```
/*  File: strtype.c
    This program illustrates the use of a type definition for strings.
*/
#include <stdio.h>
#include "strtype.h"
#define SIZE 100
void our_strprint(STRING s);
void our_strread(STRING s);
main()
{   char s[SIZE];       /* allocate space for a string */

    our_strread(s);             /* read a string */
    our_strprint(s);    /* print a string */
}

/*  Function reads a string from standard input.*/
void our_strread(STRING s)  /* declare a STRING type */
{
    while ((*s = getchar()) != '\n')
         s++;
    *s = NULL;
}

/*  Function writes a string to standard output.*/
void our_strprint(STRING s)  /* declare a STRING type */
{
    while (*s) {
         printf("%c", *s);
         s++;
    }
    printf("\n");
}

/*  File: strtype.h
    This file contains the definition of type STRING
*/

typedef char * STRING;
```

Figure 11.1: Program illustrating the `STRING` data type

If reading is successful, `gets()` returns the pointer to the string; otherwise, it returns a `NULL pointer`, i.e. a pointer whose value is zero. A returned value of `NULL` usually implies an end of file. When `gets()` reads a string, it reads the input characters until a newline is read, discards the newline, appends a `NULL` character to the string, and stores the string where `s` points. Similarly, `puts()` outputs the string, `s`, after stripping the `NULL` and appending a newline. It returns the last character value output if successful; otherwise it returns `EOF`. Note, the arguments to these functions and the return value from `gets()` are character pointers, i.e. equivalent to our `STRING` data type, and we can consider them as such. The argument of `gets()` **MUST** be a string; otherwise, the function attempts to store characters wherever the argument points, which can create a possibly fatal error when the program executes.

We will write and use a function, `ucstr()`, which converts a string to upper case. The whole program is simple: it reads a string, converts it to upper case, and prints it; and is shown in Figure 11.2 In the driver, the loop expression reads a line into `s`; if successful, the returned value is a non-zero pointer, `s`, and the loop is executed. In the loop body, the string, `s`, is converted to upper case, and printed. The function, `ucstr()`, converts a string to upper case by traversing the string and converting each character to upper case using library routine, `toupper()`, which returns the upper case version of its argument if it is a lower case letter; otherwise it returns the argument unchanged.

Sample Session:

```
***String to Upper Case***

Type strings, EOF to terminate
Hello
HELLO
Pad 19A
PAD 19A
good morning
GOOD MORNING
^D
```

The above program reads lines until end of file. As a slight variation on this task, sometimes it is desirable to loop until a blank line is entered. Here is a loop that copies lines until a blank line is entered:

```
while (*gets(s))
    puts(s);
```

Assuming that a line is read successfully, `gets()` returns `s`. The expression, `*gets()`, is the same as `*s`, which is the first character in the string, `s`. As long as the first character of `s` has a non-zero value, the loop continues. When the first character is a `NULL`, the loop terminates. If a blank line is entered by typing a *RETURN*, `gets()` reads an empty string and the loop terminates.

We can also use `gets()` in a menu driven program which requires the user to enter either a single character or a command line. In our previous menu driven programs in Chapter 4, we saw that reading a single command character required that the keyboard buffer be flushed of the newline character before reading the next command. If only one character is to be read, or if the first character of a command line is sufficient to identify a command, then it is simpler to read the

```
/*  FIle: ucstr.c
    This program reads strings, converts them to upper case, and
    prints them out.
*/

#include <stdio.h>
#include <ctype.h>                /* includes toupper() */
#include "strtype.h"
#define SIZE 100
void ucstr(STRING t);

main()
{
    char s[SIZE];     /* allocate a string  */

    printf("***String to Upper Case***\n\n");
    printf("Type strings, EOF to terminate\n");

    while (gets(s)) {
         ucstr(s);
         puts(s);
    }
}

/* Converts t to upper case string */
void ucstr(STRING t)
{

    while (*t) {                  /* loop until char is null */
         *t = toupper(*t);        /* convert char *t to upper case */
         t++;                     /* point to next char */
    }

}
```

Figure 11.2: Program to read and print strings using `gets()` and `puts()`

entire line using `gets()`, which strips the newline character from the input line, and then examine only the first character of the input string. Here is a loop for a menu driven program driver:

```
printf("H(elp, Q(uit, D(isplay\n");
while (gets(s)) {
    switch (toupper(*s)) {

         case 'H': help();
                   break;

         case 'Q': exit(0);

         case 'D': display();
                   break;

         default: ;
    }

    printf("H(elp, Q(uit, D(isplay\n");
}
```

The loop reads an input string, `s`, and passes the first character of `s`, `*s` to `toupper()` which converts it to upper case. One of the cases in the switch is selected and an appropriate function is executed. The loop repeats until `gets()` returns end of file.

We may now use library functions, `gets()` and `puts()`, in place of functions we have previously written ourselves to read and write strings. Remember, `gets()` reads an entire line of input text into a string; replacing the newline with a `NULL`. Likewise, `puts()` prints an entire `NULL` terminated string; adding a newline at the end.

11.2.2 String Manipulation: *strlen()* and *strcpy()*

As our next task, let us consider reading lines of text and finding the longest line in the input:

STRSAVE: Read text lines until end of file; save the longest line and print it.

Our approach is similar to the algorithm for finding the largest integer in a list of integers. We save the current "guess" at the longest line in a string, and, as each new line is read, we compare the length of the new line with that of the current longest line. If the length of the new line is greater than that of the current longest, we will save the new line into the longest and proceed. To begin, we initialize the longest line to an empty string; the shortest of all strings. Here is the algorithm:

```
    initialize longest to an empty string

    while not EOF, read a line
         if length of new line > length of current longest
              save new line into longest

    print longest
```

To implement this algorithm, we must consider how we can perform the required operations on the strings holding the new line and the current longest line. We already know how to read and write strings; we also need the operations of finding the length of a string and saving a string. For the former task, the standard library provides a function:

```
int strlen(STRING s);
```

which returns the length of a string,`s`, i.e. the number of characters in `s` excluding the terminating `NULL`.

For the second operation, we can consider the implementation of the maximum integer algorithm and how we saved the new maximum value — we used an assignment operator. However, this will not work for strings. Remember, the string is implemented as a character pointer. If we simply assigned one string variable to another, we would only be saving the pointer to the first string, not the string characters themselves. Then, when we read the next input line, we would overwrite the current string as well. Instead we need to *copy* the new line string into the current longest string. The standard library provides a function for this operation:

```
STRING strcpy(STRING dest, STRING source);
```

which copies a string pointed to by `source` into a location pointed to by `dest`. The function returns the destination pointer, `dest`. This is the equivalent of an assignment operation for data type, `STRING`.

