C - Function, Pointer, Array

Shuai Mu

based on the slides of Tiger Wang and Jinyang Li

Functions

C program consists of functions (aka subroutines, procedures)

Why breaking code into functions?

- Readability
- Reusability

Function

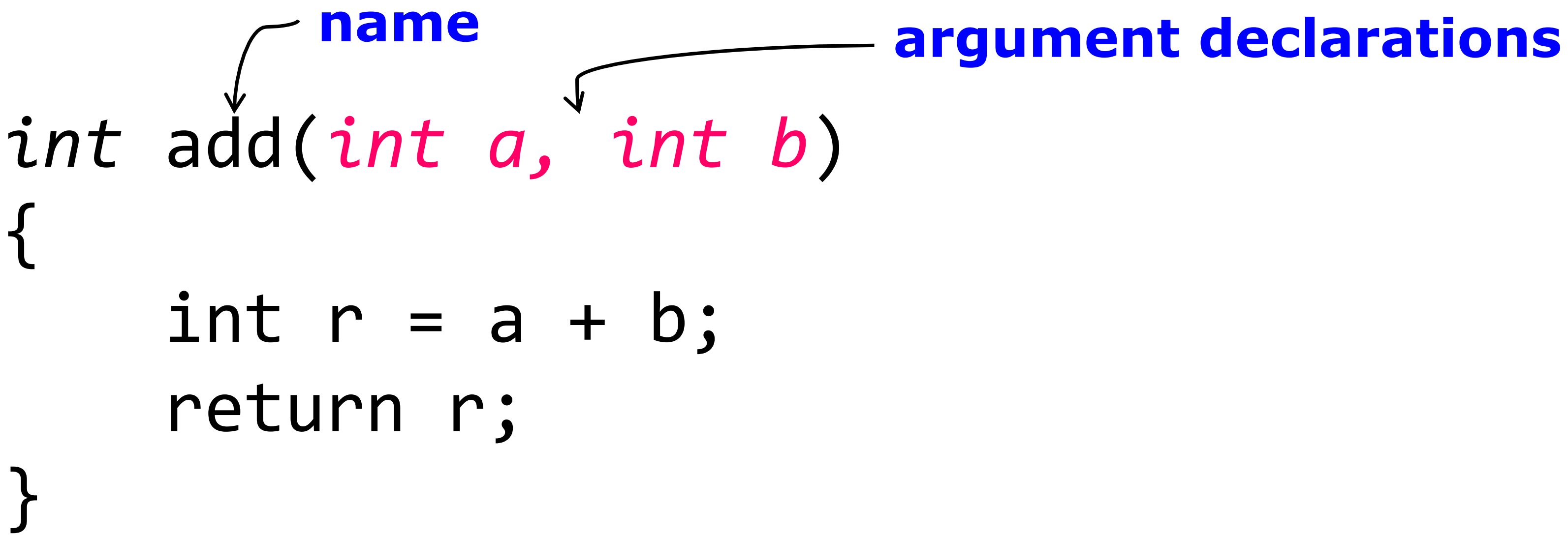
```
int add(int a, int b)
{
    int r = a + b;
    return r;
}
```

Function

name

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

Function

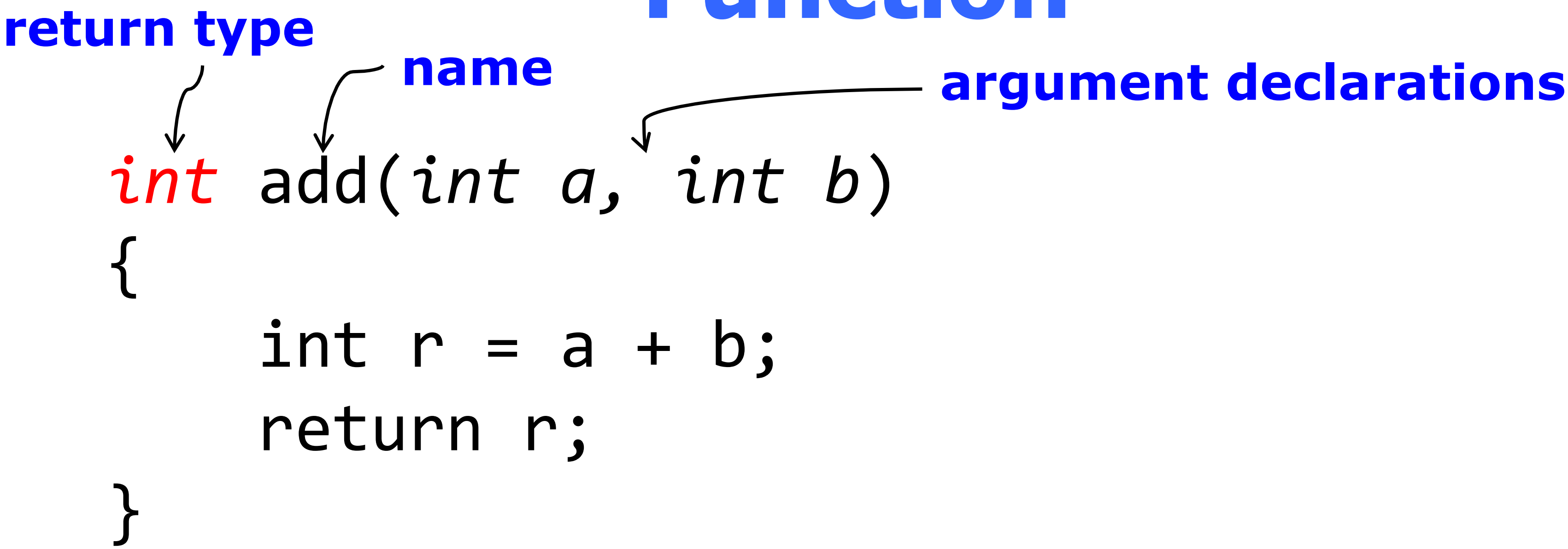


name

argument declarations

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

Function



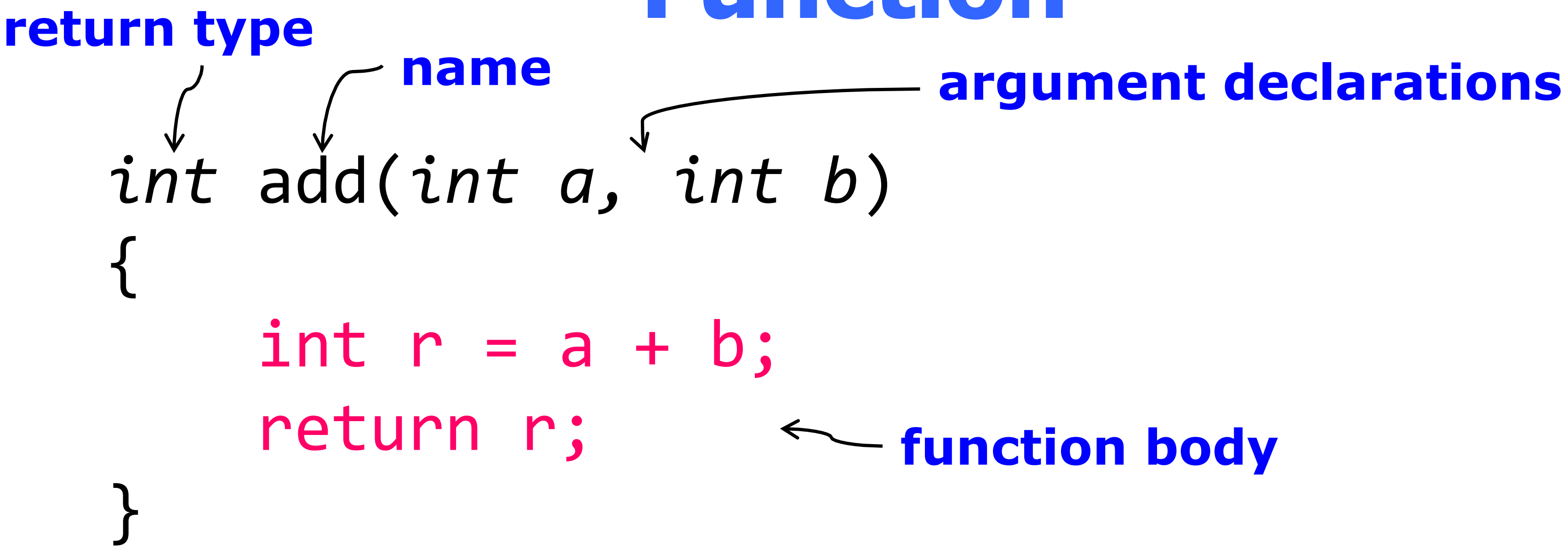
return type

name

argument declarations

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

Function



return type

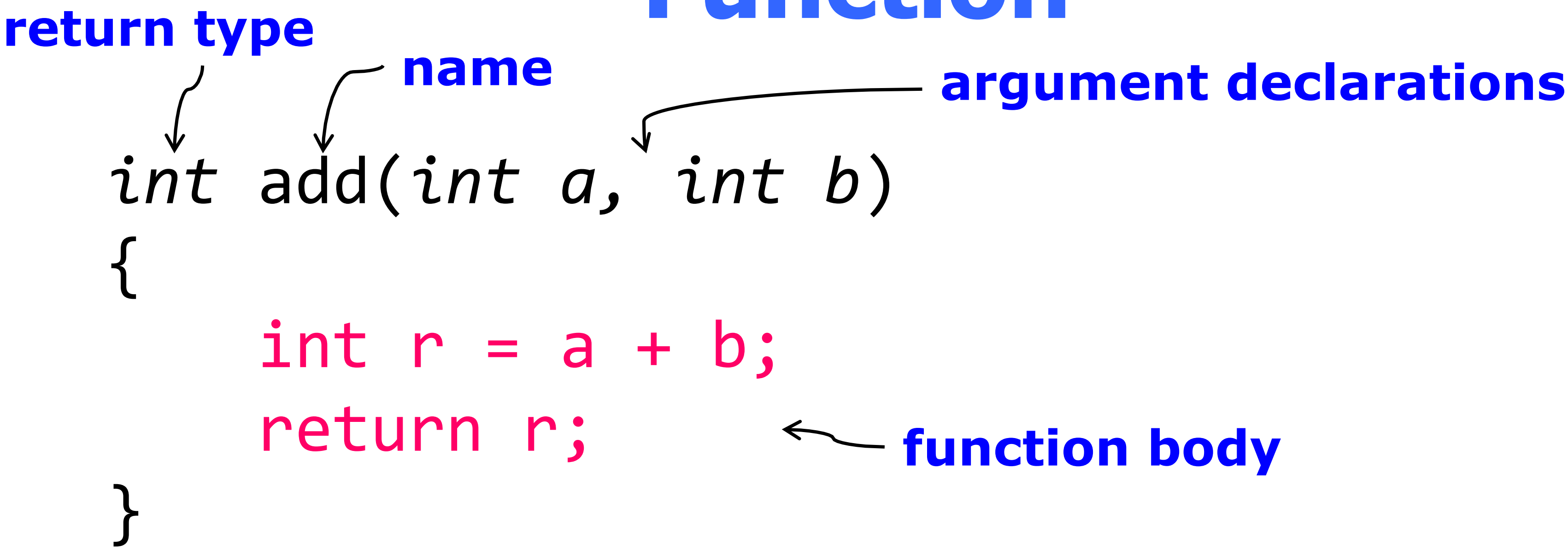
name

argument declarations

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

function body

Function



return type

name

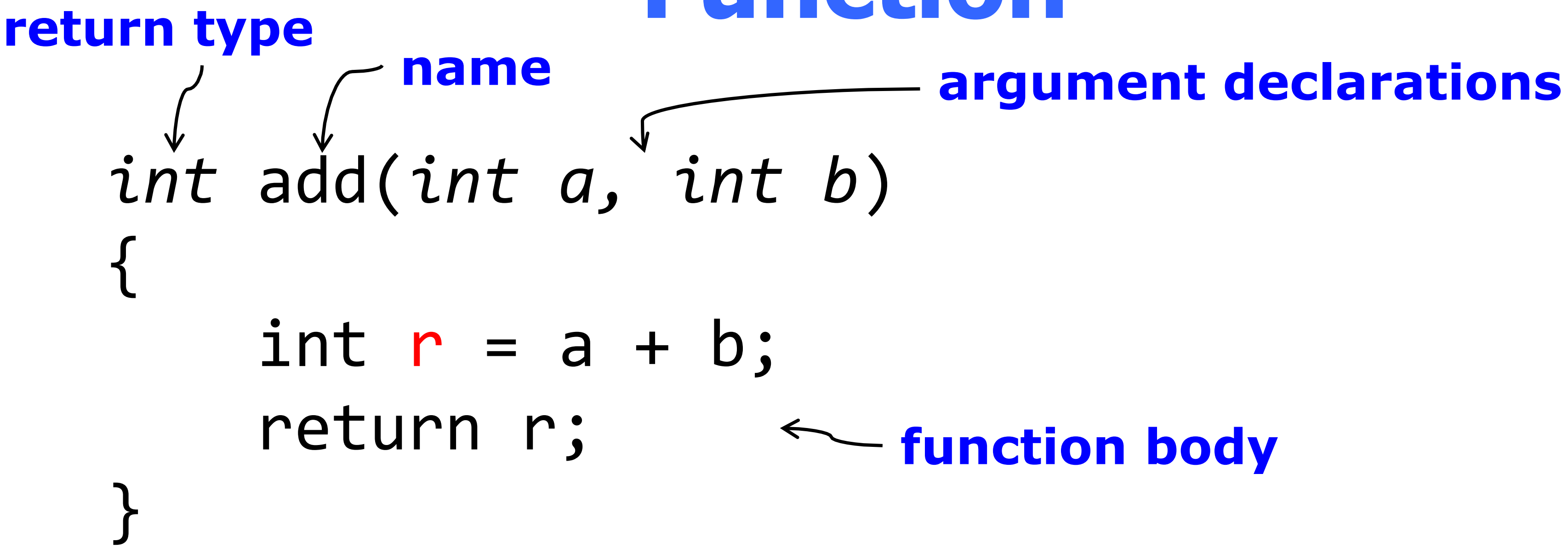
argument declarations

function body

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

```
void dummy()
{ }
```

Function



return type

name

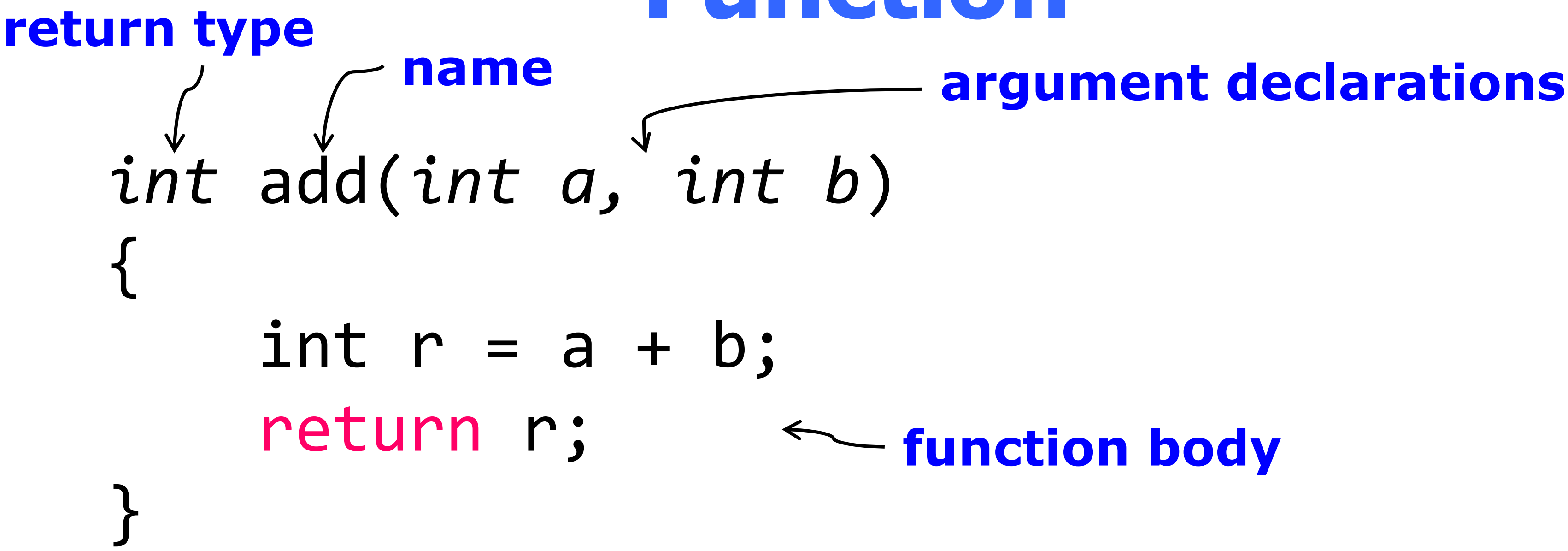
argument declarations

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

function body

r the function's local variable

Function



return type

name

argument declarations

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

function body

r the function’s local variable

return *expression*

Ideal length

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.

PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

Why small size?

