Data Structures

ARRAY

Array

A collection of elements stored in **contiguous memory locations**, where each element can be accessed directly using an index.

Arrays have a fixed size and provide constant-time access to elements.

All elements are typically of the same data type, making it efficient for storing and retrieving data by position.

One-dimensional Static Array (1D)

1. Declaration

```
int array[5];                          // 1D array with five elements
```

2. Initialization

```
int array[5] = {10, 20, 30, 40, 50};   // initialize the integer array with 10, 20, 30, 40, 50
```

3. Access

```
printf("%d", array[2]);                //prints 30 (the third element)
```

Representation: 1D Static Array

```
int array[5];                              // 1D array with five elements
```

	0	1	2	3	4
a					

```
int array[5] = {10, 20, 30, 40, 50};   // initialize the integer array with 10, 20, 30, 40, 50
```

	0	1	2	3	4
a	10	20	30	40	50

Two-dimensional Static Array (2D)

1. Declaration

```
int array2d[3][4];   // 2D array with 3 rows and 4 columns
```

2. Initialization

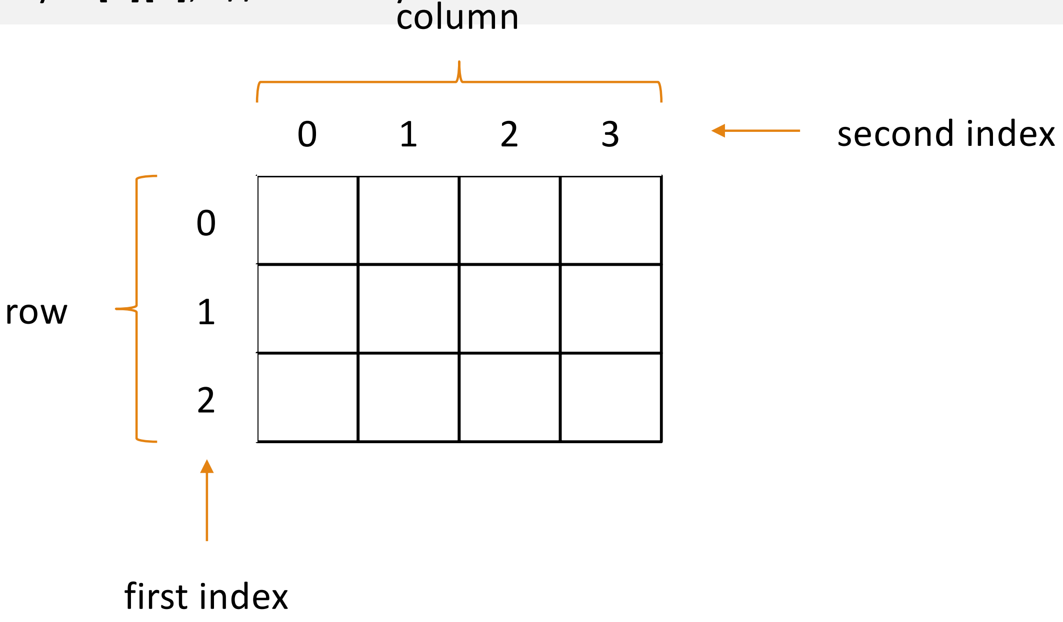
```
int array2d[3][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8},
   {9, 10, 11, 12}
};
```

3. Access

```
printf("%d", array2d[1][2]); // prints 7 (row 1, col 2)
```

Representation: 2D Static Array

```
int array2d[3][4];   // 2D array with 3 rows and 4 columns
```



Representation: 2D Static Array

```
int array2d[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Three-dimensional Static Array (3D)

1. Declaration

