Grundläggande C-programmering – del 2
Pekare och Arrayer

Ulf Assarsson

Läromoment:

- Pekare
- Absolutadressering (portar):
  - typedef, volatile, #define
- Arrayer av pekare, arrayer av arrayer

Hemuppgifter: v2.

Föregående lektion

- C-syntax
- Programstruktur, kompilering, länkning
- Bitoperationer − se övningsuppgifterna

Pekare

- A pointer is essentially a simple integer variable that holds a **memory address** of a value (variable or port), instead of holding the actual value itself.

Varför pekare?

- Kunna referera till ett objekt eller variabel, dvs utan att behöva skapa en kopia

```
char namn1[] = "Elsa";
char namn2[] = "Alice";
char namn3[] = "Maja";

char *längstNamn = namn2;
```

Man kan säga att längstNamn refererar till namn2, men vi kallar det inte referens utan pekare.
Dvs längstNamn pekar på namn2.

Varför pekare?

- Skriva till /Läsa från portar
- Indexera fortare i arrayer
- Slippa kopiera inputparametrar
- Ändra inputparametrarna…

```
#include <stdlib.h>

void inc(int x, char y)
{
    x++;
    y++;
}
```

Argumenten är ’’pass-by value’’ i C.

```
int var1 = 2;
char var2 = 7;
inc(var1, var2);
```

var1 och var 2 har fortfarande värdena 2 resp 7 efter funktionsanropet

```
#include <stdlib.h>

void inc(int *x, char *y)
{
    (*x)++;
    (*y)++;
}
```

Argumenten är ’’pass-by value’’ i C.

```
int var1 = 2;
char var2 = 7;
inc(&var1, &var2);
```

var1 och var 2 har nu värdena 3 resp 8 efter funktionsanropet

Pekare

- Pekarens värde är en adress.
- Pekarens typ berättar hur man tolkar bitarna som finns på adressen.

```
// array av chars
char str[] = ”abc";

// p pekare till char
char* p = str; // == &str[0] == &(str[0])

*p; // värdet p pekar på är 'a'

*p = 'b';
```

```
MOV R0, #'b’
LDR  R1, p
STRB R0, [R1] - skriv till *p
```

Vi hämtar värdet som ligger på adressen som ligger i p.

```
LDR  R0, =str
LDR  R1, =p
STR  r0, [R1]

LDR  R0, p
LDRB R0, [R0] - läs *p
```

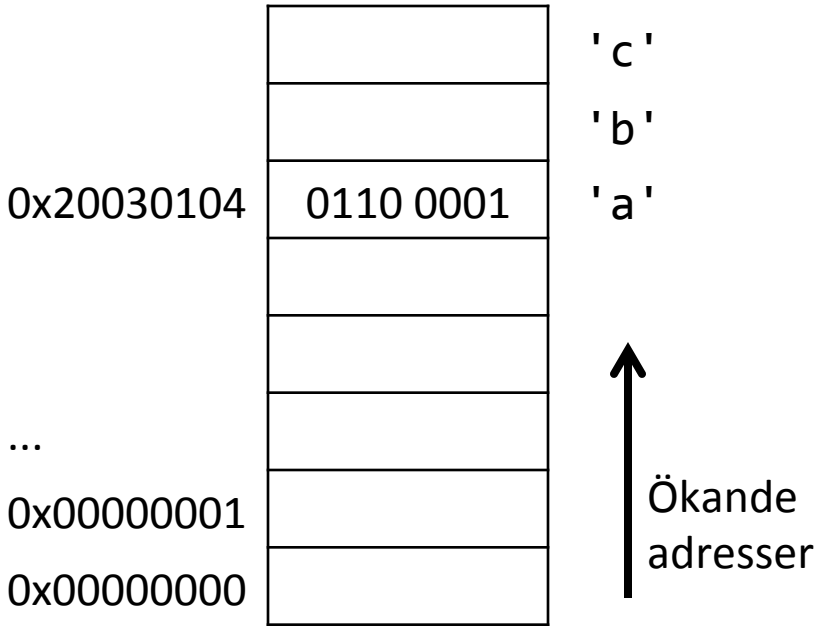
Pekare

- Pekarens värde är en adress.
- Pekarens typ berättar hur man tolkar bitarna som finns på adressen.

```
char* p = str; // == 0x20030104
```

