Fält, pekare och portar

Ur innehållet

- Fält
- Pekarvariabler, pekarkonstanter och portar
- Pekarreferenser

Målsättningar:

- Använd printf för testutskrifter på värddator
- Generera ARM/Thumb assemblerkod för fältreferenser
- Konstruera pekarreferenser för portar

Vad är ett "fält"?

Ett fält ("array") är en datastruktur som mångfaldigar förekomsten av element med samma typ.

Exempel:
Deklarationen:

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

skapar ett fält med `N` st. element av typen `int`.

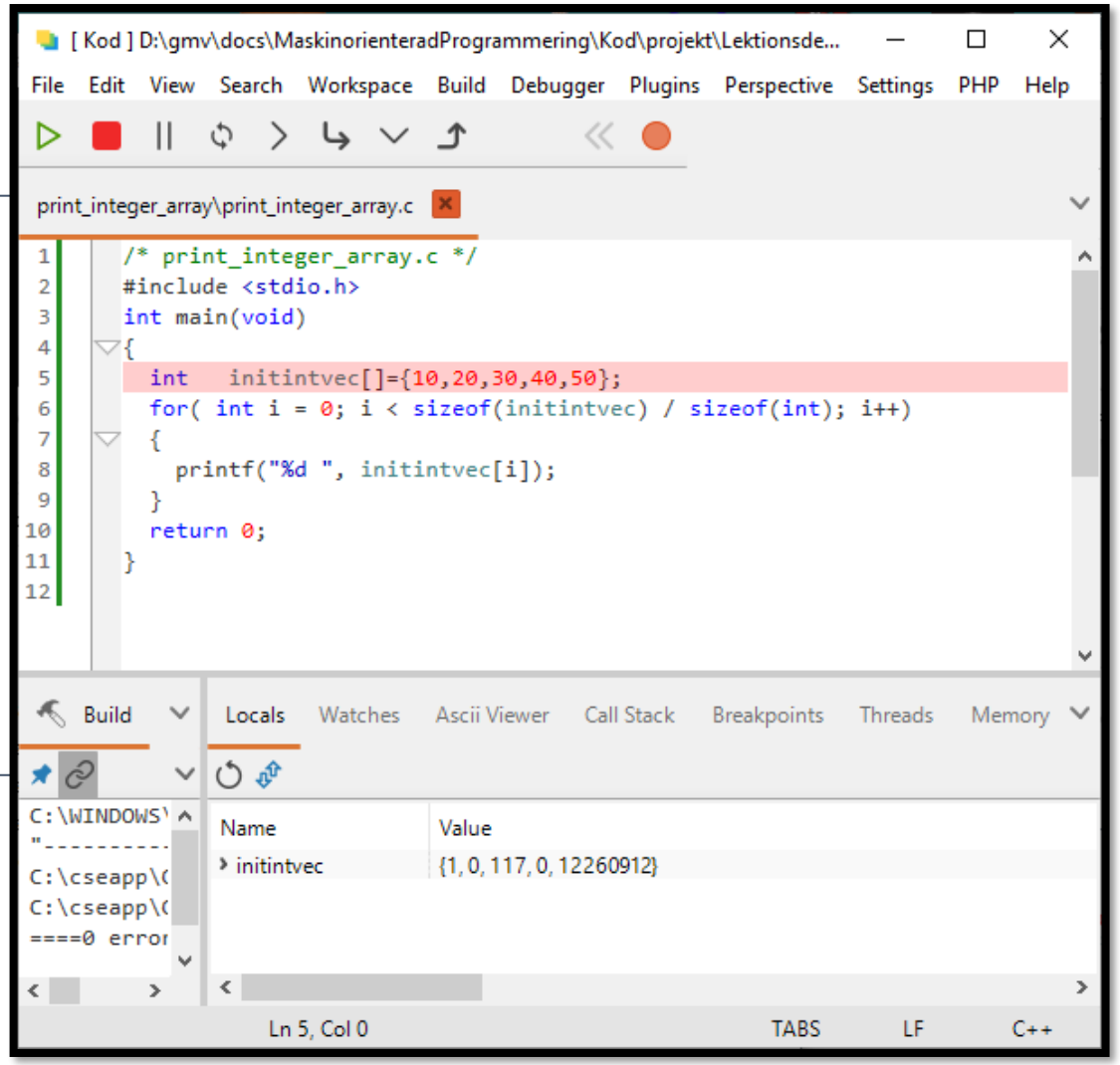
Innehållet är initialt odefinierat men ett fält kan också deklarares och initieras samtidigt:

```
int    initintvec[]={10,20,30,40,50};
```

skapar ett fält med 5 element och initierar samtidigt fältets element .

Undersök ett fält:

```c
/* print_integer_array.c */
#include <stdio.h>
int main(void)
{
  int    initintvec[]={10,20,30,40,50};
  for( int i = 0; i < sizeof(initintvec) / sizeof(int); i++)
  {
    printf("%d ", initintvec[i]);
  }
  return 0;
}
```