- It fits easily on your screen without scrolling
- It should be the code size that you can hold in your head
- It should be meaningful enough to require a function in its own right

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
- deallocated 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;
}
```

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

Function invocation

C passes the arguments by value

```
int calculator(char op,
               int x, int y)
{
    int res;
    switch(op) {
      case ‘+’:
         res = add(x, y);
      case ...
   }
   return res
}
```

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

Function invocation

```
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

main.x:	1
main.y:	2
	…
swap.a:	1
swap.b:	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 a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

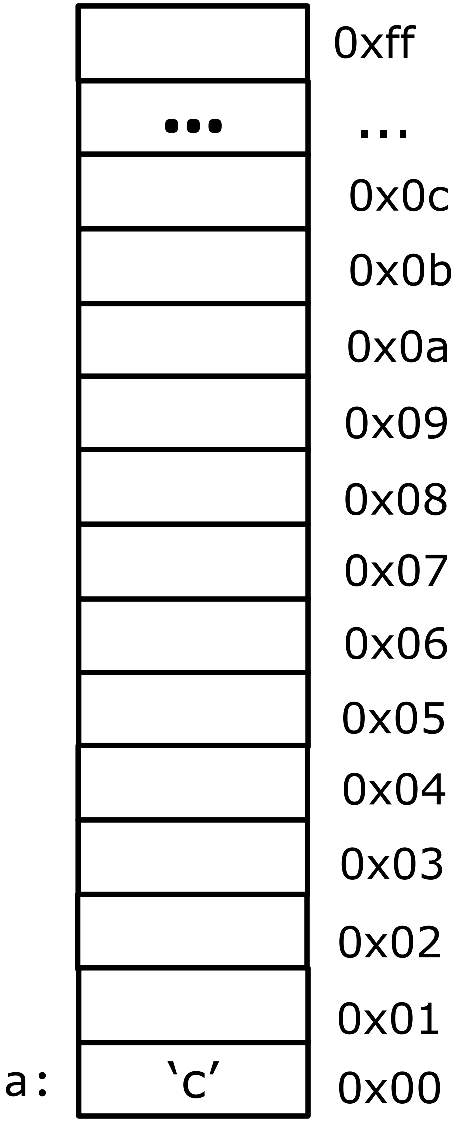
Result x: 1, y: 2

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

Pointers

Pointer is a memory address

Pointer



		0xff
	•••	…
		0x0c
		0x0b
		0x0a
		0x09
		0x08
		0x07
		0x06
		0x05
		0x04
		0x03
		0x02
		0x01
a:	‘c’	0x00

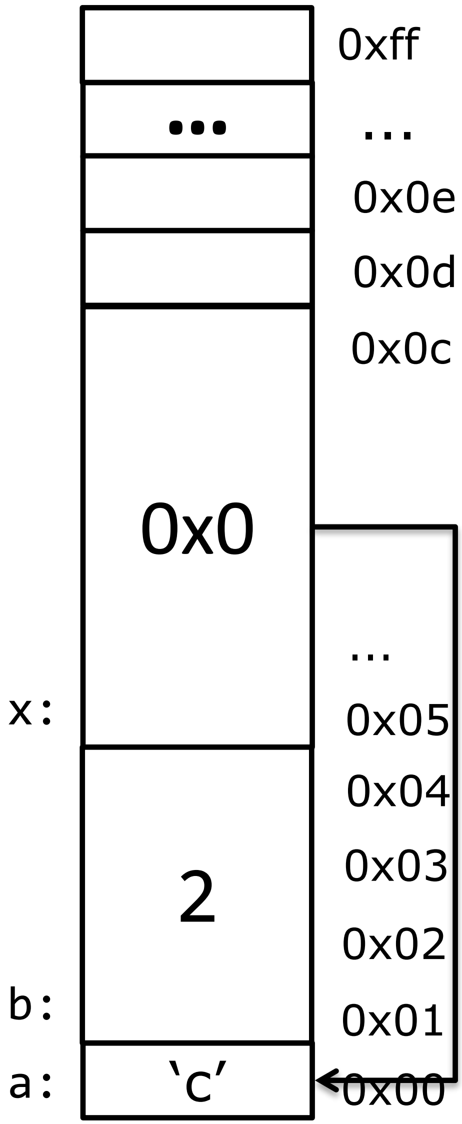
```
char a = ‘c’;
```

Pointer

```
char a = ‘c’;
int b = 2;
```

		0xff
	...	...
		0x0c
		0x0b
		0x0a
		0x09
		0x08
		0x07
		0x06
		0x05
	2	0x04
		0x03
		0x02
b:		0x01
a:	‘c’	0x00

Pointer

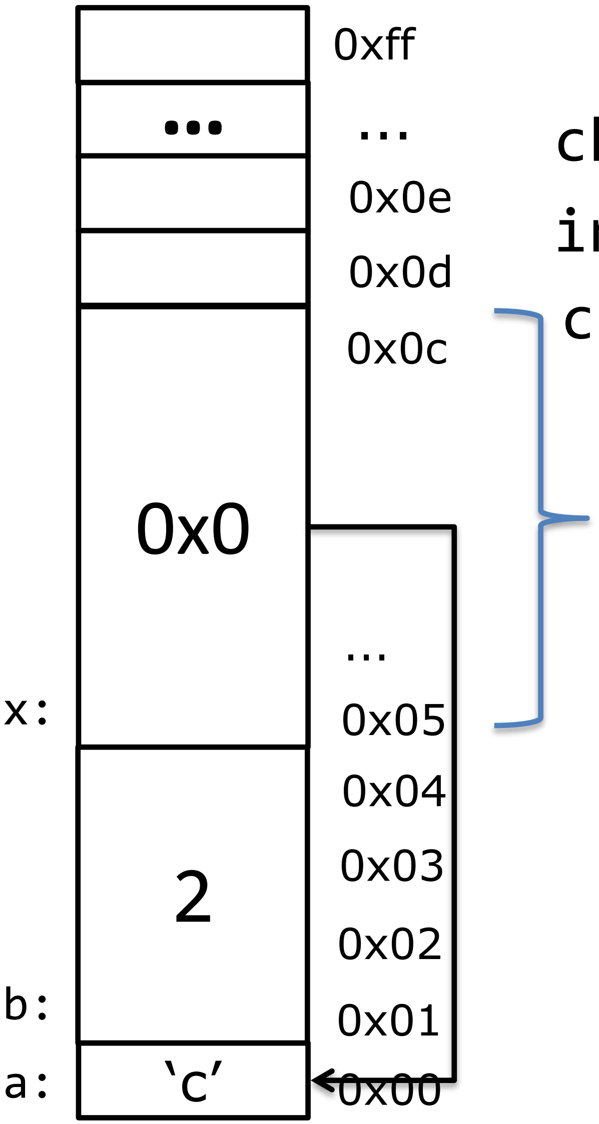


```
char a = 'c';
int b = 2;
char *x = &a;
```

& gives address of variable

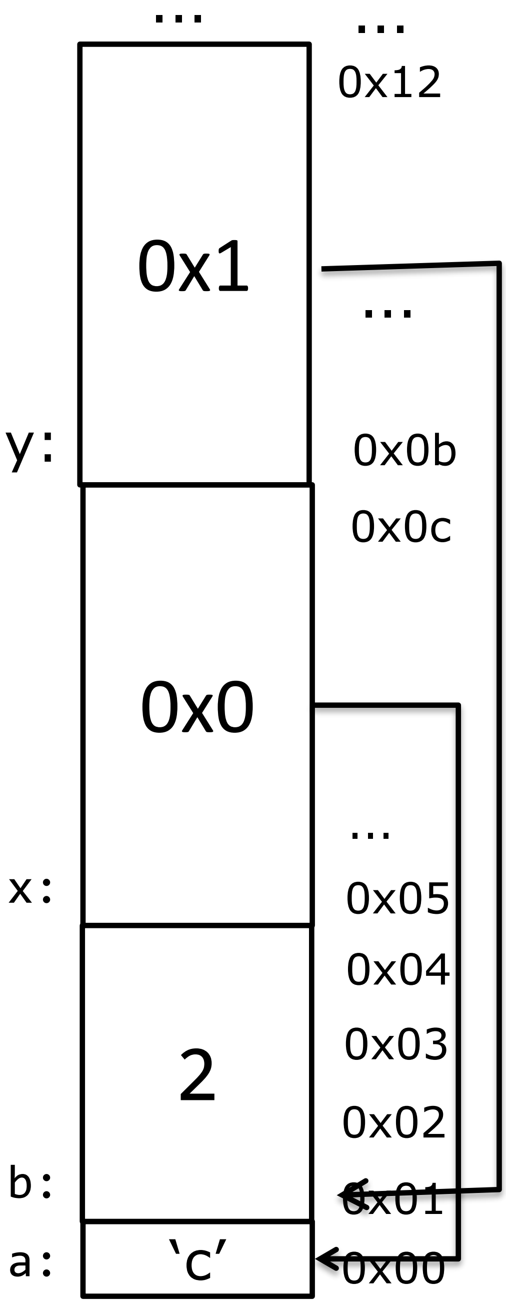
Pointer

```
char a = ‘c’;
int b = 2;
char *x = &a;
```



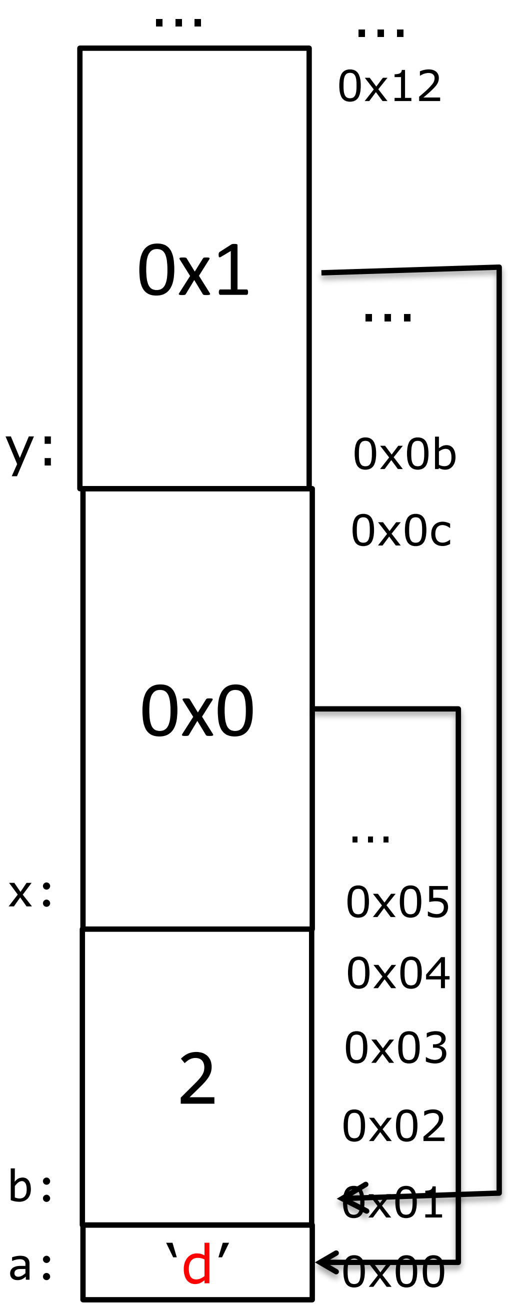
Size of pointer on a 64-bit machine: 8 bytes

Pointer



```
char a = ‘c’;
int b = 2;
char *x = &a;
int *y = &b;
```

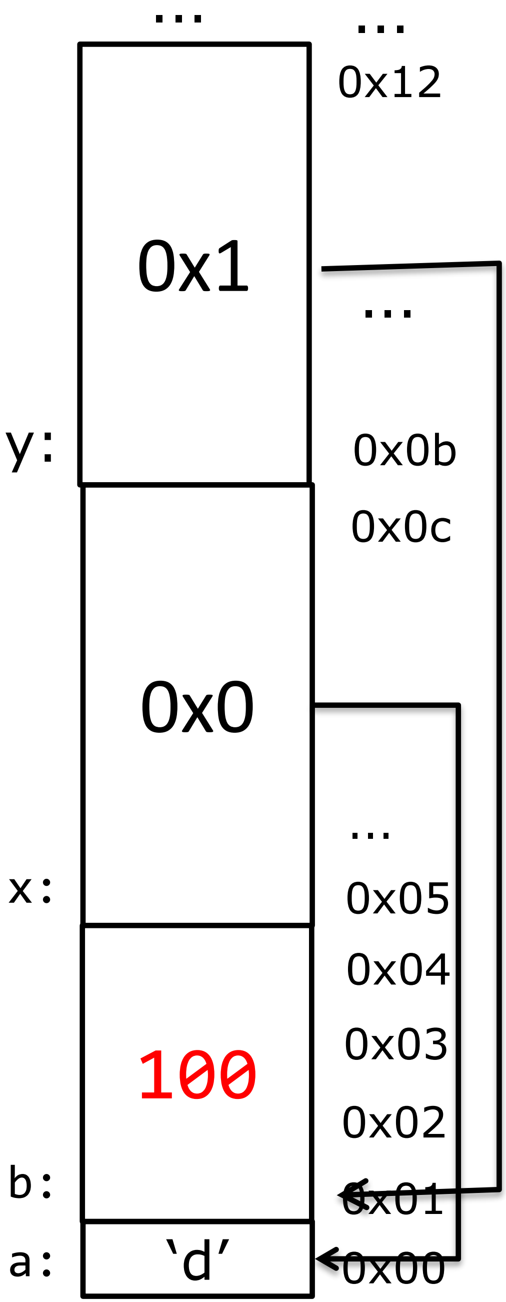
Pointer



```
char a = ‘c’;
int b = 2;
char *x = &a;
int *y = &b;

 *x = ‘d’;
```

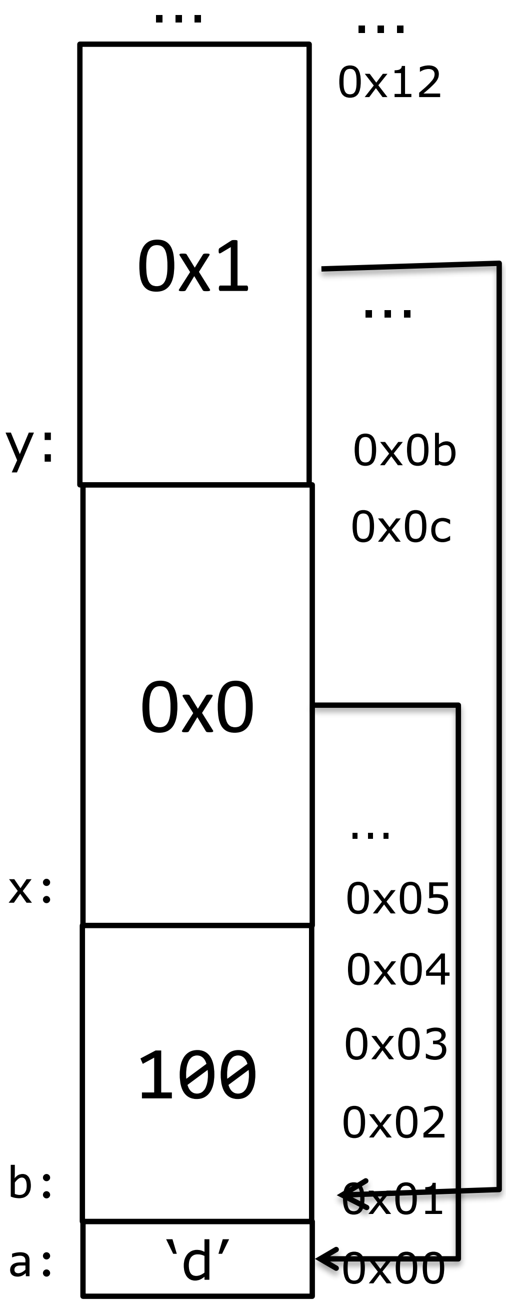
* operator dereferences a pointer

Pointer



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

 *x = 'd';
 *y = 100;
```

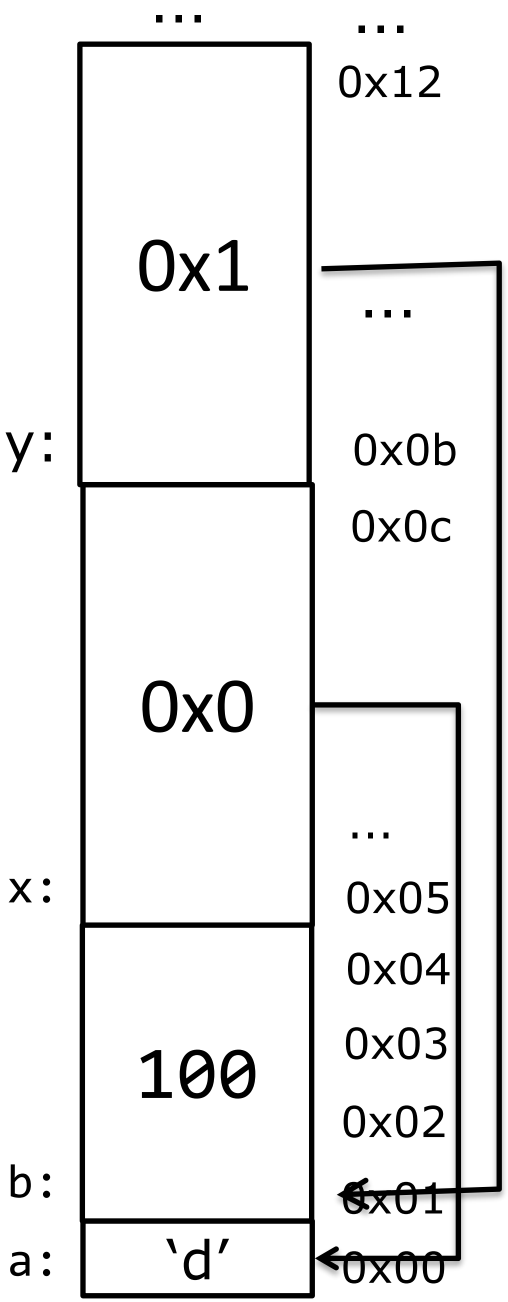
Pointer



```
char a = ‘c’;
int b = 2;
char *x = &a;
int *y = &b;

 *x = ‘d’;
 *y = 100;

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

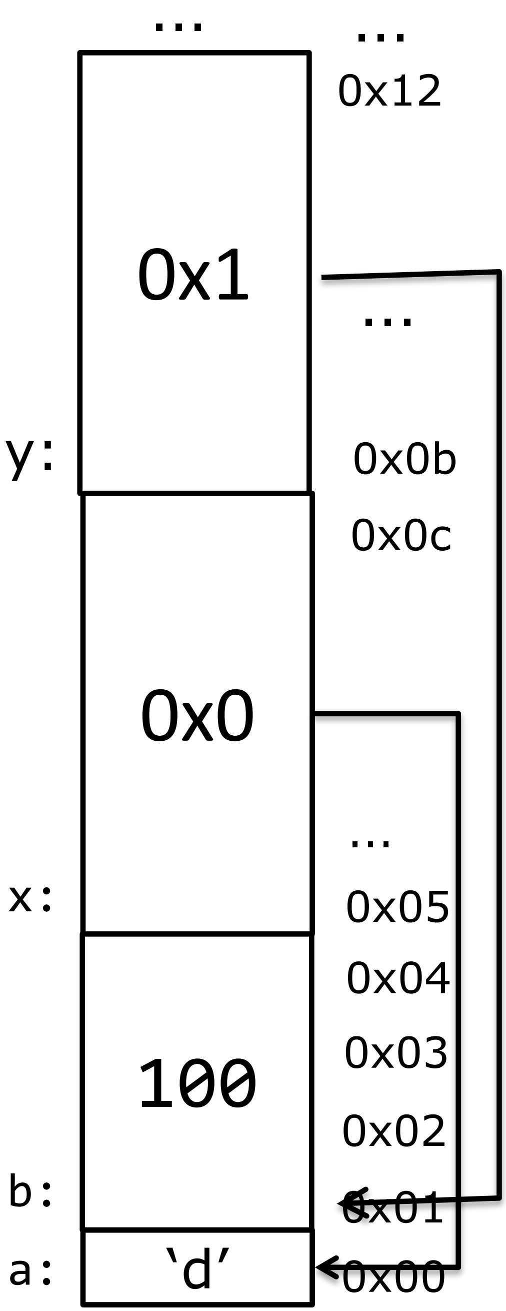
Value of xx, yy?

