

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`.

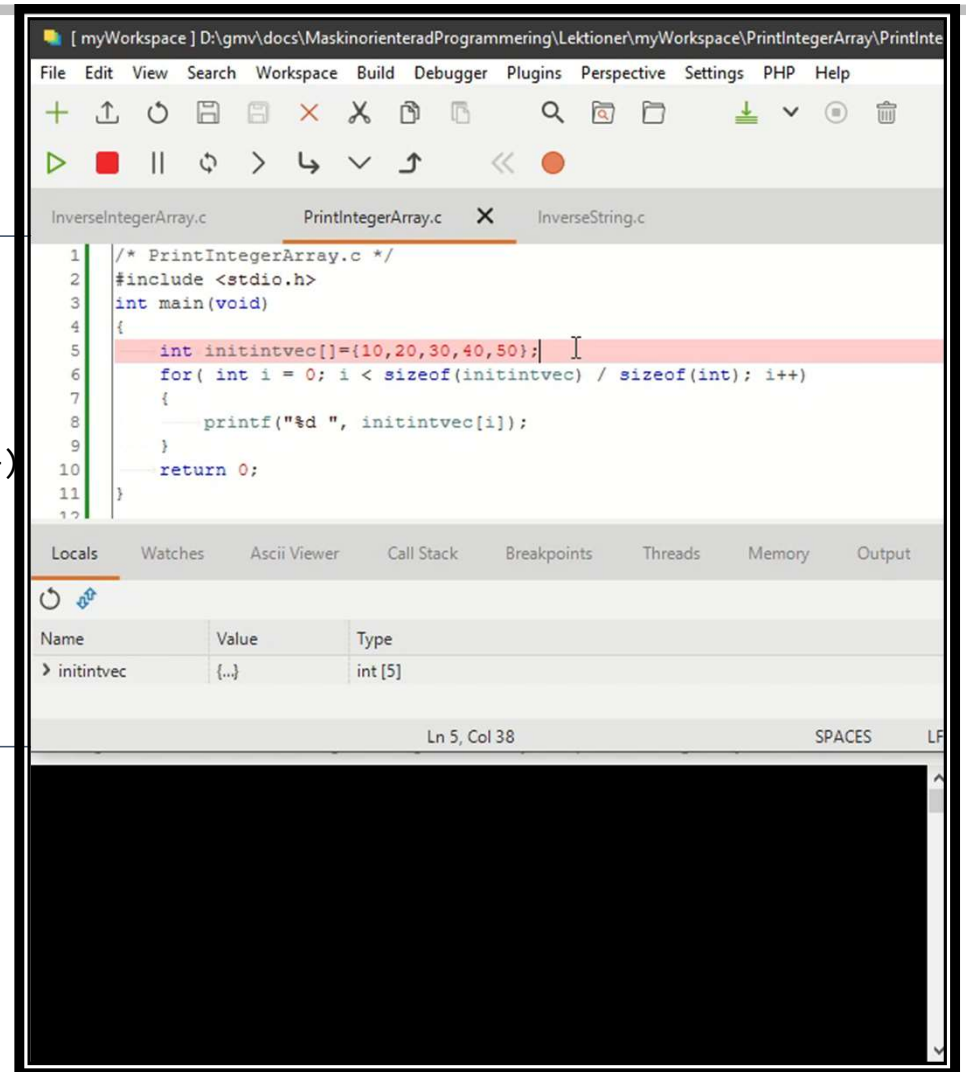
Innehållet är initialt odefinierat men ett fält kan också deklaras och initieras samtidigt:

```
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) / sizeof(int); i++)
    {
        printf("%d ", initintvec[i]);
    }
    return 0;
}
```



The screenshot shows a C program in a debugger. The program defines an array `initintvec` with values {10, 20, 30, 40, 50} and prints them. The debugger's Locals window shows the array's memory representation as a sequence of integers.

```
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 }
12
```

