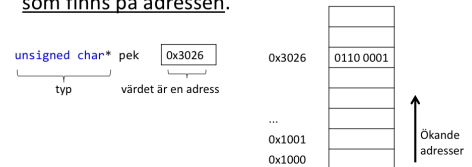


Pekare

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



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



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	(NUL)	32	20	SPACE	64	40	@	96	60	>
1	1	(START OF HEADING)	33	21	!	65	41	A	97	61	a
2	2	(START OF TEXT)	34	22	"	66	42	B	98	62	b
3	3	(END OF TEXT)	35	23	#	67	43	C	99	63	c
4	4	(FORM FEED)	36	24	\$	68	44	D	100	64	d
5	5	(BEL)	37	25	%	69	45	E	101	65	e
6	6	(BACKSPACE)	38	26	&	70	46	F	102	66	f
7	7	(TAB)	39	27	'	71	47	G	103	67	g
8	8	(BACKSPACE)	40	28	(72	48	H	104	68	h
9	9	(TAB)	41	29)	73	49	I	105	69	i
10	A	(LINE FEED)	42	2A	*	74	4A	J	106	6A	j
11	B	(VERTICAL TAB)	43	2B	+	75	4B	K	107	6B	k
12	C	(FORM FEED)	44	2C	,	76	4C	L	108	6C	l
13	D	(CARRIAGE RETURN)	45	2D	-	77	4D	M	109	6D	m
14	E	(TAB)	46	2E	.	78	4E	N	110	6E	n
15	F	(SHIFT IN)	47	2F	/	79	4F	O	111	6F	o
16	10	(DATA LINK ESCAPE)	48	30	0	80	50	P	112	70	p
17	11	(DEVICE CONTROL 1)	49	31	1	81	51	Q	113	71	q
18	12	(DEVICE CONTROL 2)	50	32	2	82	52	R	114	72	r
19	13	(DEVICE CONTROL 3)	51	33	3	83	53	S	115	73	s
20	14	(DEVICE CONTROL 4)	52	34	4	84	54	T	116	74	t
21	15	(NEGATIVE ACKNOWLEDGE)	53	35	5	85	55	U	117	75	u
22	16	(SYNCHRONOUS GATE)	54	36	6	86	56	V	118	76	v
23	17	(END OF TRANS. BLOCK)	55	37	7	87	57	W	119	77	w
24	18	(DEVICE CONTROL 5)	56	38	8	88	58	X	120	78	x
25	19	(END OF MEDIUM)	57	39	9	89	59	Y	121	79	y
26	1A	(SUBSTITUTE)	58	3A	:	90	5A	Z	122	7A	z
27	1B	(ESCAPE)	59	3B	;	91	5B	[123	7B	{
28	1C	(FILE SEPARATOR)	60	3C	<	92	5C	\	124	7C	
29	1D	(INVERTING PLANE)	61	3D	=	93	5D]	125	7D	}
30	1E	(RECORD SEPARATOR)	62	3E	>	94	5E	^	126	7E	~
31	1F	(UNIT SEPARATOR)	63	3F	?	95	5F	_	127	7F	(DEL)

Operatorer

```
#include <stdio.h>

int main()
{
    char a, b, *p; // Deklaration av pekare
    a = 'v';
    b = a; // Adressen av ...
    p = &a;

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

Utskift:
b = v, p = 0x0027F7C3 (v)
b = v, p = 0x0027F7C3 (k)

Asterisken (*) betyder

- I deklarerationer
 - Pekartyp

```
char *p;
char* p;

void foo(int *pi);
```

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

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

p är en vanlig variabel men är en pekare till en char

Om namn1 ligger precis före namn2 i minnet och vi tar bort '\0' så kommer den första Printf-satsen skriva ut till den hittar ett nullostermineringstecken. Detta kommer ske först i slutet av "Emilia" så printf(namn1) kommer ge "EmilEmilia"

Aritmetik på pekare

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

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

Array (Fält)

```
#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;
}
```

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

Likhet med pekare

- Har en adress och en typ.
- Indexering har samma resultat.

Indexering

```
#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", s1[3] );

    return 0;
}
```

$x[y]$ översätts till $*(x+y)$ och är alltså ett sätt att dereferera en pekare.

Skillnader mellan array och pekare

- Arrayer
 - Adress känd i compile-time.
 - Storlek känd i compile-time.
 - Pekar-artitmetik inte möjlig.

Oftast en relativ adress.
T ex 103 bytes efter
första instruktionen.

```
#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. Funkar bara med
initiering vid deklaration!**

10 element stor array.

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

Man kan aldrig ha Arrayer som argument. Vad man menar när man skriver som till vänster är att det egentligen är en pekare.

Funktionsargument blir pekare

```
void foo(int p1[]);
```

```
void foo(int *p1);
```

Då argument till funktioner bara är kända i run-time så kan vi inte passa arrayer.

[] - notationen finns, men betyder pekare!

Antal bytes med sizeof()

```
#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.

Dynamisk minnesallokering

- `malloc()` Allokera minne
- `free()` Frigör minne

Funktionsprototyp via:

```
#include <stdlib.h>
```

Dynamisk minnesallokering

```
#include <stdlib.h>

char s1[] = "This is a long string. It is even more than one sentence.";

int main()
{
    char* p;

    // allokera minne dynamiskt
    p = (char*)malloc(sizeof(s1));

    // gör något med minnet som vi reserverat
    // frigör minnet
    free(p);
    return 0;
}
```

Antal bytes vi allokerar

Minnesläckor

- En minnesläcka uppkommer om vi inte frigör det minne som vi allokert med `malloc()`.
- Minnesläckor kan orsaka systemhaveri om minnet tar slut.
- Minnesläckan försvinner när programmet terminerar.

Hitta minnesläckor

- Vi kommer använda en minnesanalysator DrMemory (<http://www.drmemory.org/>)
- DrMemory ersätter standard biblioteket, och analyserar anrop till `malloc()` och `free()`.

