TOBIAS WRIGSTAD

# C: STACK, HEAP, *, &

IOOP/M 2011

# *Ett enkelt stackexempel*

Syftet med det här lilla häftet är att driva hem några poänger kring skillnaderna mellan *stack* och *heap*, adresstagningsoperatorn & och avrefereringsoperatorn ∗.

Vi börjar med ett enkelt kodexempel:

```
int x = 42;
int *y = &x;
int *z = NULL;
z = y;
(*z)++;
y++;
```

Vilka värden har variablerna x, y och z? Svaret är att värdet på x är 43 men att värdena på y och z är okända[1].

[1] Däremot så vet vi att om värdet på z är heltalet $n$, så är värdet på y $n + 4$, givet att storleken på en pekare är 4 bytes.

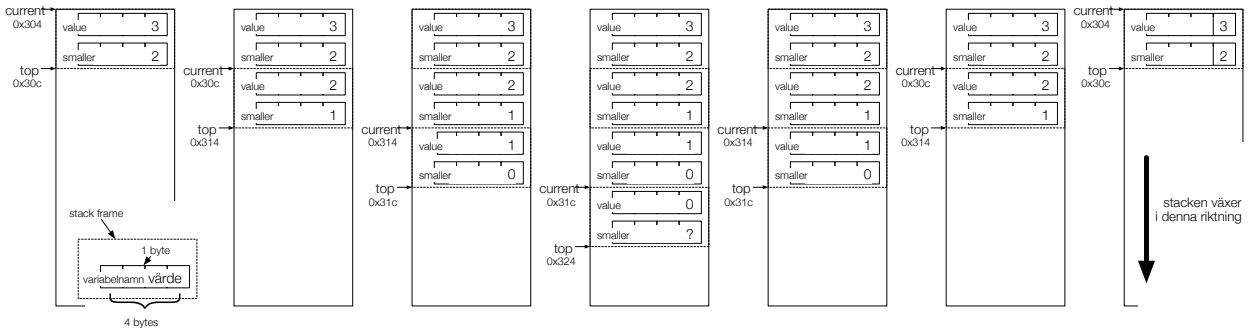
Men låt oss börja från början. För enkelhetens skull kan vi tänka oss att ett C-program har tillgång till två sorters minne: *stackminne* och *heapminne*. Stacken är det minnesutrymme som används för att lagra värdena i lokala variabler i funktioner, så kallade automatiska variabler. Betrakta följande funktion:

```
void stupid(int value) {
  if (value) {
    int smaller = value - 1;
    stupid(smaller);
  }
}
```

Funktionen har två lokala variabler, `value` och `smaller`, båda av typen **`int`**. Låt oss anta att en int är 4 bytes. Det betyder att vi kommer att behöva 8 bytes på stacken för värdena i `value` och `smaller`.[2]

[2] Det är en liten förenkling – vissa andra data kan behövas, och smarta kompilatorer gör optimeringar som trollar bort variabler som inte behövs, som t.ex. `smaller` här. Vi bortser från sånt.

Funktionen `stupid` är rekursiv; ett anrop `stupid(3)` kommer att leda till att funktionen anropar sig själv ytterligare tre gånger; varje funktion har en egen lokal variabel `value` vars värde är ett mindre än den anropande funktionens `value`-värde. Varje gång `stupid` anropas behövs ytterligare 8 bytes för att hålla värdena i dess `value` och `smaller`-variabler. Vi säger att varje funktionsanrop leder till att ny *stack frame* push:as på stacken som innehåller det minnesutrymmen




Figur 1: En bild av stacken vid anrop till `stupid(3)`.

där värdena i de lokala variablers, i detta fall `value` och `stack`, lagras.

När en funktion returnerar (är klar) behövs inte dess stack frame längre, varvid den pop:as, försvinner, från stacken.

