Pekare

Ur innehållet

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

Målsättningar:

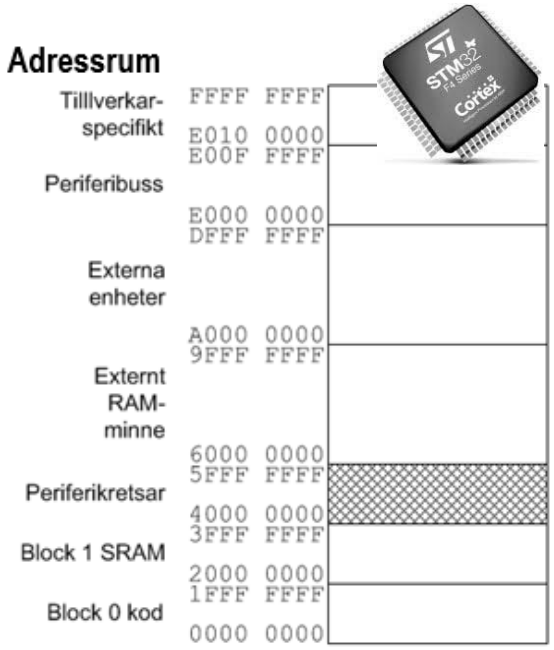
- Kunna använda pekare för absolutadressering
- Kunna använda pekare i stränghantering
- Kunna använda pekare som parametrar för returvärden

Absolut adressering

Exempel:

```
0x40021014       // en heltalskonstant (int)
// en pekare (unsigned char) som pekar på adress 0x40021414
(unsigned char *) 0x40021014
// innehåll, 8 bitar, på address 0x40021014 (dereferens)
*((unsigned char*) 0x40021014)
```

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

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

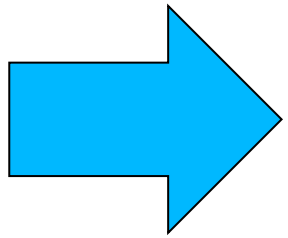


Men vi måste se upp om vi låter kompilatorn "optimera" koden...

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.

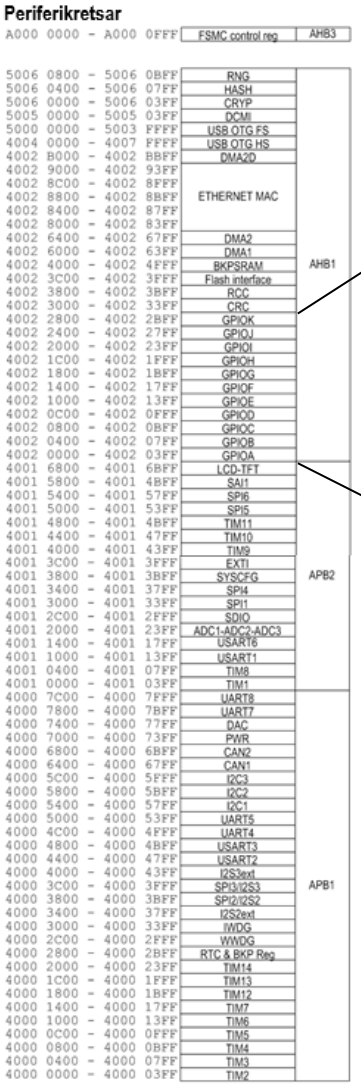
"Volatile qualifier" - bestämning/tillägg till deklaration

