C - Functions, Pointers, Arrays

Jinyang Li

Some are based on slides of Tiger Wang

Today’s lecture

- Functions
- Pointers
- Array and its relationship to pointer

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 (confining the usage of the variable)

- within the function
- local variables of the same name in different functions are unrelated

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

r’s scope is in function *add*

Local Variables / function arguments

- Function arguments are basically local variables
- Local variables’ storage policy:
  - allocated upon function invocation
  - de-allocated upon function return

```
int add(int a, int b)
{
    int r = a + b;
    return r;
}
int main()
{
    int r;
    ~~add(1, 2);~~ r = add(1, 2);
    printf(“r=%d\n”, r);
    return 0;
}
```

Global Variables

Scope

- Can be accessed by all functions

Storage

- Allocated upon program startup, 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 x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Result x: ?, y: ?

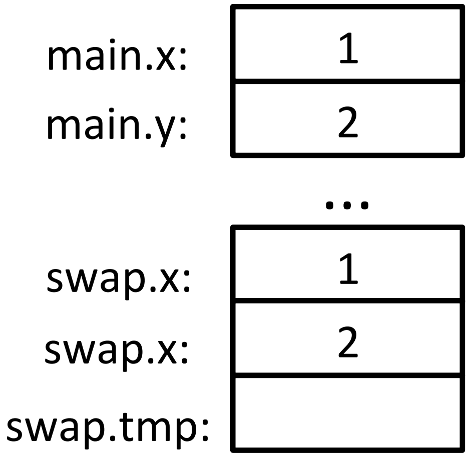
Function invocation

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 x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Result x: 1, y: 2



main.x: 1

main.y: 2

…

swap.x: 1

swap.x: 2

swap.tmp:

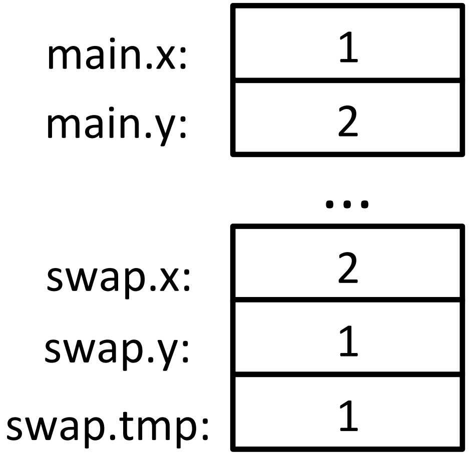
Function invocation

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 x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Result x: 1, y: 2

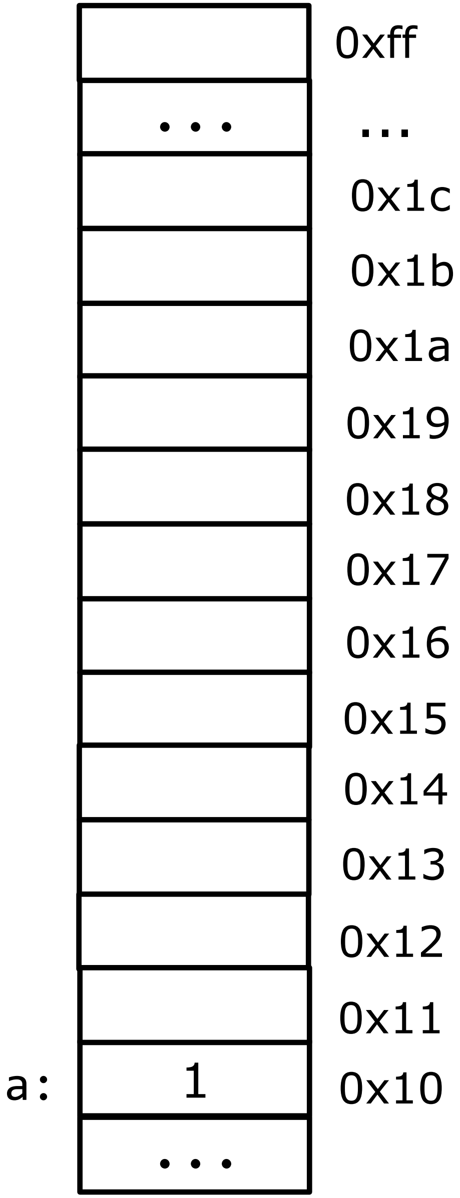


main.x: 1
main.y: 2
…
swap.x: 2
swap.y: 1
swap.tmp: 1

Pointers

Pointer is a memory address

Pointer



0xff

...

...

0x1c

0x1b

0x1a

0x19

0x18

0x17

0x16

0x15

0x14

0x13

0x12

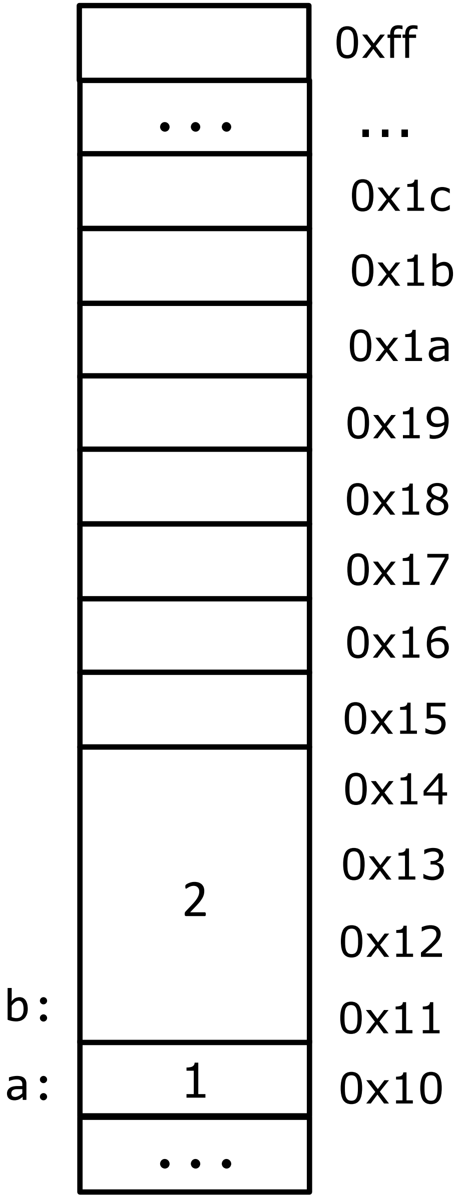
0x11

a: 1 0x10

...

```
char a = 1;
```

Pointer

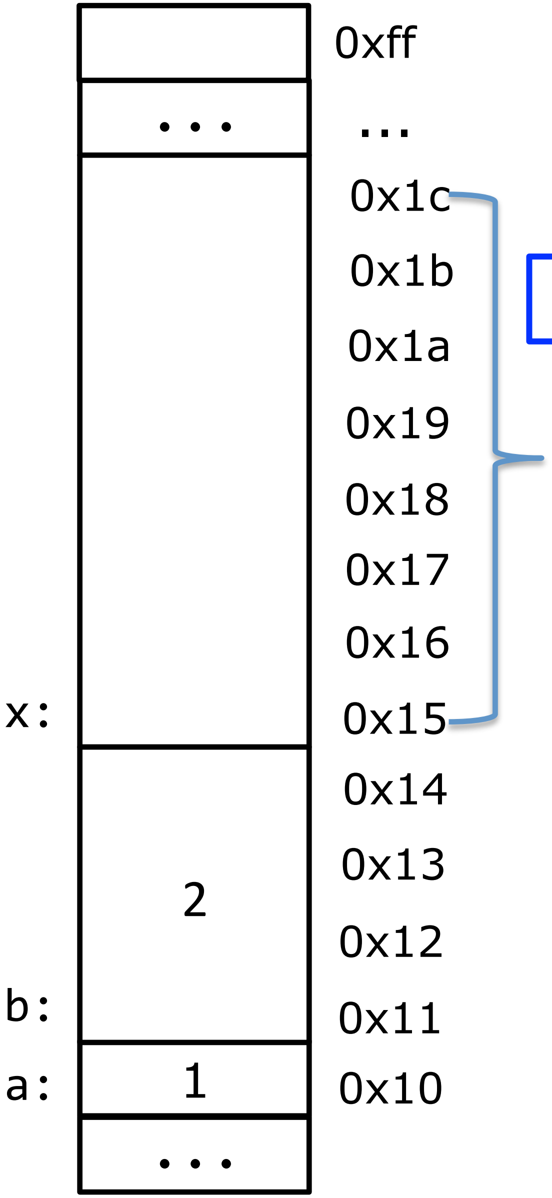


0xff
...
0x1c
0x1b
0x1a
0x19
0x18
0x17
0x16
0x15
0x14
0x13
0x12
0x11
0x10

b: 2
a: 1

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

Pointer



		0xff
	...	…
		0x1c
		0x1b
		0x1a
		0x19
		0x18
		0x17
		0x16
x:		0x15
		0x14
	2	0x13
		0x12
b:		0x11
a:	1	0x10
	...	

