C - Function, Pointer, Array

Zhaoguo Wang
A gigantic main loop

https://github.com/qemu/qemu/blob/master/vl.c
Function

Readability

Reusability
Function

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

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

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

void dummy()
{
}
```
Function

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

r the function’s local variable
Function

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

- **return type**: `int`
- **name**: `add`
- **argument declarations**: `int a, int b`
- **function body**
  ```c
  int r = a + b;
  return r;
  ```
- **r**: the function’s local variable
- **return expression**: `r`
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.

-- In Clean Code: A Handbook of Agile Software Craftsmanship,
Why small size?

It fits easily on your screen without scrolling

It's about the conceptual size that you can hold in your head

It's meaningful enough to require a function in its own right
Local Variables

Scope
– within which the variable can be used

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

r’s scope is in function add
Local Variables

Scope
– within which the variable can be used
– local variables of the same name in different functions are unrelated

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

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

Scope
– The variable that can be accessed by all functions

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

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

```c
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

```c
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);
}
```

Result  x: 1,  y: 2
Pointer

char a = 'c';

```
0xffff
...
... 0xc
0xb
0xa
0x9
0x8
0x7
0x6
0x5
0x4
0x3
0x2
0x1
0x0
a:  'c'
```
char a = ‘c’;
int b = 2;
char a = 'c';
int b = 2;
&a;
char a = ‘c’;
int b = 2;
char *x = &a;
char a = 'c';
int b = 2;
char *x = &a;

32 bits machine: 4 bytes
64 bits machine: 8 bytes
char a = 'c';
int b = 2;
char *x = &a;
int *y = &b;
char a = ‘c’;
int b = 2;
char *x = &a;
int *y = &b;

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

*x = 'd';
*y = 100;
char a = 'c';
int b = 2;
char *x = &a;
int *y = &b;

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

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

*x = ‘d’;
*y = 100;

xx: 0x5, yy: 0x9
char a = ‘c’;
int b = 2;
char *x = &a;
int *y = &b;

*x = ‘d’;
*y = 100;

What is the value of &xx and &yy?
char a = 'c';
int b = 2;
char *x = &a;
int *y = &b;

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

What is the value of &x and &y? Somewhere:-P
Pass pointers to function

```c
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```
Pass pointers to function

```c
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);
}

Result  x: 2,  y: 1
**Pointer arithmetic**

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

<table>
<thead>
<tr>
<th>++</th>
<th>p++</th>
<th>Point to the next object with int (4 bytes next to current location)</th>
</tr>
</thead>
</table>

- `a`: 0x104
- `c`: 0x105
- `0x104 ← p`
## Pointer arithmetic

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

<table>
<thead>
<tr>
<th>++</th>
<th>p++</th>
<th>Point to the next object with int (4 bytes next to current location)</th>
<th>0x108</th>
</tr>
</thead>
</table>

![Diagram showing memory layout and pointer arithmetic](image)

- `a`: 0x104
- `c`: 0x105
- `0x108`: Address after `++p` and `p++`
# Pointer arithmetic

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code></td>
<td>Point to the next object with int (4 bytes next to current location)</td>
<td>0x108</td>
</tr>
<tr>
<td><code>p++</code></td>
<td>Point to the next object with int (4 bytes next to current location)</td>
<td>0x108</td>
</tr>
<tr>
<td><code>p=p+i</code></td>
<td>Point to the ith object with int after current location</td>
<td>0x104 + i*4</td>
</tr>
<tr>
<td><code>--</code></td>
<td>Point to the previous object with int</td>
<td>0x100</td>
</tr>
<tr>
<td><code>p--</code></td>
<td>Point to the previous object with int</td>
<td>0x100</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Point to the ith object with int before current location</td>
<td>0x104 - i*4</td>
</tr>
</tbody>
</table>
**Pointer arithmetic**

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Point to the next object with short (? bytes next to current location)</td>
<td>++p++</td>
</tr>
<tr>
<td>+</td>
<td>Point to the ith object with short after current location</td>
<td>p=p+i</td>
</tr>
<tr>
<td>--</td>
<td>Point to the previous object with short</td>
<td>p--</td>
</tr>
<tr>
<td>-</td>
<td>Point to the ith object with short before current location</td>
<td>p=p-i</td>
</tr>
</tbody>
</table>
**Pointer arithmetic**

