltså helt okej i C, och det kommer ge precis samma resultat som om man skrev `97 + 3`.

J.3.2. printf

Hittills har vi använt `printf` utan att reflektera så mycket över hur den funktionen fungerar. Det ska vi råda bot på nu!

`printf` är en funktion som används för att skriva ut formaterad text (f:et i slutet av namnet står för "format"). Funktionen tar en så kallad formatsträng och ett godtyckligt antal andra värden som argument. Formatsträngen är en vanlig sträng som kan innehålla ett antal tecken med speciell betydelse.

Så kallade "escape-characters" inleds med backslash och används för att skriva ut speciella tecken, till exempel '\n' för radbrytning eller '\t' för tab. Anropet

```
printf("Hej!\nHopp\tHejsan!\n");
```

skriver ut "Hej!" och "Hopp Hejsan" på varsin rad, med en tab mellan "Hopp" och "Hejsan".

Formatsträngen kan också innehålla "formatspecifikationer", som inleds med ett procenttecken (%). Formatspecifikationer anger hur de andra argumenten till printf ska skrivas ut. Den första formatspecifikationen i strängen anger formatet för det första argumentet efter formatsträngen, den andra för den andra, och så vidare. Exempel på formatspecifikationer är

- %d – Ett heltal
- %f – Ett flyttal
- %c – Motsvarande ASCII-tecken
- %s – En sträng

Om vi antar att name är en sträng och age ett heltal så skriver följande anrop ut båda dessa värden:

```
printf("Jag heter %s och är %d år gammal", name, age);
```

Som en kort övning på att använda printf ska vi nu skriva ett program som skriver ut hela ASCII-tabellen:

- 1.) Öppna en ny fil och döp den till `ascii.c`.
- 2.) Skriv det gamla vanliga main-skelettet (eller kopiera det från en gammal fil) och inkludera `stdio.h`. Skriv en for-loop där loop-variabeln går från 0 till och med 127. Inuti loop-kroppen ska du skriva ett anrop till printf som skriver ut loop-variabelns värde både som heltal och som ASCII-tecken. När du är klar borde resultatet se ut ungefär så här:

```
1  #include <stdio.h>
2
3  int main(){
4      for (int i = 0; i < 128; ++i){
5          printf("%d\t%c\n", i, i);
6      }
7      return 0;
8  }
```

- 3.) Notera att i förekommer som argument till funktionen två gånger. Den första gången skrivs det ut som ett heltal (genom %d), och den andra gången som ett tecken (genom %c). Prova att kompilera och köra och se om utskriften verkar vettig (notera att alla ASCII-tecken inte går att skriva ut, till exempel är 10 ASCII-värdet för radbrytning).
- 4.) Som avslutning kan vi göra tabellen lite snyggare genom att sätta dit rubriker. Skriv dit följande rader högst upp i main-funktionen:

```
printf("ASCII\tChar\n");
printf("=====\t====\n");
```

Att det sitter en tab (\t) mellan kolumnerna ser till att det blir ett jämnt avstånd mellan dem.

printf erbjuder många fler möjligheter sätt att ange format (hexadecimal form, tiopotens, antal värdesiffror, vänsterjustering...). Du kan läsa man-filen för printf för att lära dig mer!

J.3.3. Strängar

På samma sätt som med tecken så finns det ingen speciell datatyp för strängar (som String i till exempel ML). En sträng representeras helt enkelt som en array av chars, som avslutas med ett NULL-tecken ('\0', ASCII-värdet 0). Strängen "IOOPM" är alltså en array bestående av sex tecken, ⟨ 'I', 'O', 'O', 'P', 'M', '\0' ⟩. Denna array är i sin tur är *precis samma sak* som en array bestående av heltalen ⟨ 73, 79, 79, 80, 77, 0 ⟩.

Nu ska vi titta på och ändra i ett program som använder strängar:

- 1.) Öppna filen `cheerleader.c`.
- 2.) Försök lista ut vad programmet gör. Kompilera sedan och kör.
- 3.) På första raden i main-funktionen deklareras en array av chars som initieras med strängen "Uppsala". Vi hade precis lika gärna kunnat skriva ut det som en array:

```
char name[] = {'U', 'p', 'p', 's', 'a', 'l', 'a', '\0'};
```

Eftersom tecken i C bara är kortformer för deras ASCII-koder hade vi också kunnat skriva

```
char name[] = {85, 112, 112, 115, 97, 108, 97, 0};
```

men det torde vara uppenbart att det är mer praktiskt att skriva som det redan står i filen (notera att det avslutande NULL-tecknet är implicit när man skriver en sträng inom citattecken).

- 4.) Loopen i funktionen cheer låter loopvariabeln `i` gå från 0 till (men inte med) det första värde där `name[i]` är `'\0'`. Loopen itererar alltså över hela strängen och terminerar när den når det avslutande NULL-tecknet i `name`.

Vad händer i loopkroppen då? Jo, först läses det `i`:te tecknet ut i variabeln `letter`, sen skrivs det ut två gånger på varsin rad (först som "Ge mig ett ...", sen som ett svar).

Efter loopen används funktionen `puts` för att skriva ut strängen "Vad blir det?", och till sist strängen `name`. Notera att `puts` automatiskt lägger till en radbrytning efter utskriften (i `printf` var vi tvungna att själva lägga till `'\n'`), och att funktionen kan skriva ut en strängvariabel direkt.

- 5.) Nu är det dags att utöka programmet! Till att börja med skulle vi vilja att det var lite mer umpf i svaren. Om man inkluderar `ctype.h` (på samma sätt som `stdio.h` är inkluderad) får man tillgång till funktioner som kontrollerar och manipulerar chars (se man-filen för `ctype`). Till exempel finns funktionen `toupper` som gör om en bokstav till motsvarande versal ("stora bokstav"). Inkludera `ctype.h` och ändra den sista raden i loopen till:

```
printf("%c!\n", toupper(letter));
```

Kompilera och kör programmet igen för att kontrollera resultatet. Prova gärna att byta strängen i `name`. Prova också att använda `tolower` för att göra om `letter` till en gemen ("liten bokstav") i den första utskriften.