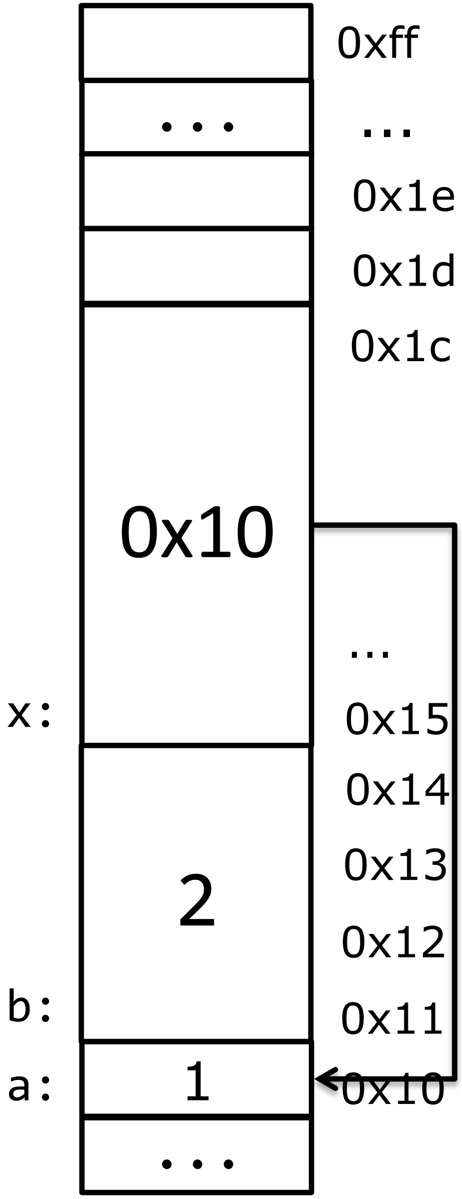
```
char a = 1;
int b = 2;
char *x;
```

char * is a (pointer) type.
char * is the same as char*

Size of pointer on a 64-bit machine?

8 bytes

Pointer



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

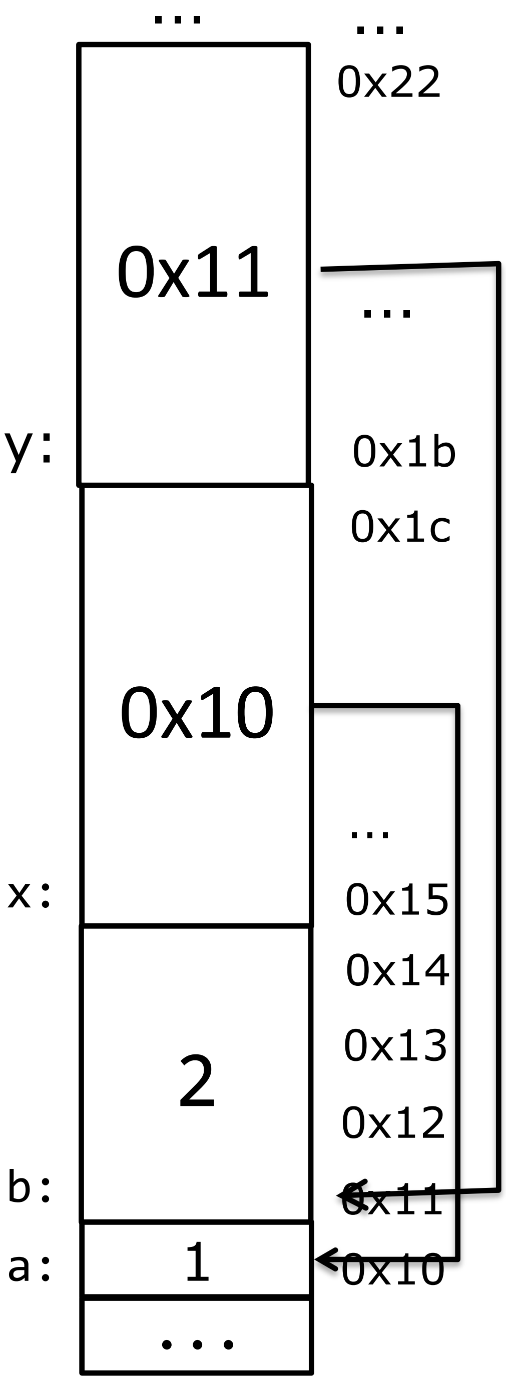
& gives address of variable

char *x; x = &a can be shorted as:
char *x = &a;

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

type mismatch!

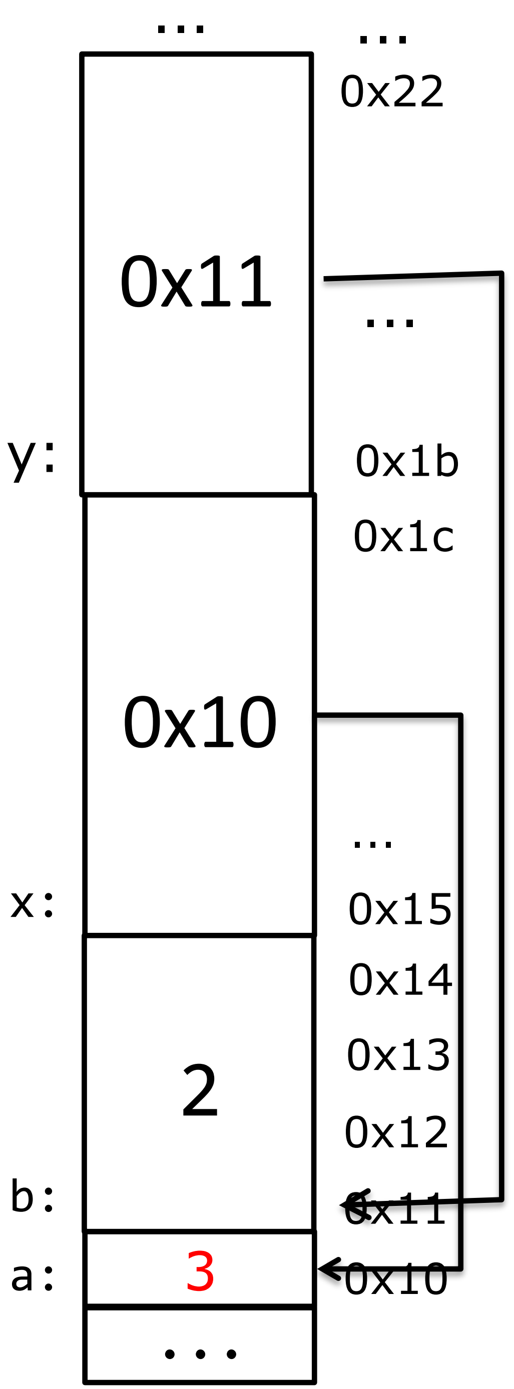
Pointer



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

How to make y a pointer that points to b?

Pointer



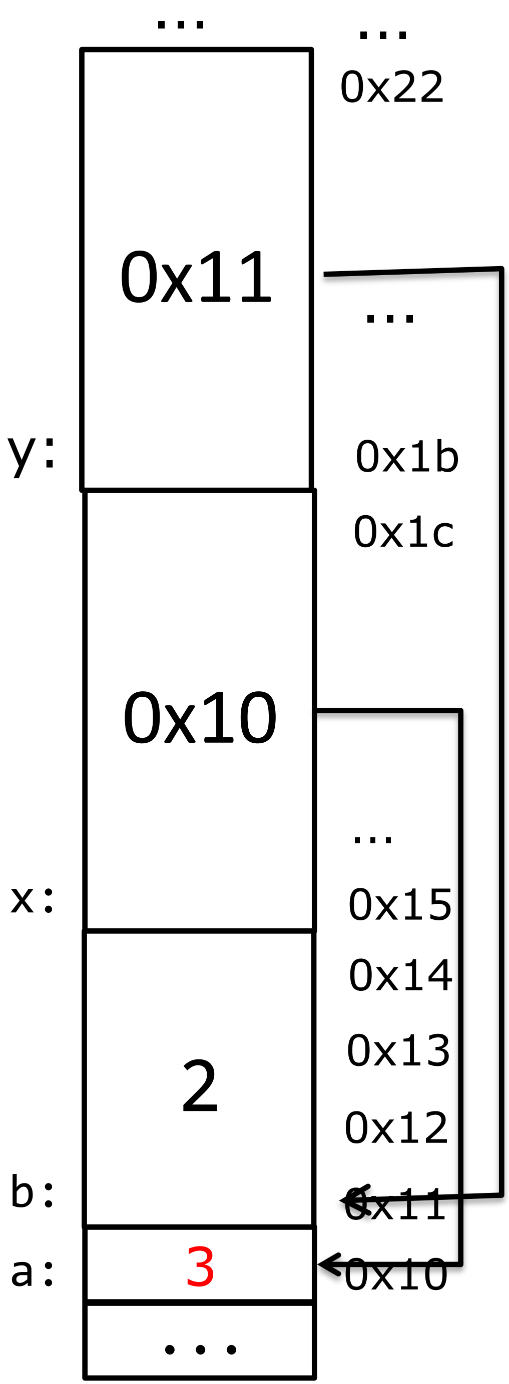
```
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;
```

```
*x = 3;
```

