THE EXPERT'S VOICE® IN C

Advanced Topics in C

Core Concepts in Data Structures

Noel Kalicharan

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

CHAPTER 1

Sorting, Searching, and Merging

In this chapter, we will explain the following:

- How to sort a list of items using selection and insertion sort
- How to add a new item to a sorted list so that the list remains sorted
- How to sort an array of strings
- How to sort related (parallel) arrays
- How to search a sorted list using *binary search*
- How to search an array of strings
- How to write a program to do a frequency count of words in a passage
- How to merge two sorted lists to create one sorted list

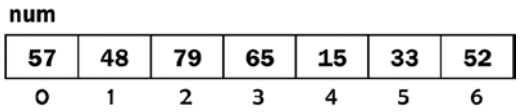
1.1 Sorting an Array: Selection Sort

S*orting* is the process by which a set of values are arranged in ascending or descending order. There are many reasons to sort. Sometimes we sort in order to produce more readable output (for example, to produce an alphabetical listing). A teacher may need to sort her students in order by name or by average score. If we have a large set of values and we want to identify duplicates, we can do so by sorting; the repeated values will come together in the sorted list.

Another advantage of sorting is that some operations can be performed faster and more efficiently with sorted data. For example, if data is sorted, it is possible to search it using binary search—this is much faster than using a sequential search. Also, merging two separate lists of items can be done much faster than if the lists were unsorted.

There are many ways to sort. In this chapter, we will discuss two of the "simple" methods: *selection* and *insertion* sort. In Chapter 10, we will look at more sophisticated ways to sort. We start with selection sort.

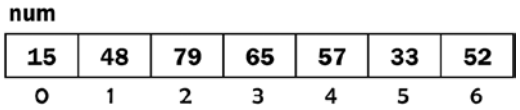
Consider the following list of numbers stored in a C array, `num`:



Sorting num in ascending order using *selection sort* proceeds as follows:

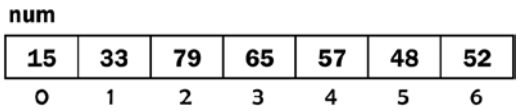
1st pass

- Find the smallest number in the entire list, from positions 0 to 6; the smallest is 15, found in position 4.
- Interchange the numbers in positions 0 and 4. This gives us the following:



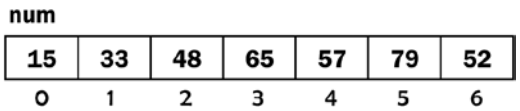
2nd pass

- Find the smallest number in positions 1 to 6; the smallest is 33, found in position 5.
- Interchange the numbers in positions 1 and 5. This gives us the following:



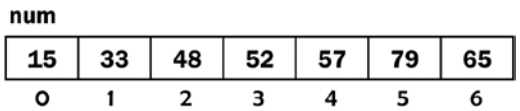
3rd pass

- Find the smallest number in positions 2 to 6; the smallest is 48, found in position 5.
- Interchange the numbers in positions 2 and 5. This gives us the following:



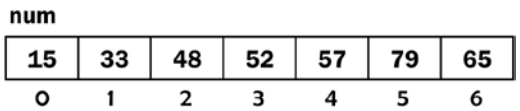
4th pass

- Find the smallest number in positions 3 to 6; the smallest is 52, found in position 6.
- Interchange the numbers in positions 3 and 6. This gives us the following:



5th pass

- Find the smallest number in positions 4 to 6; the smallest is 57, found in position 4.
- Interchange the numbers in positions 4 and 4. This gives us the following:



6th pass

- Find the smallest number in positions 5 to 6; the smallest is 65, found in position 6.
- Interchange the numbers in positions 5 and 6. This gives us the following:

num

15	33	48	52	57	65	79
0	1	2	3	4	5	6

The array is now completely sorted. Note that once the 6th largest (65) has been placed in its final position (5), the largest (79) would automatically be in the last position (6).

In this example, we made six passes. We will count these passes by letting the variable h go from 0 to 5. On each pass, we find the smallest number from positions h to 6. If the smallest number is in position s, we interchange the numbers in positions h and s.

In general, for an array of size n, we make n-1 passes. In our example, we sorted seven numbers in six passes. The following is a pseudocode outline of the algorithm for sorting num[0..n-1]:

```
for h = 0 to n - 2
   s = position of smallest number from num[h] to num[n-1]
   swap num[h] and num[s]
endfor
```

We can implement this algorithm as follows, using the generic parameter, list:

```
void selectionSort(int list[], int lo, int hi) {
//sort list[lo] to list[hi] in ascending order
   int getSmallest(int[], int, int);
   void swap(int[], int, int);
   for (int h = lo; h < hi; h++) {
      int s = getSmallest(list, h, hi);
      swap(list, h, s);
   }
}
```

The two statements in the for loop *could* be replaced by this:

```
swap(list, h, getSmallest(list, h, hi));
```

We can write getSmallest and swap as follows:

```
int getSmallest(int list[], int lo, int hi) {
//return location of smallest from list[lo..hi]
   int small = lo;
   for (int h = lo + 1; h <= hi; h++)
      if (list[h] < list[small]) small = h;
   return small;
}
```

```
void swap(int list[], int i, int j) {
//swap elements list[i] and list[j]
   int hold = list[i];
   list[i] = list[j];
   list[j] = hold;
}
```

To test whether selectionSort works properly, we write Program P1.1. Only main is shown. To complete the program, just add selectionSort, getSmallest, and swap.