```
int array3d[2][3][4];   // 3D array: 2 blocks (planes), each 3×4 matrix
```

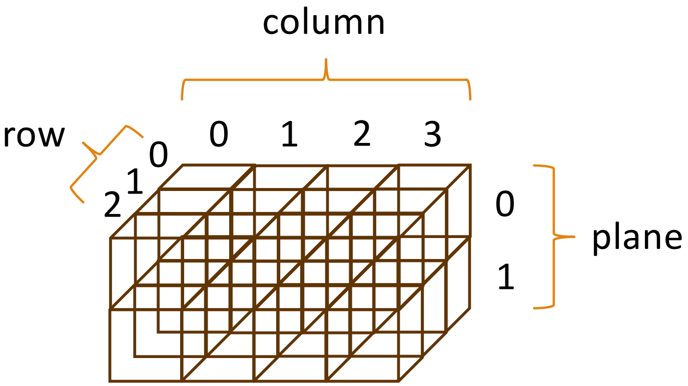
2. Initialization

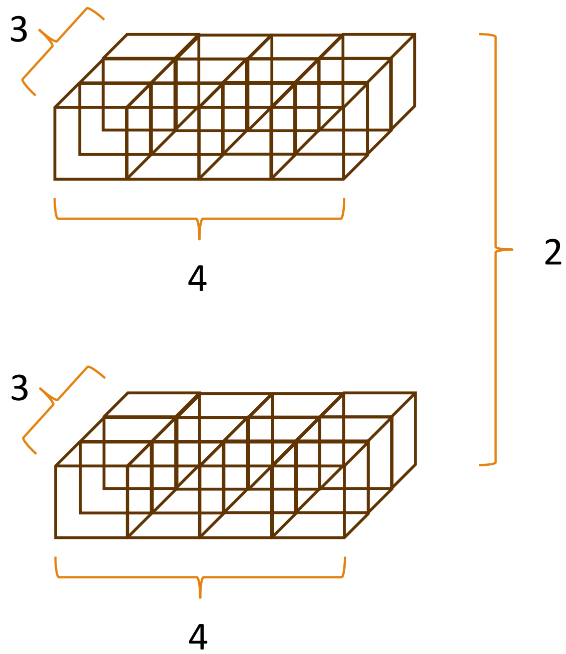
```
int array3d[2][3][4] = { { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} }, { {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24} } };
```

3. Access

```
printf("%d", array3d[1][2][3]); // prints 24
```

Representation: 3D Array





	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

	0	1	2	3
0	13	14	15	16
1	17	18	19	20
2	21	22	23	24

ADT: Array

ADT *Array* is

objects:

A set of pairs <*index, value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example. {0, …, *n*-1} for one dimension, {[(0, 0), (0, 1), (0, 2), (1, 0), (1 1), (1, 2), (2, 0), (2, 1), (2, 2)} for two dimensions, etc.

functions:

for all *A* ∈ *Array*, *i* ∈ *index*, *x* ∈ *item*, *j size* ∈ integer

Array Create(*j*, *list*) ::= **return** an array of *j* dimensions where list is a *j*-tuple whose *i*th element is the size of the *i*th dimension. *Items* are undefined.

Item Retrieve(*A*, *i*) ::= **if** (*i* ∈ *index*) **return** the item associated with index value *i* in array *A*
else return error

Array Store(A, *i*, *x*) ::= **if** (*i* in index) **return** an array that is identical to array *A* except the new pair <*i*, *x*> has been inserted
else return error

end *Array*

Size of Array

1. Static array
   - Fixed-length array
   - The number of elements is determined at **compile time** and cannot change.
2. Dynamic array
   - Variable-length array
   - The number of elements can be allocated or resized at **runtime** (e.g., using malloc in C).
   - When using malloc (or realloc) to increase the array size, remember to **free the allocated memory** after use to **avoid memory leaks**.

Dynamic Array

1. Declaration

```
int *array;
int n = 10;
array = (int *) malloc(n * sizeof(int));
```

2. Initialization

```
for(int i = 0; i < n; i++) {
        array[i] = i + 1;
}
```

Dynamic Array

3. Access