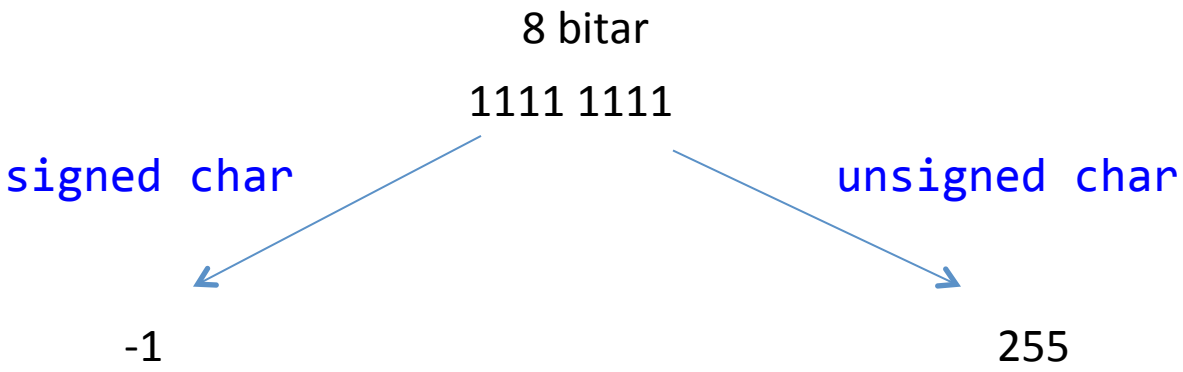
typ

p's värde är en adress



Dereferera

- När vi derefererar en pekare så hämtar vi objektet som ligger på adressen.
  - Antalet bytes vi läser beror på typen
  - Tolkningen av bitarna beror på typen



char är ett 8-bits värde, dvs ett 8-bits tal. Ett tal kan motsvara en bokstav via ASCII-tabellen.

Operatorer på pekare (& *)

```c
#include <stdio.h>

int main()
{
    char a, b, *p;
    a = 'v';

    b = a;
    p = &a;

    printf("b = %c, p = 0x%p (%c) \n", b, p, *p);

    a = 'k';
    printf("b = %c, p = 0x%p (%c) \n", b, p, *p);
}
```

Deklaration av pekare

Adressen av ...

