En kurs i C-programmering

Föreläsningsanteckningar från AD1/PK2 VT06

Jesper Wilhelmsson

March 11, 2006

Innehåll

Föreläsning 1

Introduktion till C

Välkommen till denna grundkurs i programmeringsspråket C. Denna kurs riktar sig till er som inte har någon som helst tidigare erfarenhet av programmering. Att lära sig programmera görs lättast genom att prova själv och vi kommer därför att gå igenom språket med hjälp av exempel. Det underlättar förståelsen om ni har möjlighet att på egen hand prova de exempel som visas.

Alla exempel är skrivna i ANSI-C och är fullständigt plattformsoberoende, men de exempel som visar hur man kompilerar och provkör programmen (vilket per definition är plattformsberoende) förutsätter att det är `gcc` som används. En del terminologi kan även förutsätta viss bekantskap med unix/linux.

Att programmera är att lösa problem. En av de viktigaste egenskaperna en programmerare bör ha är förmågan att abstrahera – att kunna bortse från detaljerna som rör det specifika problemet. När man bortser från detaljerna och jämför problemet man vill lösa med andra problem man redan löst kan man se likheter. Genom att utnyttja dessa likheter kan man abstrahera problemen och deras lösningar och då öppnas möjligheten att återanvända lösningar genom att anpassa den generella lösningen till det specifika problemet.

Det är viktigt att man som programmerare inte låter sig avskräckas av att en del problem ser svåra ut på grund av sin storlek. Genom att bortse från detaljer kan man dela upp problem i mindre delar som var för sig är betydligt enklare att lösa. Vi ska se hur detta går till och vilka konstruktioner som C erbjuder för att underlätta problemlösningen.

1.1 Bakgrund

Språket C har sitt ursprung i Unix. Det är en vidareutveckling av B (1970) som utvecklades för att skriva applikationer till den första Unix:en på en DEC PDP-7. Språket C såg dagens ljus 1972. 1978 definierade Kernighan och Ritchie den version av C som kom att bli dominerande. ANSI-C (1988) är en utveckling av K&R C där man filat bort en del av de kantiga hörnen och specificerat ett brett funktionsbibliotek.

C är som sagt utvecklat för att skriva applikationer till Unix och ligger därför mycket nära operativsystemet. Det finns inga skyddsmekanismer, om programmeraren gör fel så dör programmet på riktigt och har man otur så tar det andra delar av systemet med sig i fallet. Detta till trots så är ANSI-C plattformsoberoende. Så länge man inte ger sig på riktigt håriga saker i minnet, så kan man flytta en applikation från hårdvara till hårdvara från operativsystem till operativsystem utan att behöva ändra någonting i källkoden. Linux är ett lysande exempel på detta. De flesta av de program som används i Linux är skrivna i C, och alla dessa kan kompileras på vilken hårdvara som helst för att köras på Linux för till exempel x86 eller PPC eller flyttas till andra operativsystem utan att källkoden behöver förändras. (Cygwin för Windows är ett exempel på hur Linux-program flyttats till ett annat operativsystem, Geek Gadgets för AmigaOS ett annat.)

Kompileras ja. För att köra ett C-program måste man kompilera (översätta) det till maskinkod. Det finns ingen kommandorad där man kan testköra enstaka funktioner eller beräkna små uttryck

som en del andra programmeringsspråk ibland erbjuder. C är inte ett interpreterande språk. Det är viktigt att förstå att det kompilerade programmet *inte* är plattformsoberoende – endast källkoden har denna egenskap. Ett kompilerat program är anpassat för exakt den hårdvara och det operativsystem det är kompilerat för. Det kompilerade programmet startas precis på samma sätt som man startar vilken annan applikation som helst. Exakt hur detta går till återkommer vi till i slutet av denna föreläsning.

C hanterar samma datatyper som maskinen; tecken, siffror och adresser. Det finns inga inbyggda operationer för att hantera listor, strängar med mera Det finns ingen inbyggd hantering av I/O. Det C erbjuder är sekventiell kod uppbyggd av satser och språket lämpar sig mycket bra för lösning av sekventiella problem, det vill säga problemlösningar där ett antal steg ska ske i en given ordning.

"Sekventiell kod uppbyggd av satser", vad är det då? En sats är en instruktion eller ett kommando. Det kan vara allt från en enkel addition till ett avancerat matematiskt uttryck eller ett funktionsanrop. Alla satser avslutas med ett semikolon (`;`) och en sekvens av satser är helt enkelt ett antal satser som skrivs efter varandra i ett program.

Under denna föreläsning kommer vi att gå igenom ett litet C-program och vi kommer att titta närmare på de delar av C som behövs för att få programmet att fungera. Problemet som ska lösas är att beräkna arean av en cirkel. Problemet är enkelt och vi kan direkt se att dess lösning består av tre delar:

1. Inmatning (få tag i radien)
2. Beräkning (av arean)
3. Utskrift (av arean)

Utan att kunna något om programmering kan vi beskriva lösningen med pseudokod[1]. Detta är en intuitiv steg-för-steg-beskrivning av vad som behöver göras för att beräkna en cirkels area:

1. $radie \Longleftarrow$ få tag i en radie
2. $area = pi * radie^2$
3. skriv ut $area$

Det man främst är intresserad av i C-program är sidoeffekter. Sidoeffekter är saker som att skriva ut värden, uppdatera filer på hårddisken eller skriva data till en port (nätverk, skrivare och så vidare). Detta är karakteristiskt för så kallade *imperativa* språk. I andra typer av programmeringsspråk, som till exempel funktionella språk, är sidoeffekter inte det primära målet, utan man är istället ute efter ett returvärde – programmet ska beräkna ett värde som skickas tillbaka till den som anropade programmet. I vårt exempel är det dock sidoeffekten, att arean skrivs ut, som är det intressanta. Det är en vesäntlig skillnad på att skriva ut ett värde och att returnera det. Vi återkommer till returvärden i en senare föreläsning.

1.2 Variabler, Typer, Konstanter

Variabler är lagringsplatser för data. Variabler har många användningsområden men typiskt så används de för att mellanlagra värden vid beräkningar. Alla variabler som används i ett C-program måste *deklareras* innan de används. Att deklarera en variabel innebär att man talar om för C vad variabeln heter och vad den har för typ. Typen talar om vilken sorts data som variabeln kommer att innehålla. Om ingen typ anges är standardtypen `int` (heltal). Under denna kurs får ni dock inte utelämna typen utan alla variabeldeklarationer ska inkludera en typ. De inbyggda typer som C tillhandahåller visas i tabell 1.1.

[1] pseudokod = låtsaskod / nästankod

Typ	sizeof()	Kommentar
`char`	1	Heltal. Alltid en byte
`int`	4	Heltal. Oftast maskinens ordstorlek
`short`	2	Heltal. 16 bitar = -32.768 – 32.767
`long`	4	Heltal. 32 bitar = $-2.147.483.648$ – 2.147.483.647
`long long`	8	Heltal. 64 bitar = $-922 * 10^{16}$ – $922 * 10^{16}$
`float`	4	Flyttal. Oftast 32 bitar, 6 signifikanta siffror, 10^{-38} – 10^{38}
`double`	8	Flyttal. Normalt högre precision än `float`
`long double`	16	Flyttal. För riktigt stora tal

Table 1.1: Datatyper och dess storlek i minnet på en 32-bitars-maskin.

`En typisk variabeldeklaration: int x;`

Olika datatyper kan representera olika stora mängder data. Ett normalt heltal (`int`) representerar till exempel en större mängd än ett tecken (`char`). Detta innebär att olika typer kräver olika mycket minne. Den exakta siffran på hur mycket minne en variabel kräver är både beroende på kompilator och hårdvara. Främst beror det på hur stora maskinord hårdvaran kan hantera. De flesta datorer idag är 32-bitars-maskiner. 32 bitar syftar just till storleken på ett maskinord, och i en dator med 32-bitars maskinord är den "normala" storleken på en datatyp just 32 bitar (= 4 bytes). En `int` är urtypen för en normal datatyp i C. 64-bitars-maskiner börjar bli vanligare och i dessa kan alltså en `int` lagra betydligt större tal än på en 32-bitars-maskin. För att se hur mycket minne en datatyp tar (och därmed hur stora tal de kan lagra) kan man använda kommandot `sizeof()`. `sizeof(int)` ger till exempel svaret att en `int` kräver 4 bytes minne på en 32-bitars-maskin och 8 bytes på en 64-bitars. Datatypernas storlekar finns i tabell 1.1. Siffrorna i tabellen är hämtade från en 32-bitars-maskin.

I de flesta fall är alla datatyper försedda med tecken om man inte säger något annat. Det vill säga, de kan lagra både negativa och positiva tal. Vill man ändra detta kan man använda nyckelorden `signed` och `unsigned`. På detta sätt kan vi skapa variabler som endast innehåller positiva tal och därmed få dubbelt så stora tal att leka med. När det gäller `char` så beror det på plattformen om den har tecken eller ej och det enda sättet att få helt maskinoberoende program är att specificera typen som `unsigned char` eller `signed char`.

För att ge programmeraren fullständig tillgång till maskinen tillhandahåller C också en datatyp för pekare (adresser). Dessa är alltid samma storlek som ett maskinord oavsett vad de pekar på. För att deklarera en pekare lägger man till en `*` framför variabelnamnet: `int *x;` Vi återkommer till pekare i en senare föreläsning, här vill jag bara tala om att de finns.

En variabel får sitt värde genom en tilldelning, detta skrivs med likhetstecken: `x = 42;` Om man inte initierar värdet på sina variabler är deras värde ospecificerat. En del kompilatorer initierar automatiskt värdet till 0, andra gör det inte. Om variabler inte initieras kommer de att innehålla slumpmässigt skräp. Det enda sättet att vara säker på att en variabel har värdet 0 initialt är att initiera den själv.

Man kan även flytta ett värde mellan två olika variabler: `y = x;` Många gånger kan man flytta data mellan variabler av olika typ utan att C säger något om det. Detta är möjligt då hela datavärdet får plats i den nya datatypen.

```
int liten = 4711;                int heltal = 42;
long long stor;                  double flyttal;
stor = liten;                    flyttal = heltal;
```

Detta går bra eftersom en `int` är mindre än en `long long` och ett heltal kan betraktas som ett flyttal utan decimaler. Detta kallas *implicit typkonvertering* - typkonverteringen sker under ytan utan att vi behöver veta om den. Men om man vill flytta åt andra hållet så går det inte lika bra. C kommer att ge en varning, vilket vi i denna kurs betraktar som ett fel. Detta betyder naturligtvis inte att det är omöjligt (eller ens svårt) att flytta data från stora datatyper till mindre, C är ett

ganska lättövertalat språk som förutsätter att programmeraren vet bäst. För att få tyst på C måste man typkonvertera *explicit*. Man talar om för C vilken typ ett värde har genom att sätta typen inom parentes före värdet. C ignorerar då vad den egentliga typen är och decimaler och andra delar av större datatyper som inte får plats klipps bort.

```
liten = (int)stor;

int x = (int) 3.852       -> x = 3
char c = (char) 258       -> c = 2 (1 0000 0010 -> 0000 0010)
```

Man kan även deklarera att en variabels värde inte alls ska vara variabelt, utan konstant. Genom att sätta nyckelordet `const` framför en variabeldeklaration talar man om att man inte har för avsikt att ändra på variabelns värde. Exakt vad som händer om man försöker ändra på en `const`-deklarerad variabel är kompilatorberoende. `gcc` betraktar det som ett fel och kommer att rapportera det, men det gäller inte alla C-kompilatorer.

En `const`-deklarerad variabel måste få sitt värde direkt vid deklarationen. Notera även att vi skriver konstanter med stora bokstäver. Detta är inte ett krav i C, men det ingår i den gängse kodkonventionen att göra så. Under denna kurs SKA ni skriva konstanter med stora bokstäver.

```
const int TAL = 42;
```

Vi återgår till exemplet om att beräkna en cirkels area och försöker så smått börja översätta pseudokoden till C. Innan vi kan utföra några beräkningar måste vi alltså deklarera variablerna. Vi vill ha flyttal men behöver inte allt för hög precision, så `float` duger. Flera variabler kan deklareras på samma rad om de har samma typ. Notera att vi skriver talen på engelska, vi använder alltså punkt istället för komma för att separera heltalsdelen från decimaldelen i flyttal.

```
float radie, area;
const float PI = 3.1415;
```

1.3 Operatorer

När det gäller operatorerna i C kan man nog säga att allt är precis så som man vill ha det. De fyra räknesätten (`+`, `-`, `*`, `/`) finns och fungerar så som man är van vid från matematiken. Det finns även en operator för modulo (`%`).

Även de logiska operatorerna är intuitiva, likhet (`==`), större än (`>`), mindre än (`<`), skiljt från (`!=`) med flera. Värt att notera är att likhet skrivs med två likhetstecken, detta för att skilja det från tilldelning. För att knyta ihop flera logiska villkor kan man använda `&&` (och) eller `||` (eller). Negerar logiska uttryck gör man med `!`. `(x > y)` är därmed samma sak som `!(x <= y)`.

Alla logiska operatorer resulterar i ett sanningsvärde. Det finns ingen speciell typ för detta värde, utan i C:s ögon är det ett vanligt tal precis som allt annat. Falskt = 0 och sant = 1. Egentligen är allt annat än 0 sant, men det är 1 som returneras av de logiska operatorerna.

Eftersom C ligger nära maskinen finns även operatorer för att manipulera bit-mönster. Till exempel `«` och `»` som flyttar bitarna i ett tal åt vänster respektive höger. `~` ger två-komplementet, det vill säga alla ettor blir noll och alla nollor blir ett. Logiska bitoperationer kan göras med `&` (och), `|` (eller) och `^` (xor).

++

Det finns fyra olika sätt att öka en variabels värde med ett i C. Låter det onödigt? Egentligen är det bara två av sätten som finns just för att öka en variabels värde med ett, de andra två är specialfall av vanlig aritmetik. De två sätt som finns skiljer sig på en viktig punkt – när själva ökningen av värdet sker. Det är `++`-operatorn som står för ökningen.

```
x = 5;                    x = 5;
y = x++;                  y = ++x;
```

I det första fallet (till vänster) kommer värdet på `x` att användas *innan* det ökas med ett, det vill säga `y` kommer att få värdet 5. I fallet till höger kommer värdet på `x` att ökas först och det är värdet *efter* ökningen som hamnar i `y`, alltså 6. De övriga två sätten man kan öka en variabels värde med ett är som sagt bara specialfall av de vanliga aritmetiska operatorerna: `x = x + 1;` och `x += 1;`. Den senare av dessa är bara ett förkortat sätt att skriva. Uttrycken är identiska i alla avseenden. För detaljerad information om C:s operatorer och sanningstabeller för de logiska operatorerna, se det informationsblad om operatorer som finns på kurshemsidan.

Därmed kan vi börja beräkna vår cirkel-area, och som vi ser så är C-koden identisk med den intuitiva pseudokod vi skrev innan vi kunde något om C med det lilla undantaget att upphöjt inte finns som inbyggd operator i C, så vi får skriva $radie * radie$ istället.

```
area = pi * radie * radie;
```

1.4 Villkorliga satser — `if` – `else`, `switch`

`if` – `else`

Nyckelordet `if` används för att skriva villkorliga satser som styr kontrollflödet i ett program, det vill säga för att bestämma vilken kod som ska köras. Kod som kapslas in i en villkorlig sats kommer endast att köras om det givna villkoret är uppfyllt.

```
if ( villkor )
   sats;
```

Ibland vill man köra olika kod beroende på om villkoret är uppfyllt eller ej, då kan man lägga till `else`. Om villkoret är uppfyllt körs satsen direkt efter `if` (sats1), annars körs satsen efter `else` (sats2).

```
if ( villkor )
   sats1;
else
   sats2;
```

Som vi kan se förväntar sig C endast en sats efter villkoret. För att utföra lite fler saker kan man slå in koden i { }. Satser som omges med dessa "måsvingar" betraktas som en enda sats i C.

```
if ( villkor1 )
{
   sats1;
   sats2;
}
else
   sats3;
```

Koden i de villkorliga satserna kan vara precis vilken kod som helst, även nya villkorliga uttryck. Om ett nytt villkorligt uttryck följer direkt efter en `else` skriver man normalt detta på samma rad för att öka läsbarheten.

```
if ( villkor1 )
   sats1;
else if ( villkor2 )
   sats2;
else
   sats3;
```

Notera att detta är exakt samma sak som att skriva

```
if ( villkor1 )
   sats1;
else
{
   if ( villkor2 )
      sats2;
   else
      sats3;
}
```

Villkoren som används för att avgöra vilken kod som ska köras kan även de vara i princip vilken kod som helst. Det normala är att man använder logiska uttryck som resulterar i sant eller falskt. Typiska sådana är `<`, `>` och `==`. Men man kan även utföra annan kod så länge resultatet av denna är ett heltal.