Varje funktion har en stack frame som vi skulle kunna uttrycka som en vanlig `struct`. I fallet `stupid` skulle den se ut så här:[3]

[3] Det är inte så här det är implementerat under huven, men det är en utmärkt mental modell.

```
struct stack_frame_for_stupid {
  int value;
  int smaller;
};
typedef struct stack_frame_for_stupid *SFFS;
```

Figur 1 visar en översikt av stacken för de totalt fyra anrop av `stupid` som blir resultatet av `stupid(3)`. Varje stack frame är ritad som en rektangel med variablerna och deras värden klart och tydligt angivna. Stacken växer nedåt och figuren skall läsas från vänster till höger.

Som synes flyttas pekarna `top` och `current` hela tiden fram – 8 bytes i taget – vid rekursiva anrop, och flyttas tillbaka när funktioner returnerar. Dessa pekare finns inte på något sätt tillgängliga i C, utan är till för vår förståelse här. Notera att ursprungsvärdet på `top` är `0x0304`, vilket är den adress i minnet där stacken råkade börja denna körning av programmet. Varje rekursivt anrop växer `top` och `current` med 8 bytes.

### *En variabel har en adress*

Variablerna i C-koden, `value` och `smaller` är namn som vi väljer väl som programmerare eftersom de hjälper oss att läsa koden. När programmet väl kör så finns inte dessa namn kvar[4], utan de har ersatts av adresser[5].

I fallet `stupid` så är adressen för `value` adressen till den aktuella stack frame:en + 0, eftersom det är den första variabeln på stacken. Adressen för `smaller` är **`sizeof(int)`** bytes efter adressen till `value`[6].

[4] De kan visserligen finnas i den symboliska information som sparas för debugging när sådana flaggor är satta.

[5] En förenkling som inte är helt sann, men duger för oss här.

[6] Pekararitmetiskt räknas detta inte ut med `value + sizeof(int)`, utan med `value+1`. (Varför!?)

Man kan tänka på `current`-pekaren i Figur 1 som en **`char`**`*`-pekare. Vi kan typomvandla den till en **`struct`** `stack_frame_for_stupid*`; då skulle åtkomst till `value` bli `((SFFS) current)->value` alternativt `*(current+0)` och åtkomst till `smaller` bli `((SFFS) current)->smaller` alternativt `*(current+`**`sizeof(int)`**`)`.

När man tar adressen till en variabel, t.ex. `&value` får man tillbaka den adress i minnet där det värde som variabeln innehåller finns lagrat. T.ex. skulle `&value` i den tredje delfiguren från vänster i Figur 1 vara `0x314` och `&smaller` på samma stackframe vara `0x318`.

Det betyder att tilldelningen `smaller = value - 1` som sker i funktionen kommer att läsa 4 bytes[7] från `0x314` tolka dem som ett heltal, subtrahera 1 från talet, och sedan spara resultatet på minnesadressen `0x318`.

[7] Igen, givet att en **`int`** är 4 bytes.

Varje variabel innehåller alltså ett värde. Det värdet *ligger på stacken*, d.v.s. är lagrat i en stack frame. Varje variabel har också en adress, som avser den plats i minnet där dess värde ligger[8]. Själva variabelnamnet är en bekvämlighet för oss programmerare och är inte sparat i programmet[9]; programmet kan inte inspektera en variabels namn, etc.

[8] Igen, detta är en förenkling som inte är helt sann, men duger för oss här.

[9] Igen, namnen kan vara sparade för debugging.

### *Återbesök av vårt exempel*

Låt en icke-namngiven funktion ha följande funktionskropp. Denna funktion har tre lokala variabler; med samma storleksantaganden som tidigare har den en stack frame som är 12 bytes stor.

```
int x = 42;
int *y = &x;
int *z = NULL;
z = y;
(*z)++;
y++;
```

Notera att `x` är ett heltal, medan `y` och `z` är pekarvariabler – de innehåller adresser till platser i minnet. I detta fall kommer dessa adresser att avse platser på stacken (d.v.s., `y` och `z` är pekare till stacken). Värdet i t.ex. `y` är alltså en adress – om vi vill läsa värdet som `y` *pekar på* måste vi använda *avrefereringsoperatorn*, `*`, och skriva `*y` vilket betyder ”gå till den minnesadress som är lagrad i `y`, läs **`sizeof(int)`** bytes, och behandla dem som en **`int`**.”[10]

[10] Eftersom `y` är en pekare till just en **`int`**.

