BORLAND OSBORNE McGRAW-HILL

TURBO C® PROGRAMMER'S LIBRARY

TURBO C®
PROGRAMMER'S LIBRARY

Kris Jamsa

BORLAND·OSBORNE/McGRAW·HILL
PROGRAMMING SERIES

BORLAND·OSBORNE McGraw-Hill

Turbo C® Programmer's Library

Kris Jamsa

BORLAND·OSBORNE/McGRAW·HILL
PROGRAMMING SERIES

Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne **McGraw-Hill** at the above address.

A complete list of trademarks appears on page 671.

Turbo C® Programmer's Library

1234567890 DODO 898

ISBN 0-07-881394-8

Contents

To my grandparents:

For your unmatched support, encouragement, and love.

Foreword

It is a pleasure to present—with our co-publisher Osborne/McGraw-Hill—*Turbo C Programmer's Library* for the benefit of the many users of Borland's Turbo C.

The power, flexibility, and portability of Turbo C have won this complete, interactive development environment an enthusiastic acceptance by the programming community. We have responded to that endorsement with a commitment to support professional programmers and developers in every way possible: with a technically superior product, outstanding technical and customer support services, and quality books that help them expand their uses of Turbo C.

Turbo C Programmer's Library by veteran author Kris Jamsa is, therefore, an integral piece in our Turbo C support program. Here, in one indispensable volume, are the examples of good code and the techniques programmers and developers need to develop a library of hundreds of powerful Turbo C routines. In addition, Jamsa provides insight into the development of the run-time library to help users take full advantage of the library's routines.

We recommend *Turbo C Programmer's Library* as the perfect companion for developing programs with Turbo C.

Philippe Kahn
President
Borland International, Inc.

Preface

Developing a library of Turbo C routines is one of the best ways to enhance the productivity of programmers. If you work with other programmers, keeping a library of functions will increase everyone's productivity, for several reasons. First, programmers often spend a significant amount of time developing routines that already exist in other applications. A library of routines can minimize duplication of effort because programmer has access to the routines in the library. Second, programming skills are improved through exposure to "good" coding techniques. another programmer's code provides an important opportunity for information and learning new techniques. In addition, you can standardize code and documentation and minimize errors. If you are programming at home, placing your routines into a library will greatly improve the organization of your disks.

This book provides an extensive library of Turbo C routines. Each routine was developed to simplify its integration into your application programs. This library meets the needs of both the novice and experienced Turbo C programmer. The novice can create useful programs in just minutes, and the experienced programmer can learn how to increase the flexibility of applications through memory mapping and pop-up menus. All programmers will learn how to write routines that support the DOS pipe, DOS wildcard characters, and the DOS and BIOS system services.

Turbo C Run-Time Library

For those of you who are already familiar with the Turbo C run-time library, you will be pleased to find many routines presented here for the first time. By experimenting with the routines in this book, you can add significant power to your Turbo C programs.
manipulation. In other cases, they are provided to increase your appreciation of Borland's routines and to teach you how to use them more effectively.

For those of you who are already familiar with the Turbo C run-time library, you will be pleased to find many routines presented here for the first time. By experimenting with the routines in this book, you can add significant power to your Turbo C programs.

Development Philosophy

This book was written with two major goals. First, the routines had to offer new capabilities to Turbo C programmers at all levels of experience. For the novice, this book offers valuable information and an opportunity to extend his or her knowledge of Turbo C. To meet the diverse needs of advanced programmers, it offers routines for dynamic memory manipulation, pop-up menus, memory-mapped I/O, and support for DOS wildcard characters.

The second (and more important) goal was to illustrate good programming practice. Thus, each routine presented in this text has the following attributes:

- Complete documentation
- Consistent usage
- Well-structured code
- Thorough error detection
- Restriction of side effects

As you examine the routines in this text, you should note their consistent documentation. Since you must examine hundreds of routines, consistent documentation is more important than you may at first realize. You will also note that the code is quite structured. There are no goto statements, and, when applicable, functions have only one entry and exit point. Turbo C prototypes have been used extensively throughout the text to help the compiler locate as many errors as possible. If you have not yet developed your own programming standards, use the ones in this text for your foundation.

Chapter Contents

This text assumes that you are familiar with, or are in the process of learning, Turbo C. It is not intended to be a tutorial on Turbo C.

Chapter 1 provides you with an overview of the Turbo C run-time library, introduces C function prototyping, and discusses the conventions used throughout the text.

Chapter 2 is a detailed presentation of string manipulation. Several of the functions normally found in the Turbo C run-time library are enhanced to provide additional functionality.

Chapter 3 discusses string manipulation using pointers. Many of the routines presented in Chapter 2 are greatly simplified by the use of pointers. Because string manipulation is common in Turbo C, this chapter is critical to understanding the language.

Chapter 4 examines recursion. Simply stated, a recursive function invokes itself to perform a specific task.

Chapter 5 shows how to develop Turbo C programs that support the DOS pipe and I/O redirection.

Chapter 6 introduces the DOS system services. All of the commands that you normally issue from the DOS prompt (such as those used for subdirectory manipulation) can be called by your Turbo C programs. This chapter teaches you how to get the most from DOS.

Chapter 7 presents the BIOS system services that perform the basic input/output services for your computer. You can gain considerable flexibility by using the BIOS for I/O processing instead of standard Turbo C functions.

Chapter 8 introduces the ANSI driver that is available to your Turbo C programs once ANSI.SYS is installed during system startup. The ANSI driver provides enhanced screen output capabilities along with keyboard redefinition.

Chapter 9 demonstrates advanced file-manipulation techniques with Turbo C. You will learn how to support DOS wildcard characters as well as multiple command-line parameters.

Chapter 10 presents array-manipulation routines. It also presents several techniques including the use of macro procedures, to help you keep your routines as generic as possible.

Chapter 11 demonstrates several sorting and searching algorithms. You will learn the bubble, selection, Shell, and quick sorts as well as use sequential and binary searches.

Chapter 12 examines advanced I/O routines. You will develop routines that prompt for and validate integer, floating, and character string values.

Chapter 13 looks at dynamic memory manipulation. You will learn to program singly linked lists, doubly linked lists, and binary trees.

Chapter 14 presents mapped video in Turbo C. Because many of the routines update the video display, the chapter presents two assembly language routines that synchronize video memory references to the horizontal screen refresh.

Chapter 15 examines menu manipulation. It includes several routines that work for essentially any menu, and discusses video pop-up menus.

Appendix A provides an ASCII chart. Appendix B provides the calling sequence and notes for each of the routines in the Turbo C run-time library.

Disk Packages

There are thousands of lines of code in this book. All the routines are presented in their entirety, so you can type them at your computer as you need them. To save you time and testing, a disk package containing all of the routines in this book is available for $39.95 plus shipping and handling.

The Turbo C Help disk package provides you with on-line help for Turbo C statements, reserved words, and constructs, as well as the complete calling sequence and notes on each Turbo C run-time library routine. This package allows you to put your Turbo C documentation back on the shelf. Turbo C Help is available for $29.95 plus shipping and handling.

To order these packages, use the coupons on the following page.

Please send me the disk package for *Turbo C Programmer's Library.* My payment of $42.45 (39.95 plus $2.50 shipping and handling) is enclosed. (For orders to Canada and Europe, please include $7.50 for shipping and handling ($47.45).

_________ check
_________ money order

Name ____
Address ______________________________________
City ______ State ______ ZIP ______________________

Kris Jamsa Software, Inc. Box 26031 Las Vegas, Nevada 89126

This is solely the offering of Kris Jamsa Software, Inc. Osborne/ McGraw-Hill takes no responsibility for the fulfillment of this order.

Please send me Turbo Help on disk. My payment for $32.45 ($29.95 plus $2.50 shipping and handling) is enclosed. For orders to Canada and Europe, please include $5.00 for shipping and handling ($34.95).

_________ check
_________ money order

Name ____
Address ______________________________________
City ______ State ______ ZIP ______________________

Kris Jamsa Software, Inc. Box 26031 Las Vegas, Nevada 89126

This is solely the offering of Kris Jamsa Software, Inc. Osborne/McGraw-Hill takes no responsibility for the fulfillment of this order.

CHAPTER

1

Getting Started with the Turbo C Library

This book was written to save you time—development time, coding time, and testing time—as you write your Turbo C programs. This book provides you with a library (or, specifically, a collection of routines) that you can use to complete your Turbo C programs. Because the routines in this text are already written, you can simply insert them into your Turbo C programs, and, because each routine has been thoroughly tested, you can greatly reduce the testing time normally associated with program development. If you are new to Turbo C, DOS, or the IBM PC, these routines can teach you a great deal about these topics. By examining the source code presented here, your Turbo C programs should improve. Considerable time and effort has been spent to maintain the readability, modifiability, and generic characteristics of each routine.

As you progress through each chapter, keep in mind that these routines are only the start of your Turbo C library. Build on these routines and you will find that your library of Turbo C functions never seems to stop growing. Feel free to modify any of these routines to meet your individual needs. Only by experimenting with each function can you fully understand its processing. Libraries exist to make your programs easier to develop. Programming in Turbo C should be easy and fun.

Turbo C Run-Time Library

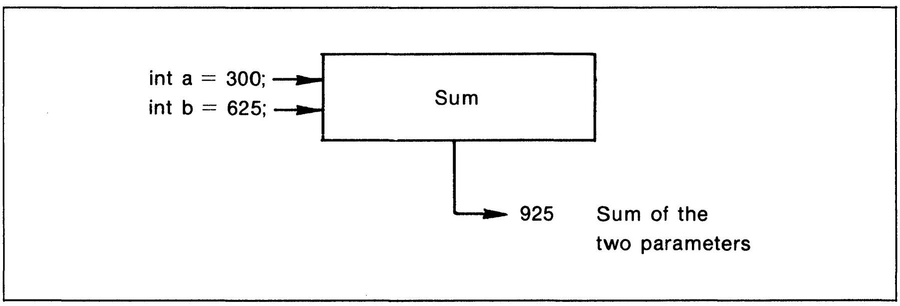
A major function of any programming library is to reduce the duplication of code. After all, if someone else has written code to perform a specific task, why reinvent the wheel? Borland International, Inc., provides you with a powerful collection of routines called the *run-time library* that you should use whenever possible. Borland developed Turbo C and employs many of the true Turbo C experts. All of the routines in the run-time library are well written and highly optimized. You should spend considerable time becoming familiar with these routines. Each of these routines is listed in Appendix B. The time you spend now becoming conversant with the run-time library routines will save you much more time in the future.

In some cases, you may wonder why routines in this text appear to duplicate functions in the run-time library. In most cases, the answer is simply for instruction. In the case of strings, Turbo C provides a powerful collection of string-manipulation routines in the run-time libraries. Because strings are so widely used, you should fully understand string manipulation. The only way to accomplish this is by examining source code. Without source code for these routines, you could never modify them to meet your individual needs. By examining the routines in this text, you will gain a much better understanding of the Turbo C run-time library.

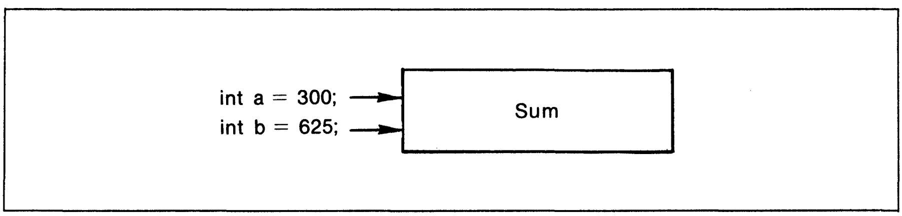
If you are performing serious Turbo C development, you should strongly consider purchasing the run-time library source code from Borland. This source code provides excellent examples of how to get the most from Turbo C.

Routine Presentation

All of the routines in this text are presented in the same fashion: first pictorially and then in source code. For example, consider the routine sum that receives two values and returns their sum. This routine is presented pictorially as follows:

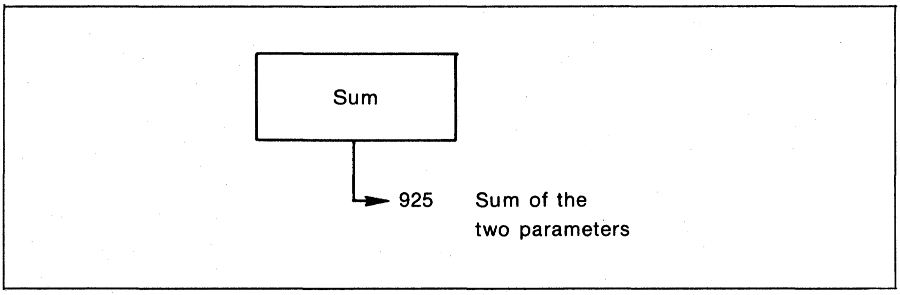


First, note the two variables passed to the routine:

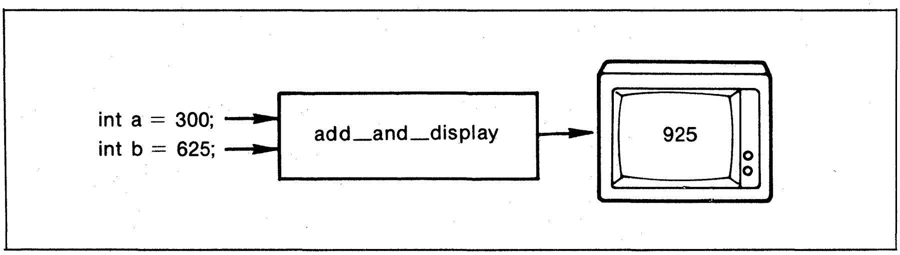


This illustration tells you that both of the variables are of type int and gives you possible values that can be assigned to each. In this case, the values 300 and 625 will be added.

Next, note how the returned value is presented:

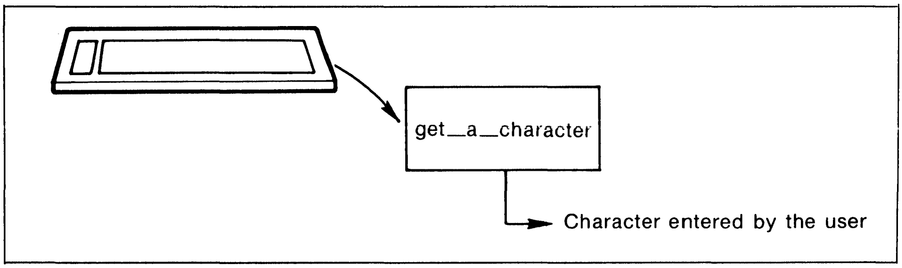


Each routine that returns a value will show the return value coming out of the bottom of the box. Consider the routine add_and_display. Rather than returning the result of the addition, this routine instead displays it to the screen, as shown here:

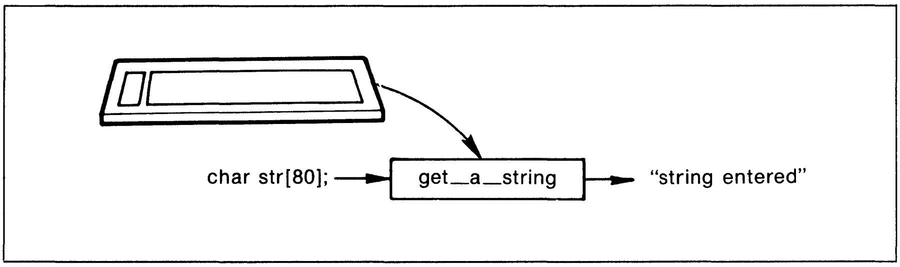


Any routine that writes data to the screen will be presented in this manner.

Similarly, if the routine get_a_character uses the keyboard, it is presented as follows:



If a routine modifies one or more of its parameters, the updated parameter is shown exiting the right-hand side of the box, as follows:



The goal of presenting each routine pictorially, is to build in your mind an image of the processing of the routine before you examine the source code. In many cases, you will find that you really do not need to know how a routine works, but rather what the routine does. These illustrations are meant to aid you in understanding the processing involved.

The source code for each routine is also presented in a consistent manner. Given the routine sum presented previously, the code will contain the following:

```
/*
 * sum (a, b)
 *
 * Return the sum of the two integer values specified.
 *
 * a (in): First value to sum.
```

```
 * b (in): Second value to sum.
 *
 * result = sum (6, 7);
 *
 */

sum (a, b)
  int a, b;
 {
  return (a + b);
 }
```

Note the descriptive header that precedes the function code:

```
/*
 * sum (a, b)
 *
 * Return the sum of the two integer values specified.
 *
 * a (in): First value to sum.
 * b (in): Second value to sum.
 *
 * result = sum (6, 7);
 *
 */
```

By examining this header information, you should be able to understand the routine's function, variables, and usage before you examine the source code that follows. Note that each parameter in the descriptive block is labeled as either (in) or (out). A parameter that does not change within a function is an (in) parameter. A function that does not use the original parameter value (but rather changes it) is an (out). If the function uses and then modifies the parameter, it is labeled (in/out).

Understanding Function Prototypes

For those who have never worked with Turbo C prototypes, you are in for a real treat. Turbo C allows you to define and declare a function. When you define a function, you provide its source code, as shown here:

```
float sum (a, b, c)
  float a, b, c;
 {
  return (a + b + c);
 }
```

When you declare a function, you tell another function information about the first function.

```
float sum ();
```

For years, many C programmers declared C functions only when the functions returned a value of a type other than int. By prototyping your functions, you can prevent many run-time errors simply because the errors are caught by the compiler.

Consider the following example:

```
float sum (a, b, c)
  float a, b, c;
 {
  return (a + b + c);
 }

main ()
 {
  float sum ();

  printf ("%f\n", sum (1.2, 2.4));
 }
```

In this case, the routine sum expects three parameters, but only two are present. Because the compiler has no knowledge about sum, the program code is acceptable. Hence, a possibly difficult-to-detect run-time error will occur.

By using prototypes you can prevent this error from occurring. Notice how you can change the function header for sum. You move the location of the parameter definitions within the parentheses, as shown here:

```
float sum (float a, float b, float c)
 {
  return (a + b + c);
 }
```

Within main, you must declare sum as a function and specify the type of each parameter, as shown here:

```
float sum (float a, float b, float c)
 {
  return (a + b + c);
 }

main ()
 {
  float sum (float, float, float);

  printf ("%f\n", sum (1.2, 2.4));
 }
```

Because main has knowledge about sum, it detects the invocation

```
printf ("%f\n", sum (1.2, 2.4));
```

as an error during compilation time.

During the development of this text, function prototyping saved me an immeasurable amount of time. You should always declare each routine you will use, along with the types of each of the routine's arguments. You will save considerable testing and debugging time in the future.

Assembly Language Routines

Chapters 14 and 15 present several functions based on two assembly language routines that provide a hardware interface. In order to use these routines, you must have either the Microsoft macro assembler or the object code library disk discussed in the Preface. A goal in any program development is to write as much of the software as possible in a high-level language. That goal has been met here.

Unfortunately, for the routines to execute fast enough to prevent snow on the screen display, the two interface routines must be written in assembly language.

To compile routines that contain in-line assembly language code, you must use the TCC command-line compiler (as opposed to the TC integrated environment).

A Final Word

Have fun and experiment. You have hundreds of routines with which to work. Your program development time should be drastically reduced. Make use of this time by studying the source code presented in this text.

CHAPTER

2

String Manipulation

The most widely used routines in any C programmer's library are those that perform string manipulation. Most C compilers provide a solid library of general-purpose string-manipulation functions and Turbo C is no exception. Because of the tremendous use of strings, however, you must fully understand how C stores and manipulates strings.

This chapter examines strings in detail. Many possible implementations could be used to solve the problems presented in this chapter. By the end of this chapter, you should be able to recognize the factors that make one solution "better" than another. This chapter will help you understand how some of Turbo C's standard library functions work while providing you with a complete set of string-manipulation routines. Your programs will exploit each of these routines on a regular basis.

Getting Started

A *character string* is a sequence of one or more characters. The C language stores character strings as arrays, in which each character in the string resides in contiguous memory locations. For example, consider the string declaration shown here:

```
char some_string [255];
```

In this case, C creates a character string variable with storage space for 255 characters. Like all C arrays, C strings are indexed beginning at offset 0. As such, the previous declaration creates an array of characters indexed as shown in Figure 2-1.

By default, C contains no built-in method to determine the number of characters contained in a C string. Instead, the standard

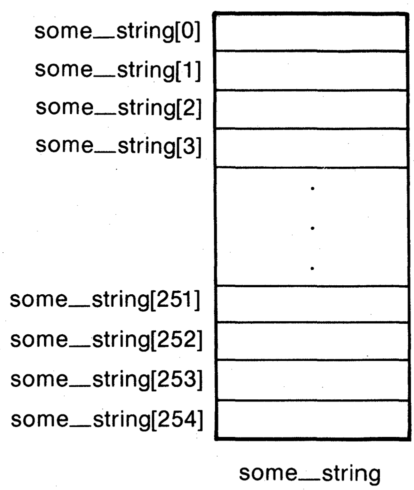


Figure 2-1. *Indexed array of characters*

is to place a null character (ASCII 0) immediately following the last character in the string. Thus, C stores the string "Turbo C" as characters in an array with the null character appended (see Figure 2-2).

Because each character string terminates with the null character, you can determine the number of characters in the string *s* simply by searching for the null character (\0), as shown here:

```
for (i = 0; s[i] != '\0'; i++)
  ;
```

Each time you specify a character string within double quotation marks, Turbo C places the null character at the end of the string for you. For example:

```
#define COMPILER "Turbo C"
```

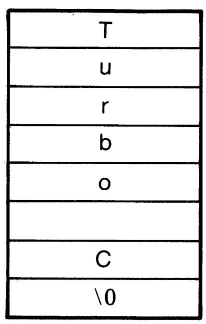


Figure 2-2. *Placement of null character in a string*

In most cases, however, ensuring that a character string is terminated by a null character becomes the responsibility of the programmer, as shown here:

```
main ()
 {
   char alphabet [27];    /* 26 letters and space for null */

   char letter;

   int index;

   for (index = 0, letter = 'A'; letter <= 'Z'; index++, letter++)
      alphabet [index] = letter;

   alphabet [index] = '\0';      /* append the null character */

   printf ("%s\n", alphabet);
 }
```

Strings as Parameters

One of the contributing factors that helps you write generic string-manipulation routines is the manner in which C treats arrays passed to functions. Assume that you have a function called string_length that returns the number of characters in a string. Invoke it from your program, as shown here:

```
count = string_length (strvar);
```

Since you are passing a character array, you can declare the string within the function with no array bounds:

```
string_length (char str[])
 {
  /* code here */
 }
```

All of the routines in the remainder of this chapter declare the formal string parameters in this manner.

String Length

The following routine returns the length of a string by examining succeeding characters for the null character:

```
/*
 * string_length (string)
 *
 * Return the number of characters in the string.
 *
 * string (in): string to return the length of.
 *
 * count = string_length (string);
 *
 */

int string_length (char string[])
 {
  int i;

  for (i = 0; string[i]; i++)
    ;

  return (i);
 }
```

Array Bounds

In most programming applications, time is always a tradeoff against other factors. In some cases, the tradeoff becomes time versus space. String-manipulation routines are no exception.

Consider this routine, which copies the contents of one string to another:

```
/*
 * void first_copy (source, target)
 *
 * Copy the contents of the source string to the target.
 *
 * s1 (in): source string containing characters to copy.
 * s2 (out): string receiving characters copied.
 *
 * first_copy ("This is a test", stringvar);
 *
 * first_copy does not perform bounds checking.
 *
 */
```

```
void first_copy (char s1[], char s2[])
 {
  int i;

  for (i = 0; s1[i] != '\0'; ++i)
    s2[i] = s1[i];

  s2[i] = '\0';
 }
```

This routine copies characters from the first string (*s1*) to the second (*s2*), one at a time, until the null character is found (see Figure 2-3).

This routine will work properly in most cases. However, consider this program, which uses first_copy:

```
main ()
 {
   s2[5];

   first_copy ("long string", s2);
 }
```

Here, first_copy appears to copy characters from *s1* to *s2*, as desired. Actually, however, the copy has exceeded the array bounds of *s2*. The character string "long string" contains 11 characters

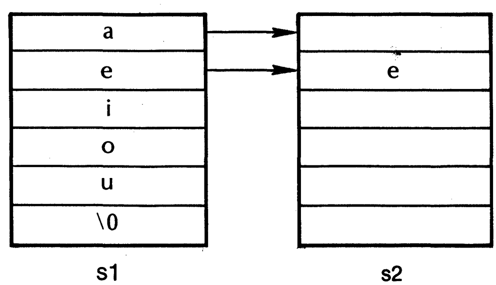


Figure 2-3. *Copying characters from first string to second string*

(including the null character), while *s2* only has space for 5. As a result, first_copy overwrites the contents of the routine's stack space and produces an error. To remedy this problem, you can include a parameter that defines the maximum number of characters to be assigned to the target string, as shown here:

```
second_copy ("long string", s, sizeof(s));
```

The following routine implements second_copy:

```
/*
 * int second_copy (source, target, array_bound)
 *
 * Copy the source string to the target string variable.
 *
 * s1 (in): Contains the characters to be copied.
 * s2 (out): Receives the characters copied.
 * maxchar (in): specifies the maximum number of characters
 *               that s2 can store.
 *
 * status = second_copy ("This is", stringvar, sizeof (stringvar));
 *
 * If the array bounds are exceeded, second_copy returns the value
 * 1; otherwise it returns the value 0.
 *
 */

int second_copy (char s1[], char s2[], int maxchar)
 {
  int index;

  maxchar--;             /* leave space for null */

  for (index = 0; (s1[index] != '\0') && index < maxchar; index++)
    s2[index] = s1[index];

  s2[index] = '\0';

  return (s1[index] && (index == maxchar));
 }
```

This routine indeed allows you to prevent the error that previously occurred. However, because you must now include the second test

```
(s1[index] != '\0') && (index < maxchar)
```

you increase the required processing time for each iteration of the loop.

Minimizing Source Code

Although the previous routines were quite readable (assuming that you are familiar with C arrays), you can simplify (reduce) the code required to implement them.

Consider this code fragment:

```
for (i = 0; s1[i] != '\0'; ++i)
  s2[i] = s1[i];

s2[i] = '\0';
```

C allows you to change this code, as shown here:

```
for (i = 0; (s2[i] = s1[i]) != '\0'; ++i)
  ;
```

In both cases, each fragment performs the identical function. In the second code fragment, Turbo C will test the value that is assigned to *s2* with each iteration of the loop. If that value is null, C terminates the loop. If not, C simply assigns the next character in *s1* to *s2*, thus repeating the test. Once the null character has been assigned to *s2*, the loop terminates. Since the code contained within the for loop has already assigned the null character to *s2*, you can eliminate the line

```
s2[i] = '\0';
```

When this code assigns the null character to *s2*, the value returned from the test

```
(s2[i] = s1[i]) != '\0'
```

is 0 (null is the ASCII 0). Since C equates the Boolean false to 0, you can again modify this code as follows:

```
for (i = 0; (s2[i] = s1[i]) ; ++i)
  ;
```

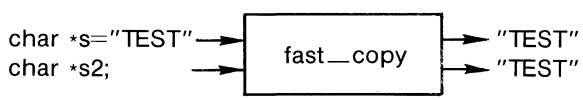
As you develop your C string-manipulation routines, keep the following in mind:

- Is execution speed more important than reliability?
- Is the code as simple as possible?
- Does the code maintain readability?

As you develop your library of string-manipulation routines, you should constantly attempt to balance the tradeoffs between speed, reliability, and readability. Actual implementation will likely be based upon your programming requirements. As such, the decision of which routine to use can often have as great an impact on your program as the code that actually implements the routine.

String Copy

Two functions implement a string copy routine. The first, fast_copy, copies the contents of the first string specified to the second without bounds checking. The second, copy_string, also performs the same processing, but with bounds checking enabled.

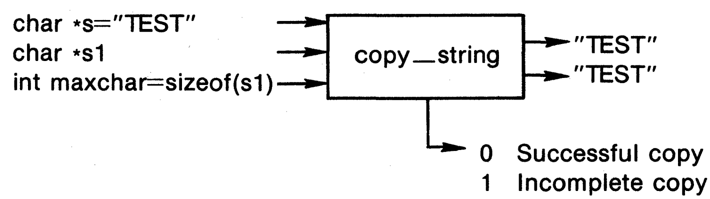


Warning: fast_copy does not perform bounds checking.

```
/*
 * void fast_copy (source, target)
 *
 * Copy the contents of the source string to the target.
 *
 * s1 (in): source string containing characters to copy.
```

```
 * s2 (out): string receiving characters copied.
 *
 * fast_copy ("This is a test", stringvar);
 *
 * fast_copy does not perform bounds checking.
 *
 */

void fast_copy (char s1[], char s2[])
 {
  int i;

  for (i = 0; (s2[i] = s1[i]) ; ++i)
     ;
 }
```

If the routine cannot successfully complete the copy, it returns the value 1. Otherwise, it returns the value 0.

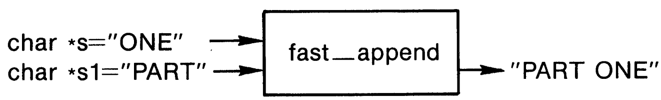



```
/*
 * int copy_string (source, target, array_bound)
 *
 * Copy the source string to the target string variable.
 *
 * s1 (in): contains the characters to be copied.
 * s2 (out): receives the characters copied.
 * maxchar (in): specifies the maximum number of characters
 *               that s2 can store.
 *
 * status = copy_string ("This is", stringvar, sizeof (stringvar));
 *
 * If the array bounds are exceeded, string_copy returns the value
 * 1; otherwise it returns the value 0.
 *
 */

int copy_string (char s1[], char s2[], int maxchar)
 {
  int i;

  maxchar--;             /* leave space for null */

  for (i = 0; (s2[i] = s1[i]) && i < maxchar; i++)
     ;

   if (i == maxchar && s1[i])   /* see if characters remain in s1 */
     {
      s2[i] = '\0';
```

```
    return (1);
    }
  else
    return (0);
}
```

String Append

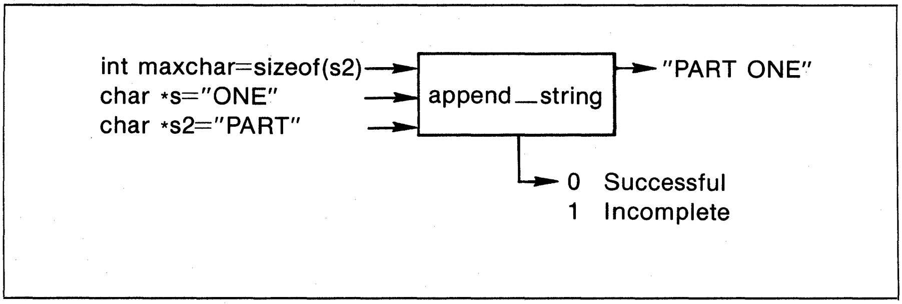
The following two routines append the contents of the first string specified to the second. The routine first locates the end of the second string (the null character) and then begins appending characters from the first string at that point. Once the null character from the first string is appended to the second string, the loop terminates. As before, the routine called fast_append does not perform bounds checking, but the routine called append_string does.



Warning: fast_append does not perform bounds checking.

```
/*
 * void fast_append (source, target)
 *
 * Append the contents of the source string to the target.
 *
 * s1 (in): source string containing characters to append.
 * s2 (out): string receiving characters copied.
 *
 * fast_append ("This is a test", stringvar);
 *
 * fast_append does not perform bounds checking.
 *
 */

void fast_append (char s1[], char s2[])
 {
  int i, j;

  for (i = 0; s2[i] ; ++i)    /* find the end of s2 */
     ;

  for (j = 0; s2[i] = s1[j]; i++, j++)  /* append s1 */
     ;

  }
```

If the following routine cannot successfully append the string, it returns the value 1. Otherwise, it returns the value 0.

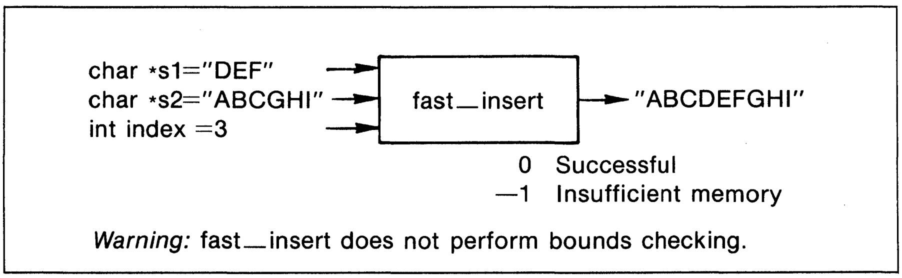



```
/*
 * insert_string (source, target, location, array_bounds)
 *
 * Insert the contents of the source string to the target
 * at the index location specified.
 *
 * s1 (in): source string containing characters to copy.
 * s2 (out): string receiving characters copied.
 * index (in): location within target to insert s1 at.
 * maxchar (in): maximum number of characters in s2.
 *
 * insert_string ("Pocket", stringvar, 6, sizeof (stringvar));
 *
 * insert_string returns one of the following:
 *   -1 insufficient memory    0 successful insertion    1 incomplete
 *
 */

int insert_string (char s1[], char s2[], int start_index, int maxchar)
{
  int i, j, len1, len2;

  char *temp;
  void *calloc(unsigned, unsigned);

  for (len1 = 0; s1[len1]; ++len1)     /* get length of s1 */
    ;

  for (len2 = 0; s2[len2]; ++len2)     /* get length of s2 */
    ;

  if (start_index > len2) start_index = len2;  /* append */

  if ((temp = (char *) calloc (1, len1+len2+1)) == '\0')
     return (-1);        /* unable to allocate memory */

  for (i = 0; i < start_index; ++i)
    temp[i] = s2[i];

  for (j = 0; temp[i+j] = s1[j]; j++)
    ;

  while (temp[i+j] = s2[i])
    ++i;
```

```
  for (i = 0; (s2[i] = temp[i]) && i < maxchar; i++)
    ;

  free (temp);

  if (i == maxchar && s2[i])     /* insertion is incomplete */
   {
    s2[i] = '\0';
    return (1);
   }
  else
    return (0);
}
```

String Insertion

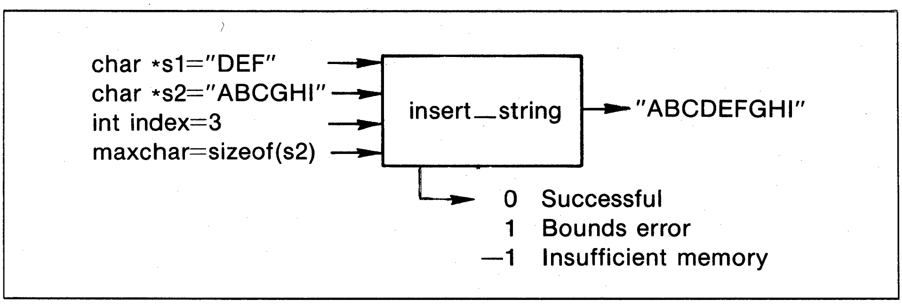
One of the keys to successful input-and-output (I/O) routines is the ability to insert one series of characters into another. The routine fast_insert does just that, but without bounds checking. If you are sure that inserting the new characters within the target string will not exceed the required storage space, this routine will indeed provide excellent performance. If you are not sure of this, you should use the routine insert_string instead.




```
/*
 * int fast_insert (source, target, index)
 *
 * Insert the contents of the source string to the target
 * at the index location specified.
 *
 * s1 (in): source string containing characters to copy.
 * s2 (out): string receiving characters copied.
 * index (in): location within target to insert s1 at.
 *
 * fast_insert ("Pocket", stringvar, 6);
 *
 * fast_insert does not perform bounds checking. If an error
 * occurs during processing, fast_insert returns -1. If the
 * insertion is successful, fast_insert returns 0.
 *
 */
```

```
int fast_insert (char s1[], char s2[], int start_index)
 {
  int i, j, len1, len2;

  char *temp;
  void *calloc(unsigned, unsigned);

  for (len1 = 0; s1[len1]; ++len1)    /* get the length of s1 */
    ;

  for (len2 = 0; s2[len2]; ++len2)    /* get the length of s2 */
    ;

  if (start_index > len2) start_index = len2;  /* append */

  if ((temp = (char *) calloc (1, len1+len2+1)) == '\0')
     return (-1);              /* unable to allocate memory */

  for (i = 0; i < start_index; ++i)
    temp[i] = s2[i];

  for (j = 0; temp[i+j] = s1[j]; j++)
    ;

  while (temp[i+j] = s2[i])
    ++i;

  for (i = 0; s2[i] = temp[i]; i++)
    ;

  free (temp);

  return (0);     /* successful insertion */
 }
```

If this routine cannot insert the string without overwriting the array bounds, it returns the value 1. If the routine cannot allocate sufficient memory, it returns −1. If the insertion is successful, the routine returns the value 0.



```
/*
 * insert_string (source, target, location, array_bounds)
 *
 * Insert the contents of the source string to the target
 * at the index location specified.
 *
 * s1 (in): source string containing characters to copy.
 * s2 (out): string receiving characters copied.
 * index (in): location within target to insert s1 at.
 * maxchar (in): maximum number of characters in s2.
 *
 * insert_string ("Pocket", stringvar, 6, sizeof (stringvar));
 *
 * insert_string returns one of the following:
 *   -1 insufficient memory   0 successful insertion   1 incomplete
 *
 */

int insert_string (char s1[], char s2[], int start_index, int maxchar)
 {
  int i, j, len1, len2;

  char *temp;
  void *calloc(unsigned, unsigned);

  for (len1 = 0; s1[len1]; ++len1)     /* get length of s1 */
    ;

  for (len2 = 0; s2[len2]; ++len2)     /* get length of s2 */
    ;

  if (start_index > len2) start_index = len2;  /* append */

  if ((temp = (char *) calloc (1, len1+len2+1)) == '\0')
     return (-1);       /* unable to allocate memory */

  for (i = 0; i < start_index; ++i)
    temp[i] = s2[i];

  for (j = 0; temp[i+j] = s1[j]; j++)
    ;

  while (temp[i+j] = s2[i])
    ++i;

  for (i = 0; (s2[i] = temp[i]) && i < maxchar; i++)
    ;

  free (temp);

  if (i == maxchar && s2[i])    /* insertion is incomplete */
   {
    s2[i] = '\0';
    return (1);
   }
  else
    return (0);
 }
```

Case Manipulation

Many programmers often choose to convert a string of characters to either uppercase or lowercase in order to simplify future processing. Although the methods for converting characters in this manner are many, you can use a simple fact about ASCII characters to speed up conversion routines: All ASCII characters use a byte (8 bits) of storage. The sixth-bit location determines the character's case. For example, if you examine the lowercase letter "a"

"a" ASCII 97 Binary 0110 0001

along with the character "A,"

"A" ASCII 65 Binary 0100 0001

you note that the only difference between them is the sixth bit:

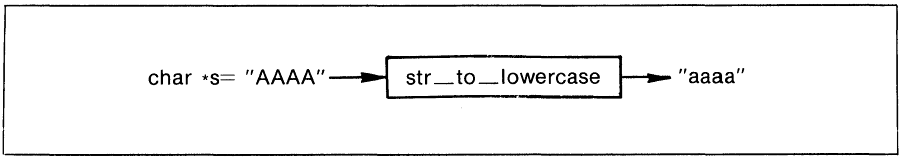
"a" Binary 0110 0001
"A" Binary 0100 0001

With this in mind, you can use C bitwise operators to perform quick comparisons:

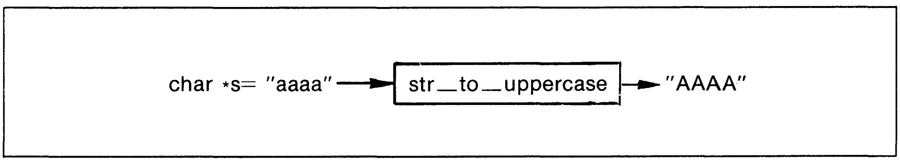
```
'A' | 32 = 'a'
0100 0001 | 0010 0000 = 0110 0001
```

```
'a' & ~32 = 'A'
0110 0001 & 1101 1111 = 0100 0001
```

The following routines convert a character string from uppercase to lowercase, and vice versa:




```
/*
 * void str_to_uppercase (s)
 *
 * Convert a string to UPPERCASE characters.
 *
 * s (in/out): string to convert to UPPERCASE.
 *
 * str_to_uppercase (filename);
 *
 * str_to_uppercase uses bit manipulation to convert characters
 * to uppercase.
 *
 */

void str_to_uppercase (char s[])
 {
  int i;

  for (i = 0; s[i]; i++)
    if (s[i] >= 'a' && s[i] <= 'z')
      s[i] = s[i] & ~32;
 }
```



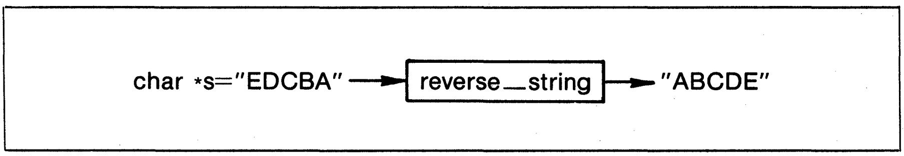

```
/*
 * void str_to_lowercase (s)
 *
 * Convert a string to lowercase characters.
 *
 * s (in/out): string to convert to lowercase.
 *
 * str_to_lowercase (filename);
 *
 * str_to_lowercase uses bit manipulation to convert characters
 * to lowercase.
 *
 */

void str_to_lowercase (char s[])
 {
  int i;
```

```
    for (i = 0; s[i]; i++)
      if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] = s[i] | 32;
  }
```

String Reversal

When a program manipulates a mathematical expression as a string, the result shows the string reversed. Just as you have many ways to convert a string from uppercase to lowercase, you also have many ways to reverse the contents of a character string.

Consider two alternatives. The first method, reverse_string, begins swapping characters in the string starting with the first and last characters, then the second and next to last, and so on. This effective method results in $1.5 * n$ exchanges, where n is the number of array elements.




```
/*
 * void reverse_string (s)
 *
 * Reverse the contents of the character string specified.
 *
 * s (in/out): string to reverse the contents of.
 *
 * reverse_string (binary_result);
 *
 * This method requires 1.5 * n exchanges.
 *
 */

void reverse_string (char s[])
 {
  char temp;

  int i, j;

  for (j = 0; s[j]; ++j)    /* find the end of string */
    ;
```

```
  for (i = 0, j--; i < j; i++, j--)
    {
      temp = s[i];
      s[i] = s[j];
      s[j] = temp;
    }
 }
```

The second method has additional overhead in the form of a call to allocate memory large enough to buffer the string contents. Once this space is allocated, the first string is simply copied to the buffer and then back to the string in reverse order.

```
/*
 * int reverse_string (s)
 *
 * Reverse the contents of the character string specified.
 *
 * s (in/out): string to reverse the contents of.
 *
 * result = reverse_string (binary_result);
 *
 * This method requires 2.0 * n exchanges. If an error in
 * processing occurs, reverse_string returns -1.
 *
 */

int reverse_string (char s[])
 {
  char *temp;
  void *calloc (unsigned, unsigned);

  int i, j;

  for (j = 0; s[j]; ++j)    /* find the end of string */
    ;

  if ((temp = (char *) calloc (1, j)) == '\0')
    return (-1);           /* couldn't allocate memory */

  for (i = 0, j--; j >= 0; i++, j--)
      temp[i] = s[j];

  for (j = 0; j < i; j++)
    s[j] = temp[j];

  return (0);
 }
```

This method results in $2 * n$ exchanges.

Given a string of 512 characters, the first method requires 768 exchanges, while the second method requires 1024. The first method is clearly superior. Your algorithm decision (even for simple routines) can have a significant impact on the performance of your program.

Exchanging Strings

Based on the preceding analysis of the string reverse routine, you can conclude that the fastest way to exchange two strings is by using a three-variable swap (see Figure 2-4).

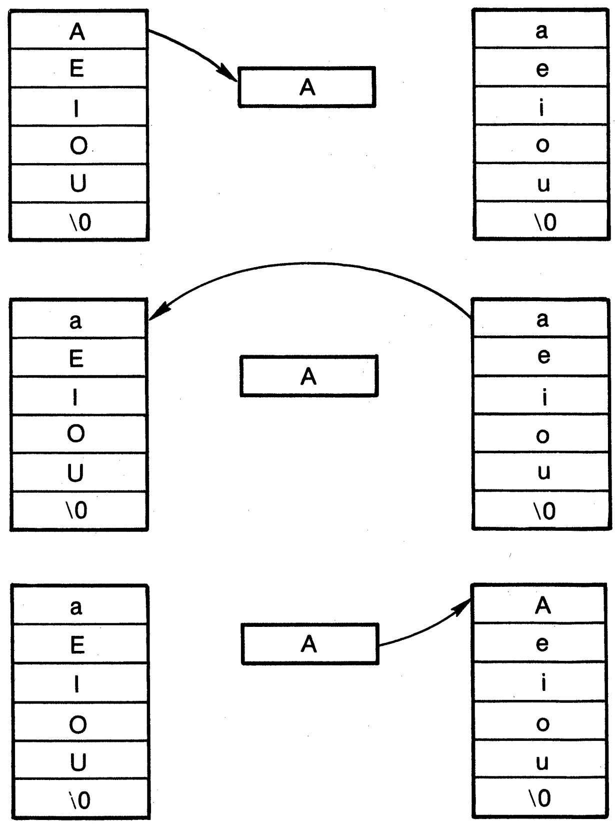
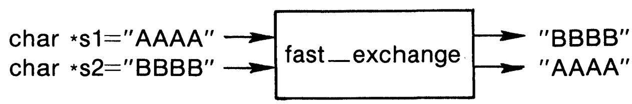


Figure 2-4. *Three-variable swap*

The following routines do just that. However, you must again consider the possibility that the user did not allocate the same amount of space for each string, as shown here:

```
char s1[32], s2[64];
```

To exchange two strings whose boundaries are not identical could have devastating results. Again, you have two alternative routines—fast_exchange, which does not perform bounds checking, and string_exchange, which does.

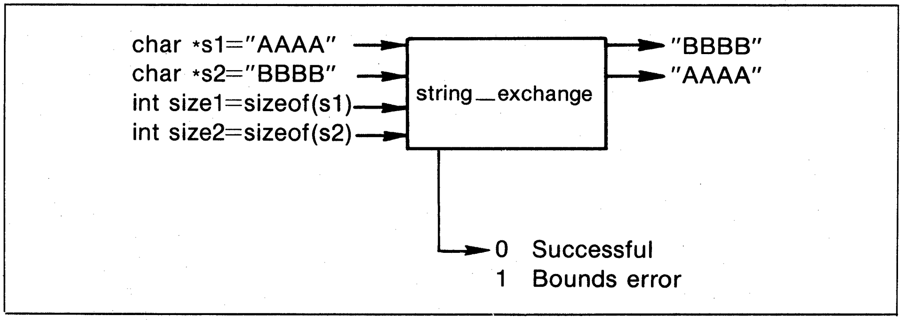


Warning: fast_exchange does not perform bounds checking.

```
/*
 * void fast_exchange (s1, s2)
 *
 * Exchange the characters contained in two character strings.
 *
 * s1 (in/out): contains characters to exchange.
 * s2 (in/out): contains characters to exchange.
 *
 * fast_exchange (oldname, newname);
 *
 */

void fast_exchange (char s1[], char s2[])
 {
  int i, j;
  char temp;

  for (i = 0; s1[i] && s2[i]; i++)
    {
      temp = s1[i];
      s1[i] = s2[i];
      s2[i] = temp;
    }

  if (s1[i])
    {
      j = i;
      while (s1[i])
        s2[i] = s1[i++];
      s2[i] = '\0';
```

```
      s1[j] = '\0';
    }
  else if (s2[i])
    {
      j = i;
      while (s2[i])
        s1[i] = s2[i++];
      s1[i] = '\0';
      s2[j] = '\0';
    }
  }
```

The following routine performs bounds checking, which prevents it from exceeding the array bounds:



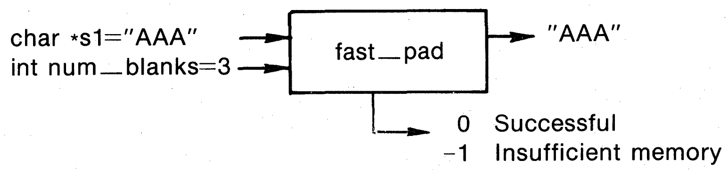

```
/*
 * string_exchange (s1, s2, size1, size2)
 *
 * Exchange the contents of two character strings.
 *
 * s1 (in/out): contains characters to exchange.
 * s2 (in/out): contains characters to exchange.
 * size1 (in): maximum number of characters in s1.
 * size2 (in): maximum number of characters in s2.
 *
 * result = string_exchange (name, a, sizeof (name), sizeof (a));
 *
 * string_exchange returns one of the following:
 * 0 successful exchange -1 Insufficient memory  1 Incomplete
 *
 */

int string_exchange (char s1[], char s2[], int size1, int size2)
 {
  int i, j;
  char temp;

  for (i = 0; s1[i]; i++)   /* get length of s1 */
    ;
```

```
if (i >= size2)          /* too large for s2 ? */
  return (1);

for (i = 0; s2[i]; i++)  /* get length of s2 */
    ;

if (i >= size1)          /* too large for s1 ? */
  return (1);

for (i = 0; s1[i] && s2[i]; i++)
  {
    temp = s1[i];
    s1[i] = s2[i];
    s2[i] = temp;
  }

if (s1[i])
  {
    j = i;
    while (s1[i])
      s2[i] = s1[i++];
    s2[i] = '\0';
    s1[j] = '\0';
  }
else if (s2[i])
  {
    j = i;
    while (s2[i])
      s1[i] = s2[i++];
    s1[i] = '\0';
    s2[j] = '\0';
  }
 return (0);
}
```

If the exchange is successful, the routine returns the value 0. If one string contained too many characters, the routine returns the value 1.

String Padding

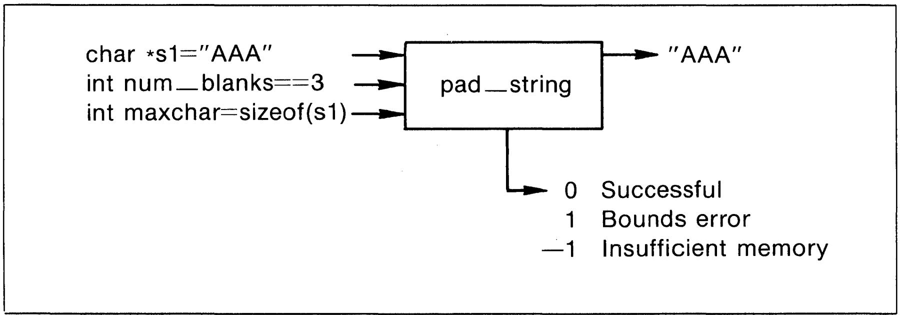
Many reports that appear aligned on the computer screen often require one or two leading blanks so that they will be aligned properly on printed output. The following routine enables you to place additional blanks in front of a string. Once again, bounds checking is a concern when producing the routines fast_pad and pad_string, as shown here:



Warning: fast_pad does not perform bounds checking.

```
/*
 * int fast_pad (s, num_blanks)
 *
 * Place the number of blanks specified at the front of a
 * string.
 *
 * s (in/out): string to pad
 * num_blanks (in): number of blanks to insert.
 *
 * result = fast_pad (s, 33);
 *
 * pad_string returns the value -1 if insufficient memory
 * prevented the insertion.
 *
 */

int fast_pad (char s[], int num_blanks)
 {
  int i, j;

  char *temp;

  void *calloc(unsigned, unsigned);

  for (i = 0; s[i]; i++)    /* get the length of s */
    ;

  if ((temp = (char *) calloc (1, i + num_blanks + 1)) == '\0')
    return (-1);           /* couldn't get memory */

  for (i = 0; i < num_blanks; i++)
    temp [i] = ' ';

  for (j = 0; temp [i] = s[j]; ++j, ++i)
    ;

  temp [i] = '\0';

  for (i = 0; s[i] = temp[i]; i++)
    ;

  free (temp);

  return (0);
 }
```

If the padding is successful, the following routine returns the value 0. If insufficient memory is available, the routine returns a value −1. If the array bounds are exceeded, the routine returns 1.



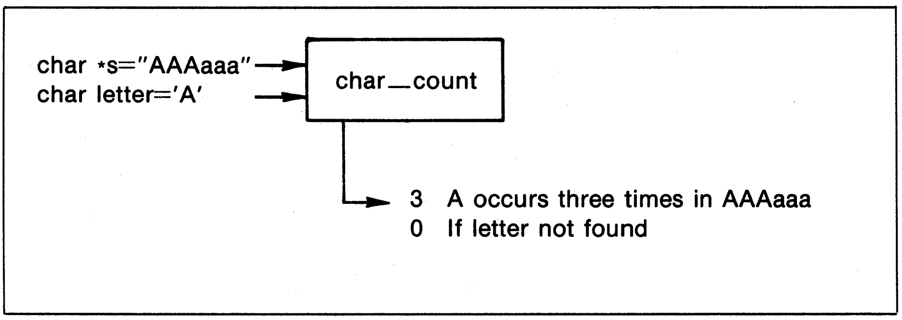

```
/*
 * int pad_string (s, num_blanks, maxchar)
 *
 * Place the number of blanks specified at the front of a
 * string.
 *
 * s (in/out): string to pad
 * num_blanks (in): number of blanks to insert.
 * maxchar (in): maximum number of characters in s.
 *
 * result = pad_string (s, 33, sizeof (s));
 *
 * pad_string returns one of the following values:
 * -1 Insufficient Memory 0 Successful 1 Incomplete
 *
 */

int pad_string (char s[], int num_blanks, int maxchar)
 {
  int i, j;

  char *temp;

  void *calloc(unsigned, unsigned);

  for (i = 0; s[i]; i++)          /* get length of s */
    ;

  if (i + num_blanks >= maxchar)  /* will blanks fit */
    return (1);
  else if ((temp = (char *) calloc (1, i + num_blanks + 1)) == '\0')
    return (-1);                  /* can't get memory */

  for (i = 0; i < num_blanks; i++)
    temp [i] = ' ';

  for (j = 0; temp [i] = s[j]; ++j, ++i)
    ;
```

```
  temp [i] = '\0';

  for (i = 0; s[i] = temp[i]; i++)
    ;

  free (temp);

  return (0);
}
```

Character Manipulation

Before discussing the more difficult routines that perform string comparisons and substring matching, you should first consider routines that manipulate a single character within a string. The following routines locate, replace, or delete a specific character within a string.

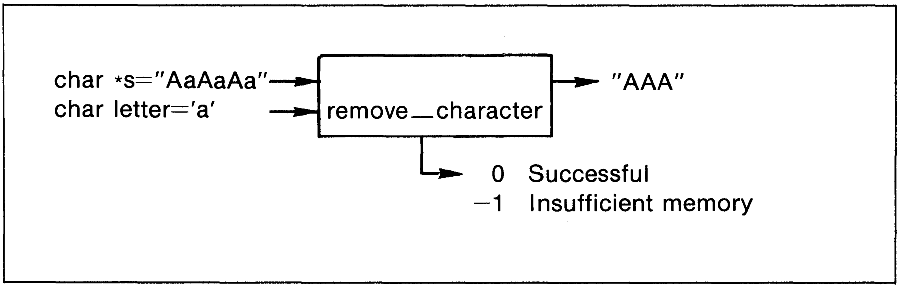
The first function, char_count, returns the number of occurrences of a character within a string:




```
/*
 * char_count (s, letter)
 *
 * Return the number of occurrences of letter in s.
 *
 * s (in): string to search.
 * letter (in): letter to search for.
 *
 * count = char_cnt ("This is a test", 's');
 *
 */

char_count (char s[], char letter)
 {
 int i, count = 0;
```

```
  for (i = 0; s[i]; i++)
    if (s[i] == letter)
      count++;

  return (count);
 }
```

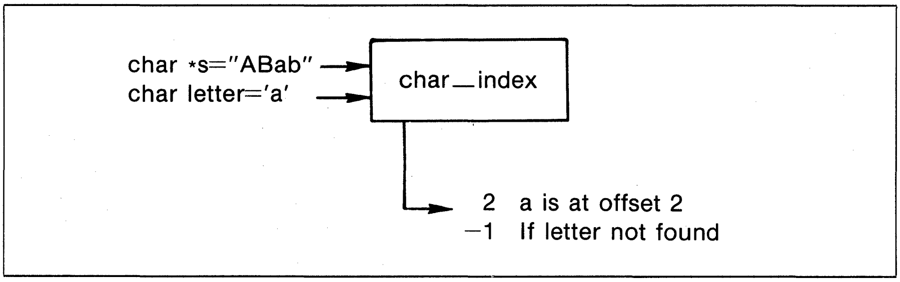
Similarly, the routine remove_character removes each occurrence of the specified character:




```
/*
 * remove_character (s, letter)
 *
 * Remove each occurrence of letter from s.
 *
 * s (in/out): string to remove the letter from.
 * letter (in): letter to remove.
 *
 * remove_character (s, 'a');
 *
 * remove_character returns the value -1 if insuffient memory
 * prevented the removal, otherwise, 0.
 *
 */

int remove_character (char s[], char letter)
 {
  int i, j;

  char *temp;

  void *calloc(unsigned, unsigned);

  for (i = 0; s[i]; i++)
    ;

  if ((temp = (char *) calloc (1, i)) == '\0')
    return (-1);

  for (i = 0, j = 0; s[i]; i++)
    if (s[i] != letter)
      temp[j++] = s[i];
```

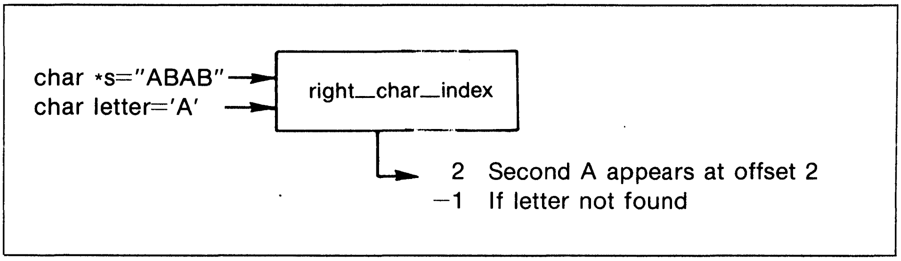
```
  for (temp[j] = '\0', i = 0; s[i] = temp[i]; i++)
    ;

  free (temp);

  return(0);
 }
```

The function char_index returns the first occurrence of a character string within a string. If the character is not found, the routine returns the value −1.



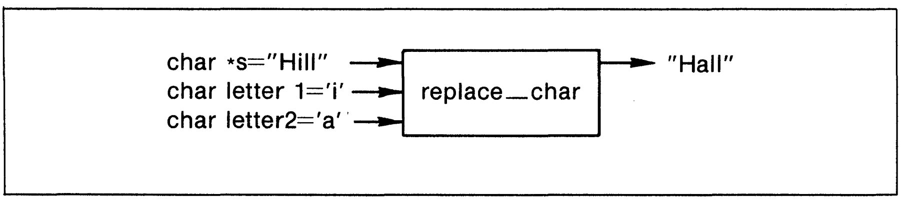

```
/*
 * char_index (s, letter)
 *
 * Return the location of the first occurrence of letter in s.
 *
 * s (in): string to search for letter.
 * letter (in): letter to search for.
 *
 * index = char_index (s, letter);
 *
 * char_index returns -1 if the letter is not found.
 *
 */

int char_index (char s[], char letter)
 {
  int i, location = -1;;

  for (i = 0; s[i] && (location == -1); i++)
    if (s[i] == letter)
      location = i;

  return (location);   /* -1 if not found */
 }
```

Similarly, the routine right_char_index returns the last occurrence of the character specified in a string or the value −1 if the character is not found.




```
/*
 * right_char_index (s, letter)
 *
 * Returns the rightmost occurrence of letter in s.
 *
 * s (in): string to search for the letter.
 * letter (in): letter to search for.
 *
 * result = right_char_index (s, 'A');
 *
 * right_char_index returns -1 if the letter is not found.
 *
 */

int right_char_index (char s[], char letter)
 {
  int i, location = -1;

  for (i = 0; s[i]; i++)
    if (s[i] == letter)
      location = i;

  return (location);    /* -1 if not found */
 }
```

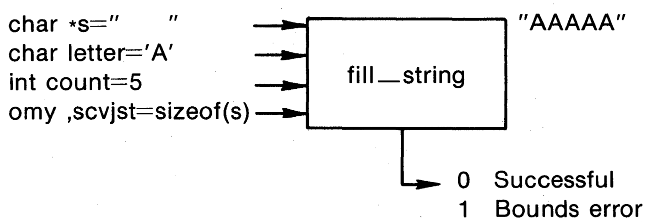
The routine replace_char replaces each occurrence of the first specified character with the second specified character.




```
/*
 * void char_replace (s, source_letter, target_letter)
 *
 * Replace each occurrence of source_letter with target_letter
 * within the string s.
 *
```

```
 * s (in/out): string to replace characters in.
 * source_letter (in): letter to replace.
 * target_letter (in): replacement letter.
 *
 * char_replace (s, 'A', 'a');
 *
 */

void char_replace (s, source_letter, target_letter)
  char s[];
  int source_letter, target_letter;
 {
  int i;

  if (source_letter != target_letter)
   for (i = 0; s[i]; i++)
    if (s[i] == source_letter)
      s[i] = target_letter;
 }
```

The routine fill_string fills a character string with a specific number of occurrences of the specified character. The routine performs bounds checking to ensure that it does not overwrite the array bounds. If the assignment is successful, the routine returns the value 0. Otherwise, the routine returns the value 1.