- 6.) Vi skulle också vilja att den avslutande utskriften var lite mer övertygande. Eftersom störst går först så löser vi det på samma sätt som sist, genom att skriva ut hela strängen som versaler. Det gör vi genom att iterera över strängen och ändra varje tecken till motsvarande versal. Skriv en sån loop innan den sista utskriften i `cheer`! En lösning står nedan:

Samma loop-huvud som den första loopen. Inuti loopen räcker name[i] till för att skriva ut versalerna. (I den första loopen var det name[i] som användes för att skriva ut versalerna.)

J.3.4. string.h

Slutligen ska vi titta på lite av vad man kan göra med de funktioner som finns i Cs strängbibliotek. För att komma åt dessa funktioner behöver du inkludera `string.h`. Vi håller oss kvar i `cheerleader.c` ett litet tag till, och lägger märke till att funktionen `cheer` permanent ändrar på sitt argument (prova att skriva `puts(name)` efter funktionsanropet). Bara för att man ville få sitt namn ropat av en hejklack är det inte säkert att man ville få sitt namn permanent ändrat till något annat. Det här ska vi fixa nu!

- 1.) Vi ska se till att det första cheer gör är att kopiera sitt argument till en ny sträng. Först låter vi cheer deklarerar en ny sträng, men för att göra det måste vi veta hur lång den ska vara. Vi vill ha en array av chars som är lika lång som strängen `name` plus ett för NULL-tecknet. I `string.h` finns en funktion som heter `strlen` som returnerar längden av en sträng (exklusive NULL-tecknet). Vi inleder cheer med följande två rader:

```
int len = strlen(name);
char newName[len + 1];
```

- 2.) Härnäst vill vi kopiera innehållet i `name` till `newName`. Det gör vi med funktionen `strncpy(s1, s2, n)`, som kopierar strängen `s2` till `s1`, dock som mest `n` tecken. Funktionen förutsätter att det finns tillräckligt med ledigt minne för att utföra kopieringen, men det har vi ju sett till i deklarationen av `newName`¹⁵.

```
strncpy(newName, name, len);
```

¹⁵Funktionen `strcpy` fungerar på samma sätt, fast sätter inget maxtak på antalet tecken. Detta gör det svårare att garantera att man inte skriver utanför `s1`s gränser.

- 3.) Nu är det bara att ändra alla förekomster av `name` i `cheer` till `newName` (kom ihåg `query-replace`). När du är klar kan du jämföra med `cheerleader_finished.c`. Prova att skriva ut `name` efter anropet till `cheer` och se att strängen inte har ändrats.¹⁶

Det finns många fler funktioner i strängbiblioteket, till exempel för att jämföra strängar, slå ihop strängar eller leta efter förekomster av ett visst tecken. Dubbelkolla alltid vad som finns i `string.h` (man `string`) innan du skriver en egen strängfunktion!

Att ta med sig

1. Ett tecken är precis samma sak som ett heltal. Det är hur man tolkar värdet som spelar roll
2. Strängar är bara arrayer av chars. Glöm inte att se till att det finns plats för NULL-tecknet också! En sträng som är n tecken lång tar upp $n + 1$ chars.

J.4. Testdriven utveckling och parprogrammering

- 1.) Testdriven utveckling (TDD) är ett sätt att programmera som utgår från skapandet av tester. En grov förenkling är denna: När man skall lägga till en funktion i ett program börjar man med att skapa ett eller flera tester för funktionen och fångar därigenom funktionens specifikation. Målet blir sedan att implementera funktionen så att testerna passerar utan fel, varefter man kan fortsätta till nästa funktion.

Vi uppmuntrar testdriven utveckling på denna kurs av flera anledningar. Bland annat för att fånga en specifikation i ett test är ett bra sätt att driva fram alla specialfall¹⁷. Att använda testdriven utveckling är ett *krav* under projektet i Fas 2.

Från och med nu, använd TDD för resten av denna sprint.

- 2.) Parprogrammering är ett enkelt sätt att bygga in granskning av kod och därmed förhöjd kodkvalitet i ett projekt som en del av utvecklingsprocessen. På denna kurs kommer vi att tillämpa parprogrammering hela tiden och specifikt en modell där en person är *driver* och den andre *co-pilot*. Dessa två roller växlar fram och tillbaka mellan medlemmarna i ett par flera gånger varje timme.

Driver är den som sitter vid tangentbordet. Co-pilot sitter brevid och läser allt som Driver gör, och ser till att fånga alla småfel som Driver naturligt gör i sin iver att komma vidare i implementationen. Det kan vara namngivning, att bryta ut upprepningar till en separat funktion, snygga till en bit oläslig kod, etc. Driver får inte skriva kod som Co-pilot inte förstår. Det är inte meningen att Co-pilot skall sitta överksam och tyst, utan all kod skrivs med fördel under dialog.

Från och med nu, använd parprogrammering under resten av kursen. Diskutera rollerna så att de är klara för er.

- 3.) I kombination med TDD tillämpar vi nu en modell där vi växlar roller (Driver blir Co-pilot och vice versa) i samband med att vi byter mellan att skriva test och faktisk implementation. Det är inte meningen att en person skall skriva alla tester och en annan person all implementation, men samma person som skrev testet bör inte skriva koden som skall passera testet (varför?!).

När ni bryter ned en uppgift i små beståndsdelar, försök hitta delar som är små nog att ta max 10 minuter att göra så att det blir många byten mellan att skriva test och implementation varje timme. Detta är givetvis svårt, speciellt i början – och det kan vara en mycket god idé att ställa en klocka på 10 minuter och byta roller när den ringer så att det blir en bra balans mellan att vara Driver och Co-pilot för alla inblandade.

Att ta med sig

1. Parprogrammering tenderar att skapa kod av högre kvalitet fortare än "ensamkodning"
2. Använd parprogrammering med klara roller (t.ex. skriva testkod/skriva programkod) och byt ofta

¹⁶Fundera gärna på om man kan skriva funktionen `cheer` utan att använda `strncpy`.

