Brief Review of Arrays

There are times when we want to write programs that process homogeneous data consisting of a set of values with all values having the same data type. Suppose we require summary statistics such as maximum, minimum, average, and median grades for a midterm test in a course with 100 students. It is possible to write a program that uses a separate identifier to record each student's grade: `grade1`, `grade2`, ..., `grade100`. Performing computations using large sequences of variables is cumbersome and error-prone. Imagine the impracticality of writing a lengthy expression that computes the sum of the values in these 100 identifiers! Further, the program would only be useful for courses containing exactly 100 students. To compute summary statistics for a course with 200 students, a new program would have to be authored with 200 discrete identifiers to record the individual grade of each student.

To write a single program that works for all course enrollment sizes, a method for working with a group of values using a single identifier is required. Computer scientists have devised a number of structures such as linked lists, trees, and graphs to group related data items together into a single composite *data structure*. The *array* is the most fundamental and simplest data structure devised by computer scientists and is supported by every programming language.

Definition of arrays

An array is a collection of *homogeneous* data of the same type organized as a linear list of *elements* in *contiguous memory*. The general form of an array definition is:

```
data-type array-name[constant-integer-expression]
```

An identifier `array-name` is specified to refer to all the elements having the same type `data-type`. Identifier `array-name` is followed by a *constant positive integer* expression in brackets that specifies the size or number of elements in the array. The term *constant positive integer expression* means the compiler *must* be able to evaluate the expression to a positive integer value during compilation (and not during run-time).

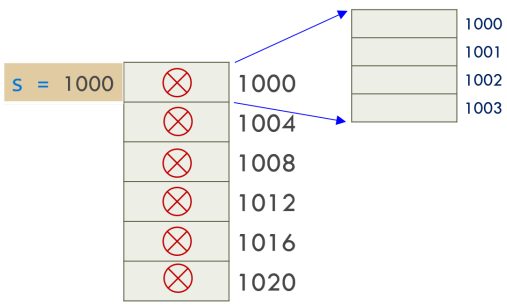
The following declaration statements define arrays `s`, `t`, and `v`:

```
int    s[6]; // s is array of 6 elements, each element of type int
double t[5]; // t is array of 5 elements, each element of type double
char   v[4]; // v is array of 4 elements, each element of type char
```

The definition

```
int s[6];
```

instructs the compiler to associate identifier `s` with a group of 6 adjacent and contiguous elements with each element capable of holding a value of type `int`. Supposing the compiler provides storage for the first element at address 1000, the memory representation of array `s` would look like this:



The compiler will provide storage for 6 adjacent elements with each element provided $sizeof\ (int) \equiv 4$ bytes necessary to store an `int` value. In total, the array requires $6 \times 4 = 24$ bytes of contiguous memory starting from address 1000 and up to and including address 1023 to store the six `int` values. Since the array is uninitialized, the garbage `int` value in each element is indicated by the $\otimes$ symbol. Finally, read-only variable `s` has type "array of `int`" and is initialized with the *base address* of the array where the first element is stored.

From this perspective, an array definition such as

```
int s[6];
```

consists of three specific parts that will be deduced by the compiler during the compilation stage:

1. The *size* of the array, that is, the number of elements associated with the array. In the case of array `s`, the array size is 6.
2. The *base type* of the array which determines how much memory each element requires and how to interpret the memory. The base type of array `s` is `int`.
3. The *base memory address* of the array, that is, the address where the first element is stored. The compiler will store the base memory address in read-only variable `s`. The actual base address of array `s` will be determined when the program is executed, but for the purposes of the example, it was assumed that base address is 1000. The compiler will then initialize read-only variable `s` with this base address 1000.

Next, consider the definition of another array `t`:

```
double t[5];
```

During compilation, the compiler will deduce the following three parts:

1. The array size is 5,
2. The base type is `double`, and
3. The base memory address, say memory address 2000, is stored in read-only variable `t`.

Arrays can be illustrated in column or row fashion with rows being more commonly used because they take up less space in a page. The following picture illustrates the memory layout in row order of array `t`:

	2000	2008	2016	2024	2032
t = 2000	⊗	⊗	⊗	⊗	⊗

Since array `t` has size 5, base type `double` that specifies $sizeof\ (double) \equiv 8$, and base address 2000, the compiler associates identifier `t` with 5 values of type `double` stored in a memory block of $5 \times 8 = 40$ bytes ranging from address 2000 and up to and including address 2039. Further, the compiler initializes read-only variable `t` with the array's base address of 2000. The $\otimes$ symbol in each element indicates the 5 memory values are uninitialized and will therefore contain garbage `double` values.

Variable length of arrays

C99 introduced the notion of variable length arrays. These arrays are declared like any other array (as described in the previous section) but with a size that is not a constant expression. Instead of array storage being allocated at compile time, the storage for variable length arrays is allocated at run time and deallocated when execution exits the block scope containing the array definition. Therefore, the following code is legal in C99:

```
printf("Enter number of students: ");
int N;
scanf("%d", &N);
double grades[N];
```

However, variable length arrays are not part of C++ and have been downgraded to an optional feature in later C standards starting from C11. For these reasons, this course will not permit the use of variable length arrays - this will be enforced by compiler option `-Werror=vla`.

