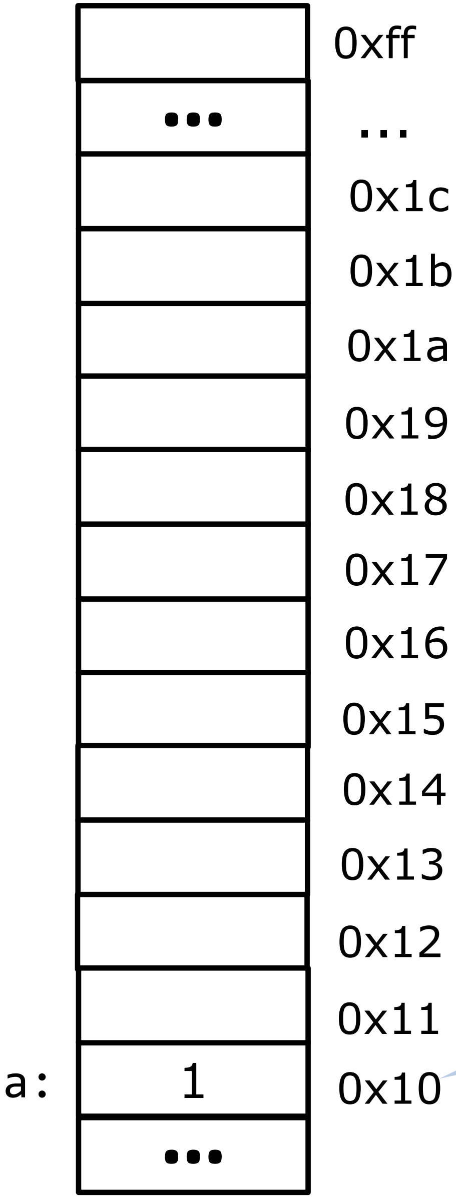
C: Pointers and Arrays

Jinyang Li

Pointers

Pointer is a memory address

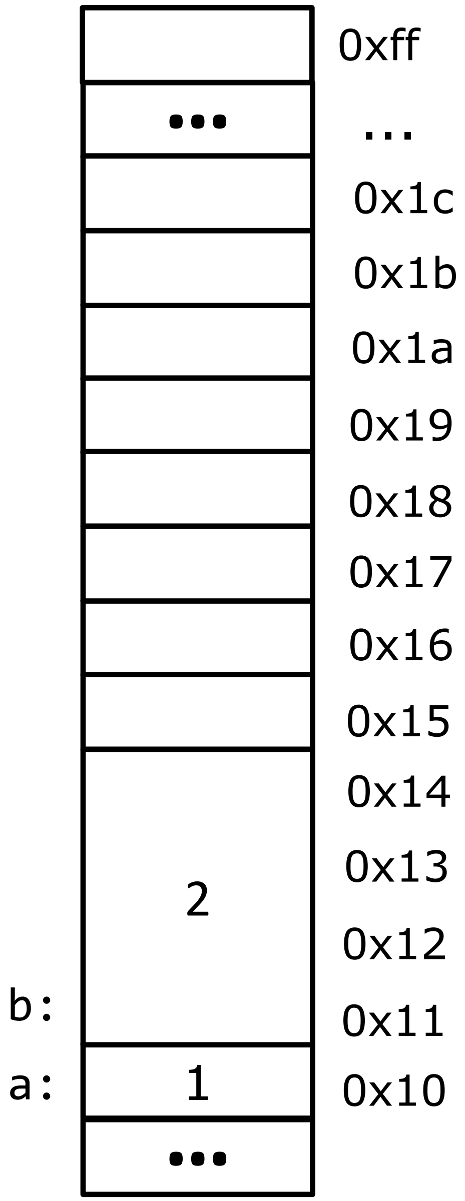
Pointer



```
char a = 1;
```

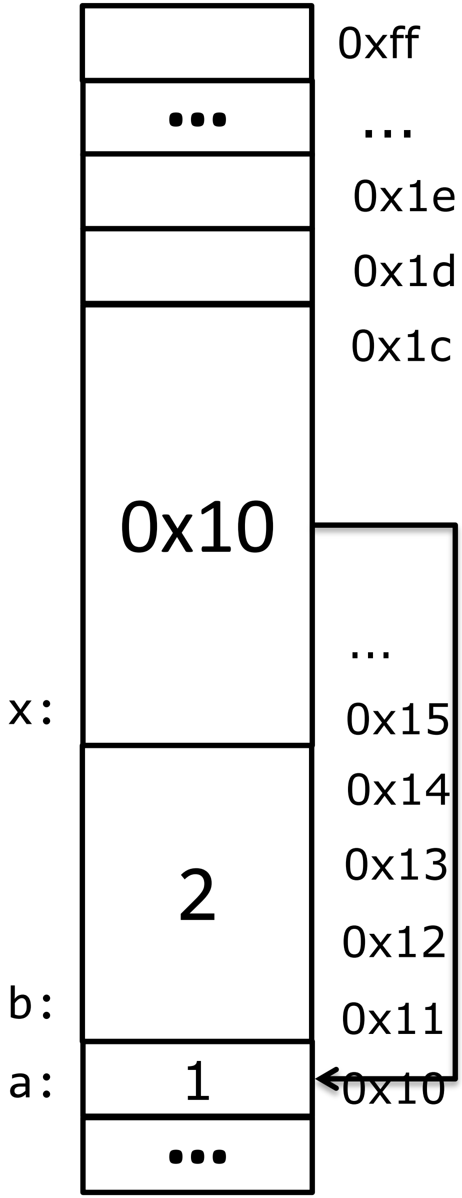
Addresses are 8-byte long on 64-bit machine; for the same of brevity, I omit leading 0s.

Pointer



```
char a = 1;
int b = 2;
```

Pointer



```
char a = 1;
int b = 2;
char *x;
x = &a;
```

Same as: char* x;
You pronounce typename from right to left

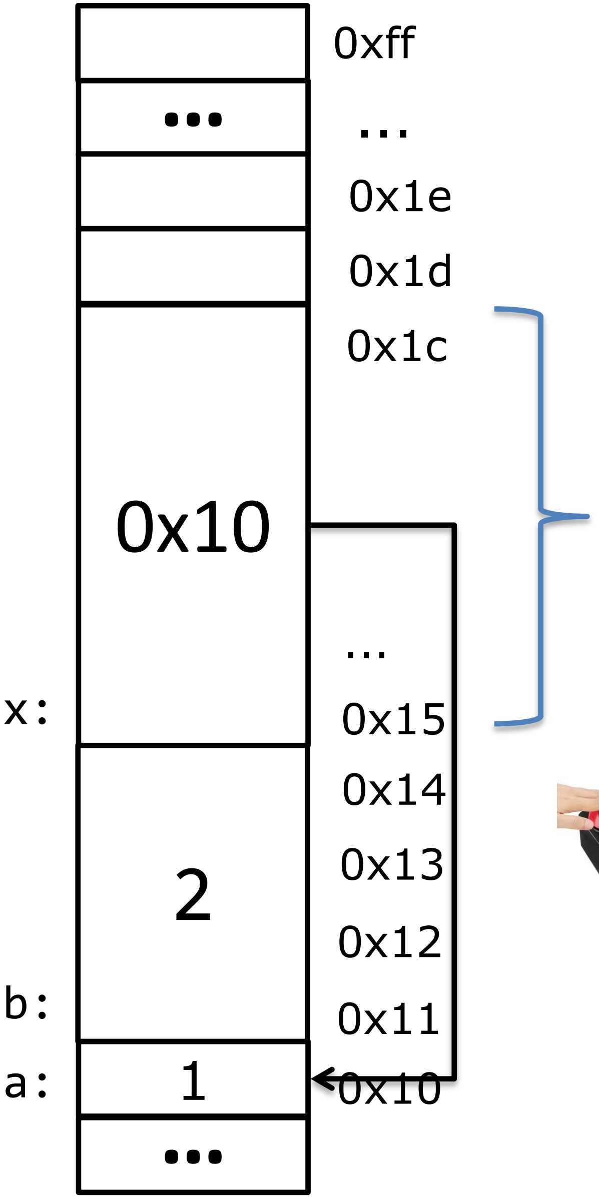
& gives address of variable

Can be combined as:
char *x = &a;

what happens if I write
char x = &a;

type mismatch!