¹⁷Läs gärna Minskys klassiker "Why Programming is a Good Medium for Expressing Poorly Understood and Sloppily-Formulated Ideas."

J.5. I/O

I/O står för "Input/Output", alltså in- och utmatning. I C och Unix (och de flesta andra programmeringsspråk) sker all I/O med hjälp av så kallade strömmar (*streams*). En ström kan ses som en kanal som man kan skriva till eller läsa ifrån. Andra änden av en ström kan vara en fil på hårddisken, ett annat program eller en nätverks-socket.

J.5.1. Standardströmmar

Det finns tre standardströmmar som är viktiga att känna till. Dessa heter `stdin`, `stdout` och `stderr`. `stdin` är data som går in i ett program, vanligtvis via terminalen. `stdout` är data som kommer ut från ett program. Om man inte anger något annat så sker alla utskrifter via `printf` och `puts` till `stdout`. Den sista strömmen, `stderr`, används för att skriva ut felmeddelanden. Vanligtvis går denna data också till terminalen, men det kan finnas anledning att separera destinationen för felmeddelanden och vanlig output (som då skrivs till `stdout`).

Nu ska vi titta på ett enkelt program som interagerar med terminalen:

- 1.) Öppna filen `io.c`. Det enda som borde vara nytt här är funktionen `scanf`, som används för att läsa från `stdin`. Dess första argument är en formatsträng, precis som den som `printf` tar emot. Formatsträngen `%s` betyder att vi kommer försöka läsa en sekvens av tecken tills vi stöter på ett blanktecken. Prova att kompilera och köra programmet. Det kommer att kräva inmatning i terminalen.
- 2.) Nu ska vi lägga till information om användarens ålder. Deklarera en `int` som heter `age` i början av programmet. Lagg sedan till en fråga om personens ålder innan `printf`-anropet, följt av raden

```
scanf("%d", &age);
```

Och-tecknet innan `age` behövs för att `scanf` vill ha en minnesadress som argument.

Formatsträngen `%d` säger att vi kommer försöka läsa ett heltal. Mer om det i nästa del! Utöka nu formatsträngen i `printf`-satsen med ett `%d` och lägg till `age` som argument. Kompilera och provkör!

- 3.) Programmet funkar nu helt okej, men om man skriver in något som inte består av siffror som ålder så kommer programmet fortsätta ändå (och eventuellt ge konstiga utskrifter). Vi skulle vilja att programmet istället upprepade frågan om ålder om ett heltal inte gick att läsa.

Vi kommer ta hjälp av att `scanf` returnerar ett värde som motsvarar hur många läsningar som lyckades. Om anropet returnerar 1 lyckades funktionen läsa ett heltal, annars returnerar den 0. Vi skriver därför en loop som säger ungefär "Så länge som `scanf` inte lyckas läsa något heltal (alltså returnerar 0) så skriver vi ut en ny fråga":

```
while(scanf("%d", &age) == 0){
    puts("Please use digits");
}
```

- 4.) Om du kompilerar och kör det här programmet och skriver in en felaktig ålder så kommer du märka att programmet fastnar och bara skriver ut "Please use digits" om och om igen (använd `C-c C-c` för att avbryta körningen). Det är för att `scanf` inte tar bort det som den läser från `stdin`. `scanf` står alltså och försöker tolka samma felaktiga inmatning i all oändlighet!¹⁸

Om inläsningen misslyckas måste vi se till att läsa förbi alla felaktiga tecken tills vi stöter på en radbrytning (varför just en radbrytning?). Vi utökar därför loopen till följande:

```
while(scanf("%d", &age) == 0){
    puts("Please use digits");
    while(getchar() != '\n');
}
```

Funktionen `getchar` läser (och konsumerar) ett tecken från `stdin`. Den inre while-loopen betyder alltså ungefär "Fortsätt att konsumera ett tecken i taget från `stdin` så länge som vi inte läser en radbrytning". Nu ska programmet fungera som det ska! Du kan jämföra med `io_finished.c` om det inte gör det.

¹⁸"Definitionen av vansinne är att upprepa samma sak och varje gång förvänta sig ett annat resultat"

- 5.) När man anropar ett program från terminalen kan man också omdirigera standardströmmarna till filer. Anropet `./io > out.txt` skickar allt som skrivs till `stdout` till filen `out.txt`. Notera att filen skrivs över om den redan finns! Anropet `./io < in.txt` kör programmet och läser `stdin` från filen `in.txt`. Man kan också kombinera dessa kommandon. Prova att skapa en fil som heter `in.txt` med innehållet

```
Silvia
xyz
69
```

och kör med kommandot `./io < in.txt > out.txt`. Undersök innehållet i `out.txt` med `cat`.

J.5.2. I/O till filer

Även om man kan göra en del med bara omdirigering så är det praktiskt att kunna öppna filer för läsning och skrivning direkt i själva programmet. Som exempel ska vi skriva ett program som läser in en fil och skriver ut den till en annan fil, fast med radnummer.

- 1.) Ett skelett till programmet finns i `linum.c`. Programmet läser det första av sina argument vid terminalanropet som ett filnamn (variabeln `infile`), och skapar en sträng som är filnamnet fast med `lin_` före (variabeln `outfile`). De nästföljande två raderna öppnar strömmen `in` som en *läsström* från `infile`, och strömmen `out` som en *skrivström* till `outfile`. Funktionen `fopen` används i båda fallen, men vilket läge som avses (läsning eller skrivning) anges med det andra argumentet. En ström har typen `FILE*`.

I slutet av programmet stängs båda strömmarna, vilket är viktigt för att undvika korrupta filer och för att se till att programmet frigör alla resurser det har allokerat.

- 2.) Om du kompilerar och kör programmet (med en fil som argument) så kommer bara en ny tom fil med prefixet `lin_` skapas. Vi ska nu skriva kod som läser från strömmen `in` och skriver till strömmen `out`, och vi skriver koden på det markerade stället.

Vi kommer behöva två hjälpvariabler. En `int` som håller koll på vilket radnummer vi är på, och en sträng som kan lagra en inläst rad i taget:

```
int line = 1;
char buffer[128];
```

Stränglängden 128 är godtycklig. Det viktiga är att en hel rad från filen vi läser får plats i `buffer`.

