

Fält, pekare och portar

Ur innehållet

- Fält

- Pekarvariabler, pekarkonstanter och portar

- Pekarreferenser

Läsanvisningar:

- Arbetsbok kap 2

- Quick-guide, instruktionslistan

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**.

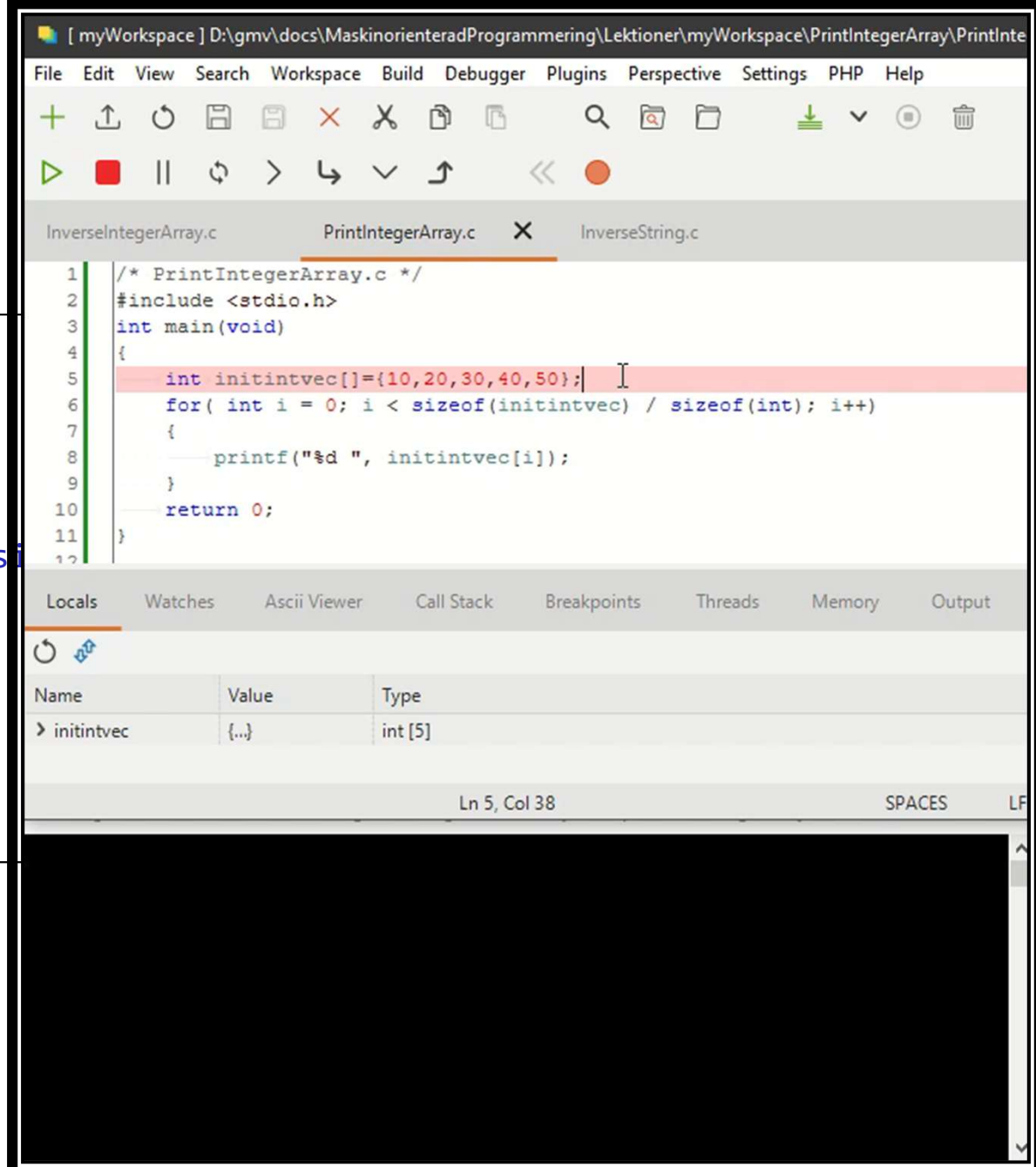
Innehållet är initialt odefinierat men ett fält kan också deklaras och initieras samtidigt:

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

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

Undersök ett fält:

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



The screenshot shows a C program in a debugger. The program is named `PrintIntegerArray.c` and is located in the workspace `D:\gmv\docs\MaskinorienteradProgrammering\Lektioner\myWorkspace\PrintIntegerArray\PrintIntegerArray.c`. The code is as follows:

```
1  /* PrintIntegerArray.c */
2  #include <stdio.h>
3  int main(void)
4  {
5      int initintvec[]={10,20,30,40,50};
6      for( int i = 0; i < sizeof(initintvec) / sizeof(int); i++)
7      {
8          printf("%d ", initintvec[i]);
9      }
10     return 0;
11 }
```

The debugger interface shows the `Locals` pane with the following information:

Name	Value	Type
initintvec	{...}	int [5]

