

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`.

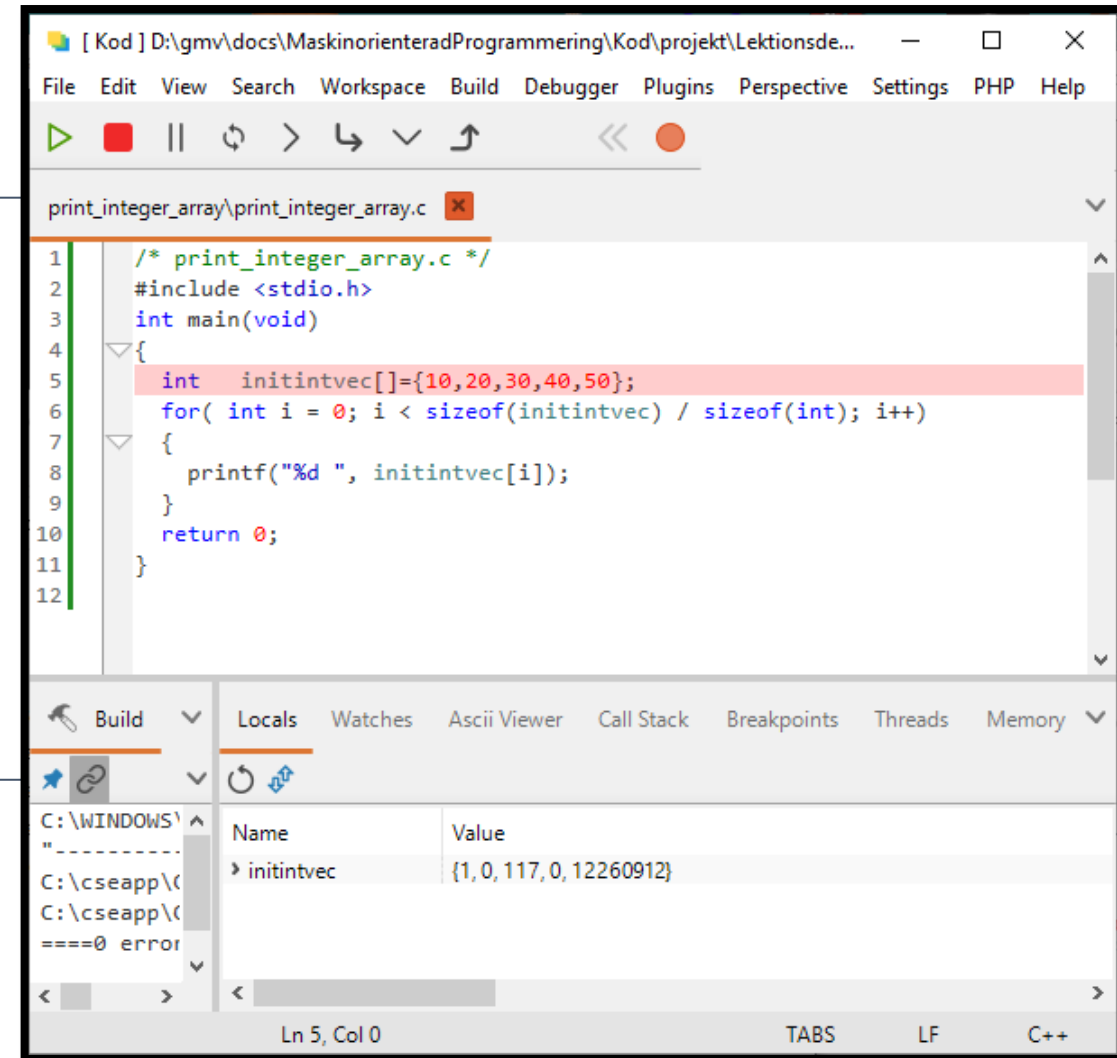
Innehållet är initialt odefinierat men ett fält kan också deklaras och initieras samtidigt:

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

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

Undersök ett fält:

```
/* 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;
}
```



The screenshot shows a C++ IDE window titled "[Kod] D:\gmv\docs\MaskinorienteradProgrammering\Kod\projekt\Lektionsde...". The menu bar includes File, Edit, View, Search, Workspace, Build, Debugger, Plugins, Perspective, Settings, PHP, and Help. The toolbar contains icons for running, stopping, stepping through, and other debugging actions. The main editor displays the source code of `print_integer_array.c`, with line 5 highlighted: `int initintvec[]={10,20,30,40,50};`. The bottom panel shows the "Locals" window with the following table:

Name	Value
initintvec	{1, 0, 117, 0, 12260912}

The status bar at the bottom indicates "Ln 5, Col 0", "TABS", "LF", and "C++".

