2. Write a code segment that uses an array of pointers to strings and `sprintf` to convert a type `double` monetary value less than `10.00` to a string for use on a check. For example, `4.83` would be converted to `"Four and 83/100 dollars"`.

Programming

1. Write a type `int` function `strtoint` and a type `double` function `strtodouble` that convert string representations of numbers to their numeric equivalents.

```
strtoint("-8") → -8
strtodouble("-75.812") → -75.812
```

9.8 String Processing Illustrated

You have been using a text editor to create and edit C programs. This is probably a fairly sophisticated program that uses special commands to move the cursor around the screen and to specify edit operations. Although you cannot develop such an editor yet, you can write a less sophisticated one that processes a single line of text.

CASE STUDY Text Editor

PROBLEM

Design and implement a program to perform editing operations on a line of text. Your editor should be able to locate a specified target substring, delete a substring, and insert a substring at a specified location. The editor should expect source strings of less than 80 characters.

ANALYSIS

The editor's main function must get the source line to edit and then repeatedly scan and process editor commands until it receives the Q (Quit) command. We will allow strings of up to 99 characters, but we will not check for overflow.

DATA REQUIREMENTS

Problem Constant

```
MAX_LEN      100      /* maximum size of a string */
```

Problem Inputs

```
char source[MAX_LEN]        /* source string */
char command                /* edit command */
```

Problem Output

```
char source[MAX_LEN]        /* modified source string */
```

DESIGN

INITIAL ALGORITHM

```
1. Scan the string to be edited into source.
2. Get an edit command.
3. while command isn't Q
       4. Perform edit operation.
       5. Get an edit command.
```

REFINEMENTS AND PROGRAM STRUCTURE

Step 4 is performed by function `do_edit`. A structure chart for the text editor is shown in Fig. 9.19; the local variables and algorithms for function `do_edit` follow.

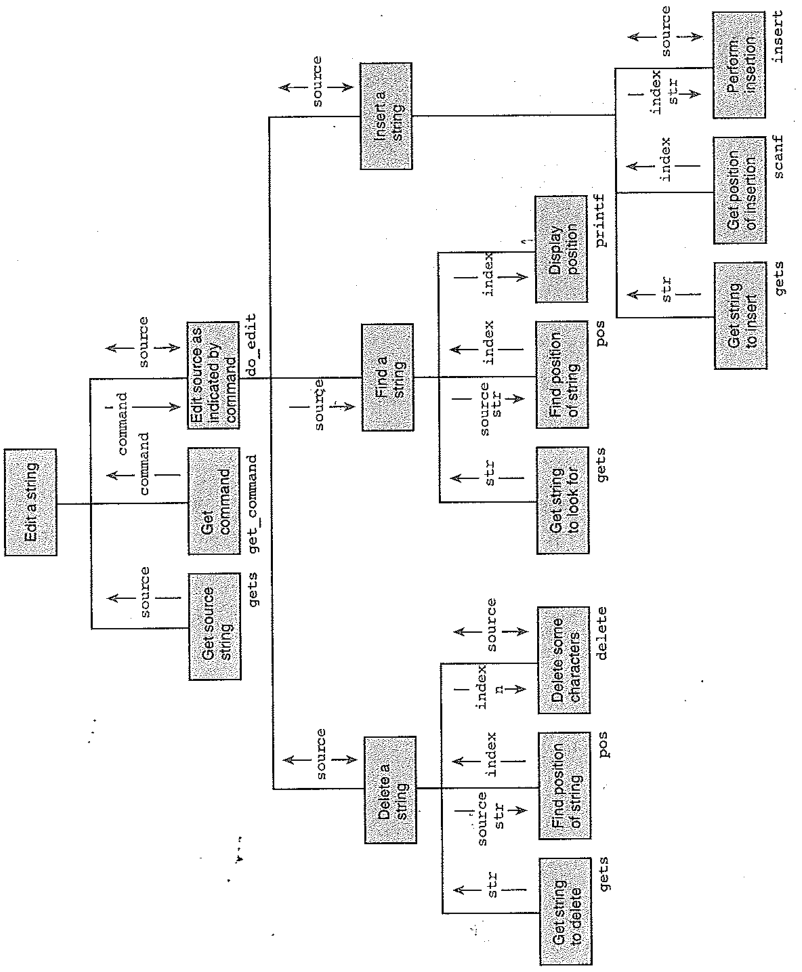
Local Variables

```
char str[MAX_LEN]           /* string to find, delete, or insert */
int  index                  /* position in source */
```