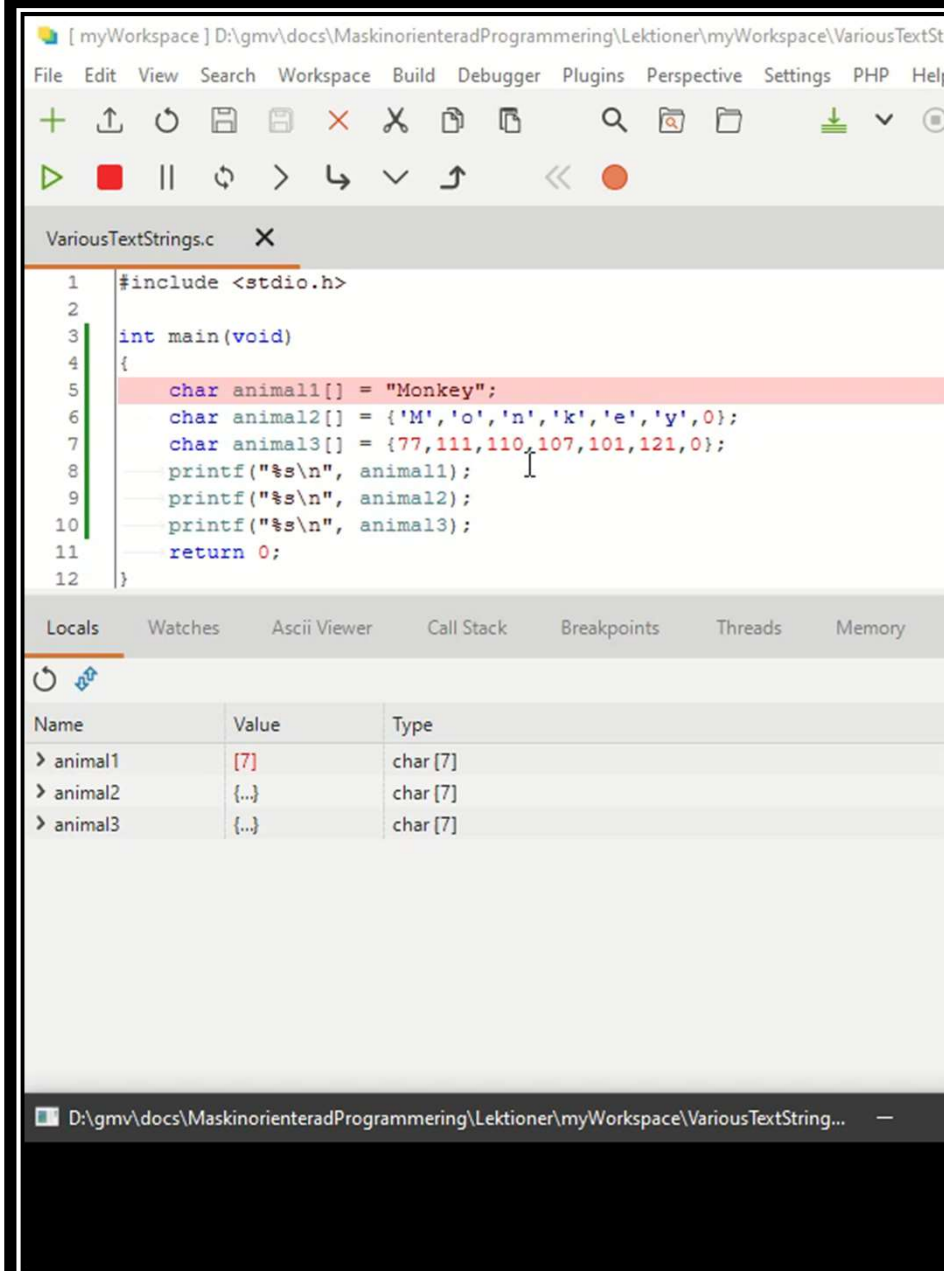
The status bar at the bottom indicates the current position is `Ln 5, Col 38`.

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

```
#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'`.

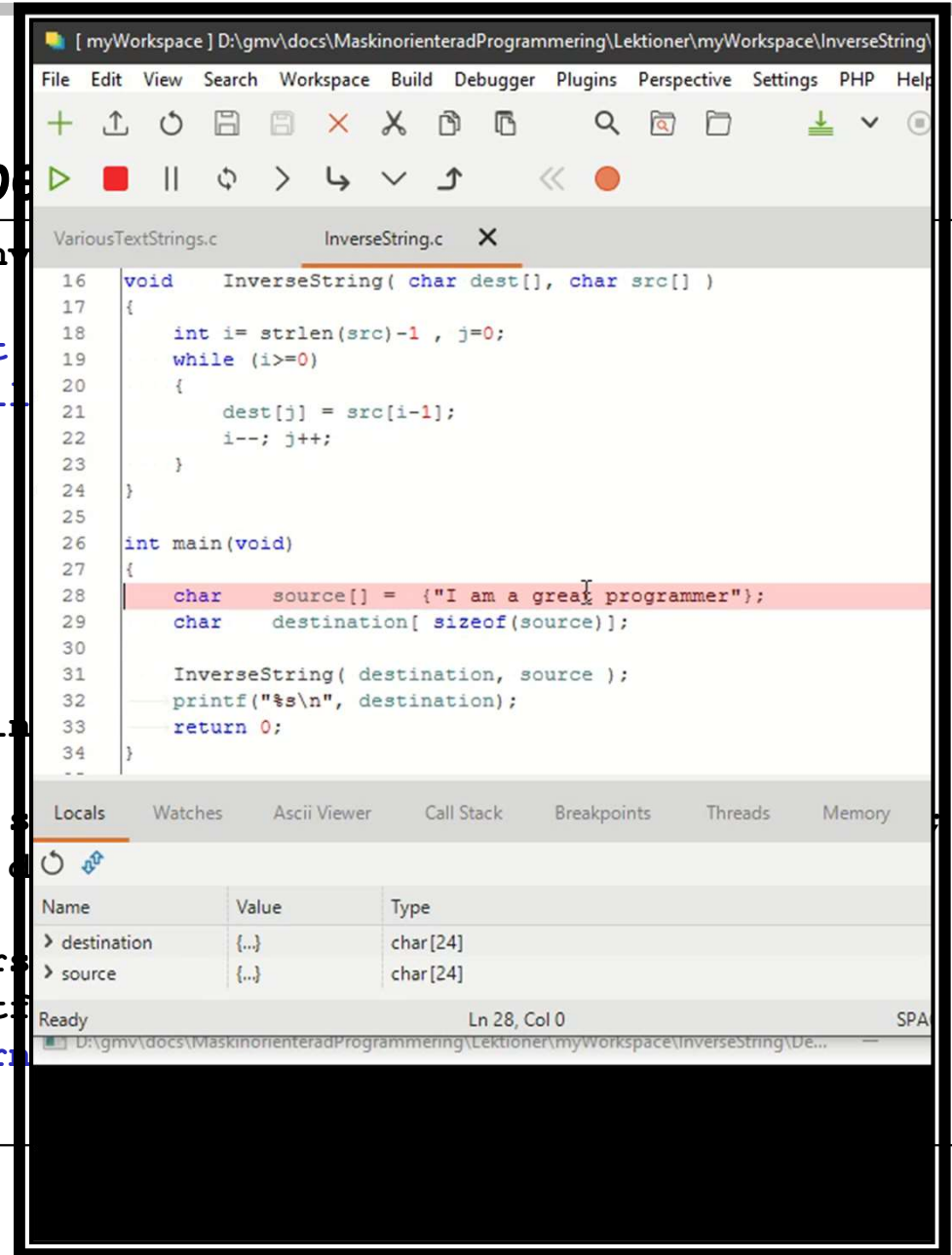
```
int strlen( char s[] )
{
    int i = 0;
    while( s[i] != '\0' )
    {
        i++;
    }
    return i + 1;
}
```

Exempel

```
void InverseString( char dest[], char src[] )
{
    int i = strlen(src)-1, j=0;
    while (i >= 0)
    {
        dest[j] = src[i];
        i--; j++;
    }
}

int main(void)
{
    char source[] = {"I am a great programmer"};
    char destination[ sizeof(source) ];

    InverseString( destination, source );
    printf("%s\n", destination);
    return 0;
}
```



The screenshot shows a C program in a debugger. The program defines a function `InverseString` that reverses a string in place. The `main` function uses `strlen` to determine the length of the source string and then calls `InverseString` to reverse it. The debugger is set to a breakpoint at line 28, and the `Locals` window shows the `destination` and `source` arrays.

