```
/*   File: sortsrch.c  */
#include <stdio.h>
#include "sortsrch.h"
#define DEBUG
/*
     Linear or sequential search of an array x[] of size lim for
     an item key.
*/
int seqsrch(int x[], int lim, int key)
{    int i;

     for (i = 0; i < lim; i++)
          if (x[i] == key)
               return(i);
     return(-1);
}
```

Figure 10.2: Code for the function `seqsrch()`

```
/*   File: sortsrch.h
     This file contains prototypes for sort and search functions.
*/
int seqsrch(int x[], int lim, int key);
```

Figure 10.3: Initial contents of `sortsrch.h`

We will implement the algorithm for search as a function, `seqsrch()` since searching an array may be required in many programs, and a function incorporating linear search can be used for many applications. The function is passed the array of integers, `x[]`, the number of elements in the array, `lim`, and the key to search for, `key`. The function is shown in Figure 10.2.

The loop compares each element of the array with `key`. If an element with the same value is found at an index `i`, `i` is returned. The loop terminates when the array limit is reached; in which case, no element equal to `key` was found in the array, and -1 is returned. If there is more than one element in the array with the same value as `key`, the function terminates the search as soon as the first element is found, returning its index.

We have defined `DEBUG` in `sortsrch.c` so that debug statements we may add to the code will be compiled. We have also included file `sortsrch.h` in `sortsrch.c` which, as shown in Figure 10.3 contains the prototypes for functions defined in `sortsrch.c` since some of the functions defined in `sortsrch.c` may be used by other functions to be written in the file. We will continue to add more functions to `sortsrch.c` and corresponding prototypes to `sortsrch.h` as we proceed through this chapter; however, we will not always show the additions of prototypes to `sortsrch.h`.

Our task now requires us to write a simple driver to repeatedly call the function, `seqsrch()`. For simplicity, we will declare an initialized array. The driver uses a function, `pr_aray_line()`, to

print an array with ten elements per line to save space. Figure 10.4 shows the program driver.

The driver first prints an initialized array, `id[]`, and then searches for items entered by the user. If an item is found in the array, its index is printed. Otherwise, a message is printed. The function, `pr_aray_line()` is included in `sortsrch.c` and its prototype in `sortsrch.h`. These additions are shown in Figure 10.5.

The function prints at successive elements on one line until the array index modulo 10 is zero when it prints a newline and continues. A sample session for the program `srcharay.c` is shown below:

```
***Sequential Search***

The id array is:
45 67 12 34 25 39

Type an integer, EOF to quit:  23
Item 23 is not in the array

Type an integer, EOF to quit:  12
Item 12 is found at index 2

Type an integer, EOF to quit:  45
Item 45 is found at index 0

Type an integer, EOF to quit:  ^D
```

10.2 Improving Search — Sorting the Data

As we mentioned above, linear search may be useful when the number of elements is small; however, when there are many data items in the array, linear search may require a long time to find the element matching the key. In the case where such an element is not in the data base, we must search the entire collection to find that out. Consider the phone book again. When we want to look up someone's phone number, we do not start at the beginning of the book and look line by line until we find the name. Instead, we make use of the fact that the data items are sorted by name in the phone book. (Of course, if we had a phone number and wanted to find the name, we would have to resort to linear search — we do not often do that).

Before we develop an algorithm to conduct a more efficient search of sorted data, we first describe several algorithms which will sort an array of data. There are numerous ways to sort data, some more suitable than others for different applications. In this section we will describe three different standard algorithms: selection sort, bubble sort, and insertion sort.

10.2.1 Selection Sort

The idea of selection sort is rather simple: we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index

```
/*   File:srcharay.c
     Other Source Files: sortsrch.c
     Header Files: sortsrch.h
     This program searches an array sequentially for items typed in
     by the user. It prints out the array index where an item is found,
     or else prints a message.
*/

#include <stdio.h>
#include "sortsrch.h"

main()
{    int id[] = {45, 67, 12, 34, 25, 39};
     int n, i;

     printf("***Sequential Search***\n\n");
     printf("The array is:\n");
     pr_aray_line(id, 6);
     printf("Type an integer, EOF to quit: ");

     while (scanf("%d", &n) != EOF) {
          i = seqsrch(id, 6, n);

          if (i >= 0)
               printf("Item %d is found at index %d\n", n, i);
          else
               printf("Item %d is not in the array\n", n);

          printf("Type an integer, EOF to quit: ");
     }
}
```

Figure 10.4: Driver to test `seqsrch()`

```
/* File: sortsrch.c - continued */
/* Prints an array horizontally. */
void pr_aray_line(int x[], int lim)
{    int i;

     for (i = 0; i < lim; i++) {
          if (i % 10 == 0)
               printf("\n");
          printf("%d  ", x[i]);
     }

     printf("\n");
}

/* File: sortsrch.h - continued */
void pr_aray_line(int x[], int lim);
```

Figure 10.5: Adding the code for `pr_aray_line()`

position. We can do this by swapping the element at the highest index and the largest element. We then reduce the *effective size* of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

For example, consider the following array, shown with array elements in sequence separated by commas:

63, 75, **90**, 12, **27**

The leftmost element is at index zero, and the rightmost element is at the highest array index, in our case, 4 (the effective size of our array is 5). The largest element in this effective array (index 0-4) is at index 2. We have shown the largest element and the one at the highest index in **bold**. We then swap the element at index 2 with that at index 4. The result is:

63, **75**, 27, **12**, 90

We reduce the effective size of the array to 4, making the highest index in the effective array now 3. The largest element in this effective array (index 0-3) is at index 1, so we swap elements at index 1 and 3 (in **bold**):

63, 12, **27**, 75, 90

The next two steps give us:

27, **12**, 63, 75, 90

12, 27, 63, 75, 90

The last effective array has only one element and needs no sorting. The entire array is now sorted. The algorithm for an array, `x`, with `lim` elements is easy to write down:

```
for (eff_size = lim; eff_size > 1; eff_size--)
     find maxpos, the location of the largest element in the effective
          array: index 0 to eff_size - 1
     swap elements of x at index maxpos and index eff_size - 1
```

The implementation of the selection sort algorithm in C, together with a driver program is shown in Figure 10.6.

Sample Session:

```
Original array:
63 75 90 12 27
Sorted array:
12 27 63 75 90
```

The driver prints the array, calls `selection_sort()` to sort the array, and prints the sorted array. The code for `selection_sort()` follows the algorithm exactly; it calls `get_maxpos()` to get the index of the largest element in an array of a specified size. Once `maxpos` is found, the element at that index is swapped with the element at index `eff_size-1`, using the temporary variable, `tmp`.

