

Pekare

Ur innehållet

- Flyktiga registerinnehåll ("volatile")
- Pekararitmetik
- Stränghantering med pekare
- Pekare till funktioner
- Pekare till pekare.. (till pekare...)

Läsanvisningar:

- Arbetsbok kap 2, avsnitt "Pekararitmetik"

Målsättningar:

- Kunna använda pekare för absolutadressering
- Kunna använda pekare i stränghantering
- Kunna använda pekare som parametrar för returvärden

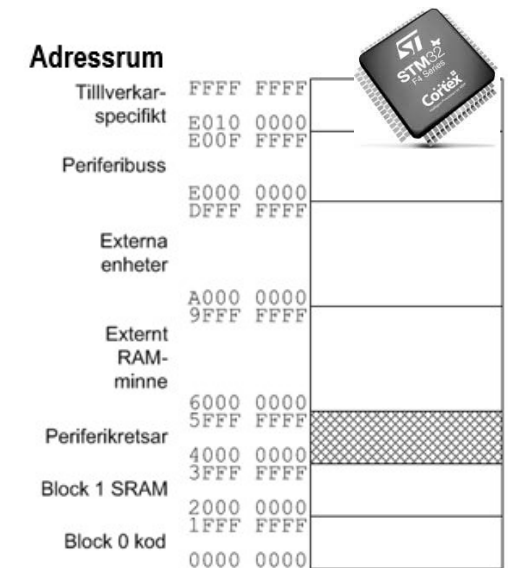
Absolut adressering

Exempel:

```
0x40021014          // en heltalskonstant (int)
// en pekare (unsigned char) som pekar på adress 0x40021014
(unsigned char *) 0x40021014
// innehåll, 8 bitar, på adress 0x40021014 (dereferens)
*((unsigned char*) 0x40021014)

// Läs från adress 0x40021014
unsigned char value = *((unsigned char *) 0x40021014);

// Skriv till adress 0x40021014
*((unsigned char *) 0x40021014) = value;
```



Men vi måste se upp om vi låter kompilatorn "optimera" koden...

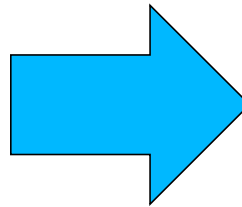
Avsikten att förbättra koden kan gå fel...

Exempel:

```
char * inport = (char*) 0x40021014;

void foo(){

    while(*inport != 0)
    {
        // ...
    }
}
```



"förbättrad kod"

```
char * inport = (char*) 0x40021014;

void foo(){
    char tmp = *inport;
    while( tmp != 0 )
    {
        // ...
    }
}
```

En optimerande kompilator kan "flytta ut" läsningen från iterationsslingan och testen reflekterar då inte längre portens värde.

"Volatile qualifier" - bestämning/tillägg till deklaration

volatile förhindrar viss optimering (som annars är bra och nödvändig) dvs. indikerar att kompilatorn måste förmoda att innehållet på en address kan ändras "från utsidan" (dvs. en ändring kan inte upptäckas vid statisk analys av koden).

```
volatile char * inport = (char*) 0x40021014;
void foo(){
    while(*inport != 0)
    {
        // ---
    }
}
```

```
volatile char * utport = (char*) 0x40021014;
void f2()
{
    *utport = 0;
    ...
    *utport = 1;
    ...
    *utport = 2;
}
```