Initialization

An array can be *optionally* initialized with declaration statements by specifying initializer values in a comma separated sequence and enclosed in curly braces. The following declaration statements not only define arrays `s`, `v`, and `t` but also initialize their elements:

```
int    s[6] = {5, -3, 2, -6, 11, -4};
char   v[5] = {'a', 'e', 'i', 'o', 'u'};
double t[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
```

If the initializing sequence is shorter than the array, then the rest of the elements are initialized to zero. Hence, to define an array of 100 double-precision floating-point elements where each element is required to be initialized to zero, the following definition suffices:

```
double grades[100] = {0.0}; // all 100 elements initialized to 0.0
```

If an array is specified without a size, but with an initialization sequence, the size is defined to be equal to the number of values in the sequence, as in:

```
int    s[] = {1, 11, 22}; // this array has 3 int elements
double t[] = {1.1, 2.2, -3.3, 4.4, -5.5}; // this array has 5 double elements
```

> *The size of an array must be specified in the declaration statement by using either a constant integer expression within brackets or by an initialization sequence within braces.*

Accessing array elements: subscript operator `[]`

Every array is defined with an identifier that applies to all the array elements. The compiler will define a read-only variable with the array identifier's name and further initialize this read-only variable with the array's base address. However, each individual element is anonymous, in that it doesn't have a specific name to identify itself. So how are individual array elements accessed or identified if they don't have associated names? Although an array element doesn't have an unique name, it has a *position* in the array. So if an element's position is known in the array, it can be accessed or identified using its position and the array's base address.

C uses *zero-based* positions to identify array elements, meaning that the first element has position at 0, the second element has position 1, and so on with the last element having position "*one less than array size*". Think of an element's position as an *offset* into the array: the first element has offset 0, the second element has offset 1, and so on. The terms *subscript* and *index* are also used to refer to an element's *position* or *offset*.

> *Array subscripts start with* 0 *and increment by* 1.

To reference an array element, the array's base address is first identified from the array identifier and then the desired element in the array is referenced using an *integral subscript*. C provides the *subscript operator* `[]` to reference array elements. The subscript operator takes two operands: the array *name* and an array *subscript* which is an integer expression that evaluates to the desired element's subscript. If `s` is defined as an array of 6 `int` elements

```
int s[6] = {5, -3, 2, -6, 11, -4};
```

subscripted variable `s[0]` (read as `s` sub-zero) refers to the first element of the array, `s[1]` to the second element of the array, and `s[5]` to the last element of the array. In general, subscripted variable `s[i]` refers to the $(i + 1)^{th}$ element of the array and has type `int`. The following figure illustrates the six elements of array `s`, the subscripted variables that reference these elements, and the corresponding memory locations that these references map to:

	1000	1004	1008	1012	1016	1020
s = 1000	5	-3	2	-6	11	-4
	s[0]	s[1]	s[2]	s[3]	s[4]	s[5]

How does the compiler actually reference the specific memory location associated with subscripted variable `s[i]`? The compiler performs the following computations to reference the memory location associated with subscripted variable `s[i]`:

1. The compiler trivially computes the *offset in elements* of subscripted variable `s[i]` from the 1^{st} element of array `s[0]` from subscript `i`:

$$offset\ in\ elements\ of\ variable\ s[i] = i - 0 = i$$

2. The compiler computes the *offset in bytes* of subscripted variable `s[i]` from the 1^{st} element by scaling the offset in elements by the memory bytes required to store each element:

$$\begin{aligned} offset\ in\ bytes\ of\ variable\ s[i] &= (offset\ in\ elements\ of\ variable\ s[i]) \times sizeof\ (int) \\ &= i \times sizeof\ (int)\ \text{bytes} \\ &= i \times 4\ \ \text{bytes} \end{aligned}$$

3. The compiler computes the *absolute address* of subscripted variable `s[i]` by adding the base address to the offset in bytes of `s[i]`:

$$\begin{aligned} absolute\ address\ of\ variable\ s[i] &= base\ address\ of\ array\ s + offset\ in\ bytes\ of\ variable\ s[i] \\ &= base\ address\ of\ array\ s + i \times 4 \end{aligned}$$

   Since base address of array `s` is stored in read-only variable `s`, the absolute address of subscripted variable `s[i]` is determined by expression `s + i*4`.

4. Since the base type of array `s` is `int`, the compiler will interpret 4 bytes starting from absolute address `s + i*4` as a value of type `int`.

The compiler doesn't remember individual addresses of each subscripted variable. Instead, it uses the base address and base type of an array to reference subscripted variables. If the compiler has to remember individual addresses of six subscripted variables of array `s`, think about the thousands of addresses that must be remembered for large arrays.

For example, consider the array definition:

```
1 double grades[1000];
```

When the compiler sees references to a subscripted variable such as `grades[3]`, it calculates the *offset in bytes* at which subscripted variable `grades[3]` is stored in memory from base address stored in `grades`. From subscript 3 in expression `grades[3]`, the compiler determines `grades[3]` is located at offset of 3 `double` variables past the base address. To compute the offset in bytes, the compiler scales offset 3 by `sizeof(double)` to get an offset in bytes of 24. Finally, the compiler computes the absolute address of subscripted variable `grades[3]` by adding the computed offset in bytes to base address in `grades`.

An important consequence of these computations that determine the absolute address of arbitrary element `s[i]` is that the compiler doesn't enforce any checks to ensure subscript `i` has a legitimate value ranging from 0 to one less than array size, that is, $i \geq 0$ and $i < size\ of\ array$. Perhaps the most common error in using arrays is accessing a nonexistent element:

```
int s[6];
s[6] = 10; // nonexistent 7th element
```

If your program accesses an array through an out-of-bounds subscript, there are no error messages. Instead, the program will quietly (or not so quietly) corrupt some memory where other variables are given storage. Except for short programs, in which the problem may go unnoticed, that corruption will make the program act unpredictably including and exhibit undefined behavior where anything is possible.