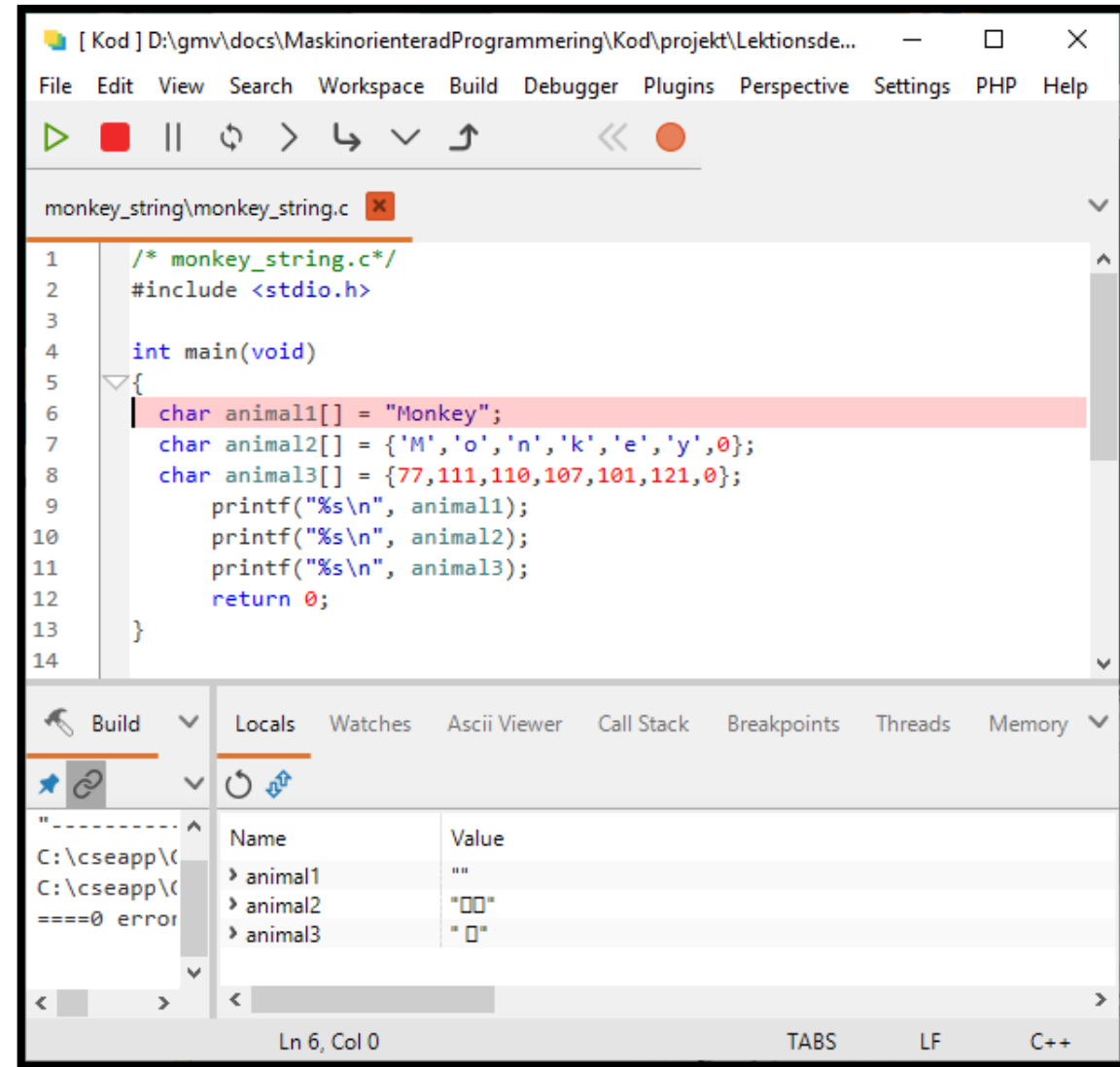
Fält och textsträngar

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;
}
```



```
[ Kod ] D:\gmv\docs\MaskinorienteradProgrammering\Kod\projekt\Lektionsde...
File Edit View Search Workspace Build Debugger Plugins Perspective Settings PHP Help

monkeys_string\monkey_string.c x
1  /* monkey_string.c */
2  #include <stdio.h>
3
4  int main(void)
5  {
6  char animal1[] = "Monkey";
7  char animal2[] = {'M','o','n','k','e','y','\0'};
8  char animal3[] = {77,111,110,107,101,121,0};
9  printf("%s\n", animal1);
10 printf("%s\n", animal2);
11 printf("%s\n", animal3);
12 return 0;
13 }
14

Build Locals Watches Ascii Viewer Call Stack Breakpoints Threads Memory
C:\cseapp\ C:\cseapp\
====0 error

Name Value
> animal1 ""
> animal2 ""
> animal3 ""