- 3.) Vi kommer vilja utföra samma sak för varje rad i filen som `in` pekar på, så vi skriver en `while`-loop med villkoret "Så länge som `in` inte har nått slutet på filen":

```
while(!(feof(in))){ ... }
```

Funktionen `feof` tittar på en ström och returnerar ett heltal som inte är noll om och endast om strömmen har nått slutet på en fil.

- 4.) Inuti loopen vill vi först läsa en rad från `in`. Det gör vi med anropet `fgets(buffer, 128, in)` som läser en sekvens av tecken från strömmen `in` tills den stöter på en radbrytning, eller som mest 127 tecken, och lagrar dem i strängen `buffer`.

För att sedan skriva ut strängen med radnummer använder vi `fprintf` som fungerar precis som vanliga `printf` förutom att den också tar strömmen den ska skriva till som argument (innan formatsträngen). Slutligen ökar vi på radnumret med ett inför nästa loop. När den är klar borde loopens se ut så här:

```
while(!(feof(in))){
    fgets(buffer, 128, in);
    fprintf(out, "%d. %s", line, buffer);
    line++;
}
```

- 5.) Spara en ny fil med tre fyra rader text som `test.txt` och provkör programmet med kommandot `./linum test.txt`. Titta sen på resultatet i `lin_test.txt` med `cat`. Prova också programmet på någon kodfil som du har skrivit!

- 6.) Slutligen kan det vara intressant att se vad som händer om man allokerar en för liten sträng att lagra raderna i. Prova att minska storleken på buffer till 5 och kompilera om programmet igen. Kör det på en fil som har rader längre än fyra tecken. Vad blir resultatet?

Det är värt att poängtera att vi hade kunnat skriva programmet `linum` med hjälp av omdirigering av `stdin` och `stdout`. En frivillig utvidgning av uppgiften är att låta programmet använda standardströmmarna när man inte ger det något argument, så att man kan anropa det både som `./linum test.txt` och `./linum < test.txt > lin_test.txt`.

Att ta med sig

1. In- och utmatning sker med hjälp av strömmar som har typen `FILE*`.
2. Standardströmmarna `stdin`, `stdout` och `stderr` är kopplade till terminalen (om inget annat anges). Funktioner som inte tar någon ström som argument läser och skriver i allmänhet via `stdin` och `stdout`.

J.6. Malloc

Förberedande screencasts om C inkluderar sammansatta datatyper, typer, typomvanling och `typedef`, samt grundläggande minneshantering.

J.6.1. Stacken är finit / Stacken är bara så djup

Ta reda på hur "djup" stacken är. Det går enkelt att göra genom att skriva ett program med en funktion som rekursivt anropar sig själv och skriver ut antalet rekursiva anrop tills det kraschar. (På grund av s.k. stack overflow – programmet får inte plats på stacken.)

```

1  #include<stdio.h>
2
3  void checkStackdepth(int depth) {
4      printf("Djup: %d\n", depth);
5      checkStackdepth(depth+1);
6  }
7
8  int main(void) {
9      checkStackdepth(0);
10     return 0;
11 }
```

Pröva att modifiera programmet ovan så att varje rekursivt anrop till `checkStackdepth` använder mer stackyta. Läs t.ex. på http://en.wikipedia.org/wiki/Stack_overflow för ledning om hur man kan göra det. Vad betyder detta i praktiken för det redan uppmätta stackdjupet?

J.6.2. Strukturer och värdesemantik

En strukt i C är en sammansatt datatyp som används för att definiera objekt med flera olika beståndsdelar. En strukt kallas ibland på svenska för "post" och består av noll eller flera datafält med sinsemellan unika etiketter. Se t.ex. [http://en.wikipedia.org/wiki/Struct_\(C_programming_language\)](http://en.wikipedia.org/wiki/Struct_(C_programming_language)).

En strukt för att skapa objekt som beskriver punkter i planet skulle kunna definieras så här:

```

1  struct point {
2      int x;
3      int y;
4  }
```

Detta deklarerar datatypen `struct point` som kan användas på efterföljande rader. Variabler av denna typ har två fält – `x` och `y` som kan läsas och skrivas individuellt. Man skulle kunna definiera en variabel `somewhere` så här:

```

1  struct point somewhere;
2  somewhere.x = 7;
3  somewhere.y = 3;
```

C stöder en smidigare typ av initiering av sammansatta datatyper med följande syntax:

```
struct point somewhere = { 7 , 3 };
```

där de initiala värdena tilldelas fälten i deklarationsordning uppifrån och ned. Detta är bräckligt eftersom en förändring av deklarationsordningen i `point` kommer att ändra kodens betydelse utan varken varningar eller fel. Ett bättre sätt är att istället ange namnen på de fält man vill initiera:

```
struct point somewhere = { .x = 7 , .y = 3 };
```

Kompilera och kör programmet `point-struct.c`. Programmets avsikt är att flytta en punkt (7,3) relativt en punkt (4,5) så att den första punkten får koordinaten (11,8) – men något är fel! Vad?

Fel! Det består i att den punkt `p` som ändras är funktionens `translate`:s egen *kopia* av den punkt som skickades in, eftersom strukturer överförs med s.k. värdesemantik.

För att lösa problemet kan vi göra två saker: vi kan antingen skicka tillbaka (en kopia av) den ändrade kopian eller använda oss av *referenssemantik* som tillåter två platser i koden att dela på samma värde. Kod för den första lösningen finns i `point-struct-return.c` – jämför programmen för att se den minimala ändringen. Sedan går vi vidare till att prata om referenssemantik.

J.6.3. Pekare och referenssemantik

En pekare är enkelt uttryckt en adress till en plats i minnet där det ligger ett annat värde¹⁹. En variabel av typen `int` innehåller ett heltalsvärde; en variabel av typen `int*` innehåller en adress till en plats i minnet där det finns ett heltalsvärde. Analogt avser typen `int**` en adress till en plats i minnet där det finns en annan adress till en plats i minnet där det finns ett heltal, etc. På svenska säger vi "en pekare till ett heltal" och på engelska "pointer".