Om man sätter samman flera logiska uttryck med hjälp av `&&` eller || så kommer de att utföras från vänster till höger. Om en tidig del av villkoret på egen hand kan bestämma om slutresultatet blir sant eller falskt kommer resten av villkoret inte att exekveras. Denna ordning är klart definierad i språket vilket möjliggör kod som kanske ser farlig ut vid en första anblick.

```
if (( x != 0 ) && (( 10 / x ) > y ))
   sats;

if (( x == 0 ) || (( 10 / x ) > y ))
   sats;
```

Division med noll är strängt förbjudet i C. Om det inträffar kommer programmet att avbrytas omedelbart. Att dela med `x` kan därför vara farligt om vi inte vet vad `x` innehåller. Tack vare att ordningen är definierad i språket är det dock ingen fara här eftersom vi vet att kontrollen att `x` är skiljt från noll i det första exemplet kommer att utföras innan divisionen. Om `x` är noll kommer divisionen aldrig att utföras. I det andra exemplet gäller samma resonemang. Om `x` är noll är uttrycket direkt sant och divisionen kommer inte att utföras.

Det kan alltså vara värt att tänka på i vilken ordning man sätter villkoren även i fall då de egentligen är oberoende av varandra. Om något villkor är betydligt dyrare att exekvera än något annat gör man bäst i att sätta det dyraste villkoret sist och hoppas på att det inte ska behöva utföras varje gång.

`switch`

Det är ganska vanligt att man har kod som ska göra en serie olika saker beroende på någon variabels värde. Detta kan skrivas som en rad av `if` – `else if` – `else if` – `else if` ... För att göra det lite tydligare (och effektivare) finns det en konstruktion som underlättar den här typen av kod. Nyckelordet `switch` används för att tala om vilken variabel det gäller och olika fall (`case`) kommer att väljas utifrån värdet på variabeln.

```
switch ( variabel )
{
   case värde1: sats1; break;
   case värde2: sats2; break;
   case värde3: sats3; break;
   default: sats4;
}
```

Om värdet på `variabel` är `värde1` kommer sats 1 att utföras, om värdet är `värde2` körs sats 2 och så vidare. Man kan även ange ett standardfall (`default`) som utförs om inget annat passar. Standardfallet måste inte finnas med, men det är en god vana att alltid täcka in det oväntade.

Kommandot `break` används för att tala om att det är dags att avbryta körningen och hoppa till koden efter `switch`-satsen. Om `break` inte skrivs ut kommer körningen att fortsätta på nästa `case`. Detta kan vara användbart om det är flera fall som ska köra samma eller delvis samma kod.

```
char ch = read_from_user();
switch ( ch )
{
   case 'a':
   case 'A': do_the_A_thing(); break;
   case 'c': create_new_thing();
   case 'm': modify_existing_thing(); break;
   default: user_is_stupid();
}
```

`switch` kan endast användas på heltalsvariabler, det vill säga `int`, `char` och andra typer av heltal.

Valet mellan en rad `if`–`else` och `switch` kan ibland vara svårt. Men det finns en enkel regel som jag tycker att man bör följa. Om det är fler än två fall bör man använda `switch`. Detta av den enkla anledningen att det blir mycket renare kod med `switch`. Man ser tydligt att det bara är en variabel som avgör vilket fall som väljs. I en rad med `if`–`else` kan det ju vara något fall som beror av någon annan variabel och det är ofta svårt att se.

1.5 Loopar — `for`, `while`, `do` – `while`

För att kunna skriva lite mer avancerade program krävs något sätt att iterera (upprepa). C tillhandahåller tre typer av loopar: `for`, `while` och `do` – `while`. Det finns ett fjärde sätt, rekursion. Rekursion betyder att en funktion anropar sig själv och det är inte att rekommendera i C. Varje gång en funktion anropas i C kommer dess lokala omgivning[2] att allokeras i datorns minne. En rekursiv loop innebär alltså att programmet äter lite minne för varje varv. Om loopen håller på för länge kommer minnet att ta slut. Därför används hellre icke-rekursiva alternativ i C.

`for ( initiering; villkor; förändring ) sats;`

`for`-loopen består av fyra delar: initiering, loop-villkor, förändring och en sats som utgör loopens kropp. Samtliga dessa delar kan i princip vara godtycklig kod, men det vanliga är att man har något i stil med:

```
for ( x = 0; x < 10; x++ )
   sats;
```

För att förstå hur en `for`-loop beter sig i ett program är det viktigt att veta i vilken ordning saker sker. Lyckligtvis är även detta klart definierat i språket.

[2] Omgivning = argument med mera – vi kommer tillbaka till det i en senare föreläsning.

1. Utför initieringskod
2. Kontrollera om villkoret är uppfyllt, om inte avsluta loopen
3. Utför koden i loop-kroppen
4. Utför förändringskoden
5. Gå tillbaka till steg 2.

Ordningen avslöjar att det alltså kan hända att kroppen i en loop aldrig exekveras. Detta sker om villkoret inte är uppfyllt då loopen startar. `for`-loopens natur gör att den är mycket lämplig att använda då man vet hur många iterationer man vill utföra. Typiskt är att man använder `for` för att stega igenom datatyper av en bestämd storlek eller liknande.

`while ( villkor ) sats;`

`while` fortsätter att iterera tills dess att villkoret inte längre är uppfyllt. Detta gör loopen lämplig att använda då man inte vet hur många gånger man vill iterera, utan vill fortsätta tills dess att något händer som falsifierar villkoret. Även i `while`-loopar kontrolleras villkoret innan kroppen exekveras. Ett vanligt fel i samband med `while`-loopar är att man glömmer uppdatera en variabel som villkoret bygger på. Detta resulterar ofta i en oändlig loop.

`for` och `while` har olika syfte men gör egentligen samma sak. Man kan alltid byta ut en `for` mot en `while` och vise versa:

```
while ( villkor ) sats;  =  for ( ; villkor; ) sats;

for ( initiering; villkor; förändring ) sats;  =
initiering; while ( villkor ) { sats; förändring; }
```

`do sats; while ( villkor );`

Både `for`-loopen och `while`-loopen kan passeras utan att koden i kroppen någonsin exekveras. Om man vill garantera att något händer minst en gång finns `do` – `while`. `do` – `while` kör loop-kroppen en gång först innan villkoret kontrolleras. För övrigt påminner den mycket om en vanlig `while`.

`break` och `continue`

De två nyckelorden `break` och `continue` kan användas tillsammans med alla tre looparna. `break` avbryter loopen helt och hoppar vidare till koden som ligger direkt efter loop-kroppen. Loopen avbryts oavsett om villkoret fortfarande är uppfyllt eller ej.

`continue` bryter loop-kroppen i förtid och hoppar till villkoret. Om villkoret är uppfyllt fortsätter loopen att snurra och ett nytt varv påbörjas vid loop-kroppens början.

1.6 Inmatning och utmatning — `stdin`, `stdout`

Hittills har vi endast pratat om inbyggda operatorer och kommandon i C. Vi har inte gått igenom alla de inbyggda konstruktionerna än, men det viktigaste är klart. Som tidigare nämnts saknar C inbyggda funktioner för I/O, så inmatning och utskrift måste ske genom funktionsanrop. Det har därför blivit dags att börja titta lite på vad som finns i standardbiblioteket.

ANSI C har ett väldefinierat standardbibliotek där det bland annat finns gott om funktioner för just I/O. Standardbiblioteket har 15 avdelningar för olika typer av funktioner. Några av de vanligaste är

`math` Matematiska funktioner

`stdio` In- och utmatning, filhantering (upptar en tredjedel av biblioteket)

`stdlib` Konvertering, minnesallokering

`string` Sträng- och minneshantering

All inmatning och utmatning i C sker genom strömmar. Man kan se en ström som ett rör där någon stoppar in tecken i den ena änden och någon annan plockar ut tecknen i den andra änden. De vanligaste strömmarna kallas `stdin` och `stdout`. (Std som förekommer i flera av namnen här är en förkortning av "standard". Namnen utläses alltså standard i/o, standard in och standard out.)

stdin

`stdin` är den ström där ett C-program oftast hämtar information från användaren. Normalt är `stdin` knuten till tangentbordet. Det är alltså tangentbordet som stoppar tecken i den ena änden av `stdin`-röret och C-programmet som plockar ut dem i den andra. För att göra detta använder man funktioner definierade i `stdio`.

`getchar()` – läs ett tecken från `stdin`. Funktionen är ganska lättanvänd. Man anropar den och den returnerar ett tecken från `stdin`. Om det inte finns något tecken att hämta i `stdin` när `getchar()` anropas kommer programmet att stanna och vänta på att ett tecken dyker upp.

`scanf(` format-sträng, ... `)` – läs in formaterad text. `scanf` är lite knepigare att använda. Man skickar med en format-sträng som styr vilken indata man vill ta emot. Formatsträngen innehåller styrkoder för till exempel heltal, tecken eller ord och för varje styrkod skickar man med en variabel dit `scanf` skriver motsvarande värde. Det är nog enklast att visa detta med ett exempel.

```
int x;
scanf("%d", &x);
```

I exemplet ovan vill vi läsa in ett tal från `stdin` och lägga detta i `x`. `%d` är alltså styrkoden för heltal. Som vi kan se så är det något märkligt framför `x` i anropet till `scanf`, ett `&`. Detta är operatorn för att hämta adressen till en variabel. Eftersom C är call-by-value så kan vi inte skicka in `x` rakt av. Det som då skulle skickas till `scanf` är ju innehållet i `x`, det vill säga något tal. Men `scanf` vill skriva ett nytt värde i variabeln `x` och för att kunna göra det krävs att `scanf` vet var i minnet den ska skriva. Därför måste vi skicka minnesadressen till `x`, inte `x`. Format-strängen kan innehålla flera styrkoder och annan text som matchas mot den inkommande strömmen av tecken.

```
stdin: Ett tal: 42 Tecken: B Ord: ost

int x;
char y;
char z[20];
scanf("Ett tal: %d Tecken: %c Ord: %s", &x, &y, &z);

x = 42    y = 'B'    z = "ost"
```

Här används `%d` för heltal, `%c` för tecken och `%s` för ett helt ord. `scanf` avslutar inläsningen av ett ord vid ett blankt tecken – mellanslag, tab, radbrytning med mera Detta gör att `scanf` endast kan läsa in enstaka ord, vill man läsa in hela meningar blir det klurigare. I exemplet ser vi hur tecken- och sträng-konstanter skrivs i C. Tecken skrivs med `'` runt om (`'B'`) och strängar med citattecken (`"hej"`).

Här ser vi även en ny typ av variabeldeklaration, `char z[20];`. Detta är en så kallad *array* och i det här fallet använder vi den för att spara ett ord i. Vi återkommer till arrayer i en senare föreläsning och just nu räcker det att veta att `z` är en variabel med plats för ord som är upp till 19 tecken långa[3].

[3]Det får bara plats 19 tecken trots att arrayen har 20 platser. Detta beror på att det krävs ett specialtecken i

`stdout`

Den andra strömmen som nämndes tidigare var `stdout`. Denna används för att skriva ut saker från programmet. Normalt hamnar dessa utskrifter i det terminalfönster som man startade programmet ifrån. Den enklaste funktionen för utmatning är `printf`.

```
printf ( sträng, ... );
```

Även `printf` tar en formatsträng och ett antal variabler, dessa kombineras ihop till den sträng som sedan skickas till `stdout`. Om strängen inte innehåller några styrkoder skrivs den ut utan förändring. `printf("Hej från C!\n");` kommer att skriva ut texten "Hej från C!" i terminalfönstret. Observera att `\n` inte är en styrkod utan tecknet för radbrytning.

```
printf("Ett tal: %d Tecken: %c Ord: %s",42,'B',"ost");
```

Raden ovan kommer att skriva ut texten "Ett tal: 42 Tecken: B Ord: ost" Strängen som skrivs ut med `%s` kan i det här fallet innehålla blanka tecken. Hela strängen kommer att skrivas ut.

`stdin` och `stdout` är som sagt de två vanligaste strömmarna, men man kan även skapa egna strömmar till filhantering och andra I/O-kanaler. Filhantering återkommer vi till i en senare föreläsning.

Det finns många fler funktioner i `stdio`. På kurshemsidan finns en länk till en beskrivning av C:s standardbibliotek, där kan ni läsa mer om funktionerna för inmatning och utskrift. Där finns även alla styrkoder förtecknade.

Nu när vi vet hur man får in data från användaren är det dags att titta på cirkel-arean igen. Det var radien vi ville läsa in och arean vi ville skriva ut. Båda värdena är flyttal och styrkoden för flyttal är `%f`.

```
scanf("%f",&radie);

printf("Area: %f\n",area);
```

1.7 Här börjar det — `main`

För att C-kompilatorn ska veta var i din källkod du vill att ditt program ska börja så finns det en väldefinierad startpunkt - `main`-funktionen. Alla C-program måste innehålla en `main`-funktion och det är där exekveringen av programmet startar.

```
main()
{
   printf("Hej från C!\n");
}
```

Det här programmet är i sin enkelhet korrekt C-kod. Det är dock inte helt komplett och för att godkännas i denna kurs krävs det lite fler detaljer.

```
#include<stdio.h>

int main()
{
   printf("Hej från C!\n");
   return 0;
}
```

slutet av ordet för att markera att det är slut.

Först måste vi tala om var `printf` finns deklarerad. Detta gör vi genom att inkludera den deklarationsfil som hör till `stdio`. Exakt vad detta innebär återkommer vi till senare i kursen, just nu räcker det att veta att det är så det ser ut. Vi måste även tala om vad funktionen har för returtyp. I `main`-fallet är denna alltid `int`. Och när vi nu säger att vi ska returnera en `int` så måste vi också göra det, därav `return 0;` Returtyp och returvärden återkommer vi till nästa föreläsning, så jag säger inte mer om det här.

Då vet vi slutligen allt som behövs för att färdigställa vårt program för att beräkna arean av en cirkel.

```
#include<stdio.h>

int main()
{
   float radie, area;
   const float PI = 3.1415;

   printf("Ange en radie: ");
   scanf("%f",&radie);
   area = PI * radie * radie;
   printf("Area: %f\n",area);

   return 0;
}
```

1.8 En kompilator — gcc

Som tidigare nämnts så måste man kompilera C-koden innan den går att köra i datorn. I denna kurs förutsätts att ni jobbar med kompilatorn `gcc` på universitetets datorer. För att ni ska slippa onödiga buggar och för att tvinga på er "god programmeringsstil" så kommer jag kräva att er källkod passerar `gcc` med några extra restriktioner.

Min rekommendation är att ni skriver källkoden i en texteditor där ni har full kontroll över vad som händer. Jag rekommenderar inte grafiska utvecklingsmiljöer där mystiska saker händer under ytan, i alla fall inte för en introducerande kurs i C. Tanken är att ni ska lära er vad som händer och kunna utföra alla stegen själva – det gör ni inte om ni har ett program som tänker åt er.

`emacs` har allt (och bra mycket mer än) man behöver för att ge en behaglig utvecklingsmiljö. Öppna din favoriteditor och skriv in programmet ovan (det som beräknade cirkelns area), spara textfilen som `cirkel.c`.

För att kompilera med `gcc` befinner man sig lämpligen i ett terminalfönster. Det går även att kompilera inne i `emacs`, men det är inget jag kommer att beskriva här. Öppna ett terminalfönster och försäkra er om att ni befinner er i samma katalog som källkodsfilerna ligger i. Unix-kommandon som `ls` och `pwd` är användbara för detta. För att kompilera programmet med `gcc` skriver man helt enkelt

```
gcc cirkel.c
```

Nu får vi en körbar fil som heter `a.out`. Provkör den med: `./a.out`
För att ge utfilen ett lite trevligare namn använder vi flaggan `-o` med `gcc`.

```
gcc cirkel.c -o cirkel
```

Nu får vi en körbar fil med namnet `cirkel`, mycket bättre. För att uppfylla de extra restriktioner som krävs för att er kod ska få godkänt ber vi `gcc` att klaga på allt som går att komma på. Detta gör vi genom att skicka med ett antal flaggor.

```
gcc -ansi -Wall -Wextra -Werror -pedantic cirkel.c -o cirkel
```

I äldre versioner av `gcc` heter flaggan `-Wextra` endast `-W`. Om ni inte arbetar på universitetets datorer så är det säkrast att kolla upp vad flaggan heter i den version av `gcc` som ni använder.

Föreläsning 2

Lättläst kod innehåller färre buggar

Man brukar säga att det i genomsnitt finns en bugg per hundra rader C-kod. Detta gäller då produktionskod skriven av erfarna programmerare, det vill säga kod som finns där ute i verkligheten och styr stora delar av samhället. För att styra upp denna skrämmande verklighet är det viktigt att ni som kommer att skriva framtidens styrprogram för flygplan och kärnkraftverk vet hur man kan minimiera risken för onödiga buggar.

Denna föreläsning kommer främst att ta upp två olika koncept som C erbjuder för att bygga källkod som är lättläst; Egna datatyper och funktionsanrop.

C har ett antal inbyggda datatyper som vi redan sett en hel del av. `int`, `float` och `char` är några av dem. Dessa typer hjälper kompilatorn att skilja mellan heltal, flyttal och tecken. Kompilatorn kan dock inte veta om en variabel av typen `int` är tänkt att innehålla ett årtal eller antalet anställda vid något företag. Det samma gäller i viss utsträckning även den programmerare som skriver källkoden. Ibland tänker man fel och ibland glömmer man bort att man ändrat innebörden av en variabel.

Det första man bör tänka på när det gäller att öka läsbarheten i ett program och för att minska risken för att man använder en variabel till fel sak är naturligtvis att ge sina variabler vettiga namn. Det är lättare att komma ihåg att en variabel ska innehålla en temperatur om den heter `temperatur` än om den heter `t` (eller ännu värre `temp` som så gott som alltid kommer att utläsas temporär).

