

Chapter 9

Strings

Problem Solving and Program Design in C
by Jeri R. Hanly and Elliot B. Koffman

Pei-yih Ting @NTU

Outline

- String **Basics**
- Library **Functions**: Assignment and Substrings
- Longer Strings: **Concatenation** and Whole-Line Input
- String **Comparison**
- Arrays of **Pointers**
- **Character Operations**
- String – Integer Conversion
- Case Study: Text Editor
- Common Programming Errors

Introduction

- Typical usages of computers
 - Processing numerical data & **textural** data.
 - Typewriter → word processing systems → preprint system
- **Strings:**
 - A **data structure** deals with a group of characters
 - In C: **null ('\0') terminated array of characters**
- **Storage of strings:**
 - arrays of type **char**, e.g. `char message[100];`
- Example: string constants used till now

```
printf("Hello World!\n");  
#define ERR_Message "Error!!"
```

Defining & Initializing Strings

- Since string is an array, the declaration of a string is the same as declaring a char array.

```
char buffer[30];
```

```
char message[20] = "Initial value";
```



```
char message[] = {'I', 'n', 'i', 't', 'i', 'a', 'l', ' ', 'v', 'a', 'l', 'u', 'e', '\0'};
```

- A string is always ended with a null character '\0' for compatibility with library functions.
- Characters after the null character are **ignored** by all predefined string processing functions.

Example Application

- Asking the user for an input file name

```
char filename[30];
```

```
FILE *fp;
```

```
...
```

```
printf("Please specify the input file name:");
```

```
scanf("%s", filename);
```

```
fp = fopen(filename, "r");
```

```
if (fp == 0) {
```

```
    printf("Error in opening the file %s!!", filename);
```

```
    exit(1);
```

```
}
```

Arrays of Strings

- An array of strings is defined as **an array in which each element is an array of characters** (i.e. a two-dimensional array of characters).
- For example
 - ❑ `#define NUM_PEOPLE 30`
 - ❑ `#define NAME_LEN 25`
 - ❑ `char names [NUM_PEOPLE][NAME_LEN];`

 - ❑ `char months[12][10] = { "January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December" };`

String Output with **printf**

- The placeholder **%s** indicates string arguments in `printf` and `scanf`.

```
printf("Topic: %s\n", title_str);
```

- The formatted string output is **right-justified** by specifying a positive number in the placeholder.

```
printf("%8s", str);
```

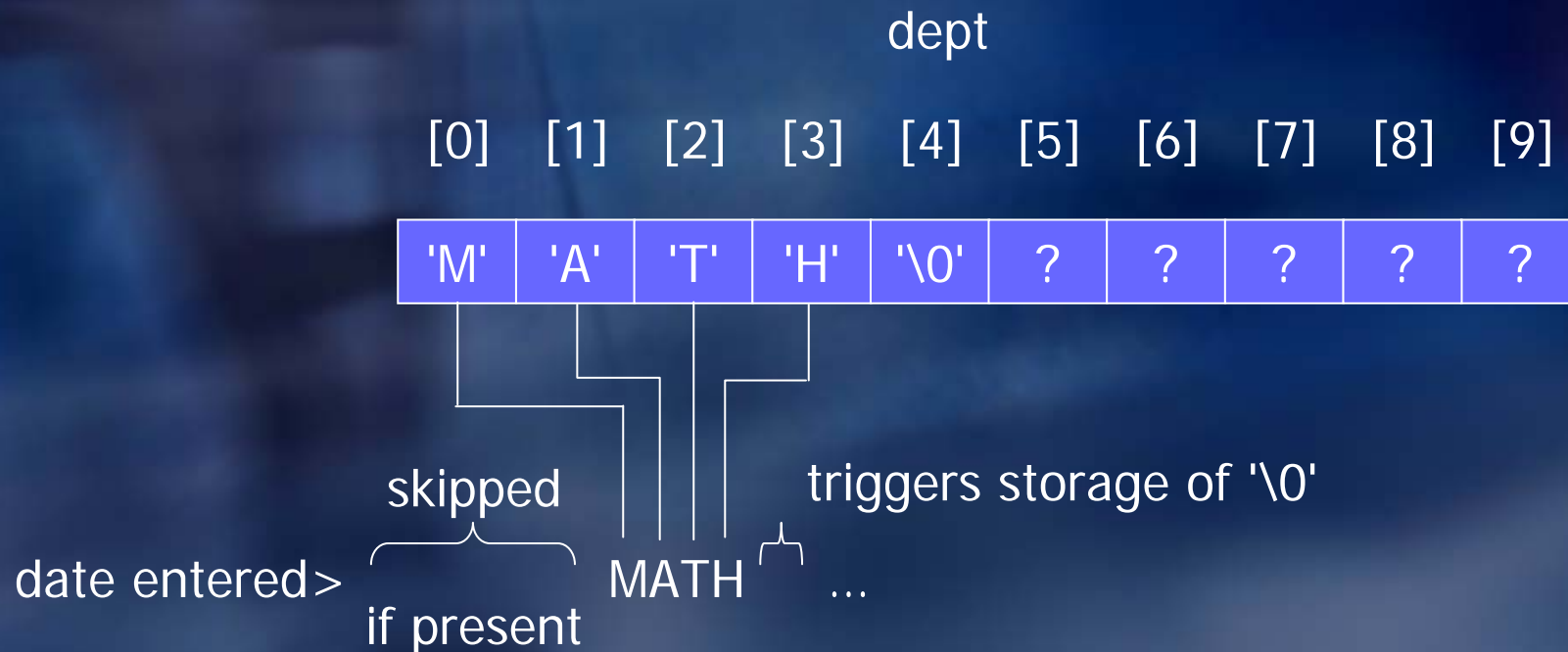
- The formatted string output is **left-justified** by specifying a negative number in the placeholder.

```
printf("%-8s", str);
```

- If the actual string is longer than the width, the displayed field is expanded with no padding.

String Input with **scanf**

- `char dept[10];`
`scanf("%s", dept);` **no & necessary**
- e.g., if the user types "MATH 1234 TR 1800," the string "MATH" along with '\0' is stored into **dept**.
- Whenever encountering a **white space**, the scanning stops and **scanf** places the null character at the end of the string.