We may be concerned about the efficiency of our algorithm and its implementation as a program. The efficiency of an algorithm depends on the number of major computations involved in performing the algorithm. The efficiency of the program depends on that of the algorithm and the efficiency of the code implementing the algorithm. Notice we included the code for swapping array elements within the loop in `selection_sort` rather than calling a function to perform this operation. A function call requires added processing time in order to store argument values, transfer program control, and retrieve the returned value. When a function call is in a loop that may be executed many times, the extra processing time may become significant. Thus, if the array to be sorted is quite large, we can improve program efficiency by eliminating a function call to swap data elements. Similarly, we may include the code for `get_maxpos()` in `selection_sort()`:

```
void selection_sort(int x[], int lim)
{    int i, eff_size, maxpos, tmp;

     for (eff_size = lim; eff_size > 1; eff_size--) {
          for (i = 0; i < eff_size; i++)
               maxpos = x[i] > x[maxpos] ? i : maxpos;
          tmp = x[maxpos];
          x[maxpos] = x[eff_size - 1];
          x[eff_size - 1] = tmp;
     }
}
```

10.2.2 Bubble Sort

An alternate way of putting the largest element at the highest index in the array uses an algorithm called *bubble sort*. While this method is neither as efficient, nor as straightforward, as selection sort, it is popularly used to illustrate sorting. We include it here as an alternate method.

Like selection sort, the idea of bubble sort is to repeatedly move the largest element to the highest index position of the array. As in selection sort, each iteration reduces the effective size of the array. The two algorithms differ in how this is done. Rather than search the entire effective

```
/* File: select.c
   This program implements selection sort.
*/
#include <stdio.h>
#define MAX 10
void selection_sort(int x[], int lim);
int get_maxpos(int x[], int lim);
void print_aray(int x[], int lim);

main()
{  int scores[MAX] = {63, 75, 90, 12, 27};

   printf("Original array:\n");
   print_aray(scores, 5);
   selection_sort(scores, 5);
   printf("Sorted array:\n");
   print_aray(scores, 5);
}
/* Selection sort function for an array x[] with lim elements. */
void selection_sort(int x[], int lim)
{  int eff_size, maxpos, tmp;

   for (eff_size = lim; eff_size > 1; eff_size--) {
     maxpos = get_maxpos(x, eff_size);
     tmp = x[maxpos];
     x[maxpos] = x[eff_size - 1];
     x[eff_size - 1] = tmp;
   }
}
/* Function returns the index of the largest element in the array x[]. */
int get_maxpos(int x[], int eff_size)
{  int i, maxpos = 0;

   for (i = 0; i < eff_size; i++)
     maxpos = x[i] > x[maxpos] ? i : maxpos;
   return maxpos;
}
/* Function prints an integer array of size lim. */
void print_aray(int x[], int lim)
{  int i;

   for (i = 0; i < lim; i++)
     printf("%d  ", x[i]);
}
```

Figure 10.6: Sorting an array using Selection Sort

array to find the largest element, bubble sort focuses on successive adjacent pairs of elements in the array, compares them, and either swaps them or not. In either case, after such a step, the larger of the two elements will be in the higher index position. The focus then moves to the next higher position, and the process is repeated. When the focus reaches the end of the effective array, the largest element will have "bubbled" from whatever its original position to the highest index position in the effective array.

For example, consider the array:

45, **67**, 12, 34, 25, 39

In the first step, the focus is on the first two elements (in **bold**) which are compared and swapped, if necessary. In this case, since the element at index 1 is larger than the one at index 0, no swap takes place.

45, **67**, **12**, 34, 25, 39

Then the focus move to the elements at index 1 and 2 which are compared and swapped, if necessary. In our example, 67 is larger than 12 so the two elements are swapped. The result is that the largest of the first three elements is now at index 2.

45, 12, **67**, **34**, 25, 39

The process is repeated until the focus moves to the end of the array, at which point the largest of all the elements ends up at the highest possible index. The remaining steps and result are:

45, 12, 34, **67**, **25**, 39
45, 12, 34, 25, **67**, **39**
45, 12, 34, 25, 39, 67

The largest element has bubbled to the top index of the array. In general, a bubble step is performed by the loop:

```
for (k = 0; k < eff_size - 1; k++)
     if (x[k] > x[k + 1])
          swaparay(x, k, k + 1);
```

The loop compares all adjacent elements at index k and k + 1. If they are not in the correct order, they are swapped. One complete bubble step moves the largest element to the last position, which is the correct position for that element in the final sorted array. The effective size of the array is reduced by one and the process repeated until the effective size becomes one. Each bubble step moves the largest element in the effective array to the highest index of the effective array.

The code implementing this algorithm is the function, `bubblesort()` shown in Figure 10.7. The function repeats bubble steps, using the function `bubblemax()`, as many times as the size of the array. This function is passed the array name and the size of the effective array. The size of the effective array is the original size reduced by one after each step. Thus, if the initial size of the array to be sorted is `lim`, the size of each successive effective array is `lim`, `lim -1`, `lim - 2`, etc. We have included a debug statement in `bubblesort()` to trace the bubble process after each bubble step. The function, `bubblemax()`, compares adjacent elements of an array of the specified size in sequence and swaps them if necessary. The function is shown in Figure 10.8 together with the function, `swaparay()` to swap elements in an array. All these functions are included in file, `sortsrch.c`, and their prototypes are included in file,`sortsrch.h`, also shown in the Figure. It should be clear that bubble sort is not as efficient as selection sort. There is a great deal of swapping required in bubble sort to "bubble" the largest element to the highest index; where in selection sort, it is done by a single swap. On the other hand, if the data is mostly sorted, then bubble sort can be made more efficient.

```
/* File: sortsrch.c - continued */
/* Sorts an array x of size lim using bubble sort. */
void bubblesort(int x[], int lim)
{    int i;

     for (i = 0; i < lim; i++)
     {    bubblemax(x, lim - i);   /* effective array size is lim - i */
          #ifdef DEBUG             /* debug statement */
               printf("Effective array of size %d: \n", lim - i);
               pr_aray_line(x, lim - i);
          #endif
     }
}
```

Figure 10.7: Code for bubble sort

```
/* File: sortsrch.c - continued */
/* bubbles the next largest element through the array x */
void bubblemax(int x[], int eff_size)
{    int k;

     for (k = 0; k < eff_size - 1; k++)
          if (x[k] > x[k + 1])
               swaparay(x, k, k + 1);
}

/* File: sortsrch.c - continued */
/* swaps elements i and j of array x */
void swaparay(int x[], int i, int j)
{    int temp;

     temp = x[i];
     x[i] = x[j];
     x[j] = temp;
}


/* File: sortsrch.h - continued */
void bubblesort(int x[], int lim);
void bubblemax(int x[], int eff_size);
void swaparay(int x[], int i, int j);
```

Figure 10.8: Code for `bubblemax()`

```
/*   File: bsrtaray.c
     Other Source Files: sortsrch.c
     Header Files: sortsrch.h
     This program uses bubble sort to sort an array of
     integers. It prints the unsorted and the sorted
     arrays. It also prints a trace at each bubble step to
     show the bubble process.
*/

#include <stdio.h>
#define DEBUG
#include "sortsrch.h"

main()
{    int id[] = {45, 67, 12, 34, 25, 39};

     printf("***Bubble Sort***\n\n");
     printf("Unsorted array: \n");
     pr_aray_line(id, 6);

     bubblesort(id, 6);
     printf("Sorted array: \n");
     pr_aray_line(id, 6);
}
```

Figure 10.9: Driver to test bubble sort

To illustrate the operation of bubble sort, we now write a program driver to exercise bubble sort shown in Figure 10.9. It uses `bubblesort()` on the same array used in our search example above. The initialized unsorted array is printed; then the array is sorted and printed. Each bubble step is explicitly shown by a debug statement. Note that `DEBUG` is defined during program development and removed when the program is debugged.

