Computer Programming: Skills & Concepts (CP)
Arrays

Mahesh Marina

Monday 17 October 2016
What is an array?

An array is a collection of variables of the same type, grouped under a single name, with individual items being picked out via ‘indexing’.

Here is an example of *declaring* an array:

```c
int a[8];
```

We can make a similar *declaration* for any standard (int, float, double, char) or user defined type (coming in week 8), for any **constant** size (8 is the size for this example).
More about arrays

The declaration of `a` creates 8 individual variables ("elements", or "cells") organised at consecutive memory locations, accessible via "indexing".

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

(subscript or index)

To access the individual variables ("cells") in the array:

- `a[0]` is the 0th cell of array `a`
  - life is less confusing if we always count from zero
- `a[1]` is the 1st (1th?) cell
- ...  
- `a[7]` is the 7th (the final) cell
- `a[i]` is the `i`-th cell of `a`
  - assuming `i` is a variable of type `int` with value in the range 0...7
fibbonacci with arrays

Remember the Fibonacci function $F(n)$ in lecture 6.

- Defined via the following recurrence

$$F(n) = \begin{cases} 
0 & n = 0 \\
1 & n = 1 \\
F(n - 1) + F(n - 2) & \text{otherwise}
\end{cases}$$

- Programs fibonacci.c, fibonacci-for.c use variables previous, current and next to compute $F(n)$.
  - (good) Efficient in terms of number of variables - we have $n$, an counting variable called count, and the 3 above.
  - (bad) Ungainly, and error-prone, in the details of updating previous, current and next within the loop.

There is, of course, a simpler way!
We can define an array (called \texttt{fib}) to store the various Fibonacci numbers \( F(n) \) up to a limit (say 100).

**Advantages and Disadvantages**

- (good) We won't have to do the delicate arranging of previous, current on each iteration of the loop.
- (bad) We will have an upper limit on the values of \( n \) we can handle, because arrays must be constant-size.
  
  - In many languages, the size of an array can be assigned dynamically at run-time, but not in standard ANSI C. There is a way to get round it, but not until later.
"program design" - straight from the recursive definition of \( F(n) \)

```c
#define MAXFIB 100

int main(void) {
    int n, i;
    int fib[MAXFIB];

    fib[0]=0;
fib[1]=1;
.... /* omitting scanf for n */
if ((n < 0) || (n > MAXFIB-1)) {
    printf("Not an appropriate integer.\n");
} else {
    for(i=2; i <= n; i++) {
        fib[i] = fib[i-1]+fib[i-2];
    }
    printf("Fibonacci number %d is %d.\n", n, fib[n]);
}
return EXIT_SUCCESS;
}
```
The first element of `fib` has index 0, and the final element has index `MAXFIB - 1` (which is 99).

We refer to the entire array as `fib`.

All the *elements* (or *cells*) of the array have type `int`. We refer to these individual elements as `fib[0]`, `fib[1]`, and so on up to `fib[MAXFIB-1]` (or `fib[99]`).

Array indices are always expressions of type `int`.

The advantage of arrays is greatest when we can/need-to *iterate through the arrays via the use of a changing index variable* (this ‘index’ is `i` in the case of `fibonacci-arr.c`)

“Arrays are pointers” – `fib` is actually an address (of the first cell `fib[0]`) in memory.
More notes on fibonacci-arr.c

- Use of `#define`
  - `#define` just substitutes the value (100) for the identifier (MAXFIB) during gcc’s pre-processing step.
  - Can’t use `const int` in Standard ANSI C if the identifier will be used for an array index.
  - A cleaner alternative is `enum { MAXFIB = 100 };` which we’ll explain later – but `#define` is traditional.

- The bound on n that we can work with?
  - An artificial bound introduced because of array use (unfortunately).
  - An entirely reasonable limit for Fibonacci numbers as it happens.
  - As i grows, the value of $F(i+1)/F(i)$ tends to $(1 + \sqrt{5})/2$, roughly 1.61. So $F(i)$ grows exponentially.
  - The max value of an `int` in C on DICE is $2^{31} − 1$.
  - **As it happens** $F(i)$ becomes greater than $2^{31} − 1$ at 47
  - ... so we see negative numbers output (“wraparound” error) for 47 onwards
  - Even we use the 'long' (64-bit integer on DICE) type for `fib`, we will exceed max size for 'long' before $F(99) = 2.18 \times 10^{20}$. 