The prototypes for these and other standard library string functions are in a header file, `string.h`. We can now write the program implementing our algorithm as shown in Figure 11.3. Notice, we initialize the current longest string by using `strcpy()` to copy an empty string into `longest`. It is also possible to initialize it as follows:

```
*longest = '\0';
```

or,

```
longest[0] = '\0';
```

Use of `strcpy()` makes it clear that an empty string is copied into `longest`. It has the *flavor* of assigning a string constant to another string, the same way `longest` is updated to the new string, `s`, within the loop body. Thus, we are sticking with our concept of an abstract data type by only using the defined functions to perform operations on data of the type, `STRING`. A sample session is shown below:

```
***Longest Line***

Type text lines, empty line to quit
hello
good morning

Longest line is:
good morning
```

Remember that assignments **cannot** be used to store strings into arrays. When a string is to be stored into a specified character array, use `strcpy()` to copy one string to another; **do NOT** use an assignment operator.

```
/*  File: long.c
    This program reads lines of text and saves the longest line.
*/
#include <stdio.h>
#include <string.h>
#define SIZE 100
#define DEBUG

main()
{   char s[SIZE], longest[SIZE];

    printf("***Longest Line***\n\n");
    strcpy(longest, "");  /* length of empty string is zero */
    printf("Type text lines, empty line to quit\n");

    while (*gets(s))
         if (strlen(s) > strlen(longest))
              strcpy(longest, s);
    printf("Longest line is: \n");
    puts(longest);
}
```

Figure 11.3: Program to find the longest string

Implementing `strcpy()`

The standard library provides the function `srtcpy()` for us to use; however, it is instructive to look at how such a function can be written. Let us write our version of `strcpy()` to copy string, `t`, into string, `s`:

```
/* File: str.c */
/* Function copies t into s */

#include "strtype.h"

STRING our_strcpy(STRING s, STRING t)
{
    while (*t != '\0') {
         *s = *t;
         s++;
         t++;
    }
    *s = '\0';
    return s;
}
```

The arguments passed to formal parameters, `s` and `t`, are of type `STRING`, i.e. character pointers. The loop is executed as long as `*t` is not `NULL`. In each iteration, a character is copied into (the string pointed to by) `s` from (the string pointed to by) `t` by the assignment of `*t` to `*s`. The pointers `s` and `t` are then incremented so they point to the next character positions in the two arrays. If `t` does not point to a `NULL`, the loop repeats and copies the next character, etc. If `t` points to a `NULL`, the loop terminates. After the loop terminates, a terminating `NULL` is appended to `s`. The function returns the pointer, `s`.

Notice, there is a problem with this implementation. The function returns the value of `s`, however, this is no longer a pointer to the destination string — `s` has been incremented as the string was copied and now points to the end of the destination string. We leave the repair of this function as an exercise (see Problem 11).

Several alternate versions of `our_strcpy()` can be written as follows (Note: these versions return `void` rather than a `STRING`):

```
/* File: str.c - continued */
void our_strcpy2(STRING s, STRING t)
{
    while ((*s = *t) != '\0') {
         s++;
         t++;
    }
}
```

In the above, the `while` condition uses the assignment expression whose value is the character assigned to check against `NULL`. If the value is `NULL`, the loop is terminated; however, the assignment places the terminating `NULL` character before the loop is terminated. Here is another variation:

```
/* File: str.c - continued */
void our_strcpy3(STRING s, STRING t)
{
    while (*s = *t) {
         s++;
         t++;
    }
}
```

In the while loop, when the assigned character is '\0', the value of the expression is zero, and therefore false. Otherwise, the character assigned is not `NULL`, and the value of the expression is true. The loop terminates correctly when it should. It is also possible to include increments in the while expression:

```
while (*s++ = *t++)
     ;
```

Here, `*t` is assigned to `*s`, and then `s` and `t` are incremented. The next version uses array indexing; otherwise, it is identical to the last version:

```
/* File: str.c - continued */
void our_strcpy4(STRING s, STRING t)
{ int i;

    i = 0;
    while (s[i] = t[i]) {
         i++;
    }
}
```

Memory Allocation for Strings

When a function is used to put values into an array, it is important that memory for the array be allocated by the *calling* function. Consider the following possible error:

```
/* COMMON BUG */
    char *s;            /* should be: char s[SIZE]; */

    strcpy(s, "Hello, good morning to all");
```

The pointer variable, `s`, can store only a pointer value; no memory is allocated for a string of characters. Nor is the pointer variable `s` initialized. The function, `strcpy()`, assumes that `s` points to memory where a string can be stored. No such memory has been allocated, nor does `s` point to any valid location — the program will crash.

A second type of error can occur if the calling function does not allocate memory for a string, but instead depends on the called function to do so. Let us consider an example in which a string copy function allocates memory for the copied string and returns a pointer to it, and see where the error leads us. Here is the function:

```
/* File: allocerr.c */
#include <stdio.h>
#include "strtype.h"

/* COMMON ERROR */
STRING scopy(STRING t)
{   char s[100];
    int i = 0;

    while (s[i] = t[i])
         i++;

    return s;
}
```

The function copies a string into an (automatic) array variable defined in the function, and returns a pointer to the array. When the function returns to the calling function, the memory for the array, `s`, is freed automatically. The value of `s` is returned, but `s` now points to garbage. Of course, the compiler does not flag an error, since the value of `s` can be legitimately returned. The fact that it now points to garbage is a program logic error.

Let us see what happens when we use this function in a program. We declare a STRING variable, `p`, which is assigned the value of the pointer returned by the above function, `scopy()`.

```
/* File: allocerr.c - continued */
/* PROGRAM BUG */
main()
{   STRING p,  scopy(STRING t);

    p = scopy("hello");
    puts(p);
}
```

The function, `scopy()`, returns a pointer to an array which has already been freed for other uses. The now freed memory, previously holding the array, must be assumed to have garbage value. The pointer to this garbage is assigned to `p`. The function, `puts()`, assumes `p` is a valid string and will print whatever garbage `p` points to, not the original meaningful string. Without a clear understanding, the above type of error is hard to pinpoint. The freed memory holding the array may or may not be immediately used for other purposes; thus, sometimes, `puts()` in the above example may print a (partly) meaningful string. At other times, it will print out all garbage.

The only solution is to declare all the needed arrays in the calling function, `main()` and pass them as arguments to called functions. The called functions can then put strings in these arrays and the calling function, `main()`, can later use these strings without any problem. The correct structure is as follows:

```
...
void scopy (STRING s, STRING t);
```

```
main()
{   char s[SIZE], t[SIZE];

    scopy(s, t);
    ...
}
```

Using String Functions with Substrings

The function, `strcpy()`, is given two character pointers, one to the destination array and one to the source string. These pointers may point to any character position within an array which corresponds to a *substring* beginning at that position, continuing to the next `NULL` in the array. We can call our string functions with arguments that are substrings of other strings. For example, we can copy a substring of `t` into any location in `s`:

```
/*  File: partstr.c
    Program shows overwriting part of a string with part of another.
*/
#include <stdio.h>
#include <string.h>
#define SIZE 100

main()
{   char s[SIZE], t[SIZE];

    printf("***Partial Strings***\n");
    strcpy(s, "This can be trouble");
    strcpy(t, "Insert string");

    printf("Old s: "); puts(s);
    printf("Old t: "); puts(t);

    strcpy(s + 3, t + 5);
    printf("New s: "); puts(s);
}
```

Sample Session:

```
***Partial Strings***
Old s:  This can be trouble
Old t:  Insert string
New s:  Thit string
```

The program copies a substring starting at `t + 5` into a location pointed to by `s + 3`. String copy terminates with a `NULL`; any remaining characters in string `s` after the first `NULL` are not part of the string.

