En kurs i C-programmering

Föreläsningsanteckningar från AD1/PK2 VT07

Jesper Wilhelmsson

1 mars 2007

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 samtidigt som ni läser.

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. När det gäller problemlösning i allmänhet och programmering i synnerhet är det viktigt att behärska konsten att abstrahera – att kunna bortse från detaljer som rör det specifika problemet. När man bortser från detaljerna kan man jämföra problemet man vill lösa med andra problem man redan löst och se likheter. Genom att utnyttja dessa likheter kan man 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 problemen i mindre delar som var för sig är betydligt enklare att lösa. Vi kommer under denna kurs att 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 och är en vidareutveckling av språket B. B utvecklades runt 1970 för att skriva program till den första Unix:en på en DEC PDP-7. Språket C såg dagens ljus 1972 (på DEC PDP-11). Språket spreds snabbt och många olika dialekter av C växte fram. 1978 gav Kernighan och Ritchie ut den första versionen av "The C programming language". Den version av C som beskrevs i boken kom att bli dominerande och benämns sedermera K&R C. ANSI C (1989) är en utveckling av K&R C där man filat bort en del av de kantiga hörnen och specificerat ett brett funktionsbibliotek. ANSI C (kallas även C89) blev den första C-versionen som (med några minimala ändringar) formaliserades som en ISO-standard (1990), känd under namnet C90. Därefter har utvecklingen av språket fortsatt och två nya versioner har sett dagens ljus; C95 och C99. C95 innehöll egentligen bara ett utökat stöd för internationella tecken, så det är först med C99 som språket fått en ordentlig uppgradering.

C99 erbjuder flera mycket intressanta nyheter. Tyvärr är stödet för C99 fortfarande ganska dåligt i de flesta stora C-kompilatorerna. `gcc` är den som har bäst stöd, men inte heller `gcc` är komplett. Andra stora kompilatortillverkare som Microsoft och Borland ligger mycket sämre till i sitt stöd för C99 (eftersom de gjort det tvivelaktiga valet att fokusera på C++ istället). Detta innebär att man i princip tvingas följa ANSI C om man vill vara säker på att koden man skriver

ska fungera i alla stora C-kompilatorer. Vi kommer därför att hålla oss till ANSI C i denna kurs. Några av tilläggen i C99 är ganska väl spridda och väldigt användbara, så vi kommer att se lite av dem också, men då talar jag om att det inte längre är ANSI C vi håller på med.

C är som sagt i grunden utvecklat för att skriva operativsystem och systemnära program och det ligger därför mycket nära hårdvaran. Man pratar ibland om att en del programmeringsspråk är högnivåspråk och att andra är lågnivåspråk. Bland högnivåspråken hittar vi till exempel Standard ML och Java. Dessa ligger på en hög nivå jämfört med hårdvaran vilket innebär att det finns ett tjockt lager av andra program som ligger mellan det program man själv skriver och hårdvaran som till slut ska köra koden. Låg nivå betyder att språket ligger nära hårdvaran och ett av de lägsta språken på denna skala är assembler. C ligger ett litet steg högre upp i skalan än assembler och är alltså ett lågnivåspråk. Detta märker man till exempel på att det inte finns några skyddsmekanismer, om programmeraren gör fel så dör programmet på riktigt och har man otur så tar det andra delar av operativsystemet med sig i fallet.

Trots att C ligger så nära hårdvaran så är ANSI C plattformsoberoende. Så länge man inte ger sig på riktigt håriga saker i minnet, så kan man flytta ett program från hårdvara till hårdvara från operativsystem till operativsystem utan att behöva ändra någonting i källkoden. Linux är ett lysande exempel på detta. Linux självt, och de flesta av de program som används i Linux är skrivna i C, och alla dessa kan kompileras på vilken hårdvara som helst och därför kan vi hitta linux på i stort sett alla plattformar som existerar – allt från "vanliga" datorer med PPC- och x86-processorer till mobiltelefoner och rymdfärjor. Programmen kan även flyttas till andra operativsystem utan att källkoden behöver förändras nämnvärt[1]. Det enda man behöver göra är att kompilera om programmet.

Kompilera ja... För att köra ett C-program måste man först 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[2]. C är inte ett interpreterande språk som till exempel Standard ML och BASIC är. Det kan vara värt att notera 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 vilket annat program som helst under det givna operativsystemet. 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.

När man kör sitt C-program säger man att datorn (eller programkörningen) befinner sig i ett tillstånd och varje sats i programmet förändrar detta tillstånd till exempel genom att flytta eller ändra värden i minnet. Dessa förändringar av ett tillstånd kallas ibland för sidoeffekter, och dessa kan även vara saker som att skriva ut saker till en skärm, 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, har man inget tillstånd som uppdateras kontinuerligt. Funktionella program är inte sekvenser av satser, utan byggs helt upp kring funktionsanrop.

[1] Cygwin för Windows är ett exempel på hur Linux-program flyttats till ett annat operativsystem, Geek Gadgets för AmigaOS ett annat.

[2] Verktyget `gdb` erbjuder bland annat möjligheten att provköra enstaka funktioner. `gdb` är ett mycket kraftfullt verktyg för felsökning och används för att hitta fel i C-program. Läs mer om `gdb` i appendix F.1

1.2 Inlämningsuppgiften — vårt stående exempel

Som jag redan sagt så är det enklaste sättet att lära sig programmera att testa själv och studera olika exempel. Exemplet som jag kommer att återkomma till under hela denna kurs är ett enkelt litet kortspel som heter Klondike. Delar av källkoden till spelet får ni given och andra delar av spelet kommer ni att få skriva själva i de fyra inlämningsuppgifterna under kursens gång. De flesta kodexempel i detta kompendium är hämtade från spelet och jag rekommenderar att ni själva letar upp koden som beskrivs i de olika exemplen och ändrar lite för att se vad som händer.

På kurshemsidan finns filerna `klondike.c`, `ansi.h`, `COURSE_cardpile.o` och `cardpile.h` – ladda hem dessa filer redan nu, jag kommer att referera till dem senare. Tillsammans bildar filerna det färdiga spelet. Filen `klondike.c` innehåller själva ramen för spelet. Här finns reglerna för hur spelet går till, utskrifter av menyer och liknande. `ansi.h` innehåller ett antal styrkoder för att hjälpa till vid utskrifter. Här finns bland annat koder för att ändra färg och flytta markören i terminalfönstret. Spelet använder sig av ett separat paket för att hantera spelkort. Filen `cardpile.h` innehåller det publika gränssnittet för detta paket, ett antal deklarationer av funktioner som hanterar högar av spelkort. Den tillhörande c-filen (`cardpile.c`) som ska innehålla själva implementationen av dessa funktioner finns dock inte ännu.

Er uppgift under denna kurs är att skriva filen `cardpile.c`. Observera att de delar ni ska skriva redan finns implementerade i `COURSE_cardpile.o`. Detta är inte en källkodsfil utan en objektfil[3]. Objektfilen är ett mellansteg i kompileringen av programmet. I den finns färdigkompilerade versioner av funktionerna ni ska skriva och `gcc` kan knyta ihop dessa med koden i `klondike.c` under slutsteget i kompileringen[4]. Den givna objektfilen gör att ni redan från början har en fullt fungerande version av spelet och tanken är att ni successivt byter ut olika delar av programmet tills dess att ni har skrivit alla funktioner som `cardpile.h` deklarerar. Exakt hur det går till att kompilera filerna beskrivs i sektion 1.9.

Titta igenom spelet i filen `klondike.c` innan ni börjar så att ni får en inblick i hur koden ni ska skriva kommer att användas. Observera speciellt att det inte förekommer några magiska konstanter eller antaganden som är implementationsberoende. Koden är dessutom välkommenterad och snyggt[5] strukturerad. All er kod ska skrivas på samma sätt! Kommentarerna i koden innehåller mycket viktig information om vad ni får förutsätta och vad ni inte får förutsätta när ni arbetar med koden. Läs även igenom `cardpile.h` så att ni vet hur strukturerna ni ska jobba med ser ut. Här finns även tips på hur ni kan testa er kod för att upptäcka vissa besvärliga fel.

Alla funktioner som deklareras i `cardpile.h` har namn som inleds med `COURSE_`. Samtliga dessa ska under kursens gång ändras så att de deklarerar er version av samma funktion. Er funktion ska ha samma namn som den existerande fast utan prefixet `COURSE_`. När ni har skrivit er egen funktion tar ni alltså bort prefixet från `cardpile.h` och anropsställena i `klondike.c` och testar att programmet fortfarande fungerar med er kod.

För att hitta alla anropsställen kan `grep` vara till hjälp. `grep` är ett unix-kommando som hittar strängar i en textfil. Inför första uppgiften kan ni prova följande i ett terminalfönster (i samma katalog där ni laddat hem källkodsfilerna):

```
grep COURSE_colormatch klondike.c
      COURSE_colormatch(COURSE_getsuit(hi), COURSE_getsuit(low)))
```

Resultatet ni får tillbaka, den andra raden, är en rad ur filen `klondike.c` där `grep` hittat texten `COURSE_colormatch`. När ni är klara med uppgift ett ska `grep` inte längre kunna hitta denna rad eftersom ni då ska ha bytt ut anropet mot ett till er egen funktion utan prefixet. När alla fyra inlämningsuppgifter är klara ska anropet "`grep COURSE klondike.c`" inte hitta en enda förekomst av prefixet `COURSE` i filen `klondike.c` (och inte i `cardpile.h` heller förstås).

[3] I själva verket finns det flera objektfiler kompilerade för olika arkitekturer, ni ska välja den som matchar den dator ni jobbar på.

[4] Att knyta ihop de olika funktionerna kallas länkning och är egentligen inte en del av själva kompileringen utan sker efteråt, men `gcc` gör allt under ytan så det behöver man inte veta.

[5] Enligt min högst personliga åsikt om vad som är snyggt förtsås.

Om att lösa och lämna in uppgifter

Ni måste följa de deklarationer som finns i `h`-filen eftersom Klondike redan använder dessa funktioner. Ni kan ändra i `h`-filerna och `klondike.c` i felsökningssyfte, men eftersom ni endast förväntas lämna in filen `cardpile.c` måste ni försäkra er om att er kod fungerar med omodifierade versioner av de givna filerna. Det enda som ni förutsätts ändra är namnen på de funktioner ni implementerar.

Alla uppgifter löses i grupper om exakt två personer. Uppgifterna ska lösas tillsammans, det är inte meningen att ni ska dela upp jobbet i mindre delar och lösa var sin del på egen hand. Alla gruppmedlemmar ska vara beredda att svara på frågor om och redogöra för hela lösningen.

På inlämningsuppgifterna lämnas något av betygen godkänd (G), komplettering (K) eller underkänd (U). Komplettering innebär att lösningen inte är tillräckligt bra utan måste förbättras och lämnas in för ny betygsättning senast vid stopptiden för nästa inlämningsuppgift. Underkänt betyg får man om man inte gjort ett seriöst försök att genomföra uppgiften och lämnat in innan stopptiden eller om eventuell komplettering inte är tillräckligt bra. (Och förstås om man fuskat.)

För att en inlämning ska räknas som ett "seriöst försök" ska följande kriterier vara uppfyllda:

- Programmet ska vara ämnat att lösa det problem som uppgiften beskriver.
- Källkoden ska passera kompilatorn (gcc) med de flaggor som uppgiften kräver (se sektion 1.9) utan anmärkningar.
- Kända fel och brister i inlämningen ska vara dokumenterade i källkoden.

Man kan alltså bli godkänd på en inlämningsuppgift på ett av två sätt:

- Genom att lämna in en tillräckligt bra lösning i tid.
- Genom att lämna in ett "seriöst försök" till lösning i tid och dessutom lämna in en tillräckligt bra lösning ("komplettering") senast vid nästa inlämningstillfälle.

Det finns en kodningsstandard på kurshemsidan för hur lösningarna skall vara utformade. Om inte denna standard följs kan resultatet bli komplettering eller underkännande utan att jag ens tittar på vad lösningen faktiskt gör.

Ni uppmuntras att diskutera era lösningar mellan grupperna, men det betraktas som fusk om flera grupper implementerar lösningarna tillsammans. Det är även att betrakta som fusk att disassemblera den givna objektfilen, även om jag betvivlar att det skulle ge någon information som underlättar lösandet av uppgifterna.

Testa er kod!

Innan ni lämnar in en källkodsfil förutsätts det att ni har testat att er kod verkligen fungerar och gör det den är avsedd att göra. Ett sätt att testa är genom att provspela Klondike med er egen kod för att se om det fungerar. Detta är troligen det minst effektiva sättet att hitta fel i er kod och ni kommer sannolikt inte att testa olika specialfall som bara inträffar under särskilda förhållanden.

Ett vanligt fel man gör när man testar sina program är att man är för snäll. Man utsätter inte programmet för särskilt brutala påhopp utan följer de regler man vet finns. Dessutom vet man oftast hur man ska hantera programmet och gör därför sällan de nybörjarfel som andra kanske gör – som att placera spelkort där de inte får ligga och så vidare. Utöver detta finns det dessutom fel som endast uppstår i specialfall, till exempel om ett rött kort läggs på en tom hög eller då man försöker placera en kung ovanpå ett ess. Detta gör att man inte hittar särskilt många fel då man provspelar sitt spel och man vaggas in i en falsk trygghet om att allt faktiskt fungerar. Vill ni testa ert program på riktigt så ska ni vara så elaka som ni bara kan. Det bästa är om ni byter program med en annan grupp och gör ert bästa för att få deras program att sluta fungera. Använd er fantasi och gör ändringar i `cardpile.h` och `klondike.c` för att få korthögarna att gå sönder!

Oavsett hur elaka ni är så kommer ni inte att kunna stöta på alla de där specialfallen som endast inträffar den tredje onsdagen i månaden när månen är full. För att hitta dessa buggar måste ni skriva egna små testprogram som hårdtestar varje funktion för sig och utsätter den för alla tänkbara korrekta och felaktiga indata.

Ett annat mycket bra sätt att testa er kod är att ändra förutsättningarna för koden. Detta kan göras på många sätt. Ett enkelt sätt är att ändra definitionerna i `cardpile.h` så att `EMPTY` inte längre är `NULL` utan något annat, oväntat värde (till exempel `((void*)4711)`). Man kan även ändra så att `ACE` inte längre är `0` utan kanske `42` eller `-7`, och prova att ändra definitionen av `suit_t`. Ett annat sätt att ändra förutsättningarna är genom att kompilera och testköra programmet på en annan hårdvara. Till exempel skiljer sig Sparc och x86 ganska radikalt åt i sin uppbyggnad, så om ett program fungerar på båda dessa plattformar kan man känna sig lite lugnare – speciellt när det gäller pekare och minneshanteringskod.

Självklart kommer jag att göra allt som står i min makt för att få era program att bryta ihop. Jag kommer att testa med olika värden på alla konstanter i `cardpile.h`, jag kommer att hårdtesta samtliga funktioner var för sig och jag kommer att kompilera och testa ert program både på Sparc (Solaris) och x86 (linux och Solaris). Jag förväntar mig att era program håller för detta.

VIKTIGT!! Gör aldrig sista-minuten-ändringar utan att kompilera om! Ett mycket onödigt men dock ganska vanligt fel är att man i sista stund ändrar något litet i koden och sedan lämnar in. Till exempel lägger man in en kommentar i koden eller tar bort någon utskrift. Självklart missar man att avsluta kommentaren eller så råkar man ta bort ett semikolon någonstans och programmet går då inte längre att kompilera. En uppgift som inte går att kompilera kommer inte att rättas, oavsett orsaken till att den inte går att kompilera. Varje år är det åtminstone några grupper som får K på en helt perfekt uppgift enbart på grund av detta. Bli inte en av dem!

Ta för vana att alltid kompilera *filen ni lämnar in.*

1.3 Variabler, Typer, Konstanter

Nu är det dags att börja titta på hur C ser ut egentligen. Ni har redan sett hur det kan se ut när ni tittade i `klondike.c`, men utan en närmare förklaring är det nog inte så lätt att förstå vad som händer där egentligen. Det första vi ska titta på är något som kallas för *variabel*. 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 i ANSI C är standardtypen `int` (heltal). I C99 är det inte längre tillåtet att utelämna typdeklarationen[6] och under denna kurs får ni inte heller utelämna typen utan alla variabeldeklarationer ska inkludera en typ. De inbyggda typer som C tillhandahåller visas i tabell 1.1.

En typisk variabeldeklaration: `int x;`

Olika datatyper kan representera olika stora mängder data. Ett normalt heltal (`int`) representerar till exempel en större mängd data än ett tecken (`char`). Detta innebär att olika typer kräver olika mycket minne. Det krävs mer minne för att lagra ett riktigt stort tal än det krävs för att lagra ett litet. 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 `int` just 32 bitar (= 4 bytes).

[6] Utöver denna inskränkning så är all ANSI C-kod kompatibel med C99.

Typdeklaration	sizeof	Kommentar
`char`	1	Heltal. Alltid en byte
`short int`	2	Heltal. 16 bitar = -32.768 — 32.767
`int`	4	Heltal. Vanligen ett maskinord stor
`long int`	4	Heltal. 32 bitar = $-2.147.483.648$ — 2.147.483.647
`long long int`	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 eller för riktigt hög precision

Tabell 1.1: Datatyper och dess storlek i minnet på en 32-bitars-maskin. `long long` introducerades i C99.