Eftersom människan har en förmåga att automatiskt fylla i luckor i en text och läsa det hjärnan vill att det ska stå snarare än det det faktiskt står, så är människan av sin natur tyvärr ganska dålig på att hitta fel i källkod. Lyckligtvis finns det flera hjälpmedel för att låta datorn hitta fel i källkoden istället. `lint` och `splint` är två exempel på program som är designade för att hitta fel i C-källkod. Prova dem gärna på era egna program för att se vilka typer av fel de hittar. Även `gcc` är bra på att hitta fel, speciellt när vi skickar med alla de flaggor som jag nämnde i den förra föreläsningen. Stavfel och bortglömda variabler hittas omedelbart när man kompilerar programmet.

2.1 Egna typer — `typedef`, `enum`

Program som `lint` och `splint` är tyvärr inte hjälpt av att variabelnamn talar om vad en variabel ska innehålla. De kan bara se typen på variabeln. För att utnyttja dessa program och få dem att klaga om man skickar fel typ av data till funktioner måste man skapa nya datatyper som beskriver vilken sorts data man hanterar.

Egna datatyper underlättar också när man vill skriva representationsoberoende kod (och det vill man alltid). Mer om det senare.

typedef

Med nyckelordet `typedef` kan vi ge ett nytt namn åt en existerande datatyp. På det sättet kan vi skapa datatyper som talar om vilken typ av värden de innehåller.

```
typedef int tidpunkt_t;

tidpunkt_t start;
tidpunkt_t slut;
```

Här har vi skapat en ny typ, `tidpunkt_t`. `gcc` kommer att betrakta denna typ som om det vore en `int` och alla operatorer som kan användas på en `int` kan även användas på en variabel av typen `tidpunkt_t`. Vi kan skapa variabler av den nya typen på samma sätt som med de inbyggda typerna. I exemplet ovan skapar vi de två variablerna `start` och `slut`, om dessa vore av typen `int` skulle det inte vara alls så självklart vad de skulle innehålla för något. Det är en god vana att märka egna typnamn på något sätt så att man lätt kan skilja dem från variabelnamn. Jag brukar sätta på ett `_t` efter namnet.

Det nya typnamnet är en bättre beskrivning av vad datatypen egentligen är och hur den ska användas i programmet. Att använda `int` för alla heltal i ett program fungerar ju naturligtvis, men genom att skapa nya typer som vi använder istället så kan vi skapa program som blir mycket lättare att läsa och underhålla.

`typedef` används ofta i representationsoberoende kod. Det vill säga kod där man inte vill att representationen av data ska vara synlig överallt i koden. Varför man vill ha det på det sättet är lättast att förklara med ett litet exempel.

Säg att vi har ett stort program (till exempel ett operativsystem) som skrevs för länge sedan. Där finns en variabel för lagra en klocka. Klockan räknas i antalet sekunder från ett givet startdatum, 1 jan 1978. Eftersom klockan bara är ett tal så valde man att lagra den i en variabel av typen `int`. Av olika anledningar överlever koden längre än någon anat (68 år) och antalet sekunder sedan 1978 börjar närma sig gränsen för vad en `int` kan lagra. Av ekonomiska skäl kan man inte kasta den gamla koden utan man blir tvungen att anlita en konsult för att byta ut representationen av klockan mot något som håller några år till.

Konsulten tycker att det enklaste blir att byta ut typen på klockan till `unsigned int` och på så sätt ge systemet ytterligare 68 år. För att programmet ska fortsätta fungera krävs dock att man hittar samtliga ställen där klockan hanteras (funktionsargument, lokala variabler, medlemmar i större datatyper och så vidare) i alla applikationer som använder klockan för att byta ut `int` mot `unsigned int`.

`grep` i all ära, men att leta efter alla förekomster av `int` i ett stort program och avgöra vilka som har med klockan att göra och vilka som är andra heltal är inte ett lätt jobb och risken för fel är stor.

Hur skulle man då ha gjort? Genom att från början definiera en egen typ för klockan skulle det vara ett lätt jobb att byta ut definitionen från `typedef int clock_t;` till `typedef unsigned int clock_t;`. Att hitta alla ställen där `clock_t` används för att se att allt fungerar skulle också vara ett lätt jobb.

enum

Förra föreläsningen nämnde jag `const` som talar om att vi som programmerare inte avser att ändra på en variabel. Eftersom man inte kan vara helt säker på att kompilatorn anmärker på försök att ändra på dessa "konstanter" blir hela poängen med att konstantdeklarera dem lite tveksam. Lyckligtvis finns det andra sätt att deklarera "riktiga" konstanter. Ett av dem är att använda `enum`.

En `enum` är en uppräkningstyp (enumeration) som definierar en lista av heltalskonstanter. Elementen har en inbördes ordning. Det första elementet får värdet 0 om inget annat anges. Värdet ökas sedan med ett för varje element. Uppräkningstypen blir en ny datatyp som vi kan deklarera variabler av. Nyckelordet `enum` blir en del av namnet på datatypen.

```
enum bool { false, true };

enum bool x;
x = true;
```

Det finns som jag tidigare nämnt ingen datatyp för sanningsvärden i C. Här har vi därför deklarerat vår egen typ. `false` kommer i detta exempel att ha värdet 0 och `true` får värdet 1. Varför har jag nu döpt denna datatyp till `bool` och inte `bool_t`? Ja, det är av den enkla anledningen att det egentligen inte är jag som har döpt den. För att öka läsbarheten i sin källkod vill man gärna ha en datatyp för just sanningsvärden. Därför har man lagt till ett externt bibliotek som heter `stdbool`. Detta deklarerar namnen `bool`, `true` och `false`.

`stdbool` är inte en del av ANSI-C utan tillkom först senare i ISO C. Även om vi i denna kurs främst håller oss till ANSI-C så kommer vi i inlämningsuppgiften att använda `stdbool`. Skulle vi köra strikt ANSI-C så skulle vi få deklarera `bool` som vi gjort ovan, men detta ger problem med program som `lint`. `lint` med flera kräver att man använder en datatyp för sanningsvärden, men den får inte vara deklarerad i programmet eftersom denna datatyp redan är deklarerad av `lint`. Det enklaste sättet att kringgå detta är därför att använda `stdbool`.

Genom att initiera det första elementet i en uppräkningstyp till något annat än noll kan vi flytta hela talföljden. Elementen kommer fortfarande att ha stigande heltalsvärden.

```
enum value_t
{
   ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
   NINE, TEN, JACK, QUEEN, KING
};
```

Typen `enum value_t` är hämtad från inlämningsuppgiften. Här har `ACE` värdet 1, `TWO` värdet 2, `THREE` har värdet 3 och så vidare.

Även om uppräkningstypen har en given ordning är det vanligt att man använder den även för att definiera medlemmar i ett set, det vill säga saker som hör ihop men som egentligen inte har någon inbördes ordning, till exempel färgerna i en kortlek, eller olika filmkategorier.

```
enum suit_t
{
   HEARTS   = 'H',
   CLUBS    = 'C',
   SPADES   = 'S',
   DIAMONDS = 'D'
};

enum category_t
{
   HORROR, SCIFI, DRAMA, COMEDY, THRILLER, WESTERN
};
```

I dessa datatyper har den numeriska ordningen ingen betydelse. I `suit_t` har konstanterna inte ens värden som följer direkt efter varandra och det är egentligen inte meningsfullt att jämföra dessa värden för annat än likhet. Samlingstyper av det här slaget är främst viktiga för att få läsbar kod. Med namngivna konstanter som dessutom har en egen typ blir det betydligt enklare att förstå ett program än om koden skulle vara kryddad med “magiska konstanter”.

Som vi ser så blir namnen på uppräkningstyperna ganska långa och eftersom de består av två ord (`enum suit_t`) så kan en del editorer bli förvirrade när till exempel färgsättning av koden ska göras. Här kan vi nu få ännu mer nytta av `typedef`. `typedef` ger ett nytt namn för en datatyp som redan finns. Detta kan vi utnyttja för att ge våra uppräkningstyper ett lättare namn som passar bättre in i kodstrukturen i C.

```
typedef enum bool bool;
```

Eftersom C vet det riktiga namnet på typen så går det bra att använda `typedef` även för typer som består av flera ord. Faktum är att `typedef` kan användas även med hela typdeklarationer.

```
typedef enum { false, true } bool;
```

Den ursprungliga uppräkningstypen blir i det här fallet anonym eftersom inget namn ges efter nyckelordet `enum`. Typen får istället ett namn av `typedef` och vi kommer att kunna använda namnet `bool` som en ny datatyp.

```
bool a;
a = true;
if (a == true)
   printf("Så är det!");
```

2.2 Strukturer — `struct`

För att skapa mer komplexa datatyper behöver man kunna slå samman flera olika värden till ett objekt. I C gör man detta med hjälp av strukturer. Med nyckelordet `struct` knyter man ihop de olika variablerna som ska bygga upp objektet.

```
struct person_t
{
   int  age;
   int  shoesize;
   bool happy;
};
```

Som vi ser kan alla typer av data finnas i strukturen, även sådana datatyper vi skapat själva. I exemplet nedan använder vi även `typedef` för att skapa ett lite enklare namn åt strukturen.

```
typedef struct
{
   char            title[40];
   int             year;
   int             length;
   category_t      category;
   struct person_t director;
} film_t;
```

Vi kan nu skapa variabler av typen `film_t` och med hjälp av en punkt (`.`) så kommer vi åt fälten i strukturen.

```
film_t alien;

strcpy(alien.title,"Alien");
alien.year = 1979;
alien.length = 117;
alien.category = HORROR;
alien.director.happy = TRUE;

printf("%s hadde premiär %d och är %d timmar och %d minuter lång.\n",
       alien.title, alien.year, alien.length / 60, alien.length % 60);
```

Här ser vi ett exempel på hur man kan använda och komma åt fälten i en struktur. Bortsett från punkten är det ingen skillnad på att använda ett fält i en struktur och en vanlig variabel. Notera även att vi kommer åt fält i `alien.director` genom att lägga på ytterligare en punkt.

Filmens titel sparas i en sträng. `strcpy` är ett kommando som kopierar en sträng till en annan. Den finns i strängbiblioteket[1]. Vi återkommer till strängar nästa föreläsning.

2.3 Funktioner

En mycket viktig egenskap för ett programmeringsspråk är förmågan att bryta upp stora stycken kod i mindre delar. Detta är viktigt ur två avseenden. Dels (än en gång) för att öka läsbarheten. Det är mycket lättare att få överblick över ett litet stycke kod än det är att få överblick över ett stort stycke kod. Den andra anledningen är att man genom att bryta ut kod som används på flera ställen i ett program kan undvika att skriva samma kod flera gånger. Kodduplicering är i allmänhet dåligt och leder ofta till fel i program när man uppdaterar koden på ett ställe men glömmer ett annat.

Det normala sättet i så gott som alla programmeringsspråk att bryta ut kod är genom att skriva funktioner. Man kan se en funktion som ett litet miniprogram som ligger inuti det stora programmet. Vi har redan sett en funktion (`main`) och anropat ett par andra (`scanf` och `printf`). Nu är det dags att på allvar titta på funktionsanrop i C.

Alla funktioner i C måste deklareras med en *returtyp*, det vill säga vilken sorts data som funktionen ska ge tillbaka till den som anropar.

```
Returtyp Namn ( Argument )
{
   Lokala variabler
   Funktionskropp
   Retursats
}
```

```
int foo(float tal)
{
   int heltal;
   heltal = (int)tal;
   return heltal;
}
```

I det här fallet har vi sagt att funktionen `foo` ska returnera ett heltal (`int`). Även argumenten till en funktion måste typdeklareras. Funktionen `foo` tar ett argument som ska vara av typen `float`. Ett anrop till `foo` kan se ut så här: `int x = foo(3.14);`

När man deklarerar nya funktioner bör man alltid placera dem ovanför den anropande funktionen i källkoden. Detta för att C-kompilatorn alltid läser källkoden från första till sista raden och endast de funktioner och variabler som deklarerats högre upp i filen får användas. Av denna anledning hamnar normalt `main`-funktionen sist i en källkodsfil.

Argument

För att få med sig värden in i en funktion måste man skicka med dem som så kallade *argument*. Vi har redan sett ett antal argument som vi skickat till bland annat `printf` och funktionen `foo` i exemplet ovan. `foo` tar ett argument av typen `float`. För att anropa `foo` måste man alltså skicka med ett flyttal.

C är vad man brukar kalla *call-by-value*. Det innebär att alla argument kommer att räknas ut innan själva funktionsanropet sker. Nedan anropar vi funktionen `foo` och anger ett uttryck (`x + 4.5`) som argument. `foo` kommer dock aldrig att se detta uttryck utan argumentet `tal` kommer att få värdet 42.

[1] Strängbiblioteket kommer man åt via `string.h`.

```
void foo(float tal)
{
   Här har tal värdet 42.
}

float x = 37.5
foo(x + 4.5);
```

Returvärden

Värdet som en funktion skickar tillbaka till den som anropar kallas *returvärde*. Funktionen deklareras med en returtyp som talar om vilken sorts data funktionen ska skicka tillbaka.

```
int bar()
{
   return 4711;
}

int x = bar();     x får värdet 4711
```

Returtypen syns på tre ställen och det är viktigt att alla tre platser hanterar samma typ. Dels är det själva funktionsdeklarationen som har en returtyp. Denna returtyp deklareras innan funktionsnamnet. Funktionen `bar` ovan har returtypen `int`. Det andra stället som ska matcha returtypen är där funktionen tar slut. Där finns kommandot `return` som kommer att utföra själva tillbakaskickandet av returvärdet. Om funktionen säger att den ska skicka tillbaka en `int` så får man se till att `return` faktiskt gör det. Det tredje stället där det måste stämma är i den anropande koden. Om en funktion returnerar ett värde måste man ta hand om det och då måste man förstås behandla det som den typ som funktionen säger att den returnerar. `gcc` kommer att klaga om man har fel typ på något ställe både när det gäller returvärden och argument.

I en del programmeringsspråk skiljer man på funktioner och *procedurer*. Funktioner har alltid ett returvärde medan procedurer aldrig returnerar något. I C finns endast funktioner, men man kan ge en funktion returtypen `void`. Detta betyder att funktionen inte returnerar något värde. (void = tomrum)

`main`

`main` är som sagt en funktion precis som alla andra, den har en returtyp (`int`) och kan ta argument. Argumenten till `main` ska se ut på ett speciellt sätt, vi återkommer till dessa lite senare.

`main`:s returvärde är även det lite speciellt. Tanken är ju att ett Unix-system ska ta emot returvärdet från programmet. Detta gör att returvärdet måste följa de konventioner som finns i Unix. Ett programs returvärde i Unix representerar en statusflagga som talar om ifall programmet lyckades med sin uppgift eller ej. Returtypen är alltid `int`. Om ett program returnerar `0` betyder det att allt gick bra, ett returvärde skilt från `0` betyder att något gick fel. I ANSI-C representeras dessa värden med två konstanter: `EXIT_SUCESS` och `EXIT_FAILURE`.

2.4 Omgivningar — Globala och lokala variabler

En variabel finns (endast) åtkomlig inom den *omgivning* den är deklarerad i. En omgivning kan till exempel vara en funktion, en loop-kropp eller ett helt program. En lokal omgivning i C markeras alltid med { }. Till exempel såg vi förra föreläsningen att man kan sätta { } vid loopar och villkorliga satser,

```
void foo()
{
   int  x = 45 + 7;
   char ch;

   Här kommer koden...
}
```

Variabeldeklarationer måste alltid stå först i en omgivning. `gcc` kommer att klaga om ni försöker blanda deklarationer och kod. Det är tillåtet att skriva kod för att initiera variabler bland deklarationerna, men detta får bara vara en sats och den måste komma direkt efter variabeldeklarationen. En funktionskropp skrivs alltid med { } och har alltid en egen lokal omgivning.

```
int plus(int x, int y)
{
   int summa = x + y;
   return summa;
}

int dela(int x, int y)
{
   if (y != 0)
   {
      int kvot = x / y;
      return kvot;
   }
   else
      return MAX_INT;
}
```

I exemplet ovan är variabeln `summa` lokal i funktionen `plus`. Det går inte att komma åt variabeln i funktionen `dela`. Variabeln `kvot` är lokal i `if`-satsen. Det går alltså inte att komma åt `kvot` i resten av funktionen `dela`, den finns endast tillgänglig inom den omgivning som deklarerats efter `if`.

Man ska alltid sträva efter att ha så kort livslängd som möjligt på sina variabler, det underlättar för `gcc`:s optimering och framför allt är det lättare att läsa källkoden om variabler deklareras nära den plats i koden där de används. Man kan deklarera en lokal omgivning var som helst i koden i ett C-program. Exemplet nedan visar ett praktiskt sätt att deklarera en lokal indexvariabel till en `for`-loop.

```
int main()
{
   Lite godtycklig kod...

   {
      int i;
      for (i = 0; i < 10; i++)
         sats;
   }

   Lite mer godtycklig kod...
}
```

Alla variabler vi sett så här långt har varit lokala. De har haft en begränsad omgivning som är tydligt markerad i koden. Även funktionsargument räknas till de lokala variablerna i en funktion.

Man kan även deklarera variabler utanför alla omgivningar. Dessa betraktas då som levande i hela programmet och kallas normalt för globala variabler. Globala variabler deklareras när programmet startar och minns sina värden genom hela programkörningen till skillnad från lokala som försvinner när man lämnar den omgivning de är deklarerade i.

