C - Functions, Pointers, Arrays

Jinyang Li

based on the slides of Tiger Wang

Today's lecture

- Pointers
- Array and its relationship to pointer
- Pointer casting
- 2D array

Functions

C program consists of functions (aka subroutines, procedures)

Why breaking code into functions?

- Readability
- Reusability

Advice from Linus

- Functions should be short and sweet, and do just one thing.
- The maximum length of a function is inversely proportional to the complexity of that function.
  - For complex tasks, break it up into pieces and use helper functions with descriptive names.
- How to measure complexity?
  - Indentation level
  - # of local variables in a function should not exceed 5-10

Google: Linux kernel coding style

Local Variables

Scope

– within which the variable can be used

```
int add(int a, int b)
{
    int r = a + b;
    return r;
}
```

r's scope is in function *add*

Local Variables / function arguments

Scope (within which the variable can be used)

- Within the function it is declared in
- local variables of the same name in different functions are unrelated

Storage:

- allocated upon function invocation
- de-allocated upon function return

```
int add(int a, int b)
{
    int r = a + b;
    return r;
}
```

```
int subtract(int a, int b)
{
    int r = a - b;
    return r;
}
```

Global Variables

- Scope
  - Can be accessed by all functions
- Storage
  - Allocated upon program start, deallocated when entire program exits

```
int r = 0;
```

```
int add(int a, int b)
{
    r = a + b;
    return r;
}
```

modifies global variable r

```
int subtract(int a, int b)
{
    int r = a - b;
    return r;
}
```

local variable r shadows global variable r

Function invocation

C (and Java) passes arguments by value

```
int main()
{
    int x = 1;
    int y = 2;
    swap(x, y);

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

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Result x: ?, y: ?

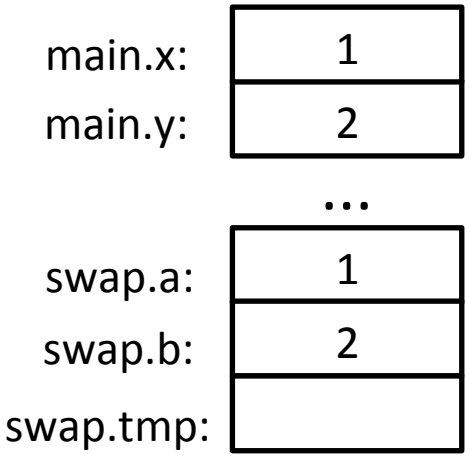
Function invocation

C passes the arguments by value

```
int main()
{
    int x = 1;
    int y = 2;
    swap(x, y);

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

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Result x: 1, y: 2



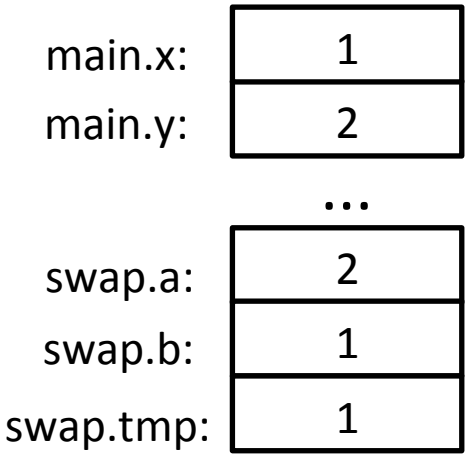
Function invocation

C passes the arguments by value

```
int main()
{
    int x = 1;
    int y = 2;
    swap(x, y);

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

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

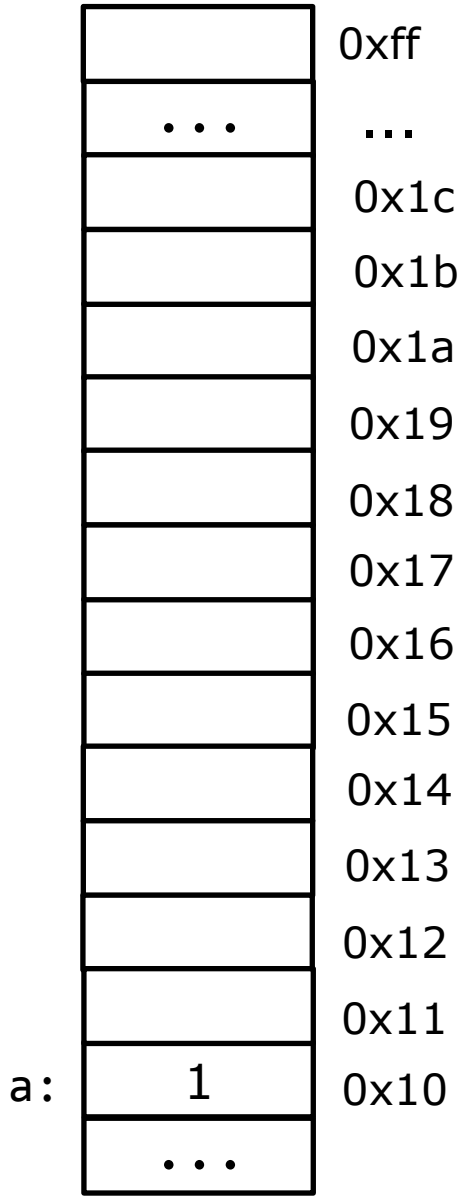
Result x: 1, y: 2



Pointers

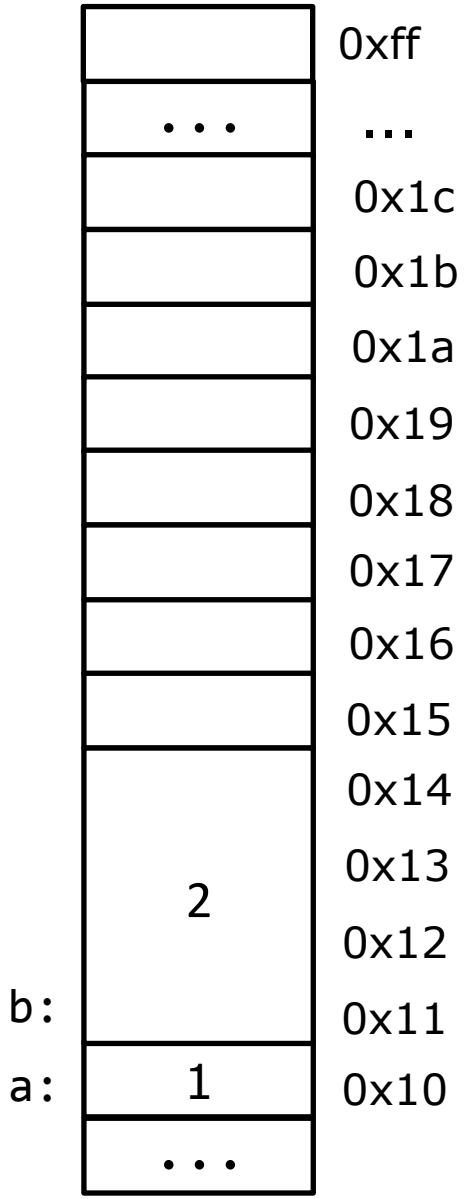
Pointer is a memory address

Pointer



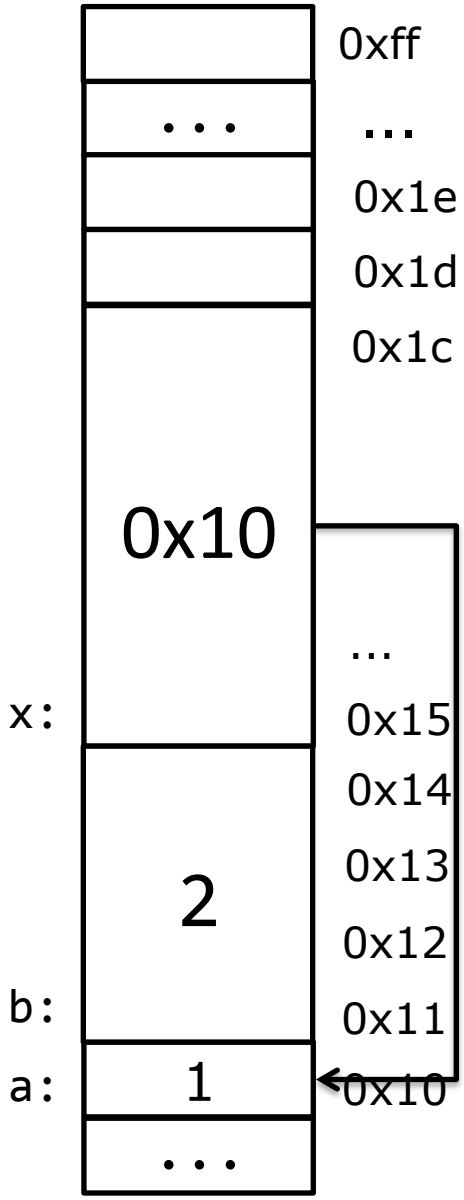