64-bitars-maskiner börjar bli vanligare och i dessa är normalt en `int` 64 bitar stor och kan alltså lagra betydligt större tal än på en 32-bitars-maskin.

Med nyckelorden `long` och `short` kan man tala om lite mer exakt hur stor man vill att en variabel ska vara. Man skriver ett av orden framför `int` vid sin deklaration. Man kan även utelämna `int` och bara skriva `long` eller `short`, det blir samma sak som om man skriver `int` där efter.

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.

Som ni märker så är allt inte helt plattformsoberoende ändå, även om man gärna vill ge det intrycket. Vill man veta de exakta gränserna finns det konstanter definierade i standardbiblioteket `limits` som anger dessa. Där hittar vi till exempel `INT_MAX` och `INT_MIN`. Notera att dessa gränser alltså varierar mellan olika plattformar – men man kan i alla fall ta reda på dem i sitt program och anpassa sig därefter. För heltalstyper med garanterad bit-storlek definierar ISO C biblioteket `inttypes` där vi kan hitta typer som `int8_t`, `uint8_t`, `int16_t` och så vidare. Detta bibliotek är dock inte en del av ANSI C så jag vill inte se några av de typerna i inlämningsuppgifterna.

Flyttal är om möjligt än mer hårdvaruberoende än heltalen. Som synes i tabellen är det ganska vaga beskrivningar om typernas storlekar. Om det är riktigt viktigt med flyttalens största och minsta värden, precision med mera kan det vara en bra idé att titta i standardbiblioteket `float` på den maskin man tänker köra på.

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). För att deklarera en pekare lägger man till en `*` efter typen: `int* x;` Vi återkommer till pekare i sektion 3.2, här vill jag bara tala om att de finns.

Utöver dessa typer finns det även något som kallas för *arrayer*. En array är en samling av variabler av samma typ. Att deklarera en array av någon typ görs på exakt samma sätt som att deklarera en enkel variabel. Man sätter bara `[]` efter namnet och i dessa anger man storleken på arrayen – alltså hur många variabler, eller element som man också kallar det, den ska innehålla.

En array med plats för tio heltal: `int siffror[10];`

Tilldelning

En variabel får sitt värde genom en tilldelning, detta skrivs med likhetstecken. Man kan lägga flera tilldelningar i rad om man har flera variabler som ska ha samma värde. Detta är möjligt eftersom tilldelningen kan betraktas som ett uttryck. Resultatet av uttrycket är det tilldelade värdet.

```
x = 42;
x = y = z = 4711;
```

Om man inte initierar[7] värdet på sina variabler är deras innehåll 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.

Variabler som ligger i en array kommer man åt genom att ange deras *index* i arrayen. Det första elementet i arrayen har index 0. För övrigt behandlar man dem som helt vanliga variabler: `siffror[3] = 42;` Vi kommer att behandla arrayer betydligt mer i sektion 3.5.

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.

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

Detta går bra eftersom en `short int` är mindre än en `long int` 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)
```

Notera att vi skriver talen på engelska, vi använder alltså punkt istället för komma för att separera heltalsdelen från decimaldelen i flyttal.

Konstanter

Man kan 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. Det rätta synsättet på en `const`-deklarerad variabel är egentligen inte att den är en konstant utan snarare en variabel som man bara kan läsa ifrån. 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;
```

Man kan deklarera flera variabler av samma typ på en gång genom att sätta komma mellan variabelnamnen. Följande variabeldeklarationer är hämtade från början av funktionen `main` i

[7] Initiera = Ge variabeln ett första värde.

`klondike.c`. Här deklareras heltalsvariabler, pekare och arrayer. Konstanterna PILES och STORAGES har fått sina värden på annat håll och typen `card_t` återkommer vi till senare, den är inte en del av ANSI C.

```
int i, j;
card_t* deck;
card_t* pile[PILES];
card_t* storage[STORAGES];
int cursor = 0;
char ch;
unsigned int seed;
```

1.4 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 (`!=`), mindre eller lika med (`<=`) 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. I ANSI C finns det ingen speciell typ för sanningsvärden, utan detta behandlas som ett vanligt tal precis som allt annat. Falskt = 0 och sant = 1. Egentligen betraktar C allt annat än 0 som sant, men det är 1 som returneras av de logiska operatorerna. ISO C däremot definierar ett bibliotek som heter `stdbool`. Där hittar vi deklarationer av `bool`, `true` och `false`. `stdbool` är en relativt välspridd utökning av ANSI C och att utnyttja dessa typer ökar läsbarheten i koden avsevärt. Därför kommer vi i inlämningsuppgiften att använda `stdbool`.

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 minst 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. Operatorn som utför ökningen med ett är `++`. De två sätten att använda denna skiljer sig på en viktig punkt – när själva ökningen av värdet sker.

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

I det första fallet (till vänster) står `++` efter variabelnamnet. Då kommer värdet på `x` att användas *innan* det ökas med ett, det vill säga `y` kommer att få värdet 5. I fallet till höger står `++` före variabelnamnet och värdet på `x` kommer då att ökas först, 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. Satserna 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.

1.5 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;
```

Villkoret är ett sanningsvärde eller rättare sagt ett uttryck som resulterar i ett sanningsvärde. Det kan vara allt från en enkel jämförelse till ett funktionsanrop. Eftersom C betraktar sanningsvärden som vilka tal som helst så kan man även använda uttryck som resulterar i något godtyckligt heltal. `0` kommer att betraktas som falskt och allt annat som sant. I denna kurs kommer vi dock enbart att använda riktiga sanningsvärden.

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 ( villkor )
{
   sats1;
   sats2;
}
else
   sats3;
```

Koden som skrivs i satserna kan vara precis vilken C-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;
```

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 beräknas. 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 ett värde med en variabel `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 frågan om `x` är skiljt från noll i det första exemplet kommer att utföras innan divisionen. Om `x` är noll kommer divisionen aldrig att utföras eftersom villkoret då blir falskt och att beräkna den andra delen av villkoret är onödigt – oavsett vad resultatet blir kommer villkoret i slutändan ändå att vara falskt. 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 viktigt att tänka på i vilken ordning man sätter uttrycken. Även i fall då uttrycken egentligen är oberoende av varandra kan det vara värt att tänka en gång till. Om någon del av villkoret är betydligt dyrare att beräkna än någon annat gör man bäst i att sätta det dyraste uttrycket sist och har man tur behöver det inte beräknas 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` kan endast användas på heltalsvariabler, det vill säga `int`, `char` och liknande.

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

Om värdet på `variabel` är `värde1` kommer `sats1` att utföras, om värdet är `värde2` körs `sats2` 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.

En något längre version av följande `switch` hittar vi i funktionen `main` i `klondike.c`. Här läser vi in ett kommando från användaren och beroende på vad som skrivits in ska olika saker ske i spelet. Om användaren skriver in något som vi inte känner till hamnar vi i `default:` där vi ignorerar vad som skrivits och läser in ett nytt kommando.

```
ch = (char)getchar();
switch (ch)
{
   case CHAR_UP: cursor -= (int)(cursor != 0); break;
   case CHAR_DOWN: cursor += (int)(cursor != PILES - 1); break;
   case CHAR_CARDS: newcards(piles, &deck); break;
   default: ch = (char)getchar();
}
```

Kommandot `break` används för att tala om att det är dags att hoppa vidare 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. I exemplet nedan kommer fallet `'a'` att direkt hoppa vidare och köra koden för fallet `'A'`, medan fallet `'A'`

endast kommer att köra sin egen kod och sedan hoppa ut ur `switch`-satsen. Fallet `'c'` kommer att köra `create_new_thing()` och sedan fortsätta med koden för nästa fall, `'m'`, och köra även den. Fallet `'m'` avslutas inte med någon `break` men eftersom det är det sista fallet gör det ingen skillnad.

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

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 flera `if`–`else` kan det ju vara något fall som beror av någon annan variabel och det är ofta svårt att se.

1.6 Loopar — `for`, `while`, `do`

För att kunna skriva lite mer avancerade program krävs något sätt att iterera (upprepa). C tillhandahåller tre typer av loopar: `for`, `while` och `do`. Det finns även ett fjärde sätt, rekursion. Rekursion betyder att en funktion anropar sig själv och på det viset åstadkommer en loop. Detta är inte att rekommendera i C. Varje gång en funktion anropas i C kommer det att gå åt lite av 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.

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 `for`-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.

I funktionen `main` i `klondike.c` hittar vi några sådana exempel. Detta är innan spelet har börjat och vi ska stega igenom alla korthögar i spelet och se till att de är tomma. `PILES` och

`STORAGES` är två konstanter som talar om hur många högar med kort vi har i spelet (spelhögar respektive lagerlokaler). `EMPTY` är även det en konstant som representerar en tom korthög. `pile` och `storage` är arrayer som håller ordning på de olika högarna.

```
for (i = 0; i < PILES; i++)
   pile[i] = EMPTY;

for (i = 0; i < STORAGES; i++)
   storage[i] = EMPTY;
```

`while ( villkor ) sats;`

Precis som `for`-loopen så fortsätter `while`-loopen att iterera tills dess att villkoret inte längre är uppfyllt. Skillnaden mellan dessa två loopar är att `while` inte har någon inbyggd initiering eller någon definierad förändringskod. 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 den information som villkoret bygger på. Detta resulterar ofta i en oändlig loop – man säger att programmet hänger sig.

I funktionen `main` i `klondike.c` hittar vi ett exempel på när det är lämpligt att använda en `while`-loop. Användaren matar in kommandon för att styra spelet och vi vill snurra i loopen tills dess att användaren trycker in kommandot för att avsluta spelet.

```
ch = (char)getchar();
while (ch != CHAR_QUIT)
{
   switch (ch)
   {
      case CHAR_UP: cursor -= (int)(cursor != 0); break;
      case CHAR_DOWN: cursor += (int)(cursor != PILES - 1); break;
   }
   ch = (char)getchar();
}
```

Vi läser in ett kommando med `getchar` och sparar det i variabeln `ch`. Så länge denna variabel inte innehåller kommandot för att avsluta snurrar vi ett varv till. Sist i loopen läser vi in ett nytt kommando och uppdaterar variabeln. `switch`-satsen som tar hand om kommandot inne i loopen har vi redan sett en gång.

`for` och `while` har som sagt 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`-loopen. `do` kör loop-kroppen en gång först innan villkoret kontrolleras. För övrigt påminner den mycket om en `while`-loop.

`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 tillbaka till villkoret. Om villkoret är uppfyllt fortsätter loopen att snurra och ett nytt varv påbörjas vid loop-kroppens början.

1.7 Inmatning och utmatning — stdin, stdout, stderr

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 biblioteksanrop. Standardbiblioteket `stdio` innehåller en mängd olika funktioner för I/O.

För att få tillgång till funktionerna i standardbiblioteken måste vi inkludera en deklarationsfil. Varje bibliotek har sin egen fil och namnet på den är det samma som namnet på biblioteket plus filändelsen `.h` (header). Inkluderar gör vi med `#include`. Notera att det inte är något semikolon sist på raden.

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

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. När man startar ett C-program skapas tre strömmar som binds till programmet, `stdin`, `stdout` och `stderr`[8]. De tre namnen deklareras i `stdio`.

Inmatning — 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.h`.

`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. Vi har redan sett ett par exempel på hur `getchar` används i `klondike.c` när man läser in kommandon från användaren.

Utmatning -- stdout

Nästa ström vi ska titta på heter `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 `putchar`. "`putchar('A');`" kommer att skicka bokstaven A till `stdout`.

Att skriva ut allting tecken för tecken kan bli lite jobbigt. Lyckligtvis finns det ganska många olika funktioner för utskrifter. Den vi kommer att se mest av under denna kurs heter `printf`.

```
printf("Fortran föddes 1954\n");
```

Anropet ovan kommer att skriva ut texten "`Fortran föddes 1954`" i terminalfönstret. `\n` är tecknet för radbrytning och gör alltså att nästa utskrift kommer att hamna i början av nästa rad. `printf` kan göra mer avancerade saker än att bara skriva ut en ren text. Texten man skickar in kallas formatsträng och kan innehålla ett antal koder som byts ut mot olika värden. Värdena skickas in som argument till `printf` tillsammans med formatsträngen och dessa kombineras sedan ihop till den textsträng som skickas till `stdout`.

```
printf("%c%c %s %d", 'M', 'L', "föddes", 1973);
```

Raden ovan kommer att skriva ut texten "ML föddes 1973". Här används koderna `%c` för tecken, `%s` för en hel textsträng och `%d` för ett heltal. Värdena som ska skrivas ut på dessa platser skickar vi in efter formatsträngen. I det här exemplet står värdena som konstanter i koden.

[8] Std som förekommer i flera av namnen här är en förkortning av "standard". Namnen utläses alltså standard i/o, standard in, standard out och standard error.

Tecken skrivs med ' runt om ('M') och strängar med citattecken ("föddes"). Vanligast är det nog dock att värdena man vill skriva ut ligger i variabler och att det är dessa man då skickar in som argument. Det finns flera exempel på utskrifter i Klondike. Utskriften nedan hittar vi i funktionen `printinfo` som skriver ut information om spelet.

```
printf("Klondike 2007    Slumpfrö: %d\n", seed);
```

En intressant egenskap som ANSI C erbjuder är att slå samman strängar om man skriver dem efter varandra. `printf("BASIC " "föddes " "1964");` blir samma sak som `printf("BASIC föddes 1964");`. Det finns flera goda skäl att använda detta sätt att dela strängar. Ett exempel är att man normalt för ökad läsbarhet vill undvika att ha rader som är längre än 80 tecken i sin källkodsfil. För att hålla raderna under denna gräns kan man alltså dela långa strängar på flera rader. Ett par andra mycket användbara användningsområden har med makron att göra och de återkommer vi till i föreläsning 5.

Felutskrifter -- stderr

Om något går fel i programmet bör man informera användaren om det. Detta görs via en speciell ström, `stderr`. Den fungerar på exakt samma sätt som `stdout` och man använder i stort sett samma funktioner för att skriva till dem. Skillnaden mellan strömmarna är att användaren kan välja att hantera dem på olika sätt. Man kan till exempel tänka sig att man skickar vanliga utskrifter till terminalfönstret och felutskrifter till en loggfil. `printf` som vi använde ovan skriver alltid till `stdout`, men det finns en snarlik funktion som tar som argument vilken ström den ska skriva till, `fprintf`.

```
fprintf(stderr, "Nu gick det åt skogen på rad %d i filen %s\n",
        __LINE__, __FILE__);
```

`__FILE__` och `__LINE__` som används här är konstanter som får sina värden av `gcc` innan kompileringen. De kommer att innehålla radnumret och filnamnet där raden står i källkodsfilen.

Utöver dessa strömmar kan man även skapa egna strömmar för 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 koder för olika typer av data förtecknade.

1.8 Kommentarer i koden — `/* ... */`

Skriver man källkod så ska man skriva kommentarer i den. Så är det bara. Speciellt i ett lågnivåspråk som C där det går att göra mycket magiskt med några oläsliga rader kod. Nu förespråkar jag inte att ni ska skriva oläslig kod, men även om man gör allt som står i ens makt för att skriva snygg C-kod så kommer man att hamna i situationer där det inte är så lätt att se direkt på koden vad den gör.

I C inleder man kommentarer med `/*` och avslutar dem med `*/`. Kommentarer kan sträcka sig över flera rader och innehålla godtycklig text eller programkod. Det enda man inte kan ha i en kommentar är en annan kommentar.

Ibland är det svårt att veta på vilken nivå man ska lägga sig när man kommenterar koden. Ska man skriva kommentarer på varje rad eller räcker det med en i början av filen?

Det enda jag kan svara på det är att kommentarer ska vara meningsfulla. Det är oftast inte intressant att tala om exakt vad koden gör, däremot är det oerhört intressant att tala om varför den gör som den gör. För att ta ett drastiskt exempel kan vi tänka oss följande rad i ett program.

```
x++;  /* Öka värdet på x med ett. */
```

Hur mycket man behöver skriva beror naturligtvis på vem som ska läsa kommentaren, men rent generellt kan man anta att den som läser en källkodsfil kan språket och vet vad `++` gör. Kommentaren här är därför fullständigt meningslös. En intressant kommentar skulle istället tala om varför värdet på x ökades med ett.

Ett minimum skulle jag säga är en kommentar inför varje funktion i programmet som talar om vad funktionen gör och hur man ska använda den. Har man avancerad kod inne i funktionen så är det troligen värt att skriva en kommentar där också. En miss som många programmerare gör är att de tror att det bara är de själva som ska läsa koden och att de vet vad de gör. Så är dock aldrig fallet. Det behöver inte gå mer än en vecka för att man ska ha glömt vad koden har för sig.

1.9 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.

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 ni tidigare sparade ner källkodsfilerna i. Unix-kommandon som `ls`, `cd` och `pwd` är användbara för detta. För att kompilera programmet med `gcc` skriver man helt enkelt

```
gcc klondike.c
```

Provar ni detta kommer ni dock att få upp en rad felmeddelanden som säger att ett antal funktioner inte hittas. Detta är de funktioner ni ska skriva i `cardpile.c` lite senare. För tillfället finns koden `gcc` vill ha i filen `COURSE_cardpile.o` så vi talar om det för `gcc`.