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)
    {
        // ---
    }
}
```

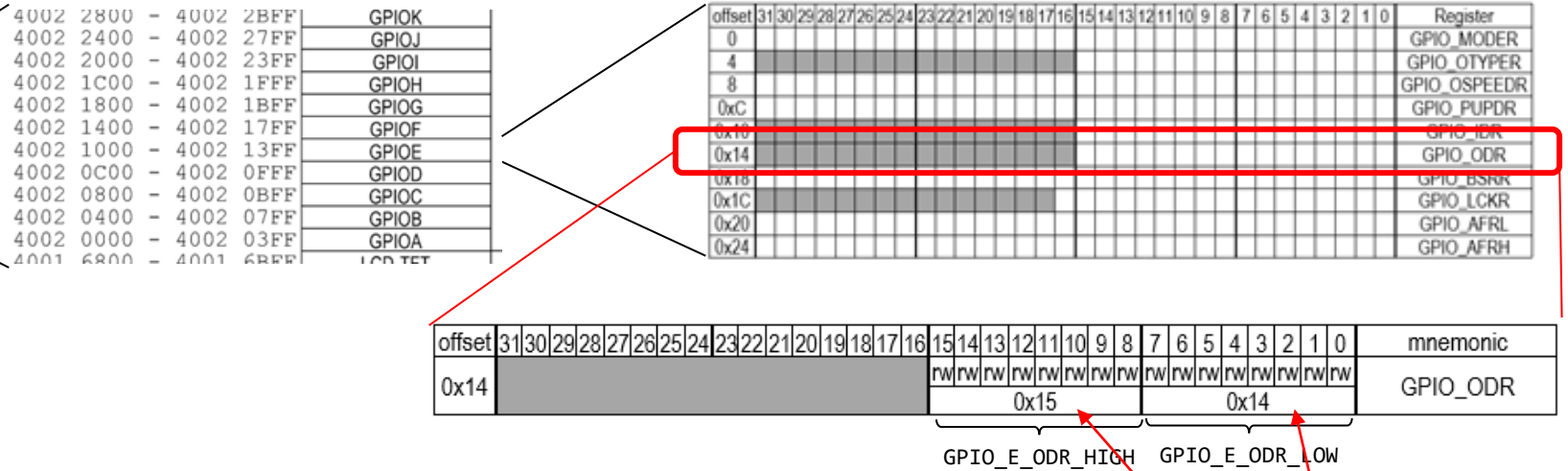
```
volatile char * utport = (char*) 0x40021014;
void f2()
{
    *utport = 0;
    …
    *utport = 1;
    …
    *utport = 2;
}
```

Portadressering



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

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

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--;
```

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:

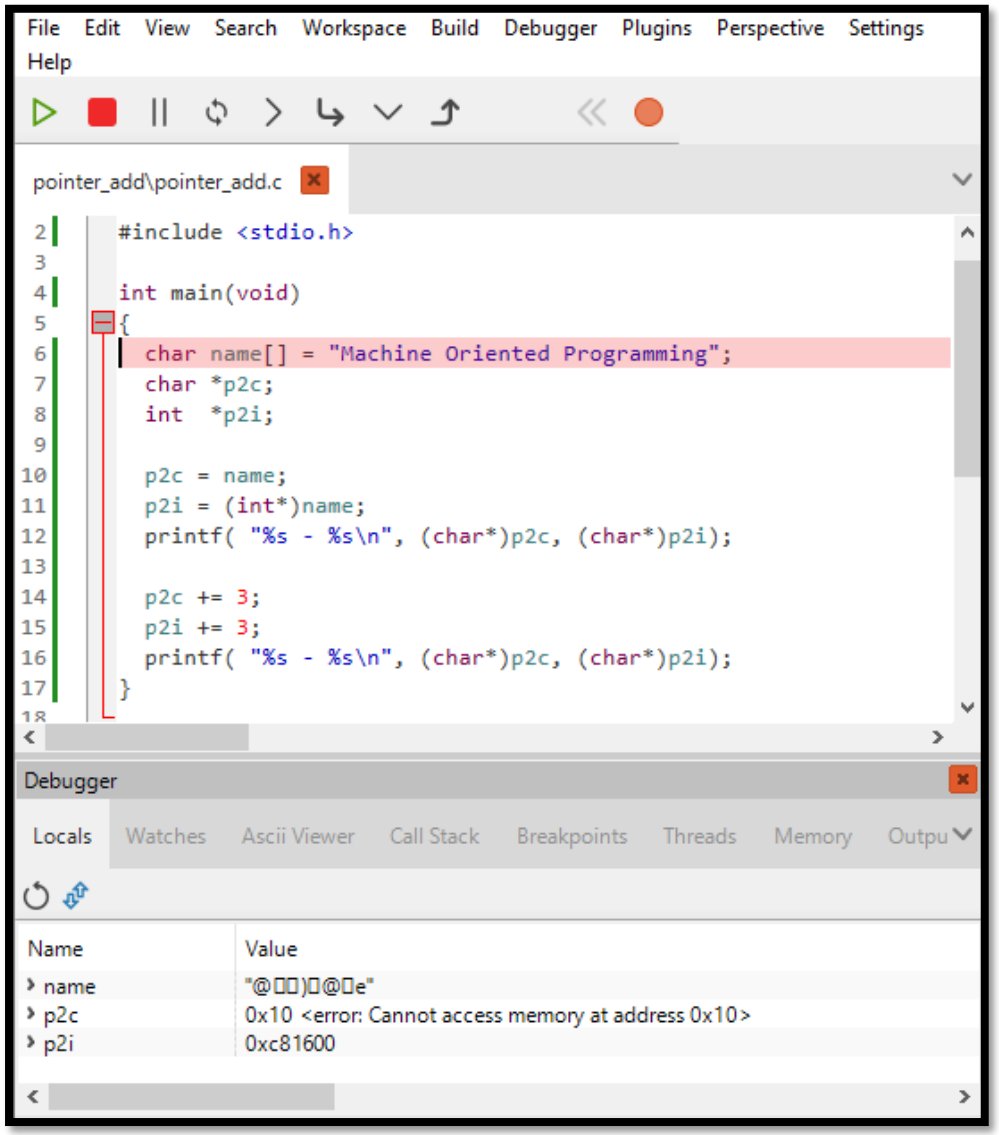
```
int *ip = (int *) 0x20001000;
 (ip+1);   // Uttryckets värde: 0x20001004
 ip++;     // ip är därefter 0x20001004
```

