2.10 Arrays

Declaring and initializing arrays

Subscripting and iterating over arrays

Arrays as function parameters

EE108 – Computing for Engineers

1

Overview

- Aims
  - Learn the motivation for and use of arrays in C
- Learning outcomes – you should be able to…
  - Declare an array (including an optional intializer)
  - Subscript an array to get or set array elements
  - Iterate over an array
  - Write a function that takes an array parameter
  - Call a function that takes an array parameter

2

Motivation for arrays

3

- Arrays are the only collection data type native to C
- They are most useful for
  - Representing an ordered sequence of values (e.g. the time ordered sequence of sample values read from an ADC)
  - Implementing lookup tables (commonly used as an optimisation in embedded systems)
  - Part of the underlying implementation collection types not directly supported by C, including a buffer (or first in first out queue) and a stack (last in first out)
  - Representing text strings in C (covered in a later lecture)

3

Array concepts

4

- An **array** is a data structure used to contain a number of data values that have the same type
- Each value or item in the array is referred to as an **array element**
- The **array size or array length** (number of elements) is fixed at declaration time
  - It cannot be changed later
- The data type of the array elements is also fixed at declaration time
  - All elements in an array must have the same type
- The simplest (and most common) arrays are one dimensional, but multidimensional arrays are also supported

4

Visualizing arrays (to be filled in class)

5

Declaring an array

- Just like variables/constants of the primitive types (int, float, etc), array variables/constants must be declared before they can be used
- Unlike other variables, the size of the array must also be specified (*)
  - The compiler needs the size to know how much memory to reserve
  - (*) you can leave out the size if you use an appropriate array value initializer

```
int intArray[5];            // an uninitialized array of 5 elements
float floatArray[10];       // an uninitialized array of 10 elements
```

- Declaring an array without an initializer (as above) reserves the space but does not initialize or set the element values
  - you should assume that their values will be garbage until/unless you set them

6

Declaring an array with an initializer

- The initial value of an array's elements can be specified by including an array initializer in the declaration

```
int intArray[4] = { 101, 102, 103, 104 };
```

  - Initializer elements are surrounded by braces and separated by commas
  - Each element in the array initializer must be a constant expression (usually just a literal value or predefined constant)

- If the initializer is shorter than the array length, remaining elements are initialized to zero

```
int arrayOne[4] = { 101, 102 };      // value is now { 101, 102, 0, 0 }
int arrayTwo[4] = {0};               // value is now { 0, 0, 0, 0 }
```


Contd.

- You can actually leave out the array length when you have an initializer – the compiler infers the length itself from the initializer
- This can be useful for long arrays of pre-defined values (E.g. lookup tables)

```
int intArray[] = { 101, 102, 103 }; // array length is 3
```

- One possible disadvantage, when the compiler infers the size like this, is that your programme doesn't automatically have a convenient constant that defines the array length. It is possible to figure out as described on a later slide

Practice questions

Q. Declare an array to hold up to 10 floating point values.

Q. Declare an array to hold the first few prime (integer) values: 2, 3, 5, 7, 11, 13, 17 and 19. Do you need to specify the array size?

Array subscripting/indexing

- Each element in an array may be accessed to get or set its value by indexing/subscripting the array
- Valid element indexes range from 0 to array length – 1

```
int x;
int arr[] = { 101, 102, 103 }; // array length is 3

// getting the value of array elements
x = arr[0]; // x is 101
x = arr[2]; // x is 103
x = arr[3]; // INVALID INDEX: x is garbage but compiler doesn't give error

// setting the value of array elements
arr[1] = 0; // arr is now { 101, 0, 103 }
arr[3] = 0; // INVALID INDEX: *might* cause programme crash or error
```

Contd.

- Since it is important not to index past the end of the array, how do you know the array length?
- **Case 1**: you specified the array length
  - **Recommendation: always use a #define constant for an array length that you know in advance – don't use a literal value**

```
#define LEN    3 /* top of file */
int arr[LEN] = { 101, 102, 103 }; // somewhere else in your code

x = arr[LEN-1]; // x is 103
```

- **Case 2**: calculate the array length using sizeof
  - This is particularly useful when array length was left out of the declaration and inferred from initializer

```
int array[] = { 101, 102, 103 };
const int LEN = sizeof(array) / sizeof(array[0]);

x = array[LEN-1]; // x is 103
```


Contd.