```
for (int i = 0; i < n; i++) {
   printf("%d ", array[i]);
}
```

4. Resize

```
n = n * 2;

array = (int *) realloc(array, n * sizeof(int));

for (int i = n/2; i < n; i++) {
          array[i] = i + 1;   // initialize new elements
}
```

Question: Dynamic Array

```
int *array;
int n = 10;

array = (int *) malloc(n * sizeof(int));

for(int i = 0; i < n; i++) {
 array[i] = i + 1;
}

for (int i = 0; i < n; i++) {
   printf("%d ", array[i]);
}

n = n * 2;

array = (int *) realloc(array, n * sizeof(int));

for (int i = n/2; i < n; i++) {
   array[i] = i + 1;   // initialize new elements
}
```

Code Review Challenge

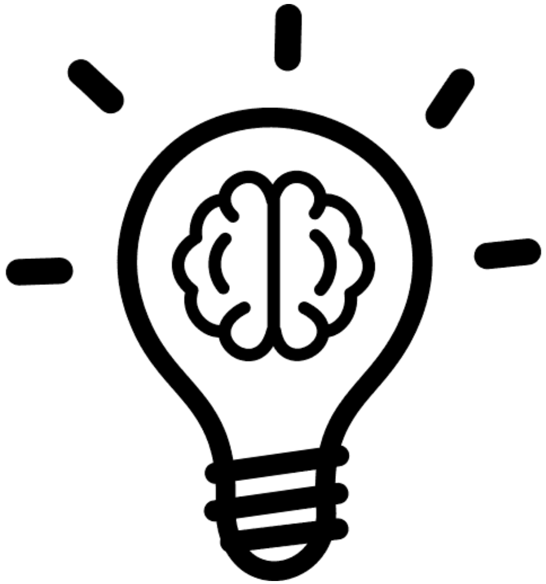


Image credit: https://uxwing.com/idea-icon/

Question: Dynamic Array

After reallocating an array to double its original size, will the starting memory address remain the same, or will it change?

Think carefully about how memory is managed in C when arrays grow.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *array;
    int n = 10;

    // Allocate memory for n integers
    array = (int *) malloc(n * sizeof(int));
    if (array == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Print the starting memory address
    printf("Initial memory address: %p\n",          );

    // Initialize elements
    for (int i = 0; i < n; i++) {
        array[i] = i + 1;
    }

    // Print elements
    printf("Initial array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    // Double the size
    n = n * 2;
    array = (int *) realloc(array, n * sizeof(int));
    if (array == NULL) {
        printf("Reallocation failed\n");
        return 1;
    }

    // Print the new memory address
    printf("After realloc memory address: %p\n",          );

    // Initialize new elements
    for (int i = n/2; i < n; i++) {
        array[i] = i + 1;
    }

    // Print all elements
    printf("Resized array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    // Free memory
    free(array);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
   int *array;
   int n = 10;

   // Allocate memory for n integers
   array = (int *) malloc(n * sizeof(int));
   if (array == NULL) {
      printf("Memory allocation failed\n");
      return 1;
   }

   // Print the starting memory address
   printf("Initial memory address: %p\n", (void*)array);

   // Initialize elements
   for (int i = 0; i < n; i++) {
      array[i] = i + 1;
   }

   // Print elements
   printf("Initial array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", array[i]);
   }
   printf("\n");

   // Double the size
   n = n * 2;
   array = (int *) realloc(array, n * sizeof(int));
   if (array == NULL) {
      printf("Reallocation failed\n");
      return 1;
   }

   // Print the new memory address
   printf("After realloc memory address: %p\n", (void*)array);

   // Initialize new elements
   for (int i = n/2; i < n; i++) {
      array[i] = i + 1;
   }

   // Print all elements
   printf("Resized array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", array[i]);
   }
   printf("\n");

   // Free memory
   free(array);
   return 0;
}
```