> *Using out-of-bounds subscripts are one of the most serious errors that can be introduced by programmers and can be difficult to detect.*

Just like other regular variables, a subscripted variable has memory storage and can thus be used anywhere a variable of similar type can be used. This means that arrays elements evaluate to *lvalue* expressions and therefore can be assigned values using assignment expressions. Any value that reduces to the proper type can be assigned to an array element. Just like ordinary variables, array elements can also be modified using pre- and post-fix increment and decrement operators. The only difference is that while an ordinary variable is referenced using its identifier, a subscripted variable is referenced using a subscript operator (and two associated operands consisting of array name and an integer subscript). The following code fragment illustrates the notion that a subscripted variable is no different than an ordinary variable:

```
int s[6] = {5, -3, 2, -6, 11, -4};

printf("%d", s[0]); // display value of subscripted variable s[0]
s[3] = 25; // assign (store) the value 25 in subscripted variable x[3]
// initialize variable sum with the sum of values in variables s[0] and s[1]
int sum = s[0] + s[1]; // sum is initialized with value 2
// add value in variables sum and s[2] and store result in variable sum
sum += s[2]; // sum is now 4
// add value in variables s[3] and sum and store result in variable s[3]
s[3] += sum; // s[3] is now 29
// add value in variables s[0] and s[3] and store result in variable s[2]
s[2] = s[0] + s[3]; // s[2] is now 34
// now increment value in s[3]
++s[3]; // s[3] is now 30
```

After the code fragment is executed, the six elements of array `s` will look like this:

	1000	1004	1008	1012	1016	1020
s = 1000	5	-3	34	30	11	-4
	s[0]	s[1]	s[2]	s[3]	s[4]	s[5]

The following code fragment re-emphasizes the need to understand the evaluation of the array subscript operator and the distinction between a *subscript value* and the corresponding *array element value*.

```
double x[8] = {16.0, 12.0, 6.0, 8.0, 2.5, 12.0, 14.0, -54.5};
int i = 5;
printf("%d %.1f\n", 4, x[4]); // displays 4 and 2.5 (value of x[4])
printf("%d %.1f\n", i, x[i]); // displays 5 and 12.0 (value of x[5])
printf("%.1f\n", x[i] + 1); // displays 13.0 (value of x[5] plus 1)
printf("%.1f\n", x[i] + i); // displays 17.0 (value of x[5] plus 5)
printf("%.1f\n", x[i + 1]); // displays 14.0 (value of x[6])
printf("%.1f\n", x[2 * i]); // error: attempt to display x[10]
printf("%.1f\n", x[2 * i - 3]); // displays –54.5 (value of x[7])
printf("%.1f\n", x[(int)x[4]]); // ok: displays 6.0 (value of x[2])
printf("%.1f\n", x[i++]); // ok: displays 12.0 (value of x[5])
                          // and then increments i to 6
printf("%.1f \n", x[--i]); // ok: assigns 5 (6 – 1) to i and then
                           // displays 12.0 (value of x[5])
```

```
x[i - 1] = x[i]; // assigns 12.0 (value of x[5]) to x[4]
x[i] = x[i + 1]; // assigns 14.0 (value of x[6]) to x[5]
x[i] - 1 = x[i]; // error: expression x[i]-1 is not an lvalue expression
```

If `i` is a subscript of array `x`, then subscripted value `x[i]` refers to the $(i + 1)^{\text{th}}$ array element. If `i` has value `0`, then the subscript value is `0` and the first element `x[0]` is referenced. If `i` has value `4`, then subscript value is `4` and the fifth element `x[4]` is referenced. Further, subscript `i` must only have values in range from `0` to `7` to refer to one of the eight elements of array `x`. If `i` has value `9`, the subscript value is `9` but the value of `x[8]` cannot be predicted because the subscript is referencing an out-of-bounds element.

Practice 1: Only use hand calculations

1. For the definition

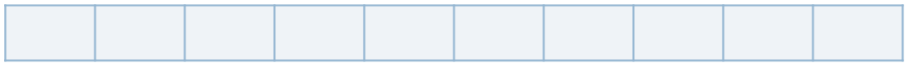
   ```
   int scores[8];
   ```

   how many memory cells are allocated for data storage? What type of data can be stored there? How does one refer to the initial array element? To the final array element?

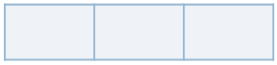
2. Define an array for storing the square roots of the integers from 0 through 10 and a second array for storing the cubes of the same integers. Give the arrays relevant and meaningful identifiers.

Show the contents of the arrays defined in each of the following sets of statements:

1. `int x[10] = {6, 3, 2};`



2. `char letters[] = {'x', 'y', 'z'};`



3. ```
   double z[4];
   z[1] = -2.5;
   z[2] = z[3] = fabs(z[1]); // fabs declared in math.h
   ```

4. Given the following code fragment

```
double x[] = {16.0, 12.0, 6.0, 8.0, 2.5, 12.0, 14.0, -54.5};
int i = 5;
printf("%d %.1f\n", 4, x[4]);
printf("%d %.1f\n", i, x[i]);
printf("%.1f\n", x[i] + 1);
printf("%.1f\n", x[i] + i);
printf("%.1f\n", x[i + 1]);
printf("%.1f\n", x[i + i]);
printf("%.1f\n", x[2 * i]);
printf("%.1f\n", x[2 * i - 3]);
printf("%.1f\n", x[(int)x[4]]);
printf("%.1f\n", x[i++]);
printf("%.1f \n", x[--i]);
x[i - 1] = x[i];
x[i] = x[i + 1];
x[i] - 1 = x[i];
```
```

determine which statements are valid and which are illegal? Explain the side effects of each valid statement. Finally, write the values of elements of array `x` after the execution of valid statements?

5. Repeat the previous question assuming line 2 is `int i = 2;`.

Using loops for sequential access to array elements

Most work with arrays is done with some kind of looping structure. For example, suppose we want to fill a `double` array `g` with values 0.0, 0.5, 1.0, ···, 10.0. Because there are 21 values, listing the values on the declaration statement would be cumbersome and tedious. Thus, the following `for` statement can be used to assign values to this array:

```
1  double g[21]; // define array g - array elements are uninitialized
2  for (int i = 0; i < 21; ++i) {
3    g[i] = i * 0.5; // assigning values to elements of array g
4  }
```

It is critical to recognize that the loop condition in the above `for` statement must specify the expression `i < 21` and not `i <= 21`, since there are 21 array elements whose indices range from `0` through `20` and are referenced as `g[0]` through `g[20]`.