Ln 6, Col 0 TABS LF C++
```

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

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

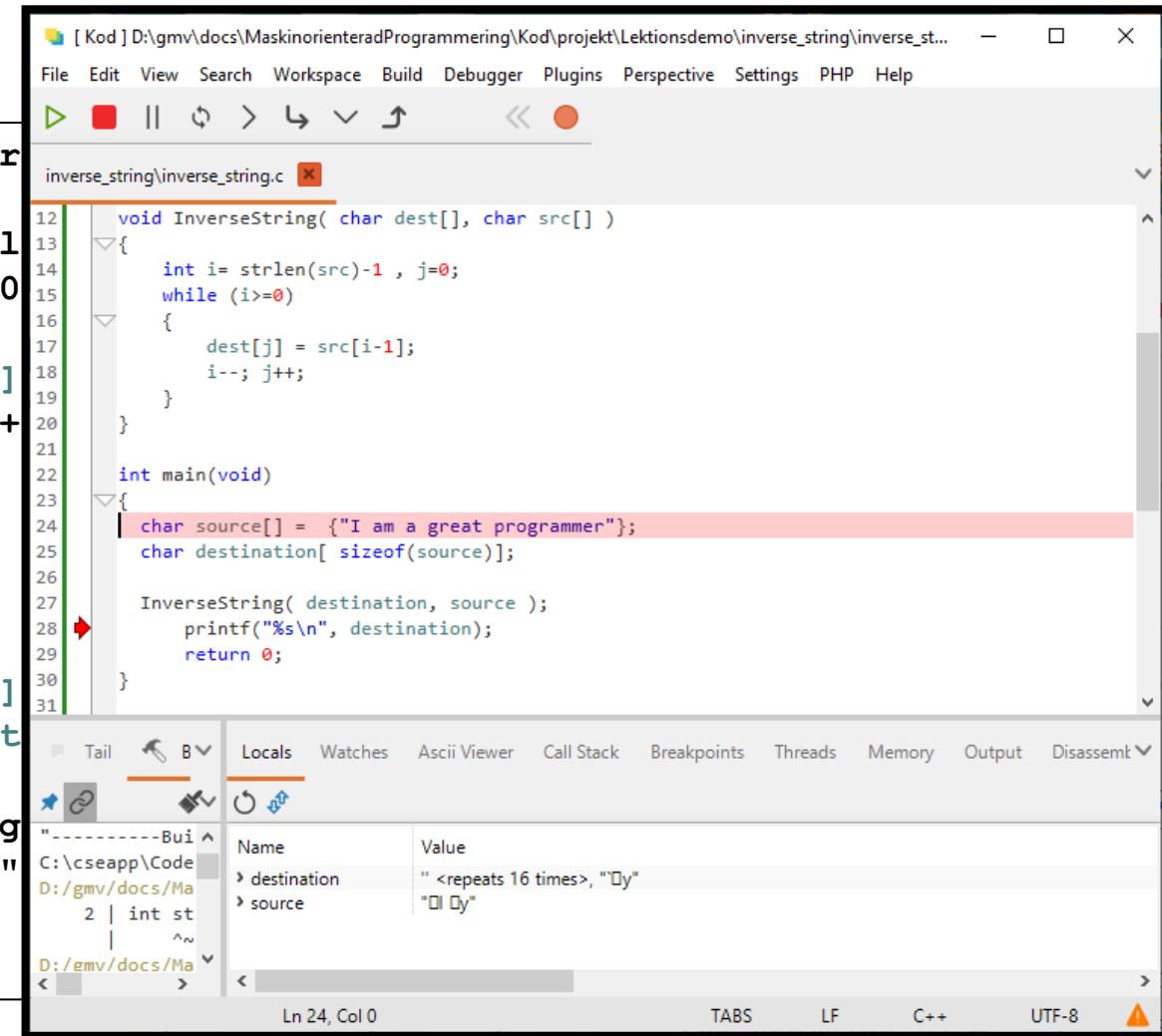
```
int strlen( char s[] )
{
    int i = 0;
    while( s[i] != '\0' )
    {
        i++;
    }
    return i + 1;
}
```

Exempel:

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

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

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



The screenshot shows a code editor window titled "inverse_string\inverse_string.c". The code implements a function to reverse a string in place. The function signature is `void InverseString( char dest[], char src[] )`. Inside the function, `int i = strlen(src) - 1` and `int j = 0` are declared. A `while (i >= 0)` loop copies characters from the end of the source string to the beginning of the destination string: `dest[j] = src[i]; i--; j++;`. The `main` function declares `char source[] = {"I am a great programmer"};` and `char destination[ sizeof(source)];`, then calls `InverseString( destination, source );` and prints the result with `printf("%s\n", destination);`.

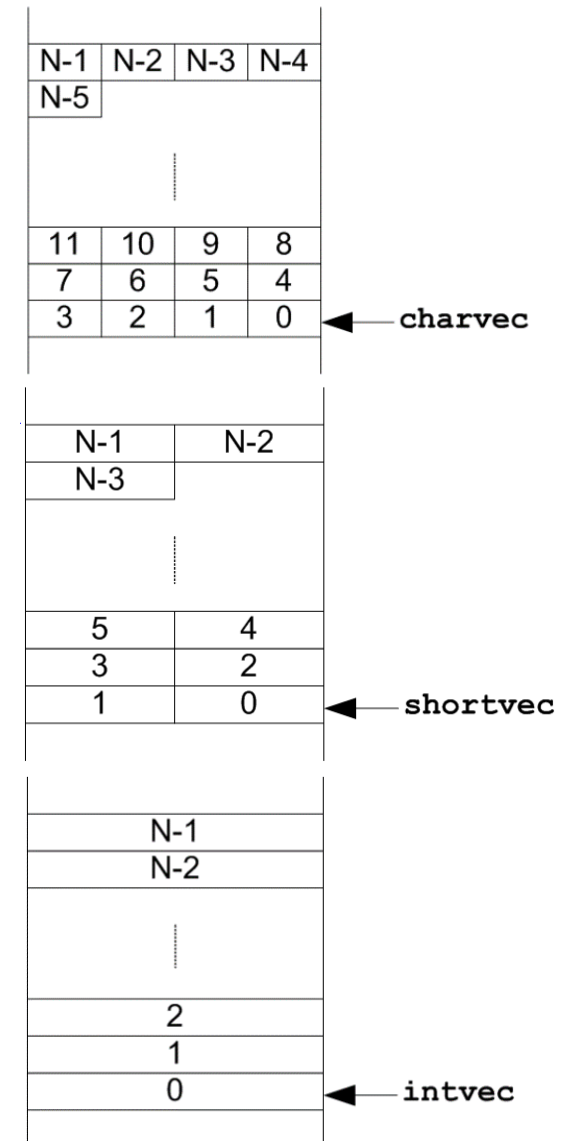
Name	Value
destination	" <repeats 16 times>, "Dy"
source	"I am a great programmer"

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

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

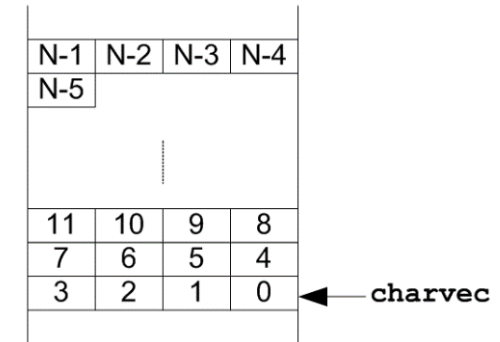
```
short shortvec[N];  
sizeof(shortvec) = N*2 bytes
```

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



Adressberäkning och adresseringssätt

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

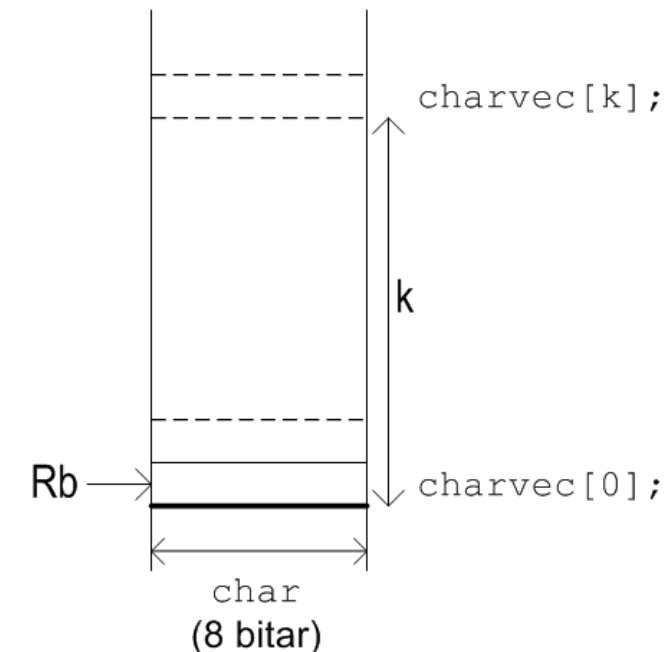
```
charvec[i];
```

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

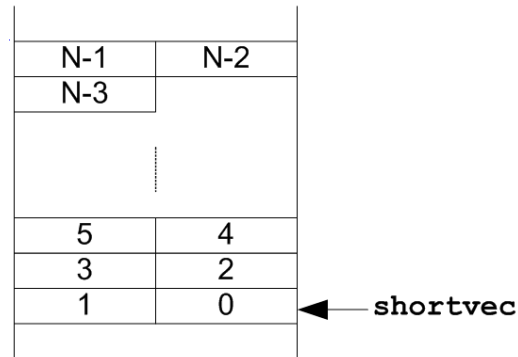
Adress: $\text{charvec} + i$

Register index:

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



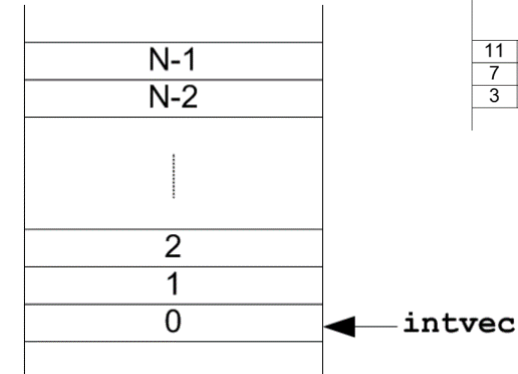
Adressberäkning och adresseringssätt



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

=shortvec + i*sizeof(short)

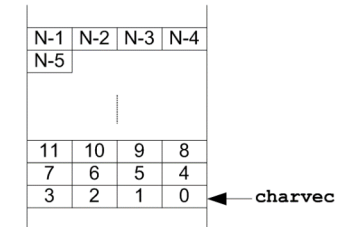
```
LDR    R0,=shortvec  
LDR    R1,i  
LSL    R1,R1,#1    @R1←R1×2  
LDRSH  R0,[R0,R1]
```



