Arrays*

September 20, 2001

Abstract

This document is meant to supplement the COS 126 course materials on arrays.

Contents

1 What is an array?

An array is a fundamental data structure used to organize data. It allows a programmer to combine many individual variables into a single aggregate set.

An array stores an *ordered* set of *homogeneous* values. Homogeneity means that all the values are of the same type, e.g., 180 integer exam scores, 512 real-valued color gradations, 5 million characters of a book, 34 billion nucleotide pairs of a DNA strand, 18 menu options, or 300 million Social Security numbers. Arrays would not be appropriate to store a heterogeneous set of values, e.g., a database record consisting of a name, birthday, salary, and social security number. The order of the values is also preserved, i.e., the character array `"comedian"` is different from the character array `"demoniac"`. It is useful to visualize an array as below.

c	o	m	e	d	i	a	n

1.1 Arrays in C

Using arrays, the programmer can write succinct code that manipulate huge numbers of variables. They serve to simplify and condense repeated code. For example, one way to store a word consisting of 8 characters is to declare 8 character variables as follows:

```
char a0, a1, a2, a3, a4, a5, a6, a7;
```

and manipulate them individually. A more useful alternative is to declare an array of 8 character variables:

```
char a[8];
```

The array is defined by specifying its *type* and its *size*. The preceding declares an array named `a` that stores 8 elements, each of type `char`. The memory location of the first item is saved along with the element type.

index	0	1	2	3	4	5	6	7
value	c	o	m	e	d	i	a	n

We can now essentially treat the array elements

```
a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7]
```

as individual variables. The notation `a[i]` is used to access the element with index `i`. The crucial gain over declaring individual variables is that the index `i` can be a variable. So, for example we can initialize an array of one million characters all to the value `'x'` with a single `for` loop.

```
char a[1000000];
int i;
for (i = 0; i < 1000000; i++)
   a[i] = 'x';
```

The advantage is obvious: using individual variables, you would need one million assignment statements!

The array values are stored *consecutively* in memory. This enables the compiler to quickly access the i^{th} element since to retrieve the i^{th} value, the compiler adds the offset i to the memory location of the 1^{st} element: there is no need to even examine the values of the other array elements.

Caveat 1. In C, array indices start at 0, not at 1. Thus, `a[0]` is the value of the 1^{st} element and `a[N-1]` is the value of the N^{th} element. This simplifies the compiler a bit at the expense of confusing novices.

Caveat 2. In C, it is the programmer's responsibility to use legal indices when accessing an array element. For example, in an N-element array, assigning a value to `a[N]` or to `a[-1]` is illegal. As with other variables, it is the programmers responsibility to ensure that array elements are initialized before they are used.

2 Examples

It is important that you understand not only *how* to use arrays, but also *when* to use them. This requires some critical thinking, and is best learned with experience. We provide several examples below, including: sorting, shuffling, statistics, text processing, and simulation.

2.1 Statistics

Arrays are useful to compute several common statistics of a set of data including: median, mode, histogram, and order statistics. The *mean* $\bar{X}$ of n values $X_1, \ldots, X_n$ is:

$$\bar{X} := \frac{1}{n} \sum_{i=1}^{n} X_i.$$

The *standard deviation* σ is defined by:

$$\sigma^2 := \frac{1}{n} \sum_{i=1}^{n} (X_i - \bar{X})^2 \;=\; \frac{1}{n} \sum_{i=1}^{n} X_i^2 \;-\; \bar{X}^2.$$

The *median* (assuming n is odd) is defined to be the value such that half of the elements are greater than it and half are smaller. The k^{th} *order statistic* is the k^{th} largest element. For example, the n^{th} order statistic is the maximum value and the 1^{st} order statistic is the minimum value. The *mode* is the data value(s) that occur most frequently.

2.1.1 Computing outliers.

We will write a program to read in a sequence of 160 scientific measurements. Then we will print out those measurements that are more than 3 standard deviations away from the mean. This is useful in data analysis to identify *outlier* data points that may require special consideration. To get started, we first have to read in the data and store it. We'll assume the data is real-valued, so we'll store them using type `double`. One option would be to create 160 variables of type `double` named `x0, x1, ..., x159` and then proceed to write 160 `scanf()` statements to read in the data. Obviously this is quite tedious, and the code would be difficult to maintain. Instead we will use a single array `x[]` that holds 160 `double`'s.

```
int i;
double x[160], value;
for (i = 0; i < 160; i++) {
   scanf("%lf", &value);
   x[i] = value;
}
```

Below, we define a macro `N` that stores the number of data points. We condense the body of the `for` loop to the single statement `scanf("%d", &x[i])`. Also, we add code to compute the mean and median. Finally, we print out those scores more than 3 standard deviations away from the mean. We use the functions `sqrt()` and `fabs()` from the math library to compute square roots and take absolute values.

Program 2.1 A program that reads in a sequence of N integers, prints their mean, standard deviation, and for each value the number of standard deviations it is above or below the mean.

```
% gcc outliers.c -lm
% a.out
fill in data
```

```
#include <stdio.h>
#include <math.h>
#define N 10

int main(void) {
   int i;
   double x[N], sum = 0.0, sumofsquares = 0.0, mean, stddev;

   for (i = 0; i < N; i++) {
      scanf("%lf", &x[i]);
      sum += x[i];
      sumofsquares += x[i] * x[i];
   }

   mean   = sum / N;
   stddev = sqrt(sumofsquares / N - mean * mean);

   printf("mean = %f\n", mean);
   printf("standard deviation = %f\n", stddev);
   for (i = 0; i < N; i++)
      if (fabs(x[i] - mean) / stddev >= 3.0)
         printf("%d %f %f\n", i, x[i], (x[i] - mean) / stddev);
   return 0;
}
```

It is important to note that arrays are not needed to compute the mean and standard deviation. (See Exercise 3.) They are needed to store away the data, so that once the mean and standard deviation are computed, we can re-examine the data and filter out the outliers.