* operator dereferences a pointer

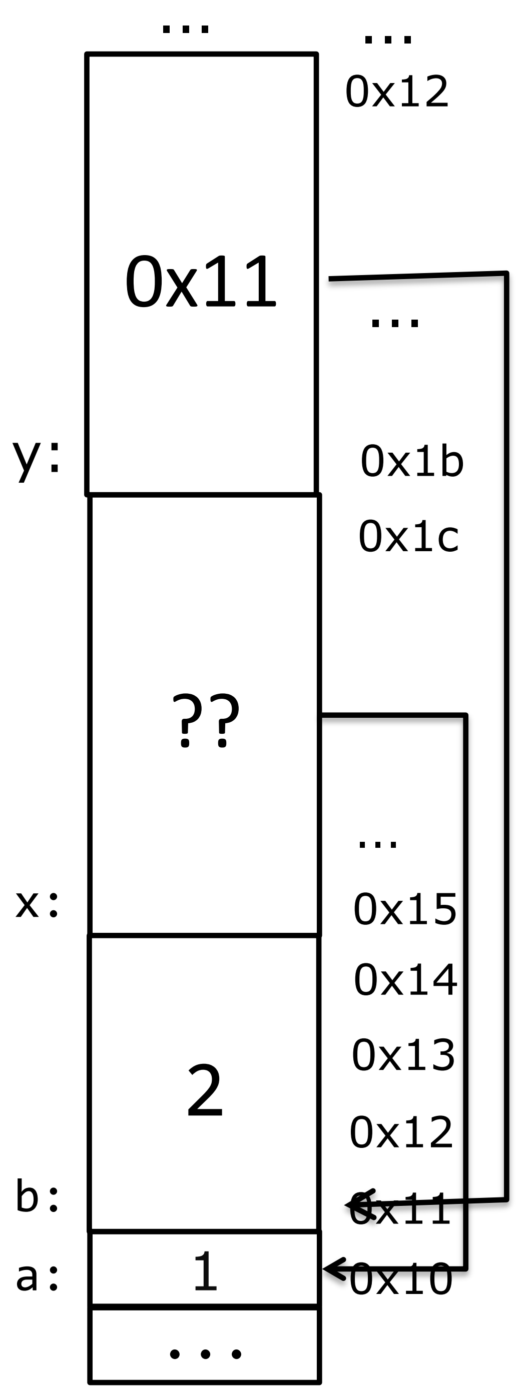
Value of variable a after this statement?

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



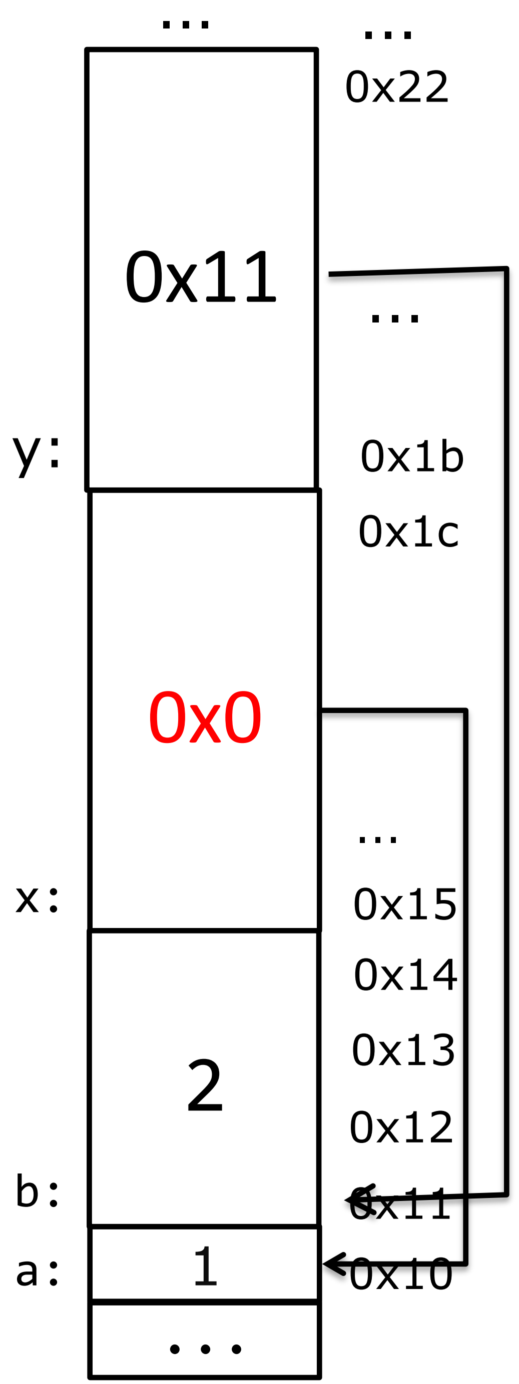
```
char a = 1;
int b = 2;
char *x ~~= &a;~~
int *y = &b;
```

what if x is uninitialized?

```
*x = 3;
```

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

Pointer

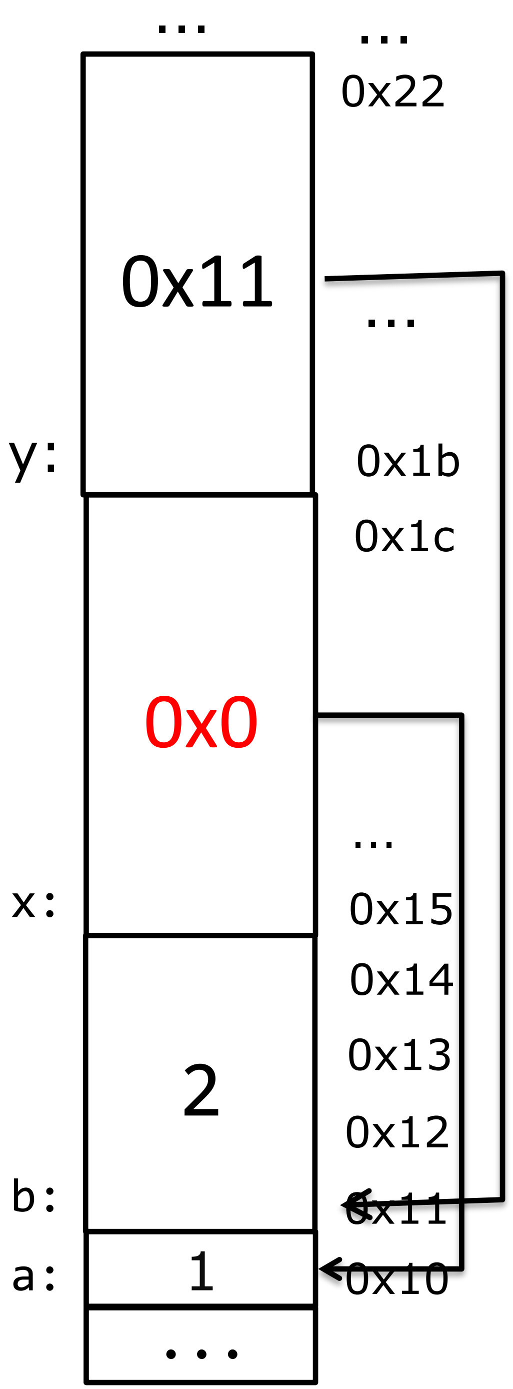


```
char a = 1;
int b = 2;
char *x = NULL;
int *y = &b;

 *x = 3;
```

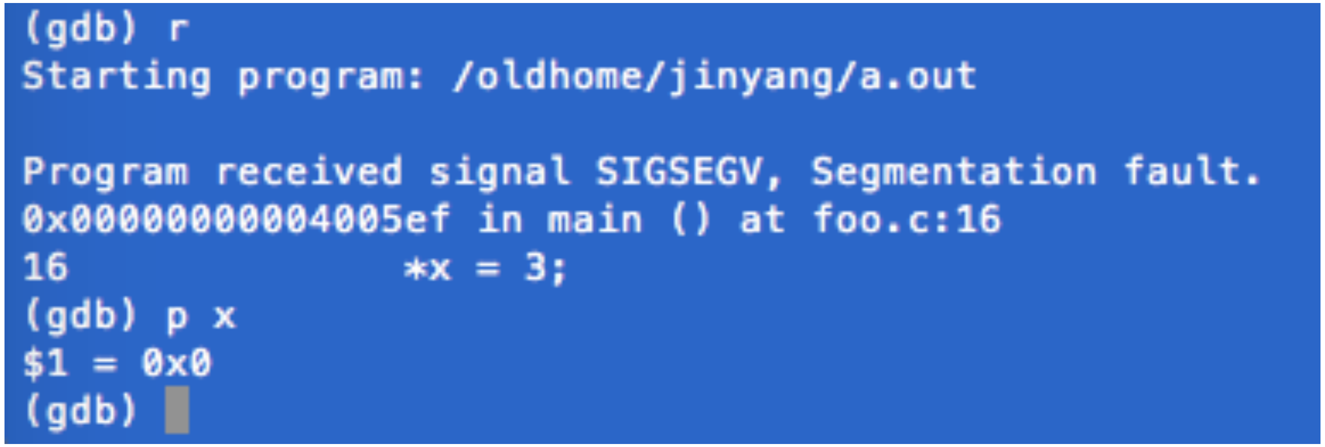
Always initialize pointers!