```
gcc klondike.c COURSE_cardpile.o
```

Nu får vi en körbar fil som heter `a.out`. Ni kan se den genom att lista filerna i katalogen med "`ls`". `a.out` är standardnamnet som `gcc` ger alla filer om man inte säger något annat. För att ge utfilen ett lite trevligare namn använder vi flaggan `-o` med `gcc` och döper vårt spel till `klondike`.

```
gcc klondike.c COURSE_cardpile.o -o klondike
```

Nu får vi en körbar fil med namnet `klondike`, mycket bättre. Ta bort `a.out` med "`rm a.out`" så att den inte förvirrar oss senare. 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 sätta ett antal flaggor (allt på en rad).

```
gcc -pedantic -ansi -Wall -Wextra -Wshadow -Wredundant-decls -Werror
    klondike.c COURSE_cardpile.o -o klondike
```

`pedantic` Följ C-standarden in i minsta detalj. En del ofarlig kod som bryter mot C-standarden tillåts normalt av `gcc`, men inte av mig.

`ansi` Stänger av vissa egenskaper som inte är kompatibla med C90. Till exempel nyckelorden `asm`, `typeof` och `inline`. Även enradskommentarer som inleds med `//` stängs av eftersom de inte inkluderades i C-standarden förrän senare.

`Wall` Slår på flera varningar för konstruktioner som anses tveksamma i C och som är lätta att undvika. Till exempel varnar `gcc` om man försöker läsa variabler innan man skrivit något till dem, om variabler eller statiska funktioner deklareras utan att senare användas, om funktioner saknar returtyp eller om kompilatorn inte kan avgöra vilken `if` en `else` hör till.

`Wextra` Slår på ytterligare varningar. `gcc` varnar nu om man till exempel har en tom kropp i en `if` eller en `else`, om man jämför en pekare med `0` istället för `NULL`, och för flera användningar av flyttalsoperationer som ofta innebär att man har gjort något fel.

`Wshadow` Varna om en lokal variabel skuggar någon annan variabel eller ett argument.

`Wredundant-decls` Varna om något deklareras mer än en gång.

`Werror` Gör så att alla varningar betraktas som fel.

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.

Nu kan vi provköra spelet. Starta det genom att skriva `./klondike`. När spelet startar ser vi följande skärm:

```
Klondike 2007    Slumpfrö: 1139405507
Välj rad att markera med pilen.
Använd 'u' - upp, och 'n' - ner för att styra.
'm' markerar kort eller flyttar markerade kort.
'k' lägger ut nya kort från kortleken.
Tryck 'q' för att avsluta

Guide: H = Hjärter  D = Ruter  C = Klöver  S = Spader
       A = Ess  J = Knekt  Q = Dam  K = Kung

-->[H: A][****][****][****][****][****][****]
   [H: 6][****][****][****][****][****]
   [D: 4][****][****][****][****]
   [C: 9][****][****][****]
   [C: 3][****][****]
   [S: 2][****]
   [H: K]


   [****]     [    ]    [    ]    [    ]    [    ]
```

Klondike startar med sju högar med ett till sju kort. Det översta kortet i varje hög är vänt så att man ser vad det är. Längst ner finns den resterande kortleken och fyra "lagerlokaler". Poängen med spelet är att få ut alla korten till lagren. Man kan endast lägga ut korten i nummerordning, med start på ess. Man kan flytta korten mellan högarna genom att bygga serier med vartannat kort svart och vartannat kort rött. Serierna ska även de vara i nummerordning. I exemplet ovan kan kortet `[H: A]` (som är hjärter ess) flyttas till ett lager, och kortet `[C: 3]` kan läggas på `[D: 4]`. Pilen visar vilken rad som är aktiv och kan styras upp och ned genom att ge kommandon enligt instruktionerna ovanför spelkorten. Varje kommando avslutas med radbrytning.

Utfallet i varje spel har sin grund i hur kortleken är blandad, och detta avgörs i sin tur av vilket slumpfrö som initierar blandningen. För att underlätta felsökningen står aktuellt slumpfrö högst upp på skärmen. Skulle ni hitta ett spel där er kod efter några drag slutar att fungera så kan ni lätt starta samma spel igen genom att skicka med slumpfröet som argument till klondike.

```
./klondike 1139405507
```

När ni börjar skriva er egen kod i den nya filen `cardpile.c` vill ni förstås även få med den när ni kompilerar. Det enda ni behöver göra då är att lägga till den på kommandoraden. Ni får då följande rad. Lär er den raden utantill eller skapa ett alias för den, för den raden kommer ni att skriva många gånger under den här kursen.

```
gcc -pedantic -ansi -Wall -Wextra -Wshadow -Wredundant-decls -Werror
        klondike.c cardpile.c COURSE_cardpile.o -o klondike
```

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`.

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 hjälpta av att variabelnamnet talar om vad en variabel ska innehålla. De kan bara se typen på variabeln. För att utnyttja dessa program fullt ut och få dem att klaga om man skickar fel typ av data till sina funktioner kan man skapa nya datatyper som bättre 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 mer specifika datatyper som bättre talar om vilken typ av värden de innehåller. Om vi till exempel vill ha en heltalsvariabel som ska innehålla ett klockslag så kan vi definiera en egen typ för detta och tydligt visa att variabeln inte innehåller vilket tal som helst utan just ett klockslag.

```
typedef int clock_t;
clock_t start;
clock_t slut;
```

Här har vi skapat en ny typ med namnet `clock_t`. Det är typen `int` vi ger ett nytt namn och `clock_t` är identisk med `int` i alla avseenden. `gcc` kommer att betrakta denna typ som om det vore en `int` och alla operatorer som kan användas på en `int` (`+`, `-`, `*` med flera) kan även användas på en variabel av typen `clock_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 deklarerades med typen `int` vore det inte 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. En vanligt förekommande konvention är att 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. Vi kan nu även utnyttja verktyg som `lint` för att hitta fel som `gcc` inte hittar. För `gcc` är det nya namnet bara ett alias och att skicka data mellan en `int`-variabel och en `clock_t` genererar inte kompileringsfel. `lint` däremot kommer att undra vad som händer och rapportera det.

`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 system (till exempel ett tidsplaneringssystem för bussnätet) som skrevs för länge sedan. Systemet har en klocka och hanterar av naturliga skäl ganska mycket tider. Klockan räknas i antalet sekunder från ett givet startdatum, 1 jan 1978 då systemet togs i bruk. Eftersom klockan bara är ett tal så valde man att lagra den i en variabel av typen `long int`. Man drog till med `long` eftersom en `int` vid den här tiden, på den aktuella hårdvaran endast var 16 bitar. Med `long int` kom man upp i det oändligt stora talet 32 bitar. Överallt där man hanterar tid i systemet använder man variabler av typen `long int`. Av olika anledningar överlever koden längre än någon anat (68 år) och antalet sekunder sedan 1978 börjar närma sig gränsen för vad ett 32-bitarstal kan lagra i sin signerade version. Av obegripliga skäl kan man inte kasta det gamla systemet 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 long 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) för att byta ut `long int` mot `unsigned long int`.

`grep` i all ära, men att leta efter alla förekomster av `int` i ett stort system 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. Speciellt som man på den tiden sparade minne genom att använda korta variabelnamn[1].

För att undvika den här situationen borde man ha tänkt rätt från början och definierat en egen typ för klockan. Då skulle det vara ett lätt jobb att byta ut definitionen från `typedef long int clock_t;` till `typedef unsigned long int clock_t;`. Att hitta alla ställen där `clock_t` används för att se att allt fungerar skulle också vara ett lätt jobb.

[1] Längden på variabelnamnen påverkar inte storleken på den körbara filen eller hur mycket minne programmet kräver för att köra. Det är bara storleken på källkodsfilen och läsbarheten som minskar.

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;
```

Här ser vi ett exempel på hur man kan deklarera en egen typ för sanningsvärden om man inte vill eller kan använda `stdbool`. `false` kommer i detta exempel att ha värdet 0 och `true` får värdet 1.

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

Typen `enum rank_t` är hämtad från `cardpile.h`. 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".

Nyckelordet `enum` blir en del av typnamnet och namnen på uppräkningstyperna blir därför ganska långa (`enum bool`, `enum suit_t`, `enum category_t`). För att göra koden lättare att överblicka vill vi gärna bli av med det extra ordet. Här kommer `typedef` till vår hjälp igen. `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;
```

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;
   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;

alien.title = "Alien";
alien.year = 1979;
alien.length = 117;
alien.category = SCIFI;
alien.director.happy = TRUE;

printf("%s hade 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. `char*` är ett sätt att referera till en sträng i C. Vi återkommer till strängar i 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 att bryta ut kod i så gott som alla programmeringsspråk är genom att skriva funktioner. Man kan se en funktion som ett litet miniprogram som ligger inuti det stora programmet. Vi har redan sett några funktionsanrop (`getchar`, `putchar` och `printf`). Nu är det dags att på allvar titta på funktioner 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. Man deklarerar även funktionens namn och vilken typ av data man kan skicka in till funktionen (*argument*).

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

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

Notera även `{ }` runt funktionskroppen. En funktion har alltid `{ }`, även om den bara skulle innehålla en sats.

I det här fallet har vi sagt att funktionen `foo` ska returnera ett heltal (`int`). Funktionen `foo` tar ett argument som ska vara av typen `float`. För att anropa `foo` måste man alltså skicka med ett flyttal. Gör man inte det kommer `gcc` att klaga, `gcc` är mycket noga med att typerna är rätt. 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.

Argument

Alla argument till en funktion måste deklareras med typ, precis som vanliga variabler. Har man flera argument sätter man komma emellan deklarationerna. Typen måste anges för varje enskilt argument även om flera argument i rad har samma typ. Om en funktion inte ska ta emot några argument kan man lämna parenteserna tomma. Vill man vara extra tydlig kan man ange *`void`* som argument för att markera att det inte ska vara några argument. Void betyder tomrum och används i flera olika sammanhang i C för att markera att man medvetet utelämnat något.

C är vad man brukar kalla *call-by-value*. Det innebär att alla argument till en funktion kommer att beräknas innan själva funktionsanropet sker. Betrakta kodexemplet nedan. Där finns en funktion och ett anrop till funktionen. Funktionen tar två argument som båda är tecken.

```
bool inorder(char c1, char c2)
{
   return c1 < c2;
}

bool b = inorder('A' + n, (char)getchar());
```

Det är inte så mycket kod och det ser inte ut som om det händer så mycket här men i själva verket är det ganska många olika instruktioner inblandade i detta exempel. När funktionsanropet sker kommer följande att hända, i ordning:

1. Uttrycket `'A' + n` beräknas och resultatet stoppas in i `c1`.

2. Funktionen `getchar` anropas och hämtar in ett tecken från användaren.

3. Returvärdet från `getchar` är en `int`, så den görs om till en `char` med hjälp av explicit typkonvertering. Tecknet läggs sedan i `c2`.

4. Funktionen `inorder` anropas.

5. Uttrycket `c1 < c2` beräknas och resultatet returneras.

6. Returvärdet från `inorder` sparas i `b`.

Returvärden

När programkörningen når en retursats så hoppar den ur funktionen den befinner sig i. Programkörningen fortsätter på instruktionen direkt efter funktionsanropet i den anropande koden som vi kunde se ovan. Eventuell kod som ligger efter retursatsen kommer aldrig att kunna köras. Man brukar kalla sådan kod för död kod. Funktionen `oktostore` nedan är hämtad ur `klondike.c`.

```
bool oktostore(card_t* card, card_t* storage)
{
   if (COURSE_isempty(storage) == true && COURSE_getrank(card) == ACE)
      return true;

   if (COURSE_isempty(storage) == false &&
       (COURSE_getrank(card) == COURSE_successor(COURSE_getrank(storage))) &&
       (COURSE_getsuit(card) == COURSE_getsuit(storage)))
      return true;

   return false;
}
```

Här ser vi att det finns flera retursatser i en funktion. Beroende på hur de olika villkoren faller ut kommer olika värden att returneras. När man har retursatser i villkorliga satser så är det viktigt att komma ihåg att ha en retursats som fångar upp de fall som inte matchar något av villkoren. Om man har deklarerat att funktionen ska returnera någon typ så **måste** den göra det i alla lägen. Programkörningen får aldrig nå slutet av funktionen utan att passera en retursats.

Det är också viktigt att alla retursatser returnerar data av rätt typ. Returtypen återkommer på tre ställen i samband med en funktion 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 `oktostore` ovan har returtypen `bool`. 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 `bool` 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.

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. En funktion med returtypen `void` behöver inte avslutas med någon `return`, men man kan sätta in `return` om man vill. Man skriver då endast nyckelordet `return` utan argument.

Precis som med variabler så får funktioner i ANSI C standardtypen `int` om man inte anger något annat. Precis som med variabler så ogillas denna notation (att utelämna typen) av både C99 och mig.

2.4 Här börjar det — main

För att C-kompilatorn ska kunna bygga ett körbart program av din källkod så måste den veta var den ska börja läsa programmet. Därför finns det en väldefinierad startpunkt - main-funktionen. Alla C-program måste innehålla en main-funktion och utseendet på denna måste uppfylla vissa krav utöver de som C ställer på andra funktioner.

Som jag nämnde tidigare så kan funktioner i C normalt bara anropa andra funktioner som ligger högre upp i källkodsfilen. Detta gäller naturligtvis även main. Av denna anledning hamnar normalt main-funktionen sist i en källkodsfil.

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

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
   printf("Hej från C!\n");
   return EXIT_SUCCESS;
}
```

Programmet till vänster är i sin enkelhet korrekt ANSI C-kod. Det är dock inte helt komplett och för att godkännas i denna kurs krävs det lite fler detaljer som vi kan se till höger. Först måste vi tala om att printf finns deklarerad i standardbiblioteket stdio. Detta gör vi genom att inkludera den deklarationsfil som hör till stdio med hjälp av #include. Inläsningar av standardbibliotek brukar alltid läggas först i källkodsfilen. Tittar vi i klondike.c så ser vi precis detta högst upp i filen. Först kommer de standardbibliotek vi använder och därefter de egna h-filer som vi vill använda. Även de egna filerna måste läsas in för att vi ska få tillgång till de datatyper och konstanter vi deklarerat där. Paket som skrivs inom <... > antas höra till standardbiblioteket och kompilatorn kommer att leta i systemets fördefinierade kataloger för att hitta filerna. Våra egna paket skriver vi med citattecken runt och kompilatorn kommer då att leta i samma katalog som källkodsfilen ligger i för att hitta h-filen.

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

#include "ansi.h"
#include "cardpile.h"
```

Vi tydliggör att main inte tar något argument genom att sätta void inom parenteserna. main kan ta argument, men de ska se ut på ett speciellt sätt så vi återkommer till dessa i en senare föreläsning. För att uppfylla kursens krav måste vi även tala om vad funktionen har för returtyp. I main-fallet är denna alltid int. Säger vi att vi ska returnera en int så måste vi också göra det, därav return EXIT_SUCCESS;.

main:s returvärde är lite speciellt. Tanken var från början att ett Unix-system skulle 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. 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_SUCCESS och EXIT_FAILURE. Dessa konstanter finns deklarerade i stdlib och därför har vi även inkluderat den deklarationsfilen i exemplet ovan. Man bör alltid använda konstanterna EXIT_SUCCESS och EXIT_FAILURE som returvärde från ett program för att hålla koden så plattformsoberoende som möjligt. Andra plattformar än Unix kan betrakta ett returvärde på andra sätt och anpassar då definitionen av felkoderna därefter. Till exempel Sony NEWS-OS version 4.0C där man av misstag definierat EXIT_SUCCESS till 1 och EXIT_FAILURE till 0, eller VMS som definierar EXIT_SUCCESS till 1 och EXIT_FAILURE till 0x10000002 (eller 3, eller 0...). En annan ganska vanlig implementation av EXIT_FAILURE är -1. Skulle man skriva exit(1); i sitt program och kompilera under AmigaOS får man troligen inte önskat resultat. Där har man en

bredare skala av felkoder som vi kan se i tabell 2.1, och felkod 1 är inte mycket att bry sig om[2]. Man kan även tänka sig inbyggda system som tolkar felkoderna helt annorlunda[3].

0	Allt är bra.
5	Mindre fel uppstod. Inget som påverkar kommande programs funktion.
10	Fel uppstod. Programmets utdata kan vara felaktig.
20	Allvarligt fel. Eventuella kringliggande script bör avbrytas.

Tabell 2.1: Vanlig tolkning av felkoder i AmigaOS.

Att avbryta i förtid

Normalt avslutas ett program när programkörningen kommer till slutet av `main`-funktionen. Men precis som med andra funktioner kan man avbryta programmet med en retursats var som helst i `main`.

Ibland vill man avsluta programmet i förtid, till exempel om något går fel, och befinner man sig då djupt inne i programmet räcker det inte alltid med `return`. `return` avslutar bara just den funktion man befinner sig i och programkörningen fortsätter efter funktionsanropet i den anropande funktionen.

För att avbryta hela programmet kan man använda kommandot `exit`. När man anropar `exit` avslutas programmet rakt av, oavsett om man är långt inne i en rad av funktionsanrop eller någonstans i `main`. `exit` tar ett argument som är den felkod som skickas tillbaka till operativsystemet.