Program P1.1

```
#include <stdio.h>
#define MaxNumbers 10
int main() {
   void selectionSort(int [], int, int);
   int num[MaxNumbers];
   printf("Type up to %d numbers followed by 0\n", MaxNumbers);
   int n = 0, v;
   scanf("%d", &v);
   while (v != 0 && n < MaxNumbers) {
      num[n++] = v;
      scanf("%d", &v);
   }
   if (v != 0) {
      printf("More than %d numbers entered\n", MaxNumbers);
      printf("First %d used\n", MaxNumbers);
   }
   //n numbers are stored from num[0] to num[n-1]
   selectionSort(num, 0, n-1);
   printf("\nThe sorted numbers are\n");
   for (int h = 0; h < n; h++) printf("%d ", num[h]);
   printf("\n");
}
```

The program requests up to ten numbers (as defined by MaxNumbers), stores them in the array num, calls selectionSort, and then prints the sorted list.

The following is a sample run of the program:

```
Type up to 10 numbers followed by 0
57 48 79 65 15 33 52 0
The sorted numbers are
15 33 48 52 57 65 79
```

Note that if the user enters more than ten numbers, the program will recognize this and sort only the first ten.

1.1.1 Analysis of Selection Sort

To find the smallest of *k* items, we make *k*-1 comparisons. On the first pass, we make *n*-1 comparisons to find the smallest of *n* items. On the second pass, we make *n*-2 comparisons to find the smallest of *n*-1 items. And so on, until the last pass where we make one comparison to find the smaller of two items. In general, on the *j*th pass, we make *n-j* comparisons to find the smallest of *n-j*+1 items. Hence:

```
total number of comparisons = 1 + 2 + ...+ n-1 = ½ n(n-1) » ½ n2
```

We say selection sort is of order O(n^2) ("big O *n* squared"). The constant ½ is not important in "big O" notation since, as *n* gets very big, the constant becomes insignificant.

On each pass, we swap two items using three assignments. Since we make *n*-1 passes, we make 3(*n*-1) assignments in all. Using "big O" notation, we say that the number of assignments is O(*n*). The constants 3 and 1 are not important as *n* gets large.

Does selection sort perform any better if there is order in the data? No. One way to find out is to give it a sorted list and see what it does. If you work through the algorithm, you will see that the method is oblivious to order in the data. It will make the same number of comparisons every time, regardless of the data.

As we will see, some sorting methods, such as mergesort and quicksort (see Chapters 6 and 10) require extra array storage to implement them. Note that selection sort is performed "in place" in the given array and does not require additional storage.

As an exercise, modify the programming code so that it counts the number of comparisons and assignments made in sorting a list using selection sort.

1.2 Sorting an Array: Insertion Sort

Consider the same array as before:

num

57	48	79	65	15	33	52
0	1	2	3	4	5	6

Now, think of the numbers as cards on a table that are picked up one at a time, in the order they appear in the array. Thus, we first pick up 57, then 48, then 79, and so on, until we pick up 52. However, as we pick up each new number, we add it to our hand in such a way that the numbers in our hand are all sorted.

When we pick up 57, we have just one number in our hand. We consider one number to be sorted.

When we pick up 48, we add it in front of 57 so our hand contains the following:

```
48 57
```

When we pick up 79, we place it after 57 so our hand contains the following:

```
48 57 79
```

When we pick up 65, we place it after 57 so our hand contains the following:

```
48 57 65 79
```

At this stage, four numbers have been picked up, and our hand contains them in sorted order.

When we pick up 15, we place it before 48 so our hand contains the following:

```
15 48 57 65 79
```

When we pick up 33, we place it after 15 so our hand contains the following:

```
15 33 48 57 65 79
```

Finally, when we pick up 52, we place it after 48 so our hand contains the following:

```
15 33 48 52 57 65 79
```

The numbers have been sorted in ascending order.

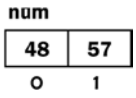
The method described illustrates the idea behind *insertion sort*. The numbers in the array will be processed one at a time, from left to right. This is equivalent to picking up the numbers from the table, one at a time. Since the first number, by itself, is sorted, we will process the numbers in the array starting from the second.

When we come to process `num[h]`, we can assume that `num[0]` to `num[h-1]` are sorted. We insert `num[h]` among `num[0]` to `num[h-1]` so that `num[0]` to `num[h]` are sorted. We then go on to process `num[h+1]`. When we do so, our assumption that `num[0]` to `num[h]` are sorted will be true.

Sorting `num` in ascending order using insertion sort proceeds as follows:

1st pass

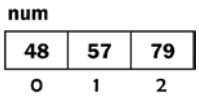
- Process `num[1]`, that is, 48. This involves placing 48 so that the first two numbers are sorted; `num[0]` and `num[1]` now contain the following:



The rest of the array remains unchanged.

2nd pass

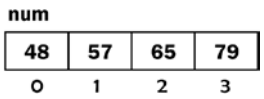
- Process `num[2]`, that is, 79. This involves placing 79 so that the first three numbers are sorted; `num[0]` to `num[2]` now contain the following:



The rest of the array remains unchanged.

