Assemblerprogrammering för ARM – del 3

Ur innehållet

- Fält och sammansatta typer (poster)
- Pekarvariabler och pekarkonstanter
- Pekararitmetik, operationer på fält

Läsanvisningar:

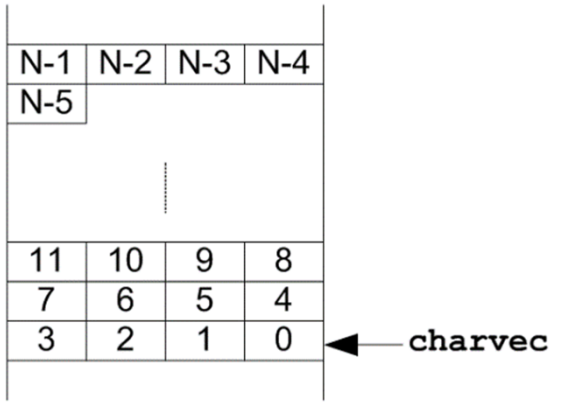
- Arbetsbok kap 2
- Quick-guide, instruktionslistan

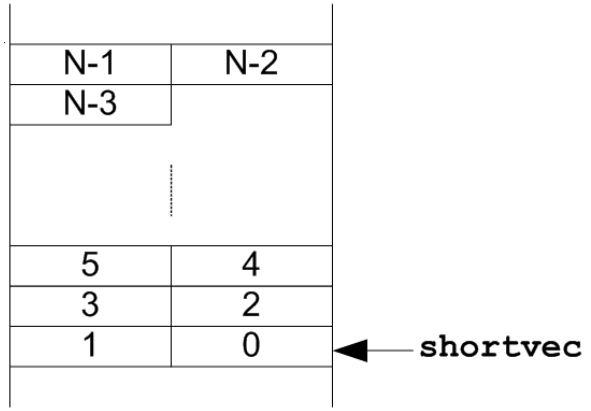
Fält – ”vektorer” – N element, indexeras 0..N-1

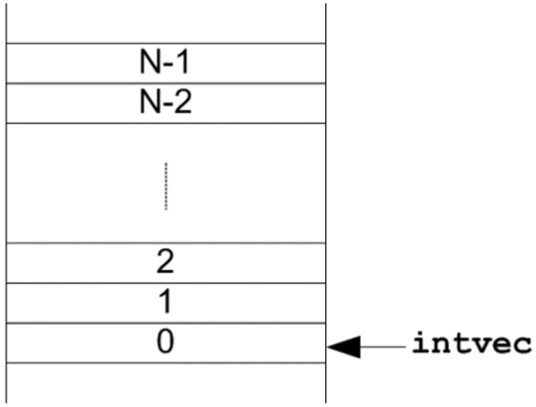
```
char  charvec[N];
sizeof (charvec) = N bytes
```

```
short shortvec[N];
sizeof(shortvec)= N*2 bytes
```

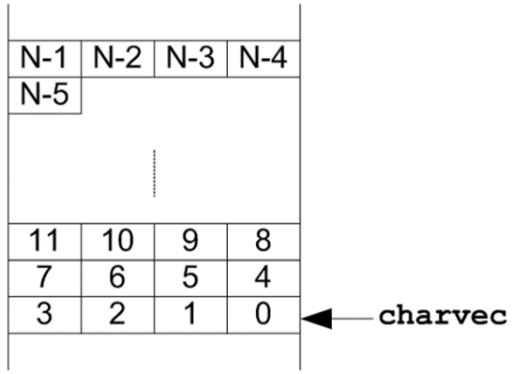
```
int   intvec[N];
sizeof (intvec) = N*4 bytes
```







Adressberäkning och adresseringssätt

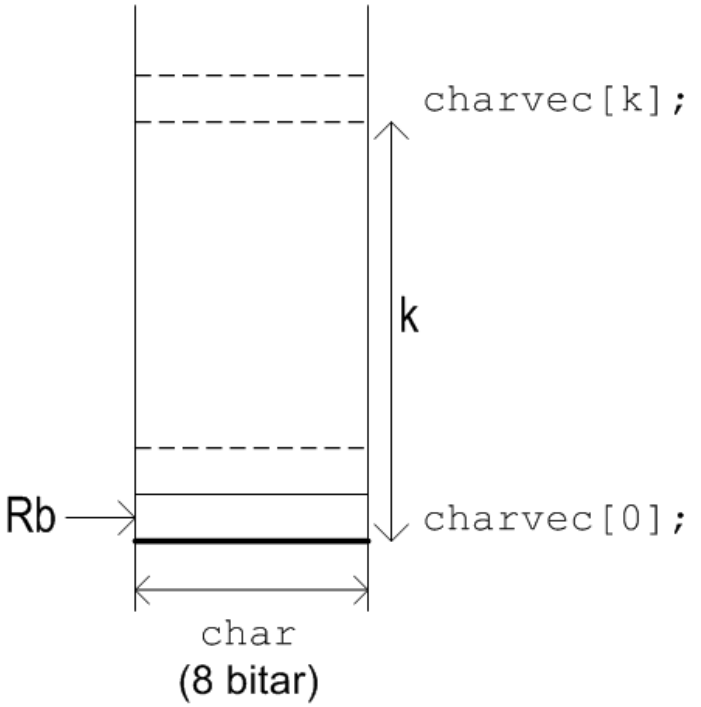



```
const k=?; (0..N-1)
char   charvec[N];

charvec[5];

LDR    R0,=charvec
LDRB   R0,[R0,#5]
```