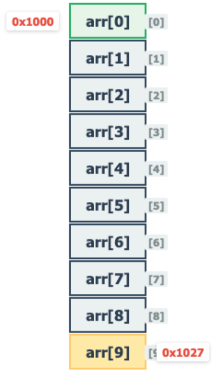
Your turn: print the ending memory address.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int *array;
   int n = 10;

   // Allocate memory for n integers
   array = (int *) malloc(n * sizeof(int));
   if (array == NULL) {
      printf("Memory allocation failed\n");
      return 1;
   }

   // Print the starting memory address
   printf("Initial memory address: %p\n", (void*)array);

   // Initialize elements
   for (int i = 0; i < n; i++) {
      array[i] = i + 1;
   }

   // Print elements
   printf("Initial array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", array[i]);
   }
   printf("\n");

   // Double the size
   n = n * 2;
   array = (int *) realloc(array, n * sizeof(int));
   if (array == NULL) {
      printf("Reallocation failed\n");
      return 1;
   }

   // Print the new memory address
   printf("After realloc memory address: %p\n", (void*)array);

   // Initialize new elements
   for (int i = n/2; i < n; i++) {
      array[i] = i + 1;
   }

   // Print all elements
   printf("Resized array: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", array[i]);
   }
   printf("\n");

   // Free memory
   free(array);
   return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *array;
    int n = 10;

    // Allocate memory for n integers
    array = (int *) malloc(n * sizeof(int));
    if (array == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Print the starting memory address
    printf("Initial memory address: %p\n", (void*)array);
    printf("Initial memory end address  : %p\n", (void*)(array + n * sizeof(int) - 1));

    // Initialize elements
    for (int i = 0; i < n; i++) {
        array[i] = i + 1;
    }

    // Print elements
    printf("Initial array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    // Double the size
    n = n * 2;
    array = (int *) realloc(array, n * sizeof(int));
    if (array == NULL) {
        printf("Reallocation failed\n");
        return 1;
    }

    // Print the new memory address
    printf("After realloc memory address: %p\n", (void*)array);
    printf("After realloc end address   : %p\n", (void*)(array + n * sizeof(n) - 1));

    // Initialize new elements
    for (int i = n/2; i < n; i++) {
        array[i] = i + 1;
    }

    // Print all elements
    printf("Resized array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    // Free memory
    free(array);
    return 0;
}
```

Array size 10

int arr[10];
sizeof(int) = 4 bytes

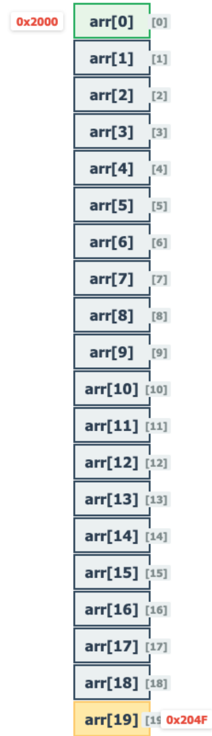



```
Address Calculation:
Start: 0x1000
End: 0x1000 + (10 × 4) - 1
End: 0x1000 + 40 - 1
End: 0x1027
```

Element Addresses:
arr[0]: 0x1000
arr[1]: 0x1004
arr[2]: 0x1008
...
arr[9]: 0x1024

Array size 20

int arr[20];
sizeof(int) = 4 bytes




```
Address Calculation:
Start: 0x2000
End: 0x2000 + (20 × 4) - 1
End: 0x2000 + 80 - 1
End: 0x204F
```

Element Addresses:
arr[0]: 0x2000
arr[1]: 0x2004
arr[2]: 0x2008
...
arr[19]: 0x204C

Key Observation of An Array