3rd pass

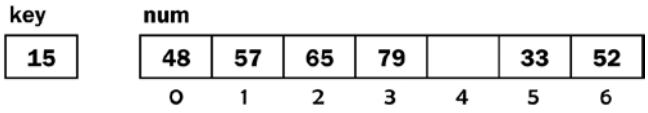
- Process `num[3]`, that is, 65. This involves placing 65 so that the first four numbers are sorted; `num[0]` to `num[3]` now contain the following:



The rest of the array remains unchanged.

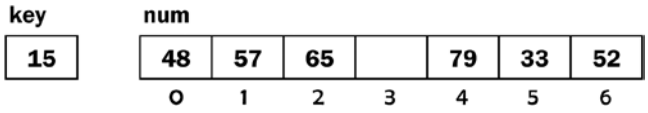
4th pass

- Process `num[4]`, that is, 15. This involves placing 15 so that the first five numbers are sorted. To simplify the explanation, think of 15 as being taken out and stored in a simple variable (key, say) leaving a "hole" in `num[4]`. We can picture this as follows:

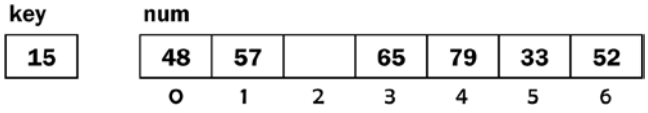


The insertion of 15 in its correct position proceeds as follows:

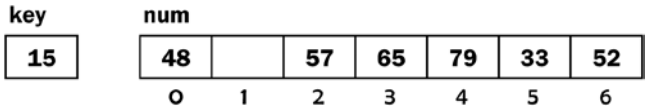
- Compare 15 with 79; it is smaller, so move 79 to location 4, leaving location 3 free. This gives the following:



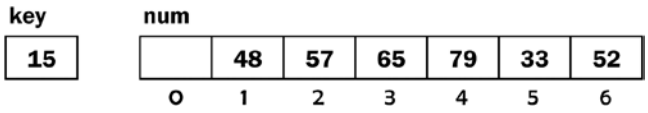
- Compare 15 with 65; it is smaller, so move 65 to location 3, leaving location 2 free. This gives the following:



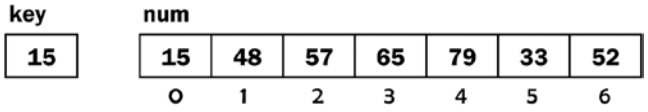
- Compare 15 with 57; it is smaller, so move 57 to location 2, leaving location 1 free. This gives the following:



- Compare 15 with 48; it is smaller, so move 48 to location 1, leaving location 0 free. This gives the following:



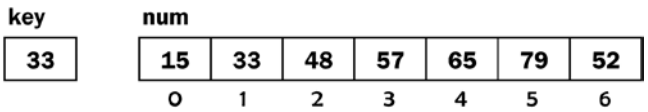
- There are no more numbers to compare with 15, so it is inserted in location 0, giving the following:



- We can express the logic of placing 15 (key) by comparing it with the numbers to its left, starting with the nearest one. As long as key is less than num[k], for some k, we move num[k] to position num[k + 1] and move on to consider num[k-1], providing it exists. It won't exist when k is actually 0. In this case, the process stops, and key is inserted in position 0.

5th pass

- Process num[5], that is, 33. This involves placing 33 so that the first six numbers are sorted. This is done as follows:
  - Store 33 in key, leaving location 5 free;
  - Compare 33 with 79; it is smaller, so move 79 to location 5, leaving location 4 free.
  - Compare 33 with 65; it is smaller, so move 65 to location 4, leaving location 3 free.
  - Compare 33 with 57; it is smaller, so move 57 to location 3, leaving location 2 free.
  - Compare 33 with 48; it is smaller, so move 48 to location 2, leaving location 1 free.
  - Compare 33 with 15; it is bigger, so insert 33 in location 1. This gives the following:

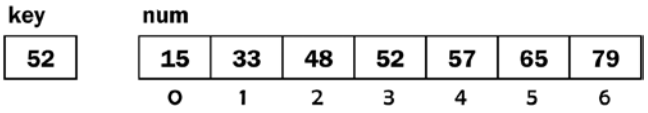


- We can express the logic of placing 33 by comparing it with the numbers to its left, starting with the nearest one. As long as key is less than num[k], for some k, we move num[k] to position num[k + 1] and move on to consider num[k-1], providing it exists. If key is greater than or equal to num[k] for some k, then key is inserted in position k+1. Here, 33 is greater than num[0] and so is inserted into num[1].

6th pass

- Process num[6], that is, 52. This involves placing 52 so that the first seven (all) numbers are sorted. This is done as follows:
  - Store 52 in key, leaving location 6 free.
  - Compare 52 with 79; it is smaller, so move 79 to location 6, leaving location 5 free.
  - Compare 52 with 65; it is smaller, so move 65 to location 5, leaving location 4 free.