Globala variabler är lite farliga att använda i större program och man bör tänka efter noga innan man skapar dessa. Anledningen är att globala variabler kan användas även utanför den källkodsfil man deklarerar dem i. Därmed öppnas möjligheterna för namnkonflikter med andra filer i applikationen, oönskad användning av variabeln och eventuell felaktig uppdatering av variabeln i kod som inte borde känna till, eller åtminstone inte ändra variabeln.

static

Med nyckelordet `static` kan man begränsa problemen med globala variabler till den aktuella källkodsfilen. Betrakta kodexemplet nedan.

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

int a = 0;
static int b = 0;

void foo(void)
{
   int c = 0;
   static int d = 0;

   printf("global: %d  static global: %d  lokal: %d  static lokal: %d\n",
          a, b, c, d);

   a++; b++; c++; d++;
}

int main(void)
{
   int i;

   for (i = 0; i < 10; i++)
      foo();

   return EXIT_SUCCESS;
}
```

När en global variabel deklareras statisk begränsas dess åtkomst till den aktuella källkodsfilen. Detta är alltså ett sätt att skydda variabler som måste vara globala i en fil från att kod i andra filer krockar med dem[2]. I kodexemplet ovan har vi två globala variabler, `a` och `b`. `a` är tillgänglig utifrån och dess värde kan alltså ändras utanför vår kontroll. `b` är statisk och därmed endast tillgänglig i vår egen kod.

`static` kan även användas för lokala variabler. Betydelsen är dock en annan. I exemplet finns två lokala variabler i funktionen `foo`. Den ena av dessa är statisk (`d`). Båda kommer initialt att ha värdet 0. `foo` anropas tio gånger från `main` och vid varje anrop ökas värdet på alla variabler med ett. Skillnaden mellan `c` och `d` är att `d` kommer att minnas sitt värde mellan funktionsanropen. Den statiska variabeln kommer alltså endast att deklareras och initieras en gång[3]. Därefter kommer raden att ignoreras - trots att där finns en tilldelning som initierar `d` till 0. Den icke-statiska variabeln (`c`) kommer att deklareras och initieras varje gång `foo` anropas.

Det finns en viktig skillnad mellan en statisk lokal variabel och en global. Den lokala variabeln är endast tillgänglig i den omgivning den deklareras i. Globala variabler är tillgängliga i hela programmet. Om man är ute efter en variabel som kommer ihåg sitt värde mellan funktionsanropen är alltså statiska lokala variabler helt klart att föredra.

Att initiera globala och statiska variabler till noll är egentligen onödigt eftersom alla dessa variabler initieras till noll automatiskt, men bör ändå göras för läsbarhetens skull.

[2]Kan jämföras med privata instansvariabler i Java.
[3]Lokala statiska variabler allokeras när programmet startas.

Föreläsning 3

Pekare är bara variabler

I den förra föreläsningen såg vi hur man kan skicka med data till funktioner via argument. Tidigare har jag nämnt att argumenten som skickas till en funktion kommer att kopieras i datorns minne. Vi ska titta på hur detta går till för att ge lite bättre förståelse för hur saker sker under ytan.

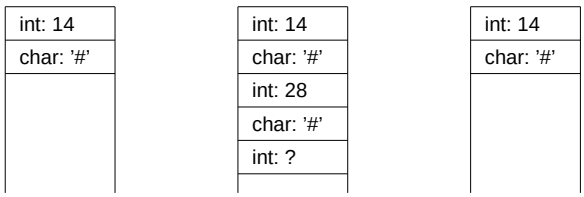
När ett program kör underhåller datorn något som kallas för en *stack*. På denna stack lägger programmet upp *stack frames*. Dessa stack frames representerar den lokala omgivningen i en funktion. Där allokeras bland annat alla lokala variabler och funktionsargument. Varje gång en funktion anropas kommer en ny stack frame att läggas upp på stacken. Storleken på denna stack frame beror naturligtvis på antalet argument och storleken på dessa.

```
 1  #include<stdlib.h>
 2  #include<stdio.h>
 3
 4  void skrivut(int antal, char ch)
 5  {
 6     int i;
 7
 8     for (i = 0; i < antal; i++)
 9        printf("%c",ch);
10
11     printf("\n");
12  }
13
14  int main()
15  {
16     int flera = 14;
17     char tecken = '#';
18
19     skrivut(flera * 2, tecken);
20
21     return EXIT_SUCCESS;
22  }
```

Figur 3.1 visar hur stacken ser ut vid ett par olika tillfällen i programkörningen[1]. I figur 3.1(a) har programmet precis startat och befinner sig på rad 18. De två lokala variablerna i `main` har skrivits till stacken. Det är nu dags att anropa funktionen `skrivut`. som jag sagt tidigare är C "call-by-value", det innebär att uttrycket `flera * 2` kommer att beräknas innan anropet sker och till stacken skriver vi endast resultatet.

Nu tittar vi på figur 3.1(b). Här är vi på rad 7 i programmet och argumenten och den lokala

[1] Detta är en mycket förenklad bild av en stack frame.



(a) Rad 18 (b) Rad 7 (c) Rad 20

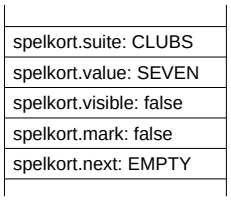
Figur 3.1: Stack frames

variabeln har kopierats till stacken. Som vi kan se har `i` inte fått något värde ännu. Jag markerar det med ett frågetecken eftersom vi inte vet vad det är för värde i en oinitierad variabel. Notera att tecknet `'#'` nu finns på två ställen på stacken. Alla argument som skickas till en funktion kommer att följa med in i den nya stack frame:en – även om de redan ligger på stacken när de skickas.

Funktionen får jobba klart och vi kommer så småningom tillbaka till `main`. Argumenten och de lokala variablerna i en funktion lever som sagt bara i den omgivning där de deklarerats och när vi lämnar omgivningen försvinner även den stack frame som hör till funktionen. På rad 20 har vi återkommit från `skrivut` och kvar på stacken finns bara den stack frame som hör till `main` (se figur 3.1(c)).

```
void getvisible(card_t spelkort)
{
   return spelkort.visible;
}
```

Vi tar en titt på en av funktionerna i Klondike, `getvisible`. Detta är en relativt enkel funktion som endast ska returnera en flagga (ett sanningsvärde) ifrån ett spelkort. Figur 3.2 visar hur stack frame:en för `getvisible` ser ut vid ett anrop där vi vill veta om klöver sju är synligt eller ej. Jag har tagit med variabelnamnen för att göra det tydligare, dessa finns inte med i en riktig stack frame. Som vi ser har hela spelkortet kopierats till stacken. Om vi tittar på hur mycket av det data vi skickar till funktionen som egentligen används så ser vi att det är en ganska liten del. Endast ett av de fem värdena som skickas in är av intresse.



Figur 3.2: Stack frame vid anrop till `getvisible`

Att kopiera allt data i onödan kostar naturligtvis en del i prestanda, framför allt i den här typen av funktion som kommer att anropas ofta i programmet och där vi inte utför något direkt arbete i själva funktionen. För att dra ner kostnaden för dessa funktionsanrop gäller det att minska mängden data vi skickar i argumenten. Vi kan inte plocka ut det intressanta fältet i spelkortet och skicka bara det – det skulle göra hela funktionen meningslös. Vi behöver ett sätt att referera till spelkortet utan att skicka med hela kortet till funktionen. Vi behöver pekare.

3.1 Adresser och pekare

Låt oss börja med att titta på hur minnet i en dator ser ut. Det är uppbyggt av en sekvens av bytes där varje byte har en egen adress. Adresserna börjar på noll och räknas upp med ett för varje byte. Eftersom datorn är uppbyggd för att hantera maskinord snarare än bytes så kan man inte lägga ut data hur som helst i minnet. variabler av typen `char` är bara en byte stora, så dessa kan man på de flesta maskiner lägga på vilken adress som helst, men vill man lägga ut till exempel en `int` i minnet så måste denna hamna på en adress som är jämnt delbar med fyra[2]. För det mesta sköter kompilatorn detta själv så det är bara i extremfall man behöver fundera på detta, men det kan vara bra att känna till om man till exempel skulle råka ut för ett "Bus Error", som betyder just att man försökt läsa eller skriva ett helt ord på en adress som inte är jämnt delbar med fyra[3].

Alla variabler som vi deklarerar i våra program kommer att hamna någonstans i datorns minne. Detta innebär att det kommer att finnas en minnesadress som pekar ut just den minnescell där variabeln ligger. Vi kan komma åt denna adress i C genom att sätta ett `&` framför variabelnamnet. När det gäller större datastrukturer som ockuperar mer än en minnescell kommer `&` att ge oss den adress där strukturen startar.

När vi nu har en operator för att plocka ut adresser kan det vara bra att ha någon variabel att lägga adressen i. Det här med att allt i C bara är siffror är förvisso fortfarande sant, även en adress är ju i grunden ett tal, men när det gäller adresser så är en C-kompilator mycket petig. Det räcker inte att tala om att det är en adress vi skickar, vi måste även tala om vilken sorts adress det är, det vill säga vilken typ av data som ligger på adressen.

För varje datatyp som finns i C, vare sig det är en inbyggd eller en egendefinierad, finns det en motsvarande *pekare*. Pekaren är en variabel som innehåller en adress. Det viktigaste att inse när det gäller pekarvariabler är att de är helt vanliga variabler. De deklareras och används på samma sätt som vilken annan typ av variabel som helst. Man säger att pekaren pekar på en adress. Detta betyder egentligen att pekarvariabeln innehåller en adress. Det finns ingen magi inblandad.

För att visa att en variabel är en pekare lägger man på en stjärna (`*`) efter typnamnet. `int*` utläses pekare till `int` och variabler av den här typen kan innehålla adresser till minnesceller där det ligger heltal av typen `int`.

```
int tal = 42;
int* adress = &tal;
```

Stjärnan är en del av typnamnet, men rent syntaktiskt är de inte bundna till varandra. Det är till exempel tillåtet att sätta mellanslag runt stjärnan. Stjärnan hör dock till typen, inte variabelnamnet, så därför kan det verka logiskt att sätta stjärnan tillsammans med typen. När vi använder variabeln `adress` ovan så skriver vi just `adress` – utan stjärna. C är dock lite förvirrande på den här punkten. När man deklarerar pekare kommer nämligen stjärnan att bindas till variabelnamnet, inte typen.

```
int* x, y;
```

Deklarationen ovan ser intuitivt ut som att den skulle deklarera två pekare, `x` och `y`. Detta är dock inte vad som sker. `x` kommer att deklareras som en pekare, `y` som en `int`. Detta fenomen kan ge upphov till riktigt konstiga fel. Om man menar att deklarera två pekare måste man skriva

```
int *x, *y;
```

och om man faktiskt menade att endast deklarera `x` som pekare bör man skriva

```
int *x, y;
```

[2]Detta gäller då ett maskinord är 32 bitar.

[3]Det kan vara värt att notera att en del datorer klarar av att skriva hela maskinord på adresser som inte är jämnt delbara med fyra. Till exempel x86 klarar detta, men det går mycket långsammare än att lägga datat på jämna adresser. På Sparc kommer man att få ett Bus Error

Genom att flytta stjärnan till variabelnamnet vid deklarationen av pekare kan man alltså undvika mycket problem. I fortsättningen kommer jag därför att sätta stjärnan vid variabelnamnet. Detta påverkar förstås inte betydelsen av stjärnan, den hör fortfarande till typen.

Operatorn *

Stjärnan kan även fungera som operator i C. Det den gör är att dereferera en pekare, det vill säga plocka ut värdet i den minnescell som pekaren pekar på. Observera att detta alltså inte är samma stjärna som men använder för att deklarera pekare. Vid deklarationen är stjärnan en del av typen, här är stjärnan en unär operator motsvarande `-` eller `!`.

```
int tal = 42;
int *adress;

adress = &tal;

(*adress) ==> 42
```

Man använder även stjärnan när man vill skriva något i den minnescell som pekaren refererar till. Det är fortfarande samma operator, `*`, som derefererar adressen så att vi kommer åt minnescellen istället för adressen.

```
*adress = 4711;

tal ==> 4711
```

3.2 Pekare och strukturer

Pekare och strukturer brukar nästan alltid användas tillsammans på olika sätt. vi har ett klassiskt exempel i Klondike, den länkade listan av spelkort. C erbjuder som tidigare nämnts inga inbyggda funktioner för att hantera listor. Vill man ha en lista i C så måste man skapa sin egen struktur som innehåller data och en pekare till nästa element i listan. I `cardpile.h` hittar vi strukturen för spelkort i Klondike. Det sista elementet, `next`, är just en pekare till nästa kort i listan.

```
typedef struct card_t
{
   suit_t  suit;         /* Färg               */
   value_t value;        /* Valör              */
   bool    visible;      /* Är kortet synligt? */
   bool    mark;         /* Är kortet markerat? */
   struct card_t *next;  /* Nästa kort i högen  */
} card_t;
```

Vi använder `typedef` för att skapa ett enkelt namn åt våra spelkort, `card_t`. Vi kan skapa pekare till spelkort genom att sätta stjärnan på den nya typen: `card_t *kortpekare`

Som vi kan se i strukturen har jag inte använt detta enkla sätt att skapa en pekare till nästa kort utan skrivit hela typnamnet (`struct card_t *`). Detta beror på att när vi deklarerar variabeln `next` och talar om vilken typ den ska ha, så är vi mitt uppe i `typedef` och den enklare typen som vi håller på att skapa finns helt enkelt inte än. Detta är också anledningen till att vi namngett strukturen. Vore den anonym så som till exempel `value_t` och `suit_t` är så skulle det inte gå att skapa en rekursiv pekare till den.

3.3 Pekare som argument och returvärde

Vi går tillbaka till `getvisible` och problemet med att skicka en struktur som argument. Problemets lösning bör vara uppenbar vid det här laget, istället för att skicka hela strukturen skickar vi bara en pekare till den. Via pekaren kan vi sedan komma åt alla fält i strukturen utan att behöva kopiera hela spelkortet till stacken. I ett program som Klondike där så gott som alla funktioner ska ta emot strukturer blir pekare extra viktigt för prestandan.

```
void getvisible(card_t *spelkort)
{
   return spelkort->visible;
}

card_t spelkort;
getvisible(&spelkort);
```

Vi ändrar `getvisible` så att den tar en pekare till ett spelkort som argument och anropet ändras så att det skickar adressen till spelkortet istället för hela strukturen. Här ser vi även en ny notation, "`->`". Pilen används för att komma åt fält i en struktur via en pekare till strukturen. Skillnaden mellan punkt och pil är alltså att punkten används när hela strukturen är på plats medan pilen används när vi bara har en pekare till strukturen.