We can even use `strcpy()` to copy part of a string to a different location in the string itself. As always, we must be sure that we are dealing with `NULL` terminated strings and must also take care

that the copy process does not overwrite useful data. For example, here is a loop that eliminates leading white space from a string, `s`:

```
strcpy(s, "    Aloha");
while (isspace(*s))
    strcpy(s, s + 1);
```

The function, `isspace()`, is a library routine that returns True if the argument is a white space. (We have indicated white space explicitly by a `*`). The loop is executed as long as `*s`, the first character of `s`, is a space. In the loop, the string starting at `s + 1` is copied into `s`, character by character. Each time the loop is executed, one leading white space is removed from `s`. Here are the successive strings starting with the original (again we use white space indicator `*`).

```
****Aloha
***Aloha
**Aloha
*Aloha
Aloha
```

When a string is copied into itself by `strcpy()`, as long as destination index is less than the source index, we overwrite only the desired characters. If the destination index is greater than the source index, destination characters will be overwritten. For example:

```
strcpy(s, "abcdef");
strcpy(s+1, s);
```

The second `strcpy()` copies `s[0]`, i.e. `'a'` into `s[1]`; then copies `s[1]` into `s[2]`; then copies `s[2]` into `s[3]`; etc. All elements of `s` are overwritten with `'a'`, even the `NULL`, resulting in a non-valid string — a logic error.

Next, let us consider moving the `NULL` position. Since the first `NULL` terminates a string, we can move the `NULL` to squeeze out unneeded trailing characters. Here is a loop that eliminates trailing white space:

```
    while (isspace(s[strlen(s) - 1]))
         s[strlen(s) - 1] = NULL;
```

Starting with the original, successive strings are shown below with an explicit terminating `NULL` (again, we use a `*` as a white space indicator):

```
Aloha****\0
Aloha***\0
Aloha**\0
Aloha*\0
Aloha\0
```

11.2.3 String Operations: *strcmp()* and *strcat()*

In the last section we saw how a string can be copied and how to determine the length of a string. Two other common operations on strings are to compare them and to join strings, i.e. *concatenate* them.

Our next task is to read lines of text, until a blank line is entered, and examine each line to see if it is the same as a "control string". If a line equals the control string, the line is ignored; otherwise, it is appended to a buffer. When a blank line is encountered in the input, the buffer is printed. The control string is assumed to be entered as the first line. Here is the task:

JOIN: Read a first line as the control string. Read other lines until a blank line is entered, either adding each line to a buffer or discarding it. A line is discarded if it equals the control string. When a line is added to the buffer, separate it from the previous text by a space. Print the buffer at the end of input.

The algorithm will require several functions: one to compare strings, another to append (i.e. concatenate) one string to another. Here is the algorithm:

```
initialize the buffer to an empty string
read the first line into the control string

while not a blank line, read a line
     if the new line is not equal to the control line
          then if the buffer is not empty, append a space to the buffer
               append the new line to the buffer

print the buffer
```

The two new string operations we will need are provided by the standard library. We will use them to implement our algorithm. The first function compares two strings:

```
int strcmp(STRING s1, STRING s2);
```

The function, `strcmp()`, compares the strings, `s1` and `s2`, and returns an integer indicating the result of the comparison. If the two strings are equal, it returns a zero value. If the two strings are not equal, the function returns the difference between the first two unequal characters in the two strings. The returned value will be positive if `s1` is *lexicographically* greater than `s2`, and negative if `s1` is less than `s2`. Thus, the `strcmp()` function is the equivalent of a *relational* operator for strings.

The second function we need is to join two strings. Again, the standard library provides a function:

```
STRING strcat(STRING s1, STRING s2);
```

which *concatenates* (i.e. joins) the two strings, `s1` and `s2`, and stores the result in `s1`. It returns `s1`, i.e. the pointer to the combined string. This is the equivalent of the *addition* operator for strings. The prototypes for these and other standard library string functions are in a header file, `string.h`.

We can now use these functions to implement our program as shown in Figure 11.4. We first

```
/*  File: text.c
    Program reads strings until a blank line is entered. The first string
    read is used as a control. If the other strings are not equal to the
    control string, they are concatenated to the buffer but separated by a
    space. It prints out the buffer at the end. A debug statement prints the
    accumulated string at each step and its length.
*/

#include <stdio.h>
#include <string.h>
#define SIZE 100
#define DEBUG

main()
{   char s[SIZE], control[SIZE], text[SIZE];

    printf("***Build Text: Exclude Control String***\n\n");
    printf("Type control string: ");
    gets(control);
    strcpy(text, "");  /* length of empty string is zero */
    printf("Type text lines, RETURN to quit\n");

    while (*gets(s)) {
         if (strcmp(s, control) != 0) {

              if (strlen(text))
                   strcat(text, " ");

              strcat(text, s);
         }

         #ifdef DEBUG
              printf("debug:length of buffer is %d: %s\n",
                      strlen(text), text);
         #endif
    }

    printf("Final buffer is: ");
    puts(text);
}
```

Figure 11.4: Code using `strcmp()` and `strcat()`

read a string into the variable, `control`, and initialize the buffer, `text`, to an empty string. The while loop then reads strings until a blank line is entered. Since the expression `gets(s)` reads a line of text and returns the destination pointer, `s`, `*gets(s)` is the first character of the string read into `s`. The expression is True if any non-empty string is entered. It is False when the first character of `s` is a `NULL` which occurs when an empty line (just a `RETURN`) is entered.

For each string read into `s`, we compare it with `control`. If they are not equal, we concatenate `text` and `s`. A space is concatenated to `text` if it is not empty, so that the concatenated strings are separated by a space. We have included a debug statement to print the accumulated buffer and its length. When the input terminates, the accumulated string, `text`, is printed. Here is a sample session:

```
***Build Text:  Exclude Control String***

Type control string:  Hello
Type text lines, RETURN to quit
Hello
debug:length of buffer is 0:
earth
debug:length of buffer is 5:  earth
calling
debug:length of buffer is 13:  earth calling
moonbase,
debug:length of buffer is 23:  earth calling moonbase,
hello
debug:length of buffer is 29:  earth calling moonbase, hello

Final buffer is:  earth calling moonbase, hello
```

Observe that string comparisons are *case distinct*, e.g. *hello* is not the same as *Hello*, so the first *Hello* in the input is discarded, while the second, *hello*, is not.

The function, `strcmp()`, can be used when we wish to search for a particular string or when we wish to order strings in lexicographic or dictionary order. Unfortunately, upper case and lower case values of a letter are not equal as shown above; therefore, we must change all strings to the same case (e.g. by using `tolower()`) for a case independent comparison.