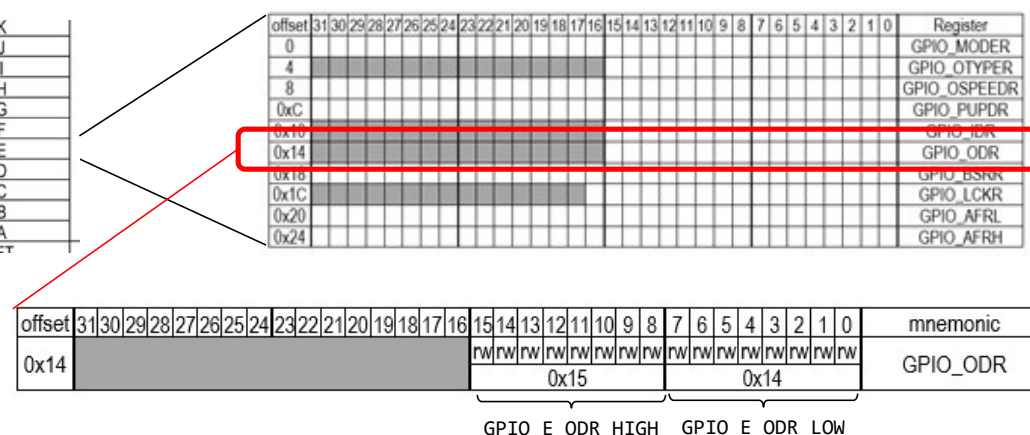
Portadressering

Periferikretsar
A000 0000 - A000 0FFF - FSMC control reg - AHB3

5006 0800 - 5006 08FF	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	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	BP2SRAM
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
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	SAT
4001 5400 - 4001 57FF	SPIS
4001 5000 - 4001 53FF	SPIS
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	SPH
4001 3000 - 4001 33FF	SPH
4001 2C00 - 4001 2FFF	SDIO
4001 2000 - 4001 23FF	ADC1, ADC2, ADC3
4001 1400 - 4001 17FF	USART6
4001 1000 - 4001 13FF	USART1
4001 0400 - 4001 07FF	TIM8
4001 0000 - 4001 03FF	TIM7
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	I2C4
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	I2S3ext
4000 3C00 - 4000 3FFF	SPD1, SPD3
4000 3800 - 4000 3BFF	SPD2, SPD4
4000 3400 - 4000 37FF	I2S2ext
4000 3000 - 4000 33FF	MDIO
4000 2C00 - 4000 2FFF	WWDG
4000 2800 - 4000 2BFF	RTC & BKUP 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

Exempel:
GPIO port E

```
#define GPIO_E           0x40021000
#define GPIO_E_MODER    ((volatile unsigned int *)  GPIO_E)
#define GPIO_E_OTYPER   ((volatile unsigned short *) (GPIO_E+4))
#define GPIO_E_OSPEEDR  ((volatile unsigned int *)  (GPIO_E+8))
...
```



```
...
#define GPIO_E_ODR          ((volatile unsigned short *) (GPIO_E+0x14))
#define GPIO_E_ODR_LOW     ((volatile unsigned char *)  (GPIO_E+0x14))
#define GPIO_E_ODR_HIGH    ((volatile unsigned char *)  (GPIO_E+0x15))
...
```

```
value = *GPIO_E_ODR_LOW; // Läs från 0x40021014
*GPIO_E_ODR_HIGH = value; // Skriv till 0x40021015
```

Pekararitmetik

Följande operationer är tillåtna på pekare:

- Adressoperator
- Dereferens
- Addition av heltalskonstant
- Subtraktion av heltalskonstant
- Subtraktion av pekare med samma typ

Exempel 1:

```
char c, *cp1, *cp2;  
cp1 = &c;      // Ok  
cp2 = &cp1;    // Warning
```

Exempel 2:

```
char *cp1;  
cp1 = cp1 + 1;  
++cp1;  
cp1++;
```

Exempel 3:

```
char *cp1;  
cp1 = cp1 - 1;  
--cp1;  
cp1--;
```

Alla andra operationer, som exempelvis skift, negation, bitvis komplement, multiplikation eller division etc. är meningslösa och därför förbjudna.

Exempel 4:

```
char *cp1, *cp2;  
int *ip;  
int i;  
i = cp1 - cp2; // Ok  
i = ip - cp1;  // Error
```

Pekararitmetik

Vid addition (ptr+n) eller subtraktion (ptr-n), pekartypen avgör hur många bytes som adderas/subtraheras

Exempel 1:

```
char *cp = (char *) 0x20001000;  
(cp+1);    // Uttryckets värde: 0x20001001  
cp++;      // cp är därefter 0x20001001
```

Exempel 2:

```
int *ip = (int *) 0x20001000;  
(ip+1);    // Uttryckets värde: 0x20001004  
ip++;      // ip är därefter 0x20001004
```

$\text{ptr} + n \Rightarrow \text{ptr} + n * \text{sizeof}(\text{ptr type})$

$\text{ptr} - n \Rightarrow \text{ptr} - n * \text{sizeof}(\text{ptr type})$

```
char name[] = "Machine Oriented Programming";
char *p2c;
int *p2i;

p2c = name;
p2i = (int*)name;
printf( "%s - %s\n", (char*)p2c, (char*)p2i);

p2c += 3;
p2i += 3;
printf( "%s - %s\n", (char*)p2c, (char*)p2i);
```

Vad blir utskrifterna?

