EEE243 – Applied Computer Programming

Arrays

ROYAL MILITARY COLLEGE OF CANADA
ELECTRICAL & COMPUTER ENGINEERING

GÉNIE ÉLECTRIQUE ET GÉNIE INFORMATIQUE
COLLÈGE MILITAIRE ROYAL DU CANADA

RMC CMR

Outline

- Review
- 1-D Arrays
- 2-D Arrays
- Multidimensional Arrays

Review

- Functions
  - declarations
  - definition
  - calls

```
float convert_temp(float temp_F);
```

```
float convert_temp(float temp_F) {
     return (5.0 / 9.0) * temp_F - (160.0 / 9.0);
}
```

```
int main() {
     float temp_F = 32;
     int temp_C = convert_temp(temp_F);
     return 0;
}
```

```
float convert_temp(float temp_F);
```

```
convert_temp(32);
```

Review

- Libraries

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

- Constants

```
#define ABSOLUTE_ZERO -459.67
```

```
EXIT_SUCCESS
```

- Global variables

- Integer division (5/9) = 0

Concept of arrays

- Imagine that we want to process 20 integers.
  - We want to read each integer from the keyboard
  - Then we also want to process them and perhaps display/print them
- In order to do that, we have to store these integers in memory for some time
- So we would need a reference to each of the 20 integers
  - each one a variable.

Concept of arrays

- Each time that we want to print or otherwise process the integers as a group, we would need 20 statements each with the appropriate variable name:

```
int number1, number2,…, number20;
…
printf("This is number1 : %d", number1);
printf("This is number2 : %d", number2);
…
printf("This is number20 : %d", number20);
```

Concept of arrays

- Although cut and paste works, we want to be able to express the same thing with fewer lines of code.
- Our example had 20 numbers, now imagine that you read a file with *20,000 integers*!!!
- We want to exploit the power of loops and *iterations*
- To use iterations we need some kind of *structure* that provides a *sequenced* collection of *information*

- The array!

One-dimensional array

- An array is a *fixed-length* sequence of elements of the *same type*
  - So an array has a type, it also has a name and a fixed number of elements
- The following is a declaration and definition of three arrays:

```
int scores[10];
char first_names[25];
float averages[12];
```

One-dimensional array

- If we want to use the power of loops to iterate through an array, we need a way to access each distinct element of the array.
- To do this we use an *index*
- An index is an integer *expression*.
- Arrays in C are *zero based*
  - the first index is 0 and the last one is the size of the declared array-1
  - ie: `int my_array[10]` is indexed from 0 to 9.

One-dimensional array

- So how does C know where the ith indexed element is?
- The name of the array is a symbolic reference to the first byte in the array: *An address*
- Because arrays are sequences of elements of the same type, the compiler can calculate an *offset* from the beginning of the array.
- If you use an *out of bounds* index for your array, the compiler will not tell you.

One-dimensional array - Init

- The initialization of an array can be done at declaration time just like variables. There are several ways of doing this init:

```
int numbers[5] = {1, 6, 9, 12, 321};
int numbers[] = {54, 8, -2};
int numbers[5] = {3, 8};
int numbers[5] ={0};
```

One-dimensional array - Init

- Accessing the value of an element:

```
numbers[2] = 1;
my_int = numbers[0];
```

- You cannot assign a complete array to another array, even if they match in type and size. You have to iterate through the arrays:

```
for (i=0; i<5; i++) {
  second_num[i] = first_num[i];
}
```

Array Indices

- MatLab array indices start at 1
- ‘C’ array indices start at 0
  - As do Python lists and sequences

- A 10-element array in MatLab is indexed
  - name(1), name(2), …, name(10)

- A 10-element array in C and Python is indexed
  - name[0], name[1], …, name[9]

Arrays and functions

- You can pass an array element to any function as long as the type of the element matches the type of the formal parameter:

```
float divide_by_3 (int myInt);
…
int numbers[5] = {7,9,14,20,323};
float result_of_div_by3[5];
…
for (i=0; i<5; i++) {
  result_of_div_by3[i] = divide_by_3(numbers[i]);
}
```

Arrays and functions

- The last example only used the values of, but did not modify the `numbers` array.
- But what happens if we want to process 20,000 values, we will have 20,000 function calls!!!
- It would be a lot more efficient to pass the two arrays as parameters and let the *called function* modify the float array in a single function call.

Arrays and functions

```
#define SIZE 5

void divide_by_3(int x[], float y[], int num_elements);

int main(void) {
    int numbers[SIZE] = {7,9,14,20,323};
    float result_of_div_by_3[SIZE];
    …
    divide_by_3(numbers, result_of_div_by_3, SIZE);
}

void divide_by_3(int x[], float y[], int num_elements) {
   int i;
   //note the use of numElements here
   for (i=0, i < num_elements, i++) {
     y[i] = x[i] / 3.0;
   }
}
```

Two-dimensional arrays

- In memory, the values of an array are stored in a sequence.
  - This makes sense since memory by definition is a sequence of bytes (an array)
- However, many problems that we want to solve in the world require that we think and be able to access certain data as two-dimensional in order to apply certain algorithms:
  - table or matrix (rows and columns)

Two-dimensional arrays

- In the C programming language, tables are treated like an array of arrays.

```
{
 {2,5,8},
 {1,9,11}
        };
```

- You declare a 2-D array like this:

```
int num_table[2][3]; //2 rows 3 columns
```

Two-dimensional arrays

- There are several ways to initialize a matrix

```
// huh?
int num_table[2][3] = {2,5,8,1,9,11}; //Huh?

// easier to read
int num_table[2][3] = {
              {2,5,8},
              {1,9,11}
                };

// not good practice
int num_table[][3] = {
              {2,5,8},
              {1,9,11}
               };
```

A {0} initialization still works. Parts not defined are filled with zeros.
An uninitialized array is filled with crap!

Two-dimensional arrays

- Accessing an element:

```
num_table[i][j] = some_int;
```

- To access elements in C you use `num_table[3][2]`
  - fourth array, third element in the array

- Parameter passing: *one element; by value*

```
void my_function(int one_int);
…
my_function(num_table[i][j]); //one elm by value
```

- When passing parameters *by value* the effect is limited to the called function

Two-dimensional arrays

- Parameter passing: *Whole array (2-D)*

```
#define MAX_ROWS     2
#define MAX_COLUMNS  3

void my_function(int table_of_ints[][MAX_COLUMNS]);

int main(void) {
   int num_table[2][3]; // Pretend some init happens here
   my_function(num_table); //The whole 2D array
}

void my_function(int table_of_ints[][MAX_COLUMNS]) {
  statements…    //process the entire array
}
```

Multidimensional arrays

- Multidimensional arrays are only an extension of 2-D arrays.
  - This works for declaration, initialization and accessing of values
- The extra dimension for a 3-D array is called a plane.

- Of course in C, we see a 3-D array as an array of arrays of arrays
  - This concept holds true for other dimensions

Questions?