```
exit(EXIT_SUCCESS);
exit(EXIT_FAILURE);
```

2.5 Omgivningar — Globala och lokala variabler

En variabel finns (endast) åtkomlig inom den *omgivning* den är deklarerad i. En omgivning kan till exempel vara en funktionskropp, 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, och i föregående sektion noterade vi även att `{ }` finns i alla funktioner. En funktionskropp har alltid en egen lokal omgivning.

I ANSI C måste variabeldeklarationer alltid stå först i en omgivning[4]. `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.

[2] `EXIT_FAILURE` är visserligen definierad till 1 även under AmigaOS så där gick lite av min poäng förlorad...

[3] Det enda man kan lita på när man programmerar för inbyggda system är att ingenting fungerar som man är van vid.

[4] I C99 blev det tillåtet att deklarera variabler senare i omgivningen.

```
 1  int plus(int x, int y)
 2  {
 3     int summa = x + y;
 4     return summa;
 5  }
 6
 7  int dela(int x, int y)
 8  {
 9     if (y != 0)
10     {
11        int kvot = x / y;
12        return kvot;
13     }
14     else
15     {
16        fprintf(stderr, "Division med noll!\n");
17        exit(EXIT_FAILURE);
18     }
19  }
```

I exemplet ovan är variabeln `summa` lokal i funktionen `plus`. Dess livslängd sträcker sig alltså över raderna 3 – 5. Det går inte att komma åt variabeln i funktionen `dela`. Variabeln `kvot` är lokal i `if`-satsen och lever därmed på raderna 11 – 13. Det går inte att komma åt `kvot` i resten av funktionen `dela`.

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. Den lokala omgivningen runt `for`-loopen tillhör alltså inte någon `if` eller loop, utan ligger helt för sig själv i koden.

```
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 livslängd i en omgivning som varit tydligt markerad i koden. Även funktionsargument räknas till de lokala variablerna i en funktion. Funktionskroppar räknas till de yttersta lokala omgivningarna. Inne i dessa kan man som vi sett skapa mindre, lokala omgivningar, och alla variabler som deklareras i lokala omgivningar betraktar vi som lokala variabler. Man kan även deklarera variabler utanför funktionerna. 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 uppstår risker 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. 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[5]. I kodexemplet nedan 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.

```
#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;
}
```

`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 kommer alltså endast att deklareras och initieras en gång[6]. Därefter kommer raden `static int d = 0;` att ignoreras - trots att där finns en tilldelning som sätter `d` till 0. Den icke-statiska variabeln (`c`) kommer att deklareras och initieras varje gång `foo` anropas. 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.

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 det alltså en statisk lokal variabel man vill ha – inte en global.

[5] Kan jämföras med privata instansvariabler i Java.

[6] Lokala statiska variabler deklareras när programmet startas precis som globala variabler.

2.6 Inlämningsuppgift 1

Nu är det dags att börja skriva lite egen kod. All er egen kod ska skrivas i en ny fil vid namn `cardpile.c`. Det är fyra funktioner som ska skrivas. Uppgifterna nedan visar hur funktionshuvudet ser ut och ger en kort beskrivning av vad varje funktion ska göra.

När ni har skrivit er egen version av någon av funktionerna nedan måste ni ändra i `cardpile.h` och ta bort `COURSE_`-prefixet i listan av funktioner längst ner i filen. Detta för att göra er funktion tillgänglig utanför `cardpile.c` och för att dölja den gamla funktionen i `COURSE_cardpile.o`. Ni måste även hitta var funktionen anropas i `klondike.c` och ta bort `COURSE_`-prefixet där också. Glöm inte att inkludera `cardpile.h` för att få tillgång till datatyper och namngivna konstanter.

A `bool colormatch(suit_t s1, suit_t s2)`
Givet två färgvärden returnerar funktionen `true` om kortfärgerna har samma fysiska färg. Det vill säga om båda korten är röda eller båda korten är svarta. I annat fall returneras `false`.

`colormatch(HEARTS, DIAMONDS)` ⟶ `true`
`colormatch(DIAMONDS, CLUBS)` ⟶ `false`
`colormatch(felaktiga indata)` ⟶ Programmet avslutas med ett felmeddelande.

B `char suittochar(suit_t suit)`
Givet en kortfärg returnerar funktionen motsvarande tecken som används för att skriva ut kortet. Observera att den enda synliga effekten av er nya kod är i guiden ovanför spelet. Själva spelkorten kommer att fortsätta skrivas ut med `COURSE_suittochar` eftersom detta anrop ligger i `COURSE_printcard` som ni inte kommer att skriva er egen version av förrän i inlämningsuppgift två.

`suittochar(SPADES)` ⟶ `'S'`
`suittochar(HEARTS)` ⟶ `'H'`
`suittochar(felaktiga indata)` ⟶ Programmet avslutas med ett felmeddelande.

C `char* ranktostring(rank_t rank)`
Givet ett kortvärde returnerar funktionen motsvarande textsträng som används för att skriva ut kortet. Notera att det inte är något mellanslag i strängarna.

`ranktostring(TEN)` ⟶ `"10"`
`ranktostring(ACE)` ⟶ `"A"`
`ranktostring(QUEEN)` ⟶ `"Q"`
`ranktostring(felaktiga indata)` ⟶ Programmet avslutas med ett felmeddelande.

D `rank_t successor(rank_t rank)`
Givet ett kortvärde returnerar funktionen det efterföljande kortvärdet.

`successor(ACE)` ⟶ `TWO`
`successor(TEN)` ⟶ `JACK`
`successor(KING)` ⟶ `ACE`
`successor(felaktiga indata)` ⟶ Programmet avslutas med ett felmeddelande.

Inlämning

Källkodsfilen `cardpile.c` med de fyra funktionerna `colormatch`, `suittochar`, `ranktostring` och `successor` skall lämnas in via ePost senast Tisdag 13/2 2007 kl. 13:10.

Föreläsning 3

Pekare är bara variabler

I den förra föreläsningen såg vi hur man kan skicka data till funktioner via argument. Argumenten som skickas till en funktion kommer att kopieras i datorns minne, från platsen där den anropande koden har lagrat värdena till de lokala argumentvariabler som deklareras i funktionen. Vi ska titta på hur detta går till för att få lite bättre förståelse för hur saker sker under ytan.

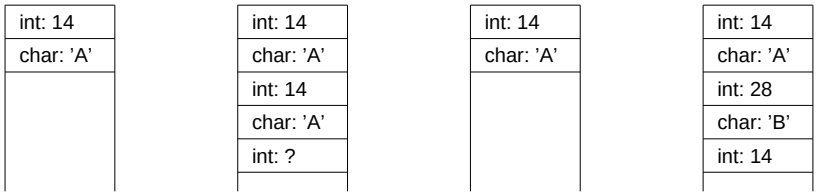
3.1 Stacken

När ett program kör använder datorn något som kallas för en *stack*. Stacken är en bit av datorns minne där lokala variabler lagras. Stacken är indelad i så kallade *stack frames*. Dessa stack frames representerar de lokala omgivningarna i funktionerna. Varje gång en funktion anropas kommer en ny stack frame att läggas upp på stacken. Storleken på denna stack frame beror 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(void)
15  {
16     int flera = 14;
17     char tecken = 'A';
18
19     skrivut(flera, tecken);
20     skrivut(flera * 2, tecken + 1);
21
22     return EXIT_SUCCESS;
23  }
```

Figur 3.1 visar hur stacken ser ut vid ett par olika tillfällen i programkörningen[1]. I figur 3.1(a)

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



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

Figur 3.1: Stack frames

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`. En ny stack frame skapas och till den skrivs argumenten. Där efter hoppar programkörningen in i funktionen och hamnar på rad 6. Även den lokala variabeln `i` får en plats på stacken.

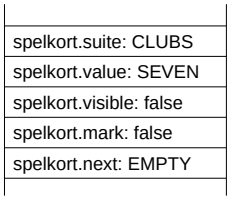
I figur 3.1(b) är vi nu på rad 7 i programmet och argumenten och den lokala variabeln har skrivits till stacken. Som vi ser har `i` inte fått något värde ännu. Jag markerar det med ett frågetecken eftersom vi inte vet vad det är för värde i en oinitierad variabel. Notera att siffran 14 och tecknet `'A'` 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`. De lokala variablerna i en funktion, vilka inkluderar argumenten, 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 det första anropet till `skrivut` och kvar på stacken finns bara den stack frame som hör till `main` (se figur 3.1(c)).

Vi förbereder oss för ännu ett funktionsanrop till `skrivut` och argumenten skrivs till stacken. C är som sagt call-by-value, så uttrycken `flera * 2` och `tecken + 1` kommer att beräknas innan anropet sker och till stacken skriver vi endast resultaten. Väl i funktionen får `i` en plats på stacken igen. Den här gången sätter jag dock inte dit ett frågetecken, se 3.1(d). Anledningen till att jag inte sätter ett frågetecken där nu är den samma som anledningen till att jag satte ett där förut. Lokala variabler initieras inte automatiskt utan det värde som råkar ligga i minnet när stack frame:n skapas ligger kvar. Första anropet visste vi inte vad som låg där sedan tidigare, nu i det andra anropet vet vi att det råkade vara värdet 14 som låg kvar i minnet.

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

Vi tar en titt på en tänkbar implementation av 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. 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 många gånger 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. Anledningen till att `getvisible` finns är ju just att plocka ut värdet ur spelkortet. Vi behöver ett sätt att referera till spelkortet utan att skicka med hela kortet till funktionen. Vi behöver pekare.

3.2 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 hur man lägger data i minnet, 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. Denna plats i minnet kommer att ha en adress som pekar ut just den minnescell där variabeln ligger. Vi kan komma åt denna adress genom att använda ett `&` framför variabelnamnet. `&` är en unär operator som sätts direkt framför namnet på den variabel man vill ha adressen till (till exempel `&x`). Exempel på andra unära operatorer är unärt minus (`-5`) och negation av logiska uttryck (`!`)[4]. 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. Vi har sett den förut, pekaren. Pekaren är en variabel som kan innehålla en adress. Den 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.

När det gäller adresser och pekare så är en C-kompilator mycket petig. Det räcker inte att bara tala om att det är en adress, vi måste även tala om vilken sorts adress det är, det vill säga vilken typ av data som ligger på adressen.

Pekartyper skrivs med en stjärna (`*`) efter typen som pekaren ska peka på. `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.

[2]Detta gäller då ett maskinord är 32 bitar. För 64 bitar gäller att adressen ska vara jämnt delbar med åtta.

[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

[4]Notera att de unära operatorerna kan ha samma symbol som binära operatorer. Unärt minus (`-5`) är inte samma instruktion som binärt minus (`7 - 5`) även om symbolen är den samma. Detta gäller även `&`, unärt `&` plockar ut adressen till en variabel, binärt `&` används för OCH på bitnivå, se appendixC för mer information.

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

Deklarationen ovan ser intuitivt ut som att den skulle deklarera två pekare, `x` och `y`. Detta är tyvärr 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;
```

Om man faktiskt menade att deklarera endast `x` som pekare är det eventuellt något tydligare att skriva "`int *x, y;`" men det är en god regel att aldrig deklarera pekare och icke-pekare på samma rad. Det bästa i det fallet skulle alltså vara att skriva:

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

Operatorn *

Stjärnan kan även fungera som en unär operator i C. Det den gör då är att dereferera en pekare, det vill säga plocka ut värdet i den minnescell som pekaren pekar på. Observera att detta visserligen är samma tecken, men är alltså inte samma stjärna som man använder för att deklarera pekare. De två har absolut ingenting med varandra att göra. Vid deklarationen är stjärnan en del av typen som talar om att det är en pekartyp. 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);`". Jämför med andra unära operatorer som till exempel `&`, `-` 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 samma operator, `*`, som derefererar adressen så att vi kommer åt minnescellen istället för adressen.

```
*adress = 4711;

tal ⟶ 4711
```

3.3 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 för 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                */
   rank_t  rank;         /* 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`. För varje datatyp som finns i C, vare sig det är en inbyggd eller en egendefinierad, finns det en motsvarande pekare.

Vi kan därför 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. Vi kan jämföra detta med ett funktionsanrop där `typedef` är funktionen och strukturdeklarationen är ett av argumenten. Precis som argumenten måste beräknas innan funktionen anropas måste strukturdeklarationen bli klar innan `typedef` kan börja. Detta är också anledningen till att vi namngett strukturen. Vore den anonym så som till exempel `rank_t` och `suit_t`[5] är så skulle det inte gå att skapa en rekursiv pekare till den.

3.4 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 viktiga för prestandan.

```
bool 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. Tidigare använde vi en punkt för att komma åt fälten i en struktur. Skillnaden mellan punkt och pil är att punkten används när strukturen finns direkt i variabeln medan pilen används när vi följer en pekare till strukturen.

Minnesregel: Punkten – här. Pilen – där borta.

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

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

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

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 lever bara i funktionens omgivning. Så snart funktionen avslutas försvinner dess stack frame tillsammans med alla värden i den. Att skicka tillbaka adresser till dessa försvunna värden är mycket farligt. Som vi sett tidigare ligger de

[5] `rank_t` och `suit_t` hittar vi också i `cardpile.h` och ni kan läsa mer om dem i sektion 2.1.

visserligen kvar i minnet ett tag, men så snart en annan funktion anropas kommer de att skrivas över av annat data. Pekare av det slaget kallas *dangling pointer*, pekaren finns kvar men inte datat som programmet tror att den pekar på. `gcc` kommer för det mesta att upptäcka detta och ge en varning, men långt ifrån alla C-kompilatorer gör detta.

Pekararitmetik

Som jag nämnt så är C ganska petigt när det gäller pekare och deras typer. Man kan inte flytta en `char*` till en variabel deklarerad som `int*` utan att använda explicit typkonvertering för att övertyga kompilatorn om att man vet vad man gör. Detta trots att alla pekare är lika stora och det inte finns någon 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 C hanterar olika pekare på olika sätt beroende på vad de pekar på.

Om man adderar ett heltal till en pekare så får man en ny pekare. Den nya pekaren kommer att referera till en ny adress som ligger lite längre fram i minnet än den ursprungliga adressen. Hur långt fram i minnet beror på pekarens typ – det är storleken på dataobjekten som avgör detta. Man hoppar nämligen alltid hela dataobjekt. 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 framåt eftersom en `int` är fyra bytes stor[6] ($4 * 5 = 20$). 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.

Ett spelkort är alltså 16 bytes stort. Hur kan det gå ihop? Titta på definitionen igen i sektion 3.3. 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 ett maskinord, det vill säga fyra bytes. Det summerar vi till 14 bytes. Var kommer då 16 ifrån?

Som vi sagt tidigare så måste alla variabler som är ett maskinord stora (`int` och pekare i det här fallet) börja på en jämn ordadress, det vill säga adressen måste vara jämnt delbar med fyra[7]. Spelkortet har därför fyllts ut med två bytes efter de två sanningsvärdena för att pekaren ska hamna på en tillåten adress.

Här ser vi att det finns en möjlighet att pussla lite om man verkligen är ute efter att spara minne. Skulle vi lagt `next`-pekaren mellan de två sanningsvärdena så hade vi fått två hål att fylla ut och strukturen skulle då bli 20 bytes stor. Det låter inte så mycket, men om vi istället säger att vi sparar 20% av minnet genom att byta plats på två variabler så blir det lite mer intressant. En C-kompilator tillåts inte flytta elementen i en struktur för att spara minne, så detta är något man får fundera på själv.

`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 generell pekartyp som går att använda, `void*`. Den kan användas för att deklarera variabler, som argumenttyp och som returtyp. Man måste alltid typkonvertera en `void*` till en "riktig" pekare innan man följer den för att plocka ut element i en struktur eller liknande.

Man bör hantera `void*` med försiktighet. Till exempel bör man aldrig använda `void*`-variabler i aritmetiska uttryck. Detta eftersom det inte är definierat vad en `void` har för storlek, och det blir därför kompilatorberoende hur långt man ska hoppa om man lägger till ett heltal till en `void*`.

[6] Storleken på datatyperna är hårdvaruberoende. Siffrorna här gäller på en 32-bitarmaskin.
[7] Se början av sektion 3.2.

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.

När man hanterar pekare vill man ofta ha ett värde som talar om att en pekare inte är giltig, till exempel för att returnera som felkod om en funktion inte lyckas hitta den adress den söker efter. 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`. I Klondike används ett annat namn för att markera tomma listor, `EMPTY`. Detta för att underlätta byte av datastruktur.

3.5 Arrayer

Arrayer har redan dykt upp lite här och där i denna kurs. Det beror på att arrayen är en datastruktur som är mycket vanlig i C-program. Vi ska nu titta närmare på hur man kan använda dem och hur de fungerar under ytan.

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

Arrayen vi skapar här heter `siffror` och har tio platser för att lagra heltal. Storleken på en array bestäms alltid när den deklareras. Man kan inte ändra storleken när arrayen väl är skapad. Siffran som anger storleken måste vara känd när koden kompileras i ANSI C. Det innebär att man som storlek kan ange siffror, namngivna konstanter och uttryck som endast innehåller dessa. Man kan inte ange variabler/argument eller funktionsanrop. Först i C99 introducerades arrayer med variabel längd – alltså vars storlek kan anges med variabler och därmed vara olika från gång till gång.

Notera att en array alltid har en typ och den kan bara innehålla värden av denna typ. I arrayen ovan kan vi alltså bara lagra heltal med typen `int`. Man kan initiera en array direkt vid deklarationen genom att skriva initialvärdena kommaseparerade inom måsvingar.

```
int heltal[] = { 10, 17, 25, 2, 37, 18, 17 };
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 i det här fallet sju 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.