Entry of Invalid Data

```
char dept[10];  
scanf("%s%d%s%d", dept, &course_num,  
      days, &time);
```

➤ MATH,1270,TR,1800

dept

space not assigned to **dept**

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

'M'	'A'	'T'	'H'	','	'1'	'2'	'7'	'0'	','	'T'	'R'	','	'1'	'8'	'0'	'0'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------

➤ Correction

▣ MATH 1270 TR 1800

String Library Functions

- The string **cannot** be copied by the assignment operator
`str = "Test String";` --- incorrect
- Initialization is OK
`char str[] = "Test String";`
- String manipulating library "string.h" (Appendix B)
- Array identifier is a constant pointer. A string constant is treated like a constant pointer to char constant.

```
void fun(char *s)
{
    ...
}
```

or

```
void fun(char s[]) { ... }
```

calling the function

```
fun("April");
```



String Library Functions

string.h

Function	Purpose	Parameters
strcpy	strcpy(s1, "hello");	char *dest const char *source
strncpy	strncpy(s2, "inevitable", 5); does not add '\0' to s2	char *dest const char *source size_t n
strcat	strcat(s1, "and more");	char *dest const char *source
strncat	strncat(s1, "and more", 5); adding '\0' if necessary	char *dest const char *source size_t n

h e l l o \0 ? ?

i n e v i ? ? ?

h e l l o a n d

m o r e \0 ? ?

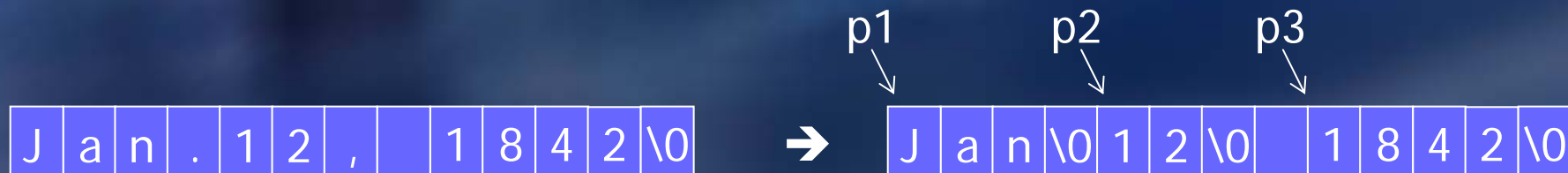
h e l l o a n d

m \0 ? ? ?

return types are all char* for the usage like
printf("%s", strcat(s1, "and more"));

String Library (cont'd)

Function	Purpose	Parameters	Return type
<code>strcmp</code>	if (<code>strcmp(n1, n2) > 0</code>) string n1 is larger than n2	<code>const char *s1</code> <code>const char *s2</code>	<code>int</code>
<code>strncmp</code>	if (<code>strncmp(n1, n2, 12) == 0</code>)	<code>const char *s1</code> <code>const char *s2</code>	<code>int</code>
<code>strlen</code>	<code>strlen("What")</code> returns 4	<code>const char *s</code>	<code>size_t</code>
<code>strtok</code>	let s1 be "Jan.12, 1842" <code>p1 = strtok(s1, "., ");</code> <code>p2 = strtok(NULL, "., ");</code> <code>p3 = strtok(NULL, "., ");</code>	<code>const char *source</code> <code>const char *delim</code>	<code>char *</code>



String Assignment **strcpy**

- Function **strcpy** copies the string in the second argument into the first argument.

```
char one_str[20];  
strcpy(one_str, "test string");
```

The **null** character is appended at the end automatically.

- If source string is longer than the destination string, the overflow characters may occupy the memory space used by other variables.
- BAD: The values of these other variables change spontaneously and unexpectedly. On rare occasions, such overflow would generate a run-time error message.

```
strcpy(one_str, "A very long test string");
```

strcpy vs. strncpy

- Function **strncpy** copies the string by specifying the number of characters to copy – safer than **strcpy**.
- If source string is **longer** than the destination string, the overflow characters are discarded automatically.

```
strncpy(dest, source, dest_len-1);  
dest[dest_len-1] = '\0'; // = 0 is the same  
// must place the null char. manually.
```

Substring Manipulation

- Frequently need to reference a **substring** of a longer character string.
- Use **strncpy** to copy a middle or an ending portion of a string.

char *strncpy(char *dest, const char *source, size_t n)

- Example:

```
char results1[10], results2[3];  
char s1[14] = "Jan. 30, 1996";
```

```
strncpy(results1, s1, 9);
```

```
results1[9] = '\0';
```

```
strncpy(results2, &s1[5], 2);
```

```
results2[2] = '\0';
```

s1
Jan. 30, 1996 \0 ?

results1
Jan. 30, ??

results2
30 ??

Separate Compounds into Components

- assuming that each element name begins with a capital letter and the first character is a capital letter

Figure 9.7

```
01 #include <stdio.h>
02 #include <string.h>
03 #define CMP_LEN 30 /* size of string to hold a compound */
04 #define ELEM_LEN 10 /* size of string to hold a component */
05 int main(void)
06 { /* string representing a compound */
07     char compound[CMP_LEN];
08     char elem[ELEM_LEN]; /* one elemental component */
09     int first, next;
10
11     /* Gets data string representing compound */
12     printf("Enter a compound> ");
13     scanf("%s", compound);
14
15     /* Displays each elemental component. These are identified
16        by an initial capital letter. */
17     first = 0;
```


Separate Compounds (cont'd)

```
18   for (next = 1; next < strlen(compound); ++next)
19       if (compound[next] >= 'A' && compound[next] <= 'Z') {
20           strncpy(elem, &compound[first], next - first);
21           elem[next - first] = '\0';
22           printf("%s\n", elem);
23           first = next;
24       }
25
26   /* Displays the last component */
27   printf("%s\n", strcpy(elem, &compound[first]));
28
29   return (0);
30 }
```

```
Enter a compound> H2SO4
H2
S
O4
```

String Length

- When writing a string-manipulating program, one usually does not know in advance the sizes of the strings used as data.
- Function **strlen** is often used to calculate the length of a string (i.e., the number of characters before the first null character).

```
char dest[10] = "Hello";  
strncat(dest, " more",  
        sizeof(dest)-1-strlen(dest));  
dest[sizeof(dest)-1] = '\0';
```

- **empty string** - a string of length zero

String Concatenation

➤ Functions **strcat** and **strncat** concatenate the first string argument with the second string argument, **strncat** adds null character if necessary.