```
char a = 1;
```

Pointer




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

Pointer




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

& gives address of variable

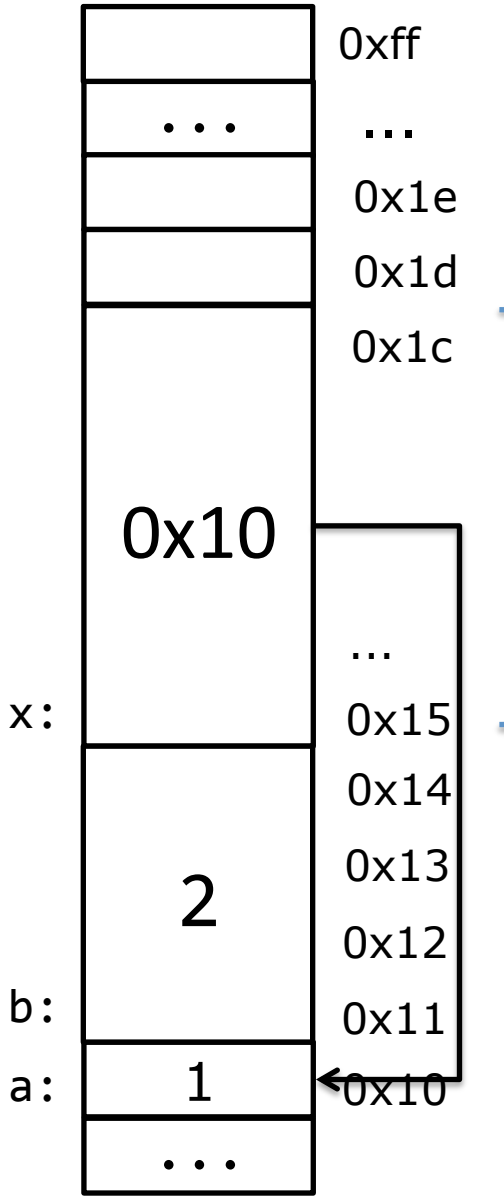
equivalent to:
char *x;
x = &a;

equivalent to:
char* x;
x = &a;

what happens if I write
char x = &a;
or
int *x = &a;

type mismatch!

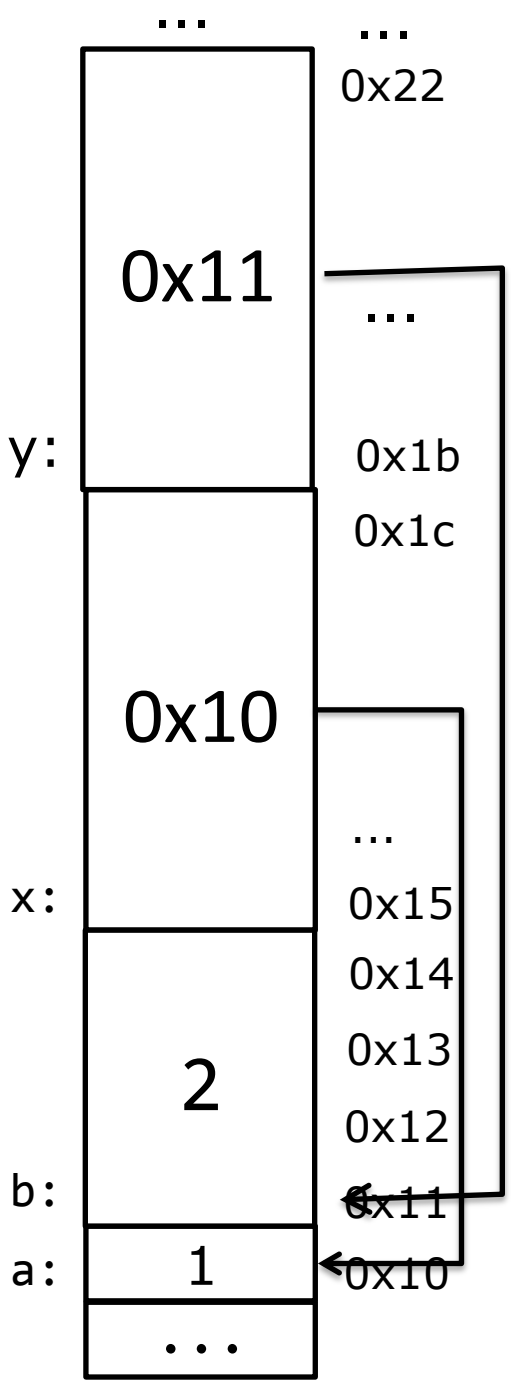
Pointer




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

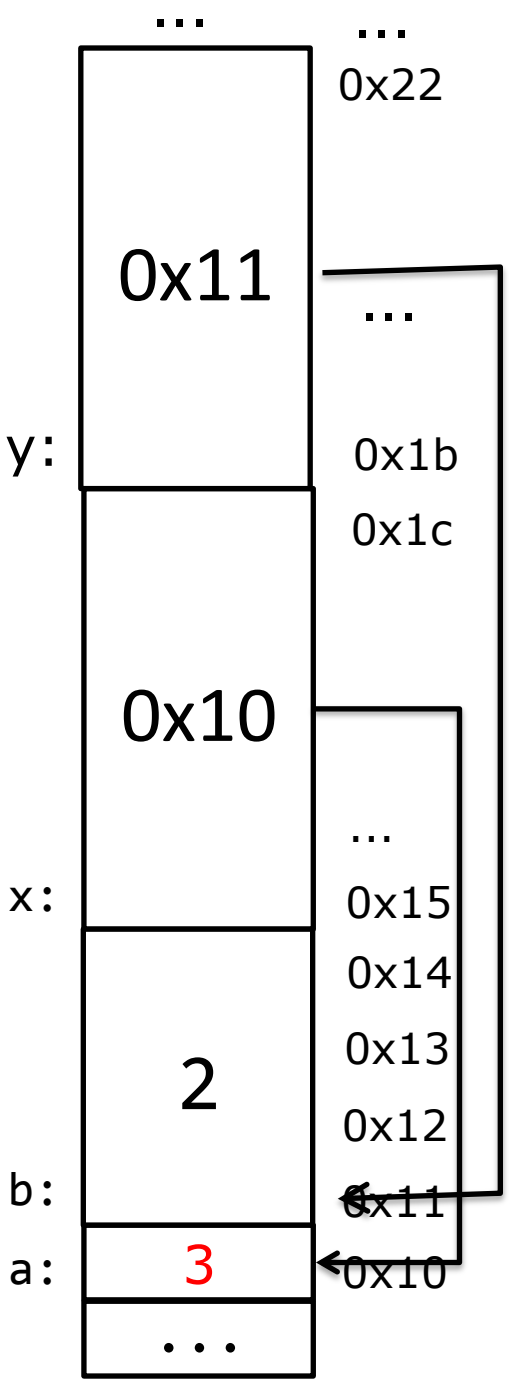
Size of pointer on a 64-bit machine?

8 bytes



Pointer

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



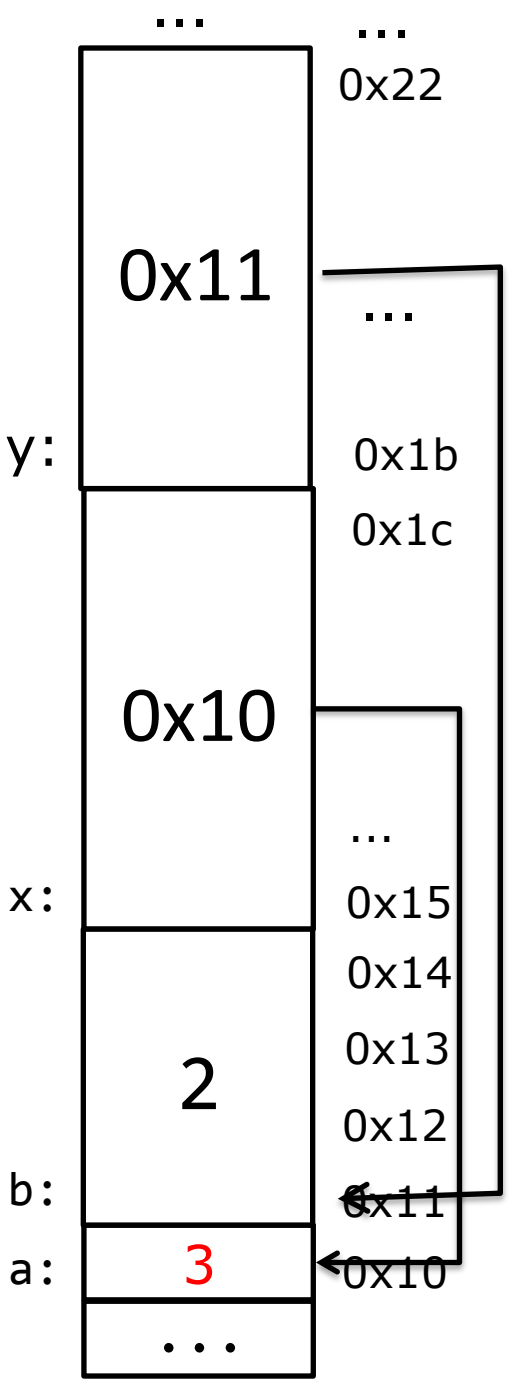
Pointer

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

```
*x = 3;
```

* operator dereferences a pointer

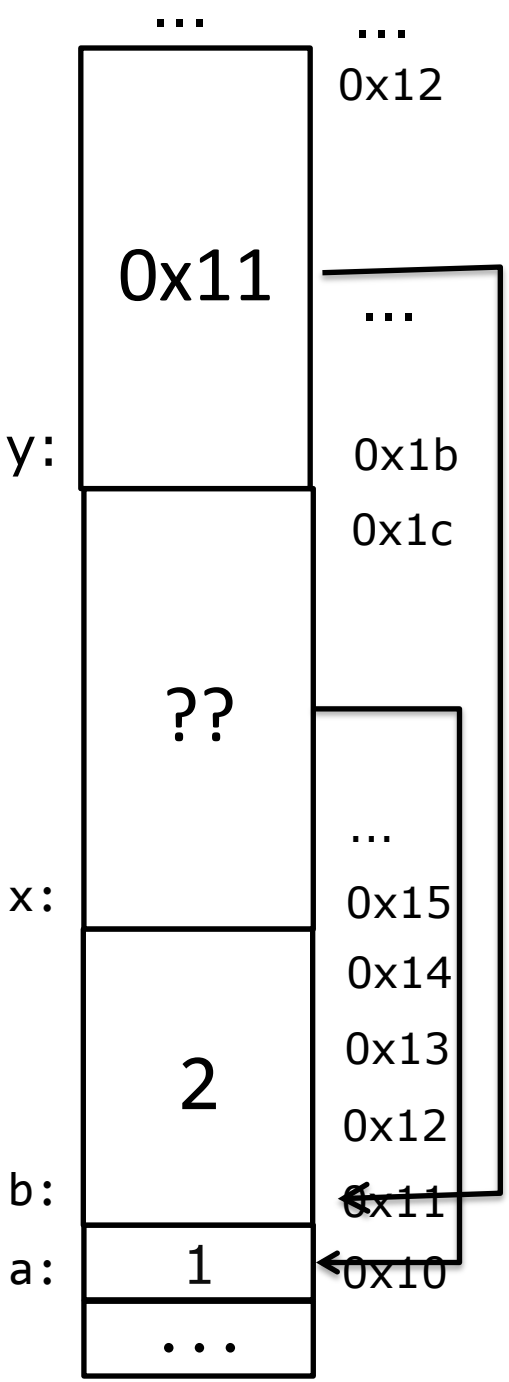
Value of variable a after this statement?



Pointer

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


 *x = 3;
 // value of variable a?
 //printf("a=%d\n", a);
```

