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

Pekare

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

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?

Bland annat:

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

Exempel 1:

```
char person1[] = "Elsa";
char person2[] = "Alice";
char person3[] = "Maja";
…
char* winner = person2;
```

Man kan säga att winner refererar till person2 men vi kallar det inte referens utan pekare.
Dvs winner **pekar** på person2.

Exempel 2:

```
int löneNivå1 = 1000;
int löneNivå2 = 2000;
int löneNivå3 = 3000;
…
int* minLön = &löneNivå3;
```

minLön pekar på löneNivå3.

Pekare

1. Pekarens värde är en adress (&).
2. Pekarens typ berättar hur man tolkar bitarna som finns på adressen.
3. * används för att läsa (dereferera) innehållet på adressen.

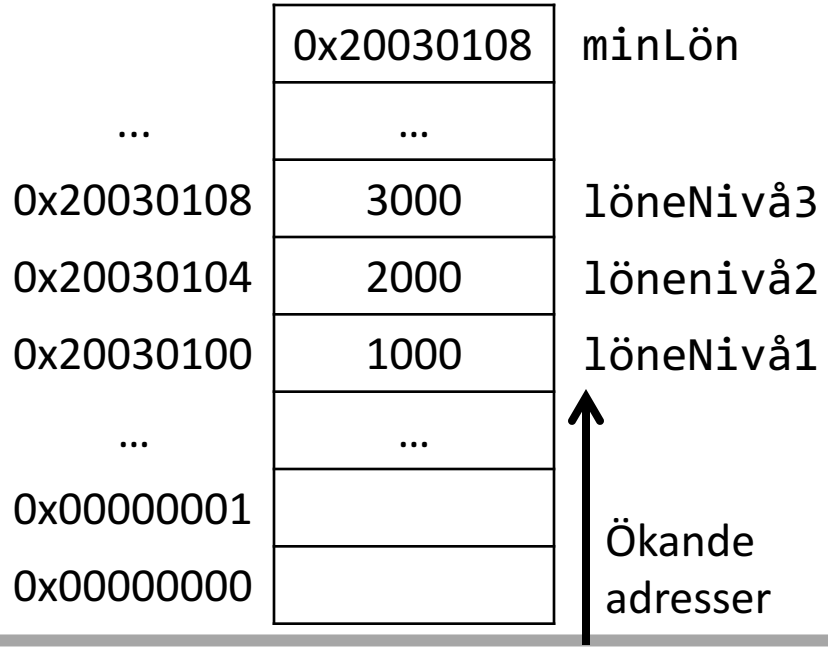
```
int löneNivå1 = 1000;
int löneNivå2 = 2000;
int löneNivå3 = 3000;

int* minLön = &löneNivå3; // == 0x20030108
```

typ

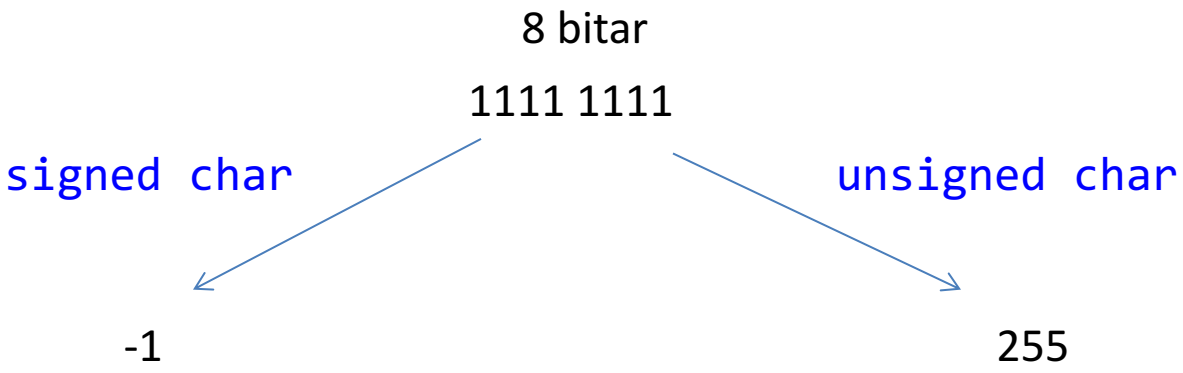
värdet är en adress

```
minLön är 0x20030108
*minLön är 3000.
```



Pekare – dereferera (*)

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



Pekare – operatorer & *

```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';
```