An equivalent `for` statement can be written with the loop condition using `<=` operator:

```
1  double g[21]; // define array g - array elements are uninitialized
2  for (int i = 0; i <= 20; ++i) {
3    g[i] = i * 0.5; // assigning values to elements of array g
4  }
```

Novice (and even experienced) programmers often introduce *off-by-one* bugs in which the loop condition is incorrectly written to specify a subscript that is one value more than the largest valid subscript. Such errors can be very insidious and difficult to debug because they access values outside the array bounds but still within the program's allotted memory. Thus, rather than causing a clean segmentation fault, the bug causes memory corruption - which can induce in the young programmer frustration, then anger, fear, and aggression. This is the dark side of programming in C. Therefore, it is important to be careful about exceeding array subscripts. Pick either the style of using array size `i < 21` or the style of using the final element index `i <= 20` and stick with it. Although both work properly, sticking with a single style will avoid casual mistakes or detect mistakes by noticing changes in patterns from `<` to `<=` or from `<=` to `<` when reading your code.

> *The dark side of programming in C involves off-by-one bugs in which the loop condition is incorrectly written to specify a subscript that is one value more than the largest valid subscript.*

An equivalent `while` statement could be used to assign values to array elements:

```
1  double g[21]; // define array g - array elements are uninitialized
2  int i = 0;
3  while (i < 21) {
4    g[i] = i * 0.5; // assigning values to elements of array g
5    ++i;
6  }
```

Arrays elements evaluate to *lvalue* expressions and therefore can be modified using pre- and post-fix increment and decrement operators. The following code fragment increments values in array `g` by `1.0`:

```
// array g defined and assigned values in previous code fragment ...
for (int i = 0; i < 21; ++i) {
  ++g[i]; // g[i] += 1; OR g[i]++ would also be equivalent ...
}
```

The Fibonacci sequence is used as the final example to illustrate the power that programmers have to define arbitrarily large arrays and construct loops that modify these arrays. The Fibonacci sequence is defined as follows: the first two elements of the sequence are 1; then subsequent elements are defined as the sum of their two predecessors:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

In code, we've the following:

```
#include <stdio.h> // for printf

enum { N = 100 };

int main(void) {
  int fib[N]; // define but not initialize array of N integers
  // assign the first two values in sequence ...
  fib[0] = fib[1] = 1;
  // assign the remaining elements up to the Nth element: fib[N-1]
  for (int i = 2; i < N; ++i) {
    fib[i] = fib[i-2] + fib[i-1];
  }
  // print the first 10 elements of sequence ...
  for (int i = 0; i < 10; ++i) {
    printf("%i%c", fib[i], (i!=9) ? ' ' : '\n');
  }
  return 0;
}
```

We can visualize the first several iterations of the loop as follows with $\otimes$ indicating that the array element is uninitialized:

`i`	`fib[0]`	`fib[1]`	`fib[2]`	`fib[3]`	`fib[4]`	`fib[5]`
2	1	1	$\otimes$	$\otimes$	$\otimes$	$\otimes$
3	1	1	2	$\otimes$	$\otimes$	$\otimes$
4	1	1	2	3	$\otimes$	$\otimes$
5	1	1	2	3	5	$\otimes$
6	1	1	2	3	5	8

The first row indicates the variables' values just before the loop executes, but after `i` is initialized to `2`. Subsequent rows indicate their values at the end of each iteration. Recall that `++i` is executed after line 11 but before the condition `i < N` is evaluated on line 10. Hence, `i` has value `3` at the end of the first iteration.

The loop of the Fibonacci computation can be rewritten as a `while` loop:

```
int i = 2;
while (i < N) {
  fib[i] = fib[i-2] + fib[i-1];
  ++i;
}
```

Practice 2: Only use hand calculations

1. Show the contents of the array `time` after the execution of the following code fragment:

```
double time[9];
for (int k = 0; k <= 8; ++k) time[k] = (k-4)*0.1;
```

Assume array `s` is defined like this: `int s[] = {3, 8, 15, 21, 30, 41};` Using hand calculations, determine the output for each of the following sets of statements:

1.
```
for (int k = 0; k <= 5; k += 2) printf("%d %d\n", s[k], s[k+1]);
```

2.
```
for (int k = 0; k < 6; ++k)
  if (0 == s[k]%2)
    printf("%d ", s[k]);
printf("\n");
```

Give the corresponding memory snapshot after each of the following sets of statements is executed (use a question mark to indicate an array element that is not initialized).

```
int t[5];
t[0] = 5;
for (int k = 0; k <= 3; ++k) t[k+1] = t[k] + 3;
```

Assignment of arrays

Recall that the compiler associates an array name with the base address in memory where array elements are given contiguous storage. Since arrays cannot be relocated after they're defined, an array name cannot appear as the left operand of the assignment operator. Therefore, given two arrays that match fully in type and size, as in

```
enum { SIZE = 25 };
double first[SIZE], second[SIZE];
```

the compiler will flag the following assignment as an error.

```
first = second; // ERROR
```

Since array `first` has value "base address in memory where elements of array are given contiguous storage" and the assignment is changing the base address of `first` to the base address of array `second` where presumably the elements of array `second` are given contiguous storage, the compiler will flag the assignment as an error.

Since you cannot write an assignment expression where one array is assigned to another array, even if they match fully in type and size, the only recourse is to copy arrays at the individual element level. For example, to copy an array of 25 integers to a second array of 25 integers, your only recourse is to loop through individual elements of the two arrays:

```
enum { SIZE = 25 };
double first[SIZE], second[SIZE];

for (int i = 0; i < SIZE; ++i) {
  first[i] = i * 0.5; // assigning values to elements of array g
}
// sequentially copy each element of array first to corresponding
// element of array second
for (int i = 0; i < SIZE; ++i) {
  second[i] = first[i];
}
```

Passing scalar values to a function

To process arrays in a large program, you've to be able to pass them to functions. You can do this in two ways: pass individual elements or pass the whole array.