The screenshot displays the Visual Studio Code interface with a C program in the editor. The program is as follows:

```

14
15
16 int main(void)
17 {
18     char name[] = "Machine Oriented Programming";
19     char *p2c;
20     int *p2i;
21
22     p2c = name;
23     p2i = (int*)name;
24     printf( "%s - %s\n", (char*)p2c, (char*)p2i);
25
26     p2c += 3;
27     p2i += 3;
28     printf( "%s - %s\n", (char*)p2c, (char*)p2i);
29     return 0;
30 }

```

The 'Locals' pane at the bottom shows the current state of the variables:

Name	Value	Type
> name	{...}	char[29]
> p2c	0x0	char*
> p2i	0x1a1650	int*

The status bar at the bottom indicates the current position is Ln 18, Col 47, and the file encoding is SPACES, LF, C++.

`x[y]` översätts till `*(x+y)` så detta är också en pekare.
Indexering är samma sak för pekare och fält...

Så, är fält då pekare?... Nej...



Fält eller pekare, i korthet

Exempel:

```
char* s1 = "Emilia";
char s2[] = "Emilia";
```

ARM/Thumb assembler:

```
2000001c <s1>:
2000001c: 20000028
```

```
20000020 <s2>:
20000020: 6c696d45 "Emil"
20000024: 00006169 "ia\0"
20000028: 6c696d45 "Emil"
2000002c: 00006169 "ia\0"
```

	s2	s1
Type:	Fält	Pekarvariabel
Adressering:	&s2 är inte möjligt - s2 är bara en symbol s2 = symbol = fältets startadress s2 = &(s2[0]) s2[0] ≡ *s2 → 'E'	&s1 = adress till variabeln s1 s1 = s1's värde = textsträngens startadress. s1 = &(s1[0]) s1[0] ≡ *s1 → 'E'
Pekararitmetik:	s2++ är inte möjligt (s2+1)[0] är OK	s1++ är OK (s1+1)[0] är OK
Typstorlek:	sizeof(s2) = 7 bytes	sizeof(s1) = sizeof(char*) = 4 bytes

s2 är en symbol (ingen variabel) för en adress känd vid kompileringstillfället.

Eftersom s2 är en adress, kan vi dereferera den på samma sätt som en pekare: *s2 → 'E'.

Stränghantering

ytterligare en variant av strlen...

```
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;
}
```

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

*s++: char tmp=*s; s=s+1;

(*s)++: tmp=*s; tmp=tmp+1;

Kopiering av textsträng - strcpy

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...

Kopiering av textsträng - strcpy

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...

```
strcpy:
@ Registeranvändning:
@ R0= &dest
@ R1= &src
@ R2= temporärregister
@ R3= lokal variabel rval
    MOV     R3,R0      @ rval = dest
.L0:
    LDRB    R2,[R1]     @ R2 = *src
    STRB    R2,[R0]     @ *dest = R2
    ADD     R1,R1,#1    @ src++
    ADD     R0,R0,#1    @ dest++
    CMP     R2,#0       @ *src != 0
    BNE     .L0
    MOV     R0,R3       @ return (dest)
    BX      LR
```

Fler skäl att använda pekare

Argument är värdeanrop, "pass-by value" i C.