2.1.2 Histogram

A *histogram* is a convenient way to summarize data graphically. The data are divided up into groups or bins, and each bin is plotted with a bar proportional to the number of values in the bin. For example, consider an exam where the scores range from 0 through 99. Instead of printing out each exam score, we might wish to know how many scores fall in each of the following ranges: $0-9, 10-19, ..., 90-99$. To accomplish this, we create an integer array `h[10]`, where `h[i]` counts the number of exam scores from $10i$ through $10i+9$.

Our program first initializes all 10 bins to 0. Then, it reads in the exam scores one at a time, incrementing the appropriate counter variable by one. This is repeated until there is no more data. Finally, we print out a number of asterisks corresponding to the number of exam scores in each bin.

Program 2.2 A program that reads in integers between 0 and 99 and plots a histogram.

```
#include <stdio.h>
 int main(void) {
    int i, j, val, h[10], bin;

    for (i = 0; i < 10; i++)
       h[i] = 0;

    while (scanf("%d", &val) != EOF) {
       bin = val / 10;   /* integer division */
       h[bin]++;
    }

    for (i = 0; i < 10; i++) {
       printf("%2d-%2d:  ", 10*i, 10*i + 9);
          for (j = 0; j < h[i]; j++)
             printf("*");
          printf("\n");
    }
    return 0;
 }
```

```
% gcc histogram.c
% a.out
80 85 80 90 76 78 92 75 83
70 83 65 96 55 66 83 83 22

 0- 9:
10-19:
20-29:  *
30-39:
40-49:
50-59:  *
60-69:  **
70-79:  ****
80-89:  *******
90-99:  ***
```

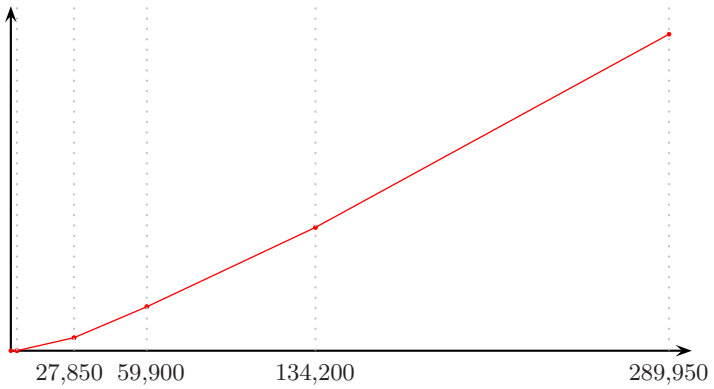
2.2 Tabulating values

Arrays are quite useful for storing information in a table for later reuse. This can greatly improve the performance and maintainability of certain programs.

2.2.1 Graduated income tax

We compute the amount of federal income tax an employee owes, given their taxable wages. Figure 2.2.1 gives the formula for computing federal income tax for an unmarried employee for the calendar year 2000.

Over	Not over	Marginal rate
$0	$2,650	00.0%
$2,650	$27,850	15.0%
$27,850	$59,900	28.0%
$59,900	$134,200	31.0%
$134,200	$289,950	36.0%
$289,950	∞	39.6%



For example, if your taxable income is $100,000 then the amount of tax you pay is:

$$0.15(27,850 - 2,650) + 0.28(59,900 - 27,850) + 0.31(100,000 - 59,900) = 25,185.$$

One ugly approach to solving this problem is using lots of `if-else` statements, as in the following atrocious code fragment:

```
if (income <= 2650)
   tax = 0.0;
else if (income <= 27850)
   tax = 0.15*(income - 2650);
else if (income <= 59900)
   tax = 0.15*(27850 - 2650) + 0.28*(income - 27850);
else if (income <= 134200)
   tax = 0.15*(27850 - 2650) + 0.28*(59900 - 27850) + 0.31*(income - 59900);
. . .
```

Although this does compute the tax correctly, it will be very difficult to maintain, and the programmer who wrote this should be fired! What happens next year when the marginal tax rates or tax brackets change? Or if we want to reuse this code to compute the tax for a married employee. There are *magic numbers* all over the place.

Now, we consider a better approach using arrays. We maintain two arrays: `bracket[]` contains the tax bracket upper limit, and `rate[]` contains the marginal tax rate for that tax bracket. To avoid having a special case for the first tax bracket, we add a dummy 0th tax bracket. The constants `DBL_MIN` and `DBL_MAX` are the smallest and largest values representable by type `double`. Using the data above,

Program 2.3 A function that computes your 2000 federal income tax, given your taxable income.

```
#include <float.h>

double calculateTax(double income) {
  int i = 1;
  double tax = 0.0;
  double rate[]    = {0.00,       0.00,     0.15,     0.28,      0.31,      0.36,     0.396};
  double bracket[] = {DBL_MIN, 2650.00, 27850.00, 59900.00, 134200.00, 289950.00,   DBL_MAX};
  while (income > bracket[i]) {
     tax += (bracket[i] - bracket[i-1]) * rate[i];
     i++;
  }
  tax += rate[i] * (income - bracket[i-1]);
  return tax;
}
```

2.2.2 Prime numbers

In this section we describe two methods for computing all prime numbers less than a given threshold value N. Prime numbers play a crucial role in many scientific applications, including cryptography. Our first method uses brute force. The second method uses arrays to implement a classic method known as the *sieve of Eratosthenes*. We discuss the space and time tradeoffs between the two approaches.