Varje plats i arrayen markeras med ett index. När man vill 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. I arrayen `heltal` hittar vi där en tvåa, för att komma åt den skriver vi `heltal[3]`.

```
Index:     0    1    2    3    4    5    6
Heltal: [ 10 | 17 | 25 |  2 | 37 | 18 | 17 ]
```

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[6]`. 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 tar emot en array som argument i en funktion anger vi ingen storlek. Argumentet deklarerar inte en ny array utan vi tar emot en array utifrån som redan skapats. Storleken på arrayen har alltså redan bestämts någon annanstans. Det enda vi gör i argumentdeklarationen är att tala om typen på argumentet – i exemplet ovan, en array av `int`. Det är därför viktigt att alltid skicka med storleken när man skickar arrayer som argument.

Det är naturligt att anta att `sizeof` kan lösa storleksproblemet. Men tar man `sizeof` på en array får man bara svaret fyra. Detta avslöjar något om vad en array egentligen är. En array är egentligen samma sak som en pekare. Man har skapat arrayen för att göra koden mer läsbar. 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[3] + 42;

*(heltal + 5) = *(heltal + 3) + 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, det måste man göra själv – mer om det i nästa föreläsning. 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 som lokala variabler och därmed allokerats 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.

```
 1  void trasig(unsigned int index)
 2  {
 3     char* pekare;
 4
 5     if (index < 50)
 6     {
 7        char array[50];
 8        pekare = array;
 9     }
10     else
11        return;
12
13     pekare[index] = 'X';
14  }
```

Koden ovan visar ett exempel på 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 endast att finnas tillgänglig i denna omgivning, på raderna 7 – 9. 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.

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 antalet platser i 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` råkar stämma med antalet element. Däremot om vi har en array av till exempel heltal (`int`) så kommer varje position att ta upp fyra bytes och mängden minne som arrayen tar upp kommer alltså att vara fyra gånger större än antalet positioner i arrayen.

Att skapa arrayer av strukturer görs på samma sätt som med enkla datatyper. Dessa kommer att fungera precis som arrayerna vi sett så här långt. På varje position i arrayen kommer det att ligga ett helt spelkort. Vi kommer åt fälten i spelkorten med hjälp av en `.` på samma sätt som förut.

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

3.6 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. `'\0'` skrivs som synes med två tecken `\` och `0`. C kommer dock att tolka detta som ett tecken precis som `'\n'`.

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[] = "CPL föddes 1963";
```

I uppgift 1 såg vi ett annat sätt att deklarera en sträng. Där finns `ranktostring` 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 a[] = "CPL blev BCPL 1967";
   char b[] = "BCPL blev B 1969";

   while (*b++ = *a++);
   printf("%s\n", b);

   return EXIT_SUCCESS;
}
```

Nu har vi lärt oss tillräckligt med C för att kunna skriva lite mer svårtolkad kod. Inte för att detta är ett mål i sig, men det kan vara kul att se hur illa det kan bli. Programmet ovan har en del fuffens för sig med pekare. Prova att testköra detta program. Försök att bena ut exakt vad som händer i programmet och varför resultatet blir som det blir.

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.

3.7 Teckenkoder — ASCII med mera

I standardbiblioteket `ctype` finns en funktion som konverterar små bokstäver till stora – `toupper`. Vi ska nu titta på hur man går till väga för att skriva en sådan funktion.

Ett sätt att lösa problemet är med en stor `switch`. Vi fick ett `'a'`, då returnerar vi ett `'A'`, vi fick ett `'b'`, då returnerar vi ett `'B'`... Men det blir väldigt mycket kod att skriva. 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 och på den lägsta nivån bara bitar, ettor och nollor. Varje tecken vi kan skriva in i datorn representeras av ett tal. Mappningen mellan tecken och tal är standardiserad och återfinns i en *teckentabell*. Det pratas ofta en tabell som kallas ASCII-tabellen. Den är egentligen bara en del av teckentabellen som vi använder, men det kan ni läsa mer om i appendix D.

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 och ö. Dessa har lagts till i ett senare skede och hamnade då i den andra halvan av tabellen.

Så alla tecken är tal, och alla tal är bara en hög med ettor och nollor. För att få full förståelse för tecknens relationer tittar vi i tabellen i appendix D. Där ser vi den binära kodningen av varje tecken. 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 OCH på bitnivå med

2-komplementet till 32, alltså det tal där alla andra bitar är ett och biten för 32 är noll. För att beräkna 2-komplementet använder vi ~.

```
        '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>

char toupper(char ch)
{
   return ch & ~32;
}

int main(void)
{
   char text[] = "B blev C 1971";
   int i = 0;

   while (text[i] != '\0')
      text[i] = toupper(text[i]);
   printf("%s\n", text);
   return EXIT_SUCCESS;
}
```

Här har vi då översatt resonemanget till C. Huvudprogrammet i `main`-funktionen går igenom strängen och för varje tecken anropas `toupper`. `toupper` maskar bort 32, det vill säga biten med värde 32 sätts till noll. Vi testkör programmet och får svaret "B". Det såg inte ut som det skulle, strängen tar slut efter B. Vad händer med mellanslaget? Titta i teckentabellen 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, tecknet med teckenkod noll terminerar strängen.

För att ordna en funktion som bara konverterar bokstäver måste vi lägga in ett villkor som kontrollerar att tecknet vi konverterar 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 teckentabellen. Dessa ligger lämpligt nog på samma sätt som den första delen av alfabetet så även där kan vi förstora bokstäver genom att maska bort 32. Den nya versionen av `toupper` finner vi nedan.

```
char toUpper(char ch)
{
   if ((ch >= 'a' && ch <= 'z') || (ch >= 'à' && ch <= 'ÿ'))
      ch &= ~32;
   return ch;
}
```

3.8 Argument till `main`

Vid det här laget har ni sett flera exempel på hur olika funktioner kan ta argument av olika slag. `main` är ju även den en funktion som kan ta argument. Argumenten till `main` skickas av operativsystemet när programmet startar och deras typ och ordning bestäms därför av anropskonventionerna i Unix (eftersom C har sina rötter i den världen). Dessa säger att `main` ska ta två argument.

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

Det andra argumentet, `argv` (argument vector), är en array av strängar. Strängarna kommer från kommandoraden där programmet startas. `argc` (argument counter), är ett heltal som talar om hur många strängar som skickades in.

I Unix finns ett kommando som heter `echo`. Man anropar det med en text och det enda som händer är att texten skrivs ut till `stdout`. Prova gärna!

```
echo Oak föddes 1991
```

Ovan ser vi ett anrop till `echo`. Vi skickar med tre ord, "Oak", "föddes" och "1991". Inne i C kommer `argc` att få värdet 4 och vi får tillgång till hela kommandoraden i `argv`.

```
argv[0] ⟶ "echo"
argv[1] ⟶ "Oak"
argv[2] ⟶ "föddes"
argv[3] ⟶ "1991"
```

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

```
#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;
}
```

Vi använder en `for`-loop för att stega igenom strängarna. Notera att vi initierar `n` till 1 för att hoppa över filnamnet som ligger på första positionen i arrayen. Vill man att alla orden ska hamna i samma sträng får man slå in dem i citattecken på kommandoraden.

```
echo "Oak blev Java 1995"

argv[0] ⟶ "echo"
argv[1] ⟶ "Oak blev Java 1995"
```

3.9 Inlämningsuppgift 2

Då är det dags för lite utskrifter och lite list-traversering. Ni behöver inte modifiera pekare i dessa deluppgifter. Funktionen `COURSE_getnext` används för att stega framåt i korthögen. Tänk på att *alltid* använda de get- och set-funktioner som finns deklarerade längst ner i `cardpile.h` när ni arbetar med spelkorten. Det är strängt förbjudet att läsa eller skriva direkt till fälten i strukturerna.

A `void printcard(card_t* card)`
Skriver ut givet spelkort på `stdout`. Denna funktion anropas vid utskriften av spelkorten. Viss frihet finns i utskriften, men det går inte att göra så mycket spännande i terminalen. Spelkortet bör inte vara mer än 6-7 tecken brett. Alla kort ska vara lika breda! Enklast är det att låta kortet vara en rad högt, men det går att göra ganska snygga saker om man leker lite med styrkoderna i `ansi.h`. Glöm inte att kolla mark-flaggan för att se om

kortet är markerat och visible-flaggan för att se om kortet ska ligga med framsidan upp eller ner. Även tomma kort ska skrivas ut (då card = `EMPTY`). Ni ska använda era egna `suittochar` och `ranktostring` vid utskriften. Ni kommer att behöva använda flera koder i formatsträngen till `printf`. På kurshemsidan finns en referens till en sida där alla dessa koder finns beskrivna.

B `void printcardpile(card_t* pile)`
Skriver ut den givna högen med spelkort på `stdout`. Denna funktion anropas vid utskriften av spelkorten. För varje kort i korthögen anropas `printcard` för att skriva ut kortet. Korten ska skrivas ut på en rad som avslutas med en radbrytning[8]. Funktionen ska även hantera utskrift av en tom korthög (då pile = `EMPTY`).

C `int countcards(card_t* pile)`
Givet en korthög returneras antalet kort i högen. Ska kunna hantera tomma korthögar.

D `bool unmarkpile(card_t* pile)`
Givet en korthög skall funktionen avmarkera de kort som är markerade i högen. Det är tillåtet att anta att alla markerade kort ligger i en obruten serie med start på högens översta kort. `COURSE_getmark` och `COURSE_setmark` används för att hantera markeringsflaggan. Funktionen ska returnera `true` om det fanns några markerade kort att avmarkera och `false` annars. Funktionen ska hantera att anropas med en tom korthög.

Inlämning

Källkodsfilen `cardpile.c` med de fyra nya funktionerna `printcard`, `printcardpile`, `countcards` och `unmarkpile` skall lämnas in via ePost senast Tisdag 20/2 2007 kl. 13:10. Filen skall även innehålla korrekta lösningar från uppgift ett.

[8] Med en rad menar jag en rad kort, inte nödvändigtvis en rad i terminalen.

Föreläsning 4

Minnesallokering — Pekare var bara början

När man pratar om minnet i datorn så finns det generellt sett två olika områden som man bör känna till, *stacken* och *heapen*. Stacken har vi redan sett, det är där alla lokala variabler finns. Alla variabler som ska finnas på stacken måste vara kända när programmet kompileras, både när det gäller storlek och antal. Ibland vill man ha en datastruktur som kan vara olika stor under programmets gång, eller så har man kanske en struktur vars storlek avgörs av användaren av programmet och man kan helt enkelt inte veta hur många element den ska ha förrän 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. Som argument skickar man med hur mycket minne man vill ha. `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 är pekartypen för argument och returvärde alltid `void*`. För att kunna använda minnet får man därför 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*`. Observera att storleken på minnet anges i bytes.

```
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 av datorns minne vi kallar heap. Man brukar sträva efter att alltid allokera stora datastrukturer på heapen. Stacken har en fast storlek och denna är normalt ganska liten jämfört med hur mycket minne det finns i datorn. Heapen däremot har tillgång till hela datorns minne[1].

[1] Egentligen tar heapen upp hela datorns RAM. Stacken allokeras i heapen när ett nytt program startar.

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. att vi lämnar tillbaka minne innan vi är klara med det, och
2. att 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. Vi har nämnt dem tidigare när vi pratade om att skapa pekare till stacken. Den här typen av fel kan vara mycket svåra att hitta eftersom det frigjorda minnet fortfarande *går* att använda tills dess att någon annan del av programmet allokerar det och skrivit över det gamla datat.

Om vi glömmer att lämna tillbaka minnet vi har allokerat så har vi en *minnesläcka*. I program som kör länge är detta ett farligt fel eftersom det innebär att minnet så småningom kommer att ta slut. Olika serverprogram till exempel eller styrprogram i maskiner av olika slag kan ha körtider på flera år (förutsatt att de inte har minnesläckor då förstås).

Det är lätt att tro 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. Minnet kommer att förbli ockuperat till dess att hela programmet avslutas, men 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[2]. På olika system kan segmenteringsfelen se lite olika ut. Under Unix och liknande får man en signal som heter `SIGSEGV`, under Windows kastas ett exception; `STATUS_ACCESS_VIOLATION`. På andra system (till exempel sådana baserade på Motorola 68000) kan det kallas adressfel.

Om man frigör minne för tidigt och fortsätter använda det är det dock inte troligt att vi får något segmenteringsfel. Anledningen till att vi kan använda minne som vi frigjort utan att något händer är att programmet inte lämnar tillbaka minnet till operativsystemet direkt när det frigörs utan håller i det ett tag under ytan. Om man allokerar nytt minne strax efter att man har frigjort minne så kommer programmet, om möjligt, att ge tillbaka samma minne som man just frigjort. Detta för att undvika systemanrop till operativsystemet när man allokerar, vilket skulle ta betydligt längre tid.

[2]Segmentering är ett sätt att skydda datorprogram från att andra "elaka" program förstör deras data. Ett annat sätt är paging.

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);

kortlek[37].rank = ACE;
kortlek[37].suit = SPADES;
```

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.

När vi pratade om arrayer sa vi att en array och en pekare egentligen är samma sak. När vi nu har ett minnesblock med plats för 52 spelkort kan vi därför betrakta det som en array av spelkort med 52 element.

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

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

kortlek[37]->rank = FOUR;
kortlek[37]->suit = HEARTS;
```

Här allokerar vi ännu en array. Den här gången tar vi storleken av en pekare till spelkort gånger 52. Vi allokerar alltså en array med plats för 52 pekare. Där efter går vi igenom arrayen och för varje element i den allokerar vi plats för ett spelkort och sparar adressen i arrayen.

De två arrayerna skiljer sig på en fundamental punkt – var själva spelkorten ligger i minnet. Allt vi har allokerat här ligger naturligtvis på heapen. Skillnaden är att i det första exemplet så ligger själva korten – alltså strukturerna med allt data – i arrayen, det vill säga de ligger på samma fysiska plats i minnet. I det andra fallet ligger arrayen på ett ställe i minnet och innehåller pekare till spelkorten som ligger utspridda på andra ställen[3]. Detta ser vi i exemplen på att vi i det första fallet använder "`.`" för att komma åt fälten i ett spelkort givet en plats i arrayen, och i det andra fallet använder vi "`->`" för att följa en pekare.

Notera även att typen på variabeln `kortlek` i det första exemplet är `card_t*` – en pekare till ett spelkort, och i det andra exemplet `card_t**` – en pekare till en pekare till ett spelkort. Pekare till pekare återkommer vi till om en liten stund.

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 läsa i minnet. Arrayen av pekare ovan initierar vi med en `for`-loop där vi allokerar kort som pekarna i arrayen får referera till. Notera att själva spelkorten, som ligger på andra ställen i minnet, inte är initierade. Den första arrayen däremot, den där korten ligger i arrayen, har vi inte initierat. Detta skulle vi förstås kunna göra genom att initiera alla fält i alla kort med hjälp av en `for`-loop även här. 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, det första är att använda `calloc` istället för `malloc`. `calloc` allokerar en array och initierar automatiskt alla positioner i den till 0. `calloc` tar två argument; antalet platser i arrayen och deras storlek. Det andra sättet är att använda `memset`. `memset` fyller en given minnesarea med ett givet värde, till exempel 0. Vi skickar med en pekare till arean vi vill fylla, värdet vi vill fylla med och storleken på arean. Kom ihåg att storleken alltid anges i bytes.

[3] Eftersom minnet allokeras i direkt följd är det ganska troligt att array och kort ligger i direkt följd i minnet, men det måste inte vara så.

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

Observera!

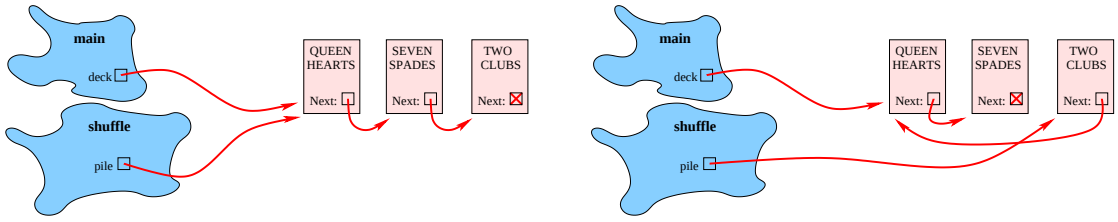
När det gäller kortleken ni ska implementera i inlämningsuppgiften så bör ni tänka på hur ni kommer att använda korten innan ni allokerar minnet i uppgift fyra. Exemplen här visar hur korten ligger i en array och hur man kan använda arrayen för att hantera korten. Ni kommer inte att ha någon nytta av 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 dem 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`. Vi tänker oss att den har följande deklaration:

```
void shuffle(card_t* pile);
```

Vi har alltså redan allokerat en korthög på heapen och nu anropar vi `shuffle` för att blanda. Som argument skickar vi med en pekare till korthögen som ska blandas. I figur 4.1(a) ser vi hur det kan se ut i minnet. 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 dess värde har därför kopierats till den lokala variabeln `pile` i `shuffle`.



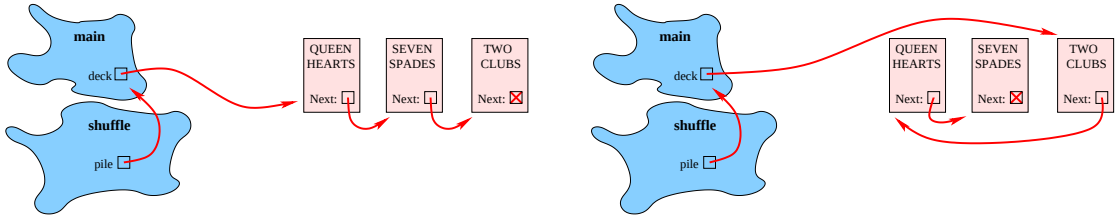
(a) Före (b) Efter

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 nu när vi returnerar från `shuffle`? Den lokala omgivningen för `shuffle` 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 vill vi kunna uppdatera `main`:s variabel `deck` innan vi avslutar `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)->rank = x;` — Lägg värdet från `x` i fältet `rank` 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).rank = x;` — Lägg värdet från `x` i fältet `rank` i det spelkort som vi kommer till om vi följer de två pekarna enligt exemplet 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.

4.5 Inlämningsuppgift 3

Då är det dags att få smutsiga fingrar! Pekare är den största orsaken till fel i C-program så det gäller att hålla tungan rätt i munnen när ni löser uppgifterna. Rita på papper så att ni vet vad ni ska göra innan ni börjar skriva kod. För enkel felsökning brukar det fungera bra att skriva ut värden på variabler och spårutskrifter som talar om var i koden man är. För lite mer avancerad felsökning rekommenderar jag `gdb`. Se appendix F.1 för mer information.

A `void placeontop(card_t** pile, card_t* card)`
Givet en korthög och ett spelkort placeras spelkortet överst på högen. Ni får förutsätta att kortet är "löst", det vill säga att det inte ingår i någon hög när ni får det. Korthögen kan vara tom.

B `card_t* drawone(card_t** pile)`
Givet en korthög returneras det översta kortet. Kortet ska plockas bort från korthögen och det ska inte vara möjligt att komma åt korthögen givet det bortplockade kortet. Om korten är slut skall det tomma kortet (`EMPTY`) returneras.

C `void shuffle(card_t** pile)`
Givet en korthög blandas den. Att blanda en länkad lista är lite krångligt. Det är betydligt lättare att skapa en array med pekare till varje kort, blanda arrayen och sedan ordna den länkade listan efter arrayen. Ett mycket enkelt sätt att blanda en array är att gå igenom den från början till slut och för varje position byta plats på det som ligger på aktuell position och innehållet på en slumpmässigt vald position. Algoritmen kallas Randomize in place och ser ut så här:

```
for (platsN = 0 ... antal kort - 1)
    platsR = slump(mellan platsN och antal kort - 1)
    byt_plats(platsN, platsR)
endfor
```

Läs mer om slumptal i appendix A. Det är tillåtet att anta att det inte finns fler än 52 kort i korthögen, men funktionen ska kunna blanda korthögar med färre kort!

D `void movemarked(card_t** from, card_t** to)`
Givet två högar med spelkort, flytta alla markerade kort från hög ett (`from`) till hög två (`to`). Korten skall hamna i samma ordning som de hade innan flytten. Efter flytten skall korten vara avmarkerade. Det är tillåtet att anta att alla markerade kort ligger brevid varandra med start i början av högen.

Innan flytt:

```
From:  3* 4* 5* 9 2 K A
To:  6 7 Q 8
```

Efter flytt:

```
From:  9 2 K A
To:  3 4 5 6 7 Q 8
```

Inlämning

Källkodsfilen `cardpile.c` med de fyra nya funktionerna `placeontop`, `drawone`, `shuffle` och `movemarked` skall lämnas in skall lämnas in via ePost senast Tisdag 27/2 2007 kl. 13:10. Filen skall även innehålla korrekta lösningar från uppgift ett och två.

Föreläsning 5

Makron — Snyggt, effektivt och lite farligt

När ni tittat i källkodsfilerna har ni kanske lagt märke till att det finns en del kod i inlämningsuppgiften som ser lite udda ut, till exempel 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 källkoden kompileras skickas den nämligen igenom en så kallad *preprocessor*. Denna kommer att ta hand om alla rader som inleds med ett `#` (preprocessordirektiv). Bland annat läser preprocessorn in `h`-filer via `#include`. `h`-filerna kommer att klistras in i källkoden på den plats där `#include`-direktivet 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`. Just det där med att klistra in text i källkoden är något som preprocessorn är väldigt bra på. Faktum är att preprocessorns främsta uppgift är att byta ut text i källkodsfilen mot annan text.

5.1 Makron som konstanter — `#define`

Preprocessorn springer naturligtvis inte runt och byter ut text hur som helst. Man måste tala om vad som ska tas bort och vad som ska stoppas in i stället. Detta gör man med så kallade *makron*. Rader som inleds med `#define` deklarerar makron. Man kan skapa allt från makron som är enkla namn på konstanter till makron som fungerar som hela funktioner. 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 `enum`. Den stora skillnaden är just att texten `EMPTY` kommer att bytas ut mot texten `NULL` innan programmet kompileras. Detta steg kallas för *makroexpansion*. I denna föreläsning kommer jag att använda symbolen $\Longrightarrow$ för att beteckna makroexpansion. Alltså: `EMPTY` $\Longrightarrow$ `NULL`.

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 enkla länkade listan av spelkort mot en skip-lista[1]. I effektiva implementationer av datastrukturer avsedda för sökning 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

[1] En skip-lista är en typ av länkad lista där sökningar kan göras mycket snabbare än i en vanlig länkad lista.

man har 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.

Det finns fler makrodefinitioner i `ansi.h`. Dessa används för att trolla lite med terminalen och ändra färg på texten. Om vi skrev in strängarna för att byta färg med mera direkt i koden skulle den bli mycket svårtydd. Man skulle behöva fundera ett tag innan man förstod vad som hände och troligen skulle man även behöva en tabell för att slå upp exakt vad de olika styrkoderna betyder. Med makron får vi nu istället snygga, förklarande namn som man direkt förstår vad de gör.

```
#define ASCII_ESC        "\033"
#define ANSI_BLACK       ASCII_ESC"[30;47m"
#define ANSI_CLEARSCREEN ASCII_ESC"[47m"ASCII_ESC"[2J"
#define ANSI_CURSOR      ASCII_ESC"[1;5;30;47m"
```

I detta exempel ser vi att det även går bra att anropa makron i själva makrot. `ASCII_ESC` är ett makro som används i definitionen av de andra makrona i listan. Makroexpansionen av till exempel `ANSI_BLACK` kommer att ske i två steg, först expanderas `ANSI_BLACK` till `ASCII_ESC"[30;47m"` och sedan expanderas `ASCII_ESC` till `"\033"`. Slutresultatet blir `"\033""[30;47m"`.

```
/* Lite konstanter */
#define PILES    7
#define STORAGES 4

/* Kontrolltecken */
#define CHAR_UP    'u'
#define CHAR_DOWN  'n'
#define CHAR_QUIT  'q'
#define CHAR_MARK  'm'
#define CHAR_CARDS 'k'
```

Även konstanterna i `cardpile.h` har definierats med makron. Det finns en anledning till att jag väljer att göra detta med makron istället för att ha `const`-deklarerade variabler, vilket ju annars skulle vara ett alternativ. Jag tycker att C:s `const`-deklaration är för svag eftersom det inte finns någon garanti för att kompilatorn skriker om man försöker ändra på en `const`-deklarerad variabel. Med ett makro vet jag att jag kommer att få kompileringsfel om jag skulle råka skriva "`PILES = x`" istället för "`PILES == x`" i någon `if`-sats till exempel. Detta kommer ju att expanderas till "`7 = x`" och lyckligtvis finns det ingen C-kompilator som tillåter att man ändrar värdet på 7.

Följande exempel är hämtat från `klondike.c`. Det är de första raderna ur `printinfo` vi ser. Här används ett flertal av de makron vi precis deklarerat.

```
printf(ANSI_CLEARSCREEN ANSI_HOME ANSI_BLACK);
printf("Klondike 2007    Slumpfrö: %d\n", seed);
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);
```

Efter makroexpansion får vi följande funktion. Det är väl främst den första raden som blir lite otrevlig utan makron. Där ser vi hur vi utnyttjar att strängar som läggs efter varandra i källkoden slås ihop till en sträng. De olika makrona representerar olika strängar och vi kan alltså skriva ut flera av dem i samma anrop till `printf` genom att skriva dem efter varandra. Observera att vi alltså skriver dessa istället för en sträng, inte i en sträng.

```
printf("\033""[47m""\033""[2J" "\033""[H" "\033""[30;47m");
printf("Klondike 2007    Slumpfrö: %d\n", seed);
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');
```

Om man vill se hur ett program ser ut efter att preprocessorn 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
```

5.2 Funktionsliknande makron

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. Inlining[2] kan fungera i vissa kompilatorer under vissa omständigheter, men det finns inga garantier för att all kod man vill ha inklistrad kommer att tas med. Funktionsliknande makron kräver oftast lite extra försiktighetsåtgärder som vi snart ska se, men det är det normalt värt 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.

Funktionsliknande makron skrivs med samma namnkonvention som funktioner. Det är inte meningen att programmeraren som använder ett bibliotek med funktioner ska behöva bry sig om ifall en funktion är implementerad med ett makro eller med en riktig funktion. På anropsstället i koden ska det se lika ut oavsett. På så sätt blir det lätt att byta implementation mellan makron och funktioner.

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.

```
#define redprint(text) printf(ANSI_RED text ANSI_RESET)
```

Makrot ovan är alltså tänkt att skriva ut något med röd text. Vi kan slänga på ANSI-koderna före och efter texten som skrivs ut så länge argumentet är en ren sträng. (Jag väljer att inte expandera ANSI-koderna för läsbarhetens skull.)

`redprint("hej")` $\Longrightarrow$ `printf(ANSI_RED "hej" ANSI_RESET)`

`printf` kan ju skriva ut mer avancerade saker än rena strängar och det är väl inte helt osannolikt att vi vill göra motsvarande saker med vårt makro. Om man till exempel vill skriva ut värdet på en heltalsvariabel så gör man det enkelt med `printf("%d", x)`. Som vårt makro `redprint` är definierat nu kan vi inte göra samma sak med det. `text` förutsätts vara en enkel sträng. Men eftersom vi kan skicka in vad som helst som argument kan vi skriva om makrot så att det istället tar emot en större del av `printf`-kommandot.

```
#define redprint(text) printf(ANSI_RED); printf text; printf(ANSI_RESET)
```

Vi ser här att ett makro kan expanderas till flera kommandon och beräkningar, det är som sagt bara text som byts ut mot annan text. Vi delar upp utskriften i tre delar och säger att det argument som ska skickas in är hela parentesen som ska stå efter `printf`. Notera att vi alltså har dubbla parenteser i anropet. De yttersta hör till makroanropet och de inre blir en del av

[2]Inlining betyder att kompilatorn istället för att skapa ett funktionsanrop klistrar in koden från funktionen vid anropet.

argumentet. Anledningen till att vi vill ta med hela parentesen och inte bara dess innehåll är att vi vill att vårt makro bara ska ta ett argument, annars blir vi lika låsta som innan.

`redprint(("%d", x))` $\Longrightarrow$ `printf(ANSI_RED); printf ("%d", x); printf(ANSI_RESET)`

De tre utskrifterna kommer att klistras in i källkoden där makroanropet finns. Jag har medvetet låtit bli att sätta semikolon efter alla makron så här långt. Detta beror på att anropen i koden kommer att se ut som vanliga funktionsanrop och det vanliga är att man sätter semikolonet efter anropet precis som med vanliga funktioner. I vårt sista makro sitter det därför semikolon efter de två första utskrifterna, medan den tredje kommer att få det semikolon som redan finns efter anropet.

`redprint(("%d", x));` $\Longrightarrow$ `printf(ANSI_RED); printf ("%d", x); printf(ANSI_RESET);`

Semikolonet efter anropet här är alltså inte en del av makrot eller makroanropet, det är bara text som råkar stå direkt efter anropet. Det är först när kompilatorn kommer och ser texten som semikolonet betyder något.

Vårt nya makro expanderas alltså till tre instruktioner. Detta kan innebära problem om man inte tänker sig för. Betrakta följande exempel.

```
if (x < 0)
   redprint(("%d", x));
else
   greenprint(("%d", x));
```

Vid en första anblick ser det bra ut, men tänker man ett steg längre inser man att `redprint` byts ut mot tre instruktioner. Endast den första kommer att ingå i `if`-satsen, därefter kommer två utskrifter som alltid kommer att skrivas ut och sedan en `else` som inte har någon matchande `if`. `gcc` kommer att skrika och för en som inte vet hur `redprint` är definierad blir det svårt att felsöka detta. Man ska inte behöva veta hur ett makro är definierat för att kunna använda det så felet ligger inte vid anropet utan vid makrodefinitionen. Det finns en viktig grundregel när man skriver makron: Ett expanderat makro ska alltid kunna betraktas som **en** sats.

OK, en sats. Men vi ville ju ha tre. Det normala sättet att trycka in fler satser där det bara får plats en är att sätta måsvingar runt dem. Vi provar...

```
#define redprint(text) { printf(ANSI_RED); printf text; printf(ANSI_RESET); }
```

`redprint` i exemplet expanderas nu till

```
if (x < 0)
   { printf(ANSI_RED); printf ("%d", x); printf(ANSI_RESET); };
else
   greenprint(("%d", x));
```

Det ser bra ut bortsett från en liten detalj, semikolonet efter anropet ligger kvar och hamnar nu efter måsvingarna. Semikolonet kommer att betraktas som en egen sats och problemet kvarstår. Vi kan alltså inte nöja oss med att bara säga att det ska vara en sats, det måste vara en sats som vi kan sätta ett semikolon efter. För att vara på den säkra sidan finns det egentligen bara ett alternativ; `do`-loopen.

```
#define redprint(text) do { printf(ANSI_RED); printf text; printf(ANSI_RESET); } while(0)
```

Det kan verka dumt att sätta in en loop här, men `do`-loopen är den enda konstruktionen i C där vi får en komplett omgivning med möjlighet att deklarera variabler och lägga in flera satser och där vi kan sätta ett semikolon efter utan att det blir två satser. Loopen i sig har ett konstant villkor som är falskt och kommer därför att optimeras bort helt. Exemplet visar också att det är viktigt att tänka sig för även om man bara har en instruktion i sitt makro. Ett semikolon sist i makrodefinitionen skapar samma situation där.

Ett funktionsliknande makro ska alltid gå att betrakta som en sats som man kan sätta ett semikolon efter.

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 den skillnaden. Satser är kommandon som inte resulterar i ett värde som vi behöver ta hand om. Ett anrop till `printf` är en sats, `x = 53;` är också en sats. Visserligen har både `printf` och `x = 53;` ett returvärde, men detta kan ignoreras och satserna har fortfarande sidoeffekter som gör att de är meningsfulla.

`5 + 3` är ett uttryck. Uttryck beräknar ett värde som vi vill ta hand om på något sätt. 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`.

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

`area(5)` $\Longrightarrow$ "`5 * 5 * PI`"

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

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. Makron är vad man brukar kalla call-by-name. 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.

`area(dyrkod())` $\Longrightarrow$ `dyrkod() * dyrkod() * PI`

Den andra faran med makron som tar argument är att när man skriver makrot kan man aldrig veta hur argumenten man kommer att få 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)
```

Makroanropet 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 läge att vara paranoid. Alla förekomster av argumentnamn i ett makro ska omges med parenteser.

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

Det samma gäller åt andra hållet också. Vi vet inte hur det kommer att se ut på anropsplatsen och till exempel ett anrop i stil med "`x / area(y)`" kommer inte att ge önskat resultat. Precis som vi slog in funktionsliknande makron i `do`-loopen för att garantera att de alltid betraktades som en sats måste vi här slå in uttrycket i parenteser för att garantera att det håller ihop oavsett hur det ser ut vid anropsstället.

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

Även med parenteser runt argumenten så måste man komma ihåg att `radie` i exemplet ovan inte är en variabel. Det skulle inte fungera att till exempel säga `(radie)++` eftersom detta till exempel skulle kunna expanderas till `(5)++` och man kan inte använda `++`-operatorn på konstanta värden. Det normala är att man alltid betraktar argument till ett makro som enbart för läsning. Man kan alltså hämta data den vägen men man ska inte skriva till argumenten om man inte vet vad man gör.

Om man vill att ett makro ska fungera som ett 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.

(sats, sats, ..., uttryck)

Vill man utföra ett antal satser i ett makro som ska betraktas som en sats kan man skriva 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 parentesuttryckets 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 även om det naturligtvis finns fall där de är mycket användbara.

uttryck_1 ? uttryck_2 : uttryck_3

Man kan inte använda `if`-satser i parentesuttryck vilket man ibland skulle vilja. Istället får man använda `?:`. Om uttrycket framför frågetecknet är sant kommer uttryck_2 att beräknas och returneras, 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))
```

Precis som parentesuttrycken så är `?:` helt vanlig C-kod som inte är specifikt knuten till makron. `?:` är mycket användbar i till exempel loopar där man har något specialfall, eller vid beräkning av returvärden. Följande exempel hittar vi i funktionen `movesleft` i `klondike.c`.

```
for (j = (i == 0 ? 1 : 0); j < PILES; (++j == i ? ++j : j))
```

Detta är alltså en loop med loop-index `j`. Den ligger inuti en annan loop som har loop-index `i` och koden i denna loop-kropp ska endast utföras då `i` och `j` inte är lika. Om `i` är noll börjar `j` på ett, annars på noll. Förändringskoden i loopen utnyttjar också `?:`. Först ökas värdet på `j` med ett som man brukar i en `for`-loop. Där efter kollar vi om `j` är lika med `i`. Om detta är fallet ökar vi `j` med ett en gång till.