➤ Example:

```
char f1[15] = "John ",  
      f2[15] = "Jacqueline ",  
      last[15] = "Kennedy";  
strcat(f1, last);  
strcat(f2, last);    /* invalid overflow */  
strncat(f2, last, 3);
```

Characters vs. Strings

- The representation of a char (e.g., 'Q') and a string (e.g., "Q") is essentially different.
 - ▣ A string is an array of characters ended with the null character '\0'.



Character 'Q'



String "Q"

String Comparison

- Suppose there are two strings, `str1` and `str2`.
 - ❑ The condition `str1 < str2` compare the **initial memory address** of `str1` and of `str2` --- nonsense
- The comparison between two strings is done by comparing each corresponding character in them.
 - ❑ The characters are compared based on their ASCII encoding.
 - ❑ "`thrill`" < "`throw`" since 'i' < 'o' lexicographic order
 - ❑ "`joy`" < "`joyous`" since "`joy`" is a substring of "`joyous`"
- The standard string comparison uses the **`strcmp`** and **`strncmp`** functions.

String Comparison

Relationship	Returned Value	Example
<code>str1</code> is less than <code>str2</code>	Negative	"Hello" is less than "Hi"
<code>str1</code> equals <code>str2</code>	0	"Hi" equals "Hi"
<code>str1</code> is greater than <code>str2</code>	Positive	"Hi" is larger than "Hello"

e.g., we can check if two strings are different by

```
if (strcmp(str1, str2) != 0)  
    printf("The two strings are different!");
```

Sentinel-Controlled Loop for String Input

```
01 printf("Enter list of words on as many lines as you like.\n");
02 printf("Separate words by at least one blank.\n");
03 printf("When done, enter %s to quit.\n", SENT);
04
05 for (scanf("%s", word);
06      strcmp(word, SENT) != 0;
07      scanf("%s", word)) {
08     /* process word */
09     ...
10 }
```

Figure 9.10

Numeric vs. String Selection Sort

Numeric	String
Comparison (in function that finds index of the smallest among remaining elements)	
if (list[i] < list[first]) first = i;	if (strcmp (list[i], list[first]) < 0) first = i;
Exchange of elements	
temp = list[index_of_min]; list[index_of_min] = list[fill]; list[fill] = temp;	strcpy (temp, list[index_of_min]); strcpy (list[index_of_min], list[fill]); strcpy (list[fill], temp);

Data Areas

1. `strcpy(temp, list[index_of_min]);`
2. `strcpy(list[index_of_min], list[fill]);`
3. `strcpy(list[fill], temp);`