Pointer

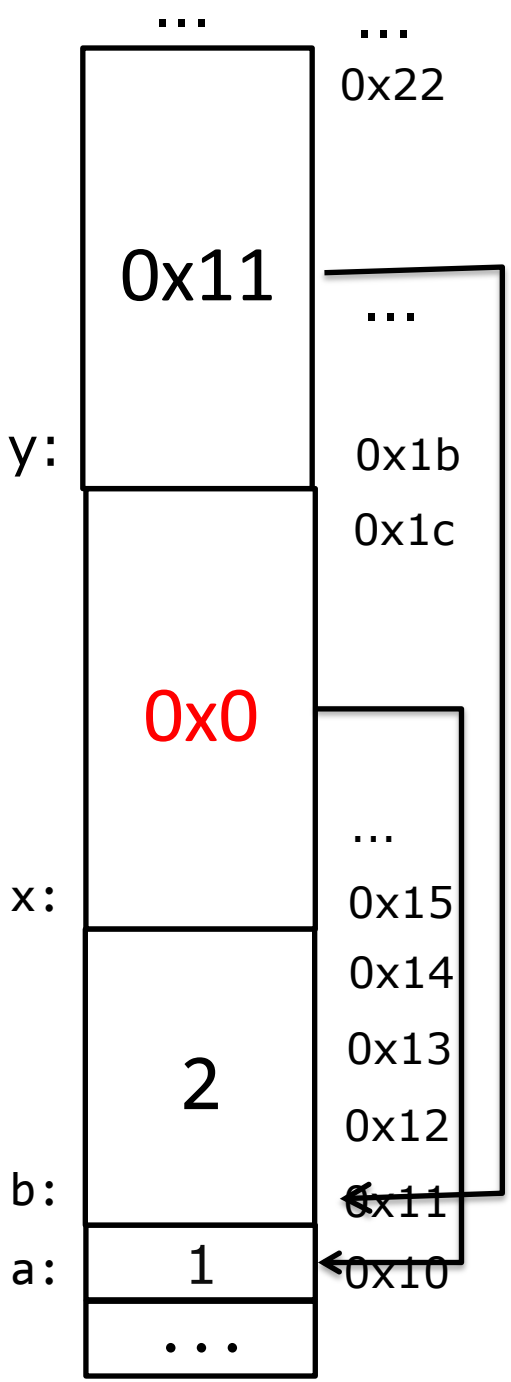



```
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;
```

what if x is uninitialized?

```
*x = 3;
```

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



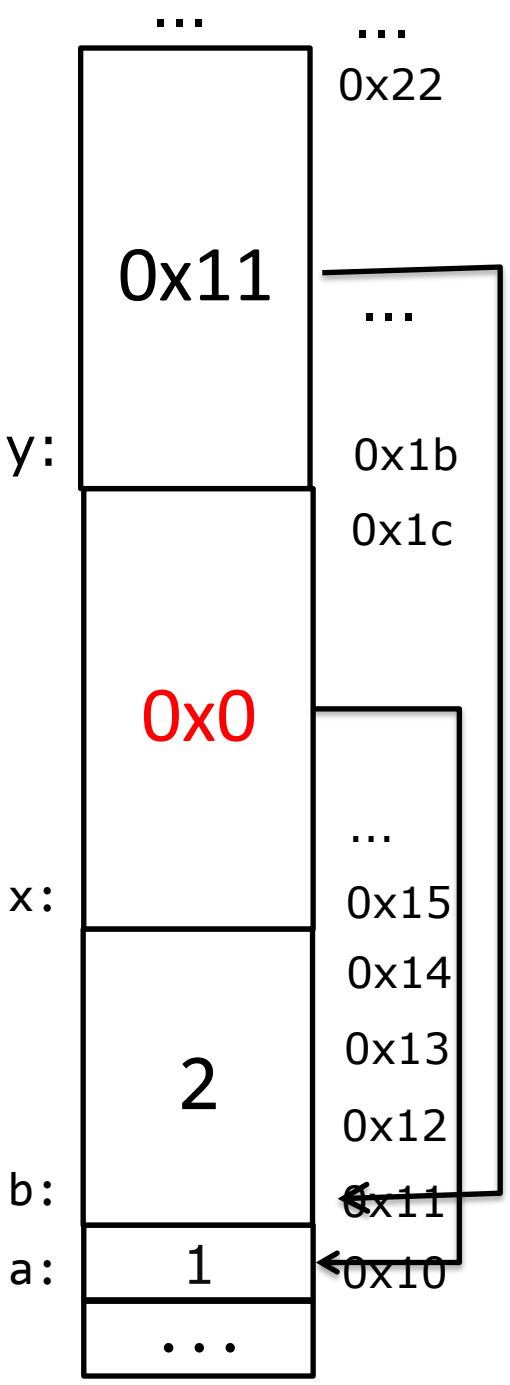
Pointer

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

Always initialize pointers!

```
*x = 3;
```

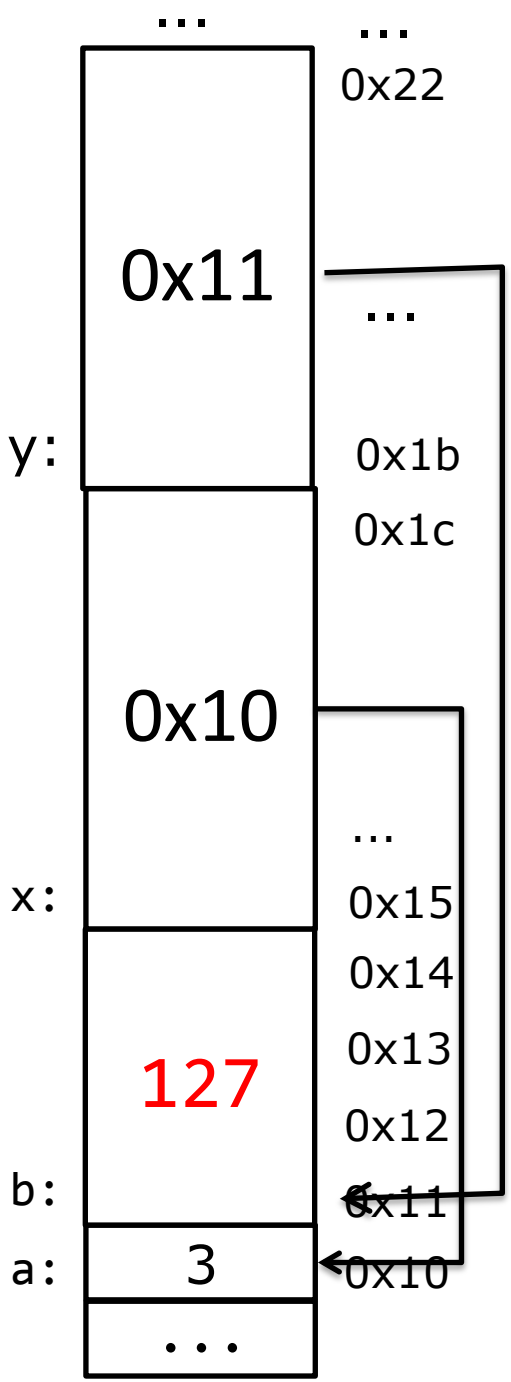
Dereferencing NULL pointer definitely results in "Segmentation fault"



Pointer

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

*x = 3;
```

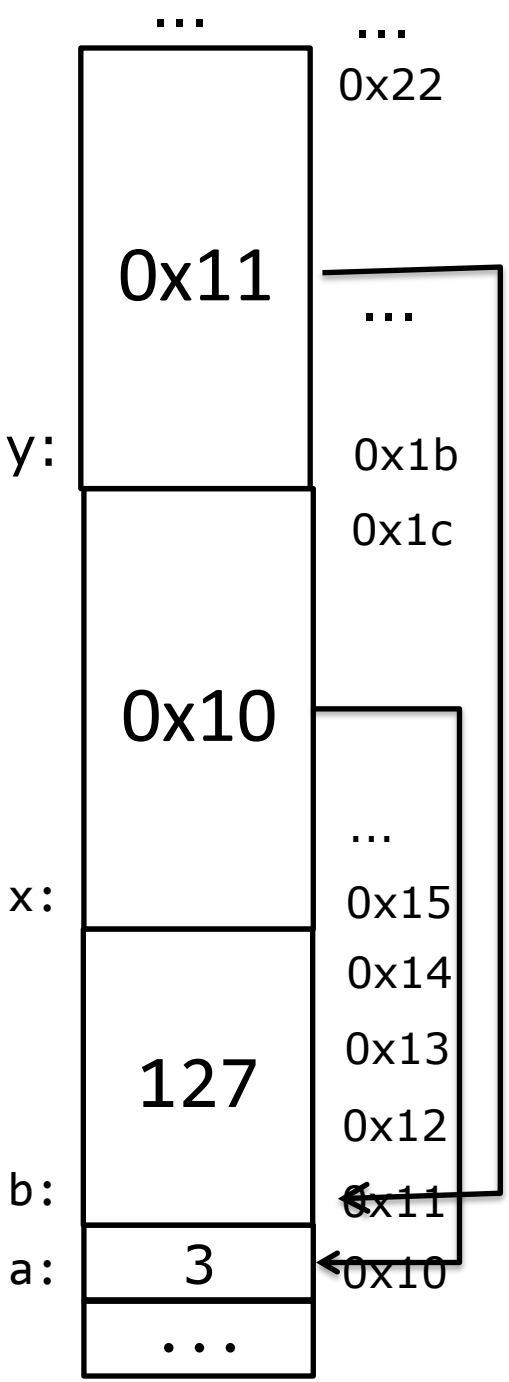
```
(gdb) r
Starting program: /oldhome/jinyang/a.out