Since individual elements are referenced as subscripted elements, they can be passed to a function like any other variable. As long as the array element type matches the function parameter type, it can be passed. Of course, it will be passed as a value parameter, which means that the function cannot change the value of the element. Consider a function `print_square` that receives an integer and prints its square to standard output. You could iterate through an array (called `base` in the code block below) and pass each array element in turn to function `print_square`:

```
#include <stdio.h>

void print_square(int x); // Function prototype
enum { SIZE = 5 };

int main(void) {
  int base[SIZE] = {3, 7, 2, 4, 5};
  for (int i = 0; i < SIZE; ++i) {
    print_square(base[i]);
  }
  return 0;
}

void print_square(int x) {
  printf("%d => %d\n", x, x*x);
}
```

Note how only one element is passed at a time by using indexed expression `base[i]`. Since the value of this expression is a single `int`, it matches the parameter type in the declaration of `print_square`. As far as `print_square` is concerned, it doesn't know or care that parameter `x` is initialized with a value that came from an array.

Arrays as function arguments and parameters

Recall the *pass-by-value* mechanism of passing values to functions described above. The function call operator will evaluate its operands which are called *arguments*. Each *parameter* variable in the called function is initialized with the value of the corresponding argument.

For a function to operate on the whole array, it is reasonable to expect that the function must be passed the whole array. However, passing whole arrays is a situation in which C deviates from passing scalar values to a function. The reason for this change is that a lot of memory and time would be used in passing large arrays around every time you want to use one in a function. For example, if an array with 1024 elements were passed by value to a function, another array with 1024 elements would have to be allocated in the called function and each element would have to be copied from one array to the other. All

of this work would significantly reduce the execution speed of C programs. So, instead of passing the whole array, how can C allow a function to efficiently reference an array's elements?

Recall that when a C compiler sees the following array definition

```
double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
```

it deduces three pieces of information:

1. The array's *base type* which determines how much memory each subscripted variable requires. For array `grades`, the base type is `double` and since `sizeof(double)` evaluates to 8, each subscripted variable requires 8 bytes.
2. The array's *size*, that is, the number of subscripted variables. In this case, the array size is 5. The compiler uses this information to reserve a block of $5 \times sizeof\ (double) = 5 \times 8 = 40$ bytes of memory to contiguously store 5 variables of type `double`.
3. The *base memory address* of the first subscripted variable which is stored in read-only variable `grades`. Notice that the read-only variable's identifier matches the array name.

Further recall how the compiler uses the base address and base type of an array to reference subscripted variables. The compiler doesn't remember individual addresses of each subscripted variable. If it has to remember individual addresses of five subscripted variables of array `grades`, think about the thousands of addresses that must be remembered for large arrays. Instead, when the compiler sees references to a subscripted variable such as `grades[3]`, it calculates the *offset in bytes* at which subscripted variable `grades[3]` is stored in memory from base address stored in `grades`. From subscript 3 of subscripted variable `grades[3]`, the compiler determines `grades[3]` is located at offset of 3 `double` variables past the base address. The compiler computes the offset in bytes of 24 by scaling offset 3 by `sizeof(double)`. Further, the compiler computes the absolute address of subscripted variable `grades[3]` by adding the computed offset in bytes of 24 to base address in `grades`.

Now, let's suppose you wish to write a function `print_arr_dbl` that will print the contents of array `grades` (or any other array whose base type is `double`) to standard output. As the previous discussion has shown, all that is required for function `print_arr_dbl` to print elements of array `grades` is the base address and base type of array `grades`. How should the function parameter be declared so that function `print_arr_dbl` can perform offset computations to reference subscripted variables? What should the declaration and definition of function `print_arr_dbl` look like?

Since all that is required for a function to refer to array elements is the array's base type and base address and base type, a convenient way to pass an array to a function involves specifying the read-only variable `grades` as the function argument. When the function call is evaluated, the corresponding parameter in the called function is initialized with the base address stored in read-only variable `grades`.

The following code fragment illustrates how to pass an array's base address to function `print_arr_dbl` and how the function can use the passed base address to reference subscripted variables in the array.

```
#include <stdio.h>
void print_arr_dbl(double x[]); // prototype declaration

int main(void) {
  double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
  print_arr_dbl(grades);
  return 0;
}

void print_arr_dbl(double arr[]) {
  for (int i = 0; i < 5; ++i) {
    printf("%.2f%c", arr[i], i==4 ? '\n' : ' ');
  }
}
```

Let's begin our analysis by examining the call to function `print_arr_dbl` on line 6 of function `main`. Since array `grades` is defined as

```
1 double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
```

the read-only variable `grades` will contain the base address at which array `grades` is given storage. Therefore, the function call on line 6

```
1 print_arr_dbl(grades)
```

will evaluate expression `grades` to the base address of array `grades`.

For a function to receive the base address of an array through a function call, the function's parameter list must use special syntax to specify that an array's base address will be received. The function declaration (prototype) for function `print_arr_dbl` must be written as

```
1 void print_arr_dbl(double arr[]);
```

indicating that parameter `arr` will be initialized with the base address of an array. After the optional parameter name, you must write an empty `[]`. The size of the array is not necessary and if present is ignored by the compiler. However, the compiler will flag an error if the unnecessary size is a negative or zero value.

The base type of the array is indicated by type specifier `double`.