Pointer

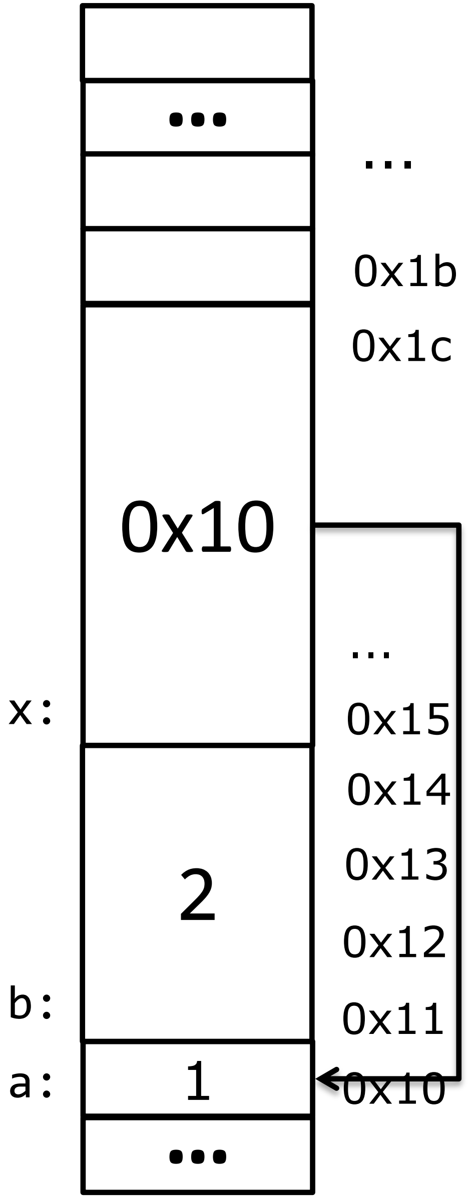


```
char a = 1;
int b = 2;
char *x = &a;
printf("x=%p\n", x);
```

You can print the value of a pointer variable with %p (leading zeros are not printed)
0x10

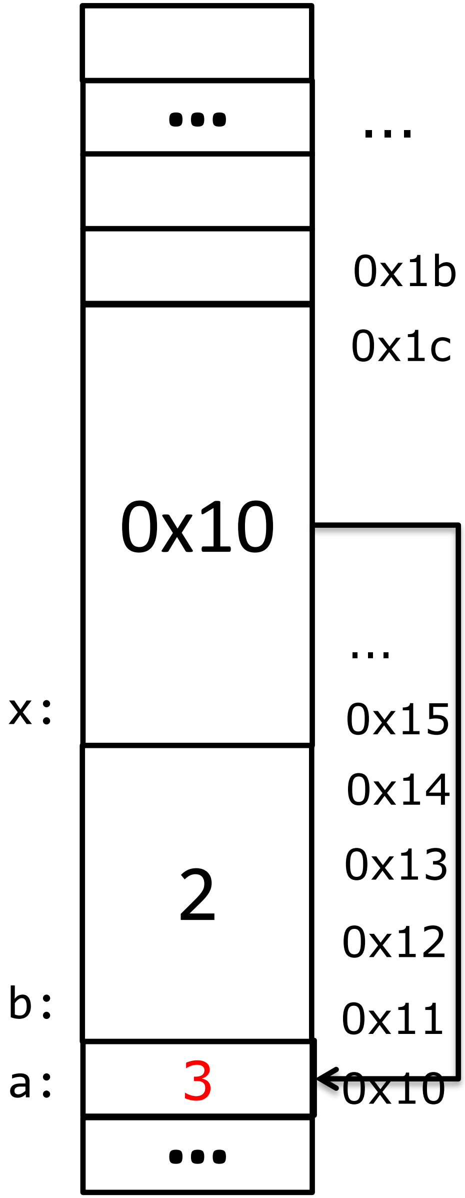
Size of pointer on a 64-bit machine?
8 bytes

Pointer



```
char a = 1;
int b = 2;
char *x = &a;
```

Pointer



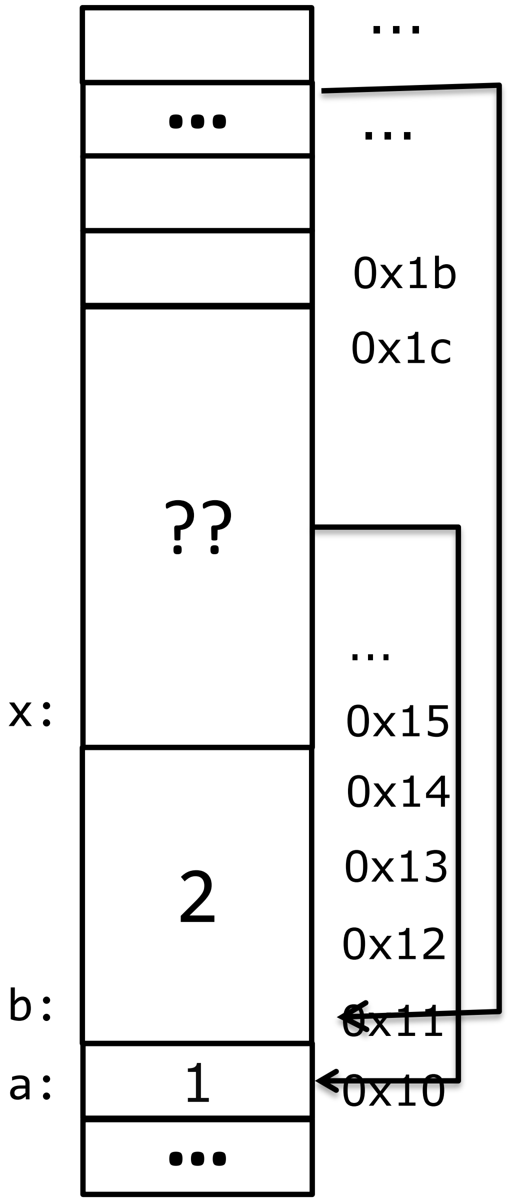
```
char a = 1;
int b = 2;
char *x = &a;
```

```
*x = 3;
```

* operator dereferences a pointer, not to be confused with the * in (char *) which is part of typename

Value of variable a after this statement?

