Chapter 6 part 1
C Arrays

C How to Program

Arrays

- An array is a group of *contiguous* memory locations that all have the *same type*.
- To refer to a particular location or element in the array, we specify the array's name and the position number of the particular element in the array.
- Figure 6.1 shows an integer array called `c`, containing 12 elements.
- Any one of these elements may be referred to by giving the array's name followed by the *position number* of the particular element in square brackets (`[]`).

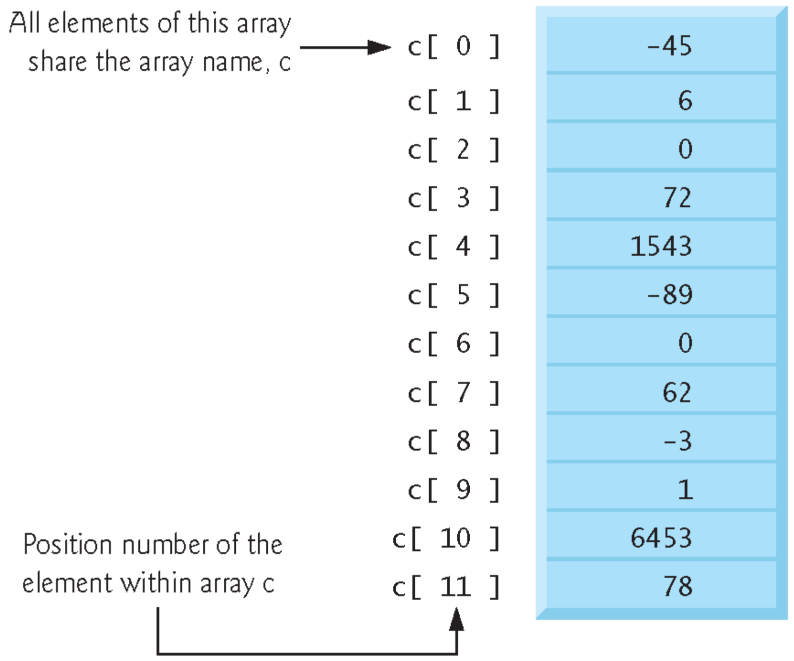

Fig. 6.1 | 12-element array.

Arrays (Cont.)

- The first element in every array is the zeroth element.
- An array name, like other variable names, can contain only letters, digits and underscores and cannot begin with a digit.
- The position number within square brackets is called a subscript.
- A subscript must be an integer or an integer expression.

Arrays (Cont.)

- For example, if `a = 5` and `b = 6`, then the statement
  - `c[ a + b ] += 2;`
- adds 2 to array element `c[11]`.
- A subscripted array name can be used on the left side of an assignment.