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

Ln 5, Col 38 SPACES LF

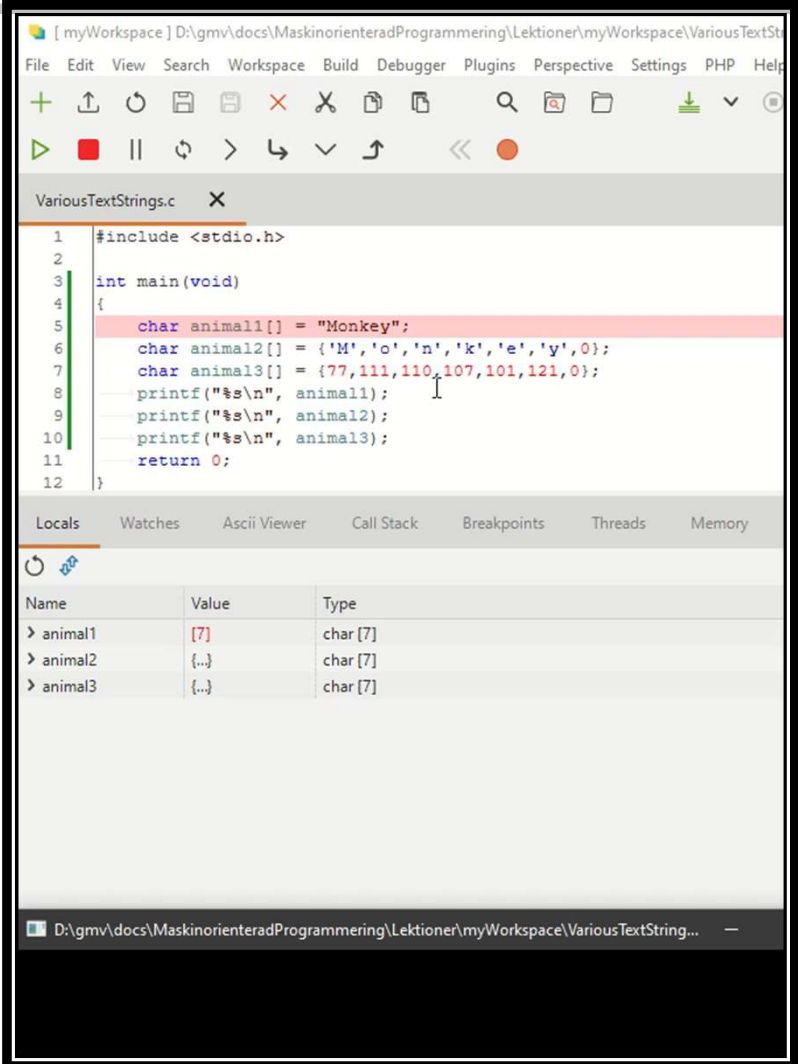
Fält och textsträngar

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;
}
```



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char animal1[] = "Monkey";
6     char animal2[] = {'M','o','n','k','e','y',0};
7     char animal3[] = {77,111,110,107,101,121,0};
8     printf("%s\n", animal1);
9     printf("%s\n", animal2);
10    printf("%s\n", animal3);
11    return 0;
12 }
```

Name	Value	Type
> animal1	[7]	char [7]
> animal2	{...}	char [7]
> animal3	{...}	char [7]

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

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

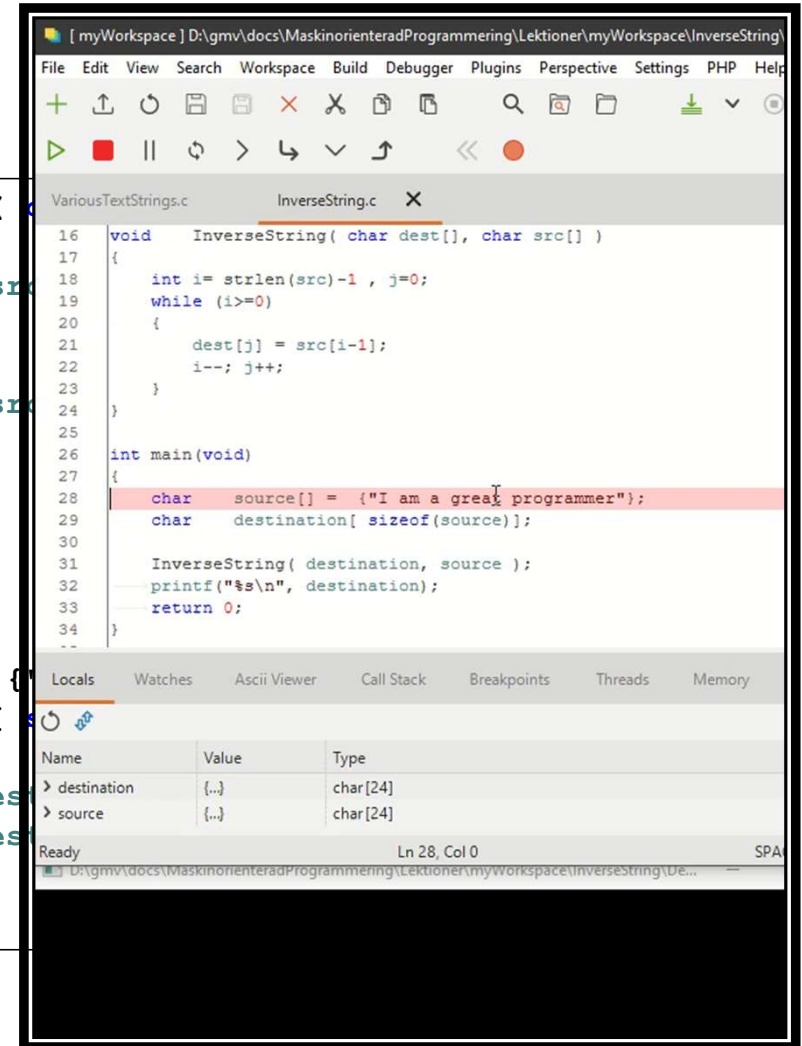
```
int strlen( char s[] )
{
    int i = 0;
    while( s[i] != '\0' )
    {
        i++;
    }
    return i + 1;
}
```

Exempel:

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



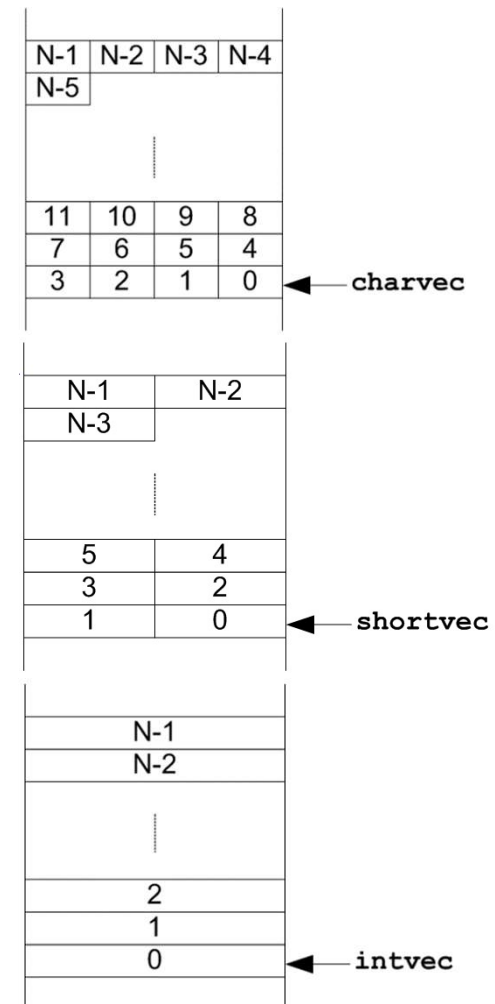
The screenshot shows a C program in a debugger. The program defines a function `InverseString` that reverses a string in place. The `main` function initializes a source string "I am a great programmer" and a destination array, then calls `InverseString` to reverse it. The debugger interface shows the source code with line numbers 16 to 34. The `Locals` pane at the bottom shows the variables `destination` and `source` as `char[24]` arrays. The status bar at the bottom indicates the program is ready at line 28, column 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: $\text{charvec} + k$

Konstant index:

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

Exempel:

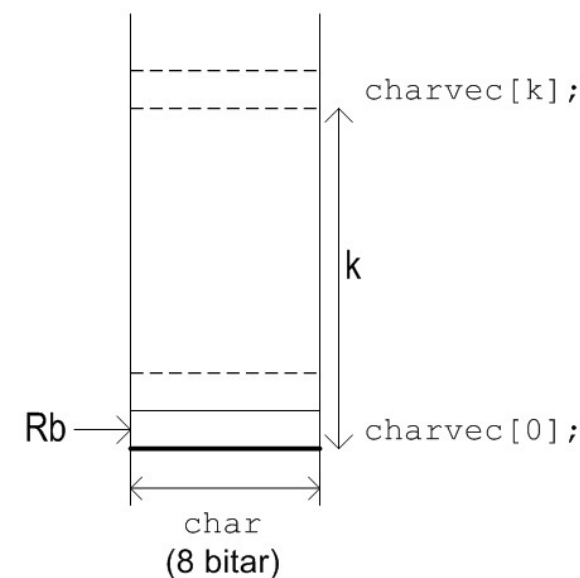
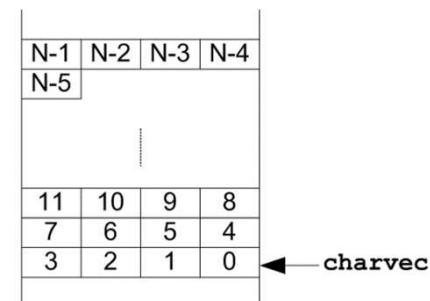
```
charvec[i];
```

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

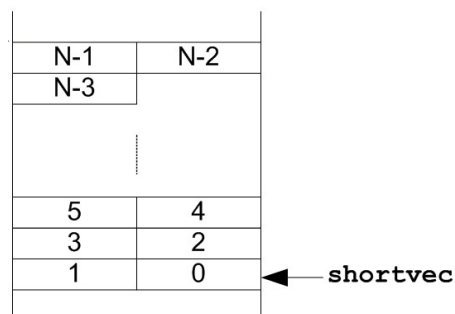
Adress: $\text{charvec} + i$

Register index:

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



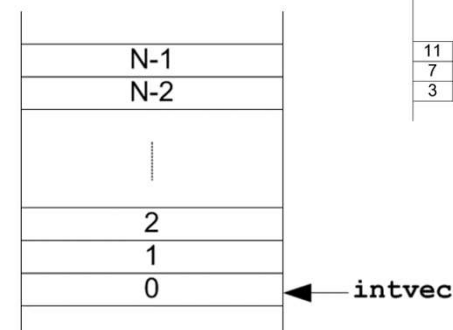
Adressberäkning och adresseringssätt



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

=shortvec + i*sizeof(short)

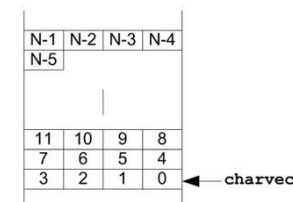
```
LDR    R0,=shortvec
LDR    R1,i
LSL    R1,R1,#1    @R1←R1×2
LDRSH  R0,[R0,R1]
```



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

Exempel:

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

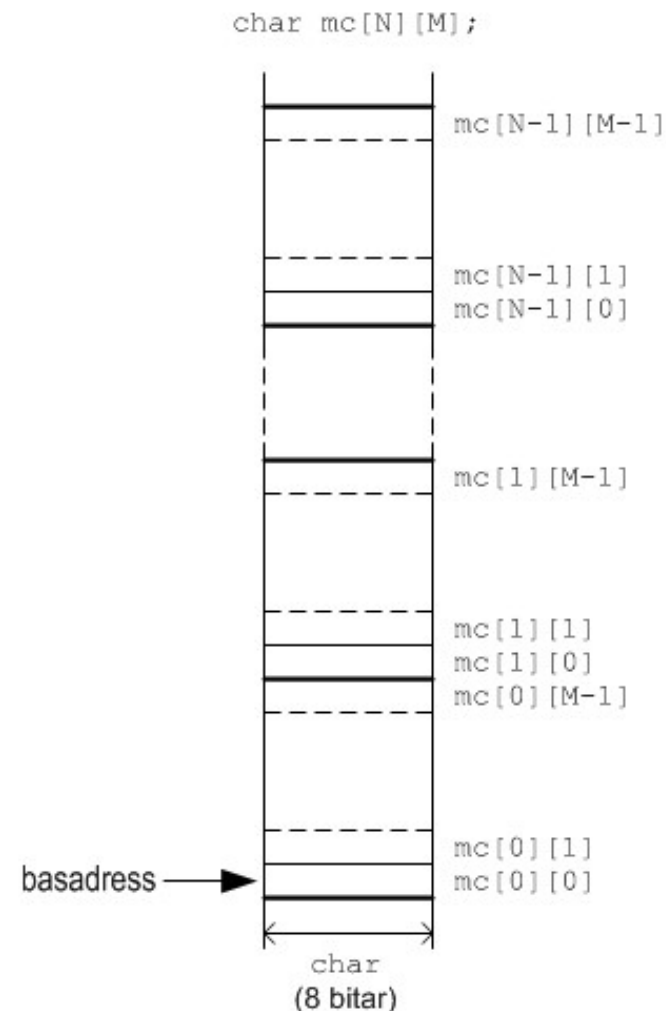
Referens: `mc[i1][i2]`

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

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

$M = 2^x + y$
Exempelvis då $M=10$:
 $10 = 2^3 + 2$

$i1 * 10 =$
 $i1 * (2^3 + 2) =$
 $i1 \ll 3 + i1 + i1$



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...

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...

a)

@ minnesadress: `=veci + j*sizeof(int)`

```
LDR    R0,j          @ R0 ← j  
LSL    R0,R0,#2      @ R0 ← j*sizeof(int)=(j*4)  
LDR    R1,=veci      @ R1 ← =veci  
LDR    R0,[R1,R0]    @ R0 ← M(=veci+(4*j)= veci[j])
```

b)

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

```
LDR    R3,i  
LSL    R2,R3,#2      @ R2 ← i*4  
ADD    R3,R2,R3      @ R3 ← (i*4)+i = (i*5)  
LDR    R2,j          @ R2 ← j  
ADD    R3,R3,R2      @ R3 ← (i*5)+j  
LSL    R3,R3,#2      @ R3 ← ((i*5)+j)*sizeof(int)  
LDR    R2,=vm  
LDR    R0,[R3,R2]    @ R0 ← M(int) ( ((i*5)+j)*4 )
```

Pekare



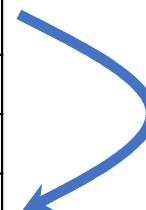
En pekare är en datatyp för en minnesadress.

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	0x20030104
...	...
0x20030108	...
0x20030104	2000
0x20030100	...
...	...
0x00000001	...
0x00000000	...



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		

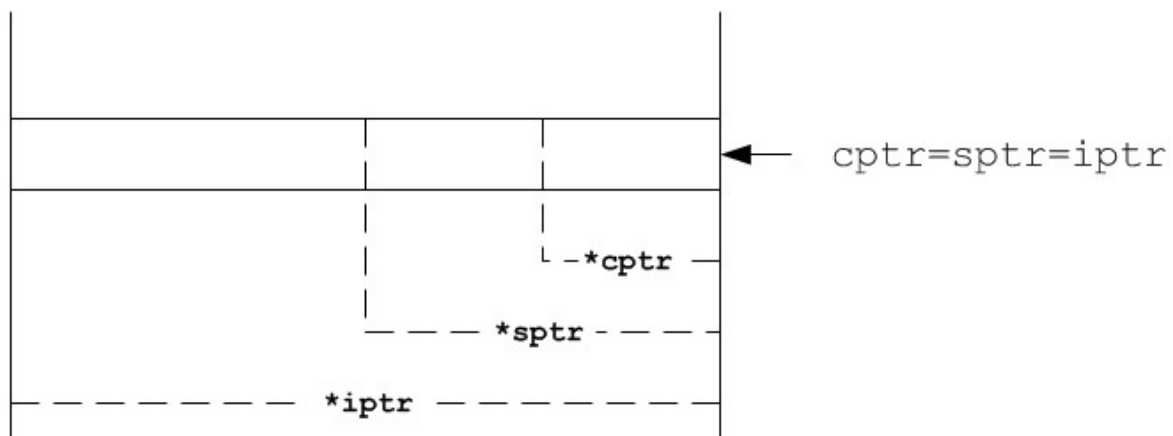
Grundläggande pekartyper, ARM/Thumb

```
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];
```