Dereferencing NULL pointer definitely results in “Segmentation fault”

Pointer

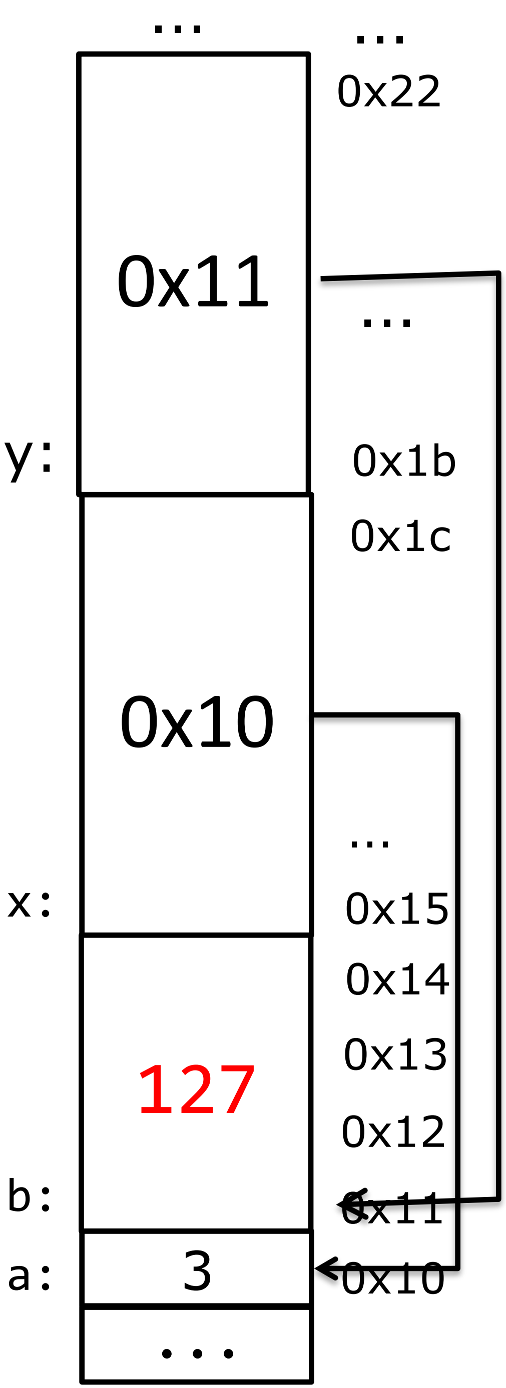


```
char a = 1;
int b = 2;
char *x = NULL;
int *y = &b;


 *x = 3;
```



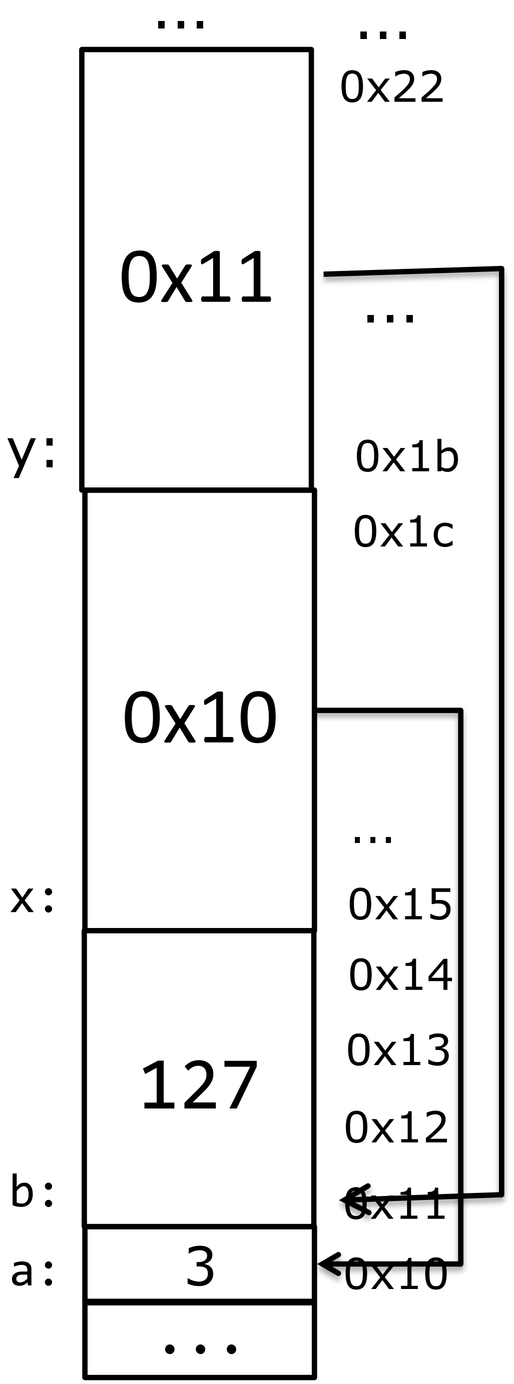
```
(gdb) r
Starting program: /oldhome/jinyang/a.out

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;
```

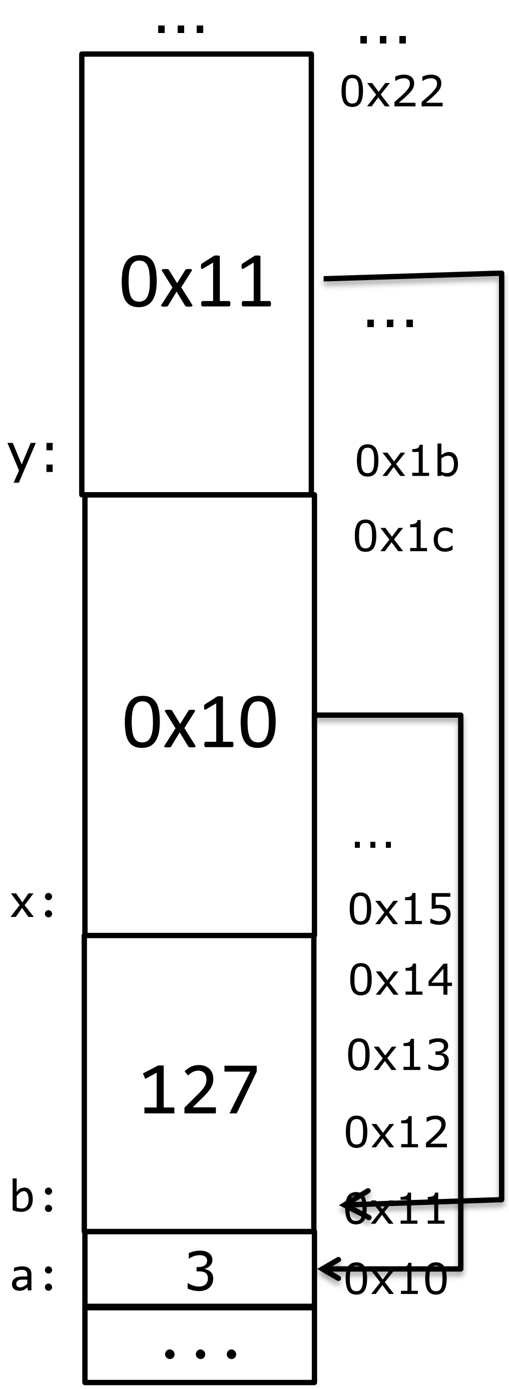
equivalent to
char **xx;
xx = &x;