To understand how these library functions work, let us write our own versions of functions `strcmp()` and `strcat()`, beginning with `our_strcmp()`. First, let us look in a little more detail of "what" `strcmp()` does. Given two strings, the comparison proceeds character by character until two unequal characters are encountered, or both the strings are exhausted. When two unequal characters are encountered, their difference is returned. If no unequal characters have been encountered when both strings have reached `NULL`, the two strings are identical, and zero is returned. Here are some examples of results using `strcmp(string1, string2)`:

```
/*  File: str.c - continued
    Compares strings s and t, returns difference of first
    unequal characters or returns zero.
*/

int our_strcmp(STRING s, STRING t)
{
    while ( *s )  {           /* terminate when s  is exhausted     */

         if (*s != *t)        /* if unequal, break loop             */
              break;

         s++;                 /* traverse the two strings           */
         t++;
    }

    return *s - *t;           /* return the difference of characters */
}
```

Figure 11.5: Code for `our_strcmp`

string1	string2	returned value	comment
hawaii	hawaiian	0 - 'a'	negative
hilo	hawaii	'i' - 'a'	positive
hawaii	hawaii	0	zero
hawhaw	hawaii	'h' - 'a'	positive
Hawaii	hawaii	'H' - 'h'	negative
haw123	hawaii	'1' - 'a'	negative

We can model our algorithm on this behavior of `strcmp()`. We traverse both strings until we arrive at a terminating `NULL` in either one. During traversal, we examine the corresponding characters in the strings to see if they are unequal. If so, we terminate the traversal loop. Otherwise, we continue the process. When the loop is terminated, we return the difference between the characters where we left off in the two strings.

Figure 11.5 shows the code implementing this algorithm. The while loop traverses strings `s` and `t` terminating when `s` points to a `NULL` character. Within the loop, the corresponding characters of the two strings are compared. If unequal characters are encountered, the loop is terminated, and the difference between the characters is returned. If the loop terminates because `*s` is zero, then no unequal characters have been encountered so far, but the string `t` may or may not be exhausted. In either case, `*s - *t`, i.e `0 - *t` is returned. In particular, if `t` points to `NULL` (the string `t` is also exhausted), then the two strings are equal and zero is returned. Otherwise, the difference between the first unequal characters is returned. Note, we do not need to test for the end of the string `t` in the while condition. If `t` terminates before `s`, then the `NULL` at the end of string `t` will not compare equal to `*s`, and the loop will terminate anyway.

```
/*  File: str.c - continued
    Concatenates s and t by appending t to s. Returns
    pointer to s. s must point to a large enough array to accommodate the
    concatenated string.
*/

STRING our_strcat(STRING s, STRING t)
{   STRING p;

    p = s;              /* save pointer s */

    while (*s)          /* increment s until it points to NULL */
         s++;

    strcpy(s, t);       /* copy t into s */
    return p;           /* return saved pointer */
}
```

Figure 11.6: Code for `our_strcat()`

To write `our_strcat()`, we must append the second string to the end of the first string; so we must traverse the first string until we find the `NULL`. We can then copy the second string at this point in the first using `strcpy()`. The function returns the pointer to the destination string, i.e. the beginning of the first string. Since the function must return a pointer to the original string, `s`, we save the original pointer in a variable, `p`. We then increment `s` until it points to the terminating `NULL`. We then copy `t` into `s` starting at the `NULL` character position using `strcpy()`, and return the saved pointer, `p`. This function performs the same task as does `strcat()`.

11.2.4 String Conversion Functions

Besides the functions for manipulating strings discussed in the previous sections (and others not discussed, but presented in Appendix C), the standard library provides several functions for converting the character (ascii) information in a string to other data types such as integers or floats.

We will illustrate the use of one such function, `atoi()`, by modifying our function `getint()` that we wrote in Chapter 4. Recall, this function reads the next valid integer from the standard input character by character, skipping over any leading white space, converts the character sequence to an integer representation, and returns the integer value. The prototype for this function is:

```
int getint(void);
```

In our previous version of this function, we made it robust enough to detect when `EOF` or invalid (non-digit) characters are present in the input. Here we will extend the utility of `getint()` to read the next white space delimited item in the input, and convert it to integer form, this time allowing a leading + or - sign, and give the user the opportunity to re-enter data for illegal character errors.

GETINT: Write a program that reads only a valid integer. If there is an error in entering an integer, it detects the error and allows the user to re-enter the data.

The program driver is quite simple; it calls the function `getint()` that returns a valid integer read from the standard input. The driver then prints the integer returned by the function. Here is the algorithm for getint():

```
initialize valid to False

while not a valid string
     read a string s
     set valid to True if s represents a valid integer

     if valid
          return an integer represented by the string s
     else
          print an error message
```

The function reads in the input as a string, and checks if it is a valid digit string for an integer. To check if a string `s` is a valid integer string, we examine whether it consists of only digits with, perhaps, a leading unary sign (`+` or `-`). The following algorithm sets `valid` to True if `s` represents a valid integer:

```
if *s is '+' or '-'
     valid = digitstr(s + 1);
else
     valid = digitstr(s);
```

If the first character of `s` is a unary sign, check the rest of the string (starting as `s + 1`) for all digits; otherwise check the entire string `s` for all digits.

If `s` is a valid digit string, the function returns an equivalent integer using the standard library function, `atoi()`. The call `atoi(s)` returns the integer represented by the string `s`. The function atoi() has the prototype (included in `stdlib.h`):

```
int atoi(STRING s);
```

If `s` is not a valid string, the user is prompted to type the input again.

To check if all characters in a string are digits, `getint()` uses the function `digitstr()`. The algorithm modules are combined and implemented in a program shown in Figure 11.7.

The driver gets an integer and prints it. The function `getint()` reads the next white space delimited string in the input using `scanf()`. If the first character is a unary operator, we check for digits string starting at the pointer `s + 1`; otherwise, we check starting at the pointer `s`. The flag, `valid`, stores the value returned by `digitstr()`. If `valid` is True, we use `atoi()` to return the integer represented by `s`; otherwise, we print a message prompting the user to re-enter the integer, flush any remaining characters on the input line, and read the new input. The flag `valid` is initialized to False, and the loop continues as long as `valid` remains False, i.e. as long as a valid integer is not entered.

The function `digitstr()` traverses the string until a `NULL` appears. If a non-digit is encountered anytime during traversal, it returns False; otherwise, at the end of traversal, it returns True. It uses the library function, `isdigit()` to check if a character is a digit.

