UMEÅ UNIVERSITET
Department of Instutition
Project

September 11, 2021

Crash Course I C för Introduktion till C och MATLAB.

Name Arvid Bergman
Username dv20arn@cs.umu.se

Contents

1 Om dokumentet

Hej! Jag som skrivit detta dokument heter Arvid och går Kand. i Datavetenskap och ska vara en av era handledare under denna kurs! Många av er har säkert inte programmerat något innan detta, och kanske känner er nervösa över hur det ska gå. Det är ingen fara! Jag ska i detta dokument försöka ge er lite bra information, tips och tricks och lite extra uppgifter som ni kan göra utöver de ni får av era lärare.

2 Ordlista.

Det finns många ord och begrepp inom programmeringsvärlden. Här är några som kan vara bra att ha koll på när man googlar(vilket ni kommer göra. Mycket).

- **IDE** - Det du programmerar i. Atom är exempelvis en IDE.
- **Curly Bracket** - I C och många andra språk så använder man funktioner för utföra saker. De omsluts av *Curly Brackets*, aka måsvingar.
- **Algoritm** - En algoritm är en ändlig uppsättning otvetydiga instruktioner som efter de körts utför något. Det finns ex. massa olika sökalgoritmer, för hur man hittar något man söker mest effektivt. Jag tror ni lär er mer om detta i slutet av kursen!
- **Kompilera** - För C måste man kompliera sina program. Det betyder att man översätter sin kod till maskinspråk. Detta är inget du behöver göra, men du behöver ge kommandotolken rätt kommandon för att utföra det. Man kan kompilera med olika flaggor som ger varningar för olika saker, men i denna kurs tror jag ni enbart kommer använda er av flaggan -Wall.
- **Integer** - ett heltal: 1, 2, 3.. ja ni fattar!
- **Double** / **Float** - Decimaltal. Double och float är lite olika i "storlek", men detta är inte så jätte relevant för er i denna kurs.
- **Boolean** - ett sanningsvärde. Det kan vara 1 eller 0 (där 1 är sant och 0 falskt) eller om man använder include <stdbool.h> så kan man skriva True eller False.
- **Semikolon** - Jag tror de flesta av er vet vad detta är, men i C (och ex. Java) så avslutar man en rad eller kommando med ;. Om man inte gör detta så kommer kompilatorn klaga!

- **Pekare** - Kan vara er värsta fiende denna kurs. Ska förklara pekare senare i dokumentet, men kort sagt så är en pekare något som pekar på allokerat minne i datorn. Känns detta luddigt så är det för det är det. Pekare är generellt det som de flesta har mest problem med, och jag har fortfarande det.

- **Loop** - En loop är som det låter. Du kan loopa på 3 sätt, med en For-loop, en While-loop och en Do-While-loop. Avslutas ert program aldrig så kan ni hamnat i en oändlig loop!
- **Iteration** - Ett varv i en loop är en iteration.
- **Nästade loopar** - Man kan ha loopar i loopar. Detta kan snabbt spåra ur, så var försiktiga med detta!
- **Tidskomplexitet** - Hur lång tid programmet tar att exekvera (väldigt ytlig beskrivning, det är mycket mer än det).
- **Ordo notation** - O(n) är Ordo notation. Det håller ni inte på med denna kurs, men kan vara bra att känna till. Om du har en loop på n iterationer så är Ordo O(n). Har du två loopar i varandra blir det istället $O(n^2)$. Man vill alltid sträva efter en så låg tidskomplexitet som möjligt.
- **Overflow** - Double, Float och Integer har alla en maximal storlek. Detta får du googla själv hur stor det är. Men tar du två tal som ligger nära denna begränsning och t.ex. multiplicerar dem så får du vad som kallas Overflow, vilket kan resultera i att programmet kraschar.
- **Pseudo-kod** - Ibland när man skriver kod så skriver man en skiss av vad koden ska göra först. Detta kallas pseudo-kod och det finns inget rätt och fel här(kanske lite). Det är till för att du skapa dig ett logiskt flöde av koden. Pseduokod kan verka meningslös ibland, men se det som ett verktyg för dig själv, en liten "ritning" för din verkliga algoritm.
- **Rekursion** - Rekursion är något som förvirrar de flesta. Kort sagt är rekursion när du kallar på "dig själv". Det kan vara en funktion som kallar på sig själv, tills ett basfall uppfylls och man ramlar ut ur rekursionen. Det förklaras nog bäst med en bild.