ptr + n ⇒ ptr + n * sizeof(ptr type)
ptr - n ⇒ ptr - n * sizeof(ptr type)

Tänkvärt pekarexempel

```
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?

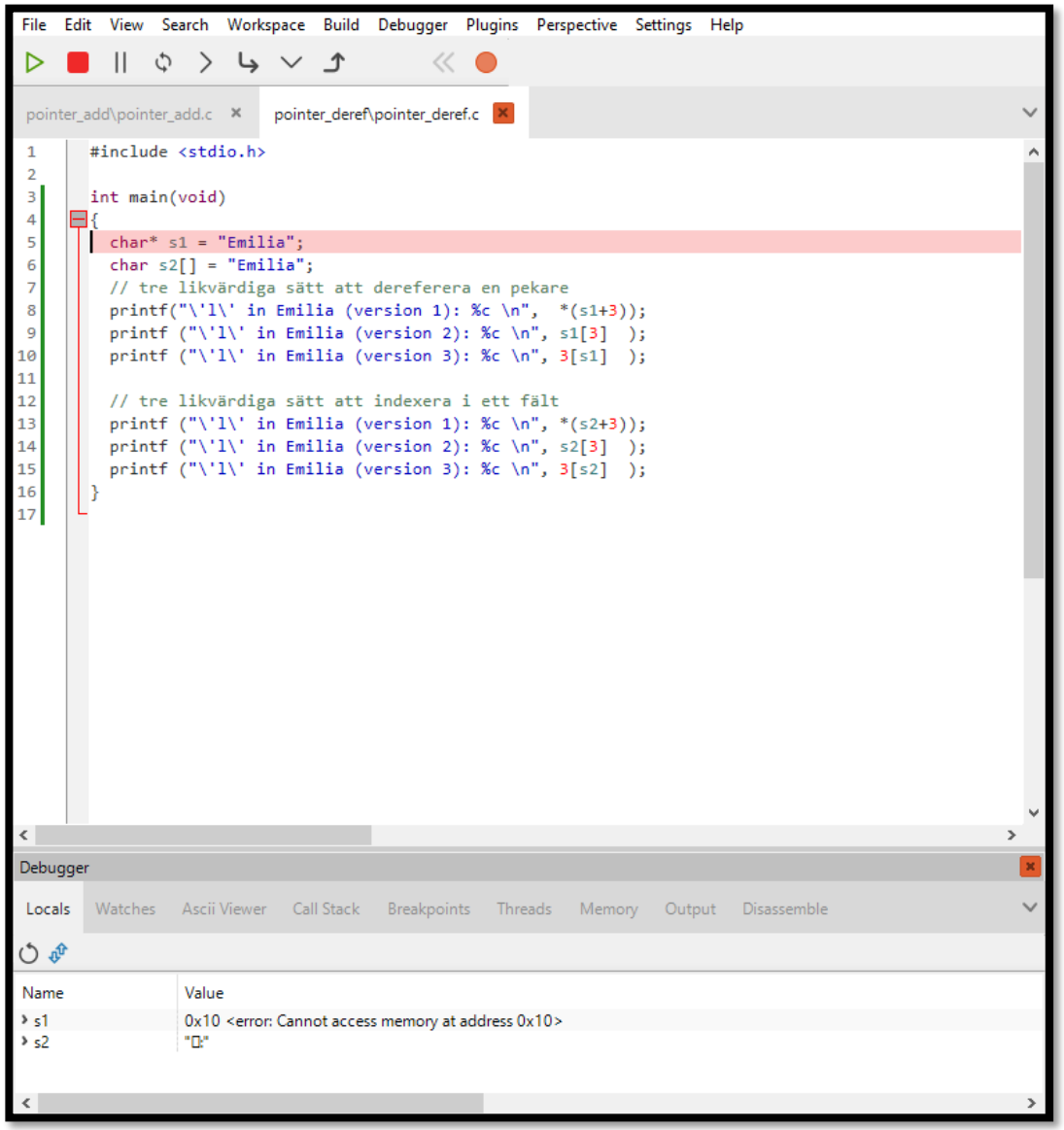


Fält eller pekare?

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