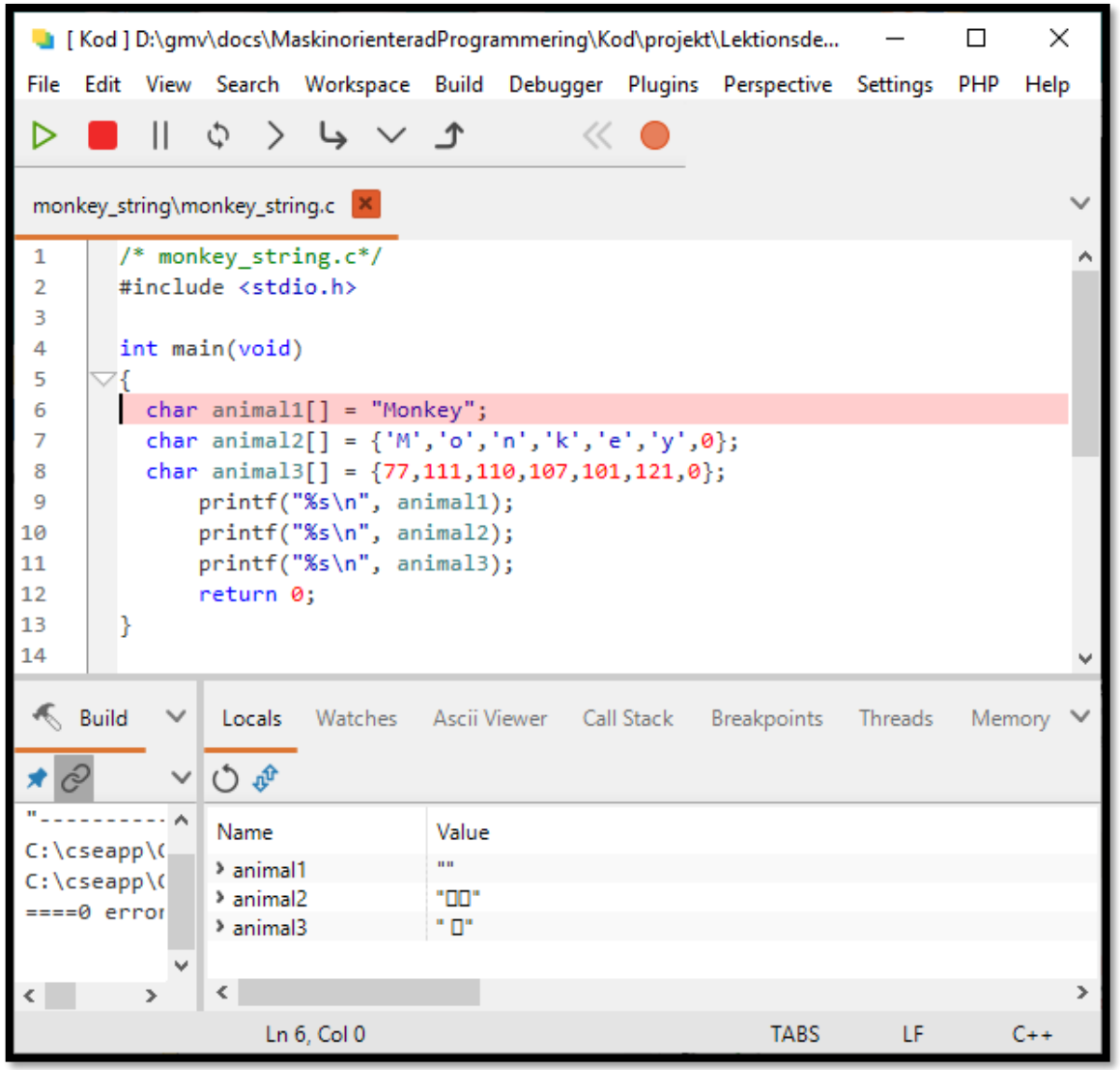


Fält och textsträngar

En textsträng är ett fält av `char`, som avslutas med 0 ('\0')

Exempel:

```
#include <stdio.h>

int main(void)
{
  char animal1[] = "Monkey";
  char animal2[] = {'M','o','n','k','e','y','\0'};
  char animal3[] = {77,111,110,107,101,121,0};
  printf("%s\n", animal1);
  printf("%s\n", animal2);
  printf("%s\n", animal3);
  return 0;
}
```



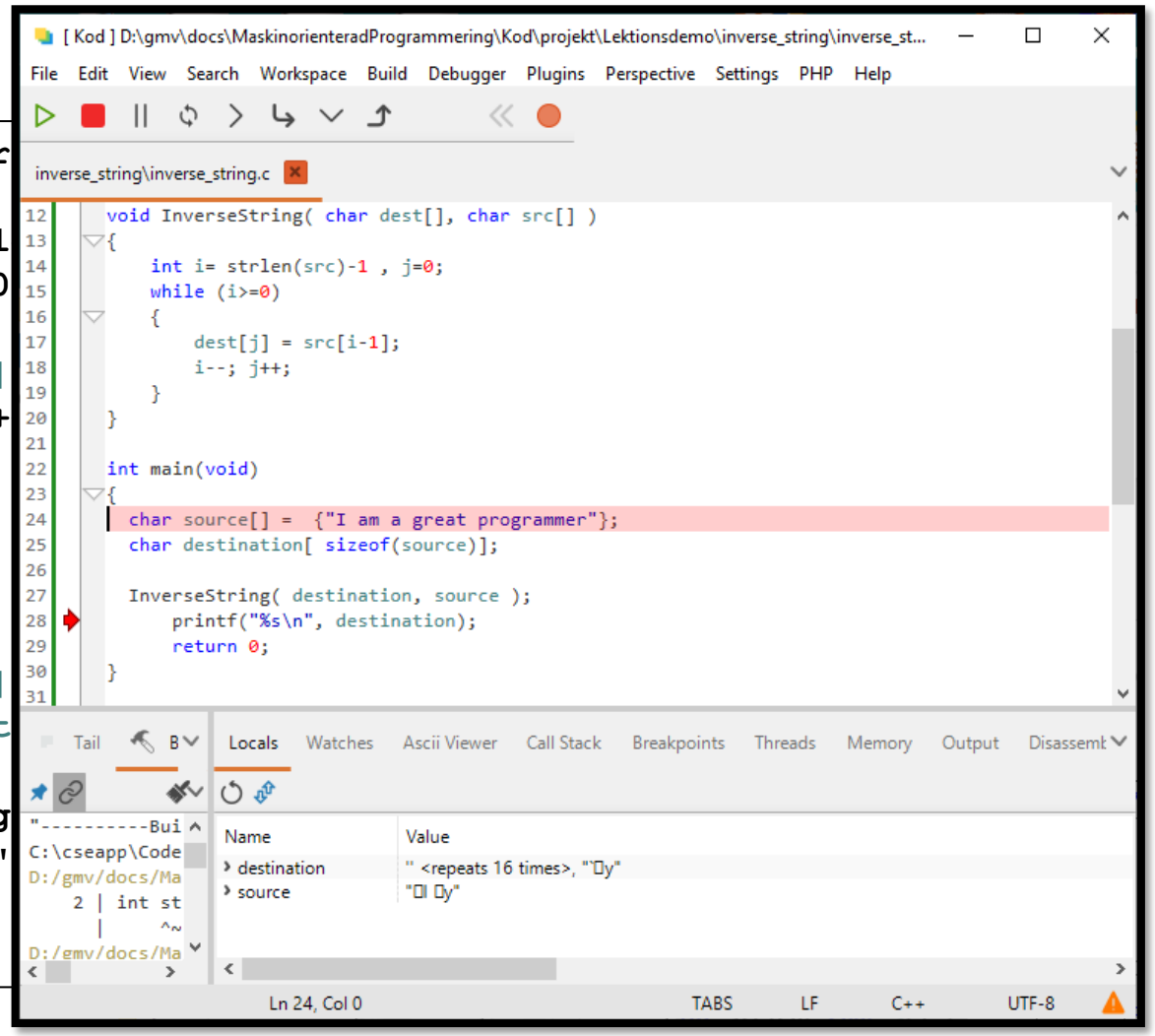
I standardbiblioteket: Längden av en textsträng: **strlen**