CP Lect 9 – slide 8 – Monday 17 October 2016
Initializing arrays

If you want to initialize an array to specific values, you can write:

```c
#define SIZE 8

/* initialize to the first 8 primes */
int a[SIZE] = { 2, 3, 5, 7, 11, 13, 17, 19 };
```

**Warning:** If you give too many values, gcc will complain; if you give too few, it will silently leave the last elements of the array uninitialized!
Where the power lies

An array index is a integer *expression*, not a *constant*, so its value isn’t determined until the program is run. The precise array element referred to by \( a[i] \) depends on the current value of \( i \)

Example:

```c
for (i = 0; i < SIZE; i++) { a[i] = 0; }
```

Effect: Initialise all elements of the array \( a \) to zero. Same as:

```c
a[0] = 0;
a[1] = 0;
...
a[SIZE - 1] = 0;
```

Be careful NOT to access cells with a later index than defined (eg \( i \) taking the value \( SIZE + 2 \)). C does not check array index limits.
#define MONTHS_IN_YEAR 12
#define DAYS_IN_WEEK 7

int main(void) {
    int day, month, days, i;
    /* WARNING: arrays start at zero, so January has index 0 */
    int daysinmonth[MONTHS_IN_YEAR] = { 31, 28, 31, 30, 31, 30,
                                      31, 31, 30, 31, 30, 31 };  
    char *daynames[DAYS_IN_WEEK] = {"Sunday","Monday", "Tuesday",
                                    "Wednesday", "Thursday",
                                    "Friday", "Saturday"};

    /* read the requested day and month in from user ... */
    days = day-1;                       /* first account for days since 1st */
    for (i=1; i < month; i++) {
        days = days + daysinmonth[i-1];
    }
    printf("It was a %s\n", daynames[(days+5)%DAYS_IN_WEEK]);
    return EXIT_SUCCESS;
}
Arrays of any type

We haven’t discussed typedef or struct formally yet . . . though we will see, in Lab sheet 4, these words used to define a type for representing points in the plane. An array of points could be used to represent a polygon with up to \( \text{MAX} \) vertices.

```c
typedef struct {
    int x, y;
} point_t;

point_t vertex[MAX];
```

Question: How do we deal with a polygon with fewer than \( \text{MAX} \) vertices?
Polygon as an array of vertices

- vertex[0] (0,0)
- vertex[1] (0,1)
- vertex[2] (1,0)
- vertex[3] (3,0)
- vertex[4] (4,1)
- vertex[5] (2,2)

Diagram:

```
(0,2)----------------------(2,2)
|                           |
|                           |
|                           |
(0,1)----------------------(4,1)
```

Array:

```
0          2          2          4          1          3          0
```

*CP Lect 9 – slide 13 – Monday 17 October 2016*
Arrays as parameters

int Max(int b[], int n) {
    /* n is the number of elements in array b. Max returns
     * the maximum element of b. NB: We lose the size of
     * the array when we pass it as a parameter */

    int i, maxSoFar;
    maxSoFar = b[0];
    for (i = 1; i < n; ++i) {
        if (b[i] > maxSoFar) { maxSoFar = b[i]; }
    }
    return maxSoFar;
}

....
printf("The maximum value is %d.\n", Max(a, 8));
void Rotate(int b[], int n) {
/* Aim: rotate the elements of a array cyclically. */
    int i;
    int temp;    /* Temporary storage (like in swap). */

    temp = b[n - 1];
    for (i = n - 1; i > 0; --i) { b[i] = b[i - 1]; }
    b[0] = temp;
}

....

Rotate(a, 8);

Question: Is a cyclically rotated or unchanged?
Arrays are “pointers”

The answer is that it is rotated. The reason? Roughly it is because an array in C is a pointer (to its zeroth element).

- The actual parameter \( a \) is a pointer to an integer.
- The formal parameter \( b[0] \) is a synonym for \( *b \).
- The formal parameter \( b[i] \) is a synonym for \( *(b+i) \).

**+ve:** Means we don’t need to use \& and \* to get the effect of “call-by-reference” with array parameters. (see swap.c).

**−ve:** We always have to incorporate an extra parameter (eg, \( n \) in Rotate) to allow the length of the array to be passed into the function.
Arrays of arrays

Array elements can themselves be arrays. So, for example, a matrix with \( N \) rows and \( M \) columns could be defined as:

```c
float matrix[N][M];
```

We’d then expect to be able to write a function that multiplies a vector \( x \) by a matrix \( a \) with header

```c
void LinTransform(float a[][],
                  float x[],
                  float y[],
                  int n, int m);
```

However C does not allow this - declaration for \( a \) must instead be of the form \( a[][][10] \) or \( a[][][8] \) or similar.

To understand why, check out Kelley & Pohl [KP, §6.12].
Reading Material

Relevant sections of Chapter 6, Kelley and Pohl.

- Specifically, 6.1, 6.4, 6.6 and 6.12