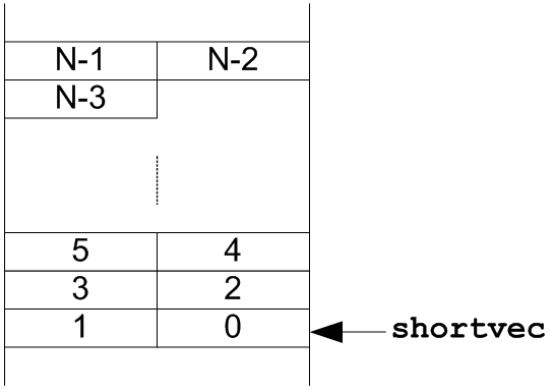
Adress: `charvec+k`
Konstant index:
`LDR  Rd,[Rb,#k]`
`@ Rd←M(Rb+k)`

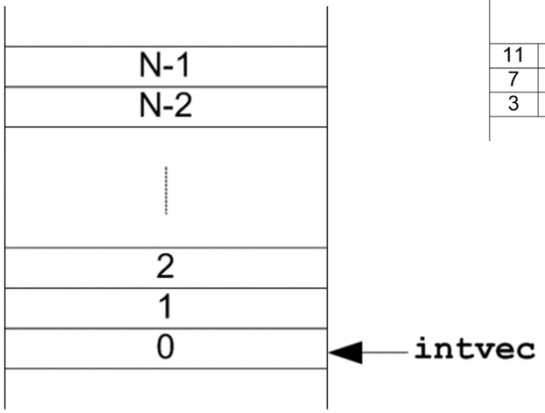
```
int i;
char   charvec[N];
charvec[i];

LDR    R0,=charvec
LDR    R1,i
LDRB   R0,[R0,R1]
```

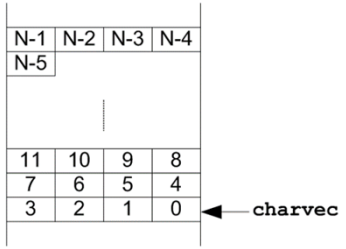
Adress: `charvec+i`
Register index:
`LDR  Rd,[Rb,Ri]`
`@ Rd←M(Rb+Ri)`



Adressberäkning och adresseringssätt








```
int i;
short shortvec[N];
shortvec[i];

=shortvec + i*sizeof(short)

LDR   R0,=shortvec
LDR   R1,i
LSL   R1,R1,#1    @R1←R1×2
LDRSH R0,[R0,R1]
```

```
int i;
int   intvec[N];
intvec[i];

=intvec + i*sizeof(int)

LDR   R0,=intvec
LDR   R1,i
LSL   R1,R1,#2    @R1←R1×4
LDR   R0,[R0,R1]
```

Tilldelning från fält

Exempel: Vi har deklarationerna:

```
char c; int i;
char vec[15];
```

Visa hur följande tilldelning kan kodas:

```
c = vec[i];
```

FLISP assembler
(antag `char` och `int` lika stora)

```
LDX  #vec
LDA  i
LDA  A,X
STA  c
```

Adress till `vec[i]` blir
`=vec + i*sizeof(char)`

THUMB assembler

```
LDR  R0,=vec
LDR  R1,i
LDRB R0,[R0,R1]
LDR  R1,=c
STRB R0,[R1]
```

Tilldelning till fält

Exempel: Vi har deklarationerna:

```
char c; int i;
char vec[15];
```

Visa hur följande tilldelning kan kodas:

```
vec[i] = c;
```

Adress till `vec[i]` blir
`=vec + i*sizeof(char)`

FLISP assembler
(antag `char` och `int` lika stora)

```
LDA  #vec
ADDA i
PSHA
PULX
LDA  c
STA  ,X
```

THUMB assembler

```
LDR  R0,=c
LDRB R0,[R0]
LDR  R1,=vec
LDR  R2,i
STRB R0,[R1,R2]
```

Flerdimensionella fält

”matriser” – N × M element, indexeras [0..N-1] [0..M-1]