```c
#include <stdio.h>
int main(void)
{
  char* s1 = "Emilia";
  char s2[] = "Emilia";
  // tre likvärdiga sätt att dereferera en pekare
  printf("\'l\' in Emilia (version 1): %c \n",  *(s1+3));
  printf ("\'l\' in Emilia (version 2): %c \n", s1[3]  );
  printf ("\'l\' in Emilia (version 3): %c \n", 3[s1]  );

  // tre likvärdiga sätt att indexera i ett fält
  printf ("\'l\' in Emilia (version 1): %c \n", *(s2+3));
  printf ("\'l\' in Emilia (version 2): %c \n", s2[3]  );
  printf ("\'l\' in Emilia (version 3): %c \n", 3[s2]  );
}
```

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...

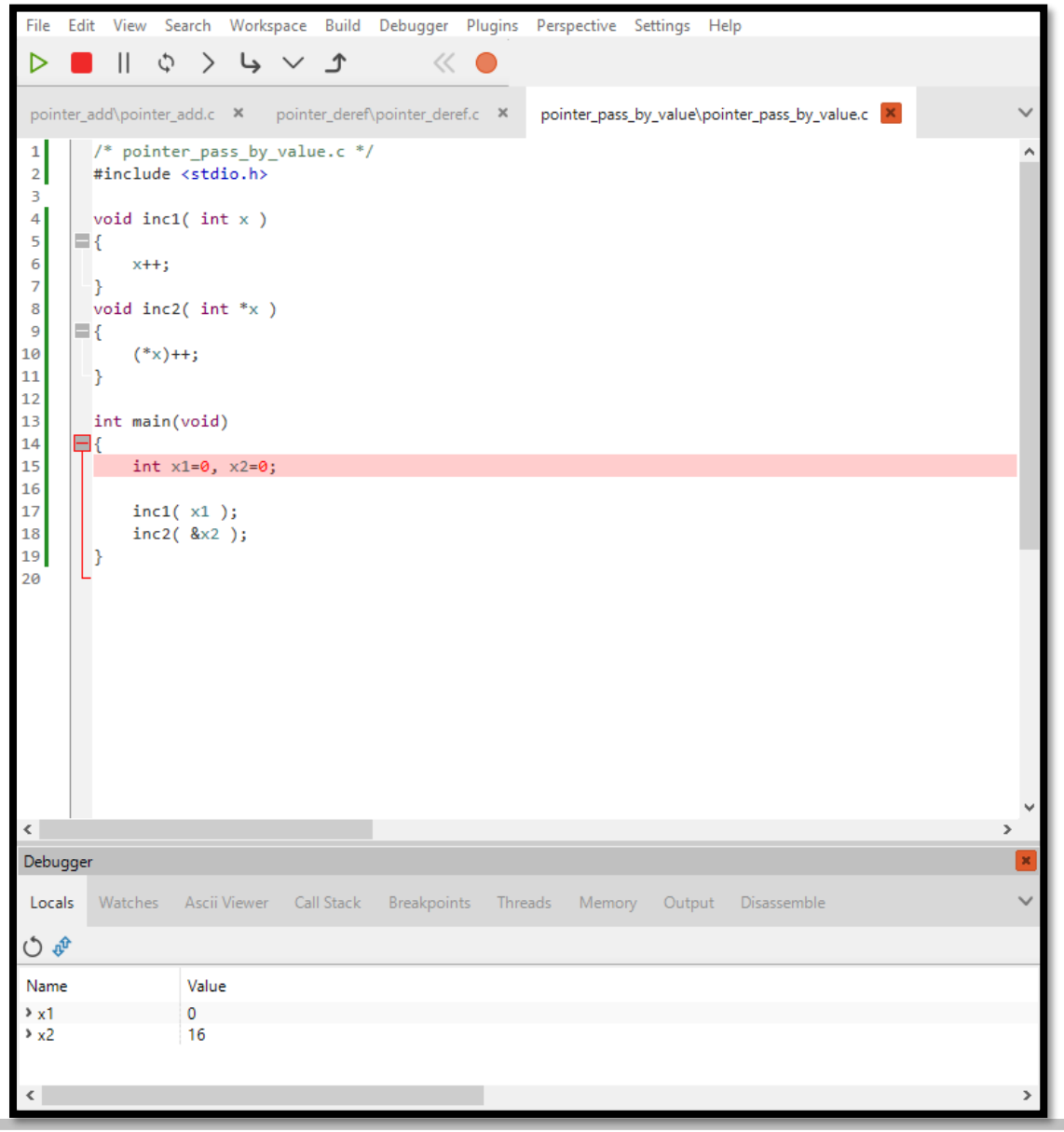
Fler skäl att använda pekare

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

```
void inc1( int x )
{
    x++;
}
void inc2( int *x )
{
    (*x)++;
}

int main(void)
{
    int x1=0, x2=0;