- Spaghetti kod - Denna är mest för min skull, men skriv inte spaghetti kod. Spaghetti kod är kod utan struktur och svårläst.
- Syntax och Semantik - Syntax är hur du skriver, och är värt att lära sig! Semantiken är vad du skriver, exempelvis hur du namnger saker.

Det finns såklart MASSVIS med fler begrepp, men jag hoppas ni får lite hjälp av dessa.

3 IDE

För att skriva kod så kan man i princip använda vad som helst för att skriva i, men det ni blir rekommenderade i denna kurs att använda är Atom. Personligen tycker jag att VS Code är betydligt bättre, men det lämnar jag till er att avgöra och testa. I er IDE så kan man ladda ner olika extensions, vilket kan göra er kodningsupplevelse betydligt mer angenäm. Problemet med vissa extensions är att de gör "för mycket" för er.

Det finns två stycken saker jag rekommenderar att ni laddar ner dock, och det är dessa två:

- **Ett trivsamt tema!** Det finns HUR många teman för din IDE som helst, här är det bara preferens som gäller. Detta gör mycket, lovar! Här är några populära teman:
  VS Code: `https://visualstudiomagazine.com/articles/2021/07/07/vs-code-themes.aspx`
  Atom: `https://www.dunebook.com/atom-themes/`
- **Bracket Pair Colorizer**. Ett av de vanligaste problemen man har när man kodar är att man tappar bort sig i alla curly brackets. Detta extension ger par en gemensam färg, vilket gör det lättare att se i vilket scope man befinner sig i.

Jag länkar sidan till VS Code: `https://code.visualstudio.com/`.
Notera att jag kan inte garantera att jag kan komma att hjälpa er med att sätta upp er VS Code utan detta får ni lösa själv, men det är inte komplicerat alls. Det som är fördelen med VS Code är att det är den överlägset mest populära IDE:en, och ni kommer mest troligt inte använda er av Atom senare i livet och studierna.

4 Att börja koda.

Äntligen kommer vi till kodandet. I C så startar alla program med en Main-funktion. En typisk start för ett C-program kan se ut såhär:

```
#include <stdio.h>    <-- Bibliotek med funktioner.

int main(void) {
    Här skriver vi kod!
}
```

Man brukar alltid skriva ett program "Hello World!" som sitt första program. För det ska vi utnyttja stdio.h:s funktion printf(). Det kan se ut såhär:

```
#include <stdio.h>    <-- Bibliotek med funktioner.

int main(void) {
    printf("Hello world!");
    return 0;
}
```

Kom ihåg semikolon! Return 0 säger åt programmet att det är slutet på programmet. För att ni ska kunna göra detta ska ni också kompliera det. För att kompliera behöver du vara i din terminal (kommandotolk) navigerad till dit du skriver ditt program. Jag lämnar denna länk för att lära er navigera terminalen:
`https://www.digitalcitizen.life/command-prompt-how-use-basic-commands/`

4.1 Kompilera ditt program.

För att kompliera ditt program skriver du in följande:
gcc -std=c99 -Wall -o namnpådinfil namnpådinfil.c

- gcc - namnet på kompilatorn. Troligtvis behöver ni installera denna! (jag tror ni får länk för detta).
- -std=c99 - Ni ska komplilera era program enligt standarden c99.
- -Wall - En varnings flagga! Denna ska ni alltid använda i denna kurs. Det finns massvis med olika flaggor, men bry er inte om dessa nu.
- -o namnpådinfil - Den komplilerade filens namn. Måste inte vara namnet på filen, men hjälper. Om man glömmer denna brukar den ge filen namnet a.out.
- namnpådinfil.c - Filen som ska kompileras.

När ni skriver era program finns de flera olika standarder att ta hänsyn till, som exempelvis namn på variabler. Börja inte ett variabelnamn med en siffra (1first

t.ex.), använd inte reserverade ord, var tydliga med vad variablen är till för. (I mer avancerade program är det rätt jobbigt om du döpt variablen till a och man vet inte vad den används till). Kom ihåg att deklarera era variabler innan ni använder dem!