Dereferering

Utskift:

```
b = v, p = 0x0027F7C3 (v)
b = v, p = 0x0027F7C3 (k)
```

Asterisken (*) betyder

- I deklarationer
  - Pekartyp

```
char *p;
char* p;

void foo(int *pi);
```

- Som operator
  - Dereferens (”av-referera”)

```
char a = *p;
*p = 'b';
```

Luring

Antag *a = 2

Vad ger:

```
int x=1.0/*a;    ?
```

Pekare - översikt

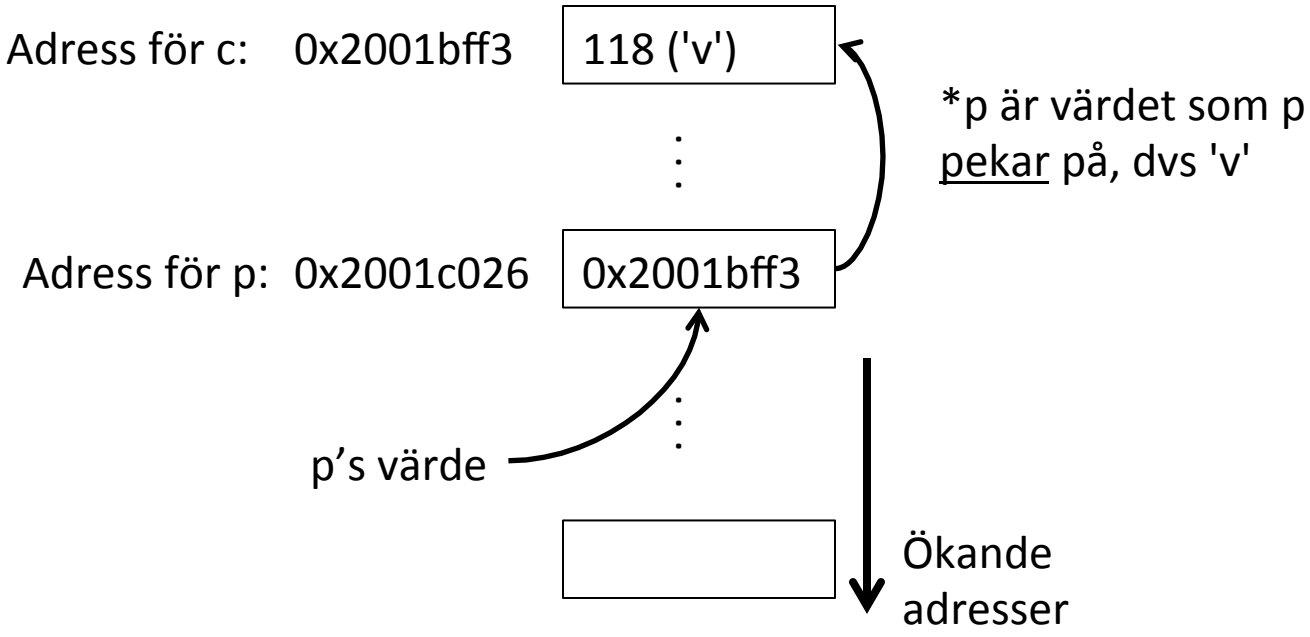
Dvs, om a's värde är en adress, så är *a vad adressen innehåller.

```
&a -> Adress till variabel a. Dvs minnesadress som a är lagrat i.
a  -> variabelns värde (t ex int, float eller en adress om a är pekarvaribel)
*a -> Vad variabel a pekar på (här måste a's värde vara en giltig
      adress och a måste vara av typen pekare). a's värde är en adress till en
      annan variabel eller port. Vi hämtar värdet för den variabeln/porten.
```

Exempel för pekarvariabel p:

```
char  c = 'v';
…
char* p = &c;

*p == c;
```



C vs Assembler

C	Assembler	Betyder
v;	v: .SPACE 4	skapa adress för v (t ex 4 bytes för integer)
&v;	LDR R0, =v	v's adress (t ex 0x2001c010)
v;	LDR R0, v	v's värde (t ex 5)

*v;
v måste här vara pekare,
t ex int *v;
	LDR R0, [v]
	men finns inte på m4
	så man får skriva:
	LDR R7, v
	LDR R0, [R7]
		värdet som ligger på adressen som ligger i v.
		T ex: v = 0x2001c004
		Ladda v's värde (R7=0x2001c004)
		Hämta innehållet på adress 0x2001c004 (4 bytes).

Pekare - lathund

`t *p;`	p får typen “pekare till typen `t`“
`p = 0;`	p blir en tom pekare (pekar ej på något)
`p = &v;`	p tilldelas adressen till variabeln v
`*p`	betyder “det som p pekar på”
`p1 = p2;`	p1 kommer peka på samma som p2
`*p1 = *p2;`	`det som` p1 pekar på kommer att ändras

Aritmetik på pekare

```
char *kurs = "Maskinorienterad Programmering";

*kurs;          // 'M’
*(kurs+2);      // 's’
kurs++;         // kurs pekar på 'a’
kurs +=4;       // kurs pekar på 'n'
```

Man ökar p med (n * typstorlek)

```
int       a[] = {2,3,4,10,8,9};
short int b[] = {2,3,4,10,8,9};
float     c[] = {1.5f, 3.4f, 5.4f, 10.2f, 8.3f, 2.9f};

int *p = a; // == &(a[0])
p++;        // *p == a[1]

short int *p2 = b;
float *p3 = &(c[2]);
```

[Övningsuppgift]

Absolutadressering

- Vid portadressering så kan vi ha en absolut adress (t ex 0x40011004).