`strlen` returnerar längden av en textsträng, inklusive terminerande '\0'.

```
int strlen( char s[] )
{
    int i = 0;
    while( s[i] != '\0')
    {
        i++;
    }
    return i + 1;
}
```

Exempel:

```
void InverseStr
{
    int i= strl
    while (i>=0
    {
        dest[j]
        i--; j+
    }
}

int main(void)
{
  char source[]
  char destinat

  InverseString
  printf("%s\n"
  return 0;
}
```

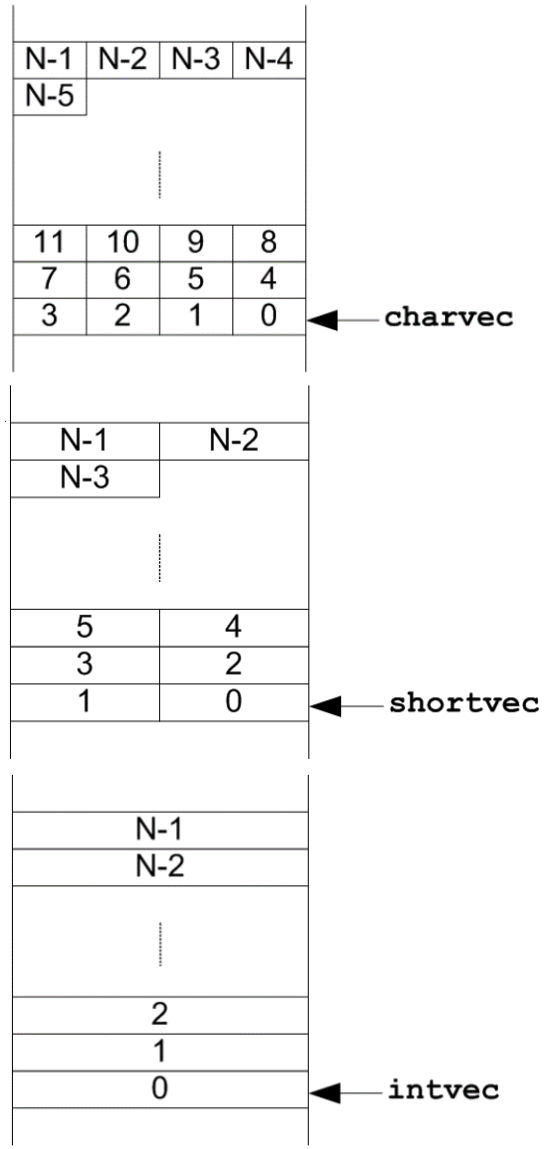


Fält – ”vektorer” – N element, indexeras 0..N-1

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

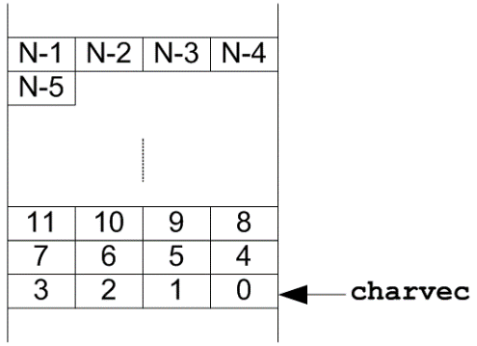
```
short shortvec[N];
sizeof(shortvec)= N*2 bytes
```

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



Adressberäkning och adresseringssätt

```
const  k; (0 ≤ k ≤ N-1)
int    i;
char   charvec[N];
```



Exempel:

```
charvec[5];

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

Adress: `charvec+k`
Konstant index:

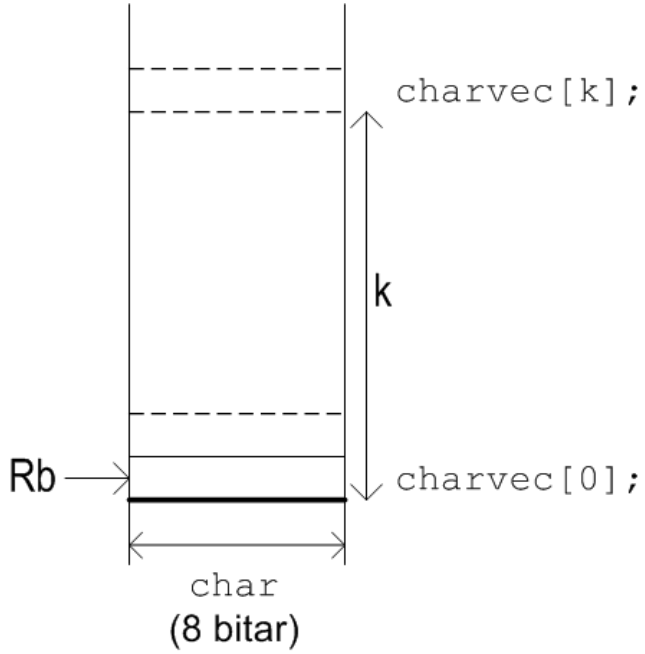
```
LDRB   Rd,[Rb,#k]
@ Rd←M(Rb+k)
```

Exempel:

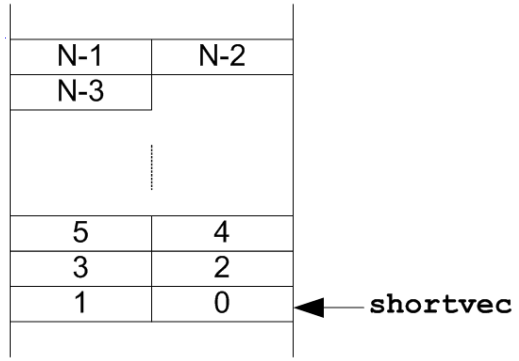
```
charvec[i];

