

# Introduction to Computing

Array and Pointers

# Array

- Many applications require multiple data items that have common characteristics

*In mathematics, we often express such groups of data items in indexed form:*

$$\blacksquare \quad x_1, x_2, x_3, \dots, x_n$$

- Array** is a *data structure* which represents a **collection of data items** of the **same datatype** (e.g. *float/int/char/...*)

## Example:

```
int A[5], i;
```

```
for (i = 0; i < 5; ++i)  
    scanf("%d", &A[i]);
```

```
for (i = 0; i < 5; ++i)  
    printf("%d", A[i]);
```

# Array(contd.)

- Declaration

- `<type> <name>[<no_of_elements>]`
- `int a[100];`
- `float b[20];`

- Initialization

- `int a[5] = {2,4,5,2,6};`
- `int b[4] = {1,3,5}`

- Accessing an element of array

- `a[2] → 5`
- `b[0] → 1`
- `b[3] → ?`
- `a[5] → ?`

- Assignment of value later on in the program

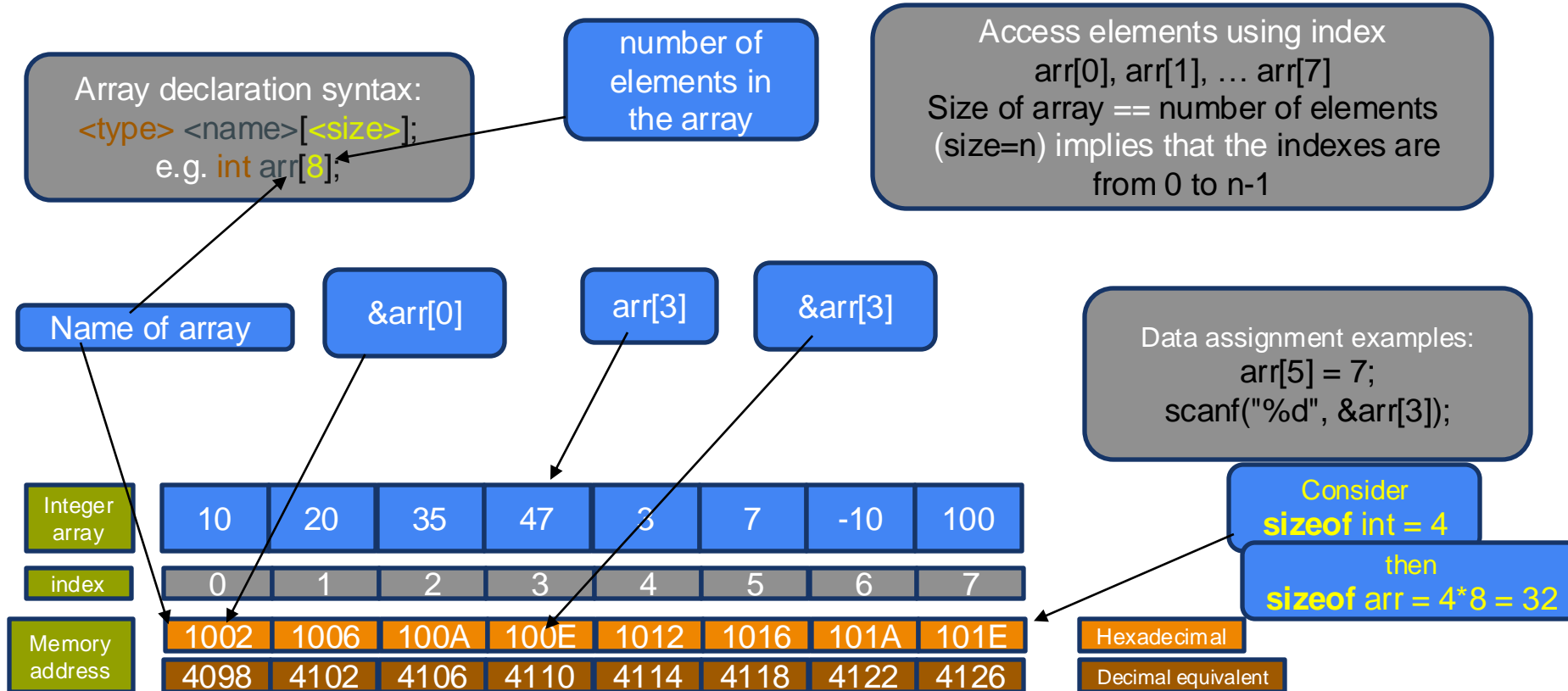
- It is same as a normal variable
- `b[3] = 3.14;`
- `a[2] = 1000;`