A brute-force approach. We first describe a brute force method for testing whether an integer `j` is prime. Divide each integer between 2 and $\sqrt{j}$ into `j`. The integer `j` is declared prime if none of these integers divides evenly into it. This method works because a positive integer is composite if and only if it is a nontrivial multiple[1] of some integer less than equal to $\sqrt{j}$. Figure 2.4 repeats this procedure for each integer `j`, and prints those integers that are prime.

[1] A nontrivial multiple of an integer j is a positive integral multiple of j other than j itself.

Program 2.4 A brute force program that prints all prime numbers less than `N`.

```
#include <stdio.h>
#define N 50

int main(void) {
   int i, j, isprime = 0;

   for (j = 2; j < N; j++) {
      isprime = 1;
      for (i = 2; i * i <= j; i++)
         if (j % i == 0) {       /* i is a factor of j */
            isprime = 0;
            break;
         }
      if (isprime == 1)
         printf("%d\n", j);
   }
   return 0;
}
```

```
% gcc prime-brute.c
% a.out
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

Sieve of Eratosthenes. The method we now describe is known as the *sieve of Eratosthenes*, after the Greek mathematician who invented it (ca 240 BC).[2] It is among the most efficient ways to find all small primes. The idea is to start with the integers 2 through N. Mark the smallest integer, say p, as prime. Then, cross out all nontrivial multiples of p. Repeat the process of choosing the smallest remaining integer and crossing out all of its nontrivial multiples. This filters out the composite numbers, and all those numbers remaining are prime. Each successive *sieving* step catches another prime.

Our implementation of this classic algorithm is shown in Figure 2.5. A sample run is illustrated in Figure 2.1. It uses an `N`-element array `a[]`. If the variable `a[j]` is 1, then `j` is prime; if it is 0, then `j` is not prime. As the algorithm progresses, it marks integers as prime or non-prime. Those variables for which the algorithm has not yet determined the primality status are assigned the value `2`. Once a variable is marked as prime or non-prime, it never changes its value. Upon termination, all variables are either marked as prime or not prime.

Initially, the algorithm sets all variables `a[j]` to be 2, since initially the primality of all integers is unknown. The crux of the algorithm consists of two nested `for` loops. The outer loop ranges over `i`, from `2` to `N-1`. If the status of integer `i` is unknown at the beginning of iteration `i`, then `i` is marked as prime. Next, the innermost loop marks all nontrivial multiples of `i` as non-prime. One crucial bit of cleverness is employed: if the integer `i` has previously been marked as non-prime, then we can skip the iteration completely. Why? If `i` is not prime, then all of its multiples were already marked as non-prime in the outer loop corresponding to one of `i`'s factor. The reason for this is simple: any multiple of `i` is also a multiple of each factor of `i`. This observation is formalized below, and establishes the correctness of the algorithm.

Property 1 *At the end of iteration i, every prime integer less than or equal to i is marked as prime, and every integer that is a nontrivial multiple of an integer less than or equal to i is marked as non-prime.*

[2] See also Sedgewick, Program 3.5

Program 2.5 A program that prints all prime numbers less than N using the sieve of Eratosthenes.

```
#include <stdio.h>
#define N 32
#define NONPRIME  0
#define PRIME     1
#define UNKNOWN   2

int main(void) {
  int i, j, a[N];

  for (j = 2; j < N; j++)
     a[j] = UNKNOWN;

  for (i = 2; i < N; i++)
     if (a[i] == UNKNOWN) {
        a[i] = PRIME;
        printf("%d\n", i);
        for (j = 2; i * j < N; j ++)
           a[i*j] = NONPRIME;
     }