Exempel:

```
char  mc [N][M];
int   i1,i2;
```

Referens: `mc[i1][i2]`

Minnesadress: `=mc + ((i1*M)+i2)*sizeof(char)`

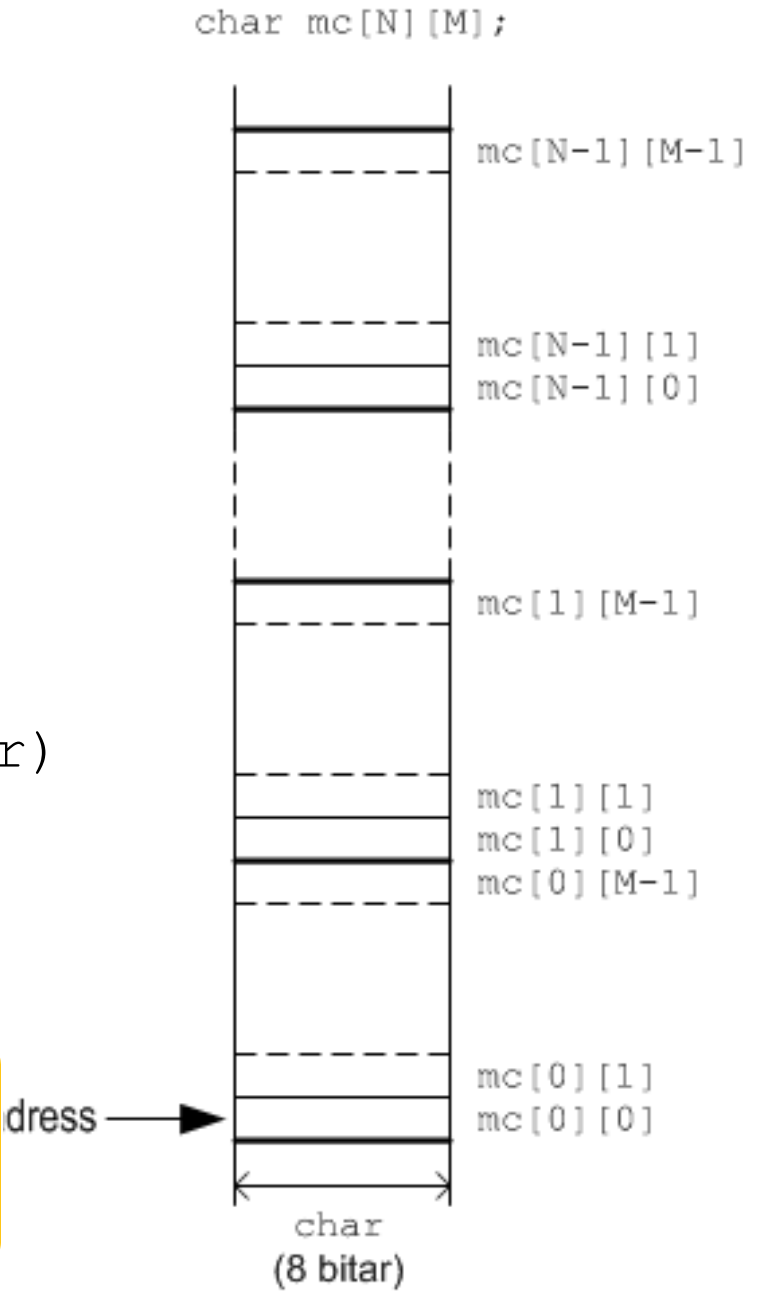
Kompilatorn ersätter multiplikationer med skift/add kombinationer eftersom `mul`-instruktionen tar betydligt fler klockcykler att utföra.

$M = 2^x + y$

Exempelvis då M=10:

$10 = 2^3+2$

```
i1*10 =
i1*(2^3+2) =
i1<<3 + i1 + i1
```



Referens av objekt i fält

Exempel: Vi har deklarationerna:

```
int i,j;
int veci[80];
int vm[10][5];
```

Visa kodsekvenser som evaluerar följande uttryck till register R0.

a) `veci[j];`

Adress:
`=veci + j*sizeof(int)`

b) `vm[i][j]`

Adress:
`=vm + ((i*5)+j)*sizeof(int)`

Vi löser på tavlan...

Sammansatta typer -”struct” (post)

Inkapsling av flera variabler, ev. med olika typer

```
struct snamn{
        …
};
typedef struct snamn TSTRUCTNAMN;
```

alternativt:

```
typedef struct snamn        /* snamn kan utelämnas här */
{
        …
} TSTRUCTNAMN;
```

EXEMPEL: Datatyp för koordinater

```
typedef struct
{
        int x,y,z;
} COORD;
```