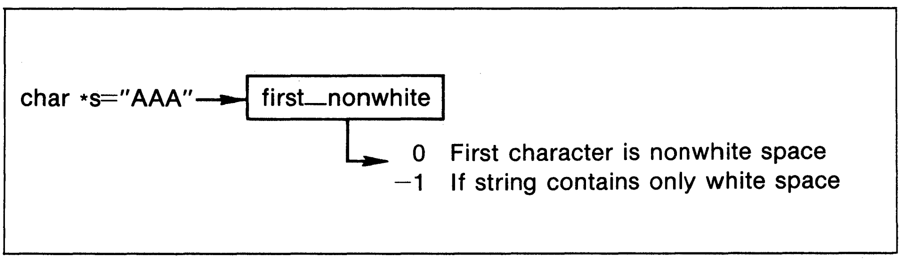



```
/*
 * fill_string (s, letter, count, maxchar)
 *
 * Place count occurrences of letter into the string s.
 *
 * s (in/out): string to fill.
 * letter (in): letter to place into the string.
 * count (in): number of times to insert the letter.
 * maxchar (in): maximum number of characters in s.
 *
 * fill_string (s, 'A', 10, sizeof (s));
 *
```

```
 * fill_string returns 1 if the fill was unsuccessful, 0 if
 * successful.
 *
 */

int fill_string (char s[], int letter, int count, int maxchar)
 {
  int i;

  if (count+1 > maxchar)       /* +1 reserves space for null */
    return (1);                /* insufficient memory */

  for (i = 0; i < count; ++i) /* fill the string */
    s[i] = letter;

  s[i] = '\0';

  return (0);
 }
```

White Space

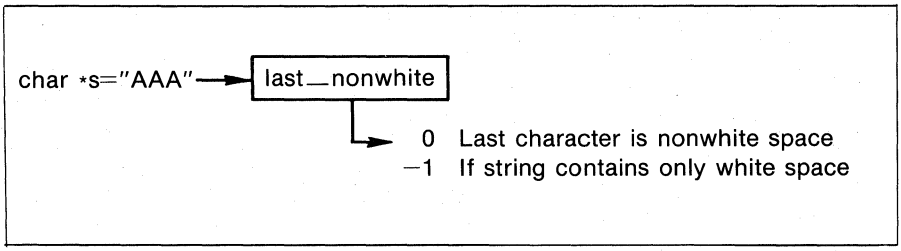
Programmers who use C often define white space as either the blank character (ASCII 32) or the tab character (ASCII 9). Just as your programs require you to place blank characters at the start of character strings, they periodically require that you remove them. The following routines allow you to locate the first and last nonwhite-space character in a string. The first function, called first_nonwhite, returns the location of the first character in the string that is not a white-space character, or the value −1 if the string contains solely white space.




```
/* first_nonwhite (s)
 *
 * Return the index of the first character that is not white
```

```
 * space (a blank or a tab).
 *
 * s (in): string to examine for nonwhite space.
 *
 * result = first_nonwhite (name);
 *
 * If the string contains all white space, -1 is returned.
 *
 */

int first_nonwhite (char s[])
 {
  int i, location = -1;

  for (i = 0; s[i] && (location == -1); ++i)
    if ((s[i] != ' ') && (s[i] != '\t'))
      location = i;

  return (location);   /* -1 if all white space */
 }
```

Similarly, the routine last_nonwhite returns the location of the last nonwhite-space character in the string, or the value −1 if the string contains only white space.




```
/* last_nonwhite (s)
 *
 * Return the index of the last character that is not white
 * space (a blank or a tab).
 *
 * s (in): string to examine for nonwhite space.
 *
 * result = last_nonwhite (name);
 *
 * If the string contains all white space, -1 is returned.
 *
 */

int last_nonwhite (char s[])
 {
  int i, location = -1;

  for (i = 0; s[i]; ++i)
    if ((s[i] != ' ') && (s[i] != '\t'))
      location = i;
```

```
 return (location);  /* -1 if all white space */
}
```

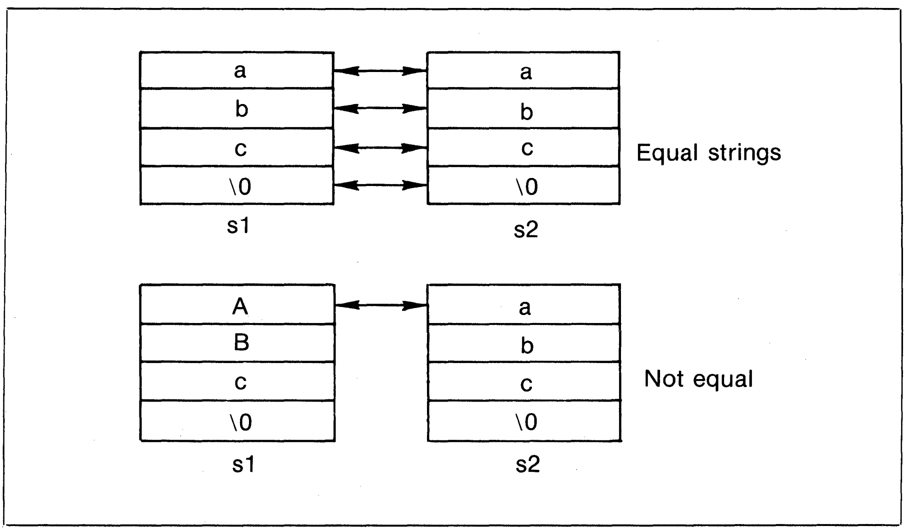
String Comparison

Most applications that perform string manipulation eventually must perform string comparisons. Many existing routines will help you determine whether two strings are equal. For example, consider this loop, which compares two strings:

```
for (i = 0; s1[i] == s2[i]; ++i)
  if (s1[i] == '\0')
    {
     printf ("Equal strings\n");
     break;
    }
```

The routine begins with the first letter in each string and compares them. As long as the letters are equal, the routine compares succeeding characters. This process continues until either two characters are not equal, or the ends of both strings are found.

Consider these examples:

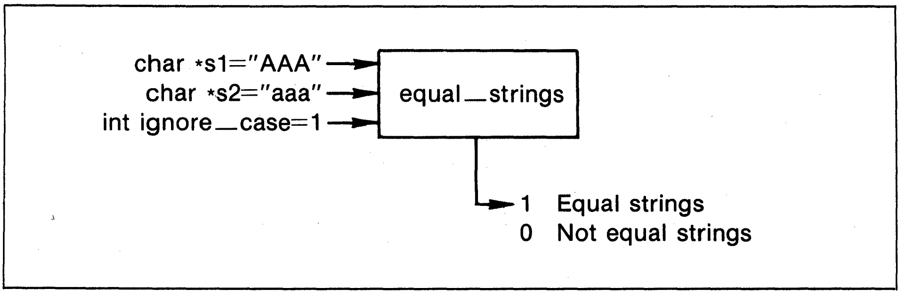


Although this routine works, often the program would like the strings to be considered equal, regardless of the case of the letters. Here, the routine should show the strings "Turbo C" and "TURBO C" as equivalent. In order to support the capability to perform case-sensitive comparisons, you must add a third parameter, as shown here:

```
equal_strings (s1, s2, ignore_case);
```

Thus, one routine serves both possible requirements of the user.

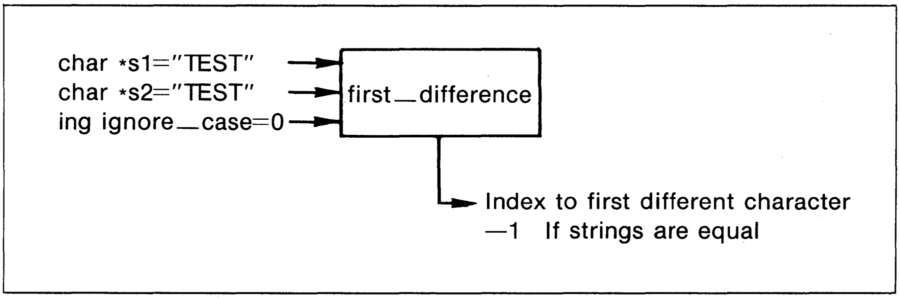
The following routine returns the value 1 if the two strings are equal, and 0 otherwise. It supports case-sensitive processing.




```
/*
 * equal_strings (s1, s2, ignore_case)
 *
 * Return 1 if the strings s1 and s2 are equal, otherwise return
 * 0. Support case sensitive processing.
 *
 * s1 (in): string to compare.
 * s2 (in): string to compare.
 * ignore_case (in): if not 0, case of letters is ignored.
 *
 * if (equal_strings ("THIS", "this", 1))
 *
 * equal_strings returns 1 if the strings are equal, 0
 * otherwise.
 *
 */

int equal_strings (char s1[], char s2[], int ignore_case)
 {
  int i;
  char a, b;

  for (i = 0; s1[i] && s2[i]; i++)
    if (s1[i] != s2[i])
     {
```

```
        if (ignore_case)
         {
          a = (s1[i] >= 'a' && s1[i] <= 'z') ? s1[i] & ~32: s1[i];
          b = (s2[i] >= 'a' && s2[i] <= 'z') ? s2[i] & ~32: s2[i];
          if (a != b)
            break;
         }
        else
          break;
      }

   if (s1[i] || s2[i])
     return (0);
   else
     return (1);
 }
```

Similarly, the following routine returns the location of the first character that differs between two strings, or −1 if no difference occurs. This routine supports case-sensitive processing.

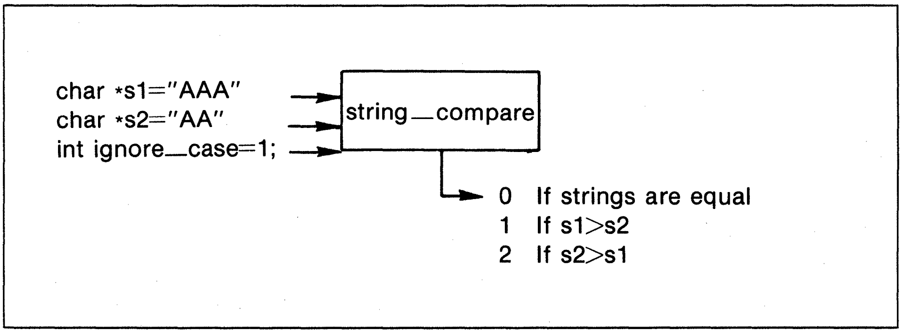



```
/*
 * first_difference (s1, s2, ignore_case)
 *
 * Return the location of the first difference between two strings
 * or the value -1 if the strings are equal.
 *
 * s1 (in): string to compare.
 * s2 (in): string to compare.
 * ignore_case (in): if 1, case is ignored.
 *
 * location = first_difference ("This", "THIS", 1);
 *
 */

int first_difference (char s1[], char s2[], int ignore_case)
 {
  int i;
  char a, b;

  for (i = 0; s1[i] && s2[i]; i++)
    if (s1[i] != s2[i])
     {
```

```
        if (ignore_case)
         {
          a = (s1[i] >= 'a' && s1[i] <= 'z') ? s1[i] & ~32: s1[i];
          b = (s2[i] >= 'a' && s2[i] <= 'z') ? s2[i] & ~32: s2[i];
          if (a != b)
            break;
         }
        else
          break;
      }

  if (s1[i] || s2[i])
    return (i);
  else
    return (-1);
}
```

The routine string_comp examines two character strings and returns one of the following values:

0	Strings are equal
1	First string is greater
2	Second string is greater




```
/*
 * string_comp (s1, s2, ignore_case)
 *
 * Compare the strings specified. Return 1 if s1 > s2, 2 if
 * s2 > s1 and 0 if the strings are equal. Support case sensitive
 * processing.
 *
 * s1 (in): string to compare.
 * s2 (in): string to compare.
 * ignore_case (in): if not 0, case of letters is ignored.
 *
 * if (string_comp ("THIS", "this", 1) == 1)
 *
 */
```

```
int string_comp (char s1[], char s2[], int ignore_case)
 {
  int i;
  char a, b;
  int result = 0;      /* 0 equal, 1 s1 greater, 2 s2 greater */

  for (i = 0; s1[i] && s2[i]; i++)
    if (s1[i] != s2[i])
     {
       if (ignore_case)
        {
         a = (s1[i] >= 'a' && s1[i] <= 'z') ? s1[i] & ~32: s1[i];
         b = (s2[i] >= 'a' && s2[i] <= 'z') ? s2[i] & ~32: s2[i];
         if (a != b)
          {
           if (a > b)
             result = 1;
           else
             result = 2;

           break;
          }
        }
       else
        {
         if (s1[i] > s2[i])
           result = 1;
         else
           result = 2;

         break;
        }
     }

  if (result == 0)
   {
    if (s1[i] == s2[i])
      result = 0;
    else if (s1[i])
      result = 1;
    else
      result = 2;
   }
  return (result);
 }
```

Substring Manipulation

Just as a character string is a sequence of characters, a *substring* is a series of characters within a string. For example, given the string "Turbo C Programmer's Library", "Turbo" is a five-character substring that begins at offset 0.

The following routines enable you to locate, replace, and count the number of occurrences of a substring within a string. The first routine, str_index, returns the starting offset of a substring within a string, or the value −1 if the substring is not found.

Given the string "McGraw-Hill" and the substring "Hill", the routine processes as follows:

1. Increment the index until string[index]==substring[0], or the end of the string is found (see Figure 2-5).
2. Increment the index of the string and substring as long as the corresponding letters are equal, or until the end of either string is found (see Figure 2-6).
3. If the substring is found, return the index within the string that corresponds to the start of the substring. Otherwise, resume Step 1.

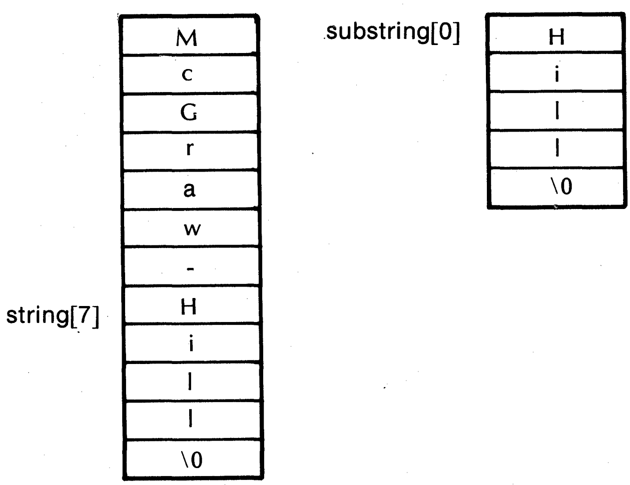


Figure 2-5. *Incrementing the index*

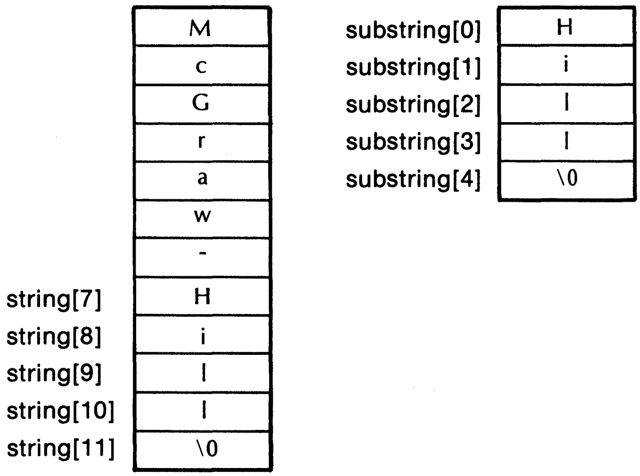
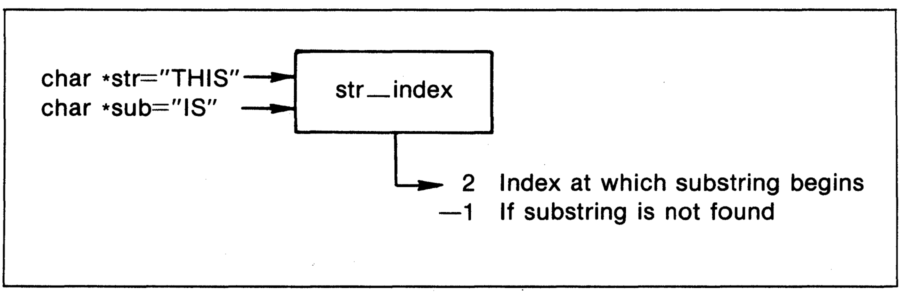


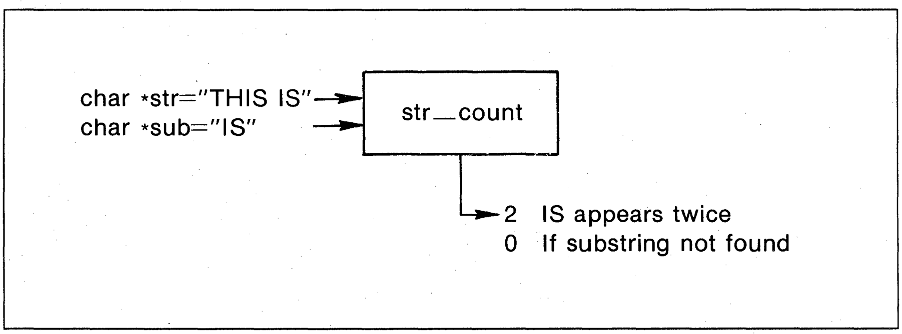
Figure 2-6. *Incrementing the index until length of string and substring are equal, or until end of string*

This code implements str_index:



```
/*
 * index (substring, string)
 *
 * Return the starting index of the substring within a string
 * or the value -1 if the substring is not found.
 *
 * substring (in): substring to search for.
 * string (in): string to examine.
 *
 * if (index ("PATH=", *ENV[1]) != -1)
 *
 */

int index (char substr[], char str[])
 {
  int i, j, k;

  for (i = 0; str[i]; i++)
    for (j = i, k = 0; str[j] == substr[k]; j++, k++)
       if (! substr[k+1])          /* end of substring */
         return (i);

  return (-1);                     /* substring not found */
 }
```

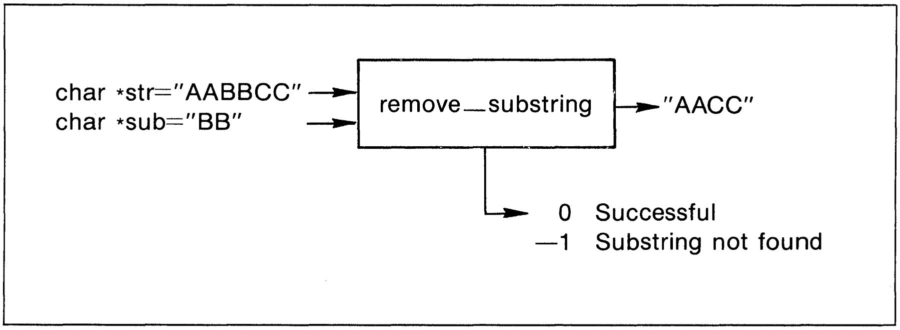
Similarly, the following routine returns a count of the number of times that a substring appears in a string:




```
/*
 * str_count (substring, string)
 *
 * Return the number of occurrences of the substring within
 * the string specified.
 *
 * substring (in): substring to search for.
 * string (in): string to examine.
 *
 * count = str_count ("is", "This is a test");
 *
 */
```

```
int str_count (char substr[], char str[])
 {
  int i, j, k, count = 0;

  for (i = 0; str[i]; i++)
    for (j = i, k = 0; str[j] == substr[k]; j++, k++)
       if (! substr[k+1])        /* end of substring */
         count++;

  return (count);                /* 0 if string not found */
 }
```

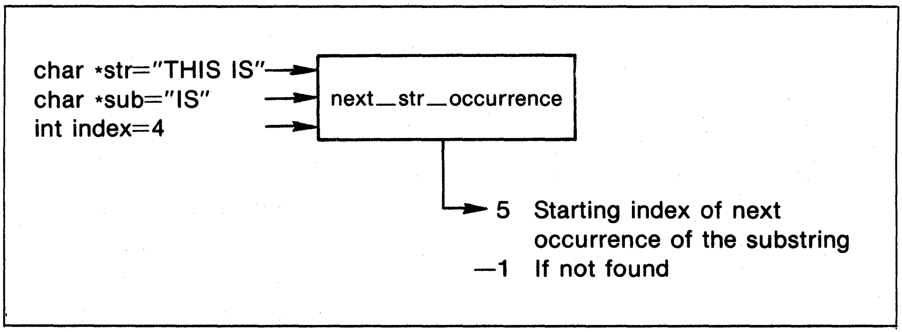
The routine remove_substring deletes each occurrence of a substring from a string:




```
/*
 * remove_substring (substring, string)
 *
 * Removes the first occurrence of a substring from within a string.
 *
 * substring (in): substring to remove.
 * string (in/out): string to remove the substring from.
 *
 * status = remove_substring ("is", strvar);
 *
 * If successful, remove_substring returns the value 0. If the
 * substring is not found, remove_substring returns the value -1.
 *
 */

int remove_substring (char substr[], char str[])
 {
  int i, j, k, location = -1;

  for (i = 0; str[i] && (location == -1); i++)
    for (j = i, k = 0; str[j] == substr[k]; j++, k++)
       if (! substr[k+1])        /* end of substring */
         location = i;

  if (location != -1)
   {
```

```
      for (k = 0; substr[k]; k++)
        ;

      for (j = location, i = location + k; str[i]; j++, i++)
        str[j] = str[i];

      str[j] = '\0';

      return (0);
    }
  else
      return (-1);                   /* substring not found */
}
```

The routine next_str_occurrence returns the next occurrence of a substring within a string that follows the index given. If the substring is not found, the routine returns the value −1.

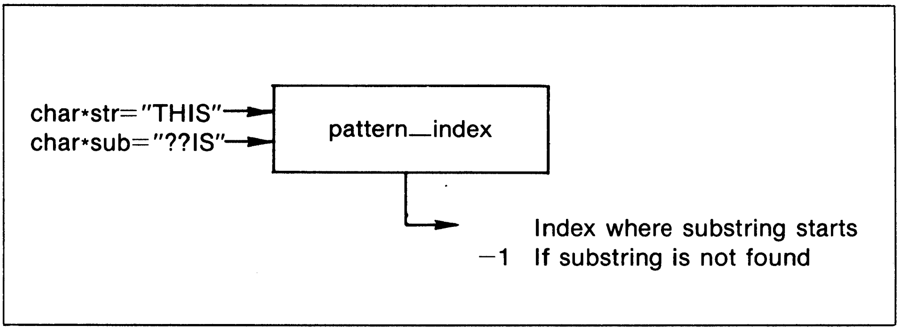



```
/*
 * next_str_occurrence (substring, string, start_index)
 *
 * Return the index of the next occurrence of the substring
 * within the string starting at the index specified.
 *
 * substring (in): substring to search for.
 * string (in): string to examine.
 * index (in): starting index of the search.
 *
 * location = next_str_occurrence ("is", "this is a test", 4);
 *
 * If the substring is not found, -1 is returned.
 *
 */

int next_str_occurrence (char substr[], char str[], int index)
 {
  int i, j, k;

  for (i = index; str[i]; i++)
    for (j = i, k = 0; str[j] == substr[k]; j++, k++)
       if (! substr[k+1])        /* end of substring */
         return (i);
```

```
  return (-1);                    /* substring not found */
 }
```

Pattern Matching

The last obstacle facing the completion of your library of string-manipulation routines is the matching of wildcard characters. For example, assume that you are concerned with only the first and last three letters of a file named ABC??FGH. Since you do not care about the middle two characters, they are replaced with question marks. With the routines just shown, you can modify the search loops, as shown here:




```
/*
 * pattern_index (substring, string)
 *
 * Return the starting index of the substring within the string.
 * Allow the user to place the ? wildcard character within the
 * substring for "don't care" letters.
 *
 * pattern_index ("this ???", "this bbb a test");
 *
 * If the substring is not found, pattern_index returns -1.
 *
 */

int pattern_index (substr, str)
  char substr[], str[];
 {
  int i, j, k;
```

```
  for (i = 0; str[i]; i++)
    for (j = i, k = 0; (str[j] == substr[k])
           || (substr[k] == '?'); j++, k++)

      if (! substr[k+1])            /* end of substring */
        return (i);

  return (-1);                      /* substring not found */
}
```

This simple addition adds considerable flexibility to your routines.

C H A P T E R

3

Pointer Manipulation

Chapter 2 created a valuable library of string-manipulation routines. To do so, each string was treated as an array of characters, as shown here:

```
char s[100];
```

Although each of the character array routines is fully functional, you can in many cases reduce to an even greater extent the amount of code required to implement each routine by using pointers. Many C programs make extensive use of pointers, so you must understand them.

By the end of this chapter, you should feel comfortable with the use and manipulation of pointers. In fact, you will be developing pointer-manipulation routines that possess tremendous capabilities.

Getting Started

All of the variables that you use in your programs are stored in memory. In order to be able to access each specific variable, C must have a means of differentiating variables (see Figure 3-1). C does this by assigning each variable a unique memory address, as shown in Figure 3-2.

The C ampersand operator (&) returns a variable's address. Note that the following program does not display the value each variable contains, but rather the location of each variable in memory:

```
main ()
 {
  int a = 5, b = 10;

  printf ("Address of a %u -- Address of b %u\n",
     &a, &b);

 }
```

By adding a simple modification to the program, you can display the address and value of each variable, as shown here:

```
main ()
 {
  int a = 5, b = 10;

  printf ("Address of a %u Value %d\n", &a, a);
  printf ("Address of b %u Value %d\n", &b, b);
 }
```

A *pointer* is a variable that contains a memory address. To declare a pointer variable, use the following format:

```
variable_type *variable_name;
```

For example, to declare a pointer to a value of type int, use the following:

```
int *int_pointer;
```

The asterisk tells C that the variable is a pointer to a memory location that contains a value of type int. To assign an address to the pointer, use the ampersand operator, as shown here:

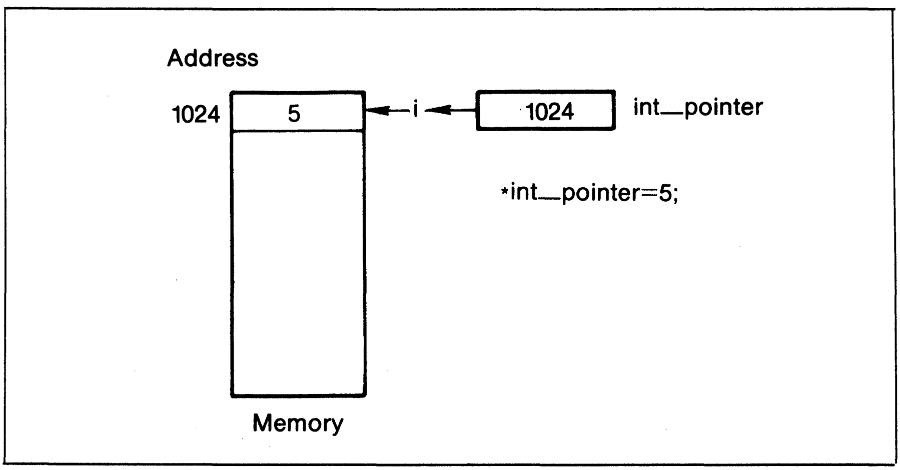
```
int_pointer = &integer_variable;
```

Notice the absence of the asterisk (*). When a program references a pointer without an asterisk, it is referring to a memory address. For example, in the previous assignment you were assigning the address of the variable *integer_variable* to the pointer *int_pointer*, so no asterisk was used. When an asterisk is used with a pointer, the value contained at the memory location referenced by the pointer is manipulated (as opposed to the address).

Consider the following example:

```
main ()
 {
  int i, *int_pointer;

  int_pointer = &i;

  *int_pointer = 5;
 }
```

This program begins by assigning the address of *i* to the pointer *int_pointer*. Following this assignment, *int_pointer* contains the memory address of *i*.



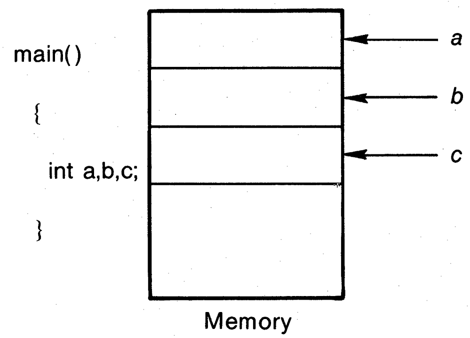


Figure 3-1. *Differentiating among variables*

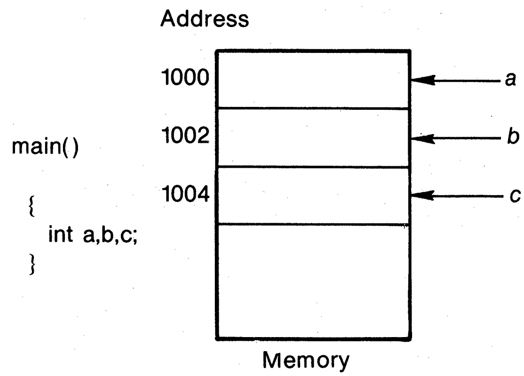
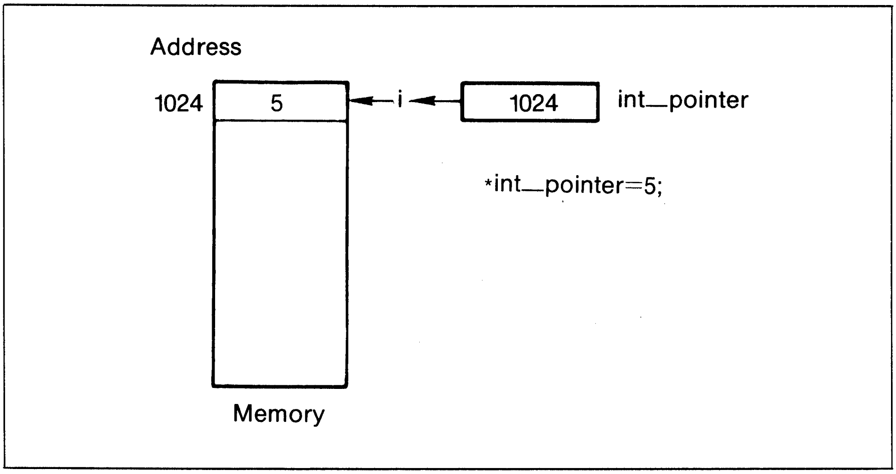


Figure 3-2. *Assigning variables unique memory addresses*

You can verify this by adding the following line of code:

```
printf ("Address of i %u Value of int_pointer\n",
        &i, int_pointer);
```

Next, the program assigns the value 5 to the memory location referenced by *int_pointer*.



In so doing, the variable *i* (which also refers to the same memory location) is assigned the value 5.

Pointers and Functions

Chapter 2 examined several functions that returned a status value indicating the success of their processing. In cases such as this where the function needs to return only one value, you have no additional processing concerns. For example, the following function returns the sum of two integer values:

```
sum (int a, int b)
 {
  return (a + b);
 }
```

In this case, the function returns one value and cannot modify the contents of its parameters.

C passes parameters to functions by using a technique known as *call by value.* Each time a parameter is passed to a function, C assigns a copy of the value in the parameter to the function parameters (*formal parameters*). Consider the following program, which passes two integer variables to a function. The function first displays the original values and then modifies each of the parameter values. However, when the program control returns to main, the original variable values remain unchanged because the function modified copies of the values contained in the variables (as opposed to the variables themselves).

```
main ()
 {
  int a = 5, b = 10;

  some_function (a, b);

  printf ("In main a = %d b = %d\n", a , b);
 }

some_function (int a, int b)
 {
  printf ("In some_function a = %d b = %d\n", a, b);

  a = 9;
  b = 11;

  printf ("In some_function a = %d b = %d\n", a, b);
 }
```

On invocation, this program displays the following:

```
In some_function a = 5 b = 10
In some_function a = 9 b = 11
In main a = 5 b = 10
```

If you want to modify the actual parameters within a function, you must use pointers. For example, assume that you have two variables (*a* and *b*) that you want to initialize by using a function. To do so, you must pass the addresses of each variable to the function by using the ampersand operator, as shown here:

```
main ()
 {
  int a, b;

  initialize (&a, &b);

  printf ("a = %d b = %d\n", a, b);
 }
```

Within the function, you must specify to C that you are using pointers, as shown here:

```
initialize (int *a, int *b)
 {
  *a = 1;
  *b = 2;
 }
```

On invocation, this program displays the following:

```
a = 1 b = 2
```

This is because the function variables referenced the same memory locations as the actual parameters (see Figure 3-3).

In a similar manner, the following function increments all three of its parameters:

```
increment (int *a, int *b, int *c)
 {
  (*a+)+;
  (*b)++;
  (*c)++;
 }
```

Once again, invoke the function as follows:

```
increment (&value1, &value2, &value3);
```

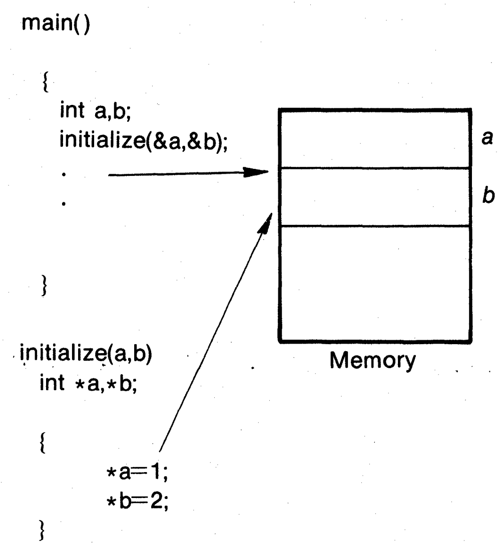


Figure 3-3. *Variables referencing same memory locations as parameters*

Pointers and Strings

Probably the greatest use of pointers in C is for string manipulation. Each time C passes an array to a function, it passes the address of the first element in the array. Remember, strings in C are treated as arrays.

Since you are dealing with memory addresses, this provides an ideal application for pointers. Consider the following function, which displays the first character in the array of characters that it receives:

```
show_first (char *s)
 {
 printf ("%c\n", *s);
 }
```

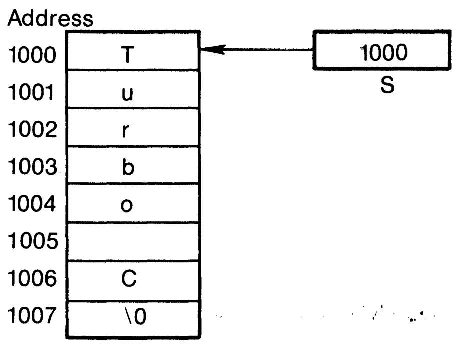


Figure 3-4. *References of* s *in "Turbo C" string*

On invocation, *s* points to the first letter in the string. Assuming that the string is "Turbo C", *s* references as shown in Figure 3-4. If you simply add 1 to the memory address, you can point to the second letter in the string, as shown here:

```
show_second (char *s)
 {
  s++;                  /* point to second character */
  printf ("%c\n", *s);
 }
```

The following routine displays the contents of the string it receives:

```
show_string (char *s)
 {
  while (*s != '\0')
     {
       printf ("%c", *s);
       s++;
     }
 }
```

With each iteration, you simply add 1 to the pointer (called *incrementing the address*).

Next, you can reduce the code to an even greater extent by using the following expression:

```
*s++
```

In this case, C will first use the value contained in the memory address referenced by *s* and then increment it. As such, the code fragments

```
a = *s++;
```

and

```
a = *s;
s++;
```

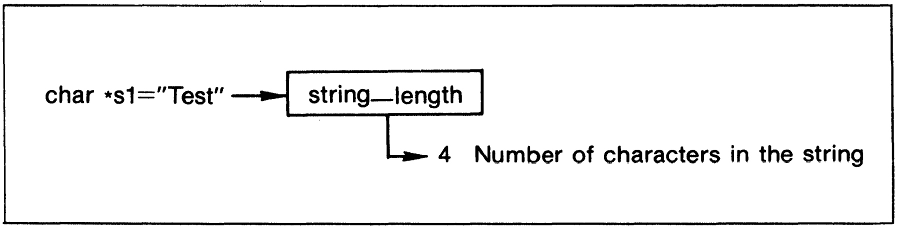
perform identical functions. With this concept in mind, you can modify the code as shown here:

```
show_string (char *s)
 {
  while (*s != '\0')
      printf ("%c", *s++);
 }
```

The following routines make extensive use of pointers. Many of these routines perform functions that are similar to the functions of routines presented in Chapter 2. You must understand the processing involved in the routines that follow.

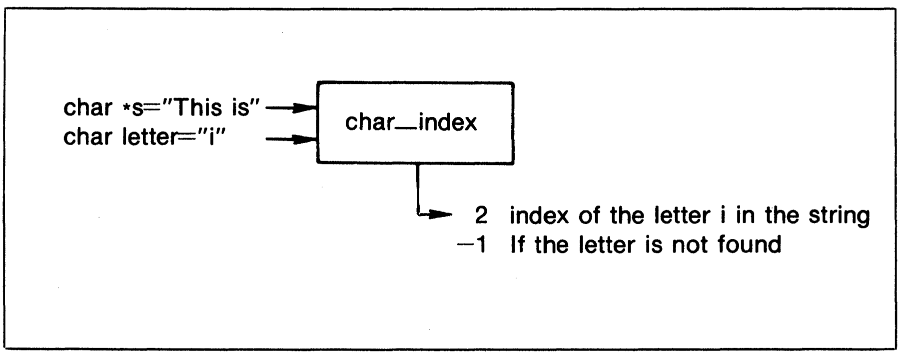
String-Manipulation Routines

The first function, string_length, returns the number of characters in a string:



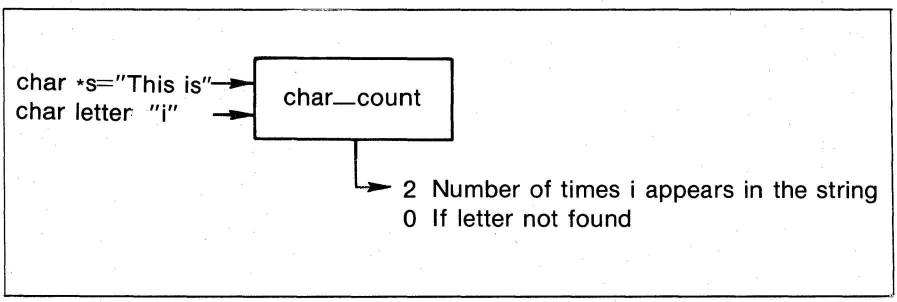

```
/*
 * string_length (s)
 *
 * Return the number of characters in the string.
 *
 * s (in): string to count the characters in.
 *
 * length = string_length ("This is a test");
 *
 */

string_length (char *s)
 {
  int len = 0;

  while (*s++)
    len++;

  return (len);
 }
```

In a similar manner, the routine char_index returns the first occurrence of the letter specified within the string. If the letter is not found, char_index returns the value −1.



```
/*
 * char_index (s, letter)
 *
 * Return the index of the first occurrence of letter in
 * the character string specified.
 *
 * s (in): string to search for the letter.
 * letter (in): character to search for.
 *
 * index_value = char_index ("This is a test", 'i');
 *
 */

char_index (char *s, char letter)
 {
  int count, location = -1;

  for (count = 0; *s && (location == -1); count++)
    if (*s++ == letter)
      location = count;

  return (location);
 }
```

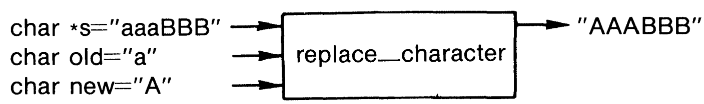
The routine char_count returns the number of occurrences of the letter specified within the string. If the letter is not found, char_count returns the value 0.




```
/*
 * char_count (s, letter)
 *
 * Return the number of occurrences of a letter in the
 * character string specified.
 *
 * s (in): string to search for the letter.
 * letter (in): character to search for.
 *
 * count = char_count ("This is a test", 'i');
 *
 */
```

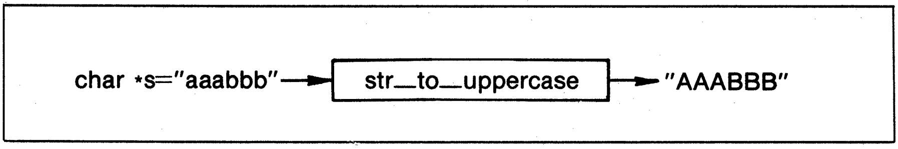
```
char_count (char *s, char letter)
 {
  int count = 0;

  while (*s)
    if (*s++ == letter)
      count++;

  return (count);
 }
```

The function replace_character replaces each occurrence of the first letter specified in the first string with the letter contained in the second string.

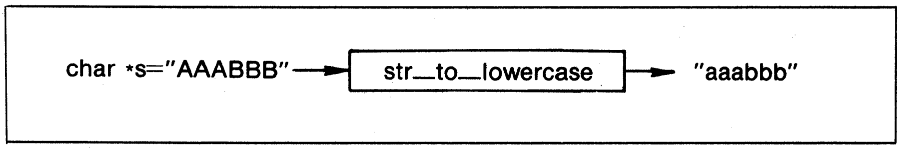



```
/*
 * void replace_character (s, oldletter, newletter)
 *
 * Replace each occurrence of oldletter in a string with
 * the character contained in newletter.
 *
 * s (in/out): string to replace the letters in.
 * oldletter (in): letter to replace.
 * newletter (in): replacement letter.
 *
 * replace_character (some_string, 'A', 'a');
 *
 */

void replace_character (char *s, char oldletter, char newletter)
 {
  while (*s)
    if (*s == oldletter)
      *s++ = newletter;
    else
      s++;
 }
```

In a manner similar to the routines in Chapter 2, the following routines use bit manipulation to convert a string to uppercase or lowercase letters:

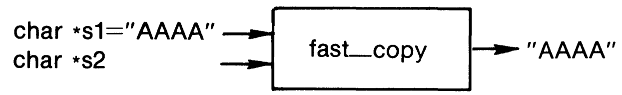



```
/*
 * void str_to_uppercase (s)
 *
 * Convert a string to UPPERCASE characters.
 *
 * s (in/out): string to convert to UPPERCASE.
 *
 * str_to_uppercase (filename);
 *
 * str_to_uppercase uses bit manipulation to convert characters
 * to uppercase.
 *
 */

void str_to_uppercase (char *s)
 {
   while (*s)
    if (*s >= 'a' && *s <= 'z')
      *s++ &= ~32;
    else
      s++;
 }
```




```
/*
 * void str_to_lowercase (s)
 *
 * Convert a string to lowercase characters.
 *
 * s (in/out): string to convert to lowercase.
 *
 * str_to_lowercase (filename);
 *
 * str_to_lowercase uses bit manipulation to convert characters
 * to lowercase.
 *
 */

void str_to_lowercase (s)
  char *s;
 {
   while (*s)
    if (*s >= 'A' && *s <= 'Z')
      *s++ |= 32;
    else
      s++;
 }
```

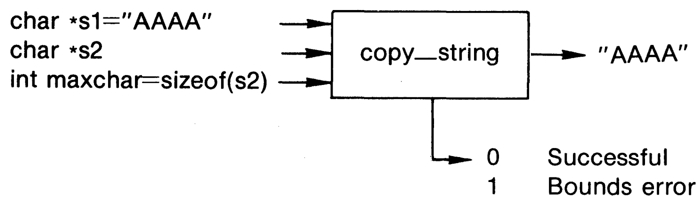
This routine copies the contents of the first string specified to the second string. This routine does not perform bounds checking.



Warning: fast_copy does not perform bounds checking.

```
/*
 * void fast_copy (source, target)
 *
 * Copy the contents of the source string to the target.
 *
 * s1 (in): source string containing characters to copy.
 * s2 (out): string receiving characters copied.
 *
 * fast_copy ("This is a test", stringvar);
 *
 * fast_copy does not perform bounds checking.
 *
 */

void fast_copy (s1, s2)
  char *s1, *s2;
 {
  while (*s2++ = *s1++)
      ;
 }
```

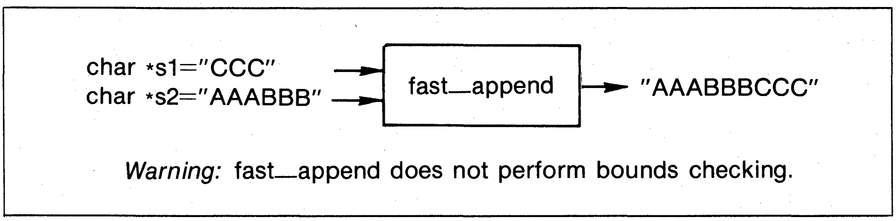
You can implement bounds checking simply by adding the maxchar qualifier (as shown in Chapter 2):




```
/*
 * int string_copy (source, target, array_bound)
 *
```

```
 * Copy the source string to the target string variable.
 *
 * s1 (in): contains the characters to be copied.
 * s2 (out): receives the characters copied.
 * maxchar (in): specifies the maximum number of characters
 *              that s2 can store.
 *
 * status = string_copy ("This is", stringvar, sizeof (stringvar));
 *
 * If the array bounds are exceeded, string_copy returns the value
 * 1; otherwise it returns the value 0.
 *
 */

int string_copy (char *s1, char *s2, int maxchar)
 {
  int i;

  maxchar--;            /* leave space for null */

  for (i = 0; (*s2++ = *s1++) && (i < maxchar); i++)
     ;

  if ((i == maxchar) && *s1)    /* see if characters remain in s1 */
    {
     *s2 = '\0';
     return (1);
    }
  else
    return (0);
 }
```

The routine fast_append appends the contents of the first specified string to the second string. No bounds checking is performed.

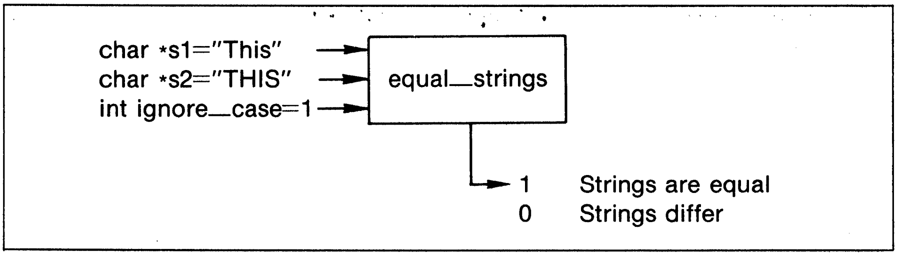



```
/*
 * void fast_append (source, target)
 *
 * Append the contents of the source string to the target.
 *
 * s1 (in): source string containing characters to append.
 * s2 (out): string receiving characters copied.
 *
 * fast_append ("This is a test", stringvar);
 *
 * fast_append does not perform bounds checking.
 *
 */
```

```
void fast_append (char *s1, char *s2)
 {
  while (*s2)     /* find the end of s2 */
     s2++;

  while (*s2++ = *s1++)   /* append s1 */
     ;
  }
```

The following routines perform case-sensitive string comparisons.

The first routine, equal_strings, returns the value 1 if two strings are identical. Otherwise, it returns 0.

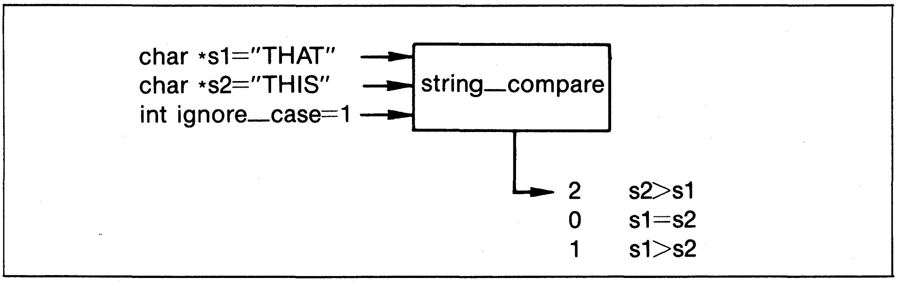



```
/*
 * equal_strings (s1, s2, ignore_case)
 *
 * Return 1 if the strings s1 and s2 are equal, otherwise return
 * 0. Support case sensitive processing.
 *
 * s1 (in): string to compare.
 * s2 (in): string to compare.
 * ignore_case (in): if not 0, case of letters is ignored.
 *
 * if (equal_strings ("THIS", "this", 1))
 *
 * equal_strings returns 1 if the strings are equal, 0
 * otherwise.
 *
 */

int equal_strings (char *s1, char *s2, int ignore_case)
 {
  char a, b;

  for (; *s1 && *s2 ; s1++, s2++)
    if (*s1 != *s2)
      {
        if (ignore_case)
         {
          a = (*s1 >= 'a' && *s1 <= 'z') ? *s1 & ~32: *s1;
          b = (*s2 >= 'a' && *s2 <= 'z') ? *s2 & ~32: *s2;
          if (a != b)
            break;
```

```
        }
      else
        break;
    }

  if (*s1 || *s2)
    return (0);
  else
    return (1);
}
```

The routine string_compare returns one of the following values:

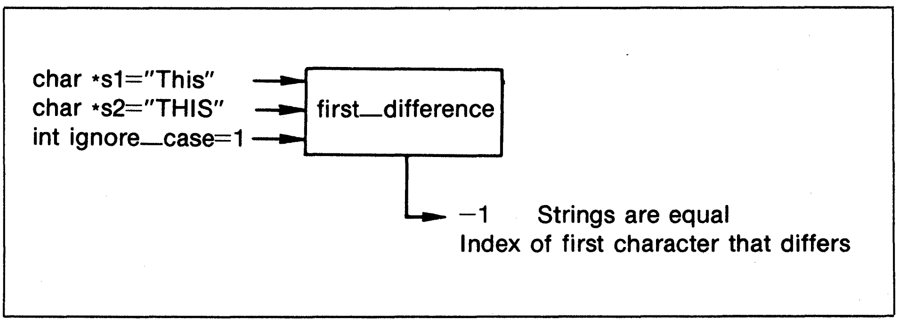
0	Strings are equal
1	String 1 > String2
2	String 1 < String2




```
/*
 * string_comp (s1, s2, ignore_case)
 *
 * Compare the strings specified. Return 1 if s1 > s2, 2 if
 * s2 > s1 and 0 if the strings are equal. Support case sensitive
 * processing.
 *
 * s1 (in): string to compare.
 * s2 (in): string to compare.
 * ignore_case (in): if not 0, case of letters is ignored.
 *
 * if (string_comp ("THIS", "this", 1) == 1)
 *
 */

int string_comp (char *s1, char *s2, int ignore_case)
 {
  char a, b;
  int result = 0;     /* 0 equal, 1 s1 greater, 2 s2 greater */

  for (; *s1 && *s2; s1++, s2++)
    if (*s1 != *s2)
```

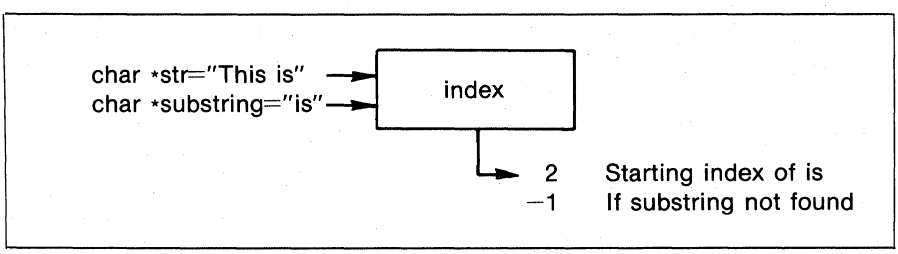
```
    {
      if (ignore_case)
        {
         a = (*s1 >= 'a' && *s1 <= 'z') ? *s1 & ~32: *s1;
         b = (*s2 >= 'a' && *s2 <= 'z') ? *s2 & ~32: *s2;
         if (a != b)
           {
            if (a > b)
              result = 1;
            else
              result = 2;

            break;
           }
        }
      else
        {
         if (*s1 > *s2)
           result = 1;
         else
           result = 2;

         break;
        }
    }

 if (result == 0)
  {
   if (*s1 == *s2)
     result = 0;
   else if (*s1)
     result = 1;
   else
     result = 2;
  }
 return (result);
}
```

The routine first_difference returns the index of the first character that differs between two strings. If the strings are equal, the routine returns the value −1.



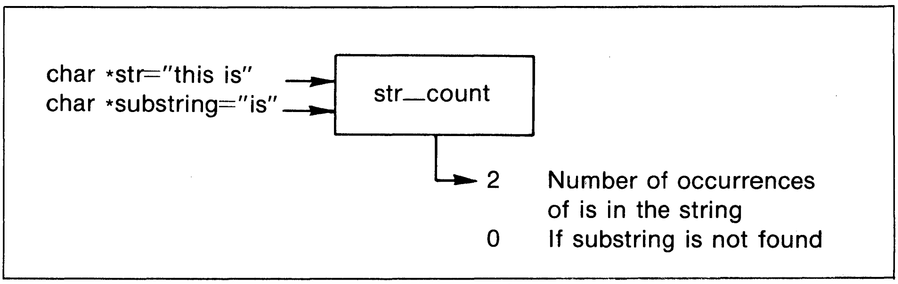
```
/*
 * first_difference (s1, s2, ignore_case)
 *
 * Return the index location of the first character that differs
 * between s1 and s2.  If the strings are equal, return the value -1.
 *
 * s1 (in): string to compare.
 * s2 (in): string to compare.
 * ignore_case (in): If 1, ignore case of letters.
 *
 * location = first_difference ("TURBO C", "turbo c", 1);
 *
 */

int first_difference (char *s1, char *s2, int ignore_case)
 {
  int i;
  char a, b;

  for (i = 0; *s1 && *s2; s1++, s2++, i++)
    if (*s1 != *s2)
     {
       if (ignore_case)
        {
         a = (*s1 >= 'a' && *s1 <= 'z') ? *s1 & ~32: *s1;
         b = (*s2 >= 'a' && *s2 <= 'z') ? *s2 & ~32: *s2;
         if (a != b)
           break;
        }
       else
         break;
     }

  if (*s1 || *s2)
    return (i);
  else
    return (-1);
 }
```

The routine index returns the starting index of a substring within a string. If the substring is not found, the routine returns the value −1.



```
/*
 * index (substring, string)
 *
 * Return the starting index of the substring within a string
 * or the value -1 if the substring is not found.
 *
 * substring (in): substring to search for.
 * string (in): string to examine.
 *
 * if (index ("PATH=", *ENV[1]) != -1)
 *
 */

int index (char *substr, char *str)
 {
  char *substring, *string, *start = str;

  while (*str)
     for (string = str++, substring = substr;
       *string == *substring; string++, substring++)
       if (! *(substring+1))          /* end of substring */
         return (str - start - 1);

  return (-1);                        /* substring not found */
 }
```

The function str_count returns a count of the number of occurrences of a substring within a string. If the substring does not occur in the string, the value 0 is returned.




```
/*
 * str_count (substring, string)
 *
 * Return the number of occurrences of the substring in the string
 * or the value 0 if the substring is not found.
 *
 * substring (in): substring to search for.
 * string (in): string to examine.
 *
```

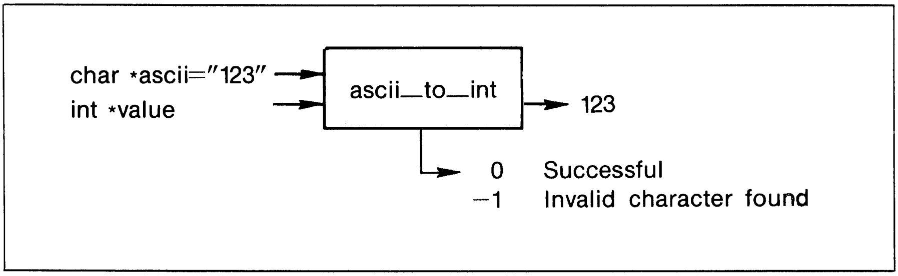
```
 * count = str_count ("is", "This is a test");
 *
 */

int str_count (char *substr, char *str)
 {
  char *substring, *string;

  int count = 0;

  while (*str)
     for (string = str++, substring = substr;
       *string == *substring; string++, substring++)
       if (! *(substring+1))        /* end of substring */
         ++count;

  return (count);                   /* substring not found */
 }
```

Conversion Routines

Periodically a program must convert a string representation of a value to its numeric format. Consider the following program:

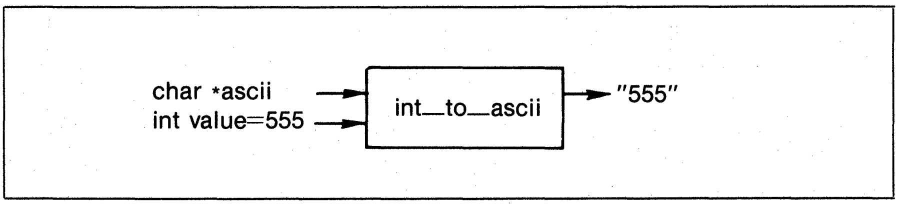
```
main ()
 {
  char agestr[5];
  int age;

  printf ("Enter your age\n");

  gets (agestr);

  if (ascii_to_int (agestr, &age) != -1)
      printf ("%d\n", age);
  else
      printf ("Invalid age entered\n");
 }
```

The program invokes the function ascii_to_int, which converts a string representation of an integer value to an actual value of type int. If the string contains "1233", the routine returns the integer value 1233. However, if the string contains an invalid character (such as "123d3"), the routine returns an error status value.




```
/*
 * ascii_to_int (str, value)
 *
 * Convert a string representation of a numeric value to the
 * actual integer value.
 *
 * str (in): string containing numeric representation.
 * value (out): actual integer value.
 *
 * if (ascii_to_int ("1112", &value) != -1)
 *
 * If the character contains invalid characters, the value -1 is
 * returned.
 *
 */

ascii_to_int (char *str, int *value)
 {
  int sign = 1;  /* -1 if negative value */

  *value = 0;

  while (*str == ' ')  /* skip leading blanks */
    str++;

  if (*str == '-' || *str == '+')
    sign = (*str++ == '-') ? -1: 1;

  while (*str)
    if ((*str >= '0') && (*str <= '9'))
      *value = (*value * 10) + (*str++ - 48);
    else
      return (-1);  /* invalid character */

  *value *= sign;

  return (0);
 }
```

In just the opposite manner, the routine int_to_ascii converts an integer value to its string representation.




```
/*
 * int_to_ascii (value, str)
 *
 * Convert an integer value to its character string representation.
 *
 * str (out): string to contain the numeric representation.
 * value (in): integer value to convert.
 *
 * int_to_ascii (str, 22);
 *
 */

int_to_ascii (int value, char *str)
 {
  int sign = value;

  char temp, *savestr = str;

  if (value < 0)
    value *= -1;

  do
   {
    *str++ = (value % 10) + 48;
    value = value / 10;
   }
  while (value > 0);

  if (sign < 0)
    *str++ = '-';

  *str-- = '\0';

  while (savestr < str)
    {
     temp = *str;
     *str-- = *savestr;
     *savestr++ = temp;
    }
 }
```

Admittedly, this has been a fast trip through pointer-manipulation routines. If you do not yet feel comfortable with the concept of a pointer, experiment with the previous routines before proceeding.

Arrays of Pointers

Just as C allows you to have an array of characters, it also allows you to create an array of pointers. For example, consider the following definition:

```
char *summer [3];
```

C will create an array indexed from 0 to 2 that contains three pointers to character strings. You can assign values to each string element and then display them, as shown here:

```
main ()
 {
  char *summer[3];

  int i;

  summer [0] = "June";
  summer [1] = "July";
  summer [2] = "August";

  for (i = 0; i < 3; i++)
    printf ("%s\n", summer [i]);
 }
```

Using this concept, you can define many useful arrays of pointers to character strings, as shown here:

```
char *DAYS[7] = {"Sunday", "Monday", "Tuesday",
                 "Wednesday", "Thursday", "Friday",
                 "Saturday" };
```

This program uses an array of pointers to display an ASCII, decimal, octal, and hexadecimal chart.

```
char *ascii[] = {"NUL", "SOH", "STX", "EXT", "EOT",
                 "ENQ", "ACK", "BEL", "BS ", "HT ",
                 "LF ", "VT ", "FF ", "CR ", "SO ",
                 "SI ", "DLE", "DC1", "DC2", "DC3",
                 "DC4", "NAK", "SYN", "ETB", "CAN",
                 "EM ", "SUB", "ESC", "FS ", "GS ",
                 "RS ", "US ", "SPACE" };
```

```
main ()
 {
   int i;

   for (i = 0; i < 33; i++)
     printf ("%03d %03o %03x %s\n", i, i, i, ascii[i]);

   for (; i < 128; i++)
     printf ("%03d %03o %03x %3c\n", i, i, i, i);
 }
```

In order to fully exploit the capabilities found in C, you must be able to understand and utilize arrays of pointers to character strings.

Command-Line Processing

Each time you enter a command from the DOS prompt, the sequence of characters you enter constitutes a command line, as shown here:

```
A> COPY SOURCE TARGET
```

One of the most powerful capabilities of C is that it allows easy access to the command line used to invoke the program. In order to exploit this access, you must define main within your program, as shown here:

```
main (int argc, char *argv[])
 {

 }
```

These two parameters provide your access to the command line. The first, *argc*, contains the number of command-line arguments. The second, *argv*, is an array of pointers to character strings that contain the actual arguments.

Before your C program executes, built-in header code (which assigns the number of command-line arguments to *argc* and the actual arguments to the elements of the array *argv*) executes. Once this processing is complete, this code invokes main. In the following command line,

```
A> COPY SOURCE TARGET
```

the variable *argc* will contain the value 3 and the elements of *argv* will point to the following:

```
argv[0] points to COPY
argv[1] points to SOURCE
argv[2] points to TARGET
```

You can verify this simply by executing the following program, which displays each of its command-line parameters:

```
main (int argc, char *argv[])
 {
  int i;
  for (i = 0; i < argc; i++)
    printf ("%s\n", argv[i]);
 }
```

The true power of command-line processing is shown when you examine the file-manipulation programs later in this text. For now, here is a program that displays the contents of the file specified by a command-line argument:

```
A> SHOW FILENAME.EXT
```

```
#include <stdio.h>

main (int argc, char *argv[])
 {
```

```
FILE *fopen (), *fp;

char str[255];

if (argc > 1)
  if (!(fp = fopen (argv[1], "r")))
    printf ("Invalid file %s\n", argv[1]);
  else
    {
      while (fgets (str, 255, fp))
        printf ("%s", str);

      fclose (fp);
    }
}
```

Accessing Environment Entries

You may be familiar with the DOS environment, which is a region of memory that DOS sets aside to store information.

For example, issue the DOS set command as follows:

```
A> SET
```

DOS will display the contents of its environment entries:

```
COMSPEC=C:\COMMAND.COM
PATH=C:\DOS;C:\TURBOC
```

To place a value in the environment, simply use the SET command as shown here:

```
A> SET FILENAME=TEST
```

To verify that the entry was successful, again issue the SET command:

```
A> SET
COMSPEC=C:\COMMAND.COM
PATH=C:\DOS;C:\TURBOC
FILENAME=TEST
```

Many programs often require access to the environment entries. As such, Turbo C allows you to access the environment in a manner similar to the command line. Once again, you must modify your definition of main:

```
main (int argc, char *argv[], char *env[])
 {

 }
```

Just as *argv* is an array of pointers to the command line, *env* is an array of pointers to the environment entries. This program uses *env* to display the current environment.

```
main (int argc, char *argv[], char *env[])
 {
  while (*env)
     printf ("%s\n", *env++);
 }
```

The following program combines command-line manipulation with environment processing to display the value of a specific environment entry.

```
A> SHOWENV PATH=
```

In this case, the program displays the value associated with the PATH entry (if it is found):

```
main (int argc, char *argv[], char *env[])
 {
  void str_to_uppercase();

  if (argc > 1)
   {
```

```
    str_to_uppercase (argv[1]);

    while (*env)
      if (index (argv[1], *env) == 0)
        printf ("%s\n", *env++);
      else
        env++;
  }
}
```

Some programmers have difficulty opening files that do not reside in a fixed directory or the current directory. The DOS environment may provide a solution. Assume that the file you need to open is called DATA.DAT. This file can reside in any DOS subdirectory. As such, you can simply place an entry in the environment that tells you where the file resides:

```
A> SET DATAFILE=C:\SOMEDIR\DATA.DAT
```

The program then uses this information to locate the file in order to successfully open the file. The program simply searches each environment entry as shown in the previous program.

Far Pointers

Each of the pointers used thus far was a 16-bit address. Such pointers are termed *near* pointers because they can only be used to access memory locations within the 64KB data segment. Most C programs never have access to memory regions beyond this.

However, an advanced program periodically has a requirement to access memory outside of this region. In such cases, the program must use *far* pointers, which contain 32-bit addresses. Unlike near pointers (which can only offset into the current 64KB data segment), a far pointer allows you to define a 16-bit segment along with a 16-bit offset address (see Figure 3-5).

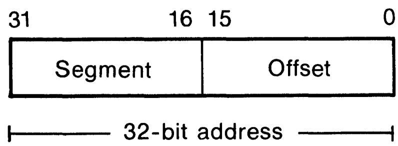


Figure 3-5. *Offset address of far pointer*

Examine the DOS memory map shown in Figure 3-6. Note that the computer uses the memory region B800:0000 to address the color video display memory. Knowing this, you can use a far pointer to reference this memory region. In so doing, you can perform memory-mapped output.

The computer displays characters by placing the ASCII code for the character into one of these memory locations, followed immediately by the character's display attribute value (color, boldface, and so on), as shown in Figure 3-7. As such, you can place the letter "A" in the upper left corner of the screen, as shown here:

```
main ()
 {
  char far *letter = 0xB8000000;
  char far *attr = 0xB8000001;

  *letter = 65;
  *attr = 7;
 }
```

DOS Power User's Guide (Kris Jamsa, Osborne/McGraw-Hill, 1988) provides several example routines that use far pointers. In fact, that book contains a chapter dedicated to memory mapping. in most cases, you will use near pointers for your manipulation. How-

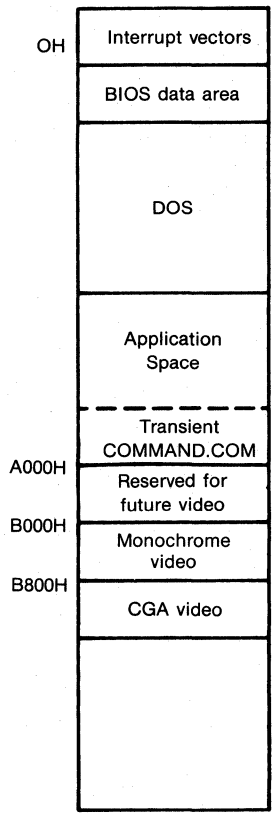


Figure 3-6. *Color video display memory address*

ever, you should understand the functional capabilities of far pointers.