    inc1( x1 );
    inc2( &x2 );
 }
```

`inc1` använder kopia av parametern
`inc2` ändrar innehållet via parametern…

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



Pekare till funktioner - "funktionspekare"

En funktionspekare definieras av:

- Returvärdet
- Argument och deras respektive typer

Funktionspekarens värde är en adress.

Exempel:

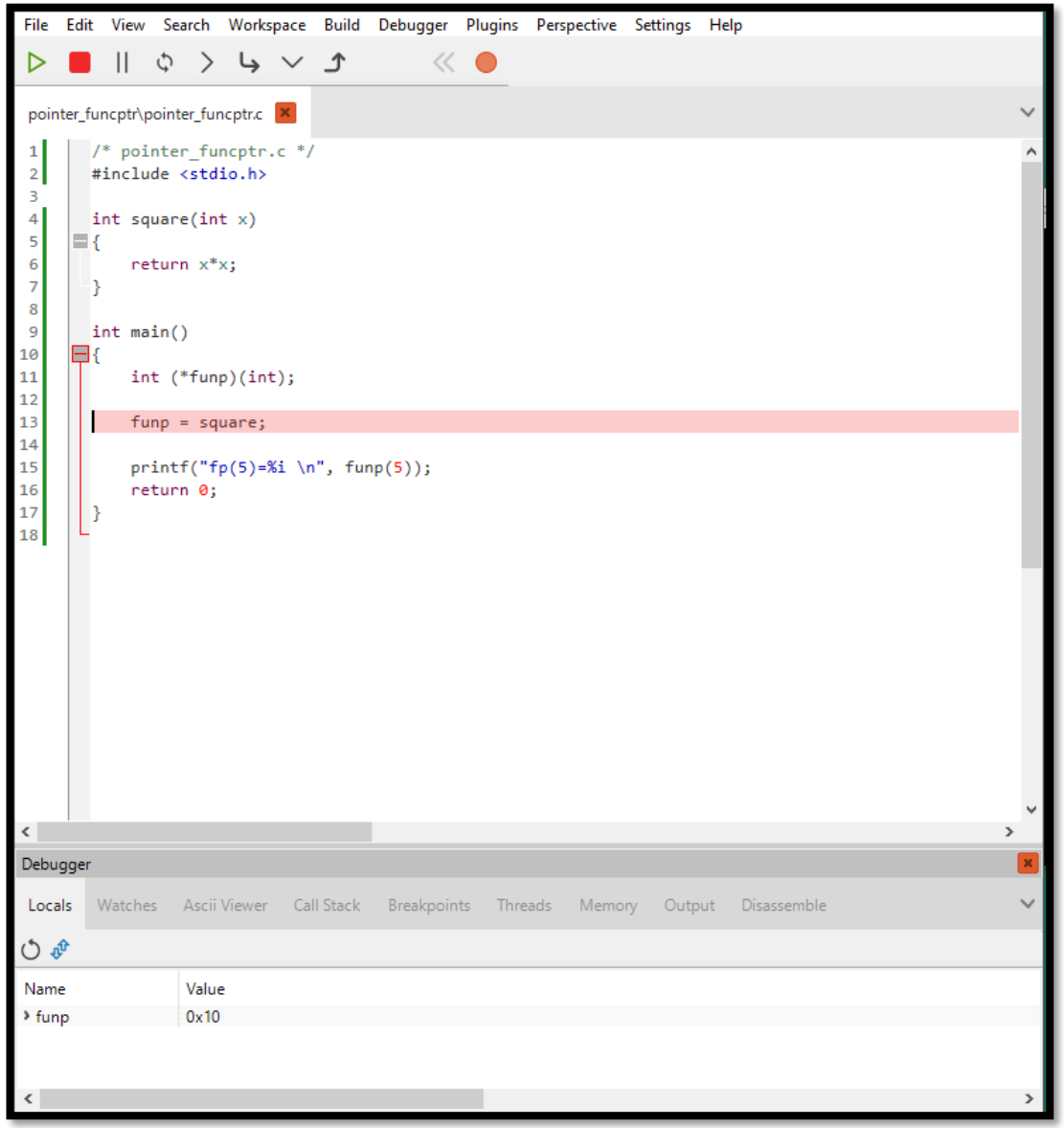
```
int foo0( void );      // Funktionsprototyp
int foo1( int x );     // Funktionsprototyp
int (*funp) (int);     // Deklaration av variabel 'funp'

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

Pekare till funktioner

Exempel:

```
#include <stdio.h>

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:

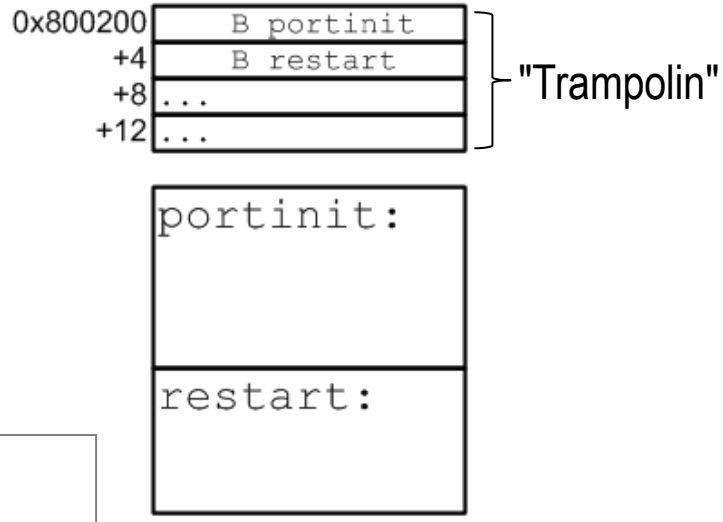
```
LDR     R0,=do_this
LDR     R1,=0x2001C000
STR     R0,[R1]
```

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]
```