```
card_t *nextcard(card_t *spelkort)
{
   return spelkort->next;
}

card_t *spelkort;
spelkort = nextcard(spelkort);
```

Man kan naturligtvis även ha funktioner som returnerar pekare. Ovan ser vi `nextcard`, en funktion som givet ett spelkort returnerar nästa kort i listan. Returtypen i `nextcard` är `card_t*`, men det vi returnerar har typen `struct card_t *`. Detta är helt OK eftersom dessa egentligen är samma datatyp och det kortare namnet är definierat av `typedef`.

När man returnerar pekare är det extremt viktigt att man tänker sig för så att man inte returnerar en pekare till stacken. Lokala variabler och argument kommer ju att försvinna så fort funktionen avslutas så att skicka tillbaka adresser till dessa är mycket farligt. `gcc` kommer att upptäcka detta och ge en varning, men långt ifrån alla C-kompilatorer gör detta.

`void*` och `NULL`

Pekartypen är som sagt något som C är mycket petig med. Det går inte att betrakta en pekare till en `int` som om den vore en pekare till `char`. Om man har en funktion som ska hantera adresser i allmänhet och inte så noga bryr sig om vilken sorts data som ligger på adresserna så finns det en speciell pekartyp som går att använda. `void*` är en generell pekare som man kan använda för att deklarera variabler, ta emot som argument och sätta som returtyp. Man måste alltid typkonvertera en `void*` till en "riktig" pekare innan man försöker följa pekaren för att plocka ut element i en struktur eller liknande.

Exempel på funktioner där vi hittar `void*` är `memcpy`, `memset`, `malloc` och `free` som finns i `stdlib` – det vill säga typiska systemfunktioner som hanterar generellt minne. Vi kommer att se mer av dessa framöver.

I funktioner som returnerar pekare vill man ofta ha en möjlighet att returnera en felkod som talar om att något gått snett och returvärdet inte är en giltig pekare. Pekare är alltid utan tecken, det vill säga de är alltid positiva. Så att returnera ett negativt värde som felkod fungerar inte. Alla positiva heltal som finns att returnera är giltiga adresser i datorn och går alltså inte heller

att välja som felkod – alla utom en. Adressen 0 är reserverad i datorn, den kan man inte använda som pekare vilket innebär att den blir en utmärkt kandidat för felkoden. Adressen 0 används så ofta i program att den till och med fått ett eget namn, `NULL`.

`NULL` används i alla sammanhang där man vill tala om att en pekarvariabel inte innehåller en giltig adress. Oanvända pekare bör få värdet `NULL` och next-pekaren i det sista elementet i en länkad lista har nästan alltid värdet `NULL`.

Pekararitmetik

Som jag nämnt så är C ganska petigt när det gäller pekare och dess typ. Man kan inte flytta en `char*` till en variabel deklarerad som `int*` utan att använda explicit typkonvertering (se föreläsning ett) för att övertyga kompilatorn om att man vet vad man gör. Detta trots att alla pekare är lika stora och det finns ingen risk för dataförlust på det sätt som sker om man flyttar ett flyttal till en heltalsvariabel.

Det finns naturligtvis flera anledningar till att C är så här petigt. En av dem är att det finns en stor felkälla i pekarhantering. Råkar man skicka en pekare till ett heltal till en funktion som förväntar sig att få in en pekare till ett spelkort så kommer det inte att gå bra när funktionen börjar följa pekaren för att komma åt fälten i strukturen.

En annan anledning är att pekare vet vad de pekar på och beter sig olika beroende på vad de har för typ. Om man adderar ett heltal till en pekare så kommer man att hamna på en ny adress som ligger lite längre fram i minnet än den adress som pekaren innehåller. Hur långt fram i minnet beror på storleken på den typ som pekaren pekar på.

Om vi adderar 5 till en `char*` så kommer vi att hoppa fem bytes fram i minnet eftersom en `char` är en byte stor. Om vi adderar 5 till en `int*` så kommer vi att hoppa tjugo bytes eftersom en `int` är fyra bytes stor ($4 * 5 = 20$).

Det samma gäller naturligtvis pekare till strukturer. Om vi har en pekare av typen `card_t*` och lägger till 5 till den så kommer vi att hoppa $5*$`sizeof(card_t)` ($= 5*16$) steg framåt i minnet.

Vad kommer det sig att ett spelkort är 16 bytes kan man undra. Strukturen innehåller fem variabler. De två första är uppräkningstyper, under ytan är dessa `int` som tar fyra bytes var. Sedan kommer två sanningsvärden, dessa kommer att implementeras som `char` och tar alltså en byte var. Det sista är en pekare och denna upptar fyra bytes. Men det borde väl bli 14 bytes, eller..?

Anledningen till att ett spelkort tar 16 bytes är att alla variabler som är ett maskinord stora (fyra bytes i det här fallet) måste börja på en jämn ordadress, det vill säga adressen måste vara jämnt delbar med fyra[4]. Spelkortet har därför fyllts ut med två bytes efter de två sanningsvärdena för att pekaren ska hamna på en tillåten adress.

3.4 Arrayer

Listor, som vi använder till våra spelkort, kan ibland vara lite krångliga att jobba med i C och de kräver en hel del extra arbete. Om man bara är ute efter att lagra ett antal saker av samma typ så finns det enklare sätt att åstadkomma detta. Den sista inbyggda typen vi kommer att ta upp i denna kurs (för det finns inte fler) är *arrayen*. En array kännetecknas av hakparenteserna (`[]`) och den är en datastruktur som är mycket vanlig i C-program.

```
int siffror[10];
```

Arrayen vi skapar här heter `siffror` och har tio platser för att lagra heltal. Det går inte att ändra storleken på en array efter att man har deklarerat den. Notera att en array alltid har en typ och den kan bara innehålla värden av den typ man angett. I arrayen ovan kan vi alltså bara lagra heltal med typen `int`.

[4]Se början av 'Adresser och pekare'.

Man kan initiera en array direkt vid deklarationen genom att skriva initialvärdena kommaseparerade inom måsvingar.

```
int heltal[] = { 10, 7, 25, 2, 37, 18, 6, 98, 3, 7 };
float flyttal[15] = { 3.45, 2.657, 5.54, 6.342 };
```

I den första arraydeklarationen ovan (`heltal`) angav vi ingen storlek på arrayen. Det innebär att antalet initialvärden kommer att avgöra storleken. Vi får då i det här fallet tio platser. I det andra exemplet, `flyttal`, anger vi en storlek men initierar endast de första fyra värdena. Övriga värden i arrayen kommer att vara oinitierade.

Att hitta en plats i en array tar konstant tid oavsett vilken plats man vill komma åt. Varje plats i arrayen markeras med ett index. När man ska komma åt en plats använder man hakparenteser (`[]`) efter namnet på arrayen och indexet som motsvarar platsen man vill komma åt. Man börjar räkna på noll. Det fjärde elementet i en array har alltså index tre och i arrayen `heltal` hittar vi där en tvåa, för att komma åt den i C skriver vi `heltal[3]`.

```
Index:     0    1    2    3    4    5    6    7    8    9
Heltal: [ 10 |  7 | 25 |  2 | 37 | 18 |  1 | 98 |  3 |  7 ]
```

En plats i en array kan användas som vilken annan variabel som helst. Vi kan skriva saker som `heltal[5]++` för att öka värdet på position fem i arrayen eller jämföra värdet på olika positioner med `heltal[2] > heltal[8]`. Man kan naturligtvis även använda variabler, uttryck och till och med funktionsanrop för att indexera i en array. Det enda kravet är att indexet i slutändan ska vara ett heltal.

```
heltal[x];
heltal[x + foo()];
heltal[(int)((5.76 * flyttal[x]) / heltal[y])];
```

Vi tittar på ett exempel för att se mer av hur arrayer kan användas. Funktionen `largest` nedan tittar igenom en array och returnerar det största elementet.

```
int largest(int arr[], int size)
{
   int i;
   int max = arr[0];

   for (i = 1; i < size; i++)
      if (arr[i] > max)
         max = arr[i];

   return max;
}
```

Det finns inget sätt att se på ett enskilt element om det är det sista i en array. Det finns heller ingen felkontroll som säger till om vi råkar gå utanför arrayen. Det är därför viktigt att vi vet storleken på en array innan vi börjar titta i den.

När vi skickar en array som argument till en funktion anger vi ingen storlek. Detta beror på att arrayen egentligen bara är syntax-socker – något som man egentligen kan klara sig utan men som ger mer läsbar kod. En array är egentligen samma sak som en pekare. Följande två kodrader är ekvivalenta. Det är inte så svårt att se varför man vill ha arrayen som ett sätt att förenkla koden.

```
heltal[5] = heltal[8] + 42;

*(heltal + 5) = *(heltal + 8) + 42;
```

Arrayen är alltså bara en pekare som pekar ut den första positionen i arrayen. Resterande positioner i arrayen kommer att ligga direkt efter i minnet så vi kan alltid komma åt dem genom att lägga på ett offset på pekaren eller genom att använda syntaxen för arrayen.

Bortsett från att det är lättare att se vad som händer i programmet när man använder arrayer finns ytterligare en fördel jämfört med att bara använda pekare. När man deklarerar en array allokeras minnet som arrayen behöver direkt av kompilatorn. Om man bara deklarerar en pekare så allokeras det inte något minne. Pekare kan man använda för att referera in i redan deklarerade arrayer. Det är dock viktigt att man tänker på vad man gör med dessa pekare. De arrayer vi har sett så här långt har alla deklarerats på stacken. Om man skapar en pekare till ett element i en stackallokerad array gäller naturligtvis precis som tidigare att man inte kan returnera en sådan pekare utanför den omgivning där arrayen deklareras – även om pekarvariabeln lever där ute.

```
void trasig(int storlek)
{
   char *pekare;

   if (storlek != 0)
   {
      char array[storlek];
      pekare = array;
   }
   else
      return;

   pekare[storlek - 1] = 'X';
}
```

Koden ovan innehåller två fel. Dels det jag ville visa, nämligen ett försök att komma åt en array utanför den omgivning där den skapades. Arrayen skapas i den lokala omgivning som hör till `if`-satsen och kommer inte att finnas tillgänglig utanför denna omgivning. Genom att använda en pekare som är deklarerad utanför omgivningen kan vi dock få med oss information om arrayen ut till koden som ligger efter `if`-satsen. Eftersom vi inte använder variabeln `array` så kommer kompilatorn inte att kunna upptäcka att det vi gör är fel. Ett fel av det här slaget kommer att vara mycket svårt att hitta och det är den här typen av buggar som kan finnas kvar i program i decennier och bara orsaka krascher varannat år.

Det andra felet är att vi har använt en variabel för att deklarera storleken på arrayen. Storleken på en array måste vara känd när programmet kompileras. Variabler och funktionsanrop är alltså inte tillåtna där[5].

I den omgivning där en array är deklarerad kan vi använda `sizeof` för att ta reda på hur mycket minne arrayen tar (räknat i bytes). Notera dock att denna storlek i regel inte stämmer överens med storleken på arrayen. Storleken på en position i en array bestäms nämligen av vilken typ av data man lagrar i sin array. Om vi lagrar tecken (`char`) så kommer varje position endast att ta upp en byte och storleken som ges av `sizeof` stämmer med antalet element. Däremot om vi har en array av till exempel heltal (`int`) så kommer varje position att ta upp fyra bytes (32 bitar) och mängden minne som arrayen tar upp kommer alltså att vara fyra gånger större än antalet positioner i arrayen.

För att få reda på hur många positioner arrayen har kan vi dela minnesstorleken för hela arrayen med minnesstorleken för en position i arrayen.

```
card_t array[52];
int storlek = sizeof(array) / sizeof(card_t);
```

Att ta reda på storleken av en array som vi fått arrayen skickad till oss via funktionsargumenten

[5]ISO C90 förbjuder arrayer av variabel storlek.

är inte helt trivialt. Argumentet som tar emot arrayen kommer nämligen alltid att betraktas som en pekare och `sizeof` på en pekare är alltid den samma oavsett storleken på arrayen. I de fall där man skickar med arrayer till funktioner som är beroende av att veta storleken på dem bör man därför även skicka med storleken via argumenten.

Som vi ser ovan är det inga problem att skapa arrayer av strukturer. Dessa kommer att fungera precis som arrayer av enkla datatyper. På varje position i arrayen kommer det att ligga ett helt spelkort. Vill vi skapa en array av pekare till spelkort är det precis lika enkelt. Vi kommer åt fälten i spelkorten med hjälp av en `.` på samma sätt som förut.

```
card_t cards[52];
cards[3].suit = HEARTS;

card_t *pile[52];
pile[42]->suit = CLUBS;
```

3.5 Strängar

Som jag sagt tidigare finns det ingen inbyggd datatyp för strängar i C. För att hantera text i C-program använder man istället arrayer av tecken.

```
char str[42];
```

Allt som sagts tidigare om arrayer gäller även strängar. Det enda som gör att strängar är strängar är att de är `NULL`-terminerade. Detta betyder att det sista tecknet i strängen är `'\0'` (`NULL`, tecknet med teckenkod noll) och att man alltså kan se när en sträng är slut utan att veta dess storlek.

Observera att detta betyder att det får plats ett tecken mindre i en sträng än man kanske förväntar sig. Om tecken-arrayen deklareras med 42 platser kan den innehålla 41 valfria tecken. Det sista tecknet måste ju vara `'\0'` för att det ska vara en sträng.

Notationen som vi använt tidigare för arrayer blir i strängar ett sätt att komma åt enskilda tecken i strängen. `str[0]` ger det första tecknet i strängen och `str[5]` det sjätte. Precis som alla andra datatyper kan strängar initieras direkt vid deklarationen.

```
char str[] = "Hej hopp!";
```

I uppgift 1 såg vi ett annat sätt att deklarera en sträng. Där finns `valuetostring` som returnerar strängar. Returtypen är satt till `char*`. Arrayer är ju som sagt bara ett annat sätt att skriva pekare, så att en array av tecken kan skrivas som `char*` är kanske inte så konstigt egentligen.

Eftersom `str` kan betraktas som en pekare även om vi deklarerade den som en array ovan, så kan vi komma åt substrängar genom att lägga till ett offset på `str`. `(str + 5)` kommer att betraktas som en pekare till strängen som börjar på det sjätte tecknet i `str`.

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

int main()
{
   char text[] = "C is the answer.";
   int i = 0;

   while (*(text + i) != '\0')
      printf("%s\n",text + i++);

   return EXIT_SUCCESS;
}
```

Programmet ovan har en del fuffens för sig med pekare. Prova att testköra detta program. Om ni skrivit av det rätt ska det ge utskriften nedan. Vad är det egentligen som händer i detta program?

```
C is the answer.
 is the answer.
is the answer.
s the answer.
 the answer.
the answer.
he answer.
e answer.
 answer.
answer.
nswer.
swer.
wer.
er.
r.
.
```

Standardbiblioteket `string` innehåller en hel del funktioner för att arbeta med strängar, titta igenom detta så att ni har lite koll på vad som går att göra.

3.6 Teckenkoder — ASCII med mera

Säg att vi vill skriva en funktion `toupper` som översätter en liten bokstav till motsvarande stor (och lämnar andra tecken oförändrade). Hur går man till väga för att skriva en sådan funktion i C?

Man kan naturligtvis lösa problemet med en stor `switch`. Vi fick ett `'a'`, då returnerar vi ett `'A'`, vi fick ett `'b'`, då returnerar vi ett `'B'`... Men det måste ju finnas något smidigare sätt att lösa detta.

För att kunna lösa problemet på ett bättre sätt måste vi förstå vad tecken är och hur datorn ser på dem. I datorn är allt bara tal. C som ligger mycket nära datorn är av samma åsikt, allt är bara tal. Vi kan prata om tecken och pekare och annat, men egentligen är det bara tal. Jag har sagt det här någon gång tidigare i denna kurs. Varje tecken vi kan skriva in i datorn representeras av ett tal. Mappningen mellan tecken och tal är standardiserad och återfinns i den så kallade *ASCII-tabellen*. ASCII-tabellen är egentligen bara en del av teckentabellen men det kan ni läsa mer om i appendix B.

Varje tal från 0 till 255 mappas till ett tecken. De första 32 är olika typer av styrkoder. Där hittar vi saker som `NULL`, radbrytning, tab med mera. Efter dessa följer ett antal olika tecken och på teckenkod 65 börjar de stora bokstäverna. Alla bokstäver ligger i ordning bortsett från de tre sista i det svenska alfabetet, å, ä, ö. Dessa har lagts till i ett senare skede och hamnade då i den andra halvan av tabellen. Det kan även vara värt att notera att alla siffror ligger i nummerordning.

Så alla tecken är tal, och alla tal är för en dator bara en hög med ettor och nollor. För att få full förståelse för tecknens relationer ser vi den binära kodningen av varje tecken i tabellen. Observera skillnaden mellan stora och små bokstäver. Endast en bit skiljer. Om vi ser till att den biten inte är ett så kommer vi att ha en stor bokstav i stället för en liten. Biten i fråga har värdet 32 och vi kan därför släcka den biten genom att utföra en logisk OCH med 2-komplementet till 32, alltså det tal där alla andra bitar är ett och biten för 32 är noll.

```
'a' = 01100001
 32 = 00100000
~32 = 11011111
'a' & ~32 = 01000001 = 'A'
```

Om binära tal känns obekanta så råder jag er att läsa på lite om detta. Det är inget som ingår i den här kursen men förståelse för hur en dator fungerar gör det mycket lättare att programmera den.

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

void toupper(char str[])
{
   int i = 0;

   while (str[i] != '\0')
      str[i++] &= ~32;
}

int main(void)
{
   char text[] = "hej hopp ökenrått a!";
   toupper(text);
   printf("%s\n",text);
   return EXIT_SUCCESS;
}
```

Här har vi då översatt resonemanget till C. Vi går igenom strängen och för varje tecken maskar vi bort 32. Kommer programmet att fungera? En testkörning ger svaret "`HEJ`". Det såg inte ut som det skulle. Varför tar strängen slut efter HEJ? Vad händer med mellanslaget? Titta i ASCII-tabellen efter tecknet för mellanslag (space). Kan man verkligen maska bort 32 i alla tecken hur som helst? Nej, det går förstås inte. Mellanslaget har teckenkod 32 vilket betyder att när vi maskar bort 32 blir det bara noll kvar, noll som terminerar strängen.

För att ordna en funktion som bara konverterar bokstäver måste vi lägga in ett villkor som kontrollerar att tecknet vi vill konvertera faktiskt ligger i rätt intervall (mellan `'a'` och `'z'`). Vi vill förstås också få med de svenska tecknen som vi hittar i den senare delen av ASCII-tabellen. Dessa ligger lämpligt nog på samma sätt som den första delen av alfabetet och även där kan vi förstora bokstäver genom att maska bort 32.

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

void toUpper(char str[])
{
   int i;

   for (i = 0; str[i] != '\0'; i++)
   {
      if ((str[i] >= 'a' && str[i] <= 'z') ||
          (str[i] >= 'à' && str[i] <= 'ÿ'))
      str[i] &= ~32;
   }
}

int main(void)
{
   char text[] = "hej hopp ökenråtta!";
   toUpper(text);
   printf("%s\n",text);
   return EXIT_SUCCESS;
}
```

3.7 Argument till `main`

Vid det här laget har ni sett flera exempel på hur olika funktioner kan ta argument av olika slag. Som vi noterat tidigare är även `main` en funktion som alla andra. Hittills har vi endast sett `main` utan argument, men det går att skicka argument även dit. Argumentens typ och ordning bestäms av anropskonventionerna i Unix (eftersom C har sina rötter i den världen) och dessa säger att `main` ska ta två argument[6].