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

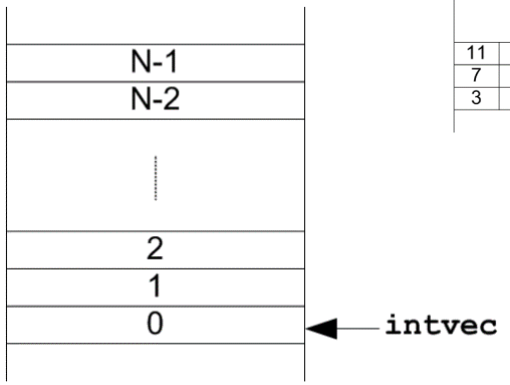
Adress: `charvec+i`
Register index:

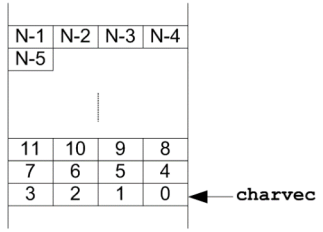
```
LDRB   Rd,[Rb,Ri]
@ Rd←M(Rb+Ri)
```



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];
```

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:

```
short s; int i;
short vec[15];
```

Visa hur följande tilldelning kan kodas:

```
vec[i] = s;
```

Adress till `vec[i]` blir

```
=vec + i*sizeof(short)
```

THUMB assembler

```
LDR   R0,=s
LDRH  R0,[R0]
LDR   R1,=vec
LDR   R2,i
LSL   R2,R2,#1
STRH  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)`

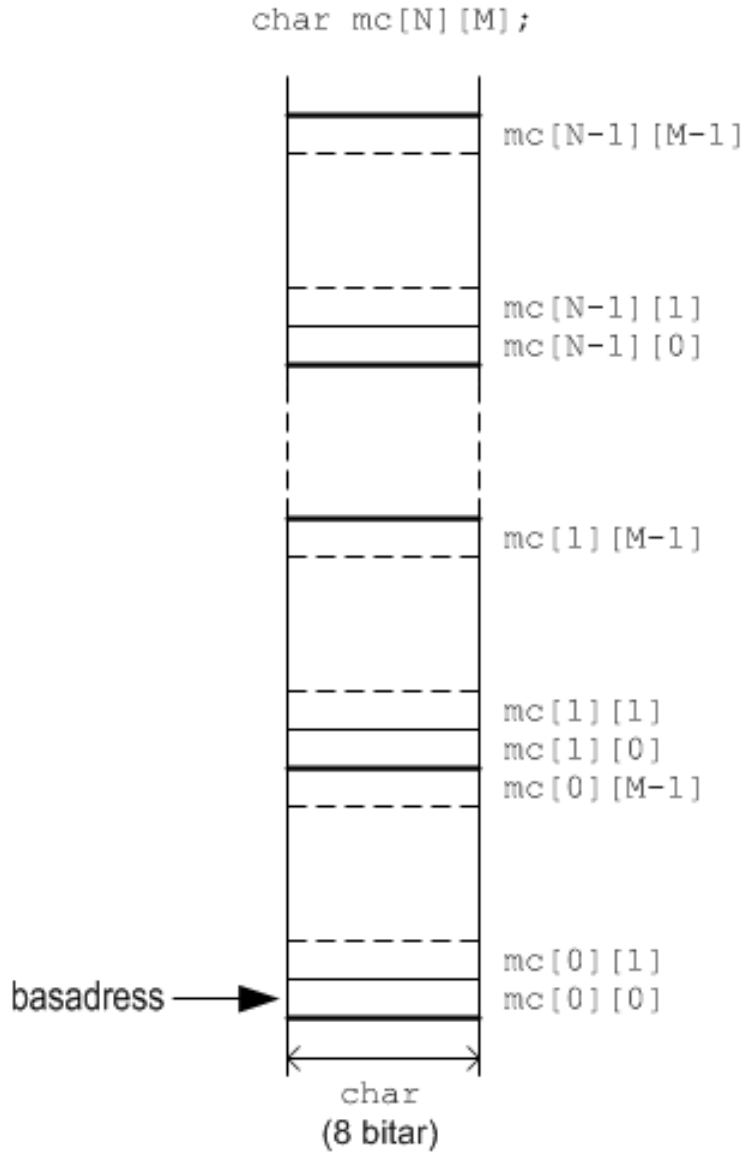
Kompilatorn ersätter multiplikationer med skift/add kombinationer eftersom `mul`-instruktionen tar betydligt fler klockcykler att utföra.

$M = 2^x + y$

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...