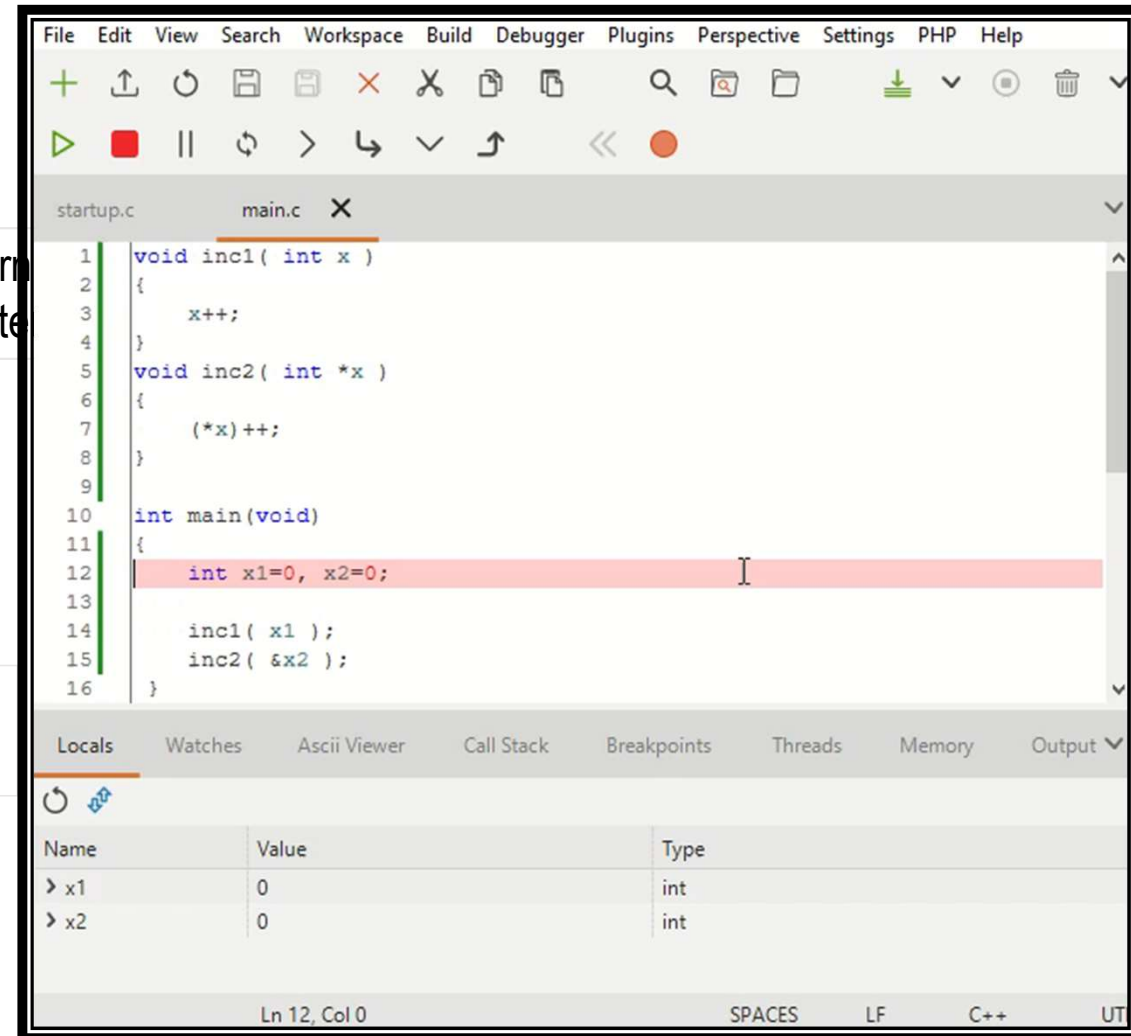
```
void inc1( int x )  
{  
    x++;  
}
```

inc1 använder kopia av parameter
inc2 ändrar innehållet via parameter

```
void inc2( int *x )  
{  
    (*x)++;  
}
```

```
int main(void)  
{  
    int x1=0, x2=0;  
  
    inc1( x1 );  
    inc2( &x2 );  
}
```

x1 har fortfarande värdet 0, men
x2 har värdet 1 efter anropen...



```
File Edit View Search Workspace Build Debugger Plugins Perspective Settings PHP Help  
+ ↕ ↺ ⌨ ✖ ✂ 📄 📄 🔍 📁 📄 ⬇️ ⌂ 🗑️  
▶ ⏸ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ ⏰ ⏱ ⏲ ⏳ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ ⏰ ⏱ ⏲ ⏳  
startup.c main.c X  
1 void inc1( int x )  
2 {  
3     x++;  
4 }  
5 void inc2( int *x )  
6 {  
7     (*x)++;  
8 }  
9  
10 int main(void)  
11 {  
12     int x1=0, x2=0;  
13  
14     inc1( x1 );  
15     inc2( &x2 );  
16 }  
Locals Watches Ascii Viewer Call Stack Breakpoints Threads Memory Output  
▶ x1 0 int  
▶ x2 0 int  
Ln 12, Col 0 SPACES LF C++ UT
```

Pekare till funktioner - "funktionspekare"

En funktionspekare definieras av:

- Returvärdet
- Argument och deras respektive typer

Funktionspekarens värde är en adress.