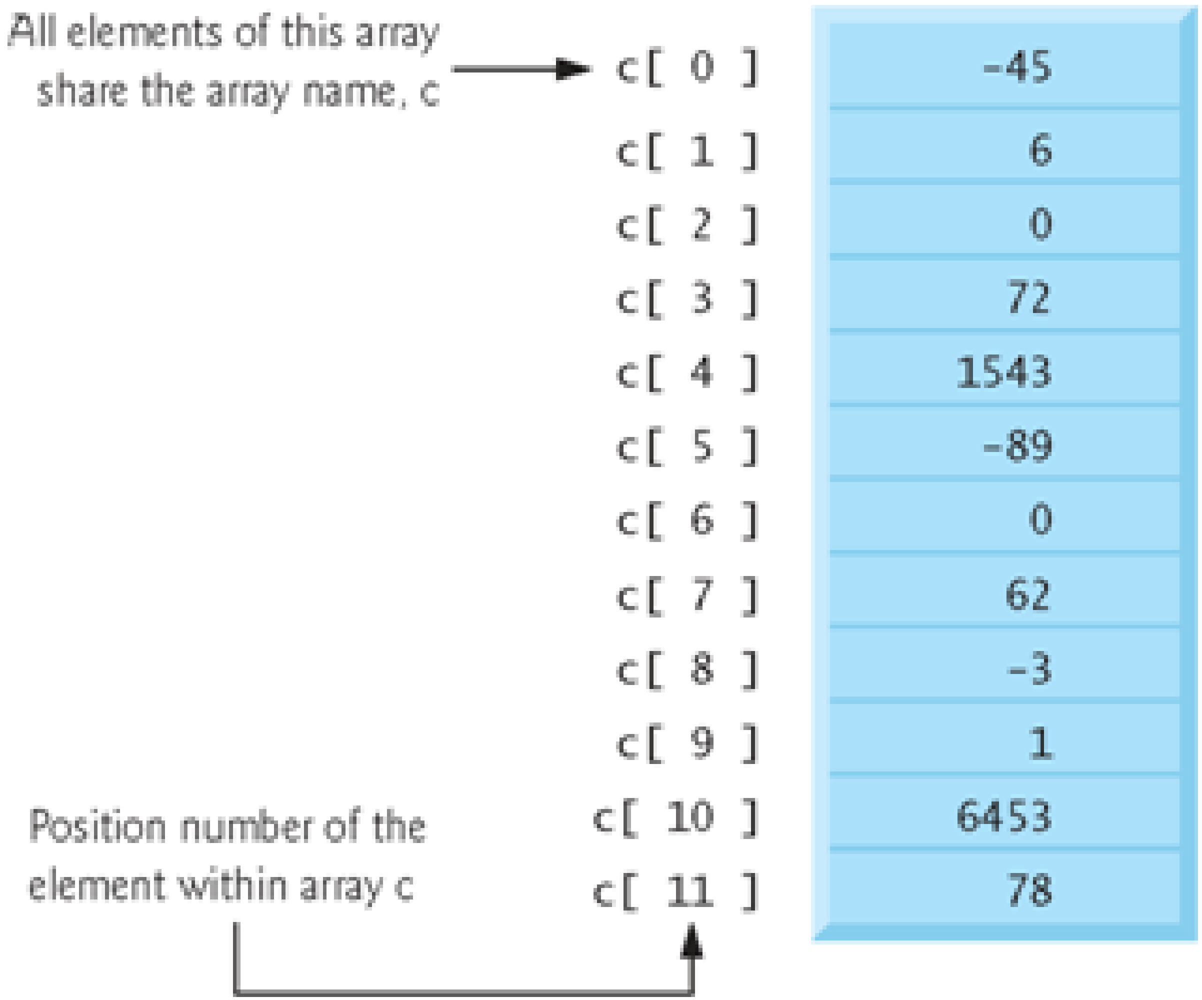

Fig. 6.1 | 12-element array.

Arrays (Cont.)

- The array's name is `c`.
- Its 12 elements are referred to as `c[0]`, `c[1]`, `c[2]`, …, `c[10]` and `c[11]`.
- The value stored in `c[0]` is `–45`, the value of `c[1]` is `6`, `c[2]` is `0`, `c[7]` is `62` and `c[11]` is `78`.
- To print the sum of the values contained in the first three elements of array `c`, we'd write
  - `printf( "%d", c[ 0 ] + c[ 1 ] + c[ 2 ] );`

Defining Arrays

- Arrays occupy space in memory.
- You specify the type of each element and the number of elements each array requires so that the computer may reserve the appropriate amount of memory.
- The following definition reserves 12 elements for integer array `c`, which has subscripts in the range 0-11.
  - `int c[ 12 ];`

Defining Arrays (Cont.)

- The definition
  - `int b[ 100 ], x[ 27 ];`
- reserves 100 elements for integer array `b` and 27 elements for integer array `x`.
- These arrays have subscripts in the ranges 0–99 and 0–26, respectively.

Array Examples

```c
// Fig. 6.3: fig06_03.c
// Initializing the elements of an array to zeros.
#include <stdio.h>

// function main begins program execution
int main( void )
{
    int n[ 10 ]; // n is an array of 10 integers
    size_t i; // counter

    // initialize elements of array n to 0
    for ( i = 0; i < 10; ++i ) {
       n[ i ] = 0; // set element at location i to 0
    } // end for

    printf( "%s%13s\n", "Element", "Value" );

    // output contents of array n in tabular format
    for ( i = 0; i < 10; ++i ) {
       printf( "%7u%13d\n", i, n[ i ] );
    } // end for
} // end main
```

```
Element        Value
      0            0
      1            0
      2            0
      3            0
      4            0
      5            0
      6            0
      7            0
      8            0
      9            0
```

Array Examples (Cont.)

- Notice that the variable `i` is declared to be of type `size_t` (line 9), which according to the C standard represents an `unsigned int` type.
- This type is recommended for any variable that represents an array's size or an array's subscripts.
- Type `size_t` is defined in header `<stddef.h>`, which is included by other headers (such as `<stdio.h>`).