4.2 Printf och Scanf

Några funktioner ni kommer använda mycket i denna kurs är printf och scanf. De gör ungefär vad namnet antyder, printf skriver ut och scanf skannar efter inputs (tangentbordet!).

```
#include <stdio.h>

int main(void) {
    int skanna_till_mig;

    scanf("%d", &skanna_till_mig);
    printf("Numret du skrev var %d", skanna_till_mig);
    return 0;
}
```

Du undrar kanske vad % och & gör. % symboliserar en platshållare, och just %d är för heltal(alltså Integers!). Det finns lite olika, här är de viktigaste för er:

- %c tecken, typ a b c. (Läs chars).
- %d heltal.
- %f flyttal m. dubbelprecision.
- %e flyttal i exp. form.
- %s Sträng. En sträng är en text, typ "Hej!".
- Skulle du vilja bestämma ex. antalet decimaler kan du skriva %5.2f, vilket ger 5 platser, 2 decimaler. (Högerjusterat!). Ex. 3.14.

& är för variabelns adress. Detta kommer ni se mycket av när ni använder pekare!

4.3 Operatorer.

Det finns en uppsjö av operatorer när det kommer till programmering. Dessa kan variera lite mellan språk, men är för de mesta lika.

- ! - Ett utropstecken betyder not. Om du skulle skriva exempelvis !true skulle det alltså betyda False. Kan användas i alla logiska operationer.
- && - Och. Alla påstående som kopplas ihop med ett && måste vara sanna!

- || - Eller. Antingen det eller det.
- % Modulo.
- <, > - Mindre än, större än.
- <=, >= - Mindre än eller lika med, större än eller lika med.
- = - Tilldelning! Denna kan vara grund till många fel i början. Om du säger 1+1 = 2 så kommer koden inte fatta vad du menar, utan = använder du för att säga ex. två = 2.
- == - Det här är lika med! Här gäller 1+1 == 2, men inte två == 2.

4.4 Val-satser

Val-satser är väldigt användbara och nödvändiga inom programmering, och används för att styra flödet av koden. Det ger dig möjligheten till att göra olika saker beroennde på olika faktorer.

4.4.1 If-satser

If-satser är rätt enkla att förstå. Du har ett villkor, om det är sant så går du in i if-satsen, annars skippar du den.

```
if(villkor) {
    gör något!
}
```

4.4.2 Else if

If-Else är en variant av if, och gör vad det låter som.

```
if(villkor) {
    gör något!
}
else if(villkor) {
    om inte if-satsen gjorde något, gör något här! OBS! bara om villkoret är sant.
}
```

4.4.3 Else

Om ingen av if eller if else satserna var sanna så kan du ha ett Else-fall. För din else behöver inget villkor uppfyllas.

```
if(villkor) {
    gör något!
}
else if(villkor) {
```

```
    gör något annat!
}
else {
    om inte if eller if else gjorde något gör något här istället!
}
```

4.5 Loopar

Det finns tre olika typer av loopar man använder sig av i C. Jag förklarar dem kort med exempel.

4.5.1 For-loop

Kanske den mest använda loopen. En for-loop gör något ett antal gånger.

```
for(int i = 0; i < iterationer; i++) {
    gör något för varje iteration!
}
```

i:et blir en loop-variabel, och iterationer hur många gånger vi ska snurra i loopen. i++ räknar upp i:et efter varje iteration. Skulle iterationer vara 3 skulle vi göra tre varv i loopen.

4.5.2 While-loop

While satsen används när du vill loopa tills ett villkor har uppfyllts. Var försiktiga så ni inte hamnar i en oändlig loop här om ert villkor aldrig uppfylls!

```
while(villkor) {
   gör något tills villkoret är uppfyllt.
}
```

4.5.3 do-while-loop

Det är typ en while sats, men man utför något innan loopen.

```
do {
   kommer alltid utföra denna rad minst en gång!
} while(villkor)
```

4.5.4 Nästlade loopar

I ordlistan nämnde jag ordet nästlade loopar, och det är något ni kommer stöta på under denna kurs. Jag ska visa hur en nästlad for-loop kan se ut.

```
...
for(int i = 0; i < iterationer; i++) {
    // Här kan du göra något
    for(int j = 0; j < iterationer; j++) {
        // Här kan du göra något
    }
    // Här kan du också göra något :)
}
...
```

i och j är vanliga variabel namn för for-loopar, där i brukar vara den yttre loopen och j den inre. Har man en tredje (rekommenderar ej!) så brukar man döpa den till k osv. Men detta är såklart bara en standard och inget måste. Antalet iterationer måste inte heller vara lika i de två for-looparna, ex. i-loopen kan köra 10 gånger medans j-loopen kan köra 5 gånger.