Absolutadressering

```
0x40011000                    // ett hexadecimalt tal
(unsigned char*) 0x40011000 // en unsigned char pekare som pekar på adress 0x40011004
*((unsigned char*) 0x40011000)        // dereferens av pekaren

// läser från 0x40011000
unsigned char value = *((unsigned char*) 0x40011000);

// skriver till 0x40011004
*((unsigned char*) 0x40011004) = value;
```

Men… vi måste lägga till volatile om vi har på optimeringsflaggor… !

Läsbarhet med typedef

```
0x40011000
(unsigned char*) 0x40011000
*((unsigned char*) 0x40011000)

// läser från 0x40011000
value = *((unsigned char*) 0x40011000);
```

```
typedef unsigned char* port8ptr;
#define INPORT_ADDR 0x40011000
#define INPORT *((port8ptr)INPORT_ADDR)

INPORT_ADDR
(port8ptr)INPORT_ADDR
INPORT

// läser från 0x40011000
value = INPORT;
```

`typedef` förenklar/förkortar uttryck, vilket kan öka läsbarheten.

```
typedef unsigned char* port8ptr;
```

typ alias/typnamn

Volatile qualifier

```
char * inport = (char*) 0x40011000;

void foo(){

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

En kompilator som optimerar kanske bara läser en gång (eller inte alls om vi aldrig skriver till adressen från programmet).

Volatile qualifier

```
volatile char * inport = (char*) 0x40011000;

void foo(){

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

`volatile` hindrar vissa optimeringar (vilket är bra och här nödvändigt!), ty anger att kompilatorn måste anta att innehållet på adressen kan ändras utifrån.

Vårt tidigare exempel, nu korrekt med volatile:

```
unsigned char value = *((volatile unsigned char*) 0x40011000); // läser från 0x40011004

*((volatile unsigned char*) 0x40011004) = value; // skriver till 0x40011004
```

Sammanfattning portar

Inport:

```
typedef volatile unsigned char* port8ptr;
#define INPORT_ADDR 0x40011000
#define INPORT *((port8ptr)INPORT_ADDR)

// läser från 0x40011000
value = INPORT;
```

Utport:

```
typedef volatile unsigned char* port8ptr;
#define UTPORT_ADDR 0x40011004
#define UTPORT *((port8ptr)UTPORT_ADDR)

// skriver till 0x40011004
UTPORT = value;
```

Array (Fält)

```c
#include <stdio.h>

char namn1[] = {'E', 'm', 'i', 'l', '\0'};
char namn2[] = "Emilia";
char namn3[10];

int main()
{
    printf("namn1: %s \n", namn1);
    printf("namn2: %s \n", namn2);
    printf("sizeof(namn2): %i \n", sizeof(namn2));

    return 0;
}
```

Om ingen storlek anges i [ ] så blir arrayen ”tillräckligt stor” .
OBS. Initiering funkar bara vid definition.

10 element stor array.

Utskrift:

```
namn1: Emil
namn2: Emilia
sizeof(namn2): 7
```

Indexering – samma för array/pekare

```
#include <stdio.h>

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

int main()
{
     // tre ekvivalenta sätt att dereferera en pekare
     printf("'l' i Emilia (version 1): %c \n",      *(s1+3)   );
     printf("'l' i Emilia (version 2): %c \n",      s1[3]     );
     printf("'l' i Emilia (version 3): %c \n",      3[s1]     );

     // tre ekvivalenta sätt att indexera en array
     printf("'l' i Emilia (version 1): %c \n",      *(s2+3)   );
     printf("'l' i Emilia (version 2): %c \n",      s2[3]     );
     printf("'l' i Emilia (version 3): %c \n",      3[s2]     );

     return 0;
}
```

x[y] översätts till *(x+y) och är alltså ett sätt att dereferera en pekare.
Indexering är samma för pekare som för array.