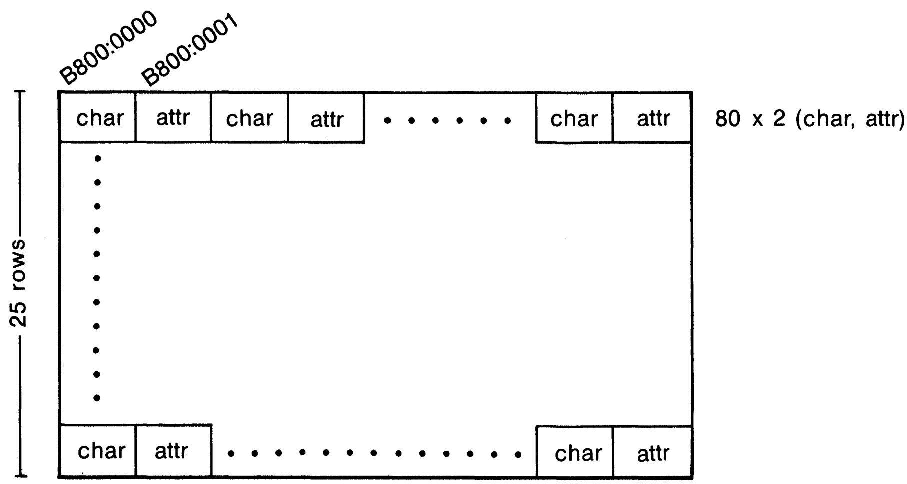


Figure 3-7. *Displaying a character on screen using ASCII codes and display attributes*

CHAPTER

4

Recursion

Earlier chapters presented several programs and functions that perform their processing by invoking other functions. With Turbo C, a function (and even a program) can invoke itself to perform a specific task. This is known as *recursion.* Many advanced programmers use recursion to greatly reduce the amount of code in their programs.

In later chapters, you will examine the manipulation of dynamic variables to perform specific tasks. Many of the algorithms for those programs will be recursive. As you examine the routines in this chapter, you will find that many of them have been previously implemented nonrecursively. In most cases you can implement a function more efficiently without recursion than with it. (The reasons for this are presented later in this chapter.) However, because many Turbo C programmers make extensive use of recursion in their programs, you should understand the general flow of control for recursive functions.

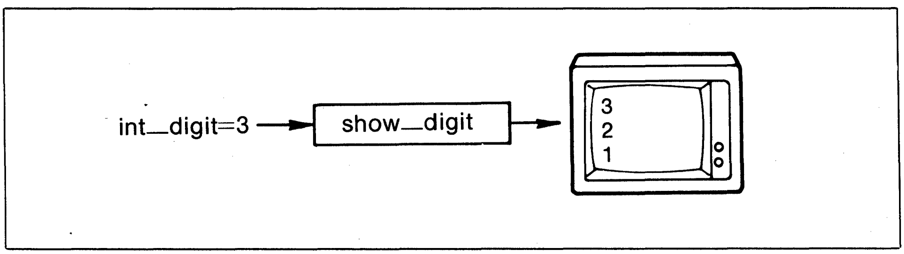
Take the time to experiment with the routines presented in this chapter, and you will find recursion to be a straightforward, powerful, and even enjoyable feature of Turbo C programming.

Getting Started with Recursion

The following program invokes the routine show_digit with the value 5. The function show_digit in turn uses printf to display the value it receives. The routine then invokes itself with the value −1 (in this case, 4). This process repeats until the value received is equal to 0. When invoked, the program displays the following:

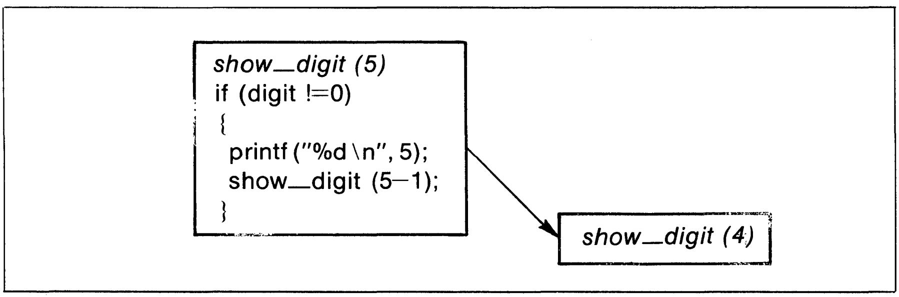
```
A> SHOWDGT
5
4
3
2
1

A>
```

The following code implements show_digit:

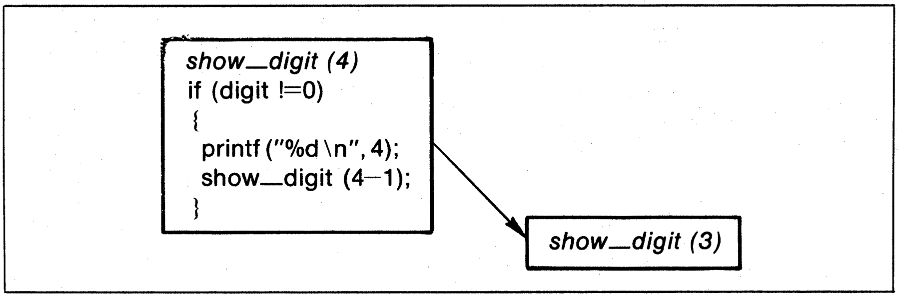


```
/*
 * void show_digit (digit)
 *
 * Recursively display the numbers from digit to zero.
 *
 * digit (in): starting number of the digits to display.
 *
 * show_digit (7);
 *
 */

void show_digit (int digit)
 {
  if (digit != 0)
   {
    printf ("%d\n", digit);
    show_digit (--digit);
   }
 }
```

On the first invocation of show_digit, the parameter digit contains the value 5. The function then displays that value and invokes itself with the value 4:

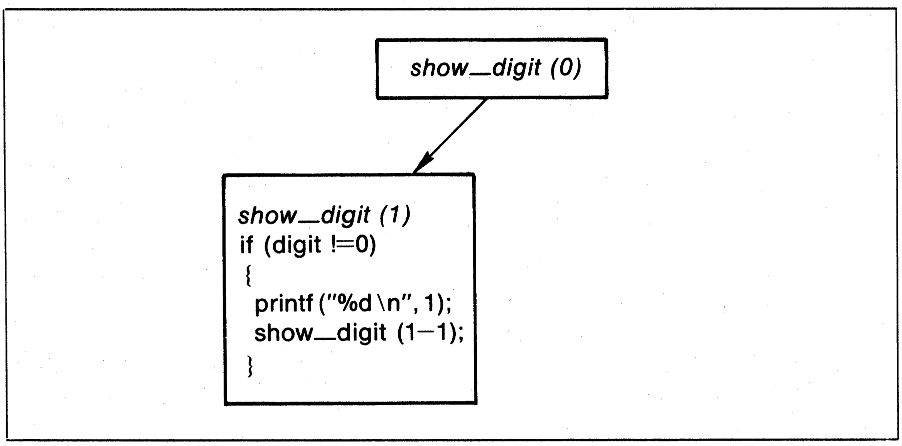


The second invocation of the function displays the value 4 and then invokes itself with the value 3:



This process repeats until the value of digit is 0, as shown in Figure 4-1.

Once digit is 0, show_digit no longer invokes itself recursively. The last invocation of the function terminates and returns control to the previous invocation.



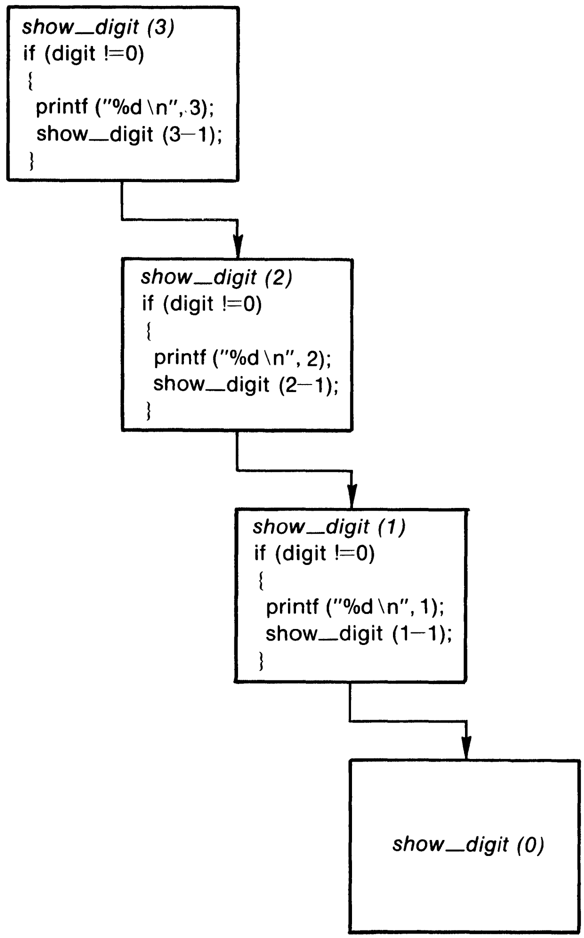
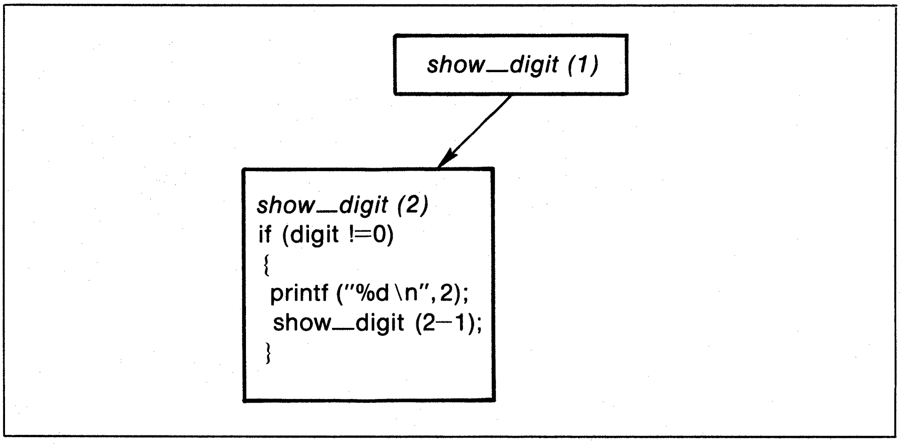


Figure 4-1. *Processing involved in show_digit program (until the value of digit is 0)*



This process repeats until no invocation of show_digit is active, as shown in Figure 4-2. A recursive function, then, is one that calls itself until an ending condition is met.

Chapter 2 discussed how Turbo C terminates strings with the null character. With this concept in mind, you can write a recursive function that determines the number of characters in a string by searching for the null character. Given the string "ABC", the function examines the first character. If the current character in the string is not the null character, the routine simply adds the value of 1 to the value returned by the next invocation of the routine string_length. Thus, by using "ABC", the processing becomes that shown in Figure 4-3.

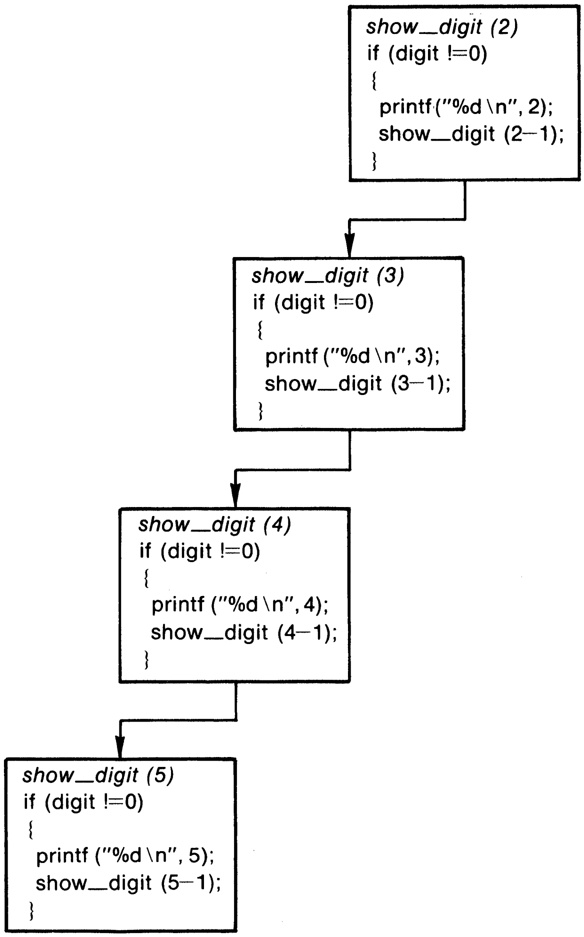


Figure 4-2. *Processing involved in show_digit program (until program is no longer active)*

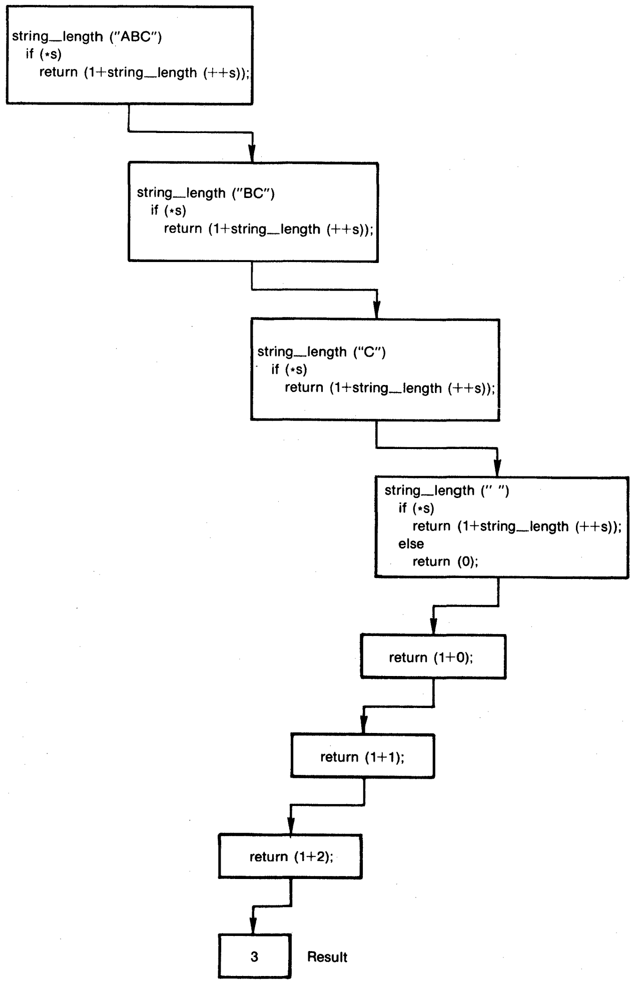
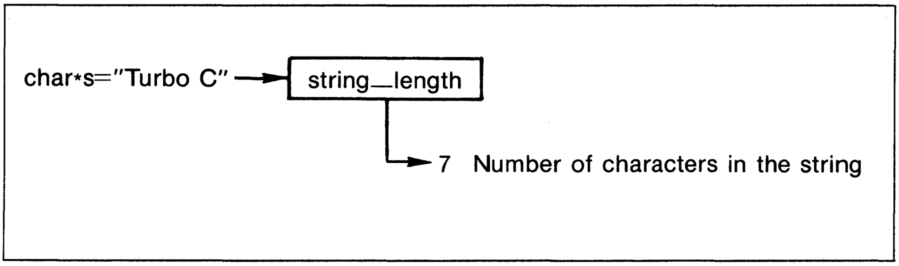


Figure 4-3. *Processing involved in string_length program*

Once it locates the null character, the routine simply works its way back through the series of recursive invocations. The following routine implements string_length:




```
/*
 * string_length (s)
 *
 * Return the number of characters in the string.
 *
 * s (in): string to return the length of.
 *
 * count = string_length (string);
 *
 */

int string_length (s)
  char *s;
 {
   return ((*s) ? 1 + string_length (++s): 0);
 }
```

Similarly, the routine display_string displays the contents of a character string by using recursion. The routine begins by examining the current character. If that character is not null, the routine displays the character and then invokes itself with the next character. This process repeats until the null character is found. Given the string "ABC", the processing becomes that shown in Figure 4-4.

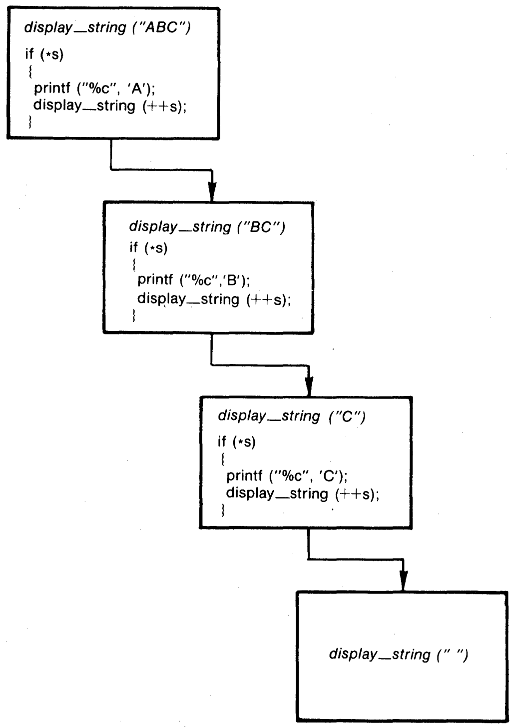
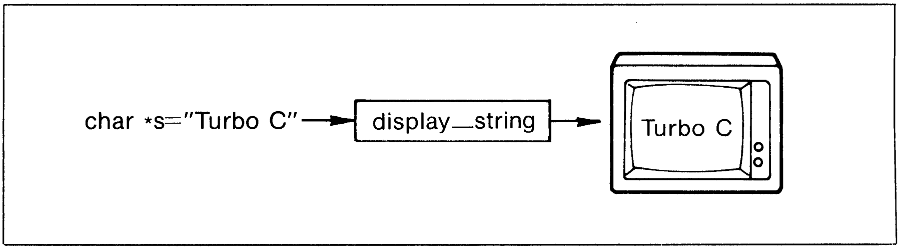


Figure 4-4. *Processing involved in display_string program*

In the following case, once the routine locates the null character, no further processing is required. The function simply returns control to the previous invocation.




```
/*
 * display_string (string)
 *
 * Display the contents of a character string on the screen display.
 *
 * string (in): character string to display.
 *
 * display_string ("This is a test");
 *
 */

display_string (char *s)
 {
  if (*s)                      /* do characters remain? */
    {
     printf ("%c", *s);        /* output current character */
     display_string (++s);     /* recursively display other characters */
    }
 }
```

By changing the following lines of code,

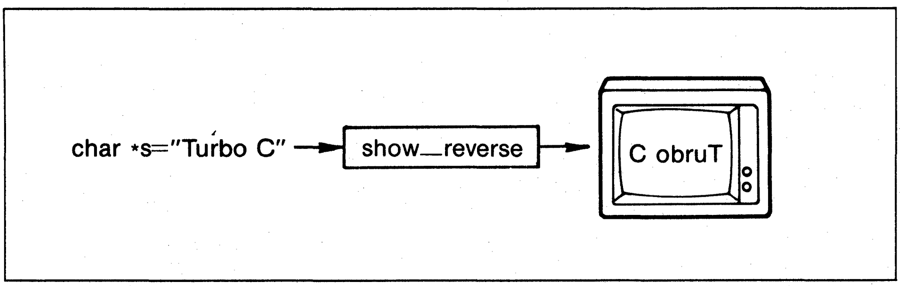
```
printf ("%c", *s);
display_string (++s);
```

to

```
display_string (s+1);
printf ("%c", *s++);
```

the routine displays the string in reverse order. This is because the function first examines the current character in the string. If the character is not the null character, the routine invokes itself recursively with the next character. This process repeats until the routine locates the null character. Once the null character is found, the routines begin working their way back through the series of invocations to display the characters in reverse order, as shown in Figure 4-5.

The following code implements show_reverse:




```
/*
 * show_reverse (string)
 *
 * Display the contents of a character string on the screen display
 * in reverse order.
 *
 * string (in): character string to display.
 *
 * show_reverse ("This is a test");
 *
 */

show_reverse (char *s)
 {
  if (*s)                    /* do characters remain? */
   {
    show_reverse (s+1);      /* recursively show other characters */
    printf ("%c", *s);       /* output current character */
   }
 }
```

Probably the most popular use of recursion is for determining the factorial of a value. Table 4-1 illustrates how to calculate the factorials for the values 1 through 5. The factorial of 5 is

5 * factorial (4)

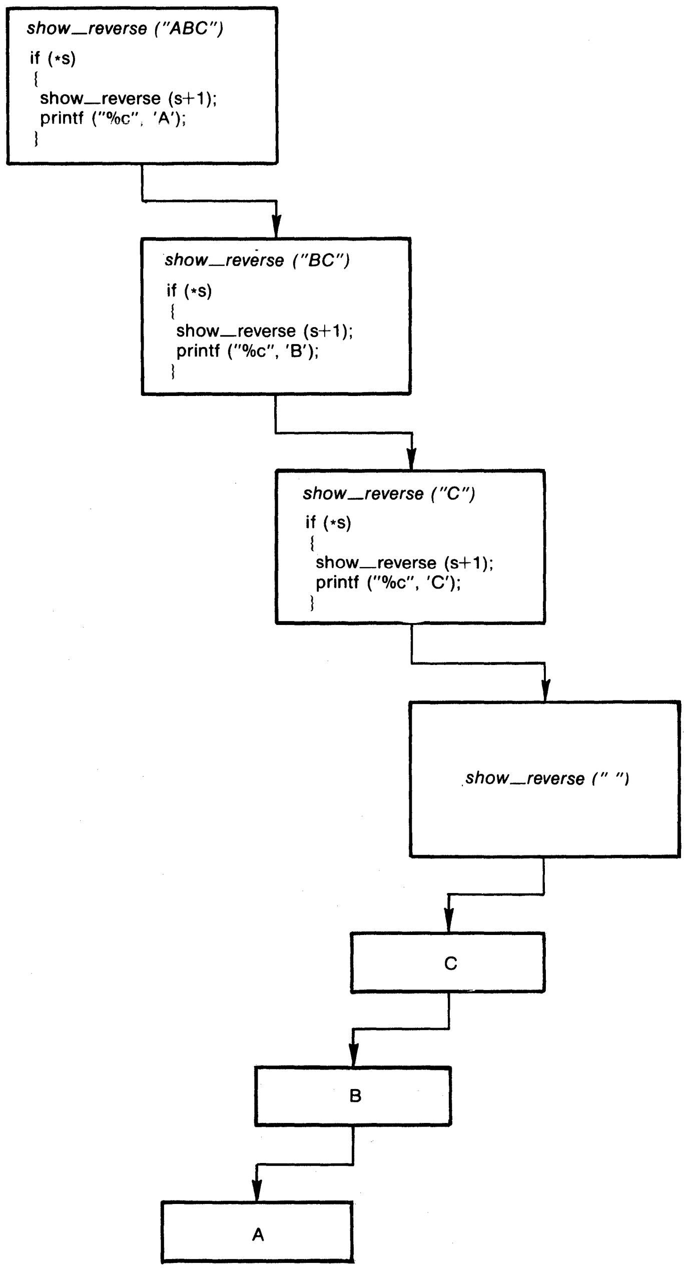


Figure 4-5. Processing involved in show_reverse program

Table 4-1. Factorials of Values 1 Through 5

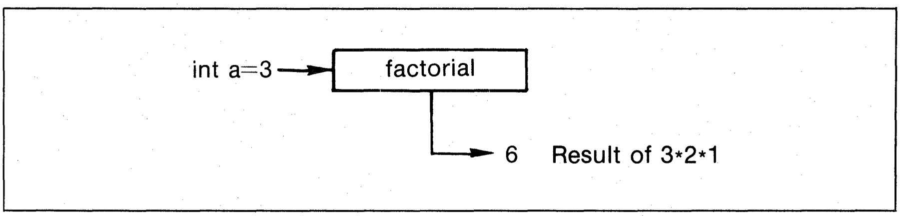
Value	Definition	Result
1	1	1
2	1*2	2
3	1*2*3	6
4	1*2*3*4	24
5	1*2*3*4*5	120

The factorial of 4 is

4 * factorial (3)

This process continues until the factorial of 1 (which, by definition, is 1) is reached. For example, to determine the factorial of the value 3, the processing shown in Figure 4-6 is performed.

The following routine implements factorial:




```
/*
 * factorial (value)
 *
 * Return the factorial of the value specified.
 *
 * value (in): value to return the factorial of.
 *
 * fact = factorial (5);
 *
 */

factorial (int value)
 {
  return ((value <= 1) ? 1: value * factorial (value-1));
 }
```

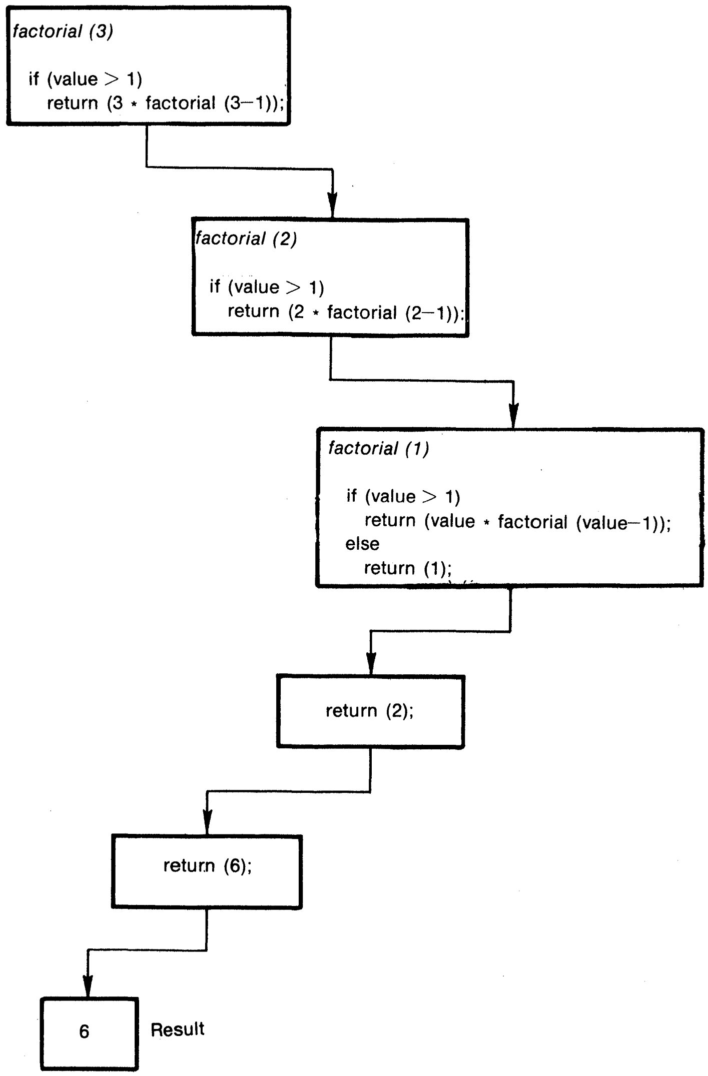
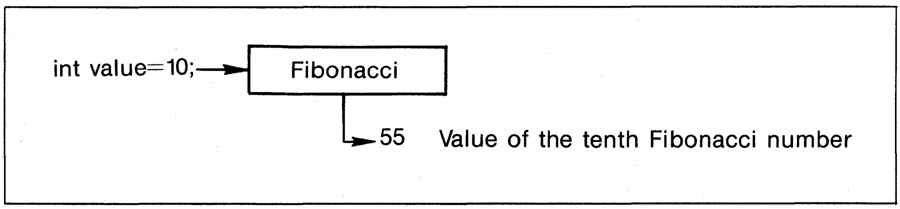


Figure 4-6. *Determining the factorial of the value 3*

Similarly, recursion can be used to compute a *Fibonacci number*. Table 4-2 shows you how to calculate the Fibonacci numbers from 1 to 5. The following code implements Fibonacci:




```
/*
 * fibonacci (value)
 *
 * Return the Fibonacci number for the value specified.
 *
 * value (in): value to return the Fibonacci number of.
 *
 * fibon = fibonacci (10);
 *
 * A Fibonacci number is the sum of the previous two Fibonacci numbers.
 *
 */

fibonacci (int value)
 {
  return ((value == 1 || value == 2) ? 1:
           fibonacci (value - 1) + fibonacci (value - 2));
 }
```

C is a *portable programming language*, which means that the code that you write on one type of computer in C will likely recompile and run on a different type of computer (with little or no modification). Portability is one of C's most important characteristics. However, exceptions to the rule always exist. Depending on your target computer, the number of bits that C uses to represent value of type int may differ from 16 to 32 bits. Thus, the range of values that each can store may also differ (see Table 4-3).

In either case, C always uses the most significant (leftmost) bit of an integer value as the sign bit, as shown in Figure 4-7. When this bit is set (1), the value contained in the lower order bits is considered a negative value. When this bit is clear (0), the value is positive. You

Table 4-2. Calculation of Fibonacci Numbers 1 Through 5

Value	Definition	Fibonacci Number
1	1	1
2	1	1
3	1+1	2
4	2+1	3
5	3+2	5

Table 4-3. Number of Bits Versus Minimum and Maximum Values

Number of Bits	Minimum Value	Maximum Value
16	−32768	32767
32	−2147483648	2147483647

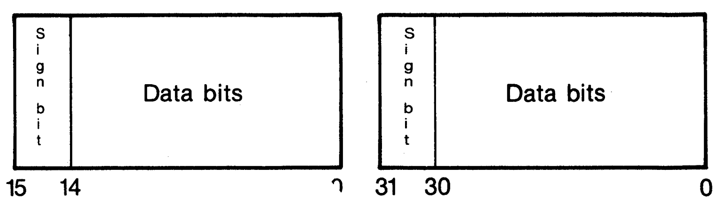


Figure 4-7. *Example of Turbo C using most significant bit of an integer value*

can use this bit to determine the number of bits Turbo C is using to store values of type int.

If you begin by assigning a value of type int the value 1, you can repeatedly shift the value to the left one location until the sign bit becomes set. When this occurs, you know the size of the variable (see Figure 4-8).

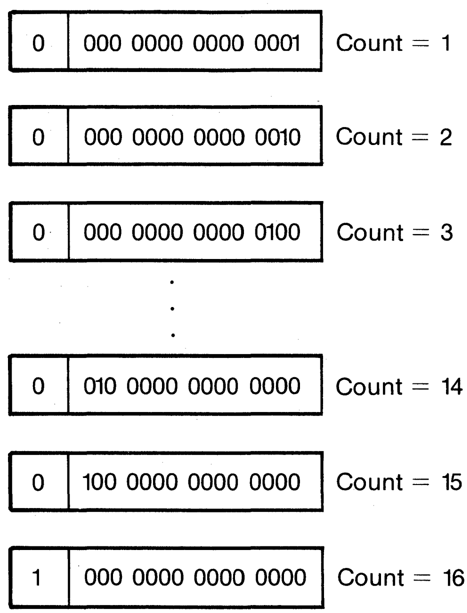
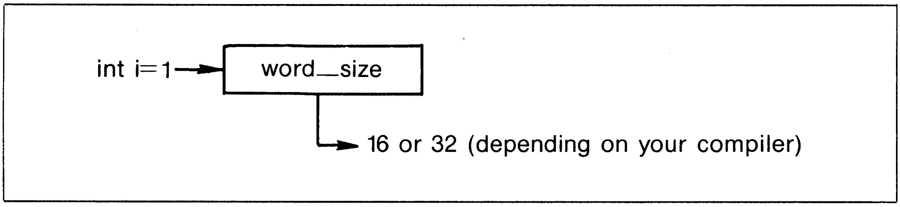


Figure 4-8. *Shifting a value left one location until sign bit is set*

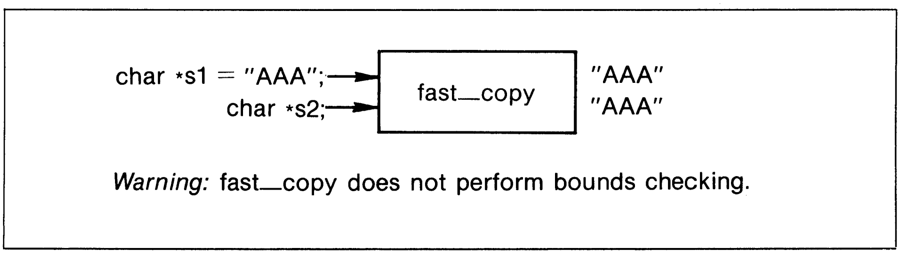
The following code implements word_size:




```
/*
 * word_count (value)
 *
 * Return the number of bits in a value of type int.
 *
 * value (in): value to shift left until negative.
 *
 * num_bits = word_count (1);
 *
 */

word_count (int a)
 {
  return ((a > 0) ? 1 + word_count (a << 1): 1);
 }
```

The following routine invokes a recursive implementation of the routine fast_copy. Given the string "ABC", the processing is that shown in Figure 4-9.

The following code implements fast_copy. Note that this routine does not perform bounds checking.



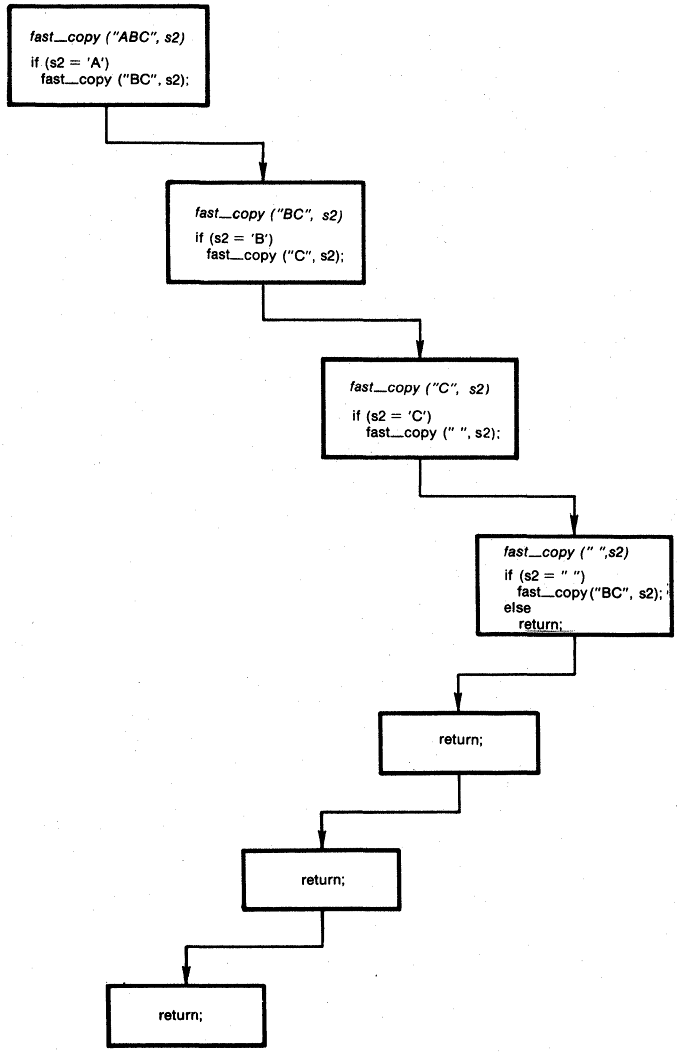


Figure 4-9. *Processing involved in fast_copy program*

```
/*
 * void fast_copy (source, target)
 *
 * Copy the contents of the source string to the target.
 *
 * s1 (in): source string containing characters to copy.
 * s2 (out): string receiving characters copied.
 *
 * fast_copy ("This is a test", stringvar);
 *
 * fast_copy does not perform bounds checking.
 *
 */

void fast_copy (s1, s2)
  char *s1, *s2;
 {
  if (*s2 = *s1)
    fast_copy (++s1, ++s2);
 }
```

The following program uses recursion to display the contents of a small text file in reverse order. Just as the string-manipulation routines presented in Chapter 2 searched a character string one letter at a time until the null character was found, the file_reverse routine searches for the end of a file. Given the following file,

```
AAAA
BBBB
CCCC
```

the processing becomes that shown in Figure 4-10.

The following program uses file_reverse to display the contents of a file specified by *argv*[1] in reverse order.

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  FILE *fp, *fopen();
```

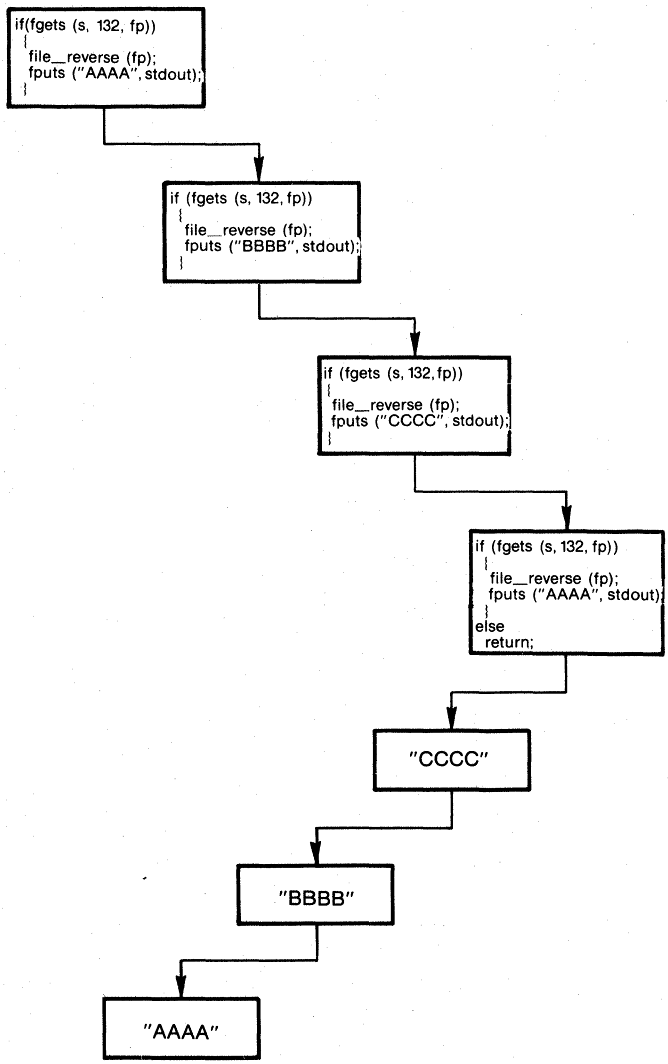


Figure 4-10. *Processing involved in file__reverse program*

```
  void file_reverse (FILE *fp);

  if (argc < 2)
    printf ("Invalid usage: FILEREV FILENAME.EXT\n");
  else if (! (fp = fopen (argv[1], "r")))
    printf ("Could not open %s\n", argv[1]);
  else
    {
     file_reverse (fp);
     fclose (fp);
    }
 }

/*
 * void file_reverse (file_pointer)
 *
 * Display the contents of the file specified last line to first line.
 *
 * file_pointer (in): pointer to the desired file.
 *
 * file_reverse (fp);
 *
 * file_reverse only works for small files.
 *
 */

void file_reverse (FILE *fp)
 {
  char string[132];

  if (fgets (string, 132, fp))
    {
      file_reverse (fp);
      fputs (string, stdout);
    }
  }
```

Similarly, the program file_pointer uses recursion to display the last ten lines of a file. For example, the command

```
A> LAST FILENAME.EXT
```

displays the last ten lines of the file FILENAME.EXT, as shown here:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  FILE *fp, *fopen();

  char *lines[10], *malloc();

  int index;
```

```
    int last (FILE *, char *[], int);

    if (argc < 2)
      printf ("Invalid usage: LAST FILENAME.EXT\n");
    else if (! (fp = fopen (argv[1], "r")))
      printf ("Could not open %s\n", argv[1]);
    else
      {
       /* allocate space for a circular buffer */
       for (index = 0; index < 10; index++)
        if (! (lines [index] = malloc (132)))
         {
           printf ("Unable to allocate necessary memory\n");
           exit (1);
         }

       last (fp, lines, 0);
       fclose (fp);
      }
   }

 /*
  * last (file_pointer, lines, index)
  *
  * Display the last 10 lines of the file specified.
  *
  * file_pointer (in): pointer to the desired file.
  * lines (in/out): buffer that 10 lines are stored in.
  * index (in): index to the current line.
  *
  * last (fp, lines, 0);
  *
  */

 last (FILE *fp, char *lines[], int index)
  {
   if (fgets (lines[index], 132, fp))
      last (fp, lines, (index + 1) % 10);

   else
     {
      int i;

      i = (index + 1) % 10;
      while (i != index)
       {
         fputs (lines[i], stdout);
         i = (i + 1) % 10;
       }
     }
  }
```

In Turbo C, even the main program is considered to be a function. You can invoke *main* in a recursive manner, as shown here:

```
main (int argc, char *argv[])
 {
  if (*++argv)
   {
    printf ("%s\n", *argv);
    main (argc, argv);
   }
 }
```

In this case, if the program has the following command-line parameters,

```
A> RECMAIN A B C
```

the program displays the first command-line parameter and then invokes itself to recursively display the second. This process continues until no parameters remain on the command line.

By simply changing the code to the following,

```
main (int argc, char *argv[])
 {
  if (*++argv)
   {
    main (argc, argv);
    printf ("%s\n", *argv);
   }
 }
```

the program now displays the command-line arguments in reverse order.

Considerations for Recursive Functions

In many cases, you can reduce the amount of code required to perform a specific task by using recursive functions.

Essentially, every routine presented thus far could be implemented recursively. The reasons why you do not do just that are speed and space.

Each time you invoke a Turbo C function, the program must place the return address and function parameters into an area of memory called the *stack*, which in turn produces overhead. In most cases, the overhead associated with functions is an acceptable tradeoff in order to achieve increased readability and modularity of code. This is not always the case with recursion, however. A recursive function may require many invocations in order to perform a specific task. With each invocation comes the overhead of placing the return address and variables onto the stack. This overhead can make recursive functions quite slow.

The second concern with recursion is stack space. With each invocation of a function, Turbo C places data onto the stack. In most cases, the stack can only store 64K of data. Thus, if you have a recursive function that requires many or large local variables, you can quickly use up your allotted stack space.

During the discussion of dynamic variables in later chapters, you will find recursion to be a powerful tool. For now, just concentrate on the flow of control for your recursive routines.

C H A P T E R

5

Pipe and I/O Redirection

By default, each time that you issue a DOS command, the operating system obtains its input from the keyboard and displays its output to the screen. Thus, the keyboard and screen make up the DOS default *standard input* source and *standard output* destination (see Figure 5-1). DOS defines the standard input source as *stdin* and standard output destination as *stdout.*

Issue the following command:

```
A> DIR
```

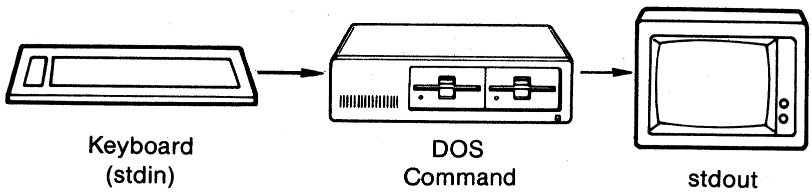


Figure 5-1. Standard input source and standard output destination

DOS displays the results of the command to the screen (stdout), as shown in Figure 5-2.

DOS also provides several I/O redirection operators that allow you to redefine stdin and stdout for a program. For example, issue the following command:

```
A> DIR > DIR.LST
```

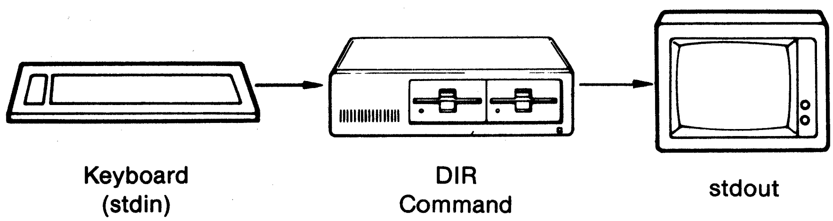


Figure 5-2. Displaying the results of a command to stdout

In this case, rather than displaying the output of the DIR command to the screen, DOS has redirected stdout to point to the file DIR.LST, as shown in Figure 5-3.

Turn on your system printer and issue the following command:

```
A> DIR > PRN:
```

This time DOS redirects the output of the DIR command from the screen to the printer, as shown in Figure 5-4.

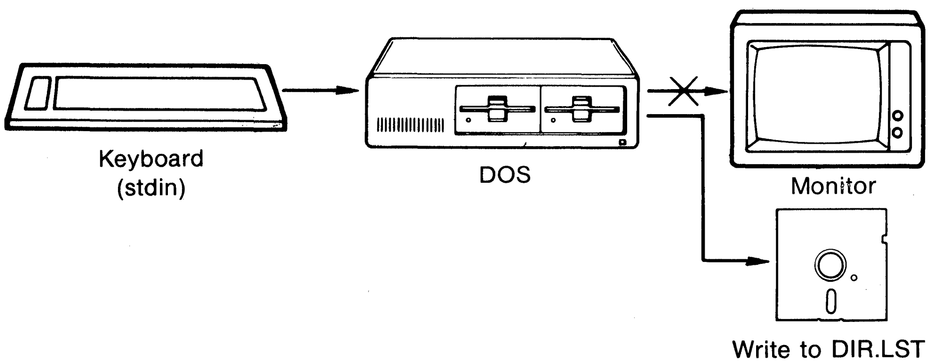


Figure 5-3. *Output of DIR to DIR.LST*

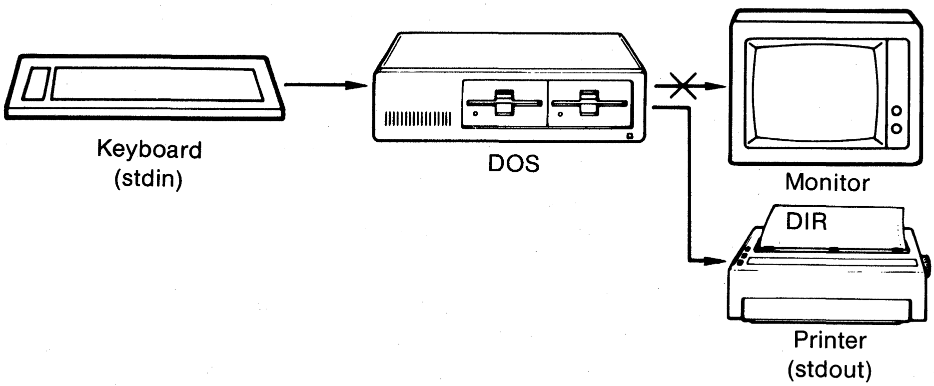


Figure 5-4. I/O redirection from screen to printer

Use an existing text file on your disk and issue the following command:

```
A> MORE < FILENAME.EXT
```

In this case, DOS leaves stdout unchanged and displays the output of the command on the screen. DOS now modifies stdin for the MORE command and redirects stdin from the keyboard to the file, as shown in Figure 5-5.

The DOS pipe operator allows you to direct the output of one command to become the input of a second command, as shown here:

```
A> DIR | SORT
```

In this case, DOS redirects stdout for the DIR command and stdin for the SORT command, as shown in Figure 5-6.

You should note that you can use many of these operators on one command line, as shown here:

```
A> SORT < FILENAME.EXT | MORE
```

Programs that support the DOS input/output (I/O) redirection operators are easy to implement with Turbo C.

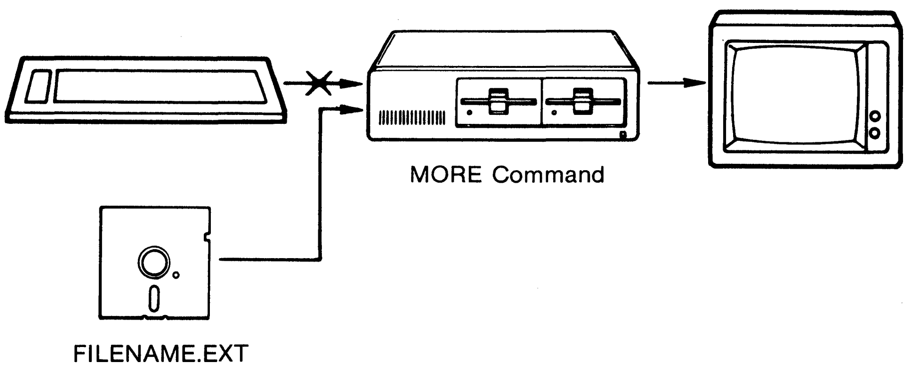


Figure 5-5. Redirection of stdin for MORE command

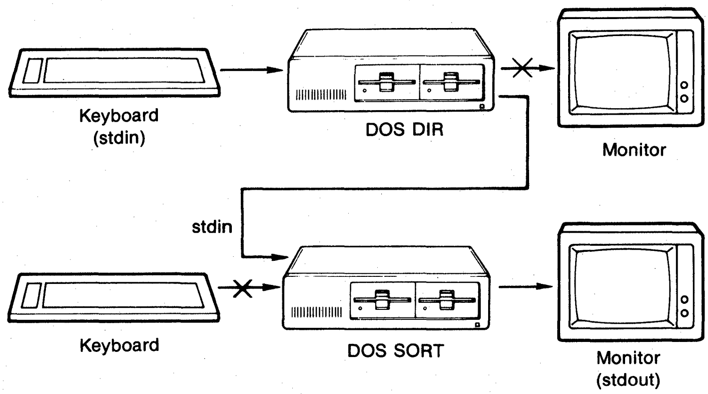


Figure 5-6. *Redirection of stdout for DIR command to stdin for SORT command*

Getting Started with I/O Redirection

The first Turbo C program example supports I/O redirection. This example program counts the number of lines of redirected input and displays the final count following the last line read. For example, if your disk contains the following files,

```
 Volume in drive A is TURBO_C
 Directory of  A:\

TEST     C       155   2-05-88  10:46a
MAIN     C      1050   6-03-87   1:00a
SHOW     C       351   1-26-88  12:19a
TEE      C       498   2-06-88   6:01p
FACT     C       402   2-05-88   5:27p
LAST     C      1241   2-05-88   6:59p
TAB      C       543   2-06-88   1:44p
F2       C       927   2-06-88   5:20p
MORE     C       362   2-06-88   5:56p
         9 File(s)    351232 bytes free
```

the command

```
A> DIR | COUNT
```

will display

```
Line count = 14
```

You can also use COUNT to display the number of lines in a file, as shown here:

```
A> COUNT < FILENAME.EXT
```

The following program implements COUNT:

```
#include <stdio.h>

main ()
 {
  int count = 0;

  char line[132];

  while (fgets (line, 132, stdin))
     count++;

  printf ("Line count = %d\n", count);
 }
```

The processing required for COUNT is straightforward. The program simply reads data from stdin until the end of the file is found.

Next, COUNT displays the count of the number of lines read.

Similarly, the program LINENUM places a line number before each of the lines it reads from stdin. The command

```
A> DIR | LINENUM
```

results in

```
1
2  Volume in drive A is TURBO_C
3  Directory of  A:\
4
5 TEST      C       155   2-05-88  10:46a
6 MAIN      C      1050   6-03-87   1:00a
7 SHOW      C       351   1-26-88  12:19a
8 TEE       C       498   2-06-88   6:01p
9 FACT      C       402   2-05-88   5:27p
10 LAST     C      1241   2-05-88   6:59p
11 TAB      C       543   2-06-88   1:44p
12 F2       C       927   2-06-88   5:20p
13 MORE     C       362   2-06-88   5:56p
14         9 File(s)     351232 bytes free
```

This following code implements LINENUM:

```
#include <stdio.h>

main ()
 {
  int line_number = 0;

  char line[132];

  while (fgets (line, 132, stdin))
     printf ("%d %s", ++line_number, line);
 }
```

Note that printf writes all of the data to stdout. Thus, you can redirect output from LINENUM, as shown here:

```
A> DIR | LINENUM | MORE
```

The program STATS combines features from the previous programs to display the number of lines, pages, words, and characters contained in a file (or redirected input). For example, given the following file,

```
This is a test file.
Don't forget about the
carriage return and line
feed at the end of each
line.
```

the command

```
A> STATS < FILENAME.EXT
```

displays

```
Pages = 0
Lines = 5
Words = 20
Characters = 100
```

The following program complements STAT.C:

```
#include <stdio.h>

#define lines_per_page 23

main ()
 {
  int lines = 0, words = 0, characters = 0;
  int in_blanks, i;

  char str[132];

  while (fgets (str, 132, stdin))
    {
     i = 0;                          /* index into string */
     in_blanks = 1;                  /* assume line starts with blanks */

     while (str[i])
       {
        characters++;                /* another character */

        if (str[i] == ' ')
          {
            if (! in_blanks)
              {
               words++;              /* blank separates words */
               in_blanks = 1;        /* two blanks in a row is not a word
              }
          }
        else if (str[i] != '\n')
            in_blanks = 0;

        else if (! in_blanks)       /* word ended at end of line */
            words++;

        i++;
       }

      lines++;                       /* get the next line */
    }

  printf ("Pages = %d\nLines = %d\nWords = %d\nCharacters = %d\n",
           lines / lines_per_page, lines, words, characters);
 }
```

The program FIRST displays the first *n* lines of the redirected input, as shown here:

```
A> FIRST 15< FILENAME.EXT
```

In this case, FIRST displays the first 15 lines. If you omit the desired number of lines, FIRST displays 10 by default.

```
A> DIR | FIRST
```

The following code implements FIRST:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  int stop_line = 10;   /* number of lines to display */

 int count = 0;        /* current line number */

 char line[132];

 int ascii_to_int (char *, int *);

 if (argc > 1)         /* see if user specified a valid number */
   if (ascii_to_int (argv[1], &stop_line) == -1)
     stop_line = 10;

 while (fgets (line, 132, stdin) && (++count <= stop_line))
    fputs (line, stdout);
}
```

The next program uses the routine LAST (presented in Chapter 4) to display the last ten lines of redirected input, as shown here:

```
A> DIR | LAST
```

The following code implements LAST:

```
#include <stdio.h>

main ()
 {
  char *lines[10], *malloc();

  int index;

  /* allocate space for a circular buffer */
```

```
 for (index = 0; index < 10; index++)
   if (! (lines [index] = malloc (132)))
      {
        printf ("Unable to allocate necessary memory\n");
        exit (1);
      }

 last (stdin, lines, 0);
}
```

Once you develop a library of powerful routines, your program development becomes much more direct.

In a manner similar to FIRST, the program TAB combines command-line processing with I/O redirection. In this case, you specify the number of spaces the output is to be shifted to the right, as shown here:

```
A> DIR | TAB 5
```

If you omit the desired number of spaces,

```
A> TAB < FILENAME.EXT
```

TAB will use the value 7 by default. Once again, the program is built by using routines presented earlier in the book.

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  int spaces = 7;        /* number of spaces to insert */

  char line[132];

  int ascii_to_int (char *, int *);
  int pad_string (char *, int, int);

  if (argc > 1)          /* see if user specified a valid number */
   {
```

```
   if (ascii_to_int (argv[1], &spaces) == -1)
     spaces = 7;
  }
 while (fgets (line, 132, stdin))
  {
    if (pad_string (line, spaces, sizeof (line)) == 1)
      {
        printf ("%c Line exceeds %d characters\n", 7, sizeof (line));
        break ;
      }

    fputs (line, stdout);
  }
}
```

The program FINDWORD displays each line of the redirected input that contains the word specified by the user:

```
A> TYPE STATES.LST | FINDWORD ARIZONA
```

In this case, the processing again becomes straightforward.

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  char line[132];

  int index (char *, char *);

  if (argc < 2)
    printf ("invalid usage: FINDWORD WORD\n");
  else
    {
      while (fgets (line, sizeof(line), stdin))
        if (index (argv[1], line) != -1)
          fputs (line, stdout);
    }
 }
```

To increase the program's capabilities, you can support the /C and /V qualifiers as follows:

/C	Display a count of the number of occurrences of the specified word
/V	Display lines that do not contain the specified word

The final program becomes as follows:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  char line[132];
  int count_only = 0, contain_word = 1, i, count = 0;

  int index (char *, char *);

  if (argc < 2)
    printf ("invalid usage: FINDWORD WORD [/C] [/V]\n");
  else
    {
      for (i = 1; i < argc; i++)
        if (index ("/C", argv[i]) != -1)
          {
           count_only = 1;
           break;
          }

      for (i = 1; i < argc; i++)
        if (index ("/V", argv[i]) != -1)
          {
            contain_word = 0;
            break;
          }

      while (fgets (line, 132, stdin))
        if (index (argv[1], line) != -1)
         {
          if (count_only)
            count++;
          else if (contain_word)
            fputs (line, stdout);
         }
        else if (! contain_word)
            fputs (line, stdout);
    }
  if (count_only)
    printf ("%s occurs %d times\n", argv[1], count);
 }
```

Just as the program FINDWORD displayed each occurrence of a word in redirected input, the program REPLACE replaces each occurrence of a word with the second word specified.

```
A> REPLACE begin BEGIN  TEST.PAS NEW.PAS
```

The following program implements REPLACE:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  char line[132];

  int location, len;

  FILE *fopen (), *infile, *outfile;

  int remove_substring (char *, char *);
  int insert_string (char *, char *, int, int);
  int next_str_occurrence (char *, char *, int);

  if (argc < 3)
    printf ("invalid usage: REPLACE TARGET NEW_WORD OLDFILE NEWFILE\n");

  else if (argc == 3)
   {
    infile = stdin;
    outfile = stdout;
   }

  else if (argc == 4)
   {
    if (! (infile = fopen (argv[3], "r")))
      {
       printf ("REPLACE error opening %s\n", argv[3]);
       exit (1);
      }
    outfile = stdout;
   }

  else if (argc == 5)
   {
    if (! (infile = fopen (argv[3], "r")))
      {
       printf ("REPLACE error opening %s\n", argv[3]);
       exit (1);
      }

    if (! (outfile = fopen (argv[4], "w")))
      {
       printf ("REPLACE error opening %s\n", argv[4]);
       exit (1);
      }
   }

   len = string_length (argv[2]);

   while (fgets (line, 132, infile))
     {

       if ((location = index (argv[1], line)) != -1)
         do
           {
```

```
              remove_substring (argv[1], &line[location]);
              insert_string (argv[2], line, location, sizeof(line));
            }
          while ((location = next_str_occurrence (argv[1], line, locatio

         fputs (line, outfile);
     }
}
```

The program MORE.C implements the DOS MORE command. Each time MORE displays a screenful of information, it pauses and waits for you to press the ENTER key to continue.