- A single variable has a name

- An array variable has a **<name>**

- It's a collection of single variables
- Variables are accessed using `<index>`
- Therefore, `<name>[<index>]` is a specific variable in an array

# Array (contd.)



## Array – examples to try

- Print all elements of an array
  - Scan elements into an array
  - Copy elements of one array into another
  - Sum of all elements in an array
  - Multiply all elements in an array
  - Find Min/Max element in an array
  - Search for an element in an array
  - Sum two equal-sized arrays element-wise, and store the results in another array
- Find minimum of a set of 10 numbers
  - Write the code in a way so that the code works for a set of any given number (i.e. not only 10)

## Array (contd.)

*Write the code in a way so that the code works for a set of any given number (i.e. not only 10)*

- Recall **const** qualifier
  - `const int size = 10;`  
`int A[size], i;`  
`for (i = 0; i < size ; ++i)`  
`scanf("%d", &A[i]);`
- Another way ...
  - `#define SIZE 10`
  - This is called a preprocessor/macro -- *we will learn about preprocessors later in the course*

# Searching for an Element (key) in an Array

- You have an array filled with integer elements
  - Can be hard coded
  - Can be user input
  - Can be read from file *<we will see how later how>*
- You take an integer (*key*) user input from user
- Search through the array to check if the *key* exists in the array
  - Go through the array one element at a time in using a loop
  - Check is the element matches the *key* or not
- Print appropriate message to show the result of the exercise
- This is called a *linear search*

# Functions (recall)

## Passing of variables

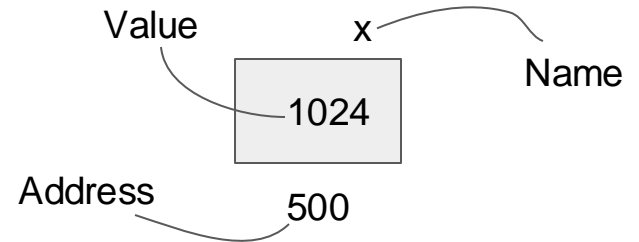
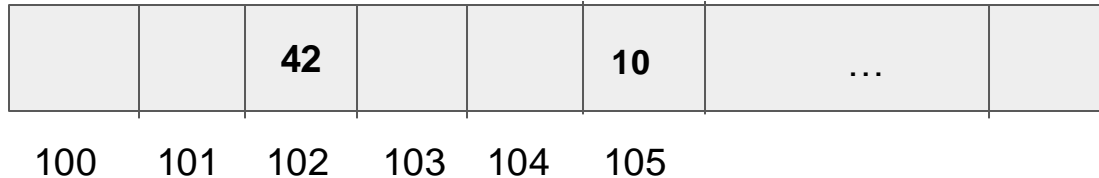
- Variables values are copied when then are passed (by calling) to a function
- The actual variables are not passed
- So a change made to a variable within a function will not reflect in the variable at the end of the caller – recall the swap function

- But scanf, which is also a function, *is able to change the values of a local variable – How does it do it?*
- Recall the AddressOf (&) operator
  - `scanf ("%d", &a);`
  - it sends (copies) the memory address of a variable
  - *scanf makes change in that memory location*
  - thereby changing the value of the variable



# Pointers

- Each memory cell (byte) has a unique address
- Each memory cell can hold a value
- Contiguous memory cells have sequential addresses



# Pointers

- Pointers are a special variables that can store memory addresses
- **Declaration** of a pointer variable
  - `<type> *<name>;`
  - Variable value (*memory address*) can be accessed using `<name>`
- **Access** the value at the stored address
  - `*<name>`
  - It will treat the value at the stored location as the declared `<type>`

