Week 3

XMUT-NWEN 241 - 2024 T2

Systems Programming

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

Content

- Arrays

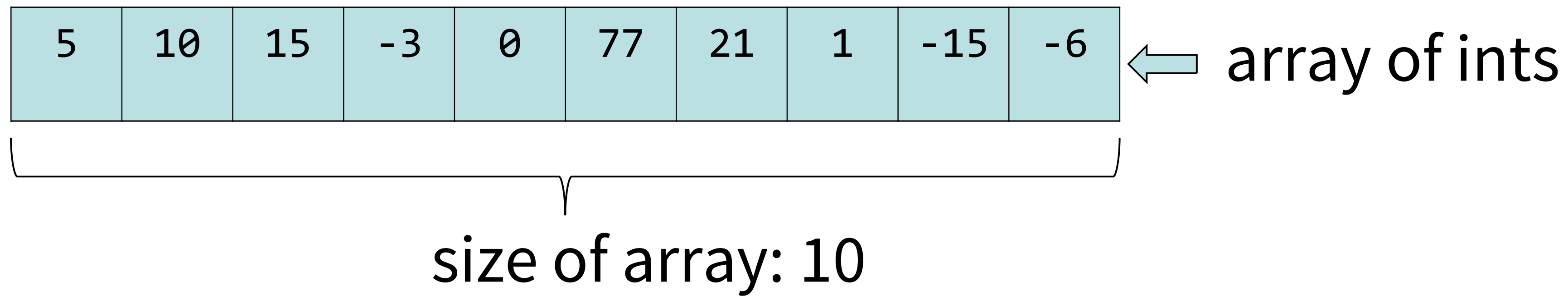
Arrays

Arrays

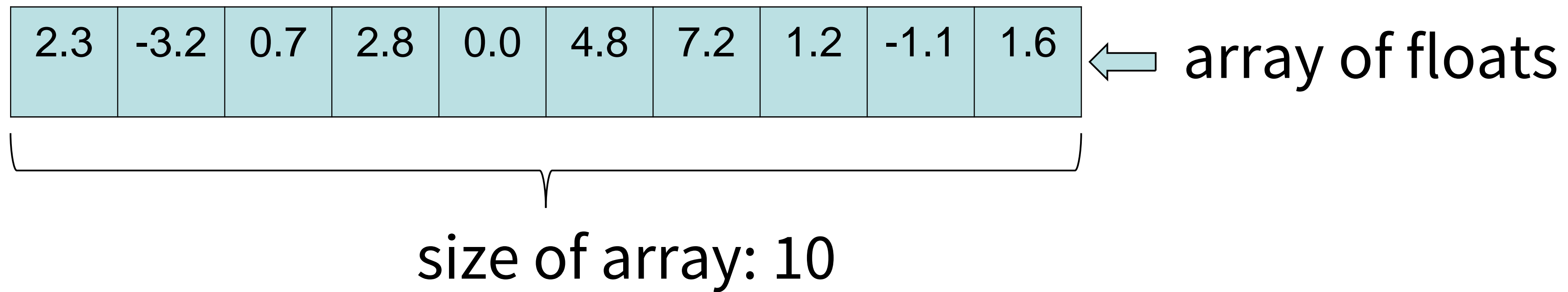
- An array is a collection of data that holds a **fixed** number of data (values) of the **same type**

- We distinguish between two types of arrays:
  - One-dimensional arrays
  - Multi-dimensional arrays
    - The C language places no limits on the number of dimensions in an array, though specific implementations may