4.6 Tilldelnings operatorer

Det finns lite smidiga förkortningar för tilldelningar, här är några och vad de betyder.

- n+=2 betyder n = n + 2
- n-=2 betyder n = n - 2
- n*2 betyder n = n * 2
- n/=2 betyder n = n / 2

4.7 Funktioner

Funktioner har en otroligt viktig funktion(haha :–)) när du programmerar. I C kallas de funktioner, i Java kallar man det metoder. Kärt barn har många namn. En funktion kallas från någon annan funktion (exempelvis din main-funktion), och har som syfte att utföra en uppgift. För att undvika att det blir spaghetti så försök låta en funktion göra en sak och enbart en sak! Med det menar jag att du inte gör en funktion som heter ”add_one_then_multiply_with_ten” utan hellre två funktioner som heter ”add_one” och ”multiply_with_ten”.

Så hur gör man en funktion? Först måste man deklarera den. Ofta gör man detta innan main, funktionen måste nämligen vara deklarerad innan den blir kallad. En funktion har några egenskaper som vi går igenom med vår funktion ”add_one”. Låt oss deklarar den först.
Vi skriver **int add_one(int my_number);**

- int - vad funktion ska returnera till den som kallar den. I detta fall blir det ett heltal(int == integer).
- ”add_one” - funktionens namn.
- (int my_number) - I parentesen skickar du med alla parametrar du vill att funktionen ska veta om. Om du deklararar en variabel i main utan att

skicka med den till funktionen så kommer funktionen **inte** veta vad det är för variabel (här finns lite undantag men det är överkurs).

- Om du inte vill få tillbaka något från funktionen eller du inte vill skicka med något kan du skriva void func(void) istället. Ibland vill man ha en funktion för att bara skriva ut text och då behöver man inte ta emot eller skicka tillbaka något.

```
#include <stdio.h>

// detta är deklarationen!
int add_one(int my_number);

int main(void) {
    int my_number = 0;
    my_number = add_one(my_number);
    return 0;
}

// detta är definitionen!
int add_one(int my_number) {
    my_number+=1;
    return my_number;
}
```

Förstår du vad koden gör? Om inte, ingen fara! Det vi gjorde först var att deklarera funktionen "add_one". Sen i vårt main så deklarerade vi en variabel my_number och satte den till 0. Sen kallar vi "add_one" till my_number. Det den då gör är att den tar my_number till funktionen, lägger till 1 till talet och returnerar det till main. Inte så komplicerat va?

Viktigt att deklarationen och definitionen stämmer överens!

4.8 Räckvidd, Define och const.

Det finns ett ord som får håren att resa sig på programmerare och det är globala variabler. Detta är definerade utanför main-blocket och är därmed synliga i hela programmet. Detta vill man **alltid** försöka undvika!

```
#include <stdio.h>

int global_variabel = 1;

void foo(void);

int main(void) {
    global_variabel är synlig här.
}

void foo(void) {
    och här! Inte bra..!
}
```

Det finns dock något som man kan använda istället. Säg att du skriver ett program som räknar matte, och vill använda pi. Pi är samma tal, hela tiden. För det kan du använda #define. För att definera pi skriver du innan main och efter dina bibliotek **#define pi 3.14**. Denna kommer kunna utnyttjas av alla funktioner och block. Const är också ett sätt definera ett tal som inte får ändras, men har som vanliga variabler begränsad räckvidd. **Const kan användas såhär: const float pi 3.14;**

4.9 Pekare

Pekar är något många, inklusive mig själv, tycker är lite små knepigt. Man kan säga att en pekare är en variabel vars värde är en adress för en annan variabel, och du kan genom pekaren direkt manipulera minnesadressen hos den variabel som pekaren pekar på. Det är inte jätte svårt att deklarera en pekare, det kan se ut såhär:

```
#include <stdio.h>

int main(void) {
    int min_variabel = 1;
    int *min_pekare;

    min_pekare = &min_variabel;

    return 0;
}
```

Notera att en pekare deklareras som en vanlig variabel fast du sätter en framför. När du tilldelar den en adress så använder du & som vi tidigare nämnde är minnesadressen för en variabel. Pekare är något ni inte kommer komma undan, och det kommer dyka upp ännu svårare i er nästa programmeringskurs (Datatyper och Algoritmer) så jag rekommenderar er starkt att **LÄR ER PEKARE**!!