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;
```

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

char*   b, c; does not work

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

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

Pass pointers to function

Pass the copies

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

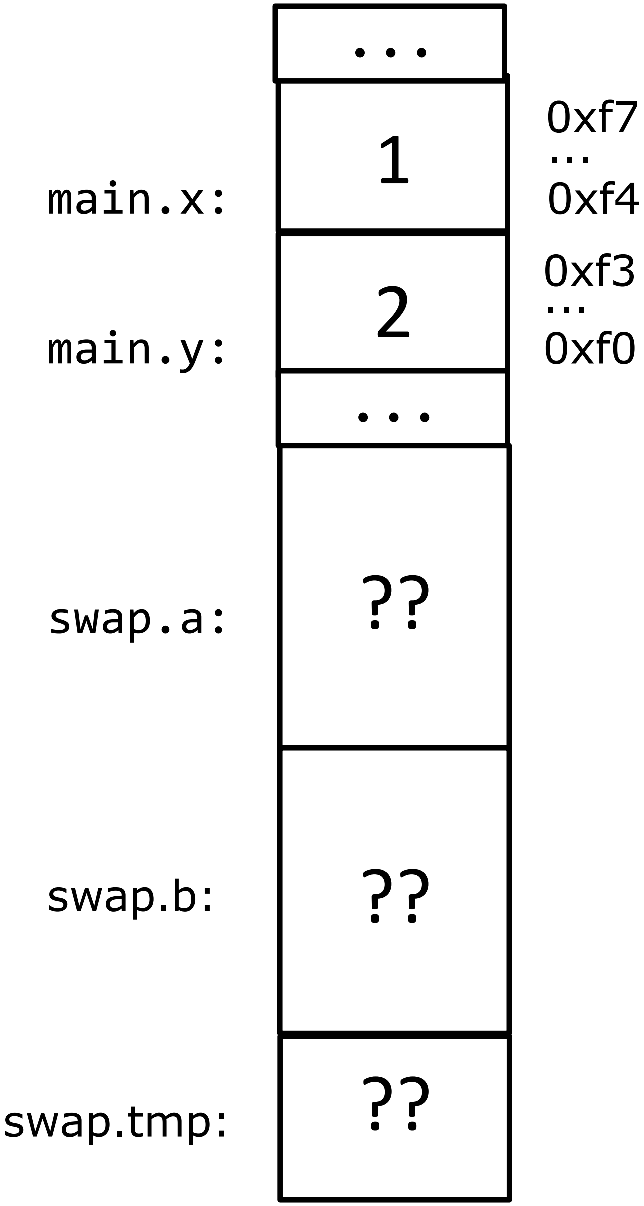
Pass pointers to function

Pass the pointers

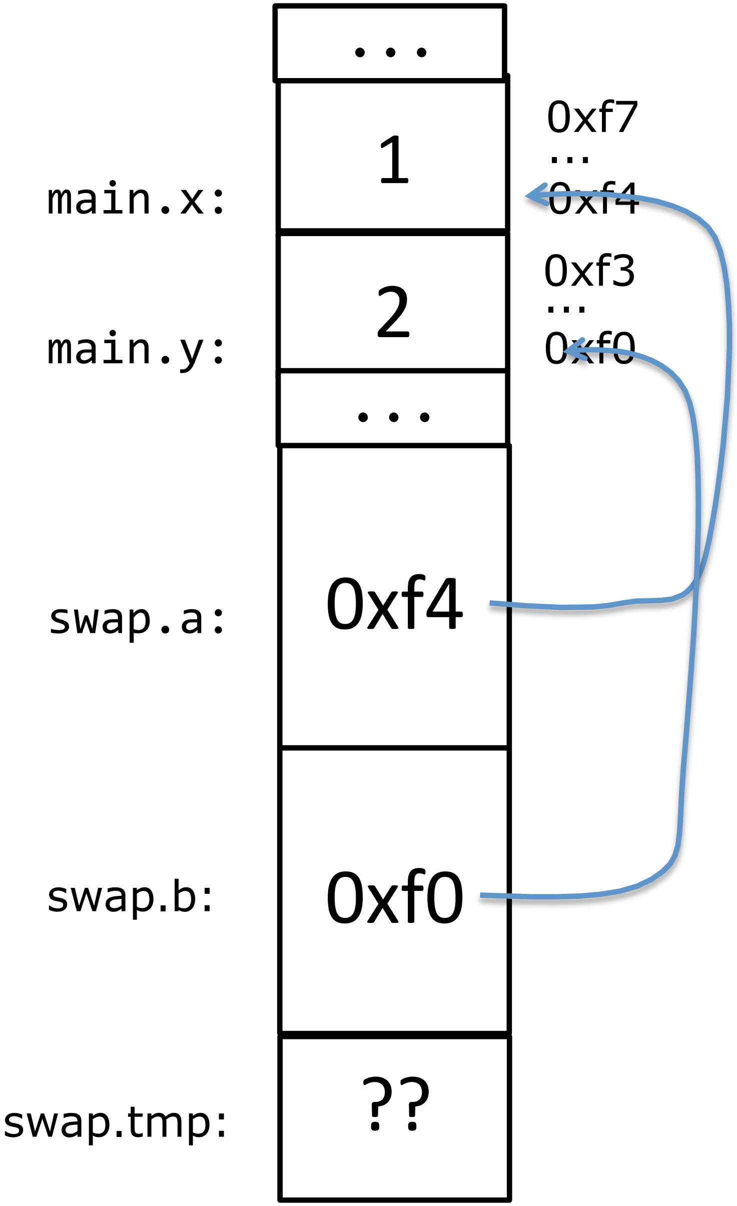
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
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);
}
```



...

main.x: 1 0xf7 ... 0xf4

main.y: 2 0xf3 ... 0xf0

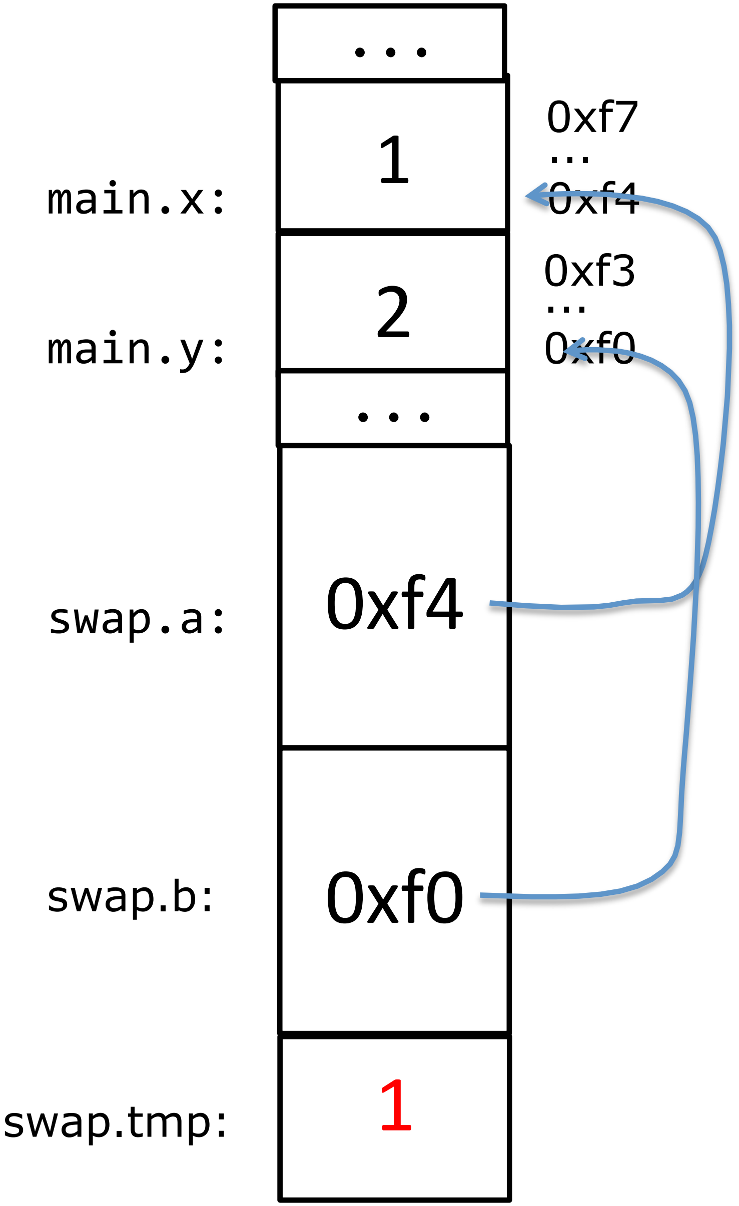
...

swap.a: 0xf4

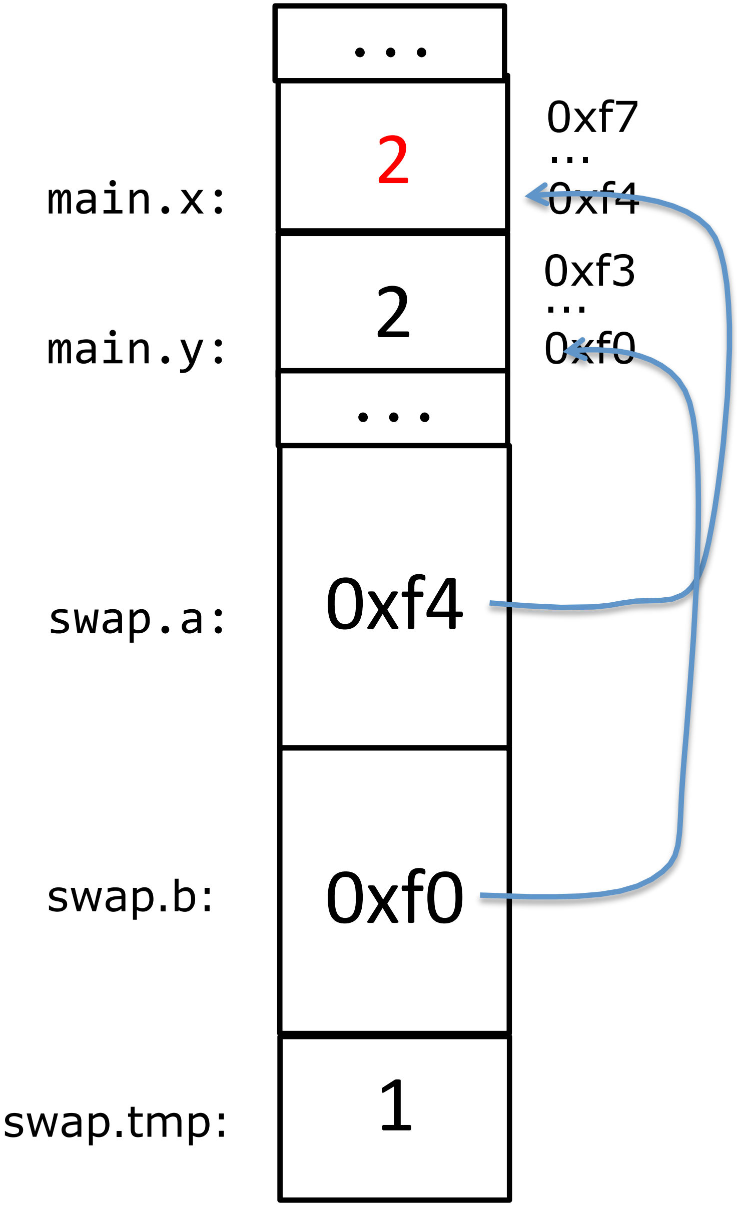
swap.b: 0xf0

swap.tmp: ??