  return 0;
}
```

```
% gcc prime-sieve.c
% a.out
2
3
5
7
11
13
17
19
23
29
31
```

We make several remarks on our implementation. First, we never use array indices 0 or 1. We choose to waste two indices for the convenience of having array index `j` correspond to the integer `j`, rather than to `j+2`. Second, it is critical that we can access any item of the array, given its index. The two loops ranging over `i` access the array elements *sequentially*, i.e., in order from smallest index to largest index. However, in the innermost loop the algorithm needs *random access* to the array elements, i.e., it does not access the array elements in order. Third, we can make a further improvement in efficiency by starting the inner `j` loop from `i` instead of from 2. Why can we do this? If `j < i` then `i*j` would have already been declared to be non-prime by the end of iteration `j`, so there is no need to recheck it. Finally, we note that although we allow the array items to take on three values, only two are needed. This was to assist in the exposition of the algorithm. It is not hard to verify that the algorithm still works if we replace all occurrences of `UNKNOWN` with `PRIME`. This would allow us to represent each array item with single bit of storage. For simplicity, we have made the rather lavish choice of using an `int` to store each bit.

Performance tradeoffs. We now compare the performance characteristics of the two methods described above in terms of speed and memory consumption. The brute force method required 9 minutes and 25 seconds to compute all primes less than $N = 10$ million, while the sieve computed them in less than 10 seconds.[3] This is a dramatic difference, and the gap only gets larger as we increase N. However, the boost in performance is not without its price. To carry out the computation, the sieve method needs to use an array of size N, while the brute-force method uses only three integer variables. If the array elements in the sieve method are stored as integers (4 bytes per `int` on the Sun-Ultra10), then this array consumes 40MB of memory. These types of time-space tradeoffs are typical when designing computer software.

2.3 Strings

A *string*[4] is "a variable-length array of characters, defined by a starting point and by a string-termination character marking the end." Strings are widely used in text processing applications, e.g., browsers use them

[3] These times were calculated on a Sun-Ultra10 with 256MB of memory.
[4] See Sedgewick, Section 3.6.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	init
1	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	i = 2
1	1	0	?	0	?	0	?	0	?	0	?	0	?	0	?	0	?	0	?	0	?	0	?	0	?	0	?	0	?	i = 3
1	1	0	?	0	?	0	0	0	?	0	?	0	0	0	?	0	?	0	0	0	?	0	?	0	0	0	?	0	?	i = 4
1	1	0	1	0	?	0	0	0	?	0	?	0	0	0	?	0	?	0	0	0	?	0	?	0	0	0	?	0	?	i = 5
1	1	0	1	0	?	0	0	0	?	0	?	0	0	0	?	0	?	0	0	0	?	0	0	0	0	0	?	0	?	i = 6
1	1	0	1	0	1	0	0	0	?	0	?	0	0	0	?	0	?	0	0	0	?	0	0	0	0	0	?	0	?	i = 7
																														. . .
1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	0	1	i = 31

FigureFloat 2.1: The sieve of Eratosthenes in action with `N = 32`. Array items marked with `0`, `1`, and `?` are non-prime, prime, and unknown, respectively. Initially, all integers are marked as unknown. In iteration `i`, if array item `i` is unknown, then (i) `i` is marked as prime and is shaded, and (ii) all nontrivial multiples of `i` are marked as non-prime and are shown with cross-hatches.

to process Web pages, and compilers use them to process C programs. One way to implement strings is using arrays. Note that arrays have fixed length, whereas strings have variable length. To create the illusion of a variable length array, we declare an array of suitably large size so that it can handle any reasonable input. However, we only use as many elements of the array as necessary. For example, we might allocate 30 characters for a string that stores someone's last name, but, for most last names, we only use the first 10. In C, the string-termination character is `'\0'`, and it delimits the end of the string. The following code demonstrates one way to store the word "hello" as a string using an array.

Program 2.6 A toy program that stores the word "hello" as a string and modifies it.

```
#include <stdio.h>
#include MAX_SIZE 10

int main(void) {
   char s[MAX_SIZE + 1];
   s[0] = 'h';  s[1] = 'e';  s[2] = 'l';  s[3] = 'l';  s[4] = 'o';  s[5] = '\0';
   printf("%s\n", s);
   s[3] = 'i';  s[4] = 'u';  s[5] = 'm';  s[6] = '\0';
   printf("%s\n", s);
   return 0;
}
```

There are several things you should observe. First, we use the type `char` to indicate that `s` will be an array of characters. There are 256 possible characters on most systems, and the character `'h'` denotes the letter h. The special character `'\0'` denotes the string-termination character. Second, note that the size of the array is bigger than the number of characters we need to store. This allows us the flexibility of later modifying the string, e.g., to "helium" as above. Third, note that if we want the string to be able to accommodate up to 10 characters, we need to declare an array of size 11. This is to allow room for the string termination character. Finally, note that the `printf("%s", s)` means to print all the characters in the array `s` until you hit the string termination character. Thus, the above program will only print the 5 characters "hello" and then the 6 characters "helium". It will not try to print anything after the string termination character.

Shortcut. To initialize a string you can use the shortcut

```
char s[MAX_SIZE+1] = "hello";
```

This is essentially identical to the declaration and 6 assignment statements in Program 2.6. Note that this shortcut is only legal within a declaration statement. In other circumstances you can use the `strcpy()`

library function. For example, `strcpy(s, "helium");` puts the word "helium" into `s` and also appends the string termination character to the end.

Let's take a closer look at how we might write a program that manipulates strings. We'll write a program that reads in words, and prints out the number of (lower-case) vowels in each word.

Program 2.7 A program that reads in words, and prints the number of vowels in each word.

```
#include <stdio.h>
#define N 100

int main(void) {
   char s[N + 1];
   int i, cnt;

   while (scanf("%s", s) != EOF) {
      for (i = 0, cnt = 0; s[i] != '\0'; i++)
         if (s[i] == 'a' || s[i] == 'e' ||
             s[i] == 'i' || s[i] == 'o' ||
             s[i] == 'u')
            cnt++;
         printf("%s has %d vowels\n", s, cnt);
   }
   return 0;
}
```

```
% gcc vowels.c
% a.out
hello there facetious aardvark

hello has 2 vowels
there has 2 vowels
facetious has 5 vowels
aardvark has 3 vowels
```

The `for` loop looks very similar to the code for traversing the elements of an array, except that the test condition is `s[i] != '\0'` instead of `i < N`. This is because we don't want to process all of the characters of the array, only those characters that precede the (first) string termination character.

2.4 Sorting

One of the most fundamental problems in computing is to sort a sequence of N items in nondecreasing order. We will consider two simple approaches here.[5] For simplicity, we'll assume the items are integers.

2.4.1 Insertion sort

Insertion sort[6] is a simple sorting solution that is a popular approach when sorting bridge hands. In insertion sort, we read in the values one at a time, and insert each value into its proper place among those already considered. We use an array `x[N]` to store the sorted values. At the beginning of the i^{th} iteration, the values `x[0]` through `x[i-1]` are in sorted order. We put the new element in position i, and then repeatedly swap `a[i]` with its neighbor to the left until it gets sifted down into the correct position. Program 2.8 illustrates the code.

[5] See Sedgewick, Algorithms in C, Chapters 6-11 for a more comprehensive treatment of this subject.
[6] See Sedgewick, Section 6.3.

Program 2.8 A program that sorts `N` real numbers using insertion sort.