```
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;
```

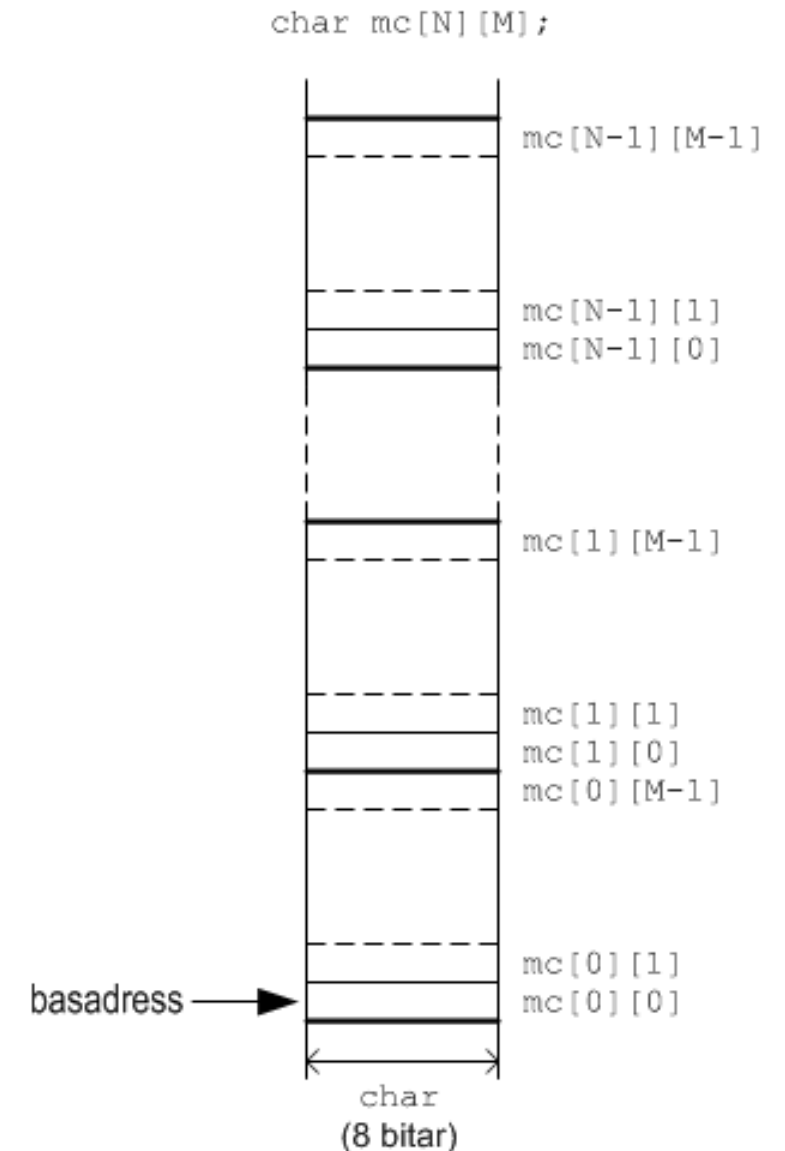
Referens: `mc[i1][i2]`

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...

Adress:

$\text{=veci} + j * \text{sizeof}(\text{int})$

Adress:

$\text{=vm} + ((i * 5) + j) * \text{sizeof}(\text{int})$

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:
 $\text{=veci} + j * \text{sizeof}(\text{int})$

b) `vm[i][j]`

Adress:
 $\text{=vm} + ((i * 5) + j) * \text{sizeof}(\text{int})$

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

0x20046670

...

0x20030108

0x20030104

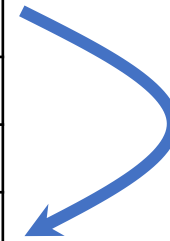
0x20030100

...

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

0x20030104

0x20030100

...

0x00000001

0x00000000

0x20030108
...
3000
2000
1000
...

mySalary

salaryLevel3

salaryLevel2

salaryLevel1

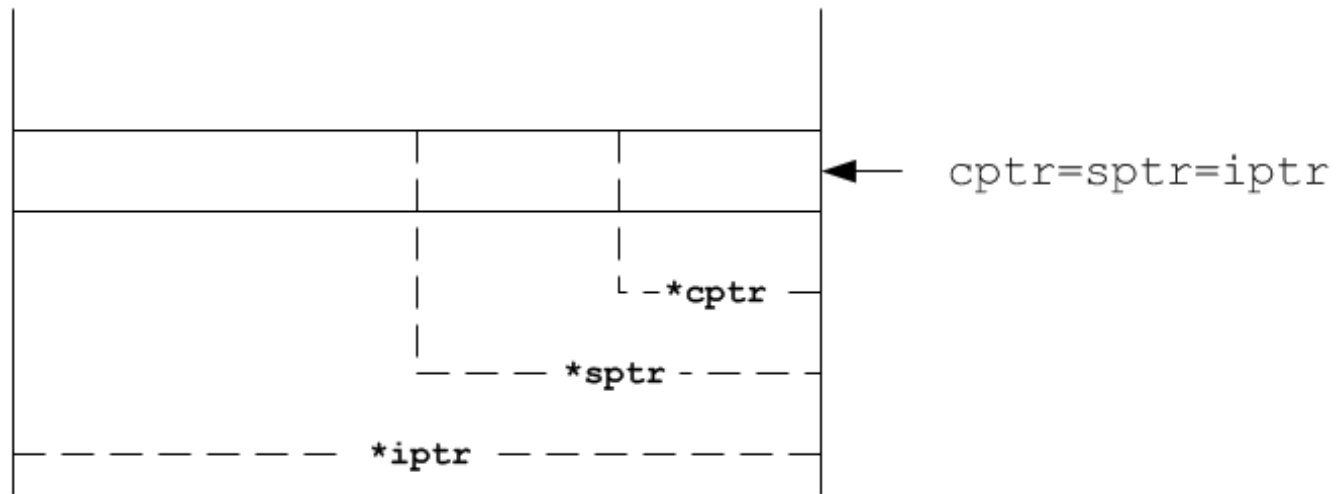
Grundläggande pekartyper, ARM/Thumb