När vi tar adressen till `x` på rad 2 returneras den adress som värdet 42 ligger på. Låt säga att att det var `0x112`. Innehållet i `y` är då adressen `0x112`, vilket vi normalt ritas som en pil till den platsen. Givet detta antagande kan vi nu besvara frågorna på sid. 1, ”vad är värdena på `x`, `y` och `z`?”

Svaret denna gång är att `x` fortfarande är 43, `z` är `0x112` och `y` är `0x116`. Att `y` är `0x116` och inte `0x113` kommer av C:s definition av pekararitmetik. När man inkrementerar eller adderar till en pekare görs detta i *steg om n* där *n* är storleken av den typ som pekas ut. Eftersom `y` pekar på en **`int`**, blir `y++` likvärdigt med `0x116`. Detta förenklar programmeringen och minskar risken för att vi råkar läsa t.ex. de sista tre tecknen av en **`int`** och det första av en annan, vilket vore resultatet om `y` var `0x113` och vi försökte läsa `*y`.[11]

[11] Detta besvarar frågan i fotnot 6.

*Om du är med så här långt borde du förstå att även om `y` och `z` pekar på samma plats i minnet (iallafallefter rad 4 och innan den sista raden) så är de olika variabler som tar upp var sina 4 bytes i minnet, och en tilldelning till den ena förändrar inte den andra.*

### *Sammansatta datatyper på stacken*

Notera att alla lokala variabler lagrar sina data på stacken. Ponera följande kod:

```
char *s1 = "Hello, world";
char s2[13];
strcpy(s2, s1);
```

Hur stor är stack frame:en för en funktion utan parametrar och med denna kropp?

Svaret är 17 bytes[12], eftersom C allokerar utrymmet för `s2` på stacken. Den första raden reserverar minnet för strängen `"Hello, world"` i programmets statiska dataarea[13], och sedan utrymme för en pekare på stacken som får peka dit. Den andra raden reserverar 13 tecken på stacken, *men ingen pekare*.

[12] Igen, givet att en pekare är 4 bytes, och att ingen avrundning sker, t.ex. till 20 som är närmaste multipel av 4, vilket kan antas vara storleken på ett ”ord” i vår hypotetiska dator. (Varför?!)

[13] Som är en ytterligare del av minnet som varken är stack eller heap, och som faller litet utanför denna text.

Men hur kan då `s2` fungera som en **`char`***? Minns att variabler “försvinner” vid kompilering, och att deras värden normalt har en adress. Det betyder att innebörden av variabeln `s1` för vårt program är en adress till en plats på stacken där det ligger en adress till en **`char`**, och att innebörden av `s2` är en adress till en plats på stacken där det ligger en **`char`**. Man kan tilldela `s1 = s2`, men inte `s2 = s1` av just denna anledning. Den första tilldelningen sparar adressen till det första tecknet i `s2` i `s1`. Den andra tilldelningen är nonsens eftersom `s1` är variabel som innehåller en adress och `s2` är en variabel som innehåller en array av **`char`**s.

Samma situation uppstår med struktar. Givet följande strukt:

```
typedef struct _point {
  int x, y;
} point, *Point;
```

kan vi skriva följande funktionskropp[14]:

```
point p = {0, 0};
Point pp = &p;
```

[14] Notera syntaxen för att initiera värdena på `x` och `y` i `p`. Man man även ange posternas namn vid initiering: `point p = {.y=1, .x=2 };`

där den första variabeln tar upp 8 bytes i minnet[15], medan den andra tar upp 4 bytes. Notera att `p` och `pp` avser samma objekt i minnet. När man vill modifiera `y` *i* `p` skriver man `p.y`; när man vill modifiera `y` *via* `pp` skriver man `pp->y`. Punkten är en operator som indexerar sig in i variabel av strukt-typ (`p.y` är likvärdigt med `*(((int*)&p)+1)`[16]) medan ”piloperatorn” (`->`) indexerar sig in i variabler av typen pekare-till-strukt.