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Expression</th>
<th>Description</th>
<th>Address after Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>p++</td>
<td>Point to the next object with short (2 bytes next to current location)</td>
<td>0x106</td>
</tr>
<tr>
<td>+</td>
<td>p=p+i</td>
<td>Point to the i\textsuperscript{th} object with short after current location</td>
<td>0x104 + i\times2</td>
</tr>
<tr>
<td>--</td>
<td>p--</td>
<td>Point to the previous object with short</td>
<td>0x102</td>
</tr>
<tr>
<td>-</td>
<td>p=p-i</td>
<td>Point to the i\textsuperscript{th} object with short before current location</td>
<td>0x104 – i\times2</td>
</tr>
</tbody>
</table>
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]

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

a:
Array

len(a[0]): 4 bytes → a[1] is 4 bytes next to a[0]
length of a[0]: 4 bytes \(\rightarrow\) a[1] is 4 bytes next to a[0]

int *p = &a[0] \(\rightarrow\) p++ points to a[1]
Array

length of a[0]: 4 bytes \(\rightarrow\) a[1] is 4 bytes next to a[0]

int *p = &a[0] \(\rightarrow\) p++ points to a[1]
\(\rightarrow\) 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++ 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++ points to a[1]
   → p + i points to a[i]

int *p = a ← int *p = &a[0]

p++ ✅
a++ ❌ Compiling 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++) ↔ p[1]
*(p + i) ↔ p[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]
*(p++)  ↔  p[1]  ❌ You are correct!!! 😊
*(p + i)  ↔  p[i]
Array

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

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

int *p = a ↔ int *p = &a[0]
*p = a ↔ p[1]
*(p + 1) ↔ p[i]
*(p + i)
int* func(int *p) {
    int *q = p;
    p = p + 1;
    return q;
}
int* func(int* p) {
    int* q = p;
    p = p + 1;
    return q;
}

#include <stdio.h>

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

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

    return 0;
}
```c
#include <stdio.h>

int* func(int *p) {
    int *q = p;
    p = p + 1;
    return q;
}

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

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

val1: 100
val2: 200
```
int* func(int *p) {
    p = p + 1;
    return p;
}
```c
int func(int *p) {
    p = p + 1;
    return *p;
}

#include <stdio.h>

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

    printf("val1: %d\n", *(++p));
    printf("val2: %d\n", *(p));
}
```
```c
#include <stdio.h>

int func(int *p) {
    p = p + 1;
    return *p;
}

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

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

    return 0;
}
```

```
++p
```

```
val1: 200
val2: 200
```
A good example or a bad example?

#include <stdio.h>

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

    printf("val of %d %d %d\n", *(p++), *(++p), *(p));
}
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
```c
int a;
int b = a + 1;
```