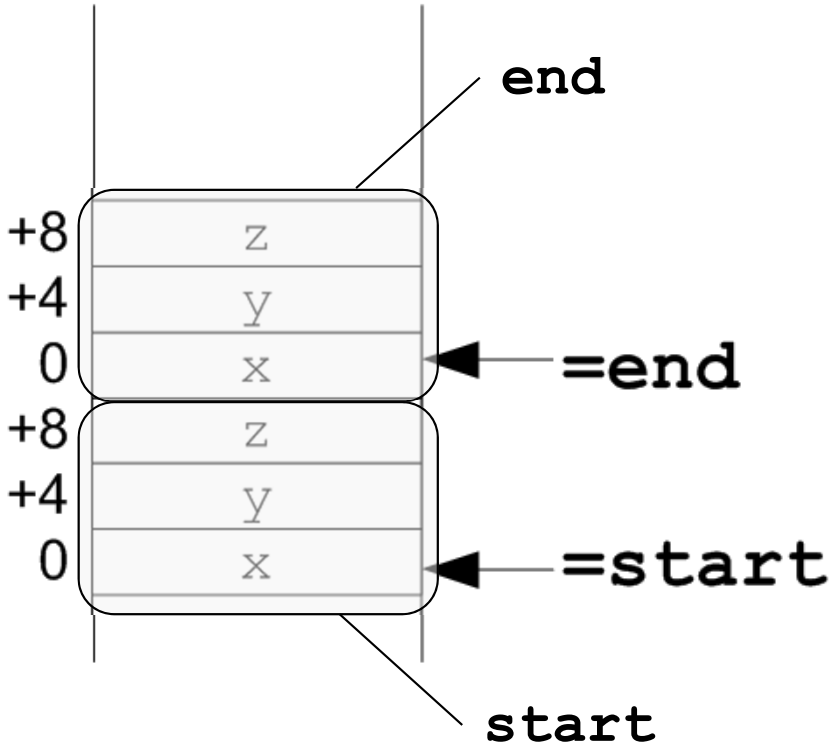
EXEMPEL: Deklaration av två koordinater

```
COORD start, end;
```

Referenser

EXEMPEL: Datatyp för koordinater

```
typedef struct
{
        int x,y,z;
} COORD;
```



EXEMPEL: Vi har deklarationerna:

```
COORD start, end;
```

Koda följande tilldelningar i assemblerspråk

```
start.x = 1;
start.z = 3;
end.y = 20;
```

Lösning:

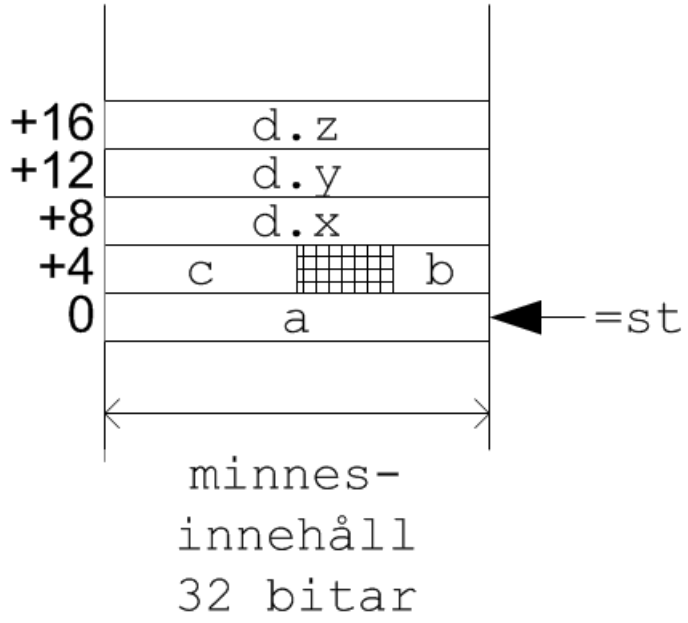
```
        LDR    R3,=start
        MOV    R2,#1
        STR    R2,[R3]
        MOV    R2,#3
        STR    R2,[R3,#8]
        LDR    R3,=end
        MOV    R2,#20
        STR    R2,[R3,#4]
```

Inbäddad post

- Ev. rättningsvillkor måste respekteras.
- En annan post kan ingå i en större post.

EXEMPEL: Datatyp för koordinater

```
typedef struct
{
      int x,y,z;
} COORD;
```




```
.ALIGN      2
offset_a = 0
offset_b = offset_a + sizeof(a)
offset_c = offset_b + sizeof(b) + align(1)
offset_d_x= offset_c + sizeof(c) + align(2)
offset_d_y= offset_d_x+ sizeof(d.x) + align(2)
offset_d_z= offset_d_x+ sizeof(d.x) + align(2)
```

Vi har deklarationen:

```
struct {
        int     a;
        char    b;
        short   c;
        COORD   d;
} st;
```

Visa kodsekvenser som evaluerar följande uttryck i register R0.

a) `st.c`

b) `st.d.y`

```
a)
LDR     R0,=st
LDRH    R0,[R0,#6]      @ R0= st + 6   (offset_c)
b)
LDR     R0,=st
LDR     R0,[R0,#12]     @ R0= st + 12  (offset_d_y)
```