- Compare 52 with 57; it is smaller, so move 57 to location 4, leaving location 3 free.
- Compare 52 with 48; it is bigger, so insert 52 in location 3. This gives the following:



The array is now completely sorted.
The following is an outline of how to sort the first n elements of an array, num, using insertion sort:

```
for h = 1 to n - 1 do
   insert num[h] among num[0] to num[h-1] so that num[0] to num[h] are sorted
endfor
```

Using this outline, we write the function insertionSort using the parameter list.

```
void insertionSort(int list[], int n) {
//sort list[0] to list[n-1] in ascending order
   for (int h = 1; h < n; h++) {
      int key = list[h];
      int k = h - 1; //start comparing with previous item
      while (k >= 0 && key < list[k]) {
         list[k + 1] = list[k];
         --k;
      }
      list[k + 1] = key;
   } //end for
} //end insertionSort
```

The while statement is at the heart of the sort. It states that as long as we are within the array (k >= 0) and the current number (key) is less than the one in the array (key < list[k]), we move list[k] to the right (list[k + 1] = list[k]) and move on to the next number on the left (--k).

We exit the while loop if k is equal to -1 or if key is greater than or equal to list[k], for some k. In either case, key is inserted into list[k + 1].

If k is -1, it means that the current number is smaller than all the previous numbers in the list and must be inserted in list[0]. But list[k + 1] *is* list[0] when k is -1, so key is inserted correctly in this case.

The function sorts in ascending order. To sort in descending order, all we have to do is change < to > in the while condition, like this:

```
while (k >= 0 && key > list[k])
```

Now, a key moves to the left if it is *bigger*.

We write Program P1.2 to test whether insertionSort works correctly. Only main is shown. Adding the function insertionSort completes the program.

Program P1.2

```
#include <stdio.h>
#define MaxNumbers 10
int main() {
   void insertionSort(int [], int);
   int num[MaxNumbers];
   printf("Type up to %d numbers followed by 0\n", MaxNumbers);
   int n = 0, v;
   scanf("%d", &v);
   while (v != 0 && n < MaxNumbers) {
      num[n++] = v;
      scanf("%d", &v);
   }
   if (v != 0) {
      printf("More than %d numbers entered\n", MaxNumbers);
      printf("First %d used\n", MaxNumbers);
   }
   //n numbers are stored from num[0] to num[n-1]
   insertionSort(num, n);
   printf("\nThe sorted numbers are\n");
   for (int h = 0; h < n; h++) printf("%d ", num[h]);
   printf("\n");
}
```

The program requests up to ten numbers (as defined by `MaxNumbers`), stores them in the array `num`, calls `insertionSort`, and then prints the sorted list.

The following is a sample run of the program:

```
Type up to 10 numbers followed by 0
57 48 79 65 15 33 52 0
The sorted numbers are
15 33 48 52 57 65 79
```

Note that if the user enters more than ten numbers, the program will recognize this and sort only the first ten.

We could easily generalize `insertionSort` to sort a *portion* of a list. To illustrate, we rewrite `insertionSort` (calling it `insertionSort1`) to sort `list[lo]` to `list[hi]` where `lo` and `hi` are passed as arguments to the function.

Since element `lo` is the first one, we start processing elements from `lo+1` until element `hi`. This is reflected in the `for` statement. Also now, the lowest subscript is `lo`, rather than `0`. This is reflected in the `while` condition `k >= lo`. Everything else remains the same as before.

```
void insertionSort1(int list[], int lo, int hi) {
//sort list[lo] to list[hi] in ascending order
   for (int h = lo + 1; h <= hi; h++) {
      int key = list[h];
      int k = h - 1; //start comparing with previous item
      while (k >= lo && key < list[k]) {
         list[k + 1] = list[k];
         --k;
      }
```

```
            list[k + 1] = key;
         } //end for
      } //end insertionSort1
```

1.2.1 Analysis of Insertion Sort

In processing item *j*, we can make as few as one comparison (if `num[j]` is bigger than `num[j-1]`) or as many as *j*-1 comparisons (if `num[j]` is smaller than all the previous items). For random data, we would expect to make ½(*j*-1) comparisons, on average. Hence, the average total number of comparisons to sort *n* items is as follows:

$$\sum_{j=2}^{n} \frac{1}{2}(j-1) = \frac{1}{2}\{1+2+\ldots+n-1\} = \frac{1}{4}n(n-1) \approx \frac{1}{4}n^2$$

We say insertion sort is of order $O(n^2)$ ("big O n squared"). The constant ¼ is not important as n gets large.

Each time we make a comparison, we also make an assignment. Hence, the total number of assignments is also ¼ n(n-1) ≈ ¼ n^2.

We emphasize that this is an average for random data. Unlike selection sort, the actual performance of insertion sort depends on the data supplied. If the given array is already sorted, insertion sort will quickly determine this by making n-1 comparisons. In this case, it runs in O(n) time. One would expect that insertion sort will perform better the more order there is in the data.

If the given data is in descending order, insertion sort performs at its worst since each new number has to travel all the way to the beginning of the list. In this case, the number of comparisons is ½ n(n-1) ≈ ½ n^2. The number of assignments is also ½ n(n-1) ≈ ½ n^2.

Thus, the number of comparisons made by insertion sort ranges from n-1 (best) to ¼ n^2 (average) to ½ n^2 (worst). The number of assignments is always the same as the number of comparisons.

As with selection sort, insertion sort does not require extra array storage for its implementation.

As an exercise, modify the programming code so that it counts the number of comparisons and assignments made in sorting a list using insertion sort.

