Week 2 Lecture 2

NWEN 241
Systems Programming

Jyoti Sahni

Jyoti.sahni@ecs.vuw.ac.nz

Content

- Function-like macros (cont.)
- Arrays
- Introduction to Strings

Recap: Function-like Macro

- Just like functions, function-like macros can take arguments
  - Insert comma-separated parameter names between (and)
  - Parameter names must be valid identifiers

```
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
```

- Invoke just like normal functions

```
z = MAX(1, 3);
```

→

```
z = ((1)>(3)?(1):(3));
```

This expression evaluates to **3**

Problems with Function-like Macros

- Suppose:

```
#define SQ(X)       X * X
```

- Then:

```
(int)SQ(r);        →   (int)r * r;
SQ(r1 + r2);       →   r1 + r2 * r1 + r2;
```

- Solution: enclose individual variables with (), including the whole replacement text

```
#define SQ(X)       ((X) * (X))
```

Problems with Function-like Macros

- Suppose:

```
#define SQ(X)       ((X) * (X))
```

- Then:

```
(int)SQ(r);      →    (int)((r) * (r));
SQ(r1 + r2);     →    ((r1 + r2) * (r1 + r2));
```

Problems with Function-like Macros

- Suppose:

```
#define SQ(X)       ((X) * (X))
```

- How about these:

```
SQ(++r);
```

→

```
((++r) * (++r));
```

`r` incremented twice

```
SQ(f());
```

→

```
((f()) * (f()));
```

`f()` invoked twice

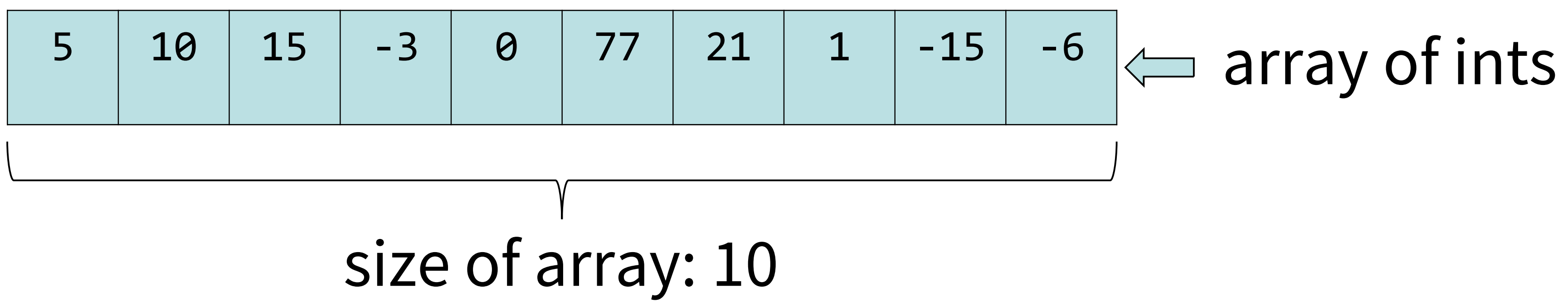
Be careful when defining and calling function-like macros!

Arrays

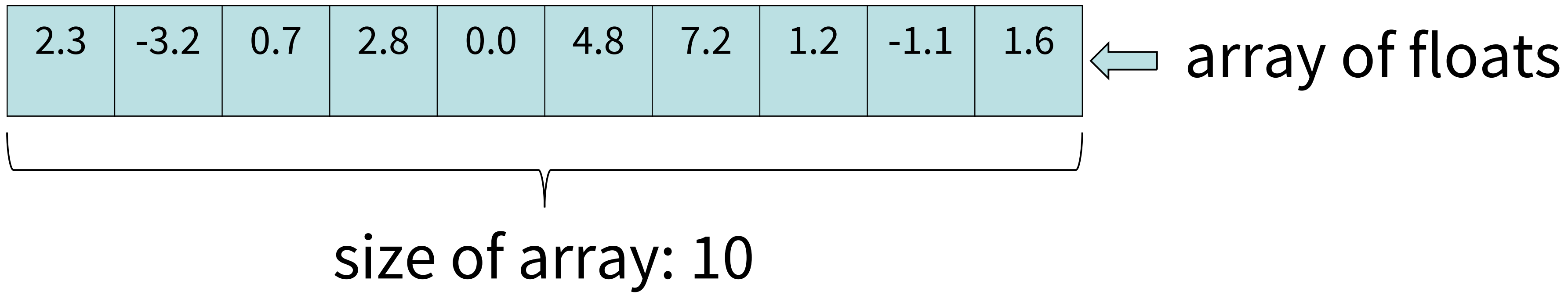
- An array is a collection of data that holds a **fixed** number of data (values) of the **same type**