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

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

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]`

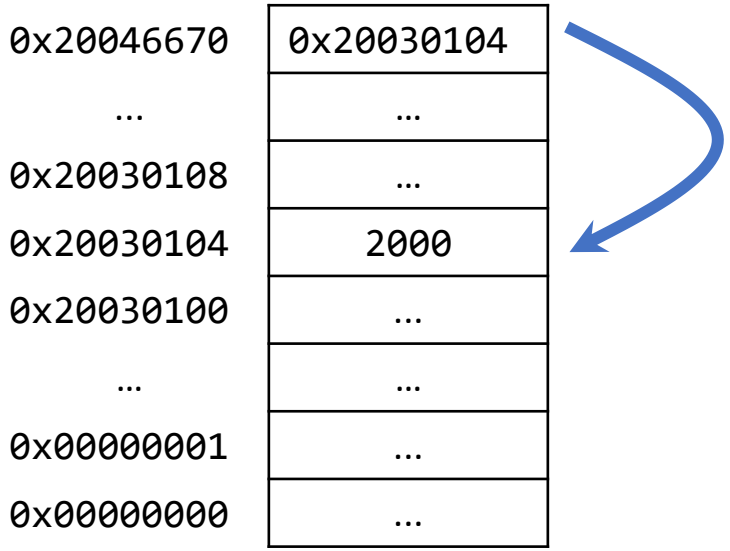
Adress:
```
=vm + ((i*5)+j)*sizeof(int)
```

Vi löser på tavlan...

Pekare

En pekare är en datatyp för en minnesadress.
En pekarvariabel innehåller minnesadressen till en variabel, port etc.
snarare än variabelns (portens) värde.

Exempel:
En *pekare* till värdet `2000`,
dvs. värdets position som är en
minnesadress



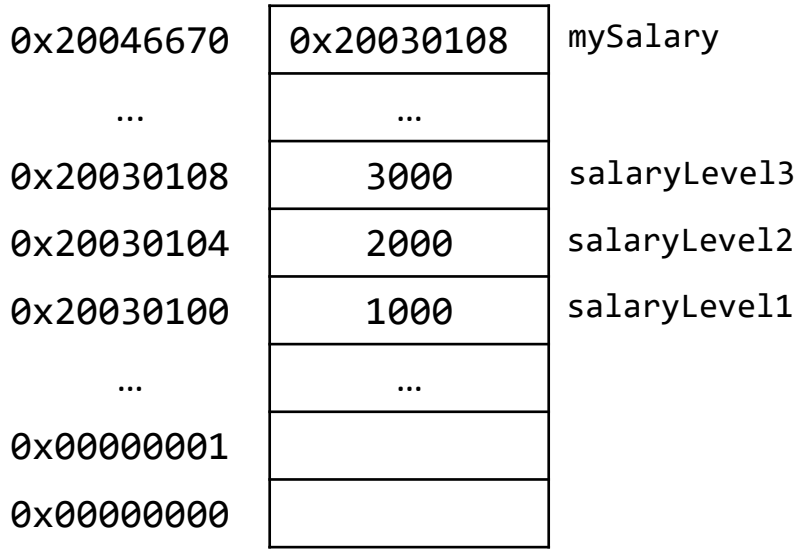
Pekaroperatorer

1. Pekarens värde är en adress (&)
2. Pekarens typ anger hur vi ska tolka bitarna hos innehållet på adressen
3. '*' (stjärna) används för att referera *innehållet på adressen* (dereferera) .

Exempel:

```
int salaryLevel1 = 1000;
int salaryLevel2 = 2000;
int salaryLevel3 = 3000;

int* mySalary = &salaryLevel2;
```

```
&mySalary är 0x20046670
 mySalary är 0x20030104
*mySalary är 2000
```



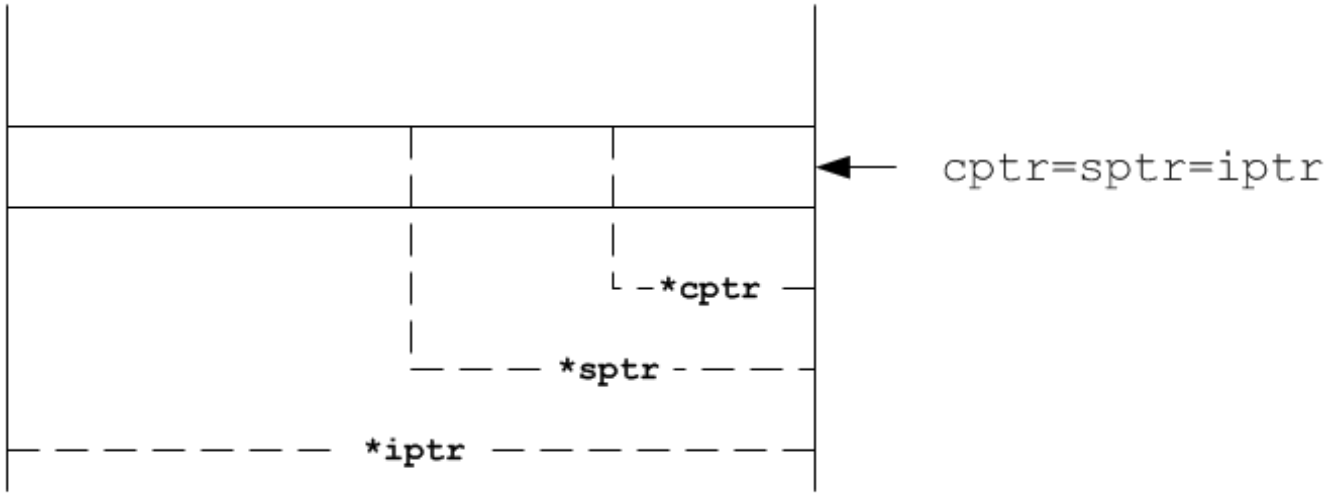
Grundläggande pekartyper, ARM/Thumb

```
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
```