1.3 Inserting an Element in Place

Insertion sort uses the idea of adding a new element to an already sorted list so that the list remains sorted. We can treat this as a problem in its own right (nothing to do with insertion sort). Specifically, given a sorted list of items from `list[m]` to `list[n]`, we want to add a new item (`newItem`, say) to the list so that `list[m]` to `list[n + 1]` are sorted.

Adding a new item increases the size of the list by 1. We assume that the array has room to hold the new item. We write the function `insertInPlace` to solve this problem.

```
void insertInPlace(int newItem, int list[], int m, int n) {
//list[m] to list[n] are sorted
//insert newItem so that list[m] to list[n+1] are sorted
   int k = n;
   while (k >= m && newItem < list[k]) {
      list[k + 1] = list[k];
      --k;
   }
   list[k + 1] = newItem;
} //end insertInPlace
```

Using `insertInPlace`, we can rewrite `insertionSort` (calling it `insertionSort2`) as follows:

```
void insertionSort2(int list[], int lo, int hi) {
//sort list[lo] to list[hi] in ascending order
   void insertInPlace(int, int [], int, int);
   for (int h = lo + 1; h <= hi; h++)
      insertInPlace(list[h], list, lo, h - 1);
} //end insertionSort2
```

1.4 Sorting an Array of Strings

Consider the problem of sorting a list of names in alphabetical order. In C, each name is stored in a character array. To store several names, we need a two-dimensional character array. For example, we can store eight names as shown in Figure 1-1.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	T	a	y	l	o	r	,		V	i	c	t	o	r	\0
1	D	u	n	c	a	n	,		D	e	n	i	s	e	\0
2	R	a	m	d	h	a	n	,		K	a	m	a	l	\0
3	S	i	n	g	h	,		K	r	i	s	h	n	a	\0
4	A	l	i	,		M	i	c	h	a	e	l	\0		
5	S	a	w	h	,		A	n	i	s	a	\0			
6	K	h	a	n	,		C	a	r	o	l	\0			
7	O	w	e	n	,		D	a	v	i	d	\0			

Figure 1-1. *Two-dimensional character array*

Doing so will require a declaration such as the following:

```
char list[8][15];
```

To cater for longer names, we can increase `15`, and to cater for more names, we can increase `8`.

The *process* of sorting `list` is essentially the same as sorting an array of integers. The major difference is that whereas we use `<` to compare two numbers, we must use `strcmp` to compare two names. In the function `insertionSort` shown at the end of Section 1.3, the `while` condition changes from this:

```
while (k >= lo && key < list[k])
```

to the following, where `key` is now declared as `char key[15]`:

```
while (k >= lo && strcmp(key, list[k]) < 0)
```

Also, we must now use `strcpy` (since we can't use `=` for strings) to assign a name to another location. Here is the complete function:

```
void insertionSort3(int lo, int hi, int max, char list[][max]) {
//Sort the strings in list[lo] to list[hi] in alphabetical order.
//The maximum string size is max - 1 (one char taken up by \0).
```

```
      char key[max];
      for (int h = lo + 1; h <= hi; h++) {
         strcpy(key, list[h]);
         int k = h - 1; //start comparing with previous item
         while (k >= lo && strcmp(key, list[k]) < 0) {
            strcpy(list[k + 1], list[k]);
            --k;
         }
         strcpy(list[k + 1], key);
      } //end for
   } //end insertionSort3
```

Note the declaration of `list` (`char list[][max]`) in the parameter list. The size of the first dimension is left unspecified, as for one-dimensional arrays. The size of the second dimension is specified using the parameter `max`; the value of `max` will be specified when the function is called. This gives us a bit more flexibility since we can specify the size of the second dimension at run time.

We write a simple `main` routine to test `insertionSort3` as shown in Program P1.3.

Program P1.3

```
#include <stdio.h>
#include <string.h>
#define MaxNameSize 14
#define MaxNameBuffer MaxNameSize+1
#define MaxNames 8
int main() {
   void insertionSort3(int, int, int max, char [][max]);
   char name[MaxNames][MaxNameBuffer] = {"Taylor, Victor", "Duncan, Denise",
      "Ramdhan, Kamal", "Singh, Krishna", "Ali, Michael",
      "Sawh, Anisa", "Khan, Carol", "Owen, David" };

   insertionSort3(0, MaxNames-1, MaxNameBuffer, name);
   printf("\nThe sorted names are\n\n");
   for (int h = 0; h < MaxNames; h++) printf("%s\n", name[h]);
} //end main
```

The declaration of `name` initializes it with the eight names shown in Figure 1-1. When run, the program produces the following output:

```
The sorted names are
Ali, Michael
Duncan, Denise
Khan, Carol
Owen, David
Ramdhan, Kamal
Sawh, Anisa
Singh, Krishna
Taylor, Victor
```

1.5 Sorting Parallel Arrays

It is quite common to have related information in different arrays. For example, suppose, in addition to `name`, we have an integer array `id` such that `id[h]` is an identification number associated with `name[h]`, as shown in Figure 1-2.

	name	id
0	Taylor, Victor	3050
1	Duncan, Denise	2795
2	Ramdhan, Kamal	4455
3	Singh, Krishna	7824
4	Ali, Michael	6669
5	Sawh, Anisa	5000
6	Khan, Carol	5464
7	Owen, David	6050

Figure 1-2. *Two arrays with related information*

Consider the problem of sorting the names in alphabetical order. At the end, we would want each name to have its correct ID number. So, for example, after the sorting is done, `name[0]` should contain "Ali, Michael" and `id[0]` should contain `6669`.