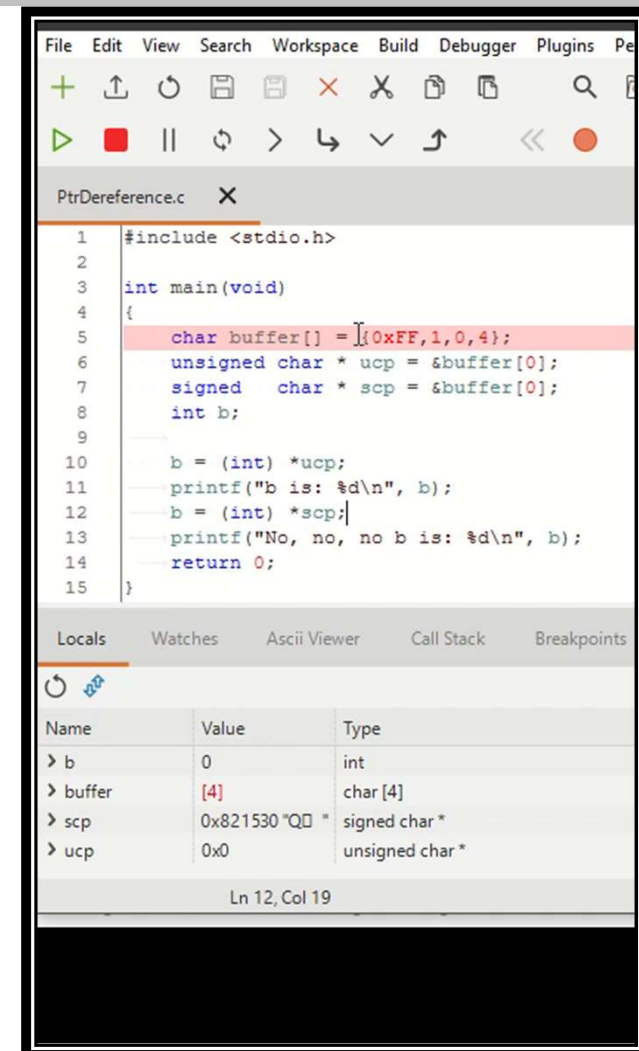
```
...
```

```
int b;
```

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

```
...
```

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



```
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 }
```

Name	Value	Type
b	0	int
buffer	[4]	char [4]
scp	0x821530 "Q"	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