- 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

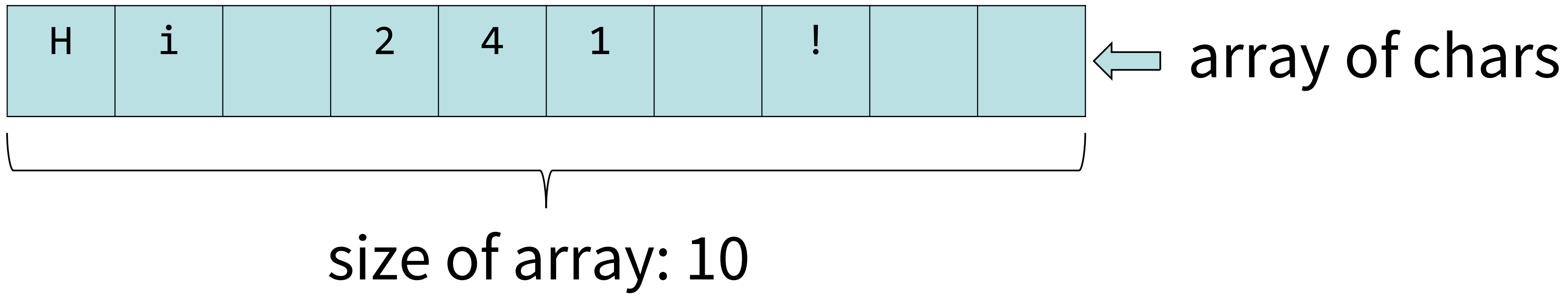
One-Dimensional Array Overview (1)



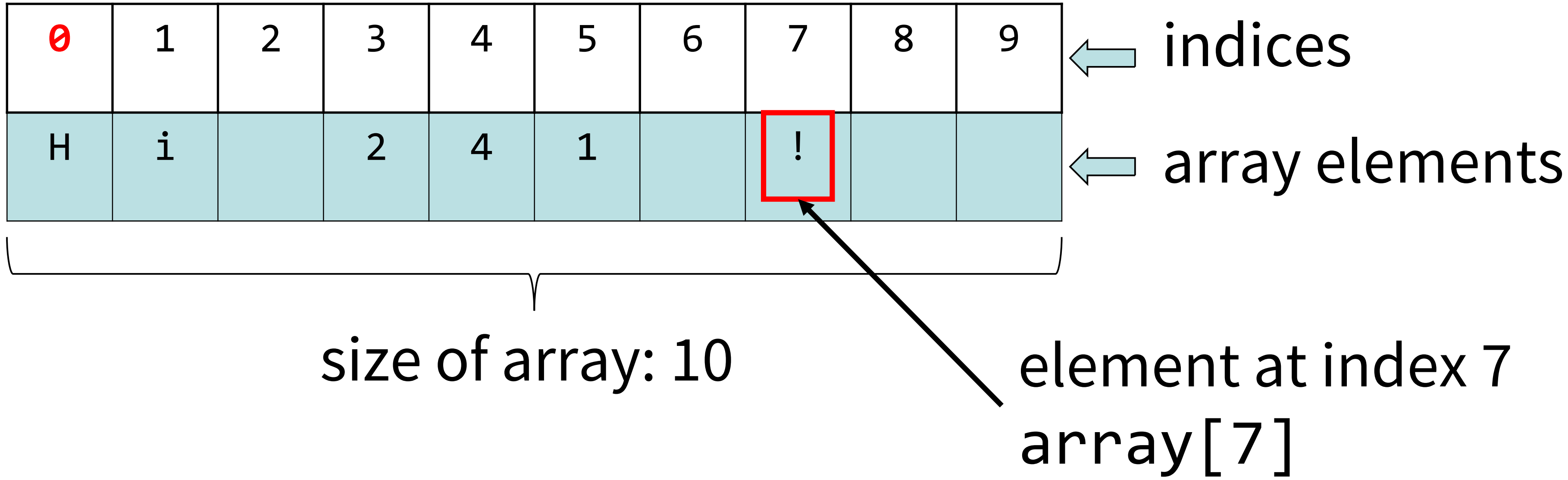
One-Dimensional Array Overview (2)



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 **indices**
  - 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:

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

- Example:
  - 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!
- Array size is fixed at compile-time, cannot be changed during run-time

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 <u>inefficient</u>

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 conjunction 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 indices 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 <u>control the limits of the array</u>!
- This type of error can be detected using static code analysis
  - For example using the <u>cppcheck</u> 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 <u>**single array element**</u> 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!