```
#include <stdio.h>
#define N 10

int main(void) {
   int i, j;
   double swap, x[N];

   for (i = 0; i < N; i++) {
      scanf("%lf", &x[i]);
      for (j = i; j > 0; j--)
         if (x[j-1] > x[j])
            swap = x[j], x[j] = x[j-1], x[j-1] = swap;
   }

   for (i = 0; i < N; i++)
      printf("%f\n", x[i]);
   return 0;
}
```

```
% gcc insertion_sort.c
% a.out
80 2185 80 3290 76
70 83 65 55 66

55
65
66
70
76
80
80
83
2185
3290
```

2.4.2 Distribution sort

We consider a second sorting solution called *distribution sort*. It is very effective when all of the items to be sorted lie in a small range, say integers between 0 and `M-1`. Two examples are: exam scores between 0 and 99 or 5 digit zip codes. The idea of this method is that we create an `M`-element array `a[]` and let the variable `a[i]` count the number of occurrences of input value i. We initialize `a[i]` to 0, and each time we read in the integer i, we increment the counter `a[i]` by one. After we've read in all of the data, it is easy to print out all of the input values in sorted order.

Distribution sort has three advantages over insertion sort when M is relatively small compared with N. First, we only need to allocate an array of size M; in insertion sort we needed to allocate an array of size N. Second, in counting sort, we don't need to know the number of items to be sorted ahead of time since the program doesn't use the value N anywhere. Third, and most importantly, counting sort runs significantly faster than insertion. The number of steps required by insertion sort to sort N elements can be proportional to N^2, whereas for counting sort it is proportional to $M + N$. The major drawback of counting sort is that the input values must lie in a small range: this is not the case in many applications, e.g, real numbers, text strings, or 9 digit Social Security numbers.

Program 2.9 A program that sorts an arbitrary number of exam scores between 0 and 99 using distribution sort.

```
#include <stdio.h>
#define VALUES 100

int main(void) {
   int i, j, b[VALUES], score;

   for (i = 0; i < VALUES; i++)
      b[i] = 0;

   while (scanf("%d", &score) != EOF)
      b[score]++;

   for (i = 0; i < VALUES; i++)
      for (j = 0; j < b[i]; j++)
         printf("%d\n", i);
   return 0;
}
```

```
% gcc distribution_sort.c
% a.out
80 85 80 90 76
70 83 65 55 66

55
65
66
70
76
80
80
83
85
90
```

3 Passing arrays to functions in C.

It is convenient to break up a large program into smaller functions. Just like with other variables, we may pass an array to a function for processing. Below is a function `sort()` that sorts the items in an array using insertion sort. Note that we must pass the number of items to be sorted to the function, since the function has no way to determine (or check) the size of the array on its own.

Program 3.1 A program that reads in `N` real numbers, calls a function to sort them, and then prints them out.

```
#include <stdio.h>
#define N 10

void sort(double x[], int n) {
   int i, j;
   double swap;
   for (i = 0; i < n; i++) {
      for (j = i; j > 0; j--)
         if (x[j-1] > x[j])
            swap = x[j-1], x[j-1] = x[j], x[j] = swap;
   }
}

int main(void) {
   int i;
   double x[N];
   for (i = 0; i < N; i++)    /* read input    */
      scanf("%lf", &x[i]);
   sort(x, N);                /* sort          */
   for (i = 0; i < N; i++)    /* print results */
      printf("%f\n", x[i]);
   return 0;
}
```

```
% gcc insertion_sort2.c
% a.out
45.43 67.65 23.23 67.64
18.93 55.52 15.56 78.66
45.41 29.00

15.560000
18.930000
23.230000
29.000000
45.410000
45.430000
55.520000
67.640000
67.650000
78.660000
```

3.1 Call by value

In C, the mechanism for passing arrays to functions is very different from how primitive data types are passed. Recall that primitive data types are *passed by value*. This means that a *copy* of the value is passed to the function. Any changes made to that copy do not propagate back to the original variable. Arrays are also passed by value; however it is a copy of the memory address (pointer) of the first array element that is passed rather than a copy of the array elements. The reason for this is simple: it would take excessive time and memory to copy all of the elements of an array each time we called a function, so instead we just pass the value of the memory location containing the first element. Since array values are stored consecutively in memory, this (along with the array element type) is all that is needed to access any element of the array. The consequence of this seemingly subtle distinction is that if you change an array element in a function, the *original* array value also changes. This explains why the array values are printed in sorted order after the function call in Program 3.1.

3.2 Shuffling

Given an array of N items, we wish to rearrange the items so that each of the $N!$ possible permutations is equally likely. We'll assume that we have a pseudo-random number generator `randomInteger(k)` that generates a random integer uniformly between 0 and k.

This procedure has many practical uses. For example, if the array consists of 52 playing cards, this procedure is analogous to shuffling a deck of cards. When testing a new drug, scientists typically use randomized clinical trials, where a number of different drugs are given to different individuals at random.

We present an algorithm that shuffles the elements by exchanging n "random" elements. This must be done carefully so that all $n!$ permutations of the original array items are equally likely. [7]

Our shuffling algorithm consists of n iterations named 0 through $n-1$. In iteration i, we generate a pseudo-random integer r in the range 0 through i, and then swap items `a[r]` and `a[i]`. Program 3.2 provides an implementation of this procedure.

We still need to claim that the above algorithm shuffles the elements uniformly. The following key property establishes this fact. It can be proved using induction.

Property 2 *At the end of iteration i, the first i array elements contains a uniform shuffling of the first i array elements.*

Program 3.2 A function that efficiently computes a pseudo-random permutation of the integers 0 to $n-1$.