- An array index can be any integral expression

```
#define N  10
int arr[N] = { 101, 102, 103 };
int i, j;

i=0;
j=2;
x = arr[i+j*2]; // i+j*2 evaluates to 4, so x is arr[4] which is 0
```

- The array index expression can even have side effects

```
#define N  10
int arr[N] = { 101, 102, 103 };
int i;

i=0;
x = arr[i++]; // corresponds to x=arr[i]; i=i+1; so x is…
arr[++i] = 2; // corresponds to i=i+1; arr[i]=2; so arr is…
```

Iterating forwards over an array

- To access all elements of an array use a loop
- To iterate forwards, start the index at 0 and increment it each time up to (array length – 1), i.e. less than array length

```
#define LEN  10
int arr[LEN] = {0};
int i, sum;

// for loop example
// initialize array to { 101, 102, 103, …, 110 }
for (i = 0; i < LEN; i++)
  arr[i] = 101 + i;

// while loop example
// calculate sum
sum = 0;
i = 0;
while (i < LEN) {
  sum += arr[i];
  i++;
}

// could also use (sizeof(arr) / sizeof(arr[0])) in place of N
// in both the for loop and while loop above
```


Iterating backwards over an array

- To iterate backwards, start the index at (arrayLength – 1) and decrement it each time down to 0 (note the greater-equals operator since 0 is a valid index)

```
#define LEN  10
int arr[LEN] = {0};
int i, sum;

// for loop example
// initialize array to { 101, 102, 103, …, 110 }
for (i = LEN-1; i >= 0; i--)
  arr[i] = 101 + i;

// while loop example
// calculate sum
i = N-1;
sum = 0;
while (i >= 0) {
  sum += arr[i];
  i--;
}

// could also use (sizeof(arr) / sizeof(arr[0])) in place of N
// in both the for loop and while loop above
```

Array as a function parameter

- To define a function that takes an array as a parameter
  - Define the parameter like an array variable with no length in the square brackets and no initializer
  - Add a second parameter to allow the length of the array to be passed in (since it cannot be calculated inside the function)

```
int calcSum(int arr[], int len) {
  int sum;
  int i;

  for (i = 0; i < len; i++) // iterate over the array
    sum += arr[i];

  return sum;
}
```

- To declare the function, copy the first line of the function definition as usual

```
int calcSum(int arr[], int len);
```


Contd.

- Then, to call the function, passing an array argument
  - Pass in the array name for the array parameter
  - Pass in the array length if the function accepts an array length parameter

```
#define VALUES_LEN    6
…
int values[VALUES_LEN] = { 1, 5, 6, 4, 3, 6 };
int result;

…
result = calcSum(values, VALUES_LEN);
```

Advanced: define a macro to calc array length

- You can use the highlighted macro in your own code if you need to calculate the size of an array
  - This only works if you have access to the original declaration of the array so you would usually do this immediately after declaring an array as shown

```
/**
 * macro to evaluate the length of an arbitrary array
 *
 * @param   the array to calculate the length of
 * @returns the number of elements in the array
 */
#define LENGTH_OF(a)     (sizeof(a) / sizeof((a)[0]))

…

int arr[] = { 101, 102, 103 };
const int LEN = LENGTH_OF(arr);      // use the macro to calculate
                                     // array length

x = arr[LEN-1]; // x is 103
```


Advanced: multi-dimensional arrays


```
#define N_ROWS    2
#define N_COLS    3

int matrixOne[N_ROWS][N_COLS]; // a 2 dimensional array with no initializer
int matrixTwo[N_ROWS][N_COLS] = { // a 2 dimensional array with initializer
    { 101, 102, 103 },      // 1st row
    { 201, 202, 203 }       // 2nd row
}

…
// iterating and indexing
int r, c, sum;

for (r = 0; r < N_ROWS; r++)
  for (c = 0; c < N_COLS; c++)
    sum += matrixOne[r][c]; // note brackets around each number, no comma
```

Code examples

Example sketches are provided on moodle which demonstrate:

- Basic array use with functions (as an input-only, or input-output parameter)
- Using arrays as lookup tables (e.g. to eliminate slow calculations)

Comments in the files explain the purpose of each and the concepts demonstrated.

19

Practice questions

Q. Write a program which reads a single digit number (1-9) and prints out the factorial of that number. Precalculate the factorials (outside the programme) and store them in a lookup table.

Q. Write a program which buffers up to 5 button clicks from SW1 or SW2. If SW2 is clicked, store a 2 in the buffer. If SW1 is clicked store a 1. When the buffer is full, print out the sequence of buttons clicked in reverse order (i.e. most recent first).

- *(Hint: iterate backwards over the array to reverse the sequence of buttons. Use an if statement to print out whether sw1 or sw2 was clicked based on each buffer element.)*

20