Initializing an Array in a Definition with an Initializer List

- The elements of an array can also be initialized when the array is defined by following the definition with an equals sign and braces, `{}`, containing a comma-separated list of array initializers.
- Figure 6.4 initializes an integer array with 10 values (line 9) and prints the array in tabular format.

```c
// Fig. 6.4: fig06_04.c
// Initializing the elements of an array with an initializer list.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   // use initializer list to initialize array n
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
   size_t i; // counter

   printf( "%s%13s\n", "Element", "Value" );

   // output contents of array in tabular format
   for ( i = 0; i < 10; ++i ) {
      printf( "%7u%13d\n", i, n[ i ] );
   } // end for
} // end main
```

```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```

Array Examples (Cont.)

- If there are *fewer* initializers than elements in the array, the remaining elements are initialized to zero.
- For example, the elements of the array `n` in Fig. 6.3 could have been initialized to zero as follows:

```
int n[ 10 ] = { 0 };
```

- This *explicitly* initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than there are elements in the array.

Array Examples (Cont.)

- It’s important to remember that arrays are not automatically initialized to zero.
- You must at least initialize the first element to zero for the remaining elements to be automatically zeroed.
- Array elements are initialized before program startup for *`static`* arrays and at runtime for *`automatic`* arrays.

Array Examples (Cont.)

- The array definition
  - `int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };`
- causes a syntax error because there are six initializers and *only* five array elements.

Array Examples (Cont.)

- If the array size is *omitted* from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list.

- For example,
  - `int n[] = { 1, 2, 3, 4, 5 };`

would create a five-element array initialized with the indicated values.

Type the following program in Codeblocks

```c
// Fig. 6.5: fig06_05.c
// Initializing the elements of array s to the even integers from 2 to 20.
#include <stdio.h>
#define SIZE 10 // maximum size of array

// function main begins program execution
int main( void )
{
   // symbolic constant SIZE can be used to specify array size
   int s[ SIZE ]; // array s has SIZE elements
   size_t j; // counter

   for ( j = 0; j < SIZE; ++j ) { // set the values
      s[ j ] = 2 + 2 * j;
   } // end for

   printf( "%s%13s\n", "Element", "Value" );

   // output contents of array s in tabular format
   for ( j = 0; j < SIZE; ++j ) {
      printf( "%7u%13d\n", j, s[ j ] );
   } // end for
} // end main
```

```
Element        Value
      0            2
      1            4
      2            6
      3            8
      4           10
      5           12
      6           14
      7           16
      8           18
      9           20
```

Symbolic Constants

- `#define SIZE 10`

  defines a symbolic constant `SIZE` whose value is `10`.

- A symbolic constant is an identifier that's replaced with replacement text by the C preprocessor before the program is compiled.

- When the program is preprocessed, all occurrences of the symbolic constant `SIZE` are replaced with the replacement text `10`.

Symbolic Constants (Cont.)

- In Fig. 6.5, we could have the first `for` loop (line 13) fill a 1000-element array by simply changing the value of `SIZE` in the `#define` directive from `10` to `1000`.
- If the symbolic constant `SIZE` had not been used, we'd have to change the program in *three* separate places.

Good Programming Practice 6.1

Use only uppercase letters for symbolic constant names. This makes these constants stand out in a program and reminds you that symbolic constants are not variables.

Self Review Exercise - Arrays

- Create and initialize an array with 10 **random** integer number elements from 1 to 100.
- Loop through the array and print all the elements of the array.
- Write a program that computes and prints the sum of all the elements in the array.

Using Arrays to Summarize Survey Results

- The next example uses arrays to summarize the results of data collected in a survey.

- Consider the problem statement.
  - Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (1 means awful and 10 means excellent). Place the 40 responses in an integer array and summarize the results of the poll.

```c
#include <stdio.h>
#define RESPONSES_SIZE 40 // define array sizes
#define FREQUENCY_SIZE 11

// function main begins program execution
int main( void )
{
   size_t answer; // counter to loop through 40 responses
   size_t rating; // counter to loop through frequencies 1-10

   // initialize frequency counters to 0
   int frequency[ FREQUENCY_SIZE ] = { 0 };

   // place the survey responses in the responses array
   int responses[ RESPONSES_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
        1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
        5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };

   // for each answer, select value of an element of array responses
   // and use that value as subscript in array frequency to
   // determine element to increment
   for ( answer = 0; answer < RESPONSES_SIZE; ++answer ) {
      ++frequency[ responses [ answer ] ];
   } // end for

   // display results
   printf( "%s%17s\n", "Rating", "Frequency" );

   // output the frequencies in a tabular format
   for ( rating = 1; rating < FREQUENCY_SIZE; ++rating ) {
      printf( "%6d%17d\n", rating, frequency[ rating ] );
   } // end for
} // end main
```

```
Rating        Frequency
     1                2
     2                2
     3                2
     4                2
     5                5
     6               11
     7                5
     8                7
     9                1
    10                3
```

Array Examples (Cont.)

- *C has no array bounds checking to prevent the program from referring to an element that does not exist.*

- Thus, an executing program can “walk off” either end of an array without warning.

- You should ensure that all array references remain within the bounds of the array.

Graphing Array Element Values with Histograms

- Our next example (Fig. 6.8) reads numbers from an array and graphs the information in the form of a bar chart or histogram—each number is printed, then a bar consisting of that many asterisks is printed beside the number.
- The nested `for` statement (line 20) draws the bars.
- Note the use of `puts( "" )` to end each histogram bar (line 24).