```
void shuffle(int a[], int n) {
   int i, r, swap;
   for (i = 0; i < n; i++) {
      r = randomInteger(i+1);
      swap = a[r]; a[r] = a[i]; a[i] = swap;
   }
}
```

a[0]	a[1]	a[2]	a[3]	a[4]	r	
?	?	?	?	?		start
0	?	?	?	?	0	Phase 0
0	1	?	?	?	1	Phase 1
0	2	1	?	?	1	Phase 2
3	2	1	0	?	0	Phase 3
3	2	1	4	0	3	Phase 4

3.2.1 An application

Suppose N kids go to a party and throw there bicycles into a big heap. Each child is blindfolded and grabs a random bike from the pile. What is the likelihood that at least one child gets their own bike back?

We can estimate the value with a computational experiment. In general this process is known as *Monte Carlo simulation* and has numerous scientific applications. First, it is convenient to label the children and

[7] For example, the naïve strategy of choosing n random indices uniformly at random (and exchanging the corresponding items) makes some permutations more likely than others.

their bikes with the integers 0 through $N-1$. Then, we randomly shuffle the bikes with the algorithm above. Child i gets their own bike back if bike i ends up in position i. The function `bike()` perform the numerical experiment once. It returns 1 if some kid gets their own bike back, and 0 otherwise. If we repeat this experiment many times and take the average, we obtain a good estimate of the true probability. Simulation reveals that the likelihood of success does not approach 0 or 1 as N goes to infinity; rather the answer is close to 37%. A rigorous mathematical analysis confirms that as N goes to infinity, the success probability converges to $1/e \approx 0.368$, a rather surprising answer!

Program 3.3 The function `randperm()` creates a random permutation of the integer 0 through $n-1$. The function `bike()` generates a random permutation, and returns 1 if at least one element i ends up in position i.

```
void randperm(int a[], int n) {
   int i;
   for (i = 0; i < n; i++)
      a[i] = i;
   shuffle(a, n);
}

int bike(int a[], int n) {
   int i;
   randperm(a, n);
   for (i = 0; i < n; i++)
      if (a[i] == i)
         return 1;
   return 0;
}
```

4 Variable length arrays

In the previous sections, we have assumed that we know the size of an array before we compile the program. In this section, we consider several techniques for handling variable-size arrays.

4.1 Arrays of unknown

Suppose we want to sort a sequence of items, but we don't know how many ahead of time. We would like to store the items in an array, but in C we must specify its length ahead of time.

To create the illusion of a variable length array, we declare an array of suitably large size so that it can handle any reasonable input. However, we only use as many elements of the array as necessary. For example, we might allocate a 30-character array to store a person's last name if we know that nobody will have a longer last name. Program 4.1 illustrates how we can implement this approach. It is capable of sorting as many as 100,000 elements, but it still works properly on smaller inputs.

Program 4.1 A program that reads in up to `MAXN` real numbers, and stores them in an array.

```
#include <stdio.h>
#define MAXN 200
int main(void) {
   int n = 0;
   double x[MAXN];
   while (scanf("%lf", &x[n]) != EOF)
      n++;
   return 0;
}
```

Sentinels. We used a variant of this approach in Section 2.3 to implement strings. As above, we allocated an array of sufficient size to handle any reasonable input. However, instead of explicitly keeping track of the number of items `n`, we store a special *sentinel* item after the last item. A sentinel item is one that will not be confused with an ordinary array item. For example, if the array is intended to store only positive integers, then we could use 0 as the sentinel. For strings, we used the `'\0'` as the sentinel.

4.2 Variable length arrays in C99

Now suppose that the number of items we want to store in the array is known before the items are read in, but not until after we compile our program. Thus, our program should allocate the array only after it has determined the appropriate size. The ANSI C99 standard supports such variable-sized arrays; however, this is illegal under ANSI C89, so it is quite possible that your compiler does not offer this extension yet. Program 4.2 shows how to use this method.

Program 4.2 A program that reads in an integer `n`, allocates a real-valued array of size `n`, and reads in `n` real values.

