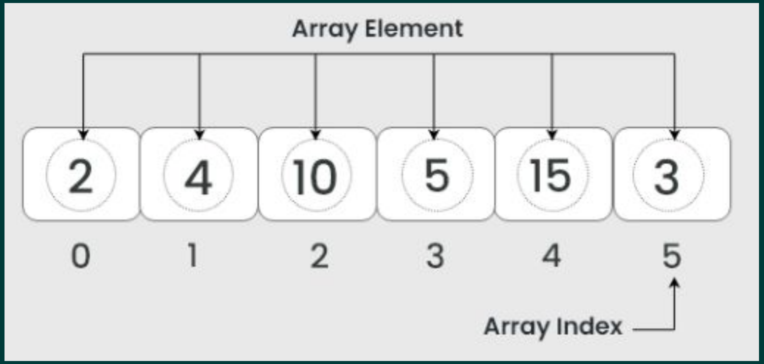
Arrays

- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
- An array (data structure) is allocated in memory at a beginning address
  - All elements of the array must be the same data type
  - Elements of the array are stored in sequential memory locations
  - Reference to specific elements use index values
- Structure of array



Example Integer Arrays

- Java array declarations:

```
int array_5[ ] = { 5, -1,  7,  8, -2 };
int array[ ] = new int[5];
```

- C array declarations

```
int myNumbers[ ] = {25, 50, 75, 100};
int my4Num[4];
```

- Python does not include arrays as a native data structure; defines lists instead

```
# List of integers
list1 = [1, 5, 7, 9, 3]
```

Note: in Python, within a list, you can have a mix of data types. Not true with most programing languages.

Using Base Address and Offset for Arrays

- The starting address for an array is its base address
  - Since an array can be defined anywhere in memory, the array base address is distinct from the memory base address
- You access elements of the array by adding a value to the array base address whose sum is the address to the appropriate element
  - The value that is added to the array base address is called the *array offset*
  - The array offset values are based on the data type
  - Array base address + array offset = array element memory address

Using Base Address and Offset for Arrays

- The offset is always an integer constant
  - Calculated using the index counter for the array element
  - Note this is identical to addressing memory in general
    - For arrays of bytes, offsets = index x 1
    - For arrays of halfwords, offsets = index x 2
    - For arrays of words, offsets = index x 4
    - For arrays of doublewords, offsets = index x 8

Note: arrays of single characters follows the byte addressing format. In some programming languages, arrays of strings are defined in a two-dimensional array. The 1st dimension is the number of elements and the 2nd dimension defines the maximum length of the string values. However, two-dimensional data structures in memory are allocated in sequential locations.

Calculating Offsets for Arrays

Example: array of int

```
int[] even = {2, 4, 6, 8, 10};
```

Base Addr = 0x10010100

	2	4	6	8	10
index	0	1	2	3	4
offset	**0**	**4**	**8**	**12**	**16**
mem addr	**0x10010100**	**0x10010104**	**0x10010108**	**0x1001010c**	**0x10010110**

Example: array of byte

```
byte[] odd = {1, 3, 5, 7, 9};
```

Base Addr = 0x10010116

	1	3	5	7	9
index	0	1	2	3	4
offset	**0**	**1**	**2**	**3**	**4**
mem addr	**0x10010116**	**0x10010117**	**0x10010118**	**0x10010119**	**0x1001011a**

Programming Offsets

- Registers are used to hold data values that change during program execution
- Working with arrays increases the requirement for registers
  - In addition to registers for array values, we need registers for the indexing, and calculating memory offsets
- The generally accepted way to generate memory addresses for sequential data values (i.e. arrays) is to calculate the actual memory address in a register
  - In code,
    - Calculate the offset using the index
    - Add the offset to the array base address to produce the memory address

Programming Offsets

- Since the memory address is fully specified in a register, the offset in the data transfer instruction is zero
  - When the hardware executes the data transfer instruction, offset zero is added to the address register which doesn't alter the memory address that was already calculated for the array access
- Arrays are normally processed via a loop structure when being sequentially accessed
- Loops will be an upcoming topic so for now, we will take a look at sequential access to see how the address computation is done

Example Program

- Using C

```
int main() {

  int fives[5] = {0, 0, 0, 0, 0};

  fives[0] = 5;
  fives[1] = 10;
  fives[2] = 15;
  fives[3] = 20;
  fives[4] = 25;

  return 0;
}
```

Obviously not the most efficient way to assign values to an array but will demonstrate addressing of arrays in memory.

<u>Register Associations</u>

s0 = array base address
s1 = array index
s2 = calculated array element offset
s3 = array element address
t0 = value to store in array

Example RISC-V Program

```
# RISC-V Program to Store Values in Array (SeqArrayFives.asm)

        .data
fives:  .word   0 0 0 0 0                   # allocate 5 element array
        .text
main:
        lui   s0, 0x10010       # base address of array in s0

        addi  t0, zero, 5       # first value to store in array
        or    s1, zero, zero              # initialize index s1 (0)
        slli  s2, s1, 2         # calculate x4 offset s2
        add   s3, s0, s2        # add offset to base address
        sw    t0, 0(s3)         # store the value to array[0]

        addi  t0, t0, 5         # next value to store in array
        addi  s1, s1, 1         # increment the index (1)
        slli  s2, s1, 2         # calculate x4 offset s2
        add   s3, s0, s2        # add offset to base address
        sw    t0, 0(s3)         # store the value to array[1]
```

