[SWE2015-41] Introduction to Data Structures (자료구조개론)

Arrays

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

Announcements

- Some proofs about asymptotic notations are uploaded
  - Check the proofs carefully

- If you have an issue when using **Codedang**, please contact us through the following channels:
  - KakaoTalk: http://pf.kakao.com/_UKraK
  - Mail: skkuding@gmail.com (if you have no KakaoTalk account)

- The solutions of coding practices will be uploaded soon
  - Try solving the problems on your own first!

(Recap) Addresses & Pointers

- **Note.** Address is often represented by hexadecimal numbers
  - The hexadecimal numbers are prefixed by `0x`
  - `0x00, ..., 0x09, 0x0a, ..., 0x0f ←  0, ...,  9, 10, ..., 15`
  - `0x10, ..., 0x19, 0x1a, ..., 0x1f ← 16, ..., 25, 26, ..., 31`
  - `0x08 + 4 = 0x0c`

- What is the address of a variable?

main.c

```c
#include <stdio.h>

int main() {
    float pi = 3.141592;
    printf("%p\n", &pi);
    return 0;
}
```

	Address	Value
&pi	0x16aedf328	3.141592
&pi+1	0x16aedf32c	...
&pi+2	0x16aedf330	...

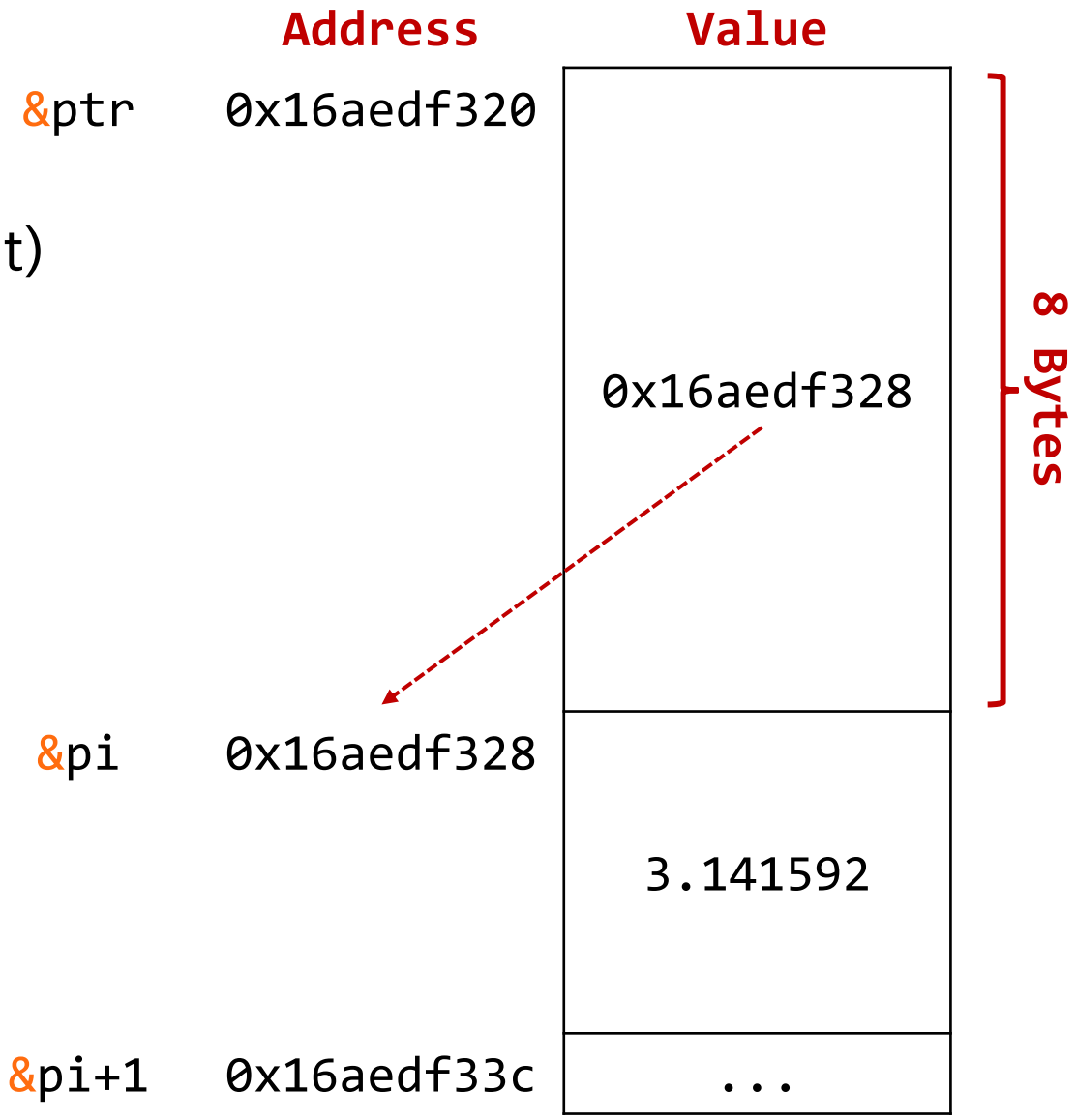
(Recap) Addresses & Pointers

- **Pointer** is the data type for address
  - Declare a pointer by * (asterisk) operator
  - Check the size of a pointer → 8 bytes (64-bit)

main.c

```c
#include <stdio.h>

int main() {
    float pi = 3.141592, *ptr = &pi;
    printf("%p\n", ptr);
    return 0;
}
```



(Recap) Addresses & Pointers

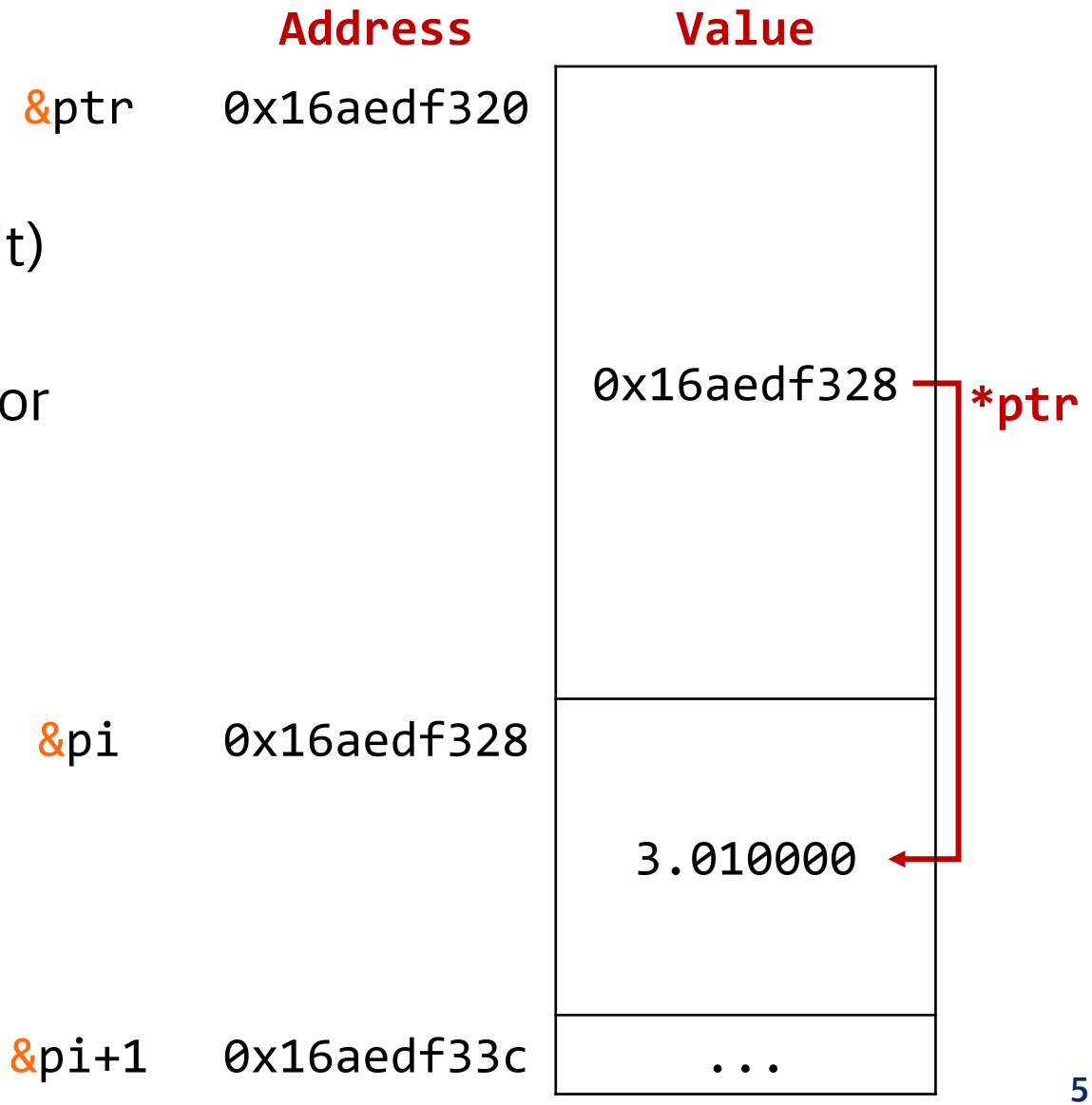
- **Pointer** is the data type for address
  - Declare a pointer by * (asterisk) operator
  - Check the size of a pointer → 8 bytes (64-bit)