Pointer



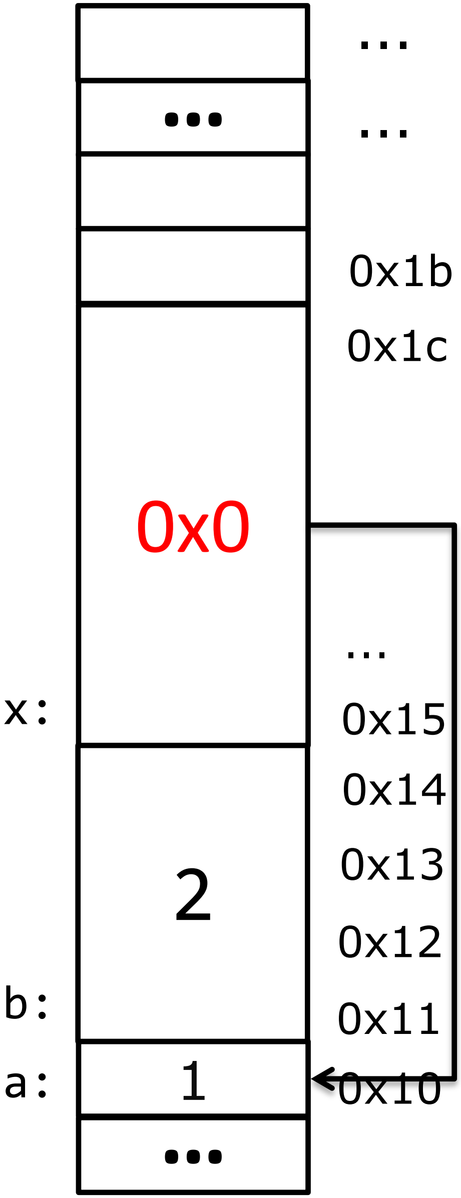
```
char a = 1;
int b = 2;
char *x ~~= &a~~;
```

what if x is uninitialized?

```
*x = 3;
```

Dereferencing an arbitrary address value may result in “Segmentation fault” or a random memory write

Pointer



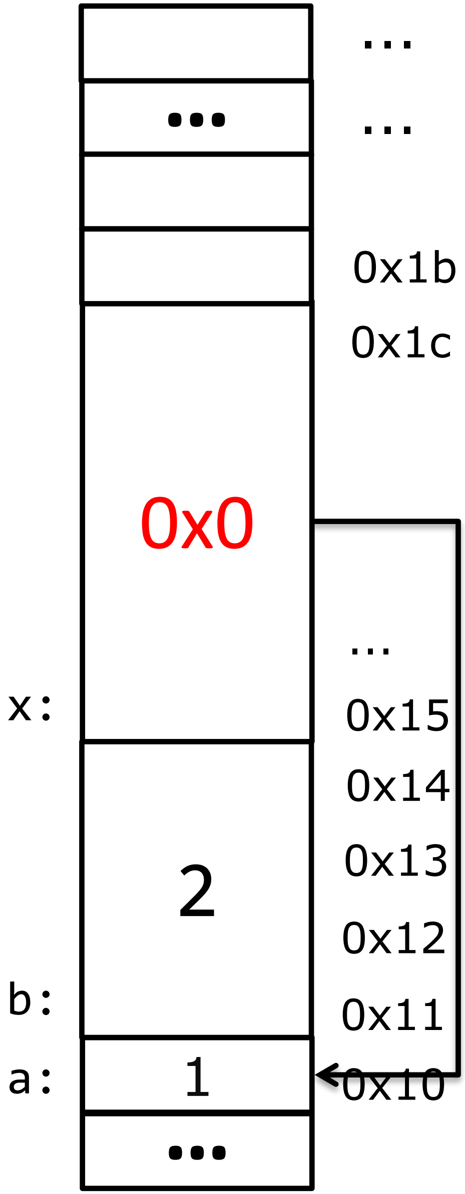
```
char a = 1;
int b = 2;
char *x = NULL;
```

Always initialize pointers!

```
*x = 3;
```

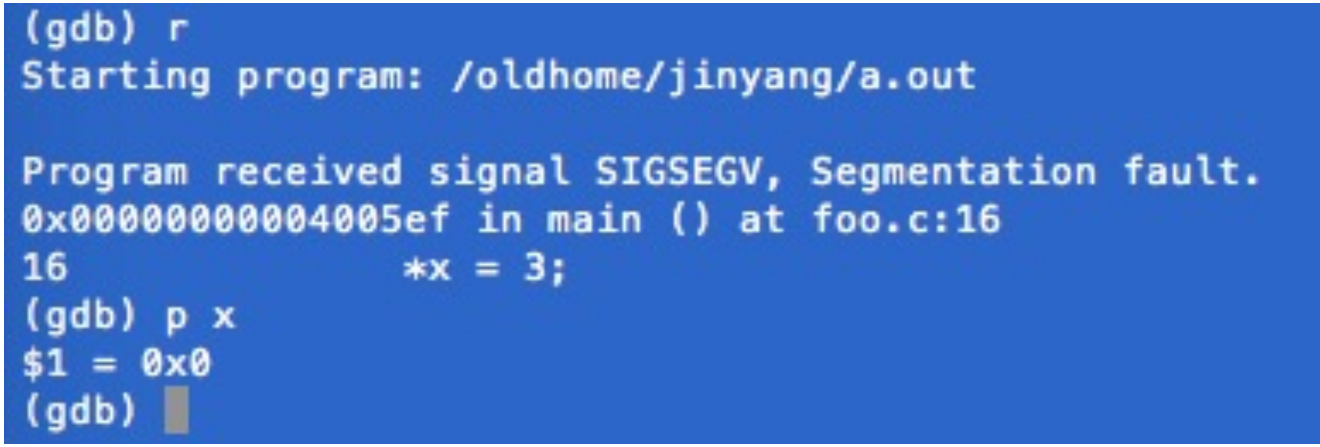
Dereferencing NULL pointer definitely results in “Segmentation fault”

Pointer



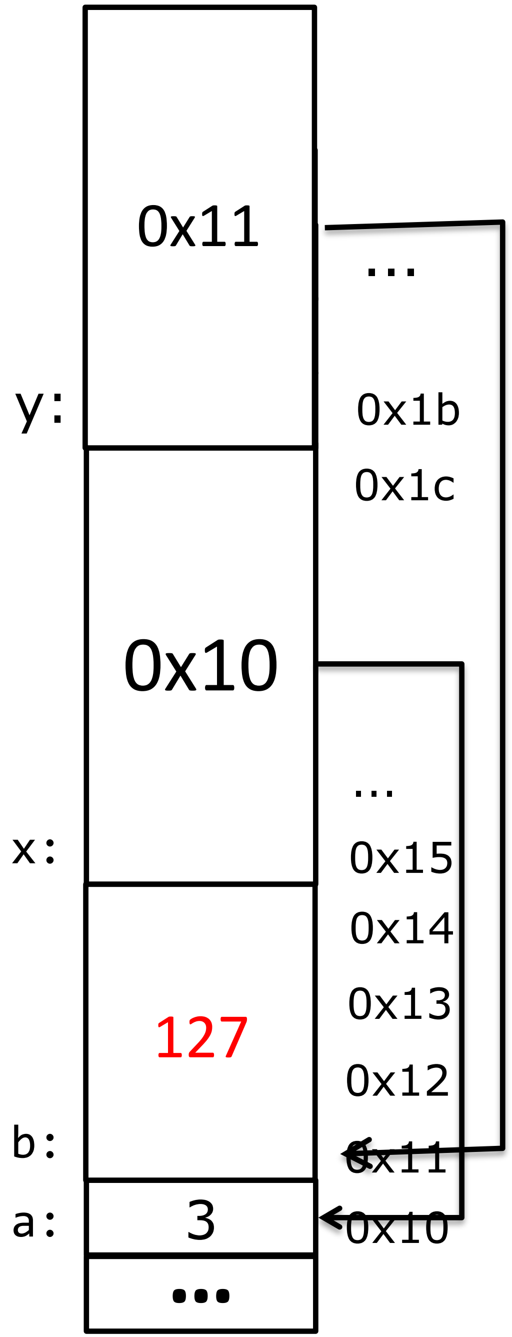
```
char a = 1;
int b = 2;
char *x = NULL;
```

```
*x = 3;
```