Pointer



```
char a = ‘c’;
int b = 2;
char *x = &a;
int *y = &b;

 *x = ‘d’;
 *y = 100;

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

xx: 0x5, yy: 0x0b



Pointer

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

 *x = 'd';
 *y = 100;

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

type and value of &xx?

Type: char ***
Value: ??? (location of variable xx)

Pass pointers to function

Pass the copies

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

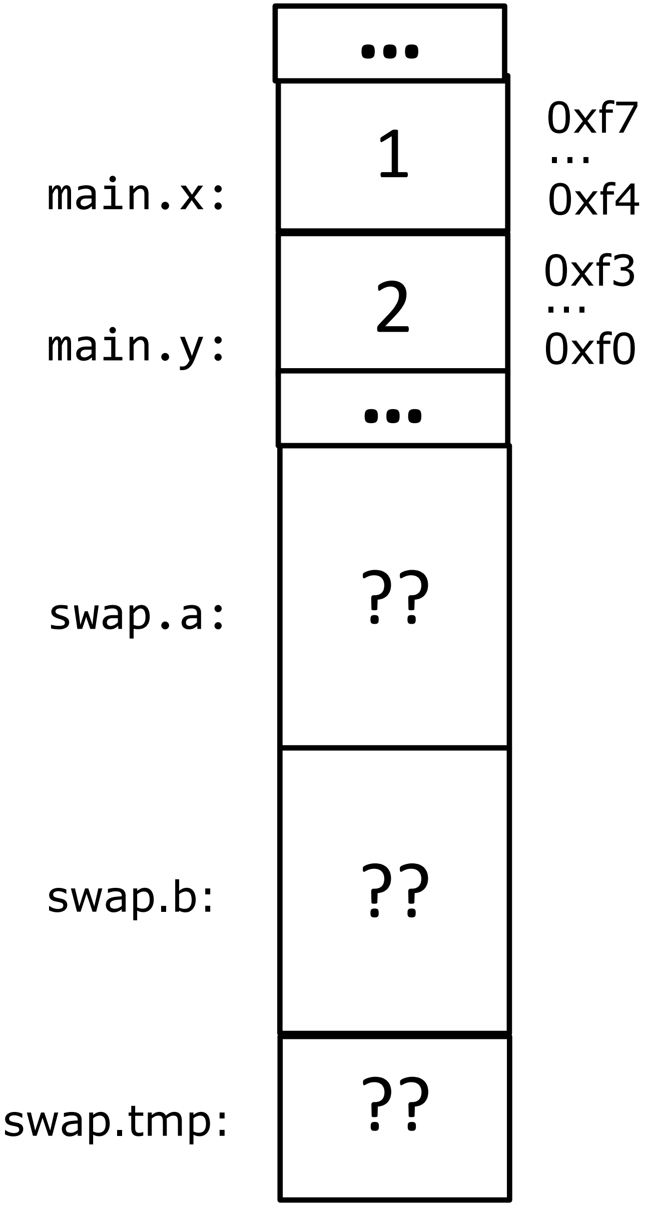
Pass pointers to function

Pass the pointers

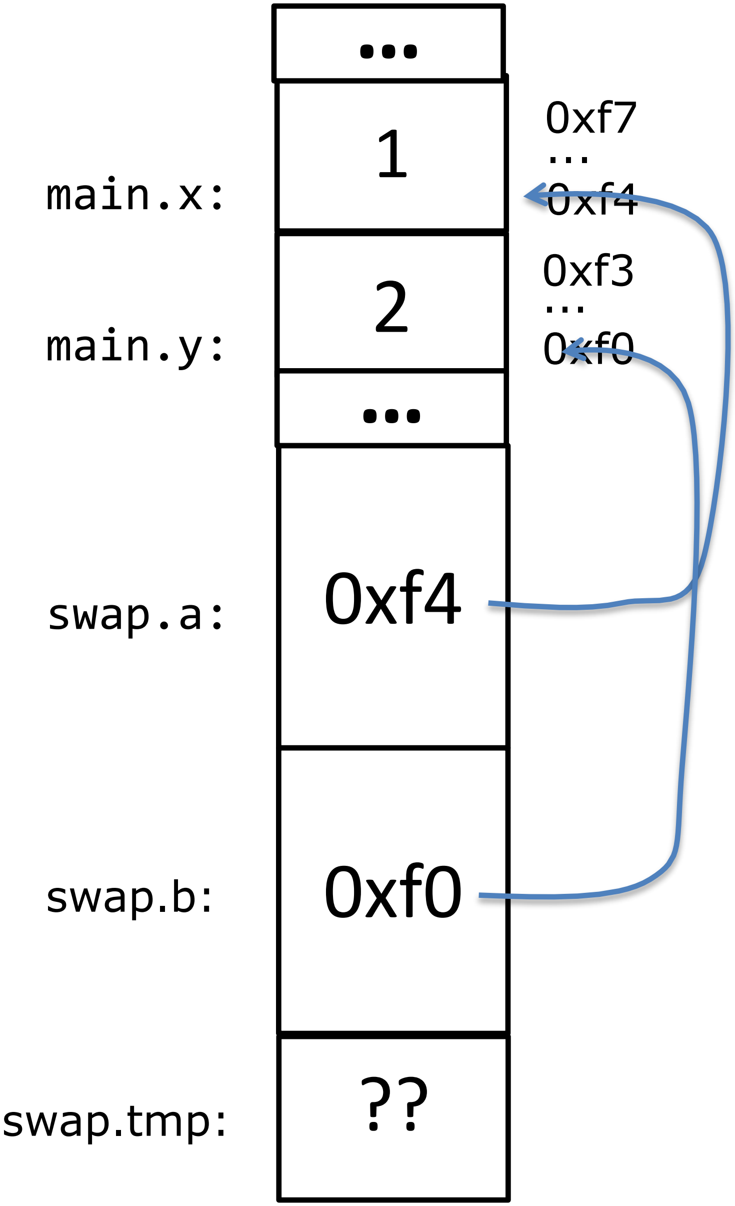
```
void swap(int *a, int *b)
{
     int tmp = *a;
      *a = *b;
      *b = tmp;
}
```

```
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);
}
```

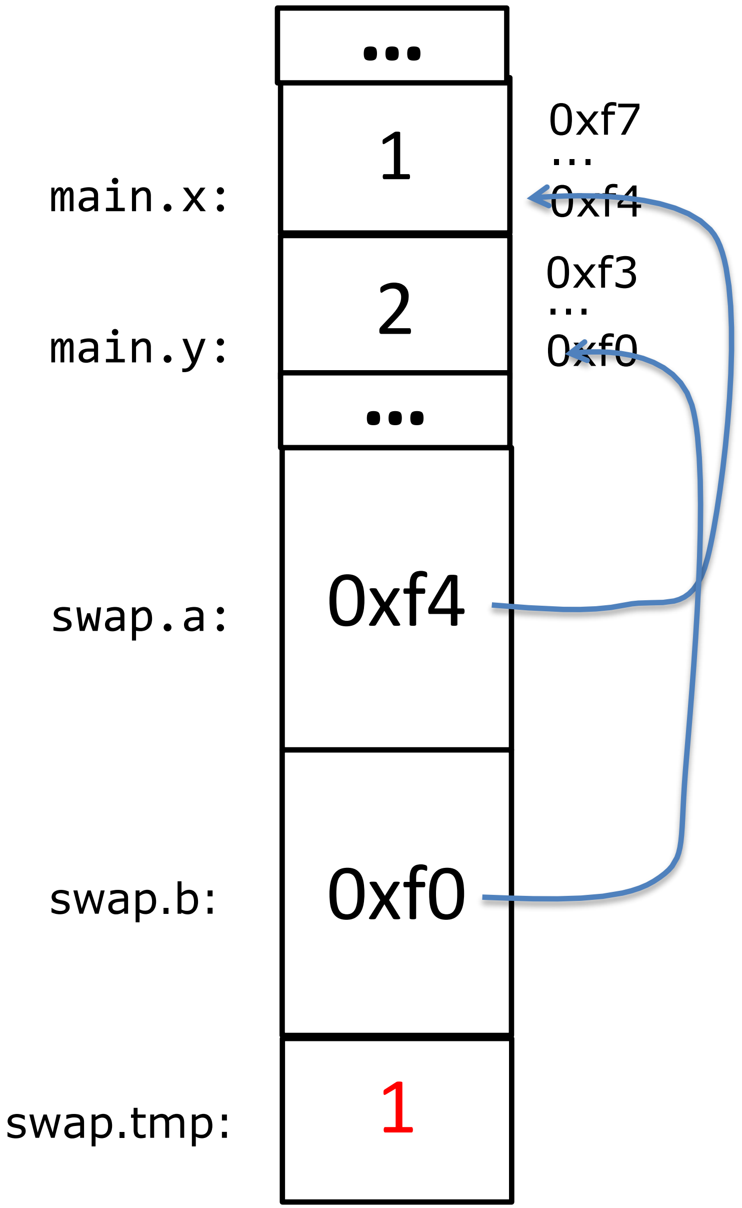
Size and value of a, b, tmp upon function entrance?



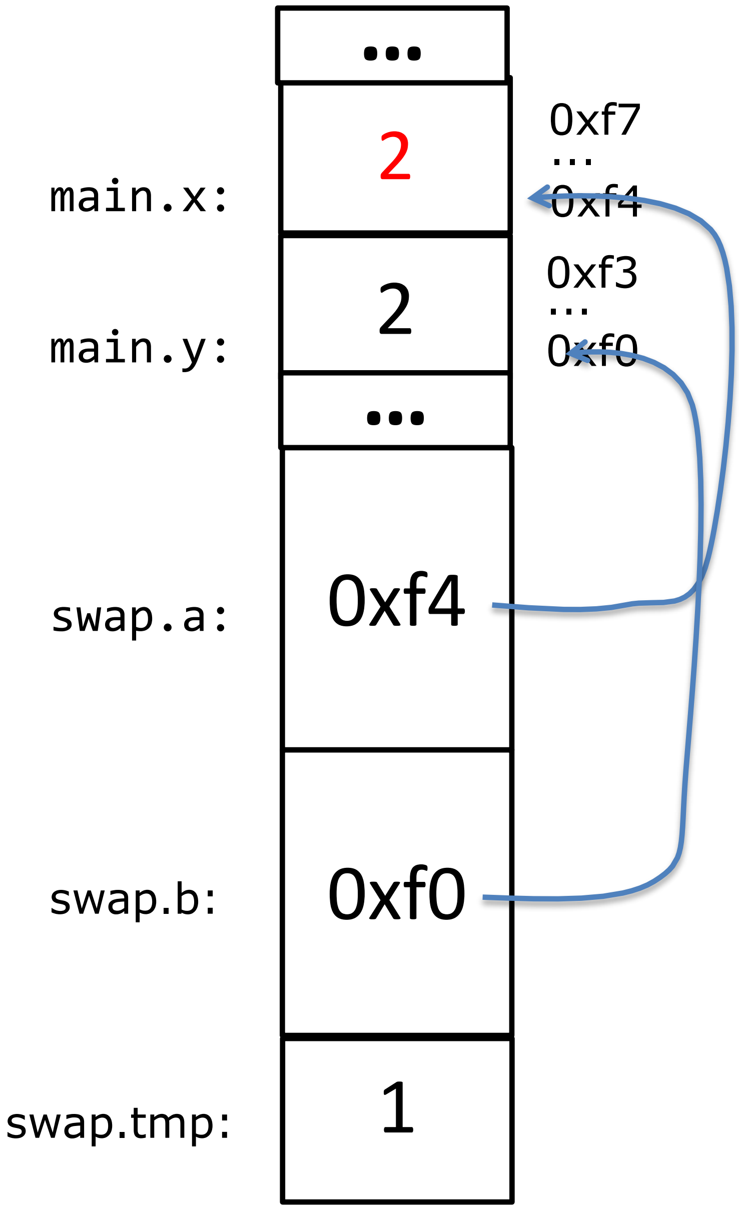
```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main()
{
    int x = 1;
    int y = 2;
    swap(&x, &y);

    printf(“x:%d, y:%d”,x,y);
}
```



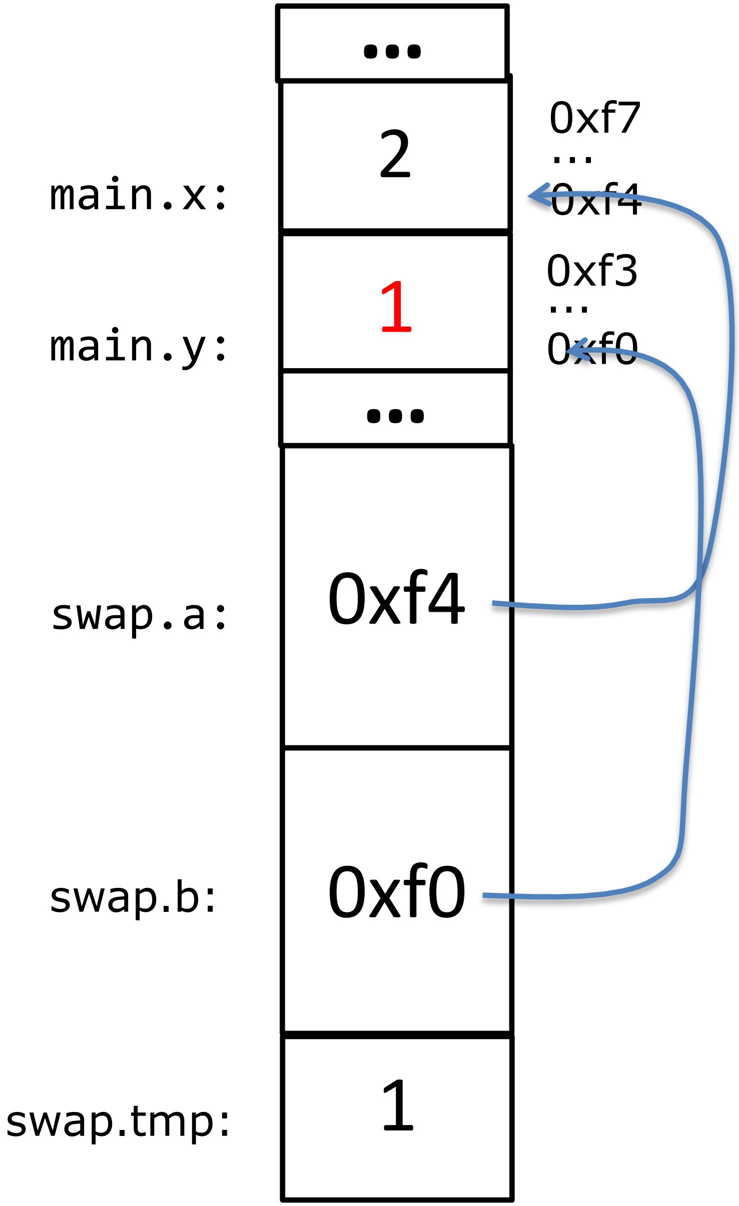
```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);