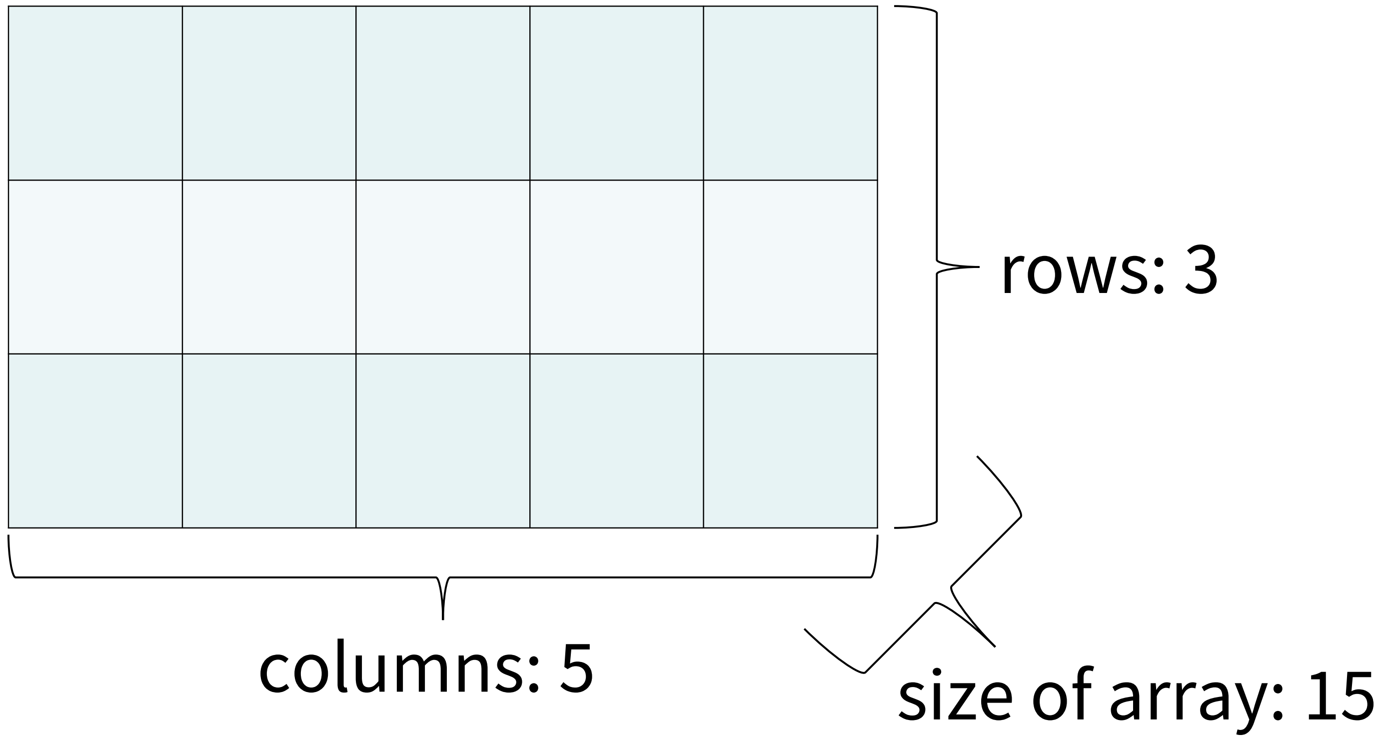
```
float average(int a[]) {
      int sum = 0;
      for (int i = 0; i < 6; ++i)
            sum += a[i];
      float avg = ((float)sum / 6);
      return avg;
}
int main(void) {
      int age[] = {18,19,20,21,22,23};
      float avg = average(age);
      printf("Average age=%.2f\n", avg);
}
```