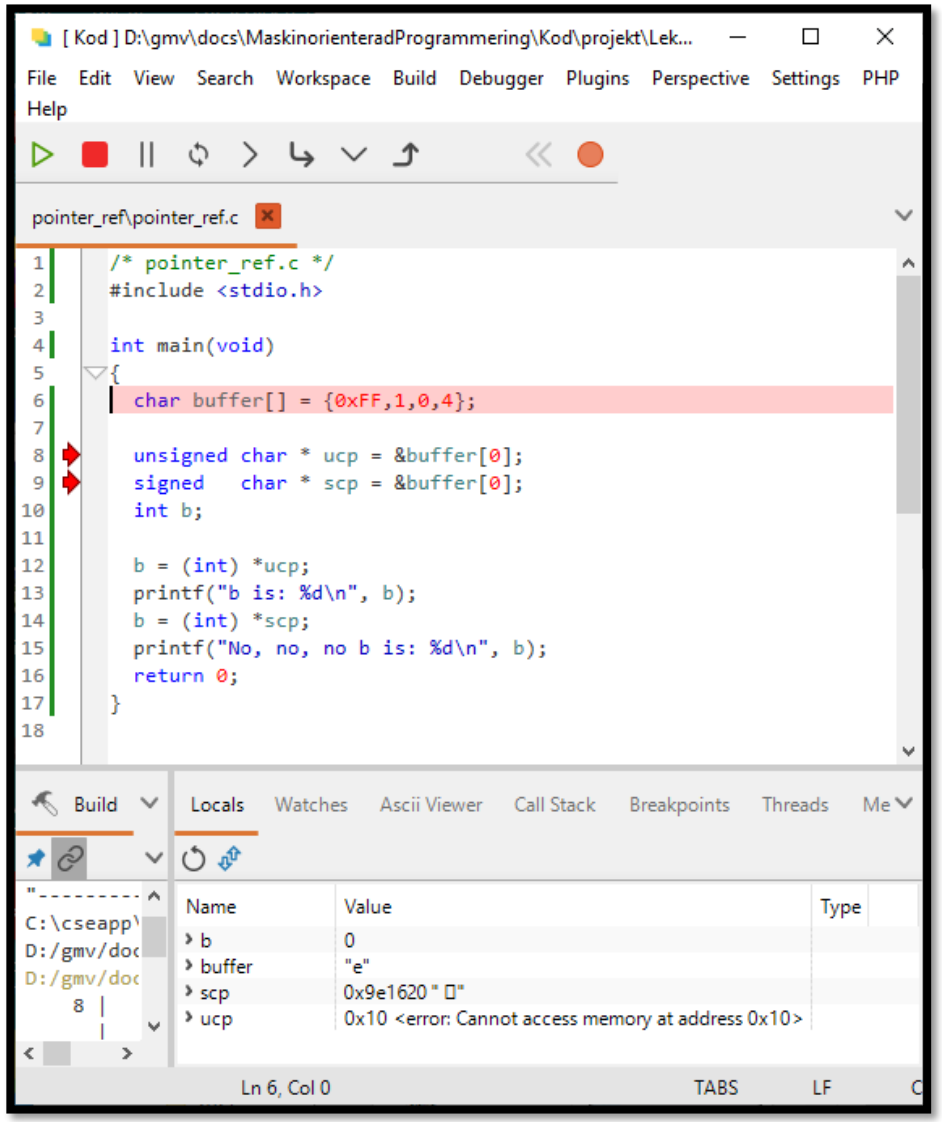


Pekare: dereferens

- När vi derefererar en pekare får vi objektet som finns på pekarens adress.
- *Antal bytes* beror då av pekarens typ
- *Tolkningen* av bitarna beror av pekarens typ

Exempel:

```
char buffer[] = {0xFF,1,0,4};

unsigned char * ucp = &buffer[0];
signed   char * scp = &buffer[0];
...
int b;
    b = (int) *ucp;
    ...
    b = (int) *scp;
```



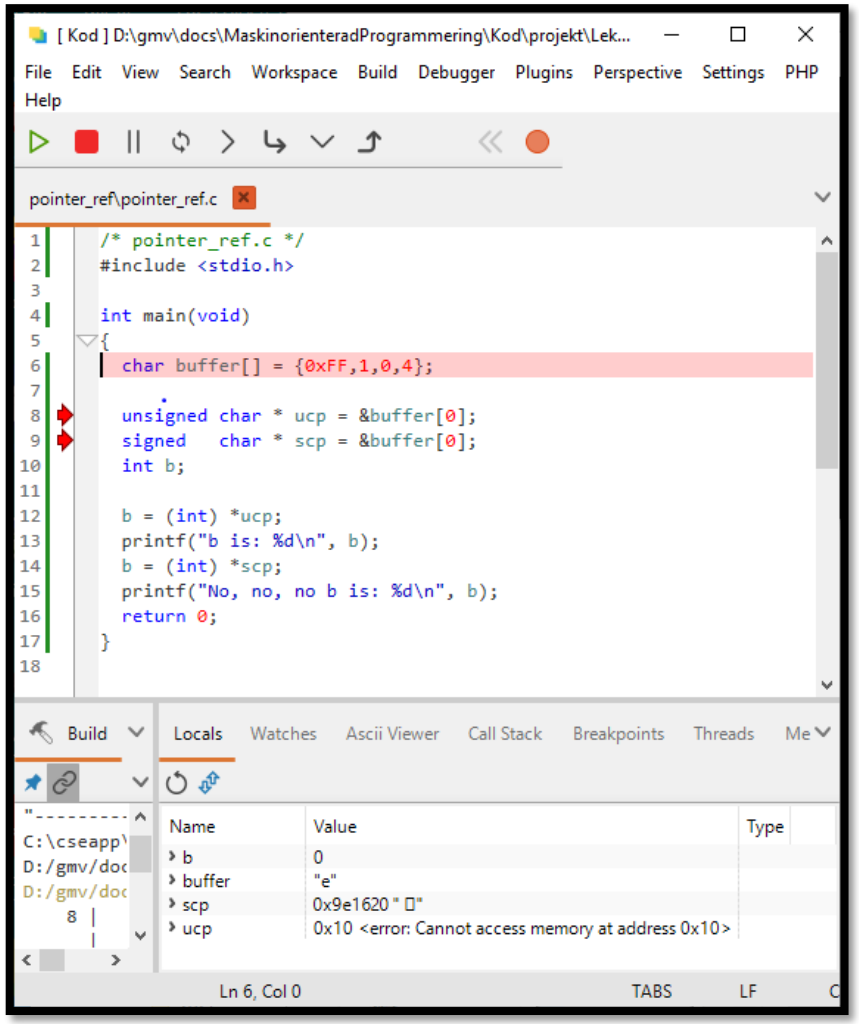
Varför behöver vi pekare?