   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);
}
```



…

main.x: 2

0xf7
…
0xf4

main.y: 1

0xf3
…
0xf0

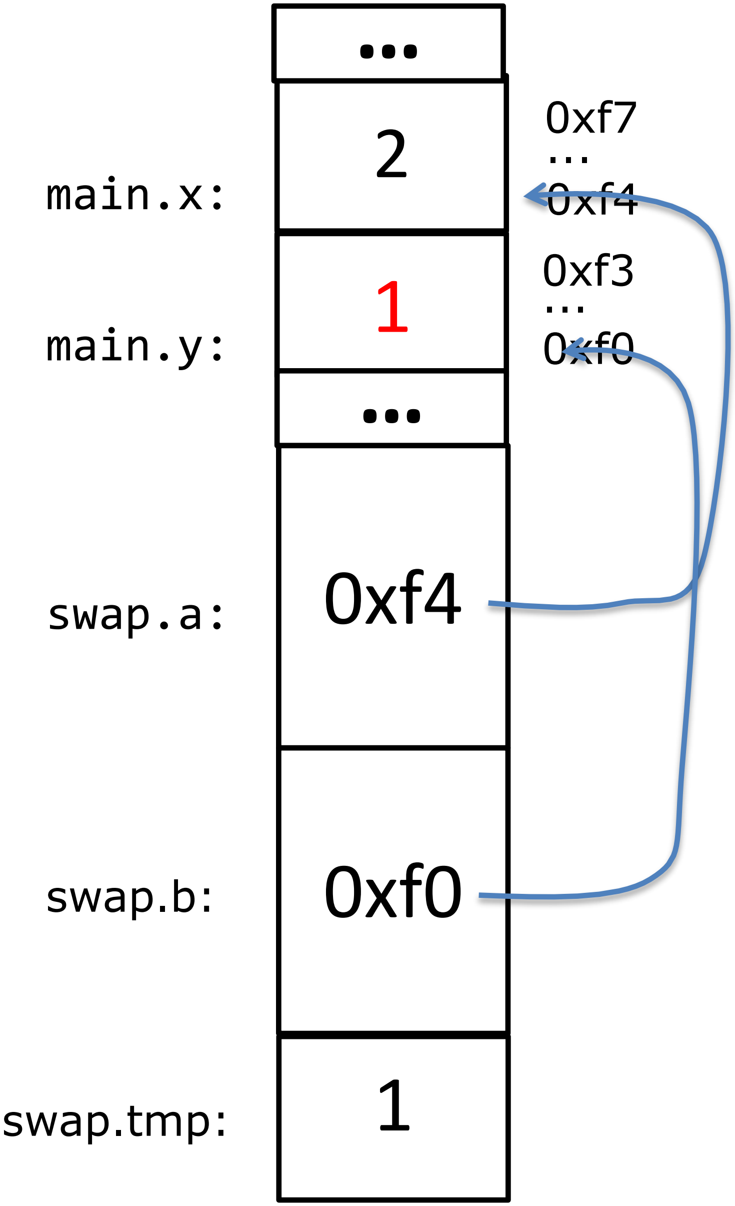
…

swap.a: 0xf4

swap.b: 0xf0

swap.tmp: 1

```
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: 2

0xf7
…
0xf4

main.y: 1

0xf3
…
0xf0

…

swap.a: 0xf4

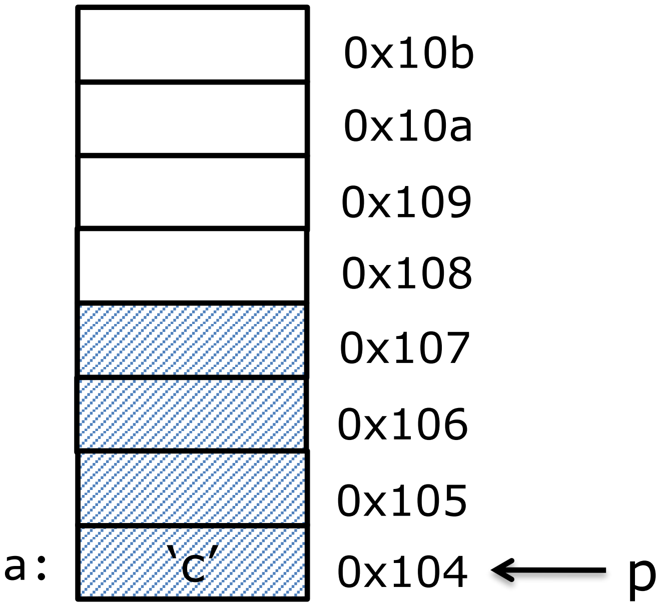
swap.b: 0xf0

swap.tmp: 1

Pointer arithmetic

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

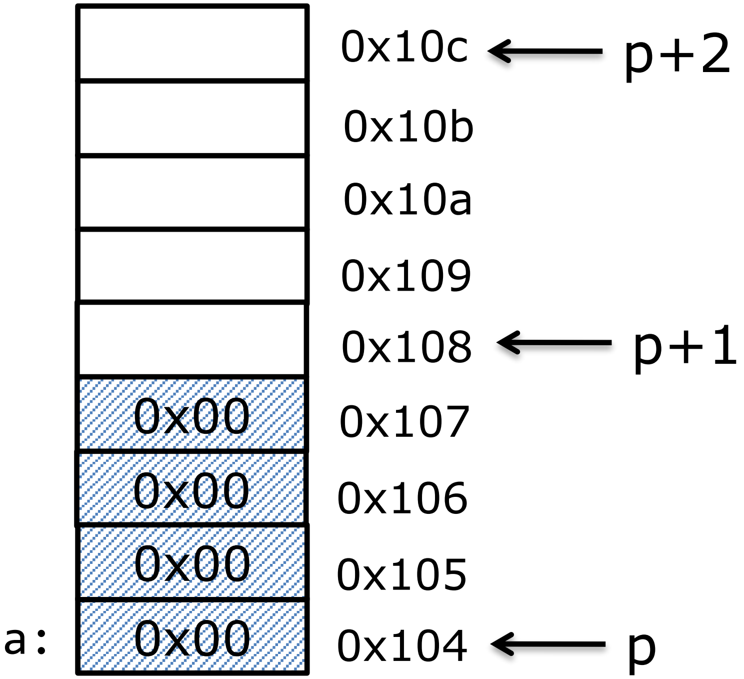
+	p+1	Point to the next object with int (4 bytes next to current location)	



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

Array

array is a collection of contiguous objects with the same type

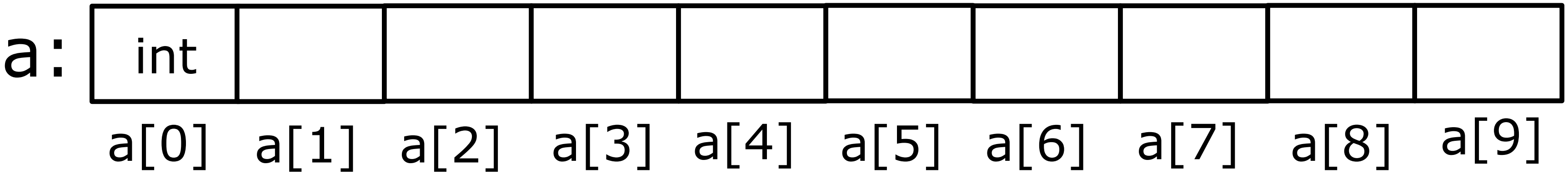
Array

Strong relationship with pointer

- Any operation achieved by array’s subscripting can also be done with pointers.

A block of n consecutive objects

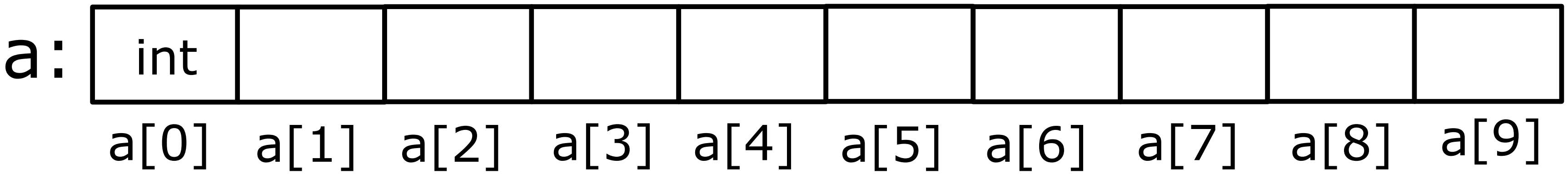
- int a[10]



a: int

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

Array

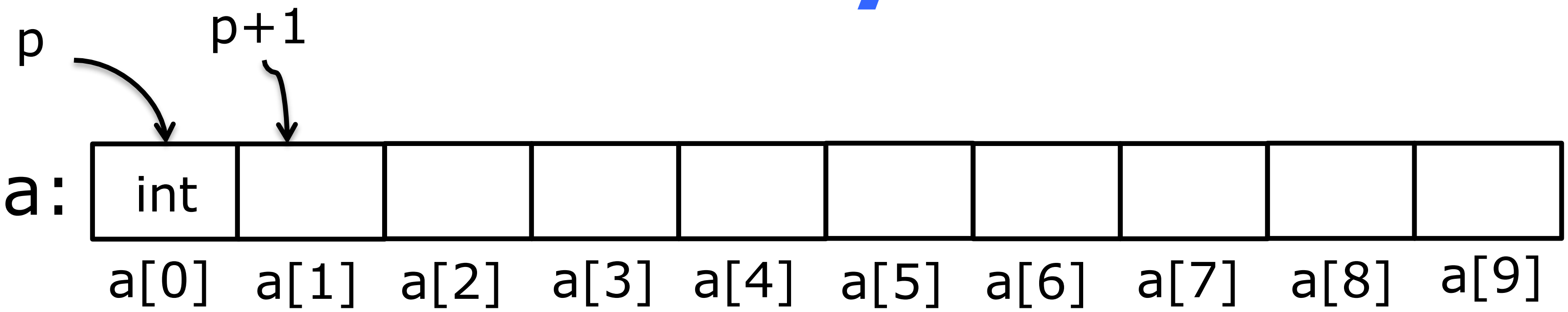


a: int

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

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

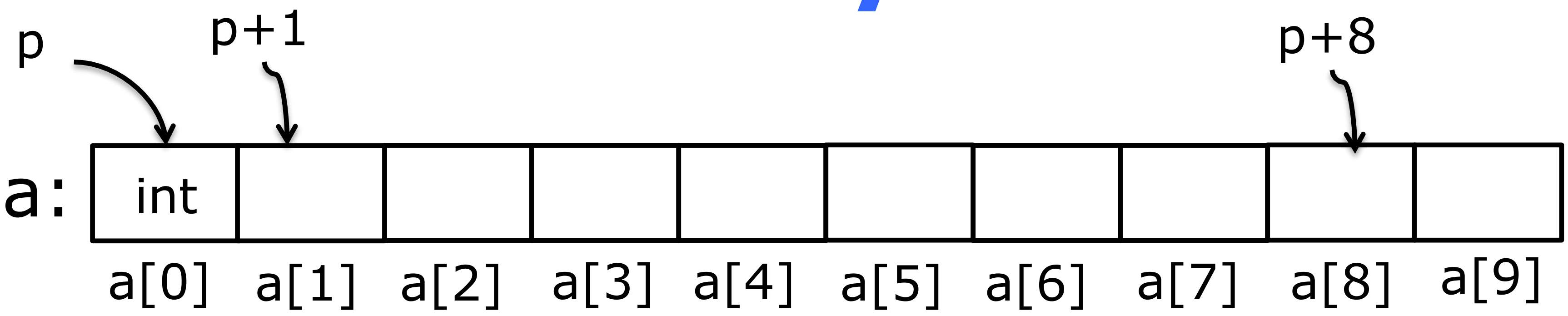
Array



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]

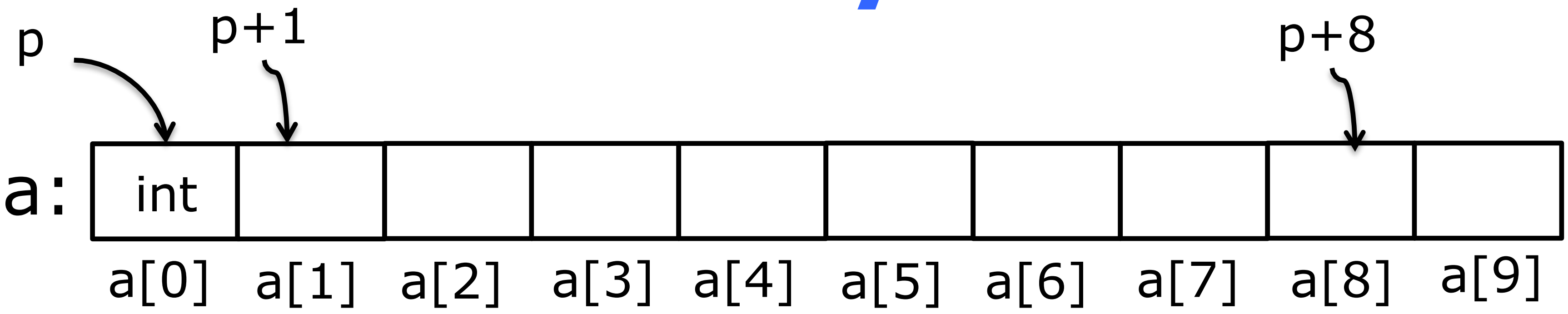
Array



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

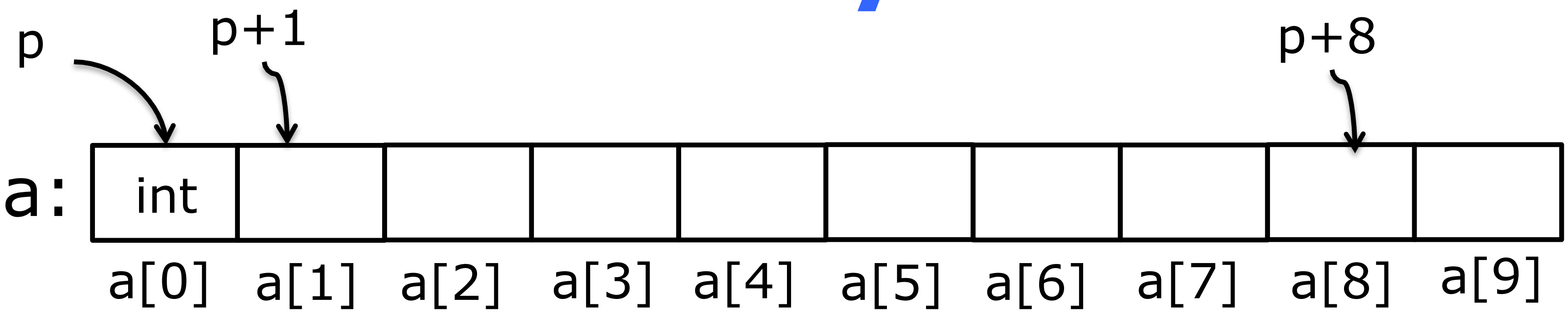


length of a[0]: 4 bytes → a[1] is 4 bytes next to a[0]

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]

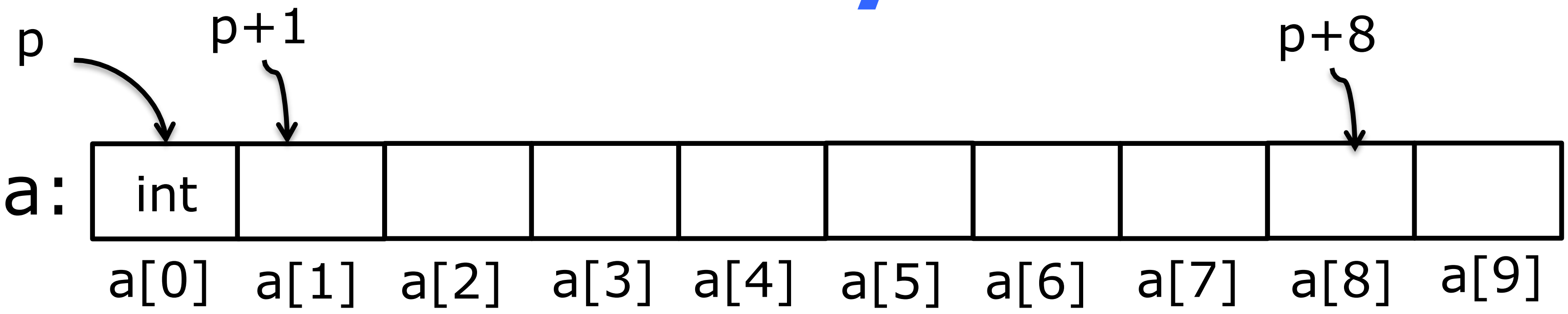
int *p = &a[0] → p+1 points to a[1]
→ p + i points to a[i]

int *p = a ⟷ int *p = &a[0]

p++ ✓

a++ ✗ compilation error

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]

p++

p++
expression’s value is
p before the increment

```
#include <stdio.h>

int main() {
  int a[3] = {100, 200, 300};
  int *p = a;

  printf("val1: %d\n", *(p++));
  printf("val2: %d\n", *(p));

}
```

p++

p++
expression’s value is
p before the increment

```
#include <stdio.h>

int main() {
  int a[3] = {100, 200, 300};
  int *p = a;

  printf("val1: %d\n", *(p++));
  printf("val2: %d\n", *(p));

}
```

```
val1: 100
val2: 200
```

++p

```
++p
expression’s value is
p after the increment
```

++p

++p
expression’s value is
p after the increment

```
#include <stdio.h>

int main() {
  int a[3] = {100, 200, 300};
  int *p = a;

  printf("val1: %d\n", *(++p));
  printf("val2: %d\n", *(p));

}
```

++p

++p
expression’s value is
p after the increment

```
#include <stdio.h>

int main() {
  int a[3] = {100, 200, 300};
  int *p = a;

  printf("val1: %d\n", *(++p));
  printf("val2: %d\n", *(p));

}
```

```
val1: 200
val2: 200
```

Another crazy example

```c
#include <stdio.h>

int main() {
  int a[3] = {100, 200, 300};
  int *p = a;

  printf("val of %d %d %d\n", *(p++), *(++p), *(p));
}
```

```
100 300 300
```