```
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];
```

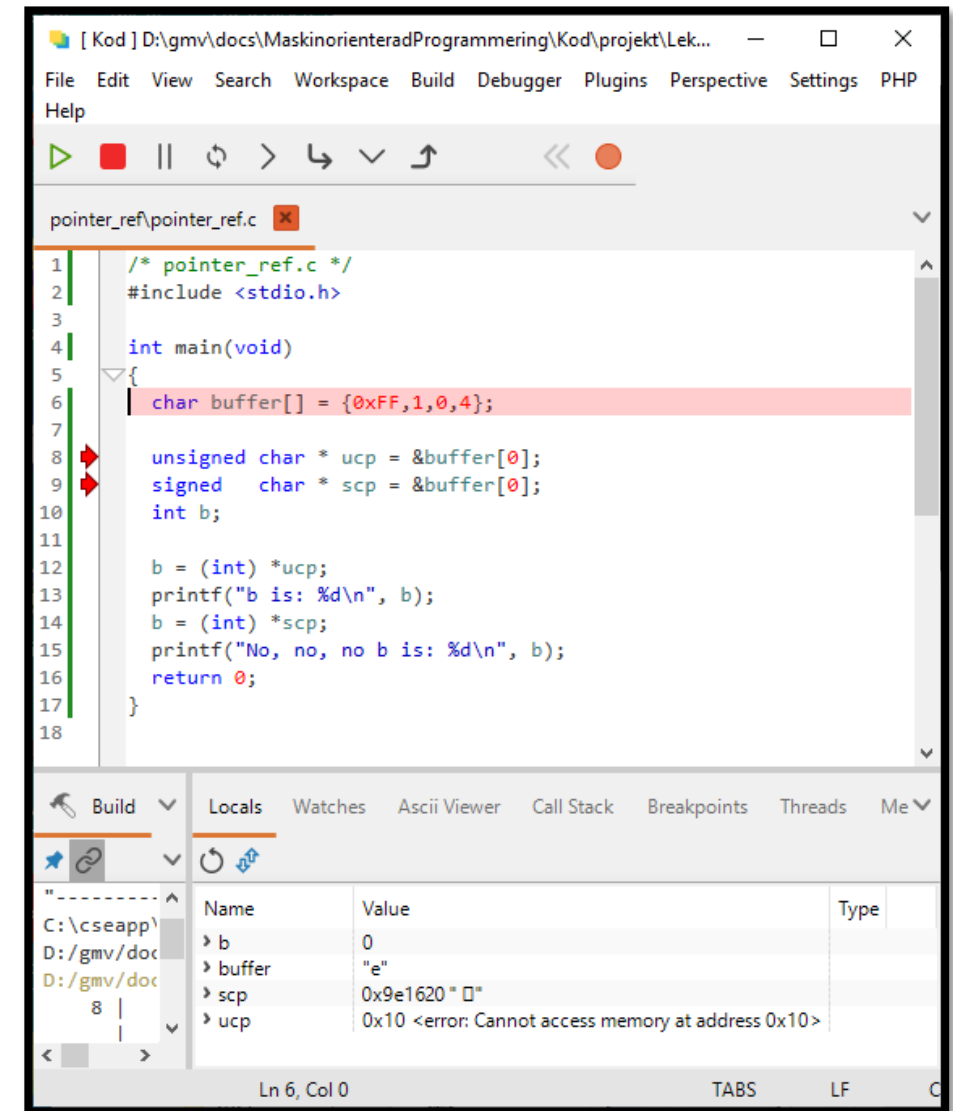
...