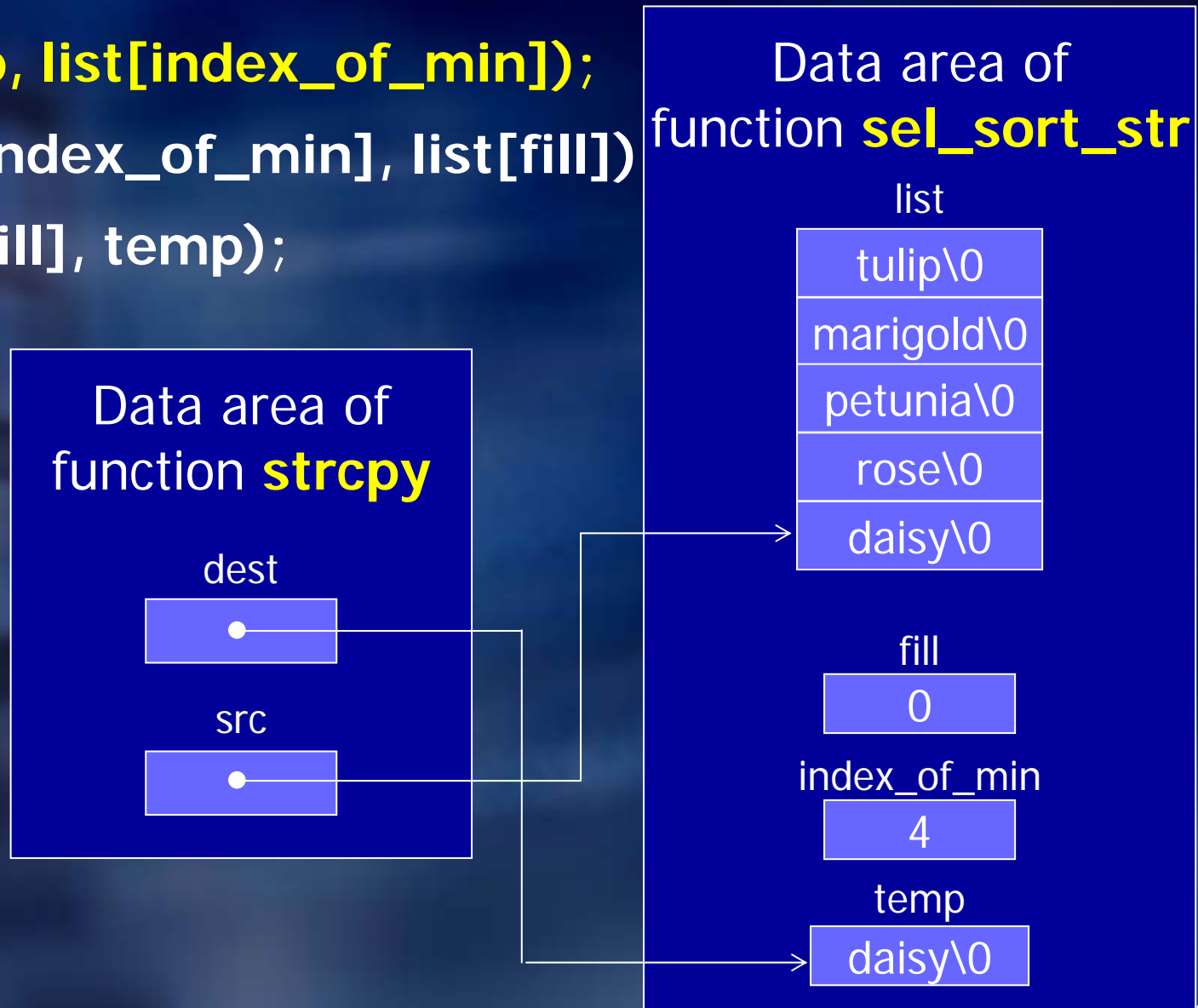


Figure 9.12

I/O of Characters and Strings

- The **stdio** library provides **getchar** function which gets the next character from the standard input.
 - ❑ **ch = getchar();** is the same as
scanf("%c", &ch);
 - ❑ Other functions: **putchar, gets, puts.**
- For IO from/to the file, the **stdio** library also provides corresponding functions.
 - ❑ **getc**: reads a character from a file.
 - ❑ Other functions: **putc, fgets, fputs.**
- Unbuffered console IO: **conio** library
 - ❑ **getch**: read a char from keyboard instantly (does not require an additional **enter** key)

Scanning a Whole Line

➤ **gets**

- ❑ Interactive input of one complete line of data

➤ Consider this code fragment:

```
char line[80];  
printf("Type in a line of data.\n> ");  
gets(line);
```

➤ The value stored in the array **line** would be

H	e	r	e		i	s		a		s	h	o	r	t		s	e	n	t	e	n	c	e	\0
---	---	---	---	--	---	---	--	---	--	---	---	---	---	---	--	---	---	---	---	---	---	---	---	----

- ## ➤ The **\n** character representing the <return> or <enter> key pressed at the end of the sentence is **NOT** stored when using **gets()**.

Implementation of **scanline** Using **getchar**

```
01 /*
02 * Gets one line of data from standard input. Returns an empty string on
03 * end of file. If data line will not fit in allotted space, stores
04 * portion that does fit and discards rest of input line.
05 */
06 char *
07 scanline(char *dest, /* output - destination string */
08 int dest_len) /* input - space available in dest */
09 {
10     int i, ch;
11     /* Gets next line one character at a time. */
12     i = 0;
13     for (ch = getchar();
14         ch != '\n' && ch != EOF && i < dest_len - 1;
15         ch = getchar())
16         dest[i++] = ch;
17     dest[i] = '\0';
18     /* Discards any characters that remain on input line */
19     while (ch != '\n' && ch != EOF)
20         ch = getchar();
21     return (dest);
22 }
```

Figure 9.15

Scanning a Whole Line (cont'd)

➤ **fgets**

□ Three arguments

1. Output parameter: string buffer
2. Maximum number of characters to store (e.g. n)
 - Never store more than n-1 characters
 - Final character stored will always be '\0'
3. File pointer to the data source stream

□ If it has room to store entire line, it will include '**\n**' before '**\0**', which is different from gets()

□ Return **0** means that it encounters the end of file (use **feof()** to test)

□ **gets()** is unsafe. Some compilers will warn you.

Scanning a Whole Line (cont'd)

```
01 #include <stdio.h>
02 #include <string.h>
03 #define LINE_LEN 80
04 #define NAME_LEN 40
05 int main(void) {
06     char line[LINE_LEN], inname[NAME_LEN], outname[NAME_LEN];
07     FILE *inp, *outp;
08     char *status;
09     int i = 0;
10     printf("Name of input file> ");
11     scanf("%s", inname);
12     printf("Name of output file> ");
13     scanf("%s", outname);
14     inp = fopen(inname, "r");
15     outp = fopen(outname, "w");
16     for (status = fgets(line, LINE_LEN, inp);
17         status != 0;
18         status = fgets(line, LINE_LEN, inp)) {
19         if (line[strlen(line) - 1] == '\n')
20             line[strlen(line) - 1] = '\0';
21         fprintf(outp, "%3d>> %s\n\n", ++i, line);
22     }
23     return (0);
24 }
```

Figure 9.8

Scanning a Whole Line (cont'd)

Input file

In the early 1960s, designers and implementers of operating systems were faced with a significant dilemma. As people's expectations of modern operating systems escalated, so did the complexity of the systems themselves. Like other programmers solving difficult problems, the systems programmers desperately needed the readability and modularity of a powerful high-level programming language.

Output file

```
1>> In the early 1960s, designers and implementers of operating
2>> systems were faced with a significant dilemma. As people's
3>> expectations of modern operating systems escalated, so did
4>> the complexity of the systems themselves. Like other
5>> programmers solving difficult problems, the systems
6>> programmers desperately needed the readability and
7>> modularity of a powerful high-level programming language.
```

Array of Pointers

```
➤ char original[][9] = {"tulip", "marigold", "petunia", "rose", "daisy"};  
char *alphap[5];  
// char ** const alphap = (char **) malloc(5*sizeof(char *));
```

alphap[?] vs. original[?]

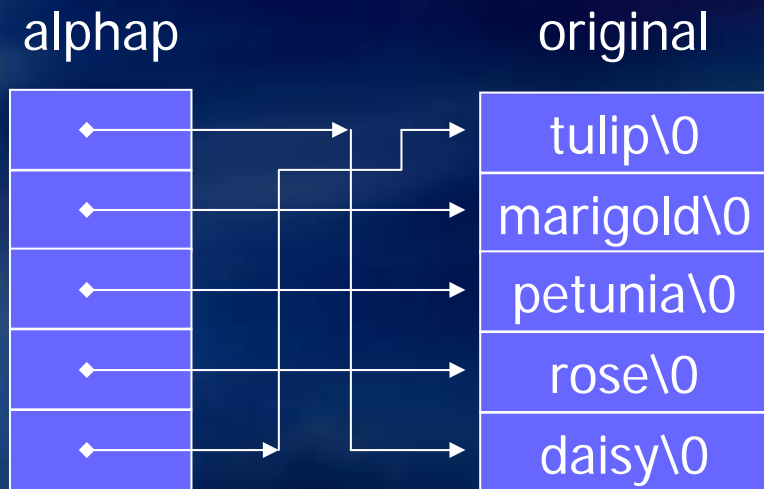


Figure 9.13

- alphap[i] refers to the same string as original[i].
- An array of pointers might represent a 2nd ordering
 - ❑ Pointer requires less space than a whole string
 - ❑ Executes faster
 - ❑ Accesses or modifications made through **original[0]** or **alphap[4]** are exactly the same.