ALGORITHM FOR DO_EDIT

```
1. switch command
       'D':  2. Get the substring to be deleted (str).
             3. Find the position of str in source.
             4. if str is found, delete it.
       'I':  5. Get the substring to insert (str).
             6. Get position of insertion (index).
             7. Perform insertion of str at index position of source.
       'F':  8. Get the substring to search for (str).
             9. Find the position of str in source.
            10. Report position.
```

FIGURE 9.19 Structure Chart for Text Editor Program



```
Otherwise:
        11. Display error message.
```

Function `do_edit` uses a function that finds the position of one string in another (`pos`) for steps 3 and 9, a function that deletes a certain number of characters from a string (`delete`) for step 4, and a function that inserts one string in another (`insert`) for step 7.

IMPLEMENTATION

Figure 9.20 shows a complete implementation of the text editor, and Fig. 9.21 shows a sample run. Read the helper functions `pos`, `insert`, and `delete` carefully as examples of functions that use C string library functions.

FIGURE 9.20 Text Editor Program

```
/*
 *  Performs text editing operations on a source string
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_LEN   100
#define NOT_FOUND -1

char *delete(char *source, int index, int n);
char *do_edit(char *source, char command);
char  get_command(void);
char *insert(char *source, const char *to_insert, int index);
int   pos(const char *source, const char *to_find);

int
main(void)
{
      char source[MAX_LEN], command;
```

(continued)

FIGURE 9.20 (continued)

```
      printf("Enter the source string:\n> ");
      gets(source);

      for (command = get_command();
           command != 'Q';
           command = get_command()) {
          do_edit(source, command);
          printf("New source: %s\n\n", source);
      }

      printf("String after editing: %s\n", source);
      return (0);
}

/*
 *  Returns source after deleting n characters beginning with source[index].
 *  If source is too short for full deletion, as many characters are
 *  deleted as possible.
 *  Pre:  All parameters are defined and
 *        strlen(source) - index - n < MAX_LEN
 *  Post:  source is modified and returned
 */
char *
delete(char *source,  /* input/output - string from which to delete part */
       int   index,   /* input - index of first char to delete          */
       int   n)       /* input - number of chars to delete              */
{
      char rest_str[MAX_LEN];  /* copy of source substring following
                                  characters to delete */

      /*  If there are no characters in source following portion to
          delete, delete rest of string */
      if (strlen(source) <= index + n) {
            source[index] = '\0';

      /*  Otherwise, copy the portion following the portion to delete
          and place it in source beginning at the index position       */
      } else {
            strcpy(rest_str, &source[index + n]);
```

(continued)

FIGURE 9.20 (continued)

```
            strcpy(&source[index], rest_str);
      }

      return (source);
}

/*
 *  Performs the edit operation specified by command
 *  Pre:  command and source are defined.
 *  Post: After scanning additional information needed, performs a
 *        deletion (command = 'D') or insertion (command = 'I') or
 *        finds a substring ('F') and displays result; returns
 *        (possibly modified) source.
 */
char *
do_edit(char *source,  /* input/output - string to modify or search */
        char  command) /* input - character indicating operation    */
{
      char str[MAX_LEN];  /* work string */
      int  index;

      switch (command) {
      case 'D':
            printf("String to delete> ");
            gets(str);
            index = pos(source, str);
            if (index == NOT_FOUND)
                  printf("'%s' not found\n", str);
            else
                  delete(source, index, strlen(str));
            break;

      case 'I':
            printf("String to insert> ");
            gets(str);
            printf("Position of insertion> ");
            scanf("%d", &index);
            insert(source, str, index);
            break;
```

(continued)

FIGURE 9.20 (continued)

```
      case 'F':
            printf("String to find> ");
            gets(str);
            index = pos(source, str);
            if (index == NOT_FOUND)
                  printf("'%s' not found\n", str);
            else
                  printf("'%s' found at position %d\n", str, index);
            break;

      default:
            printf("Invalid edit command '%c'\n", command);
      }

      return (source);
}

/*
 *  Prompt for and get a character representing an edit command and
 *  convert it to uppercase. Return the uppercase character and ignore
 *  rest of input line.
 */
char
get_command(void)
{
      char command, ignore;

      printf("Enter D(Delete), I(Insert), F(Find), or Q(Quit)> ");
      scanf(" %c", &command);

      do
         ignore = getchar();
      while (ignore != '\n');

      return (toupper(command));
}

/*
 *  Returns source after inserting to_insert at position index of
 *  source. If source[index] doesn't exist, adds to_insert at end of
 *  source.
```