Sample Session:

```
***Bubble Sort***

Unsorted array:
45 67 12 34 25 39

Effective array of size 6:
45 12 34 25 39 67

Effective array of size 5:
12 34 25 39 45
```

```
/*   File: sortsrch.c - continued */
/*   Bubble sort in a single function */
void bsrtfnc(int x[], int lim)
{    int i, k, temp;

     for (i = 0; i < lim; i++) {
          for (k = 0; k < lim - i - 1; k++)

               if (x[k] > x[k + 1]) {
                    temp = x[k];
                    x[k] = x[k+1];
                    x[k+1] = temp;
               }

               #ifdef DEBUG
                    printf("Effective array of size %d: \n", lim - i);
                    pr_aray_line(x, lim - i);
               #endif
     }
}

/* File: sortsrch.h - continued */
void bsrtfnc(int x[], int lim);
```

Figure 10.10: Code for one function bubble sort

```
Effective array of size 4:
12 25 34 39

Effective array of size 3:
12 25 34

Effective array of size 2:
12 25

Effective array of size 1:
12

Sorted array:
12 25 34 39 45 67
```

There are several ways to improve the bubble sort algorithm. First, a single function should incorporate the entire algorithm (Figure 10.10). The time overhead of a function call in a loop can be quite large if the array is large.

Next, a minor point: since an array of one element is already sorted, at most $n - 1$ bubbling

```
/*   File: sortsrch.c - continued */
/*   Bubble sort function which terminates if an array is sorted. */
#include "tfdef.h"       /* defines TRUE and FALSE */
void bsort(int x[], int lim)
{    int i, k, temp, swap = TRUE;

     for (i = 0; swap && i < lim - 1; i++)
     {    swap = FALSE;

          for (k = 0; k < lim - i - 1; k++)
               if (x[k] > x[k + 1])
               {    temp = x[k];
                    x[k] = x[k+1];
                    x[k+1] = temp;
                    swap = TRUE;
               }

               #ifdef DEBUG
                    printf("Effective array of size %d: \n", lim - i);
                    pr_aray_line(x, lim - i);
               #endif
     }
}

/* File: sortsrch.h - continued */
void bsort(int x[], int lim);
```

Figure 10.11: An improved bubble sort

steps are needed for an array of size n. The first for loop need be executed no more than `lim - 1` times. More important, if the entire array is sorted at some time in the process, no further processing is needed. An array is sorted if no elements are swapped in a bubble step. We will use a flag to keep track of any swapping. Figure 10.11 shows the revised code.

We include a file, `tfdef.h`, that defines `TRUE` and `FALSE`. In the function, we use a flag, `swap`, to keep track of any swapping in the bubble step. For each bubble step, we initially assume `swap` is `FALSE`. If there is any swapping in the bubble step, we set the flag to `TRUE`. The sort process repeats as long as swap is `TRUE`. To get the process started, swap is initialized to `TRUE`.

These improvements may be important for large arrays. If an array is sorted after the first few steps, the process can be terminated with a saving in computation time. The program, `bsrtaray.c`, can be modified to use the above `bsort()` function instead of `bubblesort()` function. A sample output of such a modified program is shown below.

Sample Session (Modified bsrtaray.c):

```
***Bubble Sort***
```

```
Unsorted array:
45 67 12 34 25 39

Effective array of size 6:
45 12 34 25 39 67

Effective array of size 5:
12 34 25 39 45

Effective array of size 4:
12 25 34 39

Effective array of size 3:
12 25 34

Sorted array:
12 25 34 39 45 67
```

Note that the process stops as soon as the effective array of size 3 is found to be sorted. If the original data is almost sorted, then bubble sort can be efficient.

10.2.3 Insertion Sort

The two sorting algorithms we have looked at so far are useful when all of the data is already present in an array, and we wish to rearrange it into sorted order. However, if we are reading the data into an array one element at a time, we can take another approach — insert each element into its sorted position in the array as we read it. In this way, we can keep the array in sorted form at all times. This algorithm is called *inertion sort*.

With this idea in mind, Let us see how the algorithm would work. If the array is empty, the first element read is placed at index zero, and the array of one element is sorted. For example, if the first element read is 23, then the array is:

```
23, ?
```

We will use the symbol, ?, to indicate that the rest of the array elements contain garbage. Once the array is partially filled, each element is inserted in the correct sorted position. As each element is read, the array is traversed sequentially to find the correct index location where the new element should be placed. If the position is at the end of the partially filled array, the element is merely placed at the required location. Thus, if the next element read is 35, then the array becomes:

```
23, 35, ?
```

However, if the correct index for the element is somewhere other than at the end, all elements with equal or greater index must be moved over by one position to a higher index. Thus, suppose the next element read is 12. The correct index for this element in the current array is zero. Each element with index zero or greater in the current partial array must be moved by one to the next higher position. To shift the elements, we must first move the last element to its (unused) higher index position, then, the one next to the last, and so on. Each time we move an element we

leave a "hole" so we can move of the adjoining element, and so on. Thus, the sequence of moving elements for our example is:

```
23, 35,  ?, ?
23,  ?, 35, ?
 ?, 23, 35, ?
```

The index zero is now vacant, and the new element, 12, can be put in that position.

```
12, 23, 35, ?
```

The process repeats with each element read in until the end of input. So, if the next element is 47, we would traverse from the beginning of the array until we find larger than 47 or until we reach the end of the filled part of the array. In this case, we reach the end of the array, and insert 47:

```
12, 23, 35, 47, ?
```

Let us develop the algorithm in more detail by observing how we insert a new item, 30. The correct position is found by traversing the partial array as long as the *new item* is greater than the *array element*. In this case, the array traversal stops at index 2, since the element at index 2, namely 35, is grater than the new element, 30. In general, the following loop finds the correct position in an array, `aray`, for the new `item`. Notice we compare the index, `i`, with the variable, `freepos`, whose value is now 4, to know when we have reached the next free position in the array.

```
for (i = 0; i < freepos && item > aray[i]; i++)
     ;
```

When this loop terminates, in our case, the variable, `i`, will be 2. Next, elements from index,`i`(2), to index `freepos`(4) are moved over one position. The highest indexed element must be moved first, then the next highest index, and so on. The following loop moves all elements, with index greater than or equal to i, in a correct order:

```
for (k = freepos; k > i; k--)
     aray[k] = aray[k - 1];
```

When this loop terminates, the loop counter, `k`, will be equal to `i`, which is the index of the "hole" created in the array:

```
12, 23, ?, 35, 47
```

Finally, the new `item` can be inserted at index, `i`. Figure 10.12 shows the complete function for inserting one new element in a sorted array, given the array, the new item, and the next free position (which, incidentally, is the current size of the array).