*Konsekvent* använt blir följden att `.`-operatorn opererar på data som bara är synligt för den körande funktionen, medan `->`-operatorn opererar på data som kan vara synligt även för andra (t.ex. för att det utpekade värdet ligger på heapen, eller på en annan stack frame). Man kan naturligtvis skriva t.ex. `(*pp).y` istället för `pp->y`, men då *måste avrefereringsoperatorn användas*, vilket igen signalerar att vi manipulerar data via en pekare, och förändringarna kan därför vara synliga utanför den aktuella funktionen.[17]

Värt att notera att C har värdesemantik, vilket betyder att följande kod skapar två *olika* punkter genom att ta en *kopia* av `p` och spara i `q`.

```
point p;
p.x = 10;
p.y = 7;
point q = p; // kopiera hela p in i q
q.x = 5;
assert(p.x == q.x); // failar
```

Samma sak händer när man skickar runt pekarvariabler – själva innehållet i variablerna, minnesadresserna, kopieras. Platsen där de ligger förblir dock orörd.

### *Minneshantering och stacken*

C hanterar minnet på stacken själv; kompilatorn räknar ut varje funktions stack frame-storlek och de faktiska motsvarigheterna till `current` och `top` flyttas fram och tillbaka ”automagiskt” vid varje nytt funktionsanrop eller -retur. Denna minneshantering är snabb och effektiv, men begränsad. Eftersom funktionen tar bort en stackframe när en funktion returnerar är följande kod nonsens:

```
char *bogus() {
  char result[16];
  strcpy(result, "Hello, world");
  return result; /* !!! */
}
```

På raden `!!!` förstörs[18] stack frame:en och därmed `result`.[19] Om vi vill allokera minne som skall ha en godtyckligt lång livstid måste vi allokera på *heapen*.

[15] **`sizeof`**`(point)` = 8, givet våra tidigare antaganden.

[16] Eftersom `y` är det andra fältet i strukten.

[17] C:s syntax visar sålunda tydligt när man följer en pekare och att man därför inte vet i vilken utsträckning den aktuella förändringen kan ses av andra! Läs gärna `->` som ”nu följer vi en pekare (till någonstans i minnet)”.

[18] Egentligen förstörs den inte förrän i och med nästa funktionanrop. Detta kommer nämligen att allokera en ny stack frame som skriver över `result`.

[19] Notera att koden **`char`** `result[] = "Hello, world";` skapar en variabel `result` som är en **`char`**-array med innehållet "Hello, worlddirekt på stacken och alltså undviker `strcpy`.

## *Heapen, mer pekare och adresstagning*

Heapminnet är tillgängligt i C via funktionerna `malloc`, `calloc` och `realloc`. De allokerar alla minne på heapen och returnerar en pekare till detta minne, d.v.s. en minnesadress.

Betrakta följande program-utsnitt:

```
void someFunc() {
  char *hbuffer = malloc(1024);
  char sbuffer[1024];
  ...
}
```

Det allokerar två buffrar av **`char`**s, `hbuffer` som ligger på heapen och `sbuffer` som ligger på stacken. Det är ingen som helst skillnad på de två buffrarna – de är lika stora, etc. – men de ligger på olika platser i minnet. Den viktigaste skillnaden är dock denna: `hbuffer` kommer att finnas kvar i programmet (d.v.s. minnesresursen är låst för annat användande) tills dess att man anropar `free(hbuffer)`, medan `sbuffer` ”försvinner” så fort `someFunc` returnerar.

Eftersom det normalt finns betydligt mindre stackminne än heapminne är det också en god idé att hålla större datastrukturer borta från stacken. I fallet `hbuffer` sparas endast en pekare[20] på stacken istället för hela det allokerade minnesutrymmet.

