Accessing arrays in a high level programming language explained

- One of the most important data structures in high level programming languages is the **array**
- A computer program needs information to do its work (solve some problem)

  An array is one of the many **data structures** that Computer Science has developed to manage/organize information

  The array is a **static** structure, i.e., the number of elements in an array is **fixed** at creation and cannot be changed (unless you destroy the array and create a new one)

  (In contrast, the **linked list** data structure is **dynamic**: the number of elements can change)

- **The Array data structure**
  - **All elements** in an array is of the **same type**

    Hence, each array element will use/occupy the **same amount** of memory

  - **Array elements** are **stored** ***cosecutively*** in the **memory**:

```
.....
+------------------------+ <----------------  Compiler remembers the
|  Compiler remembers    |                    starting location
|  the size of           | First array variable
|  each array elem       |
+------------------------+
|                        |
|                        | Second array variable
|                        |
+------------------------+
|                        |
|                        | Third array variable
|                        |
+------------------------+
....
....
```

  - When the **(Java/C) compiler** processes an **array definition**, it **records**:

    - The **starting address** of the ***first*** **array element**
    - The **size** (from the **data type**) of each **aarray element**

- **Base of an array**
  - **Base:**

    - **base** of an **array** = the ***starting*** **address** of the ***first*** **array element**

- **Accessing acrray elements**
  - To **access** an **array variable** (e.g., A[5] or A[i]), we have to **compute** the **address** of each **array element**

- The **address** of an **array element** can be **computed** as **follows**:

```
 .....    .....
+------------------------+ <----------------  Base Address
|                        |
|     Array[0]           |  Address of Array[0] = Base address
|                        |
+------------------------+ <--------- Base address + 1*size
|                        |
|     Array[1]           |  Address of Array[1] = Base address + 1*size
|                        |
+------------------------+ <--------- Base address + 1*size
|                        |
|     Array[2]           |  Address of Array[2] = Base address + 2*size
|                        |
+------------------------+
....
....
+------------------------+ <--------- Base address + k*size
|                        |
|     Array[k]           |  Address of Array[k] = Base address + k*size
|                        |
+------------------------+
....
....
```

  - **Base address** = The **starting location** (address) in the memory where the array is stored
  - **Size** = The **number of bytes (size)** of an array element

- **Example 1:**

  - **Suppose** we define the following array:

```
int  B[10];   // Each int variable occupies 4 bytes !
```

  - **Suppose** the **array** is **stored** starting at **memory address** **7000**

    Then:

```
Base Address of array B = 7000
Size of array B element = 4        (because data type is int)
```

  - We can find the **address** of array variables `B[1]`, `B[5]` and `B[k]` (for **any value** `k`) as follows:

```
Address of B[1]  = 7000 + 1*4
Address of B[5]  = 7000 + 5*4
Address of B[k]  = 7000 + k*4
```

- **Example 2:**

  - **Suppose** we define the following array:

```
short  B[10];   // Each shortvariable occupies 2 bytes of memory !
```

- **Suppose** the **array** is **stored** starting at **memory address 7000**

  Then:

```
Base Address of array B = 7000
Size of array B element = 2        (because data type is short)
```

- We can find the **address** of array variables `B[1]`, `B[5]` and `B[k]` (for **any value** `k`) as follows:

```
Address of B[1]  = 7000 + 1*2
Address of B[5]  = 7000 + 5*2
Address of B[k]  = 7000 + k*2
```

- **Examples accesing arrays**

  - **Example 1:**

```
Variable definition:

   int ans;
   int MyArray[10];       // Array with 10 elements
```

```
High level language statement:

   ans = MyArray[0];

The compiler will translate this statement into
the following assembelr instructions:

   movea.l   #MyArray, A0
   move.l    0(A0), D0
   move.l    D0, ans
```

```
High level language statement:

   ans = MyArray[1];

The compiler will translate this statement into
the following assembelr instructions:

   movea.l   #MyArray, A0
   move.l    4(A0), D0
   move.l    D0, ans
```

```
High level language statement:

   ans = MyArray[5];

The compiler will translate this statement into
```

```
 the following assembelr instructions:

    movea.l   #MyArray, A0
    move.l    20(A0), D0
    move.l    D0, ans
```

- Accessing array elements with a constant index (A[5]) - **DEMO:** click here

---

- **A more advanced example:**

```
Variable definition:

    int ans;
    int i;                  // Assume i has been initialized to some value
    int MyArray[10];        // Array with 10 elements
```

```
 High level language statement:

     ans = MyArray[i];

 The compiler will translate this statement into
 the following assembelr instructions:

    movea.l   #MyArray, A0     * A0 = base address of array

    move.l    i, D0            * D0 = index
    muls      #4, D0           * This computes: D0 = 4*D0
                               * D0 now contains the offset !!!
    adda.l     D0, A0          * Add offset to the base address
    move.l    0(A0), D0        * Gets A[i] into D0
    move.l    D0, ans          * Store it in ans
```

- Accessing array elements with a variable as index (A[i]) - **DEMO:** click here

Note:

- **Later** we will learn about a **more powerful addressing mode** that will help us **access** `MyArray[i]` more ***easier*** !!!