The function traverses the partial array until it finds either that `item` is less than or equal to the array element or that the array is exhausted. If the array is exhausted, the second loop is not executed since `i == freepos`. In this case, the item is merely inserted at the correct position. Otherwise, elements at and above index, `i`, are moved over one position, and the new element is inserted at the correct index.

We are now ready to implement insertion sort. The program logic is simple:

```
/*   File: sortsrch.h - continued */
void insert_sorted(int aray[], int item, int freepos);

/*   File: sortsrch.c - continued */
/*   Function inserts item in sorted order in array aray. Freepos
     is the next free pos. in the array.
*/
void insert_sorted(int aray[], int item, int freepos)
{    int i, k;

     i = 0;
     /* find the correct pos. */
     for (i = 0; i < freepos && item > aray[i]; i++)
          ;

     for (k = freepos; k > i; k--)    /* move elements */
          aray[k] = aray[k - 1];
     aray[i] = item;                  /* insert new item */
}
```

Figure 10.12: Code for inserting and element

```
Repeat the following until end of input:
     read a number,
     insert the number read into the array in sorted order;
     if the array is full, break out of loop.
```

The program terminates after a printing of the sorted array. The program uses the above function, `insert_sorted()`, to insert each number in sorted order into the array, and a function, `pr_aray_line()` of Figure 10.5, to print the array. These functions are included in file, `sortsrch.c`. The program driver is shown in Figure 10.13. Notice, we increment the number of elements in the array in each call to `insert_sorted()`, since we have added a new element to the array. We have included a debug statement to print out the partial array at each step. The input is terminated either when an end of file is reached or when the array becomes full.

Sample Session:

```
***Insertion Sort***

Type numbers to be sorted, EOF to quit
23

23
12

12 23
```

```
/*   File: insort.c
     Other Source Files: sortsrch.c
     Header Files: sortsrch.h
     Program uses input to fill a float array in sorted order.
*/

#include <stdio.h>
#define MAX 100
#define DEBUG
#include "sortsrch.h"

main()
{    int x, y[MAX],
               k;        /* no. of items in an array */

     printf("***Insertion Sort***\n\n");
     printf("Type numbers to be sorted, EOF to quit\n");
     k = 0;

     while (scanf("%d", &x) != EOF) {
          insert_sorted(y, x, k++);

          if (k == MAX) {
               printf("Array full\n");
               break;
          }

          #ifdef DEBUG
               pr_aray_line(y, k);
          #endif

     }
     printf("SORTED ARRAY\n");
     pr_aray_line(y, k);
}
```

Figure 10.13: Driver for Insertion Sort

```
35

12 23 35
30

12 23 30 35
47

12 23 30 35 47
10

10 12 23 30 35 47
^D

SORTED ARRAY
10 12 23 30 35 47
```

Insertion sort can be adapted to sorting an existing array. Each step works with a sub-array whose effective size increases from two to the size of the array. The element at the highest index in the sub-array is inserted into the current sub-array, the effective size is increased, etc. (see Problem 5).

10.3 Binary Search

As we saw earlier, the linear search algorithm is simple and convenient for small problems, but if the array is large and/or requires many repeated searches, it makes good sense to have a more efficient algorithm for searching an array. Now that we know how to sort the elements of an array, we can make use of that ordering to make our search more efficient. In this section, we will present and implement the *binary search* algorithm, a relatively simple and efficient algorithm.

The algorithm is easily explained in terms of searching a dictionary for a word. In a dictionary, words are sorted alphabetically. For simplicity, let us assume there is only one page for all words starting with each letter. Let us assume we wish to search for a word starting with some particular letter.

We open the dictionary at some midway page, let us say a page on which words start with M. If the value of our letter is M, then we have found what we are looking for and the word is on the current page. If the value of our letter is less than M, we know that the word would be found in the first half of the book, i.e. we should search for the word in the pages preceding the current page. If the value of our letter is greater than M, we should search the pages following the current page. In either case, the effective size of the dictionary to be searched is reduced to about half the original size. We repeat the process in the appropriate half, opening to somewhere in the middle of that and checking again. As the process is repeated, the effective size of the dictionary to be searched reduces by about half at each step until the word is found on a current page.

Binary search essentially follows this approach. For example, given a sorted array of items, say:

```
12, 29, 30, 32, 35, 49
```

```
/*   File: sortsrch.c - continued */
/*   Function uses binary search to search for item in the array y[]. */
int binsrch(int y[], int lim, int key)
{    int low, mid, high = lim - 1;

     low = 0;

     while (low <= high) {          /* Is the array exhausted?   */
          mid = (low + high) / 2;   /* If not, find middle index */

          if (key == y[mid])        /* Is the key here?          */
               return(mid);         /* If so, return index.      */

          else if (key < y[mid])    /* else if key is smaller,   */
               high = mid - 1;      /* reduce the high end;      */

          else
               low = mid + 1;       /* otherwise, increase low   */
     }
     return(-1);                    /* Not found, return -1      */
}
```

Figure 10.14: Code for Binary Search

suppose we wish to search for the position of an element equal to `x`. We will search the array which begins at some `low` index and ends at some `high` index. In our case the low index of the effective array to be searched is zero and the high index is 5. We can find the approximate midway index by integer division `(low + high) / 2`, i.e. 2. We compare our value, `x` with the element at index 2. If they are equal, we have found what we were looking for; the index is 2. Otherwise, ff `x` is greater then the item at this index, our new effective search array has a `low` index value of 3 and the high index remains unchanged at 5. If `x` is less than the element, the new effective search array has a `high` index of 1 and the low index remains at zero. The process repeats until the item is found, or there are no elements in the effective search array. The terminating condition is found when the `low` index exceeds the `high` index. The algorithm is implemented as a function in Figure 10.14.

We use the `binsrch()` function in an example program which repeatedly searches for numbers input by the user. For each number, it either gives the index where it is found or prints a message if it is not found. An array in sorted form is initialized in the declaration. The code for this driver is shown in Figure 10.15

Sample Session:

```
***Binary Search***

The array is:
12 29 30 32 35 49
```

```
/*   File: bsrcharay.c
     Other Source Files: sortsrch.c
     Header Files: sortsrch.h
     Program uses binary search to search a sorted array of numbers.
*/

#include <stdio.h>
#define MAX 100
#define DEBUG
#include "sortsrch.h"

main()
{    int i, x, y[MAX] = {12, 29, 30, 32, 35, 49};
     int k = 6;          /* no. of items in the array y[] */

     printf("***Binary Search***\n\n");
     printf("The array is:\n");
     pr_aray_line(y, k);
     printf("Type a number, EOF to quit: ");

     while (scanf("%d", &x) != EOF) {
          i = binsrch(y, k, x);

          if (i >= 0)
               printf("%d found at array index %d\n", x, i);
          else
               printf("%d not found in array\n", x);

          printf("Type a number, EOF to quit: ");
     }
}
```

Figure 10.15: Test Driver for Binary Search

```
Type a number, EOF to quit:  12
12 found at array index 0

Type a number, EOF to quit:  23
23 not found in array

Type a number, EOF to quit:  34
34 not found in array

Type a number, EOF to quit:  45
45 not found in array

Type a number, EOF to quit:  30
30 found at array index 2

Type a number, EOF to quit:  29
29 found at array index 1

Type a number, EOF to quit:  ^D
```

10.4 An Example — Payroll Data Records

So far, in the previous sections, we have seen how to search and sort an array of integers. In this section we apply the sort and search methods to our database of payroll records. The data items in a payroll record are id number, hours worked, rate of pay, regular and overtime pay. Our task is to write an interactive program which displays the pay record for a given individual.