```
 Volume in drive A is TURBO_C
  Directory of  C:\TURBOC

.            <DIR>     11-28-87   8:05p
..           <DIR>     11-28-87   8:05p
ALLOC    H         896   6-03-87   1:00a
ASSERT   H         275   6-03-87   1:00a
BIOS     H         527   6-03-87   1:00a
CONIO    H         517   6-03-87   1:00a
CTYPE    H        1345   6-03-87   1:00a
DIR      H        1222   6-03-87   1:00a
DOS      H        7316   6-03-87   1:00a
ERRNO    H        2648   6-03-87   1:00a
FCNTL    H         991   6-03-87   1:00a
FLOAT    H        2094   6-03-87   1:00a
IO       H        2407   6-03-87   1:00a
LIMITS   H         757   6-03-87   1:00a
MATH     H        2984   6-03-87   1:00a
MEM      H         906   6-03-87   1:00a
PROCESS  H        1782   6-03-87   1:00a
SETJMP   H         542   6-03-87   1:00a
SHARE    H         434   6-03-87   1:00a
--MORE--
```

This code implements MORE:

```
#include <stdio.h>

#define lines_per_page 24

main ()
 {
  int line_number = 0;
```

```
  char line[132];

  while (fgets(line, 132, stdin))
    if (++line_number % lines_per_page)
      fputs (line, stdout);
    else
      {
       fflush (stdout);
       fputs ("--MORE--\n", stdout);
       fflush (stdout);
       bioskey (0);
      }
}
```

The program TEE allows you to file intermediate results while you continue I/O redirection, as shown here:

```
A> TYPE FILENAME.EXT | SORT | TEE SORTFILE.EXT | MORE
```

This command is illustrated in Figure 5-7.

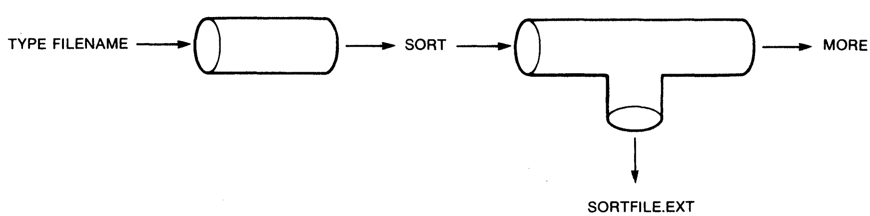


Figure 5-7. *Processing involved with program TEE*

By using TEE, you can write results to a file and also to stdout, as shown here:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  FILE *fopen(), *fp;

  char line[132];

  if (argc < 2)
   fputs ("invalid usage: TEE FILENAME\n", stdout);
  else
    {
     if (! (fp = fopen(argv[1], "w")))
       fputs ("TEE: unable to open output file\n", stdout);
     else
      {
       while (fgets (line, 132, stdin))
         {
           fputs (line, stdout);
           fputs (line, fp);
         }
       fclose (fp);
      }
    }
 }
```

Using Standard Error (stderr)

Periodically your programs will experience an error that results in an error message. If you write the following error message to stdout,

```
printf ("invalid usage: TEE FILENAME");
```

the error message will also be redirected. For this reason, you may never see the error message. To make sure you see your messages, DOS defines an output source called stderr that is guaranteed to display error messages to the screen, regardless of redirection. Your programs should write all error messages to stderr, as shown here:

```
fputs ("invalid usage: TEE FILENAME", stderr);
```

The following program modifies TEE.C to do just that:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  FILE *fopen(), *fp;

  char line[132];

  if (argc < 2)
   fputs ("invalid usage: TEE FILENAME\n", stderr);
  else
    {
     if (! (fp = fopen(argv[1], "w")))
       fputs ("TEE: unable to open output file\n", stderr);
     else
      {
       while (fgets (line, 132, stdin))
         {
           fputs (line, stdout);
           fputs (line, fp);
         }
       fclose (fp);
      }
    }
 }
```

I/O redirection is a powerful tool. Later chapters discuss how to modify many of the programs presented in this chapter so that they support I/O redirection and command-line processing. For now, experiment with the programs presented in this chapter to increase your understanding of I/O redirection.

CHAPTER

6

DOS Interface

You are probably familiar with DOS, the operating system for the IBM PC and PC compatibles. What you may not know is that a significant portion of DOS is written in C. As is the case with all operating systems, the DOS developers were faced with a monumental programming task when they wrote DOS. To simplify their task, the developers broke it into many small, manageable functions. These functions are responsible for operating system tasks such as the following:

- File manipulation (open, read, write, close operations)

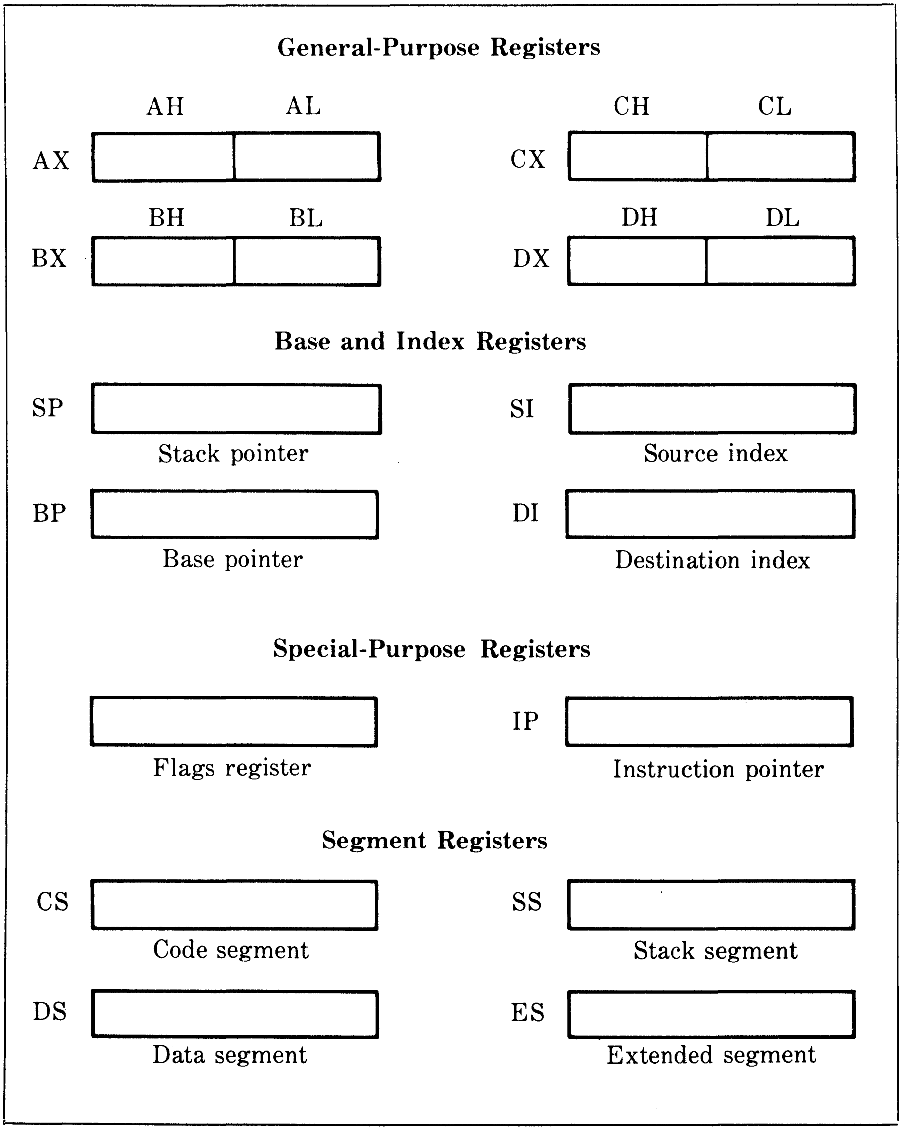
- Keyboard input
- Program startup and termination
- File-creation, file-deletion, and rename operations
- Memory management (allocate, free, modify)
- Disk-drive manipulation
- Directory manipulation

Because DOS must use these services on a continual basis in order to operate, each function must remain immediately available for use. Thus, you can make use of these services from within your programs. DOS uses the 8088 registers as its interface to the DOS system services.

Many of these routines appear in the Turbo C run-time library. Their names and parameters may differ from the routines presented here though the functionality is the same. Use whichever implementation best suits your needs, but still study these routines; they can teach you a great deal about DOS.

8088 Registers

The IBM PC and PC compatibles are based on a processor chip called the 8088. Within this chip is a set of storage locations known as *registers*. Since registers are contained within the control processing unit (CPU) itself, the 8088 can manipulate the values contained in the registers quite rapidly. The 8088 has 14 registers, each capable of storing 16 bits of data, as the following shows:


General-Purpose Registers
AH
AL
AX
CH
CL
CX
BH
BL
BX
DH
DL
DX
Base and Index Registers
SP
Stack pointer
SI
Source index
BP
Base pointer
DI
Destination index
Special-Purpose Registers
Flags register
IP
Instruction pointer
Segment Registers
CS
Code segment
SS
Stack segment
DS
Data segment
ES
Extended segment

Your programs communicate to the DOS system services through these registers. For example, assume that you want to determine the DOS version number that you are using. Place the following value into AH register and invoke the DOS interrupt (INT 21H):

AH 30H (Get DOS version number)

On completion, this service places the major and minor versions of the operating system into register AX, as shown here:

AH	Contains the minor version number
AL	Contains the major version number

The following language code fragment invokes the DOS Get Version Number system service:

```
MOV  AH, 30H
INT 21H
```

INT 21H serves as your means of executing a DOS system service.

INT 21H

An *interrupt* is a signal to the CPU from a program or hardware device instructing the CPU to suspend temporarily the function that it is performing and instead execute a different task. For example, each time you simultaneously press the SHIFT and PRINT SCREEN keys, DOS temporarily suspends what it is doing in order to print the current screen contents. DOS uses INT 21H as its interface to the DOS system services. Each time DOS encounters an INT 21H, it examines the contents of each of the 8088 registers to determine the specific DOS service to perform, along with the required parameters for the service. In most cases, DOS obtains the service number from register AH.

In the previous example, DOS found the value 30H in AH, which directed it to perform the Get DOS Version service. In this case, the DOS service 02H directs DOS to display the character contained in register DL. To invoke this routine, place the corresponding values into the 8088 registers and invoke INT 21H, as shown here:

```
MOV AH, 2        ; display character service
MOV DL, 65       ; character to display

INT 21H          ; invoke DOS service
```

A goal in developing applications is to write as much of the code as possible in a high-level language such as C (as opposed to assembly language). You must have a means of executing DOS system services from such languages. In the case of Turbo C, a routine called intdos provides your interface. To use this routine, you must include the file dos.h, as shown here:

```
#include <dos.h>
```

Remember, the DOS system services use the 8088 registers as their interface. The file dos.h contains a structure definition that allows your program to emulate the 8088 registers, as shown here:

```
struct WORDREGS
        {
        unsigned int    ax, bx, cx, dx, si, di, cflag, flags;
        };

struct BYTEREGS
        {
        unsigned char   al, ah, bl, bh, cl, ch, dl, dh;
        };

union   REGS    {
        struct  WORDREGS x;
        struct  BYTEREGS h;
        };

struct  SREGS   {
        unsigned int    es;
        unsigned int    cs;
        unsigned int    ss;
```

```
        unsigned int    ds;
        };

struct  REGPACK
        {
        unsigned        r_ax, r_bx, r_cx, r_dx;
        unsigned        r_bp, r_si, r_di, r_ds, r_es, r_flags;
        };
```

Within your C program you simply assign appropriate values to each register (member of the structure). When you later invoke intdos, that routine maps the values contained in your structure into the appropriate registers, as shown in Figure 6-1.

When the DOS system service completes, intdos again maps the register values back to your structure, as shown in Figure 6-2. The following C program displays the current DOS version:

```
#include <dos.h>

main ()
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x30;
  intdos (&inregs, &outregs);

  printf ("DOS Version %d.%d\n", outregs.h.al, outregs.h.ah);
 }
```

DOS System Services

The DOS system services are quite powerful. In fact, these services make up the toolkit that the DOS developers used to build DOS. By using these routines in your programs, you can quickly develop routines of professional quality. This section discusses the commonly used DOS system services and shows their Turbo C implementations. Most of the services are quite straightforward to use and many will greatly increase the capabilities of your application.

Note that many of these routines assume you are using the small memory model of the Turbo C compiler. These routines do not pass segment addresses of strings to the intdos routine; instead, they simply use the value of the current data segment. Since the small memory model is assumed, the routines are successful. If you are

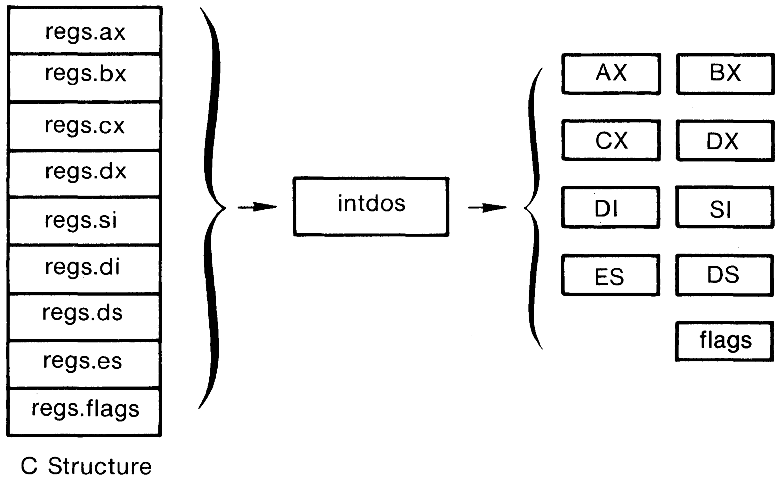


Figure 6-1. *Mapping of structure values by intdos*

using a different memory model, refer to the Osborne/McGraw-Hill text *DOS Power User's Guide*, by Kris Jamsa (Berkeley, 1988), for specifics on each system service.

Many of the routines presented in this section are also available as run-time library routines under Turbo C. However, because of the importance of the DOS system services (along with the tremendous capabilities that these services provide), the routines are presented for your examination. Experiment with the DOS system services and you should find them to be extremely useful.

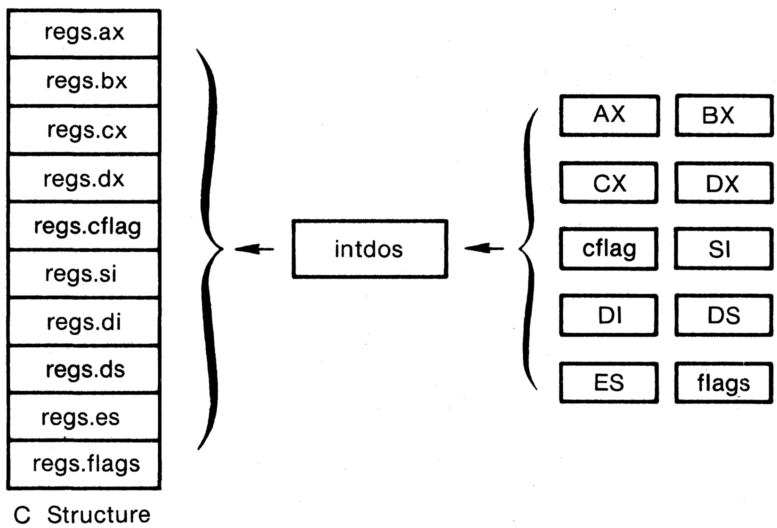
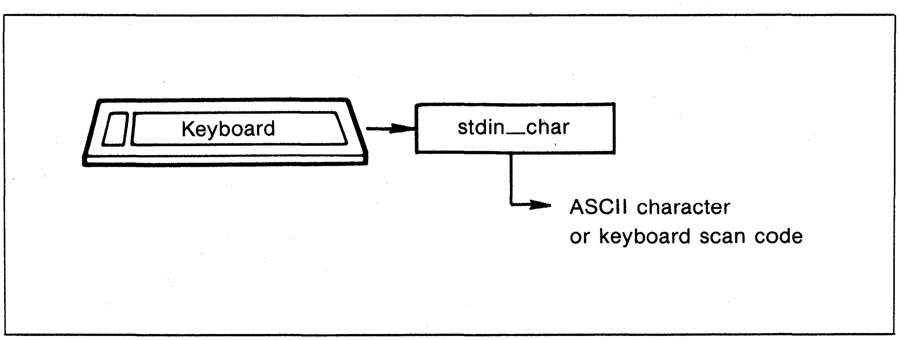
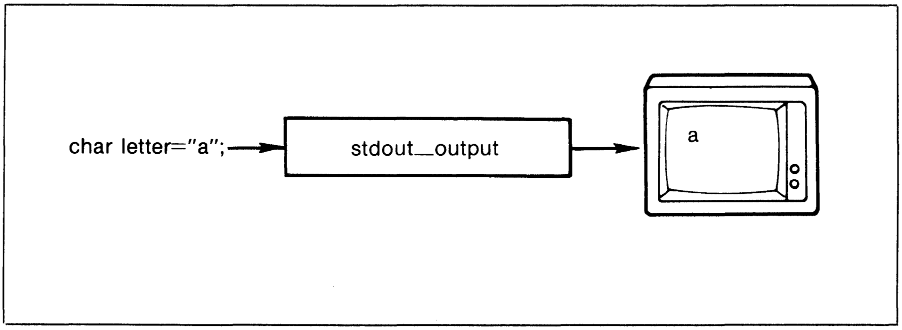


Figure 6-2. *Mapping of register values by indtdos*

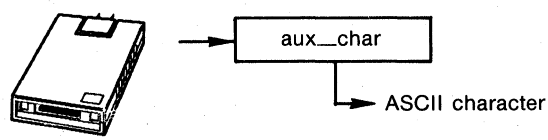


```
#include <dos.h>

/*
 * stdin_char ()
 *
 * Get a character from the standard input device.
 *
 * character = stdin_char ();
 *
 * If the uses presses a special function key, stdin_char
 * returns the null value on the first invocation.  You must
 * again invoke stdin_char to determine the scan code of the
 * special key pressed. This routine echos the character entered
 * by the user to the screen.
 *
 */

int stdin_char ()
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x01;
  intdos (&inregs, &outregs);
  return (outregs.h.al);
 }
```



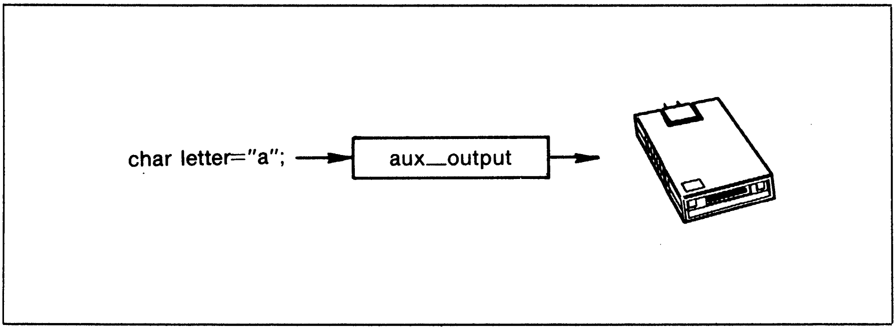

```
#include <dos.h>

/*
 * void stdout_output (character)
 *
 * Write the character specified to the standard output device.
 *
 * character (in): character to be written.
 *
 * stdout_output (65);
 *
 */
```

```
void stdout_output (char character)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x02;
  inregs.h.dl = character;

  intdos (&inregs, &outregs);
 }
```



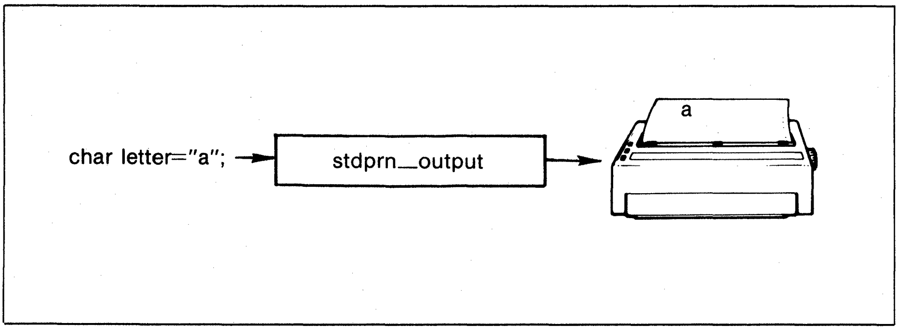

```
#include <dos.h>

/*
 * aux_char ()
 *
 * Get a character from the standard auxiliary device.
 *
 * character = aux_char ();
 *
 * If a character is not present, aux_char waits until one
 * becomes available.
 *
 */

int aux_char ()
```

```
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x03;
  intdos (&inregs, &outregs);
  return (outregs.h.al);
 }
```

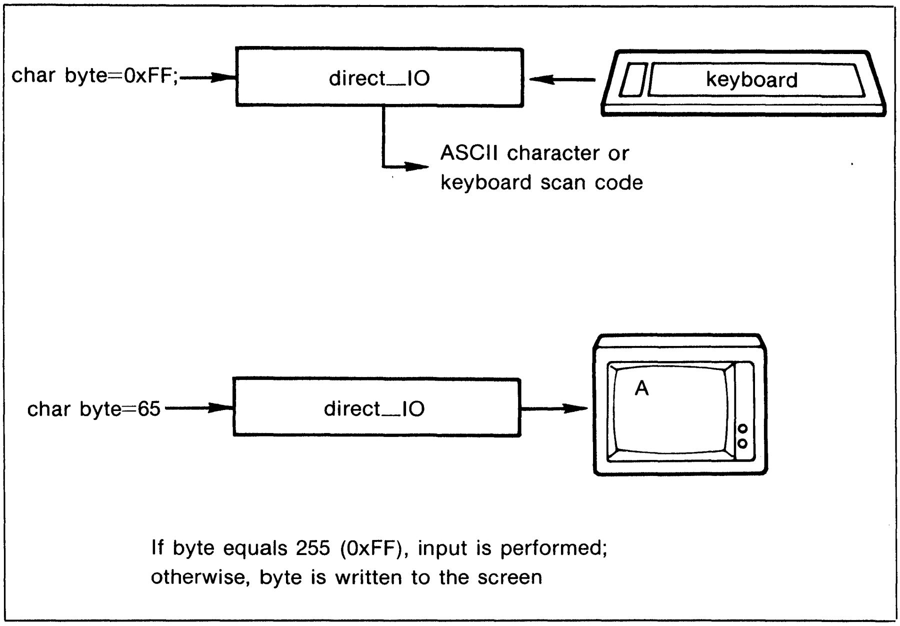



```
#include <dos.h>

/* void aux_output (character);
 *
 * Write a character to the standard auxiliary device.
 *
 * character (in): character to be written.
 *
 * aux_output (65);
 *
 * By default, DOS uses 2400 baud, no parity, 1 stop bit, and
 * 8 data bits.
 *
 */

void aux_output (char character)
 {
```

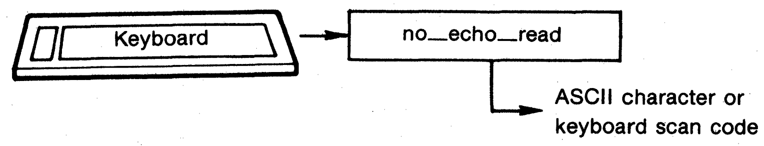
```
 union REGS inregs, outregs;

 inregs.h.ah = 0x04;
 inregs.h.dl = character;

 intdos (&inregs, &outregs);
}
```



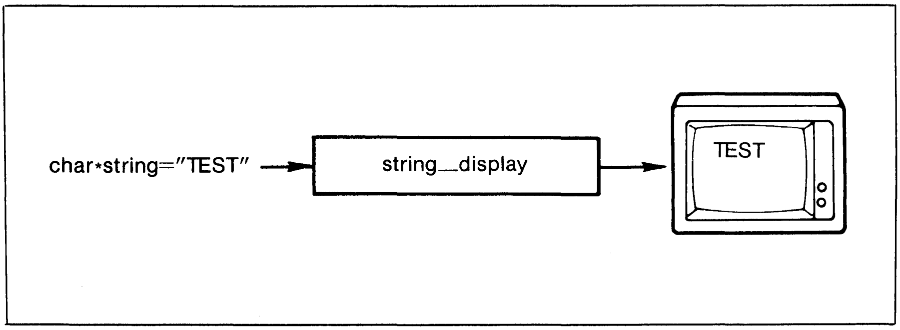

```
#include <dos.h>

/*
 * void stdprn_output (character);
 *
 * Write a character to the standard printer device.
 *
 * character (in): character to be printed.
 *
 * stdprn_output (65);
 *
 */

void stdprn_output (char character)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x05;
  inregs.h.dl = character;

  intdos (&inregs, &outregs);
 }
```



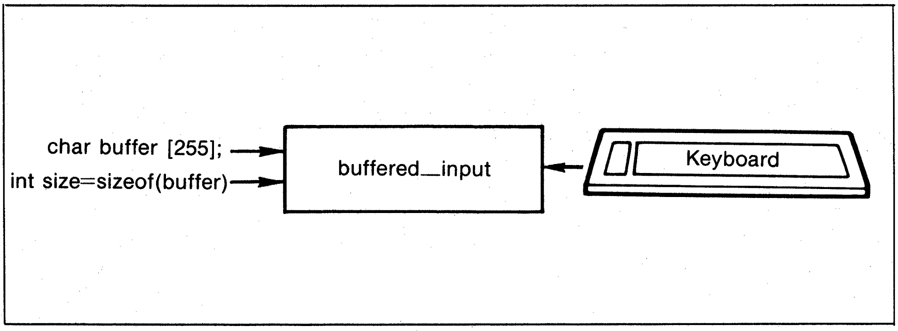

```
#include <dos.h>

/*
 * direct_IO (byte)
 *
 * Read a character from stdin or write a character to stdout
 * depending upon the value in the variable byte.
 *
 * byte (in): if 0xFF, read a character from stdin.  If byte
 *            contains any other value, write it to stdout.
 *
 * direct_IO (65);
 * character = direct_IO (0xFF);
 *
 */

int direct_IO (char byte)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x06;
  inregs.h.dl = byte;
  intdos (&inregs, &outregs);
  return (outregs.h.al);
 }
```



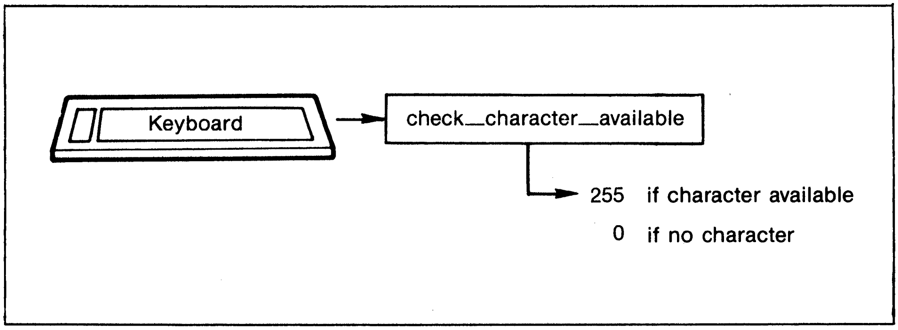

```
#include <dos.h>

/*
 * no_echo_read ()
 *
 * Read a character from stdin without echoing the character
 * back to the screen display.
 *
 * character = no_echo_read ();
 *
 * If a character is not present in the keyboard buffer, this
 * routine waits for one to become available.
 *
 * If the user presses a special function key, stdin_char
 * returns the null value on the first invocation.  You must
 * again invoke stdin_char to determine the scan code of the
 * special key pressed.
 *
 */

int no_echo_read ()
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x07;
  intdos (&inregs, &outregs);
  return (outregs.h.al);
 }
```



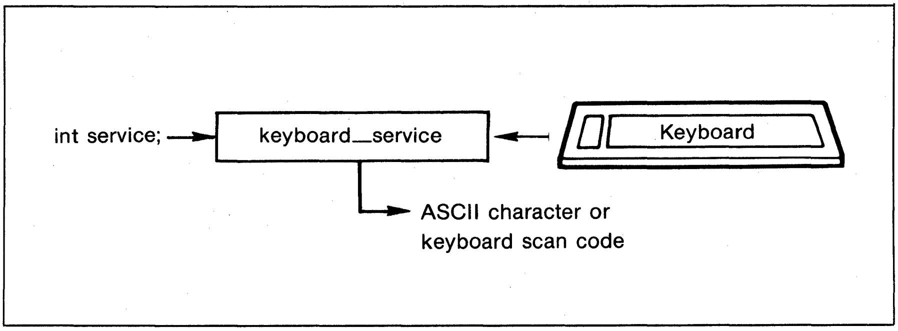

```
#include <dos.h>

/*
 * void string_display (string)
 *
 * Display the character string specified to the standard
 * output device.
 *
 * string (out): character string to be displayed.
 *
 * string_display ("Turbo C Programmer's Library");
 *
 */

void string_display (char string[])
 {
  union REGS inregs, outregs;

  int i;

  for (i = 0; string[i]; i++)
    ;

  string[i] = '$';        /* $ indicates last character to display */

  inregs.h.ah = 9;
  inregs.x.dx = string;

  intdos (&inregs, &outregs);

  string[i] = '\0';
 }
```



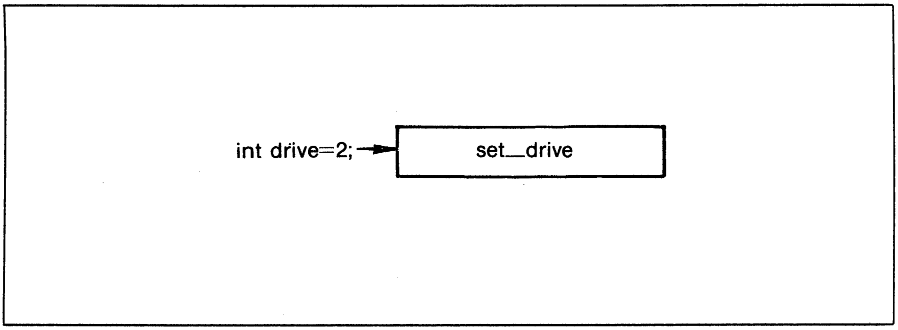

```
#include <dos.h>

/*
 * void buffered_input (buffer, size)
 *
 * Read characters from the standard input device into a user
 * defined buffer.
 *
 * buffer (in/out): buffer to store characters input.
 * size (in): maximum number of characters that the buffer can store.
 *
 * buffered_input (array, sizeof(array));
 *
 * The buffer must be defined as follows:
 *
 *     buffer[0] contains the maximum size of the buffer
 *     buffer[1] contains the number of characters read
 *     buffer[2] contains the first character read
 *
 */

void buffered_input (char buffer[], int size)
 {
  union REGS inregs, outregs;

  buffer[0] = size;            /* maximum buffer size */
  inregs.h.ah = 0xA;
  inregs.x.dx = buffer;        /* offset of buffer */

  intdos (&inregs, &outregs);
 }
```



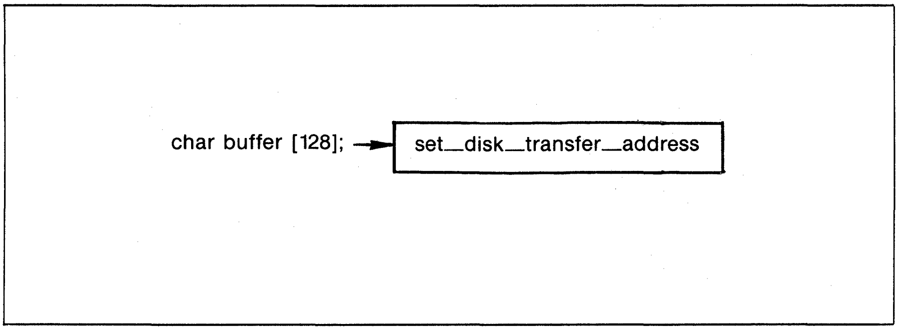

```
#include <dos.h>

/*
 * check_character_available ()
 *
 * Return the value 255 if a character is currently available in
 * the standard input device, otherwise return the value 0.
 *
 * status = check_character_available ();
 *
 */

int check_character_available ()
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x0B;
  intdos (&inregs, &outregs);
  return (outregs.h.al);
 }
```



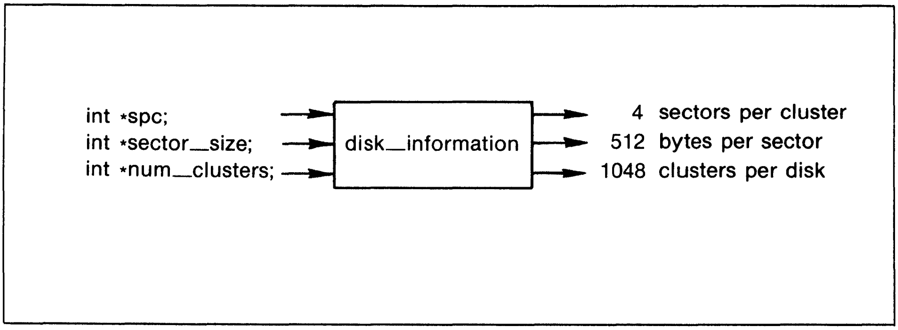

```
#include <dos.h>

/*
 * keyboard_service (service)
 *
 * Clear the keyboard buffer and perform the keyboard service
 * specified.
 *
 * service (in): DOS keyboard service to perform.
 *
 * character = keyboard_service (7);
 *
 * By invoking keyboard services in this fashion, you can insure
 * that the type ahead buffer is empty prior to your read operations.
 *
 */

int keyboard_service (int service)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x0C;
  inregs.h.al = service;
  intdos (&inregs, &outregs);
  return (outregs.h.al);
 }
```



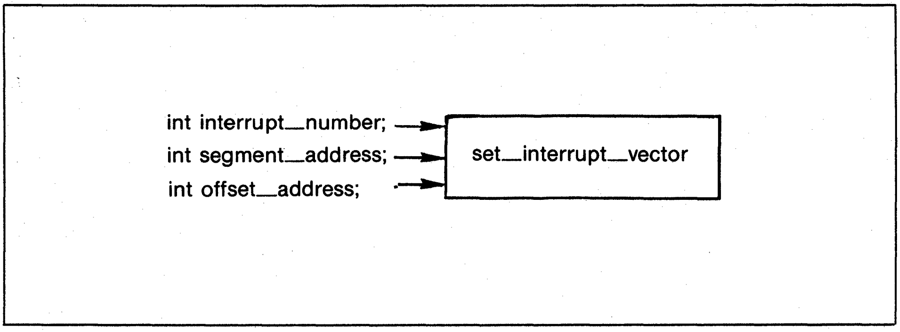

```
#include <dos.h>

/*
 * void set_drive (drive)
 *
 * Set the disk drive to the drive number specified.
 *
 * drive (in): Disk drive desired.
 *
 * set_drive (2);
 *
 * Drive numbers are defined as:
 *      A = 0, B = 1, C = 2
 *
 */

void set_drive (int drive)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x0E;
  inregs.h.dl = drive;
  intdos (&inregs, &outregs);
 }
```



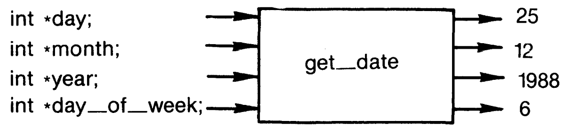

```
#include <dos.h>

/*
 * void set_disk_transfer_address (buffer)
 *
 * Define a new buffer for the DOS disk transfer area.
 *
 * buffer (in): buffer to be used as the DTA.
 *
 * set_disk_transfer_address (char_array);
 *
 * By default, the DTA points to offset 80H of the PSP.  In
 * later chapters we will use this region to perform command
 * line operations.  By modifying the DTA we can prevent DOS
 * from overwriting the command line.
 *
 */

void set_disk_transfer_address (char buffer[])
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x1A;
  inregs.x.dx = buffer;   /* minimum 128 bytes */
  intdos (&inregs, &outregs);
 }
```



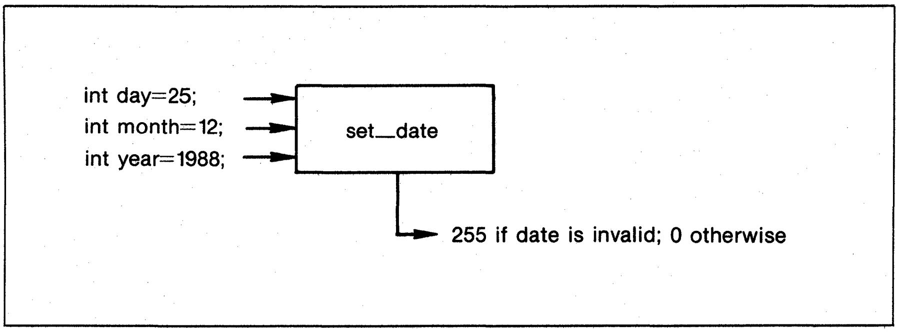

```
#include <dos.h>

/*
 * void disk_information (spc, sector_size, num_clusters)
 *
 * Return the number of sectors per cluster, the sector size,
 * and the number of clusters for the current disk drive.
 *
 * spc (out): sectors per cluster.
 * sector_size (out): bytes per sector.
 * num_cluster (out): clusters per disk.
 *
 * disk_information (&spc, &sector_size, &num_clusters);
 *
 */

void disk_information (int *spc, int *sector_size, int *num_clusters)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x1B;
  intdos (&inregs, &outregs);
  *spc = outregs.h.al;          /* sectors per cluster */
  *sector_size = outregs.x.cx;
  *num_clusters = outregs.x.dx;
 }
```



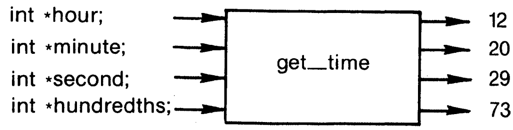

```
#include <dos.h>

/*
 * void set_interrupt_vector (interrupt_number, segment, offset)
 *
 * Specify a new interrupt handler routine for a specific interrupt.
 *
 * interrupt_number (in): Interrupt number to modify.
 * segment (in): Segment address of new routine.
 * offset (in): Offset address of new routine.
 *
 * set_interrupt_vector (5, segment_address, offset_address);
 *
 */

void set_interrupt_vector (int interrupt_number,
                           int segment, int offset)
 {
  union REGS inregs, outregs;

  struct SREGS segregs;

  inregs.h.ah = 0x25;
  inregs.h.al = interrupt_number;
  inregs.x.ds = segment;
  inregs.x.dx = offset;

  intdosx (&inregs, &outregs, &segregs);
 }
```



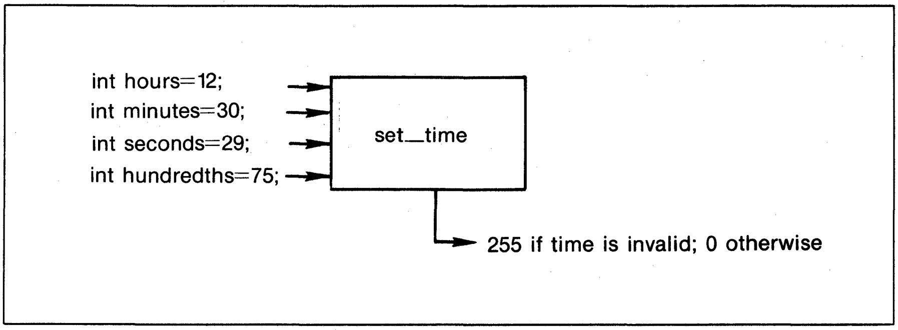

```
#include <dos.h>

/*
 * void get_date (day, month, year, day_of_week)
 *
 * Return the current system date.
 *
 * day (out): day of of the month (1-31)
 * month (out): month of the year (1-12)
 * year (out): current year (19xx)
 * day_of_week (out): current day of the week (0=Sunday, 6=Saturday)
 *
 * get_date (&day, &month, &year, &day_of_week);
 *
 */

void get_date (int *day, int *month, int *year,
               int *day_of_week)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x2A;
  intdos (&inregs, &outregs);

  *day = outregs.h.dl;
  *day_of_week = outregs.h.al;
  *month = outregs.h.dh;
  *year = outregs.x.cx;
 }
```




```
#include <dos.h>

/*
 * set_date (day, month, year)
 *
 * Set the current system date.
 *
 * day (in): day of of the month (1-31)
 * month (in): month of the year (1-12)
 * year (in): current year (19xx)
 *
 * status = set_date (&day, &month, &year);
 *
 * If the date specified is invalid, set_date returns the value 255.
 *
 */

int set_date (int day, int month, int year)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x2B;
  inregs.h.dh = month;
  inregs.h.dl = day;
  inregs.x.cx = year;

  intdos (&inregs, &outregs);

  return (outregs.h.al);
 }
```

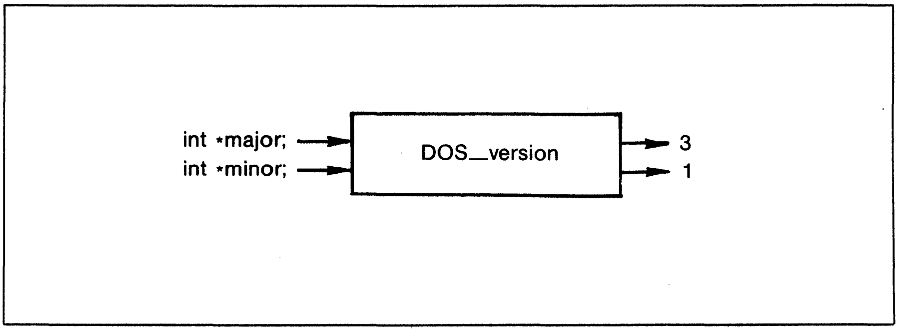



```
#include <dos.h>

/*
 * void get_time (hours, minutes, seconds, hundredths)
 *
 * Get the current system time.
 *
 * hours (out): current hour of the day.
 * minutes (out): current minute of the day.
 * seconds (out): current second of the day.
 * hundredths (out): current hundredths of seconds.
 *
 * get_time (&hours, &minutes, &seconds, &hundredths);
 *
 */

void get_time (int *hours, int *minutes, int *seconds,
               int *hundredths)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x2C;

  intdos (&inregs, &outregs);

  *hours = outregs.h.ch;
  *minutes = outregs.h.cl;
  *hundredths = outregs.h.dl;
  *seconds = outregs.h.dh;
 }
```



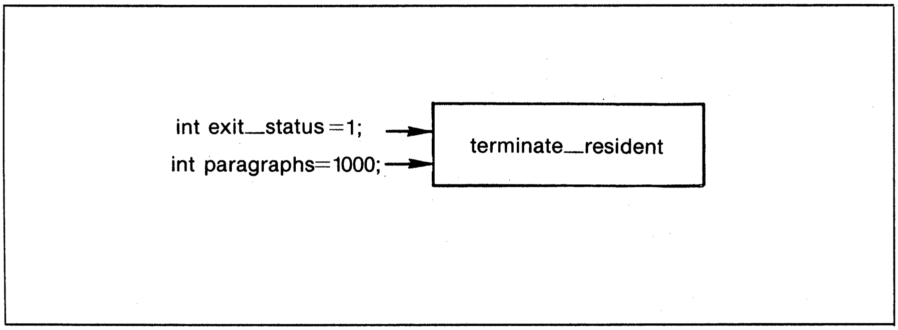

```
#include <dos.h>

/*
 * set_time (hours, minutes, seconds, hundredths)
 *
 * Set the current system time.
 *
 * hours (out): current hour of the day.
 * minutes (out): current minute of the day.
 * seconds (out): current second of the day.
 * hundredths (out): current hundredths of seconds.
 *
 * status = set_time (10, 30, 0, 0);
 *
 * If the time specified is invalid, set_time returns the value 255.
 */

int set_time (int hours, int minutes, int seconds,
              int hundredths)
 {
  union REGS inregs, outregs;
```

```
 inregs.h.ah = 0x2D;
 inregs.h.ch = hours;
 inregs.h.cl = minutes;
 inregs.h.dl = hundredths;
 inregs.h.dh = seconds;
 intdos (&inregs, &outregs);

 return (outregs.h.al);
}
```



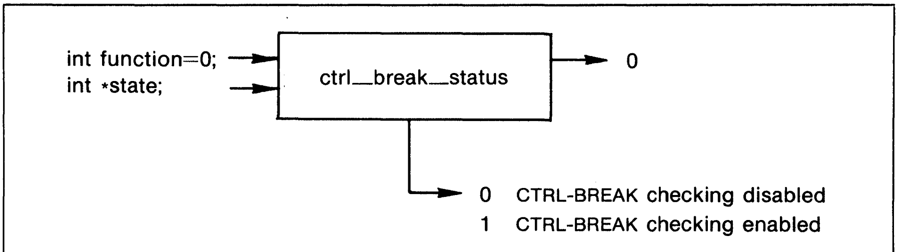

```
#include <dos.h>

/*
 * void DOS_version (major, minor)
 *
 * Return the current DOS version number.
 *
 * major (out): major version number (DOS 3.1 major is 3)
 * minor (out): minor version number (DOS 3.1 minor is 1)
```

```
 *
 * DOS_version (&major, &minor);
 *
 */

int DOS_version (int *major, int *minor)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x30;
  intdos (&inregs, &outregs);
  *major = outregs.h.al;
  *minor = outregs.h.ah;
 }
```




```
#include <dos.h>

/*
 * void terminate_resident (status, paragraphs)
 *
 * Terminate the current program resident in memory.
 *
 * status (in): exit status value for the program.
 * paragraphs (in): number of 16 byte paragraphs regions of memory
 *                  required for termination.
 *
 * terminate_resident (1, 500);
 *
 */
```

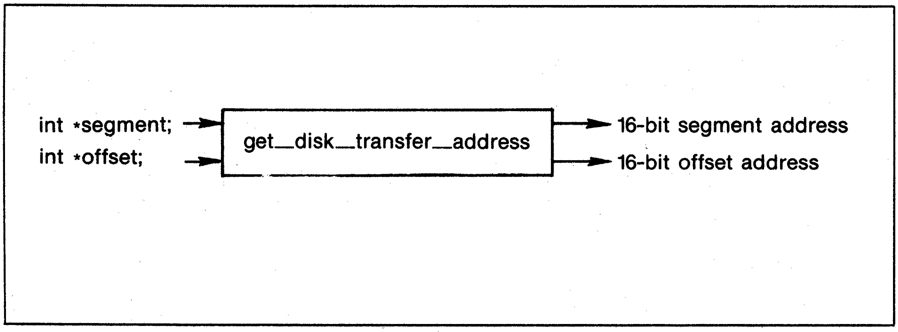
```
terminate_resident (int status, int paragraphs)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x31;
  inregs.h.al = status;
  inregs.x.dx = paragraphs;

  intdos (&inregs, &outregs);
 }
```



If the value of function is 0, the routine returns current state. If function is 1, the routine sets the current state.

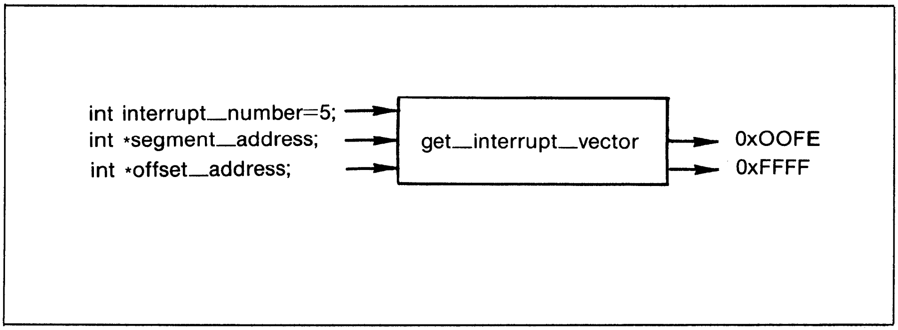
```
#include <dos.h>

/*
 * ctrl_break_status (function, state);
 *
 * Get or set the control break status.
 *
 * function (in): if function is 0, return the current Ctrl-Break state
 *                if function is 1, set the current Ctrl-Break state
 * state (in): if state is 0, disable Ctrl-Break checking
 *                 if state is 1, enable Ctrl-Break checking
 *
 * status = ctrl_break_status (0, 0);
 *
 */

int ctrl_break_status (int function, int state)
 {
```

```
  union REGS inregs, outregs;

  inregs.h.ah = 0x33;
  inregs.h.al = function;
  inregs.h.dl = state;

  intdos (&inregs, &outregs);

  return (outregs.h.dl);
 }
```



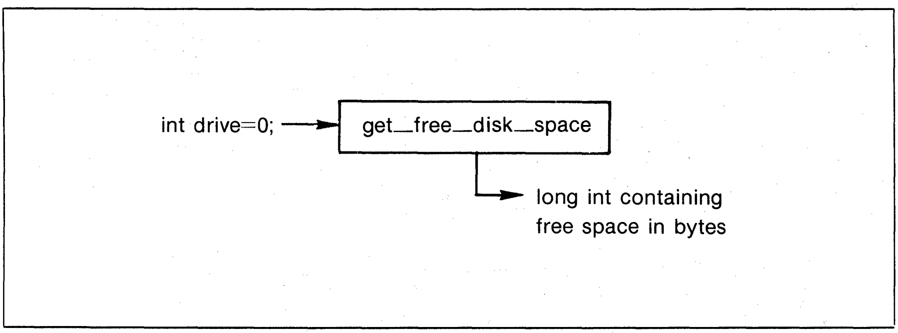

```
#include <dos.h>

/*
 * void get_disk_transfer_address (segment, offset)
 *
 * Return the segment and offset for the DOS disk transfer area.
 *
 * segment (out): segment address of the DTA.
 * offset (out): offset address of the DTA.
 *
 * get_disk_transfer_address (&segment, &offset);
 *
 * By default, the DTA points to offset 80H of the PSP.
 *
 */
```

```
void get_disk_transfer_address (segment, offset)
  int *segment, *offset;
 {
  union REGS inregs, outregs;
  struct SREGS segregs;

  inregs.h.ah = 0x2F;
  intdosx (&inregs, &outregs, &segregs);

  *segment = segregs.es;
  *offset = outregs.x.bx;
 }
```



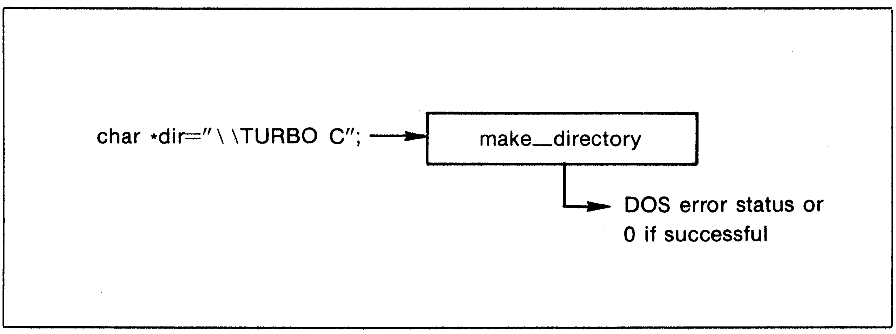

```
#include <dos.h>

/*
 * void get_interrupt_vector (interrupt_number, segment, offset)
 *
 * Return the address of the interrupt handler routine for a
 * specific interrupt.
 *
 * interrupt_number (in): Interrupt number desired.
 * segment (out): Segment address of the routine.
 * offset (in): Offset address of the routine.
 *
 * get_interrupt_vector (5, &segment_address, &offset_address);
```

```
 *
 */

void get_interrupt_vector (int interrupt_number,
                           int *segment, int *offset)
 {
  union REGS inregs, outregs;

  struct SREGS segregs;

  inregs.h.ah = 0x35;
  inregs.h.al = interrupt_number;
  intdosx (&inregs, &outregs, &segregs);

  *segment = segregs.es;
  *offset = outregs.x.bx;
 }
```



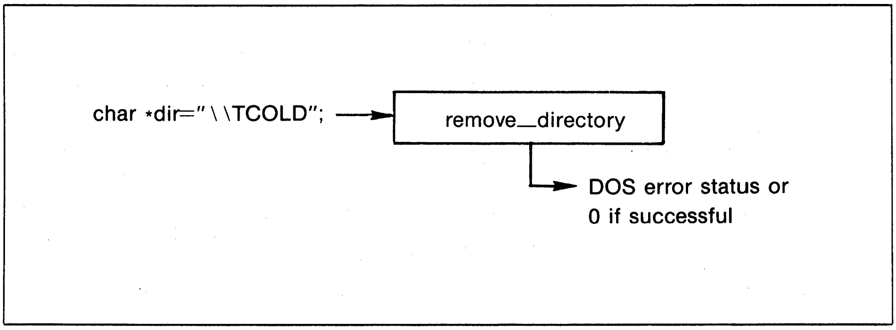

```
#include <dos.h>

/*
 * long get_free_disk_space (drive)
 *
 * Return the number of available bytes for the disk drive specified.
 *
 * drive (in): disk drive id desired.
 *
 * disk_drive = get_free_disk_space (0);
 *
 * Disk drives are specified as:
 *      0 = Current, 1 = A, 2 = B, 3 = C
 *
 */
```

```
long get_free_disk_space (int drive)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x36;
  inregs.h.dl = drive;
  intdos (&inregs, &outregs);

  return ((long) outregs.x.ax * (long) outregs.x.bx
          * (long) outregs.x.cx);
 }
```



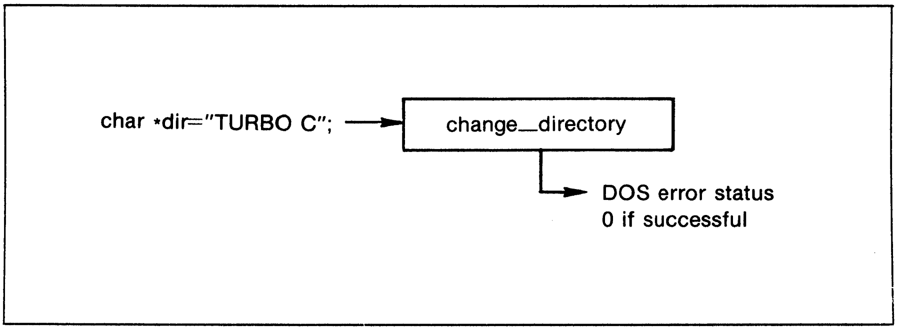

```
#include <dos.h>

/*
 * make_directory (directory)
 *
 * Create a DOS subdirectory with the name specified.
 *
 * directory (in): name of the subdirectory to create.
 *
 * status = make_directory ("\\TURBOC");
 *
 * If make_directory cannot create the directory specified, it will
 * return a DOS error status. Otherwise, make_directory returns 0.
 *
 */

make_directory (char directory[])
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x39;
```

```
    inregs.x.dx = directory;

    intdos (&inregs, &outregs);

    return ((outregs.x.cflag) ? outregs.x.ax: 0);
   }
```

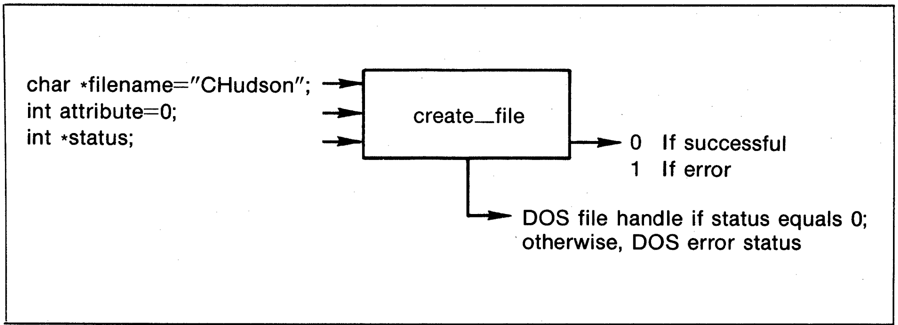



```
#include <dos.h>

/*
 * remove_directory (directory)
 *
 * Remove the DOS subdirectory with the name specified.
 *
 * directory (in): name of the subdirectory to remove.
 *
 * status = remove_directory ("\\TCOLD");
 *
 * If remove_directory cannot remove the directory specified, it will
 * return a DOS error status. Otherwise, remove_directory returns 0.
 *
 */

remove_directory (char directory[])
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x3A;
  inregs.x.dx = directory;
```

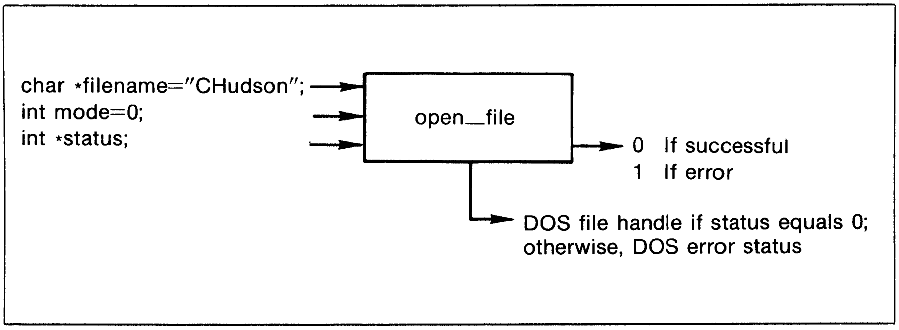
```
 intdos (&inregs, &outregs);

 return ((outregs.x.cflag) ? outregs.x.ax: 0);
}
```



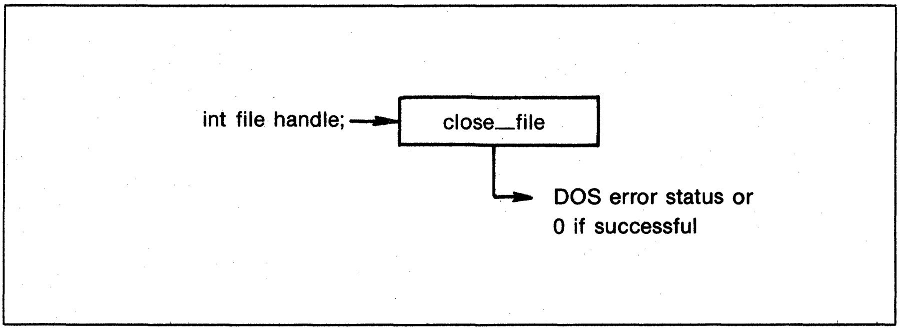

```
#include <dos.h>

/*
 * change_directory (directory)
 *
 * Set the default DOS subdirectory to the directory specified.
 *
 * directory (in): name of the subdirectory to select.
 *
 * status = change_directory ("\\TURBOC");
 *
 * If change_directory cannot select the directory specified, it will
 * return a DOS error status. Otherwise, change_directory returns 0.
 *
 */

change_directory (char directory[])
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x3B;
  inregs.x.dx = directory;

  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? outregs.x.ax: 0);
 }
```



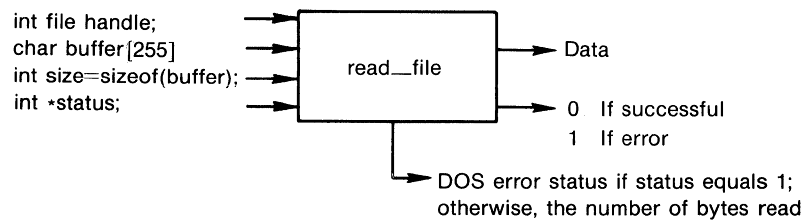

```c
#include <dos.h>

/*
 * create_file (filename, attribute, status)
 *
 * Create a DOS file with the name specified. Return a file handle
 * associated with the new file.
 *
 * filename (in): name of the file to create.
 * attribute (in): desired file attribute.
 * status (out): -1 if an error occurred, otherwise 0.
 *
 * filehandle = create_file ("CHudson"), 0, &status);
 *
 * If create_file cannot create the file as specified, it returns
 * the error status value.  If the creation is successful, create_file
 * returns a file handle to the file.
 *
 */

create_file (char *filename, int attribute, int *status)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x3C;
  inregs.x.cx = attribute;
  inregs.x.dx = filename;

  intdos (&inregs, &outregs);

  *status = (outregs.x.cflag) ? -1: 0;

  return (outregs.x.ax);
 }
```




```
#include <dos.h>

/*
 * open_file (filename, mode, status)
 *
 * Open the DOS file with the name specified in the mode given.
 * Return a file handle associated with the new file.
 *
 * filename (in): name of the file to open.
 * mode (in): specifies how the file is to be opened:
 *             0 is readonly, 1 is write only, 2 is read/write
 * status (out): -1 if an error occurred, otherwise 0.
 *
 * filehandle = open_file ("CHudson", 0, &status);
 *
 * If open_file cannot create the file as specified, it returns
 * the DOS error status value.  If the open is successful, open_file
 * returns a file handle to the file.
 *
 */

open_file (char *filename, int mode, int *status)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x3D;
  inregs.h.al = mode;
  inregs.x.dx = filename;

  intdos (&inregs, &outregs);

  *status = (outregs.x.cflag) ? -1: 0;

  return (outregs.x.ax);
 }
```

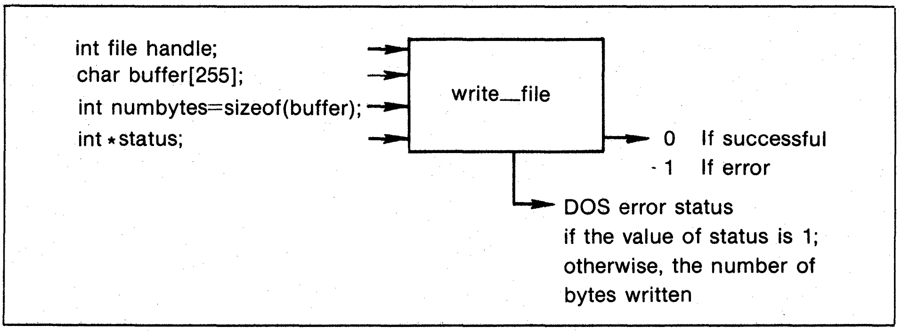



```
#include <dos.h>

/*
 * close_file (filehandle)
 *
 * Close the DOS file associated with the file handle specified.
 *
 * filehandle (in): file handle assoicated with the file to close.
 *
 * If close_file cannot close the file specified, it returns
 * the DOS error status value.  If the close is successful,
 * close_file returns the value 0.
 *
 */
close_file (int filehandle)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x3E;
  inregs.x.bx = filehandle;

  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? outregs.x.ax: 0);
 }
```



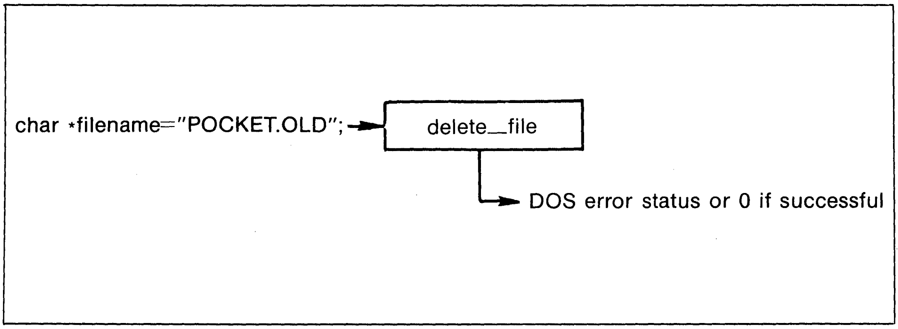

```
#include <dos.h>

/*
 * read_file (filehandle, buffer, numbytes, status)
 *
 * Read the number of bytes specfied from a given file into the
 * buffer provided.
 *
 * filehandle (in): filehandle of the desired file.
 * buffer (out): buffer to contain the bytes read.
 * numbytes (in): number of bytes to read from the file.
 * status (out): error status 1 if error, 0 if successful.
 *
 * bytes = read_file (filehandle, buffer, 255, &status);
 *
 * If an error occurs during the read operation, read_file
 * returns an error status value.  Otherwise, read_file
 * returns the number of bytes read.
 *
 */

read_file (int filehandle, char *buffer,
           int numbytes, int *status)
 {
  union REGS inregs, outregs;
```

```
    inregs.h.ah = 0x3F;
    inregs.x.bx = filehandle;
    inregs.x.cx = numbytes;
    inregs.x.dx = buffer;

    intdos (&inregs, &outregs);

    *status = (outregs.x.cflag) ? 1: 0;

    return (outregs.x.ax);
}
```



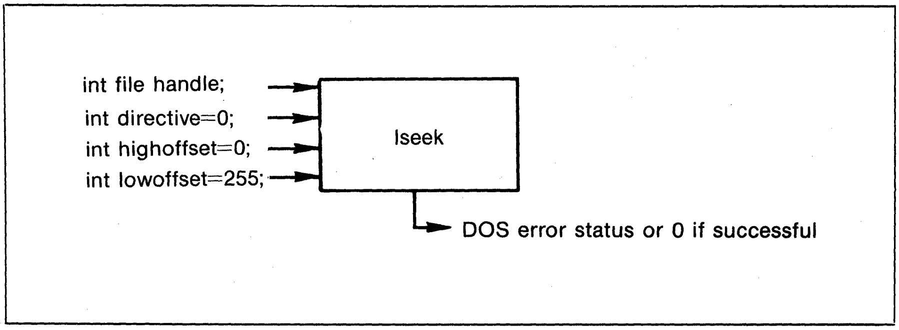

```
#include <dos.h>

/*
 * write_file (filehandle, buffer, numbytes, status)
 *
 * Write the number of bytes specfied to a given file from the
 * buffer provided.
 *
 * filehandle (in): filehandle of the desired file.
 * buffer (in): buffer containing the bytes to write.
 * numbytes (in): number of bytes to write to the file.
 * status (out): error status 1 if error, 0 if successful.
 *
 * bytes = write_file (filehandle, buffer, 255, &status);
 *
 * If an error occurs during the write operation, write_file
 * returns an error status value.  Otherwise, write_file
 * returns the number of bytes written.
 *
 */
```

```
write_file (int filehandle, char *buffer, int numbytes)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x40;
  inregs.x.bx = filehandle;
  inregs.x.cx = numbytes;
  inregs.x.dx = buffer;

  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? outregs.x.ax: 0);
 }
```



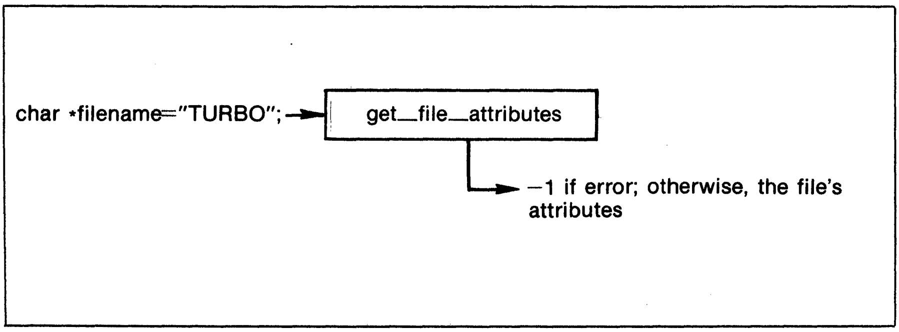

```
#include <dos.h>

/*
 * delete_file (filename);
 *
 * Delete the file with the name specified.
 *
 * filename (in): name of the file to delete.
 *
 * delete_file ("POCKET.OLD");
 *
 * If an error occurs during the delete operation, delete_file
 * returns an error status value.  Otherwise, delete_file
 * returns 0.
 *
 */

delete_file (char *filename)
```

```
{
 union REGS inregs, outregs;

 inregs.h.ah = 0x41;
 inregs.x.dx = filename;

 intdos (&inregs, &outregs);

 return ((outregs.x.cflag) ? outregs.x.ax: 0);
}
```



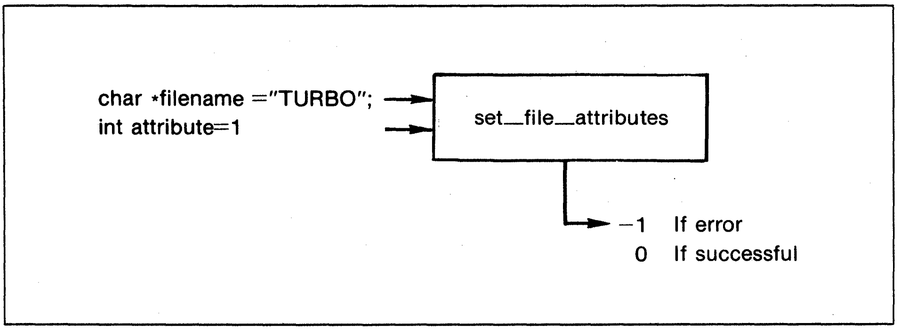