  - Access the value at the address by * operator

main.c

```c
#include <stdio.h>

int main() {
    float pi = 3.141592, *ptr = &pi;
    *ptr = 3.01;
    printf("%f\n", pi);
    return 0;
}
```



(Recap) Asymptotic Notations

- Asymptotic notations (when n is large enough)
  - **Big-O notation** $O(\cdot)$ - Asymptotic Upper Bound
  - **Omega notation** $\Omega(\cdot)$ - Asymptotic Lower Bound
  - **Theta notation** $\Theta(\cdot)$ - Asymptotic Tight Bound

  - They allow us to calculate and compare time/space complexities easier

- Examples
  - $n^2 + 10n = O(n^3)$
  - $n^2 + 10n = \Omega(n)$
  - $n^2 + 10n = \Theta(n^2)$

(Recap) Asymptotic Notations

- Asymptotic notations (when n is large enough)
  - **Big-O notation** $O(\cdot)$ - Asymptotic Upper Bound
  - **Omega notation** $\Omega(\cdot)$ - Asymptotic Lower Bound
  - **Theta notation** $\Theta(\cdot)$ - Asymptotic Tight Bound

  - They allow us to calculate and compare time/space complexities easier

- **Big-O notation** (as tight as possible) is commonly used
  - This is because it is hard to compute the tight bound
  - Lower bound is often not informative for algorithm performance analysis
  - E.g., say $n^2 + 10n = O(n^2)$ rather than $n^2 + 10n = O(n^3)$

(Recap) Time & Space Complexities

- Time complexity ≈ # of primitive operations used in your program
  - Primitive variables: `int` (integers), `float`/`double` (floating-point numbers), `char` (characters), `bool` (Boolean values), …
  - Primitive operations: operations between primitive variables (e.g., addition, multiplication, declaration of a single variable, …)

- Space complexity ≈ # of bytes used in your program
  - Each primitive variable often occupies 1~8 bytes

- Express time & space complexities using asymptotic notations: O, Ω, Θ
  - Compare the complexities when the input size n is large enough

(Recap) Time & Space Complexities

- Time complexity - $O(n^2)$

```
void print_identity_matrix(int n) {
    int i, j;
    for (i = 0; i < n; i ++) {
        for (j = 0; j < n; j ++) {
            if (i == j) printf("%d ", 1);
            else printf("%d ", 0);
        }
        printf("\n");
    }
}
```

n = 4

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

What is an Array?

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**

Declaration in C

```
type name[size] = { ... };
```

main.c

```
int numbers[10] = {
    1, 5,  9, -3, 8,
    7, 6, 10, -5, 0
};
```

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	0x16aedf33c	10
8	0x16aedf340	-5
9	0x16aedf344	0

Access Elements in Array

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**

- The `i`-th element can be accessed by `arr[i]`

`numbers[2]`

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	0x16aedf33c	10
8	0x16aedf340	-5
9	0x16aedf344	0

Access Elements in Array

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**

- The `i`-th element can be accessed by `arr[i]`

`numbers[4]` →

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	**8**
5	0x16aedf334	7
6	0x16aedf338	6
7	0x16aedf33c	10
8	0x16aedf340	-5
9	0x16aedf344	0

Access Elements in Array

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**

- The `i`-th element can be accessed by `arr[i]`

`numbers[7]`

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	0x16aedf33c	**10**
8	0x16aedf340	-5
9	0x16aedf344	0

Access Elements in Array

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**
- The `i`-th element can be accessed by `arr[i]`
- Time complexity for the access = $O(1)$
  - Why?