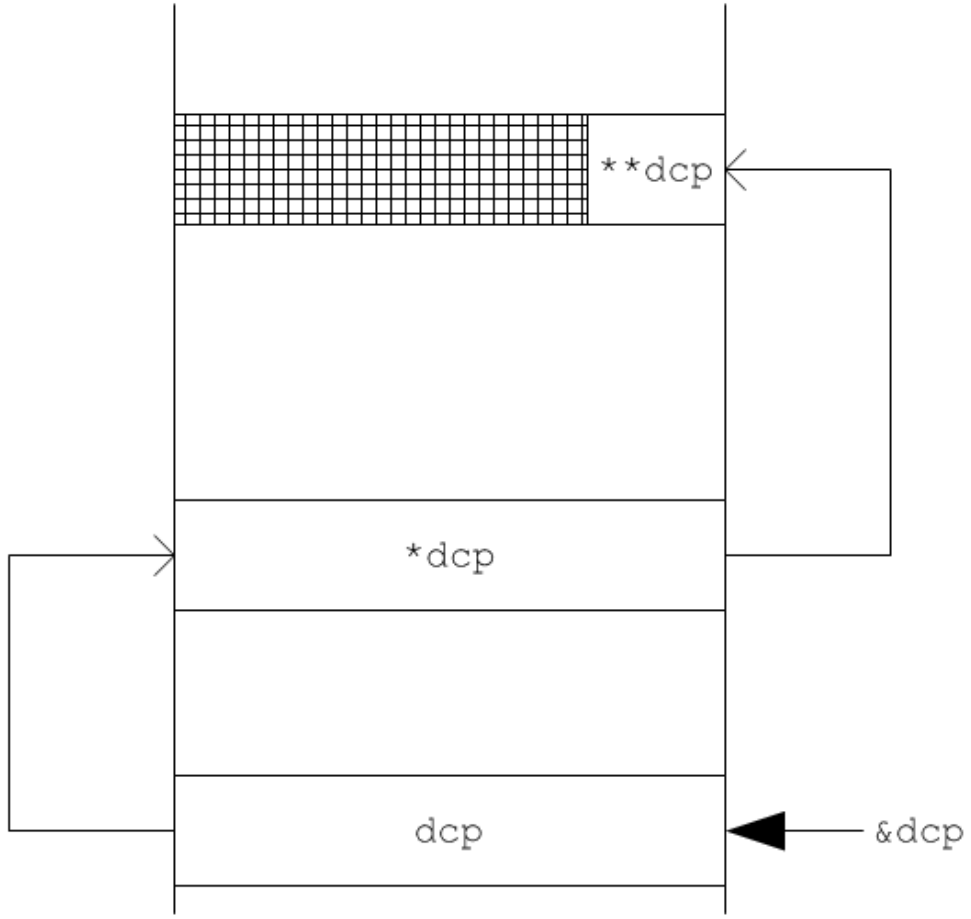
`c = **dcp;` kodas

```
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”.