Examples:

//actual variables

```
int a; float b; char c;
```

//pointer variables

```
int *iptr;   float *fptr;   char *cptr;
```

```
a = 10; //set value of a as 10
```

```
iptr = &a; //address of variable a
```

```
printf("%p", iptr); // will print address of a
```

```
printf("%d", *iptr); // will print value of a
```

## Pointers (contd.)

```
int a=10; // a is an integer variable, initialized with value 10
int *ptr; // ptr is an integer pointer variable, uninitialized
printf ("%d", a);    ⇒ 10
printf ("%p", ptr);  ⇒ <some garbage value as an memory address>
printf ("%p", &a);   ⇒ memory address of the variable a
printf ("%p", &ptr); ⇒ memory address of the variable ptr
ptr = &a;            //stores the address of a on ptr
printf ("%p", ptr);  ⇒ value of ptr / address of the variable a
printf ("%d", *ptr); ⇒ access data as integer at the location stored in ptr
printf ("%p", &ptr); ⇒ the address of the variable ptr; remains the same
```

# Pointer types: Size

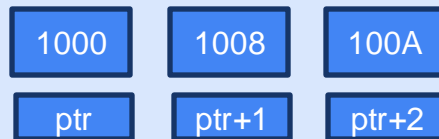
- *It depends on the maximum possible number value for address in a machine*
- **A 64-bit processor allows the machine to have 64 bit address - so it needs 8 bytes to store that address**
  - `sizeof (int) ⇒ 4, sizeof (int*) ⇒ 8`
  - `sizeof (char) ⇒ 1, sizeof (char*) ⇒ 8`
  - `sizeof (double) ⇒ 8, sizeof (double*) ⇒ 8`
  - `sizeof (long double) ⇒ 16, sizeof (long double*) ⇒ ?`

*You can check using*

*`printf("%ld %ld", sizeof (long double), sizeof (long double*))`*

# Pointer Arithmetic

```
int a;           //consider sizeof int as 8
int *ptr = &a;
```



pointer + integer

ptr + 1 will be translated as value stored in ptr + sizeof int  
ptr + 2 will be translated as value stored in ptr + 2 \* sizeof int  
i.e., ptr+i will be translated as value stored in ptr + (i \* sizeof int)

Similarly for char \*cptr; cptr+i will yield value stored in cptr + (i \* sizeof char),  
for double \*dptr; dptr+i will yield value stored in dptr + (i \* sizeof double), etc.

<type>\* ptr + <int\_val> is equivalent to ptr + <int\_val> \* sizeof(<type>)

# Array and Pointers

- Array elements are accessed using indexes
  - `int arr[10];`
    - Allocates a memory block equal to the size of 10 integers in total
    - Elements accessed as `arr[0]`, `arr[1]`, etc.
  - The **arr** is the address of the entire memory block; **it is of type `int*`** (read as *integer pointer*)
  - Therefore It can also be accessed similar to pointers variables
  - So **`*arr` is `arr[0]`**
    - How do you access the rest? → you can use pointer arithmetic

## Array and Pointers (contd.)

- *Adding 1 to a pointer variable means increasing the value of the pointer by the size of the type of that pointer*
- *Adding 1 to an `int*` variable means adding `sizeof(int)` to the value of the variable*

So,

`arr[1] == *(arr + 1), arr[2] == *(arr + 2), ...`

i.e., `arr[i] = *(arr + i)`

Also,

`arr + i = &arr[i]`

# Functions and Pointers

- Since variables passed to the functions are basically a copy
- Pointers to the variables are used instead of a variable to pass the **reference** to a variable - only when required
  - Addresses of the variable is copied
  - Changes made by function are done to the memory address
  - So when function exits, it only forgets the memory location and not the changes made at that location

So, Let's recall Swap

```
void swap (int a, int b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

```
void swap (int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```



# In The Next Class...

- You will learn about functions and arrays
- You will learn about structures
- You will learn about strings