Programmet `point-struct-pointer.c` använder sig av pekare till punkter för att dela punkter mellan huvudprogrammet och `translate`-funktionen. Detta sker genom att typen på parametern till funktionen ändrats till `struct point*`, alltså inte längre en punkt utan en pekare till en punkt. Vad som nu skickas in som argument till funktionen är adressen till en plats i minnet där en punkt finns, som kan läsas och skrivas – en plats (och en punkt) som därmed delas mellan huvudprogrammet och funktionen. Notera att syntaxen för att komma åt fält via pekare ändras från `.` till `->` (rad 7 och 8) för att tydligt visa att man gör åtkomst till en icke-lokal plats i minnet som kan var synlig för "vem som helst".

Operatören `&` kallas för "adresstagningsoperatör" och den kan användas för att ta fram adressen till ett värde. På rad 21 i `point-struct-pointer.c` används den för att komma åt de platser i minnet där `somewhere` och `other` finns eftersom `translate` nu förväntar sig denna input istället för det faktiska värdet.

Kör programmet och kontrollera att uppdateringarna av `p` i `translate` nu får förväntat genomslag.

Gå från värdesemantik till referenssemantik Ändra funktionen `printPoint` så att den också tar emot en pekare till en punkt istället för en kopia. Du måste modifiera parameterdeklarationen, åtkomsten till `x` och `y` i utskriften, samt anropsplatserna som nu måste skicka in adresser till punkter istället för de faktiska punkterna. Verifiera med hjälp av verktygen och förstås genom att köra programmet!

J.6.4. Allokering på heapen

Hittills har alla värden vi hanterat legat på stacken. Stacken medger effektiv hantering av små och kortlivade värden, men för stora och dynamiskt växande strukturer vill vi i regel använda heapminnet.

Allokering på heapen sker med hjälp av funktionen `malloc` (och några till med liknande funktion). Som argument till `malloc` anger man hur många bytes i minnet man vill allokera, och i retur får man adressen till ett minnesblock av denna storlek.²⁰

En van C-programmerare kan enkelt räkna ut hur många bytes som behövs för att lagra t.ex. en `struct point` i minnet, en siffra som varierar beroende på vilken plattform (vilken hårdvara och vilket operativsystem) man använder. För att slippa göra denna huvudräkning varje gång man skall kompilera sitt program på en ny plattform tillhandahåller C ett hjälpmacro "`sizeof`".

För att "allokera" nog med utrymme för att rymma en `struct point` kan man skriva:

```
... = malloc(sizeof(struct point));
```

¹⁹Mer exakt kan det ligga ett annat värde där. Mer om detta senare under kursen.

²⁰Mer exakt av minst denna storlek, men det finns i regel inget sätt att ta reda på storleksskillnaden.

Returtypen för malloc är något komplicerad och vi återkommer till detta senare i kursen. För stunden kan vi tänka att malloc returnerar en pekare till den struktur vars yta vi bad om, i exemplet ovan alltså en pekare till en **struct** point, alltså en **struct** point*.

Från allokering på stacken till allokering på heapen Ditt modifierade program från ovan har två variabler somewhere och other som innehåller punkter som är allokerade på stacken. Modifiera variablerna så att de blir *pekare* till punkter som är allokerade på heapen. Notera att användande av malloc kräver att man först inkluderar stdlib.h.

En viktig skillnad mellan allokering på stacken och allokering på heapen är att initieringen av strukturen måste göras explicit (vi kan alltså inte använda oss av = { 7, 3 } etc. som vi såg tidigare):

```
p->x = 7;
p->y = 3;
```

Du kan jämföra din lösning med point-struct-malloc.c.

J.6.5. Utökning: typedef

Använd **typedef** för att lägga till en ny typdefinition i ditt program (se t.ex. här för mer information: <http://en.wikipedia.org/wiki/Typedef>). På kursen använder vi med något undantag konventionen att typer med stor begynnelsebokstav avser pekare. Definiera alltså typen Point som en pekare till en **struct** point*. Du kan jämföra din lösning med point-struct-typedef.c. Tycker du att programmet blir mer eller mindre läsbart?

Att ta med sig

1. Notera att argumentet till sizeof är **struct** point utan * – det är för att vi vill veta hur mycket plats den faktiska strukturen tar, och inte hur mycket utrymme som krävs för att lagra själva adressen dit!
2. Använd alltid sizeof istället för att försöka räkna ut storlekar – det senare leder bara till fel och plattformsspecifik kod!
3. Minnesläckage och pekarfel är de vanligaste felen C-programmerare gör, oavsett om de är nybörjare eller har lång erfarenhet!

K. Enhetstestning

Enkelt uttryckt är ett enhetstest för en funktion f en samling indata $i_1 \dots i_n$ och förväntat utdata $u_1 \dots u_n$. Enhetstester för f kan t.ex. skrivas i samma kodfil som f , döpas till test-av- f -1 etc. och köras genom en funktion test-av- f -alla, men det är bättre att lägga testerna i en separat fil. Med en så långt som möjligt automatiserad process vill man sedan avgöra att $f(i_j) = u_j$, för $j = 1 \dots n$. Idealiskt kör man alla relevanta test löpande, men åtminstone innan man checkar in kod för att se till att inte trasig kod sprids till andra och förpestar deras tillvaro. Dock är det svårt att skriva bra tester. Det kräver både att man förstår problemet (koden man vill testa) och att man vet vilka de vanliga felen är som man bör testa för. Och även om man har bra tester, är det inte säkert att tiden finns att köra alla om det tar flera timmar att köra dem.