```
#include <dos.h>

/*
 * lseek (filehandle, directive, hioffset, looffset)
 *
 * Move the file pointer in the file associated with a file
 * handle as specified.
 *
 * filehandle (in): file handle of desired file.
 * directive (in): Specifies how to move the file pointer:
 *      0 beginning of file, 1 current location, 3 end of file
 * hioffset: high order 16 bits of the offset to branch to.
 * looffset: low order 16 bits of the offset to branch to.
 *
 * lseek (filehandle, 0, 0, 512);
 *
 * Offsets are treated as a 32 bit value.  As such, we specify
 * a high and low 16 bit combination.
 *
 */

lseek (int filehandle, int directive,
       int hioffset, int looffset)
 {
  union REGS inregs, outregs;
```

```
  inregs.h.ah = 0x42;
  inregs.h.al = directive;
  inregs.x.bx = filehandle;
  inregs.x.cx = hioffset;
  inregs.x.dx = looffset;

  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? outregs.x.ax: 0);
 }
```



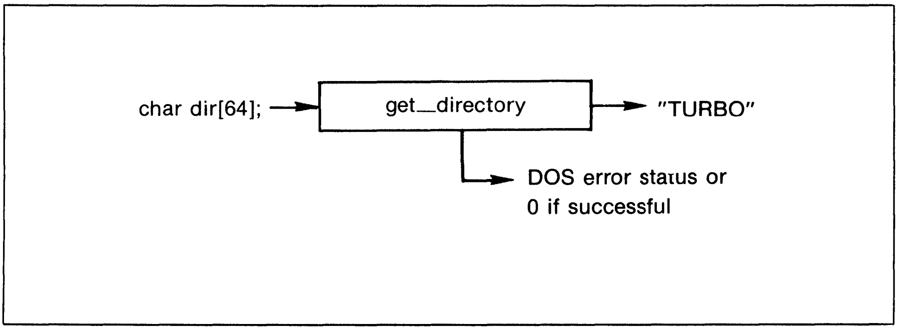

```
#include <dos.h>

/*
 * get_file_attributes (filename)
 *
 * Return the file attributes for the file specified.
 *
 * filename (in): file to return the file attributes of.
 *
 * attributes = get_file_attributes ("Turbo");
 *
 * File attributes include:
 *      1 readonly      2 hidden         4 system
 *      8 volume       16 subdirectory  32 archive
 *
 * If an error occurs, get_file_attributes returns the value -1.
 *
 */

get_file_attributes (char *filename)
 {
  union REGS inregs, outregs;

  inregs.x.ax = 0x4300;
  inregs.x.dx = filename;
```

```
  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? -1: outregs.x.ax);
}
```



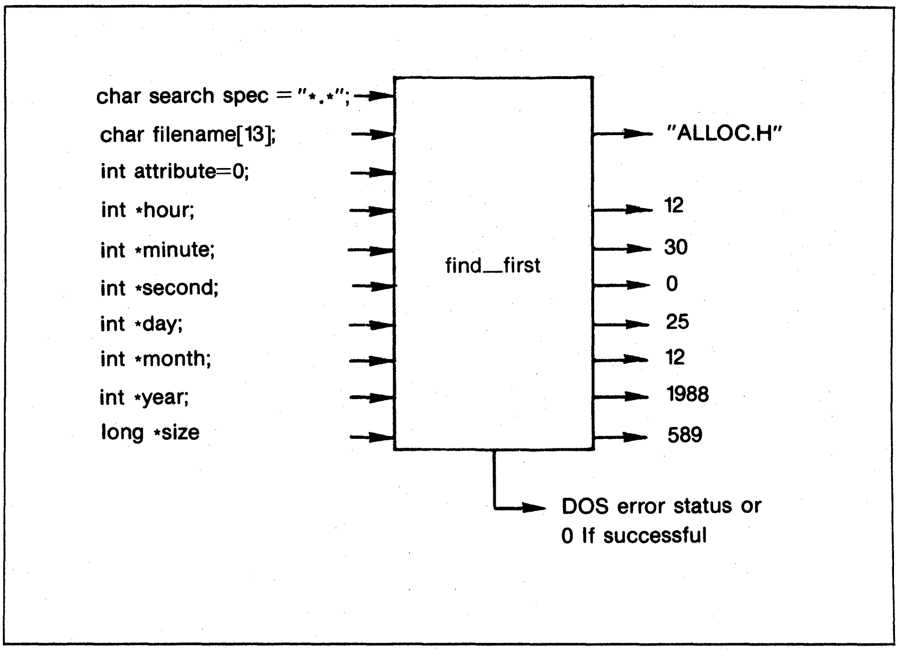

```
#include <dos.h>

/*
 * set_file_attributes (filename, attribute)
 *
 * Set the file attributes for a file as specified.
 *
 * filename (in): file to set the file attributes of.
 * attribute (in): desired file attributes.
 *
 * status = set_file_attributes ("Turbo", 32);
 *
 * File attributes include:
 *      1 readonly      2 hidden         4 system
 *      8 volume       16 subdirectory  32 archive
 *
 * If an error occurs, set_file_attributes returns the value -1.
 *
 */
```

```
set_file_attributes (char *filename, int attribute)
 {
  union REGS inregs, outregs;

  inregs.x.ax = 0x4301;
  inregs.x.cx = attribute;
  inregs.x.dx = filename;

  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? -1: 0);
 }
```



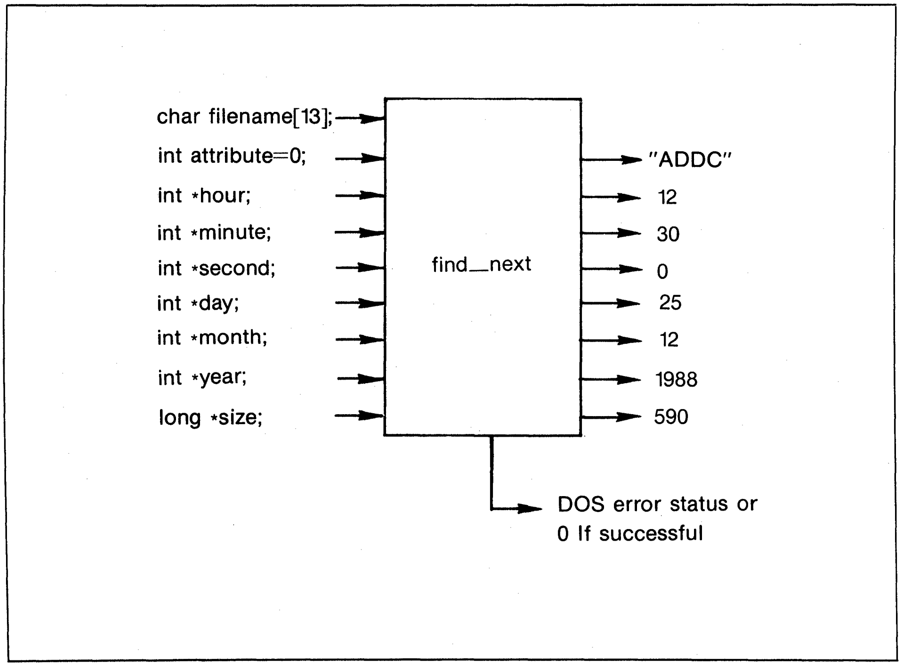

```
#include <dos.h>

/*
 * get_directory (directory, drive);
 *
 * Return the current directory for the disk drive specified.
 *
 * directory (out): current directory name.
 * drive (in): disk drive number of the drive of interest.
 *
 * status = get_directory (directory, 0);
```

```
 *
 * Disk drive numbers are specified as:
 *      0 = Current, 1 = A, 2 = B, 3 = C
 *
 */

get_directory (char *directory, int drive)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x47;
  inregs.h.dl = drive;
  inregs.x.si = directory;

  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? outregs.x.ax: 0);
 }
```



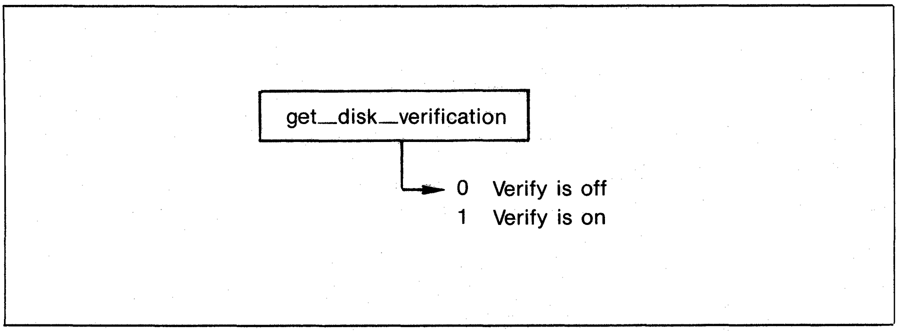
```
#include <dos.h>

/*
 * find_first (searchspec, filename, attribute, hour, minute,
 *             second, day, month, year, size)
 *
 * Return information on the first file matching the search
 * specification given.
 *
 * searchspec (in): File name or DOS wildcard characters of the
 *                  file(s) to match ("A", "TEST.C", "*.*").
 * filename (out): Name of the first matching file.
 * attribute (in): Attributes of the files we are searching for.
 * hour (out): Hour time stamp.
 * minute (out): Minute time stamp.
 * second (out): Second time stamp.
 * day (out): Day time stamp.
 * month (out): Month time stamp.
 * year (out): Year time stamp.
 * size (out): File size in bytes.
 *
 * status = find_first ("*.c", filename, 0, &hour, &minute,
 *                      &second, &day, &month, &year, &size);
 *
 * If an error occurs, find_first returns the error status value.
 * Otherwise, find_first returns the value 0.
 *
 */

find_first (char *searchspec, char *filename, int attribute,
            int *hour, int *minute, int *second,
            int *day, int *month, int *year, long int *size)
 {
  union REGS inregs, outregs;

  int segment, offset, i;
  unsigned int time, date;

  void get_disk_transfer_address (int *, int *);

  inregs.h.ah = 0x4E;
  inregs.x.dx = searchspec;
  inregs.x.cx = attribute;
  intdos (&inregs, &outregs);

  if (outregs.x.cflag)
    return (outregs.x.ax);

  get_disk_transfer_address (&segment, &offset);

  time = peek(segment, offset+22);
  date = peek(segment, offset+24);
```

```
    *year = (date >> 9) + 1980;
    *month = (date & 0x1E0) >> 5;
    *day = date & 0x1F;
    *hour = time >> 11;
    *minute = (time & 0x7E0) >> 5;
    *second = (time & 0x1F) * 2;

    *size = peek(segment, offset+28);
    *size = *size << 16;
    *size += (unsigned) peek(segment, offset+26);

    for (i = 0; i < 13; i++)
      *filename++ = peekb(segment, offset+30+i);

    *filename = '\0';

    return (0);
}
```



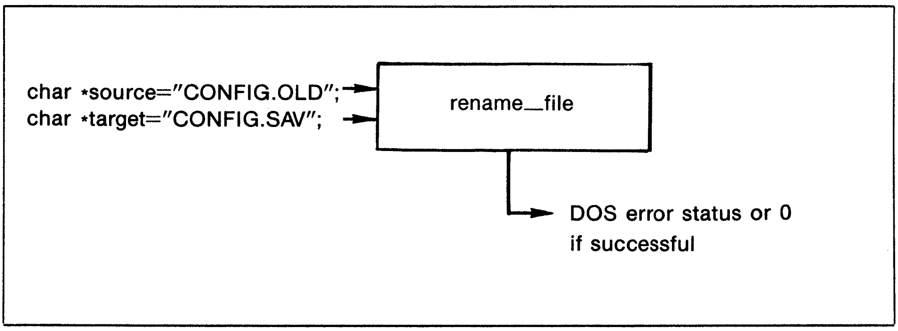
```
#include <dos.h>

/*
 * find_next (filename, attribute, hour, minute,
 *            second, day, month, year, size)
 *
 * Return information on the next file matching the search
 * specification given on a call to find_first.
 *
 * filename (out): Name of the first matching file.
 * attribute (in): Attributes of the files we are searching for.
 * hour (out): Hour time stamp.
 * minute (out): Minute time stamp.
 * second (out): Second time stamp.
 * day (out): Day time stamp.
 * month (out): Month time stamp.
 * year (out): Year time stamp.
 * size (out): File size in bytes.
 *
 * status = find_next (filename, 0, &hour, &minute,
 *                     &second, &day, &month, &year, &size);
 *
 * If an error occurs, find_next returns the error status value.
 * Otherwise, find_next returns the value 0.
 *
 */

find_next (char *filename, int attribute, int *hour,
           int *minute, int *second, int *day, int *month,
           int *year, long int *size)
 {
  union REGS inregs, outregs;

  int segment, offset, i;
  unsigned int time, date;

  void get_disk_transfer_address (int *, int *);

  inregs.h.ah = 0x4F;
  inregs.x.cx = attribute;
  intdos (&inregs, &outregs);

  if (outregs.x.cflag)
    return (outregs.x.ax);

  get_disk_transfer_address (&segment, &offset);

  time = peek(segment, offset+22);
  date = peek(segment, offset+24);

  *year = (date >> 9) + 1980;
  *month = (date & 0x1E0) >> 5;
  *day = date & 0x1F;
```

```
  *hour = time >> 11;
  *minute = (time & 0x7E0) >> 5;
  *second = (time & 0x1F) * 2;

  *size = peek(segment, offset+28);
  *size = *size << 16;
  *size += (unsigned) peek(segment, offset+26);

  for (i = 0; i < 13; i++)
    *filename++ = peekb(segment, offset+30+i);

  *filename = '\0';

  return (0);
}
```



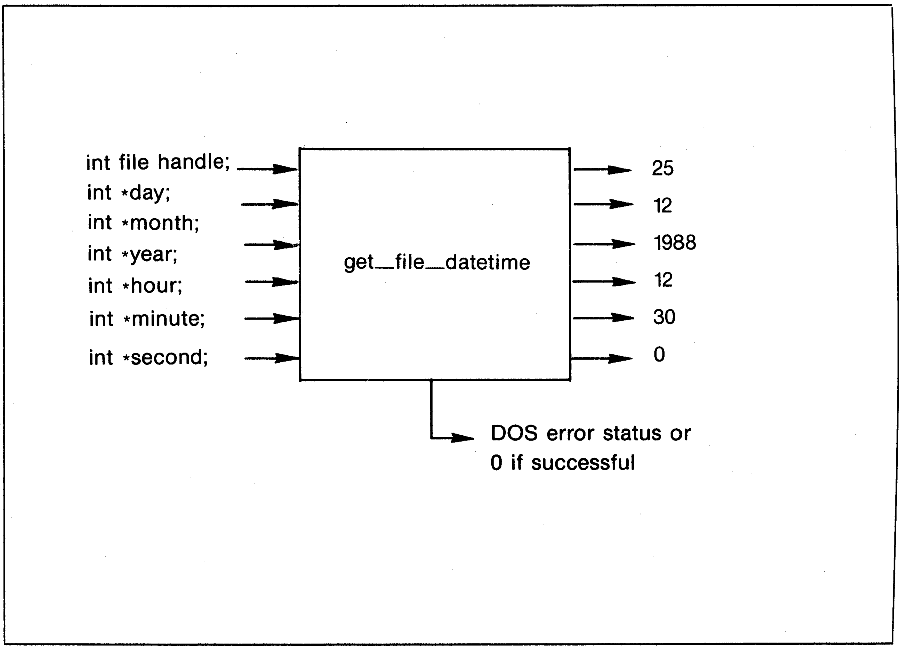

```
#include <dos.h>

/*
 * get_disk_verification ()
 *
 * Return the current state of disk verification on (1) or off (0).
 *
 * state = get_disk_verification ();
 *
 */
```

```
int get_disk_verification ()
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x54;
  intdos (&inregs, &outregs);
  return (outregs.h.al);
 }
```



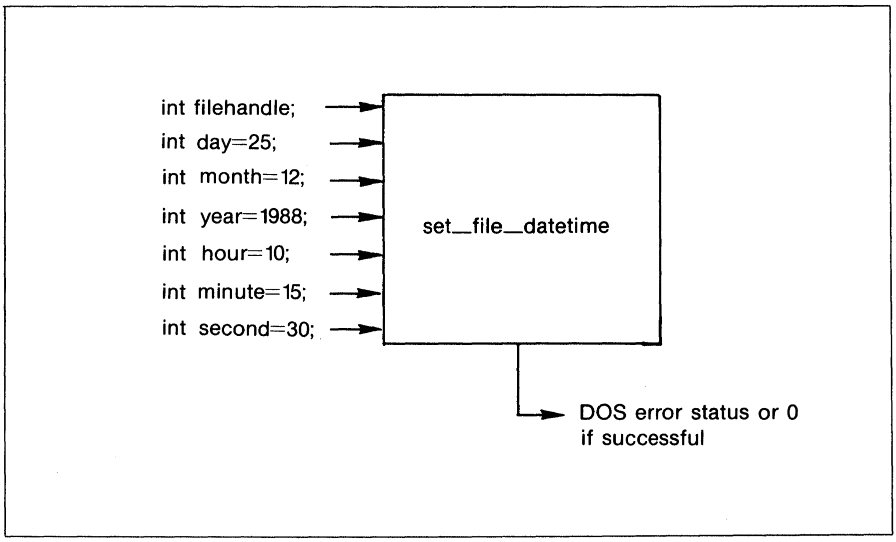

```
#include <dos.h>

/*
 * rename_file (source, target)
 *
 * Rename the file specified by source to the name given by target.
 *
 * source (in): old file name.
 * target (in): desired file name.
 *
 * rename_file ("CONFIG.OLD", "CONFIG.SAV");
 *
 * If an error occurs, rename_file returns the DOS error status code.
 * Otherwise, rename_file returns the value 0.
 *
 */
```

```
rename_file (char *source, char *target)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x56;
  inregs.x.dx = source;
  inregs.x.di = target;

  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? outregs.x.ax: 0);
 }
```

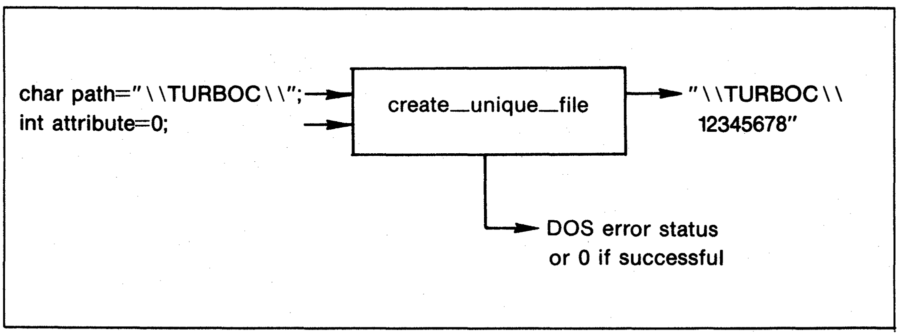



```
#include <dos.h>

/*
 * get_file_datetime (filehandle, day, month, year,
 *                    hour, minute, second);
 *
 * Return the date and time stamp for the file associated with
 * the file handle given.
 *
 * filehandle (in): file handle of the desired file.
 * day (out): day of month the file was created or modified (1-31).
 * month (out): month of year the file was created or modified (1-12).
```

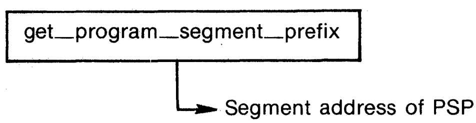
```
 * year (out): year file was created or modified (1980-2099).
 * hour (out): hour of day file was created or modified (0-23).
 * minute (out): minute of day file was created or modified (0-59).
 * second (out): second of day file was created or modified (0-59).
 *
 * get_file_datetime (filehandle, &day, &month, &year,
 *                    &hour, &minute, &second)
 *
 */

get_file_datetime (int filehandle, int *day, int *month,
                   int *year, int *hour, int *minute, int *second)
 {
  union REGS inregs, outregs;

  inregs.x.ax = 0x5700;
  inregs.x.bx = filehandle;

  intdos (&inregs, &outregs);

  *year = (outregs.x.dx >> 9) + 1980;
  *month = (outregs.x.dx & 0x1E0) >> 5;
  *day = outregs.x.dx & 0x1F;
  *hour = outregs.x.cx >> 11;
  *minute = (outregs.x.cx & 0x7E0) >> 5;
  *second = (outregs.x.dx & 0x1F) * 2;

  return ((outregs.x.cflag) ? outregs.x.ax: 0);
 }
```



```
#include <dos.h>

/*
 * set_file_datetime (filehandle, day, month, year,
 *                    hour, minute, second);
 *
 * Set the date and time stamp for the file associated with
 * the file handle given.
 *
 * filehandle (in): file handle of the desired file.
 * day (out): day of month the file was created or modified (1-31).
 * month (out): month of year the file was created or modified (1-12).
 * year (out): year file was created or modified (1980-2099).
 * hour (out): hour of day file was created or modified (0-23).
 * minute (out): minute of day file was created or modified (0-59).
 * second (out): second of day file was created or modified (0-59).
 *
 * set_file_datetime (filehandle, 25, 12, 1988, 10, 30, 0);
 *
 */

set_file_datetime (int filehandle, int day, int month,
                   int year, int hour, int minute, int second)
 {
  union REGS inregs, outregs;

  inregs.x.ax = 0x5701;
  inregs.x.bx = filehandle;
  inregs.x.dx = (year - 1980) << 9;
  inregs.x.dx += month << 5;
  inregs.x.dx += day;
  inregs.x.cx = hour << 11;
  inregs.x.cx += minute << 5;
  inregs.x.cx += second / 2;

  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? outregs.x.ax: 0);
 }
```



```
#include <dos.h>

/*
 * create_unique_file (pathname, attribute)
 *
 * Create a file with a unique name in the directory specified.
 *
 * pathname (in): directory path to place the file in.
 * attribute (in): desired attribute for the file.
 *
 * status = create_unique_file (path, attribute);
 *
 * Path names may be defined as:
 *
 *      char path [255] = "\\TURBOC\\OLDFILES\\";
 *      char path [255] = "\\";
 *
 */

create_unique_file (char *pathname, int attribute)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x5A;
  inregs.x.cx = attribute;
  inregs.x.dx = pathname;

  intdos (&inregs, &outregs);

  return ((outregs.x.cflag) ? outregs.x.ax: 0);
 }
```




```
#include <dos.h>

/*
 * get_program_segment_prefix ()
 *
```

```
 * Return the segment address of the program segment prefix for
 * the current program.
 *
 */

get_program_segment_prefix ()
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x62;
  intdos (&inregs, &outregs);
  return (outregs.x.bx);
 }
```

Using the Programs

Admittedly, this chapter has presented you with a large collection of routines. Although many of these routines appeared to perform basic functions, they become quite powerful when you use them in larger programs. Many of the programs in later chapters make extensive use of these routines.

C H A P T E R

7

Turbo C BIOS Interface

Chapter 6 discussed how to use the DOS system services to add many powerful routines to your library of Turbo C functions. Just as DOS provides a series of routines that your Turbo C programs can access, so do the IBM PC and PC compatibles. This collection of routines resides in the PC's read-only memory (ROM) and is commonly called the ROM BIOS. This is because the routines perform the Basic Input Output Services (BIOS).

As was the case in Chapter 6, this chapter does not attempt to implement all of the BIOS services; instead, it examines a select group of services that are useful in many Turbo C applications. Most of the routines in this chapter deal specifically with video control.

As with the DOS system services, you must again use the 8088 registers as your interface to the BIOS services. Thus, you must include the file dos.h at the beginning of your programs, as shown here:

```
#include <dos.h>
```

Unlike the DOS services (which used the routines intdos and intdosx), the BIOS services use int86. The calling sequence for int86 is as follows:

```
int86 (interrupt_number, &inregs, &outregs);
```

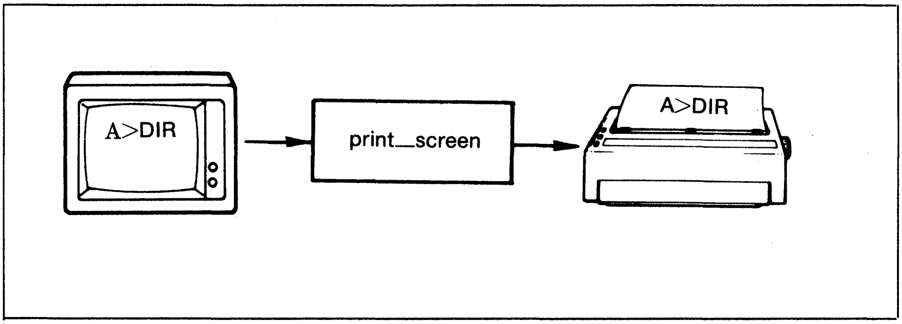
DOS services use INT 21H, as discussed in Chapter 6. The BIOS services, however, use the following interrupts:

Print screen	INT 05H
Video services	INT 10H
Equipment service	INT 11H
Memory size	INT 12H
Disk services	INT 13H
Port services	INT 14H
AT extended services	INT 15H
Keyboard services	INT 16H
Printer services	INT 17H
ROM BASIC	INT 18H
Reboot service	INT 19H
Time services	INT 1AH

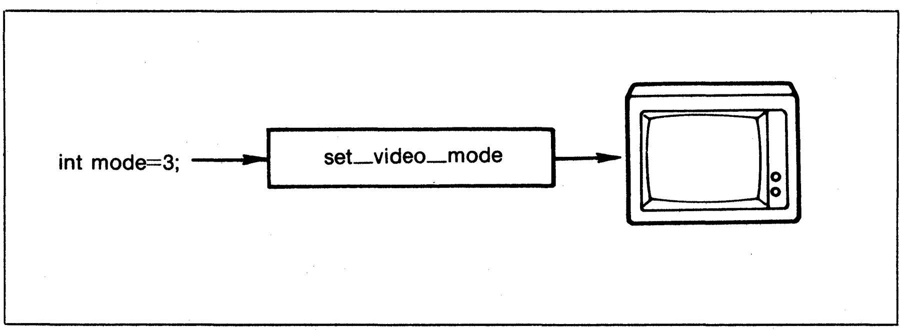
In addition to the input and output register structures, you must specify an interrupt number, as shown here:

```
int86 (0x10, &intregs, &outregs);
```

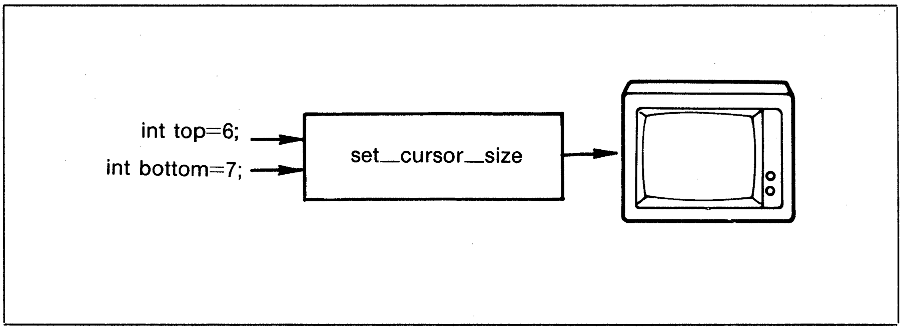
As before, many of these routines exist in the Turbo C run-time library. To help you better understand how to control your PC, however, the following routines implement a library of ROM BIOS routines:




```
#include <dos.h>

/*
 * void print_screen ()
 *
 * Print the contents of the current screen display.
 *
 * print_screen ();
 *
 */

void print_screen ()
 {
  union REGS inregs, outregs;

  int86 (5, &inregs, &outregs);
 }
```

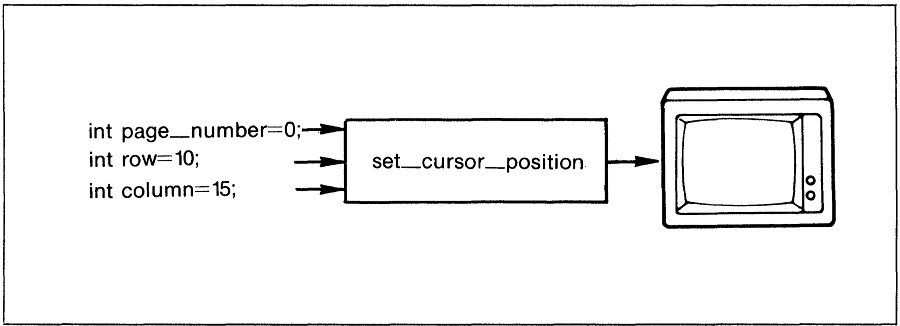


```
#include <dos.h>

/*
 * void set_video_mode (mode)
 *
 * Set the current video display mode.
 *
 * mode (in): Video mode desired. Common modes include:
 *              0 40 x 25 grey   1 40 x 25 color    2 80 x 25 grey
 *              3 80 x 25 color  4 320 x 200 color
 *              5 320 x 200 grey 6 640 x 400 graphics
 *              7 85 x 25 text
 *
 * set_video_mode (4);
 *
 */

void set_video_mode (int mode)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0;
  inregs.h.al = mode;
  int86 (0x10, &inregs, &outregs);
 }
```

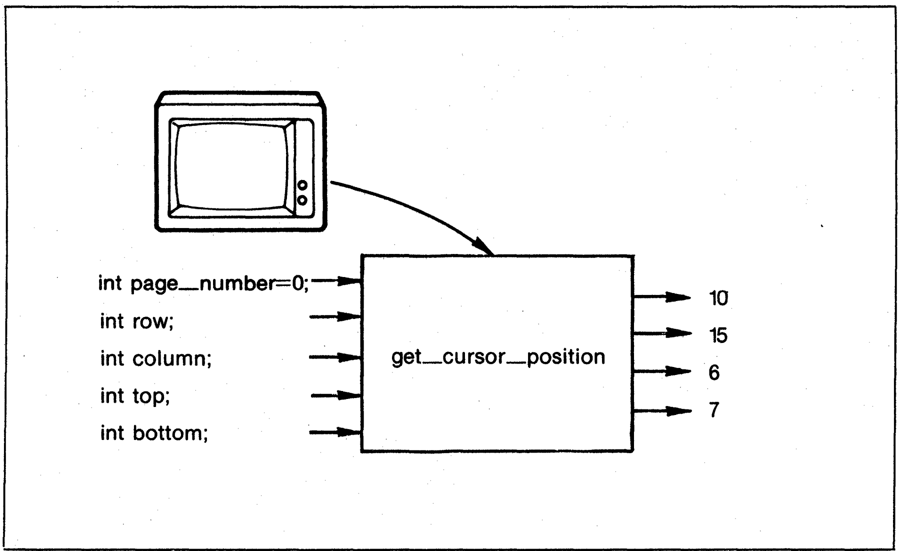



```
#include <stdio.h>

#include <dos.h>

/*
 * void set_cursor_size (start, stop)
 *
 * Set the current cursor size.
 *
 * start (in): top scan line.
 * stop (in): bottom scan line.
 *
 * set_cursor_size (8, 7);
 *
 * For CGA scan lines range from 0 to 7. For monochrome scan
 * lines range from 0 to 13. If you make the top scan line larger
```

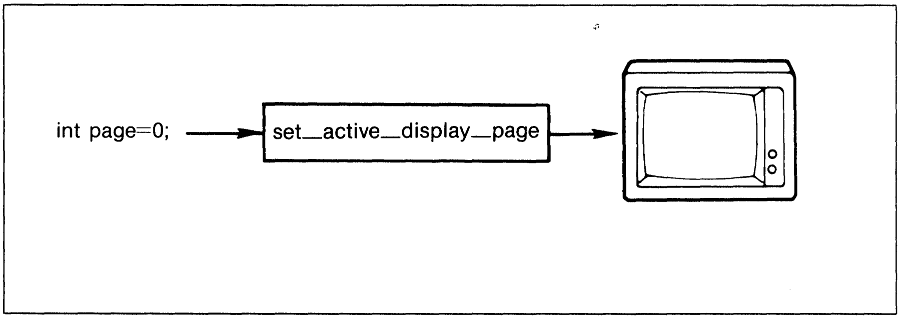
```
 * than the bottom scan line, the cursor disappears.
 *
 */

void set_cursor_size (int start, int stop)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 1;
  inregs.h.ch = start;
  inregs.h.cl = stop;
  int86 (0x10, &inregs, &outregs);
 }
```



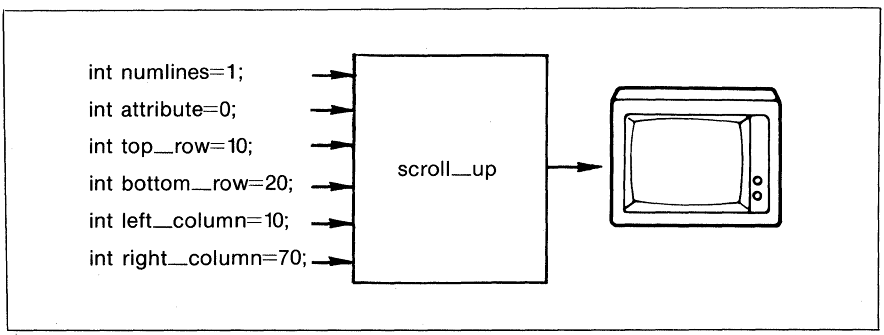

```
#include <dos.h>

/*
 * void set_cursor_position (page_number, row, column)
 *
 * Place the cursor at the row and column given for the video
 * display page specified.
 *
 * page_number (in): Video page number.
 * row (in): Desired row number.
 * column (in): Desired column number.
 *
 * set_cursor_position (0, 10, 15);
 *
 */

void set_cursor_position (int page_number, int row, int column)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 2;
  inregs.h.bh = page_number;
  inregs.h.dh = row;
  inregs.h.dl = column;
  int86 (0x10, &inregs, &outregs);
 }
```



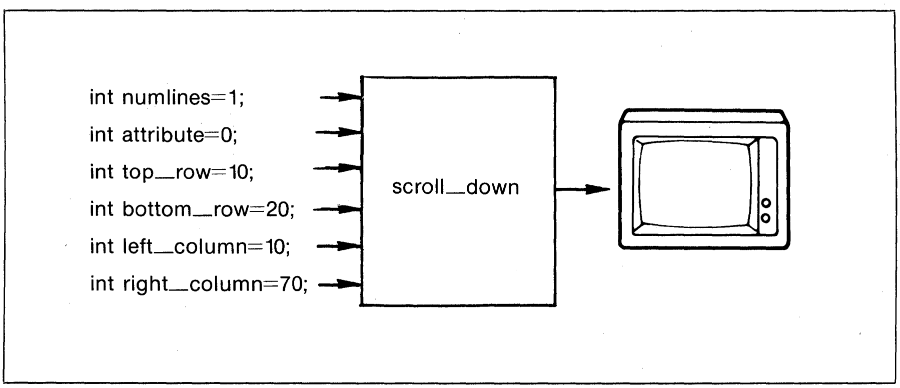

```
#include <dos.h>

/*
 * void get_cursor_position (page_number, row, column, start, stop)
 *
 * Get cursor information for the video page specified.
 *
 * page_number (in): Page number to return cursor information for.
 * row (out): Current cursor row number.
 * column (out): Current cursor column number.
 * start (out): Top cursor scan line.
 * stop (out): Bottom cursor scan line.
 *
 * get_cursor_position (0, &row, &column, &start, &stop);
 *
 */

void get_cursor_position (int page_number, int *row,
                          int *column, int *start, int *stop)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 3;
  inregs.h.bh = page_number;
  int86 (0x10, &inregs, &outregs);
  *row = outregs.h.dh;
  *column = outregs.h.dl;
  *start = outregs.h.ch;
  *stop = outregs.h.cl;
 }
```



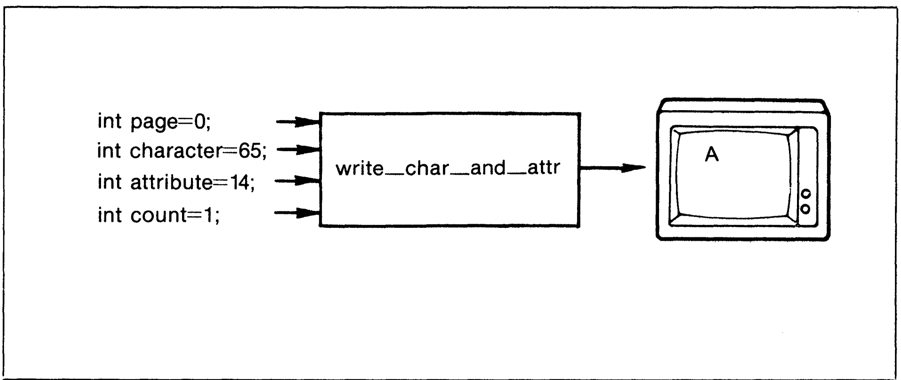

```
#include <dos.h>

/*
 * set_active_display_page (page)
 *
 * Select the video display page that is visible on the screen.
 *
 * page (in): Desired video display page.
 *
 * set_active_display_page (3);
 *
 * By writing to a nonactive video display and then selecting
 * the page as active, the video output appears instantaneous.
 *
 */

void set_active_display_page (int page)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 5;
  inregs.h.al = page;
  int86 (0x10, &inregs, &outregs);
 }
```



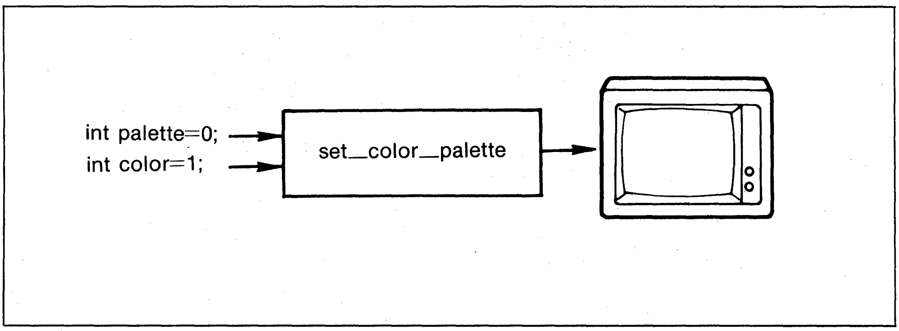
```
#include <dos.h>

/*
 * scroll_up (numlines, attribute, top_row, bottom_row,
 *            left_column, right_column)
 *
 * Scroll the text on a region of the screen up as specified.
 *
 * numlines (in): Number of lines to scroll up.
 * attribute (in): Attribute of line(s) left blank by the scroll.
 * top_row (in): Upper row of the region to scroll.
 * bottom_row (in): Lower row of the region to scroll.
 * left_column (in): Left column of the region to scroll.
 * right_column (in): Right column of the region to scroll.
 *
 * scroll_up (1, 0, 10, 20, 10, 50);
 *
 */

void scroll_up (int numlines, int attribute, int top_row,
               int bottom_row, int left_column, int right_column)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 6;
  inregs.h.al = numlines;
  inregs.h.bh = attribute;
  inregs.h.ch = top_row;
  inregs.h.dh = bottom_row;
  inregs.h.cl = left_column;
  inregs.h.dl = right_column;
  int86 (0x10, &inregs, &outregs);
 }
```



```
#include <dos.h>

/*
 * scroll_down (numlines, attribute, top_row, bottom_row,
 *              left_column, right_column)
 *
 * Scroll the text on a region of the screen down as specified.
 *
 * numlines (in): Number of lines to scroll down.
 * attribute (in): Attribute of line(s) left blank by the scroll.
 * top_row (in): Upper row of the region to scroll.
 * bottom_row (in): Lower row of the region to scroll.
 * left_column (in): Left column of the region to scroll.
 * right_column (in): Right column of the region to scroll.
 *
 * scroll_down (1, 0, 10, 20, 10, 50);
 *
 */

void scroll_down (int numlines, int attribute, int top_row,
                  int bottom_row, int left_column, int right_column)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 7;
  inregs.h.al = numlines;
  inregs.h.bh = attribute;
  inregs.h.ch = top_row;
  inregs.h.dh = bottom_row;
  inregs.h.cl = left_column;
  inregs.h.dl = right_column;
  int86 (0x10, &inregs, &outregs);
 }
```

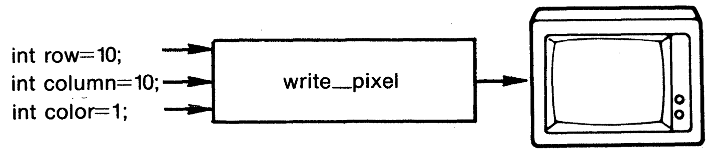


```
#include <dos.h>

/*
 * void write_char_and_attr (page, character, attribute, count)
 *
 * Write the number of occurrences specified of a given
 * character (and attribute) on the display page provided.
 *
 * page (in): Video display page to write character to.
 * character (in): ASCII character to display.
 * attribute (in): Video display attribute of character.
 * count (in): Number of times to display character.
 *
 * write_char_and_attr (0, 65, 14, 10);
 *
 */

void write_char_and_attr (int page, int character,
                          int attribute, int count)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 9;
  inregs.h.al = character;
  inregs.h.bh = page;
  inregs.h.bl = attribute;
  inregs.x.cx = count;
  int86 (0x10, &inregs, &outregs);
 }
```

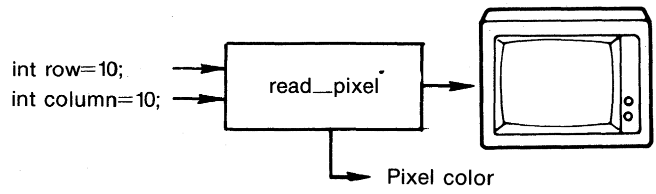



```
#include <dos.h>

/*
 * void set_color_palette (palette, color)
 *
 * Set the color palette and select a color for graphics display.
 *
 * palette (in): Desired color palette.
 * color (in): Desired color from the palette selected.
 *
 * set_color_palette (0, 1);
 *
 */
```

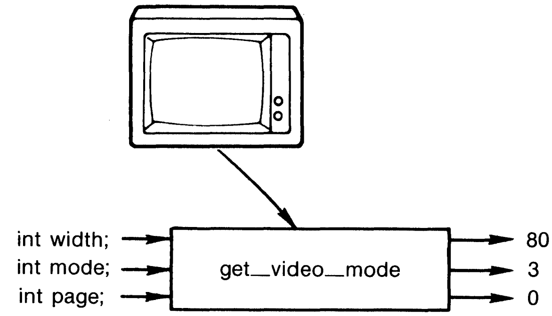
```
void set_color_palette (int palette, int color)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x0B;
  inregs.h.bh = palette;
  inregs.h.bl = color;
  int86 (0x10, &inregs, &outregs);
 }
```



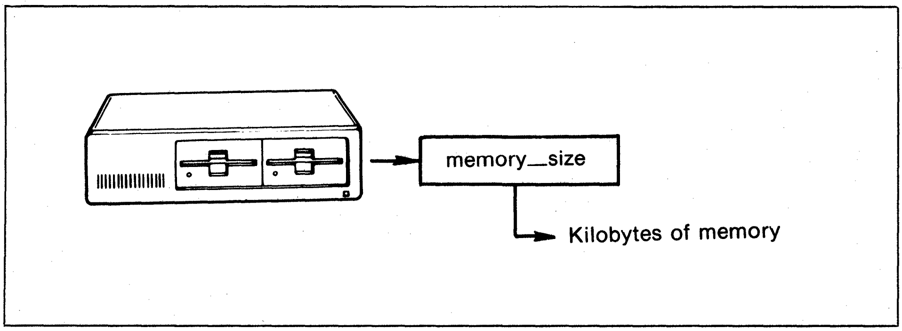

```
#include <dos.h>

/*
 * void write_pixel (row, column, color)
 *
 * Write a graphics pixel of the color given at the row and column
 * location specified.
 *
 * row (in): Pixel row position.
 * column (in): Pixel column position.
 * color (in): Pixel color.
 *
 * write_pixel (10, 10, 1);
 *
 */

void write_pixel (int row, int column, int color)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x0C;
  inregs.h.al = color;
  inregs.x.cx = column;
  inregs.h.dl = row;
  int86 (0x10, &inregs, &outregs);
 }
```



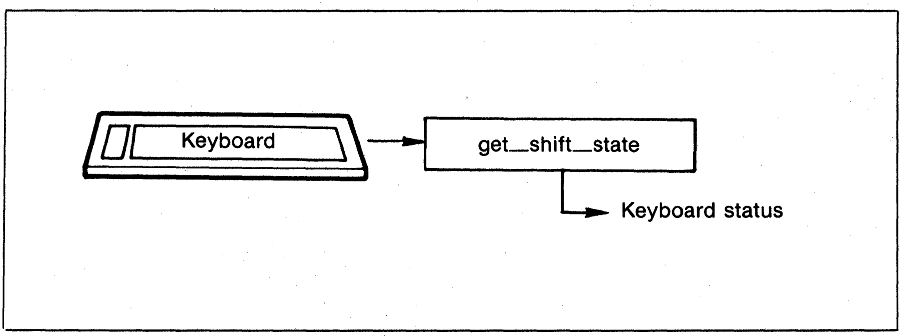

```
#include <dos.h>

/*
 * read_pixel (row, column)
 *
 * Return the color of the pixel at the row and column specified.
 *
 * row (in): Pixel row position.
 * column (in): Pixel column position.
 *
 * color = read_pixel (10, 10);
 *
 */

read_pixel (int row, int column)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x0D;
  inregs.x.cx = column;
  inregs.h.dl = row;
  int86 (0x10, &inregs, &outregs);

  return (outregs.h.al);
 }
```



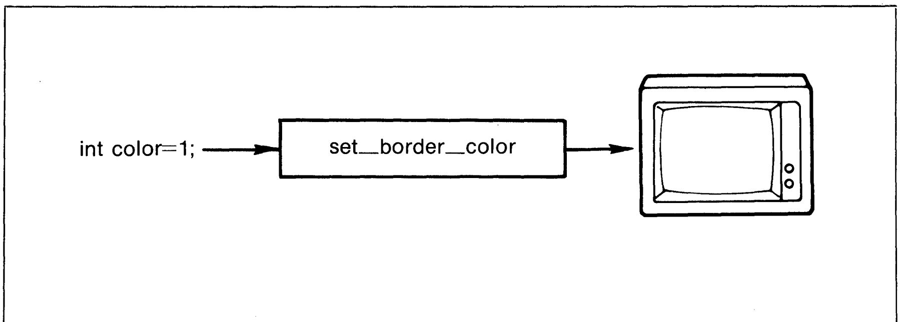

```
#include <dos.h>

/*
 * void get_video_mode (width, mode, page)
 *
 * Return the current video display status.
 *
 * width (out): Number of characters per line (40 or 80).
 * mode (out): Current video mode. (See set_video_mode)
 * page (out): Current video display page.
 *
 * get_video_mode (&width, &mode, &page);
 *
 */

void get_video_mode (int *width, int *mode, int *page)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x0F;
  int86 (0x10, &inregs, &outregs);

  *width = outregs.h.ah;
  *mode = outregs.h.al;
  *page = outregs.h.bh;
 }
```




```
#include <dos.h>

/*
 * memory_size ()
 *
 * Return the number of kilobytes of memory in the system.
 *
 * num_bytes = memory_size ();
 *
 */

memory_size ()
 {
  union REGS inregs, outregs;

  int86 (0x12, &inregs, &outregs);

  return (outregs.x.ax);
 }
```



```
#include <dos.h>

/*
 * get_shift_state ()
 *
 * Return the current keyboard state.
 *
 * state = get_shift_state ();
 *
 * get_shift_state returns a byte whose bits define:
 *      bit 0 Right shift depressed   bit 1 Left shift depressed
 *      bit 2 Ctrl depressed          bit 3 Alt depressed
 *      bit 4 scroll lock on          bit 5 num lock on
 *      bit 6 caps lock on            bit 7 ins on
 *
 */

get_shift_state ()
 {
  union REGS inregs, outregs;

  inregs.h.ah = 2;
  int86 (0x16, &inregs, &outregs);

  return (outregs.h.al);
 }
```




```
#include <dos.h>

/*
 * void set_border_color (color)
 *
 * Set the current border color for CGA monitors in text mode.
 *
 * color (in): Desired color (0 - 15).
 *
```

```
 * set_border_color (1);
 *
 */

void set_border_color (int color)
 {
  union REGS inregs, outregs;

  inregs.h.ah = 0x0B;
  inregs.h.bh = 0;
  inregs.h.bl = color;
  int86 (0x10, &inregs, &outregs);
 }
```

By using the routines provided in this chapter, you can quickly produce routines of professional quality. Experiment with these functions, and your programs should gain tremendous flexibility.

CHAPTER

8

Turbo C ANSI Support

Chapters 6 and 7 included several routines that provided video and I/O support from DOS and the ROM BIOS services. In addition to these routines, the Turbo C run-time library provides many useful I/O functions. This chapter, which completes your library of screen-manipulation routines, examines the ANSI driver provided with both MS-DOS and PC DOS.

The ANSI driver software intercepts data that is sent from the keyboard and data that is sent to the video display in search of ANSI commands (see Figure 8-1).

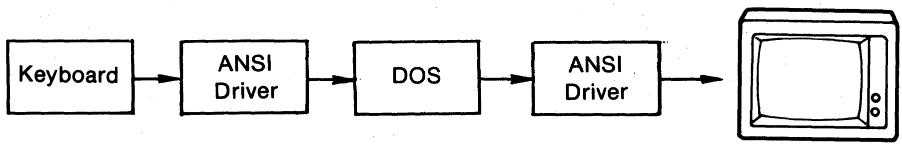


Figure 8-1. *Operation of the ANSI driver*

An ANSI command is a series of characters that begin with an ASCII 27 (commonly known as an escape character). For example, the following ANSI command clears the screen display and places the cursor in the upper-left (home) position on the screen:

ESC[2J

The following program uses this escape sequence and printf to clear the contents of the screen display (similar to the DOS CLS command):

```
main ()
 {
  printf ("\033[2J");
 }
```

All of the ANSI commands are this easy to use.

Before you can use the ANSI driver, you must be sure that the ANSI driver software is installed on your system. To do so, be sure that the file ANSI.SYS is on your boot disk by issuing the following command:

```
C> DIR ANSI.SYS

 Volume in drive C is S
 Directory of  C:\

ANSI      SYS      1651    8-01-87  12:00a
        1 File(s)    1093632 bytes free
```

Next, place the following entry in the CONFIG.SYS file and reboot:

```
DEVICE=ANSI.SYS
```

Once the system restarts, DOS will have loaded the ANSI driver support.

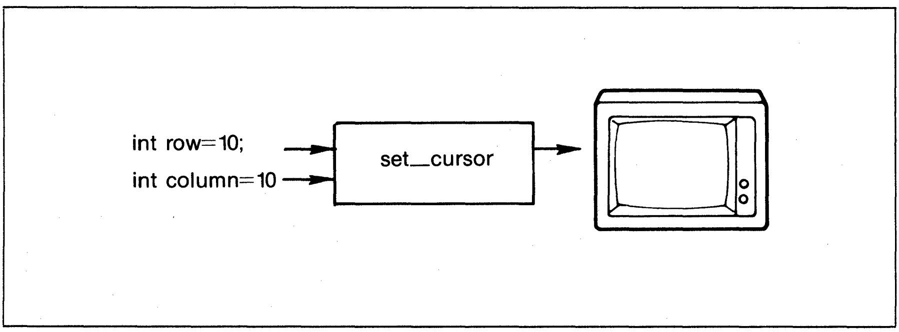
Table 8-1 summarizes the ANSI commands discussed in this chapter.

Table 8-1. *Summary of ANSI Commands*

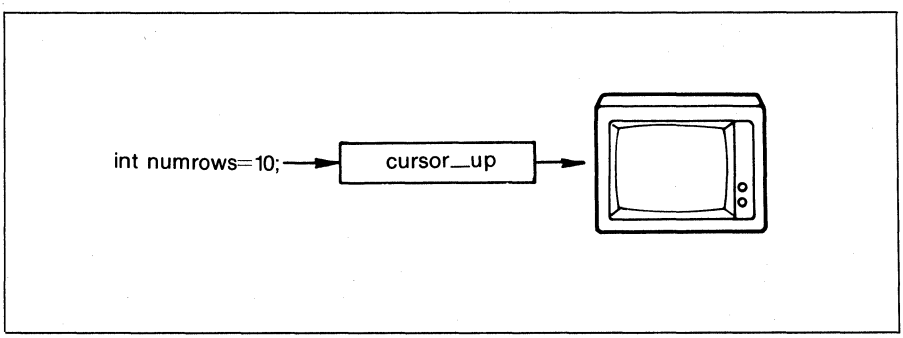
Function	ANSI Command
Set Cursor	ESC[#;#H
Cursor Up	ESC[#A
Cursor Down	ESC[#B
Cursor Forward	ESC[#C
Cursor Back	ESC[#D
Save Cursor	ESC[s
Restore Cursor	ESC[u
Clear Screen	ESC[2J
Clear EOL	ESC[K
Set Graphic	ESC[#;. . .;#m
Set Mode	ESC[=#h
Key Change	ESC[#;#p
Key Define	ESC[#;#;"text";13p

Cursor-Manipulation Routines

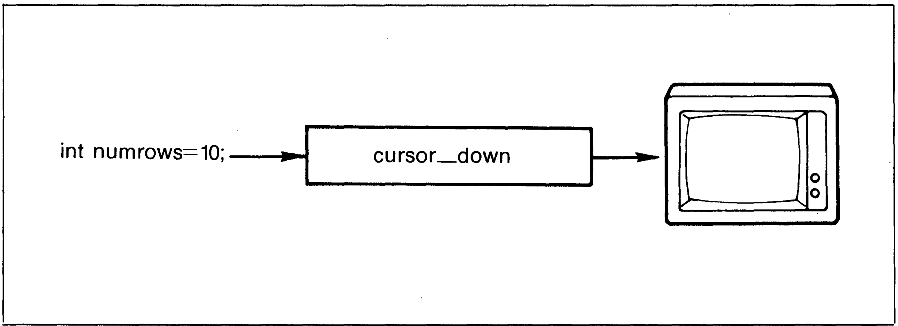
The following routines use the ANSI services to perform cursor manipulation.



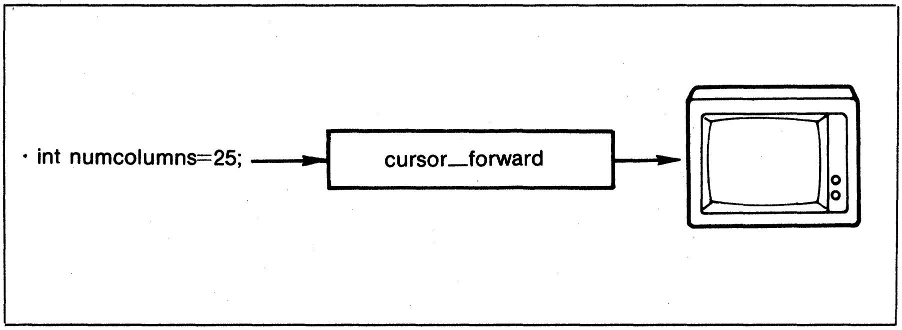

```
/*
 * void set_cursor (row, column)
 *
 * Use the ANSI driver to set the cursor to the row and
 * column position specified.
 *
 * row (in): Desired row number.
 * column (in): Desired column number.
 *
 * set_cursor (10, 10);
 *
 */

void set_cursor (int row, int column)
 {
  printf ("\033[%d;%dH", row, column);
 }
```



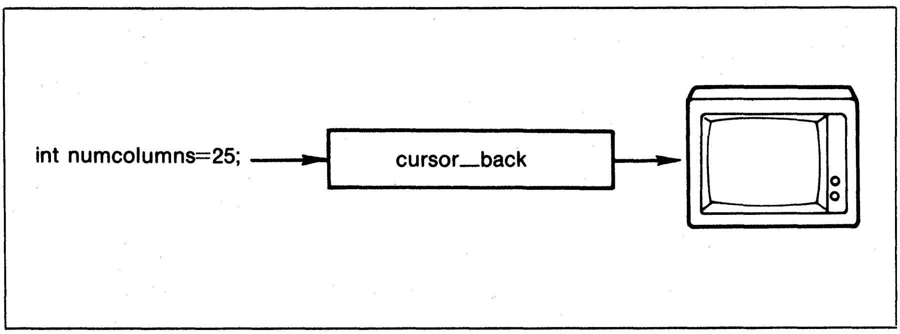
```
/*
 * cursor_up (numrows)
 *
 * Use the ANSI driver to move the cursor up the number
 * of row specified.
 *
 * numrows (in): Number of rows to move the cursor.
 *
 * cursor_up (10);
 *
 * If the cursor reaches the top of the screen, the routine
 * completes.
 *
 */

void cursor_up (int numrows)
 {
  printf ("\033[%dA", numrows);
 }
```



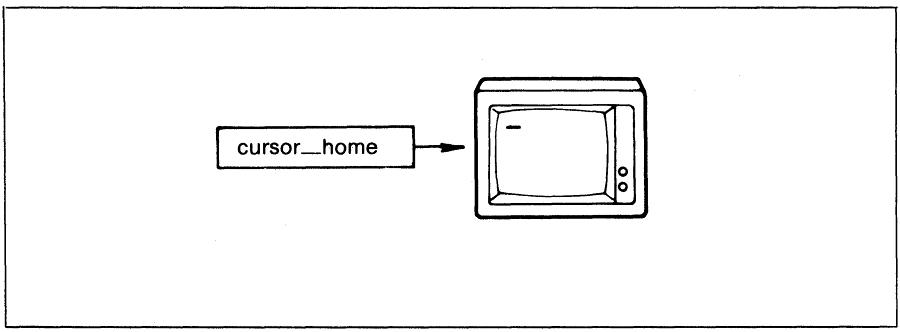

```
/*
 * cursor_down (numrows)
 *
 * Use the ANSI driver to move the cursor down the number
 * of row specified.
 *
 * numrows (in): Number of rows to move the cursor.
 *
 * cursor_down (5);
 *
 * If the cursor reaches the bottom of the screen, the routine
 * completes.
 *
 */

void cursor_down (int numrows)
 {
  printf ("\033[%dB", numrows);
 }
```



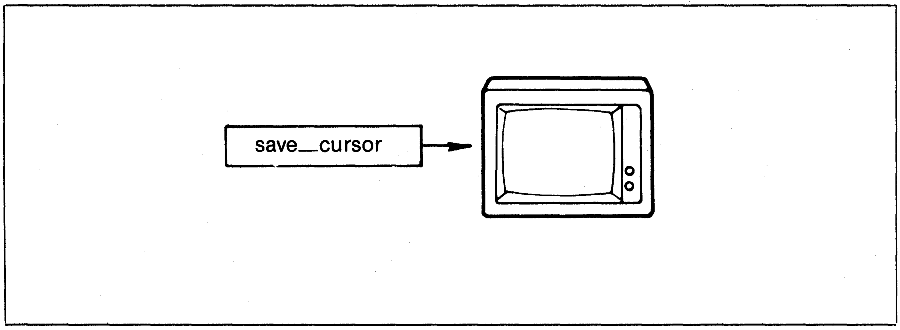

```
/*
 * void cursor_forward (numcolumns)
 *
 * Use the ANSI driver to move the cursor forward the number
 * of columns specified.
 *
 * numcolumns (in): Number of columns to move cursor forward.
 *
 * cursor_forward (10);
 *
 * If the cursor reaches the right side of the screen, the
 * routine completes.
 *
 */

void cursor_forward (int numcolumns)
 {
  printf ("\033[%dC", numcolumns);
 }
```



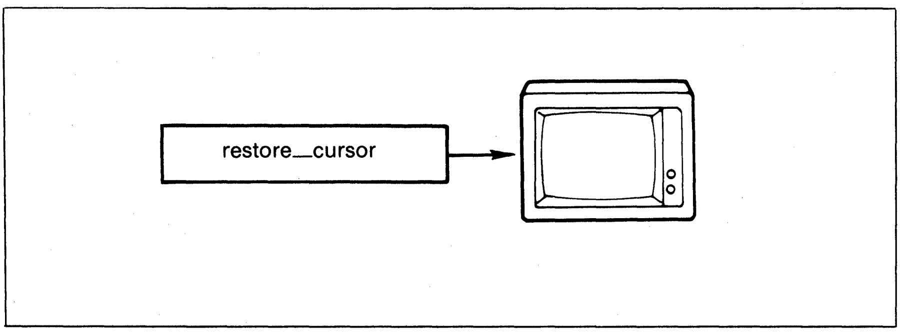
```
/*
 * void cursor_back (numcolumns)
 *
 * Use the ANSI driver to move the cursor backward the number
 * of columns specified.
 *
 * numcolumns (in): Number of columns to move cursor backward.
 *
 * cursor_backward (10);
 *
 * If the cursor reaches the left side of the screen, the
 * routine completes.
 *
 */

void cursor_back (int numcolumns)
 {
  printf ("\033[%dD", numcolumns);
 }
```




```
/*
 * void cursor_home ()
 *
 * Use the ANSI driver to place the cursor in the home position.
 *
 * cursor_home ();
 *
 */

void cursor_home ()
 {
  printf ("\033[H");
 }
```



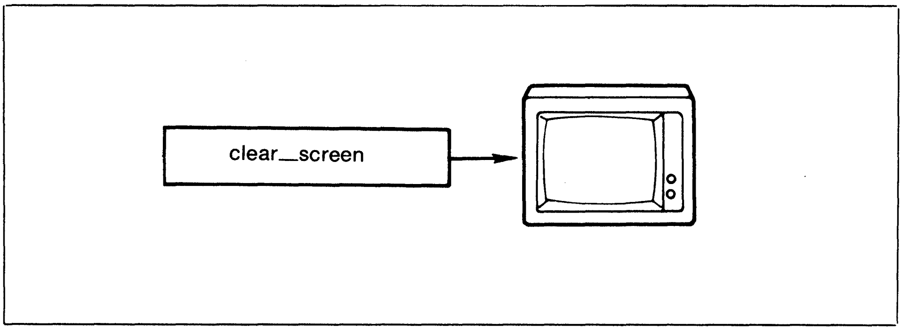

```
/*
 * void save_cursor ()
 *
 * Use the ANSI driver to save the current cursor position for
 * later restoration by restore_cursor.
 *
 * save_cursor ();
 *
 */

void save_cursor ()
 {
  printf ("\033[s\n");
 }
```



```
/*
 * void restore_cursor ()
 *
 * Use the ANSI driver to restore the cursor position that
 * was saved by a previous call to save_cursor.
 *
 * restore_cursor ();
 *
 */

void restore_cursor ()
 {
  printf ("\033[u\n";;
 }
```

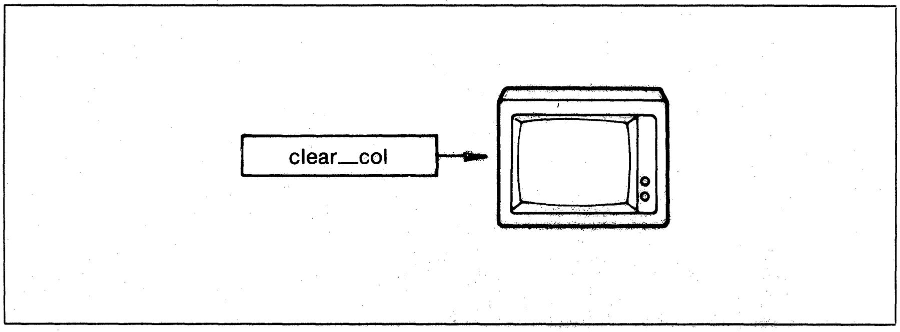
Erasing

The following set of routines uses the ANSI commands to erase the entire screen display, and to erase the screen display from the current cursor position to the end of the line:




```
/*
 * void clear_screen ()
 *
 * Use the ANSI driver to clear the current screen contents
 * placing the cursor in the home position.
```

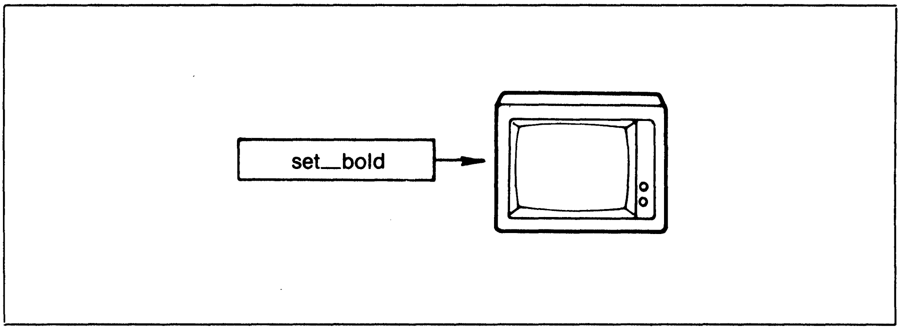
```
 *
 * clear_screen ();
 *
 */

void clear_screen ()
 {
  printf ("\033[2J");
 }
```



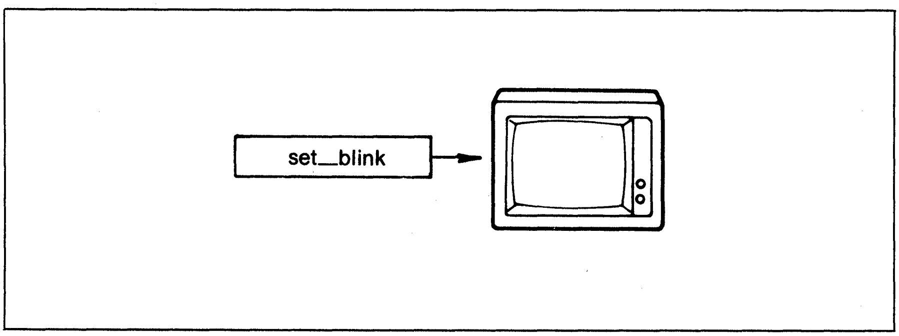

```
/*
 * void clear_eol ()
 *
 * Use the ANSI driver to clear the current line from the
 * current cursor position.
 *
 * clear_eol ();
 *
 */

void clear_eol ()
 {
  printf ("\033[K");
 }
```

Screen Attributes

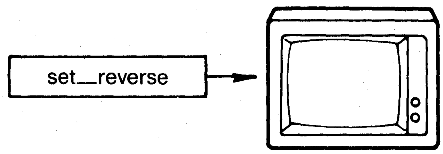
The following routines use the ANSI services to modify the video display and the output attributes of data on the screen:



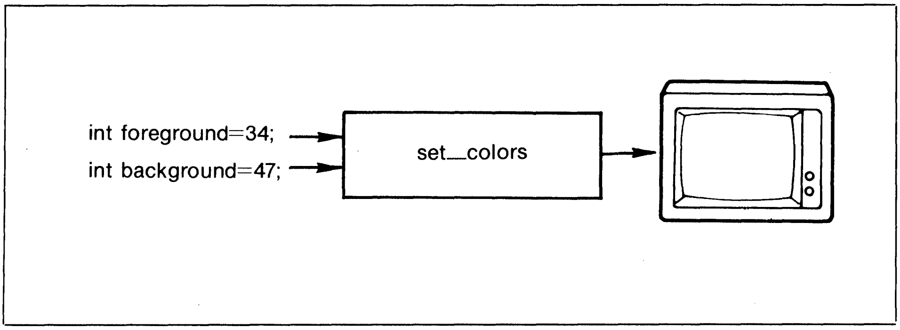

```
/*
 * void set_bold (command)
 *
 * Use the ANSI driver to enable or disable bold text display.
 *
 * command (in): If command is 1, bolding is enabled, otherwise
 *               bolding is disabled.
 *
 * set_bold (1);  printf ("BOLD TEXT");
 * set_bold (0);  printf ("NORMAL TEXT");
 *
 */

void set_bold (int command)
 {
  printf ("\033[%dm", (command) ? 1: 0);
 }
```



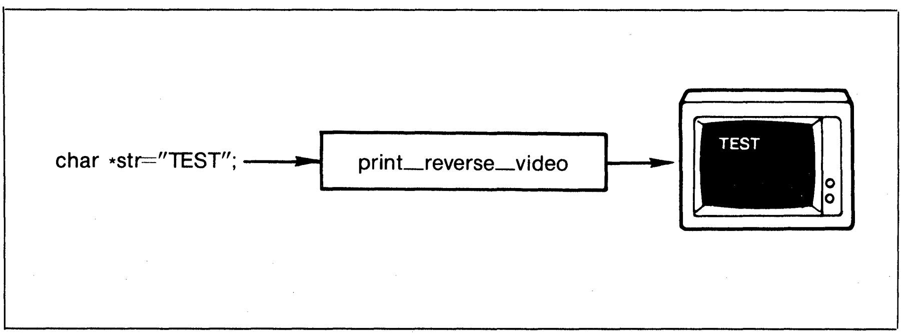
```
/*
 * void set_blink (command)
 *
 * Use the ANSI driver to enable or disable blinking text display.
 *
 * command (in): If command is 1, blinking is enabled, otherwise
 *               blinking is disabled.
 *
 * set_blink (1);  printf ("BLINKING TEXT");
 * set_blink (0);  printf ("NORMAL TEXT");
 *
 */

void set_blink (int command)
 {
  printf ("\033[%dm", (command) ? 5: 0);
 }
```



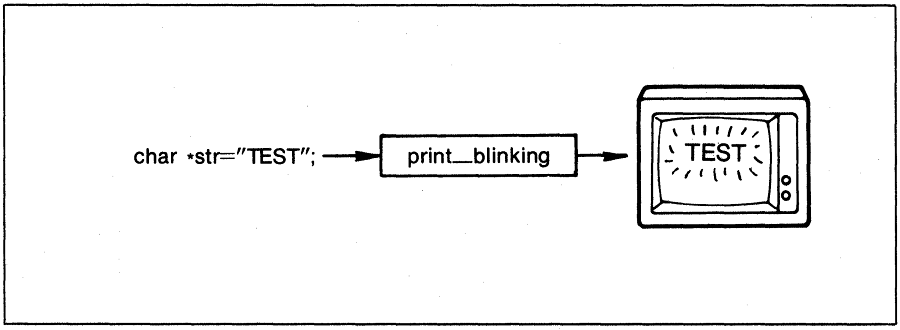

```
/*
 * void set_reverse (command)
 *
 * Use the ANSI driver to enable or disable reverse video text display.
 *
 * command (in): If command is 1, reverse video is enabled, otherwise
 *               reverse video is disabled.
 *
 * set_reverse (1);  printf ("REVERSED TEXT");
 * set_reverse (0);  printf ("NORMAL TEXT");
 *
 */

void set_reverse (int command)
 {
  printf ("\033[%dm", (command) ? 7: 0);
 }
```



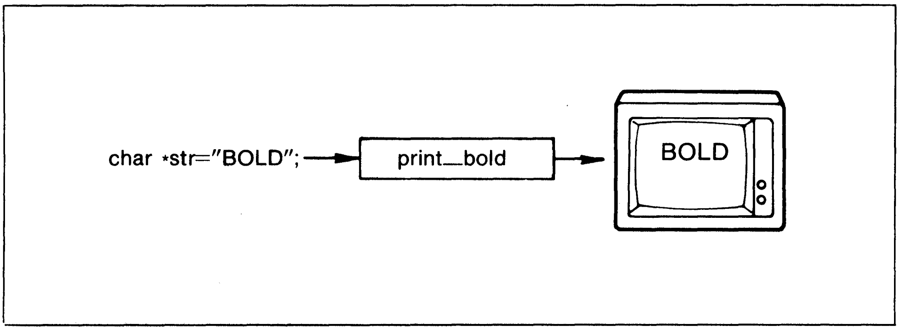

```
/*
 * void set_colors (foreground, background)
 *
 * Use the ANSI driver to set the foreground and background
 * colors for text display.
 *
 * foreground (in): Desired foreground color:
 *      30 black  31 red     32 green  33 yellow
 *      34 blue   35 magenta 36 cyan   37 white
 * background (in): Desired background color:
 *      40 black  41 red     42 green  43 yellow
 *      44 blue   45 magenta 46 cyan   47 white
 *
 * set_colors (31, 47);
 *
 */

void set_colors (int foreground, int background)
 {
  if ((foreground >= 30) && (foreground <= 37))
    if ((background >= 40) && (background >= 47))
      printf ("\033[%d;%dm", foreground, background);
 }
```



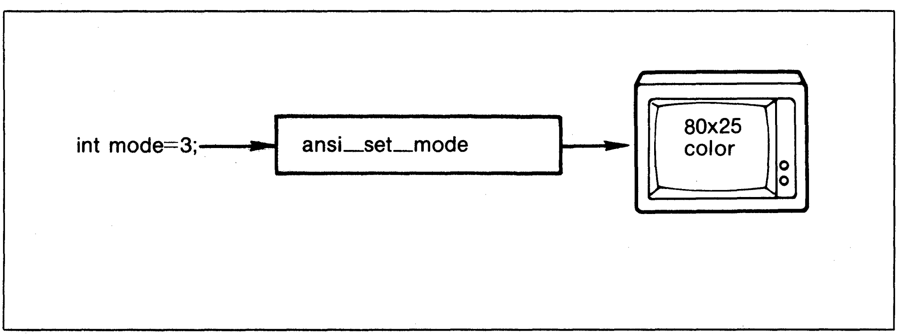
```
/*
 * void print_reverse_video (string)
 *
 * Use the ANSI driver to print the string specified in reverse
 * video.
 *
 * string (in): Character string to display.
 *
 * print_reverse_video ("TEST STRING");
 *
 */

void print_reverse_video (char *string)
 {
  printf ("\033[7m%s\033[0m", string);
 }
```




```
/*
 * void print_blinking (string)
 *
 * Use the ANSI driver to print the string specified blinking.
 *
 * string (in): Character string to display.
 *
 * print_blinking ("TEST STRING");
 *
 */

void print_blinking (char *string)
 {
  printf ("\033[5m%s\033[0m", string);
 }
```




```
/*
 * void print_bold (string)
 *
 * Use the ANSI driver to print the string specified bold.
 *
 * string (in): Character string to display.
 *
 * print_bold ("TEST STRING");
 *
 */

void print_bold (char *string)
 {
  printf ("\033[1m%s\033[0m", string);
 }
```

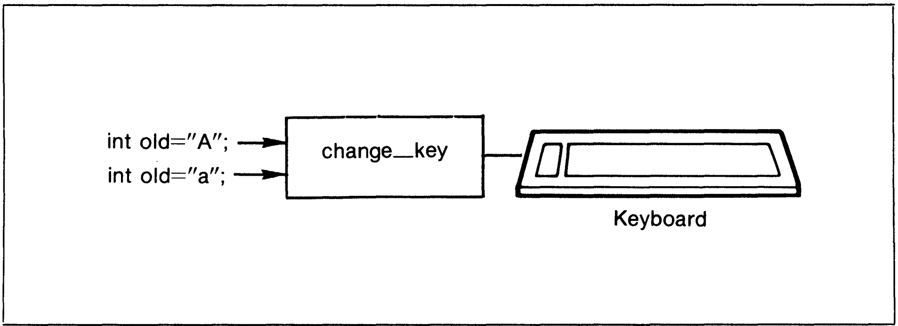


```
/*
 * void ansi_set_mode (mode)
 *
 * Use the ANSI driver to select the mode specified.
 *
 * mode (in): Desired video mode:
 *            0 40 x 25 bw      1 40 x 25 color
 *            2 80 x 25 bw      3 80 x 25 color
 *            4 320 x 200 color 5 320 x 200 bw
 *            6 640 x 200 bw    7 wrap at end of line
 *
 * ansi_set_mode (3);
 *
 */

void ansi_set_mode (int mode)
 {
  if ((mode >= 0) && (mode <= 7))
    printf ("\033[=%dh", mode);
 }
```

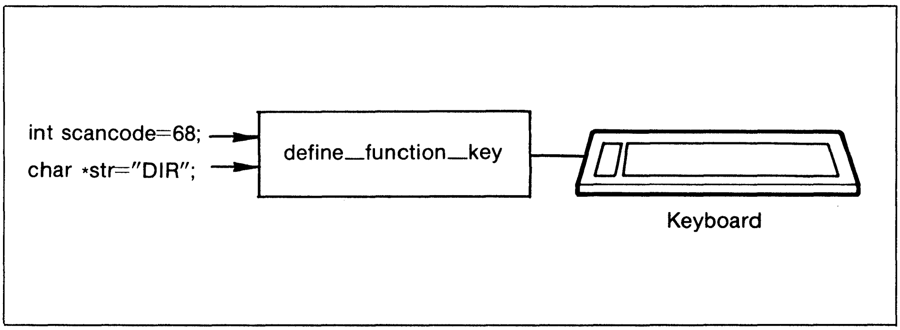
Keyboard Reassignment

The routines presented in this section enable you to trap data from the keyboard and to replace that data with either a different keystroke or a series of keystrokes. This means that the ANSI driver enables you to redefine a DOS function key (such as F10) with a DOS command (such as DIR). Once defined, each time you press the F10 key from the DOS prompt, DOS will respond with the DIR command, as shown here:

```
C> DIR

 Volume in drive C is S
 Directory of  C:\TURBOC

.            <DIR>     11-28-87   8:05p
..           <DIR>     11-28-87   8:05p
ALLOC    H         896  6-03-87   1:00a
ASSERT   H         275  6-03-87   1:00a
BIOS     H         527  6-03-87   1:00a
CONIO    H         517  6-03-87   1:00a
CTYPE    H        1345  6-03-87   1:00a
DIR      H        1222  6-03-87   1:00a
DOS      H        7316  6-03-87   1:00a
ERRNO    H        2648  6-03-87   1:00a
FCNTL    H         991  6-03-87   1:00a
      9 File(s)   1058816 bytes free
```




```
/*
 * void change_key (old, new)
 *
 * Use the ANSI driver to redefine the ASCII code associated
 * with the key specified.
 *
 * old (in): ASCII code of key to redefine.
 * new (in): ASCII code of new key.
 *
 * change_key ('A', 'a');
 *
 */

void change_key (int old, int new)
 {
   printf ("\033[%d;%dp", old, new);
 }
```




```
/*
 * void define_function_key (scancode, string)
 *
 * Use the ANSI driver to associate a character string with
 * a DOS function key.
 *
```

```
 * scancode (in): Scan code of the key to reassign.
 * string (in): String to associate with the key.
 *
 * define_function_key (68, "DIR");   (68 is F10)
 *
 */

void define_function_key (int scancode, char *string)
 {
  printf ("\033[0;%d;\"%s\";13p", scancode, string);
 }
```

The ANSI functions are indeed quite powerful and quite convenient. However, to avoid problems, be sure that the user has installed the ANSI driver. If your program uses the ANSI escape sequences to perform I/O operations and the ANSI driver is not installed, the screen will contain a strange combination of characters. If this occurs, be sure the user installs the ANSI driver as previously explained.

Since Turbo C provides several powerful routines for controlling your output (see Appendix B), you may choose to use the run-time library routines in place of the ANSI routines. In either case, if you program under DOS, it is important that you understand that the ANSI driver capabilities exist.

C H A P T E R

9

File Manipulation

Chapter 5 examined many programs that use the DOS pipe and I/O redirection operators to perform file and stream manipulation. This chapter builds on those programs to enable support for DOS command-line processing and I/O redirection, as shown here:

```
A> DIR | FIRST
A> FIRST FILENAME.TXT
```

In the latter case, many of the programs presented in this chapter will also support DOS wildcard characters by using the routines find_first and find_next, which were presented in Chapter 6.

```
A> FIRST *.*
```

By supporting both command-line arguments and DOS I/O redirection, the programs presented in this chapter will provide you with maximum flexibility.

Understanding find_first and find_next

Before examining the text file-manipulation routines presented later in this chapter, you should first understand how wildcard processing is performed. The following program, LS.C, performs a DOS directory command. You can invoke the program as follows:

```
A> LS                          (List all files by default)
A> LS FILENAME.EXT             (List all C files)
A> LS *.C
```

The program will display all of the information normally displayed by the DOS DIR command. For example, if your directory contains the following,

```
 Volume in drive A is C_FILES
 Directory of  A:\

APPENDIX C       2688  11-04-87  11:55a
ALPHA    C        320   2-15-88   6:58p
ASCIIINT C        562   2-15-88   6:58p
AUXCHAR  C        446   2-15-88   6:58p
AUXWRITE C        596   2-15-88   6:58p
ANSITEST C         50   2-15-88   6:58p
ASETCUR  C        431   2-15-88   6:58p
        7 File(s)    353280 bytes free
```

the command

```
A> LS
```

will display the following:

```
APPENDIX.C      11/04/1987   11:55:44   2688 bytes
ALPHA.C         02/15/1988   18:58:00    320 bytes
ASCIIINT.C      02/15/1988   18:58:00    562 bytes
AUXCHAR.C       02/15/1988   18:58:00    446 bytes
AUXWRITE.C      02/15/1988   18:58:00    596 bytes
ANSITEST.C      02/15/1988   18:58:00     50 bytes
ASETCUR.C       02/15/1988   18:58:00    431 bytes
```

The program begins by passing either the contents of *argv*[1] or the character string *.* to the routine find_first. If find_first locates a matching file, the program displays the related data. Otherwise, the program terminates.

If find_first successfully locates a file, the program invokes the routine find_next to locate the next file matching the original search specification. If a new file is found, the program displays the file information, and this process repeats. Otherwise, the program terminates. The following code implements LS:

```
main (argc, argv)
  int argc;
  char *argv[];
 {
  int day, month, year, hour, minute, second, status;
  long size;
  char filename[13];

  int find_first (char *, char *, int, int *, int *, int *
                  int *, int *, int *, long int *);
  int find_next (char *, int, int *, int *, int *,
                 int *, int *, int *, long int *);

  if (argc < 2)
    status = find_first ("*.*", filename, 0, &hour, &minute,
                         &second, &day, &month, &year, &size);
  else
    status = find_first (argv[1], filename, 0, &hour, &minute,
                         &second, &day, &month, &year, &size);

  while (status == 0)
   {
    printf ("%-15s %02d/%02d/%d\t%02d:%02d:%02d %91d bytes\n",
         filename, month, day, year, hour, minute, second, size);
    status = find_next (filename, 0, &hour, &minute, &second,
                        &day, &month, &year, &size);
   }
 }
```

The program ATTR.C enhances the DOS ATTRIB command, which sets or displays a file's attributes. This program supports the attributes shown in Table 9-1.

Table 9-1. Attributes Supported by ATTR.C Program

Attribute	Meaning
0	Normal
1	Read-only
2	Hidden
4	System
8	Volume label
16	Subdirectory
32	Archive

Invoke the ATTR.C program as follows:

```
A> ATTR *.*          (Display file attributes)
A> ATTR 1 *.C        (Set C files to read-only)
A> ATTR 0 *.*        (Set files to normal attributes)
```

If you use ATTR to set a file to read-only and later try to delete or modify the file, DOS will display the following:

```
Access Denied
```

The following code implements ATTR.C:

```
main (argc, argv)
  int argc;
  char *argv[];
 {
  int status, attributes, day, month, year, hour, minute, second;
  long size;
  char filename[13];

  if (argc == 1)
   {
    printf ("ATTR invalid usage: ATTR [attribute] FILESPEC\n");
    exit (1);
   }
  else if (argc == 2)
    status = find_first (argv[1], filename, 0, &hour, &minute, &second,
                         &day, &month, &year, &size);
  else
   {
    if (ascii_to_int (argv[1], &attributes) == -1)
      {
       printf ("ATTR invalid attribute %s\n", argv[1]);
       exit (1);
      }
    status = find_first (argv[2], filename, 0, &hour, &minute, &second,
                         &day, &month, &year, &size);
   }

  while (status == 0)
    {
     if (argc == 2)
       printf ("%-15s %d\n", filename, get_file_attributes (filename));
     else if (set_file_attributes (filename, attributes) == -1)
         printf ("ATTR Error modifying %s\n", filename);
     status = find_next (filename, 0, &hour, &minute, &second,
                         &day, &month, &year, &size);
    }
 }
```

The program STAMP.C allows you to set a date and time stamp for a file (or files) to the current system date. Invoke the program as follows:

```
A> STAMP FILENAME.EXT
A> STAMP *.C
A> STAMP *.*
```

The following code implements STAMP.C:

```
main (argc, argv)
  int argc;
  char *argv[];
 {
  int file, status, fday, fmonth, fyear, fhour, fminute, fsecond;
  int sys_dow, sys_day, sys_month, sys_year, sys_hour,
      sys_minute, sys_second, sys_hundredths;

  long int size;

  char filename[13];

  if (argc < 2)
    {
     printf ("STAMP invalid usage: STAMP FILESPEC\n");
     exit (1);
    }

  status = find_first (argv[1], filename, 0, &fday, &fmonth,
                   &fyear, &fhour, &fminute, &fsecond, &size);

  get_time (&sys_hour, &sys_minute, &sys_second, &sys_hundredths);
  get_date (&sys_day, &sys_month, &sys_year, &sys_dow);

  while (status == 0)
    {
     file = open_file (filename, 1, &status);
     if (status == -1)
       printf ("STAMP error modifying %s\n", filename);

     status = set_file_datetime (file, sys_day, sys_month, sys_year,
                                 sys_hour, sys_minute, sys_second);
     if (status == -1)
       printf ("STAMP error modifying %s\n", filename);

     close (file);

     status = find_next (filename, 0, &fday, &fmonth, &fyear,
                         &fhour, &fminute, &fsecond, &size);
   }
 }
```

The routines find_first and find_next add tremendous flexibility to your programs. Each of these routines will be used extensively throughout this chapter.

File-Manipulation Routines

The following utility programs deal exclusively with file manipulation that is based on command-line arguments. The first program, DISPLAY.C, enhances the functional capabilities of the DOS TYPE command by supporting wildcard characters and multiple command-line arguments, as shown here:

```
A> DISPLAY *.*
A> DISPLAY TEST.C TEST.H DISPLAY.C
```

The following code implements DISPLAY.C:

```
main (int argc, char *argv[])
 {
  int i, status, day, month, year, hour, minute, second;
  long size;
  char filename[13], buffer[132];

  FILE *fopen (), *fp;

  int find_first (char *, char *, int, int *, int *, int *,
                  int *, int *, int *, long int *);
  int find_next (char *, int, int *, int *, int *,
                 int *, int *, int *, long int *);

  i = 1;

  if (argc == 1)
    printf ("DISPLAY invalid usage: DISPLAY FILESPEC [...]\n");
  else
   do {
    status = find_first (argv[i++], filename, 0, &hour, &minute,
                         &second, &day, &month, &year, &size);

    while (status == 0)
      {
       if (! (fp = fopen (filename, "r")))
```

```
        printf ("DISPLAY error opening %s\n", filename);
     else
        while (fgets (buffer, sizeof(buffer), fp))
          fputs (buffer, stdout);

     fclose (fp);

     status = find_next (filename, 0, &day, &month, &year,
                         &hour, &minute, &second, &size);
    }
  }
 while (i < argc);
}
```

The program FILECOPY uses the DOS low-level file-manipulation routines presented in Chapter 6 to copy the contents of the first file specified to the second, as shown here:

```
A> FILECOPY SOURCE.EXT TARGET.EXT
```

The program does not support DOS wildcard characters. The following code implements FILECOPY.C:

```
main (int argc, char *argv[])
 {
  int source_file, target_file, status, num_bytes;

  char buffer[255];

  int open_file (char *, int, int *);
  int create_file (char *, int, int *);
  int read_file (int, char *, int, int *);
  int write_file (int, char *, int, int *);
  int close_file (int);

  if (argc < 3)
   {
    printf ("FILECOPY invalid usage: FILECOPY SOURCE TARGET\n");
    exit (1);
   }
  else
   {
    source_file = open_file (argv[1], 0, &status);
    if (status == -1)
      {
       printf ("FILECOPY error opening %s\n", argv[1]);
       exit (1);
      }
```

```
    }

  target_file = create_file (argv[2], 0, &status);
  if (status == -1)
    {
     printf ("FILECOPY error opening %s\n", argv[2]);
     exit (1);
    }

  while (num_bytes = read_file (source_file, buffer,
                      sizeof(buffer), &status))
    {
     if (status == -1)
       {
        printf ("FILECOPY error reading %s\n", argv[1]);
        exit (1);
       }

     write_file (target_file, buffer, num_bytes, &status);
     if (status == -1)
       {
        printf ("FILECOPY error writing %s\n", argv[1]);
        exit (1);
       }
    }

      close_file (source_file);
      close_file (target_file);
    }
```

Utility Programs

The following programs help complete your library of DOS file-manipulation routines. Each program presented in this section supports both command-line arguments and DOS I/O redirection. As such, these programs maximize your command-line flexibility.

The first program, MORE.C, modifies the program presented in Chapter 5 to support command-line processing and I/O redirection, as shown here:

```
A> MORE FILENAME.EXT
A> MORE *.*
A> DIR | MORE
```

The program begins by examining its command-line parameters. If none are present, MORE assumes that its input is redirected I/O, as shown here:

```
A> DIR | MORE
```

If the user has instead specified a file,

```
A> MORE FILENAME.EXT
```

MORE uses the specified file. The following code implements MORE.C:

```
#include <stdio.h>

#define lines_per_page 24

main (int argc, char *argv[])
 {
  FILE *file, *fopen();

  int status, i = 1;
  int hour, minute, second, day, month, year;
  long int size;

  char filename[13];

  void show_file (FILE *);
  int find_first (char *, char *, int, int *, int *, int *,
                  int *, int *, int *, long int *);
  int find_next (char *, int, int *, int *, int *,
                 int *, int *, int *, long int *);

  if (argc == 1)
    show_file (stdin);
  else
    {
     do {
        status = find_first (argv[i], filename, 0, &hour, &minute,
                             &second, &day, &month, &year, &size);
```

```
        while (status == 0)
         {
          if (! (file = fopen (filename, "r")))
            printf ("MORE error opening %s\n", filename);
          else
            show_file (file);
          fclose (file);
          status = find_next (filename, 0, &hour, &minute,
                              &second, &day, &month, &year, &size);
         }
       }
     while (++i < argc);
   }
 }


  void show_file (FILE *file)
   {
    char line[132];

    int line_number = 0;

  while (fgets(line, sizeof(line), file))
    if (++line_number % lines_per_page)
      fputs (line, stdout);
    else
      {
       fflush (stdout);
       fputs ("--MORE--\n", stdout);
       fflush (stdout);
       bioskey (0);
      }
 }
```

In a similar manner, the program LAST.C displays the last ten lines of a file or redirected input, as shown here:

```
A> LAST FILENAME.EXT
A> TYPE FILENAME.EXT | LAST
A> LAST *.*
```

The following code implements LAST.C:

```
#include <stdio.h>

main (argc, argv)
  int argc;
  char *argv[];
 {
```

```
FILE *file, *fopen();

int i = 1, status, index, hour, minute, second, day, month, year;

long int size;

char *lines[10], filename[13], *malloc();

int last (FILE *, char *[], int);
int find_first (char *, char *, int, int *, int *, int *,
                int *, int *, int *, long int *);
int find_next (char *, int, int *, int *, int *,
               int *, int *, int *, long int *);

/* allocate space for a circular buffer */
for (index = 0; index < 10; index++)
 if (! (lines [index] = malloc (132)))
  {
    printf ("Unable to allocate necessary memory\n");
    exit (1);
  }
 else
   *lines[index] = '\0';

if (argc == 1)
  last (stdin, lines, 0);
else
  {
   do {
      status = find_first (argv[i], filename, 0, &hour, &minute,
                           &second, &day, &month, &year, &size);
      if (status != 0)
        printf ("LAST file not found\n");

      while (status == 0)
       {
        if (! (file = fopen (filename, "r")))
          printf ("LAST error opening %s\n", filename);
        else
           last (file, lines, 0);
        fclose (file);
        status = find_next (filename, 0, &hour, &minute,
                           &second, &day, &month, &year, &size);
       }
     }
   while (++i < argc);

  }
 }
```

In the opposite manner, the program FIRST.C displays the first *n* lines of a file or redirected input, as shown here:

```
A> FIRST 100 FILENAME.EXT
A> FIRST *.*
A> TYPE FILENAME.EXT | FIRST
```

The following code implements FIRST.C:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  int stop_line = 10;   /* number of lines to display */

  int i = 2, done = 0, status, index, hour, minute, second,
      day, month, year;

  long int size;

  char filename[13];

  FILE *fopen(), *file;

  void first (FILE *, int);
  int find_first (char *, char *, int, int *, int *, int *,
                  int *, int *, int *, long int *);
  int find_next (char *, int, int *, int *, int *,
                 int *, int *, int *, long int *);
  int ascii_to_int (char *, int *);

  if (argc == 1)
   {
    first (stdin, stop_line);    /* user entered FIRST */
    done = 1;
   }

  else if (argc == 2)            /* FIRST value or FIRST file */
   {
    if (ascii_to_int (argv[1], &stop_line) == -1)
      {
        stop_line = 10;
        i = 1;
      }
    else
     {
      first (stdin, stop_line);
      done = 1;
     }
   }
  else if (argc > 2)     /* FIRST value file or FIRST file file */
   {
    if (ascii_to_int (argv[1], &stop_line) == -1)
      {
        stop_line = 10;
        i = 1;
      }
   }

  if (! done)
    do {
        status = find_first (argv[i], filename, 0, &hour, &minute,
```

```
                            &second, &day, &month, &year, &size);
        if (status != 0)
          printf ("FIRST file not found\n");

        while (status == 0)
         {
          if (! (file = fopen (filename, "r")))
            printf ("FIRST error opening %s\n", filename);
          else
             first (file, stop_line);
          fclose (file);
          status = find_next (filename, 0, &hour, &minute,
                        &second, &day, &month, &year, &size);
         }
       }
     while (++i < argc);
 }


void first (file, stop_line)
  FILE *file;
  int stop_line;
 {
  int count = 0;        /* current line number */

  char line[132];

  while (fgets (line, sizeof(line), file) && (++count <= stop_line))
     fputs (line, stdout);
 }
```

The program FINDSTR.C displays each occurrence of a specified string either in a file (or files) or in redirected input, as shown here:

```
A> FINDSTR ARIZONA STATES.LST
A> FINDSTR DOS *.*
A> TYPE TEST.PAS | FINDSTR begin
```

The program REPLACE.C replaces each occurrence of a word with a second word in either a file or redirected input, as shown here:

```
A> REPLACE BEGIN begin TEST.PAS NEWFILE.EXT
A> TYPE TEST.PAS | REPLACE BEGIN begin
```

Note that REPLACE does not perform wildcard processing. The following code implements REPLACE.C:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  char line[132];

  int location, len;

  FILE *fopen (), *infile, *outfile;

  int remove_substring (char *, char *);
  int insert_string (char *, char *, int, int);
  int next_str_occurrence (char *, char *, int);

  if (argc < 3)
    printf ("invalid usage: REPLACE TARGET NEW_WORD OLDFILE NEWFILE\n");

  else if (argc == 3)
   {
    infile = stdin;
    outfile = stdout;
   }

  else if (argc == 4)
   {
    if (! (infile = fopen (argv[3], "r")))
       {
        printf ("REPLACE error opening %s\n", argv[3]);
        exit (1);
       }
    outfile = stdout;
   }

  else if (argc == 5)
   {
    if (! (infile = fopen (argv[3], "r")))
       {
        printf ("REPLACE error opening %s\n", argv[3]);
        exit (1);
       }

    if (! (outfile = fopen (argv[4], "w")))
       {
        printf ("REPLACE error opening %s\n", argv[4]);
        exit (1);
       }
```

```
 }

 len = string_length (argv[2]);

 while (fgets (line, 132, infile))
   {
      if ((location = index (argv[1], line)) != -1)
        do
          {
            remove_substring (argv[1], &line[location]);
            insert_string (argv[2], line, location, sizeof(line));
          }
         while ((location = next_str_occurrence (argv[1], line,
           location + len)) != -1);

        fputs (line, outfile);
    }
}
```

The program TAB.C enables you to precede lines of a file or redirected input, as shown here:

```
A> TAB FILENAME.EXT NEWFILE.EXT
A> TAB 25 FILENAME.EXT NEWFILE.EXT
A> DIR | TAB 7
```

The following code implements TAB.C:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  FILE *fopen (), *infile, *outfile;

  int spaces = 7;        /* number of spaces to insert */

  int i = 2;

  char line[132];

  int ascii_to_int (char *, int *);
  int pad_string (char *, int, int);

  infile = stdin;
  outfile = stdout;

  if (argc > 1)          /* see if user specified a valid number */
   {
    if (ascii_to_int (argv[1], &spaces) == -1)
     {
      spaces = 7;
```

```
      i = 1;
    }

  if (*argv[i])
    if (! (infile = fopen (argv[i], "r")))
      {
       printf ("TAB error opening %s\n", argv[i]);
       exit (1);
      }

  if (*argv[i] && *argv[i+1])
    if (! (outfile = fopen (argv[i+1], "w")))
      {
       printf ("TAB error opening %s\n", argv[i+1]);
       exit (1);
      }
 }

while (fgets (line, 132, infile))
 {
   if (pad_string (line, spaces, sizeof (line)) == 1)
     {
      printf ("%c Line exceeds %d characters\n", 7, sizeof (line));
      break ;
     }

   fputs (line, outfile);
 }
}
```

The programs EXTRACT.C and REMOVE.C enable you to select or remove various portions of a file or redirected input. The first, EXTRACT.C, writes selected lines of a file (or redirected input) to a second file or to the screen, as follows:

```
A> EXTRACT 0 50 TEST.C TEST.NEW
A> DIR | EXTRACT 0 25
```

Assuming that the file C.DAT contains the following,

```
1 AAAA
2 BBBB
3 CCCC
4 DDDD
5 EEEE
6 FFFF
7 GGGG
8 HHHH
9 IIII
```

the command

```
A> EXTRACT 3 5 C.DAT
```

will display the following:

```
3 CCCC
4 DDDD
5 EEEE
```

The following code implements EXTRACT.C:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
```

```
    FILE *fopen (), *infile, *outfile;

    int start_line, stop_line, count;

    char line[132];

    int ascii_to_int (char *, int *);

    infile = stdin;
    outfile = stdout;

    if (argc < 3)
      {
        printf ("EXTRACT invalid usage: EXTRACT # # FILE FILE\n");
        exit (1);
      }

    if (ascii_to_int (argv[1], &start_line) == -1)
      printf ("EXTRACT invalid start line %d\n", argv[1]);

    if (ascii_to_int (argv[2], &stop_line) == -1)
      printf ("EXTRACT invalid stop line %d\n", argv[2]);

    if (argc >= 4)
      if (! (infile = fopen (argv[3], "r")))
        {
          printf ("EXTRACT error opening %s\n", argv[3]);
          exit (1);
        }

    if (argc == 5)
      if (! (outfile = fopen (argv[4], "w")))
        {
          printf ("EXTRACT error opening %s\n", argv[4]);
          exit (1);
        }

    for (count = 1; fgets (line, sizeof(line), infile); count++)
     {
       if (count >= start_line)
         fputs (line, outfile);

       if (count == stop_line)
         break ;
     }
  }
```

In a similar manner, the program REMOVE.C deletes lines from a file (or redirected input) and writes the result to the screen or to a second file, as shown here:

```
A> REMOVE 0 10 FILENAME.EXT NEWFILE.EXT

A> TYPE FILENAME.EXT | REMOVE 0 10  NEWFILE.EXT
```

Given the file C.DAT, the command

```
A> REMOVE 3 5 C.DAT
```

will display the following:

```
1 AAAA
2 BBBB
6 FFFF
7 GGGG
8 HHHH
9 IIII
```

The following code implements REMOVE.C:

```
#include <stdio.h>

main (int argc, char *argv[])
 {
  FILE *fopen (), *infile, *outfile;

  int start_line, stop_line, count;

  char line[132];

  int ascii_to_int (char *, int *);


  infile = stdin;
  outfile = stdout;

  if (argc < 3)
    {
      printf ("REMOVE invalid usage: REMOVE # # FILE FILE\n");
      exit (1);
    }

  if (ascii_to_int (argv[1], &start_line) == -1)
    printf ("REMOVE invalid start line %d\n", argv[1]);

  if (ascii_to_int (argv[2], &stop_line) == -1)
    printf ("REMOVE invalid stop line %d\n", argv[2]);

  if (argc >= 4)
    if (! (infile = fopen (argv[3], "r")))
      {
        printf ("REMOVE error opening %s\n", argv[3]);
```

```
        exit (1);
      }

  if (argc == 5)
    if (! (outfile = fopen (argv[4], "w")))
      {
        printf ("REMOVE error opening %s\n", argv[4]);
        exit (1);
      }

  for (count = 1; fgets (line, sizeof(line), infile); count++)
     if ((count < start_line) || (count > stop_line))
       fputs (line, outfile);
}
```

With Turbo C, developing useful utility programs is quite straightforward. Experiment with the programs in this chapter and you should be able to assemble a library of countless utility programs.

CHAPTER

10

Array Manipulation

Because of the tremendous use of arrays in string manipulation, most Turbo C programmers have a solid foundation from which to build a library of array-manipulation routines. Throughout this text, routines have been as generic as possible. This practice has greatly increased the number of applications that can use the functions without modification of the code of the routine. This chapter examines routines that manipulate arrays. In an effort to limit the duplication of code, a reduction has been made to the amount of coding and testing that must be performed when modifications are made.

Array Considerations

One of the most difficult functions to consider when developing a library of array-manipulation routines is how to deal with different array types. For example, the following routine returns the sum of the values contained in an array of type float:

```
float sum_array (float array[], int num_elements)
 {
  float result = 0.0;

  int i;

  for (i = 0; i < num_elements; i++)
    result += array[i];

  return (result);
 }
```

The array type and value returned from the function are of the type float. Although this routine works for floating-point values, the routine must be duplicated for an array of type int. Although this appears to be a simple fix, remember that Turbo C has many types, including the following:

int	float	char	long int
unsigned int	double	short int	

As a result, you can quickly create several different functions, each of which performs an identical task.

When you create array-manipulation routines, you have three alternatives. The first, which was just discussed, is simply to create duplicate routines for the required type, as shown here:

```
long int sum_array (int array[], int num_elements)
 {
  int result = 0;

  int i;

  for (i = 0; i < num_elements; i++)
    result += array[i];

  return (result);
 }
```

However, the shortcoming of this solution is the proliferation of routines required for different array types.

The second alternative is to develop a routine based on the two user-defined types shown here:

```
typedef int array_type;
typedef int result_type;
```

The array-manipulation routine is now defined as follows:

```
result_type sum_array (array_type array[], int num_elements)
 {
  result_type result = 0;

  int i;

  for (i = 0; i < num_elements; i++)
    result += array[i];

  return (result);
 }
```

To use this routine for an array of type float, change the user-defined types, as shown here:

```
typedef float array_type;
typedef float result_type;
```

This processing restricts duplication of code, but it too has limitations. With each application you must recompile the array-manipulation routines to be sure that the correct types are used. In addition, if your program must use several arrays of differing types, this method supports only one array type.

The third alternative requires the user to specify the array type as a parameter, as shown here:

```
sum_values (array, num_elements, type);
```

In this case, type is defined as

0	char
1	int
2	float
3	long int
4	unsigned int
5	double
6	short int

Rather than passing an array of type int, float, or double to the routine, the user instead passes an array whose type is defined by a union, as shown here:

```
union array_types {
  char cval;
  int ival;
  float fval;
  unsigned int uval;
  double dval;
  short int sval;
 } ;
```

Within the routine, you access the correct type based on the type of variable, as shown here:

```
double sum_array (union array_types array[],
                  int num_elements, int type)
 {
  double result = 0.0;

  int i;

  for (i = 0; i < num_elements; i++)
    switch (type) {
      case 0: result += (double) array[i].cval;
              break;
      case 1: result += (double) array[i].ival;
              break;
      case 2: result += (double) array[i].fval;
              break;
      case 3: result += (double) array[i].uval;
              break;
      case 4: result += (double) array[i].dval;
              break;
      case 5: result += (double) array[i].sval;
              break;
     }

  return (result);
 }
```

The following program uses this routine to display the sum of the values in several types of arrays by using a single array to sum them:

```
main ()
 {
  union array_types a[10], b[10], c[10];

  double sum_value (union array_types *, int, int);

  int i;

  for (i = 0; i < 10; i++)
    {
     a[i].ival = 5;
     b[i].cval = 1;
     c[i].fval = 3.0;
    }

  printf ("int ARRAY %f\n", sum_array (a, 10, 1));
  printf ("char ARRAY %f\n", sum_array (b, 10, 0));
  printf ("float ARRAY %f\n", sum_array (c, 10, 2));
 }
```

The difficulty of this type of routine is that you must now assign values to the correct union members, as shown here:

```
for (i = 0; i < 10; i++)
  {
   a[i].ival = 5;
   b[i].cval = 1;
   c[i].fval = 3.0;
  }
```

This may be an unreasonable requirement to place on all your programs.

Multiple array types can be quite frustrating to Turbo C programmers. The development of your array-manipulation routines is a tradeoff among the following factors:

- Duplication of code for each type
- Code recompilation with each application
- Impact upon code outside of the function (unions)

The routines in the remainder of this chapter are based on the types array_type and return_type. For example, if your array types were of type int, you would simply place the following typedef statement at the beginning of your program:

```
typedef int array_type;
typedef int result_type;
```

If you were using arrays of type float, you would use the following:

```
typedef float array_type;
typedef float result_type;
```

Within your programs, you define your array in terms of these two types:

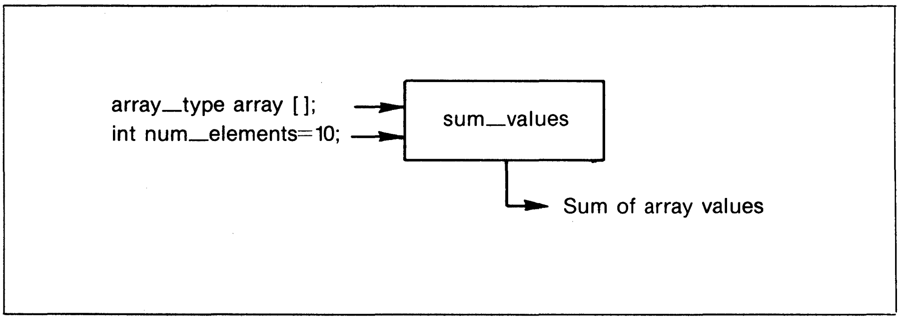
```
typedef float result_type;
typedef float array_type;

main()
  {
    array_type salary[50];
  }
```

If you are building a library of routines, you may want to change the names of each routine to reflect its type, as shown here:

float_sum_values
int_sum_values
double_sum_values

Array-Manipulation Routines

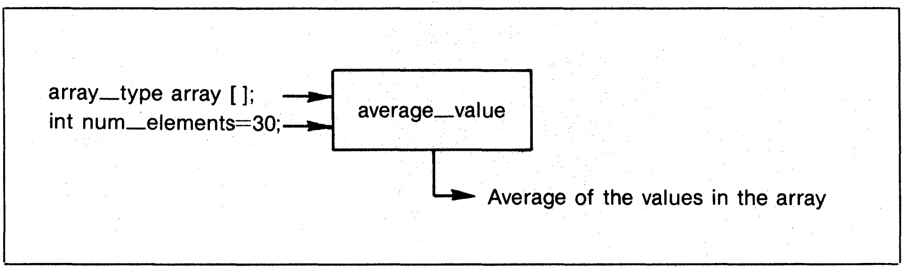
The first routine, sum_array returns the sum of all of the values contained in an array:




```
/*
 * result_type sum_array (array, num_elements);
 *
 * Return the sum of the values in an array.
 *
 * array (in): array containing the values to sum.
 * num_elements (in): number of elements in the array.
 *
 * sum = sum_array (scores, 10);
 *
```

```
 * This routine requires you to define the types result_type
 * and array_type as required depending upon your array type.
 *
 */

result_type sum_array (array_type array[], int num_elements)
 {
  int i;

  result_type result = 0;

  for (i = 0; i < num_elements; i++)
    result += array[i];

  return (result);
 }
```

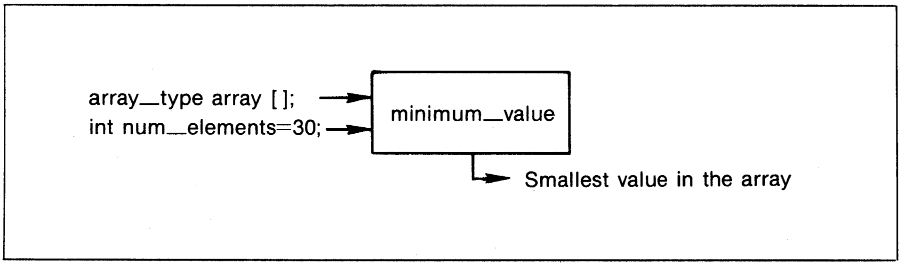
The next routine, average_value, returns the average of the values contained in an array.




```
/*
 * result_type average_value (array, num_elements);
 *
 * Return the average value in an array.
 *
 * array (in): array containing of values to compute the average of.
 * num_elements (in): number of elements in the array.
 *
 * avg = average_value (scores, 10);
 *
 * This routine requires you to define the types result_type
 * and array_type as required depending upon your array type.
 *
 */

result_type average_value (array_type array[], int num_elements)
 {
  int i;
```

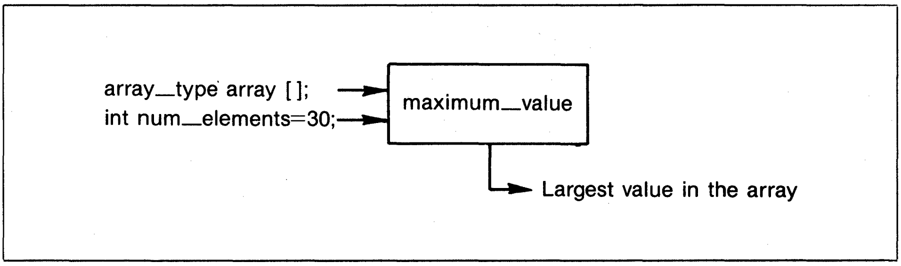
```
  result_type result = 0;

  for (i = 0; i < num_elements; i++)
    result += array[i];

  return (result / num_elements);
 }
```

The routine minimum_value searches the elements of an array and returns the smallest value found, as follows:



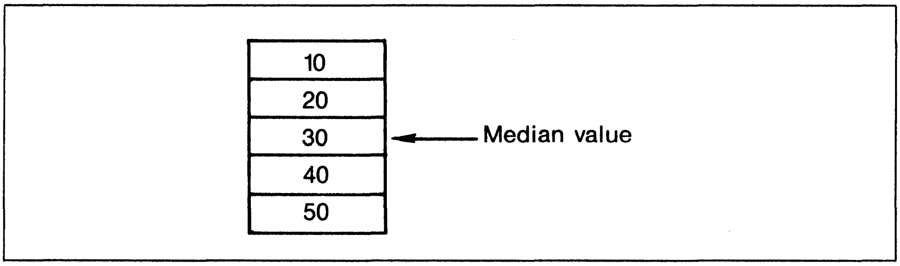

```
/*
 * result_type minimum_value (array, num_elements);
 *
 * Return the smallest value in an array.
 *
 * array (in): array of values to return minimum value from.
 * num_elements (in): number of elements in the array.
 *
 * min = minimum_value (scores, 10);
 *
 * This routine requires you to define the types result_type
 * and array_type as required depending upon your array type.
 *
 */

result_type minimum_value (array_type array[], int num_elements)
 {
  int i;

  result_type minimum = array[0];

  for (i = 1; i < num_elements; i++)
    if (minimum > array[i])
      minimum = array[i];

  return (minimum);
 }
```

The routine maximum_value returns the largest value in an array, as shown here:

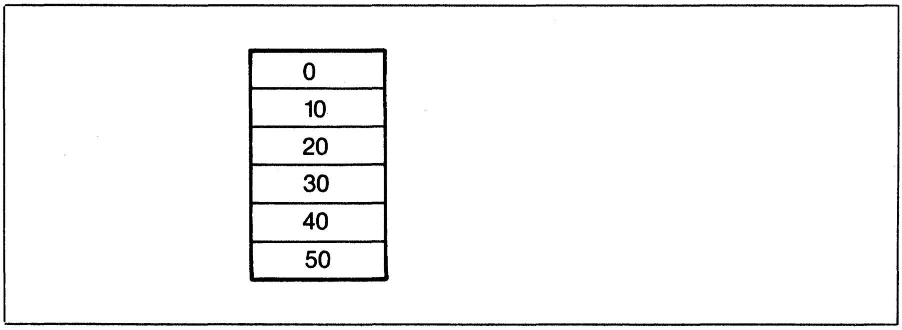



```
/*
 * result_type maximum_value (array, num_elements);
 *
 * Return the largest value in an array.
 *
 * array (in): array of values to return maximum value from.
 * num_elements (in): number of elements in the array.
 *
 * max = maximum_value (scores, 10);
 *
 * This routine requires you to define the types result_type
 * and array_type as required depending upon your array type.
 *
 */

result_type maximum_value (array_type array[], int num_elements)
 {
  int i;

  result_type maximum = array[0];

  for (i = 1; i < num_elements; i++)
    if (maximum < array[i])
      maximum = array[i];

  return (maximum);
 }
```

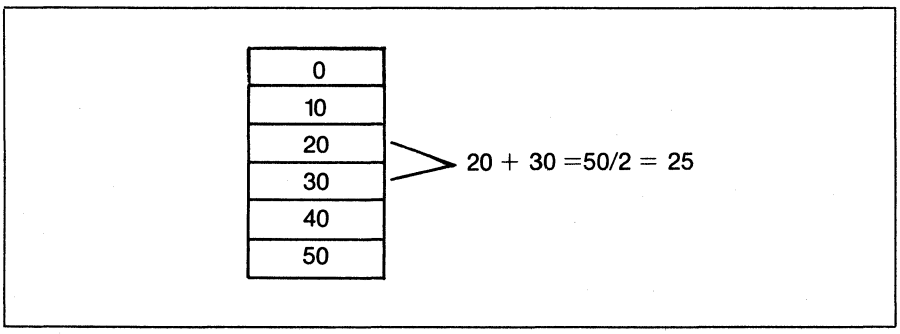
The routine median_value returns the median value (middle value) contained in an array. Given the array



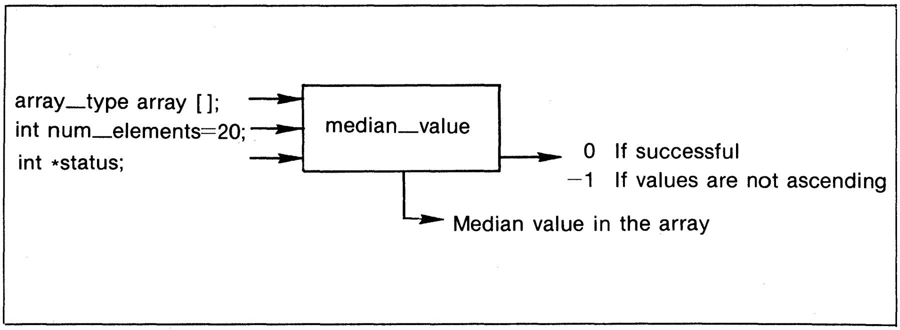
the routine median_value returns the value 30. However, if the array contains an even number of elements,



the routine returns the value 25, as calculated here:

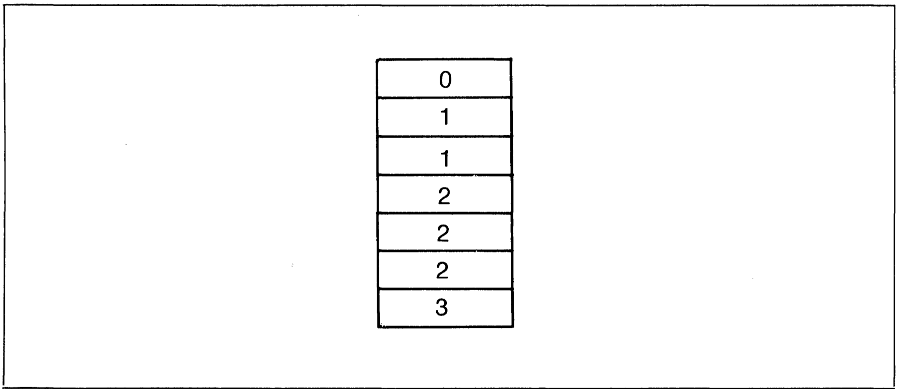


The values must be in ascending order or median_value returns the error status −1.

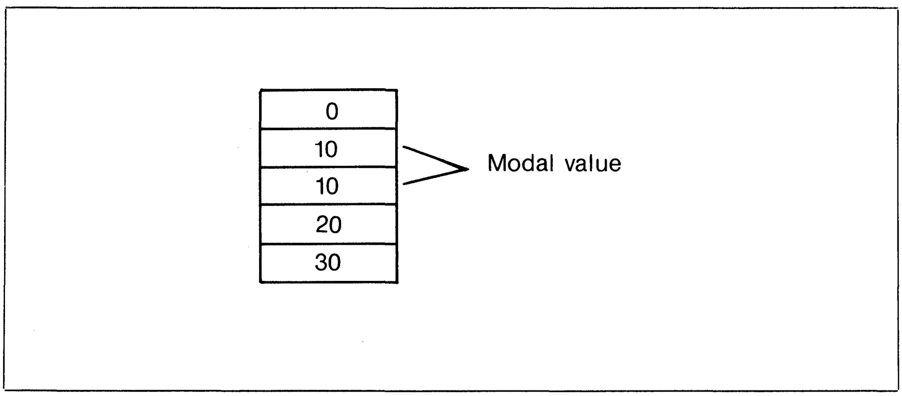



```
/*
 * result_type median_value (array, num_elements, status);
 *
 * Return the largest value in an array.
 *
 * array (in): array of values to return median value from.
 * num_elements (in): number of elements in the array.
 * status (out): 0 if successful, -1 if elements are not ascending.
 *
 * median = median_value (scores, 10);
 *
 * This routine requires you to define the type array_type
 * as required depending upon your array type.
 *
 */

float median_value (array_type array[], int num_elements, int *status)
 {
  int i;

  result_type median;

  *status = 0;

  /* insure array is ascending */
  for (i = 0; i < num_elements-1; i++)
    if (array[i] > array[i+1])
      *status = -1;

  if (! *status)
    if (num_elements % 2)
      median = array[num_elements / 2];
    else
      median = (array[num_elements / 2] +
                array[num_elements - (num_elements / 2) - 1]) / 2;

  return (median);
 }
```

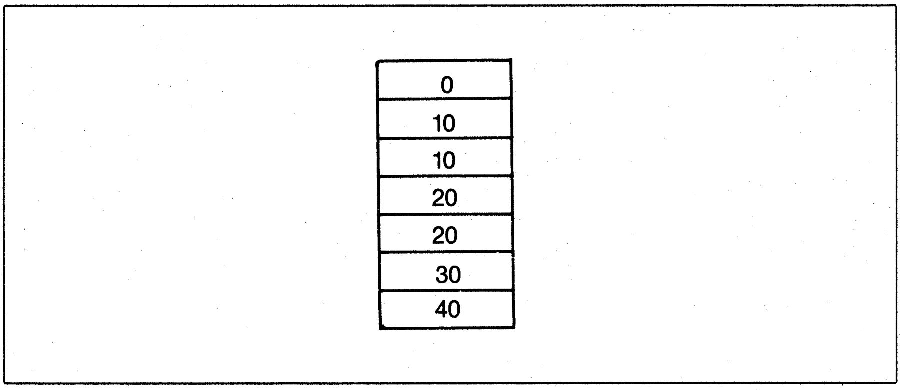
The routine modal_value returns the modal value of an array. The modal value is simply the value that occurs most often in the array. For example, given the following array,



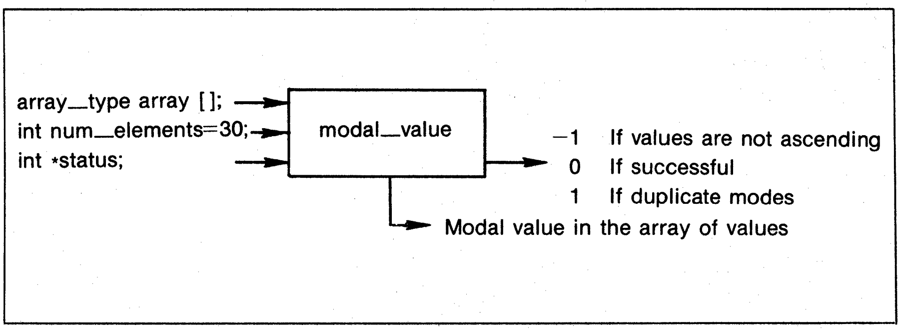
the modal value is 2. As was the case, with the routine median_value, the values of the array must be in ascending order. Given the following array,



the routine modal_value returns the value 10. If duplicate modal values exist,



the routine returns the status value 1.




```
/*
 * result_type modal_value (array, num_elements, status);
 *
 * Return the modal value of an array.
 *
 * array (in): array of values to return modal value from.
 * num_elements (in): number of elements in the array.
 * status (out): 0 if successful, -1 if values not ascending,
 *               1 if duplicate modes
 *
 * mode = modal_value (scores, 10, &status);
 *
 * This routine requires you to define the types result_type
 * and array_type as required depending upon your array type.
 *
 */
```

```
result_type modal_value (array_type array[], int num_elements,
                         int *status)
 {
  int i, current_count, max_count = -1;

  array_type current_value, max_value = 0.0;

  *status = 0;

  /* insure array is ascending */
  for (i = 0; i < num_elements-1; i++)
    if (array[i] > array[i+1])
      *status = -1;

  if (*status != -1)
   {
    i = 0;

    while (i < num_elements)
     {
      current_value = array[i];
      current_count = 0;
      while ((array[i] == current_value) && (i < num_elements))
       {
        current_count++;
        i++;
       }
      if (current_count > max_count)
       {
        max_count = current_count;
        max_value = current_value;
        *status = 0;
       }
      else if (current_count == max_count)
        *status = 1;                        /* duplicate mode */
     }
   }

  return (max_value);
 }
```

Variance and Standard Deviation

Two of the most widely used statistical tools are variance and standard deviation. Statisticians use them to analyze the expected value, or average of a population. For example, if 100 different programs are run on two computers, an expected value can be computed that represents how much faster the first computer is in comparison to the second. Statisticians can use either the variance or standard deviation to determine the accuracy of the expected value by describing the average deviation from the sample mean.

The variance is computed by using the following equation:

$$V = \frac{1}{N} \sum_{i=1}^{N} (D_i - M)^2$$

The standard deviation is computed by taking the square root of the variance, as shown here:

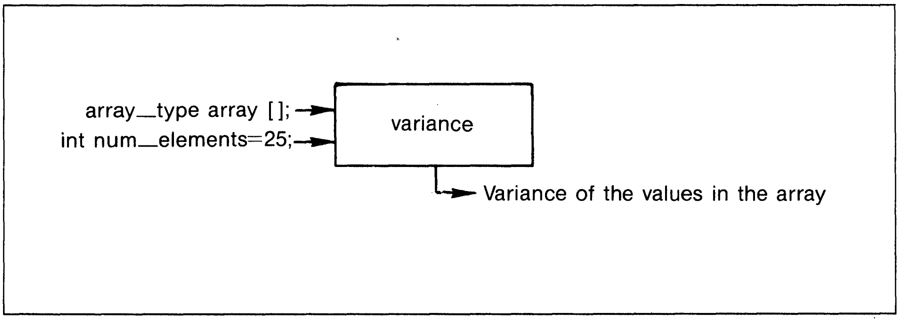
$$std = \sqrt{\frac{V*N}{N-1}}$$

In both equations, N represents the number of elements in the mean of the sample.

The standard deviation is more important than variance because its result is more readily understood. In the example just given, if the first computer averages 3 seconds faster than the second computer, possible values would be as follows:

Expected value:	3 seconds
Variance:	4 $(\text{seconds})^2$
Standard deviation:	2 seconds

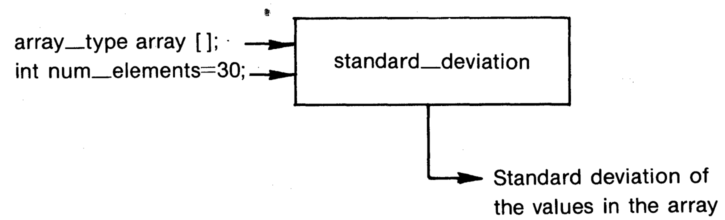
When the variance is calculated, the difference between each element and the mean is squared to produce only positive values. The following routine determines the variance of the values in an array:




```
/*
 * float variance (array, num_elements)
 *
 * Return the variance of values in an array.
 *
 * array (in): array of values to return variance of.
 * num_elements (in): number of elements in the array.
 *
 * var = variance (scores, 10);
 *
 * This routine requires you to define the type array_type
 * as required depending upon your array type.  This routine
 * uses the routine average_value contained in the array library.
 *
 */

float variance (array_type array[], int num_elements)
 {
  int i;

  float sum = 0.0;

  array_type average, average_value (array_type *, int);

  average = average_value (array, num_elements);
```

```
  for (i = 0; i < num_elements; i++)
    sum += (array[i] - average) * (array[i] - average);

  return (sum / num_elements);
}
```

The routine standard_deviation returns the standard deviation of the values in an array:

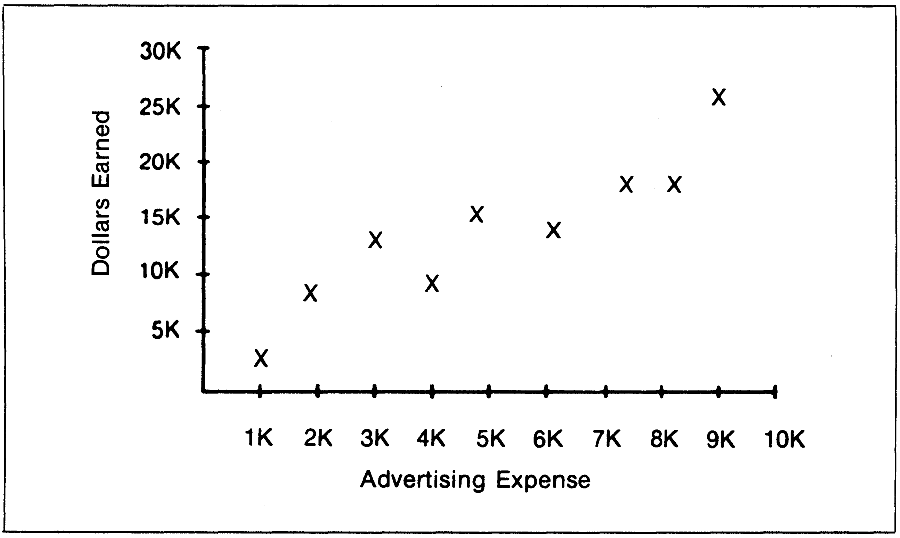



```
/*
 * float standard_deviation (array, num_elements)
 *
 * Return the standard deviation of values in an array.
 *
 * array (in): array of values to return standard deviation of.
 * num_elements (in): number of elements in the array.
 *
 * stddev = standard_deviation (scores, 10);
 *
 * This routine requires you to define the type array_type
 * as required depending upon your array type.  This routine
 * uses the routine variance contained in the array library.
 *
 */
```

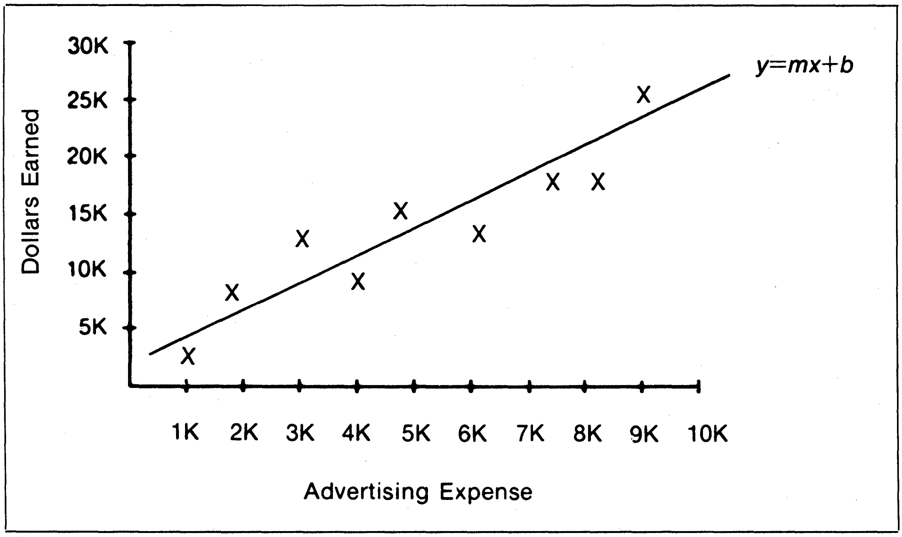
```
#include <math.h>

float standard_deviation (array_type array[], int num_elements)
 {
  float variance (array_type *, int);

  return (sqrt((variance (array, num_elements) * num_elements) /
               (num_elements - 1)));
 }
```

Least Squares Fit

The least squares fit is one of the simplest methods used to determine the linear equation that best fits a collection of data values. For example, given the following distribution of values,

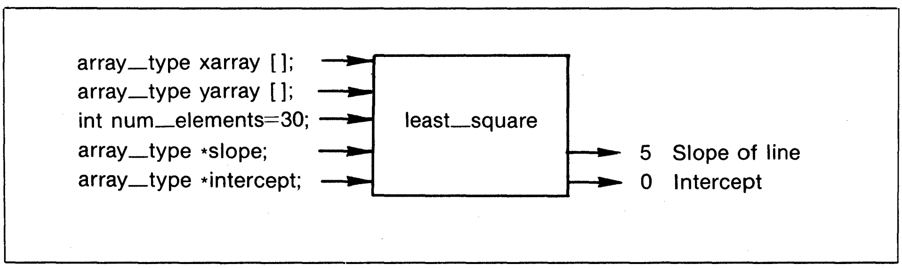


the least squares algorithm provides a linear equation that best fits the data, as shown here:



The line shown is called the *line of best fit.* The slope and intercept of this line are used to determine missing points in your data. Linear equations are expressed in slope-intercept format.

The procedure least_square returns the slope and intercept of the line that best fits the data.