Union

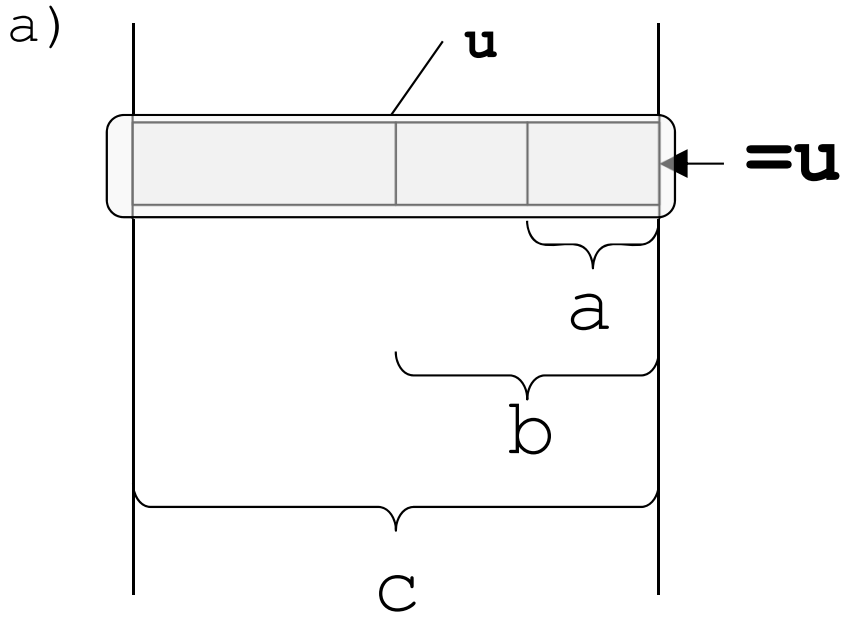
Storleken (`sizeof(u)`) är garanterad att rymma den största, av de typer, som ingår i unionen.

Vi har deklarationen:

```
union {
        char  a;
        short b;
        long  c;
} u;
```

```
.ALIGN      2
offset_a = 0
offset_b = 0
offset_c = 0
```

a) Visa hur u representeras i minnet.



Visa kodsekvenser som evaluerar följande uttryck i register R0.

b) `u.a;`

c) `u.b;`

d) `u.c;`

```
b)      LDR   R3,=u
        LDRB  R0,[R3]
c)      LDR   R3,=u
        LDRH  R0,[R3]
d)      LDR   R3,=u
        LDR   R0,[R3]
```

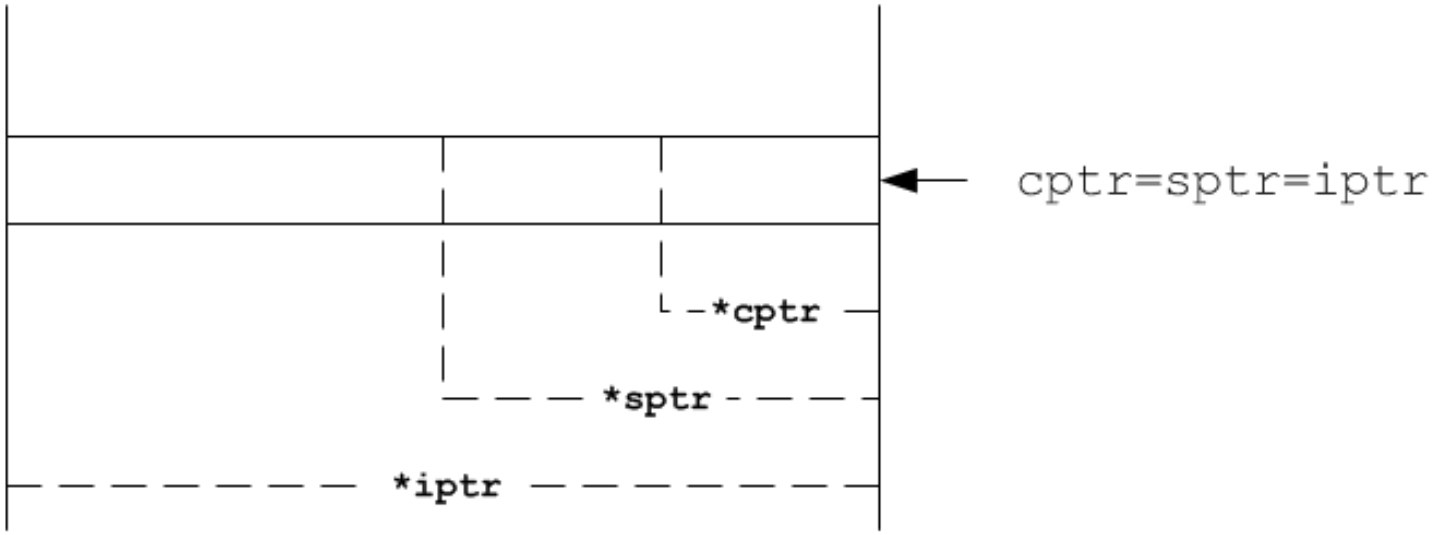
Grundläggande pekartyper, ARM-M4

```
char   *cptr; /* pekar på 8-bitars datatyp */
short  *sptr; /* pekar på 16-bitars datatyp */
int    *iptr; /* pekar på 32-bitars datatyp */
```

Adressutrymmet är 32 bitar. PC, SP och ALLA pekartyper är 32 bitar

Exempel:

```
cptr = ( char  *) 0x40021010
sptr = ( short *) 0x40021010
iptr = ( int   *) 0x40021010
```