[20] Typiskt 4 el. 8 bytes.

Funktionen `malloc` har en lista över tillgänligt minne (den s.k. free-listan) och en lista över vilka block som är allokerade. När man allokerar minne med `malloc` sparas en pekare till det allokerade utrymmet i listan över allokerade block, samt en storlek. När man anropar `free` med den minnesadressen som argument frigörs minnet och återförs på free-listan, där det eventuellt slås samman med andra angränsande block för att minska fragmentering.

Ponera följande program-utsnitt:

```
int *x = ...; /* ignorera initieringen */
*x = 5;
```

Nu är (värdet på) `x` en adress i minnet, säg `0x004` för enkelhetsskull, där ett heltal med värdet 5 är lagrat. Värdet på `*x` är 5 – vi tänker oss att `*x` betyder ”gå till minnesplatsen `0x004` och hämta värdet där” istället för `x` som betyder ”adressen `0x004`”. Vi kan använda `&`-operatorn på `x` och få ut *adressen i minnet där `x`:s värde, `0x004`, är lagrat*. Vi kan göra så här:

```
... /* som ovan */
int **y = &x;
```

Variabeln `y` innehåller nu alltså adressen till `x`, som råkar ligga på samma stack frame som `y`.[21] Även om variablerna `y` och `x` båda innehåller adresser har de *inte samma typ*. C håller nämligen reda på

[21] Förmodligen gäller `&y - &x = sizeof(int*) = 4`.

att (värdet på) `y` är en minnesadress på vilken det ligger en annan minnesadress (till en plats där det ligger en **`int`**), medan (värdet på) `x` är en minnesadress på vilken det ligger en **`int`**.

Att skriva t.ex. `&&x` är nonsens eftersom `&`-operatorn tar adressen till ett värde i en variabel eller array, inte till ett "fristående värde". Följande är fullt legalt, och de tre initieringarna av `first`, `second` och `third` visar olika sätt att komma åt adresser till minnesplatser.

```
int ints[] = {1,2,3};
int *first = ints; // pekare till 1:an
int *second = &(ints[1]); // pekare till 2:an
int *third = ints+2; // pekare till 3:an
```

På samma sätt som vi kan komma åt adressen till ett element i en array kan vi också komma åt adressen till ett fält i en strukt.

```
point p;
Point pp = &p; // p och pp avser nu samma point
int *y1 = &(p.y);
int *y2 = &(pp->y);
int *y3 = &((*pp).y);
int *y4 = ((int*)&p) + 1;
int *y5 = ((char*)pp) + sizeof(int);
assert(y1 == y2 && y2 == y3 && y3 == y4 && y4 == y5); // OK
```

Här pekar nu variablerna `y1` till `y5` på `p`-struktens `y`-fält. D.v.s. om startadressen till `p` är `0x400` så blir värdet på t.ex. `y1` = `0x404`. Variabeln `pp` ligger på stacken och innehåller en adress (vilken!?). Var ligger innehållet i `p` – stacken eller heapen?[22]

Man kan se adresstagningoperatorn `&` och avrefereringsoperatorn `*` som varandras motsatser. Den förstnämnda ger alltid en minnesadress där ett värde ligger medan den sistnämnda går från en minnesadress till det värde som ligger på den adressen.

## *Slutkommentarer*

Förhoppningsvis har denna text utrett begreppen stack, heap, pekare och adress ytterligare. Vi har sett att stacken hanteras automatiskt av C, medan heapen hanteras manuellt med `malloc` (etc.) och `free`. Pekare är adresser till platser i minnet; man kan peka på både platser på stacken och på heapen. Avrefereringsoperatorn `*` tillåter oss att enkelt följa en pekare till dess utpekade värde. Adresstagningsoperatorn `&` låter oss ta adressen till ett värde i minnet, t.ex. värdet i en variabel eller adressen till ett specifikt element i en array eller fält i en strukt.

[22] På stacken. Likaså pekar `pp` ut en adress på stacken, `0x400`, samma som `p`.