Jag är alldels för dålig på pekare för att förklara det bra för er, så jag överlåter det till era framtida bästa vänner: indiska youtubers.
`https://www.youtube.com/watch?v=f2iOCnUOniA`

4.10 Arrayer

Arrayer kan ses som en lång rad av pekare. En array är på ett sätt en pekare till pekare. Det låter virrigt först, men arrayer är faktiskt inte så komplicerade. En array deklareras såhär: <typ> <namn>[<antal element i arrayen]; Om du vill ha en array med 10 heltal kan du skriva **int arr[10]**. En array indexeras från 0, så om du vill få tag i det första elementet ur en array så skriver du arr[0].

```
#include <stdio.h>

int main(void) {
    int arr[3] = {1, 2, 3};
    int  first_Element = arr[0];
    int last_Element = arr[2]; // notera att det inte är arr[3]!
}
```

Man kan även göra tvådimensionella arrayer, det är bara lägga till en []! Så exempelvis int arr[2][2]. Vi gör ett lite mer avancerat program där vi använder en for-loop för att fylla en array:

```
#include <stdio.h>

int main(void) {
    int my_array[10]; // glöm inte skriva in storleken på arrayen.
    for(int i = 0; i < 10; i++) {
        my_array[i] = i;
    }
}
Detta genererar arrayen som är fylld från 0-9, alltså:
my_array = {0,1,2,3,4,5,6,7,8,9}
```

4.11 Structs

En struct är en egendefinerad data struktur. Om du vill ha en variabel för datum så kanske du inte vill ha tre variabler för dag, månad, år. Det är här structs kommer in! En struct är en struktur som kan innehålla variabler av olika slag (till och med pekare :)) som gör livet lättare för dig. En struct för datum kan se ut såhär:

```
typedef struct date {
    int date;
    int month;
    int year;
} date;
```

Det finns lite regler till hur du initialiserar din datatyp. Vi skriver ett kort program för att se hur man initaliserar och hur man får reda på information från structen.

```
#include <stdio.h>

// Skriv inte denna i main!
typedef struct date {
    int date;
    int month;
    int year;
} date;

int main(void) {
    date todays_date = {01, 10, 2021};
    printf("Current year is %d", todays_date.year);
}
```

Det är viktigt att man initialiserar structen som ovan. En struct ska tilldelas värden när den skapas!

4.12 Andra bibliotek

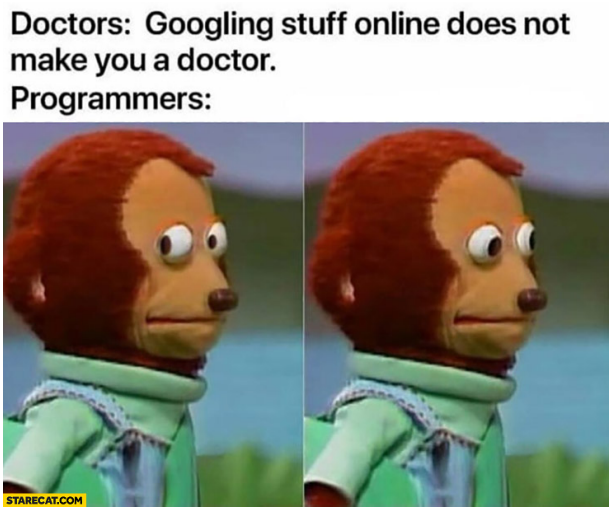
Det finns som sagt massvis med olika bibliotek med massvis med funktioner (även om de i C är rätt primitiva, jämfört med andra språk). Ni kommer under denna kurs troligtvis mest använda er av dessa:

- <**stdio.h**> - Detta är nästan alltid med!
- <**stdlib.h**> - Innehåller funktioner som rand och srand.
  `https://www.tutorialspoint.com/c_standard_library/stdlib_h.htm`
- <**time.h**> - används ofta tillsammans med srand.
  `https://www.tutorialspoint.com/c_standard_library/time_h.htm`
- <**ctype.h**> - Innehåller massvis med smidiga funktioner. Kolla upp med lärare / handledare innan ni använder dessa i labb om det inte står specifierat!
  `https://www.tutorialspoint.com/c_standard_library/ctype_h.htm`
- <**stdbool.h**> - Låter dig använda true och false istället för 1 och 0 som sanningsvärden.
  `https://riptutorial.com/c/example/11458/using-stdbool-h`
- <**string.h**> - Innehåller massa smidiga funktioner kopplat till strängar. Som de i ctype, kolla innan ni använder dessa!
  `https://www.tutorialspoint.com/c_standard_library/string_h.htm`

4.13 Slutord kring att börja programmera

Det finns hur mycket som helst man kan göra i C, men mycket lär man sig från att googla. Under denna kurs så ska ni inte göra några jätteavancerade program men de kan ändå kändes övermäktiga. Om det känns jobbigt så är det bästa man kan göra att ta en paus, gå bort från tangentbordet och tänka på något annat(bara ni inte gör det för ofta). Ni kan alltid maila mig också, men jag kan inte garantera att jag kan svara på allt / har tid.

Det är inte fusk att googla, utan en tillgång. MEN! Använd inte kod du inte förstår vad den gör, och kopiera inte direkt över, utan låt det ge dig en knuff i rätt riktning istället.



5 Tips och tricks

- Ha en bra arbetsergomi när du programmerar! Det är lätt att få ont i leder / nacke om man sitter dåligt, och då blir det ganska bokstavligt en pina att koda.
- Lär dig ge funktioner och variabler bra namn! Ett bra namn är att man förstår vad en funktion eller variabel gör utan att kolla på vad de faktiskt gör.
- Kommentera gärna din kod - men inte för mycket! Gör ni något konstigt i koden kan det vara värt att skriva en kort kommentar vad som händer, men ni behöver inte skriva att ni gör en loop.

  (`https://www.youtube.com/watch?v=k238XpMMn38&ab_channel=shounic`)

  Vidare (detta är lite personlig preferens) ska inte en kommentar beskriva HUR en funktion gör något, utan VAD den gör.

```
Detta är inte en jättebra kommentar.
/*
Searches the array for a specific number given as
a parameter by using a for-loop and checking if
index of the array matches the searched number.
If it matches the function returns true,
otherwise it returns false!
*/

Detta är en bättre kommentar.
/*
Function searches array for number supplied as
parameter.
```

```
Returns true if found, else false.
```

Den ”Bättre” kommentaren kan anses vara i underkant hur kort de ska vara, men ni förstår poängen. Skriv inte mycket bara för att, utan skriv vad funktionen faktiskt gör. Ni kommer senare (tror det är under DOAn) lära er kommentera er kod bättre.

- Returnera sanningsvärden kan man ofta göra rätt simpelt.

```
bool isTwo(int number) {
    if(number == 2) {
        return true;
    }
    else {
        return false;
    }
}
```

```
Detta fungerar! Men det går att göra bättre.
```

```
bool isTwo(int number) {
    return number == 2;
}
```

```
Denna gör samma sak, men blir betydligt lättare
att läsa och snyggare.
```

- Lär er att använda switch-sats där det är befogat. Har du 3+ if else kan det vara dags att fundera på en switch-sats istället.
- Kommentarer igen - ibland kan det vara bra att skriva mental notes i koden för sig själv, men ta gärna ut dem så det inte blir fullt av små konstiga kommentarer i koden.

```
...
int *p = &pp; // this will point to..
...
```

- Fråga om hjälp när det behövs! Ibland skiter det sig, och även om handledare ofta inte kan ge dig lösningen på dina problem kan en knuff vara allt man behöver.
- Träna! Koda lär man sig bäst av att just koda. Man kan läsa om kod mycket men inte lära sig något ändå. Det finns hur mycket resurser online som helst, använd dem!
- Appropå träna, slår ett slag för hemsidan `https://open.kattis.com/`. Där finns det massvis med uppgifter i problemlösning du löser med kod och som testas av deras system. Om du får godkänd lösning får du poäng, som du kan klättra på leaderborden med. Kom ihåg att registrera dig på Umeå Universitet!
  (OBS: Många kattisuppgifter kan vara rätt dryga att göra i C).