```
void InverseString( char dest[], char src[] )
{
    int i = strlen(src)-1, j=0;
    while (i >= 0)
    {
        dest[j] = src[i];
        i--; j++;
    }
}

int main(void)
{
    char source[] = {"I am a great programmer"};
    char destination[ sizeof(source) ];

    InverseString( destination, source );
    printf("%s\n", destination);
    return 0;
}
```

Name	Value	Type
destination	{...}	char[24]
source	{...}	char[24]

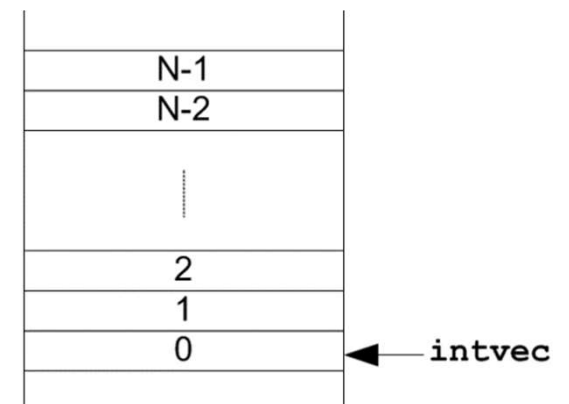
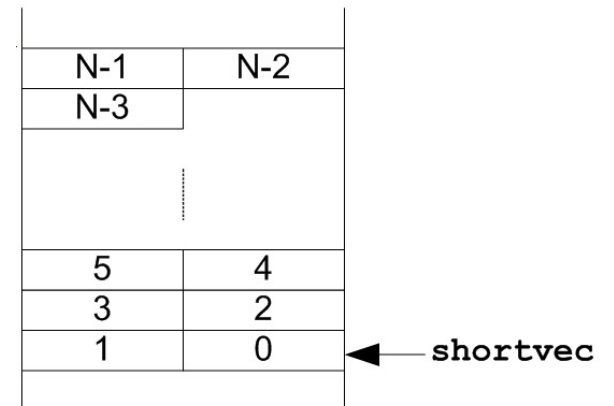
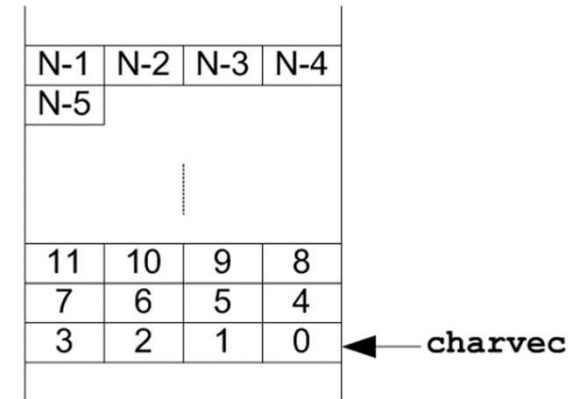
Ready Ln 28, Col 0 SPA

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 adresseringsätt

```
const k; ( $0 \leq k \leq N-1$ )
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:

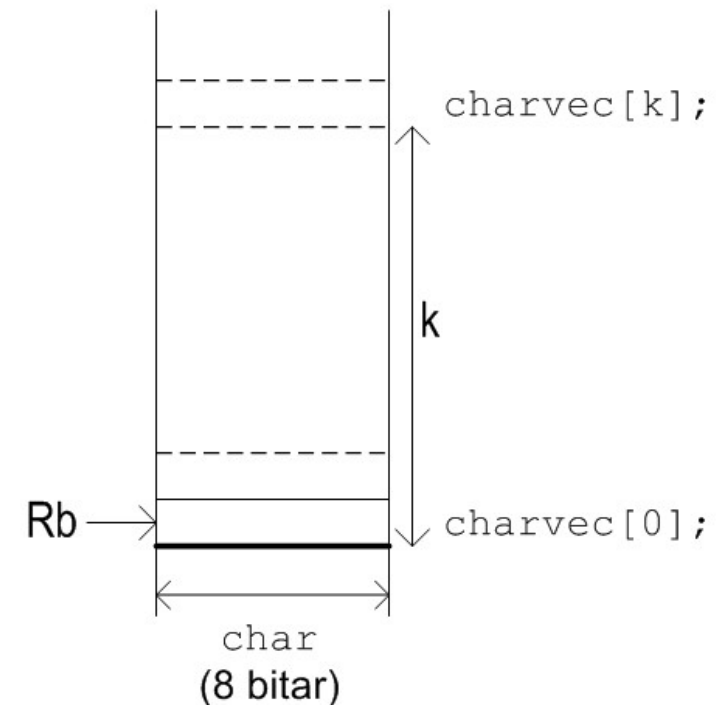
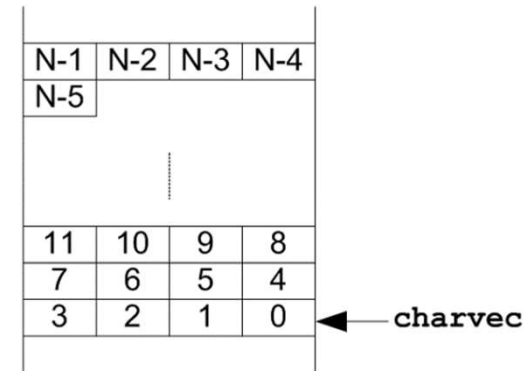
charvec[i];