```
printf("Index %d -> Value: %d, Address: %p\n", i, array[i], (void*)&array[i]);
```

1. Array index
   - array[0]
   - array[3]
2. Value of the array
   - array[0] → 0
   - array[3] → 2
3. Array memory location
   - (void*)&array[0]
   - (void*)&array[3]

Question: Dynamic Array

After reallocating an array to double its original size, will the starting memory address remain the same, or will it change?

Think carefully about how memory is managed in C when arrays grow.

Now, practice by yourself.

STL: std::array

Element access

`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data`	direct access to the underlying contiguous storage (public member function)

Iterators

`begin` `cbegin`	returns an iterator to the beginning (public member function)
`end` `cend`	returns an iterator to the end (public member function)
`rbegin` `crbegin`	returns a reverse iterator to the beginning (public member function)
`rend` `crend`	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)

Operations

`fill`	fill the container with specified value (public member function)
`swap`	swaps the contents (public member function)

https://en.cppreference.com/w/cpp/container/array.html

STL: std::vector

Element access

`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data`	direct access to the underlying contiguous storage (public member function)

Iterators

`begin` `cbegin` (C++11)	returns an iterator to the beginning (public member function)
`end` `cend` (C++11)	returns an iterator to the end (public member function)
`rbegin` `crbegin` (C++11)	returns a reverse iterator to the beginning (public member function)
`rend` `crend` (C++11)	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)
`reserve`	reserves storage (public member function)
`capacity`	returns the number of elements that can be held in currently allocated storage (public member function)
`shrink_to_fit` (DR*)	reduces memory usage by freeing unused memory (public member function)

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements (public member function)
`insert_range` (C++23)	inserts a range of elements (public member function)
`emplace` (C++11)	constructs element in-place (public member function)
`erase`	erases elements (public member function)
`push_back`	adds an element to the end (public member function)
`emplace_back` (C++11)	constructs an element in-place at the end (public member function)
`append_range` (C++23)	adds a range of elements to the end (public member function)
`pop_back`	removes the last element (public member function)
`resize`	changes the number of elements stored (public member function)
`swap`	swaps the contents (public member function)

https://en.cppreference.com/w/cpp/container/vector.html

std::array vs. std::vector

Array

Element access

`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data`	direct access to the underlying contiguous storage (public member function)

Iterators

`begin` `cbegin`	returns an iterator to the beginning (public member function)
`end` `cend`	returns an iterator to the end (public member function)
`rbegin` `crbegin`	returns a reverse iterator to the beginning (public member function)
`rend` `crend`	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)

Operations

`fill`	fill the container with specified value (public member function)
`swap`	swaps the contents (public member function)

Vector

Element access

`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data`	direct access to the underlying contiguous storage (public member function)

Iterators

`begin` `cbegin` (C++11)	returns an iterator to the beginning (public member function)
`end` `cend` (C++11)	returns an iterator to the end (public member function)
`rbegin` `crbegin` (C++11)	returns a reverse iterator to the beginning (public member function)
`rend` `crend` (C++11)	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)
`reserve`	reserves storage (public member function)
`capacity`	returns the number of elements that can be held in currently allocated storage (public member function)
`shrink_to_fit` (DR*)	reduces memory usage by freeing unused memory (public member function)

std::array vs. std::vector

Aspect	`std::array` (Static)	`std::vector` (Dynamic)
Size	Fixed at compile-time	Variable at runtime
Memory	Stack (usually)	Heap
Performance	Fastest access	Fast, slight overhead
Memory Usage	Minimal	Extra capacity buffer
Flexibility	Limited	High
Use Case	Known, fixed data size	Varying data requirements

Why STL?

STL (Standard Template Library) is C++'s implementation of fundamental **Abstract Data Types (ADTs)**.

It provides ready-to-use, optimized data structures that abstract away implementation details while offering consistent interfaces.

Reference: https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/array

Application

Think: the Integer Array

64	34	25	12	22	11	90	8

Think: the Integer Array

0	1	2	3	4	5	6	7
64	34	25	12	22	11	90	8

Sort the Integer Array

Sorting

- Ascending order: 1, 3, 5, 7, 8, 20
- Descending order: 20, 8, 7, 5, 3, 1

Our goal: sorting the integer array by ascending order

original

64	34	25	12	22	11	90	8

sorted in ascending order (from smallest to largest)

8	11	12	22	25	34	64	90

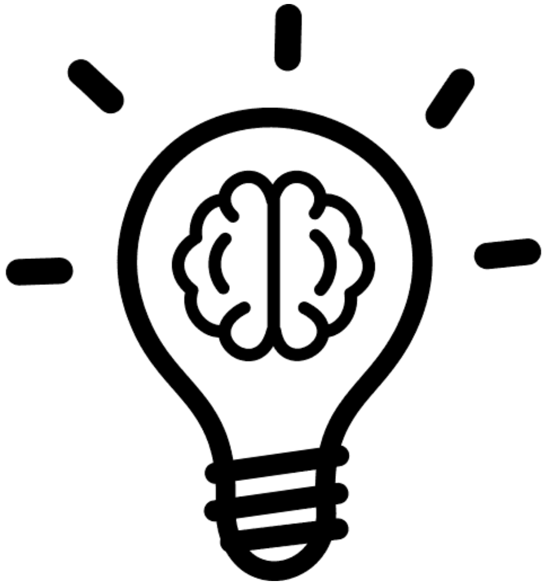


Image credit: https://uxwing.com/idea-icon/

Proposal

Solution 1:

Solution 2:

Solution 3:

…

Sort the Integer Array

1. Bubble sort
2. Selection sort
3. Insertion sort

Bubble Sort