En pekarvariabel innehåller minnesadressen till en variabel, port etc. snarare än variabelns (portens) värde.

Pekare tillåter oss att referera en variabel (eller ett objekt) utan att först skapa en kopia

Exempel:

```
#include <stdio.h>

char person1[] = "Elsa";
char person2[] = "Alice";
char person3[] = "Maja";
char *winner;

int main(void)
{
    winner = person2;
    printf("%s is the winner\n", winner);
    winner = person1;
    printf("No, %s is the winner\n", winner);
    return 0;
}
```

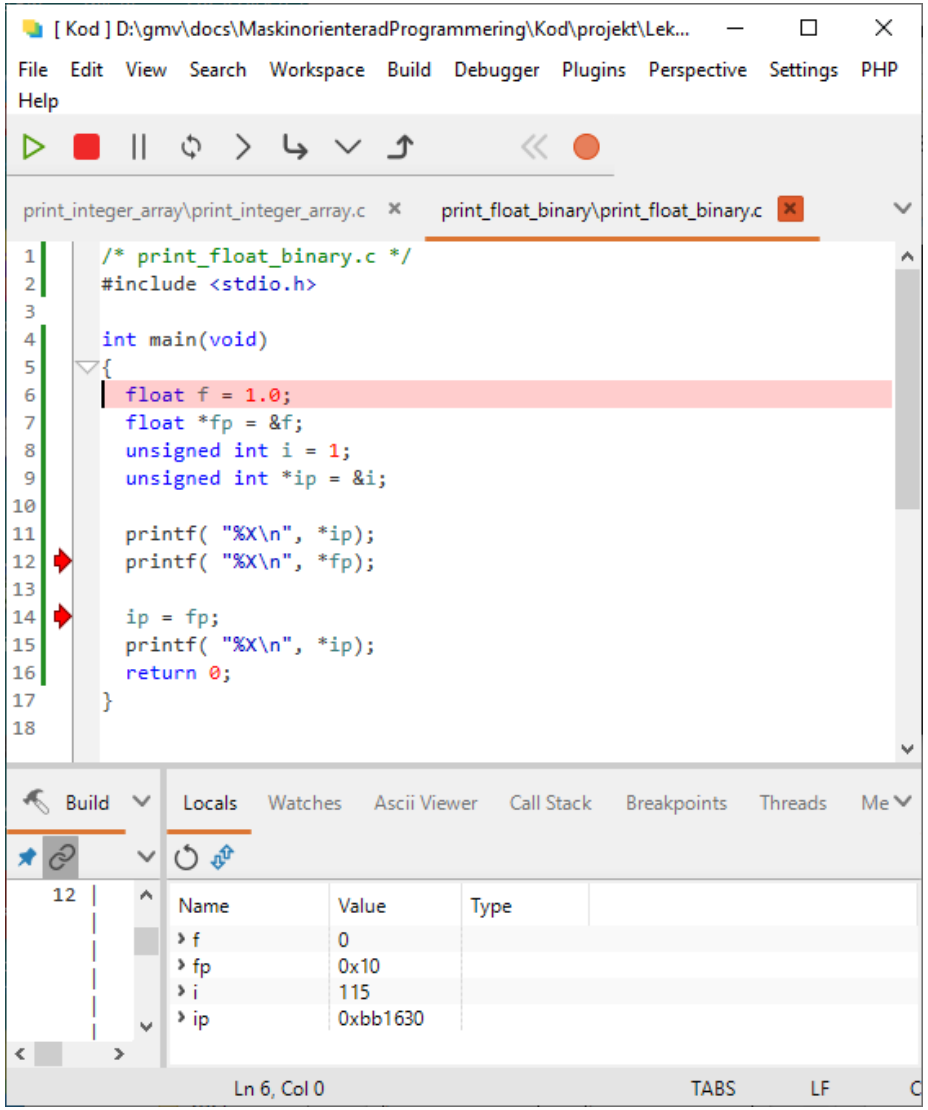


Möjligheter med pekare...

Exempel:
Antag att vi vill skriva ut det hexadecimala värdet av en godtycklig datatyp med `printf`.
Kompilatorns implicita typkonverteringar och typkontroll kan då ställa till problem (prova själv).

Exempelvis har ett talvärde helt olika representationer i typerna `int` och `float`.

.. vilket följande program kan ge en anvisning om...



Vad betyder stjärnan..? "*"

I en deklaration anger stjärnan en pekartyp

Exempel:

```
char  *cp;
float *fp;
unsigned int *ip;
void f(int *ip);
char *g(void);
```

I ett uttryck, dvs. som operator anger stjärnan en dereferens.

Exempel:

```
char a = *cp;
*cp = 'b';
printf( "%X\n", *ip );
a = 5 * *cp;
```

Stränghantering

Pekare används ofta vid hantering av textsträngar

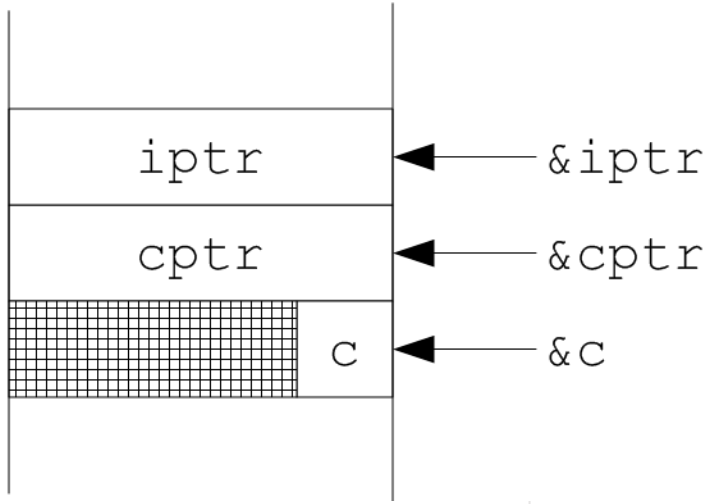
```
int strlen( char s[] )
{
    int i = 0;
    while( s[i] != '\0')
    {
        i++;
    }
    return i + 1;
}
```

```
int strlen( char *s )
{
    int i = 0;
    while( *s )
    {
        s++; i++;
    }
    return i + 1;
}
```

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 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.