We saw how we could implement such a database in Chapter 7. There, the data record for a specific id is stored at the same index in several different arrays as shown in Figure 10.16. In our application, we will search the database to find the payroll record given a specific id number as the key. Therefore, we will need to sort the database by the id number field. When we search for the key, we will get the index for the element, if any, which matches the sought after id. With that index in id number array, we can access the remaining information for that data record.

As we saw in the previous section, when we sort an array, we rearrange the positions of the array elements. When we sort data records, we must rearrange the positions of all fields of the data records; i.e. if a data record is spread over several arrays, we must rearrange the elements of all of these arrays in an identical manner. In this way, we will still be able to access a data record using the index determined by a key. To sort the database, we can use either selection sort or bubble sort for our task. We will assume that the input data is mostly sorted, requiring little rearrangement, and therefore will choose to use bubble sort. This is a reasonable assumption since records are usually kept in a file in sorted order. Only new records entered may be out of place. We will modify `bubblesort()` of the last section to handle data records.

The input data record is spread over three arrays; namely, `id[]`, `hrs[]`, and `rate[]`. Since we are sorting records by id numbers, the decision whether to swap records is determined by the elements of the `id[]` array; however, if we swap elements of `id[]`, we must also swap corresponding

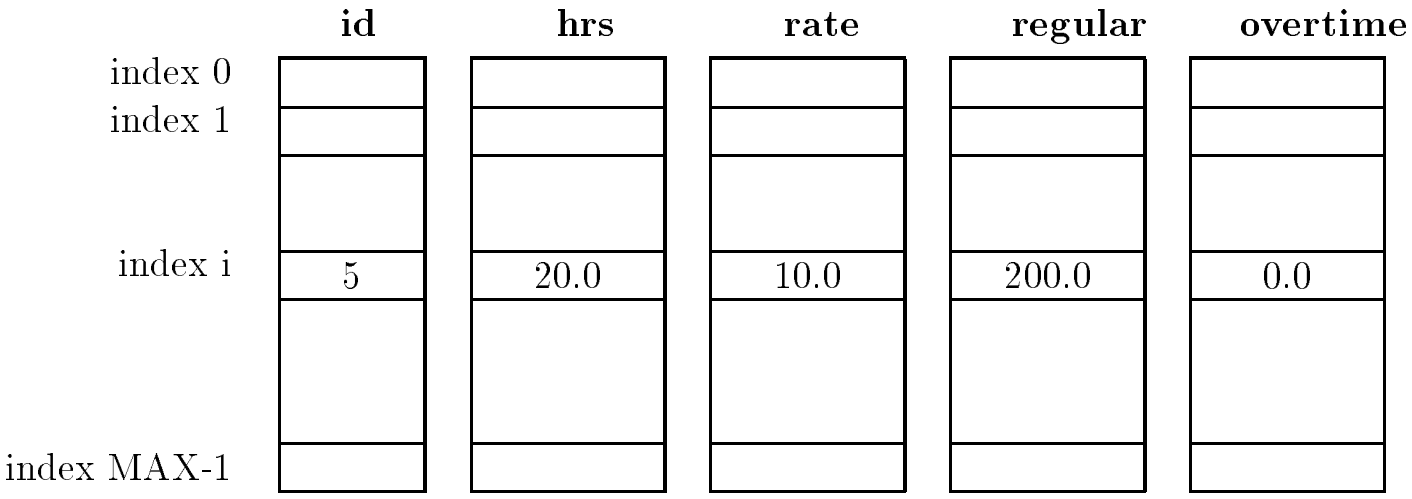

Figure 10.16: A Data Record Across Arrays

elements of the other two arrays.

The code for the modified sort function, called `sortdata()`, that sorts input payroll records is shown in Figure 10.17. We write the code in the file `payutil.c` and add its prototype in `payutil.h`. These files have other payroll functions and prototypes developed in Chapter 7, including: `getdata()` which reads the input data, `calcpay()` which calculates regular and overtime pay, and `printdata()` which prints the pay records in a table. We also use the file `tfdef.h` that defines TRUE and FALSE.

We can now use the above function in a payroll program that sorts the input data before processing it. The main purpose of this program is to test the operation of creating a sorted database of records before later modifications to the program. The driver is very simple and consists of functions that get data, sort data, calculate pay, and print data as seen in Figure 10.18. Notice we have performed the calculate pay step after the database has been sorted, as the arrays containing this data are not rearranged by our sort function. The sample session is shown below.

Sample Session:

```
***Payroll Program - Sorted Data***

ID <zero to quit>:  2
Hours Worked:  20
Rate of Pay:  5

ID <zero to quit>:  5
Hours Worked:  40
Rate of Pay:  10

ID <zero to quit>:  12
Hours Worked:  45
Rate of Pay:  12.50
```

```
/* File: payutil.c - continued */
#include "tfdef.h"
void sortdata(int id[], float hrs[], float rate[], int lim)
{    int i, k, temp, swap = TRUE;
     float ftmp;

     for (i = 0; swap && i < lim - 1; i++) {
          swap = FALSE;

          for (k = 0; k < lim - i - 1; k++)
               if (id[k] > id[k + 1]) {
                    temp = id[k];
                    id[k] = id[k + 1];
                    id[k + 1] = temp;

                    ftmp = hrs[k];
                    hrs[k] = hrs[k + 1];
                    hrs[k + 1] = ftmp;

                    ftmp = rate[k];
                    rate[k] = rate[k + 1];
                    rate[k + 1] = ftmp;
                    swap = TRUE;
               }
     }
}

/* File: payutil.h - continued */
void sortdata(int id[], float hrs[], float rate[], int lim);
```

Figure 10.17: Code for `sortdata()` and header file entry

```
/*    File: paysrt.c
      Other Source Files: payutil.c
      Header Files: payutil.h
      Program calculates payroll data for a number of id's. It
      gets data, sorts data, calculates pay, and prints data for
      all id's.
*/

#include <stdio.h>
#include "payutil.h"
#define MAX 10

main()
{     int i, n = 0, key, id[MAX];
      float hrs[MAX], rate[MAX], regpay[MAX], overpay[MAX];

      printf("***Payroll Program - Sorted Data***\n\n");
      n = getdata(id, hrs, rate, MAX);
      sortdata(id, hrs, rate, n);
      calcpay(hrs, rate, regpay, overpay, n);
      printdata(id, hrs, rate, regpay, overpay, n);
}
```

Figure 10.18: Test driver for `sortdata()`

```
ID <zero to quit>:  8
Hours Worked:  50
Rate of Pay:  14

ID <zero to quit>:  0

***PAYROLL: FINAL REPORT***
 ID    HRS   RATE     REG    OVER     TOT

  2  20.00   5.00  100.00    0.00  100.00
  5  40.00  10.00  400.00    0.00  400.00
  8  50.00  14.00  560.00  210.00  770.00
 12  45.00  12.50  500.00   93.75  593.75
```

The program file, `paysrt.c`, containing the driver, and the source file, `payutil.c`, with the new function, `sortdata()` added, must be compiled and linked. Observe that the input is almost sorted by id number; only the last record is out of place. Bubble sort can sort this data in one pass.