A sample session is shown below:

```
/*  File: intchk.c
    This program reads and prints an integer. It detects errors in
    input and asks the user to retype.
*/
#include <stdio.h>
#include "tfdef.h"      /* defines TRUE, FALSE *.
#include <stdlib.h>     /* prototype for atoi() */
#include <ctype.h>
#define SIZE 100
main()
{   int n;
    printf("***Valid Integer Input***\n\n");
    printf("Type an integer: ");
    n = getint();
    printf("Integer is %d\n", n);
}
/* Function gets a valid integer. */
int getint(void)
{   char s[SIZE];
    int valid = FALSE;       /* flag for valid string */

    while(!valid) {
         scanf("%s",s);               /* read a string delim by ws */
         if (*s == '+' || *s == '-')   /* if first char is + or -, */
              valid = digitstr(s + 1); /* check rest of string; */
         else                          /* otherwise, */
              valid = digitstr(s);     /* check the entire string */

         if (valid)               /* if a valid string */
              return(atoi(s));    /* return its equivalent integer */
                                  /* otherwise, */
         printf("***Error in input\n");/* print an error mesg */
         printf("Re-enter your integer: ");
         while(getchar() != '\n'); /* flush remainder of input */
    }
}
/* File: str.c - continued */
/* Checks if a string t is all digits  */
int digitstr(STRING  t)
{
    while (*t)
         if (!isdigit(*t))        /* if any character in t is */
              return FALSE;       /* NOT a digit, return FALSE */
         else  t++;               /* else point to next char. */
    return TRUE;        /* if all chars are digits, return TRUE */
}
```

Figure 11.7: Code for `getint()`

```
***Valid Integer Input***

Type an integer:  123e
***Error in input
Re-enter your integer:  =123
***Error in input
Re-enter your integer:  -+123
***Error in input
Re-enter your integer:  -123
Integer is -123
```

11.2.5 File I/O with Strings

Earlier in this chapter, we described library functions to do string I/O with the standard input and output. The library also provides functions to do I/O with files. Here we will illustrate the use of these functions with our next task; to search for the presence of a string in the lines of a text file.

GETLNS: Search for a control string in the lines of a file. Each line that contains the control string is to be written to an output file and to the standard output.

The algorithm is written easily if we write a function, `srchstr()`, that searches for the presence of one string in another. Here is the algorithm:

```
get the control string control
open files

while not EOF, read a line s from the input file
     if srchstr(s, control) is True
     then write the line to output file and stdout
```

We could use character I/O to read from an input file, but it is easier to use library string I/O functions: `fgets()` and `fputs()`.

```
int fgets(STRING s, int n, FILE *fp);
int fputs(STRING s, FILE *fp);
```

These functions are similar to `gets()` and `puts()` with minor differences. The function `fgets()` reads a string from a stream, `fp`, into a buffer, `s`. The maximum size, `n`, of the string buffer must be specified to `fgets()` and must allow for a terminating `NULL` character. The function reads a string until a newline character is encountered or the specified maximum size of buffer is reached. It adds the terminating `NULL`, but it does **NOT** strip the newline character as does `gets()`. The `NULL` is added **after** the `\n` character and `fgets()` returns the buffer pointer if successful, or `NULL` otherwise.

The function `fputs()` outputs a string to a stream `fp`. It strips the terminating `NULL` from the string, but does **NOT** add a newline character as does `puts()`. The function returns the last character output if successful, `EOF` otherwise. The prototypes for these functions are included in `stdio.h`.

```
/*  File: srchstr.c

    This program searches for a string in an input file. Every line
    that contains the string is printed out.
*/
#include <stdio.h>
#include "tfdef.h"
#include "strtype.h"
#define SIZE 100

main()
{   FILE *input, *output;
    char infile[15], outfile[15];
    char s[SIZE], control[SIZE];

    printf("***String Search***\n\n");
    printf("Type a string to be searched for: ");
    gets(control);

    printf("Input file : ");
    gets(infile);
    printf("Output file : ");
    gets(outfile);
    input = fopen(infile, "r");
    if (input == NULL) {
         puts("*** Can't open input file ***");
         exit(0);
    }
    output = fopen(outfile, "w");
    if (output == NULL) {
         puts("*** Can't open output file ***");
         exit(0);
    }

    while (fgets(s, SIZE - 1, input))

         if (srchstr(s, control)) {
              puts(s);
              fputs(s, output);
         }

    fclose(input);
    fclose(output);
}
```

Figure 11.8: Driver for Text Searching Program

```
/*  File: srchstr.c - continued */
/* Function tests if str is in s */
int srchstr(STRING s, STRING str)
{

    while (*s)
         if (compare(s, str))     /* if str is at the start of s */
              return TRUE;        /* return True */
         else s++;                /* otherwise, go to the next pos. */

    return FALSE;                 /* string exhausted, return False */
}
```

Figure 11.9: Code for `srchstr()`

The program driver for our task is easy to write as shown in Figure 11.8. The program driver first reads the control string to search for. It then opens the input and output files. The while loop reads lines from the input file until end of file. Each line read is tested by `srchstr()` for the presence of the control string. If the control string is present in the line, it is written to both `stdout` and the output file. We will need `TRUE` and `FALSE` definitions for `srchstr()`, so we have included the header file, `tfdef.h`.

The function, `srchstr()` traverses the string, `s`, and tests if the control string is present at each position in `s`. If it is present, it returns `TRUE`; otherwise, it goes to the next position. The function, `srchstr()`, uses a function, `compare()`, to see if a string is present at the start of another string. This is different than `strcmp()` since the string we are searching in may not terminate at the end of the control string. The code for `srchstr()` is shown in Figure 11.9. Given a string, `s`, and a control string, `str`, it starts at the first character of `s`, and calls `compare()` to see if `str` is present in `s` starting at the first character. If it is, it returns `TRUE`; otherwise, it increments `s` to point to the next character. If the string is exhausted, it returns `FALSE`.

The code for `compare()` is shown in Figure 11.10. It traverses `str` and `s` until `str` is exhausted. If it encounters corresponding characters that are not the same in the two strings, it returns `FALSE`. When `str` is exhausted, it returns `TRUE`. Here is a sample session:

```
***String Search***

Type a string to be searched for:  while
Input file :  ucstr.c
Output file :  xyz
while (gets(s)) {
```

The file `ucstr.c` contains only one line with the string *while* in it. That line is written to the file `xyz` and to `stdout`.

For this task, we have written our own function to compare `str` with the first several characters in string `s` because we do not expect `s` to terminate at the end of the control string, `str`. If `n` is

```
/*  File: srchstr.c - continued */
/* Function tests if str is at the start of s */

compare(STRING s, STRING str)
{
    while (*str)
         if (*str++ != *s++)
              return FALSE;

    return TRUE;
}
```

Figure 11.10: Code for `compare()`

the length of `str`, then we require a comparison of the first `n` characters in the two strings. There is a standard library function, `strncmp()`, which does just that:

```
int strncmp(STRING s, STRING t, unsigned n);
```

It compares the first `n` characters of `s` and `t`, and returns the difference of the first unequal characters, or it returns zero if they are all equal, just like `strcmp()`. So, instead of `compare(s, str)`, we could have used:

```
strncmp(s, str, strlen(str))
```

A similar library function, `strncpy()`, is also available:

```
STRING  strncpy(STRING s, STRING t, unsigned n);
```

which copies `n` characters from the source string, `t`, into the destination string, `s`, without adding a terminating `NULL`. It returns `s`.