6 Lite svårare uppgifter.

Gör gärna dessa, men gör de ni får från kursen först.

6.1 Funktioner och Pekare

- Deklarera en funktion som inte returnerar något, men tar in tre parametrar av olika typer.
- Definiera en funktion som inte returnerar något som tar adressen till ett heltalsobjekt som argument och ökar värdet på objektet med 1.
- Gör en swap-funktion. Funktionen ska heta swap och ta två parametrar, och efter funktion har körts ska de två parametrarna bytt värde.

```
int a = 1, b = 2;
swap(a, b);
// a = 2, b = 1
```

- Skriv ett program som läser in två olika decimaltal till två variabler och sen byter värdet på dem (hint: en swap funktion?). Innan programmet avslutas ska allt skrivas ut som nedan:

```
...
Före swap: a = 3.14, b = 2.71
Efter swap: a = 2.71, b = 3.14
```

6.2 Val satser

- Definera en funktion boom_bang som givet ett heltal **n** skriver ut talet **n** för:

```
- om n är delbart med 3 skriv ut "boom".
- om n är delbart med 5 skriv ut "bang".
- om n är delbart med 3 och 5 skriv ut "boom bang".
```

  Ni får välja om ni vill använda if-satser eller switch-sats, eller gör den för båda!

- Skriv en val-sats som beroende på om den får 0, 1, 2 eller 3 skriver ut Hjärter, Spader, Ruter eller Klöver.

- (lite svårare) Skriv en rekursion som skriver ut Fibonacci sekvensen av ett tal. Fibonacci av 5 är t.ex. 0 1 1 2 3.

- Skriv definitionen av en funktion som returnerar summan av alla heltal från 1 till **n**, där **n** ges som aktuell parameter till funktionen. Använd rekursion!

6.3 Iterationer

- (lite svårare) Definiera en funktion med deklarationen **void print_figure(int type, int size)** som skriver ut följande fem typer av figurer. Vilken av dem bestäms av **int type**.

```
type=0  type=1  type=2  type=3   type=4
XXXXX   XXXXX   X        XXXXX   XXXXX
XXXXX   X   X   XX        XXXX   X
XXXXX   X   X   XXX        XXX   X
XXXXX   X   X   XXXX        XX   X
XXXXX   XXXXX   XXXXX        X   X
```

  Tips: Nästlade loopar. Värdet på loop-variablerna kan säkert hjälpa er..!

- Definiera en funktion via iteration som skriver ut alla heltal mellan talen a och b (inkl. a och b) som är udda, d'r a och b är parametrar, samt a > 1 och b $>$ a. I utskriften ska alla tal skiljas åt med ett tomt tecken(mellanslag). Ex: a $= 3$ och b $= 8$ så ska ert program skriva ut 3 5 7.

6.4 Arrayer

- Skriv en funktion som tar en två-dimensionell array av int som argument (samtargument för de två dimensionernas storlekar) och returnerar medelvärdet av deingående elementen. Kom ihåg att använda const där det är befogat. Testkör din kod flera gånger så att du är säker på att den räknar rätt.
- Skriv en funktion som skriver ut innehållet i en sträng baklänges, ett tecken i taget. Funktionen ska ha endast en formell parameter av typen pekare till char. Tänk på att använda dig av ”slutet-på-strängen”-tecknet. Kom ihåg att använda const där det är befogat.
- Skriv en funktion som byter ut alla vokaler i en sträng mot ett mellanrum.
- Skriv en funktion som lägger till en sträng till slutet av en annan.

6.5 Structs

- Deklarera en strukturtyp som representerar en person (förnamn, efternamn, födel-sedatum).
- Vi vill representera n stycken punkter(xi, yi)i planet. Ett sätt är att lagra varje punkt i en struktur och sedan lagra strukturerna i en array av storlek n. Ett annat sätt är att lagra alla x-värden i en array av storlek n och alla y-värden i en annanarray av storleknoch sedan lagra de två arrayerna i en struktur. Översätt bägge alternativen till deklarationer av typer och variabler.