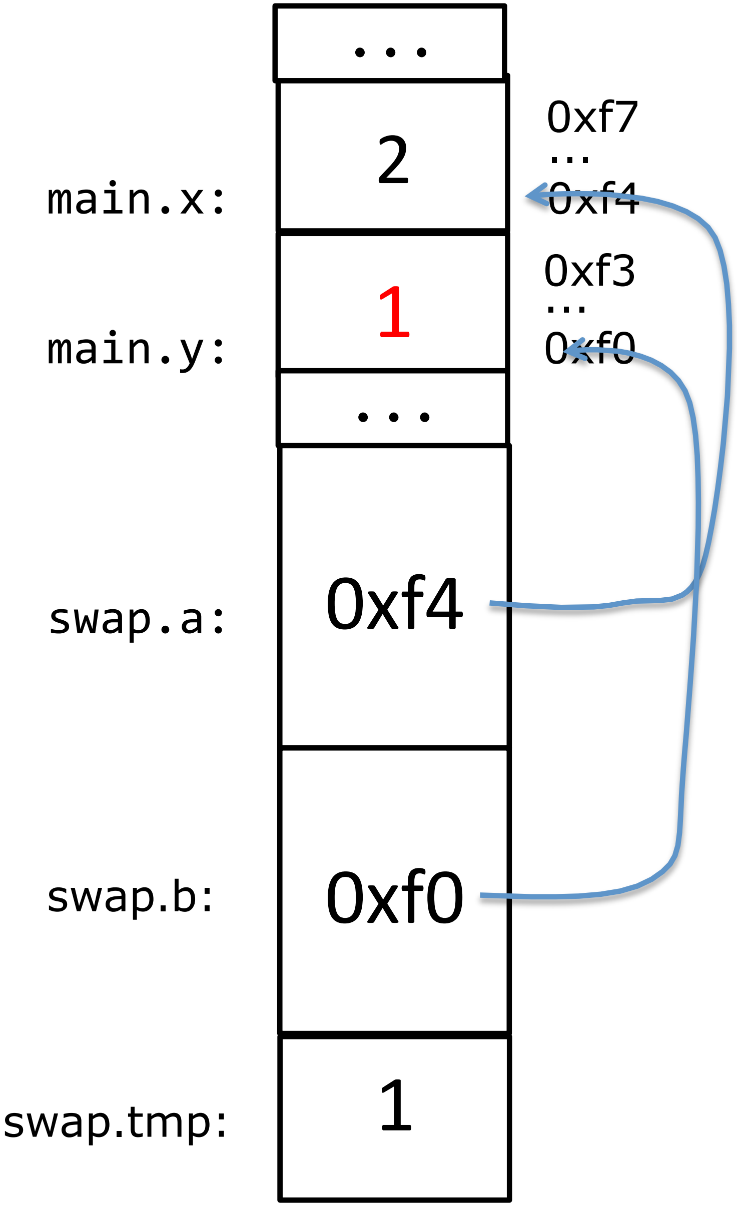
```
void swap(int* a, int* b)
{
     int tmp = *a;
     *a = *b;
     *b = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

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



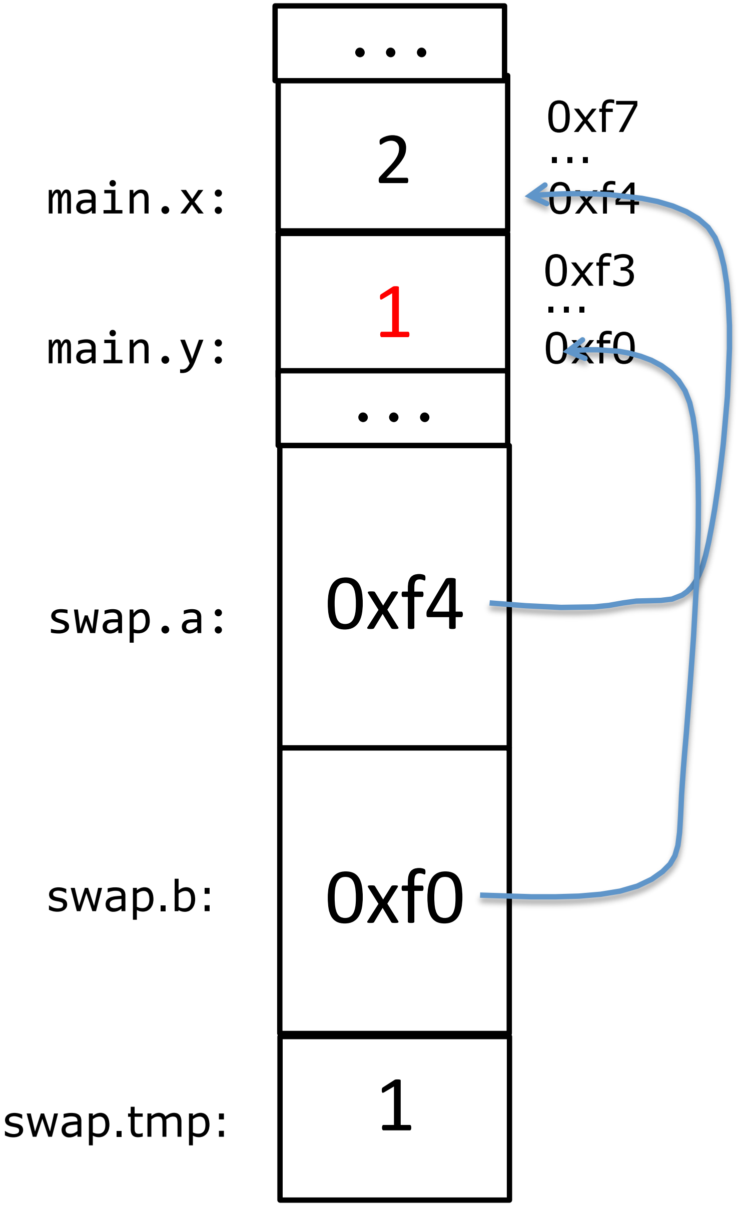
```
void swap(int* a, int* b)
{
     int tmp = *a;
     *a = *b;
     *b = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    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);
}
```

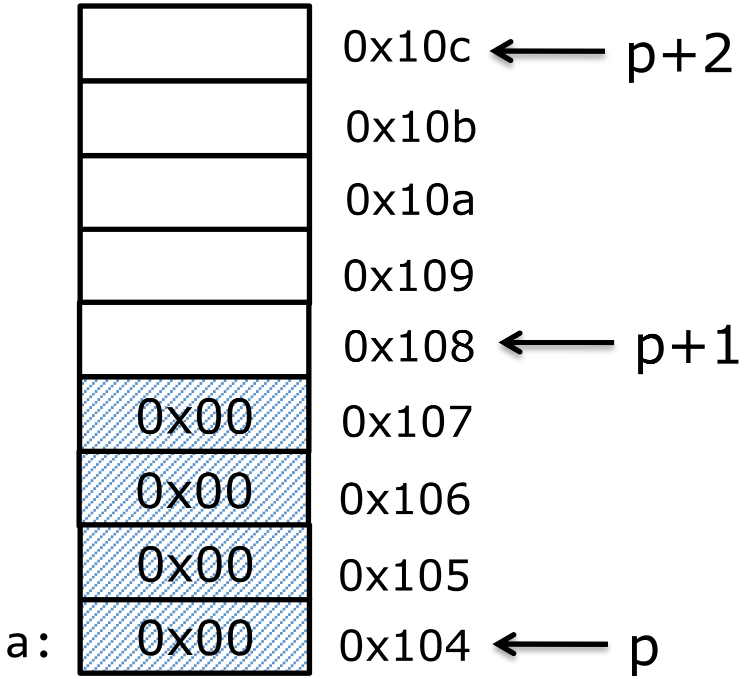


Pointer arithmetic

```
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 ith 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 ith 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 ith 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 ith 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 ith 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