We close this section by emphasizing the difference between `gets(s)` and `fgets(s, n, fp)`. Let us assume an input string "Hawaii\n" is in the standard input, and that the string `s` is large enough to accommodate the example string with `n` selected appropriately. The string, `s` is shown below after the use of each function:

```
gets(s):              Hawaii\0     /* newline stripped, NULL appended */
fgets(s, n, stdin):   Hawaii\n\0   /* newline present, NULL appended */
```

Similarly, the output of the functions `puts(s)` and `fputs(s, fp)` is shown below:

```
puts(s):              Hawaii\n  /* NULL stripped, newline appended */
fputs(s, stdout):     Hawaii\n  /* NULL stripped */
```

11.3 More Example Programs

In the previous section we have discussed some of the string utility functions provided by the C standard library and illustrated their use with examples. Additional string functions can be found in Appendix C. We close this chapter with a few additional example programs making use of these string processing functions.

11.3.1 Palindromes

Our next task is:

PALI: Read strings and check if each is a palindrome.

A palindrome is a string that reads the same forwards and backwards, for example:

able was i ere i saw elba

The algorithm is simple: compare the reverse of the string with the original string. If they are the same, the string is a palindrome.

```
while not EOF, read a string s
     copy reverse of s into t

     if s and t are equal,
          s is a palindrome
     else
          s is not a palindrome
```

The driver follows the algorithm closely, as seen in Figure 11.11. We will use a function, `revcpy()`, to copy the reverse of the string.

We must write the function `revcpy()` to copy one string into another in reverse order. To see how the algorithm for this function should proceed, we will work with the indices in the source and destination strings as shown below:

```
src:   hello\0
sind:  <—4

dest:  olleh\0
dind:  0—>
```

The string, `src`, is shown with the terminating `NULL` and the source index, `sind`, must start at the last character of `src` and decrease as each character is copied. In the destination string, `dest`, the index, `dind`, must start at 0 and increase as each character is copied. When the source index becomes negative, all characters have been copied in reverse order from the source. After all the characters are copied, a terminating `NULL` must be added to the destination string. Here is the algorithm:

```
initialize sind to the last index of src and dind to 0
while sind is >= 0
     copy from src to dest
     increment dind and decrement sind
add a NULL to dest
```

```
/*  File: pali.c
    Program reads a string and tests whether it is a palindrome.
    It repeats the process until EOF.
*/

#include <stdio.h>
#include <string.h>
#include "strtype.h"
#define SIZE 100

main()
{   char s[SIZE], t[SIZE];

    printf("***Palindrome Test***\n\n");
    printf("Type strings, EOF to quit\n");

    while (gets(s)) {
         revcpy(t, s);            /* copy reverse of s into t */

         if (strcmp(s, t) == 0)
              printf("%s: a palindrome\n", s);
         else
              printf("%s: not a palindrome\n", s);
    }
}
```

Figure 11.11: Driver for Palindrome

```
/*  File: pali.c - continued
    Function copies string src in reverse order into string dest.
*/

void revcpy(STRING dest, STRING src)
{   int sind, dind = 0;      /* dest index is 0 */

    sind = strlen(src) - 1;  /* source index at last character */

    while (sind >= 0)        /* loop while source index is non-neg. */
         dest[dind++] = src[sind--];   /* copy character, and update */

    dest[dind] = NULL;                 /* append a NULL */
}
```

Figure 11.12: Code for `revcpy()`

The function is shown in Figure 11.12.

Here is a sample session:

```
***Palindrome Test***

Type strings, EOF to quit
this is it
this is it:  not a palindrome
able was i ere i saw elba
able was i ere i saw elba:  a palindrome
^D
```

Our function, `revcpy()`, will work fine as long as the source and destination strings are different strings. We could write a function to reverse a string in place. We can follow the same procedure of copying from the source index to the destination index; however, since the source and destination strings are the same string, characters at source index as well as at destination index must be swapped rather than simply assigned. Otherwise, copying a character from the source index to the destination index will overwrite a character.

```
      s:  hello\0
   sind:  <—4

   dind:  0—>
  new s:  olleh\0
```

Furthermore, the characters need only be swapped as long as source index is greater than destination index. When the source index is less than the destination index, all characters have been swapped. If the two indices are equal, the corresponding characters are the same and need no swapping. Finally, a terminating `NULL` need not be added since it is already present in the correct position. Figure 11.13 shows the code for the function `revself()`.

```
/*  File: str.c - continued
    Function reverses string s in place.
*/
void revself(STRING s)
{   int c, sind, dind = 0;   /* c used as temp. storage during a swap */
                             /* dest index at 0 */

    sind = strlen(s) - 1;    /* src index at last char. */

    while (dind < sind) {    /* loop while chars need swapping */
         c = s[dind];        /* swap characters, */
         s[dind++] = s[sind]; /* and update indices */
         s[sind--] = c;
    }
}
```

Figure 11.13: Code for `revself()`

11.3.2 Words

Our next task is to break up a string into words delimited by white space.

WDS: Read strings; break up each string into its constituent words.

The algorithm starts by skipping over leading white space. If the string is not exhausted, a word starts at that position and continues until the next white space. Here is the algorithm.

```
while not EOF, read a string s

    initialize string index to 0

    while not NULL
         skip over leading white space
         initialize word index to 0

         copy the next word into wd
         terminate word with a NULL
         print the word
```

In our algorithm, a word is any sequence of characters delimited by white space. Figure 11.14 shows the program. It reads lines until `EOF` scanning each line until a `NULL` is reached. Each scan first skips over white space, then copies a word into a string, `wd`, while characters are non-white space and non-NULL. A terminating `NULL` is added to the word and it is printed. Here is sample session:

```
***Words in Strings***
```

```
/*  File: strwds.c
    This program reads strings until EOF. For each string read, it copies each
    of the words into another string and prints it.
*/

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define SIZE 80
#define WDSIZE 40

main()
{   char s[SIZE], wd[WDSIZE];
    int i, j;

    printf("***Words in Strings***\n\n");
    printf("Type strings, EOF to quit\n");

    while (gets(s)) {                  /* read lines until EOF */
         i = 0;

         while (s[i]) {                /* repeat while s[i] is not NULL */

              while (isspace(s[i]))    /* skip leading white space */
                   i++;

              j = 0;                   /* initialize for a new word */

              while (s[i] && !isspace(s[i])) /* while non-NULL AND non-white */
                   wd[j++] = s[i++];         /* copy word */

              wd[j] = NULL;                  /* terminate string */
              puts(wd);                      /* print word */
         }
    }
}
```

Figure 11.14: Code for separating words from a string

```
Type strings, EOF to quit
This is a test
This
is
a
test
^D
```

11.3.3 Substrings

In string manipulations, it is frequently necessary to find a substring of a string. A substring is a string that is part of another string. It can be parameterized by specifying where the substring starts and how long it is. Our next task is to write a program that finds a substring of a string at a given position and of a specified size.