Divided by zero
```c
int a = 1 / 0;
```

Signed integer overflow
```c
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

Divided by zero

Signed integer overflow
Classic undefined behaviors

- Use uninitialized variables
- Avoid memory write

- Divided by zero

- Signed 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.

Signed integer overflow
- X86 silently wrap around
- 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.
#include <stdio.h>

void foo(int a) {
    if(a+100 < a) {
        printf("tiger is evil\n");
        return;
    }

    printf("tiger is handsome\n");
}

int main() {
    foo(100);
    foo(0x7fffffff);
}
#include <stdio.h>

void foo(int a) {
    if(a+100 < a) {
        printf("tiger is evil\n");
        return;
    }

    printf("tiger is handsome\n");
}

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

gcc removes the check with O3
Undefined behaviors in real world

https://github.com/torvalds/linux/blob/master/lib/vsprintf.c
Line 2129

The linux kernel uses GCC’s –fno-strict-overflow to disable such optimizations


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`**

<table>
<thead>
<tr>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sizeof(int)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(long)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(float)</code></td>
<td></td>
</tr>
<tr>
<td><code>sizeof(double)</code></td>
<td></td>
</tr>
</tbody>
</table>

64 bits machine
# function `sizeof`

<table>
<thead>
<tr>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sizeof(int)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>sizeof(long)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>sizeof(float)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>sizeof(double)</code></td>
<td>8</td>
</tr>
</tbody>
</table>

64 bits machine
# function `sizeof`

<table>
<thead>
<tr>
<th>expr</th>
<th>sizeof()</th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a = 0;</td>
<td>sizeof(a)</td>
<td></td>
</tr>
<tr>
<td>long b = 0;</td>
<td>sizeof(b)</td>
<td></td>
</tr>
<tr>
<td>int a = 0; long b = 0;</td>
<td>sizeof(a + b)</td>
<td></td>
</tr>
<tr>
<td>char c[10]</td>
<td>sizeof(c)</td>
<td></td>
</tr>
<tr>
<td>int ai[10]</td>
<td>sizeof(ai)</td>
<td></td>
</tr>
<tr>
<td>int ai[10]</td>
<td>sizeof(ai[8])</td>
<td></td>
</tr>
</tbody>
</table>

64 bits machine
## function `sizeof`

<table>
<thead>
<tr>
<th>expr</th>
<th><code>sizeof()</code></th>
<th>result (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int a = 0;</code></td>
<td><code>sizeof(a)</code></td>
<td>4</td>
</tr>
<tr>
<td><code>long b = 0;</code></td>
<td><code>sizeof(b)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>int a = 0; long b = 0;</code></td>
<td><code>sizeof(a + b)</code></td>
<td>8</td>
</tr>
<tr>
<td><code>char c[10]</code></td>
<td><code>sizeof(c)</code></td>
<td>10</td>
</tr>
<tr>
<td><code>int ai[10]</code></td>
<td><code>sizeof(ai)</code></td>
<td>10 * 4 = 40</td>
</tr>
<tr>
<td><code>int ai[10]</code></td>
<td><code>sizeof(ai[8])</code></td>
<td>4</td>
</tr>
</tbody>
</table>

64 bits machine
How to calculate the number of elements of an array?

```c
int main() {
    int a[] = {1,2,3,4,5,6,7,8,9,10...};
    // your code here
    int num = ???;
}
```
How to calculate the number of elements of an array?

```c
int main() {

    int a[] = {1,2,3,4,5,6,7,8,9,10...};

    // your code here
    int num = sizeof(a) / sizeof(int);
}
```
Exercise

Write a function that add every element of the array by 1.

```c
int main() {

    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10...};

    // your function invocation
    add(...);
}
```
Exercise

Write a function that add every element of the array by 1.

```c
void add(int a[]) {
    int size = sizeof(a) / sizeof(int);
    for (int i = 0; i < size; i++) {
        a[i] = a[i] + 1;
    }
}
```
Exercise

Write a function that add every element of the array by 1.

```c
void add(int a[], int size) {
    for (int i = 0; i < size; i++) {
        a[i] = a[i] + 1;
    }
}
```
Exercise

Write a function that add every element of the array by 1.

```c
void add(int *a, int size) {
    for (int i = 0; i < size; i++) {
        a[i] = a[i] + 1;
    }
}
```
Exercise

Write a function that add every element of the array by 1.

```c
int main() {
    int a[] = {1,2,3,4,5,6,7,8,9,10...};
    add(a, sizeof(a)/sizeof(int));
}
```
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 3\textsuperscript{rd} element of the array \texttt{arr}?
Recap pointer and array

```c
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

<table>
<thead>
<tr>
<th>nums</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
</table>

[tmp]


Solution I

<table>
<thead>
<tr>
<th>nums</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solution I

<table>
<thead>
<tr>
<th>nums</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solution I

nums

| 0 | 1 | 0 | 3 | 12 |

tmp

| 1 | 3 |   |   |   |
# Solution I

<table>
<thead>
<tr>
<th>nums</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solution I

<table>
<thead>
<tr>
<th>nums</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solution I

<table>
<thead>
<tr>
<th>nums</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Solution I

<table>
<thead>
<tr>
<th>nums</th>
<th>1</th>
<th>3</th>
<th>12</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

| tmp   | 1 | 3 | 12 | 0 | 0 |
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

```c
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

<table>
<thead>
<tr>
<th>nums</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

↑
Solution II

nums

| 0 | 1 | 0 | 3 | 12 |
Solution II

```plaintext
nums

1 1 0 3 12
```
# Solution II

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
</table>

`nums`
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

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

nums

| 0 | 1 | 0 | 3 | 12 |

↑  ➠  ▲
### Solution II

<table>
<thead>
<tr>
<th>nums</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
</table>

Arrows indicate movement through the sequence.
Solution II