To achieve this, each time a name is moved during the sorting process, the corresponding ID number must also be moved. Since the name and ID number must be moved "in parallel," we say we are doing a *parallel sort* or we are sorting *parallel arrays*.

We rewrite `insertionSort3` to illustrate how to sort parallel arrays. We simply add the code to move an ID whenever a name is moved. We call it `parallelSort`.

```
void parallelSort(int lo, int hi, int max, char list[][max], int id[]) {
//Sort the names in list[lo] to list[hi] in alphabetical order, ensuring that
//each name remains with its original id number.
//The maximum string size is max - 1 (one char taken up by \0).
   char key[max];
   for (int h = lo + 1; h <= hi; h++) {
      strcpy(key, list[h]);
      int m = id[h];        // extract the id number
      int k = h - 1;        //start comparing with previous item
      while (k >= lo && strcmp(key, list[k]) < 0) {
         strcpy(list[k + 1], list[k]);
         id[k + 1] = id[k];  // move up id number when we move a name
         --k;
      }
      strcpy(list[k + 1], key);
      id[k + 1] = m;        // store the id number in the same position as the name
   } //end for
} //end parallelSort
```

We test `parallelSort` by writing the following `main` routine:

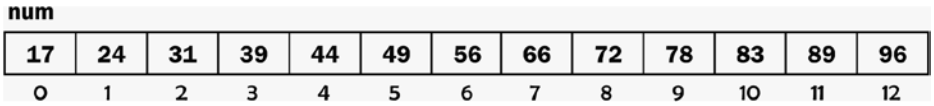
```
#include <stdio.h>
#include <string.h>
#define MaxNameSize 14
#define MaxNameBuffer MaxNameSize+1
#define MaxNames 8
int main() {
   void parallelSort(int, int, int max, char [][max], int[]);
   char name[MaxNames][MaxNameBuffer] = {"Taylor, Victor", "Duncan, Denise",
      "Ramdhan, Kamal", "Singh, Krishna", "Ali, Michael",
      "Sawh, Anisa", "Khan, Carol", "Owen, David" };
   int id[MaxNames] = {3050,2795,4455,7824,6669,5000,5464,6050};

   parallelSort(0, MaxNames-1, MaxNameBuffer, name, id);
   printf("\nThe sorted names and IDs are\n\n");
   for (int h = 0; h < MaxNames; h++) printf("%-18s %d\n", name[h], id[h]);
} //end main
```

When run, it produces the following output:

```
The sorted names and IDs are
Ali, Michael      6669
Duncan, Denise    2795
Khan, Carol       5464
Owen, David       6050
Ramdhan, Kamal    4455
Sawh, Anisa       5000
Singh, Krishna    7824
Taylor, Victor    3050
```

1.6 Binary Search

Binary search is a fast method for searching a list of items for a given one, *providing the list is sorted* (either ascending or descending). To illustrate the method, consider a list of 13 numbers, sorted in ascending order and stored in an array `num[0..12]`.



Suppose we want to search for 66. The search proceeds as follows:

1. First, we find the middle item in the list. This is 56 in position 6. We compare 66 with 56. Since 66 is bigger, we know that if 66 is in the list at all, it *must* be *after* position 6, since the numbers are in ascending order. In our next step, we confine our search to locations 7 to 12.
2. Next, we find the middle item from locations 7 to 12. In this case, we can choose either item 9 or item 10. The algorithm we will write will choose item 9, that is, 78.

3. We compare 66 with 78. Since 66 is smaller, we know that if 66 is in the list at all, it *must* be *before* position 9, since the numbers are in ascending order. In our next step, we confine our search to locations 7 to 8.

4. Next, we find the middle item from locations 7 to 8. In this case, we can choose either item 7 or item 8. The algorithm we will write will choose item 7, that is, 66.

5. We compare 66 with 66. Since they are the same, our search ends successfully, finding the required item in position 7.

Suppose we were searching for 70. The search will proceed as described above until we compare 70 with 66 (in location 7).

1. Since 70 is bigger, we know that if 70 is in the list at all, it *must* be *after* position 7, since the numbers are in ascending order. In our next step, we confine our search to locations 8 to 8. This is just one location.

2. We compare 70 with item 8, that is, 72. Since 70 is smaller, we know that if 70 is in the list at all, it *must* be *before* position 8. Since it can't be after position 7 *and* before position 8, we conclude that it is not in the list.

At each stage of the search, we confine our search to some portion of the list. Let us use the variables `lo` and `hi` as the subscripts that define this portion. In other words, our search will be confined to `num[lo]` to `num[hi]`.

Initially, we want to search the entire list so that we will set `lo` to `0` and `hi` to `12`, in this example.

How do we find the subscript of the middle item? We will use the following calculation:

```
mid = (lo + hi) / 2;
```

Since integer division will be performed, the fraction, if any, is discarded. For example, when `lo` is `0` and `hi` is `12`, `mid` becomes 6; when `lo` is 7 and `hi` is `12`, `mid` becomes 9; and when `lo` is 7 and `hi` is 8, `mid` becomes 7.