SUBSTR: Read substring parameters. For each line of input, find the appropriate substring.

For example, consider the string:

Source string: This is a test string\0

(The terminating `NULL` is shown explicitly). A substring of this string starting at index position 2 and containing 5 characters is:

Destination string: is is\0

We will write a function to extract such a substring. The function must be passed several arguments: the source string (pointer), a destination string where the specified substring is to be copied (pointer), the starting position of the substring in the source string (integer), and the number of characters for the substring (integer).

It may or may not be possible for the function to extract the string. For example, if the starting position is outside the string, no substring can be extracted. We will assume that the function returns the destination string (pointer) if successful in extracting a string; otherwise, it returns a `NULL` pointer to indicate failure. We will also assume that the function will extract as many characters as possible upto the specified number. The function prototype should be:

```
STRING substr(STRING src, STRING dest, int startpos, int nchrs);
```

The parameter, `src`, points to the source string, and `dest` points to an array where the substring of `src` is to be copied. The next two arguments provide the starting position and the number of characters. The **calling** function must allocate memory for the destination string. The starting position, `startpos` is an index into the array — it must be 0 or greater. The parameter, `nchrs` is the maximum number of characters to copy into the substring.

Since the program depends primarily on `substr()`, let us first develop an algorithm for it. The function must start copying characters from the starting position startpos. If we use an array index, `src[startpos]` accesses the character at the start position if `startpos` is in the source string. If `startpos` is not in the source string, we will return a `NULL` to indicate failure to extract a substring.

Next, we must copy up to a maximum of `nchrs` characters into `dest`. When the source string is exhausted or `nchrs` characters are copied, we stop the copy process and append a `NULL` to the

substring. If even one character is copied into the substring, we will return the destination pointer. Here is the algorithm:

```
if startpos >= strlen(src)
     return NULL

j = 0;
while j is less than nchrs and src is not exhausted
     copy a character: dest[j] = src[startpos + j]
     increment j

terminate dest with NULL
return dest
```

The program driver reads the start position and the number of characters. It then reads strings until end of file and finds the substring for each string if possible. The code for the driver and `substr()` is shown in Figure 11.15. The program prints the substring if it can be extracted; otherwise, it prints a message. Here is a sample session:

```
***Substring Extraction***

Type start position and number of characters:  2 5
Type text lines, EOF to quit
this is a test string
is is
hello
llo
well
ll
he
Substring cannot be extracted
then
en
^D
```

11.4 Common Errors

1. Failure to include library header files, e.g. `string.h`. Prototypes for library string routines are not included resulting in default assumptions and consequent problems.

2. We have already discussed common string related errors in Chapter 7 and in this chapter. Always allocate space for an array where a string is to be stored. Once space is allocated for a string, pointer variables can be used to access strings.

3. Array names must not be used as Lvalues.

```
/*  File: substr.c
    Program extracts a substring and prints it.
*/
#include <stdio.h>
#include "strtype.h"
#define SIZE 80
STRING substr(char src[], char dest[], int startpos, int nchrs);

main()
{   char s[SIZE], sub[SIZE];
    int start, n;

    printf("***Substring Extraction***\n\n");
    printf("Type start position and number of characters: ");
    scanf("%d %d%*c", &start, &n);     /* suppresses newline */
    printf("Type text lines, EOF to quit\n");

    while (gets(s)) {
         if (substr(s, sub, start, n)) /* if substring, */
              puts(sub);               /* print it */
         else
              printf("Substring cannot be extracted\n");
    }
}

/*  Function copies a substring of src, starting at i and n characters
    long, into dest. It returns dest if success; NULL otherwise.
*/
STRING substr(STRING src, STRING dest, int startpos, int nchrs)
{   int j;

    if (startpos >= strlen(src))
         return NULL;

    for (j = 0; j < nchrs && src[startpos + j]; j++) {
         dest[j] = src[startpos + j];
    }

    dest[j] = NULL;
    return dest;
}
```

Figure 11.15: Code for the substring program

11.5 Summary

This chapter has discussed a very common data type in C programs: the string. We have briefly introduced the concept of an *abstract data type* as consisting of a data declaration and a set of operations on data items of that type. We have defined a user defined type, `STRING`, for string data and used it throughout the chapter. (While our string data type does not completely satisfy the definition of an abstract data type, the basic concept is seen).

Many common operations on string data are provided through the standard library. We have described a few of these; in particular functions for I/O: `gets()` and `puts()`, and file I/O: `fgets()` and `fputs()` whose prototypes are defined in `stdio.h`. In addition the functions for string manipulation, `strlen()` and `strcpy()` as well as string operation, `strcmp()` and `strcat()`, have been described. Other functions described include `atoi()`, `strncmp()`, and `strncpy()`.

Throughout the chapter we have shown numerous examples of programs for string processing.

11.6 Exercises

1. If the characters in an array, `s` are: string\0 of characters\0

   What does each of the following print? Show each character.

```
printf("%s", s);
puts(s);
fputs(s, stdout);
```

2. If the input of characters is:

```
string of characters\n
```

   What does each of the following read? Show each character, including `NULL`.

```
scanf("%s", s);
gets(s);
fgets(s, sizeof(s) - 1, stdin);
```

3. Assume `s` is a string array. Under what condition is each of the following True?

```
s
*s
!*s
gets(s)
*gets(s)
!*gets(s)
```

   Find and correct any errors in the following and determine the outputs where feasible. The input is shown when appropriate.

4.
```
main()
{   char s[80], t[80];

    s = "this is a message";
    if (s == t)
         printf("Equal\n");
    else
         printf("Not equal\n");
    puts(s);
}
```

5.
```
main()
{   char s[80], t[80];

    scanf("%s", s);
    printf("%s", s);
}
```
Input: This is a message

6.
```
main()
{   char *s;

    s = "this is a message";
    printf("%d  %s\n", s, s);
    puts(s);
}
```

7.
```
main()
{   char *s;

    gets(s);
    puts(s);
}
```

8.
```
main()
{   char s[80];

    while (*s) {
         putchar(*s);
         s++;
    }
}
```

9.
```
main()
{   char *s;

    strcpy(s, "hello");
    puts(s);
}
```

10.
```
main()
{   char name[80];

    name = get_str(name);
    puts(s);
}

char *get_str(char *s)
```

```
    {
        gets(s);
        return s;
    }
```

11.
```
int cmpstr(char *s, char *t)
    {
        if (s == t)
             return TRUE;
        else
             return FALSE;
    }
```

11.7 Problems

Write program drivers for each of the following. The driver should read appropriate data until end of file, call the functions described below, and print the results.

1. Write a function that returns the index where a character, `c`, occurs in a string, `s`. The function returns -1 if `c` is not present in `s`. Use array indexing.

2. Repeat 1 using pointers.

3. Write a function that returns the index where a character, `c`, occurs in a string `s`; the search for `c` starting at a specified index, `i` in `s`. The function returns -1 if `c` is not present in `s` starting at the index, `i`. Use array indexing.

