In this lab, we’ll look in greater detail at the declaration, definition, and use of programmer-defined functions in C.

Pairs

Here are the pairs for this lab — the same pairs (modulo absences) for both sessions. Drivers are on the left.

- Yuyin Sun and Mattori Birnbaum
- Erhaan Ahmad and Jong Hoon Bae
- Zachary Susag and Gemma Nash
- Josh Lavin and Cory McCartan
- Eli Salm and Jae Hong Shin
- Sanjay Sudhir and Anna Blindermann
- Lex Martin and Faizaan Ali
- Sophie Gaschott and Saung Thuya
- Lilly Webster and Ying Zhang
- Tyler Williams and Tanner Tufto
- Dennis Chan and Colin Greenman

Ella Nicolson will replace anyone who is absent.

Example: Iteration and recursion

To find the sum of the elements of a given vector in Scheme we might use a recursion over natural numbers to produce the elements of the vector one at a time, recovering each by its index. We could use the same strategy in C:

```c
double array_sum(double arr[], int remaining) {
    int index;
    if (remaining == 0)
        return 0.0;
    else {
        index = remaining - 1;
        return arr[index] + array_sum(arr, index);
    }
}
```

The context of a call to this function might be something like this:

```c
#define DANCE_STEPS 8
/* ... */
double durations[] = { 1.0, 0.2, 1.0, 0.2, 1.0, 0.2, 1.0, 4.0 };
(void) printf("Total duration: %.2f seconds\n",
    array_sum(durations, DANCE_STEPS));
```

Most C programmers, however, would use a loop construction rather than recursion in this case:

```c
double array_sum(double arr[], int size) {
    double total = 0.0;
    int index;
```
for (index = 0; index < size; index++)
    total += arr[index];
return total;
}

The imperative model of computation encourages the programmer to approach this
problem by setting up a storage location, total, for the result, and updating this storage
location at each step of the traversal of the array. At the beginning of each iteration of the
for-loop, and at the end, the value stored in total is the sum of all the array elements
so far encountered. Once we have encountered them all, we return the accumulated sum.

Exercises

Exercise 0000: Write and test an iterative C function that takes as arguments a
one-dimensional array of long values and the length of that array and returns the largest
value in the array. (It is a precondition of this procedure that the array has at least one
element.)

Exercise 0001: Write and test an iterative C function that takes three arguments: a
one-dimensional array of double values, the length of that array, and a “scaling factor”
of type double, and overwrites every element of the array with a “scaled” version of that
element (that is, one that has been multiplied by the scaling factor). Confirm that the side
effects of the replacement of the array elements inside the function persist in the calling
context after the function has returned.

Exercise 0010: Write and test an iterative C function that takes a positive integer num
as its argument and returns the sum of its proper divisors, that is, the sum of the positive
integers less than num that evenly divide num. For instance, if num is 18, the function
should return 1 + 2 + 3 + 6 + 9, or 21.

Exercise 0011: Using the function you defined in the previous exercise, write, compile,
and run a program that finds and prints out four numbers that are equal to the sum of their
own proper divisors. (For instance, one such number is 28, because the proper divisors of
28 are 1, 2, 4, 7, and 14, and their sum is 28.)

Exercise 0100: If you have two arrays of numbers that are equal in length, their dot
product can be computed by multiplying together the values that are in corresponding
positions in the two arrays and adding up the products. For instance, the dot product of
\{ 1, 5, 6 \} and \{ 2, 8, 3 \} is 1\cdot2+5\cdot8+6\cdot3, which works out to be 60. (If the elements
of the arrays are integers, the dot product is always an integer; if they are doubles, the
dot product is a double.) Write and test a C function that takes three arguments — two
arrays of doubles, and their common size — and returns their dot product.

Exercise 0101: Write and test an iterative C function that takes an array of int
values and the size of the array and reverses the array “in place,” as a side effect. That
is: When the function returns, the array should contain the same elements that it held
originally, but in the opposite order: The element that was originally at index 0 should
now be at index size − 1, and so on.

Exercise 0110: One (very inefficient) way to determine whether an unsigned int
value is even or odd is to write a pair of mutually recursive functions, is_even and is_odd,
each of which takes an unsigned int as its argument and returns a Boolean value. Each
begins by testing whether its argument is zero. If so, is_even returns true, and is_odd
returns false. Otherwise, each invokes the other to determine whether the predecessor
of its argument — the next smaller integer — is even or odd, then negates the result. (The
unary operator ! negates a Boolean value.)
In implementing and testing these procedures, you'll notice that, since each contains a call to the other, there's a difficulty in conforming to C's rule that the signature of each function should appear in the text of a program before any call to that function. It seems that, whichever of the mutually recursive functions you define first violates this rule by containing a call to the other. What mechanism is provided in C for averting this vicious circle?

Exercise 0111: Our textbook provides a function that tests whether a given integer is prime. Using this function, write, compile, and run a program that declares an array of one thousand integers and then fills in that array from left to right with the first thousand prime numbers (so that 2 is stored at index 0 in the array, 3 at index 1, 5 at index 2, and so on). Finally, your program should print out the values at indices 99, 199, 299, ..., 999 of the array: the hundredth, two-hundredth, three-hundredth, ..., thousandth prime numbers.

Exercise 1000: Write, compile, and run a program that creates an array with space for two thousand values of type short, then, using the function that you created in exercise 0010 above, for each i from 1 up to and including 1999, compute the sum of the proper divisors of i and place that sum in position i of the array. Then use this constructed array to find and print out a pair of integers that are amicable, meaning that each is equal to the sum of the divisors of the other. (There are two pairs of amicable numbers in the range covered by the array — you might want to try to find both of them.)

Exercise 1001: Write and test a C function that takes as arguments two arrays of double values, equal in length, and an int indicating their common size, and overwrites the contents of the second array with the cumulative totals of the elements of the first array, building up the total from left to right.

Exercise 1010: Chapter 9 of our textbook provides an implementation of the Quicksort algorithm for sorting an array of numbers. Rewrite the function so that it computes and returns the number of times an array element is moved from one storage location to another (the number of “data movements”) during the sorting process.

Test your revised function by writing a program that determines the number of data movements quicksort makes in sorting three arrays of one hundred elements each: (a) an array in which the elements are already in ascending order before the sort begins; (b) an array in which the elements start out in reverse order; (c) an array in which the first thirty-four elements are the multiples of 3 from 0 up to 99, the next thirty-three are the successors of multiples of 3, from 1 up to 97; and the remaining thirty-three are the predecessors of multiples of 3, from 2 up to 98.

Exercise 1011: The grade of an array of distinct numbers is a second array, equal in length, indicating the position that each element of the first array would occupy if the elements of that array were sorted into ascending order. For instance, the grade of the four-element array \{15, 9, 6, 12\} is \{3, 1, 0, 2\}, because if the original array were sorted, the element currently in position 0 would wind up in position 3, the element currently in position 1 would remain in position 1, and so on.

Write and test a C function that takes as arguments two arrays of int values, equal in length, and an int indicating their common size, and overwrites the second array with the grade of the first array. Your function may assume, as a precondition, that the first array contains no duplicate values.

(Hint: The value at position i in the grade can be regarded as a tally of the values in the first array that are less than the value at position i in the first array and so would wind up placed to the left of that value if the array were fully sorted. These tallies can}
be computed by comparing every pair of elements in the first array and incrementing the count at the appropriate position in the grade array, depending on the outcome of the comparison.)