```
procedure bubbleSort(A[1..n]):
    for i from 1 to n-1:
        for j from 1 to n-i:
            if A[j] > A[j+1]:
                swap A[j] and A[j+1]
```

Bubble Sort

```
procedure bubbleSort(A[1..n]):
    for i from 1 to n-1:
        for j from 1 to n-i:
            if A[j] > A[j+1]:
                swap A[j] and A[j+1]
```

A0	64	34	25	12	22	11	90	8
A1	34	25	12	22	11	64	8	90
A2	25	12	22	11	34	8	64	90
A3	12	22	11	25	8	34	64	90
A4	12	11	22	8	25	34	64	90
A5	11	12	8	22	25	34	64	90
A6	11	8	12	22	25	34	64	90
A7	8	11	12	22	25	34	64	90

Selection Sort

```
procedure selectionSort(A[1..n]):
    for i from 1 to n-1:
        minIndex = i
        for j from i+1 to n:
            if A[j] < A[minIndex]:
                minIndex = j
        swap A[i] and A[minIndex]
```

Selection Sort

```
procedure selectionSort(A[1..n]):
    for i from 1 to n-1:
        minIndex = i
        for j from i+1 to n:
            if A[j] < A[minIndex]:
                minIndex = j
        swap A[i] and A[minIndex]
```

A0	64	34	25	12	22	11	90	8
A1	8	34	25	12	22	11	90	64
A2	8	11	25	12	22	34	90	64
A3	8	11	12	25	22	34	90	64
A4	8	11	12	22	25	34	90	64
A5	8	11	12	22	25	34	90	64
A6	8	11	12	22	25	34	90	64
A7	8	11	12	22	25	34	64	90

Insertion Sort

```
procedure insertionSort(A[1..n]):
    for i from 2 to n:
        key = A[i]
        j = i - 1
        while j > 0 and A[j] > key:
            A[j+1] = A[j]
            j = j - 1
        A[j+1] = key
```

Insertion Sort