Having observed that the database is correctly sorted, we can now complete the task to search for a specific record to display. We will use binary search on the data in sorted order. The search for an id number in the array, `id[]`, returns an index if it is found, and returns -1 otherwise. If the index is non-negative, the same index is used to access the rest of the data record spread over the other arrays. We modify the above test program to read the data records and calculate the pay, then repeatedly call `binsrch()` to return the index of a data record for a specified id number. If a data record exists, it is printed by `printrec()`. We have already implemented the function `binsrch()` and included it in the file, `sortsrch.c`. We will soon write `printrec()`. The program that implements our task is shown in Figure 10.19.

The first part of the program reads data, sorts data, and calculates pay. The second part of the program reads an id number and calls `binsrch()` to locate its index in the array. If the index is non-negative, the program uses a function, `printrec()`, to print a data record at that index. If the index is negative, the program prints an error message. The function, `printrec()`, is shown in Figure 10.20 and added to the file, `payutil.c`.

A sample session for the search part of the program is shown below. The input data is assumed identical to that in the sample session for the previous program `paysrt.c`.

```
***Payroll Program - Search Data***

Type an id <zero to quit>:  8
***PAYROLL RECORD FOR ID 8***

 ID    HRS   RATE     REG    OVER     TOT
  8  50.00  14.00  560.00  210.00  770.00

Type an id <zero to quit>:  10
Error - no such id

Type an id <zero to quit>:  2
***PAYROLL RECORD FOR ID 2***
```

```
/*   File: paysrch.c
     Other Source Files: payutil.c, sortsrch.c
     Header Files: payutil.h, sortsrch.h
     Program sorts and calculates payroll data for a number of id's. It
     then uses sequential search to find and print data records for
     specified id numbers.
*/

#include <stdio.h>
#include "payutil.h"
#include "sortsrch.h"
#define MAX 10

main()
{    int i, n = 0, key, id[MAX];
     float hrs[MAX], rate[MAX], regpay[MAX], overpay[MAX];

     printf("***Payroll Program - Search Data***\n\n");
     n = getdata(id, hrs, rate, MAX);
     sortdata(id, hrs, rate, n);
     calcpay(hrs, rate, regpay, overpay, n);
     printdata(id, hrs, rate, regpay, overpay, n);

     printf("Type an id <zero to quit>: ");
     while (scanf("%d", &key) != EOF && key != 0) {
          i = binsrch(id, n, key);

          if (i >= 0)
               printrec(id, hrs, rate, regpay, overpay, i);
          else printf("Error - no such id\n");

          printf("Type an id <zero to quit>: ");
     }
}
```

Figure 10.19: Code for searching the database

```
/* File: payutil.c - continued */
/* Function prints a single data record at a specified index. */
void printrec(int id[], float hrs[], float rate[],
              float reg[], float over[], int i)
{
     printf("***PAYROLL RECORD FOR ID %d***\n\n", id[i]);
     printf("%10s%10s%10s%10s%10s%10s\n", "ID", "HRS",
              "RATE", "REG", "OVER", "TOT");

     printf("%10d%10.2f%10.2f%10.2f%10.2f%10.2f\n",
              id[i], hrs[i], rate[i], reg[i], over[i],
              reg[i] + over[i]);
}

/* File: payutil.h - continued */
void printrec(int id[], float hrs[], float rate[],
              float reg[], float over[], int i);
```

Figure 10.20: Code for `printrec()` and header file entry

```
 ID    HRS  RATE     REG  OVER     TOT
  2  20.00  5.00  100.00  0.00  100.00
Type an id <zero to quit>:  0
```

10.5 Polymorphic Data Type

Very often in programs, a generic operation must be performed on data of different types. For example, in our bubble sort algorithm for the payroll records, when elements were found out of order in the `id[]` array, we needed to swap the integer elements in that array as well as the float elements in the `hrs[]` and `rate[]` arrays. If we decided to implement this swapping operation as a function, we would need to write two functions: one to swap integers, and another to swap floating point values; even though the algorithm for swapping is the same in both cases. (We wrote a swap function for integers using pointers in Chapter 6).

The C language provides a mechanism which allows us to write a single swapping function which can be used on any data type. This mechanism is called a *polymorphic data type*, i.e. a data type which can be transformed to any distinct data type as required. An item of polymorphic data type is created by the use of a *generic pointer*. A generic pointer is simply a byte address without an associated type. In other words, a generic pointer does not point to an object of a specific type; it just points to some location in the memory of the computer. In ANSI C, a generic pointer is declared as a void pointer (in old C, a generic pointer is a char pointer). It is only when the actual operations must be performed on the data that generic pointers are cast to pointers to specific types and dereferenced.

Using the concept of a generic pointer, we can assume the following prototype for a function

```
/* File: payutil.c - modified */
#include "tfdef.h"
void sortdata(int id[], float hrs[], float rate[], int lim)
{    int i, k, temp, swap = TRUE;
     float ftmp;

     for (i = 0; swap && i < lim - 1; i++) {
          swap = FALSE;

          for (k = 0; k < lim - i - 1; k++)
               if (id[k] > id[k + 1]) {
                 gen_swap((void *)(id + k), (void *)(id + k + 1), 'd');
                 gen_swap((void *)(hrs + k), (void *)(hrs + k + 1), 'f');
                 gen_swap((void *)(rate + k), (void *)(rate + k + 1), 'f');

                 swap = TRUE;
               }
     }
}
```

Figure 10.21: Modified code for `sortdata()` using generic swap

to swap two data items of any type:

```
void gen_swap(void * x, void * y, char type);
```

Here, `x` and `y` are generic pointers to two data items, and `type` specifies the type of the data using a single character. With this information, we can now rewrite the function, `sortdata()`, in the file, `payutil.c` using `gen_swap` to swap all data items. The code is shown in Figure 10.21. Notice in the calls to `gen_swap()` we cast the pointers to the integer array elements (`id + k` and `id + k + 1`) to void pointers. Similarly, the pointers to the float data items in the `hrs[]` and `rate[]` arrays are cast to void pointers. We pass the character constants `'d'` for integer, or `'f'` for float to tell `gen_swap()` the type of the data it is to swap.

We can now write the code for `gen_swap()` as seen in Figure 10.22. We have declared two temporary variables, `temp` and `ftmp` to hold an integer or float value, respectively when we do the swapping. The variable, `type` is used to switch to the appropriate code sequence to swap the two data items.