As long as `lo` is less than or equal to `hi`, they define a nonempty portion of the list to be searched. When `lo` is equal to `hi`, they define a single item to be searched. If `lo` ever gets bigger than `hi`, it means we have searched the entire list and the item was not found.

Based on these ideas, we can now write a function `binarySearch`. To be more general, we will write it so that the calling routine can specify which portion of the array it wants the search to look for the item.

Thus, the function must be given the item to be searched for (`key`), the array (`list`), the start position of the search (`lo`), and the end position of the search (`hi`). For example, to search for the number 66 in the array `num`, shown earlier, we can issue the call `binarySearch(66, num, 0, 12)`.

The function must tell us the result of the search. If the item is found, the function will return its location. If not found, it will return `-1`.

```
int binarySearch(int key, int list[], int lo, int hi) {
//search for key from list[lo] to list[hi]
//if found, return its location; otherwise, return -1
   while (lo <= hi) {
      int mid = (lo + hi) / 2;
      if (key == list[mid]) return mid; // found
      if (key < list[mid]) hi = mid - 1;
      else lo = mid + 1;
   }
   return -1; //lo and hi have crossed; key not found
} //end binarySearch
```

If item contains a number to be searched for, we can write code as follows:

```
int ans = binarySearch(item, num, 0, 12);
if (ans == -1) printf("%d not found\n", item);
else printf("%d found in location %d\n", item, ans);
```

If we want to search for `item` from locations `i` to `j`, we can write the following:

```
int ans = binarySearch(item, num, i, j);
```

1.7 Searching an Array of Strings

We can search a sorted array of strings (names in alphabetical order, say) using the same technique we used for searching an integer array. The major differences are in the declaration of the array and the use of `strcmp`, rather than == or <, to compare two strings. The following is the string version of `binarySearch`:

```
int binarySearch(int lo, int hi, char key[], int max, char list[][max]) {
//search for key from list[lo] to list[hi]
//if found, return its location; otherwise, return -1
   while (lo <= hi) {
      int mid = (lo + hi) / 2;
      int cmp = strcmp(key, list[mid]);
      if (cmp == 0) return mid; // found
      if (cmp < 0) hi = mid - 1;
      else lo = mid + 1;
   }
   return -1; //lo and hi have crossed; key not found
} //end binarySearch
```

The function can be tested with `main` shown in Program P1.4.

Program P1.4

```
#include <stdio.h>
#include <string.h>
#define MaxNameSize 14
#define MaxNameBuffer MaxNameSize+1
#define MaxNames 8
int main () {
   int binarySearch(int, int, char [], int max, char [][max]);
   int n;
   char name[MaxNames][MaxNameBuffer] = {"Ali, Michael","Duncan, Denise",
      "Khan, Carol","Owen, David", "Ramdhan, Kamal",
      "Sawh, Anisa", "Singh, Krishna", "Taylor, Victor"};
   n = binarySearch(0, 7, "Ali, Michael", MaxNameBuffer, name);
   printf("%d\n", n);  //will print 0, location of Ali, Michael
   n = binarySearch(0, 7, "Taylor, Victor", MaxNameBuffer, name);
   printf("%d\n", n); //will print 7, location of Taylor, Victor
   n = binarySearch(0, 7, "Owen, David", MaxNameBuffer, name);
   printf("%d\n", n); //will print 3, location of Owen, David
```

```
    n = binarySearch(4, 7, "Owen, David", MaxNameBuffer, name);
    printf("%d\n", n); //will print -1, since Owen, David is not in locations 4 to 7
    n = binarySearch(0, 7, "Sandy, Cindy", MaxNameBuffer, name);
    printf("%d\n", n); //will print -1 since Sandy, Cindy is not in the list
} //end main
```

This sets up the array `name` with the names in alphabetical order. It then calls `binarySearch` with various names and prints the result of each search.

One may wonder what might happen with a call like this:

```
n = binarySearch(5, 10, MaxNameBuffer, "Sawh, Anisa", name);
```

Here, we are telling `binarySearch` to look for "Sawh, Anisa" in locations 5 to 10 of the given array. However, locations 8 to 10 do not exist in the array. The result of the search will be unpredictable. The program may crash or return an incorrect result. The onus is on the calling program to ensure that `binarySearch` (or any other function) is called with valid arguments.

1.8 Example: Word Frequency Count

Let's write a program to read an English passage and count the number of times each word appears. The output consists of an alphabetical listing of the words and their frequencies.

We can use the following outline to develop our program:

```
while there is input
   get a word
   search for word
   if word is in the table
      add 1 to its count
   else
      add word to the table
      set its count to 1
   endif
endwhile
print table
```

This is a typical "search and insert" situation. We search for the next word among the words stored so far. If the search succeeds, we need only increment its count. If the search fails, the word is put in the table, and its count set to 1.

A major design decision here is how to search the table, which, in turn, will depend on where and how a new word is inserted in the table. The following are two possibilities:

1. A new word is inserted in the next free position in the table. This implies that a sequential search must be used to look for an incoming word since the words would not be in any particular order. This method has the advantages of simplicity and easy insertion, but searching takes longer because more words are put in the table.

2. A new word is inserted in the table in such a way that the words are always in alphabetical order. This may entail moving words that have already been stored so that the new word may be slotted in the right place. However, since the table is in order, a binary search can be used to search for an incoming word.