```
procedure insertionSort(A[1..n]):
    for i from 2 to n:
        key = A[i]
        j = i - 1
        while j > 0 and A[j] > key:
            A[j+1] = A[j]
            j = j - 1
        A[j+1] = key
```

A0	64	34	25	12	22	11	90	8
A1	34	64	25	12	22	11	90	8
A2	25	34	64	12	22	11	90	8
A3	12	25	34	64	22	11	90	8
A4	12	22	25	34	64	11	90	8
A5	11	12	22	25	34	64	90	8
A6	11	12	22	25	34	64	90	8
A7	8	11	12	22	25	34	64	90

Further Thinking (Pros & Cons Strategy)

Pros

- Static data with random access

0	1	2	3	4	5	6	7
8	11	12	22	25	34	64	90

Further Thinking (Pros & Cons Strategy)

Cons

- Insert into a Sorted Array
- Delete from a Sorted Array

Reasons

- Frequent insertions and deletions: costly shifts.

Further Thinking (Pros & Cons Strategy)

0	1	2	3	4	5	6	7
8	11	12	22	25	34	64	90

delete 22

0	1	2	3	4	5	6	7
8	11	12	25	34	64	90	90

Further Thinking (Pros & Cons Strategy)

0	1	2	3	4	5	6	7
8	11	12	22	25	34	64	90

insert 28

0	1	2	3	4	5	6	7	8
8	11	12	22	25	34	34	64	90

insert 28

0	1	2	3	4	5	6	7	8
8	11	12	22	25	28	34	64	90

Building Blocks of Data: Arrays Across Types

1. **Integer** array
   - An integer array is a collection of elements of type int stored in contiguous memory locations.
2. **Character** array
   - A character array is a collection of elements of type char stored in contiguous memory.
3. **String** array
   - In C, a string is essentially a character array terminated by '\0' (null character).
   - A string array can mean either:
     - A single character array that holds a string (e.g., "Hello").
     - An array of strings (e.g., list of words).

Declaration in C

1. Integer array

```
int array[5];                          // 1D array of 5 integers
array[5] = {10, 20, 30, 40, 50};
```

2. Character array

```
char letters[5];                       // an array of 5 characters
letters[5] = ['a', 'b', 'c', 'd', 'e’];
```

3. String array

```
char string[6] = "Hello";              // string (with '\0' at the end)
char *words[3] = {"cat", "dog", "fish"};   // array of strings
```

Integer Array

```
int array[5];                                   // 1D array of 5 integers
array[5] = {10, 20, 30, 40, 50};
```

{Integer} array can be any numeric data types including

1. Integer type (signed or unsigned)
   1. Basic: short, int, long, long long
   2. Fixed-width: int8_t, int16_t, int32_t, int64_t (signed); uint8_t, uint16_t, uint32_t, uint64_t (unsigned);
2. Floating-point type: float, double, long double

String Array