```c
// Fig. 6.8: fig06_08.c
// Displaying a histogram.
#include <stdio.h>
#define SIZE 10

// function main begins program execution
int main( void )
{
   // use initializer list to initialize array n
   int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
   size_t i; // outer for counter for array elements
   int j; // inner for counter counts *s in each histogram bar

   printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );


    // for each element of array n, output a bar of the histogram
    for ( i = 0; i < SIZE; ++i ) {
       printf( "%7u%13d        ", i, n[ i ]) ;

       for ( j = 1; j <= n[ i ]; ++j ) { // print one bar
          printf( "%c", '*' );
       } // end inner for

       puts( "" ); // end a histogram bar
    } // end outer for
 } // end main
```

```
Element        Value        Histogram
      0           19        *******************
      1            3        ***
      2           15        ***************
      3            7        *******
      4           11        ***********
      5            9        *********
      6           13        *************
      7            5        *****
      8           17        *****************
      9            1        *
```

Fig. 6.8 | Displaying a histogram. (Part 2 of 2.)

Using Character Arrays to Store and Manipulate Strings

- We now discuss storing *strings* in character arrays.
- A string such as `"hello"` is really an array of individual characters in C.
- A character array can be initialized using a string literal.
- For example,
  - `char string1[] = "first";`

  initializes the elements of array `string1` to the individual characters in the string literal `"first"`.

Using Character Arrays to Store and Manipulate Strings

- `char string1[] = "first";`

- In this case, the size of array `string1` is determined by the compiler based on the length of the string.

- The string "`first`" contains five characters *plus* a special *string-termination character* called the null character.

- Thus, array `string1` actually contains six elements.
- The character constant representing the null character is `'\0'`.

- All strings in C end with this character.

Using Character Arrays to Store and Manipulate Strings

- Character arrays also can be initialized with individual character constants in an initializer list.
- The preceding definition is equivalent to
  - `char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };`
- Because a string is really an array of characters, we can access individual characters in a string directly using array subscript notation.
- For example, `string1[0]` is the character `'f'` and `string1[3]` is the character `'s'`.

Using Character Arrays to Store and Manipulate Strings

- We also can input a string directly into a character array from the keyboard using `scanf` and the conversion specifier `%s`.

- For example,
  - `char string2[ 20 ];`

  creates a character array capable of storing a string of *at most 19 characters* and a *terminating null character*.

- The statement
  - `scanf( "%19s", string2 );`

  reads a string from the keyboard into `string2`.

Using Character Arrays to Store and Manipulate Strings

- `scanf( "%19s", string2 );`

- The name of the array is passed to `scanf` without the preceding `&` used with nonstring variables.

- The `&` is normally used to provide `scanf` with a variable's *location* in memory so that a value can be stored there.

- This is because the value of an array name *is the address of the start of the array*; therefore, the `&` is not necessary.

Using Character Arrays to Store and Manipulate Strings

- `scanf( "%19s", string2 );`
- Function `scanf` will read characters until a *space*, *tab*, *newline* or *end-of-file indicator* is encountered.
- The `string2` should be no longer than 19 characters to leave room for the terminating null character.
- If the user types 20 or more characters, your program may crash or create a security vulnerability.
- For this reason, we used the conversion specifier `%19s` so that `scanf` reads a maximum of 19 characters and does not write characters into memory beyond the end of the array.

Manipulating Strings

- It's your responsibility to ensure that the array into which the string is read is capable of holding any string that the user types at the keyboard.
- Function `scanf` does ***NOT*** check how large the array is.
- Thus, `scanf` can write beyond the end of the array.

Manipulating Strings (Cont.)

- A character array representing a string can be output with `printf` and the `%s` conversion specifier.
- The array `string2` is printed with the statement
  - `printf( "%s\n", string2 );`
- Function `printf`, like `scanf`, does not check how large the character array is.
- The characters of the string are printed until a terminating null character is encountered.

Array Examples (Cont.)

- Figure 6.10 demonstrates initializing a character array with a string literal, reading a string into a character array, printing a character array as a string and accessing individual characters of a string.

Type the following program in Codeblocks