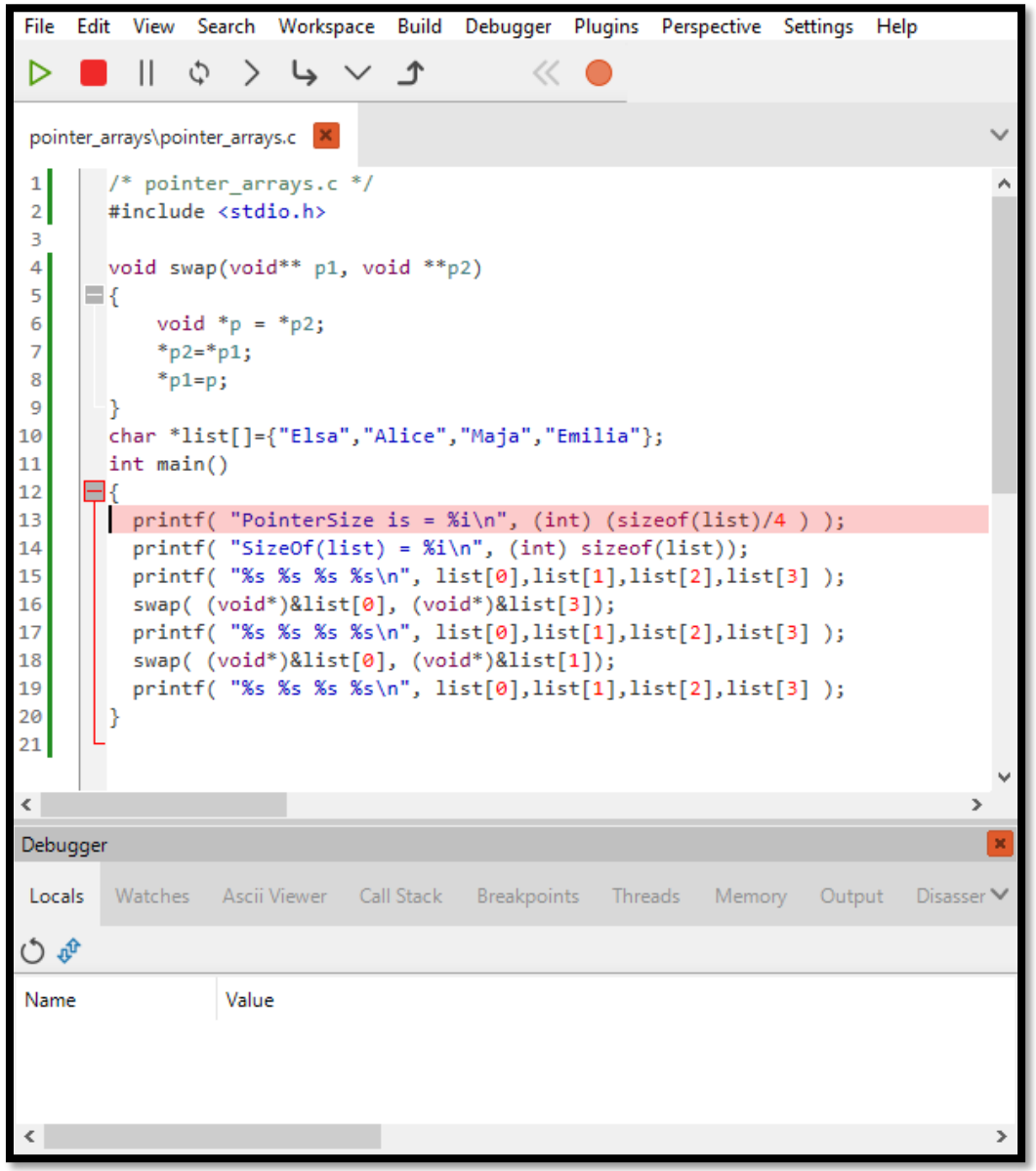
Pekarfält

Exempel: Ett fält med pekare till textsträngar