4. Repeat 3 using pointers.

5. Write a function, `how_many()`, that returns the number of times a character, `ch`, occurs in a string, `s`. Use array indexing.

6. Repeat 5 using pointers.

7. Write a function that substitutes a new character, `newc`, for the first occurrence of a character, `c`, in a string, `s`. Use array indexing.

8. Repeat 7 using pointers.

9. Write a function that substitutes a new character, `newc`, for every occurrence of a character, `c`, in a string, `s`. Use array indexing.

10. Repeat 9 using pointers.

11. Rewrite the function, `our_strcpy()` in Section 11.2.2 so that it properly returns the pointer to the destination string.

12. Write a function that takes two strings, `s` and `t`, as arguments. Copy string `s` into `t`, but remove all white space and punctuation. Use array indexing.

13. Repeat 12, but use pointers.

14. Write a function that takes a string of characters and removes all white space and punctuation in that same string. Use array indexing.

15. Repeat 14 using pointers.

16. Write a function, `xwslead()`, that removes all leading white space from a string. Use array indexing.

17. Repeat 16 using pointers.

18. Write a function, `xwstrail()`, that removes all trailing white space from a string. Use array indexing.

19. Repeat 18 using pointers.

20. Write a function, `xws()`, that removes all leading and trailing white space from a string. Use array indexing.

21. Repeat 20 using pointers.

22. Write a function, `squeeze()`, that removes all white space from a string. Use array indexing.

23. Repeat 22 using pointers.

24. Write a function, `compare()`, that takes two strings as arguments and compares them for equality after leading and trailing blanks are removed. If the strings are equal after the leading and trailing blanks are removed, the function returns True. Otherwise, it returns False.

25. Write a function, compstrip(), that takes two strings as arguments and compares stripped versions of them. A stripped string is one from which all white space and punctuation are removed. Function returns True if the strings are equal after they are stripped.

26. Write a function, `palindrome()` that checks if a given string is the same forwards and backwards. Use pointers.

27. Write a function that checks if a string is a palindrome ignoring all white space. Example:

```
i ia wah hawaii
```

28. Write a function that takes two string arguments, `s` and `t`. Copy `s` into `t` in reverse order, except that a sequence of white space is squeezed to a single space.

29. Write a function that takes a single string argument, and reverses the string itself, except that white space is squeezed to a single space.

30. Write a function that removes the first word from a string. Write a program that uses the function to remove a specified number of leading words from a string.

31. Write a function that removes the last word in a string. Write a program that uses the function to remove a specified number of trailing words from a string.

32. Write a function that takes two strings, `s1` and `s2` as arguments. It returns the index where `s2` occurs in `s1`, or it returns -1 if `s2` is not in `s1`.

33. Write a function that substitutes a new string, `repl_str`, for the first occurrence of a string, `str` in a string, `src`.

34. Write a function that replaces a new string, `repl_str`, for every occurrence of a string, `str`, in `src`.

35. Write a function that detects the presence of a whole word, `wd`, in a string, `s`.

36. Write a function that converts a string into an integer. The conversion is terminated when a non-digit is encountered.

37. Write a function that converts a string into a float. The conversion is terminated when a character that does not belong in a decimal number is encountered.

38. Write a function that converts an integer to a string.

39. Write a function that converts a float to a string.

40. Write a function that converts a string of binary digits to an integer.

41. Write a function that converts an unsigned integer into a string of binary digits.

42. Write a function, `nexttok()`, that gets the next token from a string, starting at a specified array index, called `cursor`. The function returns the new value of `cursor`, the token itself, and the type of the token. Leading white space is skipped. A longest valid token is built as long as the characters belong to a token type. The token is complete when a character that does not belong to a token type being built is encountered.

    A valid token type is either an identifier, an integer, a float, an invalid, or an EOS, end of string. An identifier starts with a letter, and may be followed by letters and/or digits. An integer starts with a digit, and may be followed by digits. A float must start with a digit, may be followed by digits, may be followed by a decimal point, and may be followed by a sequence of digits. A character other than white space, letters, and digits is an invalid type token containing that one character. EOS type of token is returned when the `NULL` character is reached.

    Write a program that reads in lines of input from a file, and use the above function to print out the tokens in each line until `EOF`.

Chapter 12

Structures and Unions

So far, we have seen one kind of compound (user defined) data type — the array and in Chapters 7 and 9 have seen how we can group information into one common data structure. However, the use of arrays is limited to cases where all of the information to be grouped together is of the same type. In this chapter we present the other compound data type provided in C — the structure, which removes the above limitation. We will discuss structures, pointers to structures, and arrays of structures. As with our previous data types, we will see how such structures can be declared; how information in them can be accessed, and how we can pass and return structures in functions. We will also see how arrays of structures are sorted and searched. We illustrate these points with several example programs.

Finally, we will introduce unions which are similar to structures; however, the elements in the union share the same memory cells. In a union, different types of data may be stored in a variable but at different times.

12.1 Structures

In C, a *structure* is a derived data type consisting of a collection of member elements and their data types. Thus, a variable of a structure type is the name of a group of one or more *members* which may or may not be of the same data type. In programming terminology, a structure data type is referred to as a *record data type* and the `members` are called `fields`. (We will use these two terms interchangebly).

12.1.1 Declaring and Accessing Structure Data

As with any data type, we need to be able to declare variables of that type. In particular for structures, we must specify the names and types of each of the fields of the structure. So, to declare a structure, we need to describe the number and types of fields in the form of a *template*, as well as declare variables of that type. We illustrate with an example: a program that maintains temperatures in both celsius and fahrenheit degrees. A variable, `temp`, is to be used to maintain the equivalent temperatures in both celsius and fahrenheit, and thus requires two fields, both of them integers. We will call one field `ftemp` for fahrenheit temperature and the other `ctemp` for celsius. The program, shown in Figure 12.1, reads a temperature to the `ftemp` field of the variable, `temp`, and uses a function, `f_to_c()`, to convert the temperature from fahrenheit to celsius and store it in the `ctemp` field. In looking at this program, we see that the variable `temp` is declared

```
/*   File: fctemp.c
     Program reads temperature in fahrenheit, converts to celsius, and
     maintains the equivalent values in a variable of structure type.
*/
#include <stdio.h>
main()
{    struct trecd {
          float ftemp;
          float ctemp;
     } temp;
     double f_to_c(double f);
     char c;

     printf("***Temperatures - Degrees F and C***\n\n");
     printf("Enter temperature in degrees F : ");
     scanf("%f",&temp.ftemp);
     temp.ctemp = f_to_c(temp.ftemp);
     printf("Temp in degrees F = %3.1f\n", temp.ftemp);
     printf("Temp in degrees C = %3.1f\n", temp.ctemp);
}

/*   This routine converts degrees F to degrees C */
double f_to_c(double f)
{
     return((f - 32.0) * 5.0 / 9.0);
}
```

Figure 12.1: Code for Simple Structure Program