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	**0x16aedf33c**	**10**
8	0x16aedf340	-5
9	0x16aedf344	0

Access Elements in Array

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**
- The `i`-th element can be accessed by `arr[i]`
- Time complexity for the access = $O(1)$
  - Address computation requires $O(1)$

```
numbers = &numbers[0] = 0x16aedf320
&numbers[7] = &numbers[0] + 7
            = 0x16aedf33c
```

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	**0x16aedf33c**	**10**
8	0x16aedf340	-5
9	0x16aedf344	0

Access Elements in Array

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**
- The `i`-th element can be accessed by `arr[i]`
- Time complexity for the access = $O(1)$
  - Address computation requires $O(1)$

```
numbers = &numbers[0] = 0x16aedf320
&numbers[i] = &numbers[0] + i
```

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	**0x16aedf33c**	**10**
8	0x16aedf340	-5
9	0x16aedf344	0

Access Elements in Array

- An array is a collection of elements of **the same data type** in **a contiguous block of memory**
- The `i`-th element can be accessed by `arr[i]`
- Time complexity for the access = $O(1)$
  - Address computation requires $O(1)$
  - Value modification also requires $O(1)$

`numbers[7] = 2`

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	**0x16aedf33c**	**2**
8	0x16aedf340	-5
9	0x16aedf344	0

Insert An Element into Array

- Consider an array of n=5 integer elements
  - The maximum size of the array = 10

main.c

```
int n = 5;
int arr[10] = {
    1, 5,  9, -3, 8,
};
```

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
n=5	0x16aedf334	null
6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

Insert An Element into Array

- Consider an array of `n=5` integer elements
  - The maximum size of the array = 10

main.c

```c
int n = 5;
int arr[10] = {
    1, 5,  9, -3, 8,
};
```

- How to insert an item at the end of `arr`?

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
n=5	0x16aedf334	null
6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

Insert An Element into Array

- Consider an array of `n=5` integer elements
  - The maximum size of the array = 10

main.c

```
int n = 5;
int arr[10] = {
    1, 5,  9, -3, 8,
};
```

- How to insert an item at the end of `arr`?

main.c

```
arr[n] = 7; // Add new item
n = n + 1; // Increase size
```

main.c

```
// Short version
arr[n++] = 7;
```

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
n=6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

Insert An Element into Array

- How to insert an item at a certain position of `arr`?

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
n=6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

Previous

Insert 6 at the 3rd position

Index	Address	Value
0	0x16aedf320	?
1	0x16aedf324	?
2	**0x16aedf328**	**6**
3	0x16aedf32c	?
4	0x16aedf330	?
5	0x16aedf334	?
6	0x16aedf338	?
n=7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

New

Insert An Element into Array

- How to insert an item at a certain position of `arr`?

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
n=6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

Previous

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	**0x16aedf328**	**6**
3	**0x16aedf32c**	**9**
4	**0x16aedf330**	**-3**
5	**0x16aedf334**	**8**
6	**0x16aedf338**	**7**
n=7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

New

Insert 6 at the 3rd position

Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index


    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = position; i < size; i ++)
        arr[i+1] = arr[i];
    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

- (Q) Does this code correct?

1	5	9	-3	8	7	null

Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

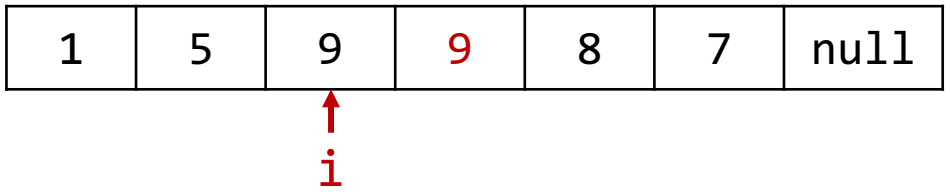
```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = position; i < size; i ++)
        arr[i+1] = arr[i];
    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

- (Q) Does this code correct?



Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

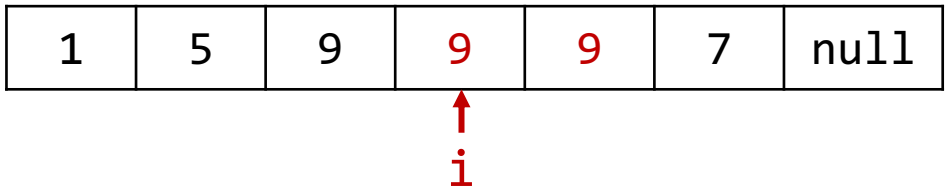
```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = position; i < size; i ++)
        arr[i+1] = arr[i];
    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

- (Q) Does this code correct?



Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

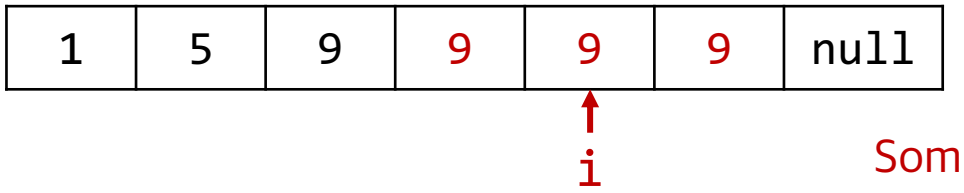
```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = position; i < size; i ++)
        arr[i+1] = arr[i];
    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

- (Q) Does this code correct?



Something wrong …

Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = size-1; i >= position; i --)
        arr[i+1] = arr[i];
    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

- (Q) How about this?

1	5	9	-3	8	7	null

Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

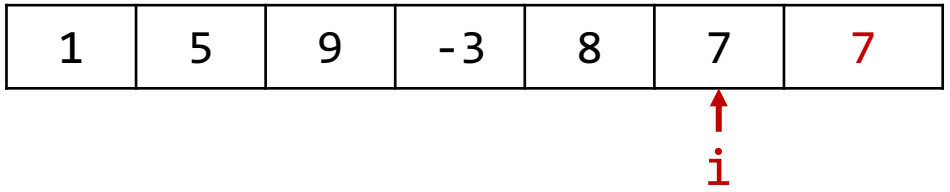
```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = size-1; i >= position; i --)
        arr[i+1] = arr[i];
    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

- (Q) How about this?



Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

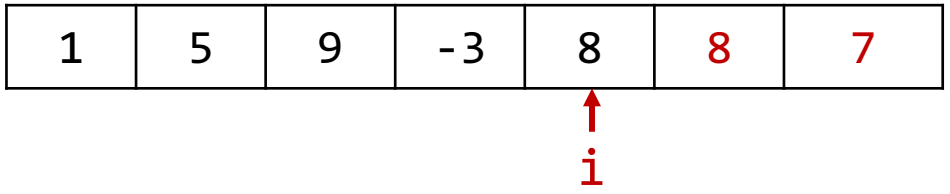
```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = size-1; i >= position; i --)
        arr[i+1] = arr[i];
    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

- (Q) How about this?



Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

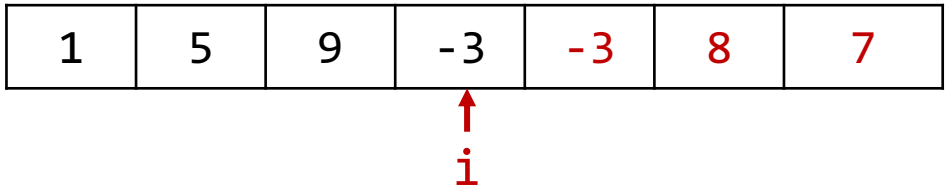
```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = size-1; i >= position; i --)
        arr[i+1] = arr[i];
    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

- (Q) How about this?



Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

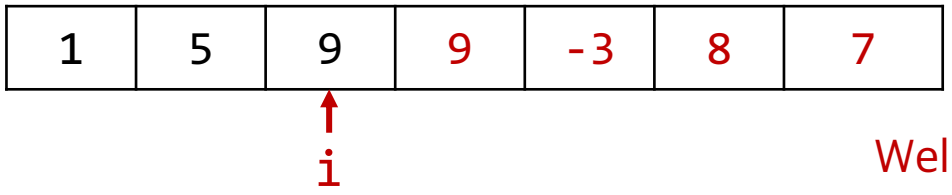
```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = size-1; i >= position; i --)
        arr[i+1] = arr[i];
    // 2. Put item into the position index

    // 3. Increase size

    return size;
}
```

- (Q) How about this?



Well-implemented!

Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

```c
#include <stdio.h>

int insert(int *arr, int size, int item, int position) {
    // 1. Push elements from the position index
    for (int i = size-1; i >= position; i --)
        arr[i+1] = arr[i];
    // 2. Put item into the position index
    arr[position] = item;
    // 3. Increase size
    size += 1;
    return size;
}
```

  - Time complexity for this insertion = $O(n)$ where n is the number of elements
    - Why?