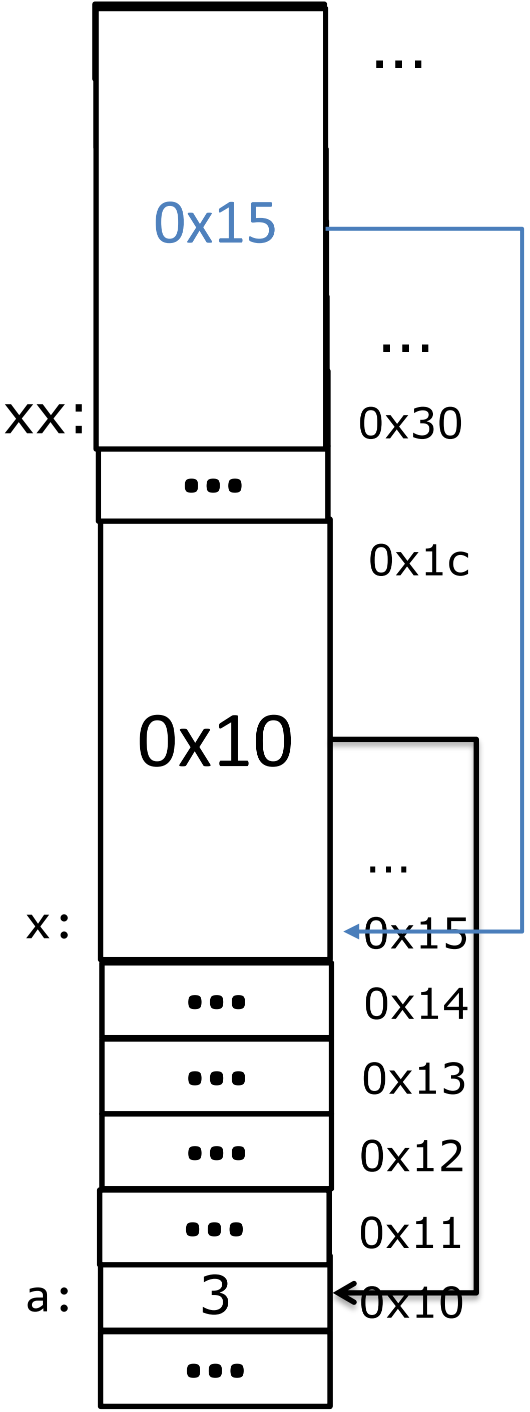
```
(gdb) r
Starting program: /oldhome/jinyang/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005ef in main () at foo.c:16
16              *x = 3;
(gdb) p x
$1 = 0x0
(gdb) 
```

Pointer has different types



```
char a = 1;
int b = 2;
char *x = &a;
 *x = 3;

 int *y = &b;
 *y = 127;
```

what if I write
char *y = &b;



Double Pointer

```
char a = 1;
int b = 2;
char *x = &a;
*x = 3;

char **xx = &x;
```

Same as:
char **xx;
xx = &x;

what if I write
char *xx =&x;

char **xx is the same as char** xx;

printf(“xx=%p *xx=%p **xx=%d\n”, xx, *xx, **xx);

Common confusions on *

* has two meanings!!

1. part of a pointer type name, e.g. char *, char **, int *
2. the deference operator.

```
char a = 1;
char *p = &a;
*p = 2;

char *b, *c;
char **d,**e;

char *f=p, *g=p;
char **m=&p, **n=&p;
```

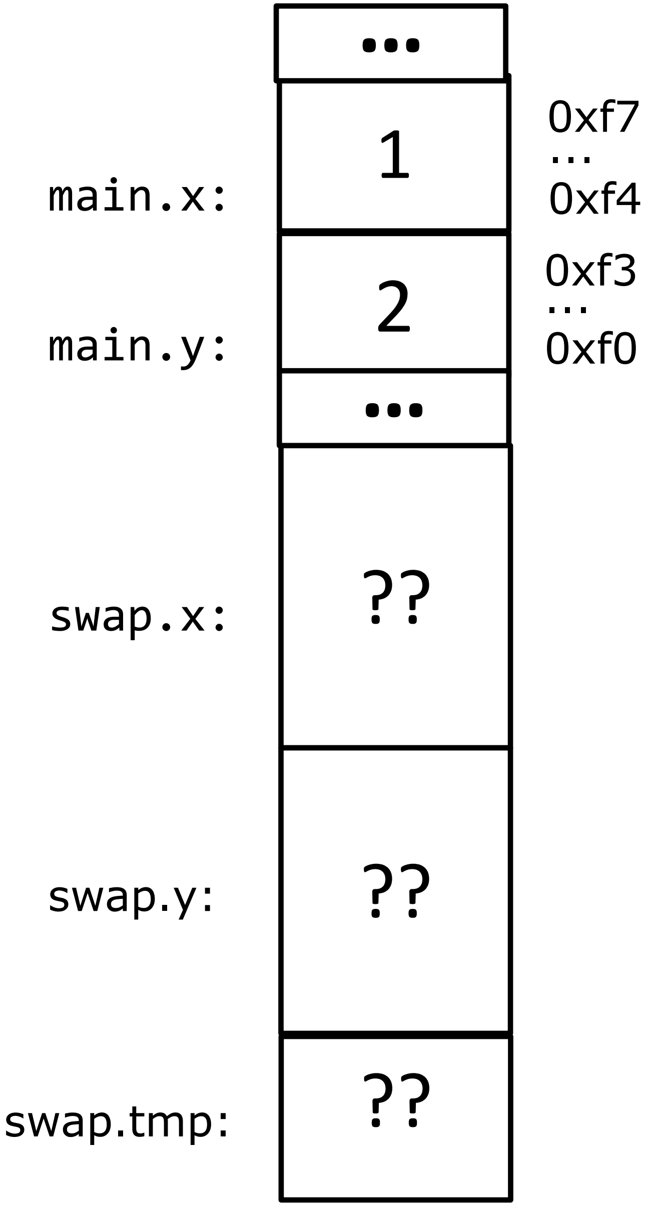
C’s syntax for declaring multiple pointer variables on one line

char*   b, c; does not work

C’s syntax for declaring and initializing multiple pointer variables on one line

Pass pointers to function

```
void swap(int* x, int* y)
{
     int tmp = *x;
     *x = *y;
     *y = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf("x:%d, y:%d",x,y);
}
```



•••

main.x: 1 0xf7 … 0xf4

main.y: 2 0xf3 … 0xf0

•••

swap.x: ??