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

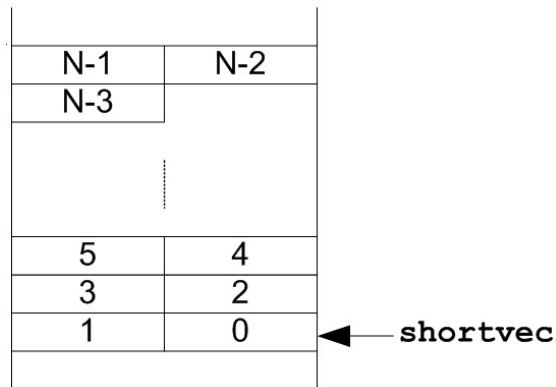
Adress: charvec+i

Register index:

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



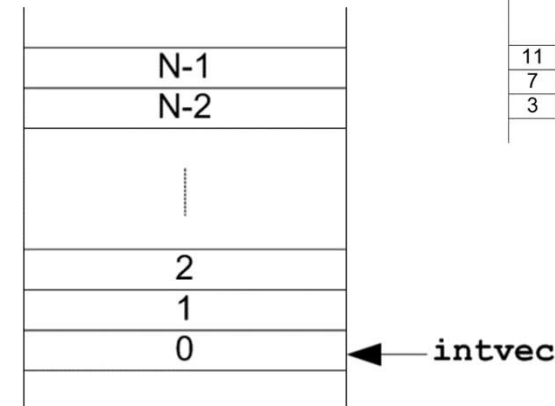
Adressberäkning och adresseringsätt



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

=shortvec + i*sizeof(short)

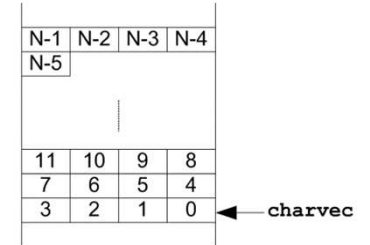
```
LDR    R0,=shortvec
LDR    R1,i
LSL    R1,R1,#1    @R1←R1×2
LDRSH  R0,[R0,R1]
```



```
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" – $N \times M$ element, indexeras $[0..N-1][0..M-1]$

Exempel:

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

Referens: $mc[i1][i2]$

Minnesadress: $=mc + ((i1 * M) + i2) * \text{sizeof}(\text{char})$

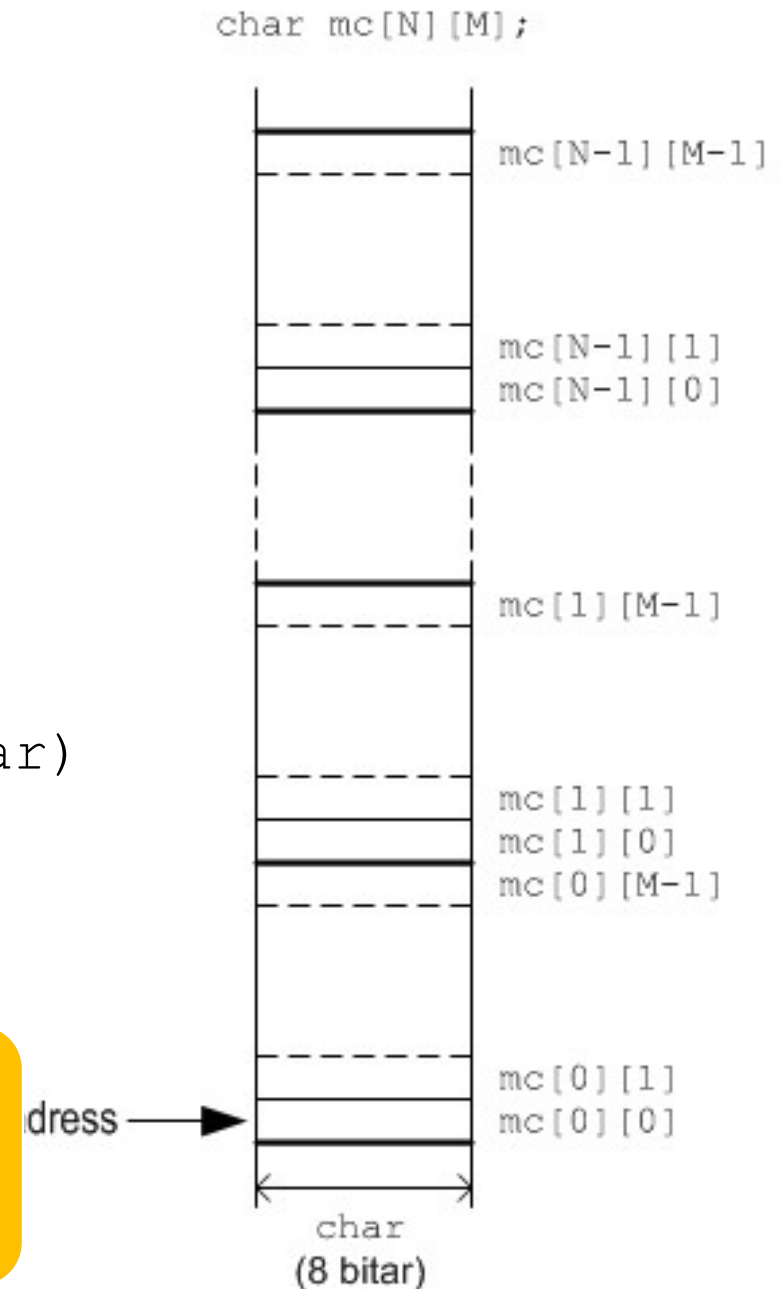
Kompilatorn ersätter multiplikationer med skift/add kombinationer eftersom `mul`-instruktionen tar betydligt fler klockcykler att utföra.

$$M = 2^x + y$$

Exempelvis då $M=10$:

$$10 = 2^3 + 2$$

$$\begin{aligned} i1 * 10 &= \\ i1 * (2^3 + 2) &= \\ i1 \ll 3 + i1 + i1 \end{aligned}$$



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...

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:

a) `veci[j]`;

1 Adress:
=veci + j*

b) `vm[i][j]`

1 Adress:
=vm + ((i*5) + j)



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

0x20046670

...

0x20030108

0x20030104

0x20030100

...

0x00000001

0x00000000

0x20030104

...

...

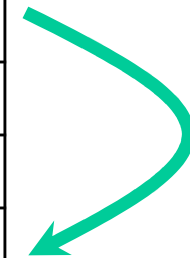
2000

...

...

...

...



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

0x20046670	0x20030108	mySalary
...	...	
0x20030108	3000	salaryLevel3
0x20030104	2000	salaryLevel2
0x20030100	1000	salaryLevel1
...	...	
0x00000001		
0x00000000		

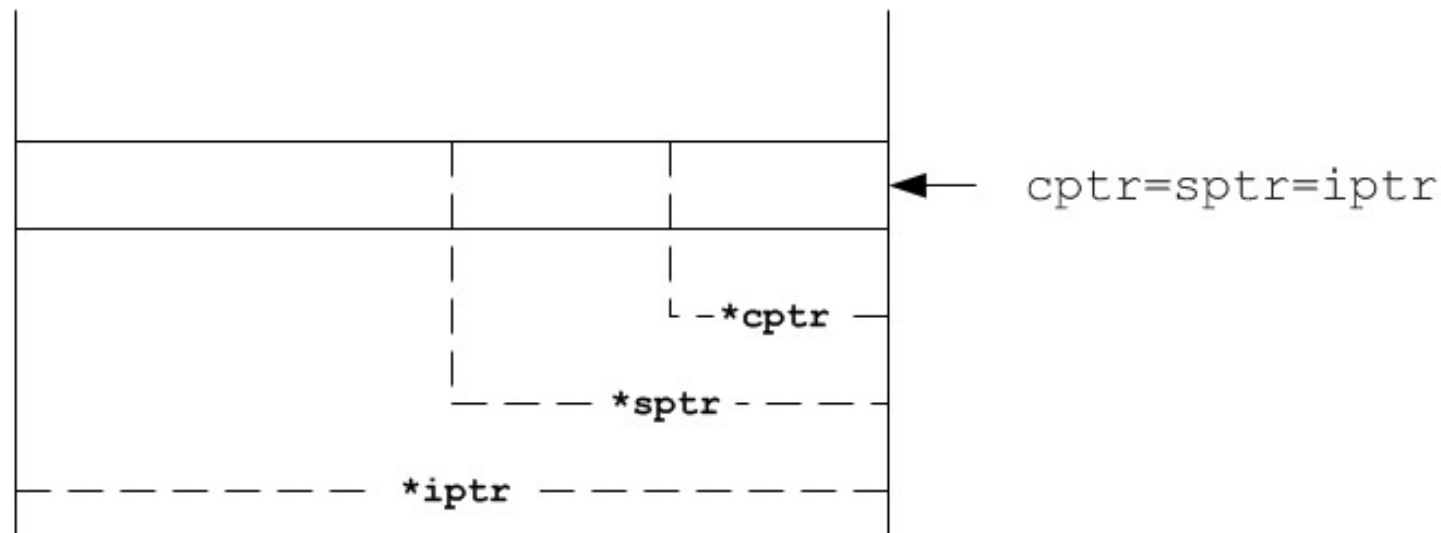
Grundläggande pekartyper, ARM/Thumb