(program continued on next slide)

Continued from Previous Slide

```
      addi  t0, t0, 5        # next value to store in array
      addi  s1, s1, 1        # increment the index (2)
      slli  s2, s1, 2        # calculate x4 offset s2
      add   s3, s0, s2       # add offset to base address
      sw    t0, 0(s3)        # store the value to array[2]

      addi  t0, t0, 5        # next value to store in array
      addi  s1, s1, 1        # increment the index (3)
      slli  s2, s1, 2        # calculate x4 offset s2
      add   s3, s0, s2       # add offset to base address
      sw    t0, 0(s3)        # store the value to array[3]

      addi  t0, t0, 5        # last value to store in array
      addi  s1, s1, 1        # increment the index (4)
      slli  s2, s1, 2        # calculate x4 offset s2
      add   s3, s0, s2       # add offset to base address
      sw    t0, 0(s3)        # store the index value to array[4]

exit: ori   a7, zero, 10
      ecall
```

Before & After Results of Code

array (after allocation & before code execution)

0x10010000					
	0	**0**	**0**	**0**	**0**
index	0	1	2	3	4
offset	0	4	8	12	16

array (after code execution)

0x10010000					
	5	**10**	**15**	**20**	**25**
index	0	1	2	3	4
offset	0	4	8	12	16

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x00000005	0x0000000a	0x0000000f	0x00000014	0x00000019

Example Exercise

- Write a sequential program to create an array with 5 32-bit numbers that start with 10 and decrement by 5 for each subsequent element

10	**5**	**0**	**-5**	**-10**

- Plan registers first: needed

<u>Recommended:</u>

  - memory base address s0
  - array base address s1
  - array index value s2
  - array offset value s3
  - calculated array element memory address s4
  - calculated element value t0
  - starting value t0 (reuse)
  - decrement value t1

Example Exercise (cont)

- Data segment:

```
# Program to create an array containing values [10 5 0 -5 10]

        .data
start:  .byte   10
decr:   .byte   5
array:  .word   0 0 0 0 0    # allocate 5-element array of words
```

- Alternatively, you could use the .space assembler directive to allocate empty space for the array
  - This would be preferable if a large array needed to be allocated

```
array:  .space  20  # allocate array space
```

Example Exercise (cont)

- Text segment:

```
        .text
main:   lui   s0, 0x10010  # memory base address
        addi  s1, s0, 4    # array base address
```

- Memory base address is defined with lui
- You will need to know the offset for the array relative to the memory base address
  - In the data segment, the array was defined after the two byte values start and decr
  - Since the array is defined as words, alignment restriction applies forcing the array to begin at an offset that is a multiple of 4

Example Exercise (cont)

- Continuing the text segment:
  - Code the instructions to load values from memory
  - Code an instruction to set the starting index value = 0

```
lb    t0, 0(s0)        # load start
lb    t1, 1(s0)        # load decr
or    s2, zero, zero       # set index (counter) = 0
```

- Continue with a sequence of instructions to
  - Calculate the array offset using the index value
  - Calculate the array element address (array base address + offset)
  - Store the first element value
  - Increment the index for the next element of the array

Example Exercise (cont)

- Program thus far:

```
        .data
start:  .byte   10
decr:   .byte   5
array:  .word   0 0 0 0 0       # allocate 5-element array of words

        .text
main:   lui     s0, 0x10010     # memory base address
        addi    s1, s0, 4       # array base address

        lb      t0, 0(s0)       # load start
        lb      t1, 1(s0)       # load decr
        or      s2, zero, zero  # set index (counter)

        slli    s3, s2, 2       # calculate array offset (x4)
        add     s4, s1, s3      # calculate array element address
        sw      t0, 0(s4)       # store start as first element value
        addi    s2, s2, 1       # increment index
```

Example Exercise (cont)

- Next instruction sequence calculates next values for offset and value for storing the next array element

```
slli  s3, s2, 2     # calculate array offset
add   s4, s1, s3    # calculate array element address
sub   t0, t0, t1    # calculate next element value
sw    t0, 0(s4)     # store next element value
addi  s2, s2, 1     # increment index
```

- Note that this sequence ends with the instruction to increment the index for the next array element
  - This is done for a reason – eventually defining the array access in a loop where the index value will be used for loop termination – but here we are writing sequential code
- What comes next?

Example Exercise (cont)

- How many values have we written to the array?
  - The first two values
- How many values are left to be written?
  - Three more
- So, repeat the previous code segment without any changes three more times prior to the exit code.
  - On the last time, don't include the index increment because we have no more elements to write

```
      slli  s3, s2, 2      # calculate array offset
      add   s4, s1, s3     # calculate array element address
      sub   t0, t0, t1     # calculate next element value
      sw    t0, 0(s4)      # store next element value

exit: ori   a7, zero, 10
      ecall
```

Example Exercise (cont)

- Test the program by stepping through the code to observe how the register values are updated to produce the memory address for the next element of the array.
  - Specifically look for and track the following:
    - index increment
    - offset calculation
    - array address calculation
    - array value calculation
    - stored value into the array
  - Verify all values in the array are correct