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

Argumenten kommer från kommandoraden där programmet startas. Det första argumentet, `argc` (argument counter), är ett heltal som talar om hur många argument som skickades in till programmet och det andra, `argv` (argument vector), är en array av strängar där alla programmets argument ligger lagrade.

```
echo hello world 42
```

Ovan ser vi ett anrop till programmet `echo`. Vi skickar med tre argument, "hello", "world" och "42". Inne i C kommer `argc` att få värdet 4 och vi får tillgång till både programnamn och argument i `argv`.

```
argv[0] -> "echo"
argv[1] -> "hello"
argv[2] -> "world"
argv[3] -> "42"
```

Notera speciellt att även tal som skickas in som argument kommer att komma fram som strängar. För att konvertera tillbaka dem till tal finns en användbar funktion i `stdlib` som heter `atoi`.

[6]Det finns även en version av `main` med tre argument, men den ingår inte i denna kurs.

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

int main(int argc, char *argv[])
{
   int n;

   for (n = 1; n < argc; n++)
      printf("%s ",argv[n]);

   printf("\n");
   return EXIT_SUCCESS;
}
```

`echo` skriver ut sina argument till `stdout`. Vi använder en `for`-loop för att stega igenom argumenten. Notera att vi initierar `n` till 1 för att hoppa över filnamnet som ligger på första positionen i arrayen.

Föreläsning 4

Minnesallokering — pekare var bara början

När man pratar om minnet i C-program så finns det två olika områden som man bör känna till, *stacken* och *heapen*. Stacken har vi redan sett, det är där alla lokala variabler finns. Allting som ska finnas på stacken måste vara känt när programmet kompileras, både när det gäller storlek och antal. Ibland har man en datastruktur som kan vara olika stor under programmets körning, eller så vet man kanske inte hur många element en struktur ska ha för än man har startat programmet. Då behöver man en möjlighet att få tag i mer minne medan programmet kör, minne som inte var klart att vi skulle behöva när programmet kompilerades. Standardbiblioteket `stdlib` erbjuder dynamisk minneshantering genom bland annat `malloc` och `free`.

4.1 Dynamisk minneshantering — `malloc`, `free`

Med ett anrop till `malloc` talar man om att man behöver mer minne. Man skickar med hur mycket minne man vill ha som argument. `malloc` returnerar en pekare till det minne operativsystemet ger till programmet. När man är klar med minnet ska det lämnas tillbaka till operativsystemet. Detta görs med kommandot `free`. Till `free` skickar man med den pekare som man fick från `malloc` för att tala om vilket minne det är som ska lämnas tillbaka. Man brukar säga att man *allokerar* och *frigör* minne.

Eftersom `malloc` och `free` inte kan veta vilken typ av data som ska lagras i det nya minnet så kommer de alltid att returnera och ta emot `void*`. För att kunna använda minnet får man typkonvertera pekaren man får från `malloc` till den typ man använder i sitt program. När man lämnar tillbaka minnet måste man typkonvertera pekaren tillbaka till `void*`.

```
int storlek = sizeof(card_t);
card_t *kort = (card_t*)malloc(storlek);

free((void*)kort);
```

Allt minne som vi allokerar på det här sättet kommer att hamna i den delen vi kallar heap. Man brukar sträva efter att alltid allokera stora datastrukturer på heapen. Stacken har som vi sagt tidigare en fast storlek och denna är normalt ganska liten jämfört med hur mycket minne det finns i datorn. Heapen däremot har tillgång till allt övrigt minne i datorn.

4.2 Faran med manuell minneshantering

Om man bortser från att själva hanteringen av pekare i sig kan vara besvärlig och leda till många svårhittade fel i program, finns det främst två faror med att vara personligt ansvarig för att lämna

tillbaka minne till operativsystemet.

De två senarier som vi kan tänka oss är

1. vi lämnar tillbaka minne innan vi är klara med det, och
2. vi glömmer att lämna tillbaka minnet.

Om vi lämnar tillbaka minnet för tidigt kommer vi att sitta med så kallade *dangling pointers*. Dessa är pekare som vi fortfarande använder i programmet trots att de pekar på minne som vi frigjort. Den här typen av fel kan vara mycket svåra att hitta eftersom det frigjorda minnet fortfarande går utmärkt att använda tills dess att man någon annan del av programmet allokerar det. Hur det kommer sig att minnet går att använda efter det är frigjort återkommer vi till strax.

Om vi glömmer att lämna tillbaka minnet vi har allokerat så har vi en minnesläcka. Små program med körtider på några sekunder lider inte så mycket av att minnet inte lämnas tillbaka. Minnet kommer att förbli ockuperat till dess att hela programmet avslutas, men när detta sker kan operativsystemet återta allt minne som programmet allokerat. I program som kör länge är detta däremot ett farligt fel eftersom det innebär att minnet så småningom kommer att ta slut. Olika server-program till exempel eller styrprogram i maskiner av olika slag kan ha körtider på flera år.

Det är lätt att tänka att man inte behöver fundera så mycket på det där med att lämna tillbaka minnet om man skriver små program som på sin höjd ska köra ett par sekunder. När programmet är slut kommer minnet ändå att frigöras helt automatiskt. Det är farligt att tänka så. För det första är det endast sant att minnet frigörs automatiskt om man har ett operativsystem som erbjuder den tjänsten. I moderna persondatorer fungerar det så, men majoriteten av alla C-program som skrivs kommer inte att köras på en modern persondator utan snarare på en något äldre processormodell i ett inbyggt system någonstans. Dessa har ofta starkt begränsade operativsystem om ens det och att minnet lämnas tillbaka av sig självt när programmet slutar ska man inte räkna med.

En annan risk med att inte frigöra minne är att man döljer fel i programmet. Man kanske tror att man är klar med en bit minne och borde frigöra den, men någonstans finns en pekare som man glömt att uppdatera som fortfarande pekar till detta minne. Om man frigör minnet finns det en chans att felet orsakar problem och upptäcks (man har då en dangling pointer), om man inte frigör minnet har man ett fel som är betydligt svårare att hitta.

Man bör alltid ha som vana att frigöra det minne man allokerar. Om inte annat så bara för att visa att man vet vad man håller på med.

Segmentation fault

Om man försöker komma åt minne som programmet inte har tillgång till kommer operativsystemet att klaga. Man får då ett segmenteringsfel vilket helt enkelt betyder att man försöker komma åt ett minnessegment som man inte har allokerat.

Hur kommer det sig att vi kan använda minne som vi frigjort utan att något händer? Anledningen är att programmet inte lämnar tillbaka minnet direkt när det frigörs utan håller i det ett tag. Om man allokerar nytt minne strax efter att man har frigjort minne så kommer programmet att se till att man får tillbaka samma minne som man just frigjort. Detta för att slippa göra systemanrop till operativsystemet när man allokerar, vilket skulle ta betydligt längre tid.

4.3 Allokera arrayer

En av anledningarna till att vi vill allokera minne på egen hand istället för att låta allt hamna på stacken är att stacken har en begränsad storlek. Stora datastrukturer läggs därför lämpligen på heapen istället. Vi ska nu titta på hur vi kan skapa en heapallokerad array av spelkort.

```
card_t *kortlek = (card_t*)malloc(sizeof(card_t) * 52);
```

Ja, svårare än så är det inte. Vi har en pekare (`kortlek`) där vi sparar adressen till kortleken. Adressen får vi från `malloc` när vi ber om ett stycke minne. Adressen som `malloc` returnerar är av typen `void*` och vi typkonverterar den till en pekare till spelkort för att kunna lagra den i vår variabel. Storleken som vi skickar till `malloc` beräknar vi genom att ta storleken på ett spelkort och multiplicera med 52. Detta ger en array med plats för 52 spelkort.

```
card_t *kortlek = (card_t*)malloc(sizeof(card_t*) * 52);

for (i = 0; i < 52; i++)
   kortlek[i] = malloc(sizeof(card_t));
```

Här allokerar vi ännu en array. Den här gången tar vi storleken av en pekare till spelkort gånger 52. Vi har alltså allokerat en array med plats för 52 pekare. De två arrayerna skiljer sig på en fundamental punkt – var själva spelkorten ligger i minnet. Hur menar jag nu..? Allt vi har allokerat här ligger ju på heapen. Skillnaden är att i det första exemplet så ligger själva korten i arrayen – de ligger på samma fysiska plats i minnet. I det andra fallet ligger arrayen på ett ställe och innehåller pekare till spelkorten som ligger utspridda på andra ställen.

Precis som när vi deklarerar variabler på stacken så är minnet vi får av `malloc` oinitierat. Vi kan alltså inte förutsätta något om innehållet utan måste själva skriva dit värden innan vi kan börja använda minnet. Arrayen av pekare ovan initierar vi med en `for`-loop där vi allokerar kort som pekarna i arrayen får referera till. Den första arrayen, den där korten ligger i arrayen, har vi dock inte initierat ännu. Detta skulle vi förstås kunna göra genom att initiera alla fält i alla kort med hjälp av en `for`-loop. Men låt oss säga att vi inte är intresserade av att ge alla kort riktiga värden ännu, utan bara vill se till att det inte ligger något gammalt skräp kvar i korten. Det finns två sätt att göra detta, antingen kan man använda `calloc` istället för `malloc`. `calloc` initierar automatiskt allt allokerat minne till 0. Detta betyder att det på varje minnesadress i hela minnesarean man allokerar står värdet 0. Man kan även använda `memset`. `memset` fyller en given minnesarea med ett givet värde, till exempel kan man säga att vi vill fylla hela arrayen av spelkort med värdet 0. Vi skickar med en pekare till arean vi vill fylla, värdet vi vill fylla med och storleken på arean.

```
memset(kortlek,0,sizeof(card_t) * 52);
```

Notera!

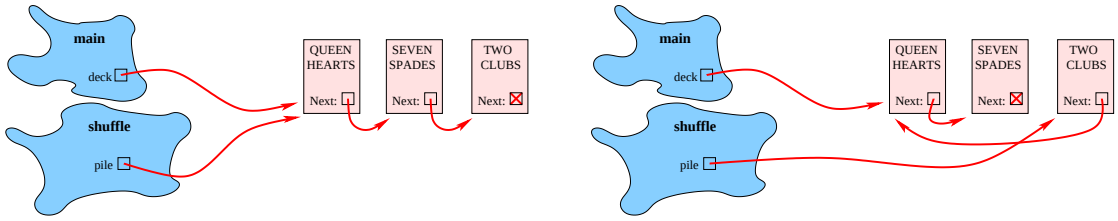
När det gäller kortleken ni ska implementera i inlämningsuppgiften så bör ni tänka på hur ni kommer att använda korten innan ni allokerar minnet i uppgift fyra. Exemplen här visar hur korten ligger i en array och hur man kan använda arrayen för att hantera korten. Ni kommer inte att ha någon nytta av att korten ligger i en array i uppgiften. Spelkorten i uppgiften ligger i länkade listor och ni kommer att flytta runt korten mellan olika listor under hela spelet. Att tänka på spelkorten som om de ligger i en array kommer därför endast att försvåra förståelsen för uppgiften. Ni kommer dessutom att frigöra minnet i flera omgångar - en korthög i taget. Så att allokera alla kort i en minnesklump kommer inte att fungera. Min rekommendation är att ni allokerar varje kort för sig, så att ni ser den som de fristående objekt ni kommer att behandla dem som.

4.4 Pekare till pekare — **

I uppgift tre ska ni bland annat skriva en funktion som blandar kortleken, `shuffle`. Låt oss se vad som händer i den funktionen.

```
void shuffle(card_t *pile)
{
}
```

Vi har allokerat en korthög på heapen och anropat `shuffle`. Som argument skickade vi med en pekare till korthögen som ska blandas. I figur 4.1(a) ser vi hur det kan se ut. Vi har den länkade listan av spelkort och två lokala omgivningar, `main` och `shuffle`. I `main` finns en pekare, `deck`, som refererar till korthögen. Detta är pekaren som vi skickade som argument till `shuffle` och återfinns därför i den lokala variabeln `pile` i `shuffle`.



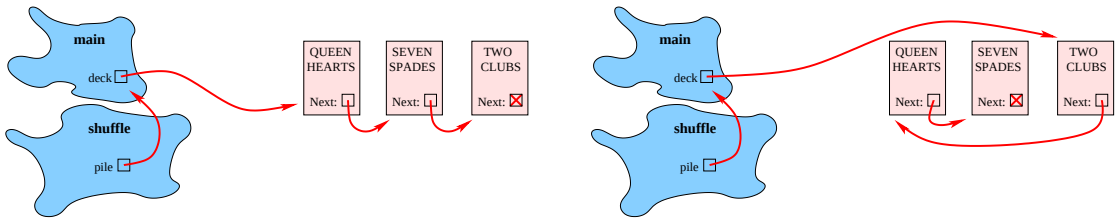
(a) Före (b) Efter

Figur 4.1: En korthög före och efter blandning.

Vi blandar korten och uppdaterar `pile` så att den pekar ut det kort som nu ligger först i listan, se figur 4.1(b). Vad händer då när vi returnerar från `shuffle`? Den lokala omgivningen försvinner och vi kommer tillbaka till `main`. Variabeln `deck` i `main` har vi inte ändrat på. Den pekar fortarande på det kort som låg först i listan före blandningen. Som vi kan se tappar vi här bort kort och resultatet är inte riktigt vad vi hade tänkt oss. För att lösa detta måste vi få tillgång till `main`:s variabel `deck` inne i `shuffle`.

```
void shuffle(card_t **pile)
{
}
```

När jag introducerade pekare sa jag att det går att skapa pekare till alla datatyper som finns i C. Detta gäller förstås även datatyper för pekare. Vi ändrar därför definitionen av `shuffle` till att ta en pekare till en pekare som argument. Notera de två stjärnorna (`**`) vid argumenttypen. Istället för att skicka med en pekare till korthögen när vi anropar `shuffle` skickar vi alltså en pekare till variabeln `deck`.



(a) Före (b) Efter

Figur 4.2: Pekare till pekare före och efter blandning.

Figur 4.2(a) visar det nya tillståndet. Variabeln `pile` pekar nu inte på korthögen utan på variabeln `deck` i `main`. När vi blandar korten och uppdaterar pekaren till det första kortet så är

det nu inte den lokala variabeln `pile` vi uppdaterar, utan `deck` i `main`. För att åstadkomma detta rent praktiskt i C använder vi operatorn `*`. `*` ger oss värdet på den plats som en pekare pekar på. När vi skriver `*pile` följer vi alltså pekaren från `pile` till `deck` och opererar på värdet vi hittar där.

`pile = x;` — Lägg värdet från `x` i den lokala variabeln `pile`

`*pile = x;` — Lägg värdet från `x` i den variabel som `pile` pekar på, alltså `deck` i vårt exempel.

`(*pile)->value = x;` — Lägg värdet från `x` i fältet `value` i det spelkort som `deck` pekar på.

`**pile` — Följ pekaren från `pile` till `deck`, följ sedan den pekare som finns där. Vi kommer nu att hamna på ett spelkort och typen på detta uttryck är `card_t` – alltså inte längre en pekare.

`(**pile).value = x;` — Lägg värdet från `x` i fältet `value` i det spelkort som `deck` pekar på.

Notera att det är en avsevärd skillnad på stjärnan i "`card_t *pile`" och stjärnan vi använder för att följa en pekare. De två har absolut ingenting med varandra att göra. I "`card_t *pile`" hör stjärnan till typen, kom ihåg att vi från början skrev det som "`card_t* pile`". Denna stjärna talar om att `pile` är en pekare (till ett spelkort).

Operatorn `*` som vi använder för att följa en pekare är just en operator, ett kommando. Man kan se det som en funktion som tar en pekarvariabel som argument och returnerar innehållet på den plats i minnet som pekaren pekar på, "`innehåll = *(pile);`". Den enda skillnaden jämfört med ett funktionsanrop är att vi även kan skriva `*pile` på vänster sida om likhetstecknet som i exemplen ovan.

Varför `**`?

Man kan undra varför vi väljer att krångla till det med dubbla pekare istället för att bara returnera den nya pekaren. Om `shuffle` skickade tillbaka pekaren till kortet som ligger först i listan efter blandningen så skulle vi ju kunna ta emot det i `main` med `deck = shuffle(deck);`.

För `shuffle` fungerar detta, men tänk på vad som händer om vi till exempel vill flytta de översta korten från en hög till en annan. `movemarked` i uppgift tre gör precis detta. Båda högarna kommer att ha andra kort överst efter flytten. Vi kan inte returnera två värden i C så minst en av högarna måste skickas in till funktionen som en pekare till en pekare.

Funktionen `drawone` tar bort det översta kortet i högen och returnerar det. Även där är det ett nytt kort som hamnar överst på högen. Vi kan inte returnera en pekare till den nya högen eftersom vi ska returnera kortet som vi tog bort. För att skapa ett enhetligt gränssnitt till funktionerna i cardpile har jag valt att använda pekare till pekare även i de enstaka fall där det egentligen skulle gå att returnera den nya högen.

Föreläsning 5

Makron - Snyggt och effektivt

Ni har säkert lagt märke till att det finns en del kod i inlämningsuppgiften som ser lite udda ut, främst i början av `klondike.c`. Det är rader som börjar med `#`, och många av dem slutar med ett \. Dessa rader är inte C-kod som en kompilator skulle svälja, om de tog sig fram till `gcc`:s kompileringssteg skulle de generera syntaxfel. Lyckligtvis kommer de aldrig så långt. Innan programmet kompileras skickas det nämligen igenom en så kallad pre-processor. Denna kommer att ta hand om alla rader som inleds med ett `#`. Bland annat läser pre-processorn in `.h`-filer via kommandot `#include`. `.h`-filerna kommer att klistras in i koden på den plats där `#include` står och kompilatorn kommer att se all kod som en enda fil. Det är därför man aldrig behöver skriva `.h`-filernas namn på kommandoraden till `gcc`.

5.1 Makron — `#define`

Rader som inleds med `#define` deklarerar makron. Förenklat kan man säga att man med ett makro kan byta ut en text i sitt program mot någon annan text. Man kan skapa allt från makron som är enkla namn på konstanter till makron som fungerar som hela funktioner. Oavsett hur makrot ser ut och vad det gör så handlar det om att byta ut en text mot en annan. Makron används för att förenkla och förtydliga kod. Det öppnas inga principiella nya möjligheter med makron men de underlättar till exempel representationsoberoende programmering och tillåter konstruktioner som skulle bli mycket krångliga att implementera på annat sätt. I `cardpile.h` hittar vi en av de enklare typerna av makron.