Since the parameter name is optional in function declarations, the function declaration can also be written as:

```
1 void print_arr_dbl(double []);
```

Now, let's consider the definition of function `print_arr_dbl`:

```
1 void print_arr_dbl(double arr[]) {
2   for (int i = 0; i < 5; ++i) {
3     printf("%.2f%c", arr[i], i==4 ? '\n' : ' ');
4   }
5 }
```

Using the base address in parameter `arr` and base type `double`, function `print_arr_dbl` can now make references to the subscripted variables of array `grades` in the calling function. Expression `arr[i]` is evaluated like this:

1. Using subscript `i`, the compiler trivially computes the *offset in elements* of subscripted variable `arr[i]` from the 1st element `arr[0]`:

$$offset\ in\ elements\ of\ variable\ arr[i] = i - 0 = i$$

2. The compiler computes the *offset in bytes* of subscripted variable `arr[i]` from the 1st element by scaling the offset in elements by the memory bytes required to store each element:

$$\begin{aligned} offset\ in\ bytes\ of\ variable\ arr[i] &= (offset\ in\ elements\ of\ variable\ arr[i]) \times sizeof\ (double) \\ &= i \times sizeof\ (double)\ \text{bytes} \\ &= i \times 8\ \ \text{bytes} \end{aligned}$$

3. The compiler computes the *absolute address* of subscripted variable `arr[i]` by adding the base address in parameter `arr` to the offset in bytes of `arr[i]`:

$$\begin{aligned} absolute\ address\ of\ variable\ arr[i] &= base\ address\ stored\ in\ arr + offset\ in\ bytes\ of\ variable\ arr[i] \\ &= base\ address\ in\ parameter\ arr + i \times 8 \end{aligned}$$

The absolute address of subscripted variable `arr[i]` is determined by expression `arr + i*8`.

4. Since the base type of variable `arr` is `double`, the compiler will interpret 8 bytes starting from absolute address `arr + i*8` as a value of type `double`.

Function `print_arr_dbl` has a potentially serious problem. Can you see it? What happens if the array `grades` in `main` has only 3 elements? Remember our discussion of what happens if the subscript gets out of range? This catastrophic scenario is depicted in the following code:

```
#include <stdio.h>
void print_arr_dbl(double x[]); // prototype declaration

int main(void) {
  double grades[3] = {11.1, 22.2, 33.3};
  print_arr_dbl(grades);
  return 0;
}

void print_arr_dbl(double x[]) {
  for (int i = 0; i < 5; ++i) {
    printf("%.2f%c", x[i], i==4 ? '\n' : ' ');
  }
}
```

Array `grades` in calling function `main` is defined to have size 3 but the called function `print_arr_dbl` makes the mistake of assuming that `grades` contains at least 5 elements. The function will access non-existent fourth and fifth elements by accessing out-of-bounds memory and the behavior of the program will be catastrophic - the compiler will not complain and the runtime program may or may not crash while printing values of non-existent array elements.

Although the array name when used as a function argument provides the base type and base memory address of its first element, the array argument doesn't tell the function the array size. When the code in the function body is executed, the run-time environment knows where the array begins in memory, and how much memory each subscripted variable uses, but it doesn't know how many subscripted variables the array has. That is why it is necessary to have an additional `int` argument telling the function the array size. The general convention is for the function's first parameter to be initialized with the base type and base memory address of the array and for the second parameter to be initialized with the number of elements to process from the base memory address (specified in the first parameter).

A function becomes more flexible when it is passed two parameters to specify the information of the array that it must process. The parameter initialized with the array's base type and base memory address specifies the first element of the array to be processed while the parameter initialized with the array size specifies the number of elements to be accessed starting from the base memory address. The first advantage is that the same function can be used for many arrays with different sizes but having the same base type. The second advantage is that the same function can be used for many different slices of the same array. That is, an array with 100 elements can provide the function the base memory address for its first element (having subscript 0) and a second parameter of `25` so that the function can process the first 25 elements of the array (ranging from subscripts `0` to `24`). In an other situation, the same function can be used to process the second half of the array by providing the function the base address for the 51^{st} element and a second parameter of `50`.

Using this information about how C thinks about arrays, the declaration and definition of function `print_arr_dbl` are implemented as shown in the following code fragment:

```
#include <stdio.h>
// declaration of print_int_array function
```

```c
void print_arr_dbl(double array[], int num_elements);

int main(void) {
  double grades[5]  = {11.1, 22.2, 33.3, 44.4, 55.5};
  double more_grades[] = {1.0, 2.0, 3.0, 4.0, 5.0,
                          6.0, 7.0, 8.0, 9.0, 10.0};

  /*
  The array name, in this case grades, means the address of the first
  index variable of an array of double variables.
  The first parameter provides the base type (double) and the base memory
  address of the first element of the array (with index 0).
  The second parameter provides the number of elements to be accessed
  starting from the first element of the array.
  */
  print_arr_dbl(grades, 5);
  /*
  Print all elements of array more_grades
  Array name more_grades evaluates to base address of array more_grades
  The 2nd argument evaluates to the array's size (number of elements)
  */
  print_arr_dbl(more_grades, sizeof(more_grades)/sizeof(more_grades[0]));
  // now print only the first 6 values in array more_grades
  print_arr_dbl(more_grades, 6);
  /*
  Print last 5 elements of array more_grades
  The 1st argument evaluates to address of sixth element of array at
  subscript (or offset or index) 5. Since 2nd argument is 5, the function
  will reference the last 5 elements of the 10 elements in the array
  */
  print_arr_dbl(&more_grades[5], 5);
  return 0;
}

/*
Parameter array is initialized with the base address of the first indexed variable
of an array of int variables that the function must access.
Remember, array doesn't have to be the first element of the array.
Parameter num_elements specifies the number of indexed variables to be accessed
*/
void print_arr_dbl(double arr[], int num_elements) {
  for (int i = 0; i < num_elements; ++i) {
    printf("%.2f%c", arr[i], i==num_elements-1 ? '\n' : ' ');
  }
}
```

These array parameters may seem a little strange, but this strangeness is highly prized by C programmers. The advantage of the strange declaration and definition is that `print_arr_dbl` does what its name implies - the same function can be used to print an array of any size, as long as the base type of the array is `int`. This is so because the memory address of the first indexed variable and the number of elements are separate arguments.