2nd Ordering of a List

```
001 /*
002  * Maintains two orderings of a list of applicants: the original
003  * ordering of the data, and an alphabetical ordering accessed through an
004  * array of pointers.
005  */
006
007 #include <stdio.h>
008 #define STRSIZ 30 /* maximum string length */
009 #define MAXAPP 50 /* maximum number of applications accepted */
010
011 int alpha_first(char *list[], int min_sub, int max_sub);
012 void select_sort_str(char *list[], int n);
013
014 int main(void)
015 {
016     char applicants[MAXAPP][STRSIZ]; /* list of applicants in the */
017     char *alphap[MAXAPP];           /* order in which they applied */
018     int num_app,                    /* list of pointers to applicants */
019         i;                          /* actual number of applicants */
020     char one_char;
```

Figure 9.14

2nd Ordering of a List (cont'd)

```
022  /* Gets applicant list */
023  printf("Enter number of applicants (0 . . %d)\n> ", MAXAPP);
024  scanf("%d", &num_app);
025  do /* skips rest of line after number */
026      scanf("%c", &one_char);
027  while (one_char != '\n');
028
029  printf("Enter names of applicants on separate lines\n");
030  printf("in the order in which they applied\n");
031  for (i = 0; i < num_app; ++i)
032      gets(applicants[i]);
033
034  /* Fills array of pointers and sorts */
035  for (i = 0; i < num_app; ++i)
036      alphap[i] = applicants[i]; /* copies ONLY address */
037  select_sort_str(alphap, num_app);
038
039  /* Displays both lists */
040  printf("\n\n%-30s%5c%-30s\n\n", "Application Order", ' ', "Alphabetical Order");
041  for (i = 0; i < num_app; ++i)
042      printf("%-30s%5c%-30s\n", applicants[i], ' ', alphap[i]);
043
044  return(0);
045 }
```

2nd Ordering of a List (cont'd)

```
046
047 /*
048 * Finds the index of the string that comes first alphabetically in
049 * elements min_sub..max_sub of list
050 * Pre: list[min_sub] through list[max_sub] are of uniform case;
051 *       max_sub >= min_sub
052 */
053 int alpha_first(char *list[], /* input - array of pointers to strings */
054                 int min_sub, /* input - minimum and maximum subscripts */
055                 int max_sub) /* of portion of list to consider */
056 {
057     int first, i;
058
059     first = min_sub;
060     for (i = min_sub + 1; i <= max_sub; ++i)
061         if (strcmp(list[i], list[first]) < 0)
062             first = i;
063
064     return (first);
065 }
```

2nd Ordering of a List (cont'd)

```
067 /*
068 * Orders the pointers in array list so they access strings
069 * in alphabetical order
070 * Pre: first n elements of list reference strings of uniform case; n >= 0
071 */
072 void                                /* input/output - array of pointers being */
073 select_sort_str(char *list[], /* ordered to access strings alphabetically */
074                 int n)          /* input - number of elements to sort */
075 {
076     int fill, /* index of element to contain next string in order */
077         index_of_min; /* index of next string in order */
078     char *temp;
079     for (fill = 0; fill < n - 1; ++fill) {
080         index_of_min = alpha_first(list, fill, n - 1);
081         if (index_of_min != fill) {
082             temp = list[index_of_min];
083             list[index_of_min] = list[fill];
084             list[fill] = temp;
085         }
086     }
087 }
```

2nd Ordering of a List (cont'd)

Enter number of applicants (0 . . 50)

> 5

Enter names of applicants on separate lines
in the order in which they applied

SADDLER, MARGARET

INGRAM, RICHARD

FAATZ, SUSAN

GONZALES, LORI

KEITH, CHARLES

Application Order

SADDLER, MARGARET

INGRAM, RICHARD

FAATZ, SUSAN

GONZALES, LORI

KEITH, CHARLES

Alphabetical Order

FAATZ, SUSAN

GONZALES, LORI

INGRAM, RICHARD

KEITH, CHARLES

SADDLER, MARGARET

Naming a string constant

- Naming a string constant

```
char *message = "File not found";
```

or

```
char *message;
```

```
message = "File not found";
```

- **Example 1:**

```
if (some error)
```

```
    printf(message); // printf("File not found");
```

- Errors: `message[4] = ':';`

```
    strcpy(message, "hello");
```

- **Example 2:**

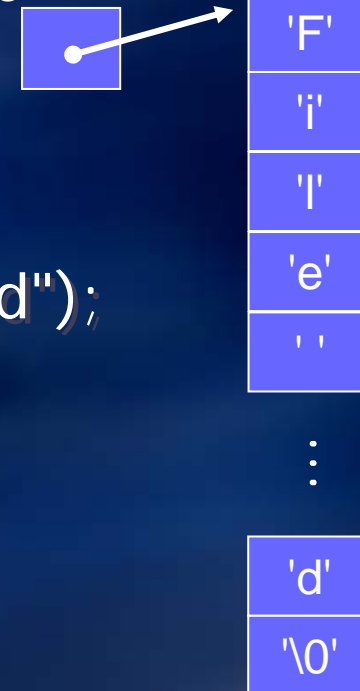
```
int result;
```

```
char *format = "Here is the result: %d";
```

```
...
```

```
printf(format, result);
```

message



Arrays of Strings

➤ C permits the use of an array of pointers to store a list of strings.

