C - Functions, Pointers, Arrays

Jinyang Li

based on the slides of Tiger Wang
Functions
C program consists of functions (aka subroutines, procedures)

Why breaking code into functions?

- Readability
- Reusability
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.
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

```c
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

```c
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

```c
int r = 0;

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

```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
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

```c
int main()
{
    int x = 1;
    int y = 2;
    swap(x, y);
    printf("x: %d, y: %d", x, y);
}
```

Result  x: 1, y: 2
Pointers

Pointer is a memory address
Pointer

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

Size of pointer on a 64-bit machine?
8 bytes
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

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

what if x is uninitialized?

Dereferencing an invalid address likely results in "Segmentation fault"
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”
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

*x = 3;
*y = 127;
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;

value of xx?
printf("xx=%p", xx);  xx=0x5
char a = 'c';
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=0x0b
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);
}

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);
}
```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);
}
```
# Pointer arithmetic

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

<table>
<thead>
<tr>
<th>p+1</th>
<th>Point to the next object with type int (4 bytes after current object of address p)</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10c</td>
<td>p+2</td>
<td></td>
</tr>
<tr>
<td>0x10b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>p+1</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>

`a:`

| 0x00  | 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>Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+i</td>
<td>Point to the <em>i</em>th object of type <code>int</code> after object with address <code>p</code></td>
<td>0x104 + i*4</td>
</tr>
<tr>
<td>p-i</td>
<td>Point to the <em>i</em>th object with <code>int</code> before object with address <code>p</code></td>
<td>0x104 – i*4</td>
</tr>
</tbody>
</table>
## Pointer arithmetic

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p+i</code></td>
<td>Point to the (i)th object with type short after object with address (p)</td>
<td>???</td>
</tr>
<tr>
<td><code>p-i</code></td>
<td>Point to the (i)th object with type short before object with address (p)</td>
<td>???</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></th>
<th>Description</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+i</td>
<td>Point to the ith object with type short after object with address p</td>
<td>0x104 + i*2</td>
</tr>
<tr>
<td>p-i</td>
<td>Point to the ith object with type short before object with address p</td>
<td>0x104 - i*2</td>
</tr>
</tbody>
</table>
Pointer arithmetic

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

<table>
<thead>
<tr>
<th>p+i</th>
<th>Point to the ith object with type char * after object with address p</th>
<th>0x104 + i*8</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-i</td>
<td>Point to the ith object with type char * before object with address p</td>
<td>0x104 – i*8</td>
</tr>
</tbody>
</table>
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];

\[
\begin{array}{ccccccccccc}
\text{int} & & & & & & & & & & \\
\end{array}
\]
Array


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

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

int *p = &a[0] \( \rightarrow \) 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 \(\rightarrow\) a[1] is 4 bytes next to a[0]

\[
\text{int } *p = \&a[0] \rightarrow p+1 \text{ points to a[1]}
\]
\[
\rightarrow p + i \text{ points to a[i]}
\]

\[
\text{int } *p = a \quad \leftrightarrow \quad \text{int } *p = \&a[0]
\]
Array

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

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

int *p = a \(\leftrightarrow\) int *p = &a[0]

p++ ✔

a++ ✗ compilation error

p = &a ✗
Array

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

```plaintext
int *p = a ↔ int *p = &a[0]
*(p+1) ↔ p[1]
*(p + i) ↔ p[i]
```
```c
#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");
}
```

Output? 400 200 300

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

Output:
```
100 400 300
```
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");
}

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;

What are the values of *c in hex? (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?
int a = 0x12345678;
int *p = &a;
char *c = (char *)p;

*(c+1) is 0x56
Pointer casting

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

*(c+1) is 0x56

What about big endian?
Example use of pointer casting

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

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

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> expression</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>
<tr>
<td><code>sizeof(int *)</code></td>
<td></td>
</tr>
</tbody>
</table>

64 bits machine
### function `sizeof`

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

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

- out of bound array access
  ```c
  int a[2] = {1, 2};
  int *p = a
  *(p+3) = 3;
  ```

- Divide by zero
  ```c
  int a = 1 / 0;
  ```

- 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
• 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.
#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);
}
#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);
}
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 \(3^{rd}\) element of the array \(arr\)?
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

nums

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

tmp

|   |   |   |   |   |
Solution I

nums

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

↑

tmp

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
</table>
### Solution I

<table>
<thead>
<tr>
<th>nums</th>
<th></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>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tmp</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</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></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></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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nums</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tmp</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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];
    }
}
```
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>

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 |

- Red arrow: Start index
- Black arrow: End index
## Solution II

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

The diagram shows the array `nums` with values 1, 1, 0, 3, and 12. The arrows indicate the movement of elements within the array.
Solution II

nums

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

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

Red arrow points to 1, black arrow points to 0.
Solution II

nums

1 3 0 3 12
Solution II

nums

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

nums

| 1 | 3 | 12 | 3 | 12 |

[Diagram showing an arrow from a number to another number]
Solution II

nums

| 1 | 3 | 12 | 0 | 0 |

↑  ↓
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></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**

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

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

nums

\[
\begin{array}{cccccc}
1 & 3 & 0 & 0 & 12 \\
\end{array}
\]
**Solution III**

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

The arrows indicate the movement of elements in the array.
**Solution III**

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

| tmp  |     |     |     |     |     |
Solution I

nums
0 1 0 3 12

tmp
1
# Solution I

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tmp</strong></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

nums

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

↑ ↑
**Solution II**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nums</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Solution II
**Solution II**

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

![Arrow Diagram]
Solution II

nums

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

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

The arrows indicate the movement of pointers or elements in the array.
Solution II

nums

1  3  0  3  12
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;
}
```
Solution III

nums

0 1 0 3 12

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

nums

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

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

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

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

[nums
0 12 0 3 1
]

[0, 1, 0, 3, 12], \text{ val: 1}
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;
}