```c
// Fig. 6.10: fig06_10.c
// Treating character arrays as strings.
#include <stdio.h>
#define SIZE 20

// function main begins program execution
int main( void )
{
   char string1[ SIZE ]; // reserves 20 characters
   char string2[] = "string literal"; // reserves 15 characters
   size_t i; // counter

   // read string from user into array string1
   printf( "%s", "Enter a string (no longer than 19 characters): " );
   scanf( "%19s", string1 ); // input no more than 19 characters

   // output strings
   printf( "string1 is: %s\nstring2 is: %s\n"
           "string1 with spaces between characters is:\n",
           string1, string2 );

   // output characters until null character is reached
   for ( i = 0; i < SIZE && string1[ i ] != '\0'; ++i ) {
      printf( "%c ", string1[ i ] );
   } // end for

   puts( "" );
} // end main
```

```
19 characters): Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Self Review Exercise – Strings

- Palindrome: a word that reads the same backward as forward, e.g., madam.

- Write a program that checks if a given string is Palindrome.

- Get a word from the user and save it in a String
- Reverse the string
- Use a for loop to compare both strings

Static Local Arrays and Automatic Local Arrays

- A `static` local variable exists for the *duration* of the program but is *visible* only in the function body.

- We can apply `static` to a local array definition so the array is not created and initialized each time the function is called and the array is *not* destroyed each time the function is exited in the program.

- This reduces program execution time, particularly for programs with frequently called functions that contain large arrays.

Static Local Arrays and Automatic Local Arrays

- Arrays that are `static` are initialized once at program startup.
- If you do not explicitly initialize a `static` array, that array's elements are initialized to *zero* by default.

Common Programming Error 6.6

Assuming that elements of a local `static` array are initialized to zero every time the function in which the array is defined is called.

```c
// Fig. 6.11: fig06_11.c
// Static arrays are initialized to zero if not explicitly initialized.
#include <stdio.h>

void staticArrayInit( void ); // function prototype
void automaticArrayInit( void ); // function prototype

// function main begins program execution
int main( void )
{
   puts( "First call to each function:" );
   staticArrayInit();
   automaticArrayInit();

   puts( "\n\nSecond call to each function:" );
   staticArrayInit();
   automaticArrayInit();
} // end main

```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 1 of 5.)

```c
// function to demonstrate a static local array
void staticArrayInit( void )
{
   // initializes elements to 0 first time function is called
   static int array1[ 3 ];
   size_t i; // counter

   puts( "\nValues on entering staticArrayInit:" );

   // output contents of array1
   for ( i = 0; i <= 2; ++i ) {
      printf( "array1[ %u ] = %d  ", i, array1[ i ] );
   } // end for

   puts( "\nValues on exiting staticArrayInit:" );

   // modify and output contents of array1
   for ( i = 0; i <= 2; ++i ) {
      printf( "array1[ %u ] = %d  ", i, array1[ i ] += 5 );
   } // end for
} // end function staticArrayInit

```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 2 of 5.)

```c
// function to demonstrate an automatic local array
void automaticArrayInit( void )
{
   // initializes elements each time function is called
   int array2[ 3 ] = { 1, 2, 3 };
   size_t i; // counter

   puts( "\n\nValues on entering automaticArrayInit:" );

   // output contents of array2
   for ( i = 0; i <= 2; ++i ) {
      printf("array2[ %u ] = %d  ", i, array2[ i ] );
   } // end for

   puts( "\nValues on exiting automaticArrayInit:" );

   // modify and output contents of array2
   for ( i = 0; i <= 2; ++i ) {
      printf( "array2[ %u ] = %d  ", i, array2[ i ] += 5 );
   } // end for
} // end function automaticArrayInit
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 3 of 5.)

```
First call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 0  array1[ 1 ] = 0  array1[ 2 ] = 0
Values on exiting staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5

Values on entering automaticArrayInit:
array2[ 0 ] = 1  array2[ 1 ] = 2  array2[ 2 ] = 3
Values on exiting automaticArrayInit:
array2[ 0 ] = 6  array2[ 1 ] = 7  array2[ 2 ] = 8
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 4 of 5.)

```
Second call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5
Values on exiting staticArrayInit:
array1[ 0 ] = 10  array1[ 1 ] = 10  array1[ 2 ] = 10

Values on entering automaticArrayInit:
array2[ 0 ] = 1  array2[ 1 ] = 2  array2[ 2 ] = 3
Values on exiting automaticArrayInit:
array2[ 0 ] = 6  array2[ 1 ] = 7  array2[ 2 ] = 8
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 5 of 5.)