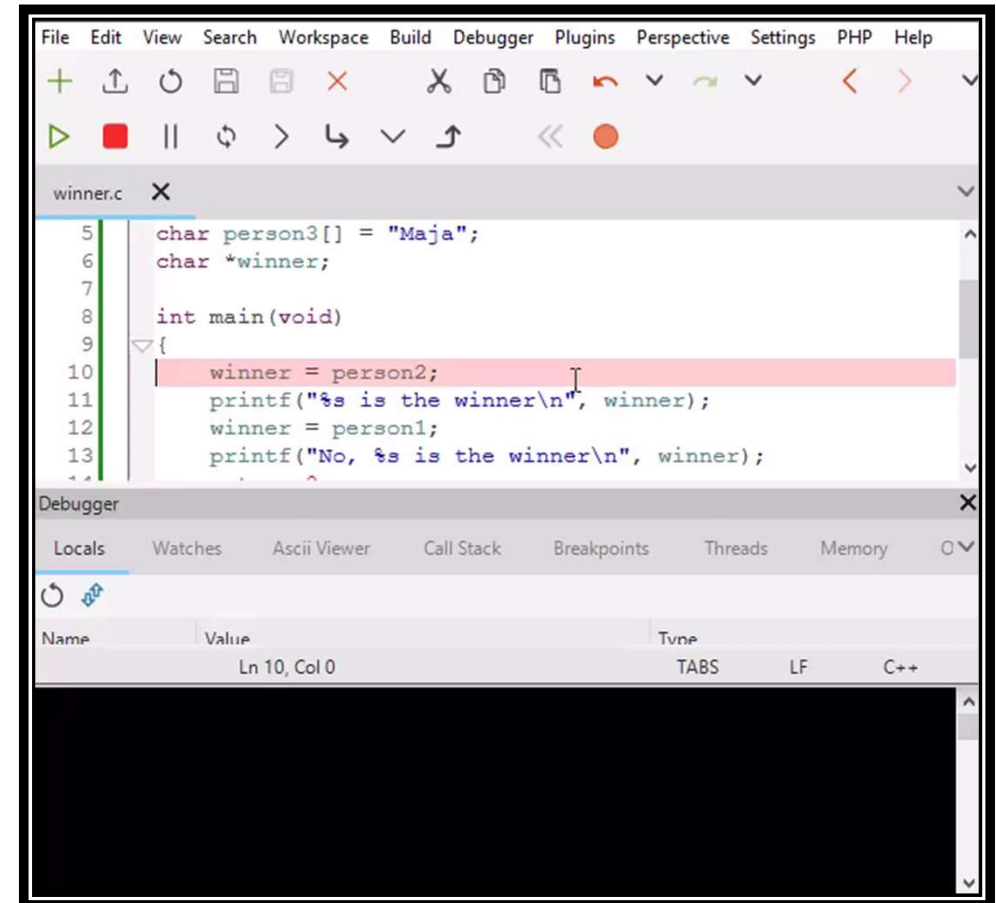
Exempel:

```
#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...

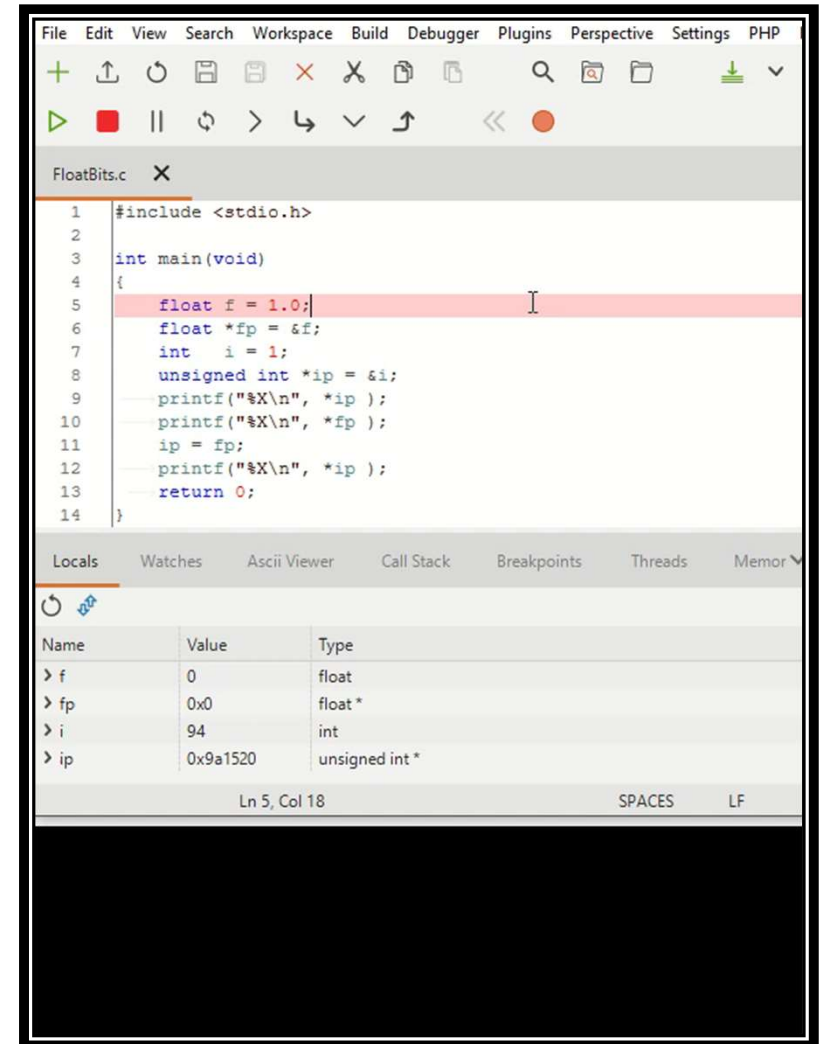
Exempel:

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 variable 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 line 5, column 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;
}
```

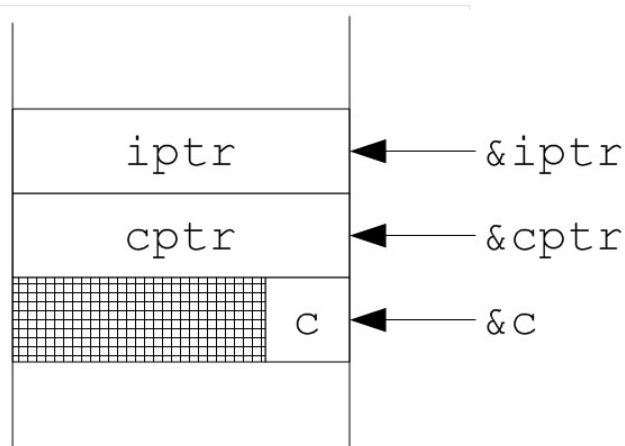


```
int strlen( char *s )
{
    int i = 0;
    while( *s )
    {
        s++; i++;
    }
    return i + 1;
}
```

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.

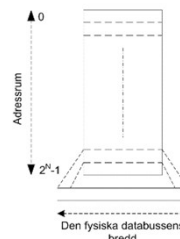
Lösning:

```
@ 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
```

Och vad är en "port"?



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



Adressrum	
Tillverkar-specifikt	FFF FFFF
Periferibuss	E010 0000 E00F FFFF
Externa enheter	E000 0000 DFFF FFFF
Extern RAM-minne	A000 0000 9FFF FFFF
Periferikretsar	6000 0000 5FFF FFFF
Block 1 SRAM	4000 0000 3FFF FFFF
Block 0 kod	2000 0000 1FFF FFFF 0000 0000