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



Pointer

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

*x = 3;
*y = 127;
```

Pointer



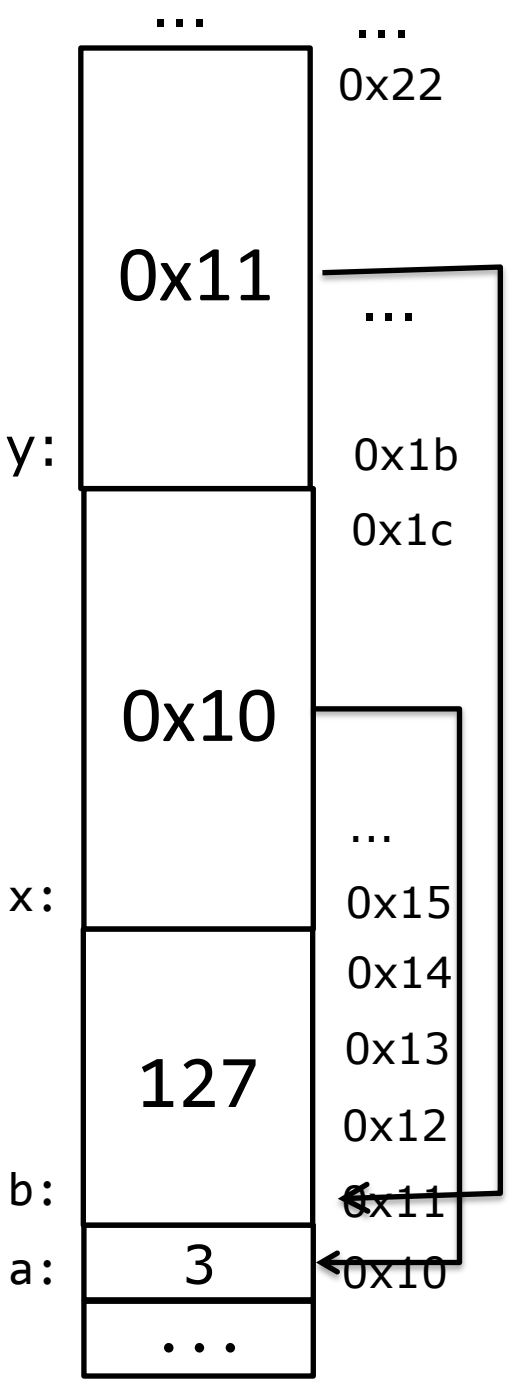

```
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

*x = 3;
*y = 127;

char **xx = &x;
```

equivalent to
char **xx;
xx = &x;

equivalent to
char** x x;
xx = &x;

what happens if I write
char* xx;
xx = &x;

value of xx?
printf("xx=%p", xx); xx=0x15



Pointer

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

*x = 3;
*y = 127;

char **xx = &x;
int **yy = &y;
```

value of yy?
printf("yy=%p", yy); yy=0x1b

Common confusions on *

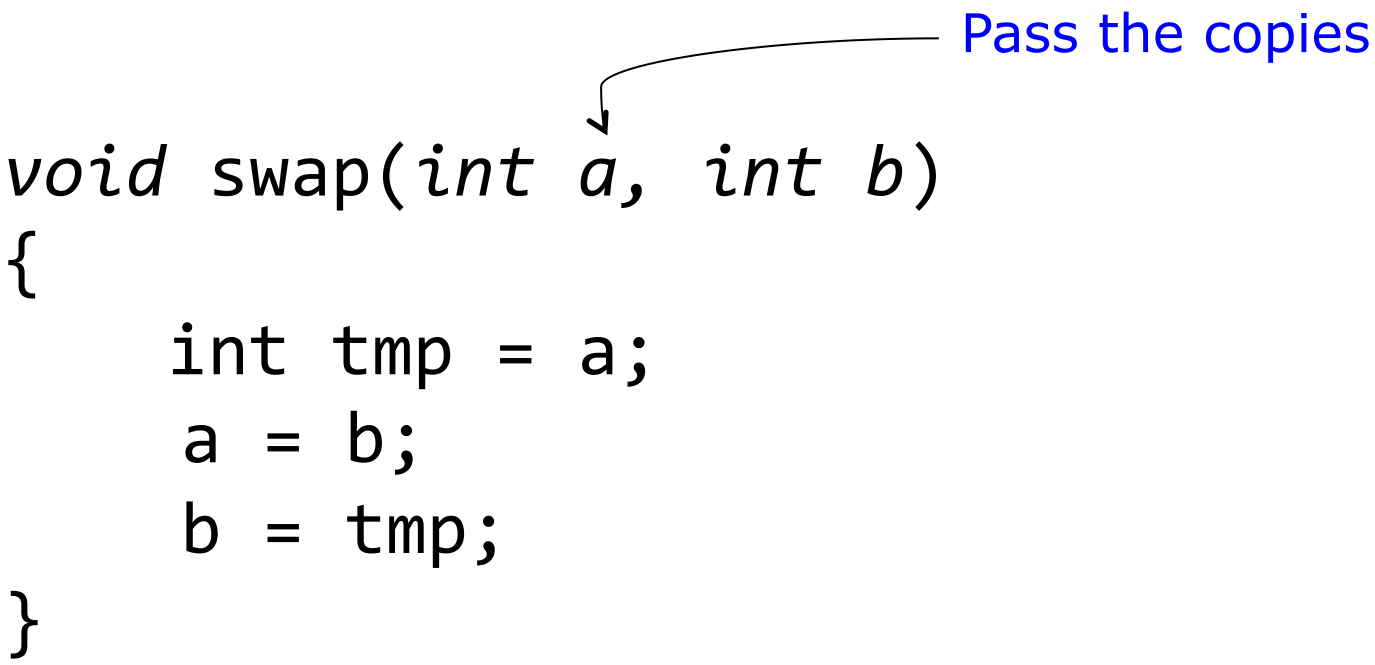
* has two meanings!!
  1. part of a pointer type name, e.g. char *, char **, int *
  2. the deference operator.

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

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

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

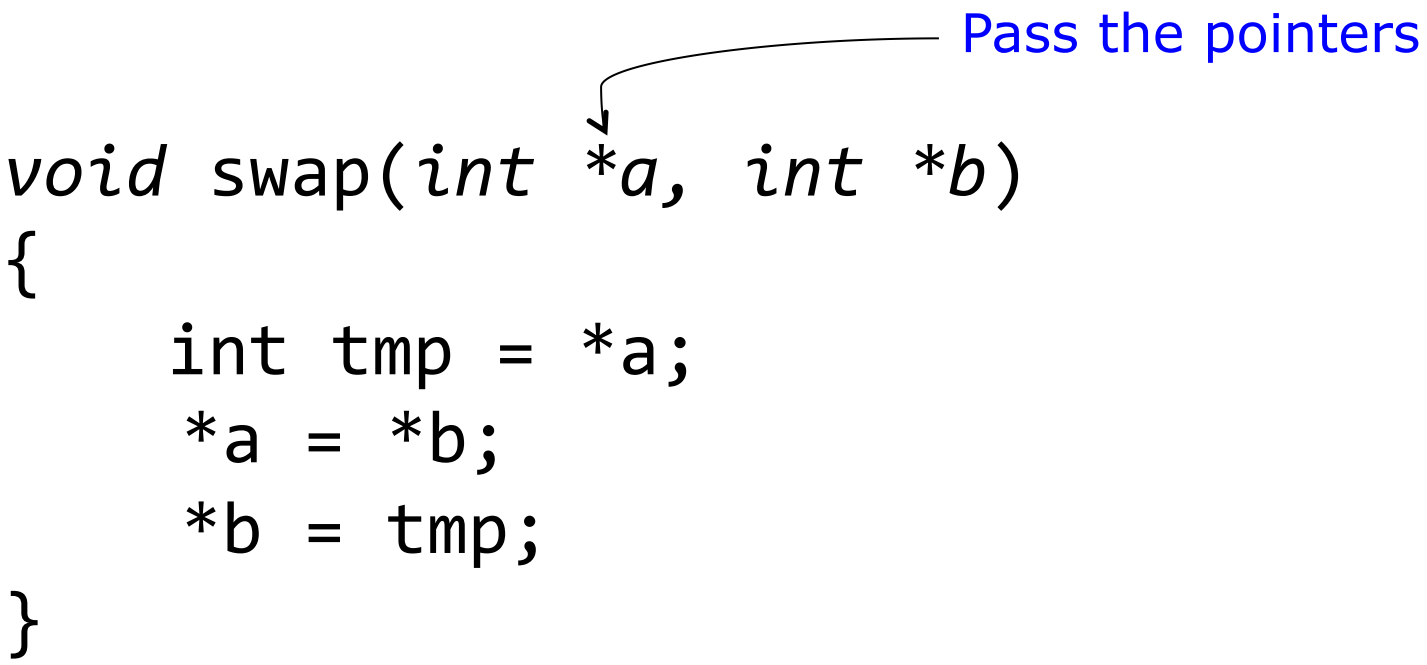
C's syntax for declaring multiple pointer variables on one line
char* b, c; does not work