Operatorer för pekare

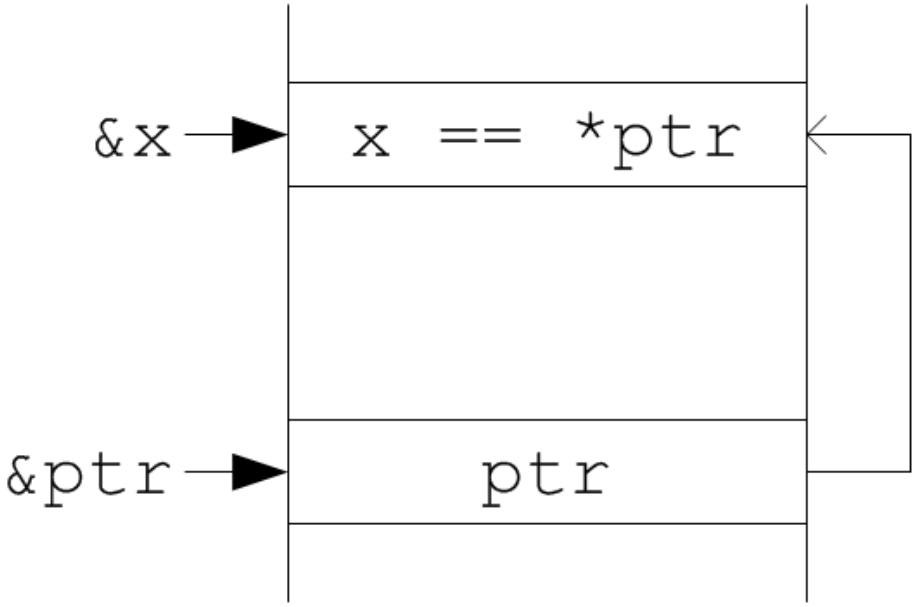
&

*

Vi har:

```
int i, *ptr;
```

Om pekarvariabeln `ptr` innehåller adressen till `x` gäller att `ptr == &x` , `ptr` är en referens till `x` och att: `*ptr == x` är en dereferens av `ptr`



Dvs:

`ptr` är en minnesadress

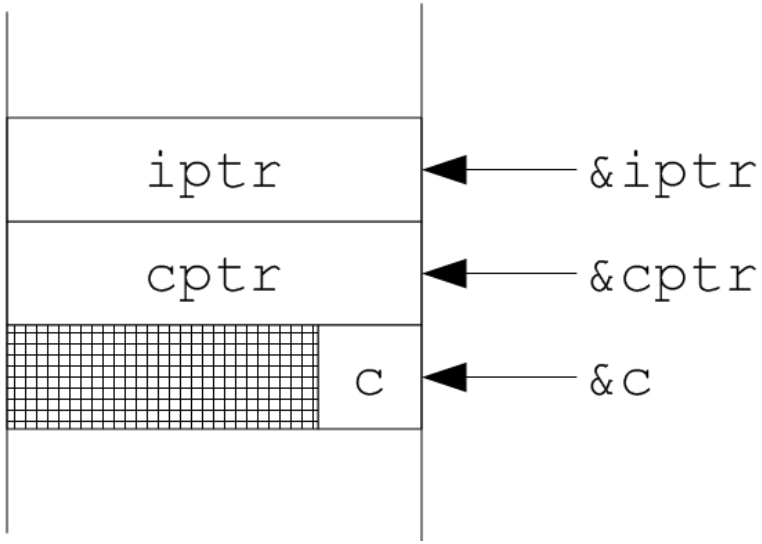
`*ptr` är innehållet på denna minnesadress

`&ptr` är pekarvariabelns adress i minnet

Referenser, dereferenser och tilldelningar

Exempel: Vi har deklarationerna:

```
char  c,*cptr;
int   *iptr;
```



Koda följande tilldelningar i assemblerspråk

```
a) cptr = iptr;
b) *cptr = *iptr
c) cptr = &c;
d) c = *cptr;
```

```
@ a)    cptr = iptr;
    LDR     R0,iptr
    LDR     R1,=cptr
    STR     R0,[R1]
```

```
@ b)    *cptr = *iptr
    LDR     R0,iptr
    LDR     R0,[R0]
    LDR     R1,cptr
    STRB    R0,[R1]
```

```
@ c)    cptr = &c;
    LDR     R0,=c
    LDR     R1,=cptr
    STR     R0,[R1]
```

```
@ d)    c = *cptr;
    LDR     R0,cptr
    LDRB    R0,[R0]
    LDR     R1,=c
    STRB    R0,[R1]
```

Pekare och portar

En "port" är i själva verket ett register som finns på en konstant adress i minnesutrymmet.

För att läsa från, eller skriva till registret måste vi därför kunna dereferera en konstant.