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;
   }
}

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

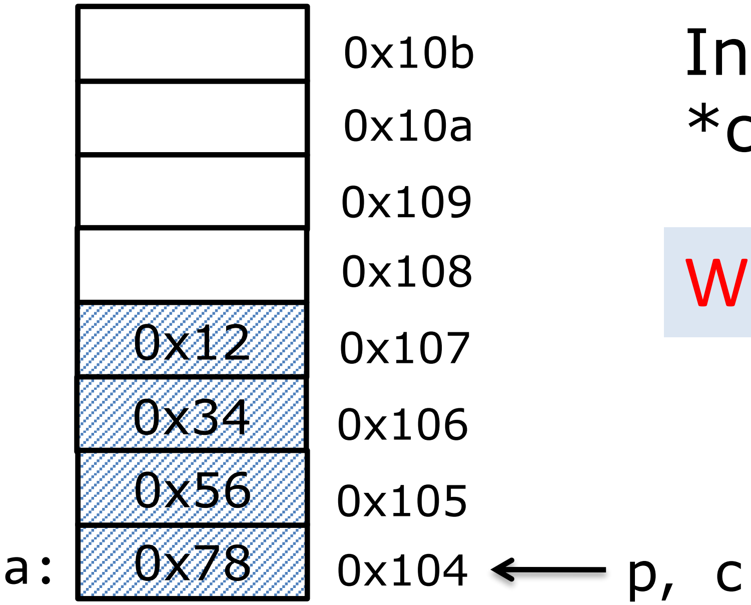
Pointer casting

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

What are the values of *c in hex ?
(Intel laptop)

Pointer casting

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



0x10b

0x10a

0x109

0x108

0x12 0x107

0x34 0x106

0x56 0x105

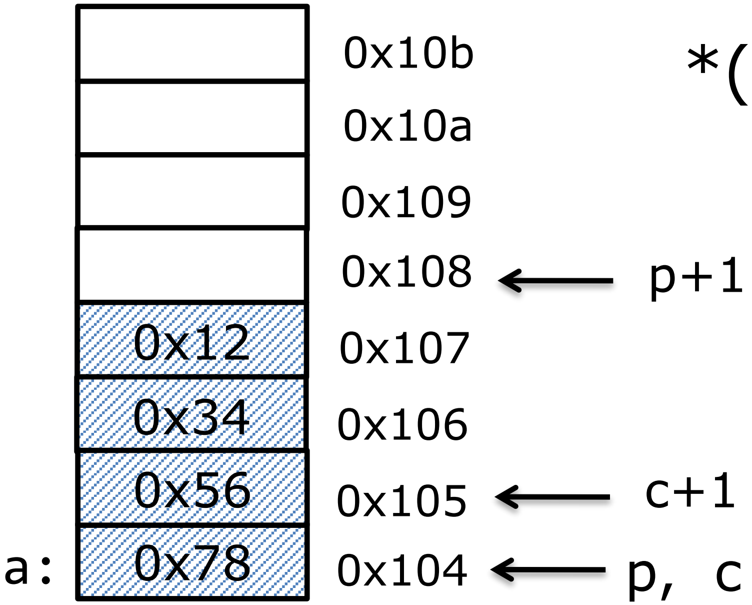
a: 0x78 0x104 ← p, c

Intel laptop is small endian
*c is 0x78

What is c+1? p+1?

Pointer casting

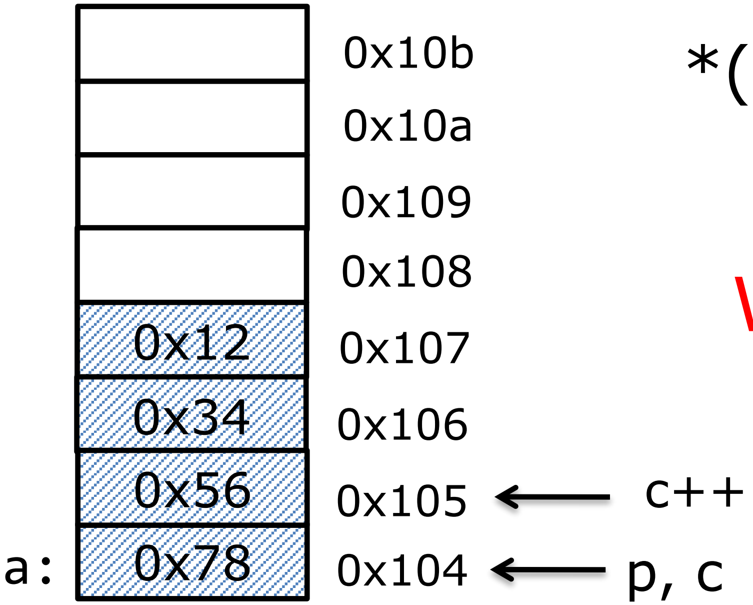
```
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;
```



*(c+1) is 0x56

Pointer casting

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



*(c+1) is 0x56

What about big endian?

function *sizeof*

sizeof(type)

- – Returns size in bytes of the object representation of type

sizeof(expression)

- – Returns size in bytes of the variable representation 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

Undefined behavior

In computer programming, undefined behavior (UB) is the result of executing computer code whose behavior is not prescribed by the language specification.

Classic undefined behaviors

- Use an uninitialized variable

```
int a;
int b = a + 1;
```

- out of bound array access

```
int a[2] = {1, 2};
int *p = a
*(p+3) = 3;
```

- Divide by zero

```
int a = 1 / 0;
```

- integer overflow

```
int a = 0x7fffffff
int b = a + 1
```

Why does C have undefined behavior?

Simplify compiler’s implementation

Enable better performance

Classic undefined behaviors

- Use uninitialized variables
  - Avoid memory write
- Out-of-bound array access
  - Avoid runtime bound checking
- Divided by zero

?

- integer overflow

Classic undefined behaviors

At instruction set level, different architectures handle them in different ways:

Divided by zero

- X86 raises an exception
- MIPS and PowerPC silently ignore it.

integer overflow

- X86 wraps around (with flags set)
- MIPS raises an exception.

Classic undefined behaviors

Assumption: Unlike Java, C compilers trust the programmer not to submit code that has undefined behavior

The compiler optimizes this code under this assumption

→ Compiler may remove the code or rewrite the code in a way that programmer did not anticipate

Classic undefined behaviors

```
#include <stdio.h>

void foo(int a) {
  if(a+100 < a) {
    printf(”overflowed\n");
     return;
  }

  printf(”normal is boring\n");
}

int main() {
  foo(100);
  foo(0x7fffffff);
}
```

Classic undefined behaviors

```
#include <stdio.h>

void foo(int a) {
  ~~if(a+100 < a) {~~
~~    printf(“overflowed\n");~~
~~     return;~~
~~  }~~

  printf(“normal is boring\n");
}

int main() {
  foo(100);
  foo(0x7fffffff);
}
```

gcc removes the check with O3

Recap pointer and array

```
int arr[3] = {1, 2, 3};
int *p = arr;
int *q = p + 1;
int **r = &p;
```

How many ways to access the 3rd element of the array arr?

Recap pointer and array

```
int arr[3] = {1, 2, 3};
int *p = arr;
int *q = p + 1;
int **r = &p
```

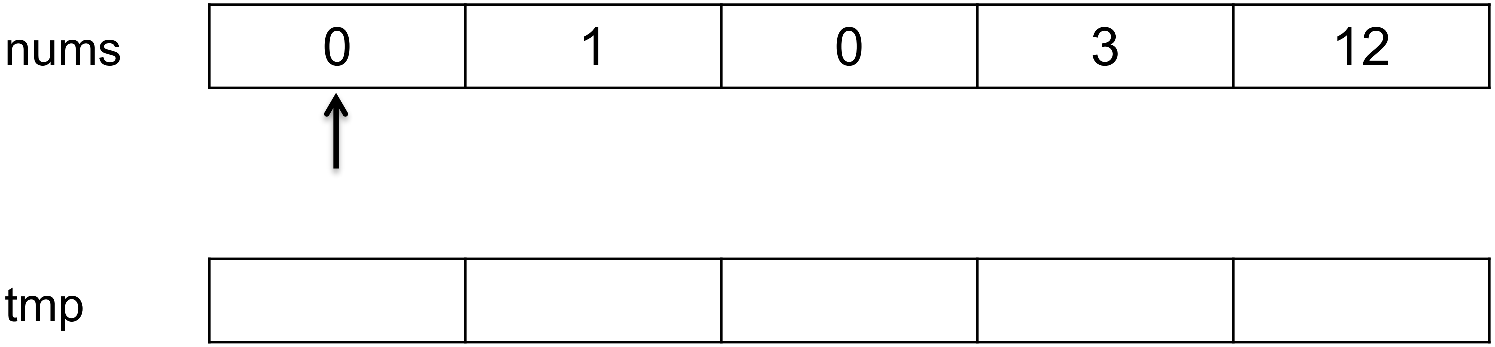
```
arr[2],  *(arr + 2),
p[2],    *(p + 2),
q[1],    *(q + 1),
(*r)[2], *(*r + 2)
```

Exercise

Move zeros

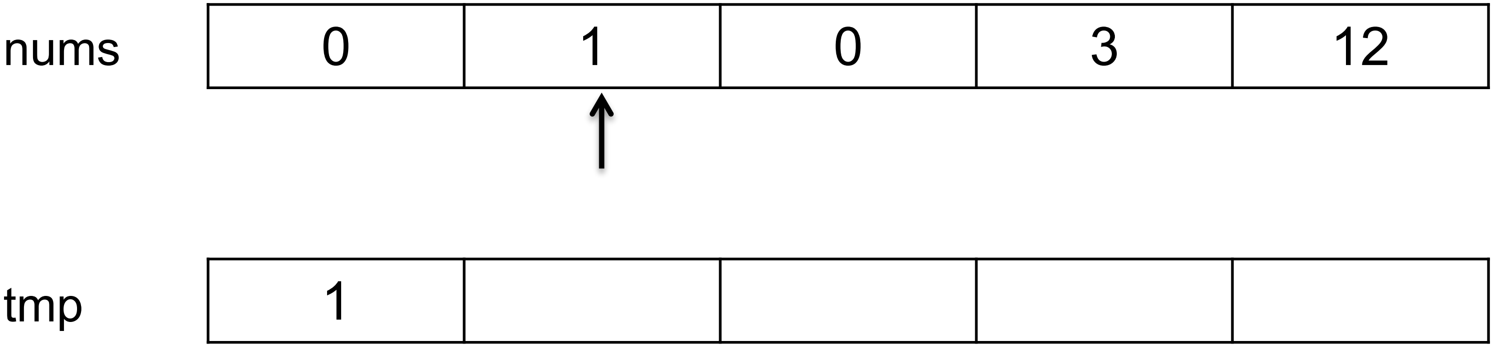
- Given an int array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.
- For example, given nums = [0, 1, 0, 3, 12], after calling your function, nums should be [1, 3, 12, 0, 0]
- Assume you can dynamically allocate an int array with function dynamic_alloc(n):
  - *int* dynamic_alloc(int len)*