Insert An Element into Array

- How to insert an item at a certain position of `arr`?

main.c

```c
#define MAX_SIZE 1000

int insert(int *arr, int size, int item, int position) {
    if (size >= MAX_SIZE) return -1; // Check whether the array memory is full
    // ...
}

int main() {
    int arr[MAX_SIZE] = { 1, 2, 3 }, n = 3;
    insert(arr, n, ...);
}
```

  - You need to check whether the array memory is already full to prevent memory leak/overflow
  - Other corner cases?

Delete An Element from Array

- How to delete an item from `arr`?

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
n=6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

Previous

Index	Address	Value
0	0x16aedf320	?
1	0x16aedf324	?
2	0x16aedf328	?
3	0x16aedf32c	?
4	0x16aedf330	?
n=5	0x16aedf334	null
6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

New

Delete 3rd item (value=9, index=2)

Delete An Element from Array

- How to delete an item from `arr`?

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
n=6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

Previous

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	-3
3	0x16aedf32c	8
4	0x16aedf330	7
n=5	0x16aedf334	null
6	0x16aedf338	null
7	0x16aedf33c	null
8	0x16aedf340	null
9	0x16aedf344	null

New

Delete 3rd item (value=9, index=2)

Delete An Element from Array

- How to delete an item from `arr`?

main.c

```c
#include <stdio.h>

int delete(int *arr, int size, int position) {
    if (size <= 0 || position < 0 || position >= size) return -1; // Corner cases

    // 1. Pull elements util the position index
    for (int i = position; i < size-1; i ++)
        arr[i] = arr[i+1];
    // 2. Decrease size
    size -= 1;
    return size;
}
```

1	5	9	-3	8	7

Delete An Element from Array

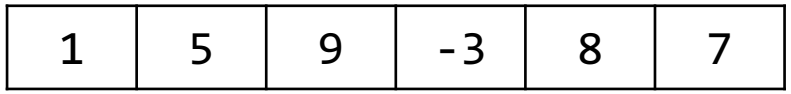
- How to delete an item from `arr`?

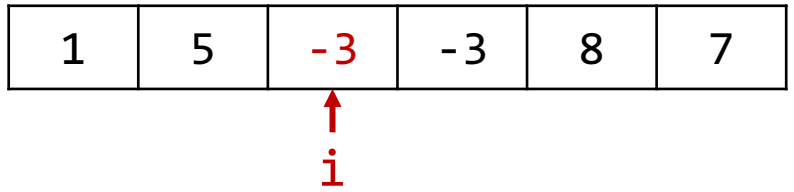
main.c

```c
#include <stdio.h>

int delete(int *arr, int size, int position) {
    if (size <= 0 || position < 0 || position >= size) return -1; // Corner cases

    // 1. Pull elements util the position index
    for (int i = position; i < size-1; i ++)
        arr[i] = arr[i+1];
    // 2. Decrease size
    size -= 1;
    return size;
}
```





Delete An Element from Array

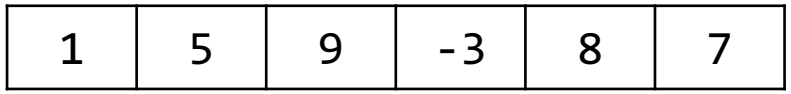
- How to delete an item from `arr`?

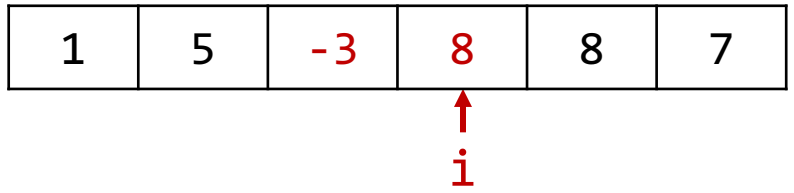
main.c

```c
#include <stdio.h>

int delete(int *arr, int size, int position) {
    if (size <= 0 || position < 0 || position >= size) return -1; // Corner cases

    // 1. Pull elements util the position index
    for (int i = position; i < size-1; i ++)
        arr[i] = arr[i+1];
    // 2. Decrease size
    size -= 1;
    return size;
}
```





Delete An Element from Array

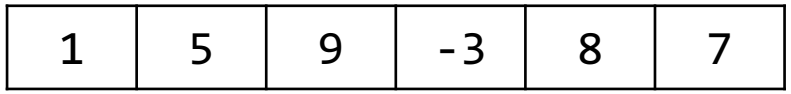
- How to delete an item from `arr`?