Alltså måste konstanten först typkonverteras till en pekare, syntaxen kräver då:

```
*(( typ *) konstant)
```

Exempel:

Då en 16 bitars port på address 0x40021010 refereras enligt:

```
*((unsigned short*) 0x40021010);
```

kodas detta i assembler:

```
LDR  R0,=0x40021010
LDRH R0,[R0]
```

Användning av konstant pekare

Exempel:
Visa, såväl i C som i assembler, hur de fyra minst signifikanta bitarna hos en läs- och skrivbar port på address 0x40021014 sätts till 1, medan övriga bitar lämnas opåverkade.

Vi löser på tavlan

offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic
0x14																	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	GPIO_ODR
																	0x15								0x14								

Pekare till fält

Exempel:
De globala variablerna `i`, och `vecc` är definierade enligt:

```
int i;
char  vecc[20];
```

Visa en kodsekvens som evaluerar uttrycket `&vecc[i-1]` till register R0.

Lösning:

```
@ minnesadress: =vecc + i-1
    LDR    R0,=vecc
    LDR    R1,i
    SUB    R1,R1,#1
    ADD    R0,R0,R1
```

Exempel:
De globala variablerna `i`, och `mc` är definierade enligt:

```
int i;
char  mc[12][8];
```

Visa en kodsekvens som evaluerar uttrycket `&mc[i][4]` till register R0.

Lösning:

```
@ minnesadress: =mc + i*8 + 4
    LDR    R0,=mc
    LDR    R1,i
    LSL    R1,R1,#3
    ADD    R0,R0,R1
    ADD    R0,R0,#4
```

Vad gäller för `int mc[12][8];` ?

Pekare till sammansatta typer

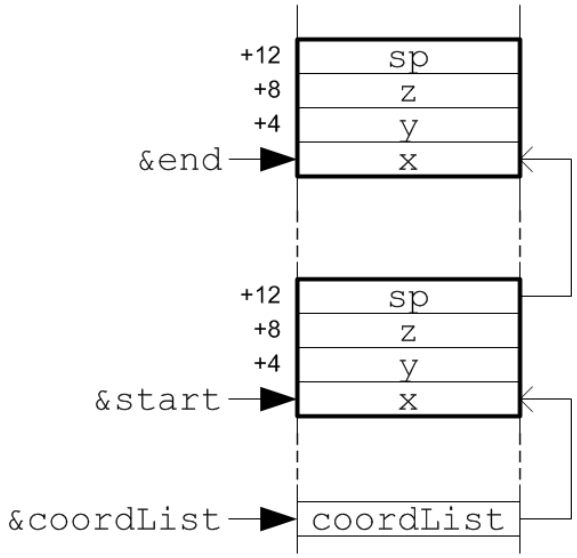
Exempel:

```
typedef struct coord
{
   int x,y,z;
   struct coord *sp;
} COORD;
COORD   start;
COORD   end;
COORD   *coordList;
```

Koda tilldelningarna:

```
coordList =      &start;
start.sp =       &end;
coordList->sp =  &end;
```

```
(*coordList).sp = &end;
coordList->sp = &end;
```




```
@ start.sp = &end;
    LDR     R3,=end
    LDR     R2,=start
    STR     R3,[R2,#12]
```

```
@ coordList = &start;
    LDR     R3,=start
    LDR     R2,=coordList
    STR     R3,[R2]
```

```
@ coordList->sp = &end;
    LDR     R3,=end
    LDR     R2,=coordlist
    LDR     R2,[R2]
    STR     R3,[R2,#12]
```

Pekare till funktioner

Vi har följande funktioner:

```
void  do_this( void ) {}
void  do_that( void ) {}
```

Vi kan deklarera en variabel `func`, som en pekare till en sådan funktion.

```
void (*func)(void);
```

Vi har sedan ytterligare ett instrument att påverka programflödet:

```
if( ... )
   func = &do_this;
else
   func = &do_that;
   ...
func();  /* do_this eller do_that... */
```

```
LDR  R2,=do_this
LDR  R3,=func
STR  R2,[R3]
```

```
LDR  R3,func
BLX  R3
```

Pekare till funktioner

Exempel:

Visa i C och assembler hur en pekare till funktionen

```
void do_this( void )
```

placeras på address 0x2001C000 i minnet.

Tilldelningen kodas i C:

```
*((void (**)(void) ) 0x2001C000 ) = &do_this;
```

Tilldelningen kodas i assembler:

```
LDR     R0,=do_this
LDR     R1,=0x2001C000
STR     R0,[R1]
```

Pekare till pekare

Exempel:

`cp = *dcp;` kodas

```
LDR   R3,=dcp   @ R3←&dcp
LDR   R3,[R3]   @ R3←dcp
LDR   R3,[R3]   @ R3←*dcp
LDR   R2,=cp
STR   R3,[R2]
```