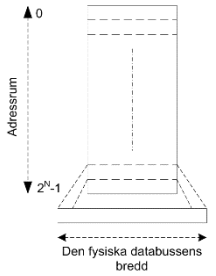
Exempel:
De globala variablerna `i`, och `mi` är definierade enligt:

```
int  i;
int  mi[12][8];
```

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

Och vad är en "port"?

32-bit data,
32-bit adress
2^{32} bytes = 4 294 967 296 bytes
= 4 Giga bytes



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:

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

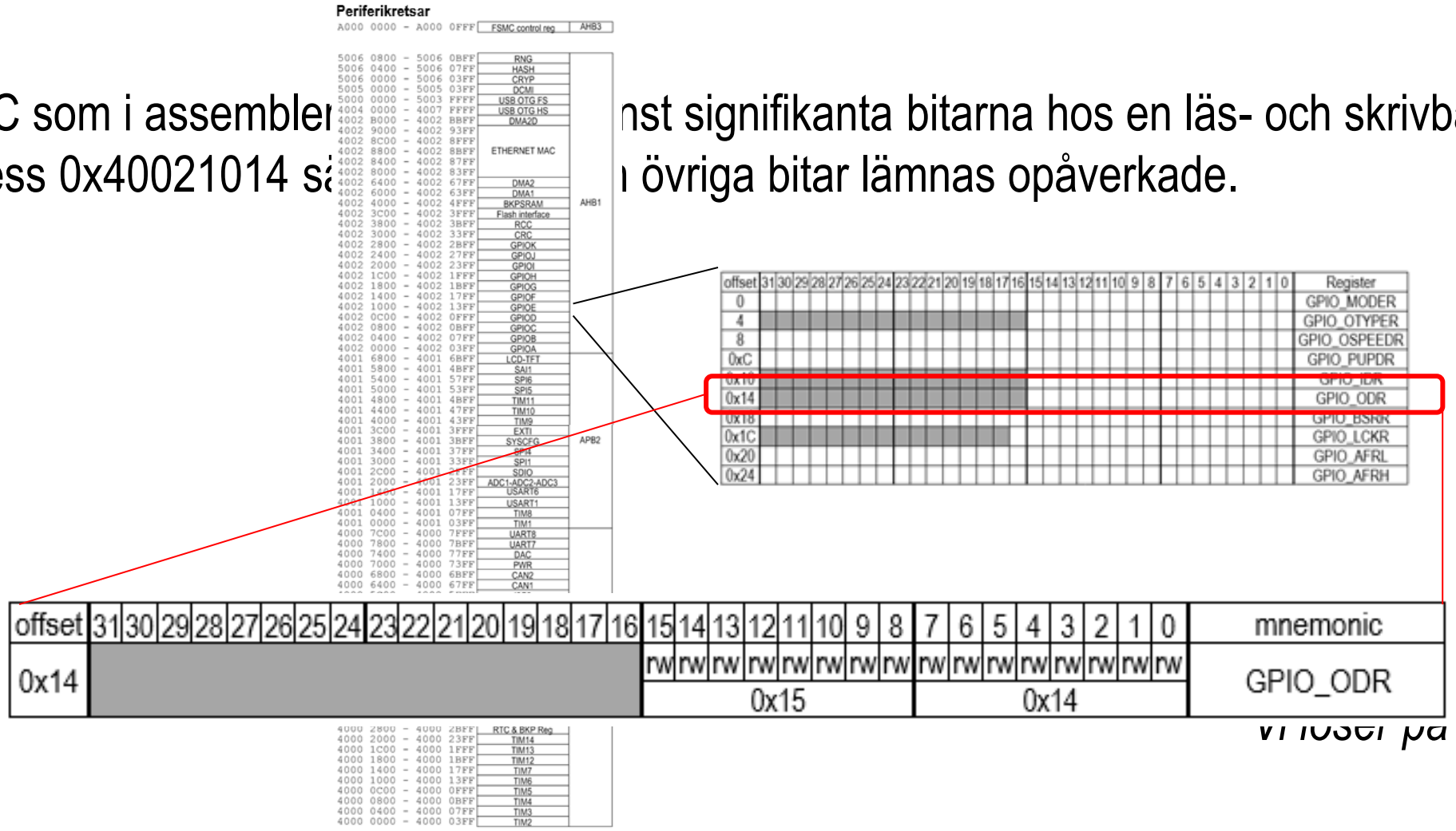
kodas detta i assembler:

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

Användning av konstant pekare

Exempel:

Visa, såväl i C som i assembler, hur man kan sätta de två mest signifikanta bitarna hos en läs- och skrivbar port på address 0x40021014 så att de får värdet 0b10 och övriga bitar lämnas opåverkade.



Visa på tavlan

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.

Ett första projekt med MD407

Demonstration: basic_io

```
void app_init ( void )
{
  * ( (unsigned long *) 0x40020C00) = 0x00005555;
}
void main(void)
{
  unsigned char c;
  app_init();
  while(1){
    c = (unsigned char) *(( unsigned short *) 0x40021010) ;
    * ( (unsigned char *) 0x40020C14) = c;
  }
}
```

Ett första projekt med MD407

Ett första projekt med MD407

Demonstration: basic_io

```
void app_init ( void )
{
  * ( (unsigned long *) 0x40020C00) = 0x00005555;
}
void main(void)
{
  unsigned char c;
  app_init();
  while(1){
    c = (unsigned char) *(( unsigned short *) 0x40021010) ;
    * ( (unsigned char *) 0x40020C14) = c;
  }
}
```