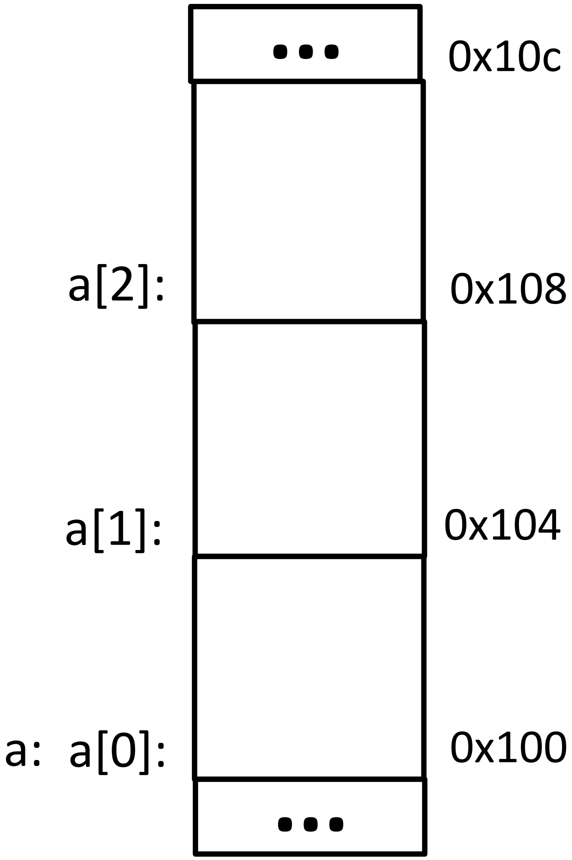
What we’ve learnt and today’s plan

- Bitwise operations
- Pointers
  - Pointers are addresses
  - With pointers arguments, a callee can modify local variables in the caller.
- Today’s lesson:
  - Array and its relationship with pointer
  - Pointer casting

Array: a collection of contiguous objects with the same type

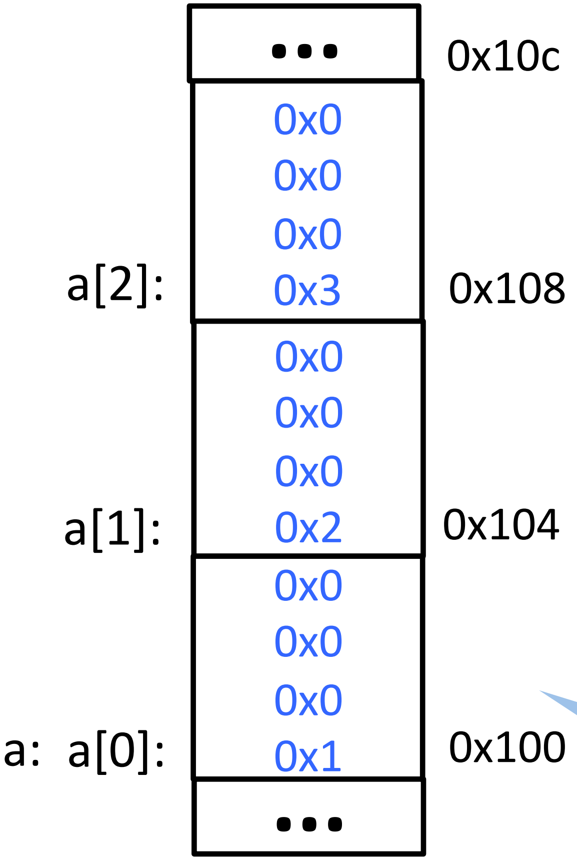
Array

```
int a[3];
```



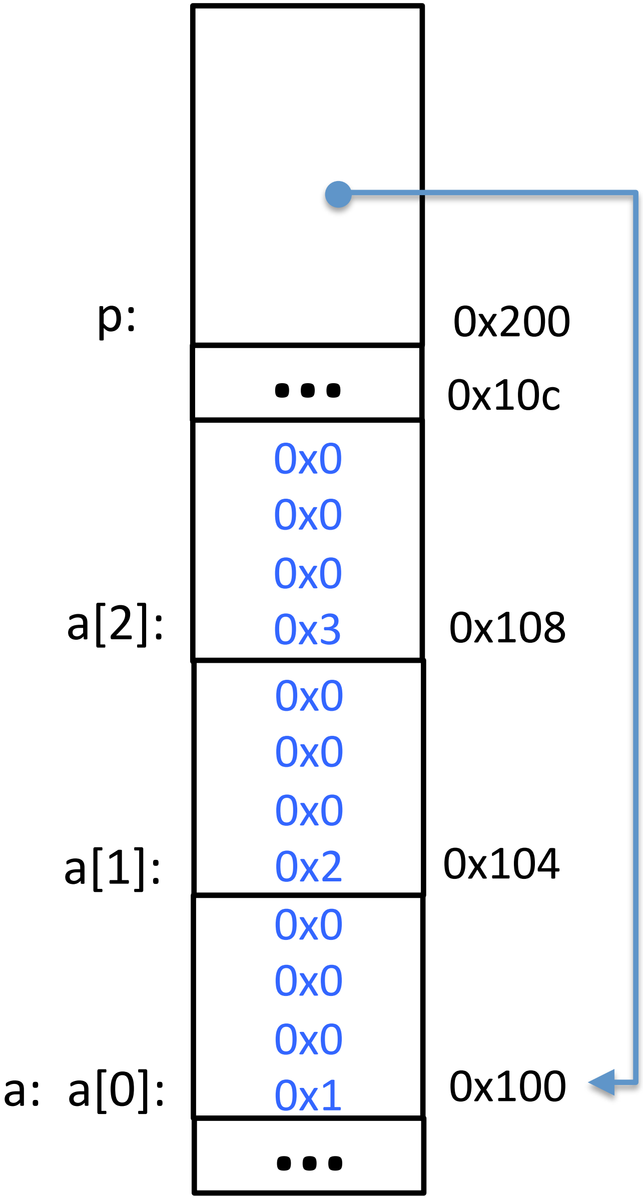
Array

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



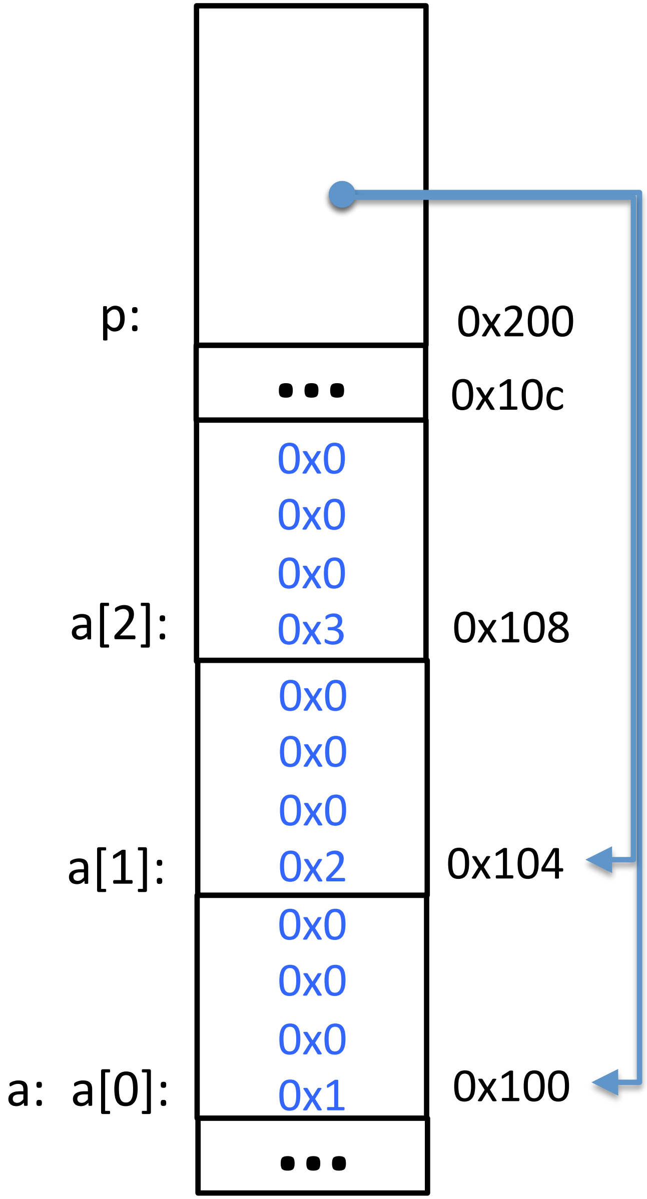
If machine is little Endian

Array



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

int *p;
p = &a[0]; //equivalent to p = a;