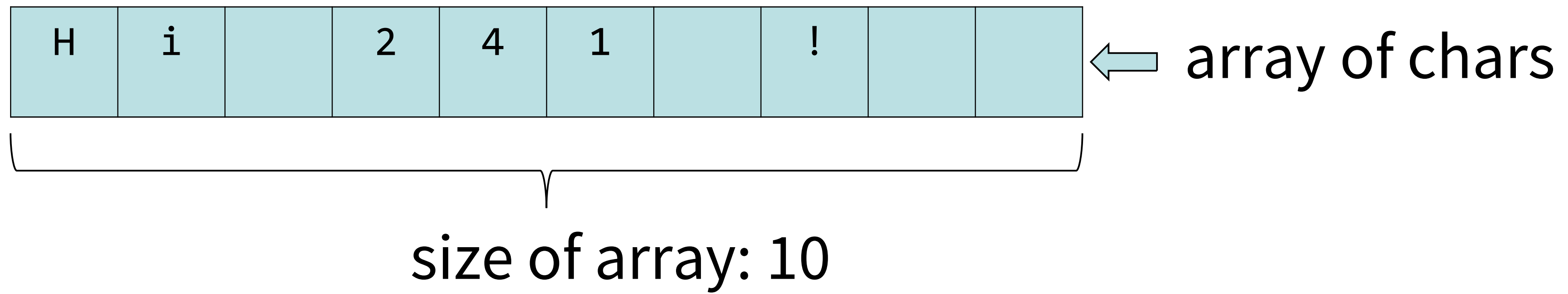
One-Dimensional Array Overview (1)



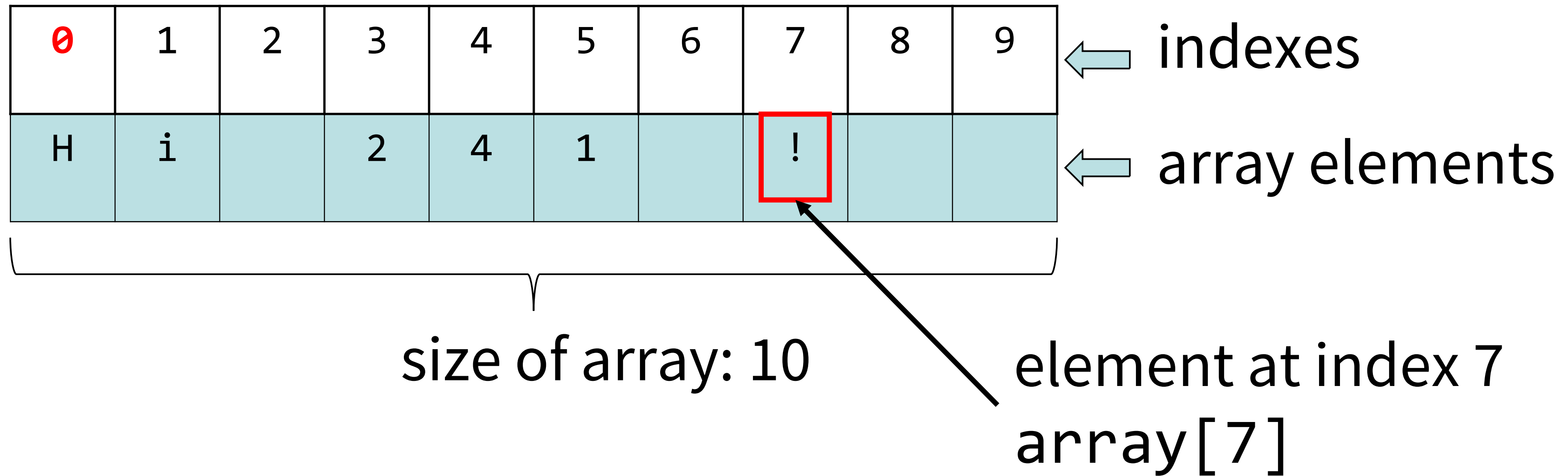
One-Dimensional Array Overview (2)



One-Dimensional Array Overview (3)



One-Dimensional Array Overview (4)



Arrays

- The simplest interpretation of an array is one-dimensional array, often referred to as a list

- The individual elements of the array can be accessed via **indixes**
  - The first index of an array starts at `0`
  - If the size of an array is `n`, to access the last element the index `n-1` is used
  - This is because the index in C is actually an *offset* from the beginning of the array
    - The first element is at the beginning of the array, and hence has zero offset

Declaring Arrays

- Declaring arrays in C differs slightly compared to Java
- Syntax for **declaring** a one-dimensional array:

- Example:

```
data_type array_name[size];
```

  - We declare an array named **data** of **float** type and size **4** as:

```
float data[4];
```

  - It can hold 4 floating-point values

- The **size** and **type** of arrays **cannot** be changed after their declaration!