Exempel:

```
int foo0( void );      // Funktionsprototyp
int foo1( int x );     // Funktionsprototyp
int (*funp) (int);     // Deklaration av variabel 'funp'

funp = foo0;           // Warning, typfel
funp = foo1;           // Ok
```

Exempel:

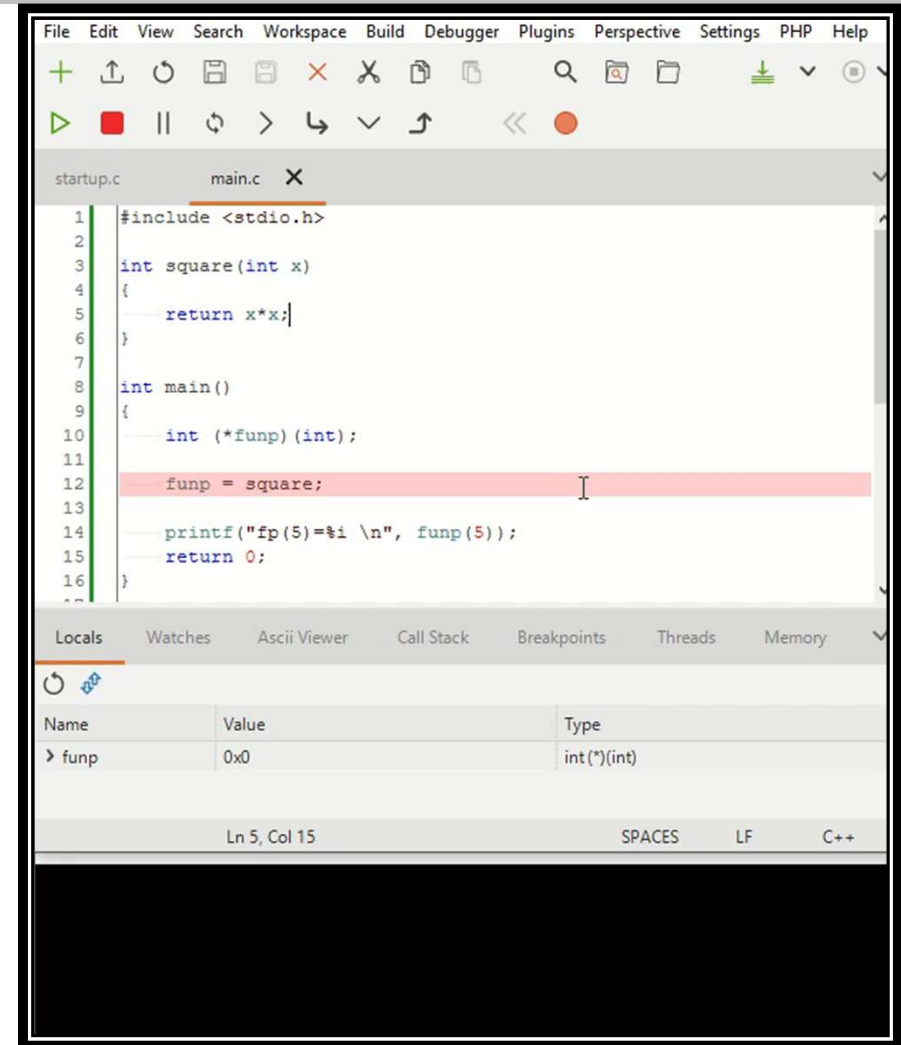
```
int square(int x)
{
    return x*x;
}
```

```
int main()
{
    int (*funp)(int);

    funp = square;

    printf("fp(5)=%i \n", funp(5));
    return 0;
}
```

Funktionspekare



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 - på plats i minnet

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 C-funktioner på fast adress i minnet

Exempel:

En parameterlös subrutin `portinit` för att initiera hårdvaran i MD407 finns med ingång på adress adress `0x08000200` i minnet. Visa hur denna kan anropas:

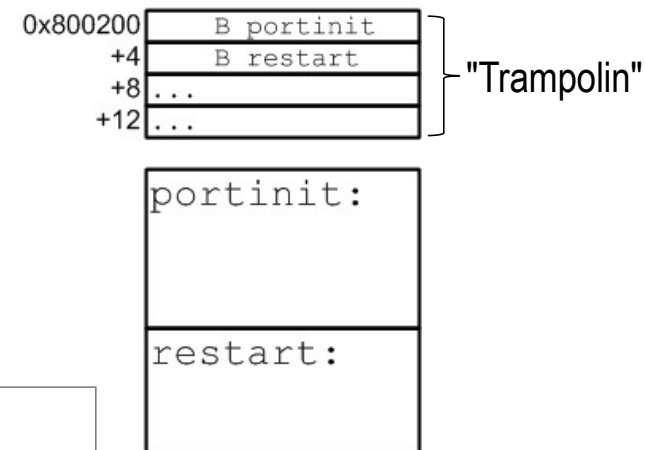
- a) i assemblerspråk
- b) från ett C-program

a)

```
LDR    R0,=0x8000200
BLX    R0
```

b)

```
#define portinit ((void (**)(void) ) 0x8000200 )
void xxx( void )
{
    portinit();
}
```



Fält som parameter till funktion

- Ett fält i en parameterlista är en pekare

```
void foo(int i[]);
```

[] – notationen finns men betyder pekare, dvs. detta är samma sak som:

```
void foo(int* i);
```

Undviker kopiering av hela fält inför funktionsanropet. Längden av fältet behöver inte vara känd.

Dock: fältet kan ändras i den anropade funktionen

```
int sumElements(int *a, int l)
{
    int sum = 0;
    for (int i=0; i<l; i++) {
        sum += a[i];
    }
    return sum;
}

...
int array[] = {5,4,3,2,1};
int x;
x = sumElements(array, 5);
```

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

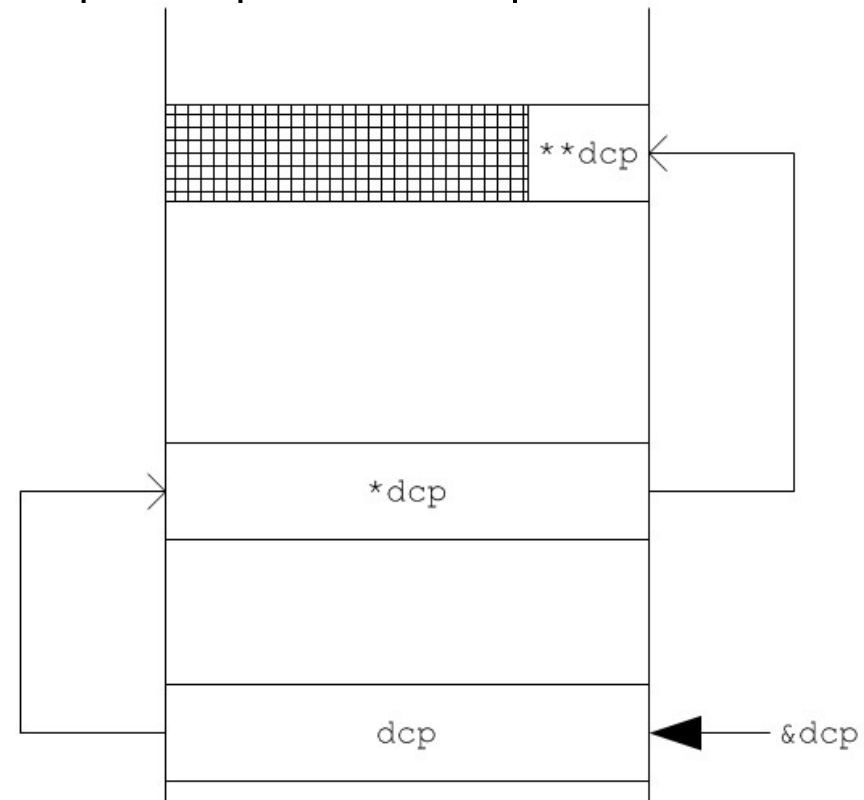
LDR R3, =c

STRB R2, [R3]

Deklarationen :

char **dcp;

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



Exempel: Ett fält med pekare till textsträngar

Byt plats mellan två pekarvariabler

Ett fält med pekare

Exempel: Innehåll och adress

```
arrayOfArrays[i][j] = arrayOfArrays+i*4+j
```

Pekare, klipp 5, 2021:1