```
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];
```

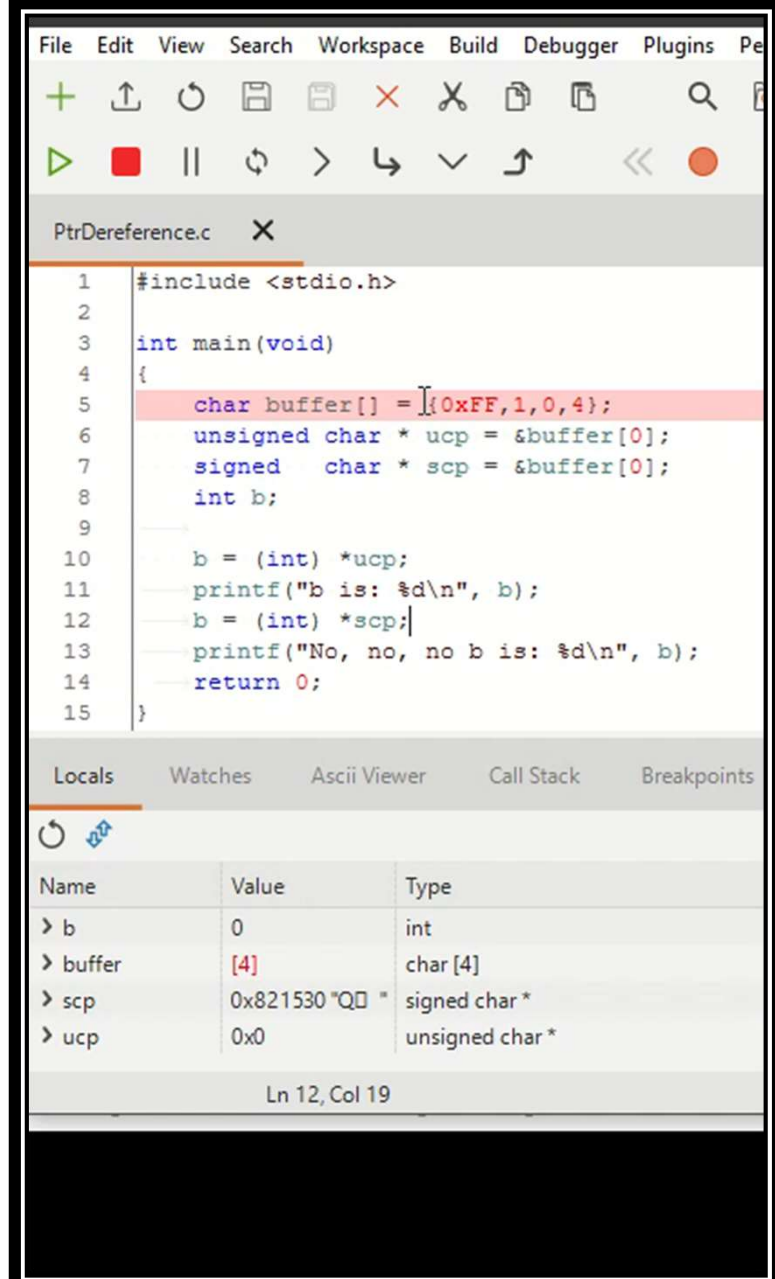
```
...
```