➤ For example

❑ char **month**[12][10] = **Two dimensional character array**

```
{ "January", "February", "March", "April",  
  "May", "June", "July", "August",  
  "September", "October", "November", "December" };
```

❑ char ***month**[12] = **Array of char pointers to string constants**

```
{ "January", "February", "March", "April",  
  "May", "June", "July", "August",  
  "September", "October", "November", "December" };
```

Character Analysis & Conversion

```
#include <ctype.h>
```

Facility	Checks	Example
isalpha	If argument is a letter of the alphabet	<pre>if (isalpha(ch)) printf("%c is a letter\n", ch);</pre>
isdigit	If argument is one of the ten decimal digits	<pre>dec_digit = isdigit(ch);</pre>
islower (isupper)	If argument is a lowercase (or uppercase) letter of the alphabet	<pre>if (islower(fst_let)) { printf("\nError: sentence"); printf("should begin with a"); printf("capital letter.\n"); }</pre>
ispunct	If argument is a punctuation character	<pre>if (ispunct(ch)) printf("Punctuation mark: %c\n", ch)</pre>

Table 9.3

Character Analysis & Conversion

Facility	Converts	Example
isspace	If argument is a whitespace character	<pre>c=getchar(); while (isspace(c) && c!=EOF) c=getchar();</pre>
tolower (toupper)	Its lowercase (or uppercase) letter argument to the uppercase (or lowercase) equivalent and returns this equivalent as the value of the call	<pre>if (islower(ch)) printf("Capital %c=%c\n", ch, toupper(ch))</pre>

Table 9.3

String – Number Conversion

- The `<stdlib.h>` defines some basic functions for conversion from strings to numbers:
 - ❑ `atoi("123")` converts a string to an integer.
 - ❑ `atol("123")` converts a string to a long integer.
 - ❑ `atof("12.3")` converts a string to a float.
- However, there is **no** functions such as `itoa`, `ltoa`, `ftoa`, ...etc,
 - ❑ because there is a function called `sprintf` which can convert many formats to a string.

sprintf and sscanf

- The **sprintf** function substitutes values for placeholders just as **printf** does except that it stores the result into a character array

```
int mon, day, year;  
sprintf(s, "%d%d%d", mon, day, year);
```

- The **sscanf** function works exactly like **scanf** except that it takes data from the string as its input argument.

```
int num; double val; char word[20];  
sscanf(" 85 96.2 hello", "%d%lf%s", &num, &val, word);
```

Placeholders Used with **printf**

Value	Placeholder	Outputs
'a'	%c	a
	%3c	□□a
	%-3c	a□□
-10	%d	-10
	%2d	-10
	%4d	□-10
	%-5d	-10□□
49.76	%.3f	49.760
	%.1f	49.8
	%10.2f	□□□□□49.76
	%10.3e	□4.976e+01
"fantastic"	%s	fantastic
	%6s	fantastic
	%12s	□□□fantastic
	%-12s	fantastic□□□
	%3.3s	fan

Table 9.5

max field width

%3.3s

min field width

Date Format Conversion

```
001 /*
002 * Functions to change the representation of a date from a string containing
003 * day, month name and year to three integers (month day year) and vice versa
004 */
005
006 #include <stdio.h>
007 #include <string.h>
008
009 #define STRSIZ 40
010 char *nums_to_string_date(char *date_string, int month, int day,
011                          int year, const char *month_names[]);
012 int search(const char *arr[], const char *target, int n);
013 void string_date_to_nums(const char *date_string, int *monthp,
014                        int *dayp, int *yearp, const char *month_names[]);
015
016 /* Tests date conversion functions */
017 int
018 main(void)
019 {
020     char *month_names[12] = {"January", "February", "March", "April", "May",
021                            "June", "July", "August", "September",
022                            "October", "November", "December"};
023     int m, y, mon, day, year;
024     char date_string[STRSIZ];
```

Figure 9.18

Date Format Conversion (cont'd)

```
025     for (y = 1993; y < 2010; y += 10)
026         for (m = 1; m <= 12; ++m) {
027             printf("%s", nums_to_string_date(date_string,
028                                             m, 15, y, month_names));
029             string_date_to_nums(date_string, &mon, &day, &year, month_names);
030             printf(" = %d/%d/%d\n", mon, day, year);
031         }
032     }
033     return (0);
034 }
035
036 /*
037  * Takes integers representing a month, day and year and produces a
038  * string representation of the same date.
039  */
040 char *
041 nums_to_string_date(char *date_string, /* output - string representation */
042                   int month, /* input - */
043                   int day, /* representation */
044                   int year, /* as three numbers */
045                   const char *month_names[])
046 { /* input - string representations of months */
047     sprintf(date_string, "%d %s %d", day, month_names[month - 1], year);
048     return (date_string);
049 }
```

Date Format Conversion (cont'd)

```
051 #define NOT_FOUND -1 /* Value returned by search function if target
052                        not found */
053 /*
054 * Searches for target item in first n elements of array arr
055 * Returns index of target or NOT_FOUND
056 * Pre: target and first n elements of array arr are defined and n>0
057 */
058 int
059 search(const char *arr[], /* array to search */
060        const char *target, /* value searched for */
061        int n) /* number of array elements to search */
062 {
063     int i,
064         found = 0, /* whether or not target has been found */
065         where; /* index where target found or NOT_FOUND*/
```

Date Format Conversion (cont'd)

```
067     /* Compares each element to target */
068     i = 0;
069     while (!found && i < n) {
070         if (strcmp(arr[i], target) == 0)
071             found = 1;
072         else
073             ++i;
074     }
075
076     /* Returns index of element matching target or NOT_FOUND */
077     if (found)
078         where = i;
079     else
080         where = NOT_FOUND;
081     return (where);
082 }
```


Date Format Conversion (cont'd)

```
084 /*
085  * Converts date represented as a string containing a month name to
086  * three integers representing month, day, and year
087  */
088 void
089 string_date_to_nums(const char *date_string, /* input - date to convert */
090                   int *monthp, /* output - month number */
091                   int *dayp, /* output - day number */
092                   int *yearp, /* output - year number */
093                   const char *month_names[])
094 /* input - names used in date string */
095 {
096     char mth_nam[STRSZ];
097     int month_index;
098
099     sscanf(date_string, "%d%s%d", dayp, mth_nam, yearp);
100
101     /* Finds array index (range 0..11) of month name. */
102     month_index = search(month_names, mth_nam, 12);
103     *monthp = month_index + 1;
104 }
```

```
15 January 1993 = 1/15/1993
15 February 1993 = 2/15/1993
...
15 December 2003 = 12/15/2003
```

Case Study: Text Editor

Step 1: **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.