Multi-dimensional Arrays

- In C, you can create array of an array known as multidimensional array

- The simplest interpretation of a multi-dimensional array is a table, i.e. a **two-dimensional array**
  - each row has the same number of columns

Two-Dimensional Arrays Overview (1)



Two-Dimensional Arrays Overview (2)

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

⇦ array of ints

Two-Dimensional Arrays Overview (3)

1.0	2.0	3.0	4.0	5.0
6.0	7.0	8.0	9.0	10.0
11.0	12.0	13.0	14.0	15.0

⇐ array of floats

Two-Dimensional Arrays Overview (4)

	0	1	2	3	4
0	H	e	l	l	o
1		W	o	r	l
2	d		?	!	

element at row 3 column 4
`array[2][3]`

array elements

size of array: 15

Two-Dimensional Arrays

- Declaring a char array with 3 rows and 5 columns

```
char two_d[3][5];
```

  - The array can hold 15 char elements
- Accessing a value

```
char ch;
ch = two_d[2][4];
```

- Modifying a value

```
two_d[0][0] = 'x';
```

- The array can be initialized in one of the following ways

```
int two_d[2][3] = {{5, 2, 1}, {6, 7, 8}};
int two_d[2][3] = {5, 2, 1 , 6, 7, 8};
int two_d[][3] = {{5, 2, 1}, {6, 7, 8}};
```

  - The number of columns must be explicitly stated. The compiler will find the appropriate amount of rows based on the initializer list

Passing 2D Arrays to Functions (1)

- Passing a <u>single array element</u> 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[2][3] = { {18, 19, 20}, {21, 22, 23} };

   display(age[1][2]); /* Passing element age[1][2] only */

   return 0;
}
```

Passing 2D 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!

```
void enterData(int d[][10]) {
        /* Code for reading and saving data into 2D array */
}

int main(void)
{
        int data[10][10];

        enterData(data);
}
```

What is String in C?

- C language **does not support strings** as a basic data type
- A C string is just an array that contains ASCII characters terminated by the **null character `'\0'`**
- A C string is stored in an array of chars

H	i		2	4	1		!	!	\0

`"Hi 241 !!"`

H	i		2	4	1		!		

Not a valid string

String Length

- Number of bytes/characters **excluding** the null character

Entire string occupies 10 bytes

H	i		2	4	1		!	!	\0

String length = 9 bytes

- strlen() function in <string.h> returns the string length

String Literal vs String Variable

- In C, we distinguish between **string literals** and **string variables**
- A **string literal** refers to the string constant value which is stored in the read-only memory area of the program
- A **string variable** refers to a string that is stored in an array which can be modified

String Literal (1)

- Enclosed in double quotes (") and can contain character literals (plain and escape characters)
- Can be broken up into multiple lines (each line ends with \) or separated by whitespaces

```
"Hello, world"
```

```
"Hello" ", " "world"
```

```
"Hello, \
world"
```

String Literal (2)

- String literals may contain as few as **one** or **even zero** characters

- Do not confuse a single-character string literal, e.g. `"A"` with a character constant, `'A'`
  - The former is actually two characters, because of the null-terminator stored at the end

- An **empty string**, `""`, consists of only the null-terminator, and is considered to have a string length of zero, because the null-terminator does not count when determining string lengths

String Literal (3)

- String literals are passed to functions as *pointers* to a stored string. For example, given the statement:

```
printf("Hello world!\n");
```

  - The string literal `"Hello world!\n"` will be stored somewhere in memory, and the address will be passed to `printf()`
  - The first argument to `printf()` is actually defined as a `char *`

- We will revisit this when we talk about pointers

Operations on String Literals

- String literals may be subscripted

```
printf("%c\n", "Hello"[2]);    /* will print 'l' */
```

- Attempting to modify a string literal results in undefined behaviour, and may cause problems in different ways depending on the compiler, *e.g.*

```
"Hello"[2] = 'e';
```

Symbolic String Constants

- Similar to integer and float symbolic constants, symbolic string constants can be declared using `const` qualifier or `#define` pre-processor

```
const char *MSG = "Hello, world";
const char *MSG_A = "Hello, \
world";
const char *MSG_B = "Hello" ", " "world";
```

```
#define MSG   "Hello, world";
#define MSG_A "Hello, \
world"
#define MSG_B "Hello" ", " "world"
```

Next Lecture

- Strings (cont.)

- Structures