(continued)

FIGURE 9.20 (continued)

```
 *  Pre:  all parameters are defined, space available for source is
 *        enough to accommodate insertion, and
 *        strlen(source) - index - n < MAX_LEN
 *  Post: source is modified and returned
 */
char *
insert(char       *source,    /* input/output - target of insertion */
       const char *to_insert, /* input - string to insert           */
       int         index)     /* input - position where to_insert
                                         is to be inserted           */
{
      char rest_str[MAX_LEN]; /* copy of rest of source beginning
                                 with source[index] */

      if (strlen(source) <= index) {
            strcat(source, to_insert);
      } else {
            strcpy(rest_str, &source[index]);
            strcpy(&source[index], to_insert);
            strcat(source, rest_str);
      }

      return (source);
}

/*
 *  Returns index of first occurrence of to_find in source or
 *  value of NOT_FOUND if to_find is not in source.
 *  Pre:  both parameters are defined
 */
int
pos(const char *source,   /* input - string in which to look for to_find */
    const char *to_find)  /* input - string to find                      */

{
      int  i = 0, find_len, found = 0, position;
      char substring[MAX_LEN];

      find_len = strlen(to_find);
```

(continued)

FIGURE 9.20 (continued)

```
     while (!found  &&  i <= strlen(source) - find_len) {
         strncpy(substring, &source[i], find_len);
         substring[find_len] = '\0';

         if (strcmp(substring, to_find) == 0)
               found = 1;
         else
               ++i;
     }

     if (found)
           position = i;
     else
           position = NOT_FOUND;

     return (position);
}
```

FIGURE 9.21 Sample Run of Text Editor Program

```
Enter the source string:
> Internet use is growing rapidly.
Enter D(Delete), I(Insert), F(Find), or Q(Quit)> d
String to delete> growing▨
New source: Internet use is rapidly.

Enter D(Delete), I(Insert), F(Find), or Q(Quit)> F
String to find> .
'.' found at position 23
New source: Internet use is rapidly.

Enter D(Delete), I(Insert), F(Find), or Q(Quit)> I
String to insert>  ▨expanding
Position of insertion> 23
New source: Internet use is rapidly expanding.

Enter D(Delete), I(Insert), F(Find), or Q(Quit)> q
String after editing: Internet use is rapidly expanding.
```

TESTING

Choose test cases that check various boundary conditions as well as middle-of-the-road data. For example, to check the Delete command, try to delete the first few characters of `source`, the last few, and a substring in the middle of `source`. Also try a substring that appears more than once to verify that only the first occurrence is deleted. Attempt two impossible deletions, one of a substring that does not resemble any part of `source`, and one of a substring that matches a part of `source` except for its last character. Also test insertions at the beginning of `source`, exactly at the end of `source`, at a position several characters beyond the end of `source`, and in the middle of `source`. Use the Find command to look for all of `source`, single-letter pieces of `source` from the beginning, middle, and end, and multiple-character substrings from the beginning, middle, and end of the `source` string. Be sure to look for a substring not present in `source`, too.

9.9 Common Programming Errors

When manipulating string variables, the programmer must use great care in the allocation and management of memory. When we work with numeric values or single characters, we commonly compute a value of interest in a function, storing the result value temporarily in a local variable of the function until it is returned to the calling module using the `return` statement. One cannot use this approach in string functions, for such functions do not actually return a string *value* in the same way that an `int` function returns an integer value. Rather a string function returns the *address* of the initial character of a string. If we were to use the same strategy in a string function as we do in many numeric functions, we would build our result string in a local variable and return the address of the new string as the function value. The problem with this approach is that the function's data area is deallocated as soon as the `return` statement is executed, so it is not valid to access from the calling module the string the function constructed in its own local variable. Figure 9.22 shows a poor rewrite of the `scanline` function from Fig. 9.15. Rather than requiring the calling function to provide space in which to construct the function result as the earlier `scanline` did (and as is the practice of the functions in the C string library), this faulty `scanline` returns a string built in local storage. As a consequence, the string that the `printf` function tries to print is in a section of memory that neither `main` nor `printf` has any legitimate right to access and that may be overwritten with new values at any moment. What makes this type of error particularly grievous is that on some C implementations it will compile and pass unit testing without creating any error in the output.