`const` type qualifier or parameter modifier

When a function receives the address of the first indexed variable of an array, the function can use subscript operator `[]` to change the values stored in individual array elements. This is usually fine for certain functions but not for other functions such as `print_arr_dbl` whose sole reason for existence is to *read and print* but not *modify* the contents of an array of `double`s. As seen from the definition of `print_arr_dbl`, care has been taken to not alter the array contents. However, in a complicated function definition, it is possible for the programmer to write code that inadvertently changes one or more of the

values in an array, even though the array should not be modified at all. As a precaution, function authors can tell the compiler that during function execution, the intent is to not change the values of any indexed variables. The compiler will then enforce this request by the function's author and prevent successful compilation if any inadvertent changes are made to the values of indexed variables.

To tell the compiler that an array parameter should not be changed by the function, type qualifier `const` is inserted before the array declaration. Since function `print_arr_dbl` is designed to only read values previously stored in an array's indexed variables and print them to standard output, one can ensure that any inadvertent writes to the array's indexed variables are caught by the compiler by redeclaring the function as:

```
1  void print_arr_dbl(double const array[], int num_elements);
```

The declaration of the first parameter of function `print_arr_dbl` is read from right-to-left as: `array` is a *read-only* array of `double`'s. The definition of `print_arr_dbl` doesn't change except for the function header:

```
1  void print_arr_dbl(double const array[], int num_elements) {
2    for (int i = 0; i < num_elements; ++i) {
3      printf("%.2f ", array[i], i == num_elements-1 ? '\n' : ' ');
4    }
5  }
```

Brief note on `sizeof` operator

Line 24 in an earlier code example exercises a C/C++ operator called the `sizeof` operator:

```
1  print_arr_dbl(more_grades, sizeof(more_grades)/sizeof(more_grades[0]));
```

What does the `sizeof` operator do? The `sizeof` operator computes an *unsigned value* representing the number of bytes reserved for storage of a specified variable or variable type. That it is an operator rather than a function is of little importance, except to note that you don't always have to put parentheses around the argument. The parentheses are only required for variable type arguments and is optional for expressions. The form of using `sizeof` is:

```
1  sizeof(typename)
2  sizeof variable-name
3  sizeof expression
```

where `typename` can be variable name or type, `variable-name` is the name of a previously declared variable, and `expression` is any legal C expression consisting of operands and operators. For instance, the following are legitimate uses of `sizeof` operator:

```
1  int x = 10, y = 11;
2  printf("sizeof int is: %lu bytes.\n", sizeof(int));
3  printf("sizeof int is: %lu bytes.\n", sizeof y);
4  printf("sizeof int is: %lu bytes.\n", sizeof x*y); // strange output?
```

As expected, since variables of type `int` require 4 bytes of memory storage, the first two `printf` statements print value 4 to standard output. Unexpectedly, the third and final statement prints the value 44 to standard output. Why? Since `sizeof` is an operator, precedence comes into play. The `sizeof` operator has a higher precedence than `*` operator, and therefore the expression `sizeof x*y` is evaluated as `(sizeof x)*y` which further evaluates to `4*11`.

When an array is provided as an operand to the `sizeof` operator, the result of the expression is the number of bytes occupied by the array. That is, if an array `x` is defined as follows:

```
// define array x as an array of n elements, each element of some type
T x [ n ];
```

then expression `sizeof(x)` will evaluate to the size of the array `n` times the size in bytes of an object of type `T`. Therefore, in the following code fragment, the first `printf` statement will print 400, the second `printf` statement will print 800, while the third `printf` statement will print 100 to standard output.

```
int x[100];
double y[100];
printf("size in bytes of array x: %lu\n", sizeof(x));
printf("size in bytes of array y: %lu\n", sizeof(y));
printf("number of elements in array x: %lu\n", sizeof(x)/sizeof(int));
```

The expression in the third `printf` statement provides a way for us to determine array size if it is not explicitly specified during its declaration.

Common array algorithms

In the following sections, some of the most common algorithms for processing sequences of values are presented. The algorithms are implemented as functions whose first parameter is initialized with an array's base address and the second parameter initialized with the number of elements to process. This will allow the algorithms and functions to be useful not only with entire arrays but also with partial slices of arrays.

Filling arrays with values

The following function fills the array with a value that is specified by the calling function:

```
void fill_arr_dbl(double arr[], int size, double value) {
  for (int i = 0; i < size; ++i) {
    arr[i] = value;
  }
}
```

Instead of a `for` statement, each array element can be referenced using a `while` statement:

```
void fill_arr_dbl(double arr[], int size, double value) {
  int i = 0;
  while (i < size) {
    arr[i] = value;
    ++i;
  }
}
```

The two separate statements in lines 4 and 5:

```
arr[i] = value;
++i;
```

can be combined into a single statement using post-fix increment operator, as in:

```
arr[i++] = value;
```

Expression `arr[i++]` references subscripted variable `arr[i]` so that the right operand to the `=` operator is assigned to subscripted variable `arr[i]` and then the post-fix increment operator increments subscript `i`'s value to `i+1` so that the next adjacent subscripted variable can be referenced in the next iteration. Which one is better? Remember, the objective is to write cool programs rather than flex the language's syntax. As long as you can read and understand both styles, you shouldn't care about syntax. Modern compilers can generate optimal code for both styles and therefore you should only be concerned about making your code easy to read, understand, and debug by yourself and other programmers.