```
int b;
```

```
    b = (int) *ucp;
```

```
...
```

```
    b = (int) *scp;
```



```
File Edit View Search Workspace Build Debugger Plugins Pe
+ ↑ ↺ ⌨ ✖ ✂ 📄 🔍
▶ ■ || ⌂ > ↶ ∨ ↗ << ●
PtrDereference.c X
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char buffer[] = {0xFF, 1, 0, 4};
6     unsigned char * ucp = &buffer[0];
7     signed char * scp = &buffer[0];
8     int b;
9
10    b = (int) *ucp;
11    printf("b is: %d\n", b);
12    b = (int) *scp;
13    printf("No, no, no b is: %d\n", b);
14    return 0;
15 }
Locals Watches Ascii Viewer Call Stack Breakpoints
🔄 🔍
Name Value Type
> b 0 int
> buffer [4] char [4]
> scp 0x821530 "QQ" signed char *
> ucp 0x0 unsigned char *
Ln 12, Col 19
```

Varför behöver vi pekare?

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

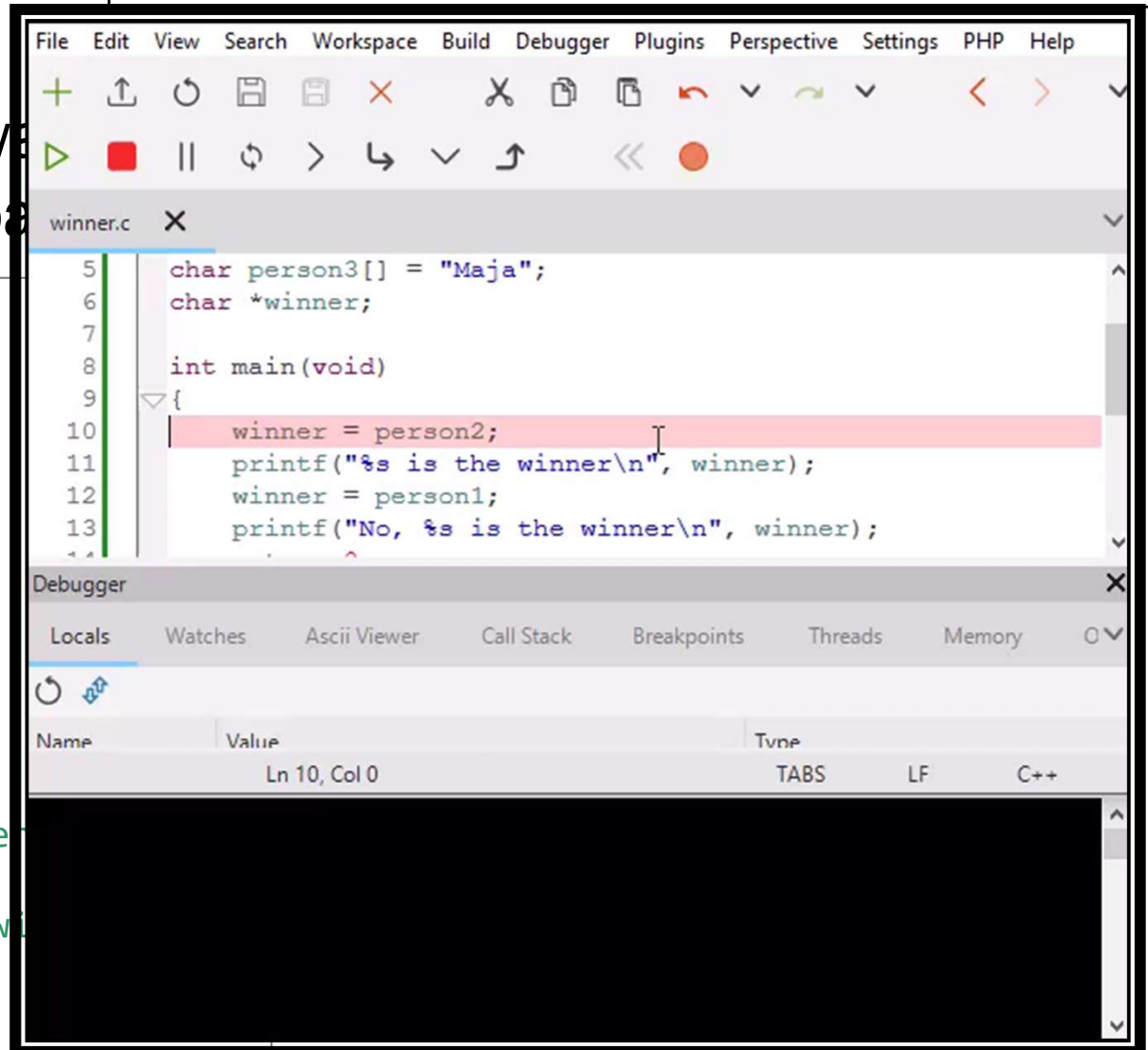
Exempel:

```
#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;
}
```

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



Möjligheter med pekare...

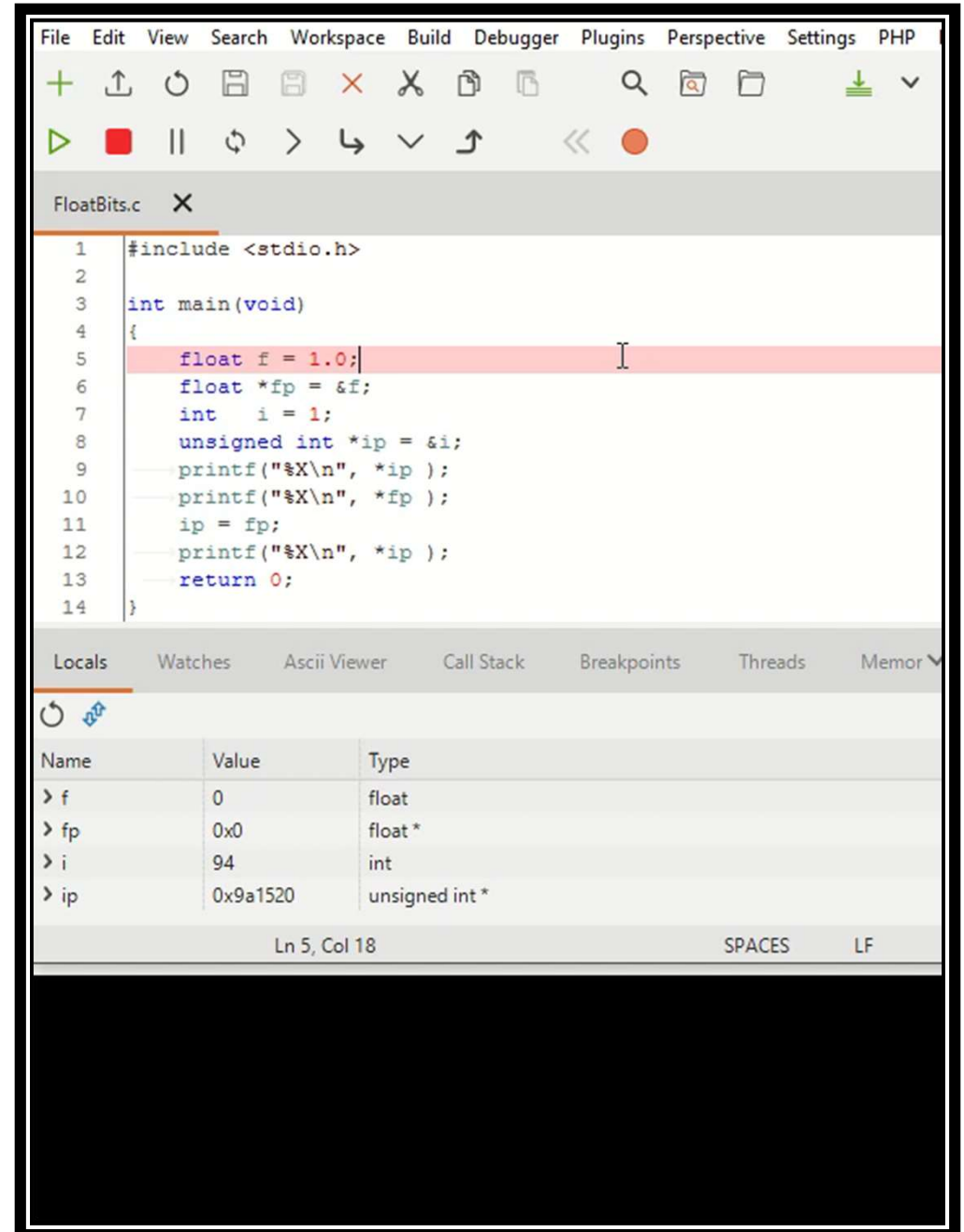
Exempel:

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...



The screenshot shows a C program named `FloatBits.c` in a debugger. The code is as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     float f = 1.0;
6     float *fp = &f;
7     int i = 1;
8     unsigned int *ip = &i;
9     printf("%X\n", *ip);
10    printf("%X\n", *fp);
11    ip = fp;
12    printf("%X\n", *ip);
13    return 0;
14 }
```

The debugger's **Locals** window shows the following variables and their values:

Name	Value	Type
> f	0	float
> fp	0x0	float *
> i	94	int
> ip	0x9a1520	unsigned int *

The status bar at the bottom indicates the cursor is at **Ln 5, Col 18**.

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

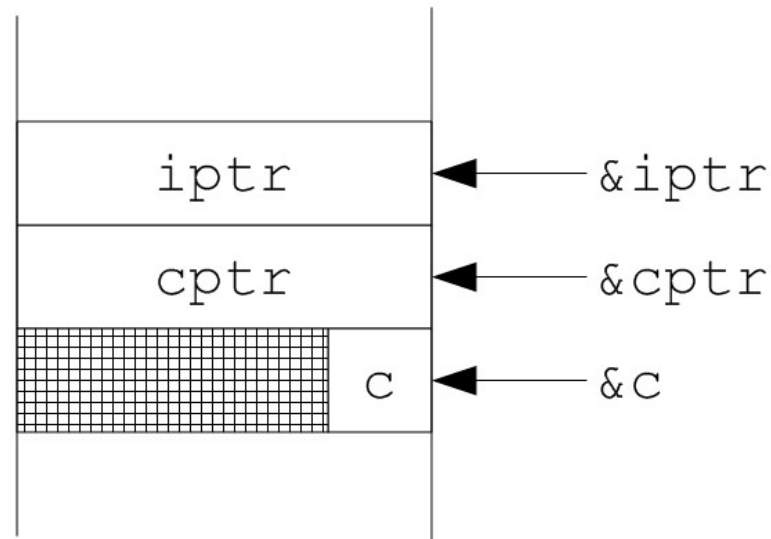
```
int strlen( char s[] )
{
    int i = 0;
    while( s[i] != '\0' )
    {
        i++;
    }
    return i + 1;
}
```

```
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.

Lösning:

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

Exempel:

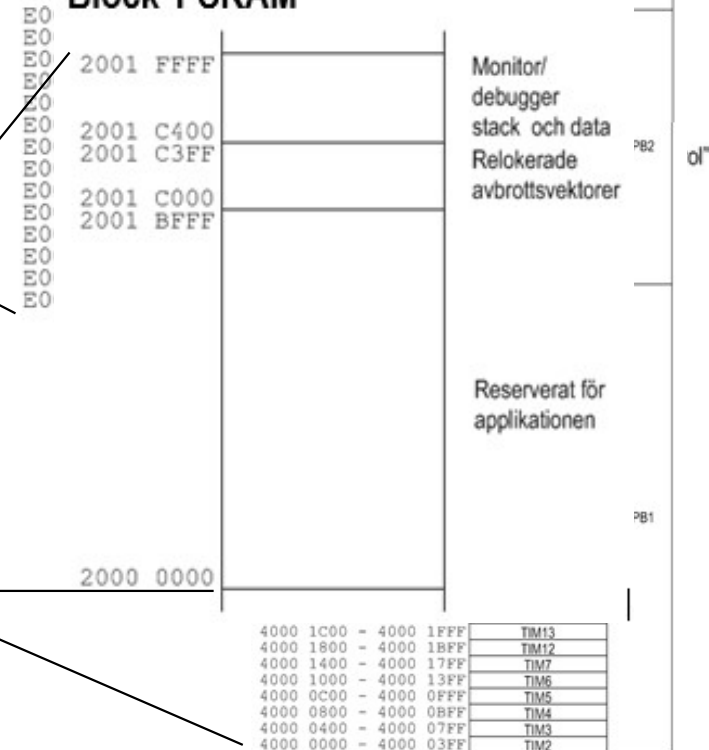
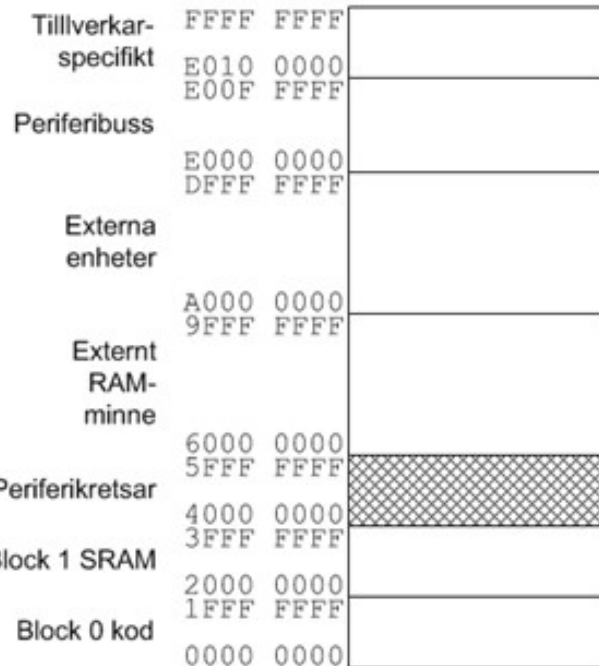
De globala variablerna `i`, och `mi` är definierade enligt:

```
int i;  
int  mi[12][8]; Visa en  
kodsekvens som evaluerar uttrycket  
&mi[i][4] till register R0.
```

Lösning:

```
@ minnesadress: =mi + (i*8+4) * 4  
    LDR    R0, =mi  
    LDR    R1, i  
    LSL    R1, R1, #3  
    ADD    R1, R1, #4  
    LSL    R1, R1, #2  
    ADD    R0, R0, R1
```


The diagram illustrates a database structure. It features a vertical axis on the left labeled 'Adressum' (Address) with an upward arrow at '0' and a downward arrow at ' 2^L-1 '. To the right of this axis is a rectangular structure representing the database. The structure has a wide base and a narrower top section, connected by a tapered middle section. Dashed lines indicate internal divisions within the structure. Below the structure, a horizontal double-headed arrow is labeled 'Den fysiska databassens bredd' (The physical database's width).



Periferikretsar

A000	0000	-	A000	0FFF	FSMC control reg	AHB3
5006	0800	-	5006	0BFF	RNG	
5006	0400	-	5006	07FF	HASH	
5006	0000	-	5006	03FF	CRYP	
5005	0000	-	5005	03FF	DCMI	
5000	0000	-	5003	FFFF	USB OTG FS	
4004	0000	-	4007	FFFF	USB OTG HS	
4002	8000	-	4002	BBFF	DMA2D	
4002	9000	-	4002	93FF		
4002	8C00	-	4002	8FFF		
4002	8800	-	4002	8BFF	ETHERNET MAC	
4002	8400	-	4002	87FF		
4002	8000	-	4002	83FF		
4002	6400	-	4002	67FF	DMA2	
4002	6000	-	4002	63FF	DMA1	
4002	4000	-	4002	4FFF	BKPSRAM	AHB1
4002	3C00	-	4002	3FFF	Flash interface	
4002	3800	-	4002	3BFF	RCC	
4002	3000	-	4002	33FF	CRC	
4002	2800	-	4002	2BFF	GPIOK	
4002	2400	-	4002	27FF	GPIOJ	
4002	2000	-	4002	23FF	GPIOI	
4002	1C00	-	4002	1FFF	GPIOH	
4002	1800	-	4002	1BFF	GPIOG	
4002	1400	-	4002	17FF	GPIOF	
4002	1000	-	4002	13FF	GPIOE	
4002	0C00	-	4002	0FFF	GPIOD	

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, bitarna hos en läs- och skriftbarhets maskin, som sätts till 1, medan övriga bitar sätts till 0.



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;
    }
}
```

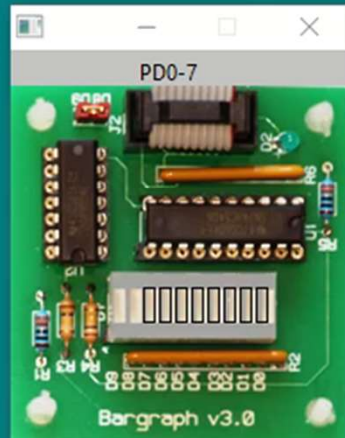
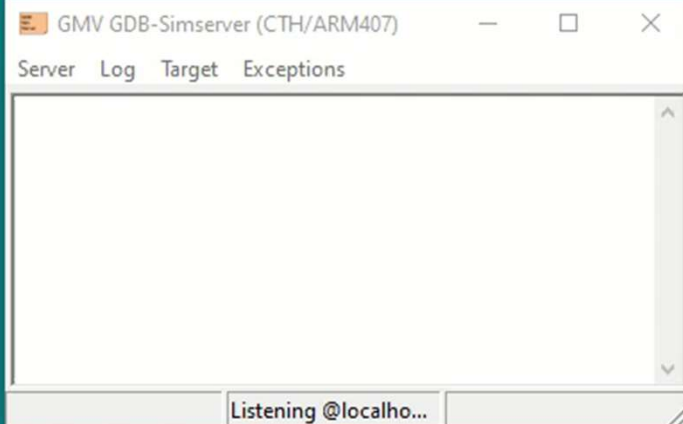
Ett första projekt med MD407

Ett första projekt med MD407

Demonstration: basic_io

```
void app_init ( void )
```

```
{  
}  
}  
void  
{  
  us  
  a  
  w  
}
```



```
[ Workbook ] D:\gm\docs\MaskinorienteradProgrammering\Exempelkod\basic_io\startup.c
File Edit View Search Workspace Build Debugger Plugins Perspective Settings PHP Help

startup.c X
1  /*
2  *  startup.c
3  *  Anslut GPIO E pin 0-7 till '8 bit dipswitch'
4  *  Anslut GPIO D pin 0-7 till '8 segment bargraph'
5  */
6  __attribute__((naked)) __attribute__((section (".start_section")))
7  void startup ( void )
8  {
9      __asm__ volatile(" LDR R0,=0x2001C000\n");          /* set stack */
10     __asm__ volatile(" MOV SP,R0\n");
11     __asm__ volatile(" BL main\n");                    /* call main */
12     __asm__ volatile(".L1: B .L1\n");                  /* never return */
13 }
14 void app_init ( void )
15 {
16     * ( (unsigned long *) 0x40020C00 ) = 0x00005555;
17 }
18 void main(void)
19 {
20     unsigned char c;
21     app_init();
22     while(1){
23         c = (unsigned char) *(( unsigned short *) 0x40021010 ) ;
24         * ( (unsigned char *) 0x40020C14 ) = c;
25     }
26 }
27

Ln 24, Col 46      TABS      LF      C++      UTF-8
```