```
#include <stdio.h>
int main(void) {
   int n;
   scanf("%d", &n);
   double a[n];                  // legal in C99
   for (i = 0; i < n; i++)
      scanf("%lf", &a[i]);
   return 0;
}
```

The advantage of this technique over the previous two is that we allocate exactly the right amount of space to store the array items.

4.3 Variable length arrays with `malloc()`

If your compiler does not yet support variable-length arrays, you can use the standard library function `malloc()` to allocate the appropriate amount of space.

Program 4.3 A program that reads in an integer `n`, uses `malloc()` to allocate a real-valued array of size `n`, and reads in `n` real values.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int n;
   double *a;
   scanf("%d", &n);

   a = malloc(n * sizeof(*a));
   if (a == NULL) {
      printf("Ran out of memory.\n");
      exit(EXIT_FAILURE);
   }

   for (i = 0; i < n; i++)
      scanf("%lf", &a[i]);
   return 0;
}
```

5 Exercises

1. Fix the following broken C code fragment to do what the programmer intended.

```
int a[N] = {58, 89, 89, 72, 76};
int b[N] = {87, 80, 91, 89, 83};
int c[N];
c = a + b;
```

2. What does the following program do?

```
#include <stdio.h>
#define N 100
int main(void) {
   double a[N], sum = 0.0, avg = 0.0;
   int i, big = 0;
   for (i = 0; i < N; i++) {
      scanf("%lf", &a[i]);
      sum += a[i];
   }
   avg = sum / N;

   for (i = 0; i < N; i++)
      if (a[i] > avg)
         ++big;
   printf("%d\n", big);
   return 0;
}
```

3. Suppose we only want to compute the mean and standard deviation (and not filter out the outliers). Modify Program 2.1 to do this *without* using arrays.

4. Write a program that reads in a real-valued N-dimensional vector, and prints the *normalized vector*. That is read in N values X_0 through X_{N-1} and print out Y_0 through Y_{N-1} where:

$$Y_i := \frac{X_i}{\sqrt{\sum_{i=0}^{N-1} X_i^2}}$$

.

5. Write a program to read in a sequence of 1,000 real numbers and print a pair that is:

   (a) *closest* in value.

   (b) *farthest* in value.

   Avoid using arrays if possible.

6. Write a C program to read in a sequence of N integers between 0 and 99 and print a *stem-and-leaf plot*. Hint: modify Program 2.2, and use 100 bins.

```
% gcc stemleaf.c
% a.out
80 85 80 90 76 77 72 91 56
70 83 65 55 72 72 66 80 85
 0- 9:
10-19:
20-29:
30-39:
40-49:
50-59:  5
60-69:  566
70-79:  022276
80-89:  000355
90-99:  01
```

Output 5.1: A stem-and-leaf plot.

7. Write a program to read in an arbitrarily long sequence of real numbers, and print the smallest 10 values. Hint: use the basic idea from insertion sort, but only use an array of size 10.

8. Write a program that reads in a list of words, and prints out only those that have no vowels, e.g., *rhythm* but not *biorhythm*.

9. Write a program that reads in an integer n and prints its binary representation. Use the following procedure:

   - $i = 0$
   - repeat until $n = 0$
     - if n is even, set `b[i] = 0`; else set `b[i] = 1`
     - divide n by 2 using integer division
     - increment i by 1
   - print the array in reverse order

10. Write a program that reads in a length N array and prints out only each item in the array exactly once, even if an item occurs multiple times. This is useful to remove duplicate entries from a mailing list. For simplicity, assume the entries are characters, e.g., if the input is *abstractionist*, then the output is *abcinorst*. The order of the output is unimportant, so long as each letter appears exactly once.

    (a) Create an array that counts the number of occurrences of each letter in the word. Print out each character once if it has a count greater than or equal to 1.

    (b) Sort the items in the array, e.g., sorting the word *abstractionist* would yield *aabciinorssttt*. Now, all duplicate letters will appear consecutively. Scan through the array, printing each unique letter exactly once.

11. Redo the graduated income tax to compute the federal income tax in 2000 for a married employee using the following tax rates. Compare the amount of code you need to change in the array based solution and the `if-else` based solution.

Over	Not over	Marginal rate
\$0	\$6,450	00.0%
\$6,450	\$48,400	15.0%
\$48,400	\$101,000	28.0%
\$101,000	\$166,000	31.0%
\$166,000	\$292,900	36.0%
\$292,900	∞	39.6%

12. Write a function `int isSorted(int a[], int n)` that returns 1 if the n-element array `a[]` is in increasing order, and 0 otherwise.

13. A drug company wishes to test the efficacy of 4 drugs on 1,000 individuals. Their experiment protocol requires that each drug is assigned to exactly 250 individuals, and the assignment should be random. Write a function to do this.

14. **Powerball Lottery.** Write a program that chooses M integers between 0 and $N - 1$ at random, and prints them to standard output. Typical values are $M = 6$ and $N = 80$. The basic strategy is to repeatedly generate random integers in the range 0 through $N - 1$ until you end up with M distinct integers. The challenge is to avoid duplicates. Hint: random permutation.

15. Modify Program 4.1 so that it gives the user a warning if they exceed the pre-specified number of array elements.

16. Modify insertion sort so that it sorts any number of input values up to `MAXN` instead of `N`.

6 Solutions

1. Arrays are treated as second-class citizens in C: you can't add them directly. Instead, write a loop.

```
int a[N] = {58, 89, 89, 72, 76};
int b[N] = {87, 80, 91, 89, 83};
int c[N], i;
for (i = 0; i < N; i++)
   c[i] = a[i] + b[i];
```

2. It reads in 100 real numbers, computes their mean, and prints the number of elements greater than the average.

3. This is a valuable programming lesson: don't waste space in storing values in an array unless you really need to.

```
#include <stdio.h>
 #include <math.h>
 #define N 10

 int main(void) {
    int i;
    double val, sum = 0.0, sumofsquares = 0.0, mean, stddev;

    for (i = 0; i < N; i++) {
       scanf("%lf", &val);
       sum += val;
       sumofsquares += val * val;
    }
    mean   = sum / N;
    stddev = sqrt(sumofsquares / N - mean * mean);
    printf("mean = %f\n", mean);
    printf("standard deviation = %f\n", stddev);
    return 0;
 }
```