Solution I



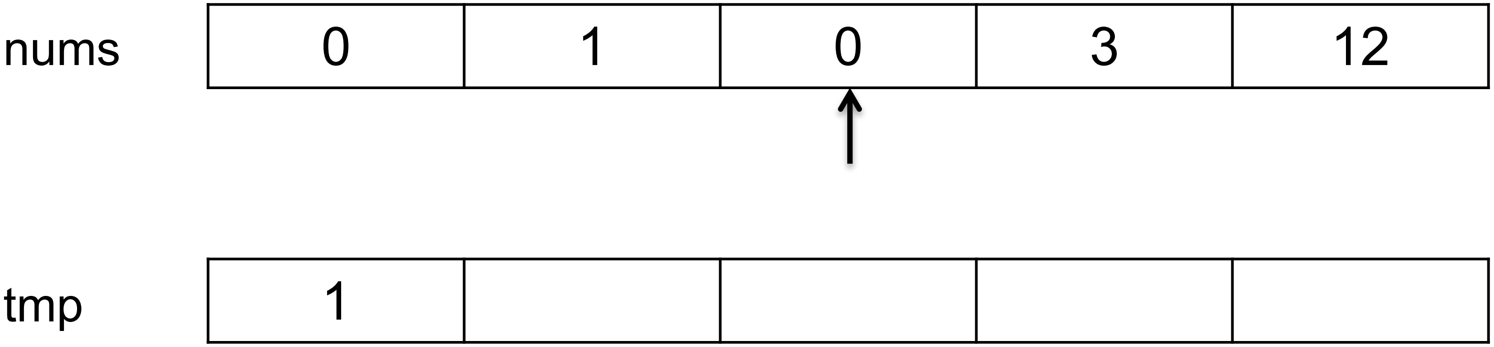
nums	0	1	0	3	12
tmp					

Solution I



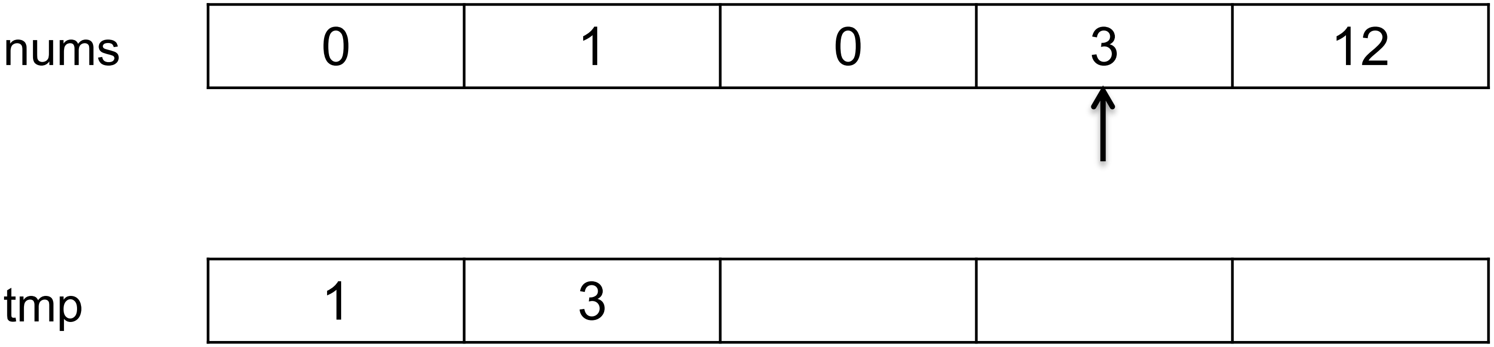
nums	0	1	0	3	12
tmp	1				

Solution I



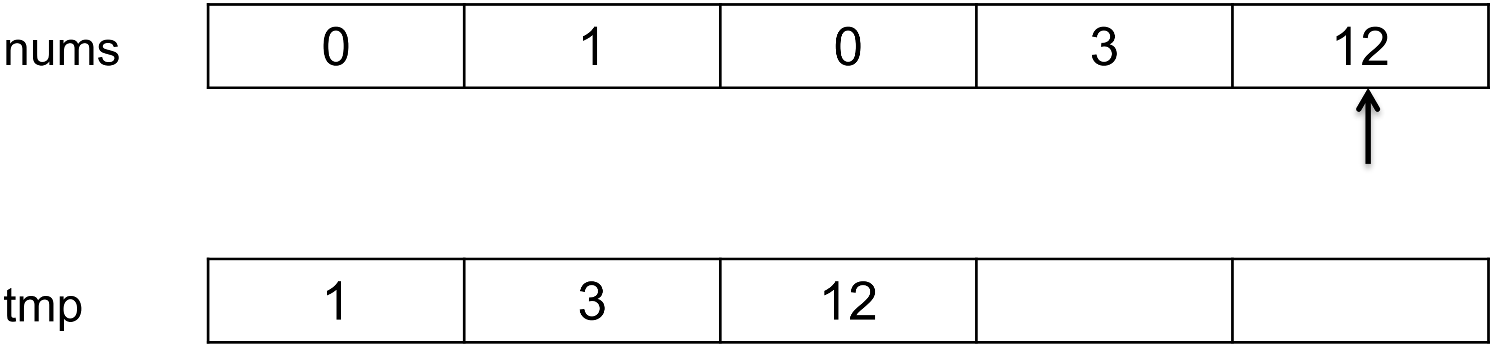
nums	0	1	0	3	12
tmp	1				

Solution I



nums	0	1	0	3	12
tmp	1	3			

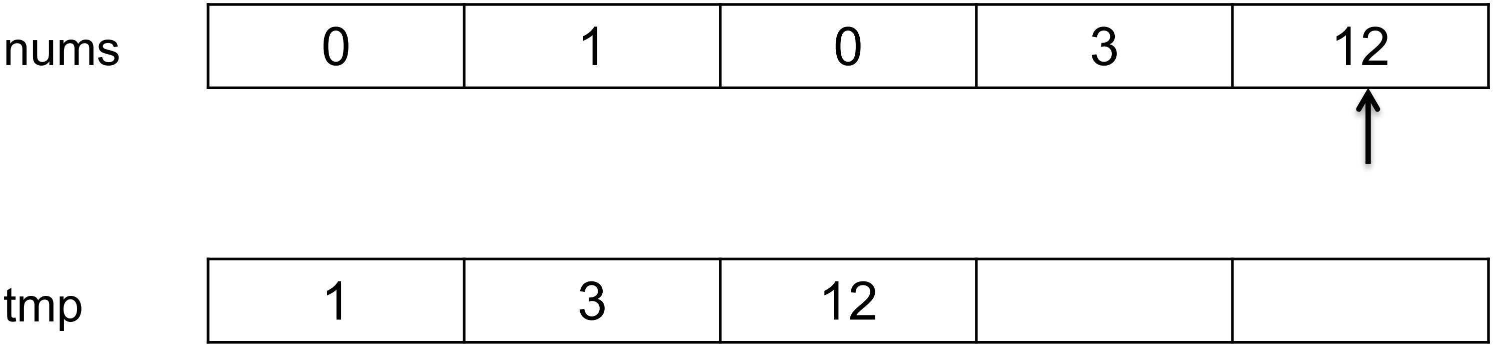
Solution I



nums	0	1	0	3	12

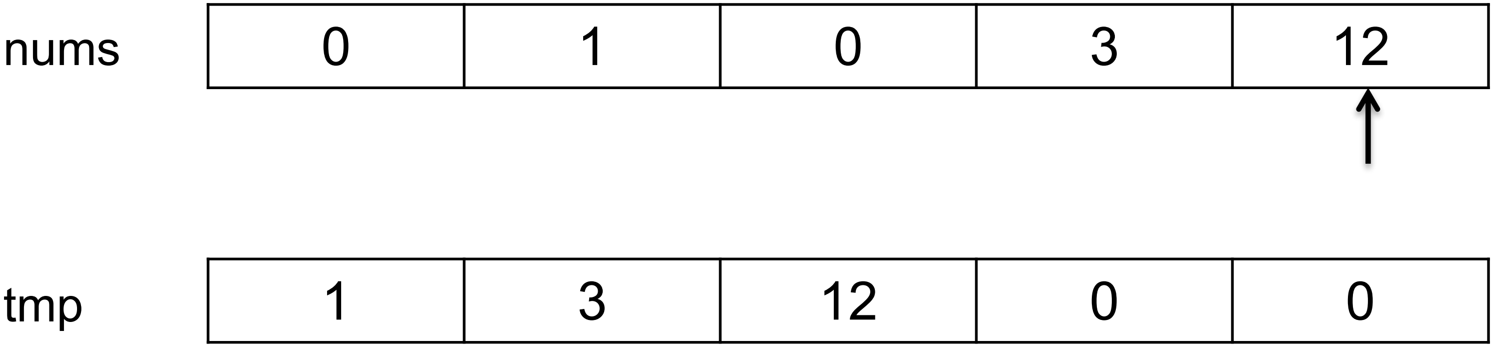
tmp	1	3	12		

Solution I



nums	0	1	0	3	12
tmp	1	3	12		

Solution I



nums	0	1	0	3	12
tmp	1	3	12	0	0

Solution I

nums	1	3	12	0	0
tmp	1	3	12	0	0

Solution I

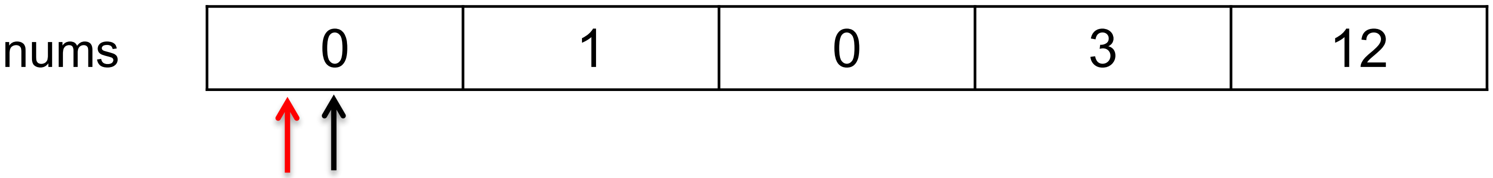
```
void moveZeroes(int* nums, int numsSize) {

    int* tmp = dynamic_alloc(numsSize);
    int index = 0;
    for(int i = 0; i < numsSize; i++){
         if(nums[i] != 0) {
            tmp[index] = nums[i];
            index = index + 1;
        }
    }
    for(int i = index; i <numsSize; i++) {
        tmp[i] = 0;
    }

    for(int i = 0; i < numsSize; i++) {
        nums[i] = tmp[i];
    }
}
```

Solution I

```
void moveZeroes(int* nums, int numsSize) {

    int* tmp = dynamic_alloc(numsSize);
    int index = 0;
    for(int i = 0; i < numsSize; i++){
         if(nums[i] != 0) {
            tmp[index] = nums[i];
            index = index + 1;
        }
    }
    for(int i = index; i <numsSize; i++) {
        tmp[i] = 0;
    }

    for(int i = 0; i < numsSize; i++) {
        nums[i] = tmp[i];
    }
}
```

Can we avoid dynamic extra space?

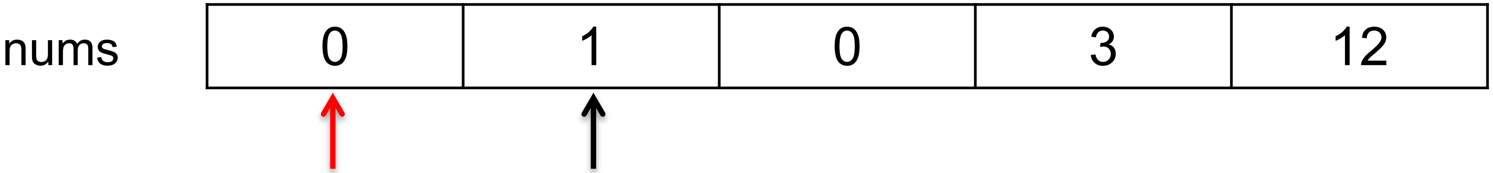
Solution II



nums	0	1	0	3	12

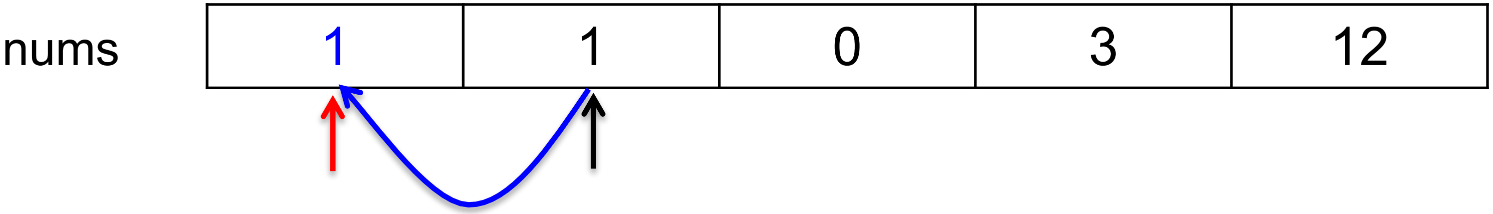
Black (fast): point to the next element to be checked
Red (slow): point to the next slot to be replaced

Solution II



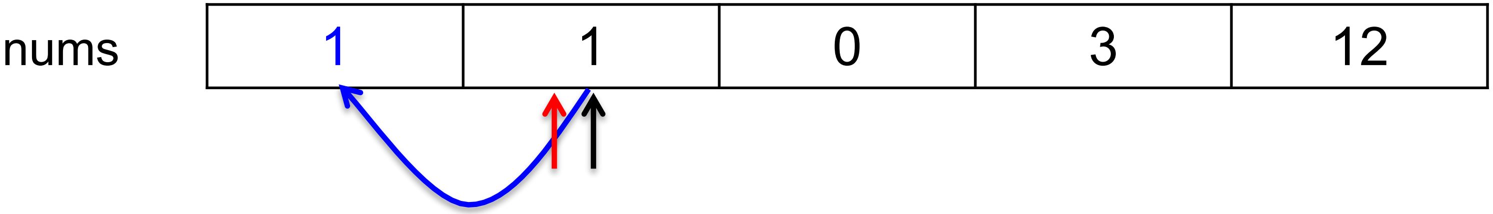
nums	0	1	0	3	12

Solution II



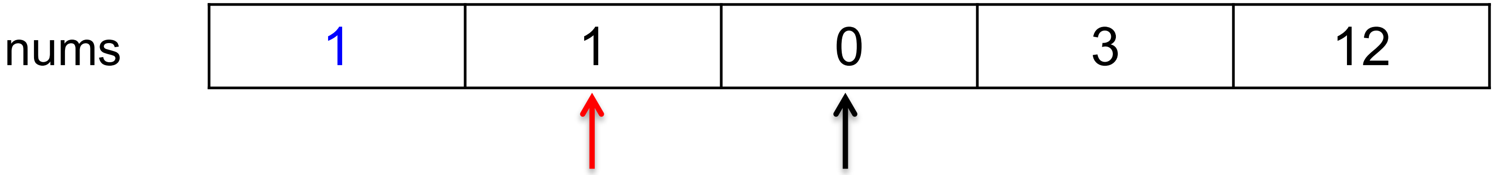
nums	1	1	0	3	12

Solution II



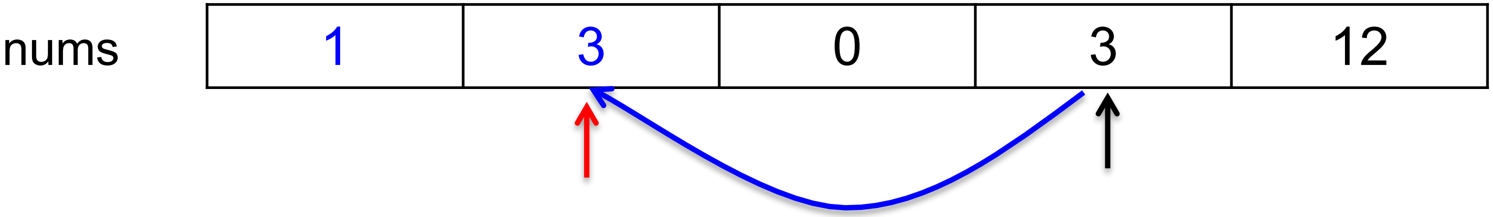
nums	1	1	0	3	12

Solution II



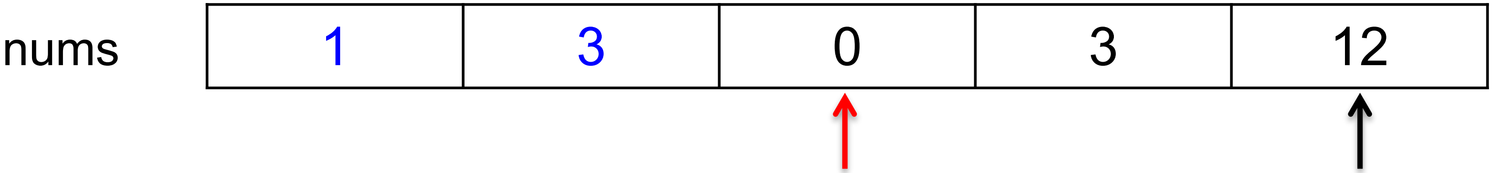
nums	1	1	0	3	12

Solution II



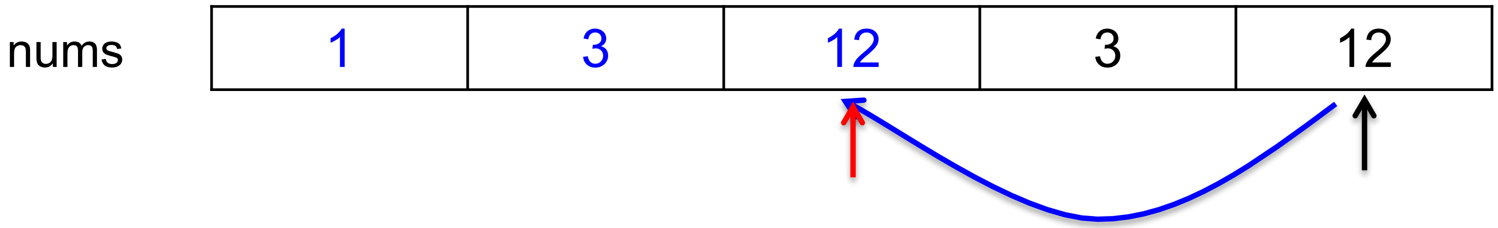
nums	1	3	0	3	12

Solution II



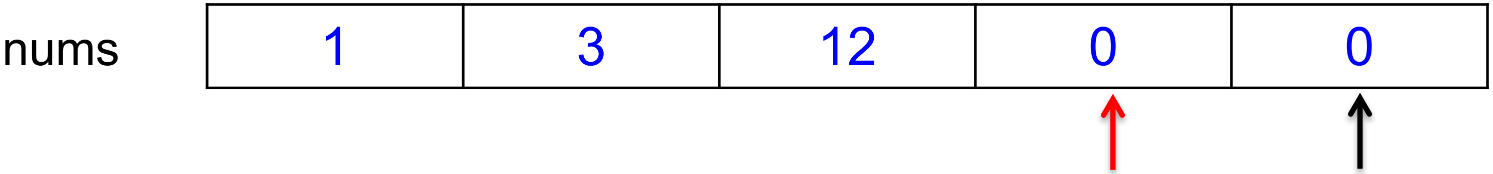
nums					
	1	3	0	3	12

Solution II



nums	1	3	12	3	12

Solution II

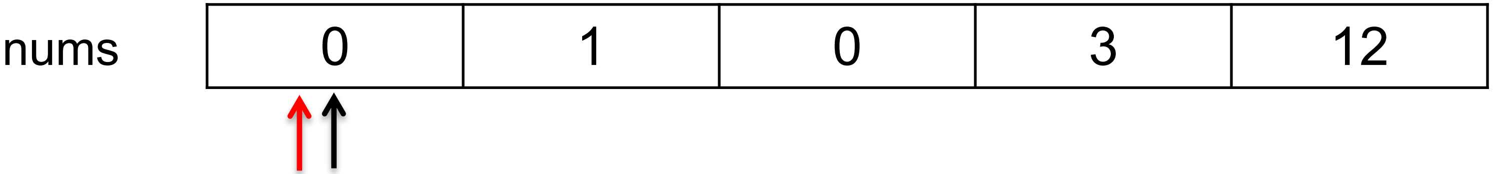


nums	1	3	12	0	0

Solution II

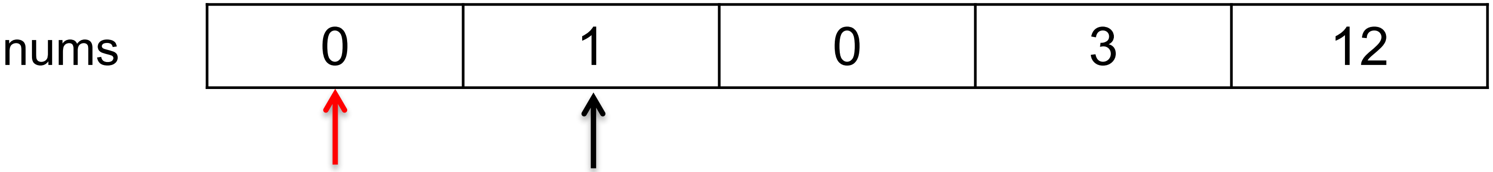
```
void moveZeroes(int* nums, int numsSize) {

    int nextReplace = 0;
    for (int i = 0; i < numsSize; i++) {
        if (nums[i] != 0) {
            nums[nextReplace++] = nums[i];
        }
    }

    for (int i = nextReplace; i < numsSize; i++) {
        nums[i] = 0;
    }

}
```