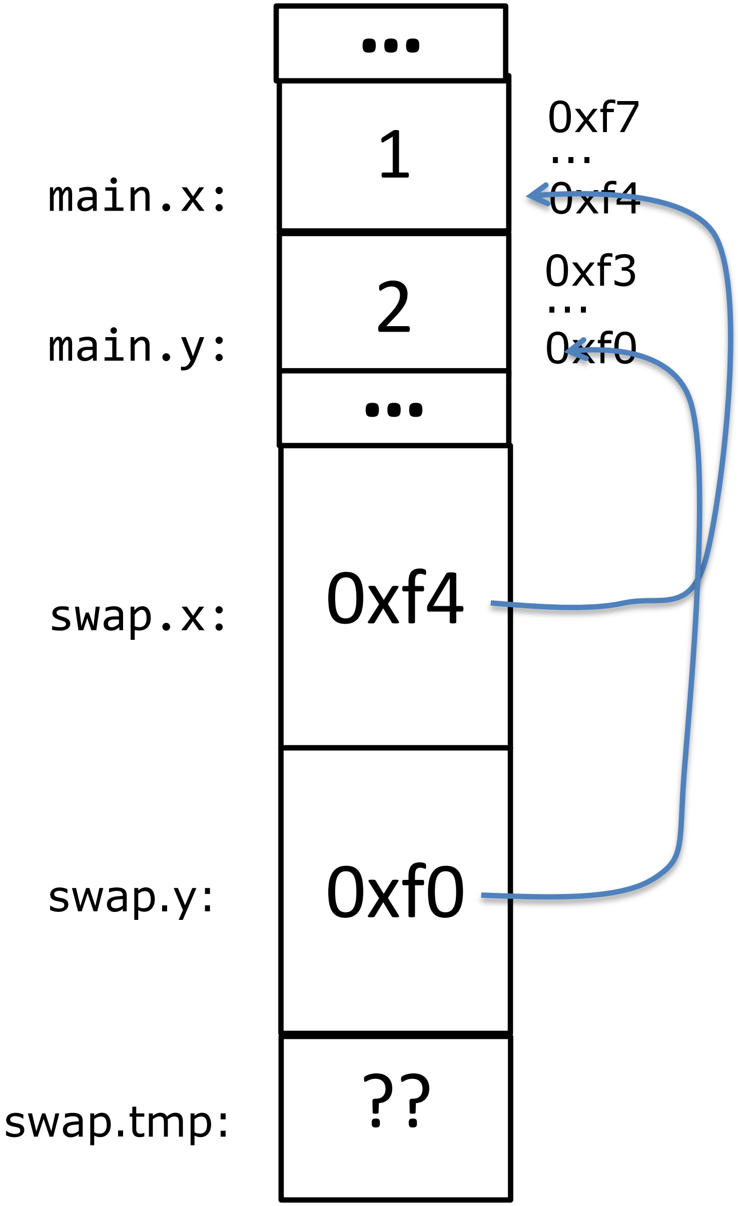
swap.y: ??

swap.tmp: ??

Size and value of x, y, tmp
in swap upon function entrance?

Pass pointers to function

```
void swap(int* x, int* y)
{
     int tmp = *x;
     *x = *y;
     *y = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf(“x:%d, y:%d”,x,y);
}
```



…

main.x: 1 0xf7 … 0xf4

main.y: 2 0xf3 … 0xf0

…

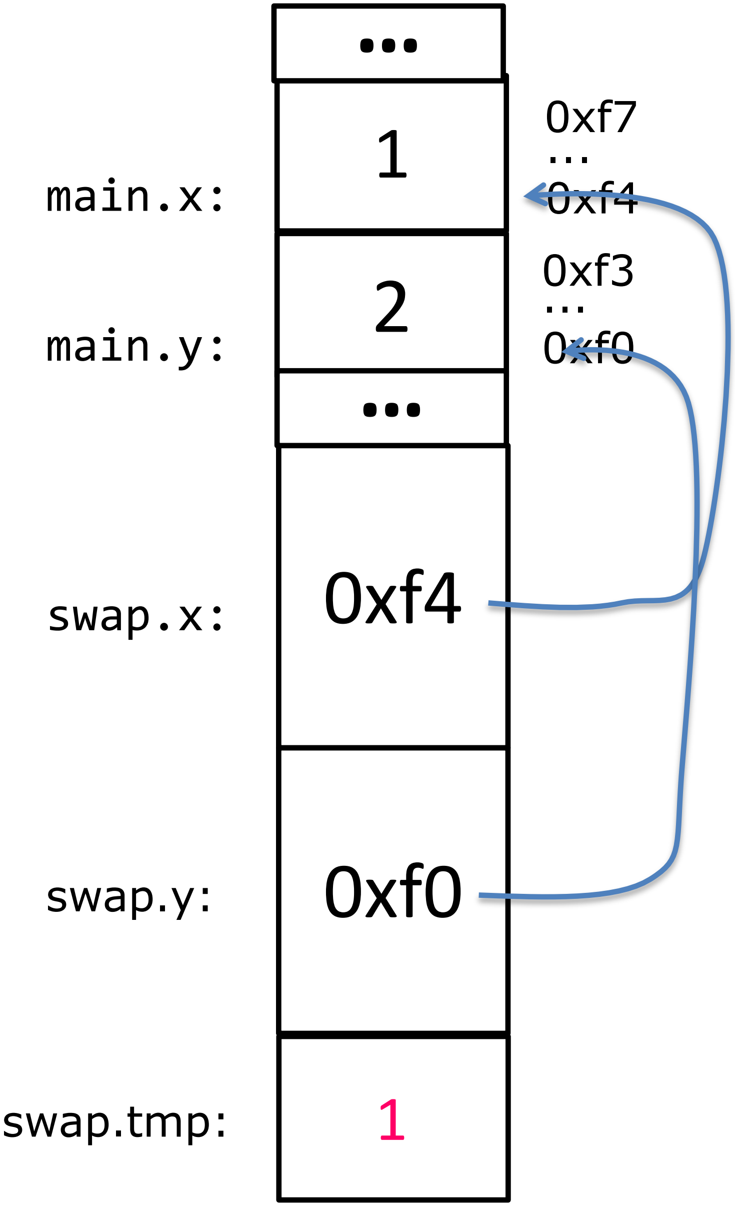
swap.x: 0xf4

swap.y: 0xf0

swap.tmp: ??

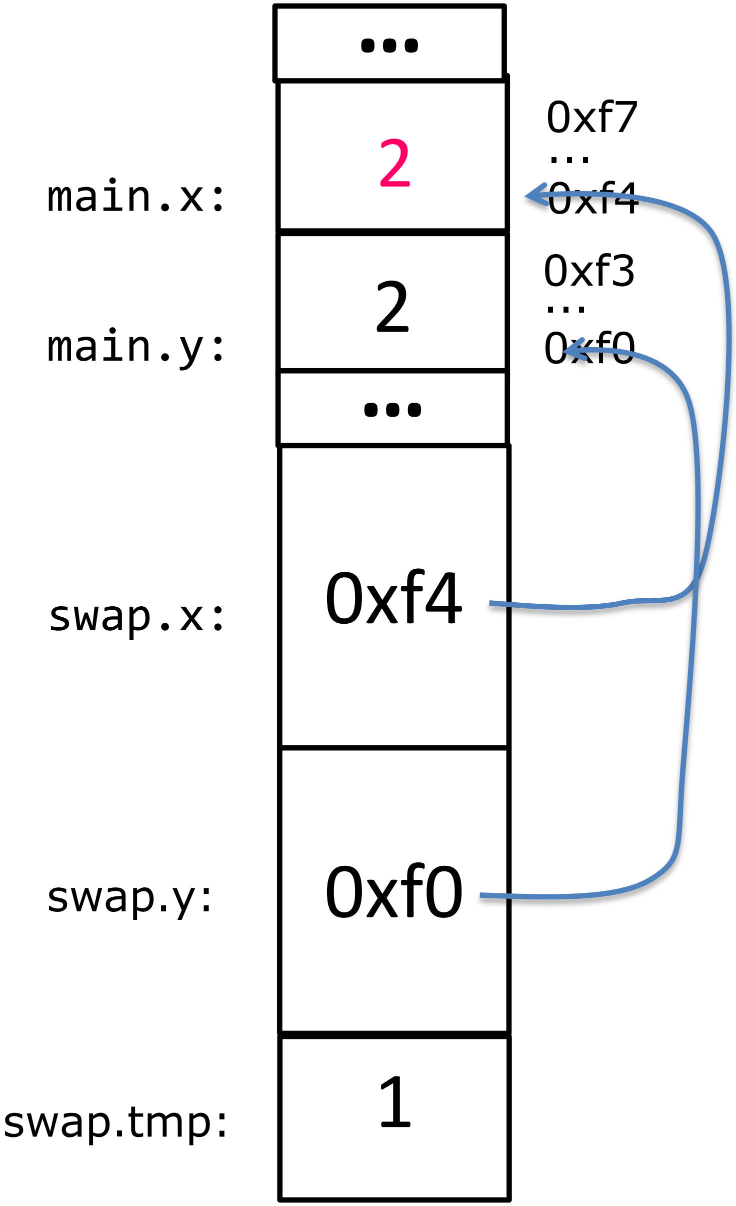
Pass pointers to function

```
void swap(int* x, int* y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf("x:%d, y:%d",x,y);
}
```