5.4 Makron med flera rader — \

Ett makro kan bestå av hur mycket eller lite text som helst. Det kan vara allt från enkla namn på konstanter till avancerade funktioner med argument och returvärde. För att det ska bli läsbart vill man oftast dela makrodefinitionen över flera rader. En makrodefinition förväntas dock bara ta en rad. Preprocessorn börjar läsa på rader som inleds med `#` och läser fram till nästa radbrytning. Man kan skriva makron som sträcker sig över flera rader genom att sätta ett `\` sist på raden.

```
#define cardcheck(pile, n) do {                                          \
      if (COURSE_isempty(pile) == false) {                               \
         int count = COURSE_countcards(pile);                            \
         if (count != (n)) {                                             \
            printf("I got %d cards, but was expecting %d at %s:%d\n",    \
                   count, n, __FILE__, __LINE__);                        \
            exit(EXIT_FAILURE);                                          \
         }                                                               \
      } else if ((n) != 0) {                                             \
         printf("I got an empty pile but was expecting %d cards at %s:%d\n", \
                n, __FILE__, __LINE__);                                  \
         exit(EXIT_FAILURE);                                             \
      }                                                                  \
   } while(0)
```

Notera att det inte är något \på sista raden. Makroexpansion handlar som sagt om rent utbyte av text och när preprocessorn är klar med detta kommer det att se ut så här[3]:

```
cardcheck(deck, 52); Longrightarrow
do {
   if (COURSE_isempty(deck) == false) {
      int count = COURSE_countcards(deck);
      if (count != (52)) {
         printf("I got %d cards, but was expecting %d at %s
                count, 52, "klondike.c", 389);
         exit(1);
      }
   } else if ((52) != 0) {
      printf("I got an empty pile but was expecting %d card
             52, "klondike.c", 389);
      exit(1);
   }
} while(0);
```

Vi ser att `pile` har bytts ut mot `deck` och att `n` har bytts ut mot 52. Det är några saker till som har hänt med texten. `EXIT_FAILURE` är ett makro som här har bytts ut mot 1 och de magiska namnen `__FILE__` och `__LINE__` har bytts ut till `"klondike.c"` och 389. Dessa namn är definierade av preprocessorn själv och kan alltså användas för att få tag i namnet på källkodsfilen och radnummret där koden står. Detta är mycket användbart vid felsökning. Det finns fler namn som preprocessorn definierar:

`__DATE__` Aktuellt datum

`__TIME__` Aktuell tid

`__FILE__` Aktuellt filnamn

`__LINE__` Aktuellt radnummer

`__STDC__` Ett värde skiljt från noll

Om namnet `__STDC__` är definierat indikerar det att man har en ANSI-kompatibel kompilator. Inget av dessa namn kan omdefinieras av programmet.

5.5 Preprocessormagi

```
#define setsuit(card, s) (card)->suit = (s)
```

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

`setsuit(kort, CLUBS)` $\Longrightarrow$ (kort)-¿suit = (CLUBS)

I definitionen av `setsuit` ovan använder 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 nämligen smartare än så, endast hela ord byts ut. Om ett argumentnamn förekommer i en annan text i makrot kommer det inte att bytas ut.

Hur gör vi då om vi faktiskt vill att en del av en text i makrot ska bytas ut..? Vi kan tänka oss ett antal räknare i en databas som för lite statistik över hur ofta saker läggs in och plockas bort. Vi slänger på ett prefix på räknarna för att inte blanda ihop dem med andra variabler i programmet. Vi deklarerar ett makro och två variabler.

```
#define inc(name) counter_name++
int counter_insert, counter_delete;
```

För att få koden så ren som möjligt vill vi dock slippa skriva ut prefixet vid makroanropen. Vi vill alltså anropa makrot med "`inc(insert)`" eller "`inc(delete)`". Tanken är nu att preprocessorn ska byta ut `name` mot det vi skickar in som argument till makrot. Detta kommer dock inte att fungera. Argumentet `name` återfinns inte i makrotexten utan det enda som finns där är `counter_name` som betraktas som ett enda ord. För att skilja på de två delarna av variabelnamnet och tillåta att den andra delen byts ut av preprocessorn använder man `##`. Genom att lägga in `##` mellan `counter_` och `name` blir `name` ett separat ord som kan bytas ut av preprocessorn.

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

Det är ganska vanligt att man har en del kod i sitt program som finns där enbart för att fånga fel under utvecklingen. 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 preprocessorn göra åt oss. Det går nämligen att fråga preprocessorn 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.

```
#ifdef DEBUG
#define dprint(msg) printf([DEBUG: %s @ line:%d] ", __FILE__, __LINE__);  \
                    printf msg; printf("\n");
#else
#define dprint(msg)
#endif
```

Om namnet `DEBUG` är definierat så kommer `dprint` att expanderas till en utskrift, annars blir det ingenting kvar av makroanropet. Namnet `DEBUG` kan definieras någonstans tidigare i koden. Detta makro behöver inte ha någon "kropp" eftersom vi endast är intresserade av att namnet som sådant ska vara definierat. Det kan alltså vara tomt efter själva namnet: `#define DEBUG`

Man kan även deklarera makron vid kommandoraden när man kompilerar programmet. Man skickar då med flaggan `-D` tillsammans med makrodefinitionen.

```
gcc -DDEBUG ...
```

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 så att preprocessorn kan ta hand om dessa anrop kommer de att finnas 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. Man kanske är beroende av att veta åt vilket håll stacken växer i minnet för att kunna utföra viss pekararitmetik[4], man kanske vill veta om maskinen har flyttalsregister eller ej för att kunna utnyttja dessa om de finns eller så måste man kanske veta hur bitarna ligger i ett maskinord för att kunna utföra bitmanipulationer[5].

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 inkluderar även `klondike.h` i början av `klondike.c`. När preprocessorn nu börjar klistra in `h`-filer så kommer den först att titta på `klondike.c`. `klondike.c` inkluderar `klondike.h` och innehållet i `h`-filen klistras in i `c`-filen. Sedan ser den att `klondike.h` vill ha `cardpile.h` och klistrar in den. Därefter fortsätter preprocessorn i `klondike.c` och ser att även här ska `cardpile.h` klistras in. Resultatet blir att alla datatyper som finns i `cardpile.h` klistras in och 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.

5.7 Inlämningsuppgift 4

Nu är det dags att allokera lite minne. Än en gång: Tänk på att minne och pekare är den största orsaken till fel i C-program. Rita på papper och tänk igenom koden en extra gång innan ni börjar implementera något.

[4] På x86 växer stacken från låga adresser mot högre, på de flesta andra arkitekturer växer stacken från höga adresser mot lägre.

[5] x86 är vad man kallar little endian vilket betyder att bytes i ett maskinord ligger med den minst signifikanta byten först. Bitarna hamnar då i "fel" ordning mot hur man normalt tänker sig att det ska se ut. Andra arkitekturer är big endian vilket betyder att man börjar med den mest signifikanta byten och allt blir rätt. Namnen little endian och big endian härstammar från sagan om Gullivers resor där lilleputtfolket låg i krig över i vilken ände man ska börja äta ett ägg - den stora änden eller den lilla änden.

A `card_t* initdeck(void)`
Funktionen skall initiera och returnera en ny kortlek. Detta innebär att skapa en lista med 52 kort och ge dem rätt värden. Inga kort i en ny kortlek är synliga eller markerade. Här kommer ni att allokera minne (`malloc`) för varje kort (var för sig!).

B `void cleanupdeck(card_t** pile)`
Givet en korthög städas alla kort i den bort. Minnet för korten ska avallokeras (`free`) och den givna kortpekaren ska sättas till `EMPTY`. Funktionen ska hantera tomma högar.

C Längst ner i `cardpile.h` finns en lista med ett antal funktioner som ni känner väl vid det här laget. Det är funktioner för att plocka ut och uppdatera värden i ett spelkort. Alla dessa ska ni nu skriva om som makron. Extern-deklarationerna i `h`-filen ska bort helt och istället ska där finnas makro-definitioner för var och en av funktionerna. Tänk på att makron inte har någon deklarerad returtyp, men resultatet av makrot måste ändå följa den returtyp som funktionerna tidigare haft. Detta gäller även argumenten.

   - `getvisible(card)` — Talar om ifall ett spelkort är synligt eller ej. Resultatet är värdet på `visible`-flaggan i kortet `card`.
   - `setvisible(card, val)` — Sätter `visible`-flaggan i kortet `card` till `val`.
   - `getmark(card)` — Talar om ifall ett spelkort är markerat eller ej. Resultatet är värdet på `mark`-flaggan i kortet `card`.
   - `setmark(card, val)` — Sätter `mark`-flaggan i kortet `card` till `val`.
   - `getsuit(card)` — Ger tillbaka spelkortets valör. Resultatet är värdet på variabeln `suit` i kortet `card`.
   - `setsuit(card, val)` — Sätter spelkortets valör, det vill säga värdet på variabeln `suit` i kortet `card` till `val`.
   - `getrank(card)` — Ger tillbaka spelkortets värde. Resultatet är värdet på variabeln `rank` i kortet `card`.
   - `setrank(card, val)` — Sätter spelkortets värde, det vill säga värdet på variabeln `rank` i kortet `card` till `val`.
   - `getnext(card)` — Ger tillbaka nästa kort i högen. Resultatet är värdet på variabeln `next` i kortet `card`.
   - `setnext(card, val)` — Sätter nästa kort i högen. Värdet på variabeln `next` i kortet `card` sätts till `val`.
   - `isempty(pile)` — Svarar på om korthögen `pile` är tom eller om det finns kort i den. Resultatet är `false` om det finns kort i högen och `true` om den är tom.

När denna uppgift är klar kan ni plocka bort filen `COURSE_cardpile.o`. Ni har nu ersatt alla funktioner i den. Det ska inte heller finnas några `COURSE_`-prefix kvar i `klondike.c`, `cardpile.c` eller `cardpile.h`.

Inlämning

Källkodsfilen `cardpile.c` med de två nya funktionerna `initdeck` och `cleanupdeck`, och källkodsfilen `cardpile.h` med makrodefinitioner för `setvisible`, `getvisible`, `setmark`, `getmark`, `setsuit`, `getsuit`, `setrank`, `getrank`, `setnext`, `getnext` och `isempty` skall lämnas in via ePost senast Tisdag 6/3 2007 kl. 13:10. Filen skall även innehålla korrekta lösningar från uppgift ett, två och tre.

Appendix A

Slumptal

Funktionen för att generera slumptal 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

Standardbiblioteket

ANSI C har ett väldefinierat standardbibliotek med 15 delar för olika typer av funktioner. Dessa bibliotek kan man förvänta sig att de alltid finns – om de inte gör det har kompilatorn inte stöd för hela ANSI C vilket är ganska ovanligt men förekommer på vissa inbyggda system. Alla konstanter som definieras i dessa bibliotek har samma namn oavsett plattform och kompilator, men deras värden kan variera. Vill man skriva plattformsoberoende program är det alltså viktigt att använda dessa konstanter istället för att ange numeriska värden direkt i koden. För att komma åt funktionerna i dessa inkluderar man respektive `h`-fil med till exempel "`#include <stdlib.h>`".

`assert` Felsökning och dignostik

`ctype` Funktioner för att få information om tecken

`errno` Felkoder som returneras av en del biblioteksfunktioner

`float` Implementationsbestämda gränser för flyttal

`limits` Implementationsbestämda gränser

`locale` Landsspecifik information

`math` Matematiska funktioner

`setjmp` Icke-lokala hopp

`signal` Signaler

`stdarg` Variabla argumentlistor

`stddef` Standarddefinitioner som `NULL`, `size_t` med flera

`stdio` In- och utmatning, filhantering

`stdlib` Konvertering, minnesallokering

`string` Sträng- och minneshantering

`time` Tid och datum

Appendix C

Operatorer och sanningstabeller

Teckenförklaring

v En variabel av valfri typ.

e Ett uttryck av valfri typ.

n Ett tal. Till exempel av typen `char`, `int`, `float` med flera.

p En pekare. Till exempel av typen `char*`, `int*`, `float*` med flera.

t Det logiska värdet SANT.

f Det logiska värdet FALSKT.

b Ett logiskt värde, d.v.s. **t** eller **f**.

C.1 Unära operatorer

`+`	Unärt plus	`+e`
`-`	Unärt minus	`-e`
`&`	Adressen av	`&v`
`*`	Indirektion	`*p`
`~`	1-komplement	`~n`
`!`	Logisk negation	`!b`

```
n++
n--
++n
--n
```

C.2 Binära operatorer

`=`	Tilldelning	`v = e`
`+`	Addition	`e + e`
`-`	Subtraktion	`e - e`
`*`	Multiplikation	`e * e`
`/`	Division	`e / e`
`%`	Restdivision (modulo)	`e % e`
`<`	Mindre än	`e < e`
`>`	Större än	`e > e`
`<=`	Mindre än eller lika med	`e <= e`
`>=`	Större än eller lika med	`e >= e`
`==`	Lika	`e == e`
`!=`	Inte lika	`e != e`
`&&`	Logisk OCH	`e && e`
`||`	Logisk ELLER	`e || e`
`&`	OCH på bitnivå	`e & e`
`|`	ELLER på bitnivå	`e | e`
`^`	XOR på bitnivå	`e ^ e`
`<<`	Bit shift åt vänster	`e << e`
`>>`	Bit shift åt höger	`e >> e`

C.3 Sanningstabeller

Inte klart ännu...

Appendix D

Teckentabell ISO-8859-1

Detta är ISO-8859-1 som den definierades av IANA 1992 för trafik på internet. Denna teckentabell skall enligt definitionen användas av all HTTP-trafik med MIME-typ som inleds med "`text/`". Notera att denna tabell skiljer sig från Windows-1252 som används av trasiga operativsystem.

Denna teckentabell kallas ofta ASCII-tabell vilket bara är halva sanningen, mer precist den första halvan. Tecknen 0-127 följer den tabell som ANSI föreslog på 60-talet under namnet American Standard Code for Information Interchange (ASCII).

ISO-8859-1 inkluderar ECMA-94 Latin Alphabet No 1 (Latin-1) som även är känd under namnet ISO 8859-1 (notera bindestrecket). Skillnaden mellan ISO-8859-1 och ECMA-94 är styrkoderna med teckenkod 0-31 och 128-159. Dessa positioner saknar definition i ECMA-94 där de endast är reserverade för styrkoder. Den betydligt äldre ASCII definierar styrkoderna för positionerna 0-31 och trots (eller kanske tack vare) att ECMA-94 inte innehåller någon definition av dessa koder har de alltså överlevt till ISO-8859-1.

Dec	Hex	Binär	Tecken – Information	Dec	Hex	Binär	Tecken – Information
0	00	00000000	`CTRL-@` – NUL (`'\0'`) (Null prompt)	128	80	10000000	– PAD (Padding)
1	01	00000001	`CTRL-A` – SOH (Start of heading)	129	81	10000001	– HOP (High octet preset)
2	02	00000010	`CTRL-B` – STX (Start of text)	130	82	10000010	– BPH (Break permitted here)
3	03	00000011	`CTRL-C` – ETX (End of text)	131	83	10000011	– NBH (No break here)
4	04	00000100	`CTRL-D` – EOT (End of transmission)	132	84	10000100	– IND (Index)
5	05	00000101	`CTRL-E` – ENQ (Enquiry)	133	85	10000101	– NEL (Next line)
6	06	00000110	`CTRL-F` – ACK (Acknowledge)	134	86	10000110	– SSA (Start of selected area)
7	07	00000111	`CTRL-G` – BEL (`'\a'`) (Bell)	135	87	10000111	– ESA (End of selected area)
8	08	00001000	`CTRL-H` – BS (`'\b'`) (Backspace)	136	88	10001000	– HTS (Character tabulation set)
9	09	00001001	`CTRL-I` – HT (`'\t'`) (Horizontal tab)	137	89	10001001	– HTJ (Character tabulation with justification)
10	0a	00001010	`CTRL-J` – LF, NL (`'\n'`) (Linefeed, New line)	138	8a	10001010	– VTS (Line tabulation set)
11	0b	00001011	`CTRL-K` – VT (`'\v'`) (Vertical tab)	139	8b	10001011	– PLD (Partial line down)
12	0c	00001100	`CTRL-L` – FF, NP (`'\f'`) (Formfeed, New page)	140	8c	10001100	– PLU (Partial line up)
13	0d	00001101	`CTRL-M` – CR (`'\r'`) (Carriage return)	141	8d	10001101	– RI (Reverse line feed)
14	0e	00001110	`CTRL-N` – SO (Shift out)	142	8e	10001110	– SS2 (Single shift 2)
15	0f	00001111	`CTRL-O` – SI (Shift in)	143	8f	10001111	– SS3 (Single shift 3)
16	10	00010000	`CTRL-P` – DLE (Data link escape)	144	90	10010000	– DCS (Device control string)
17	11	00010001	`CTRL-Q` – DC1 (Device control 1, XON)	145	91	10010001	– PU1 (Private use 1)
18	12	00010010	`CTRL-R` – DC2 (Device control 2)	146	92	10010010	– PU2 (Private use 2)
19	13	00010011	`CTRL-S` – DC3 (Device control 3,XOFF)	147	93	10010011	– STS (Set transmit state)
20	14	00010100	`CTRL-T` – DC4 (Device control 4)	148	94	10010100	– CCH (Cancel)
21	15	00010101	`CTRL-U` – NAK (Negative aclowledge)	149	95	10010101	– MW (Message waiting)
22	16	00010110	`CTRL-V` – SYN (Synchronous idle)	150	96	10010110	– SPA (Start of protected area)

Dec	Hex	Binär	Tecken – Information	Dec	Hex	Binär	Tecken – Information
23	17	00010111	CTRL-W – ETB (End of transmission block)	151	97	10010111	– SPE (End of protected area)
24	18	00011000	CTRL-X – CAN (Cancel)	152	98	10011000	– SOS (Start of string)
25	19	00011001	CTRL-Y – EM (End of medium)	153	99	10011001	– SGCI (Single graphic caracter introducer)
26	1a	00011010	CTRL-Z – SUB (Substitute)	154	9a	10011010	– SCI (Single caracter introducer)
27	1b	00011011	CTRL-[– ESC (Escape)	155	9b	10011011	– CSI (Control sequence introducer)
28	1c	00011100	CTRL-\ – FS (File separator)	156	9c	10011100	– ST (String terminator)
29	1d	00011101	CTRL-] – GS (Group separator)	157	9d	10011101	– OSC (Operating system command)
30	1e	00011110	CTRL-^ – RS (Record separator)	158	9e	10011110	– PM (Privacy message)
31	1f	00011111	CTRL-_ – US (Unit separator)	159	9f	10011111	– APC (Application program command)
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

Dec	Hex	Binär	Tecken – Information	Dec	Hex	Binär	Tecken – Information
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
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

Dec	Hex	Binär	Tecken – Information	Dec	Hex	Binär	Tecken – Information
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 E

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 referens till en förteckning över de vanligaste koderna finns på kurshemsidan.