4.
```
 #include <stdio.h>
 #define N 50
 int main(void) {
    int i;
    double x[N], y[N], ss;
    for (i = 0, ss = 0.0; i < N; i++) {
       scanf("%lf", &x[i]);
       ss += x[i] * x[i];
    }
    for (i = 0; i < N; i++)
       printf("%f\n", x[i] / sqrt(ss));
    return 0;
 }
```

Confusingly, `%lf` is the correct `scanf()` code for reading in doubles, whereas `%f` is used with `printf()`. The acute reader may observe that it would be more efficient to compute the square root once at the beginning of the loop, store it in a variable, and then use this stored value inside the loop instead of recomputing from scratch each time. This is true, although a good optimizing compiler will do this automatically.

5. This purpose of this question is also to make you think about when you need arrays and when you don't.

   (a) The simplest way is to store the 1,000 values in an array and compute the distance between each pair.[8]

```
#include <stdio.h>
#include <math.h>
#define N 1000

int main(void) {
   int i, j;
   double x[N], dist, bestdist;

   for (i = 0; i < N; i++)
      scanf("%lf", &x[i]);

   bestdist = fabs(x[0] - x[1]);
   for (i = 0; i < N; i++)
      for (j = i + 1; j < N; j++) {
         dist = fabs(x[i] - x[j]);
         if (dist < bestdist)
            bestdist = dist;
      }
   printf("closest distance = %f\n", bestdist);
   return 0;
}
```

   (b) For this problem, using an array is unnecessary and wastes space. Instead, compute the minimum and maximum values: this defines a pair that is furthest apart.

```
#include <stdio.h>

int main(void) {
   double val, min, max;
   int n = 0;
   while (scanf("%lf", &val) != EOF) {
      if (n == 0)
         min = max = val;
      else if (val < min)
         min = val;
      else if (val > max)
         max = val;
      n++;
   }
   printf("furthest distance = %f\n", max - min);
   return 0;
}
```

6. Stem-and-leaf plot.

7. The array `x[]` stores the `K` smallest values in nondecreasing order. For convenience, we store the value just read in element `x[K]`, i.e., the `K+1`st. Then we swap it down into place, as in insertion sort.

[8] A faster alternative is to first sort the values. Now the closest pair occurs consecutively. See the Complexity exercises for more details.

```
#include <stdio.h>
#include <limits.h>
#define K 10

int main(void) {
   int i, swap, x[K+1], val;

   for (i = 0; i < K; i++)
      x[i] = INT_MAX;            /* value of largest int */

   while(scanf("%d", &val) != EOF) {
      x[K] = val;
      for (i = K; i >= 1; i--)
         if (x[i-1] > x[i])
            swap = x[i], x[i] = x[i-1], x[i-1] = swap;
   }

   for (i = 0; i < K; i++)
      printf("%d\n", x[i]);
   return 0;
}
```

8. The following modification to Figure 2.7 will print out only those words with no vowels. It breaks out of the for loop as soon as the first vowel is encountered. Thus, the word has no vowels if and only if `(s[i] == '\0')` upon termination of the for loop.

```
#include <stdio.h>
#define N 100

int main(void) {
   char s[N + 1];
   int i, cnt;

   while (scanf("%s", s) != EOF) {
      for (i = 0, cnt = 0; s[i] != '\0'; i++)
         if(s[i] == 'a' || s[i] == 'e' || s[i] == 'i' ||
            s[i] == 'o' || s[i] == 'u')
           break;
      if (s[i] == '\0')
         printf("%s\n", s);
   }
   return 0;
}
```

9.
```
#include <stdio.h>
#define MAXDIGITS 100

int main(void) {
   int digits[MAXDIGITS];
   int i, n, x;
   scanf("%d", &x);

   for (n = 0; n < MAXDIGITS && x > 0; n++) {
      digits[n] = x % 2;
      x /= 2;
```

```
   }

   for (i = n - 1; i >= 0; i--)
      printf("%d", digits[i]);
   printf("\n");
   return 0;
}
```

10. Put solutions for duplicated letter exercise here.

    (a) frequency count

    (b) sort Note that this solution is more general than the one in part (a) because it works even if the elements are not characters or small integers.

11. Put answer here.

12.
```
int isSorted(int a[], int n) {
   int i;
   for (i = 1; i < n; i++)
      if (a[i] < a[i-1])
         return 0;
   return 1;
}
```

13. For convenience, we label the drugs 0 through 3. We fill up an array with $n/4$ copies of each letter, rounded appropriately if n is not a multiple of 4. Finally, we call the shuffling procedure.

```
#define DRUGS 4
   int randomDrug(int a[], int n) {
   int i;
   for (i = 0; i < n; i ++)
      a[i] = i % DRUGS;
   shuffle(a, n);
}
```

14. Powerball. Caveat: never use the library function `rand()` when writing gambling application. The pseudo-random numbers generated are poor – someone will figure this out and you will go broke!

15.
```
#include <stdio.h>
#define MAXN 200
int main(void) {
   int i n = 0;
   double x[MAXN], value;
   while (scanf("%lf", &value) != EOF) {
      if (n >= MAXN) {
         printf("Warning:  ignored all but first %d values\n", MAXN);
         break;
      }
      x[n] = value;
      n++;
   }
   return 0;
}
```