For (2), searching is faster, but insertion is slower than in (1). Since, in general, searching is done more frequently than inserting, (2) might be preferable.

Another advantage of (2) is that, at the end, the words will already be in alphabetical order and no sorting will be required. If (1) is used, the words will need to be sorted to obtain the alphabetical order.

We will write our program using the approach in (2). The complete program is shown as Program P1.5.

Program P1.5

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#define MaxWords 50
#define MaxLength 10
#define MaxWordBuffer MaxLength+1
int main() {
   int getWord(FILE *, char[]);
   int binarySearch(int, int, char [], int max, char [][max]);
   void addToList(char[], int max, char [][max], int[], int, int);
   void printResults(FILE *, int max, char [][max], int[], int);
   char wordList[MaxWords][MaxWordBuffer], word[MaxWordBuffer];
   int frequency[MaxWords], numWords = 0;

   FILE * in = fopen("passage.txt", "r");
   if (in == NULL){
      printf("Cannot find file\n");
      exit(1);
   }

   FILE * out = fopen("output.txt", "w");
   if (out == NULL){
      printf("Cannot create output file\n");
      exit(2);
   }

   for (int h = 0; h < MaxWords; h++) frequency[h] = 0;

   while (getWord(in, word) != 0) {
      int loc = binarySearch (0, numWords-1, word, MaxWordBuffer, wordList);
      if (strcmp(word, wordList[loc]) == 0) ++frequency[loc]; //word found
      else //this is a new word
         if (numWords < MaxWords) { //if table is not full
            addToList(word, MaxWordBuffer, wordList, frequency, loc, numWords-1);
            ++numWords;
         }
         else fprintf(out, "'%s' not added to table\n", word);
   }
   printResults(out, MaxWordBuffer, wordList, frequency, numWords);
} // end main
```

```
int getWord(FILE * in, char str[]) {
// stores the next word, if any, in str; word is converted to lowercase
// returns 1 if a word is found; 0, otherwise
   char ch;
   int n = 0;
   // read over white space
   while (!isalpha(ch = getc(in)) && ch != EOF) ; //empty while body
   if (ch == EOF) return 0;
   str[n++] = tolower(ch);
   while (isalpha(ch = getc(in)) && ch != EOF)
      if (n < MaxLength) str[n++] = tolower(ch);
   str[n] = '\0';
   return 1;
} // end getWord

int binarySearch(int lo, int hi, char key[], int max, char list[][max]) {
//search for key from list[lo] to list[hi]
//if found, return its location;
//if not found, return the location in which it should be inserted
//the calling program will check the location to determine if found
   while (lo <= hi) {
      int mid = (lo + hi) / 2;
      int cmp = strcmp(key, list[mid]);
      if (cmp == 0) return mid; // found
      if (cmp < 0) hi = mid - 1;
      else lo = mid + 1;
   }
   return lo; //not found; should be inserted in location lo
} //end binarySearch

void addToList(char item[], int max, char list[][max], int freq[], int p, int n) {
//adds item in position list[p]; sets freq[p] to 1
//shifts list[n] down to list[p] to the right
   for (int h = n; h >= p; h--) {
      strcpy(list[h+1], list[h]);
      freq[h+1] = freq[h];
   }
   strcpy(list[p], item);
   freq[p] = 1;
} //end addToList

void printResults(FILE *out, int max, char list[][max], int freq[], int n) {
   fprintf(out, "\nWords        Frequency\n\n");
   for (int h = 0; h < n; h++)
      fprintf(out, "%-15s %2d\n", list[h], freq[h]);
} //end printResults
```

When Program P1.5 was run with this data, it produced the output that follows:

```
The quick brown fox jumps over the lazy dog. Congratulations!
If the quick brown fox jumped over the lazy dog then
Why did the quick brown fox jump over the lazy dog?
To recuperate!
```

Here is the output:

```
Words       Frequency
brown           3
congratula      1
did             1
dog             3
fox             3
if              1
jump            1
jumped          1
jumps           1
lazy            3
over            3
quick           3
recuperate      1
the             6
then            1
to              1
why             1
```

The following are comments on Program P1.5:

- For our purposes, we assume that a word begins with a letter and consists of letters only. If you want to include other characters (such as a hyphen or apostrophe), you need change only the `getWord` function.
- `MaxWords` denotes the maximum number of distinct words catered for. For testing the program, we have used `50` for this value. If the number of distinct words in the passage exceeds `MaxWords` (50, say), any words after the 50th will be read but not stored, and a message to that effect will be printed. However, the count for a word already stored will be incremented if it is encountered again.
- `MaxLength` (we use `10` for testing) denotes the maximum length of a word. Strings are declared using `MaxLength+1` (defined as `MaxWordBuffer`) to cater for `\0`, which must be added at the end of each string.
- `main` checks that the input file exists and that the output file can be created. Next, it initializes the frequency counts to `0`. It then processes the words in the passage based on the outline shown at the start of Section 1.8.
- `getWord` reads the input file and stores the next word found in its string argument. It returns `1` if a word is found and `0`, otherwise. If a word is longer than `MaxLength`, only the first `MaxLength` letters are stored; the rest are read and discarded. For example, `congratulations` is truncated to `congratula` using a word size of `10`.