```
/*
 * void least_square (x, y, num_elements, slope, intercept)
 *
 * Return the slope and intercept of the line that best fits
 * the x and y data values given.
 *
 * x (in): array of x coordinates.
 * y (in): array of y coordinates.
 * num_elements (in): number of elements in the array.
 * slope (out): slope of the line.
 * intercept (out): intercept of the line.
 *
 * least_square (x, y, 10, &slope, &intercept);
 *
 * This routine requires you to define the type array_type
 * based upon the type of your array.
 *
 */

void least_square (array_type x[], array_type y[], int num_elements,
                   float *slope, float *intercept)
 {
  float xsum = 0.0, ysum = 0.0, xsquared_sum = 0.0, xy_sum = 0.0;

  int i;

  for (i = 0; i < num_elements; i++)
    {
     xsum += x[i];
     ysum += y[i];
     xsquared_sum += x[i] * x[i];
     xy_sum += x[i] * y[i];
    }

  *slope = ((xsum * ysum) - (num_elements * xy_sum)) /
           ((xsum * xsum) - (num_elements * xsquared_sum));

  *intercept = (ysum - (*slope * xsum)) / num_elements;
 }
```

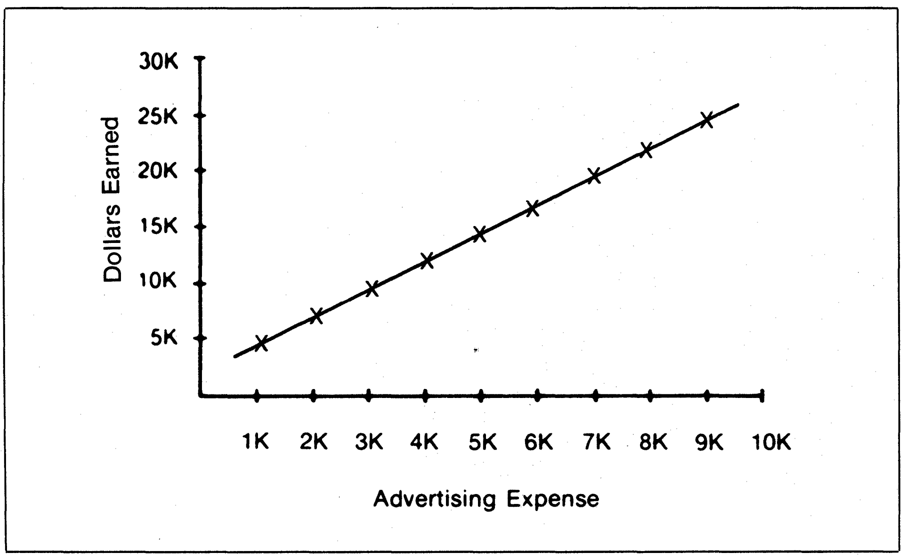
Once you know the slope and intercept for the data, you can use them to estimate missing points, as shown here:

$$\begin{aligned}
&\text{slope} = 5 \qquad\qquad \text{intercept} = 1\\
&x = 1.5\\
&y = \text{slope} * \chi + \text{intercept}\\
&y = \text{slope} * 1.5 + \text{intercept}\\
&y = 5 * 1.5 + 1\\
&y = 7.5 + 1\\
&y = 8.5
\end{aligned}$$

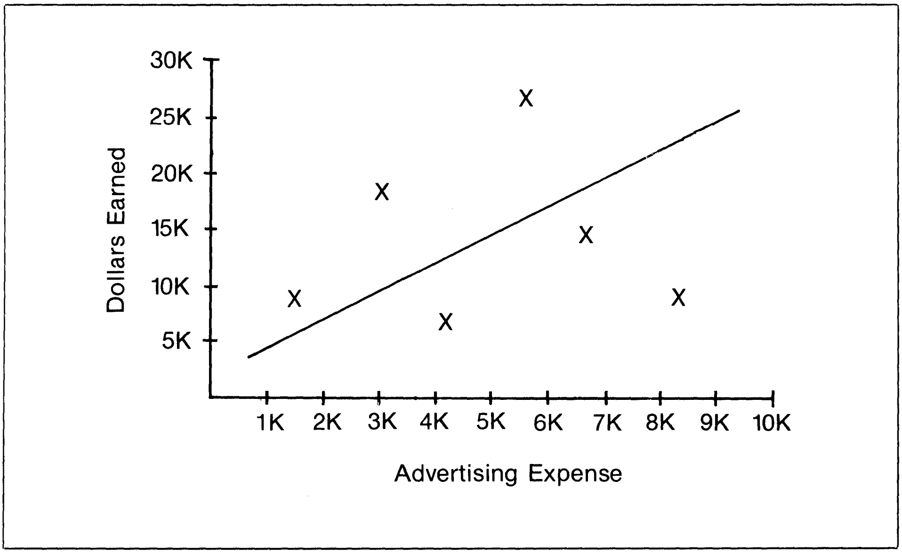
Statisticians use residuals to determine the goodness of fit of the linear equation produced by least_square. A *residual value* is the distance between each value and the line of best fit. For example, the difference between the actual value of Y and the approximated value Y′ can be computed by the following equation:

RESIDUAL = Y − Y′;

The sum of the residuals for an array can determine the validity of the linear equation. For example, if the data is linear,



the sum of the residuals will be 0. As the line less approximates the data this sum will be greater, as shown here:



Using Macros

Many Turbo C programmers must often balance the increased code size produced by C macros (over that of functions) against their increased flexibility. Developing generic array-manipulation routines in Turbo C is not an easy task. However, you can often use macros instead to increase the flexibility of your code. For example, the following macro returns the sum of the elements in an array:

```
#define sum_array(array, num_elements, rslt)        \
          {                                         \
            int index = 0;                          \
            rslt = 0;                               \
            while (index < (num_elements))          \
              rslt = rslt + array[index++];         \
          }
```

Note the differences between the macro code and the routine presented previously in this chapter. First, because the code is a macro and not a function, it does not return a value. You must pass a parameter to the macro that will store the result, as shown here:

```
#define sum_array(array, num_elements, rslt)       \
```

Second, because the macro does not define an array type, it will work without modification for arrays of any type. For example, the following program obtains the sum of arrays of type int and float by using the single macro:

```
#define sum_array(array, num_elements, rslt)       \
          {                                        \
            int index = 0;                         \
            rslt = 0;                              \
            while (index < (num_elements))         \
              rslt = rslt + array[index++];        \
          }

main ()
 {
  int int_array[10], int_result;
  float float_array[10], float_result;

  int i;

  for (i = 0; i < 10; i++)
   {
     int_array[i] = i;
     float_array[i] = i;
   }

  sum_array(int_array, 10, int_result);
  sum_array(float_array, 10, float_result);

  printf ("Result of int_array sum %d\n", int_result);
  printf ("Result of float_array sum %f\n", float_result);
 }
```

All of the routines in this chapter can be implemented as macros in this fashion. In many cases, your code will actually execute slightly faster since you do not have the stack overhead associated with functions. The tradeoff, however, is increased code size.

Multidimensional Arrays

All of the arrays presented thus far have been single-dimensional arrays like the following:

```
float scores [10]
int grades[5];
```

However, many applications require arrays of multiple dimensions, as shown here:

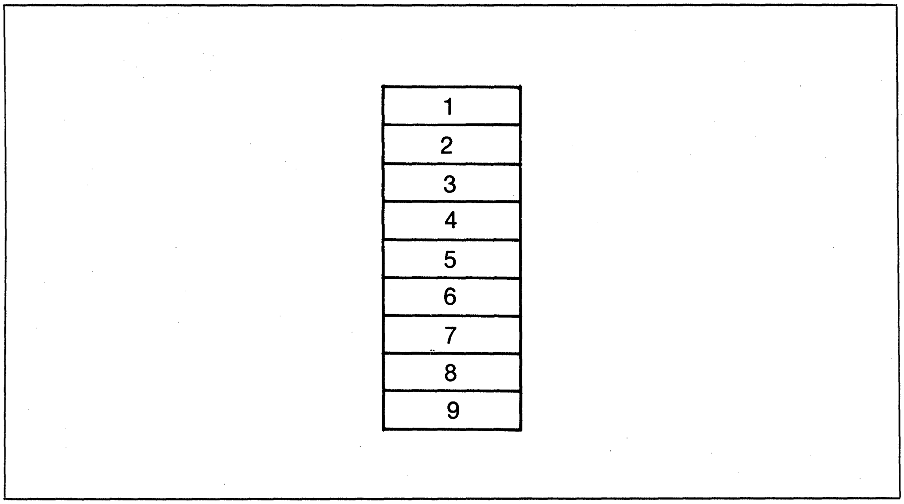
```
float box [3][3];
int tax_table [3][5][10];
```

Because of the way that Turbo C usually stores arrays, you can normally pass single-dimensional or multidimensional arrays to your functions without modification to the code. For example, assume you have the array shown here with the values given:

int X[3] [3];

1	2	3
4	5	6
7	8	9

In most cases, Turbo C actually stores the array as a single-dimensional array, as follows:



Thus, you can normally pass the array to your array-manipulation routines regardless of the number of dimensions. Simply remember to pass the correct number of elements, as shown here:

float box [3][3] has 3 * 3, or 9, elements

int tax_table [3][5][10] has 3 * 5 * 10, or 150, elements

In most cases, despite the number of dimensions, the routines presented work without modification. Experiment with these routines and you should find that they are quite flexible. If you are using arrays of pointers, be aware of the fact that Turbo C is not required to store arrays in a linear manner. The only ANSI requirement is that array[i] is directly equivalent to *((array)+(i)). As a result, you will likely need to modify your array-manipulation routines.

Admittedly, many of the routines presented in this chapter are quite simple. However, by creating libraries of routines like these, your program development becomes much simpler and much faster. Remember that these library routines are your building blocks for larger programs.

CHAPTER

11

Searching and Sorting

Many applications that use arrays to store data also perform table look-ups. These applications search for specific values (or process the data contained in the array) with the assumption that the data is in either ascending or descending order. Searching and sorting operations are important aspects of most computer applications. Choosing the correct sorting or searching algorithm has a significant impact on the execution time of your programs.

Computer scientists have thoroughly researched the characteristics of sorting and searching algorithms. They found that several algorithms execute much faster because fewer iterations are required to sort an array or to locate a specific value.

This chapter discusses sequential and binary searches. The sorting algorithms in this chapter include the bubble, selection, Shell, and quick sorts. As in Chapter 10, the routines are as generic as possible. Each routine is based on the user-defined type array_type. To minimize duplicate sorting routines, the sorting order (either ascending or descending) is a parameter to each routine.

Searching

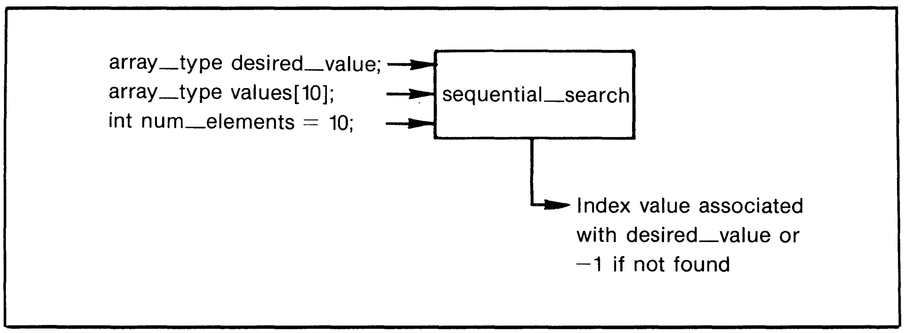
Many programming applications must search arrays for specific values. For example, assume that the following arrays contain employee information:

Index	EMPLOYEE	ID NUMBER	SALARY
0	Boy	1111	30000
1	Burnham	2222	45000
2	Byrd	3700	38000
3	Davis	4201	25000
4	Eubank	5001	60000
5	Grant	5500	35000
6	Jones	6200	45000
7	Kempf	7777	50000
8	Rosaschi	8001	55000
9	Watson	8372	32000

If management wants to access the salary of Jones, a program can sequentially search the array employee until the name is found. The program can then locate the salary associated with the index value that points to Jones. In this case, index 6 points to a salary of $45,000. The goal of the search routines presented here is to locate a value and return the corresponding index. If the value is not found, each routine will return the value −1.

Sequential Search

The sequential search is the simplest searching algorithm. The values to be examined here are stored as elements in the array. Each time a value must be found, the sequential search starts with the first element in the array. The search examines the elements one after another until either the value is found or the array elements are exhausted. In the example of the employee information array, the search would first test the element Boy and then examine successive values of the array until it found the element Jones. The following routine implements the sequential search:




```
/*
 * sequential_search (value, array, num_elements)
 *
 * Sequentially search the array of values given in order to locate
 * a specific value.  Return the index of the value, or -1 if the
 * value is not found.
```

```
 *
 * value (in): Value to search for.
 * array (in): Array of values to examine.
 * num_elements (in): Number of elements in the array.
 *
 * index = sequential_search (5, scores, 10);
 *
 * If multiple occurrences of the same value are present in the
 * array, this routine returns the index of the first occurrence.
 *
 */

int sequential_search (array_type value, array_type values[],
                       int num_elements)
 {
  int i, location = -1;

  for (i = 0; (i < num_elements) && location == -1; i++)
    if (value == values[i])
      location = i;

  return (location);
 }
```

Keep in mind that you can again use C macros in order to increase the generic nature of each routine. For example, this macro implements a generic segmental search:

```
#define seq_search(value, values, num_elements, location)      \
 {                                                              \
  int k;                                                        \
  location = -1;                                                \
                                                                \
  for (k = 0; (k < num_elements - 1) && location == -1; k++)   \
    if (value == values[k])                                     \
      location = k;                                             \
 }
```

This technique can be used for all of the routines presented in this chapter. The drawback, however, is increased code size.

Binary Search

In the previous application, the sequential search successfully found the desired information. In many cases, you can reduce the number of iterations the routine must perform to find the data by using a different searching algorithm, such as the binary search. To perform a binary search, the values in the array must be in order, traditionally ascending order (lowest to highest).

The binary search is one of the quickest searching algorithms used by programmers. Unlike the sequential search (which examines successive elements of the array), the binary search reduces the number of elements that must be examined (by a factor of two) with each iteration, until the desired record is found.

This process is similar to the one you use when you look up a telephone number. Assume that you are looking up the name Jones. You will probably start near the middle of the book. If the names on that page begin with a letter other than "J," you have effectively cut in half the number of pages you must search for the name. If the name Jones is not on this page, you simply repeat this process until the name is found.

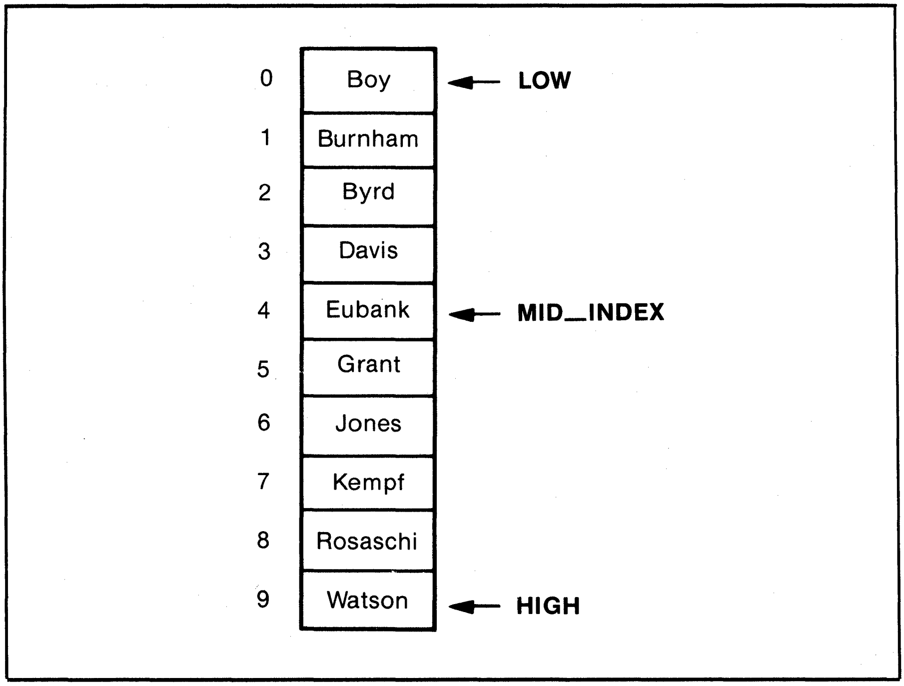
Admittedly, you could have used a sequential search to find the name Jones by starting at the first page in the telephone book and examining every page. In a small town, this might not take long. However, in a city such as New York, a sequential search would take far too much time.

You could use the sequential search to search the array of employee information for the salary of Jones. The process requires seven iterations, but the binary search requires only four iterations. This obviously reduces the execution time of the program. The decrease in execution time becomes important as array sizes increase.

The first iteration of the binary search examines the entire array (just as you examined the entire telephone book). Using the employee information example, the variables low and high are assigned the values 0 and 9. The variable mid_index is the middle element in the search range. The array element values[mid_index] contains the value that you will compare to desired_value. To calculate mid_index, use the following line:

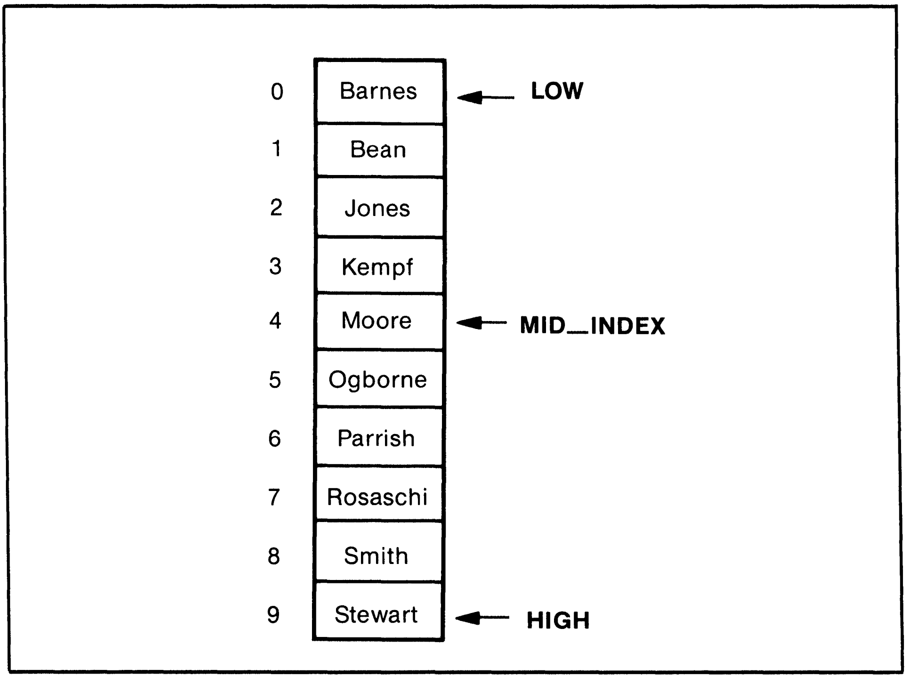
```
mid_index = (high + low) / 2;
```

Since the routine performs integer division, mid_index is assigned the value 4, as shown here:



If the value contained in values[mid_index] equals the desired value, the search is completed by setting a variable called found to true.

If the value contained in values[mid_index] is greater than the desired value, the algorithm modifies the search range because the value indicates that searching past that point for the desired value is not necessary. For example, if the array contains the following,



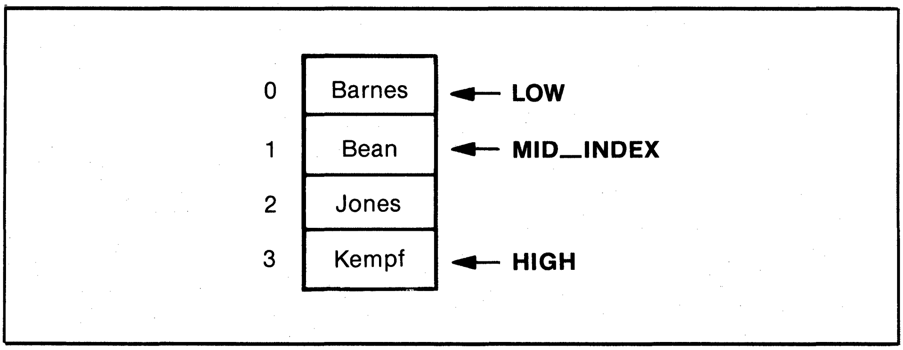
and you are searching for Jones, you have no reason to search above values[index] for the value. The search then modifies the value of high as follows:

```
high = mid_index - 1;
```

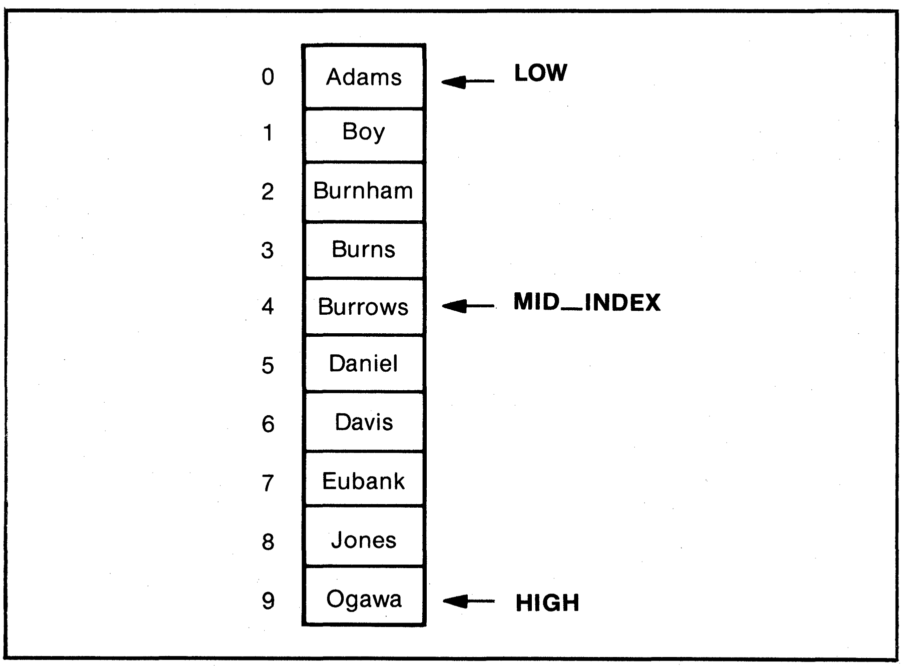
In effect, this process creates a new range of names to examine. The value of mid_index must also be modified, as shown here:

```
mid_index = (high + low) / 2;
```

The new range of names then contains the following:



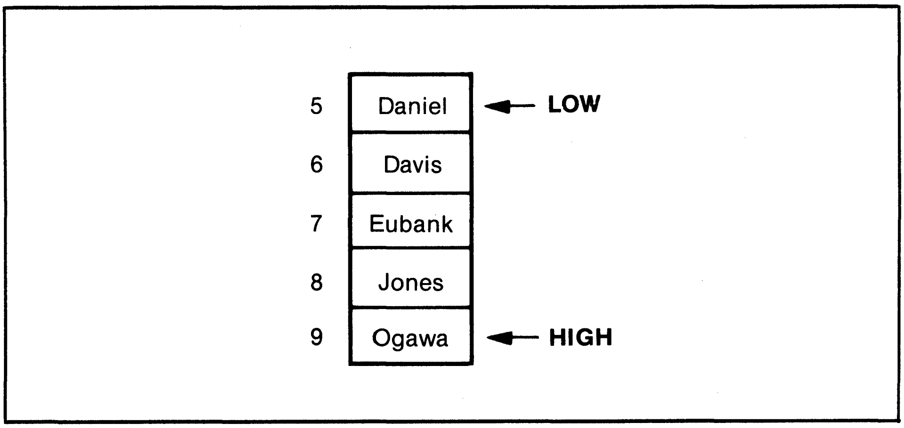
Similarly, if the initial array contains the following,



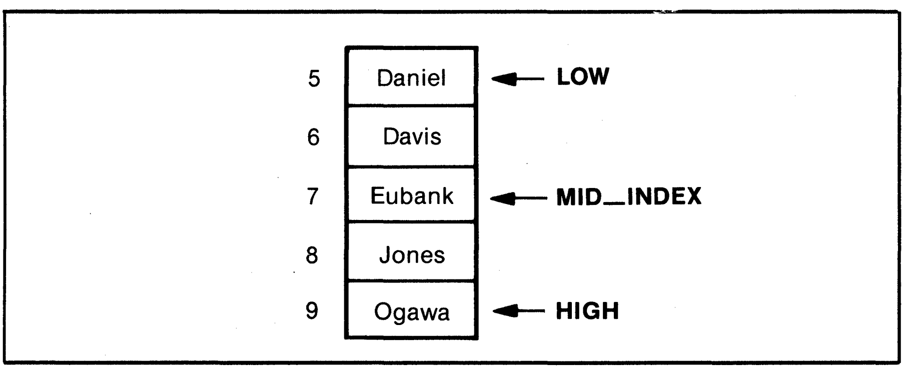
and you are searching for Jones, you need not search below values[mid_index] for the value. The search modifies the value contained in low by the following statement:

```
low = mid_index + 1;
```

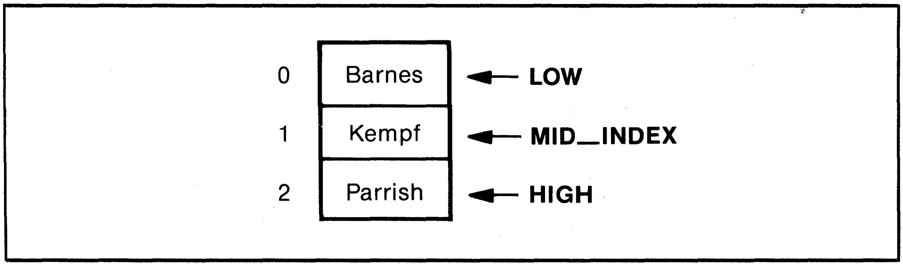
This statement then produces this new range:



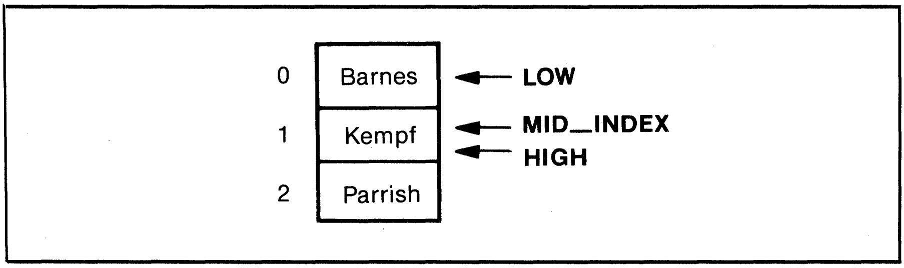
The algorithm recomputes mid_index to yield



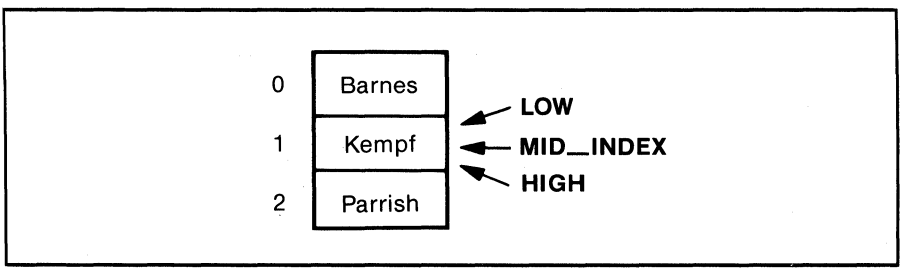
After the desired value is found, the variable found terminates the search. If the value is not found, a secondary test is required. For example, if the array contains



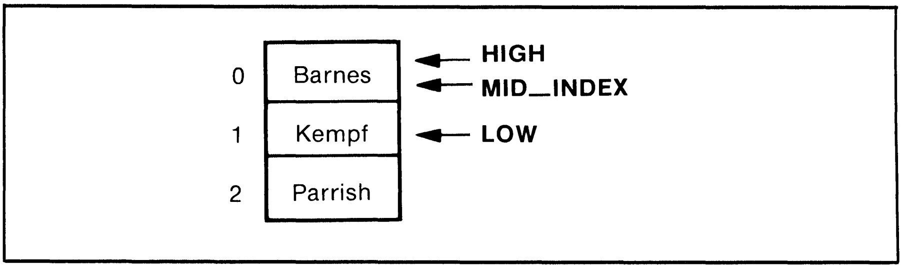
and the desired value is Jones, the first iteration modifies high and mid_index as follows:



The second iteration produces



The third iteration illustrates the error that occurs if the algorithm does not perform the secondary test that prevents the array boundaries from being overrun, as shown here:

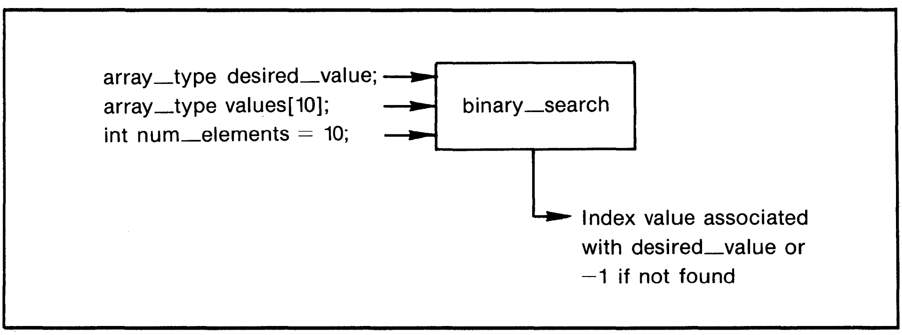


If the desired value is not found, the algorithm attempts to access invalid subscripts, which results in an error. The complete test necessary to prevent this error becomes

```
while ((! found) && (high >= low))
```

If the desired value is not found, the variable found remains false and should be examined by the calling routine.

The following routine implements the complete binary search:



```
/*
 * binary_search (value, array, num_elements)
 *
 * Use a binary search of the array of values given in order to locate
 * a specific value.  Return the index of the value, or -1 if the
 * value is not found.
 *
 * value (in): Value to search for.
 * array (in): Array of values to examine.
 * num_elements (in): Number of elements in the array.
 *
 * index = binary_search (5, scores, 10);
 *
 */

int binary_search (array_type value, array_type values[],
                   int num_elements)
 {
  int found = 0;
  int high, low, mid_index;

  low = 0;
  high = num_elements;
  mid_index = (high + low) / 2;

  while ((! found) && (high >= low))
   {
    if (value == values[mid_index])
      found = 1;
    else if (value < values[mid_index])
      high = mid_index - 1;
    else if (value > values[mid_index])
      low = mid_index + 1;
    mid_index = (high + low) / 2;
   }

  return ((found) ? mid_index : -1);
 }
```

Sorting

Many programming applications require that data be processed in either ascending (lowest to highest) or descending (highest to lowest) order. In such instances, a sorting algorithm must be used to place the data in order. The sorting algorithms introduced in this chapter are the bubble sort, the selection sort, the Shell sort, and the quick sort.

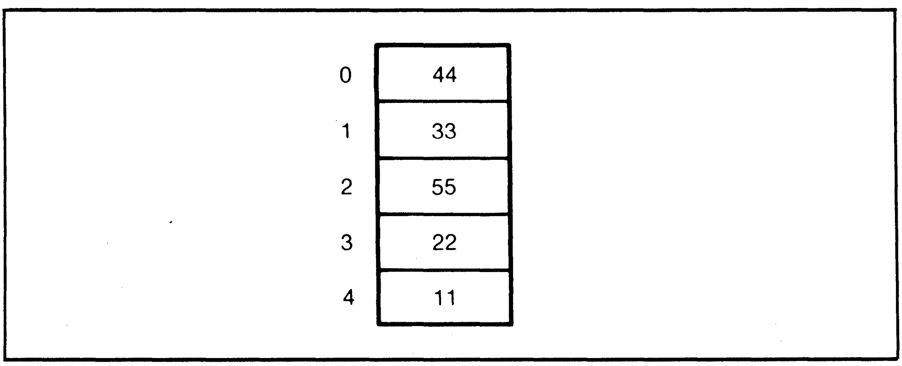
All of the array-manipulation routines in Chapter 10 were based on the type array_type. Consequently, duplicate routines for arrays of different types did not have to be developed. Remember, if you develop two routines to sort data in ascending and descending order, your programming efforts have been needlessly duplicated. To avoid duplicate routines, the sorting algorithms provided in this chapter allow you to specify as a parameter the desired order (ascending or descending), as shown here:

Value	**Sort Order**
0	Ascending
1	Descending

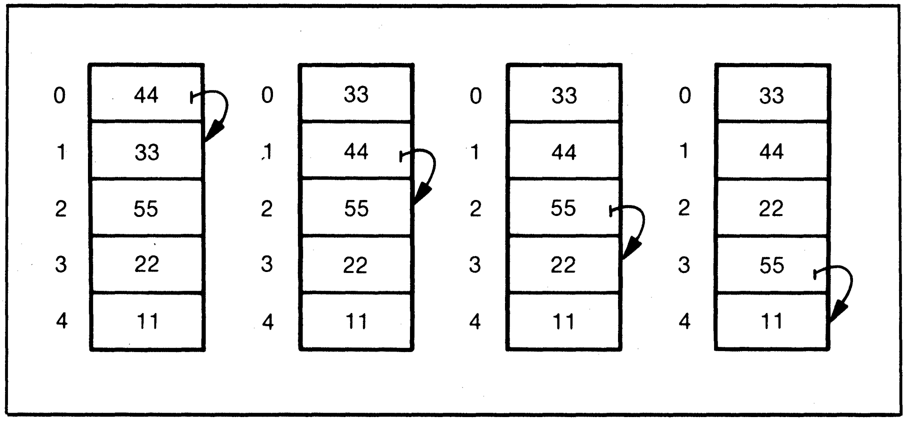
Bubble Sort

The *bubble sort* is a popular sorting algorithm because of its simplicity. It is so named because with each iteration, a value rises (like a bubble) to the top of the array. The bubble sort gets values in the correct order by comparing adjacent array elements and exchanging those that are out of sequence. Because of this, the number of iterations the bubble sort requires makes it an inefficient sort for large arrays. If your array contains more than 30 elements, you should use either the Shell sort or the quick sort.

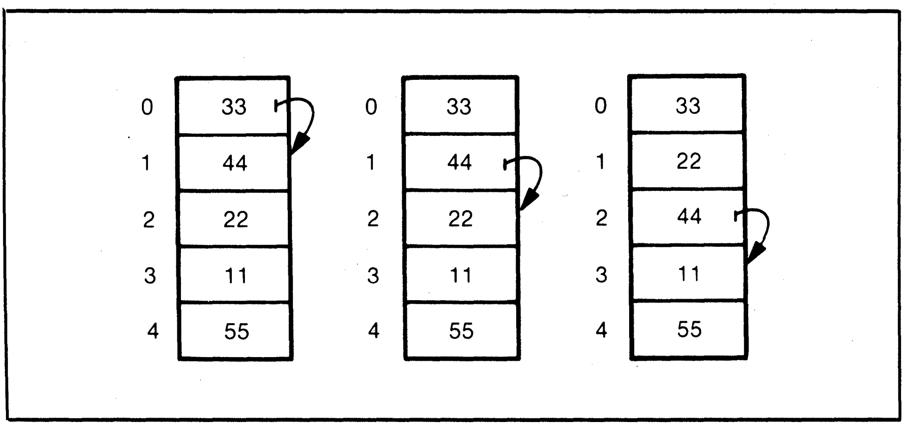
Assume that the array values contain the following:



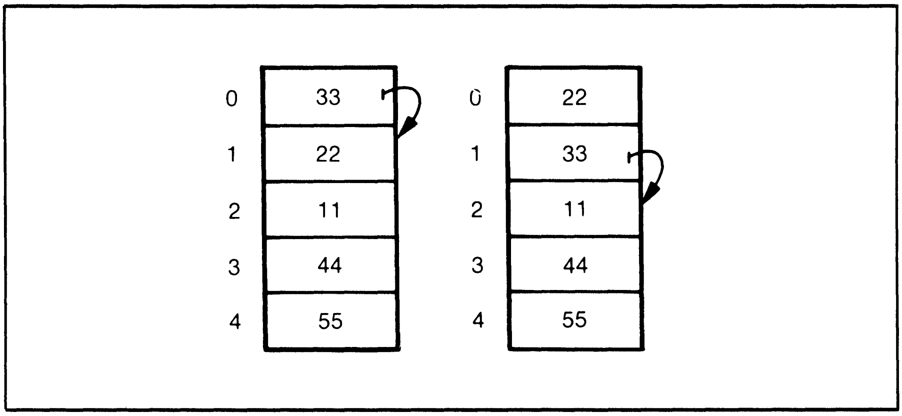
The first iteration of the bubble sort for ascending order will perform four evaluations:



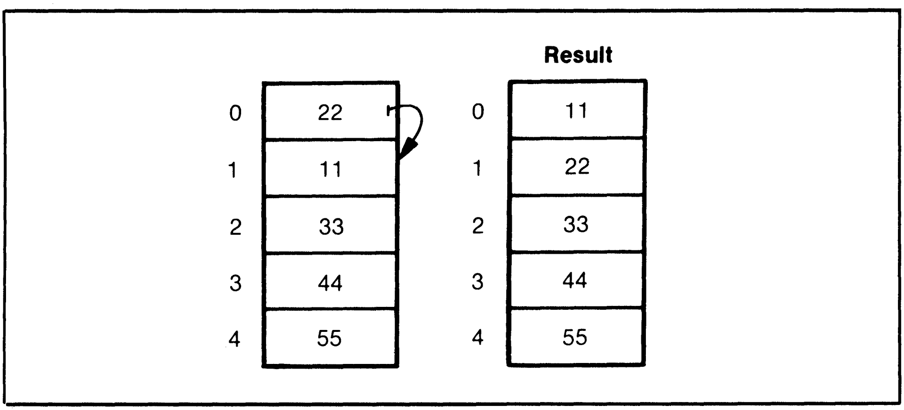
Since the largest value in the array is in the correct location after the first iteration, the algorithm examines only the first four elements on the second iteration:



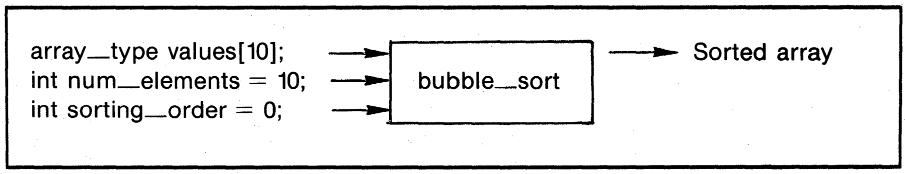
In the third iteration, only two elements are examined:



The final iteration ensures that the first two array elements are in the correct order:



The following routine implements the bubble sort:



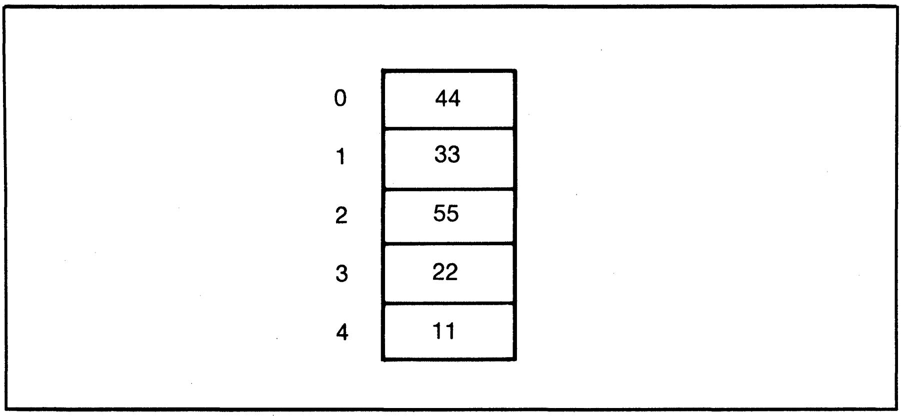

```
/*
 * void bubble_sort (values, num_elements, order)
 *
 * Sort the array of values in the order specified.
 *
 * values (in/out): Array of values to sort.
 * num_elements (in): Number of elements in the array.
 * order (in): Desired sorting order:
 *              0 for ascending
 *              1 for descending
 *
 * bubble_sort (values, 10, 1);
 *
 */

void bubble_sort (array_type values[],
                  int num_elements, int order)
 {
  array_type temp;

  int i, j;

  for (i = 0; i < num_elements - 1; i++)
    for (j = i + 1; j < num_elements; j++)
      if ((! order && (values[i] > values[j])) ||
          (order && (values[i] < values[j])))
        {
         temp = values[i];
         values[i] = values[j];
         values[j] = temp;
        }
 }
```

Selection Sort

The *selection sort* is another simple sorting algorithm. Although most schools teach the bubble sort, many programmers find that the selection sort is easier to understand and to use without losing efficiency.

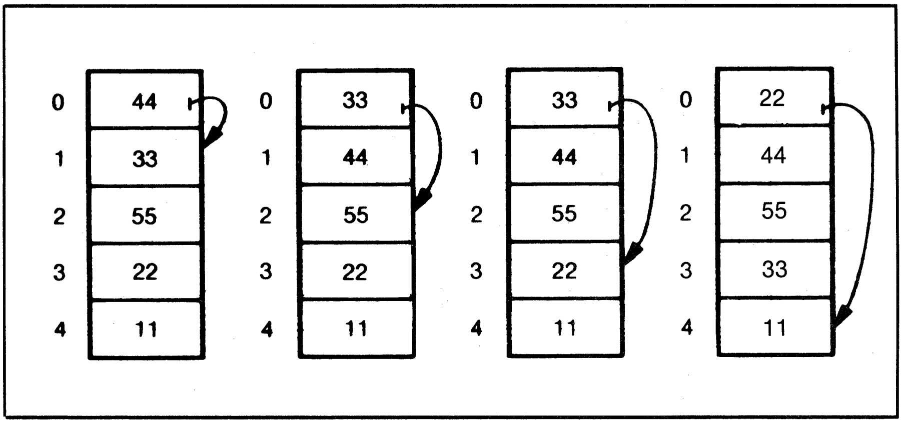
In the selection sort, elements are sorted by selecting the maximum or minimum value (depending on ascending or descending order) with each iteration.

Assume that the following array is sorted in ascending order:

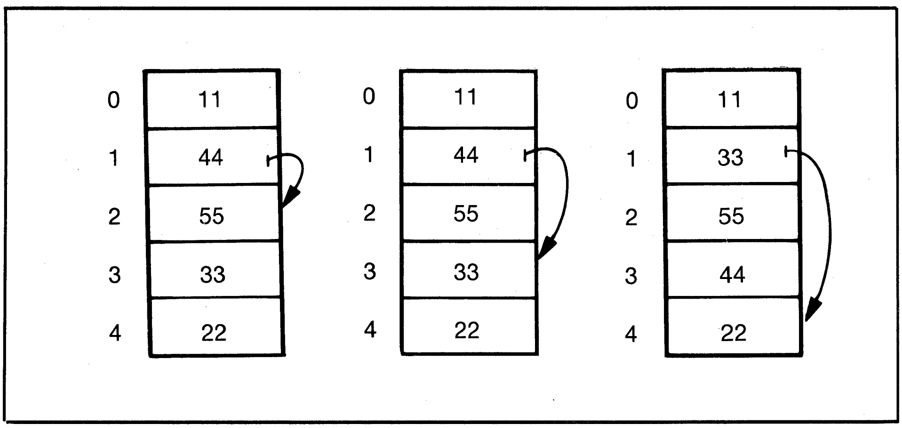


The first iteration selects the minimum value and places it in the first element. To accomplish this iteration, the sort first selects the first element as the current index. The sort then compares elements in the array to values[current]. If one of the values is greater, the two values are exchanged.

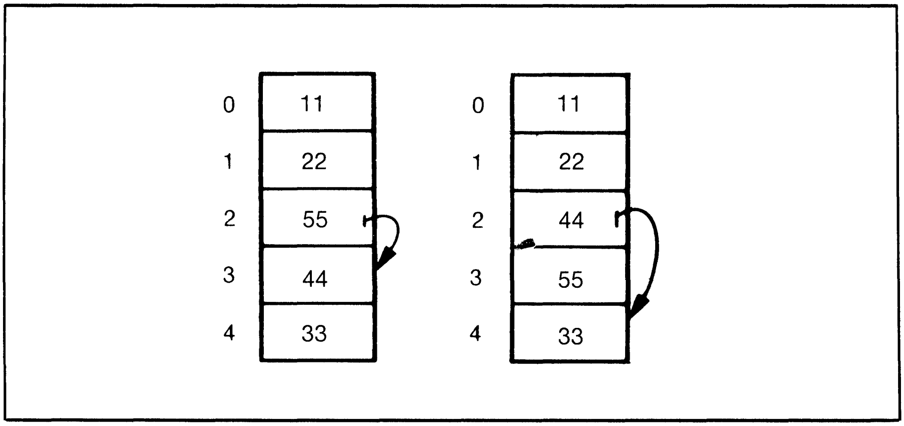
The first iteration selects the minimum value as follows:



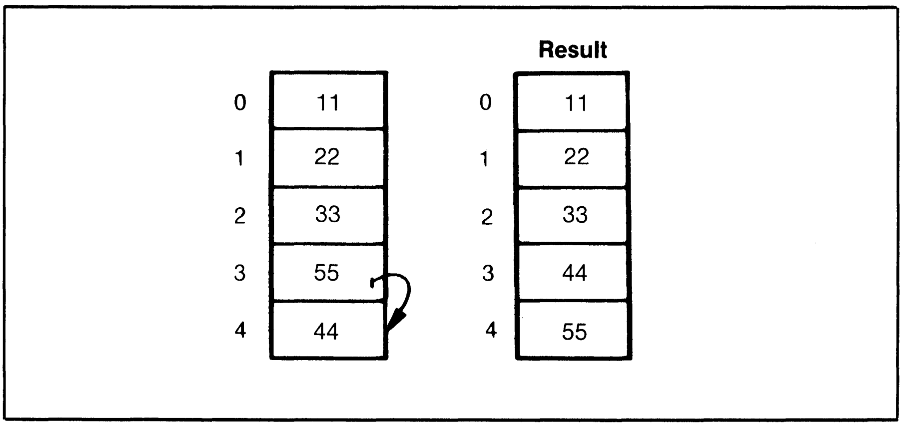
The second iteration places the second smallest value in element 2:



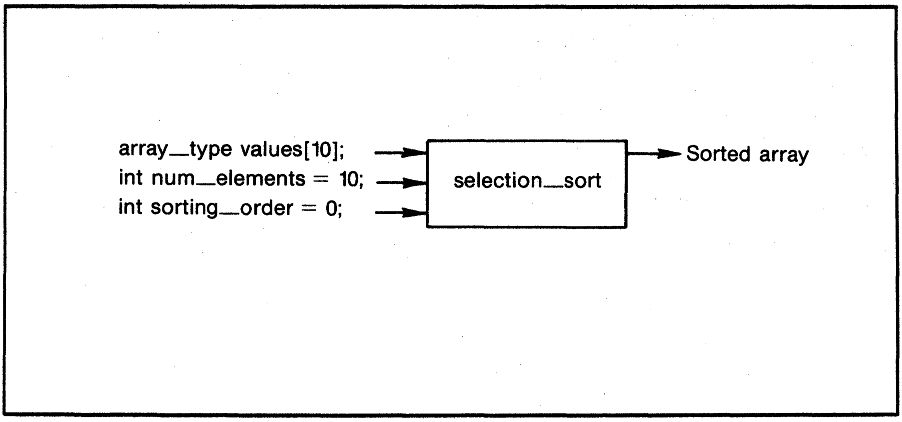
The third iteration selects the third smallest value:



The fourth iteration results in the sorted array:



The following routine implements the selection sort:




```
/*
 * void selection_sort (values, num_elements, order)
 *
 * Sort the array of values in the order specified.
 *
 * values (in/out): Array of values to sort.
 * num_elements (in): Number of elements in the array.
 * order (in): Desired sorting order:
 *             0 for ascending
 *             1 for descending
 *
 * selection_sort (values, 10, 1);
 *
 */

void selection_sort (array_type values[], int num_elements, int order)
 {
  array_type temp;

  int j, current;

  for (current = 0; current < num_elements - 1; current++)
   {
    for (j = current + 1; j < num_elements; j++)
      if ((! order && (values[current] > values[j])) ||
          (order && (values[current] < values[j])))
        {
         temp = values[current];
         values[current] = values[j];
         values[j] = temp;
        }
   }
 }
```

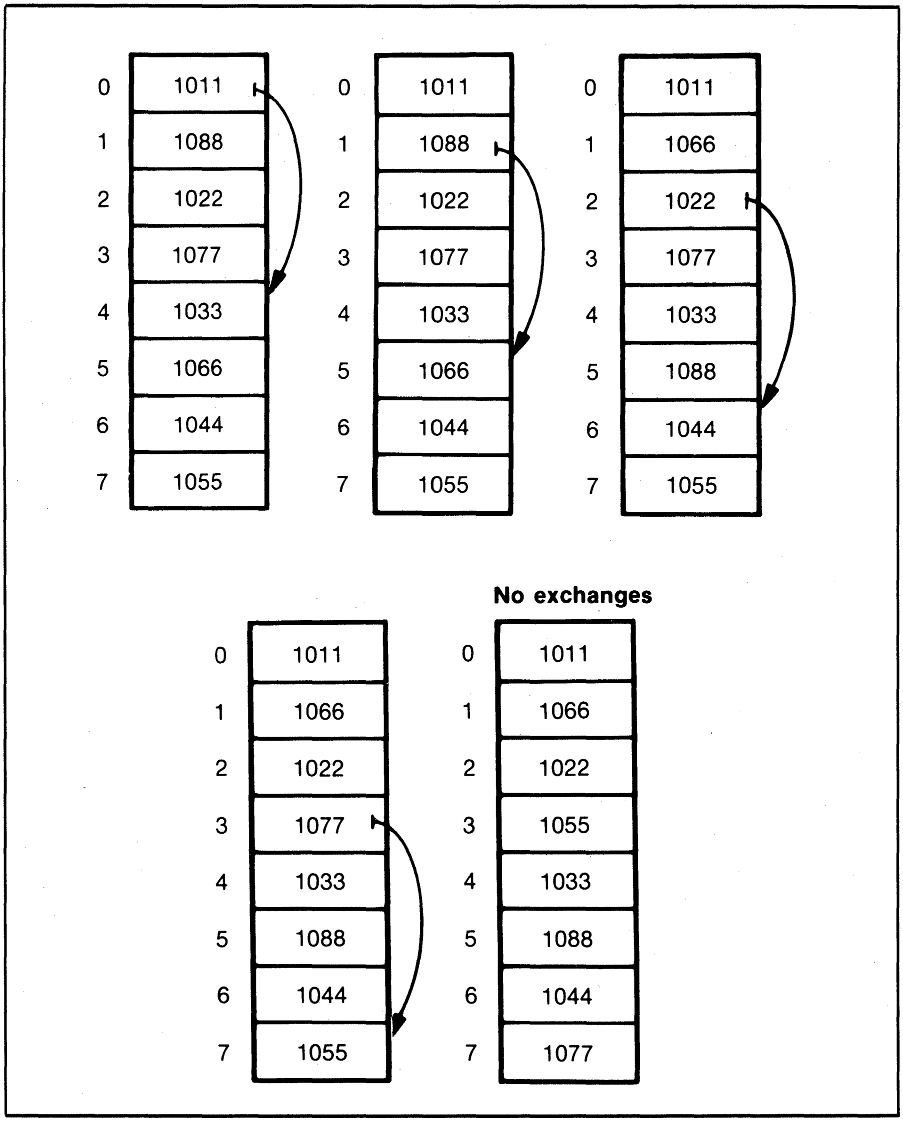
Shell Sort

To enhance the efficiency of sorting algorithms for large arrays, Donald Shell created a sorting algorithm that is now called the *Shell sort.* The Shell sort differs from the bubble sort in that it compares elements that are spaced farther apart before comparing adjacent elements. This removes much of the array's disorder in early iterations.

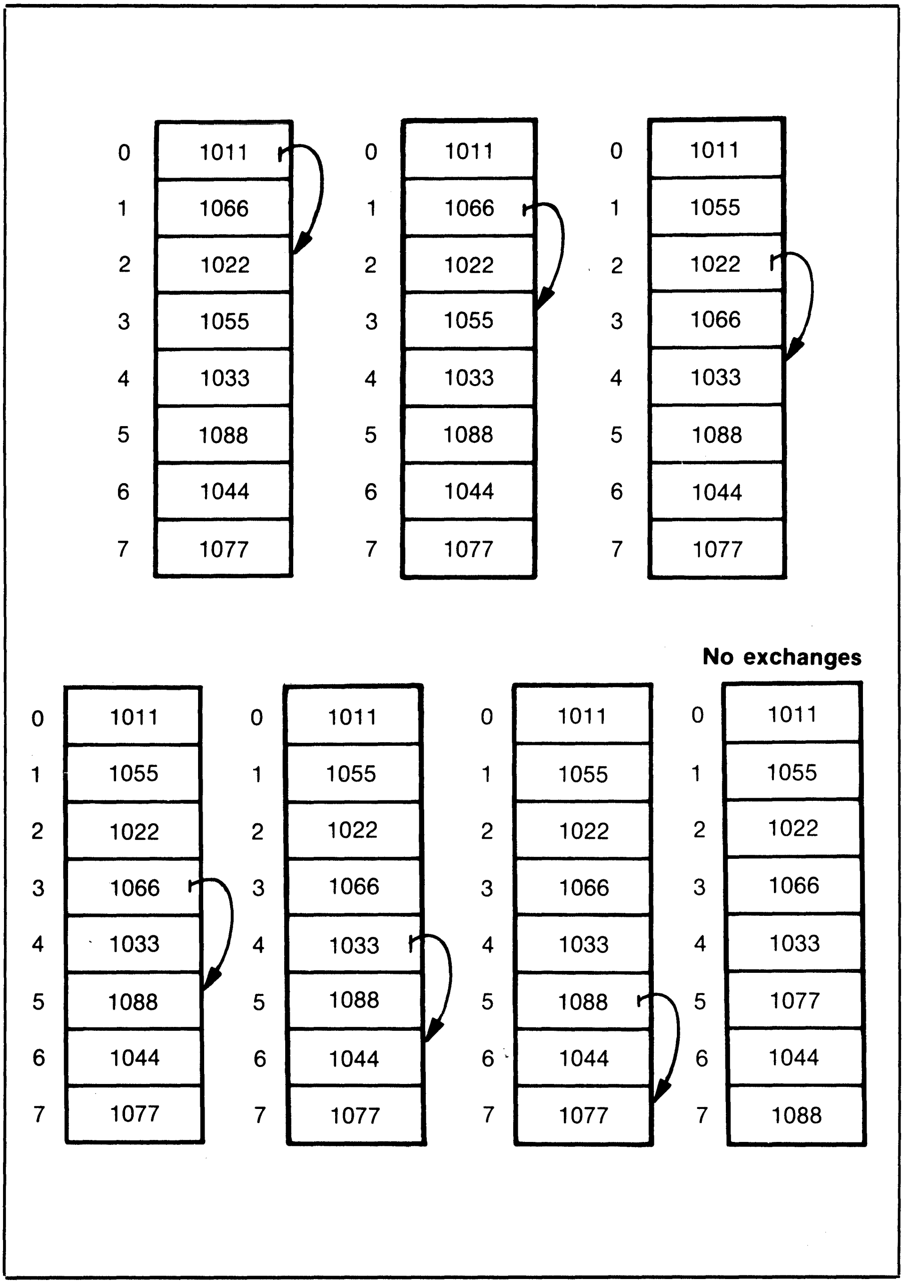
The Shell sort uses a variable called gap that is initially set to the value of one-half of the number of elements in the array. The value of gap specifies the distance between each pair of comparison elements in the array. In the following example, the elements compared will initially be separated by a gap of 4:

0	1011
1	1088
2	1022
3	1077
4	1033
5	1066
6	1044
7	1055

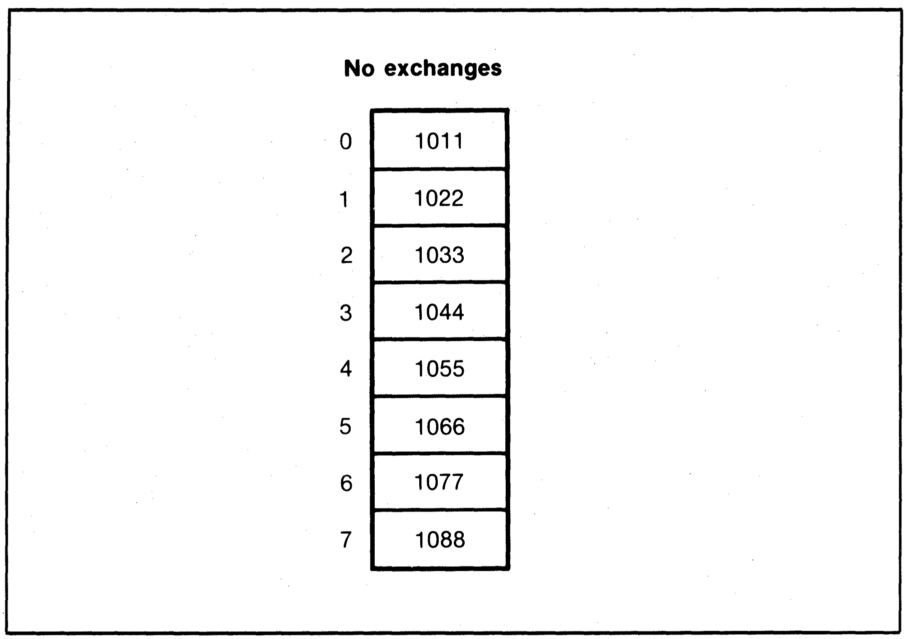
For the first iteration of the sort, gap is assigned a value of 4. This iteration of the array compares all of the elements separated by this distance. The process is repeated until no exchanges occur with a gap of 4:



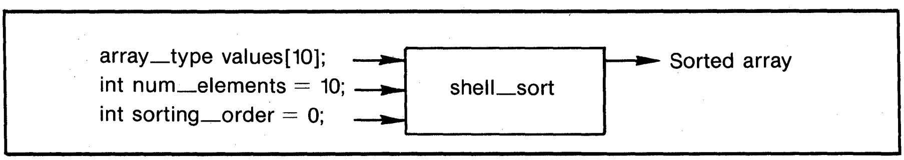
When no more exchanges can occur with a gap of 4, the algorithm modifies gap to gap / 2. The elements separated by a gap of 2 are then compared until no exchanges occur:


0 1 2 3 4 5 6 7
1011 1066 1022 1055 1033 1088 1044 1077
1011 1066 1022 1055 1033 1088 1044 1077
1011 1055 1022 1066 1033 1088 1044 1077
1011 1055 1022 1066 1033 1088 1044 1077
1011 1055 1022 1066 1033 1088 1044 1077
1011 1055 1022 1066 1033 1088 1044 1077
No exchanges
1011 1055 1022 1066 1033 1077 1044 1088

When no more exchanges can occur with a gap of 2, gap is again modified to gap / 2, and the process is continued with a gap of 1. When no exchanges occur with a gap of 1, gap is assigned the value of gap / 2. In this case, integer division assigns gap the value of 0, which is the ending condition:



The following routine implements the Shell sort:

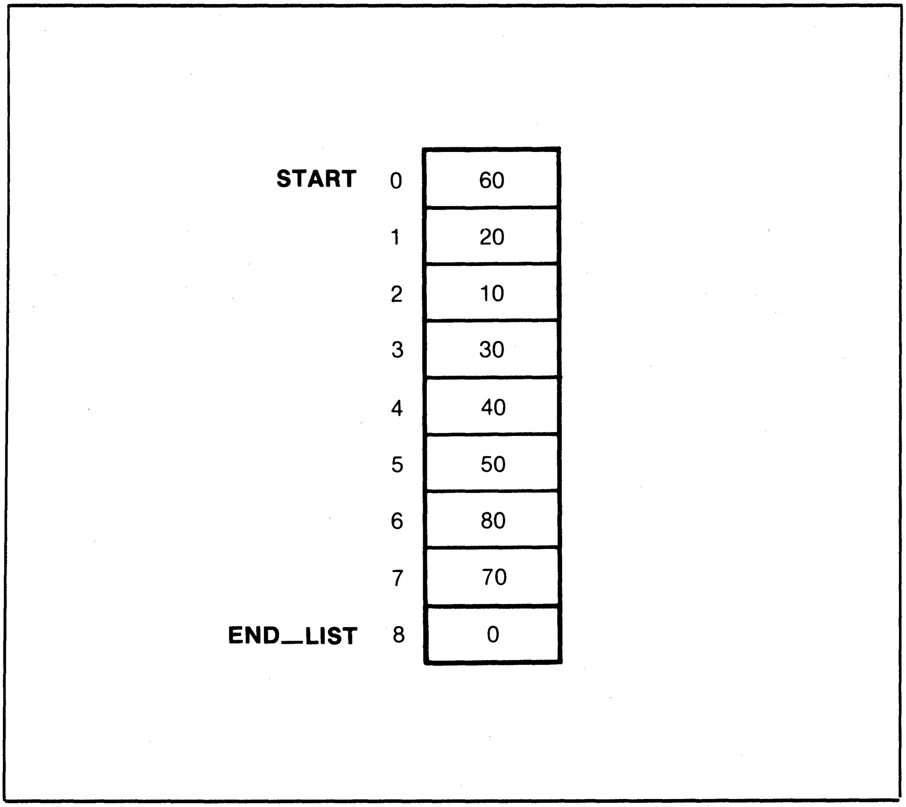


```
/*
 * void shell_sort (values, num_elements, order)
 *
 * Sort the array of values in the order specified.
 *
 * values (in/out): Array of values to sort.
 * num_elements (in): Number of elements in the array.
 * order (in): Desired sorting order:
 *              0 for ascending
 *              1 for descending
 *
 * shell_sort (values, 10, 1);
 *
 */

void shell_sort (array_type values[],
                 int num_elements, int order)
 {
  array_type temp;

  int i, gap, exchange_occurred;

  gap = num_elements / 2;

  do
    do {
      exchange_occurred = 0;

      for (i = 0; i < num_elements - gap; i++)
        if ((! order && (values[i] > values[i+gap])) ||
          (order && (values[i] < values[i+gap])))
         {
          temp = values[i];
          values[i] = values[i+gap];
          values[i+gap] = temp;
          exchange_occurred = 1;
         }
     }
    while (exchange_occurred);
  while (gap = gap / 2);
 }
```

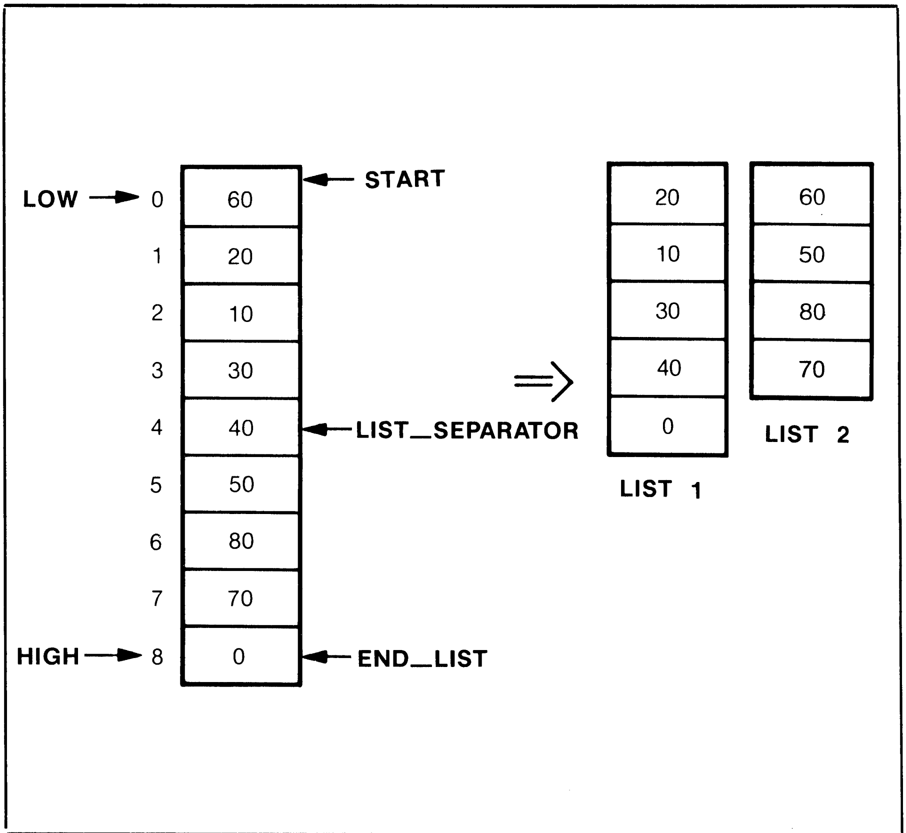
Quick Sort

Although the efficiency of the Shell sort increases as the number of elements in the array increases, it, too, has limitations. The *quick sort* (which is often implemented recursively) increases the speed of the sort as the number of elements in the array approaches 150 to 200 elements. In fact, the quick sort is one of the fastest array-sorting algorithms in use today.

The quick sort sorts data by breaking a list of values into a series of smaller sorted lists. For example, if the array



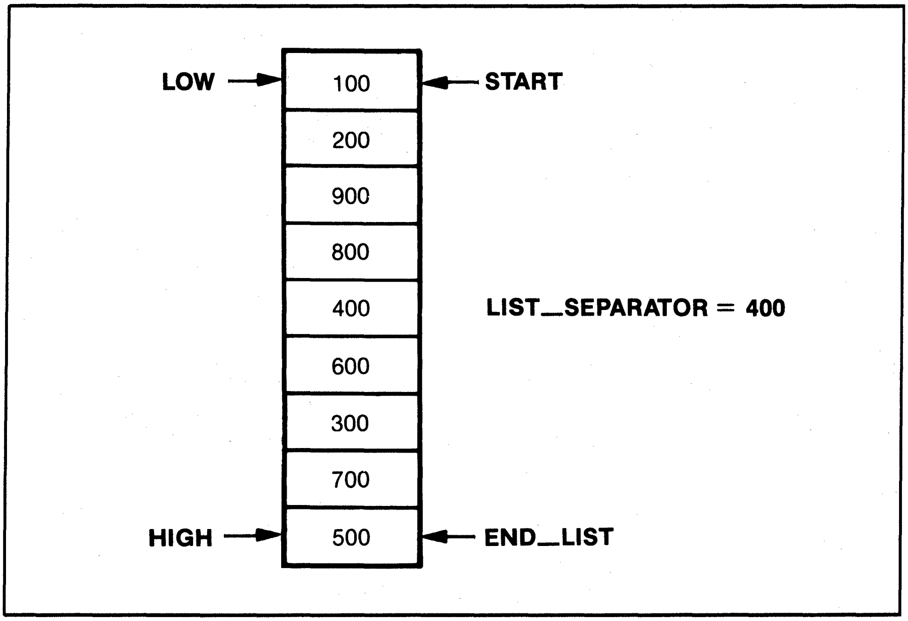
is passed to the quick sort routine, the algorithm will select the value contained in values[(start+end_list) / 2] (which in this case is values[4]) as the list separator. Any values in the list that are less than or equal to the list separator are placed in one list, and the values that are greater than the list separator are placed into a second list, as shown here:



The same process is carried out on each sublist, or range, until each contains only one element. At that point, the array will be sorted.

Figure 11-1 illustrates the sequence in which the sublists are constructed. The sort splits the list into two parts: The smaller items are placed in the left-hand list, and the larger items into the right-hand list. The process is repeated until there is only one item in each list and the items are sorted from left to right.

As another example, imagine that the following array



is passed to the quick sort routine to be sorted in ascending order. It is first divided into two lists. The variable low is assigned to the first element in the list. The variable high is assigned to the last element in the list. The variable low is then incremented until values[low] contains a value that is greater than or equal to the list_separator (for descending order, the value must be less than the list_separator).