Appendix F

Verktyg för felsökning

F.1 gdb

The GNU Debugger, gdb, är ett av de mer kompetenta felsökningsverktygen som existerar. Man kan stoppa programkörningen precis när man vill och titta på innehållet i variabler, register, stacken och mycket mer. Man kan stega fram i programmet instruktion för instruktion och se exakt vad som händer. Har man ett program som kraschar så kan gdb direkt tala om exakt var i koden det hände och man kan lätt se vad olika variabler innehöll vid kraschen. gdb ger även möjligheten att provköra enskilda funktioner i programmet och är därför ett utmärkt testverktyg även för program som inte har några uppenbara fel.

gdb är ett verktyg som används från ett terminalfönster, det är helt textbaserat. Det finns ett grafiskt skal till gdb som heter Data Display Debugger (ddd) som jag inte kommer att beskriva närmare här. För att kunna använda gdb med sitt program måste man skicka med en extra flagga till gcc, -ggdb. Resten av denna sektion visar en testkörning där jag använder de vanligaste kommandona i gdb för att titta lite på insidan av klondike.

Vi börjar med att kompilera programmet. Flaggan -ggdb skickas med på kommandoraden. Därefter startas gdb. Namnet på programmet vi ska felsöka i skickas med som argument till gdb. Alla rader i exemplet nedan som inleds med "(gdb)" är ställen där jag skrivit in något kommando till gdb, "(gdb)" är gdb:s prompt.

```
bash$ gcc -ggdb klondike.c COURSE_cardpile_x86.o -o klondike
bash$ gdb klondike
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db library
"/lib/tls/i686/cmov/libthread_db.so.1".
```

gdb skriver ut sitt välkomstmeddelande och vi startar Klondike med kommandot run.

```
(gdb) run
Klondike 2007    Slumpfrö: 1171990446
Välj rad att markera med pilen.
Använd 'u' - upp, och 'n' - ner för att styra.
'm' markerar kort eller flyttar markerade kort.
'k' lägger ut nya kort från kortleken.
Tryck 'q' för att avsluta

Guide: H = Hjärter  D = Ruter  C = Klöver  S = Spader
       A = Ess  J = Knekt  Q = Dam  K = Kung

-->[C: 2][****][****][****][****][****][****]
   [D: K][****][****][****][****][****]
   [S: A][****][****][****][****]
   [D: 4][****][****][****]
   [S: 7][****][****]
   [C: K][****]
   [C: 6]


   [****]     [    ]    [    ]    [    ]    [    ]

Program received signal SIGINT, Interrupt.
0xffffe410 in __kernel_vsyscall ()
```

Klondike startar och skriver ut sin meny och spelkorten. Spelet väntar sedan på att vi ska skriva in något kommando. Det gör vi inte. Istället trycker vi på `CTRL-c` för att avbryta programmet. Normalt när man stoppar program på det sättet kommer man tillbaka till terminalen, men nu har `gdb` tagit över och vi kommer istället tillbaka till `gdb`:s prompt.

```
(gdb) break klondike.c:161
Breakpoint 1 at 0x8048979: file klondike.c, line 161.
```

Vi sätter nu en brytpunkt i programmet. En brytpunkt är ett ställe i koden där vi vill att programmet ska avbrytas när programmkörningen når dit. Vi anger denna med filnamn och radnummer i källkodsfilen. Rad 161 i filen `klondike.c` är i början av funktionen `oktostore`. Man kan även skriva till exempel ett funktionsnamn istället för radnummer om man vill att körningen ska stanna när den når en viss funktion. Sedan fortsätter vi programmkörningen med kommandot `continue`. Klondike kommer då att återuppta sin körning precis där den var då vi avbröt med `CTRL-c`.

```
(gdb) continue
Continuing.
m

Breakpoint 1, oktostore (card=0x804c158, storage=0x0) at klondike.c:161
161         if (COURSE_isempty(storage) == true && COURSE_getrank(card) == ACE)
```

Nu står klondike och väntar på att vi ska skriva något igen. Vi skriver ett 'm' för att markera ett kort i spelet. Detta kommer att orsaka anrop till `oktostore`. Brytpunkten aktiveras och vi kommer tillbaka till `gdb`. Rad 161 skrivs ut för att visa var i programmet den stannade.

```
(gdb) list
156     /*
157      * Kollar om vi kan lägga det givna kortet på den givna lagerhögen.
158      */
159     static bool oktostore(card_t* card, card_t* storage)
160     {
161        if (COURSE_isempty(storage) == true && COURSE_getrank(card) == ACE)
162           return true;
163
164        if (COURSE_isempty(storage) == false &&
165            (COURSE_getrank(card) == COURSE_successor(COURSE_getrank(storage))) &&
```

Med kommandot `list` kan vi se lite av källkoden runt raden där programmkörningen står just nu.

```
(gdb) print card
$1 = (card_t *) 0x804c158
(gdb) p *card
$2 = {suit = CLUBS, rank = TWO, visible = true, mark = false, next = 0x804c470}
```

Vi kan nu börja titta lite i minnet och se vad våra variabler innehåller. `print` är ett av de kommandon man använder mest i `gdb`. Vi skriver ut innehållet i variabeln `card` och får veta att `card` är en pekare till ett spelkort (`card_t*`) och att den innehåller adressen `0x804c158`[1].
Adressen säger kanske inte så mycket så vi följer pekaren och skriver ut det som den pekar på. Vi skriver på exakt samma sätt som i C. Nu får vi se alla fälten i spelkortet. `gdb` känner till våra egendefinierade konstanter och kan därför använda namnen `CLUBS` och `TWO` i utskriften.

```
(gdb) step
164        if (COURSE_isempty(storage) == false &&
```

Vi stegar fram ett steg i programmet med kommandot `step`. Kommandot som utförs nu är det vi stannade innan tidigare, alltså `if`-satsen på rad 161. Rad 164 som skrivs ut nu är nästa rad att köra.

```
(gdb) print COURSE_isempty(storage)
$3 = 1
```

Vi använder kommandot `print` för att anropa funktionen `COURSE_isempty`. Vi kan anropa vilka funktioner som helst i programmet och undersöka deras resultat med hjälp av `print`. Nu fick vi svaret 1 som betyder `true`, högen vi tittade på var alltså tom.

```
(gdb) next
169        return false;
```

Här använder vi `next` för att stega fram ett steg i programmet. Nu utfördes alltså rad 164, nästa rad som kommer att köras är 169. Skillnaden mellan `step` och `next` är att `step` kommer att gå in i funktioner och stega igenom även dessa medan `next` kommer att utföra funktionsanrop i ett svep utan att stega in i dem.

```
(gdb) bt
#0  oktostore (card=0x804c158, storage=0x0) at klondike.c:169
#1  0x08048a33 in movetostore (pile=0xbfcdcc88, storage=0xbfcdcca4)
    at klondike.c:183
#2  0x08048c2d in move (pile=0xbfcdcc88, storage=0xbfcdcca4, curs=0)
    at klondike.c:254
#3  0x080496af in main (argc=1, argv=0xbfcdcdd4) at klondike.c:435
(gdb) f 2
#2  0x08048c2d in move (pile=0xbfcdcc88, storage=0xbfcdcca4, curs=0)
    at klondike.c:254
254           if (COURSE_isempty(pile[curs]) == false &&
(gdb) p COURSE_isempty(pile[curs])
$4 = 0
```

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) quit
```

F.2 `lint`

Appendix G

Kodkonventioner

Följande kodkonventioner är hämtade från verkligheten. De används av utvecklarna i minnesgruppen hos JRockit - BEA Systems. Diskussioner om kodkonventionerna förs kontinuerligt och uppdateras efter behov. Konventionerna är daterade 1 feb. 2007. Vissa exempel har "anonymiserats" eller tagits bort för att inte offentliggöra delar av JRockits källkod.

Konventionerna förespråkar inte strikt ANSI C som vi kan se på att de till exempel tillåter enradskommentarer med `//`. Det förs även diskussioner om man ska börja tillåta variabeldeklarationer av C99-stil, det vill säga inne i kodblock, och hurvida samtliga argument till funktioner måste beskrivas i kommentarerna.

Allmänt

- Alla regler kan brytas mot, om man har goda skäl och minst två personer (utvecklaren + kodgranskaren) håller med om, och kan försvara det.
- Inga regler ska orsaka ett omedelbart refaktoriserande, utan ska införas efterhand som man skriver nytt eller uppdaterar gammal kod.
- När man gör en förändring, så ska den genomföras fullt ut så långt det är vettigt. Försökt vara konsekvent, och städa gärna upp i efterhand om du inte hinner med att införa det nya överallt i första vändan.

Kodformattering

- Använd space-tecken istället för tab-tecken. De flesta editorer går att ställa in så att de gör så.
- Måsvingar (`{}`) kan placeras antingen på samma rad (K&R/Java-stil) eller på en egen rad (BSD-stil). En funktion är minsta enheten för koherent måsvinge-standard - om du ändrar i en funktion så måste du antingen anpassa dina ändringar till existerande stil, eller omvandla alla till din stil.
- I en funktionsdeklaration ska alltid första måsvingen stå ensam först på en rad.
- Man ska alltid använda måsvingar (`{}`) i `if`-satser etc.
- I funktionsdeklarationer kan man ha ett argument per rad om man har många argument och man tycker det ökar läsligheten, men annars inte.
- Efter ett block med variabeldeklarationer kommer alltid en tomrad.
- Asterisken i pekardeklarationer kan binda antingen till typen eller variabelnamnet, dvs antingen "`void* foo`" eller "`void *foo`". Försök följ filens konvention. För nya typer, inför gärna en FooP så kringgår du problemet.

- Det är tillåtet att deklarera flera variabler på en rad, typ "`int i,j;`", men undvik gärna.
- Det är inte tillåtet att deklarera flera variabler på en rad, om en av dem är en pekare, d.v.s. "`int* a, b;`" är inte accepterat.
- Vid definition och deklaration av funktioner som inte tar några parametrar, skall void alltid anges så man får en korrekt prototyp, dvs "`void foo(void);`" snarare än "`void foo();`".

Kommentarer

- Använd `//` och undvik `/* ... */` inne i funktioner. Undantaget är långa utläggningar ("C-uppsatser"), som kan få ha `/* ... */`. Kom ihåg att man kan "kommentera ut" långa kodblock med "`#if 0 ... #endif`".
- Alla globala/"modulglobala" variabler och typdefinitioner ska dokumenteras, antingen med en Javadoc-style kommentar före (`/** ... */`) eller en `//` efteråt.
- Alla icke-uppenbara delar i koden ska dokumenteras. Det innefattar icke-uppenbara statiska variabler och funktioner, alla sidoeffekter på parametrar, och icke-uppenbara returvärden.

Namngivning

- Namngivning är viktigt, och kan vara svårt att få till rätt, men är värt att anstränga sig för. Bra namngivning kan säga mer än kommentarer.
- Alla namn ska vara CamelCase (undantaget defines). (Ang. förkortningar, se nedan.)
- Samtliga funktioner ska ha ett modulgemensamt prefix. T.ex. modulen Foo har prefixet foo.
- Såväl privata (statiska) som publika funktioner ska ha samma namnprefix.
- Globala/"modulglobala" variabler ska ha samma namnprefix som funktionerna i filen.
- Typer (typedefs, enums, structs, etc) ska ha samma namnprefix som funktioner, men med första bokstaven versal.
- Om man har ett prefix typ "foo", så bör funktionen heta "fooDoThis". Jfr med det objektorienterade namnet "Foo.doThis".

Boolska uttryck

- "`foo == TRUE`" är trasigt, och ska inte användas. Punkt.
- "`foo == FALSE`" ska då av symmetriskäl undvikas, och "`!foo`" användas istället.
- Bara boolska uttryck ska finnas i `if`- och `while`-satsers villkor. "`myPtr`" är inget boolskt uttryck. "`myPtr != NULL`" är det, däremot.
- Tilldelning och jämförelse i samma uttryck bör undvikas, d.v.s satser som denna: "`if ((n = getNum()) == 0) ...`". I `while`-slingor kan det behövas för att undvika kodduplicering och kan där godtas om utvecklaren önskar det, men i `if`-satser kan och ska alltid tilldelningen placeras före jämförelsen.

Kodplacering

- Makron ska undvikas i görligaste mån - använd inline-funktioner istället.
- Globala/ "modulglobala" variabler och typdefinitioner placeras alltid överst i filen.
- Funktioner ska om möjligt rymmas på en skärmsida. Ett krav är att slingor eller andra block alltid ska rymmas på en skärmsida. Långa funktioner som bara gör en mängd anrop i följd är undantagna från kravet.
- Statiska hjälpfunktioner placeras gärna precis före den som använder den, speciellt om man brutit ut funktionalitet från en funktion. De ska åtminstone ligga framför funktionen, så man slipper en extra deklaration om det är möjligt, och gärna "nära".
- En funktion ska vara antingen modulintern och därmed `static`-deklarerad, eller publik och därmed deklarerad i en headerfil.
- I en headerfil får endast header-filer av typen `_types.h` eller `_structs.h` includeras, för att undvika rekursiva inkluderingskedjor.
- Includes av headerfiler ska göras i strikt alfabetisk ordning, baserat på filens sökväg. Om include-listan är lång kan gärna tomrader läggas in mellan toppnivå-katalogerna. Denna placering gäller även c-filens "egen" headerfil.

Programflöde

- Goto ska undvikas, men kan i undantagsfall accepteras om alla alternativ är sämre.
- Undvik callbacks, om möjligt.
- Undvik globala funktionspekare, om möjligt.
- Gör programflödet så tydligt som möjligt. T.ex. bör man undvika flera nivåer av villkorsblock och slingor, om möjligt. Modellera om logiken eller bryt ut delar till egna funktioner.
- break-satser ska undvikas i görligaste mån. Om det är nödvändigt att använda, se till att det är så enkelt och tydligt som möjligt. För oändliga slingor, använd `for (;;)` istället för `while (TRUE)`.
- for ska användas endast på uppräkningsmängder. Är avslutningsvillkoret eller stegvillkoret mer komplext, använd while.
- Lokala variabler ska deklareras i det innersta scope som är möjligt.
- Tilldelning till lokala variabler kan ske antingen vid deklaration, eller vid användning - använt sunt förnuft för att avgöra vilket som känns bäst.

Koddesign

- `size_t` ska användas för typer som är relaterade till minnesstorlek, främst för sådant som mäter antal bytes av minnet, men även för räknare av sådant som är O(adressrymd).
- Vi bör undvika globala variabler om möjligt.
- Globala variabler ska alltid accessas genom getters och setters. Enda möjliga undantaget är i filen de definieras i.
- Funktioner får inte ha icke-uppenbara sidoeffekter. T.ex. ska getters och setters bara göra get/set.

- Tanken bör vara ”en funktion - en sak att göra”.
- All död/oanvänd kod ska plockas bort när den upptäcks - vill man ha den igen finns den i versionshanteringssystemet. Är det test-/debugkod, så ska det finnas tester som kör den (och då är den per definition inte oanvänd.)

Kodgranskning

- Om kodgranskaren ställer en fråga, ska den egentligen besvaras genom att koden i fråga skrivs om eller kommenteras.
- Idealet vid kodgranskning är att två personer kodgranskar, en som verkligen behärskar koden, och en som inte är alls insatt i den.
- Vid större förändringar bör vi göra inkrementell kodgranskning.
- Vi bör gå igenom gamla filer i en kontinuerlig kodgranskningsprocess. Två slumpvis ihopparade personer ska sitta en timme per vecka med en fil.