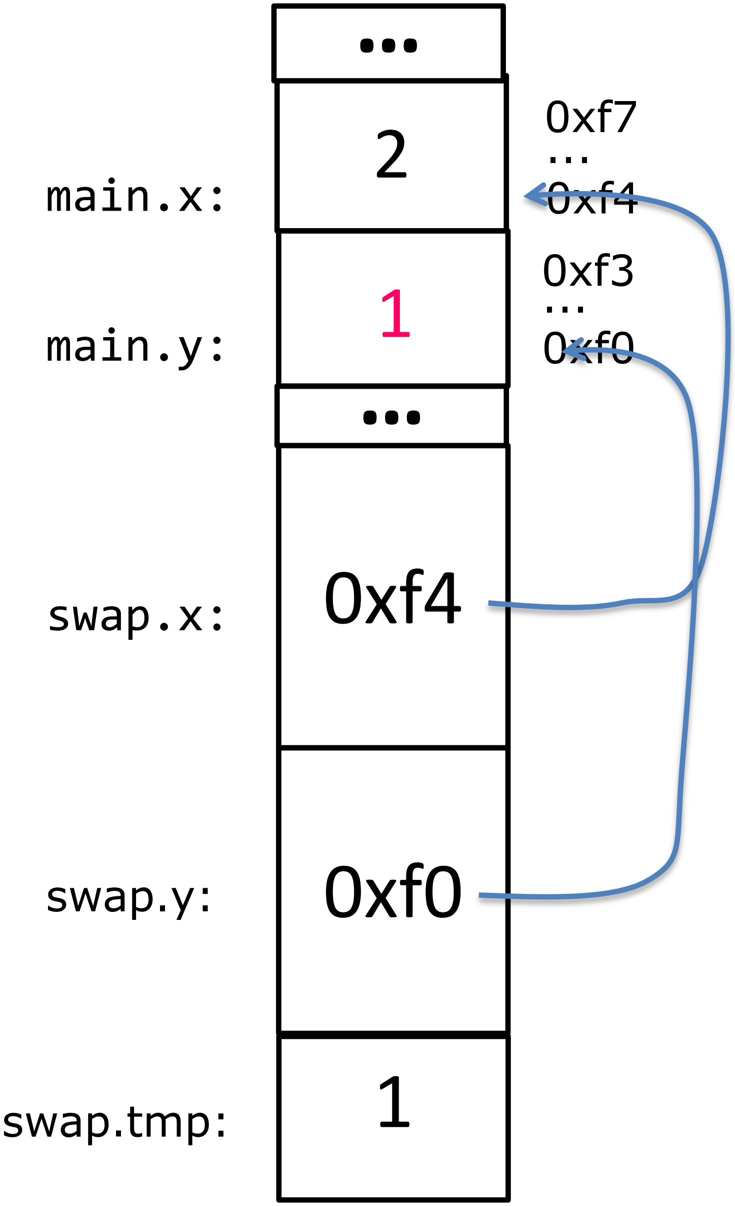
Pass pointers to function

```
void swap(int* x, int* y)
{
     int tmp = *x;
     *x = *y;
     *y = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf("x:%d, y:%d",x,y);
}
```



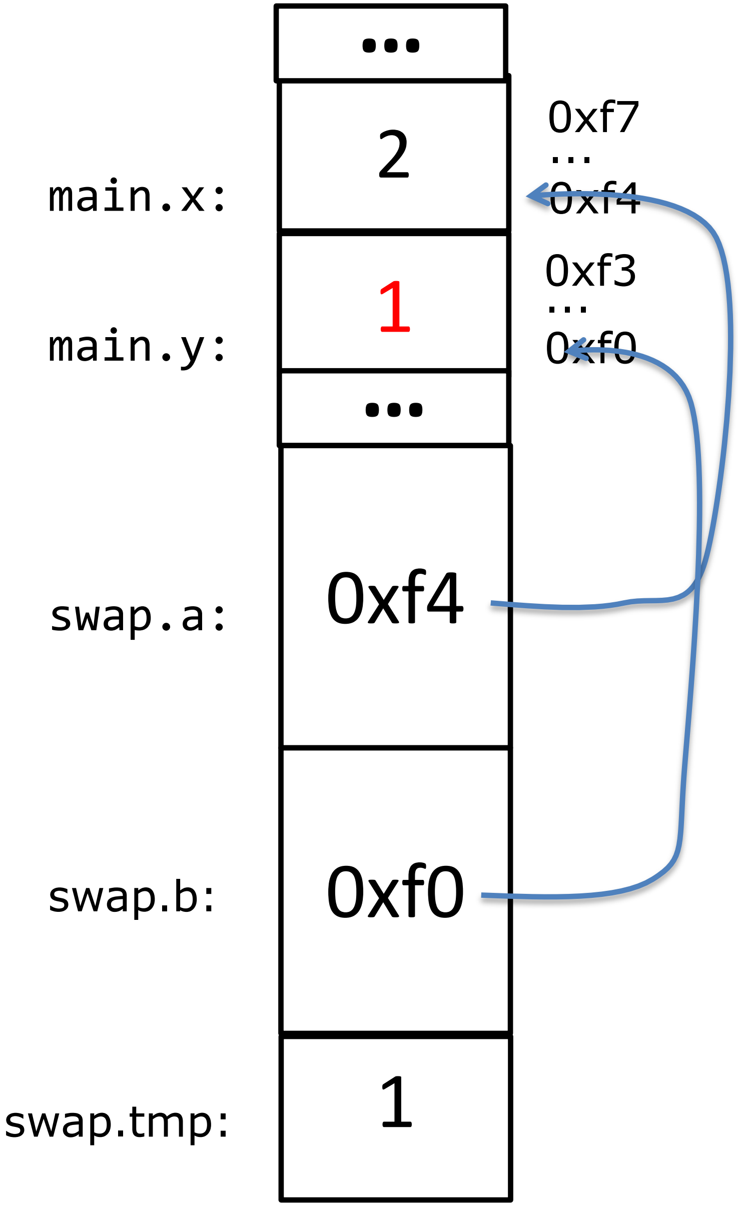
Pass pointers to function

```
void swap(int* x, int* y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf("x:%d, y:%d",x,y);
}
```



Pass pointers to function

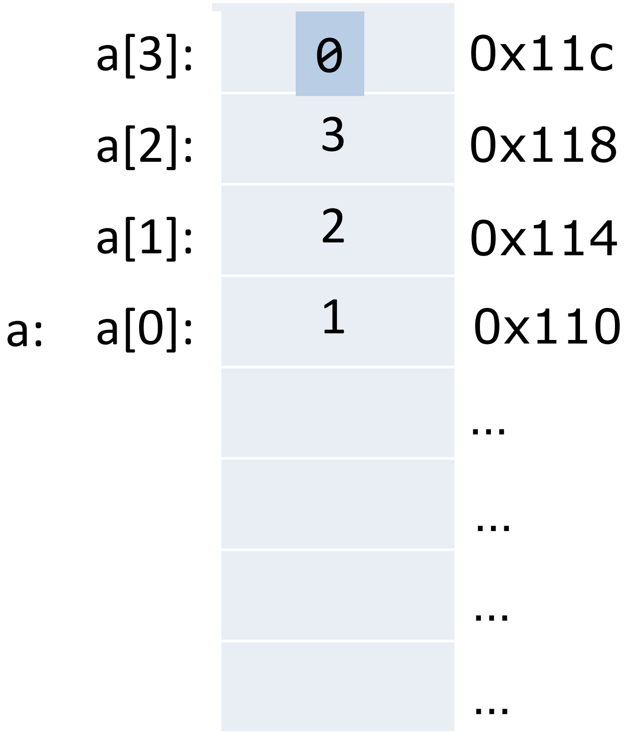
```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf(“x:%d, y:%d”,x,y);
}
```