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

Pass pointers to function

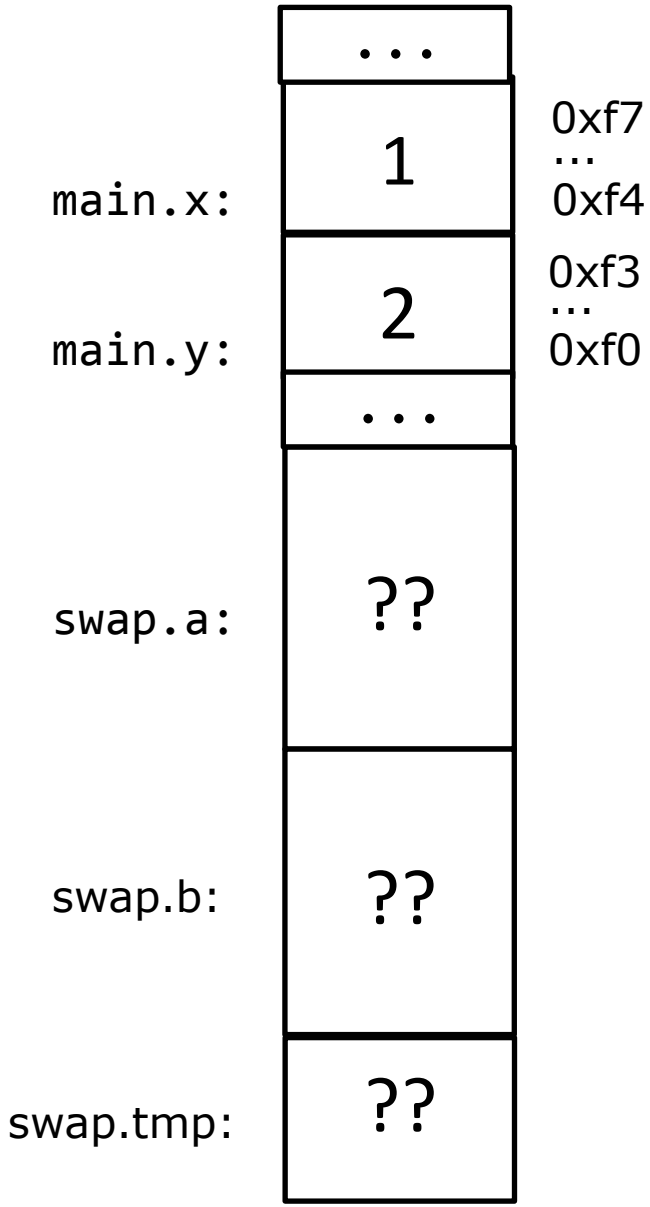


Pass pointers to function

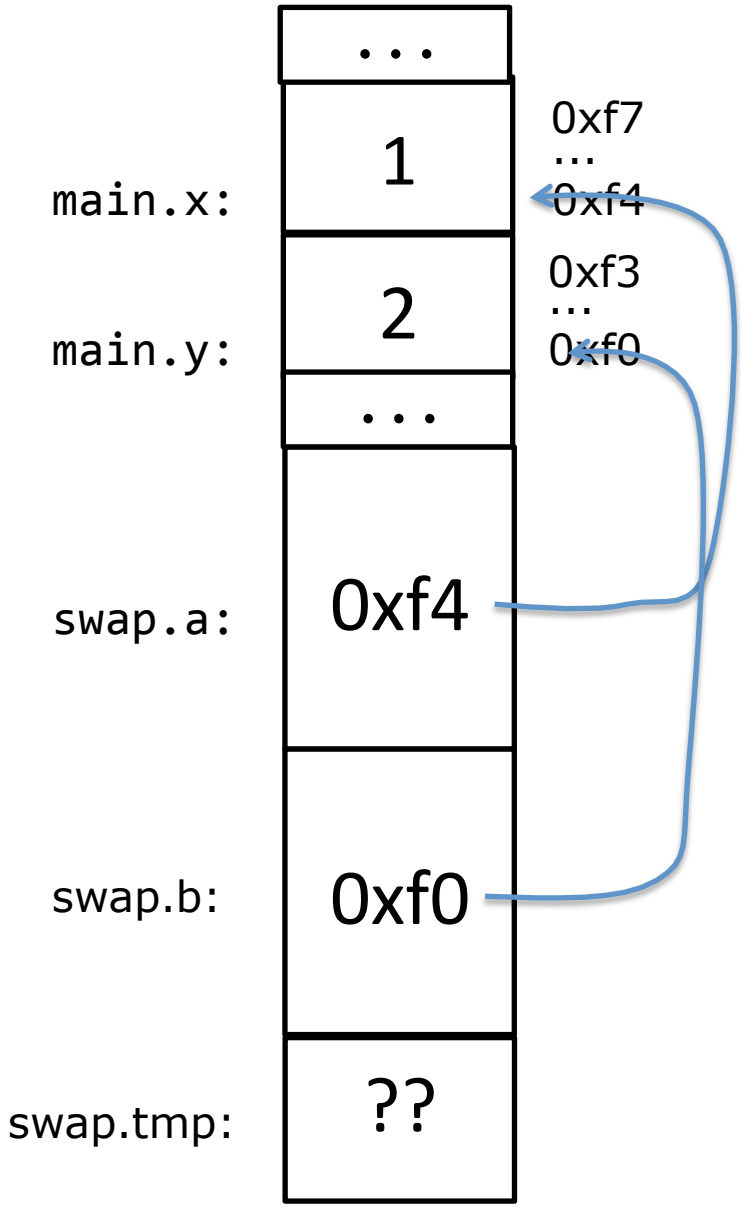


```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

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

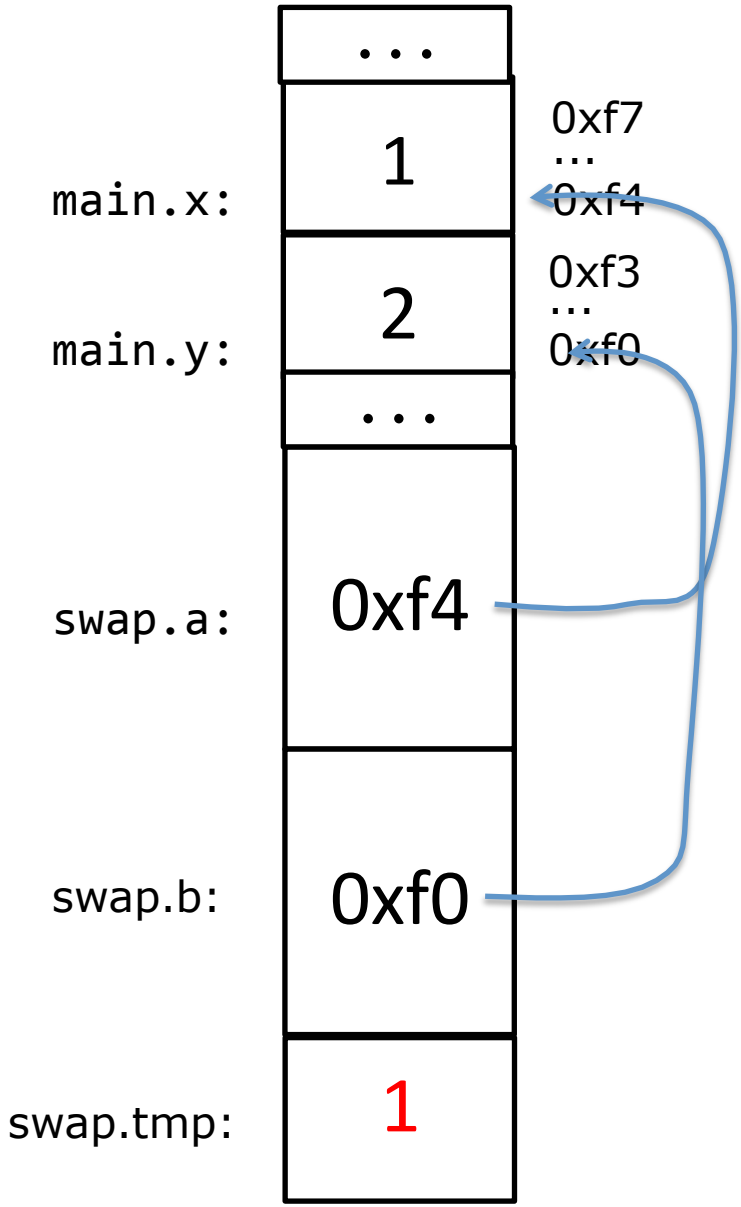
Size and value of a, b, tmp upon function entrance?



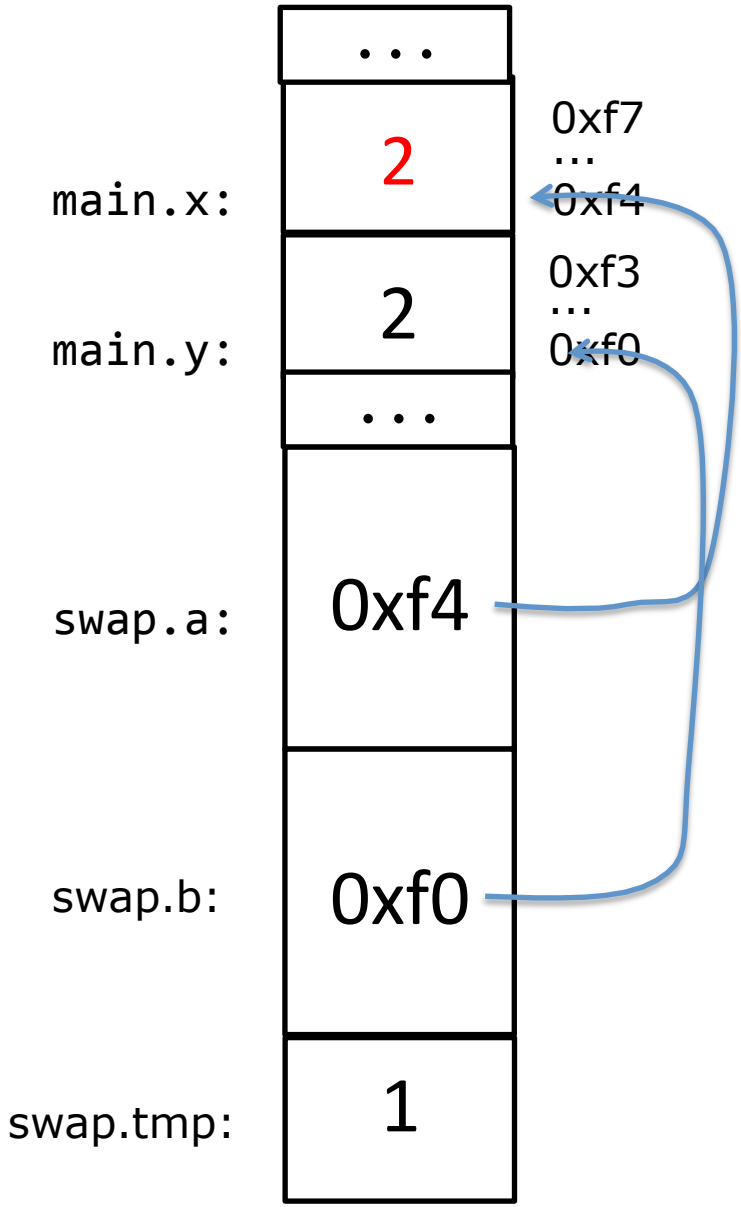
```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