Periferikretsar Block 1 SRAM

E000 0000	2001 FFFF
E000 0000	2001 C400
E000 0000	2001 C3FF
E000 0000	2001 C000
E000 0000	2001 BFFF
E000 0000	2000 0000

Monitor/
debugger
stack och data
Relokerade
avbrottsvektorer

Reserverat för
applikationen

Periferikretsar		AHB3	
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 83FF	DMA2D	
4002 9000	4002 93FF		
4002 8C00	4002 8FFF	ETHERNET MAC	
4002 8800	4002 8BFF		
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	GPIOE	
4002 1000	4002 13FF	GPIOE	
4002 0C00	4002 0FFF	GPIOC	
4002 0800	4002 0BFF	GPIOC	
4002 0400	4002 07FF	GPIOB	
4002 0000	4002 03FF	GPIOA	
4001 6800	4001 6BFF	LCD-TFT	
4001 5800	4001 5BFF	SAI1	
4001 5400	4001 57FF	SPB	
4001 5000	4001 53FF	SPB	
4001 4800	4001 4BFF	TIM11	
4001 4400	4001 47FF	TIM10	
4001 4000	4001 43FF	TIM9	
4001 3C00	4001 3FFF	EXTI	
4001 3800	4001 3BFF	SYSCFG	APB2
4001 3400	4001 37FF	SPB	
4001 3000	4001 33FF	SPB	
4001 2C00	4001 2FFF	SDIO	
4001 2000	4001 23FF	ADC1-ADC3-ADC3	
4001 1400	4001 17FF	USART6	
4001 1000	4001 13FF	USART1	
4001 0400	4001 07FF	TIM8	
4001 0000	4001 03FF	TIM1	
4000 7C00	4000 7FFF	UART8	
4000 7800	4000 7BFF	UART7	
4000 7400	4000 77FF	DAC	
4000 7000	4000 73FF	PWR	
4000 6800	4000 6BFF	CAN2	
4000 6400	4000 67FF	CAN1	
4000 5C00	4000 5BFF	I2C3	
4000 5800	4000 57FF	I2C2	
4000 5400	4000 53FF	I2C1	
4000 5000	4000 53FF	UART5	
4000 4C00	4000 4FFF	UART4	
4000 4800	4000 4BFF	USART3	APB1
4000 4400	4000 47FF	USART2	
4000 4000	4000 43FF	I2Sext	
4000 3C00	4000 3FFF	SPB/I2S3	
4000 3800	4000 3BFF	SPB/I2S2	
4000 3400	4000 37FF	I2Sext	
4000 3000	4000 33FF	WWDG	
4000 2C00	4000 2FFF	WWDG	
4000 2800	4000 2BFF	RTC & BKP Reg	
4000 2000	4000 23FF	TIM14	
4000 1C00	4000 1FFF	TIM13	
4000 1800	4000 1BFF	TIM12	
4000 1400	4000 17FF	TIM7	
4000 1000	4000 13FF	TIM6	
4000 0C00	4000 0FFF	TIM5	
4000 0800	4000 0BFF	TIM4	
4000 0400	4000 07FF	TIM3	
4000 0000	4000 03FF	TIM2	

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, en konstant signifikanta bitarna hos en läs- och skrivbar port på address 0x40021014 så att de övriga bitar lämnas opåverkade.

Periferikretsar
A000 0000 - A000 0FFF FSMC control reg. AHBS

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 8BFF	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	SRAM
4002 3C00 - 4002 3FFF	Flash interface
4002 3800 - 4002 3BFF	RCC
4002 3000 - 4002 33FF	CRC
4002 2800 - 4002 2BFF	GPIOA
4002 2400 - 4002 27FF	GPIOB
4002 2000 - 4002 23FF	GPIOC
4002 1C00 - 4002 1FFF	GPIOE
4002 1800 - 4002 1BFF	GPIOF
4002 1400 - 4002 17FF	GPIOG
4002 1000 - 4002 13FF	GPIOH
4002 0C00 - 4002 0FFF	GPIOI
4002 0800 - 4002 0BFF	GPIOJ
4002 0400 - 4002 07FF	GPIOK
4002 0000 - 4002 03FF	GPIOA
4001 6800 - 4001 6BFF	LCD-TFT
4001 5800 - 4001 5BFF	SAI1
4001 5400 - 4001 57FF	SPB
4001 5000 - 4001 53FF	SPB
4001 4800 - 4001 4BFF	TIM11
4001 4400 - 4001 47FF	TIM10
4001 4000 - 4001 43FF	TIM9
4001 3C00 - 4001 3FFF	EXTI
4001 3800 - 4001 3BFF	SYSCFG
4001 3400 - 4001 37FF	AFIO
4001 3000 - 4001 33FF	SPI1
4001 2C00 - 4001 2FFF	SDIO
4001 2000 - 4001 23FF	ADC1-ADC3
4001 1400 - 4001 17FF	USART16
4001 1000 - 4001 13FF	USART11
4001 0400 - 4001 07FF	TIM8
4001 0000 - 4001 03FF	TIM1
4000 7C00 - 4000 7FFF	UART8
4000 7800 - 4000 7BFF	UART7
4000 7400 - 4000 77FF	DAC
4000 7000 - 4000 73FF	PWR
4000 6800 - 4000 6BFF	CAN2
4000 6400 - 4000 67FF	CAN1

GPIO_ODR Register

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	Register
0																																GPIO_MODER	
4																																GPIO_OTYPER	
8																																GPIO_OSPEEDR	
0xC																																GPIO_PUPDR	
0x10																																GPIO_IDR	
0x14																																GPIO_ODR	
0x18																																GPIO_BSRR	
0x1C																																GPIO_LCKR	
0x20																																GPIO_AFR1	
0x24																																GPIO_AFRH	