Text Editor (cont'd)

➤ Sample run

Figure 9.21

```
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.
```

Text Editor (cont'd)

Step 2: **Analysis**

- Problem Constant
 - ◆ MAX_LEN 100
- Problem Inputs
 - ◆ char source [MAX_LEN]
 - ◆ char command
- Problem Output
 - ◆ char source [MAX_LEN]

Text Editor (cont'd)

Step 3: **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

□ Refinement And Program Structure

Local Variables

```
char str[MAX_LEN]
```

```
int index
```

Text Editor (cont'd)

Step 3: Design

□ 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

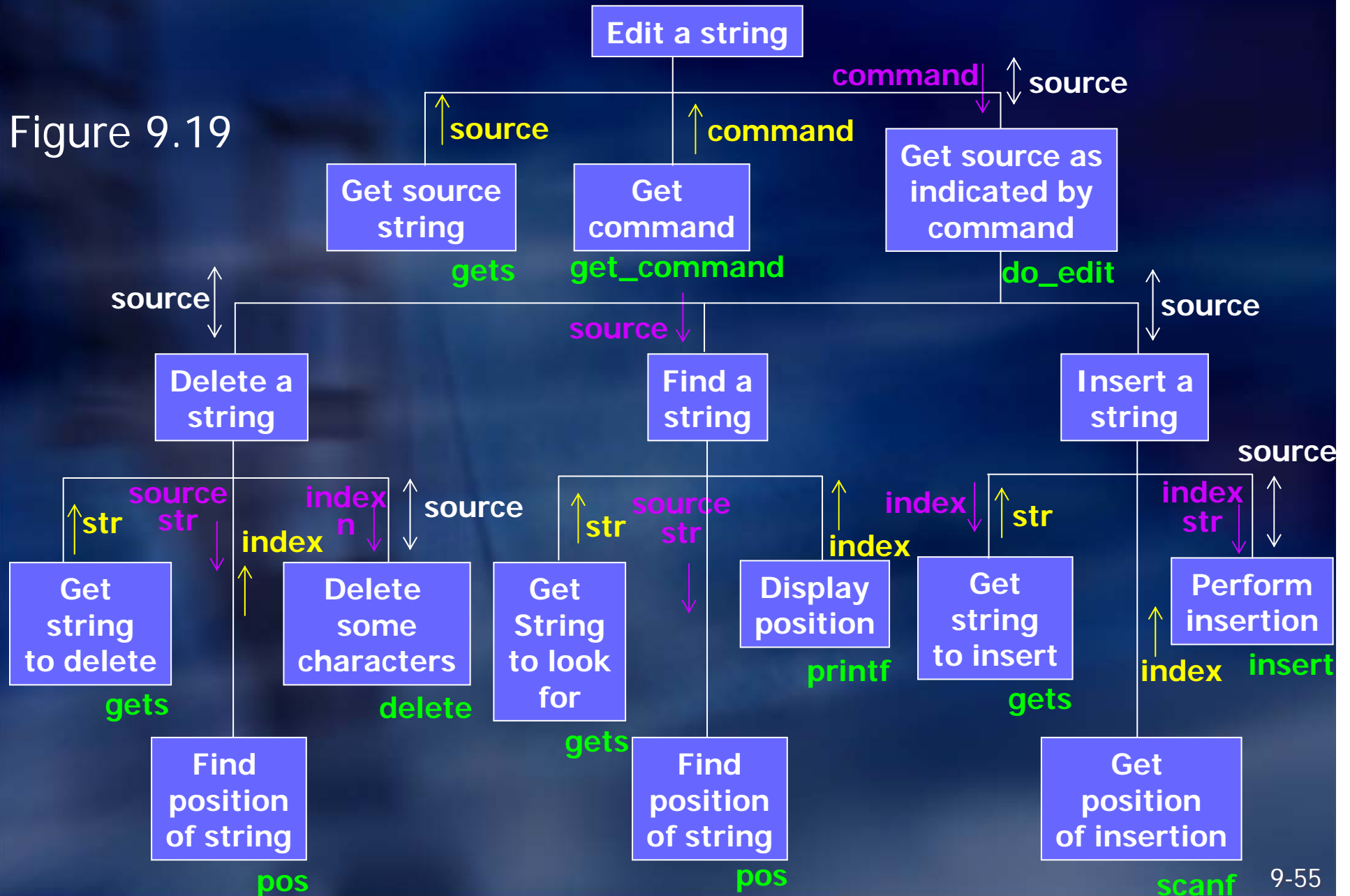
'I' : 5. Get the substring to insert (str)
 6. Get position of insertion (index)
 7. Perform insertion of str at index position of source

'F' 8. Get the substring to search for (str)
 9. Find the position of str in source
 10. Report position

default: 11. Display error message

Structure Chart

Figure 9.19



Text Editor (cont'd)

```
001 /*
002  * Performs text editing operations on a source string
003  */
004 #include <stdio.h>
005 #include <string.h>
006 #include <ctype.h>
007
008 #define MAX_LEN 100
009 #define NOT_FOUND -1
010
011 char *delete(char *source, int index, int n);
012 char *do_edit(char *source, char command);
013 char get_command(void);
014 char *insert(char *source, const char *to_insert, int index);
015 int pos(const char *source, const char *to_find);
016
017 int
018 main(void)
019 {
020     char source[MAX_LEN], command;
021
022     printf("Enter the source string:\n> ");
023     gets(source);
```