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



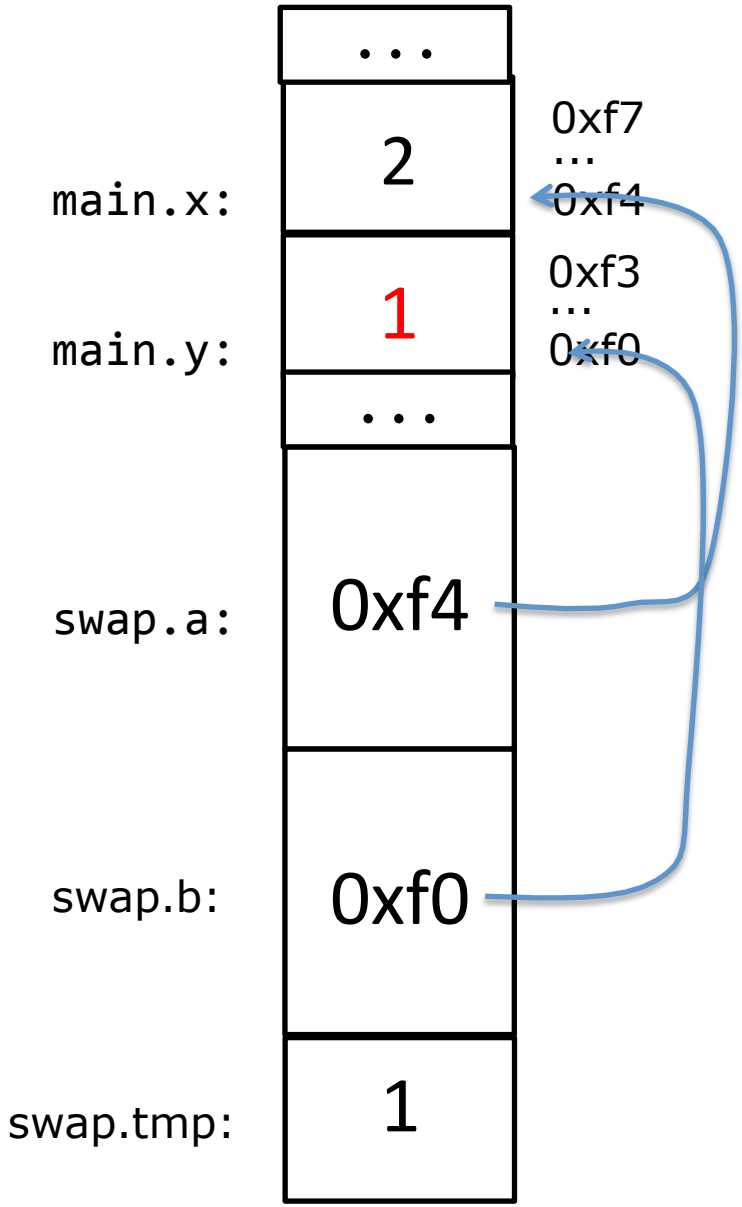
```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

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



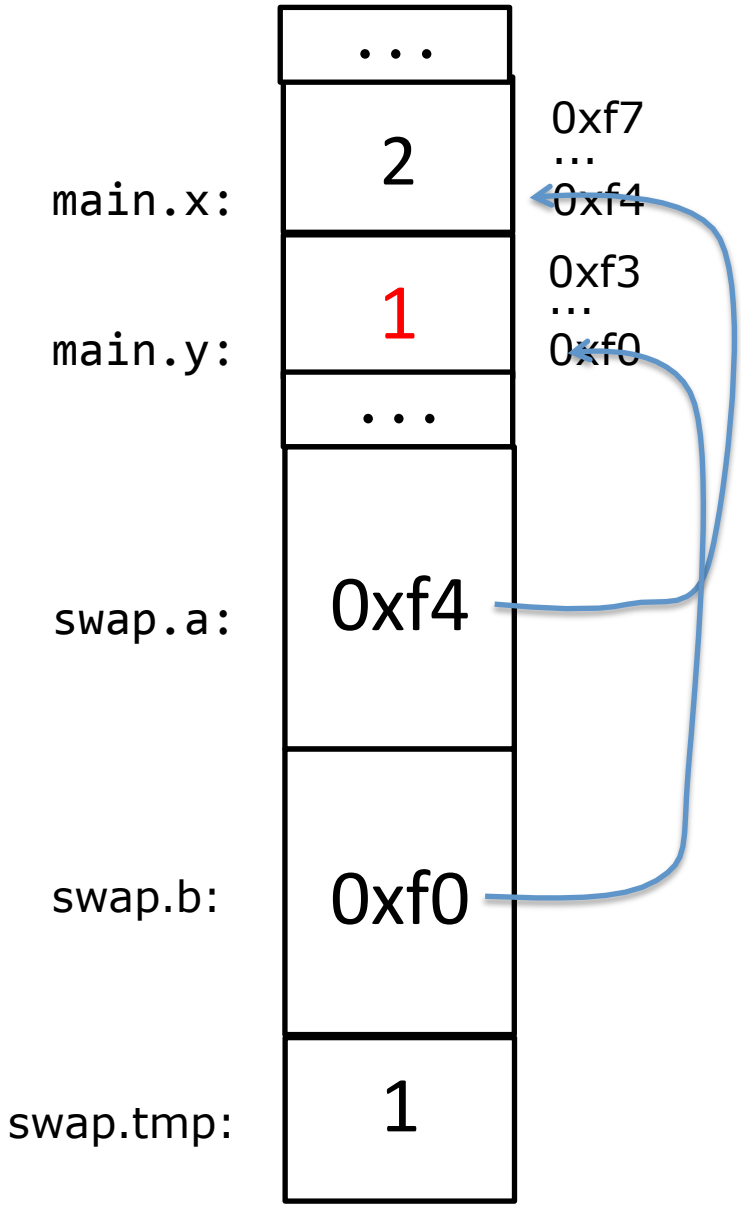
```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

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



```
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);
}
```



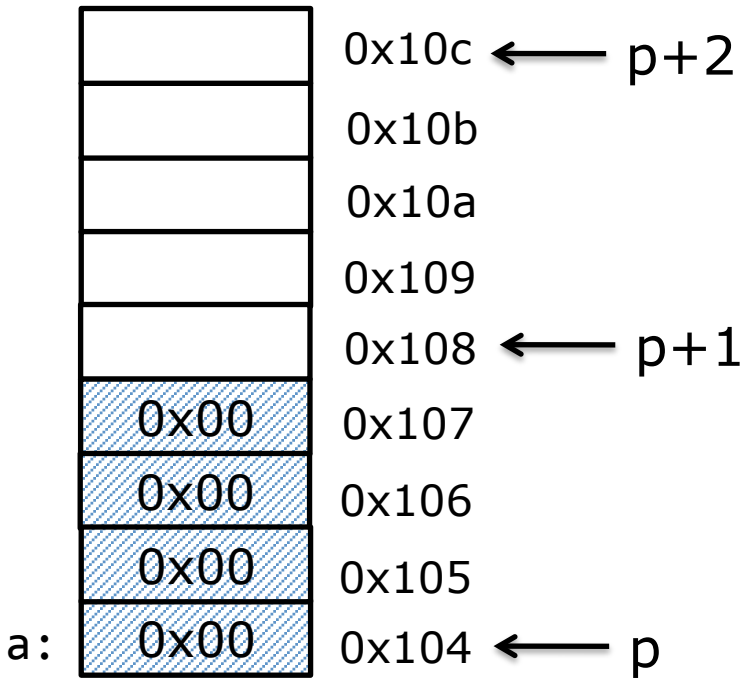
```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

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



Pointer arithmetic

```
int a = 0;
int *p = &a;   // assume the address of variable a is 0x104
```

p+1	Point to the next object with type int (4 bytes after current object of address p)	???



Pointer arithmetic

```
int a = 0;
int *p = &a;   // assume the address of variable a is 0x104
```

`p+i`	Point to the `i`th object of type int after object with address p	0x104 + i*4
`p-i`	Point to the ith object with int before object with address p	0x104 – i*4

Pointer arithmetic

```
short a = 0;
short *p = &a; // assume the address of variable a is 0x104
```

`p+i`	Point to the `i`th object with type short after object with address p	???
`p-i`	Point to the ith object with type short before object with address p	???

Pointer arithmetic

```
short a = 0;
short *p = &a; // assume the address of variable a is 0x104
```

`p+i`	Point to the `i`th object with type short after object with address p	0x104 + i*2
`p-i`	Point to the ith object with type short before object with address p	0x104 - i*2

Pointer arithmetic

```
char *a = NULL;
char **p = &a; // assume the address of variable a is 0x104
```

`p+i`	Point to the `i`th object with type char * after object with address p	???
`p-i`	Point to the ith object with type char * before object with address p	???

Pointer arithmetic