Arrays

Array is a collection of contiguous objects with the same type

Array



a[3]: 0 0x11c

a[2]: 3 0x118

a[1]: 2 0x114

a: a[0]: 1 0x110

…

…

…

…

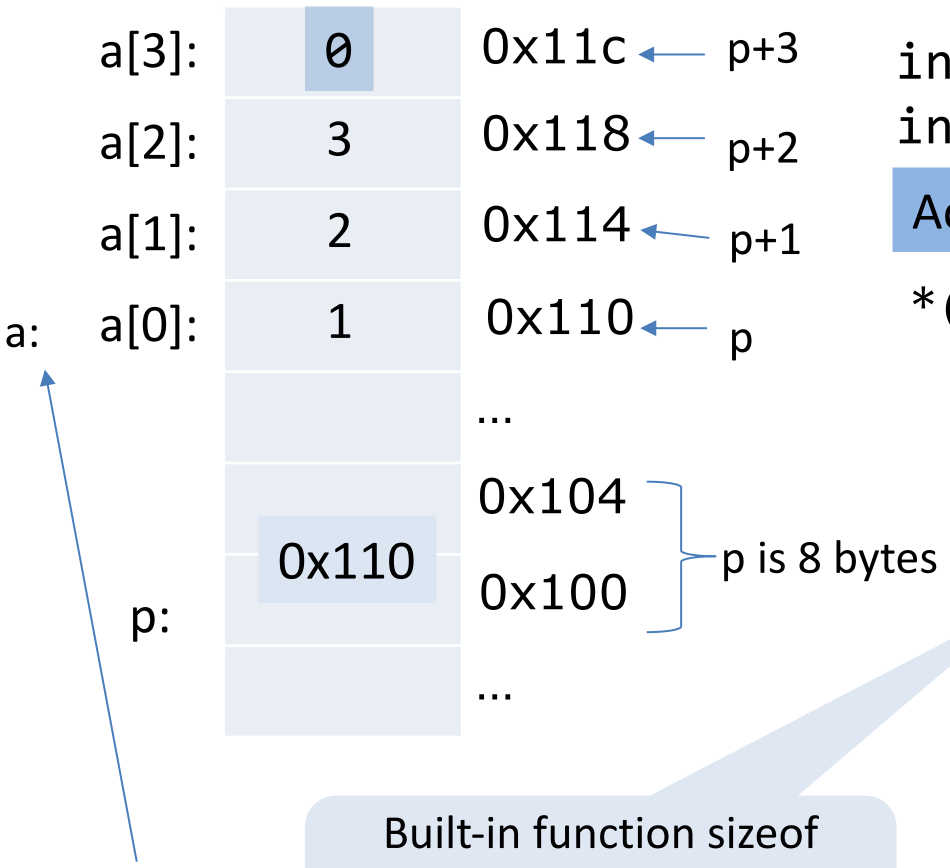
```
int a[4] = {1, 2, 3, 4};
```

Access method-1: use index

```
a[3] = 0;
```

🚫 There’s no meta-data (e.g. capacity, length) associated/stored with the array

Array access using pointer



	a[3]:	0	0x11c	p+3
	a[2]:	3	0x118	p+2
	a[1]:	2	0x114	p+1
a:	a[0]:	1	0x110	p
			…	
	p:	0x110	0x104	p is 8 bytes
			0x100	
			…	

```
int a[4] = {1, 2, 3, 4};
int *p = a;
```

Access Method-2: use pointer (arithmetic)

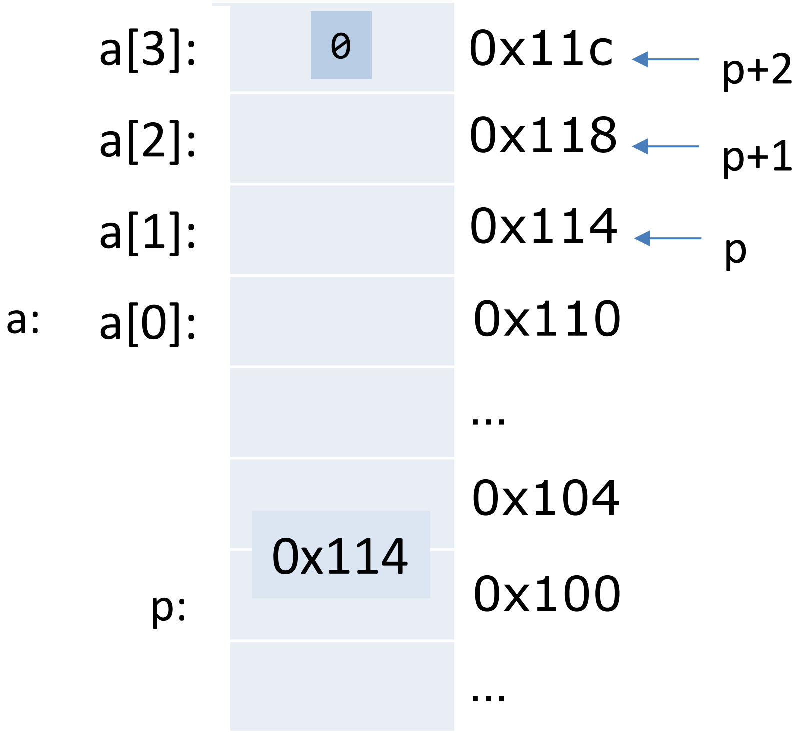
```
*(p+3)=0;
```

- `p+i` points to the i-th element after the one pointed to by p
  i.e. `p+i` is calculated as: `p's value + sizeof(*p) * i`
  `*(p+i)` is syntactically equivalent to `p[i]`
- `p-i` points to the i-th element before the one pointed to by p

Built-in function sizeof returns size (in bytes) of a given type or expression

a (array name) is aliased to be the memory address of the first element.
a is effectively a constant, not a variable, cannot be changed

Array access using pointer

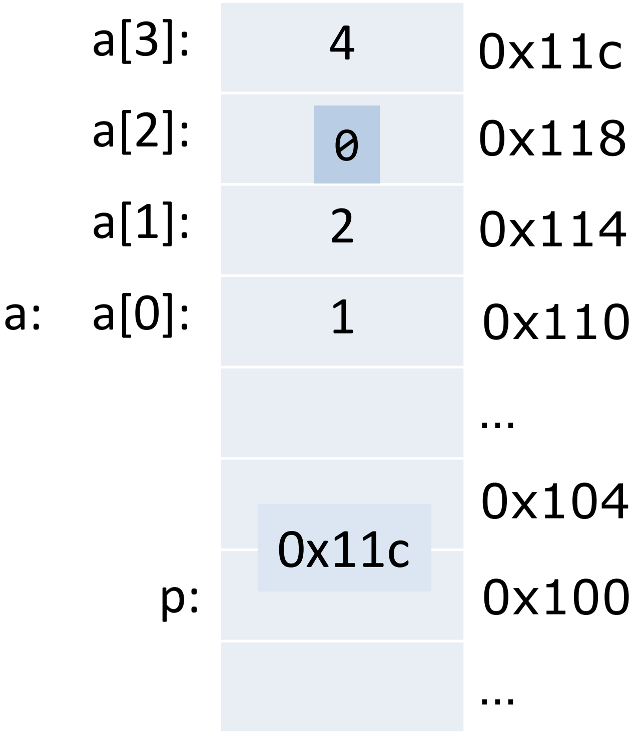


```
int a[4];
int *p = &a[1];
*(p+2)=0;
```