Pekare – sammanfattning

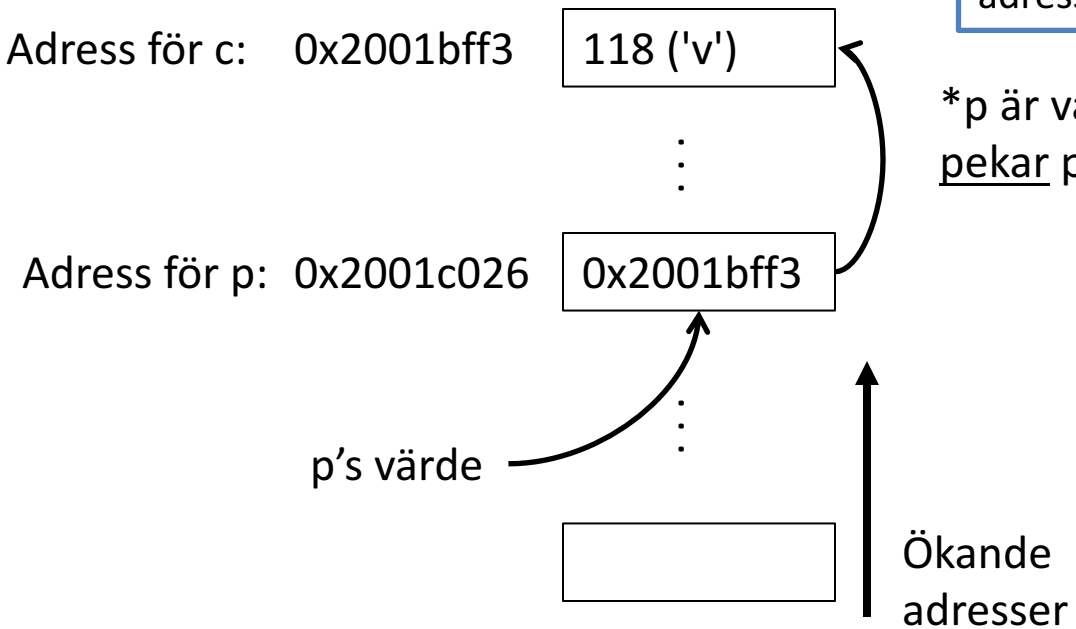
```
&a -> Adress till variabel a. Dvs minnesadress som a är lagrat i.
a  -> variabelns värde (t ex int, float eller en adress om a är pekarvaribel)
*a -> Vad variabel a pekar på. Här måste a's värde vara en giltig adress (t ex till en
      annan variabel eller port) och a måste vara av typen pekare.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

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

Exempel:

```
char  c = 'v';
…
char* p = &c;
```

&p är 0x2001c026
p är 0x2001bff3
*p är 'v'



*p är värdet som p pekar på, dvs 'v'

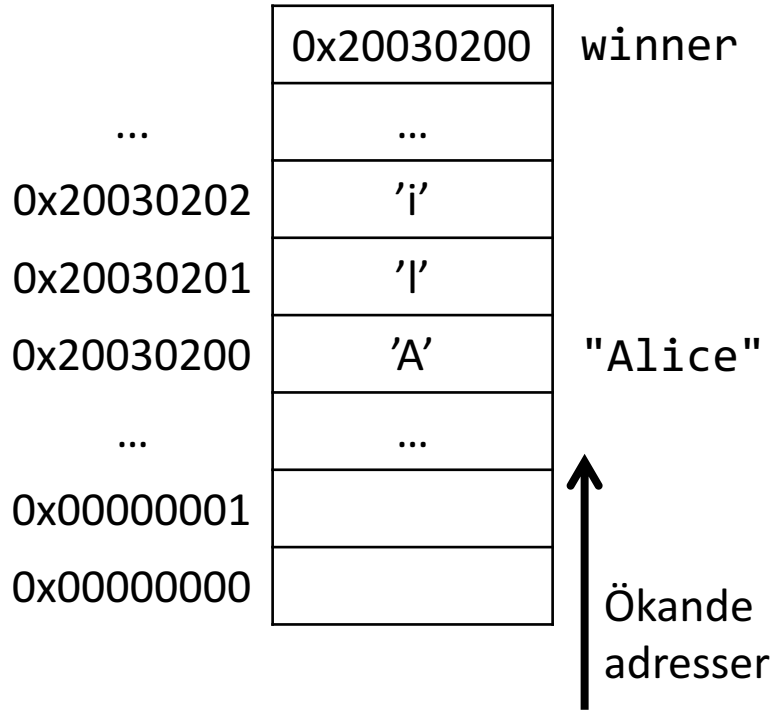
Pekare – ett till exempel

```
char person1[] = "Elsa";
char person2[] = "Alice";
char person3[] = "Maja";
…
char* winner = &person2[0]; // == 0x20030200
```

typ

värdet är en adress

```
winner är 0x20030200
*winner är "Alice".
```



Pekare – fler exempel

```
int       a[] = {2,3,4,10,8,9};
int       *pa = &a[0];

short int b[] = {2,3,4,10,8,9};
short int *pb = b;

float     c[] = {1.5f, 3.4f, 5.4f, 10.2f, 8.3f, 2.9f};
float     *pc = &c[3];
```

Pekare till sträng:

```
char kurs[] = "Maskinorienterad Programmering";
char *pKurs = kurs;
```

Vanlig skrivbar array på stacken eller i programmets datasegment.

Eller direkt så här:

```
char *pKurs = "Programmering av inbyggda system";
```

Men här lägger C-kompilatorn strängen i skrivskyddat strängliteralminne i programmets datasegment

Pekare – C vs Assembler

C	Assembler	Betyder
int 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 finns ej i thumb
	LDR R0, =v
	LDR R0, [R0]
		v's värde (t ex 5 eller 0x2001c004)

*v;
(v måste här vara pekare,
t ex int *v;)
	LDR R0, [v] finns ej i m4
	LDR R0, =v
	LDR R0, [R0]
	LDR R0, [R0]
		värdet som ligger på
		adressen som ligger i v.

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 till det som p2 pekar på.

Mer 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

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};

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

int *p3 = a + 3;
```

[Övningsuppgifter]

1. Skapa två pekare, pa respektive pb, till:
   ```
   int a = 5;
   char b = 's';
   ```

2. Ändra a och b's värden via pekarna.

3. Gör en funktion som modifierar en variabel, te x adderar 5:

```
void add5(…)
{
…
}

int a;
add5( … );  // ska öka a med 5
```

[Övningsuppgifter]

1. Skapa två pekare, pa respektive pb, till:
   ```
   int a = 5;
   char b = 's';
   ```

2. Ändra a och b's värden via pekarna.

3. Gör en funktion som modifierar en variabel, te x adderar 5:

```
void add5(…)
{
…
}

int a;
add5( … );  // ska öka a med 5
```

Svar:

1.
```
int *pa = &a;
char *pb = &b;
```

2.
```
*pa = 10;
*pb = 't';
```

3.
```
void add5(int* p)
{
   *p += 5;
}
…
int a;
add5( &a );
```

Pekare för Absolutadressering

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

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

INPORT_ADDR
(port8ptr)INPORT_ADDR
INPORT

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

Evalueras av preprocessorn till:

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

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

`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 char * utport = (char*)
                        0x40011000;
void f2()
{
    *utport = 0;
    *utport = 1;
    *utport = 2;
}
```

`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;
```

Pekare och Arrayer

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.

Indexering – samma för array/pekare

```c
#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.

Så är arrayer pekare? Nej…

Array – Likhet/olikhet med pekare

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

- Båda har en adress och en typ.
  - `char s2[] = "Emilia";`
    - `sizeof(s2) = 7`
  - `char* s1 = "Emilia";`
    - `sizeof(s1) = sizeof(char*) = 4`

- Indexering har samma resultat.
  - 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 = symbol = arrayens startadress. s2 = &(s2[0]) s2[0] = *s2 = 'E'	&s1 = adress för variabeln s1. s1 = s1's värde = 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 symbol (ej variabel) för en adress som är känd i compile time.
Eftersom s2 är en adress kan vi dereferera den precis som för en pekare: *s2 = 'E'.

Indexering – fler exempel

```
#include <stdio.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.
                      // s2 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] = "hej"; // b blir 10 element stor.
    // b = "då"; // här försöker vi ändra b's värde, men det går inte via "…"-syntax (C stödjer inte det).
    b[0] = 'd';  // OK
    b[1] = 'å';  // OK

    return 0;
}
```

Arrayer som funktionsargument blir pekare

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

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

[ ] – 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 i.

(En struct kopieras och läggs på stacken).

```
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};
sumElements(array, 5);
```

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;
}
```

[Övningsuppgifter]

1. Skapa en port till en char som ligger på adress 0x40004000.
2. Skapa en pekare till en sträng som ligger i skrivskyddat strängliteralminne i datasegmentet.
3. Skapa en pekare till en sträng som ligger på stacken.
4. Använd typedef för att skapa ett typalias byteptr till en unsigned byte.
5. Vad gör volatile.

[Övningsuppgifter]

1. Skapa en port till en int som ligger på adress 0x40004000.
2. Skapa en pekare till en sträng som ligger i skrivskyddat strängliteralminne i datasegmentet.
3. Skapa en pekare till en sträng som ligger på stacken.
4. Använd typedef för att skapa ett typalias byteptr till en unsigned byte.
5. Vad gör volatile.

Svar:

1.
```
typedef volatile int* port8ptr;
#define PORT_ADDR 0x40004000
#define PORT *((port8ptr)PORT_ADDR);
```

2.
```
char *p = “hej”;
```

3.
```
void fkn()
{
    char s[] = “hej”; // på stacken
    char* p = s;
}
```

4.
```
typedef unsigned char *byteptr;
```

5. Läsning/skrivning av volatile-variabel optimeras ej bort. Volatile därför nödvändigt för portar.

Nästa föreläsning:

Structs, funktionspekare