```
#define EMPTY NULL
```

Detta makro definierar namnet `EMPTY`. I koden kan vi använda detta namn som om det vore en konstant deklarerad med `const` eller `enum`. Den stora skillnaden är just att texten `EMPTY` kommer att bytas ut mot texten `NULL` innan programmet kompileras. Detta görs som sagt av pre-processorn och kallas för *makroexpansion*.

Ett makro av det här slaget underlättar till exempel om man vill bygga program där det är möjligt att byta datarepresentation på ett enkelt sätt. Säg till exempel att vi vill byta ut den länkade listan av spelkort mot ett träd, mer specifikt ett AA-träd[1]. I AA-träd använder man aldrig NULL för att markera att en nod är sist utan man har en speciell så kallad null-nod. `EMPTY` kommer då att definieras som en pekare till en null-nod. Om man sett till att använda `EMPTY` överallt istället för `NULL` så kan man byta ut denna definition utan att behöva ändra något mer i hela programmet.

Mer avancerade makron kan man skriva om man har ett stycke kod som ska utföras på många ställen i ett program, till exempel en utskrift av en meny.

[1] Exakt hur ett AA-träd ser ut och fungerar har ingen betydelse i sammanhanget.

```
#define PRINT_MENU {                                                    \
   printf(ANSI_CLEARSCREEN ANSI_HOME ANSI_BLACKCARD);                   \
   printf("Klondike 2006 v1.0");                                        \
   printf("Välj rad att markera med pilen.\n");                         \
   printf("Använd '%c' - upp, och '%c' - ner för att styra.\n",         \
          CHAR_UP, CHAR_DOWN);                                          \
   printf("'%c' markerar kort eller flyttar markerade kort.\n", CHAR_MARK);  \
   printf("'%c' lägger ut nya kort från kortleken.\n", CHAR_CARDS);     \
   printf("Tryck '%c' för att avsluta\n", CHAR_QUIT);                   \
}
```

Här ser vi att raderna avslutas med "\". Detta betyder att makrodefinitionen fortsätter på nästa rad. En annan sak som kan vara värd att notera är att vi kan använda makron i makron. Ovan använder vi ett antal makron som är definierade i `ansi.h`.

```
if (timetoprint == true)
   PRINT_MENU;
```

Anrop till makrot görs genom att skriva texten `PRINT_MENU` där man vill att koden ska klistras in. Om vi sätter ett semikolon efter makroanropet eller inte har inte så stor betydelse. Makroexpansion handlar som sagt om rent utbyte av text och när pre-processorn är klar med detta kommer det att se ut så här[2]:

```
if (timetoprint == true)
{
   printf("\033""[47m""\033""[2J" "\033""[H" "\033""[30;47m");
   printf("Klondike 2006 v1.0");
   printf("Välj rad att markera med pilen.\n");
   printf("Använd '%c' - upp, och '%c' - ner för att styra.\n",
          'u', 'n');
   printf("'%c' markerar kort eller flyttar markerade kort.\n", 'm');
   printf("'%c' lägger ut nya kort från kortleken.\n", 'k');
   printf("Tryck '%c' för att avsluta\n", 'q');
};
```

Notera att samtliga makron nu har blivit utbytta. Semikolonet hamnar efter måsvingen och gör varken till eller från. Min rekommendation är dock att man sätter dit ett semikolon ändå eftersom det get ett enhetligt utseende på koden där själva anropet står. Här kan vi också se hur viktigt det är att man är lite paranoid när man skriver sina makron. Egentligen vinner makrot inget på att slås in i måsvingar. Vi har inga lokala variabler eller annat som motiverar att vi skapar en ny omgivning. Vid anropet kan vi dock se att det är oerhört viktigt att hela makrot är inslaget i måsvingar. Utan måsvingar skulle det ju bara vara den första satsen i makrokoden som låg i `if`-satsen, resten skulle hamna utanför och utföras varje gång oavsett om det var dags att skriva ut eller ej. Man kan aldrig veta hur man får för sig att anropa ett makro så därför bör man alltid slå in makron i måsvingar om dessa innehåller fler än en sats.

Om man vill se hur ett program ser ut efter att pre-processorn har gjort sitt kan man anropa `gcc` med flaggan `-E`. `gcc` kommer då skriva ut källkoden till `stdout` vilket normalt hamnar i terminalfönstret. Eftersom det kan bli ganska mycket kod som skrivs ut då alla `.h`-filer inkluderats och makron expanderats så kan det vara lämpligt att omdirigera `stdout` till en textfil när man gör detta. I ett unix shell (`sh`, `bash`) görs detta med '`>`'.

```
gcc -E klondike.c > namn_på_textfil
```

[2]Egentligen blir det inte riktigt så prydligt eftersom hela makrot kommer att betraktas som en enda rad

5.2 Makron med argument

```
#define AREA(radie) radie*radie*PI
```

Man kan skicka med argument till ett makro på samma sätt som man gör vid funktionsanrop. Namnet på argumentet kommer att identifieras i makrotexten och bytas ut mot det man skickar med. Det är ett par saker man måste tänka på här. För det första så pratar vi fortfarande om rent utbyte av text. Argumenten till ett makro kommer inte att beräknas innan makroexpansionen som fallet är vid funktionsanrop. De kommer att kopieras in i makrotexten utan förändring. Detta innebär att om man har en tung beräkning som argument till ett makro, och detta argument förekommer flera gånger i makrotexten, så kommer den tunga beräkningen att utföras flera gånger.

```
#define PRINT_DOUBLE(value) printf("Double: %d\n",value + value)

PRINT_DOUBLE(count_atoms_on_earth());
```

Den andra faran med makron som tar argument är att man när man skriver makrot aldrig kan veta hur argumenten man får in ser ut. I en funktion vet vi vilken typ argumentet har och vi vet att det är en variabel som vi kan räkna med. Detta är inte fallet i ett makro. Vi har ingen typkontroll på argument till makron och det är inte säkert att argumentet beter sig som en variabel trots att det ser ut som en då vi skriver makrot.

```
AREA(3+2);
```

Anropet ovan kommer att expanderas till `3+2*3+2*PI`. Arean av en cirkel med radien 5 bör bli 78,5, men detta anrop kommer att ge 15,3 eftersom multiplikation binder hårdare än addition. Det är därför än en gång läge att vara paranoid. Alla förekomster av argumentnamn i ett makro bör förses med parenteser.

```
#define AREA(radie) (radie)*(radie)*PI
```

Vi vet sedan tidigare att funktionsanrop kopierar argumenten till stacken. Detta kan bli kostsamt, speciellt om man har funktionsanrop som ligger i en loop och utförs tusentals gånger. Det är därför ofta värt de extra försiktighetsåtgärderna som krävs när man skapar makron för att öka prestandan i ett program. Det är extremt vanligt att man till exempel använder makron till get- och set-funktioner eftersom dessa förekommer överallt i vanliga program.

```
#define SET_SUIT(card,s) {(card)->suit = (s)}

SET_SUIT(kort,CLUBS);  ->  {(kort)->suit = (CLUBS)}
```

##

I `SET_SUIT` ovan anger vi argumentnamnet `s`. Detta innebär att alla förekomster av `s` i makrotexten ska bytas ut mot det som skickas in vid anropet (`CLUBS`). Men som vi kan se i den expanderade koden är det inte alla `s` som bytts ut, det står inte "`{(kort)->CLUBSuit = (CLUBS)}`". Makroexpansionen är smartare än så. Endast hela ord byts ut. Om ett argumentnamn förekommer i en annan text i makrot kommer det inte att bytas ut.

```
#define INC(number) var_number++;

int var_one, var_two;

INC(one);
INC(two);
```

Tanken är att makrot `INC`, via argumentet, ska välja rätt variabel att lägga till ett på. Detta kommer dock som vi sett inte att fungera. Argumentet `number` återfinns inte i makrotexten utan det enda som finns där är `var_number` som betraktas som ett enda ord. För att skilja på de två delarna av variabelnamnet och tillåta att den andra delen av det byts ut av pre-processorn finns `##`. Genom att lägga in `##` mellan `var_` och `number` kan vi uppnå önskat resultat. Med "`var_##number`" blir `number` ett separat ord som kan bytas ut av pre-processorn.

5.3 Makron som uttryck

Hittills har vi inte direkt reflekterat över att det finns en skillnad mellan uttryck och satser. När det gäller makron är det dock viktigt att man förstår skillnaden. Satser är kommandon som inte resulterar i ett värde som vi tar hand om, medan uttryck beräknar just ett värde som vi vill ta hand om på något sätt. Ett anrop till `printf` är en sats. Visserligen har `printf` ett returvärde, men detta ignoreras så gott som alltid. `5 + 3` är ett uttryck. Om vi inte tar hand om resultatet är koden meningslös. Uttryck kan skrivas överallt där vi förväntar oss ett värde; vid tilldelningar, som argument vid funktionsanrop eller som villkor i en `if`.

Man använder ofta makron för att beräkna vanliga uttryck. `AREA` ovan är ett exempel på ett makro som fungerar som uttryck. Vi kan till exempel skriva "`x = AREA(5);`"

(sats, sats, sats)

För att ett makro ska fungera som uttryck är det viktigt att det inte slås in i måsvingar. Måsvingarna skapar en omgivning, och en omgivning har inget returvärde. Man kan inte skriva "`x = {5 + 3;};`" Istället slår man alltid in makron som ska användas som uttryck i parenteser. Detta medför att man får problem om man vill utföra flera satser i ett makro som i slutändan ska betraktas som ett uttryck.

Lyckligtvis tillåter C att man skriver kommaseparerade satser inom parenteser. Man byter helt enkelt ut semikolonet efter satserna mot komma och sätter parenteser runt. Det som skrivs på den sista positionen i parentesen kommer att fungera som hela parentesens värde. När vi anropar `SOMETHING` nedan så kommer "Hej!" att skrivas ut och `x` får värdet 42.

```
#define SOMETHING (printf("Hej!\n"),42)

x = SOMETHING;
```

Parentesuttrycken är helt vanlig C-kod och kunde tagits upp betydligt tidigare i kursen. Men det är först när man börjar med makron som dessa uttryck blir riktigt motiverade. I vanlig kod bidrar de mest till att göra koden svårläst.

uttryck_1 ? uttryck_2 : uttryck_3

Man kan inte använda `if`-satser i parentesuttryck vilket man ibland skulle vilja. För att skapa makron med villkorliga satser finns istället `?`. Om uttrycket framför frågetecknet är sant kommer uttryck_2 att beräknas, annars uttryck_3. Ett klassiskt exempel på detta är `MAX` som returnerar det största av två tal.

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

5.4 Villkorlig kompilering — `#ifdef`, `#endif`

Det är ganska vanligt att man har en del kod i sitt program som finns där enbart för att fånga buggar. I klondike har vi till exempel ett antal kontroller som räknar hur många kort det är i de olika korthögarna, och ett antal utskrifter som visar hur initieringen av spelet går. Kod av

det här slaget är mycket användbar medan man utvecklar sitt program, men oftast vill man (av prestandaskäl) ta bort koden innan man lämnar ifrån sig en slutgiltig version. Att klippa bort koden är inte ett hållbart alternativ eftersom den behövs igen då man lite senare hittar fel i programmet eller vill vidareutveckla det.

Det som behövs här är villkorlig kompilering – ett sätt att tala om att delar av koden ska finnas med vid vissa tillfällen och utelämnas vid andra. Detta kan pre-processorn göra åt oss. Det går nämligen att fråga pre-processorn om ett visst makronamn är definierat eller ej, och välja att ta med eller utelämna kod beroende på detta. Med `#ifdef` frågar vi om ett namn är definierat. All kod som ligger mellan `#ifdef` och `#endif` kommer att finnas kvar och skickas till kompileringen om namnet man frågar efter finns definierat. Detta påminner mycket om en vanlig `if` i C och precis som i C-koden kan en `#ifdef` följas av en `#else` om man vill göra något annat när namnet inte är definierat.

```
#define DEBUG

#ifdef DEBUG
#   define DPRINT(text) printf text;
#else
#   define DPRINT(text) ;
#endif
```

Om namnet `DEBUG` är definierat så kommer `DPRINT` att expanderas till `printf`, annars blir det bara ett semikolon kvar av makroanropet. Namnet `DEBUG` definieras någonstans tidigare i koden och detta makro behöver inte ha någon makrotext eftersom vi endast är intresserade av att namnet som sådant ska vara definierat.

Man måste i de flesta fall ha med `#else`-biten också även om man som i det här fallet inte vill att makrot ska expanderas till något om `DEBUG` inte är definierat. Detta beror på att man har anrop till makrot i C-koden. Om namnet `DPRINT` inte definieras av pre-processorn kommer den heller inte att ta hand om dessa anrop vilket betyder att de finns kvar när kompilatorn sätter tänderna i koden. `DPRINT` är inget som kompilatorn känner till och vi får då ett kompileringsfel.

Det finns naturligtvis många andra användningsområden för `#ifdef` också. Till exempel är det vanligt i många C-program att man har kod som ligger mycket nära hårdvaran. Vi kan tänka oss en drivrutin för skrivare till exempel. Olika skrivarmodeller kan ha olika sätt att kommunicera med drivrutinen. Så även om resten av drivrutinskoden är identisk mellan olika modeller blir drivrutinen ändå beroende av att kommunikationskoden kompileras för rätt skrivare. I dessa fall är det vanligt att man definierar ett makronamn som talar om vilken kod som ska kompileras.

Ett annat fall där man har stor nytta av villkorlig kompilering är då man skriver `.h`-filer. Som jag sa i början av denna föreläsning så betyder `#include` att man klistrar in hela innehållet i `.h`-filen i koden. Detta kan leda till problem om man inte är försiktig.

Vi tänker oss att vi skriver en `.h`-fil till spelet klondike, `klondike.h`. I denna använder vi datatyper som vi definierat i cardpile och därför måste vi inkludera `cardpile.h` i början av vår nya `.h`-fil. Vi skriver sedan kod i klondike som använder vår nya `.h`-fil och naturligtvis inkluderar vi `klondike.h` i början av `klondike.c`. När pre-processorn nu börjar klistra in `.h`-filer så kommer den först att se att `klondike.c` vill ha `klondike.h` och `cardpile.h` och klistra in dessa i koden. Sedan ser den att `klondike.h` vill ha `cardpile.h` och klistrar in den en gång till. Resultatet blir att alla datatyper som deklareras i `cardpile.h` deklareras två gånger. Det kommer kompilatorn att klaga på.

För att undvika att samma `.h`-fil läses in mer än en gång i ett program bör man därför definiera ett unikt namn för varje `.h`-fil och fråga om detta namn redan finns definierat innan man läser in filen. Detta görs i två steg. I början av `.h`-filen frågar man om filens unika namn är definierat, om inte fortsätter man med att omedelbart definiera namnet för att markera att denna `.h`-fil nu är inläst. Sedan följer hela filens innehåll innan man avslutar med en `#endif`. För att fråga om något namn *inte* är definierat använder man `#ifndef`, med ett 'n'. Detta är naturligtvis gjort i `cardpile.h` så ni kan se där hur det ser ut i verkligheten.

```
#ifndef __CARDPILE_H__
#define __CARDPILE_H__

.h-filens innehåll hamnar här

#endif /* __CARDPILE_H__ */
```

Jag har valt att tillverka det unika namnet `__CARDPILE_H__` för filen `cardpile.h`. Detta är en ganska vanlig konvention och det är lätt att förstå vad namnet betyder. Som vi kan se ovan har jag även valt att sätta en kommentar vid `#endif`. Det är en bra idé att göra det eftersom det blir ganska långt mellan `#ifdef` och `#endif`. Även i fall där det bara är något tiotal rader mellan `#ifdef` och `#endif` tycker jag att man ska sätta dit kommentaren för att göra det tydligt. I stora program är det lätt att det blir många olika namn som definieras och en del `#ifdef` kommer att hamna i varandra. Så det är lika bra att ha denna goda vana redan från start.

Appendix A

Slumptal

Algoritmen som vi använder för att blanda kortleken vill ha tag i ett slumptal. Funktionen för att generera slumptal i C finns i `stdlib` och heter `rand`. `rand` returnerar ett heltal mellan (och inklusive) 0 och `RAND_MAX`. `RAND_MAX` är ett hyfsat stort tal, ANSI-C definierar det till minst 32767 (maxvärdet för ett 16-bitars `signed int`).

Normalt när man vill ha slumptal har man dock sin egen maxgräns. För att anpassa slumptalet till denna gräns delar vi talet vi får av `rand` med `RAND_MAX + 1`. På det sättet får vi ett slumptal mellan 0 och 1 som vi sedan kan multiplicera med vår egen övre gräns. Vi vill att talet vi får ut efter divisionen ska ligga i intervallet [0,1), det vill säga det ska inkludera 0, men inte 1. Eftersom `rand` kan returnera `RAND_MAX` måste vi därför dividera med ett tal som är lite större, därav `+1`. Intervallet är viktigt för att ge en jämn fördelning av resultatet[1]. Om vi dividerar två heltal så kommer C att använda heltalsdivision, resultatet kommer alltså att trunkeras[2] till ett heltal som i det här fallet (när resultatet ligger mellan 0 och 1) alltid blir 0. För att tvinga fram flyttalsdivision måste minst en av täljare och nämnare vara flyttal. När allt är klart gör vi om resultatet till en `int` igen.