Enhetstester är enkla men kraftfulla. I grund och botten ligger två frågor – gäller $f(i) = u$ (som förväntat) eller gäller $f(i) \neq u$ (som förväntat)? I många ramverk för enhetstester (vi möter dem senare i kursen) har dessa frågor namn som "assertTrue" och "assertFalse". Om vi vill pröva att funktionen strlen, som räknar ut längden på en given sträng, är korrekt skulle följande satser vara exempel på sanna utsagor²¹ som vi förväntar oss håller i en korrekt implementation av strlen:

Test	Uttryckt i kod
Längden av "Hej" är 3.	strlen("Hej") == 3
Längden av "" är 0.	strlen("") == 0
Längden av "Hokus pokus filiookus" är 20.	strlen("Hokus pokus filiookus") == 29

Saker som man bör testa i enhetstest är extremvärden, tomma strängar, stora värden. Prova att en loop är korrekt för noll varv, ett varv och många varv. Prova att termineringsvillkoret stämmer!

²¹Observera att översättningen till kod är felaktig (29 istf 20). Poängen är att även tester kan vara felaktiga.

```

1   for(int i=0; i<M; ++i) {
2       doSomething(i);
3   }

```

Hitta testfall som gör M till 0, 1 och något stort tal. M är i regel någon slags input eller någon funktion av input.

I exemplet ovan med test av `strlen` vill vi pröva med strängar av längden 0, 1 och en längre sträng.

Försök att få så hög "code coverage" som möjligt. Det betyder att testerna skall pröva så mycket som möjligt av koden, och helst alla möjliga vägar genom koden (tänk nästlade villkorssatser).

Att bara testa med slumpvisa värden är inte ett bra test utan varje test har en anledning som härrör från funktionen man testar och dess omständigheter. Försök att ha en anledning till varje test och att dokumentera dessa anledningar!

Du kan inkludera testkod i ditt program i form av kod som anropar de funktioner du vill testa. I avsnitten nedan skall du skriva tester för det program som du implementerar så att du kan vara (mer) säker på att det är korrekt. Detta kommer att tvinga dig att fundera kring vad som är vettigt att testa, och vilka värden etc. som är lämpliga.

Här är till sist implementationen av enhetstestet för `strlen` (titta på koden i `minute.h` och `minute.c`²² och kör programmet nedan för att se vad det gör. Flera tester kan enkelt läggas till som nya funktioner som också anropas från `main`.)

```

1   #include "minute.h"
2   #include<string.h>
3
4   void testStrlen() {
5       assertTrue(strlen("Hej") == 3);
6       assertTrue(strlen("") == 0);
7       assertTrue(strlen("Hokus pokus filiokus") == 20);
8   }
9
10  int main(void) {
11      testStrlen();
12      report();
13      return 0;
14  }

```

Kompilera med `make unit` och kör med `./unit-test-example`. Detta ger följande output:

```

foo> make unit
foo> ./unit-test-example
Testing strlen("Hej") == 3 ... PASSED
Testing strlen("") == 0 ... PASSED
Testing strlen("Hokus pokus filiokus") == 20 ... PASSED
All 3 test passed!

```

Att ta med sig

1. Enhetstesterna är dina vänner
2. Skriv alltid enhetstester till den kod som du skriver
3. Automatisera körandet av tester så att alla tester går att köra på ett enkelt sätt

L. Kodgranskning

Välj en uppgift ur nedanstående lista av fem små program att implementera. Implementera programmet och gå sedan vidare till själva uppgiften, som är beskriven i L.2.

L.1. Fem små program

L.1.1. Gissa talet (enkel)

Skriv ett gissa talet-spel för två spelare. När programmet startar frågar det efter ett tal. Detta matas in av spelare 1 medan spelare 2 tittar bort. När spelare 1 har matat in talet får spelare 2 gissa vilket

²²Trots sin enkelhet använder detta program flera koncept som är för avancerade för att hitta in i kursen redan nu.

tal spelare 1 matade in. Spelet svarar med "För litet!", "För stort!" eller "Mitt i prick!". När rätt tal matats in anges också hur många gissningar som krävdes för att hitta rätt innan spelet avslutas. Vid 10 felaktiga gissningar avslutas spelet och "Du förlorar!" skrivs ut.

En frivillig utökning av detta spel är att Spelare 1 ersätts av datorn som slumpar fram ett tal mellan givna intervall som kan vara konstanter i koden eller ges som kommandoradsargument.

L.1.2. Frekvenstabell för vokaler i en text (enkel)

Vokalerna i det engelska språket är "aoueiy". Skriv ett program som läser in en fil tecken för tecken och räknar förekomsterna av varje vokal. När filen är processerad skall en frekvenstabell skriva ut, t.ex.:

a	15
o	9
u	3
e	25
i	8
y	2

Ovanstående siffror stämmer för texten inklusive tabellen och kan användas som ett test.

En frivillig utökning av uppgiften är att sortera tabellen i fallande frekvensordning.

L.1.3. Begränsad storlek på omdirigering av utdata (medelsvår)

Som tidigare tagits upp i detta dokument kan man i UNIX omdirigera in- och utdata med hjälp av operatorerna `<` och `>`²³.

Ett problem vid tidigare års kurser har varit när studenter dirigerat utdata från ett program med en oändlig loop till en fil på sitt konto och sedan blivit utloggade på grund av att de överskridit sin kvota. En enkel lösning på detta är att skriva programmet `cap` som läser från standard in tecken för tecken och skriver allt den läser till standard out, upp till en viss gräns, som kan vara en konstant i programmet t.ex. en megabyte (1024×1024 tecken), eller hellre ett kommandoradsargument.

L.1.4. Sortering av värden i en array (medelsvår)

Skriv ett program som läser in positiva heltalsvärden från en radorienterad fil, d.v.s. på varje rad i filen finns ett positivt heltal följt av ett `\n`-tecken. Filens första rad anger hur många rader som följer. Samtliga tal skall läsas in i en array med lika många poster som antalet tal. När samtliga tal lästs in skall arrayen sorteras i stigande ordning.

L.1.5. N-fakultet (enkel)

Skriv ett program som läser in positiva heltalsvärden från användaren i terminalen och räknar ut fakulteten. För ett heltal större än noll är fakulteten lika med produkten av alla heltal från 1 upp till och med talet självt, d.v.s. fem-fakultet är $1 \times 2 \times 3 \times 4 \times 5 = 120$.