```
#include <stdio.h>
void swap(void** p1, void **p2)
{
    void *p = *p2;
    *p2=*p1;
    *p1=p;
}
char *list[]={"Elsa","Alice","Maja","Emilia"};
int main()
{
  printf( "PointerSize is = %i\n", (int) (sizeof(list)/4 ) );
  printf( "SizeOf(list) = %i\n", (int) sizeof(list));
  printf( "%s %s %s %s\n", list[0],list[1],list[2],list[3] );
  swap( (void*)&list[0], (void*)&list[3]);
  printf( "%s %s %s %s\n", list[0],list[1],list[2],list[3] );
  swap( (void*)&list[0], (void*)&list[1]);
  printf( "%s %s %s %s\n", list[0],list[1],list[2],list[3] );
}
```

Byt plats mellan två pekarvariabler

Ett fält med pekare



Flerdimensionella fält

Exempel: Innehåll och adress

```
#include <stdio.h>
int arrayOfArrays[3][4] =
{ {1,2,3,4},
  {5,6,7,8},
  {9,10,11,12} };
int main()
{
  int i,j;
  for( i=0; i<3; i++) {
    for ( j=0; j<4; j++)
      printf("%d ", arrayOfArrays[i][j]);
    printf("\n");
  }
  for( i=0; i<3; i++) {
    for ( j=0; j<4; j++)
      printf("%p ", &arrayOfArrays[i][j]);
    printf("\n");
  }
  return 0;
```

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