Solution III



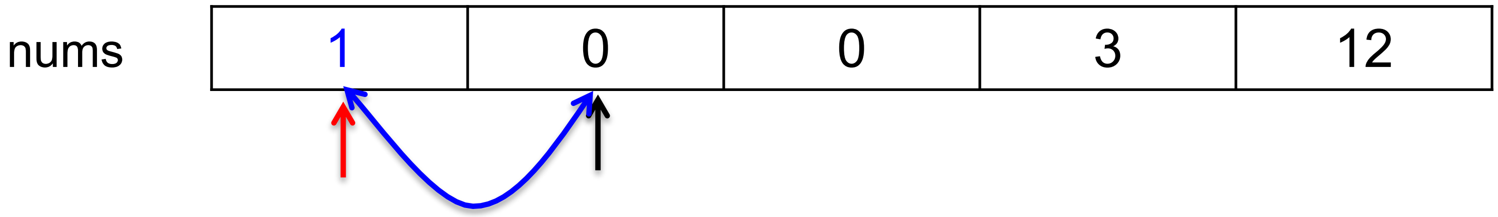
nums	0	1	0	3	12

Solution III



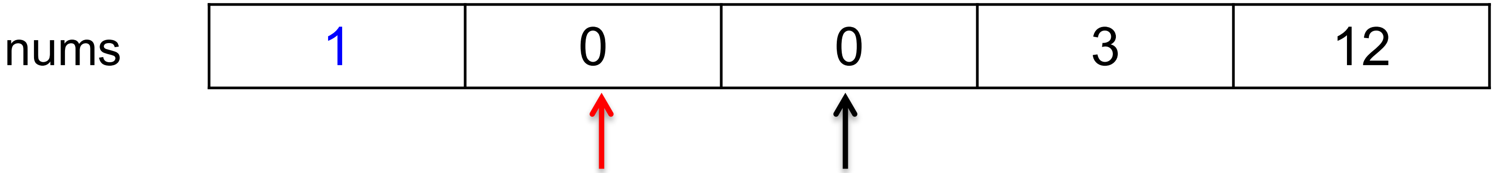
nums	0	1	0	3	12

Solution III



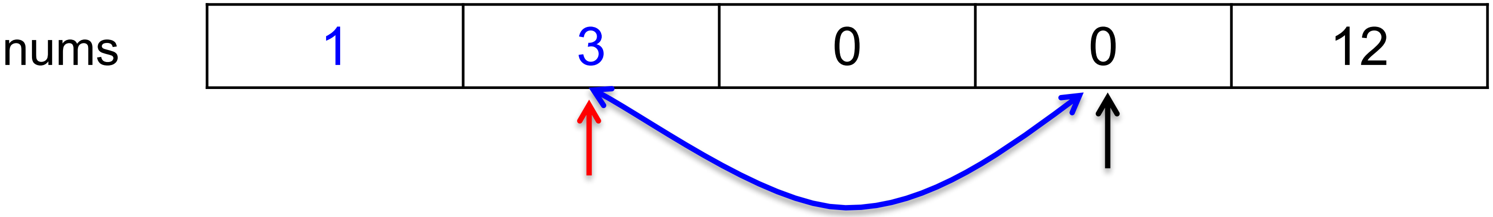
nums					
	1	0	0	3	12

Solution III



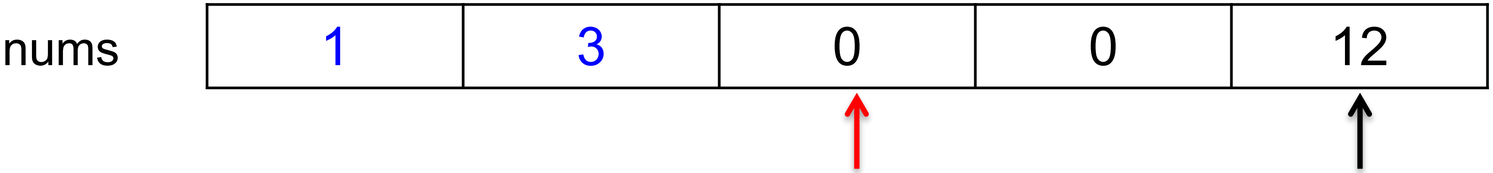
nums	1	0	0	3	12

Solution III



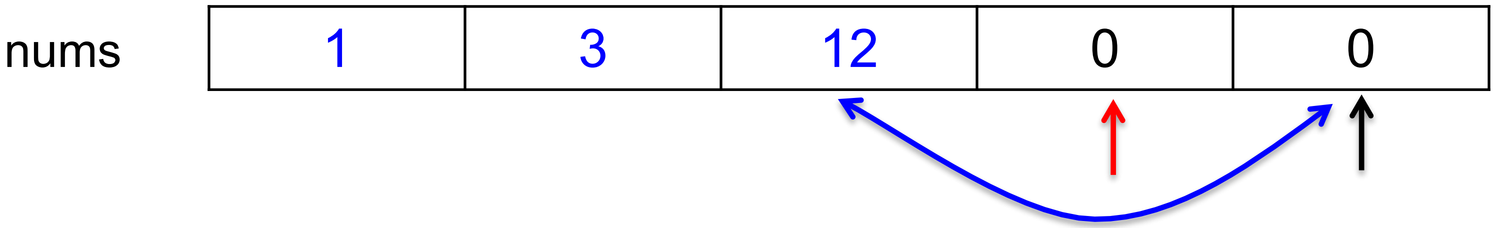
nums	1	3	0	0	12

Solution III



nums	1	3	0	0	12

Solution III



nums	1	3	12	0	0

Solution III

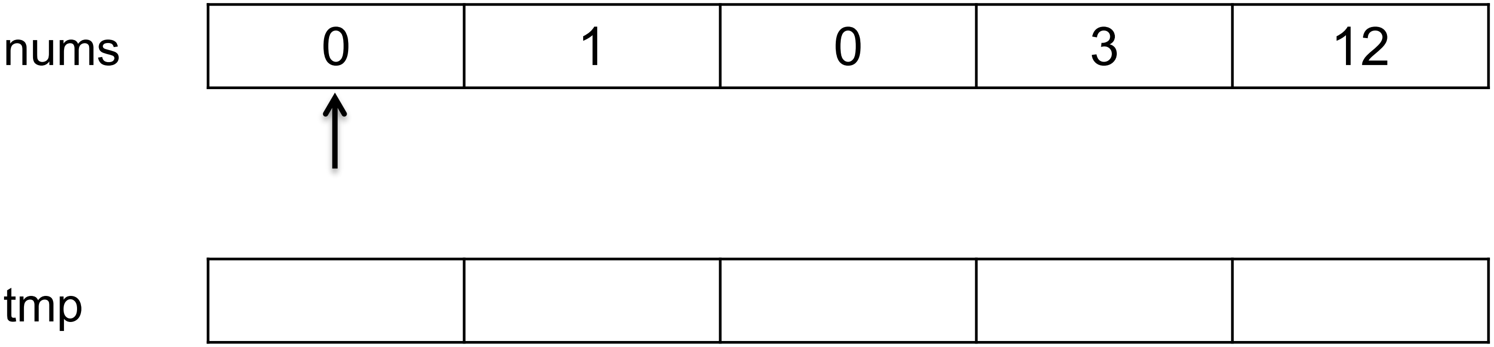
```
void moveZeroes(int* nums, int numsSize) {

    int nextSwap = 0;
    for (int i = 0; i < numsSize; i++) {
        if (nums[i] != 0) {
            swap(&nums[nextSwap++], &nums[i])
        }
    }
}
```

Exercise

Remove Elements

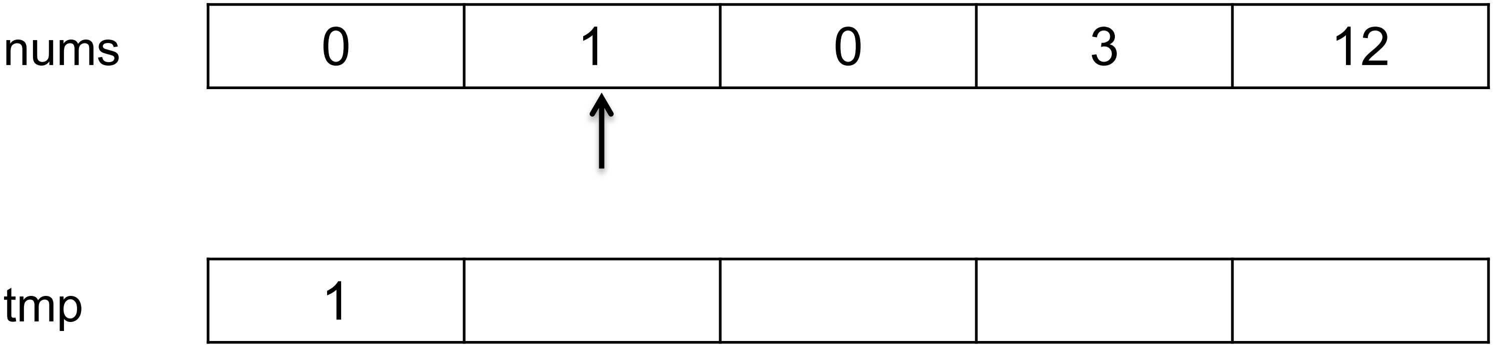
- Given an array and a value, remove all instances of that value in place and return the new length.
- For example, given nums = [0, 1, 0, 3, 12], value is 0 calling your function, nums should be [1, 3, 12, *, *] and 3
- Assume you can dynamically allocate an int array with function dynamic_alloc(n):
  - *int* dynamic_alloc(int len)*

Solution I



nums	0	1	0	3	12
tmp					

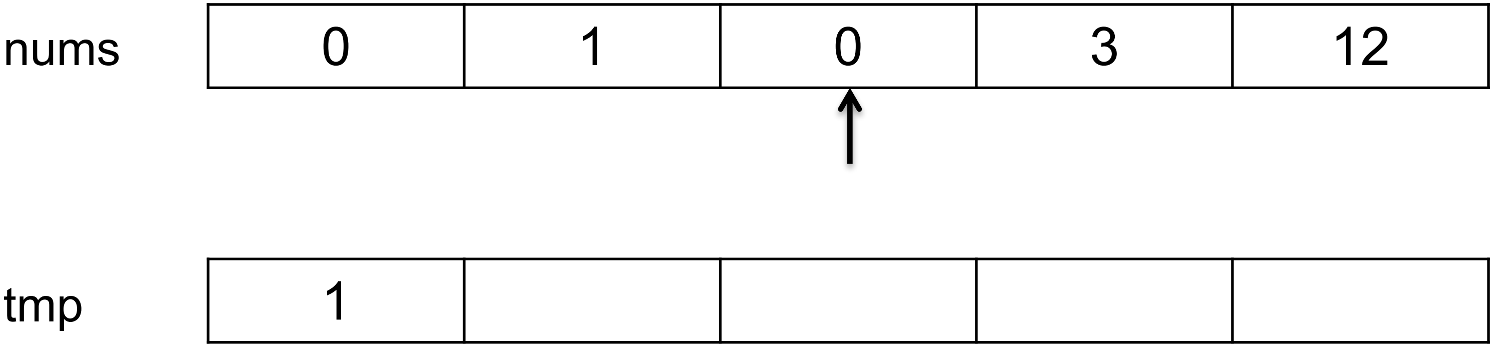
Solution I



nums	0	1	0	3	12

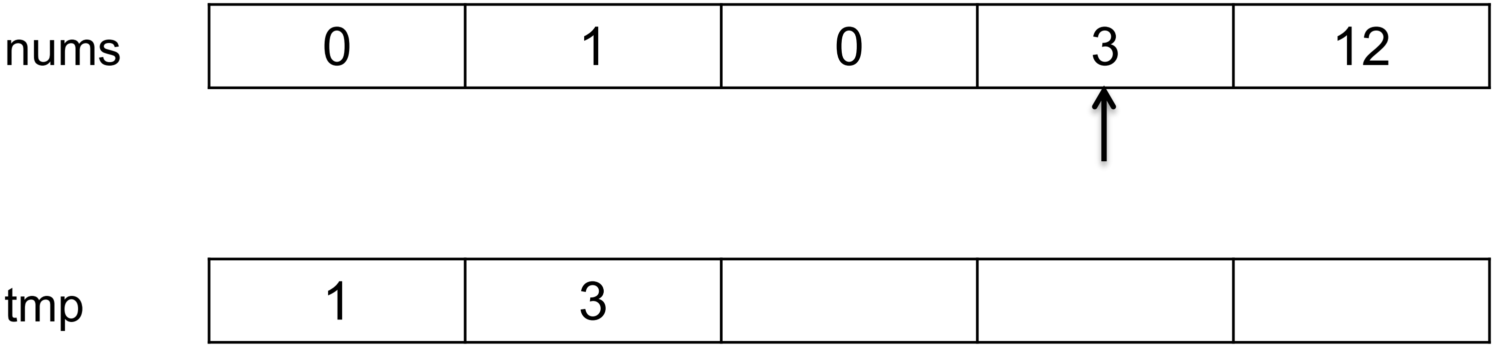
tmp	1				

Solution I



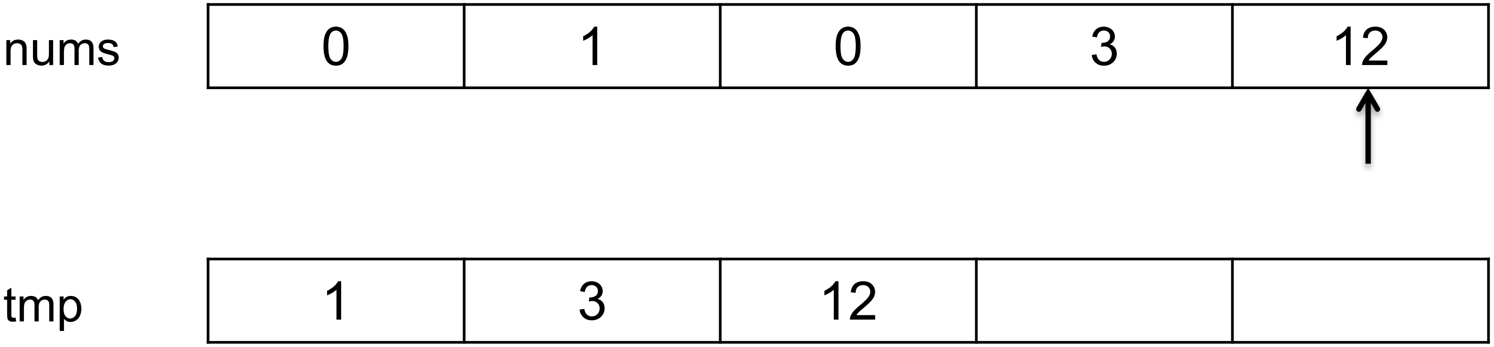
nums	0	1	0	3	12
tmp	1				

Solution I



nums	0	1	0	3	12
tmp	1	3			

Solution I



nums	0	1	0	3	12
tmp	1	3	12		

Solution I

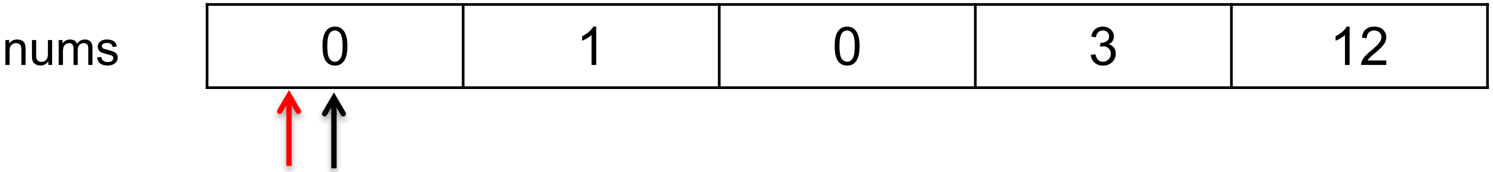
```
int remove(int* nums, int numsSize, int val) {

    int* tmp = dynamic_alloc(numsSize);
    int index = 0;
    for(int i = 0; i < numsSize; i++){
        if(nums[i] != val) {
            tmp[index] = nums[i];
            index = index + 1;
        }
    }

   for(int i = 0; i < index; i++) {
        nums[i] = tmp[i];
    }
    return index
}
```

Solution I

```
int remove(int* nums, int numsSize, int val) {

    int* tmp = dynamic_alloc(numsSize);
    int index = 0;
    for(int i = 0; i < numsSize; i++){
        if(nums[i] != val) {
            tmp[index] = nums[i];
            index = index + 1;
        }
    }

   for(int i = 0; i < index; i++) {
        nums[i] = tmp[i];
    }
    return index
}
```

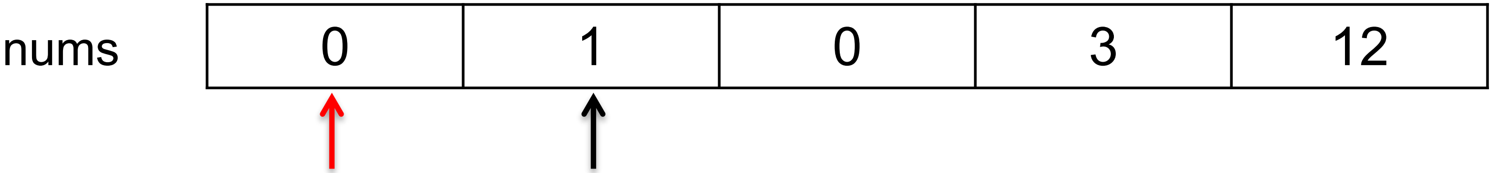
Can we avoid dynamic extra space?