```
int b;
```

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

...

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



The screenshot shows a code editor window titled "[Kod] D:\gm\docs\MaskinorienteradProgrammering\Kod\projekt\Lek...". The code is in a file named "pointer_ref\pointer_ref.c". The code defines a character array "buffer" with values {0xFF, 1, 0, 4}. It then declares two pointers, "ucp" (unsigned char) and "scp" (signed char), both pointing to the first element of "buffer". It also declares an integer "b". The program prints the value of "b" using both pointers. The execution state is shown in the bottom panel, which includes a "Locals" tab. The "Locals" tab shows the following variables and their values:

Name	Value	Type
b	0	
buffer	"e"	
scp	0x9e1620 " "	
ucp	0x10 <error: Cannot access memory at address 0x10>	

Varför behöver vi pekare?

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

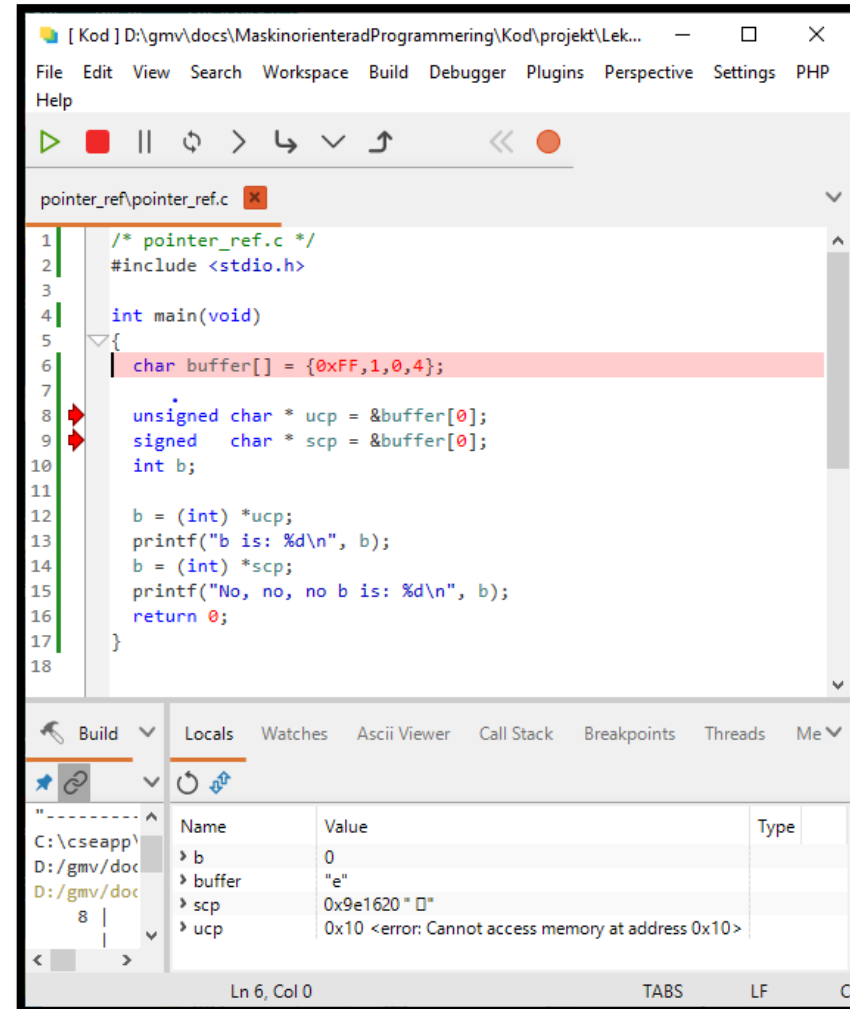
Exempel:

```
#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.



The screenshot shows a code editor window titled "pointer_ref\pointer_ref.c" with the following code:

```
1  /* pointer_ref.c */  
2  #include <stdio.h>  
3  
4  int main(void)  
5  {  
6      char buffer[] = {0xFF,1,0,4};  
7  
8      unsigned char * ucp = &buffer[0];  
9      signed char * scp = &buffer[0];  
10     int b;  
11  
12     b = (int) *ucp;  
13     printf("b is: %d\n", b);  
14     b = (int) *scp;  
15     printf("No, no, no b is: %d\n", b);  
16     return 0;  
17 }  
18
```

Below the code editor, the "Locals" pane shows the memory state:

Name	Value	Type
b	0	
buffer	"e"	
scp	0x9e1620 * □	
ucp	0x10 <error: Cannot access memory at address 0x10>	

The status bar at the bottom indicates "Ln 6, Col 0" and "TABS LF".

Möjligheter med pekare...

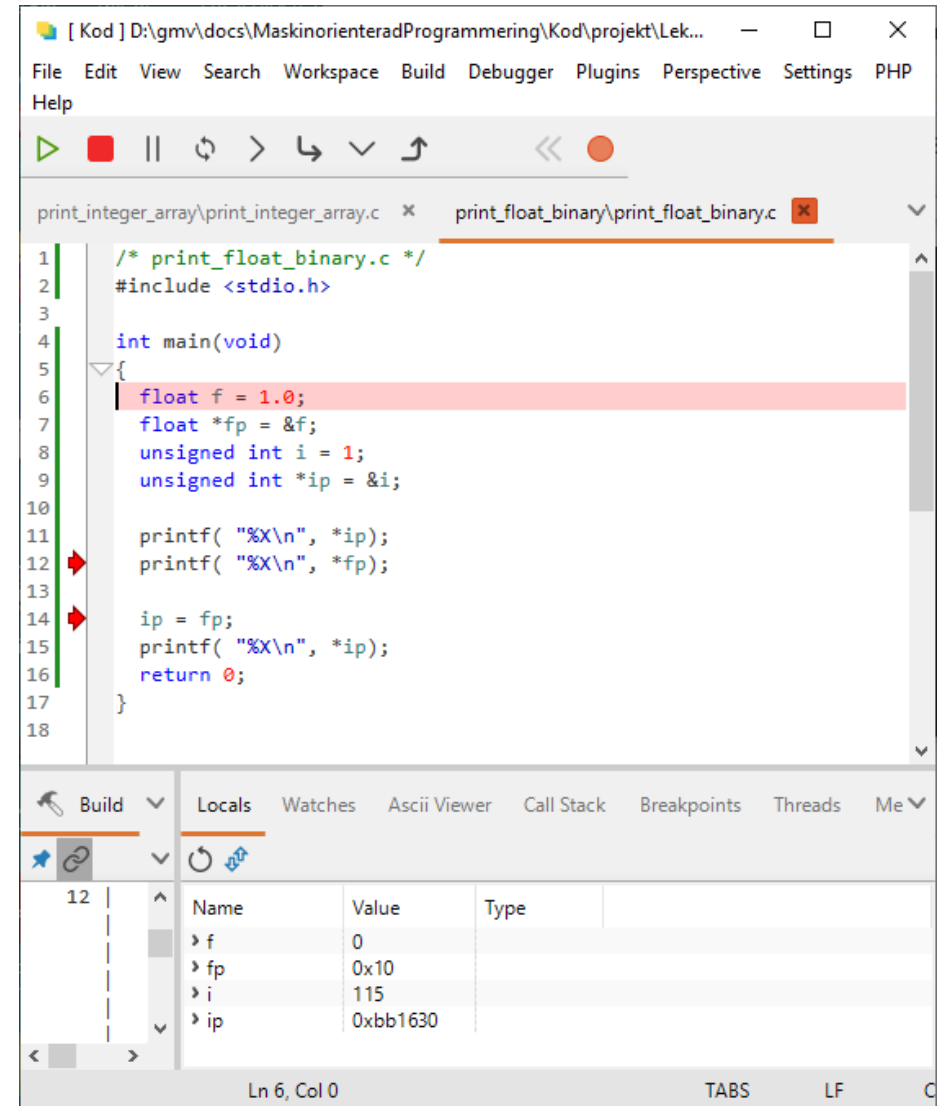
Exempel:

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 in a debugger. The code is as follows:

```
1  /* print_float_binary.c */
2  #include <stdio.h>
3
4  int main(void)
5  {
6      float f = 1.0;
7      float *fp = &f;
8      unsigned int i = 1;
9      unsigned int *ip = &i;
10
11     printf( "%X\n", *ip);
12     printf( "%X\n", *fp);
13
14     ip = fp;
15     printf( "%X\n", *ip);
16     return 0;
17 }
18
```

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

Name	Value	Type
f	0	
fp	0x10	
i	115	
ip	0xbb1630	

The status bar at the bottom indicates the current position is Ln 6, Col 0.

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;
}
```

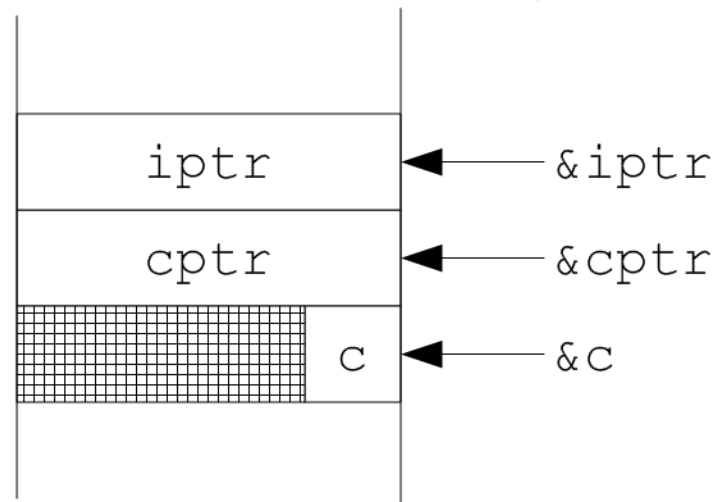


```
int strlen( char *s )
{
    int i = 0;
    while( *s )
    {
        s++; i++;
    }
    return i + 1;
}
```

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:

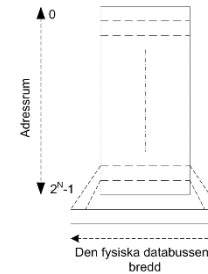
```
int    i;  
int    mi[12][8];
```

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



Adressrum

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

Periferihus Block 1 SRAM

E0	2001 FFFF
E0	
E0	
E0	
E0	
E0	2001 C400
E0	2001 C3FF
E0	
E0	2001 C000
E0	2001 BFFF
E0	
E0	
E0	
E0	

Monitor/
debugger
stack och data
Relokerade
avbrottsvektorer

Reserverat för
applikationen

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 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	BKPSRAM	AHB1
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	GPIOH	
4002 1800 - 4002 1BFF	GPIOG	
4002 1400 - 4002 17FF	GPIOF	
4002 1000 - 4002 13FF	GPIOE	
4002 0C00 - 4002 0FFF	GPIOD	
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	SP16	
4001 5000 - 4001 53FF	SP15	
4001 4800 - 4001 4BFF	TIM11	
4001 4400 - 4001 47FF	TIM10	
4001 4000 - 4001 43FF	TIM9	
4001 3C00 - 4001 3FFF	EXTI	APB2
4001 3800 - 4001 3BFF	SYSCFG	
4001 3400 - 4001 37FF	SPH	
4001 3000 - 4001 33FF	SP1	
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 5FFF	I2C3	
4000 5800 - 4000 5BFF	I2C2	
4000 5400 - 4000 57FF	I2C1	
4000 5000 - 4000 53FF	UART5	
4000 4C00 - 4000 4FFF	UART4	
4000 4800 - 4000 4BFF	USART3	
4000 4400 - 4000 47FF	USART2	
4000 4000 - 4000 43FF	I2Sext	
4000 3C00 - 4000 3FFF	SP13/I2S3	APB1
4000 3800 - 4000 3BFF	SP12/I2S2	
4000 3400 - 4000 37FF	I2Sext	
4000 3000 - 4000 33FF	IWDG	
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, hur de fyra minst signifikanta bitarna hos en läs- och skrivbar port på adress 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;
    }
}
```