Array – Likhet/olikhet med pekare

- Har en adress och en typ.
  - `char s2[] = "Emilia";`
    - `sizeof(s2) == 7`
    - `Men sizeof(char*) == 4;`

- Indexering har samma resultat.
  - `char* s1 = "Emilia";`
  - `char s2[] = "Emilia";`
    - s1[0] = 'E';
    - s2[0] = 'E';
    - *s1 == 'E';
    - *s2 == 'E'; // eftersom s2 är adress så kan vi dereferera den
      // precis som för en pekare

Array – Likhet/olikhet med pekare

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

	s2	s1
Typ:	Array	Pekarvariabel
Adressering:	**&s2 ej möjligt** ty s2 endast symbol s2 = arrayens startadress. s2 = &(s2[0]) s2[0] = *s2 = 'E'	&s1 = adress för variabeln s1. s1 = strängens startadress. s1 == &(s1[0]) s1[0] = *s1 = 'E'
Pekararitmetik:	**s2++ ej möjligt** (s2+1)[0] helt OK	s1++ helt OK (s1+1)[0] helt OK
Typstorlek:	sizeof(s2) == 7 bytes	sizeof(s1) == sizeof(char*) == 4 bytes

s2 är endast symbol (ej variabel.) för adress som är känd i compile time.
Eftersom s2 är adress så kan vi dereferera den precis som för en pekare: *s2 == ’E’

Indexering – fler exempel

```
#include  <stdio.h>
#include <conio.h>

char * s1 = "Emilia"; // s1 är pekare. Variabeln s1 är en variabel som går att ändra,
                      och vid start tilldelas värdet av adressen till 'E':
char s2[] = "Emilia"; // s2 är array. Värdet på symbolen s2 är känt vid compile time. Symbolen s2 är konstant,
                       dvs ingen variabel som går att ändra. Är adressen till 'E'.

int main()
{
    // tre ekvivalenta sätt att dereferera en pekare
    printf("'l' i Emilia (version 1): %c \n",          *(s1+3)    );
    printf("'l' i Emilia (version 2): %c \n",          s1[3]      );
    printf("'l' i Emilia (version 3): %c \n",          3[s1]      );
    printf("'l' i Emilia (version 3): %c \n",          *(s2+3)    );
    printf("'l' i Emilia (version 3): %c \n",          (s2+3)[0]  );

    char a[] = "hej";
    (a+1)[0] = 'o';
    char* p = a;
    p = "bye"; // funkar. Strängen ”bye" allokeras i compile time i strängliteralminne.

    char b[10]; // b blir 10 element stor och får adressvärde
    // b = "då"; // här försöker vi ändra b's värde och det GÅR inte.
    printf("%s\n", p);

    return 0;
}
```

Arrayer som funktionsargument blir pekare

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

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

[ ] – notationen finns, men betyder pekare!

Undviker att hela arrayen kopieras. Längd inte alltid känd i compile time.
Adressen till arrayen läggs på stacken och accessas via stackvariabeln pi.

(En struct kopieras och läggs på stacken).

Antal bytes med `sizeof()`

```c
#include <stdio.h>

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

int main()
{
    printf("sizeof(char):  %i \n", sizeof(char) );
    printf("sizeof(char*): %i \n", sizeof(char*) );
    printf("sizeof(s1):    %i \n", sizeof(s1) );
    printf("sizeof(s2):    %i \n", sizeof(s2) );

    return 0;
}
```

```
sizeof(char):  1
sizeof(char*): 4
sizeof(s1):    4
sizeof(s2):    7
```

Sizeof utvärderas i compile-time. En (av få) undantag där arrayer och pekare är olika.

Array av pekare

```c
#include <stdio.h>

char *fleraNamn[] = {"Emil", "Emilia", "Droopy"};

int main()
{
    printf("%s, %s, %s\n", fleraNamn[2], fleraNamn[1], fleraNamn[0]);

    return 0;
}
```

```
Droopy, Emilia, Emil
```

```
sizeof(fleraNamn) = 12; // 3*sizeof(char*) = 3*4 = 12
```

Array av arrayer

```c
#include <stdio.h>

char kortaNamn[][4] = {"Tor", "Ulf", "Per"};

int main()
{
    printf("%s, %s, %s\n", kortaNamn[2], kortaNamn[1], kortaNamn[0]);

    return 0;
}
```

Per, Ulf, Tor

sizeof(kortaNamn) = …

Array av arrayer

```
#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++) {
        printf("arrayOfArray[%i] = ", i);
        for ( j=0; j<4; j++)
            printf("%i ", arrayOfArrays[i][j]);
        printf("\n");
    }

    return 0;
}
```

Nästa föreläsning:

Structs, funktionspekare