If we made these modifications to `payutil.c` and recompiled our program, it would behave exactly as it did in the last section. Of course, as we stated earlier, we may not want to use a function to perform the swap in bubble sort because of the overhead in calling and returning from a function. As another example of the use of the polymorphic data type, consider writing a function that will print an array, regardless of type, with five elements per line. We may have an array of integers and an array of floats to be printed and wish to use the same function to format the lines of output. Figure 10.23 shows a driver program and the function, `praray()`.

The function calls to `praray()` pass the array pointers after first casting them to generic pointer types. In the function, `praray()`, we use the array index to determine when a new line is

```
/* File: payutil.c - continued */
void gen_swap(void * x, void * y, char type)
{  int temp;
   float ftmp;

   switch(type)
   { case 'd' : temp = *(int *)x;
                *(int *)x = *(int *)y;
                *(int *)y = temp;
                return;
     case 'f' : ftmp = *(float *)x;
                *(float *)x = *(float *)y;
                *(float *)y = ftmp;
                return;
     default  : printf("Error in gen_swap: %c not a legal type\n",type);
   }
}
```

Figure 10.22: Code for `gen_swap`

needed. Since we wish to print five elements to a line, a newline is printed every time the index, `i`, is a multiple of 6, i.e. `i % 6` is zero. When the index `i` is zero, no newline is needed.

The void pointer, `y`, points to the array and the `type` value is a character, `'d'` for integers, and `'f'` for float, as before. Each element of the array is printed by means of a `switch` statement. The switch cases are selected by the type of the array passed. If the type is `'d'`, a decimal integer is printed; if the type is `'f'`, a float is printed. If desired, the function can be extended to handle other types as well. Let us examine the printing of an i^{th} element of an integer array. For a type `'d'`, the argument expression in printf() is:

```
*((int *) y + i)
```

The void pointer, `y`, is first cast to the desired type, i.e. `int *`; then, the `int *` is increased by `i` so as to point to the i^{th} element of an integer array. This pointer is finally dereferenced to access the i^{th} element of the array. Thus, `printf()` prints the value of the i^{th} element of an integer array. Similarly, a float array element is printed out by first casting the generic pointer to a float pointer. A sample output is shown below.

```
***Generic Pointers and Polymorphic Data Types***

Integer array is:
12 23 34 45 72

Float array is:
12.240000 23.350000 43.570000 82.209999
```

Use of polymorphic data types makes for compact programs; however, their use is not recommended for beginning programmers. For the most part, we will not use them in this text.

```
/*   File: genptr.c
     Program shows the use of generic pointers to implement a
     polymorphic data type. An integer and a float array are printed
     out by the same primitive function praray().
*/
#include <stdio.h>
void praray(void * y, int lim, char type);

main()
{    int x[] = {12, 23, 34 ,45, 72};
     float y[] = {12.24, 23.35, 43.57, 82.21};

     printf("***Generic Pointers and Polymorphic Data Types***\n\n");
     printf("Integer array is:\n");
     praray((void *) x, 5, 'd');
     printf("Float array is:\n");
     praray((void *) y, 4, 'f');
}

/*   Function prints an array of any type, int or float. */
void praray(void * y, int lim, char type)
{    int i;

     for (i = 0; i < lim; i++) {
          if (i != 0 && i % 6 == 0)     /* add a newline every 6th item */
               printf("\n");

          switch(type) {
               case 'd': printf("%d ", *((int *) y + i));
                         break;
               case 'f': printf("%f ", *((float *) y + i));
                         break;
               default: printf("Error in printing array\n");
          }

     }
     printf("\n");
}
```

Figure 10.23: Code for printing arbitrary arrays

10.6 Common Errors

1. In insertion sort, the elements are shifted incorrectly. Shift the highest index element first, then the next highest, and so forth.

2. The argument in binary search that specifies the high index is incorrect. If the size of the array is passed as the highest index, there is a problem. If the size of the array is n, the index n is outside the array. The argument should be n - 1.

3. Generic pointers should be used with care. In traditional C, use char pointer instead of void pointer.

10.7 Summary

In this chapter we have developed algorithms for searching a collection of data for a specific element, called the *key*. We saw a simple algorithm, *linear search* which started at the beginning of the data, and compared each element against the key until it was found, or the data was exhausted. However, linear search is not very efficient if the number of elements to search is large. In order to develop more efficient algorithms, we need to take advantage of the order of the data. Therefore, we next discussed how we can arrange the elements in an array in a specified order — a process called *sorting*. We developed three sorting algorithms: *selection sort*, *bubble sort*, and *insertion sort*. The first two of these are useful when all of the data is already stored in an array, and insertion sort can be used to sort the data as it is being read into the array.

Once we have the data sorted, we developed a more efficient searching algorithm — *binary search*. This algorithm worked by dividing the data in half, and deciding in which half the key would occur. With each step, then, we can eliminate half of the data from further consideration.

These searching and sorting techniques are general and may be applied to any type of data. We used them in our payroll task to find individual payroll records in a database given an id as the key.

Finally, we discussed the use of the polymorphic data type, or *generic pointers* to implement a common operation that may be applied to data of different types.

10.8 Exercises

Find and correct errors if any.

1.
```
main()
{    int x[10];

     x[10] = {12, 23, 45};
}
```

2.
```
main()
{    int x[10];

     x = {12, 23, 45};
}
```

3.
```
main()
{    int i, x[10];

     for (i = 0; i < 10; i++)
          x = 0;
}
```

4. Should you use a function or a macro to swap values in bubble sort? Explain your reasons.

5. Bubble sort moves the largest value to the highest index. Modify the bubble sort code to move the smallest element to the lowest index.

6. Insertion sort inserts a new element into the array. Modify the insertion sort method to apply it to an unsorted array with n elements. Do not use another array.

7. Modify the bubble sort to apply it to an array of characters housing a string. The number of elements in the string are unknown, but terminated by a NULL.

10.9 Problems

1. Write a function that sorts an array of integers in decreasing order.

2. Write a function that sorts an array of integers in either increasing or in decreasing order as specified by an argument.

3. Write a binary search function that searches an array of integers sorted in decreasing order.

4. Write a binary search function that searches an array of integers sorted in either increasing or in decreasing order as specified by an argument.

5. Write a function that uses insertion sort to sort an array of input numbers, either in increasing or in decreasing order.

6. Write a function that uses insertion sort to sort an existing array of integers.

7. Write a program to read an array of integers from a file. Write a function that takes two arguments: low and high. Low and high specify the low and high indices of an effective array. Function finds indices for the maximum and the minimum elements in the specified effective array. Use the function with zero for low and the highest valid index for high. Print the values of maximum and minimum.

8. Repeat the last problem, but this time swap the largest element in the effective array with the one at the high index, say index n; swap the smallest element with the one at the low index.

9. Repeat the last problem, but this time after the swap of the elements change the effective array so low is 1 and high is n - 1. Repeat the process so the largest and smallest elements are found in the array from index 1 through n - 1. Swap the next largest element with the one at index n - 1, and swap the next smallest with the one at index 1. The next swap considers the array from index 2 through n - 2, etc until all elements of the array are in increasing order. This is another way of sorting an array.