`&a[i]` is syntactically equivalent to: `a+i`

`*(p+i)` is syntactically equivalent to: `p[i]`

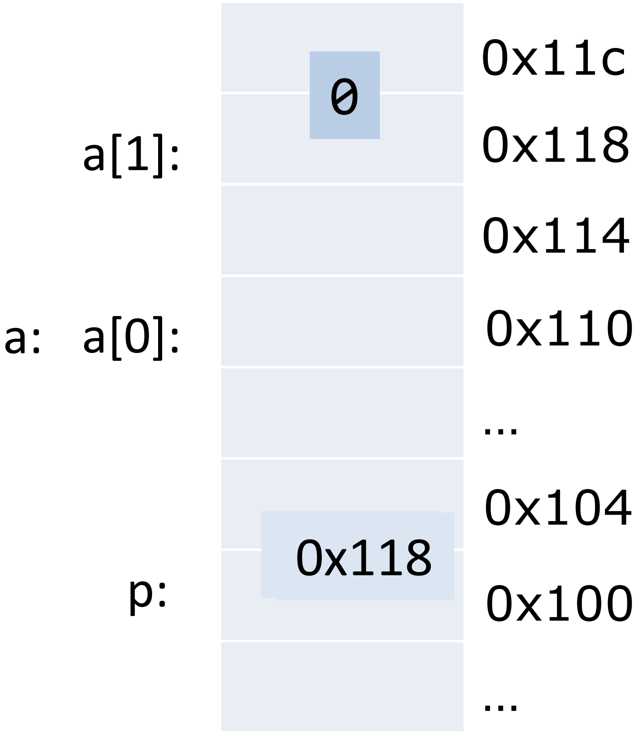
Array access using pointer



	a[3]:	4	0x11c
	a[2]:	0	0x118
	a[1]:	2	0x114
a:	a[0]:	1	0x110
			…
		0x11c	0x104
	p:		0x100
			…

```
int a[4] = {1, 2, 3, 4};
int *p = &a[3];
 p--;
 *p=0;
```

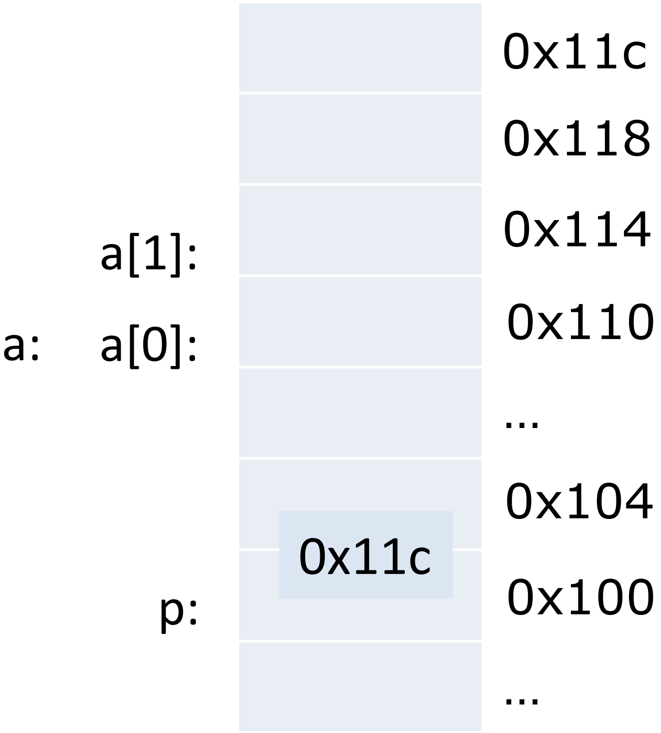
Array access using pointer



```
char *a[2];
char**    p = &a[0];
p++;
*p=NULL;
```

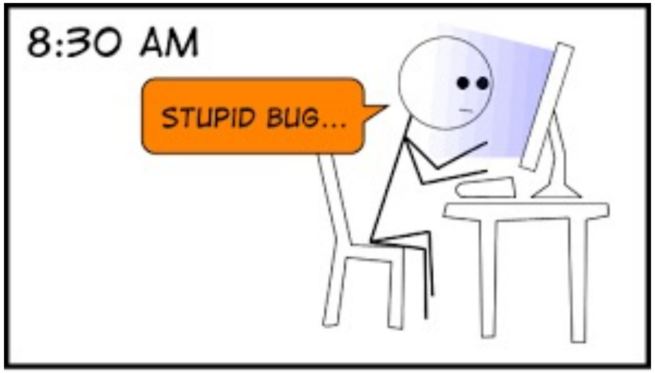
Equivalent to:
p = a

Out-of-bound access results in (potentially silent) memory error



			0x11c
			0x118
	a[1]:		0x114
a:	a[0]:		0x110
			…
		0x11c	0x104
	p:		0x100
			…

```
int a[2];
int *p = a;
 p += 3;
*p=0;
```



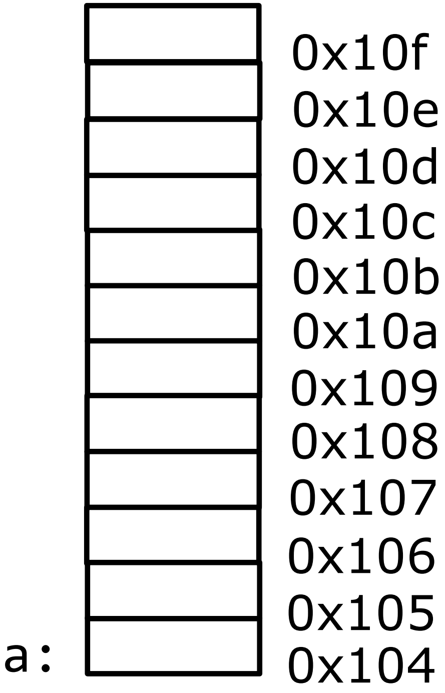
Pass array to function via pointer

```
// multiply every array element by 2
void multiply2(int *a) {
    for (int i = 0; i < ???; i++) {
        a[i] *= 2;
    }
}

int main() {
    int a[2] = {1, 2};
    multiply2(a);
    for (int i = 0; i < 2; i++) {
        printf(“a[%d]=%d”, i, a[i]);
    }
}
```

Pass array to function via pointer

```
// multiply every array element by 2
void multiply2(int *a, int n) {
    for (int i = 0; i < n; i++) {
        a[i] *= 2; // (*(a+i)) *= 2;
    }
}

int main() {
    int a[2] = {1, 2};
    multiply2(a, 2);
    for (int i = 0; i < 2; i++) {
        printf(“a[%d]=%d”, i, a[i]);
    }
}
```

Pointer casting



a:

0x10f
0x10e
0x10d
0x10c
0x10b
0x10a
0x109
0x108
0x107
0x106
0x105
0x104

```
int a = 0x12345678;
char *cp = (char *)&a;

// What is *cp?

cp++;
// What is *cp?
```

Assume 64-bit small endian machine

Another example use of pointer casting

```
unsigned int extract_float_bit_pattern(float f)
{

    unsigned int i = *(unsigned int *)&f;
    return i;


}
```

Summary

- Pointers are memory addresses
  - p =&x; (p has address of variable x)
  - *p … (refers to the variable pointed to by p)
- Arrays:
  - No array meta-data associated/stored. No bound checking
  - equivalence of pointer arithmetic and array access
    - `p+i` same as `&p[i]`
    - `*(p+i)` same as `p[i]`
    - Value of `p+i` is computed as `p+sizeof(*p)*I`
- Pass pointers to functions
- Pointer casting