L.2. Uppgiftens andra del

Uppgiftens andra del handlar om kodgranskning. Ni skall nu byta kod med ett annat par. Ni ger dem en utskrift av er kod och får i utbyte en utskrift av deras kod.

Er uppgift är nu att läsa denna tillbytt kod och utföra en *kodgranskning*. Kodgranskning är en viktig del av programutveckling som hittar och rättar fel i programkod. Kodgranskning är också en aktivitet som bidrar till att göra granskarna till bättre programmerare, något som framhålls från många håll både inom akademi och industri. En kompletterande del av "magin" är att vetskapen att den kod man skriver kommer att granskas – detta leder till att koden man skriver blir bättre från början.

Normalt är att granska ca 150 rader kod per timme och att inte granska mer än 2 timmar per dag. Mer än 150 rader kod/timme är ofta för snabbt för att hitta fel, och för lång tid leder ofta till slarv p.g.a. "kodblindhet".

När ni är klara med er kodgranskning (ca 1 timme) skall ni byta kommentarer med varandra och gå igenom dem tillsammans.

²³Om du är obekant med detta koncept, läs t.ex. här: http://www.acc.umu.se/~handbok/05_unix/omdirigering.html

Nedan finns en checklista med frågor som ni skall särskilt beakta under denna kodgranskning? (En sådan lista kallas för ett kodgranskningsprotokoll. Det finns många varianter och vissa för specifika programspråk. Detta är ett förenklat protokoll för denna tidiga del av kursen. Det är lika viktigt att förstå frågorna som att förstå koden man skall besvara dem för.)

- ☐ Har hela programmet implementerats?
- ☐ Är koden välstrukturerad och konsekvent formaterad?
- ☐ Finns det död kod? (T.ex. funktioner som aldrig anropas eller kod som aldrig kan nås.)
- ☐ Finns det upprepningar som skulle kunna bryta ut och läggas i separata funktioner?
- ☐ Har alla variabler namngivits på ett vettigt, tydligt och konsekvent sätt?
- ☐ Finns det överflödiga (redundanta eller oanvända) variabler?
- ☐ Är alla loopar, villkorssatser och andra logiska konstruktioner kompletta och korrekta?
- ☐ Prövas det mest sannolika fallet först i alla villkorssatser?
- ☐ Är det tydligt att alla loopar alltid terminerar?
- ☐ Deklareras och initieras temporära variabler i anslutning till den kod där de används?

Reflektion I samband med att ni jämför kommentarer, diskutera också aktiviteten kodgranskning. Fundera över: ditt eget och andras ego, hur ger man kritik, varför detta används i verkligheten i industrin, vad läsbarhet är med avseende på kod, samt om ni tror att sättet ni kodar på påverkas av det faktum att någon annan skall kunna läsa koden?

M. Att spåra minnesläckage

Programmet Valgrind är ett program som kan hjälpa till att hitta minnesfel i ett program. För att komma åt det i datorsalarna måste man logga in på en av Linux-maskinerna²⁴. Kommandot `linuxlogin` öppnar en terminal på en maskin som har lediga resurser. Du kan också logga in manuellt med kommandot `rsh` följt av en av adresserna nedan.

```
tussilago.it.uu.se  
vitsippa.it.uu.se  
gullviva.it.uu.se
```

Du kommer ha tillgång till din hemkatalog och alla kataloger du kan nå vanligtvis. Du behöver alltså inte skicka några filer mellan maskinerna.

Läs på om Valgrind på Internet eller med hjälp av man-sidor i terminalen (på Linux-maskin). Hitta informationen med hjälp av Google. Försök att besvara följande frågor:

1. Vilken typ av fel kan Valgrind hitta?
2. Vilken typ av fel kan Valgrind inte hitta?
3. Kan Valgrind rapportera s.k. "false positives" alltså rapportera något som ett läckage som faktiskt inte är ett läckage?
4. Hur kör man Valgrind?

Kör nu Valgrind på ert program från uppgift J.6 och se om du hittar några fel. Tyd utskrifterna från programmet och laga alla eventuella fel. Fortsätt tills inga läckage rapporteras.

Att ta med sig

1. Valgrind (eller motsvarande program) är din vän!
2. Ca 40% av alla fel som görs i C-program har att göra med hantering av minnet
3. Använd Valgrind regelbundet utöver kompilatorn, lint och enhetstester för att hitta fel under utveckling!

N. En första redovisning

Om du inte är bekant med hur denna kurs fungerar med kunskapsmål, etc. – läs först om detta på kursens webbsida <http://wrigstad.com/ioopm>. Där finns också en förteckning över samtliga mål med tillhörande förklaringar.

²⁴IT-institutionen kommer på sikt att övergå helt till Scientific Linux. Läs mer på <http://www.it.uu.se/datordrift/maskinpark/linux>

Du får börja redovisa mål vid labbtillfällena från och med vecka två i kursen. I allmänhet ska *alla* redovisningar ske med kod och exempel från kursens utdelade uppgifter. Om du vill kan du dock välja att redovisa **ett** av de nedanstående kunskapsmålen med hjälp av uppgifter från detta dokument. En anledning till att vi inte vill att du skall använda dessa uppgifter för att bocka av andra kunskapsmål är för att de är alldeles för små och tillrättalagda för att vara tillräckligt lärorika. *De tar dig istället till en nivå där det blir vettigt att börja redovisa.*

A 1 F 13 M 36 M 38 R 50 T 53

Se denna första redovisning som en övning i att redovisa! På det första gruppmötet *skall* du också redovisa målet X 62.

Att tänka på inför sin första labbredovisning När du redovisar är ditt mål att övertyga labbassen om att du uppfyller de kunskapsmål etc. som du vill bocka av. Börja med att berätta vilka kunskaper du vill redovisa och fortsätt sedan med att berätta din plan för hur redovisningen skall gå till. Genomför sedan själva redovisningen (räkna med att bli avbruten och behöva svara på frågor) och avsluta med att berätta varför du anser att du nu har visat att du uppfyller de aktuella kunskapsmålen. Om du inte har tänkt igenom din redovisning eller inte är tillräckligt tydlig med vad du vill uppnå kommer labbassen att avbryta redovisningen och gå vidare till nästa person på schemat och ge dig tid att tänka igenom allt innan du provar igen.