10. Compare the operations involved in the above sorting with that for bubble sort. What are the approximate comparisons required to sort an array of n items by the two methods.

11. Write a menu_driven program that allows the following commands: get data from a file, add data, delete data, sort data, search data, save data to a file, help, quit. Assume that data records consist of id numbers and exam scores. Sorting must be done either by id numbers or by exam scores and it must be either in ascending or descending order.

12. Repeat the last problem, but get data uses insertion sort to read data in sorted form by id numbers.

13. Write a program that reads integers into an array A. Use another array P of the same size to store each index of the array A in the following way. The index in A with the smallest element is stored at index 0 of P, the index of the next smallest element in A is stored at index 1 of P, and so on. Print the array A, and print the elements of A ordered in the sequence given by each succeeding index stored in P.

14. Repeat Problem 11, but use the approach of Problem 13 to sort the data.

15. Repeat Problem 11, but use insertion sort to read data in sorted form by id numbers. The sort command then uses approach of Problem 13 to sort the data by exam scores.

16. Compare the operations required for sorting in Problems 11 and 14.

17. Write a program to merge two sorted arrays A and B into a third array S as follows. Start with initial index, ia, of the element to be merged from A and also the index, ib, of the element from B. If the element from A is smaller than the element from B, append that element of A to the array S and increment ia; if the element of B is smaller than that of A, add the element from B to S and increment ib. If they are equal, add both elements to S and increment both ia, ib. If either array is exhausted, copy the elments from the other array into S. Repeat until both arrays are exhausted. Print A, B, and S.

18. Develop a sort method using the merging of two arrays as in the last problem. Given an array with 8 elements, assume it is split into as many arrays as there are elements. Merge each adjacent pair into 4 arrays of two elements each. Merge each pair again into 2 arrays of 4 elements each. Finally, merge the two into a sorted array. The method can be applied to any array and is called merge sort method. Write a program that sorts an array by merge sort.

19. Write a program that sorts the characters in a string.

20. Write a program that reads strings; for each string compute the frequency of occurrence of each character.

21. Write a program that reads text from a file and computes cumulative frequency of occurrence of each character.

22. Write a program that searches a string for a specified character and returns the index of its first occurrence.

23. Write a program that returns the first occurrence of a character in a string starting at some specified index.

24. Write a program to find all occurrences of a character in a string.

25. Write a program that replaces all the occurrences of a character in a string by another character.

26. Write a program that sorts characters in a string according to a different order than that of the ASCII values. Use a function to compare two characters and return whether one is greater than, equal to, or less than the other. The function first converts all lower case letters to upper case and then compares their ASCII values. The program sorts characters ignoring case.

27. Write a sort program similar to the above problem for integers. Use a function that compares the absolute values of two integers. The program sorts by absolute values.

28. Write a character sort program that uses a function, `cmpchrs()`, to compare two characters by an arbitrary criteria. Assume that an array stores all ASCII printable characters in an assumed increasing order: vowels first, consonants next, digits next, all others next. The function, `cmpchrs()`, looks up the corresponding indices of characters to compare characters. The result of comparing the indices is returned by the function. The program sorts characters in an order specified by `cmpchrs()`.

Chapter 11

String Processing

These days, computers are not used only for processing numbers, but, as we have seen in previous chapters, they are also used for processing textual data. As we saw in Chapter 7, in C, textual data is represented using arrays of characters called a *string*. If we are to manipulate strings in any reasonable way we must have several basic operations available to us. Since string is not a basic type in C, string operations must be developed as functions. A library of such functions can then be used as a part of the language. In other words, we can effectively treat string as an *abstract data type* [1] once we have string operations written in function form. The Standard Library provides a rich set of functions for performing operations on strings such as copying and comparing them, breaking strings up into parts, joining them, finding substrings, substituting one string for another, and so forth. In this chapter, we will discuss such string processing using the built-in library functions as well as look at how some of these functions can be written.

We begin the chapter by defining a user defined data type for strings and then use this new type throughout. We then describe the library functions available for performing string operations, including reading and writing strings, copying a string and finding its length, comparing and joining strings and converting the information in a string to other data types. We conclude the chapter with several example programs using these string operations.

11.1 The Data Type `STRING`

Because a string is such a common data structure in programs, it may be convenient to define it as its own data type. We can then define functions to perform operations on operands of the defined data type; effectively treating the defined type as a new data type (an abstract data type). As we have seen previously, a string is implemented as an array of characters, and an array is implemented as a contiguous collection of cells and a *pointer* pointing to the beginning of the block. The name of the array is associated with this pointer cell, rather than with the data cells themselves. When we pass an array to a function, we pass this pointer to the array. So when we are processing strings, passing them to functions, returning them as values, we are handling pointer values. Therefore, we can define a data type, `STRING`, as a pointer to a character as follows:

```
typedef char * STRING;
```

[1] To define an abstract data type, we must define a way to declare variables of that type together with operations that can be performed on such data items. A full description of the concept of abstract data types is beyond the scope of this text; however, the basic idea is presented here.

We can then define string variables in terms of the data type, `STRING`:

```
STRING s, t;
```

The variables, `s` and `t`, are character pointers. We can access the characters in the string by dereferencing the pointer or using array type indexing. But remember, declaring a pointer type only allocates space for the pointer; it does not allocate cells for an array; it does not initialize the pointer. We cannot use a `STRING` variable to store a string of characters, but merely to point to a pre-allocated string. We must always declare a character array to store a string of characters, and can then initialize a `STRING` variable to point to this array. `STRING` does not serve to allocate memory for a string. As such, the concept of abstract data type is not totally satisfied by the above type definition; however, with the above caveat, we can otherwise treat it as such.

We illustrate the use of the `STRING` type by writing a rogram to read and print a string as shown in Figure 11.1. Note, in `main()`, a character array, `s`, is declared. The name of the array, `s`, is an (initialized) character pointer — the same as our type, `STRING`, and may therefore be passed to the function `our_strprint()` which expects a `STRING` argument. Notice, we have placed the `typedef` for `STRING` in a header file, `strtype.h` since it will be useful for other programs we will write.

11.2 Library String Functions

With a data type defined, we may now proceed to define functions to implement the operations on data of this type. As mentioned above, the C built-in library provides a rich set of string processing functions. We describe some of the more common ones here; others are described in Appendix C.

11.2.1 String I/O: *gets()* and *puts()*

One of the first operations we may need for strings is the ability to read or write strings from the standard input or to the standard output. For example, if we had a task:

STR0: Read strings until end of file, convert each string to upper case, and print the modified string.

we could easily write an algorithm for the task:

```
while not EOF, read a string
     convert string to upper case
     print string
```

To implement this algorithm, we need three functions: read a string from standard input stream, write a string to standard output, and convert a string to upper case. We have shown crude versions of these first two functions in the previous, section; however, the standard library provides these operations as well. Library function, `gets(s)` reads a string from the standard input into an array pointed to by `s`; and, `puts(s)` writes a string pointed to by `s` to the standard output. The prototypes for these functions, declared in `stdio.h`, are: .

```
STRING  gets(STRING  string);
int puts(STRING  string);
```