Copying arrays

The intention here is to copy the elements of a source array to a destination array. Since array assignment is illegal, the elements have to be individually copied from the source to destination array. The general convention is for the calling function to ensure that the size of the destination array is at least as large as the number of elements to be copied. Otherwise, the run time behavior of the function will be catastrophic! There are many ways to specify the parameters and the return value - the following definition for copying `int` values is a simplified version of equivalent copy algorithms in the C standard library:

```
// notice that src array is declared read-only while dest is not
void copy_int(int dest[], int const src[], int size) {
  for (int i = 0; i < size; ++i) {
    dest[i] = src[i];
  }
}
```

Summing values in array

The following algorithm returns the sum of values of array elements:

```
int accumulate_int(int const arr[], int size) {
  int sum = arr[0];
  for (int i = 1; i < size; ++i) {
    sum += arr[i];
  }
  return sum;
}
```

Finding the maximum value in an array

A common requirement in many applications is the need to identify the largest value in a sequence of values. The strategy is straightforward: use a variable to keep track of the largest value encountered so far. Here is an implementation for `int` arrays:

```
int max_int(int const arr[], int size) {
  int maxval = arr[0]; // largest value seen so far ...
  for (int i = 1; i < size; ++i) {
    maxval = (arr[i] > maxval) ? arr[i] : maxval;
  }
  return maxval;
}
```

Finding the minimum value in an array

The implementation of a function to identify the smallest value in a sequence of values is straightforward: use a variable to keep track of the smallest value encountered so far:

```
int min_int(int const arr[], int size) {
  int minval = arr[0]; // smallest value seen so far ...
  for (int i = 1; i < size; ++i) {
    minval = (arr[i] < minval) ? arr[i] : minval;
  }
  return minval;
}
```

Linear search

You often need to search for the subscript at which a particular value is located in an array so that you can replace or remove it. The linear search algorithm visits all elements until you have found a match or you have come to the end of the array. If the you've reached the end of the array, then any negative value can be returned to signal to the caller that the value doesn't exist in the array.

```
int linear_search_int(int const arr[], int size, int value) {
  for (int i = 0; i < size; ++i) {
    if (value == arr[i]) {
      return i;
    }
  }
  return -1;
}
```

Filling an array with specific number of values from standard input

Arrays are often used to store information that is read from data files. For example, suppose that data file named `grades.txt` contains about 100 integer grades from a recent midterm test:

```
 46  54  68  28  75  64  65  94  59  82  70  29  37  30  75  27  11  43
 56  45  72  95  15  14  69   2  70  94  46  23 100  59  44  33  87  85
 98  51  44  56  99  13  51   2  43  25  29  55  35  52  66   6  46  81
 88  82  83  57  41  96  46   6  54  56  40  40  40   3  57  85  25  21
 98  76  23   7   0  19  62   1  37  94   8  83  41  62  30  90  18  37
 85  30  44  38  87  84  78  93  53   0
```

Also suppose the user redirects the contents of this data file to standard input `stdin`. The following function reads a specific number of these values (specified by parameter `count`) from standard input:

```
/*
Read values of type int from input stream and assign them - in sequence -
to indexed variables in array.
Fill the array with no more than count number of int values. or until
end-of-file is reached, whichever comes first.
Return number of values that have been read.
Notice that arr is NOT declared as a read-only array since we want to
assign or write int values to the array.
*/
int fill_arr_int_stdin(int arr[], int count) {
  int i = 0;
  while (i < count) {
```

```
    if (fscanf(stdin, "%d", &arr[i]) != 1) {
      break;
    }
    ++i;
  }
  return i;
}
```

The definition of function `fill_arr_int_stdin` is quite comprehensive. It anticipates the scenario where the number of values in the input stream is smaller than `const`. We can take advantage of the return value of `fscanf` to determine if a value is read from the input stream. Recall that `fscanf` returns the number of values that it successfully converted and stored. If `fscanf` doesn't successfully read a `int` value from the input stream, it will not return value `1`. The `while` condition and the return value from `fscanf` can be combined into a single `while` condition:

```
int fill_arr_int_stdin(int arr[], int count) {
  int i = 0;
  while ( (i < count) && (1 == fscanf(stdin, "%d", &arr[i])) ) {
    ++i;
  }
  return i;
}
```

In earlier tutorials, we have shown that the C standard library can be used to create an input stream from a data file to the program. In contrast to redirecting the contents of a data file to the standard input stream, programs can now directly read from any number of data files using these input file streams. Function `fill_arr_int_stdin` can be made more flexible by specifying the input stream as a function parameter, as in:

```
int fill_arr_int_stream(int arr[], int count, FILE *input_str) {
  int i = 0;
  while ( (i < count) && (1 == fscanf(input_str, "%d", &arr[i])) ) {
    ++i;
  }
  return i;
}
```

The function will be called and used in the calling environment like this:

```
#include <stdio.h>
// declaration of print_int_array function
void print_arr_int(int array[], int num_elements);
int fill_arr_int_stream(int arr[], int count, FILE *input_str);

int main(void) {
  int values[10];
  // read and store no more than 10 values from stdin
  int how_many_vals_read = fill_arr_int_stream(values, 10, stdin);
  // how_many_vals_read says how many were actually read
  print_arr_int(values, how_many_vals_read);
  return 0;
}
```

Practice 3

Assume that we've defined the following variables

```
int k = 6;
double grades[] = {1.5, 3.2, -6.1, 9.8, 8.7, 5.2};
```

Adapting the `max_int` function presented in this section for `double` arrays, give the value of each of the following expressions:

1. `max_dbl(grades, 6);`
2. `max_dbl(data, 5);`
3. `max_dbl(data, k-3);`
4. `max_dbl(data, k%5);`

Consider the following code fragment:

```
int mystery[10] = {0};
printf("Enter a number: ");
long value;
scanf("%ld", &value);
while (value > 0) {
  ++mystery[value%10];
  value/=10;
}
// now print mystery array
```

What are the contents of array `mystery` if the following numbers are provided as inputs:

1. 28212
2. 2147483647
3. 939577
4. 41271092