Figure 9.20

Text Editor (cont'd)

```
025     for (command = get_command();
026         command != 'Q';
027         command = get_command()) {
028         do_edit(source, command);
029         printf("New source: %s\n\n", source);
030     }
032     printf("String after editing: %s\n", source);
033     return (0);
034 }
035 /*
036  * Returns source after deleting n characters beginning with source[index].  If
037  * source is too short for full deletion, as many characters are deleted as possible.
039  * Pre:  All parameters are defined and strlen(source) - index - n < MAX_LEN
041  * Post: source is modified and returned
042  */
043 char * delete(char *source, /* input/output-string from which to delete part */
044             int index,    /* input - index of first char to delete           */
045             int n)        /* input - number of chars to delete           */
046 {
047     char rest_str[MAX_LEN]; /* copy of source substring following
048                             characters to delete */
```

Text Editor (cont'd)

```
051     /* If there are no characters in source following, delete rest of string */
052     if (strlen(source) <= index + n) {
053         source[index] = '\0';
054     /* Otherwise, copy the portion following the portion to delete
055        and place it in source beginning at the index position      */
056     } else {
057         strcpy(rest_str, &source[index + n]);
058         strcpy(&source[index], rest_str);
059     }
060     return (source);
061 }
062 /*
063 * Performs the edit operation specified by command
064 * Pre:  command and source are defined.
065 * Post: After scanning additional information needed, performs a deletion
066 *       (command = 'D') or insertion (command = 'I') or finds a substring
067 *       ('F') and displays result; returns (possibly modified) source.
068 */
069 char *do_edit(char *source, /* input/output - string to modify or search */
070             char command) /* input - character indicating operation */
071 {
072     char str[MAX_LEN]; /* work string */
073     int index;
```

Text Editor (cont'd)

```
080     switch (command) {
081     case 'D':
082         printf("String to delete> ");
083         gets(str);
084         index = pos(source, str);
085         if (index == NOT_FOUND)
086             printf("%s' not found\n", str);
087         else
088             delete(source, index, strlen(str));
089         break;
090
091     case 'I':
092         printf("String to insert> ");
093         gets(str);
094         printf("Position of insertion> ");
095         scanf("%d", &index);
096         insert(source, str, index);
097         break;
```

Text Editor (cont'd)

```
099     case 'F':
100         printf("String to find> ");
101         gets(str);
102         index = pos(source, str);
103         if (index == NOT_FOUND)
104             printf("%s' not found\n", str);
105         else
106             printf("%s' found at position %d\n", str, index);
107         break;
108
109     default:
110         printf("Invalid edit command '%c'\n", command);
111     }
112
113     return (source);
114 }
```

Text Editor (cont'd)

```
115 /*
116  * Prompt for and get a character representing an edit command and
117  * convert it to uppercase. Return the uppercase character and ignore
118  * rest of input line.
119  */
120 char
121 get_command(void)
122 {
123     char command, ignore;
124
125     printf("Enter D(Delete), I(Insert), F(Find), or Q(Quit)> ");
126     scanf(" %c", &command);
127
128     do
129         ignore = getchar();
130     while (ignore != '\n');
131
132     return (toupper(command));
133 }
```

Text Editor (cont'd)

```
134 /*
135  * Returns source after inserting to_insert at position index of source.
136  * If source[index] doesn't exist, adds to_insert at end of source.
137  *
138  * Pre: all parameters are defined, space available for source is enough to
139  *       accommodate insertion, and strlen(source) - index - n < MAX_LEN
140  *
141  * Post: source is modified and returned
142  */
143 char *insert(char *source,          /* input/output - target of insertion */
144             const char *to_insert, /* input - string to insert */
145             int index)             /* input - position where to_insert
146                                     is to be inserted */
147 {
148 {
149     char rest_str[MAX_LEN]; /* copy of rest of source beginning
150                             with source[index] */
151
152     if (strlen(source) <= index) {
153         strcat(source, to_insert);
154     } else {
155         strcpy(rest_str, &source[index]);
156         strcpy(&source[index], to_insert);
157         strcat(source, rest_str);
158     }
159
160     return (source);
161 }
```

Text Editor (cont'd)

```
162 /*
163  * Returns index of first occurrence of to_find in source or
164  * value of NOT_FOUND if to_find is not in source.
165  * Pre: both parameters are defined
166  */
167 int pos(const char *source, /* input - string in which to look for to_find */
168         const char *to_find) /* input - string to find */
169 {
170     int i = 0, find_len, found = 0, position;
171     char substring[MAX_LEN];
172     find_len = strlen(to_find);
173     while (!found && i <= strlen(source) - find_len) {
174         strncpy(substring, &source[i], find_len);
175         substring[find_len] = '\0';
176         if (strcmp(substring, to_find) == 0)
177             found = 1;
178         else
179             ++i;
180     }
181     if (found)
182         position = i;
183     else
184         position = NOT_FOUND;
185     return (position);
186 }
```

Common Programming Errors

- Figure 9.22 shows a poor rewrite of **scanline** function of Figure 9.15.
 - ❑ return the address of a deallocated variable: lifetime of a variable vs. scope of a variable

Flawed **scanline**

```
01 /*
02 * Gets one line of data from standard input. Returns an empty string on end
03 * of file. If data line will not fit in allotted space, stores portion that
04 * does fit and discards rest of input line.
05 **** Error: returns address of space that is immediately deallocated.
06 */
07 char *scanline(void)
08 {
09     char dest[MAX_STR_LEN];
10     int i, ch;
11     /* Get next line one character at a time. */
12     i = 0;
13     for (ch = getchar();
14         ch != '\n' && ch != EOF && i < MAX_STR_LEN - 1;
15         ch = getchar())
16         dest[i++] = ch;
17     dest[i] = '\0';
18     /* Discard any characters that remain on input line */
19     while (ch != '\n' && ch != EOF)
20         ch = getchar();
21     return(dest); returns address of a local variable
22 }
```

Figure 9.22

Programming Errors (cont'd)

- Misuse or neglect of the & operator
 - String is equivalent to “array of chars”, the name of the array represents the address constant of the first element of the array
- Overflow of character arrays allocated for strings.
- All strings end with the null character, '\0'