Initializing Arrays (1)

- Arrays can be initialized **one-by-one**
- For example:

```
float data[4];
data[0] = 22.5;
data[1] = 23.1;
data[2] = 23.7;
data[3] = 24.8;
```

- In the case of large arrays this method is inefficient

Initializing Arrays (2)

- Arrays can be also initialized when they are **declared** (just as any other variables):

```
float data[4] = {22.5, 23.1, 23.7, 24.8};
```

- An array may be **partially initialized**, by providing fewer data items than the size of the array

```
float data[4] = {22.5, 23.1};
```

  – The remaining array elements will be automatically initialized to zero

- If an array is to be completely initialized, the dimension (size) of the array is not required

```
float data[] = {22.5, 23.1, 23.7, 24.8};
```

  – The compiler will automatically size the array to fit the initialized data

Arrays and Loops

- Arrays are commonly used in connection with **loops**
  - in order to perform the same calculations on all (or some part) of the data items in the array:

```
int array[10] = {1, 2};
```

```
int idx = 0;
while(idx < 10) {
	/* do something with array[idx] */
	idx++;
}
```

```
for (int idx = 0; idx < 10; idx++){
	/* do something with array[idx] */
}
```

Off-By-One Error

- The most common mistake when working with arrays in C is forgetting that indixes start at `0` and stop one less than the array size
  - We often refer to this issue as "**off-by-one error**"

```
int data[]={1,2,3,4,5}; /* number of elements is 5 */
for (int idx = 0; idx <= 5; idx++){
      /* do something with data[idx] */
}
```

- The compiler does not control the limits of the array!
- This type of error can be detected using static code analysis
  - For example using the cppcheck tool

Determining Size of Array

- The size of an array can be determined using the `sizeof()` operator
- It will return the ***number of bytes*** *the array "occupies" in the memory*
- To determine the number of elements in the array, the **returned** value must be **divided** by the **number of bytes** reserved for the **data type** **!**

Determining Size of Array

```
int data[] = {1, 2, 3, 4, 5};
int bytes, len;

/* Print number of bytes used by array */
bytes = sizeof(data);
printf("Bytes used: %d\n", bytes);

/* Print number of elements or items in array */
len = sizeof(data)/sizeof(int);
printf("Number of items: %d\n", len);

/* To traverse array, use number of elements as limit */
for (int idx = 0; idx < len; idx++) {
      /* do some stuff on element data[idx] */
}
```

Passing 1D Arrays to Functions (1)

- Passing a **single array element** to a function
  - can be passed in a similar manner as passing a variable to a function

```
void display(int a) {
   printf("%d", a);
}

int main(void) {
   int age[] = { 18, 19, 20 };

   display(age[2]); /* Passing element age[2] only */

   return 0;
}
```

Passing 1D Arrays to Functions (2)

- Passing an **entire array** to a function
  - When passing an array as an argument to a function, it is passed by its memory address (starting address of the memory area) and not its value (call-by-address)!
  - Because a function accesses the original array values, we must be very careful that we do not inadvertently (accidentally) change values in an array within a function.

```
float average(int a[]) {
      int sum = 0;
      for (int i = 0; i < 6; ++i)
            sum += a[i];
      float avg = ((float)sum / 6);
      return avg;
}
int main(void) {
      int age[] = {18,19,20,21,22,23};
      float avg = average(age);
      printf("Average age=%.2f\n", avg);
}
```

Passing 1D Arrays to Functions (2)

- Better design

```
float average(int a[], int len) {
        int sum = 0;
        for (int i = 0; i < len; ++i){
                sum += a[i];
         }
        float avg = ((float)sum / len);
        return avg;
}
int main(void) {
        int age[] = {18,19,20,21,22,23}, len;

        len = sizeof(age) / sizeof(int);

        float avg = average(age, len);
        printf("Average age=%.2f\n", avg);
}
```