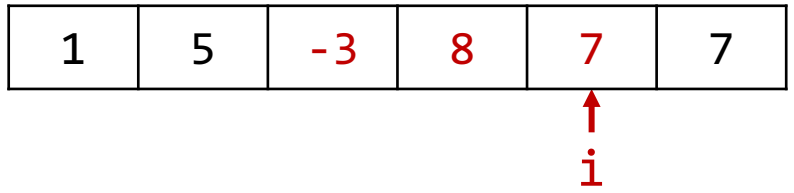
main.c

```c
#include <stdio.h>

int delete(int *arr, int size, int position) {
    if (size <= 0 || position < 0 || position >= size) return -1; // Corner cases

    // 1. Pull elements util the position index
    for (int i = position; i < size-1; i ++)
        arr[i] = arr[i+1];
    // 2. Decrease size
    size -= 1;
    return size;
}
```





Delete An Element from Array

- How to delete an item from `arr`?

main.c

```c
#include <stdio.h>

int delete(int *arr, int size, int position) {
    if (size <= 0 || position < 0 || position >= size) return -1; // Corner cases

    // 1. Pull elements util the position index
    for (int i = position; i < size-1; i ++)
        arr[i] = arr[i+1];
    // 2. Decrease size
    size -= 1;
    return size;
}
```

  - Time complexity for this deletion = $O(n)$ where n is the number of elements

Find An Element by Value in Array

- How to find an item of the specific value?
  - `arr[] = { 1, 5, ... }, n = 10;`
  - `target value = 6`

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	0x16aedf33c	10
8	0x16aedf340	-5
9	0x16aedf344	0

Find An Element by Value in Array

- How to find an item of the specific value?
  - `arr[] = { 1, 5, ... }, n = 10;`
  - `target value = 6`
  - Check all elements in the sequential order

```c
int find(int *arr, int size, int val) {
    for (int i = 0; i < size; i ++)
        if (arr[i] == val)
            return i;
    return -1; // Not found
}
```

  - What is time complexity for sequential search?

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	0x16aedf33c	10
8	0x16aedf340	-5
9	0x16aedf344	0

Find An Element by Value in Array

- How to find an item of the specific value in the sorted array?
  - `arr[] = { -5, -3, ... }, n = 10;`
  - `target value = 6`

  - Sequential search in $O(n)$

```c
int find(int *arr, int size, int val) {
    for (int i = 0; i < size; i ++)
        if (arr[i] == val)
            return i;
    return -1; // Not found
}
```

  - Is sequential search the fastest solution?
    - For sorted arrays, you can use binary search!
    - Its time complexity is $O(\log n)$

Index	Address	Value
0	0x16aedf320	-5
1	0x16aedf324	-3
2	0x16aedf328	0
3	0x16aedf32c	1
4	0x16aedf330	5
5	0x16aedf334	6
6	0x16aedf338	7
7	0x16aedf33c	8
8	0x16aedf340	9
9	0x16aedf344	10

Coding Practices - Array Statistics

- Compute statistics of `n` numbers
  - Sum, minimum, maximum, …

```
int getSum(int *arr, int n);
int getMin(int *arr, int n);
int getMax(int *arr, int n);
```

- **(Q)** Can you extend the functions compute statistics of `i`-th ~ `j`-th items?
  - Can you efficiently compute the statistics for any `(i,j)` pair?
  - Can you consider a scenario where item modification is allowed?
    - You will learn more efficient data structures for these challenging scenarios
    - E.g., segment tree

Coding Practices - `struct` Array

- **How to effectively implement** the array structure?

  (Q) What type of data should be stored?

  (Q) What operations are necessary?

Coding Practices - `struct` Array

- **How to effectively implement** the array structure?

  (Q) What type of data should be stored?

  - `items[]` - the physical memory allocated for storing elements
  - `size` - the number of the elements stored

  (Q) What operations are necessary?

Coding Practices - `struct` Array

- **How to effectively implement** the array structure?

  (Q) What type of data should be stored?

  - `items[]` - the physical memory allocated for storing elements
  - `size` - the number of the elements stored

  (Q) What operations are necessary?