GPIO_ODR Register (Detailed)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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																																	GPIO_ODR

GPIO_ODR Register (Bit Fields)

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
--------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---

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 adress 0x40021014 sätts till 1, medan övriga bitar lämnas opåverkade.

```
#define GPIO_ODR ((volatile unsigned short *)0x40021014)
*GPIO_ODR |= 0xF;
```

```
LDR      R3,=0x40021014 @ R3 = 0x40021014
LDRH     R2,[R3]         @ R2 = (short) M(0x40021014)
MOV      R3,#0xF         @ R3 = 0xF
ORR      R2,R2,R3        @ R2 = (short) M(0x40021014) | 0xF
STRH     R2,[R3]         @ M(0x40021014) = (short) M(0x40021014) | 0xF
```

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;
    }
}
```

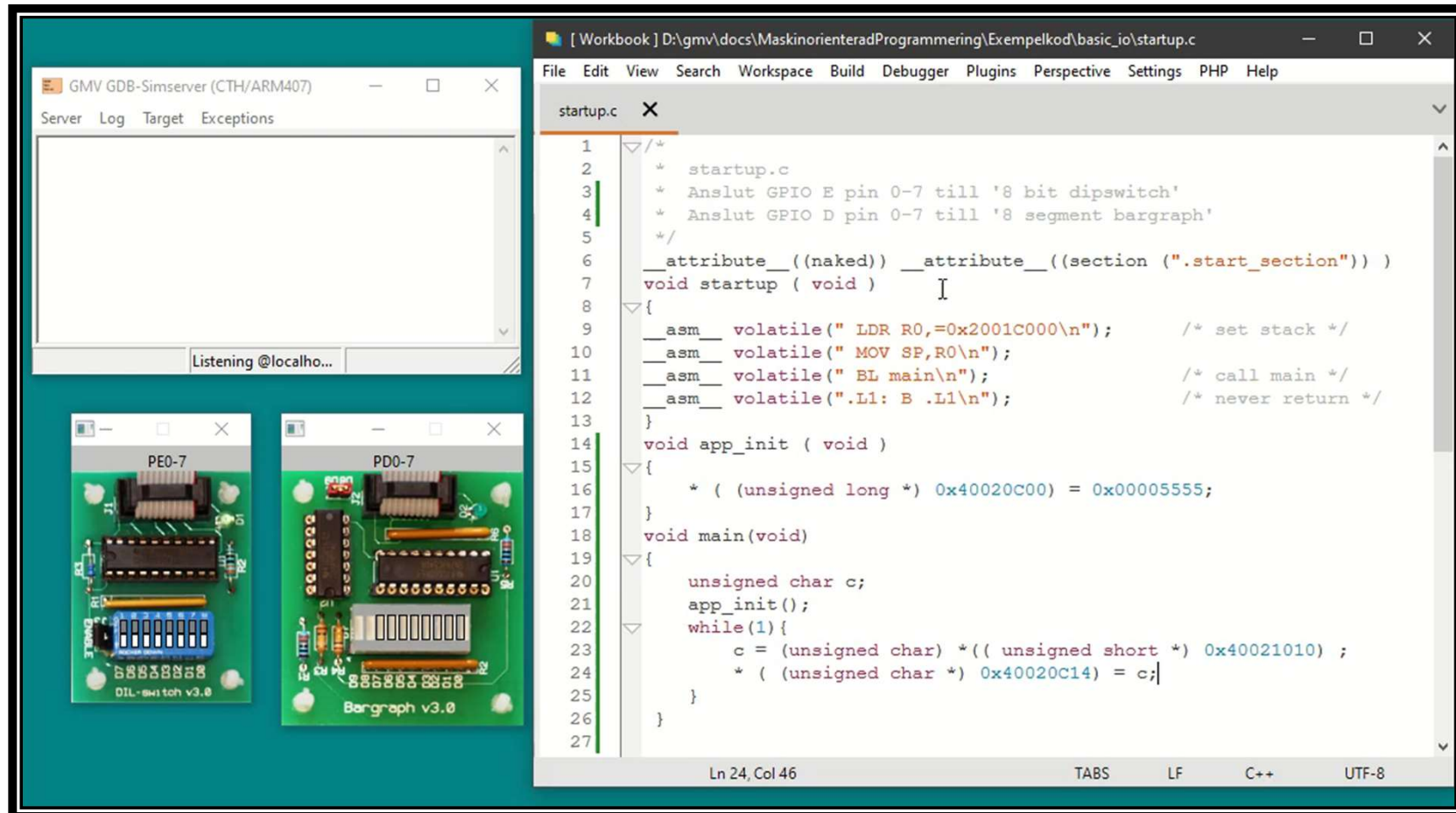
Ett första projekt med MD407

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;
    }
}
```



The screenshot displays the GMV GDB-Simserver (CTH/ARM407) interface. The top window shows the startup.c file with the following code:

```
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
```

Below the code window, two hardware components are shown: PE0-7 (DIP-switch v3.0) and PD0-7 (Bargraph v3.0).