```
char string[6] = "Hello";                    // string (with '\0' at the end)
char *words[3] = {"cat", "dog", "fish"};     // array of strings
```

string[6] = “Hello”;

H	e	l	l	o	\0

*words[3] = {"cat", "dog", "fish"};

c	a	t	\0		
d	o	g	\0		
f	i	s	h	\0	

Search in Array

Search 22 in an array or not and report its index

1) unsorted

64	34	25	12	22	11	90	8

2) sorted

8	11	12	22	25	34	64	90

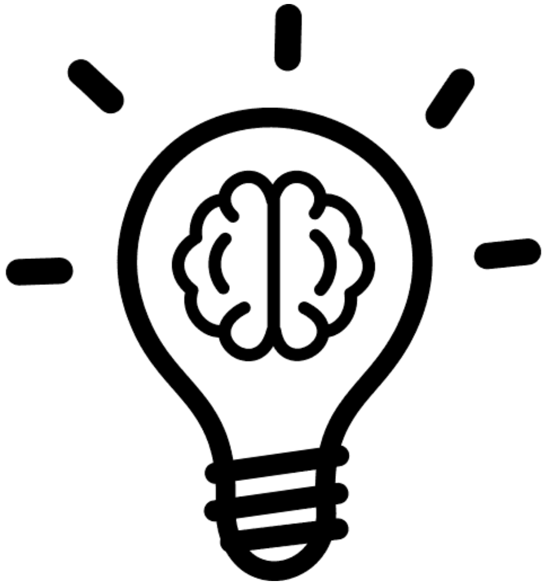


Image credit: https://uxwing.com/idea-icon/

Search in Array

1. Unsorted array
   - Target can be in any position.
   - Only go through the entire elements to ensure the existence of the target number
2. Sorted array
   - Target can be in the certain position.
   - Number of steps is guaranteed to ensure the existence of the target number

Search in Array

Unsorted array
- Linear search

Sorted array
- Linear search
- Binary search (improved and why)

ADT: Array

Create(*n*): Create an array of size *n*.

Access(*A*, *i*): Return the element at index *i*.

Update(*A*, *i*, *x*): Replace the element at index *i* with value *x*.

Insert(*A*, *i*, *x*): Insert value *x* at index *i* (may require shifting elements).

Delete(*A*, *i*): Delete element at index *i* (may require shifting elements).

Traverse(*A*): Visit each element of the array in order.

Search(*A*, *x*): Find index of value *x* (linear or binary depending on sorting).

Resize(*A*, *m*): Increase or decrease the size of the array (dynamic array using malloc/realloc in C)

Multi-dimensional arrays: Operations (insert/delete/resize) need extra care because rows/columns can be represented differently in **row-major order** or **pointers-to-pointers** style in C.

Complexity Analysis

Operation	**Complexity**	**Notes**
Access	O(1)	Direct index lookup
Update	O(1)	Replace at index
Insert	O(n)	Requires shifting elements
Delete	O(n)	Requires shifting elements
Traverse	O(n)	Visit all elements
Search	O(n) / O(log n)	Linear for unsorted, binary for sorted

Summary

1. Create: static or dynamic
2. Retrieve:
   - Random access
   - Search the target
     - Unsorted
     - Sorted
3. Update
   - Target
   - Insert
   - Delete

Bonus: AI Prompt for Studying "Array" (1)

I have basic knowledge of arrays and can create simple programs with them. Now I want to advance my understanding to intermediate level. Please help me with:

1. **Array Types and Memory**:
   - Static vs Dynamic arrays - when to use each?
   - How arrays are stored in memory (stack vs heap)
   - Memory layout and why it matters for performance

2. **Advanced Operations**:
   - Searching algorithms (linear search, binary search)
   - Sorting algorithms (bubble sort, selection sort, insertion sort)
   - Array manipulation (insertion, deletion, resizing)

Bonus: AI Prompt for Studying “Array” (2)

3. **Dynamic Memory Management**:
   - Using malloc/calloc/realloc/free in C
   - Understanding memory leaks and how to prevent them
   - Error handling for memory allocation failures

4. **Performance Analysis**:
   - Time complexity of different array operations
   - Space complexity considerations
   - When arrays are efficient vs inefficient

Bonus: AI Prompt for Studying "Array" (3)

5. **Practical Applications**:
   - Implementing data structures using arrays (stacks, queues)
   - Multi-dimensional arrays and their uses
   - String manipulation using character arrays

6. **Best Practices**:
   - Code organization and modularity
   - Error handling and defensive programming
   - Memory management best practices

Real-world Example

Purpose: IP-based access control module for Nginx that implements allow and deny directives to restrict client access based on IP addresses.

Core Functionality

- Parses configuration: allow 192.168.1.0/24, deny 10.0.0.1, allow all
- Stores rules in arrays: Uses ngx_array_t to maintain lists of access rules
- Evaluates requests: Checks client IP against rules during request processing
- Supports multiple protocols: IPv4, IPv6, and Unix domain sockets

https://github.com/nginx/nginx/blob/bc71625dcca1f1cbd0db7450af853feb90ebba85/src/http/modules/ngx_http_access_module.c

https://github.com/nginx/nginx/blob/master/src/core/ngx_array.c

https://github.com/nginx/nginx/blob/master/src/core/ngx_array.h