Solution II



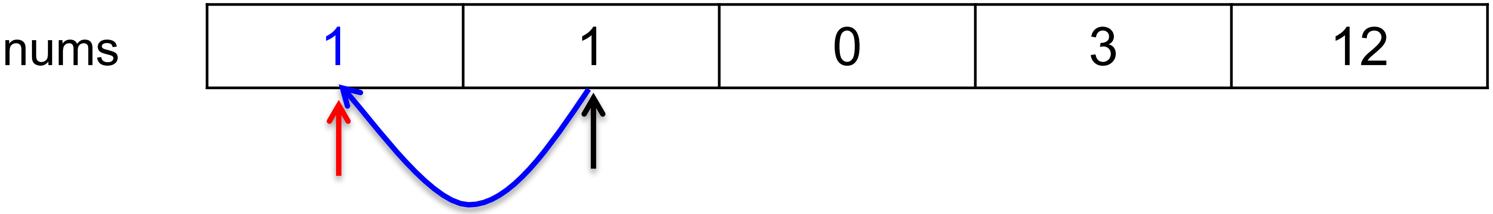
nums	0	1	0	3	12

Solution II



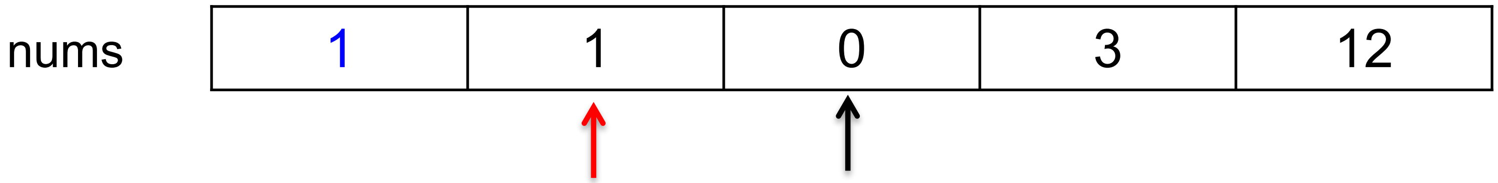
nums	0	1	0	3	12

Solution II



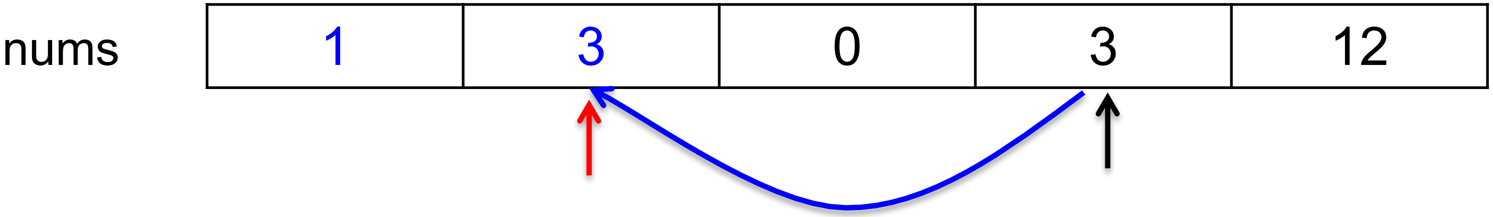
nums	1	1	0	3	12

Solution II



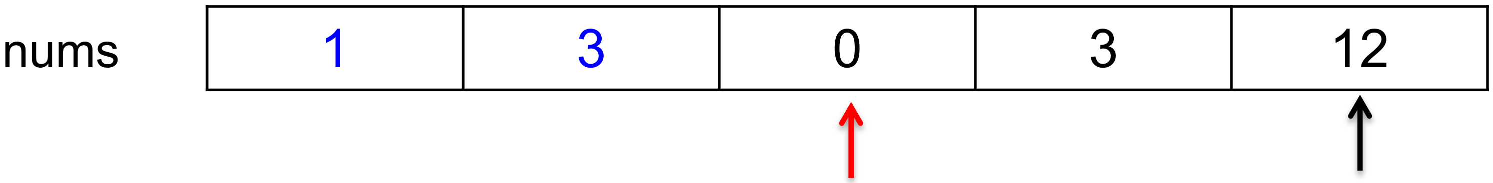
nums	1	1	0	3	12

Solution II



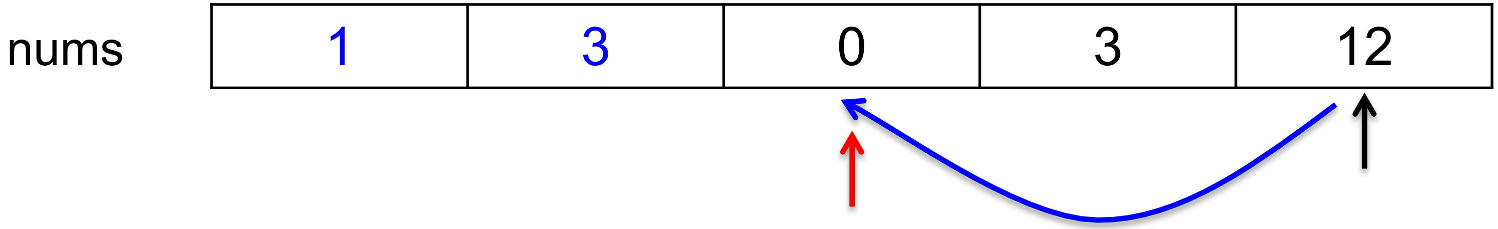
nums	1	3	0	3	12

Solution II



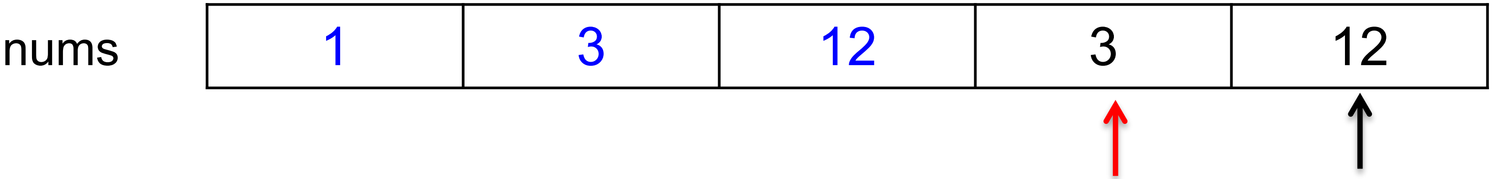
nums	1	3	0	3	12

Solution II



nums	1	3	0	3	12

Solution II



nums	1	3	12	3	12

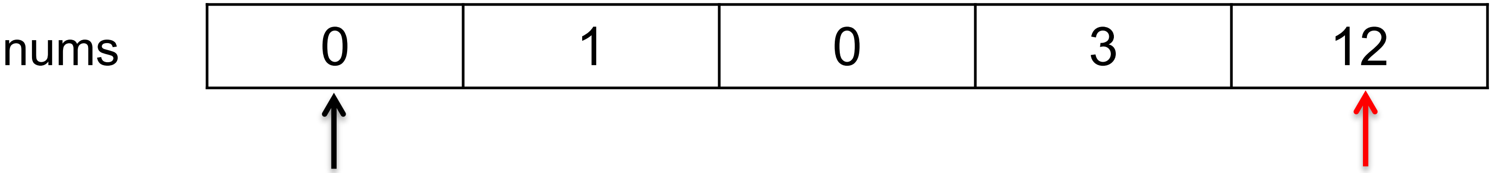
Solution II

```
int remove(int* nums, int numsSize, int val) {

    int nextReplace = 0;
    for (int i = 0; i <numsSize; i++) {
        if (nums[i] != val) {
            nums[nextReplace++] = nums[i];
        }
    }

    return nextReplace

}
```

Solution II

```
int remove(int* nums, int numsSize, int val) {

    int nextReplace = 0;
    for (int i = 0; i <numsSize; i++) {
        if (nums[i] != val) {
            nums[nextReplace++] = nums[i];
        }
    }

    return nextReplace;

}
```

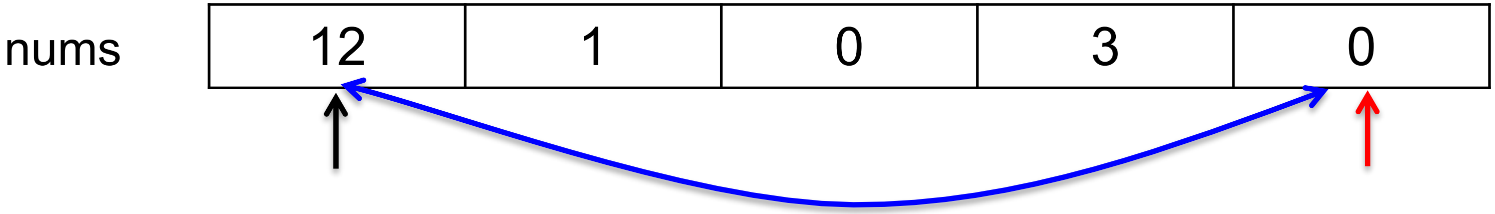
Solution III



nums	0	1	0	3	12

[0, 1, 0, 3, 12], val: 0

Solution III

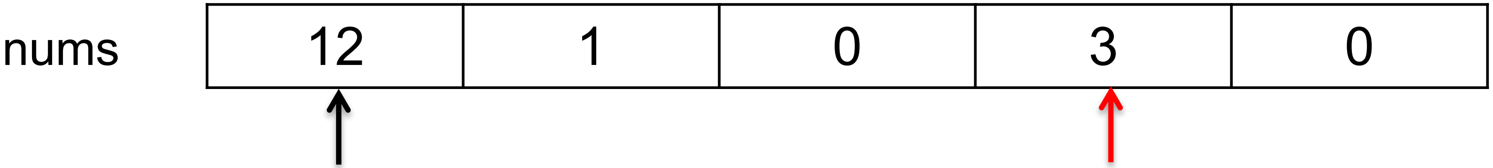


nums

12	1	0	3	0

[0, 1, 0, 3, 12], val: 0

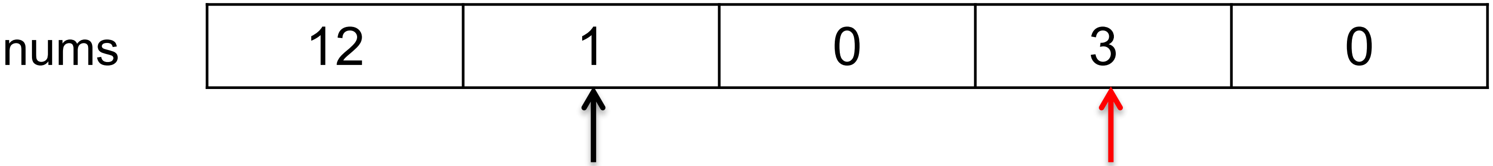
Solution III



nums	12	1	0	3	0

[0, 1, 0, 3, 12], val: 0

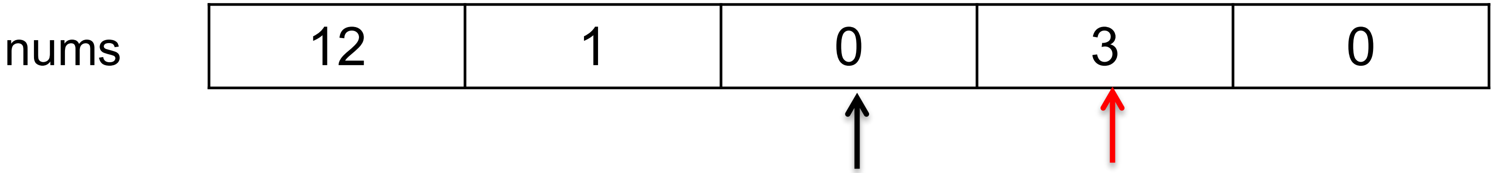
Solution III



nums	12	1	0	3	0

[0, 1, 0, 3, 12], val: 0

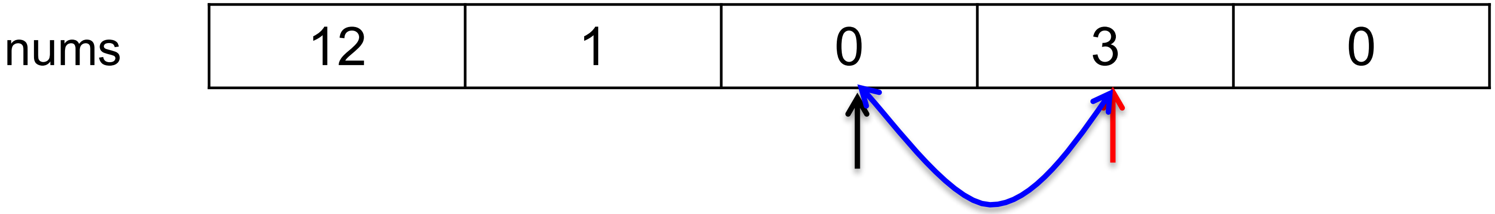
Solution III



nums	12	1	0	3	0

[0, 1, 0, 3, 12], val: 0

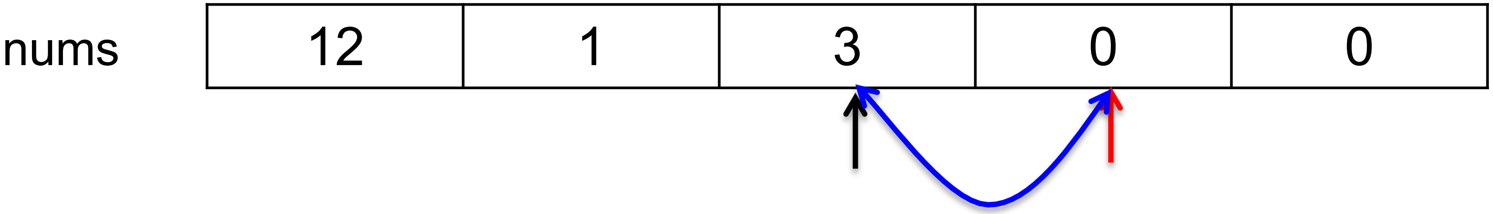
Solution III



nums	12	1	0	3	0

[0, 1, 0, 3, 12], val: 0

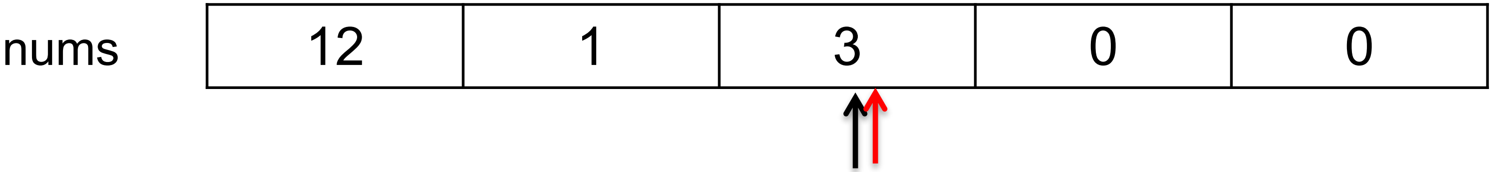
Solution III



nums	12	1	3	0	0

[0, 1, 0, 3, 12], val: 0

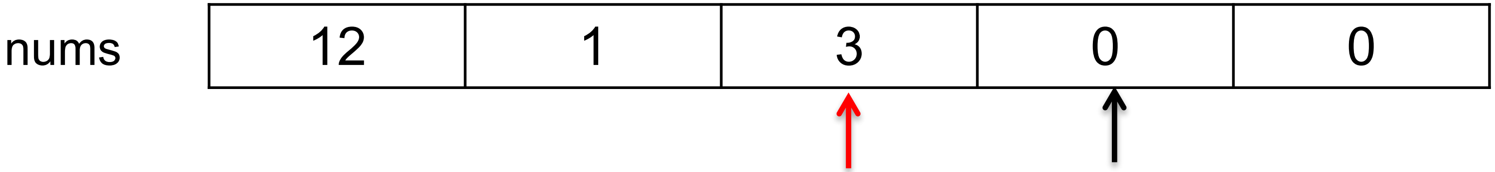
Solution III



nums	12	1	3	0	0

[0, 1, 0, 3, 12], val: 0

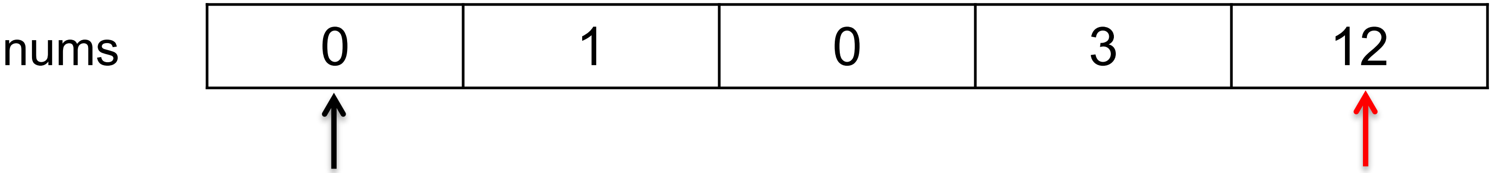
Solution III



nums	12	1	3	0	0

[0, 1, 0, 3, 12], val: 0

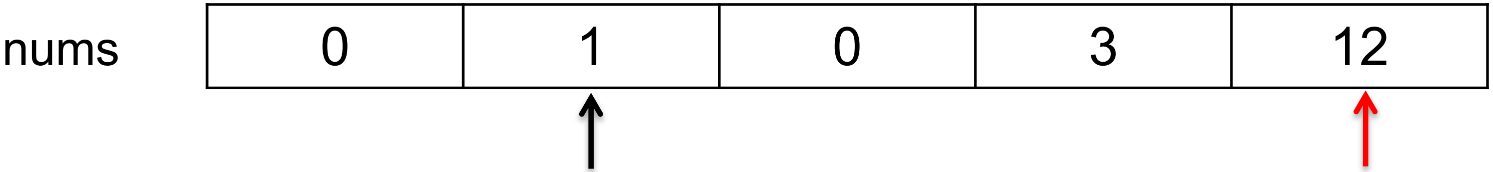
Solution III



nums	0	1	0	3	12

[0, 1, 0, 3, 12], val: 1

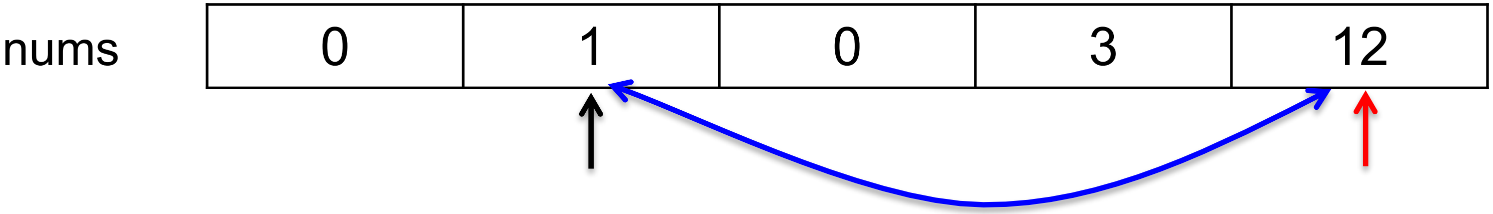
Solution III



nums	0	1	0	3	12

[0, 1, 0, 3, 12], val: 1

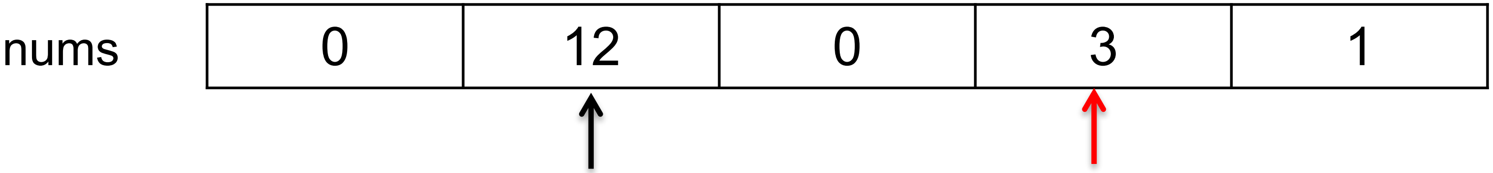
Solution III



nums	0	1	0	3	12

[0, 1, 0, 3, 12], val: 1

Solution III



nums	0	12	0	3	1

[0, 1, 0, 3, 12], val: 1

Solution III

```
int remove(int* nums, int numsSize, int val) {

      int i = 0;
     int n = numsSize - 1;
     while (i <= n) {
        if (nums[i] == val) {
            nums[i] = nums[n];
            n--;
       } else {
          i++;
       }
       }

       return n + 1;
}
```