```
char *a = NULL;
char **p = &a; // assume the address of variable a is 0x104
```

`p+i`	Point to the `i`th object with type char * after object with address p	0x104 + i*8
`p-i`	Point to the ith object with type char * before object with address p	0x104 – i*8

Arrays

Array is a collection of contiguous objects with the same type

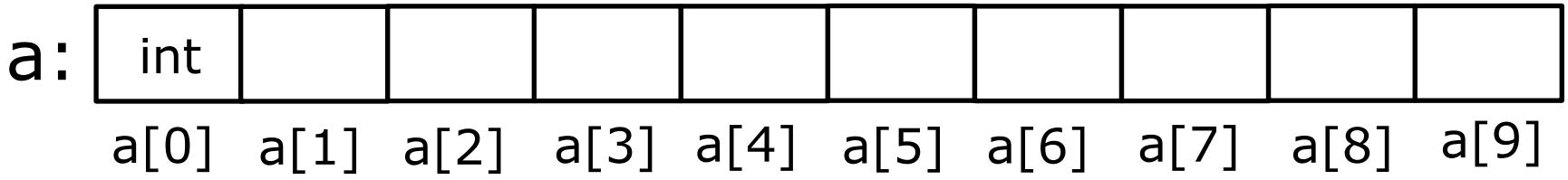
Array

Strong relationship with pointer

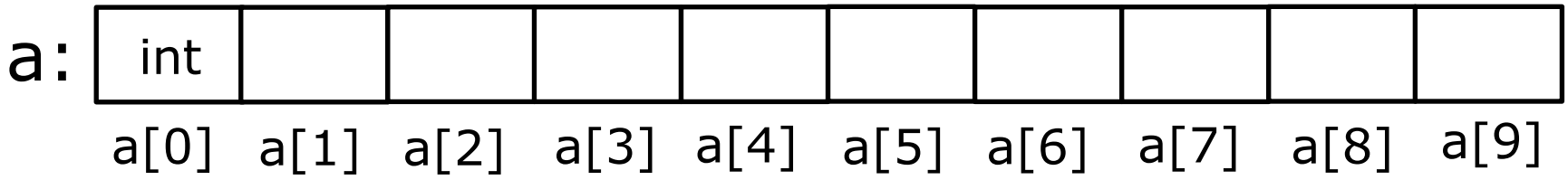
- array access can be done using pointers.

A block of n consecutive objects.

- int a[10];

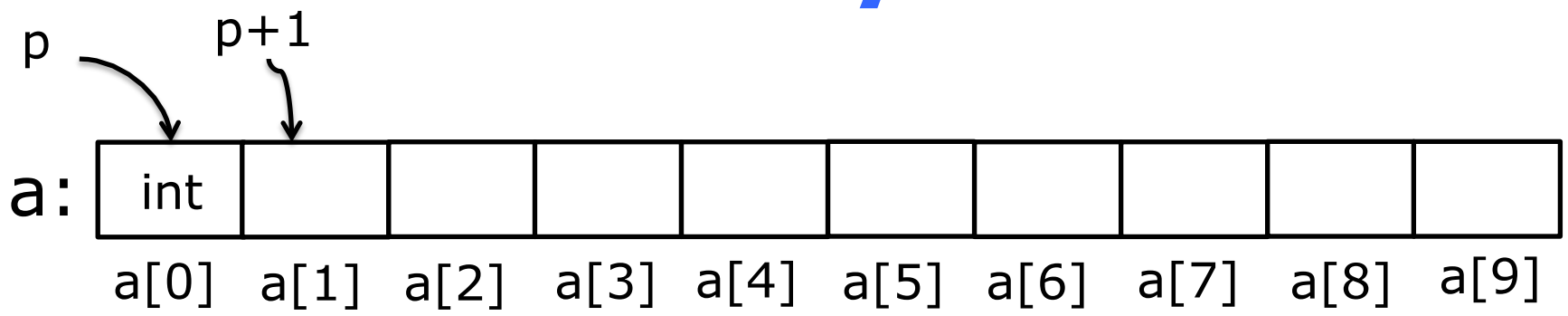


Array



length of a[0]: 4 bytes → a[1] is 4 bytes next to a[0]

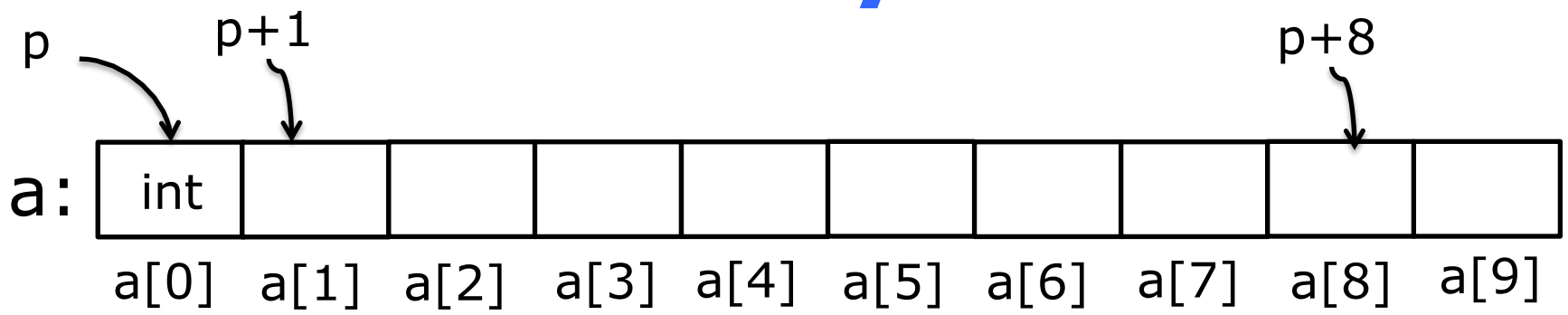
Array



length of a[0]: 4 bytes → a[1] is 4 bytes next to a[0]

int *p = &a[0] → p+1 points to a[1]

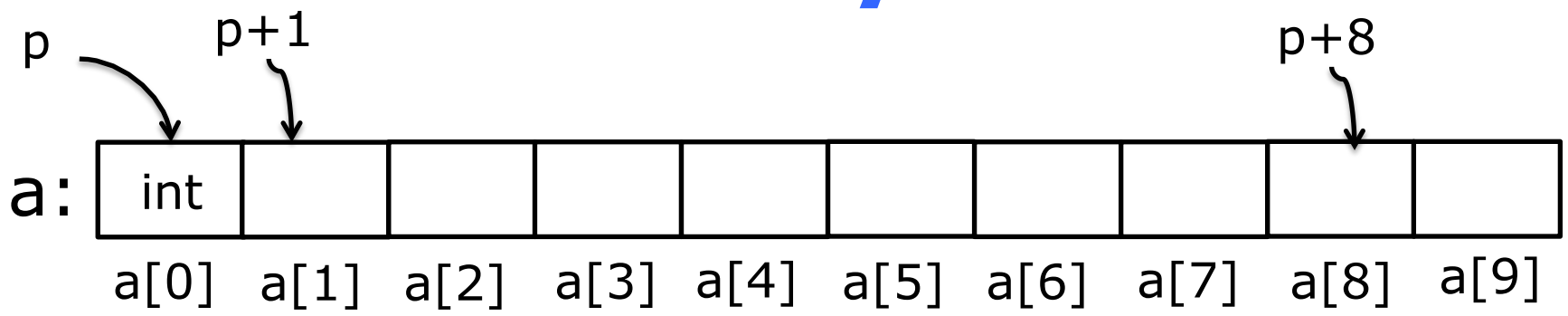
Array



length of a[0]: 4 bytes → a[1] is 4 bytes next to a[0]

int *p = &a[0] → p+1 points to a[1]
→ p + i points to a[i]

Array

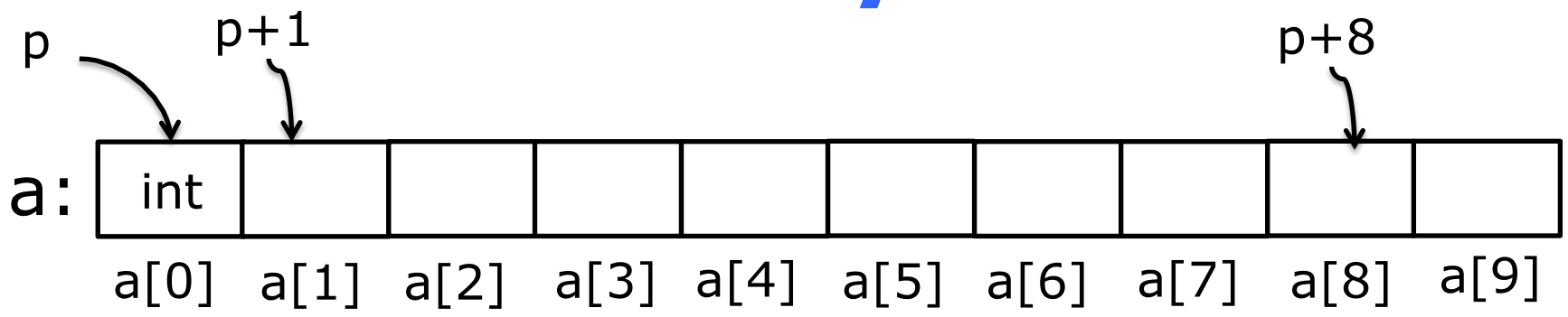


length of a[0]: 4 bytes → a[1] is 4 bytes next to a[0]

int *p = &a[0] → p+1 points to a[1]
→ p + i points to a[i]

int *p = a ⟷ int *p = &a[0]

Array



length of a[0]: 4 bytes → a[1] is 4 bytes next to a[0]

int *p = &a[0] → p+1 points to a[1]
→ p + i points to a[i]

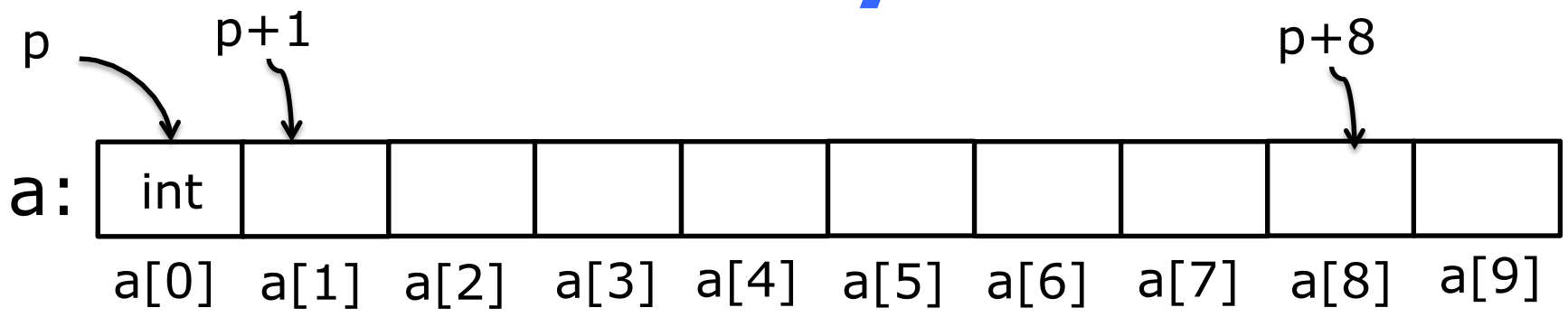
int *p = a ⟷ int *p = &a[0]

p++ ✓

a++ ✗ compilation error

p = &a ✗

Array



length of a[0]: 4 bytes → a[1] is 4 bytes next to a[0]

int *p = &a[0] → p + 1 points to a[1]
→ p + i points to a[i]

int *p = a ⟷ int *p = &a[0]

*(p+1) ⟷ p[1]

*(p + i) ⟷ p[i]

Example