printf(“%p\n”, p); //output?  0x100

printf(“%d\n”, *p); //output?  1
```

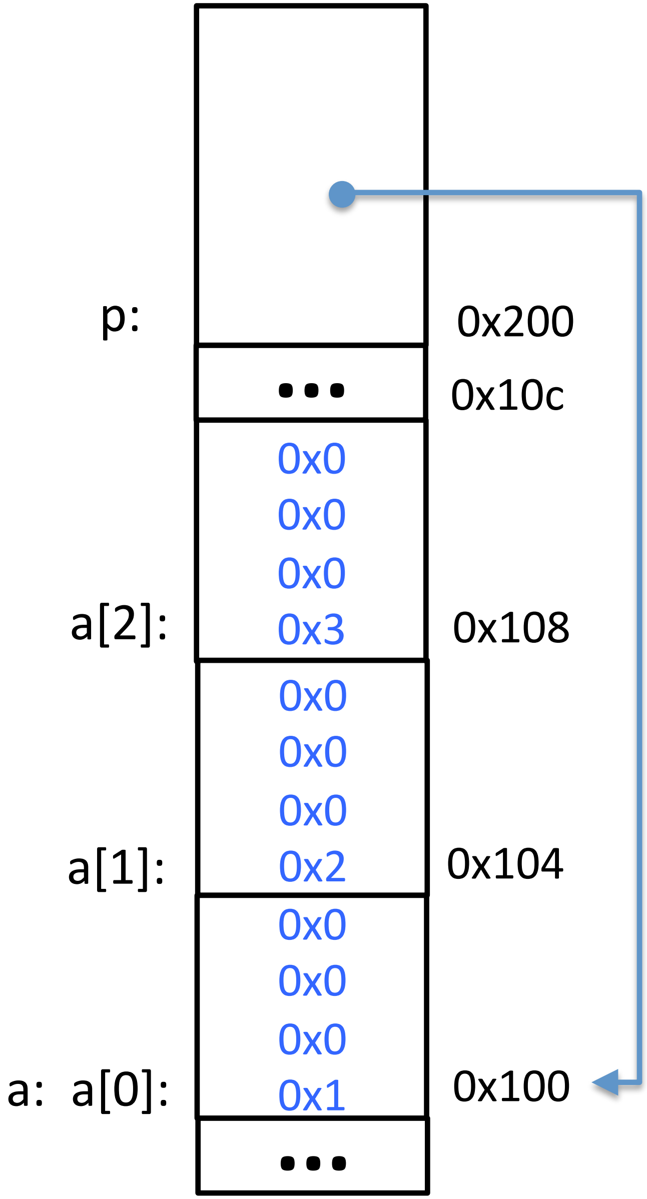
Pointer arithmetic



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

int *p;
p = &a[0];

p = p + 1; //equivalent to p++


printf("%p\n", p); //output?  0x104

printf("%d\n", *p); //output?  2
```

Rule of pointer arithmetic:
p+i has address of *i*-th object after p, i.e.
p+i's value is p's value plus i*sizeof(int)

Pointer arithmetic

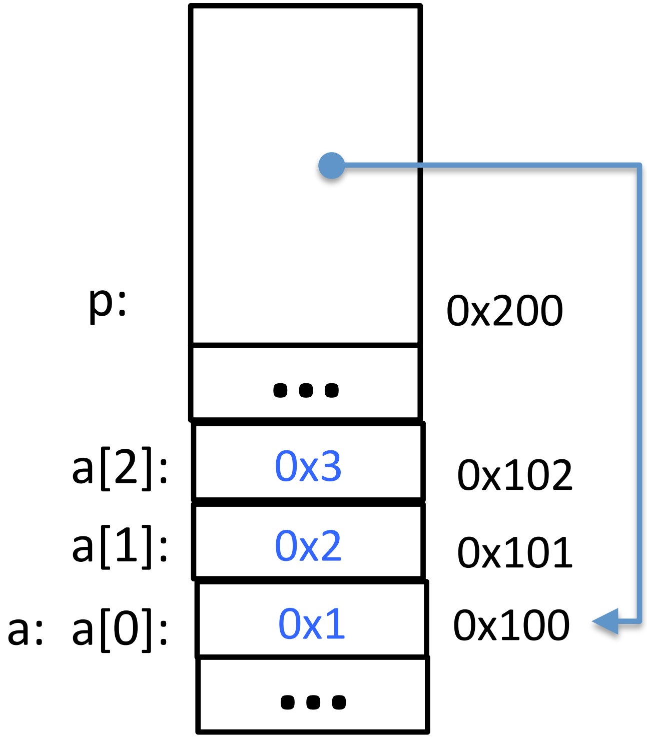


```
int a[3] = {1, 2, 3};

int *p;
p = &a[0];

printf("%p\n", p+2); //output?      0x108

printf("%d\n", *(p+2)); //output?  3
```

Rule of pointer arithmetic:
p+i has address of *i*-th object after p, i.e.
p+i's value is p's value plus i*sizeof(int)

Pointer arithmetic

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

char *p;
p = &a[0];

printf(“%p\n”, p+2); //output?      0x102

printf(“%d\n”, *(p+2)); //output?  3
```



Rule of pointer arithmetic:
p+i has address of *i*-th object after p, i.e.
p+i’s value is p’s value plus i*sizeof(int)

Array and pointer

```
int a[10];
int *p;

p = &a[0]; // a is alias for &a[0];

for (int i = 0; i < 10; i++) {

   *(p+i) = 0; // p[i] is alias for *(p+i)

}
```

Array and pointer

```
int a[10];
int *p;

p = a;      // a is alias for &a[0];

for (int i = 0; i < 10; i++) {

   p[i] = 0;   // p[i] is alias for *(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”);
}
```

same as:
int *p;
p = a;

same as:
p[0] = 400;

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]);
   }
}
```

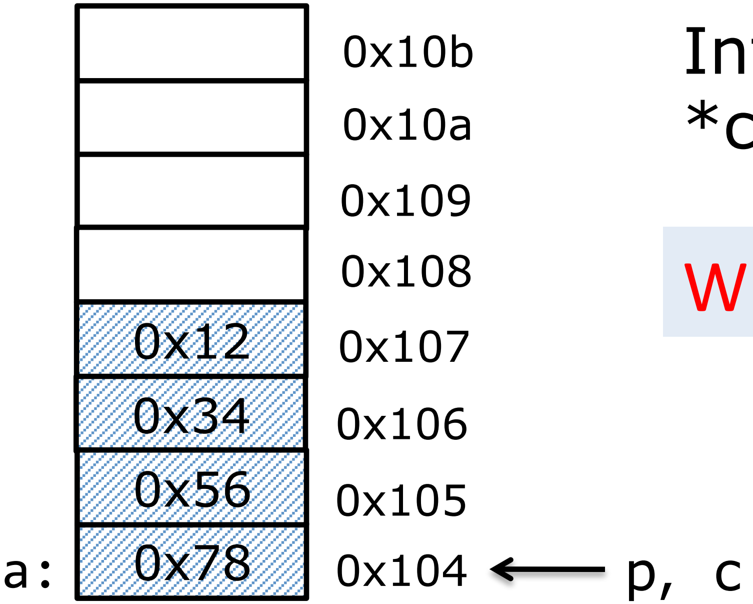
Pointer casting

```
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;
```

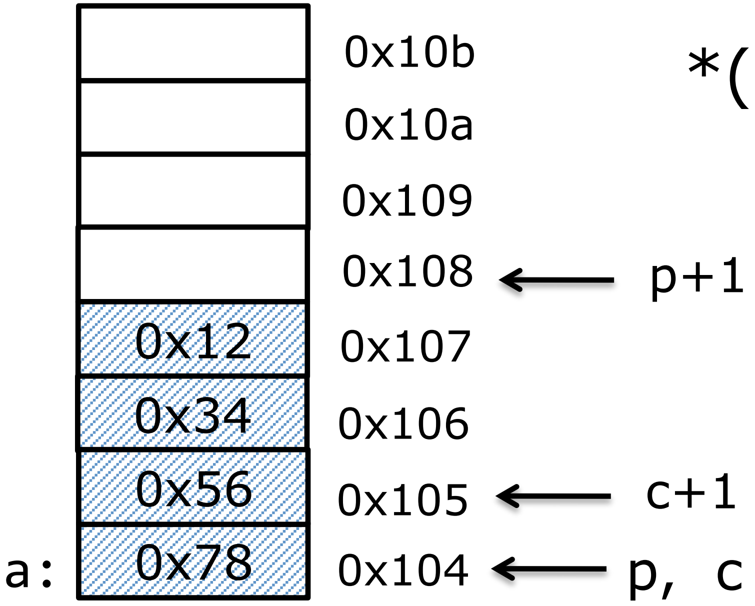


Intel laptop is small endian
*c is 0x78

What is c+1? p+1?

Pointer casting

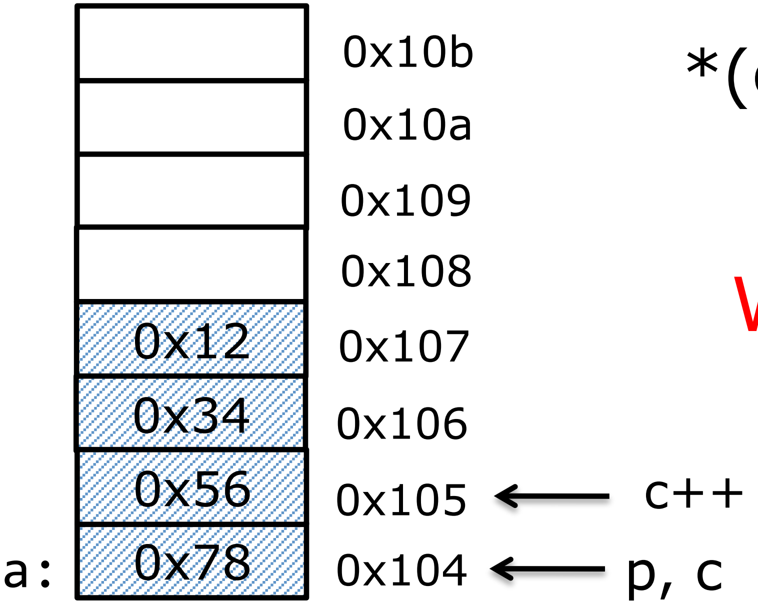
```
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```



*(c+1) is 0x56

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