```
(int)((rand() / ((double)RAND_MAX + 1)) * max)
```

Slumptalsmotorn i C använder ett så kallat frö för att initiera slumptalsgenereringen. Ett givet frö kommer alltid att resultera i samma slumptalsföljd. Man kan ange sitt eget frö med funktionen `srand`. För att det inte ska bli samma slumptalsföljd varje gång krävs att vi har tillgång till ett tal som med stor sannolikhet inte är det samma körning efter körning. I en dator är nästan allt samma varje gång men en av de få sakerna vi kan utnyttja för att hitta ett slumpfrö är systemklockan. I systembiblioteket `time` finns funktionen `time` som talar om vad klockan är.

```
srand(time(NULL));
```

Följande kodexempel visar hur vi kan använda slumptal för att indexera slumpmässigt valda positioner i en array. Programmet räknar upp den valda positionen med ett och efter 10.000.000 varv i loopen skriver programmet ut värdet på varje position i arrayen.

[1] Multiplicerar vi ett tal i intervallet [0,1) med 52 kommer vi att få tal i intervallet [0,51], alla med samma sannolikhet.

[2] Trunkera = klippa bort de delar som inte får plats. I det här fallet decimalerna.

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

int main()
{
   int i;
   int tal[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
   int summa;

   srand(time(NULL));

   for (i = 0; i < 10000000; i++)
   {
      int slump = (int)((rand() / ((double)RAND_MAX + 1)) * 10);
      tal[slump]++;
   }

   summa = 0;
   for (i = 0; i < 10; i++)
   {
      printf("tal[%d]: %d\n",i,tal[i]);
      summa += tal[i];
   }

   printf("----------------\nTotal: %d\n",summa);
   return EXIT_SUCCESS;
}
```

Om vi har en bra slumpfunktion ska fördelningen mellan platserna i arrayen vara jämn, och en provkörning visar att detta är fallet i C.

```
tal[0]: 998406
tal[1]: 1001061
tal[2]: 1000668
tal[3]: 999001
tal[4]: 1000694
tal[5]: 999921
tal[6]: 998424
tal[7]: 1000289
tal[8]: 1000778
tal[9]: 1000758
----------------
Total: 10000000
```

Om ni väljer att fortsätta programmera i C kommer ni med tiden att bygga upp ert eget bibliotek av små behändiga funktioner. Slumptalsgenereringen är just en sådan funktion och jag hoppas att makrot nedan kan vara till hjälp.

```
/* RND(min, max)
 * Genererar slumptal i intervallet [min,max).
 */
#define RND(min,max) ((int)((rand() / ((double)RAND_MAX + 1)) *       \
                            ((max) - (min))) + (min));
```

Appendix B

Teckentabell

ASCII – American Standard Code for Information Interchange – är en tabell som definierar teckenkoderna för de vanliga tecknen och symbolerna. Den föreslogs på 60-talet av ANSI och har sedan dess varit standard i de flesta datorer. ASCII definierar bara de 128 första tecknen (0 – 127), resten av tabellen är ett tillägg som gjorts senare av ISO. Tabellen nedan följer ISO 8859-1 (Latin-1).

Dec	Hex	Binär	Tecken – Information	Dec	Hex	Binär	Tecken – Information
0	00	00000000	`CTRL-@ (\0)` – NULL (Null prompt)	128	80	10000000	–
1	01	00000001	`CTRL-A` – SOH (Start of heading)	129	81	10000001	–
2	02	00000010	`CTRL-B` – STX (Start of text)	130	82	10000010	–
3	03	00000011	`CTRL-C` – ETX (End of text)	131	83	10000011	–
4	04	00000100	`CTRL-D` – EOT (End of transmission)	132	84	10000100	–
5	05	00000101	`CTRL-E` – ENQ (Enquiry)	133	85	10000101	–
6	06	00000110	`CTRL-F` – ACK (Acknowledge)	134	86	10000110	–
7	07	00000111	`CTRL-G (\a)` – BEL (Bell)	135	87	10000111	–
8	08	00001000	`CTRL-H (\b)` – BS (Backspace)	136	88	10001000	–
9	09	00001001	`CTRL-I (\t)` – HT (Horizontal tab)	137	89	10001001	–
10	0a	00001010	`CTRL-J (\n)` – LF, NL (Linefeed, New line)	138	8a	10001010	–
11	0b	00001011	`CTRL-K (\v)` – VT (Vertical tab)	139	8b	10001011	–
12	0c	00001100	`CTRL-L (\f)` – FF, NP (Formfeed, New page)	140	8c	10001100	–
13	0d	00001101	`CTRL-M (\r)` – CR (Carriage return)	141	8d	10001101	–
14	0e	00001110	`CTRL-N` – SO (Shift out)	142	8e	10001110	–
15	0f	00001111	`CTRL-O` – SI (Shift in)	143	8f	10001111	–
16	10	00010000	`CTRL-P` – DLE (Data link escape)	144	90	10010000	–
17	11	00010001	`CTRL-Q` – DC1 (Device control 1 – XON)	145	91	10010001	–
18	12	00010010	`CTRL-R` – DC2 (Device control 2	146	92	10010010	–
19	13	00010011	`CTRL-S` – DC3 (Device control 3 – XOFF)	147	93	10010011	–
20	14	00010100	`CTRL-T` – DC4 (Device control 4	148	94	10010100	–
21	15	00010101	`CTRL-U` – NAK (Negative aclowledge)	149	95	10010101	–
22	16	00010110	`CTRL-V` – SYN (Synchronous idle)	150	96	10010110	–
23	17	00010111	`CTRL-W` – ETB (End of transmission block)	151	97	10010111	–
24	18	00011000	`CTRL-X` – CAN (Cancel)	152	98	10011000	–
25	19	00011001	`CTRL-Y` – EM (End of medium)	153	99	10011001	–
26	1a	00011010	`CTRL-Z` – SUB (Substitute)	154	9a	10011010	–
27	1b	00011011	`CTRL-[` – ESC (Escape)	155	9b	10011011	–
28	1c	00011100	`CTRL-\` – FS (File separator)	156	9c	10011100	–
29	1d	00011101	`CTRL-]` – GS (Group separator)	157	9d	10011101	–
30	1e	00011110	`CTRL-^` – RS (Record separator)	158	9e	10011110	–
31	1f	00011111	`CTRL-_` – US (Unit separator)	159	9f	10011111	–

Dec	Hex	Binär	Tecken – Information	Dec	Hex	Binär	Tecken – Information
32	20	00100000	– Space	160	a0	10100000	– Non-breaking space
33	21	00100001	! – Exclamation mark	161	a1	10100001	¡ – Inverted exclamation mark
34	22	00100010	" – (Double) Quotation mark	162	a2	10100010	¢ – Cent sign
35	23	00100011	# – Number sign	163	a3	10100011	£ – Pound sterling sign
36	24	00100100	$ – Dollar sign	164	a4	10100100	¤ – General currency sign
37	25	00100101	% – Percent sign	165	a5	10100101	¥ – Yen sign
38	26	00100110	& – Ampersand	166	a6	10100110	¦ – Broken vertical bar
39	27	00100111	' – Apostrophe, Single quote mark	167	a7	10100111	§ – Section sign
40	28	00101000	(– Left parenthesis	168	a8	10101000	¨ – Umlaut
41	29	00101001	) – Right parenthesis	169	a9	10101001	© – Copyright sign
42	2a	00101010	* – Asterisk	170	aa	10101010	ª – Feminine ordinal
43	2b	00101011	+ – Plus sign	171	ab	10101011	« – Left angle quote
44	2c	00101100	, – Comma	172	ac	10101100	¬ – Logical not sign
45	2d	00101101	- – Minus sign, Hyphen	173	ad	10101101	– Soft hyphen
46	2e	00101110	. – Period, Decimal point, Full stop	174	ae	10101110	® – Registered trademark sign
47	2f	00101111	/ – Slash, Virgule, Solidus	175	af	10101111	¯ – Macron accent
48	30	00110000	0 – Digit 0	176	b0	10110000	° – Degree sign
49	31	00110001	1 – Digit 1	177	b1	10110001	± – Plus-or-minus sign
50	32	00110010	2 – Digit 2	178	b2	10110010	² – Superscript 2
51	33	00110011	3 – Digit 3	179	b3	10110011	³ – Superscript 3
52	34	00110100	4 – Digit 4	180	b4	10110100	´ – Acute accent
53	35	00110101	5 – Digit 5	181	b5	10110101	µ – Micro sign
54	36	00110110	6 – Digit 6	182	b6	10110110	¶ – Paragraph sign
55	37	00110111	7 – Digit 7	183	b7	10110111	· – Middle dot
56	38	00111000	8 – Digit 8	184	b8	10111000	¸ – Cedilla
57	39	00111001	9 – Digit 9	185	b9	10111001	¹ – Superscript 1
58	3a	00111010	: – Colon	186	ba	10111010	º – Masculine ordinal
59	3b	00111011	; – Semicolon	187	bb	10111011	» – Right angle qoute
60	3c	00111100	< – Left angle bracket, Less than	188	bc	10111100	¼ – Fraction 1/4
61	3d	00111101	= – Equal sign	189	bd	10111101	½ – Fraction 1/2
62	3e	00111110	> – Right angle bracket, Greater than	190	be	10111110	¾ – Fraction 3/4
63	3f	00111111	? – Question mark	191	bf	10111111	¿ – Inverted question mark
64	40	01000000	@ – Commercial at sign	192	c0	11000000	À – Capital A, grave accent
65	41	01000001	A – Capital A	193	c1	11000001	Á – Capital A, acute accent
66	42	01000010	B – Capital B	194	c2	11000010	Â – Capital A, circumflex
67	43	01000011	C – Capital C	195	c3	11000011	Ã – Capital A, tilde
68	44	01000100	D – Capital D	196	c4	11000100	Ä – Capital A, umlaut
69	45	01000101	E – Capital E	197	c5	11000101	Å – Capital A, ring
70	46	01000110	F – Capital F	198	c6	11000110	Æ – Capital AE ligature
71	47	01000111	G – Capital G	199	c7	11000111	Ç – Capital C, cedilla
72	48	01001000	H – Capital H	200	c8	11001000	È – Capital E, grave accent
73	49	01001001	I – Capital I	201	c9	11001001	É – Capital E, acute accent
74	4a	01001010	J – Capital J	202	ca	11001010	Ê – Capital E, circumflex
75	4b	01001011	K – Capital K	203	cb	11001011	Ë – Capital E, umlaut
76	4c	01001100	L – Capital L	204	cc	11001100	Ì – Capital I, grave accent
77	4d	01001101	M – Capital M	205	cd	11001101	Í – Capital I, acute accent
78	4e	01001110	N – Capital N	206	ce	11001110	Î – Capital I, circumflex
79	4f	01001111	O – Capital O	207	cf	11001111	Ï – Capital I, umlaut
80	50	01010000	P – Capital P	208	d0	11010000	Ð – Capital eth
81	51	01010001	Q – Capital Q	209	d1	11010001	Ñ – Capital N, tilde
82	52	01010010	R – Capital R	210	d2	11010010	Ò – Capital O, grave accent

Dec	Hex	Binär	Tecken – Information	Dec	Hex	Binär	Tecken – Information
83	53	01010011	S – Capital S	211	d3	11010011	Ó – Capital O, acute accent
84	54	01010100	T – Capital T	212	d4	11010100	Ô – Capital O, circumflex
85	55	01010101	U – Capital U	213	d5	11010101	Õ – Capital O, tilde
86	56	01010110	V – Capital V	214	d6	11010110	Ö – Capital O, umlaut
87	57	01010111	W – Capital W	215	d7	11010111	× – Multiplication sign
88	58	01011000	X – Capital X	216	d8	11011000	Ø – Capital O, slash
89	59	01011001	Y – Capital Y	217	d9	11011001	Ù – Capital U, grave accent
90	5a	01011010	Z – Capital Z	218	da	11011010	Ú – Capital U, acute accent
91	5b	01011011	[– Left square bracket	219	db	11011011	Û – Capital U, circumflex
92	5c	01011100	\ – Backslash, Reverse solidus	220	dc	11011100	Ü – Capital U, umlaut
93	5d	01011101	] – Right square bracket	221	dd	11011101	Ý – Capital Y, acute accent
94	5e	01011110	^ – Circumflex accent	222	de	11011110	Þ – Capital thorn
95	5f	01011111	_ – Underscore	223	df	11011111	ß – Small sz ligature
96	60	01100000	‘ – Grave accent, Back apostrophe	224	e0	11100000	à – Small a, grave accent
97	61	01100001	a – Small a	225	e1	11100001	á – Small a, acute accent
98	62	01100010	b – Small b	226	e2	11100010	â – Small a, circumflex
99	63	01100011	c – Small c	227	e3	11100011	ã – Small a, tilde
100	64	01100100	d – Small d	228	e4	11100100	ä – Small a, umlaut
101	65	01100101	e – Small e	229	e5	11100101	å – Small a, ring
102	66	01100110	f – Small f	230	e6	11100110	æ – Small ae ligature
103	67	01100111	g – Small g	231	e7	11100111	ç – Small c, cedilla
104	68	01101000	h – Small h	232	e8	11101000	è – Small e, grave accent
105	69	01101001	i – Small i	233	e9	11101001	é – Small e, acute accent
106	6a	01101010	j – Small j	234	ea	11101010	ê – Small e, circumflex
107	6b	01101011	k – Small k	235	eb	11101011	ë – Small e, umlaut
108	6c	01101100	l – Small l	236	ec	11101100	ì – Small i, grave accent
109	6d	01101101	m – Small m	237	ed	11101101	í – Small i, acute accent
110	6e	01101110	n – Small n	238	ee	11101110	î – Small i, circumflex
111	6f	01101111	o – Small o	239	ef	11101111	ï – Small i, umlaut
112	70	01110000	p – Small p	240	f0	11110000	ð – Small eth
113	71	01110001	q – Small q	241	f1	11110001	ñ – Small n, tilde
114	72	01110010	r – Small r	242	f2	11110010	ò – Small o, grave accent
115	73	01110011	s – Small s	243	f3	11110011	ó – Small o, acute accent
116	74	01110100	t – Small t	244	f4	11110100	ô – Small o, circumflex
117	75	01110101	u – Small u	245	f5	11110101	õ – Small o, tilde
118	76	01110110	v – Small v	246	f6	11110110	ö – Small o, umlaut
119	77	01110111	w – Small w	247	f7	11110111	÷ – Division sign
120	78	01111000	x – Small x	248	f8	11111000	ø – Small o, slash
121	79	01111001	y – Small y	249	f9	11111001	ù – Small u, grave accent
122	7a	01111010	z – Small z	250	fa	11111010	ú – Small u, acute accent
123	7b	01111011	{ – Left brace (curly bracket)	251	fb	11111011	û – Small u, circumflex
124	7c	01111100	| – Vertical bar	252	fc	11111100	ü – Small u, umlaut
125	7d	01111101	} – Right brace (curly bracket)	253	fd	11111101	ý – Small y, acute accent
126	7e	01111110	~ – Tilde accent	254	fe	11111110	þ – Small thorn
127	7f	01111111	– DEL (Delete)	255	ff	11111111	ÿ – Small y, umlaut

Appendix C

Escape-sekvenser / ANSI-koder

ANSI – American National Standards Institute satte för många år sedan ihop en lista över koder för att styra terminaler. Tanken var att en server skulle kunna styra terminalen hos de klienter som kopplade upp sig mot den. Detta sätt att styra fjärr-terminaler blev mycket populärt och de flesta terminaler blev ANSI-kompatibla.

Varje kod är en sekvens av tecken som följer ett speciellt mönster. Alla koder inleds med teckenkoden för escape och koderna kallas därför ibland för escape-koder eller escape-sekvenser. Koderna kan användas till det mesta som behövs för att ha kontroll över en terminal. Man kan flytta markören, ändra färger, rulla texten upp eller ner, rensa skärmen och mycket mycket mer.

Ni kommer att stöta på ANSI-koderna i Klondike. Även om koderna på sätt och vis har spelat ut sin roll nu när allt styrs via grafiska gränssnitt så tycker jag att det tillhör allmänbildning för en programmerare att veta hur dessa koder används.

Koderna är som sagt en sekvens av tecken och vad man gör är helt enkelt att man skriver ut dessa tecken i terminalfönstret. Koderna styr terminalen och förändrar dess tillstånd. Dessa förändringar ligger helt i terminalen och har inget med ert program att göra efter att de skickats. Om man till exempel sätter textfärgen till röd så kommer den att fortsätta vara röd tills dess att man säger något annat – även om man avslutar programmet.

En förteckning över de vanligaste koderna finns på kurshemsidan.