```
#include <stdio.h>

int main() {
  int a[3] = {100, 200, 300};
  int *p = a;
  *p = 400;
  for (int i=0; i<3; i++) {
    printf("%d ", a[i]);
  }
  printf("\n");
}
```

equivalent to p[0] = 400;

What if change to: *(p+1) = 400;

Output: 100 400 300

Output? 400 200 300

Another Example

```
#include <stdio.h>

int main() {
  int a[3] = {100, 200, 300};
  int *p = a;
  p++;
  *p = 400;
  for (int i=0; i<3; i++) {
    printf("%d ", a[i]);
  }
  printf("\n");
}
```

equivalent to
*(++p) = 400;

Output? 100 400 300

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]);
    }
}
```

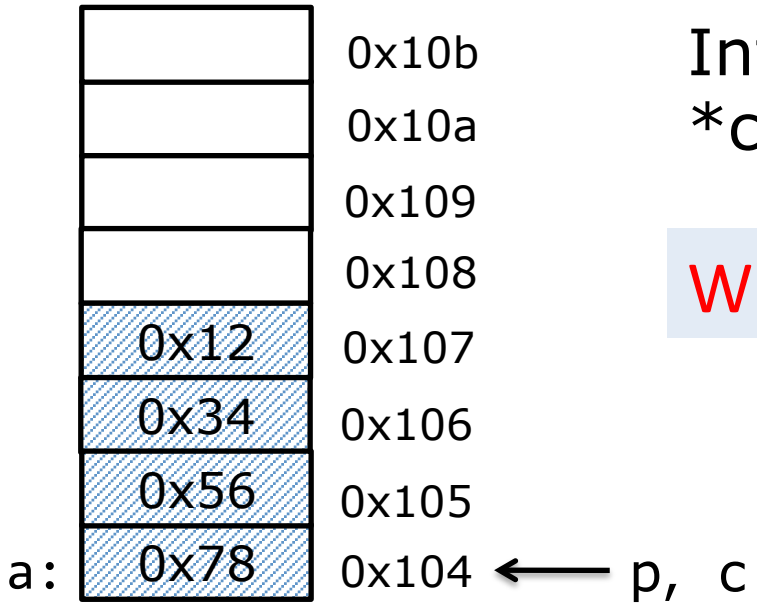
Pointer casting

```
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
printf("%x\n", *c);
```

Output? (when running on Intel laptop)

Pointer casting

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

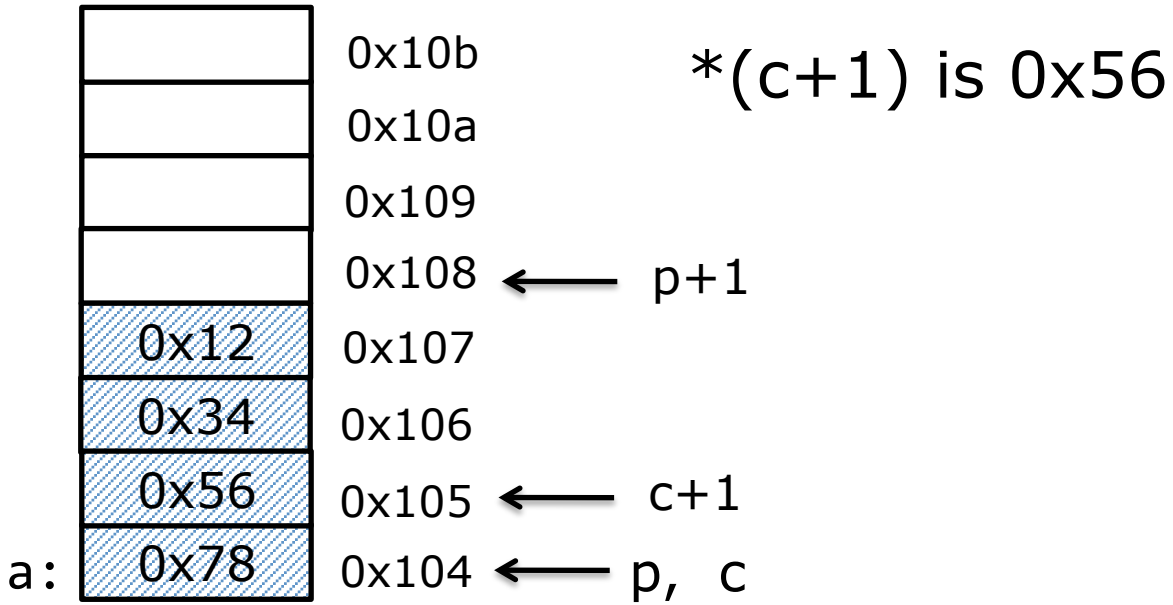


Intel laptop is small endian
*c is 0x78

What is c+1? p+1?

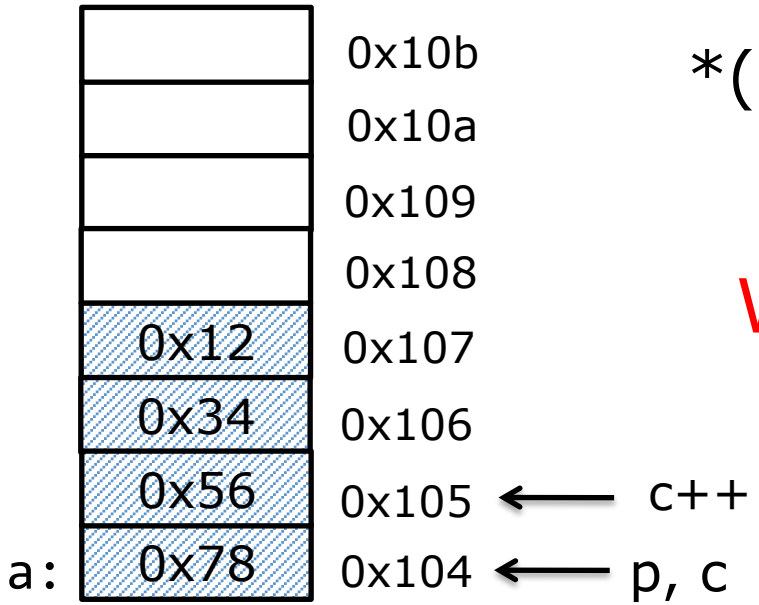
Pointer casting

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



Pointer casting

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



*(c+1) is 0x56

What about big endian?

Another example of pointer casting

```
bool is_normalized_float(float f)
{




}
```

Another example of pointer casting

```
bool is_normalized_float(float f)
{
    unsigned int i;
    i = *(unsigned int *)&f;

    unsigned exp = (i&0x7fffffff)>>23;
    return (exp != 0 && exp != 127);

}
```

function *sizeof*

sizeof(`type`)

- Returns size in bytes of the object representation of type

sizeof(expression)

- Returns size in bytes of the type that would be returned by expression, if evaluated.

function *sizeof*

sizeof()	result (bytes)
sizeof(int)	
sizeof(long)	
sizeof(float)	
sizeof(double)	
sizeof(int *)	

64 bits machine

function *sizeof*

sizeof()	result (bytes)
sizeof(int)	4
sizeof(long)	8
sizeof(float)	4
sizeof(double)	8
sizeof(int *)	8

64 bits machine

function *sizeof*

expr	sizeof()	result (bytes)
int a = 0;	sizeof(a)	
long b = 0;	sizeof(b)	
int a = 0; long b = 0;	sizeof(a + b)	
char c[10];	sizeof(c)	
int arr[10];	sizeof(arr)	
	sizeof(arr[0])	
int *p = arr;	sizeof(p)	

64 bits machine

function *sizeof*

expr	sizeof()	result (bytes)
int a = 0;	sizeof(a)	4
long b = 0;	sizeof(b)	8
int a = 0; long b = 0;	sizeof(a + b)	8
char c[10];	sizeof(c)	10
int arr[10];	sizeof(arr)	10 * 4 = 40
	sizeof(arr[0])	4
int *p = arr;	sizeof(p)	8

64 bits machine