```python
nums = [1, 1, 0, 3, 12]
```
Solution II

nums

| 1 | 1 | 0 | 3 | 12 |

Diagram showing the array nums with values 1, 1, 0, 3, and 12.
Solution II

nums

\[
\begin{array}{c|c|c|c|c|c}
1 & 1 & 0 & 3 & 12 \\
\end{array}
\]
Solution II

ums

1 3 0 3 12
Solution II

nums

1 3 0 3 12
Solution II

nums

\[
\begin{array}{cccc}
1 & 3 & 12 & 3 & 12 \\
\end{array}
\]
Solution II

nums

1 3 12 0 0

↑

↑
Solution II

```c
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

<table>
<thead>
<tr>
<th>num</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
</table>

*nums*
Solution III

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
</table>

nums
Solution III

nums

1 0 0 3 12
Solution III

nums

| 1 | 0 | 0 | 3 | 12 |

↑  ↓
Solution III

nums

|   | 1 | 3 | 0 | 0 | 12 |

Arrow pointing from 1 to 3, and from 3 to 0.
<table>
<thead>
<tr>
<th>num</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Solution III

nums
Solution III

ums

1 3 12 0 0
Solution III

```c
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

<table>
<thead>
<tr>
<th>nums</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tmp</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Solution I

nums

| 0 | 1 | 0 | 3 | 12 |

arrow

tmp

1
Solution I

nums

| 0 | 1 | 0 | 3 | 12 |

tmp

| 1 |   |   |   |   |
Solution I

\[
\begin{array}{c|c|c|c|c|c}
\text{nums} & 0 & 1 & 0 & 3 & 12 \\
\text{tmp} & 1 & 3 & & & \\
\end{array}
\]
**Solution I**

<table>
<thead>
<tr>
<th>nums</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tmp</th>
<th>1</th>
<th>3</th>
<th>12</th>
<th></th>
</tr>
</thead>
</table>
Solution I

```c
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

```c
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

<table>
<thead>
<tr>
<th>nums</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>12</th>
</tr>
</thead>
</table>

Arrows indicate the movement of elements.
### Solution II

| nums | 0 | 1 | 0 | 3 | 12 |

**Arrow Indicators:**
- Red arrow pointing to 0
- Black arrow pointing to 1
Solution II

nums

| 1 | 1 | 0 | 3 | 12 |

1 1 0 3 12
### Solution II

<table>
<thead>
<tr>
<th>nums</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>
Solution II

nums

| 1 | 3 | 0 | 3 | 12 |
### Solution II

<table>
<thead>
<tr>
<th>nums</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>
### Solution II

<table>
<thead>
<tr>
<th>nums</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>
Solution II

nums

| 1 | 3 | 12 | 3 | 12 |

↑  

↑
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

```c
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;
}
```

How to reduce the move operations
[0, 1, 0, 3, 12], val: 1
Solution III

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

Solution III

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

nums

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

\[
\begin{array}{c|c|c|c|c}
12 & 1 & 3 & 0 & 0 \\
\end{array}
\]

[0, 1, 0, 3, 12],  val: 0
**Solution III**

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>1</th>
<th>3</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

```
[0, 1, 0, 3, 12],  val: 0
```
Solution III

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>1</th>
<th>3</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

[nums = [0, 1, 0, 3, 12], val: 0]
Solution III

[nums: 0, 1, 0, 3, 12], val: 1
Solution III

nums:

\[
\begin{array}{cccc}
0 & 1 & 0 & 3 & 12 \\
\end{array}
\]

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

nums

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

[nums]

[0, 1, 0, 3, 12],  val: 1
```c
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;
}
```
Pointer casting

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

What are the values of *(c++) ?
(Intel laptop)
Pointer casting

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

What are the values of *(c+1) ?
(Intel laptop)
**Pointer casting**

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

Intel laptop is small endian
Pointer casting

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

```
0x78
```

```
a: 0x78 0x104 0x105 0x106 0x107 0x108 0x109 0x10a 0x10b
```

```
*(c+1) is 0x56
```

```
0x12 0x34 0x56
```

```
p / c
```

```
c++
```
Pointer casting

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

*`(c+1)` is 0x56*

What about big endian?