Det är inte labbassens ansvar att se till att du blir godkänd – ansvaret är ditt. Om labbassen anser att du inte blir godkänd på något eller alla mål, eller inte kan följa ditt resonemang kan du begära att hen förklarar varför, men du kan inte räkna med att det blir en diskussion eller argumentera att ”du visst skall bli godkänd”.

Att tänka på inför sin första gruppredovisning Kvaliteten på din presentation avgör kvaliteten på undervisningstillfället för minst fem andra studenter. En av anledningarna till att grupperna finns är för att ge möjlighet till diskussion och reflektion i en mindre grupp. Se det som ett mål att inte bara övertyga grupppassen, utan att lära ut något till de andra. Om du inte är säker på något är det bättre att säga det än att dra till med något och spela tvärsäker så slipper de som inte redan visste lära sig något som är fel. Målet är inte att vara perfekt utan att vara tillräckligt bra, och vad tillräckligt bra är kommer du att få klart för dig över tid.

En bra taktik i en presentation är att försöka vara den första att peka ut alla fel innan examinatoren hinner göra det. Det kan vara en bra idé att ha bilder att visa (dator finns att låna om det behövs) för att varva med sitt i förväg uttänkta demo, eller hur du nu har tänkt redovisa. Samma ansvar som vid en labbredovisning gäller. Om du försöker komma på ett sätt att redovisa på plats kommer du att märka att det blir ostrukturerat och tar för lång tid. Du kommer då att bli avbruten och ingen blir glad av det!

Fusk och redovisningar Det är rimligt att tro att situationen uppstår då du vill redovisa *X* men någon annan har redan redovisat *X* vid ett tidigare möte. I detta sammanhang är det tillåtet att *upprepa endast i rimlig utsträckning*. Notera att om du anger källa (”den här bilden har jag tagit från Martins presentation förra veckan”) blir det inte plagiat²⁵ längre, utan ett citat²⁶. Kursens upplägg möjliggör tillräckligt stor variation för att du inte skall bli lidande av kravet att bara upprepa i rimlig utsträckning. Att återanvända någons bild som visar en översikt av alla datatyper i C är rimligt men att återanvända någons lösning är det inte. Vid tveksamheter, diskutera alltid med kursansvarig eller din grupp.

Observera! Labbassarna, grupppassarna och de kursansvariga är på din sida. Deras uppgift är att du skall lära dig detta ämne så bra som möjligt. De är absolut inte ett hinder som du skall överkomma. Vi skall arbeta tillsammans för att maximera utkomsten av den här kursen!

O. Sammanfattning

Ett av målen med denna kurs är att du skall skapa ett fungerande arbetsflöde som innehåller verktyg som stöder dig i din programmering. Idealiskt skall du uppleva att verktygen är ”naturliga förlängningar av dig själv” och du skall kunna hoppa sömlöst mellan olika verktyg och aktiviteter, t.ex. från editering

²⁵Ett brott som inte sällan bestraffas med avstängning.

²⁶Som anses mycket fint.

till terminalen för att använda Make, och in i gdb när programmet kraschar (allt detta kan göras inifrån Emacs förstås).

Vad som passar i ett arbetsflöde är naturligtvis personligt. Av flera anledningar²⁷ kommer vi dock på denna kurs att kräva bland annat:

1. Emacs för all editering (ev. kommer vi att kräva Netbeans under Fas 1)
2. Make för bygghantering
3. Valgrind för minneskontroll
4. Git för versionshantering
5. gdb för debuggning
6. Doxygen och/eller JavaDoc för generering av API-dokumentation
7. L^AT_EX för alla rapporter

Det är med verktygskedjor som med programmering: man måste nöta och nöta och nöta för att till slut bemästra dem. Det är sannolikt att du kommer att uppleva att vissa saker i denna påtvingade verktygskedja är dåligt, och andra saker bra, och några fantastiska. Då har du en bra bas att stå på för att börja designa din egen verktygskedja och ditt eget optimala arbetsflöde!

Ytterligare råd

1. Kompilera ofta, gärna mellan varje ändring
 2. Åtgärda alltid kompileringsfel uppifrån och ned!
 3. Använd Emacs för att kompilera och navigera till den plats i koden där kompileringsfel finns
- Varje hopp mellan t.ex. Emacs och terminalen gör att du riskerar att tappa bort var du håller på med. *Skapa ett fungerande arbetsflöde för att hålla fokus!*

Det var det!

Grattis, nu har du redan stött på de flesta verktyg etc. som kommer att användas på kursen, och sett en hel del av den imperativa programmeringen i C! Tanken med denna inledande duvning är inte att du skall tänka "nu är jag av med det" utan tvärtom – nu vet jag vilka verktyg jag skall bemöda mig om att använda i mesta möjligaste mån under resten av kursen!

Efter denna sprint återstår två sprintar till i denna fas, som alltså behandlar imperativ programmering exemplifierat med C. På fredag eller måndag skall du redovisa X 62 och därefter tillsammans med alla andra i gruppen diskutera hur man lämpligen går vidare, d.v.s. vilka mål man vill satsa på att bocka av, och vilka uppgifter man därför bör göra.

Kursen fortsätter nu på detta sätt: varje sprint kommer du att göra minst en uppgift som du använder för att redovisa mål. (Det är en ohyggligt dålig idé att först välja uppgift och sedan "se vilka mål man kan redovisa".) Du redovisar mål vid labbtillfällen och gruppmöten som i de flesta fall sker fredagen eller måndagen i anslutning till den sista helgen av varje sprint.

Den sista fasen är projektarbetet och då är arbetet något friare. Observera att du förväntas använda projektet till att bocka av mål, och att målet med projektet inte är att göra en perfekt implementation av en specifikation.

Avslutningsvis finns det bara ett sätt att lära sig programmering och det är genom att skriva så mycket egen kod som möjligt och läsa så mycket kod skriven av andra som möjligt! Gör't!

²⁷Dessa går vi igenom på annan plats.