  - `insert()` - insert an element to the array
  - `delete()` - delete an element from the array
  - `getSize()` - count the number of elements in the array
  - `isEmpty()` - check whether the array is empty or not
  - `isFull()` - check whether the array is full or not
  - …

Coding Practices - `struct` Array


```c
#include <stdbool.h>        // This enables to use bool type
#define MAX_SIZE 10000      // Maximum size of our array structure

typedef struct _IntArray { // Array structure for integer values
    int items[MAX_SIZE];
    int size;
} IntArray;

// IntArray operations:
void insert(IntArray *arr, int value, int index);
void delete(IntArray *arr, int index);
bool isFull(IntArray *arr);
bool isEmpty(IntArray *arr);
int getSize(IntArray *arr);
int getMax(IntArray *arr);
int getMin(IntArray *arr);
int getSum(IntArray *arr);
...
```

Two-Dimensional Arrays (2D Arrays)

- You can declare a 2D array as follows:

  Declaration in C

  ```
  type name[rows][cols] = { ... };
  ```

```
int arr[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

- The `i`th-row `j`th-column element can be accessed by `arr[i][j]`

Two-Dimensional Arrays (2D Arrays)

- You can declare a 2D array as follows:

  Declaration in C

  ```
  type name[rows][cols] = { ... };
  ```

```
int arr[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

- The `i`th-row `j`th-column element can be accessed by `arr[i][j]`

- Address manipulation for 2D Arrays
  - `arr` is the pointer of a 2D array → 1 unit = single row = 3 integers = 12 bytes
  - `arr+i` is the `i`th-row address

Two-Dimensional Arrays (2D Arrays)

- You can declare a 2D array as follows:

  Declaration in C

  ```
  type name[rows][cols] = { ... };
  ```

```
int arr[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

- The `i`th-row `j`th-column element can be accessed by `arr[i][j]`

- Address manipulation for 2D Arrays
  - `arr` is the pointer of a 2D array → 1 unit = single row = 3 integers = 12 bytes
  - `arr+i` is the `i`th-row address
  - `*(arr+i)` is the pointer of the `i`th-row 1D array → 1 unit = 1 integer = 4 bytes
  - `*(arr+i)+j` is the address of the `i`th-row `j`th-column element

Two-Dimensional Arrays (2D Arrays)

- You can declare a 2D array as follows:

  Declaration in C

  ```
  type name[rows][cols] = { ... };
  ```

```
int arr[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

- The `i`th-row `j`th-column element can be accessed by `arr[i][j]`

- Address manipulation for 2D Arrays
  - `arr` is the pointer of a 2D array → 1 unit = single row = 3 integers = 12 bytes
  - `arr+i` is the `i`th-row address
  - `*(arr+i)` is the pointer of the `i`th-row 1D array → 1 unit = 1 integer = 4 bytes
  - `*(arr+i)+j` is the address of the `i`th-row `j`th-column element
  - `*(*(arr+i)+j)` is the value of the `i`th-row `j`th-column element

Array with Elements of A Structure Type

- You can use `struct` for the array type

```
struct Rectangle {
    int height, width;
};
struct Rectangle rectangles[5] = {
    { 4, 3 }, { 5, 4 },
    { 1, 2 }, { 10, 1 },
    { 2, 8 },
};
```

- How to access?
  - `rectangles[i].height`
  - `(rectangles+i)->height`

Index	Address	Value	
0	0x16aedf320	height	4
	0x16aedf324	width	3
1	0x16aedf328	height	5
	0x16aedf32c	width	4
2	0x16aedf330	height	1
	0x16aedf334	width	2
3	0x16aedf338	height	10
	0x16aedf33c	width	1
4	0x16aedf340	height	2
	0x16aedf344	width	8

When Array Structure Is Useful?

- In general, array operations need the following time complexities

Operation	Time Complexity
Insertion	$O(n)$
Insertion at End	$O(1)$
Deletion	$O(n)$
Deletion at End	$O(1)$
Search by Index	$O(1)$
Search by Value	$O(n)$

- When array structures are inefficient?
  - If `insertion` or `deletion` at the middle frequently occurs, array is inefficient
  - If `search by value` is frequently required, array is inefficient

When Array Structure Is Useful?

- In general, array operations need the following time complexities

Operation	Time Complexity
Insertion	$O(n)$
Insertion at End	$O(1)$
Deletion	$O(n)$
Deletion at End	$O(1)$
Search by Index	$O(1)$
Search by Value	$O(n)$

- When array structures are useful?
  - If `insertion` and `deletion` only occurs at the end, array is efficient
  - If `search by value` is not required, array is efficient
  - Examples: stacks, queues, matrix (in linear algebra), …

Any Questions?