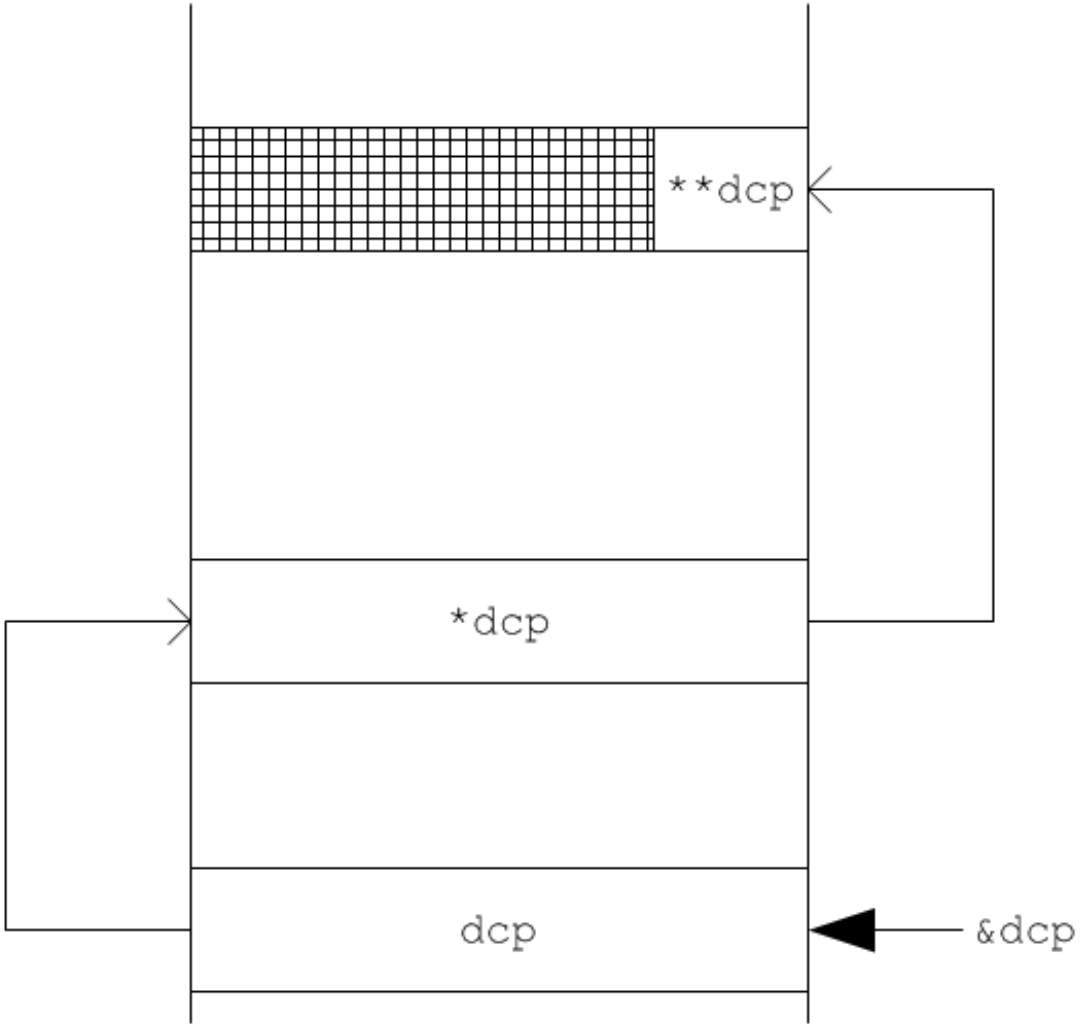
`c = **dcp;` kodas

```
LDR   R3,=dcp  @ R3←&dcp
LDR   R3,[R3]  @ R3←dcp
LDR   R3,[R3]  @ R3←*dcp
LDRB  R2,[R3]  @ R3←**dcp
LDR   R3,=c
STRB  R2,[R3]
```

Deklarationen :

`char **dcp;`

läses ”dcp är en pekare till en pekare till en char”.



Increment/Decrement operatorer

Operatorer för att öka eller minska med 1:

```
int i,j;
i = j++;      /* i←j; j←j+1; */  postinkrement
i = ++j;      /* j←j+1; i←j; */  preinkrement
i = j--;      /* i←j; j←j-1; */  postdekrement
i = --j;      /* j←j-1; i←j; */  predekrement
```

Samma operatorer för pekare ger ökning/minskning baserad på typ:

```
char  *cp;   +/- sizeof( char)      cp++ ; /* cp←cp+1; */
short *sp;   +/- sizeof( short)     sp++ ; /* sp←sp+2; */
int   *ip;   +/- sizeof( int)       ip++ ; /* ip←ip+4; */
```

Pekararitmetik

Exempel:
I standard-C biblioteket finns funktionen **`strcpy`**, för att kopiera en ASCII-sträng i minnet. Strängslut markeras med tecknet **`'\0'`**, dvs. 0 och detta tecken ska vara det sista som kopieras. Funktionen returnerar `dest` och kan implementeras på följande sätt, visa hur **`strcpy`** kan kodas i assemblerspråk.

```
char *strcpy( char *dest, char * src )
{
  char *rval = dest;
  do{
    *dest++ = *src++;
  } while( *(src-1) );
  return rval;
}
```

Vi löser på tavlan...