```
while (values [low] < list_separator)
      low++;
```

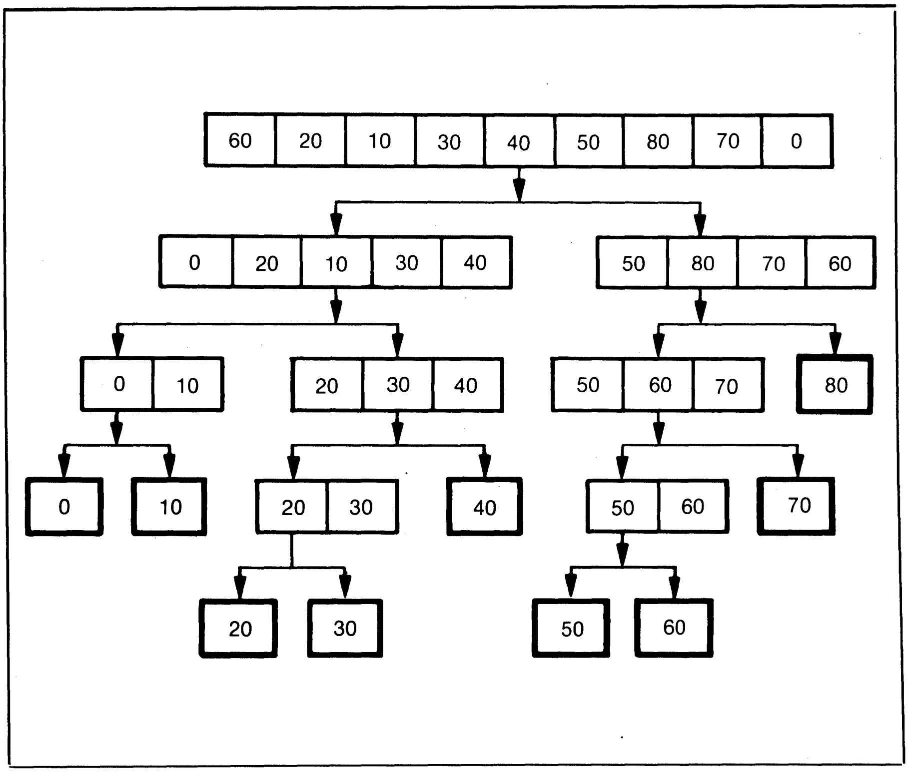
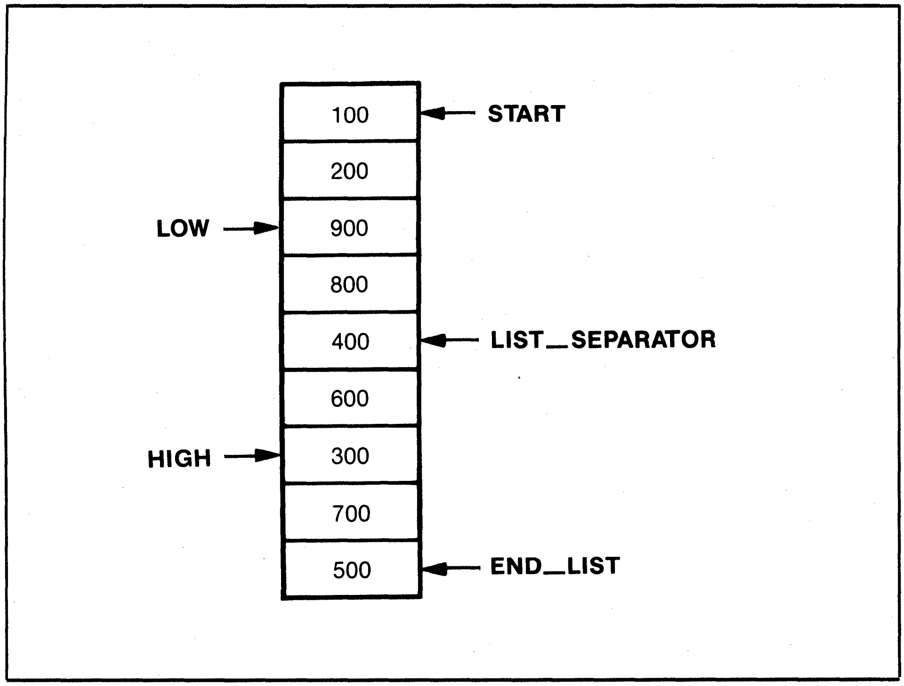


Figure 11-1. *Sequence of sublist construction using a quick sort*

When values[low] contains a value that is greater than or equal to the value contained in the list_separator the while loop terminates. The value in high is then decremented until values[high] contains a value that is less than or equal to the list_separator:

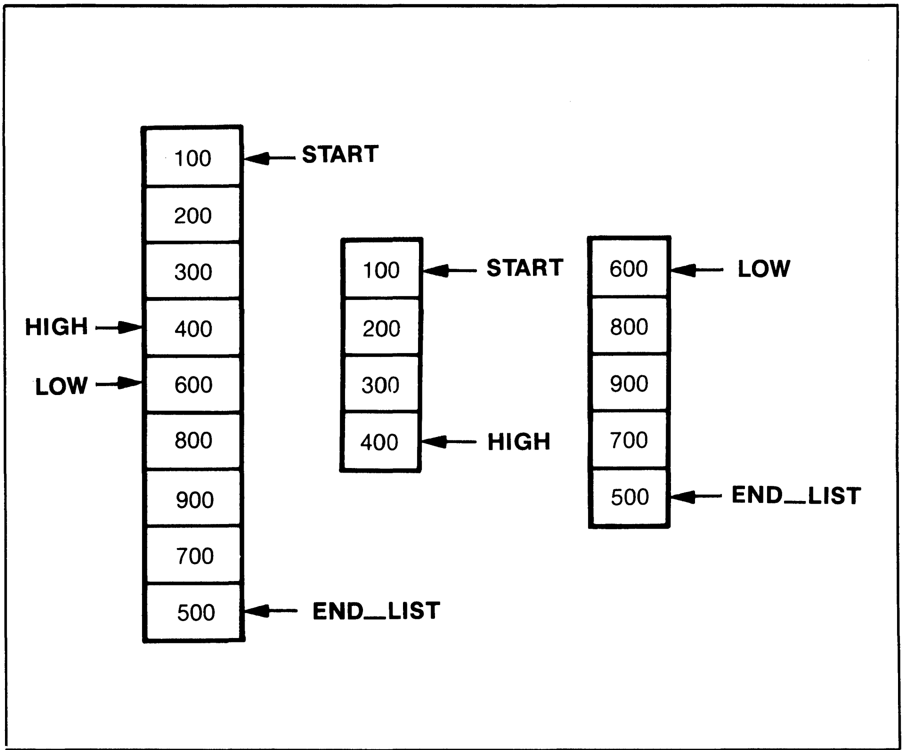


When the value contained in values[high] is less than or equal to the value contained in the list_separator the while loop terminates and the values contained in low and high are compared. If the value in low is less than the value in high, the values are exchanged; the value in low is incremented while the value in high is decremented, as shown here:

```
if (low <= high)
 {
  temp = values[low];
  values[low++] = values[high];
  values[high--] = temp;
 }
```

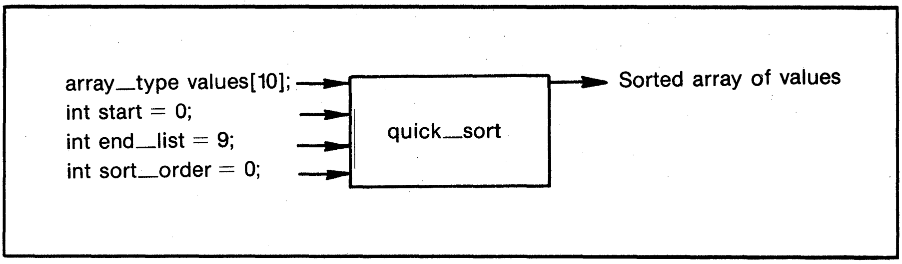
This process is repeated until low is greater than high.

Once the value in low is greater than the value in high, you have two lists. The first contains the elements from start to high, and the second contains the elements from low to end_list:



Each list then is passed recursively to the routine, and each is also subdivided into lists. This process will continue until each list contains only one element.

The following routine implements the quick sort:




```
/*
 * void quick_sort (values, first, last, order)
 *
 * Sort the array of values in the order specified.
 *
 * values (in/out): Array of values to sort.
 * first (in): Index of the first element in the list to sort.
 * last (in): Index of the last element in the list to sort.
 * order (in): Desired sorting order:
 *              0 for ascending
 *              1 for descending
 *
 * quick_sort (values, 0, 9, 1);
 *
 */

void quick_sort (array_type values[], int start,
                 int end_list, int order)
 {
  array_type temp;
  int low = start;
  int high = end_list;
  int list_separator = values [(start+end_list) / 2];

  do {
   if (! order)   /* ascending */
     {
       while (values[low] < list_separator)
         low++;

       while (values[high] > list_separator)
         high--;
     }
   else         /* descending */
    {
       while (values[low] > list_separator)
         low++;

       while (values[high] < list_separator)
         high--;
```

```
    }
  if (low <= high)
   {
    temp = values[low];
    values[low++] = values[high];
    values[high--] = temp;
   }
  }
 while (low <= high);

 if (start < high)
   quick_sort (values, start, high, order);
 if (low < end_list)
   quick_sort (values, low, end_list, order);
}
```

Arrays of Character Strings

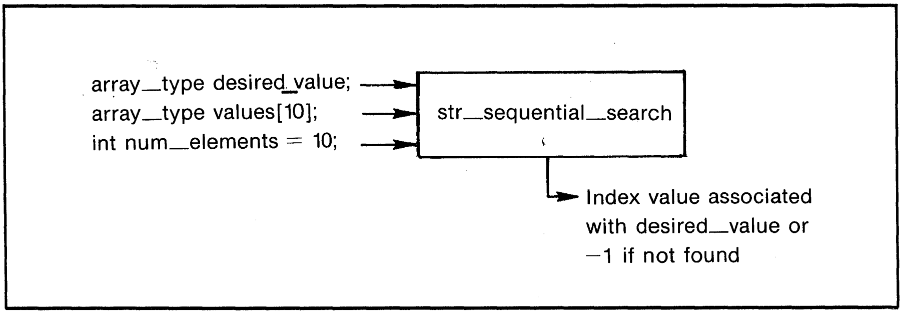
All of the arrays presented thus far have been arrays of type, int, float, double, and so on. Turbo C also allows you to create arrays of character strings (argv and env). Just as you traverse arrays of type float with an index value,

```
for (i = 0; i < 10; i++)
  printf ("%f\n", floating_point_values [i]);
```

arrays of character strings are manipulated the same way:

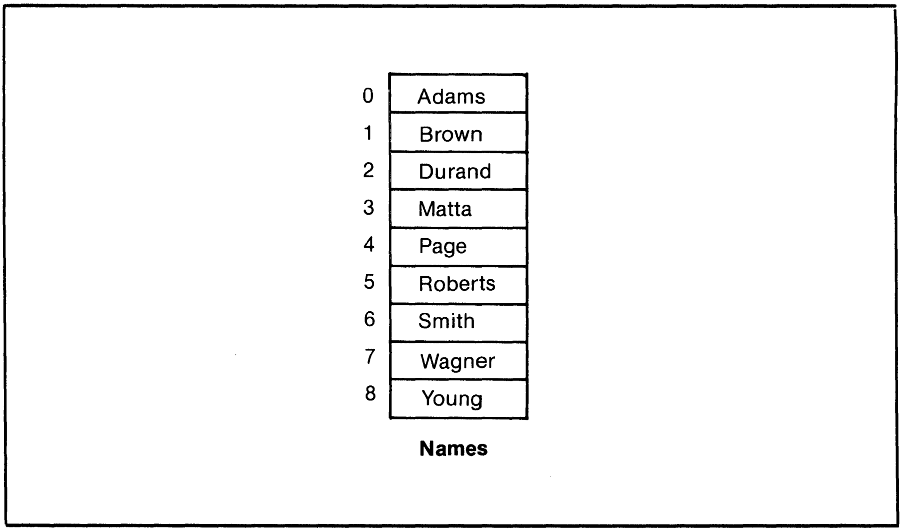
```
main (argc, argv)
  int argc;
  char *argv[];
 {
  int i;
  for (i = 0; i < argc; i++)
    printf ("%s\n", argv[i]);
 }
```

Although the arrays contain character strings, manipulation of the arrays is essentially the same. The same holds true for sorting and searching algorithms. The only difference is the code used to perform element comparisons. For example, the following program uses the routine equal_strings from Chapter 3 to implement a sequential search:




```
/*
 * str_sequential_search (value, array, num_elements)
 *
 * Sequentially search the array of values given in order to locate
 * a character string.  Return the index of the value, or -1 if the
 * value is not found.
 *
 * value (in): String to search for.
 * array (in): Array of values to examine.
 * num_elements (in): Number of elements in the array.
 *
 * index = str_sequential_search ("Monday", days, 10);
 *
 * If multiple occurrences of the same value are present in the
 * array, this routine returns the index of the first occurrence.
 *
 */

int str_sequential_search (array_type value, array_type values[],
                           int num_elements)
 {
  int i, location = -1;

  for (i = 0; (i < num_elements) && location == -1; i++)
    if (equal_strings (values[i], value, 0))
      location = i;

  return (location);
 }
```

Assuming that your array contains the following,



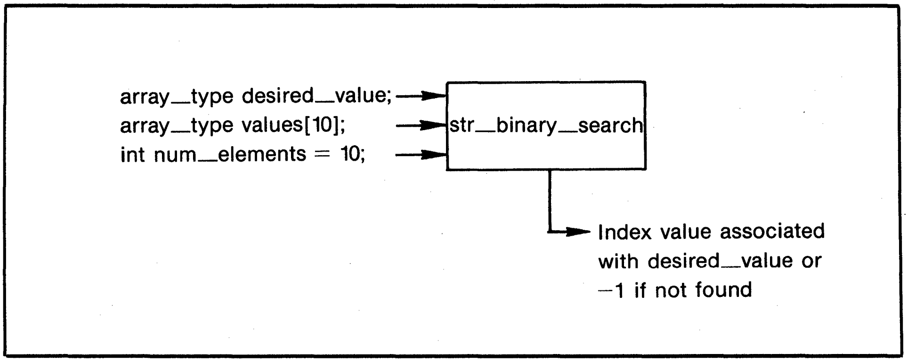
the function invocation

```
index = str_sequential_search ("Roberts", Names, 9);
```

returns the index value 5. Likewise, the invocation

```
index = str_sequential_search ("Kellie", Names, 9);
```

returns the value −1 since the array does not contain the string. The following routine uses compare_strings (presented in Chapter 3) to implement a binary search:




```
/*
 * str_binary_search (value, array, num_elements)
 *
 * Use a binary search of the array of values given in order to
 * locate a character string.  Return the index of the value, or
 * -1 if the value is not found.
 *
 * value (in): String to search for.
 * array (in): Array of values to examine.
 * num_elements (in): Number of elements in the array.
 *
 * index = str_binary_search ("Monday", days, 10);
 *
 */

int str_binary_search (array_type value, array_type values[],
                       int num_elements)
 {
  int i, found = 0;
  int high, low, mid_index, result;

  low = 0;
  high = num_elements;
  mid_index = (high + low) / 2;

  while ((! found) && (high >= low))
   {
    result = string_comp (value, values[mid_index], 0);

    if (result == 0)
      found = 1;
    else if (result == 2)
      high = mid_index - 1;
    else if (result == 1)
      low = mid_index + 1;

    mid_index = (high + low) / 2;
   }

  return ((found) ? mid_index: -1);
 }
```

The only real difference between the string-searching routines and those presented at the beginning of this chapter is the code that performs the element comparisons.

```
if (value == values[i])
```

Versus

```
if (equal_strings (value, values[i]))
```

You can indeed develop a single routine to handle both types. To do so, you have a couple of alternatives.

The first alternative is to pass, as a parameter to the routine, the address of a function that is to perform the actual element comparisons, as shown here:

```
generic_search (value, array, num_elements, compare_routine);
```

For an array of type float, this routine would be defined as follows:

```
float_compare (float a, float b)
 {
  return (a == b);
 }
```

Within a program that uses the array, you would pass the address of float_compare to the generic_search routine, as shown here:

```
typedef float array_type;

main ()
 {
  float values[10];

  int generic_search (array_type, array_type [], int, int (*)());

  int i;

  for (i = 0; i < 10; i++)
    values[i] = i * 1.0;

  printf ("Location of %f in array is %d\n", 3.0,
           generic_search (3.0, values, 10, float_compare));
 }
```

The following code implements the generic search:

```
int generic_search (array_type value, array_type values[],
                    int num_elements,
                    int (*compare)(array_type, array_type))
 {
  int i, location = -1;

  for (i = 0; (i < num_elements) && (location == -1); i++)
    if ((*compare) (value, values[i]))
      location = i;

  return (location);
 }
```

Note the definition of the routine that performs the actual comparison of array elements:

```
int (*compare)(array_type, array_type)
```

If you examine the contents of the first set of parentheses, you find that compare is a pointer. The second set of parentheses indicates that it is a pointer to a function. The type int defines the type of value returned by the function. This differs greatly from the declaration

```
int *compare (array_type, array_type)
```

which declares compare to be a function that returns a pointer to a value of type int.

Knowing this, you can later pass an array of character strings and the routine equal_strings to generic_search, as shown here:

```
index = generic_search ("MONDAY", days, 10, equal_strings);
```

Remember that, although you are using the same routine for each type, you must define the type array_type and recompile for each type of array.

Although this algorithm seems powerful because of its use of a pointer to a function, it has several drawbacks. First, you may have to create several additional functions to perform your element comparisons:

```
float_compare (float a, float b)
 {
  return (a == b);
 }

int_compare (int a, int b)
 {
  return (a == b);
 }

double_compare (double a, double b)
 {
  return (a == b);
 }
```

Remember, function invocations add overhead. This routine greatly increases the number of invocations required, since each comparison is now a function call. As such, the routine will run slower than previous routines.

A second alternative is to pass the parameter that describes the type of array, as shown here:

Value	Type
0	Nonstring array
1	Array of character strings

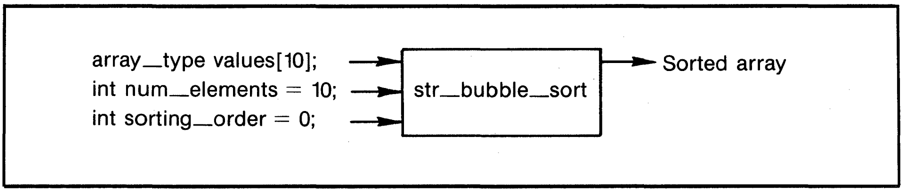
Within the routine, you simply perform a comparison based on this type, as shown here:

```
int generic_search (array_type value, array_type values[],
                    int num_elements, int type)
 {
  int i, location = -1;

  for (i = 0; (i < num_element) && (location == -1); i++)
    if ((type == 0) && (value == values[i]))
      location = i;
    else if ((type == 1) && (equal_strings (value, values[i])))
      location = i;

  return (location);
 }
```

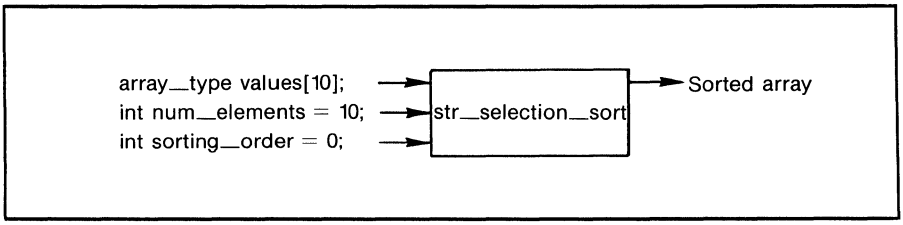
This routine requires additional function invocations. You must still recompile this routine with each array type. As such, most programmers find it simpler to develop a library of routines for string arrays and nonstring array types:

```
int_bubble_sort (values, num_elements, order);
float_bubble_sort (values, num_elements, order);
string_bubble_sort (values, num_elements, order);
```

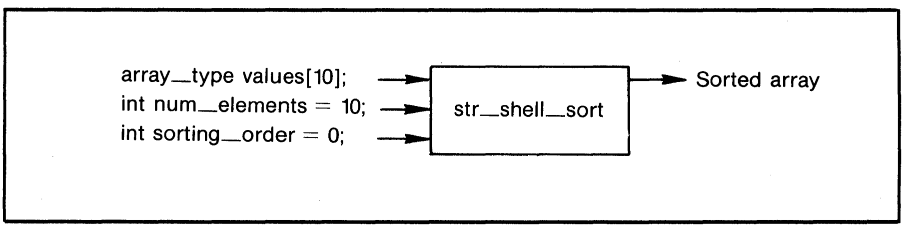
Since you have already seen how the sorting routines work, no discussion of implementation is presented here for arrays that contain character strings. The following routines implement the bubble, selection, Shell, and quick sorts for arrays of character strings:



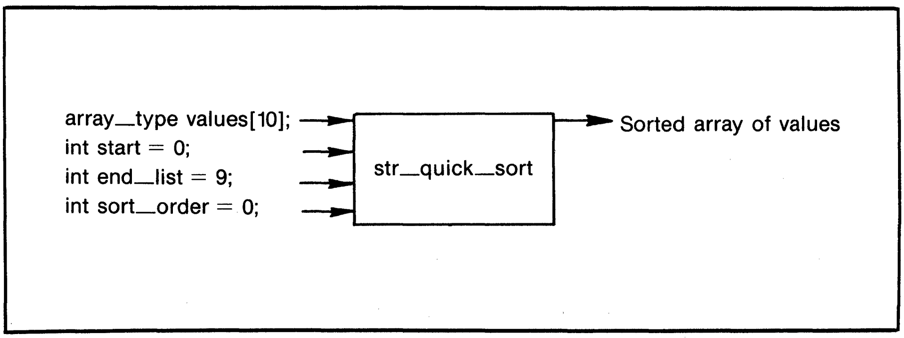

```
/*
 * void str_bubble_sort (values, num_elements, order)
 *
 * Sort the array of strings in the order specified.
 *
 * values (in/out): Array of values to sort.
 * num_elements (in): Number of elements in the array.
 * order (in): Desired sorting order:
 *             0 for ascending
 *             1 for descending
 *
 * str_bubble_sort (values, 10, 1);
 *
 */

void str_bubble_sort (array_type values[], int num_elements, int order)
 {
  array_type temp;

  void fast_exchange ();

  int i, j;

  for (i = 0; i < num_elements - 1; i++)
    for (j = i + 1; j < num_elements; j++)
      if ((! order && (string_comp (values[i], values[j]) ==1)) ||
           (order && (string_comp (values[i], values[j]) == 2)))
              fast_exchange (values[i], values[j]);
 }
```




```
/*
 * void str_selection_sort (values, num_elements, order)
 *
 * Sort the array of strings in the order specified.
 *
 * values (in/out): Array of strings to sort.
 * num_elements (in): Number of elements in the array.
 * order (in): Desired sorting order:
 *             0 for ascending
 *             1 for descending
 *
 * str_selection_sort (values, 10, 1);
 *
 */

void str_selection_sort (array_type values[],
                         int num_elements, int order)
 {
  int i, j, current;

  void fast_exchange ();

  for (i = 0; i < num_elements - 1; i++)
   {
    current = i;
    for (j = i + 1; j < num_elements; j++)
      if ((! order && (string_comp (values[current], values[j]) ==1)) ||
          (order && (string_comp (values[current], values[j]) == 2)))
             fast_exchange (values[current], values[j]);

   }
 }
```



```
/*
 * void str_shell_sort (values, num_elements, order)
 *
 * Sort the array of strings in the order specified.
 *
 * values (in/out): Array of strings to sort.
 * num_elements (in): Number of elements in the array.
 * order (in): Desired sorting order:
 *              0 for ascending
 *              1 for descending
 *
 * str_shell_sort (values, 10, 1);
 *
 */

void str_shell_sort (array_type values[],
                     int num_elements, int order)
 {
  array_type temp;

  int i, gap, exchange_occurred;

  gap = num_elements / 2;

  do
    do {
      exchange_occurred = 0;

      for (i = 0; i < num_elements - gap; i++)
        if ((! order && (string_comp (values[i], values[i+gap]) == 1)) |
           (order && (string_comp (values[i], values[i+gap]) == 2)))
          {
           temp = values[i];
           values[i] = values[i+gap];
           values[i+gap] = temp;
           exchange_occurred = 1;
          }
     }
    while (exchange_occurred);
  while (gap = gap / 2);
 }
```



```
/*
 * void str_quick_sort (values, first, last, order)
 *
 * Sort the array of values in the order specified.
 *
 * values (in/out): Array of values to sort.
 * first (in): Index of the first element in the list to sort.
 * last (in): Index of the last element in the list to sort.
 * order (in): Desired sorting order:
 *              0 for ascending
 *              1 for descending
 *
 * str_quick_sort (values, 0, 9, 1);
 *
 */

void str_quick_sort (array_type values[], int start,
                     int end_list, int order)
 {
  array_type temp;
  int low = start;
  int high = end_list;
  int list_separator_index = (start+end_list) / 2;

  void fast_exchange ();

  char list_separator [128];

  copy_string (values[list_separator_index], list_separator,
               sizeof(list_separator));

  do {
   if (! order)   /* ascending */
     {
       while (string_comp (values[low], list_separator) == 2)
         low++;

       while (string_comp (values[high], list_separator) == 1)
         high--;
     }
   else        /* descending */
    {
       while (string_comp (values[low], list_separator) == 1)
         low++;

       while (string_comp (values[high], list_separator) == 2)
         high--;
     }

   if (low <= high)
     fast_exchange (values[low++], values[high--]);
   }
  while (low <= high);
  if (start < high)
    str_quick_sort (values, start, high, order);

  if (low < end_list)
    str_quick_sort (values, low, end_list, order);
 }
```

This chapter was written with two goals. First, to present several routines that you can get up and running in a hurry. Second, and perhaps more important, to teach you how several of the most popular sorting and searching algorithms work. It also is important to point out the bsearch, qsort, and lsearch routines in the Turbo C run-time library. These routines provide generic sorting and searching functions that you can use readily within your Turbo C programs.

You might wonder, if Borland can develop generic sorting and searching routines, why is it not possible for you to do so. The answer is that you can. The difficulty becomes doing so in a manner that is still readily understandable.

Because you are already conversant with the sequential search, it will be used as a test case. However, remember that the following code uses a significant number of pointers, casts, and redirections.

As before, you must define functions to compare two values. In this case, you will be searching an array of type int and one of type float.

```
int_cmp (int *x, int *y)
 {
  if (*x == *y)
    return (1);
  else
    return (0);
 }

flt_cmp (float *x, float *y)
 {
  if (*x == *y)
    return (1);
  else
    return (0);
 }
```

Next, you must define the routine that will perform the actual search:

```
int generic_seq_search (void *value, void *values,
                        int num_elements, int width,
                        int (*compare) (void *, void *))
 {
  int i, location = -1;

  for (i = 0; (i < num_elements - 1) && location == -1; i++)
    if ((*compare)(value,(void *)((char *) values + (i*width))))
      location = i;

  return (location);
 }
```

Note that both the desired value and the array are defined as void pointers. Here is where the routine lays its generic foundation.

The next confusing fragment is the actual comparison.

```
if ((*compare)(value,(void *)((char *) values + (i*width))))
```

Since you are dealing with pointers, you are yet to be concerned with the array type. The goal in the comparison is to pass the address of the desired value along with the address of the current array element. The routine that performs the comparison is the only code fragment concerned with the array type. For example, the routine int_cmp simply uses the addresses it receives as pointers to the type int. Since this was the original goal, the routines work as desired.

This program passes arrays of type int and float to the search routine. Note that the desired value *must* be passed by address:

```
main ()
 {
  int flt_cmp (float *, float *);
  int int_cmp (int *, int *);

  int int_values[10];
  float float_values[10];

  int int_value = 2;
  float float_value = 2.0;

  int i;

  for (i = 0; i < 10; i++)
   {
    int_values[i] = i * 1;
    float_values[i] = i * 1.0;
   }

  printf ("int result %d\n",
        generic_seq_search (&int_value, int_values, 10,
                            sizeof(int), int_cmp));

  printf ("float result %d\n",
        generic_seq_search (&float_value, float_values, 10,
                            sizeof(float), flt_cmp));
 }
```

Admittedly, this code is generic. Its difficulty lies in its readability. You can apply this concept to all of the routines in this chapter. This is how the searching and sorting routines in the Turbo C run-time library work.

Selecting the proper sorting and searching algorithm has a definite impact on the execution time of your routines. Compilers that provide a sorting routine often use the quick sort. If you experiment with each routine using arrays of 30, 300, and 3000 elements, you should find that for 30 elements, the execution time of each routine is almost the same; that the bubble sort is much slower than the other sorts for 300 elements; and that the quick sort is much faster than the others for an array of 3000 elements. However, you also will find that the recursive nature of the quick sort makes it slower for smaller lists.

Each sorting routine has attributes that make it more efficient for a specific application. You will find that you can increase the speed of your sorts by examining the number of elements in the array and invoking the sort that is best suited to the application.

CHAPTER

12

Input/Output Routines

Probably the most important aspect of any computer program is its user interface. Good programs are too often under-utilized simply because users find them awkward. Unless your programs are easy to use, they will be of little use to others.

Everyone in the computer industry has a definition of "user friendly." To many, "user friendly" means a mouse-driven application. However, to most programmers, the command line is more than adequate to make a program user friendly. Writing user-friendly programs depends to a great extent on the target audience. A program that is designed for an advanced user to operate from the command line will be terribly frightening to a novice. A mouse-driven system can have tremendous overhead, which frustrates advanced users. Perhaps a more suitable goal is to make your programs "user consistent." A good program should make the user's next response obvious.

For example, assume that you need the user to enter a mailing address, as follows:

Name:
Address:
City: State: Zip:

Some programs will prompt the user a line at a time, as shown here:

```
Enter name: Kevin Shafer

Enter address:
```

To minimize surprising the user (and to help put the user at ease), a program should first display all of the prompts to which the user must respond:

```
Name:

Address:

City:                State:     Zip:
```

Next, the user can begin filling in the fields, as shown here:

```
        Name: Kevin Shafer

        Address:

        City:               State:      Zip:
```

This allows the user to build a mental image of the screen and to have a good concept as to what comes next.

If you use the routines presented in this chapter, developing user-consistent programs will be much easier. Rather than forcing you to worry about input/output (I/O) processing when you develop your programs, these routines enable you to concentrate on the task at hand. Most experienced programmers will testify that they spend the majority of their time strictly on I/O processing. The goal of this chapter is to develop powerful I/O routines once, and then to use them many times in the future.

Output Routines

The first collection of routines performs output operations. These routines output strings, integer values, and floating-point values. You do not use printf in these routines because of its slow speed and limited capabilities.

Since the routines presented in this chapter are based on the routine write_char_and_attr from Chapter 7, they bypass Turbo C's output routines and directly access the BIOS services. This enables you to control your own output. More important, by using write_char_and_attr, your programs can easily specify the display attributes of each character on the screen display. Each of the routines in this chapter allows you to specify the video display page that is currently active. Most of you will always use display page 0. However, advanced programs often write output to one display page and then select that page as active (by using routines from Chapter 7). In so doing, the output appears instantaneously.

The IBM PC and PC compatibles allow you to define the display attribute of every character on the screen. Display attributes include color, boldness, and even blinking. Each character displayed on the screen has an 8-bit attribute byte associated with it. Table 12-1 defines the function of each bit in the attribute byte.

The following program displays each of the attribute values by showing the number of the attribute that uses the attribute value. Use this program to help you select desirable attributes for your programs.

```
#include <stdio.h>

main ()
 {
  char str[4];

  int attr, i;

  for (attr = 0; attr <= 255; attr++)
   {
    int_to_ascii (attr, str);

    for (i = 0; str[i]; i++)
     {
      set_cursor_position (0, 10, 39 + i);
      write_char_and_attr (0, str[i], attr, 1);
     }
    getchar();
   }
 }
```

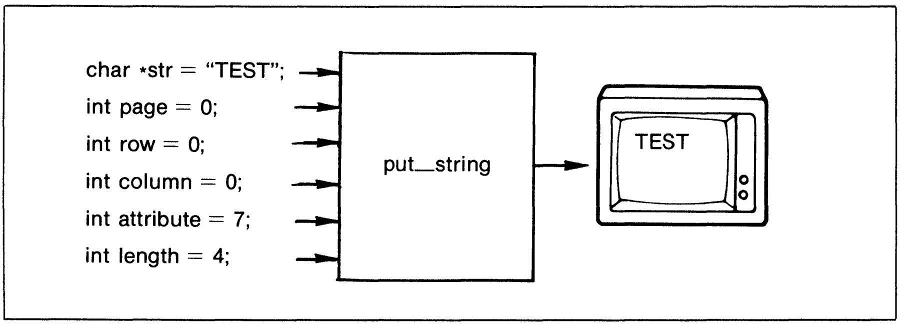
The routine put_string enables you to specify the screen row and column location, along with the display attributes for a given string.

```
main ()
 {
  int page = 0, row = 10, column = 10, attribute = 7;

  put_string ("User Prompt:", page, row, column, attribute, 80);
 }
```

Table 12-1. Functions of Bits in Attribute Byte

Bit	Color
0	Blue foreground
1	Green foreground
2	Red foreground
3	Bold
4	Blue background
5	Green background
6	Red background
7	Blinking

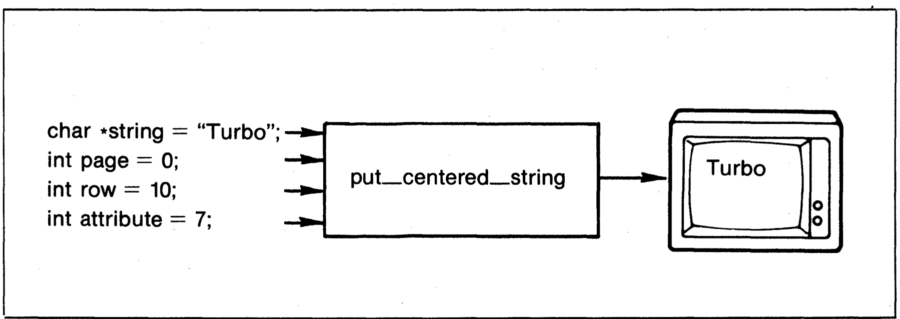
By using your previously developed library of routines, put_string is quite straightforward.




```
/*
 * void put_string (string, page, row, column, attribute, length)
 *
 * Output a character string at the row and column specified.
 *
 * string (in): String to be displayed.
 * page (in): Desired video display page.
 * row (in): Screen row to write the string at.
 * column (in): Screen column to write the string at.
 * attribute (in): Video display attribute for the string.
```

```
 * length (in): Maximum number of characters to display.
 *
 * put_string ("User Prompt:", 0, 10, 10, 7, 11);
 *
 */

void put_string (char *str, int page, int row,
                 int column, int attribute, int length)
 {
  int count = 0;
  void write_char_and_attr (int, int, int, int);
  void set_cursor_position (int, int, int);

  while ((*str) && (count < length))
    {
      set_cursor_position (page, row, column + count++);
      write_char_and_attr (page, *str++, attribute, 1);
    }
 }
```

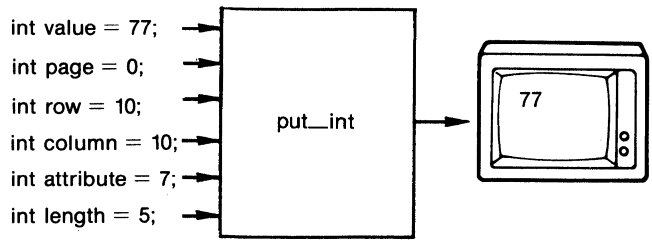
The routine put_centered_string enables you to output a character string centered on the specified row. The routine is based on an 80-column screen display.




```
/*
 * void put_centered_string (string, page, row, attribute)
 *
 * Center a character string on the screen row specified.
 *
 * string (in): Character string to be displayed centered.
 * page (in): Desired video display page.
 * row (in): Desired screen row position for the string.
 * attribute (in): Desired display attribute.
 *
 * put_centered_string ("Turbo C Programmer's Library", 0, 5, 7);
 *
 * This routine assumes an 80 column screen display.
 *
 */
```

```
void put_centered_string (char *str, int page,
                          int row, int attribute)
 {
  int count = 0;
  int column;

  void write_char_and_attr (int, int, int, int);
  void set_cursor_position (int, int, int);

  while (*(str + count))
    count++;

  column = 39 - (count / 2);

  while (*str)
    {
      set_cursor_position (page, row, column++);
      write_char_and_attr (page, *str++, attribute, 1);
    }
 }
```

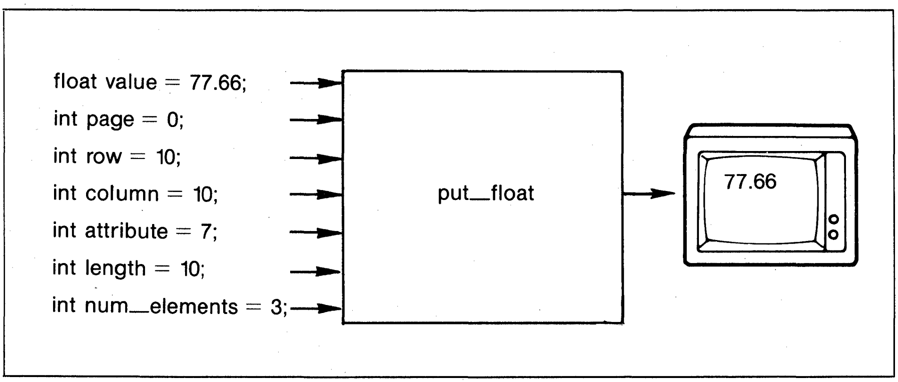
The routine put_int enables you to output an integer value at a specific row and column location. As before, this routine also enables you to specify the display attribute and desired video page.




```
/*
 * void put_int (value, page, row, column, attribute, length)
 *
 * Output an integer value at the row and column specified.
 *
 * value (in): Integer value to be displayed.
 * page (in): Desired video display page.
 * row (in): Screen row to write the string at.
 * column (in): Screen column to write the string at.
 * attribute (in): Video display attribute for the string.
 * length (in): Maximum number of characters to display.
```

```
 *
 * put_int (12345, 0, 10, 10, 7, 11);
 *
 */

void put_int (int value, int page, int row,
              int column, int attribute, int length)
 {
  int count = 0;

  void write_char_and_attr (int, int, int, int);
  void set_cursor_position (int, int, int);

  char str[132];

  int_to_ascii (value, str);

  while ((str[count]) && (count < length))
    {
      set_cursor_position (page, row, column + count);
      write_char_and_attr (page, str[count++], attribute, 1);
    }
 }
```

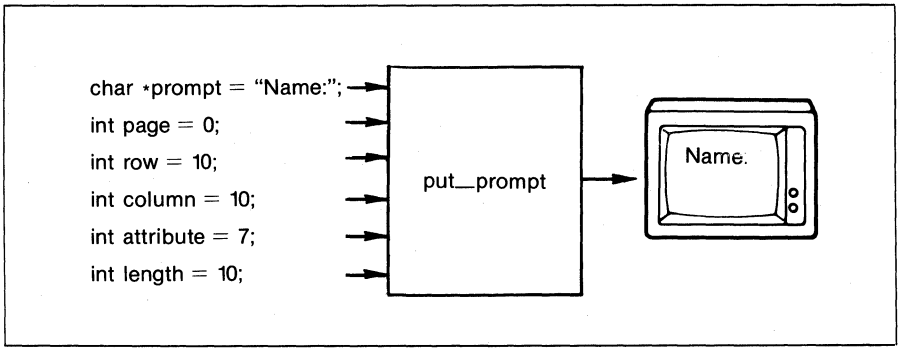
The routine put_float places a floating-point value anywhere on the screen. The field-length specifier enables you to suppress the display of insignificant digits (values of type float have seven digits of significance).




```
/*
 * void put_float (value, page, row, column, attribute,
 *                 length, num_decimals)
 *
 * Output a floating point value at the row and column specified.
 *
```

```
 * value (in): Floating point value to be displayed.
 * page (in): Desired video display page.
 * row (in): Screen row to write the string at.
 * column (in): Screen column to write the string at.
 * attribute (in): Video display attribute for the string.
 * length (in): Maximum number of characters to display.
 * num_decimals (in): Maximum number of digits to the right of
 *                    the decimal point.
 *
 * put_float (12345.67, 0, 10, 10, 7, 11, 2);
 *
 */

void put_float (float value, int page, int row,
                int column, int attribute, int length,
                int num_decimals)
 {
  int count = 0;

  void write_char_and_attr (int, int, int, int);
  void set_cursor_position (int, int, int);

  char *str;

  int sign, decimal_position;

  char *ecvt (double, int, int *, int *);

  str = ecvt ((double) value, length, &decimal_position, &sign);

  while ((str[count]) && (count < length))
    {
      set_cursor_position (page, row, column + count);
      if (count == decimal_position)
       {
        write_char_and_attr (page, '.', attribute, 1);
        count++;
       }
      else
        write_char_and_attr (page, str[count++], attribute, 1);

      if (count > (num_decimals + decimal_position))
        break;
    }
 }
```

The routine put_prompt enables you to output prompt text to the screen in a manner similar to put_string. Unlike put_string, put_prompt displays the text and then removes any other characters remaining on that line. Thus, you must not worry about the previous screen contents when you display your prompt.




```
/*
 * void put_prompt (prompt, page, row, column, attribute, length)
 *
 * Output a prompt at the row and column specified. Clear the
 * text remaining on the line following the prompt.
 *
 * prompt (in): Prompt to be displayed.
 * page (in): Desired video display page.
 * row (in): Screen row to write the string at.
 * column (in): Screen column to write the string at.
 * attribute (in): Video display attribute for the string.
 * length (in): Maximum number of characters to display.
 *
 * put_prompt ("Enter Name:", 0, 10, 10, 7, 15);
 *
 */

void put_prompt (char *str, int page, int row,
                 int column, int attribute, int length)
 {
  int count = 0;
  void write_char_and_attr (int, int, int, int);
  void set_cursor_position (int, int, int);

  while ((*str) && (count < length))
    {
      set_cursor_position (page, row, column + count++);
      write_char_and_attr (page, *str++, attribute, 1);
    }

  /* clear characters remaining on the line */

  column += count;

  while (column < 80)
    {
      set_cursor_position (page, row, ++column);
      write_char_and_attr (page, 0, 0, 1);
    }
 }
```

Input Routines

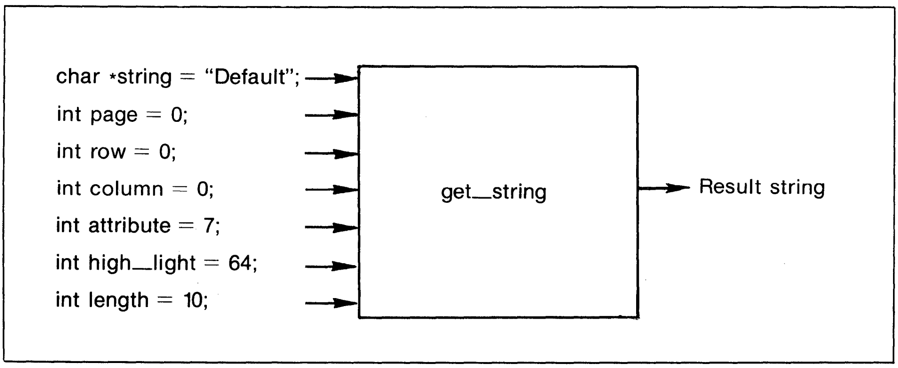
The next collection of routines enables you to get a character string, integer value, or floating-point value from the user in a controlled manner. Each of the routines presented in this section is based on the routine get_string.

Many C programs often prompt a user to enter data that is stored internally as a character string. Because C programs use strings so frequently, a powerful function that enables you to control string input can be quite convenient.

The following routine provides several significant features. It enables you to specify the screen location and maximum number of characters in the string.

The routine allows you to provide a default string for the user to either select or edit. For example, if you are developing a mailing-list program and the most frequently used state is AZ, the user should not have to type in these letters with each new entry. Instead, you provide the letters "AZ" as the default.

Because you can provide a default string, the user should be able to edit it. The routine get_string enables you to use the right arrow and left arrow keys, the BACKSPACE key, and the INS key to insert text. Because all of the routines in this section have been based on get_string, each routine provides all of these editing capabilities.

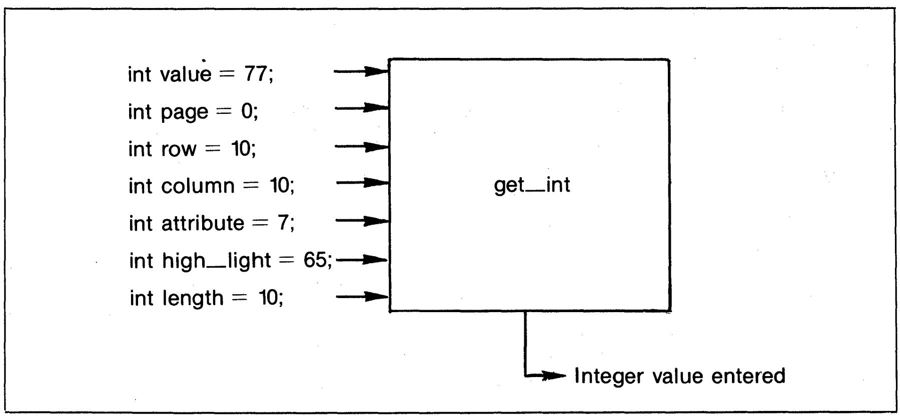


```
/*
 * get_string (string, page, row, column, attribute,
 *             high_light, length);
 *
 * Display a default string to the user allowing the user to
 * edit the string as desired.
 *
 * string (in): Default string which once edited becomes result.
 * page (in): Desired video display page.
 * row (in): Screen row to input the string at.
 * column (in): Screen column to input the string at.
 * attribute (in): Display attribute of the string.
 * high_light (in): Display attribute of the blank space remaining
 *                  in the input field.
 * length (in): Maximum number of characters in the input field.
 *
 * get_string (string, 0, 10, 10, 7, 65, 10);
 *
 */

void get_string (char *str, int page, int row,
                 int column, int attribute,
                 int high_light, int length)
 {
  void write_char_and_attr (int, int, int, int);
  void set_cursor_position (int, int, int);
  int no_echo_read (void);
  int get_shift_state (void);

  int i, count = 0, done = 0, letter, scan_code;

  /* display the default string */
  while ((str[count]) && (count < length))
    {
      set_cursor_position (page, row, column + count);
      write_char_and_attr (page, str[count++], attribute, 1);
    }

  set_cursor_position (page, row, column + count);

  if (length - count)
    write_char_and_attr (page, 32, high_light, length - count);

  count = 0;
  set_cursor_position (page, row, column + count);

  while (! done)
    {
      letter = no_echo_read ();      /* get the keystroke */

      switch (letter) {
        case 8: /* back space */
                if (count > 0)       /* no characters to delete */
                 {
                  if (count + 1 == length)
                   {
                    if (str[count])
                      str[count] = '\0';
```

```
              else
                set_cursor_position (page, row, column + --count);

              str[count] = '\0';

              write_char_and_attr (page, 32, high_light, 1);

              break;
             }
            else              /* shift all following char down */
             {
               for (i = --count + 1; i < length; i++)
                 str[i-1] = str[i];

               i = 0;
               while ((str[i]) && (i < length))
                {
                 set_cursor_position (page, row, column + i);
                 write_char_and_attr (page, str[i++], attribute, 1
                }

               set_cursor_position (page, row, column + i);

               if (length > i)
                 write_char_and_attr (page, 32, high_light,
                                      length - i);

               set_cursor_position (page, row, column + count);
             }
           }

         break;

case 13: /* carriage return */
         done = 1;
         break;

case 0:  scan_code = no_echo_read ();
         if (scan_code == 77)  /* right arrow */
          {
           if ((str[count]) && ((count + 1) != length))
            {
             write_char_and_attr (page, str[count++], attribute,
             set_cursor_position (page, row, column + count);
            }
          }
         else if (scan_code == 75)  /* left arrow */
          if (count)
            set_cursor_position (page, row, column + --count);
         break;

default: if (get_shift_state () & 128)
           {
            for (i = length -1; i > count; i--)
              str[i] = str[i-1];

            for (i = count+1; i < length; i++)
               if (str[i])
```

```
                        {
                          set_cursor_position (page, row, column + i);
                          write_char_and_attr (page, str[i], attribute,
                        }
                      else
                        break;

                    set_cursor_position (page, row, column+count);
                  }

                str[count] = letter;
                write_char_and_attr (page, letter, attribute, 1);
                if (count + 1 != length)
                   set_cursor_position (page, row, column + ++count);

                break;
       }
     }

    for (i = 0; str[i] && (i < length); i++)
       count++;

   str[count] = '\0';
 }
```

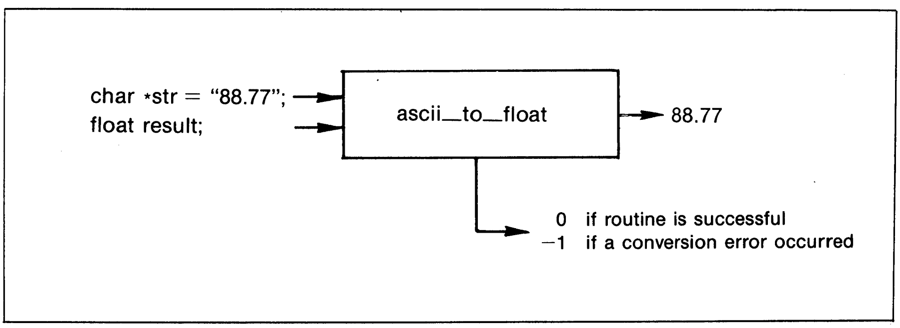
The routine get_int enables you to get an integer value from the user at any location on the screen. This routine uses the routines ascii_to_int and int_to_ascii (presented previously in this text). This code implements get_int:




```
/*
 * get_int (value, page, row, column, attribute,
 *              high_light, length);
 *
 * Display a integer value to the user allowing the user to
 * edit the value as desired.
```

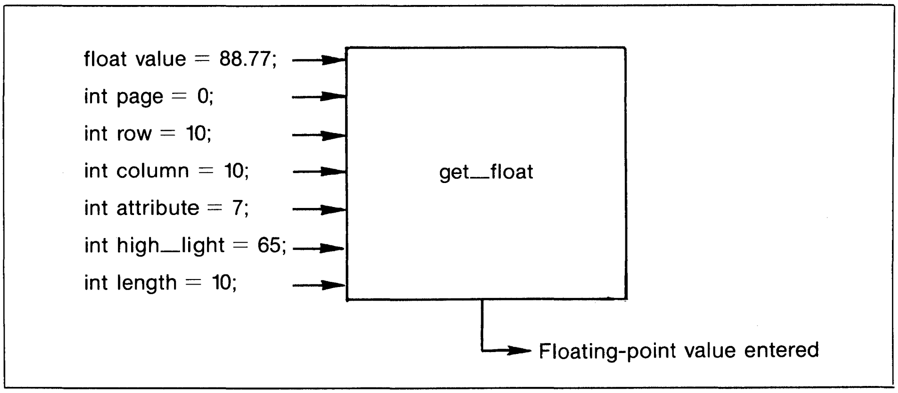
```
 *
 * value (in): Default value which once edited becomes result.
 * page (in): Desired video display page.
 * row (in): Screen row to input the value at.
 * column (in): Screen column to input the value at.
 * attribute (in): Display attribute of the value.
 * high_light (in): Display attribute of the blank space remaining
 *                  in the input field.
 * length (in): Maximum number of characters in the input field.
 *
 * result = get_int (12345, 0, 10, 10, 7, 65, 10);
 *
 */

get_int (int value, int page, int row, int column,
         int attribute, int high_light, int length)
 {
  void get_string (char *, int, int, int, int, int, int);

  char str[132];

  int done = 0;
  int status;

  while (! done)
   {
     int_to_ascii (value, str);
     get_string (str, page, row, column, attribute,
                 high_light, length);

     status = ascii_to_int (str, &value);

     if (status != -1)
       done = 1;
   }

  return (value);
 }
```

The routine get_float allows you to get a floating-point value from the user. The routine requires the routine ascii_to_float, which converts an ASCII representation of a floating-point value to a numeric value, as shown here:



```
/*
 * ascii_to_float (string, result)
 *
 * Convert an ASCII representation of a floating point value
 * to its numeric equivalent.
 *
 * string (in): String to convert.
 * result (out): Floating point result.
 *
 * status = ascii_to_float ("123.333", &result);
 *
 * If successful, ascii_to_float returns the value 0.
 *
 */

ascii_to_float (char *str, float *result)
 {
  int count, sign = 1;

  double pow10(int);

  *result = 0.0;

  while (*str == ' ')
    str++;

  if ((*str == '-') || (*str == '+'))
    sign = (*str++ == '-') ? -1: 1;

  while (*str)
    if ((*str >= '0') && (*str <= '9'))
      *result = *result * 10.0 + (*str++ - '0');
    else if (*str++ == '.')
      break ;
    else
      return (-1);

  if (*str)
    for (count = 1; *str; ++count, ++str)
    if ((*str >= '0') && (*str <= '9'))
      *result = *result + ((*str - '0') / pow10(count));
     else
       return (-1);

  *result = *result * sign;

  return (0);
 }
```

This routine implements get_float:




```
/*
 * get_float (value, page, row, column, attribute,
 *            high_light, length);
 *
 * Display a floating point value to the user allowing the user to
 * edit the value as desired.
 *
 * value (in): Default value.
 * page (in): Desired video display page.
 * row (in): Screen row to input the value at.
 * column (in): Screen column to input the value at.
 * attribute (in): Display attribute of the value.
 * high_light (in): Display attribute of the blank space remaining
 *                  in the input field.
 * length (in): Maximum number of characters in the input field.
 *
 * result = get_float (12345.67, 0, 10, 10, 7, 65, 10);
 *
 */

float get_float (float value, int page, int row, int column,
                 int attribute, int high_light, int length)
```

```
{
 void get_string (char *, int, int, int, int, int, int);
 char *ecvt (double, int, int *, int *);
 int ascii_to_float (char *, float *);

 char *str;

 int done = 0, decimal_pt, sign;

 int status, i, len;

 while (! done)
  {
    str = ecvt ((double) value, length+1, &decimal_pt, &sign);

    for (len = 0; str[len]; len++) ;

    for (i = len; i > decimal_pt; i--)
      str[i] = str[i - 1];

    str[decimal_pt] = '.';

    if (sign)
     {
      for (i = len+1; i > 0; i--)
        str[i] = str[i - 1];

      str[0] = '-';
     }

    get_string (str, page, row, column, attribute,
                high_light, length);

    status = ascii_to_float (str, &value);

    if (status != -1)
      done = 1;
  }

 return (value);
}
```

User-Consistent I/O

Programs should provide a constant interface that minimizes the possibility of user surprise (or confusion). Consider the previous example of the mailing-list program:

```
Name:

Address:

City:                    State:     Zip:
```

You can produce code to obtain this information from the user, as shown here:

```
main ()
 {
  void put_string (char *, int, int, int, int, int);

  put_string ("Name:", 0, 1, 10, 1, 10);
  put_string ("Address:", 0, 3, 10, 1, 10);
  put_string ("City:", 0, 5, 10, 1, 10);
  put_string ("State:", 0, 5, 30, 1, 10);
  put_string ("Zip:", 0, 5, 50, 1, 10);
 }
```

Although this program is much improved over a one-line-at-a-time interface discussed at the beginning of this chapter, you can make it even better. First, the program should make the current prompt (such as Name:) distinct from other prompts by changing the display attribute of the current prompt. The program should display a single line of explanatory text at the bottom of the screen, as shown here:

```
Name:

Address:

City:                State:     Zip:




Enter your full name (Example: Tom Burns)
```

Once the user enters the data required for that entry, the prompt attribute should reset and the descriptive (or help text) should disappear.

The following routines are basically all-in-one routines that allow you to prompt the user for a string, integer, or floating-point value. These routines enable you to specify the following:

- PROMPT (such as Name:)—Row, Column, Attribute
- Input/Output value (such as namestr)—Row, Column, Attribute
- Help or Descriptive text—Row, Column, Attribute

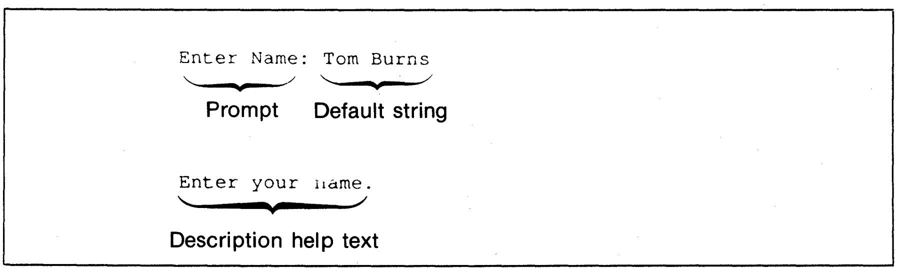
Before examining these routines, consider a possible invocation of get_prompted_string.

```
char str[132] = "Tom Burns";

get_prompted_string (str, 0, 10, 20, 30, 7, 64, "Enter Name:",
                     10, 7, 13, "Enter your name.", 23, 7,
                     40, 7);
```

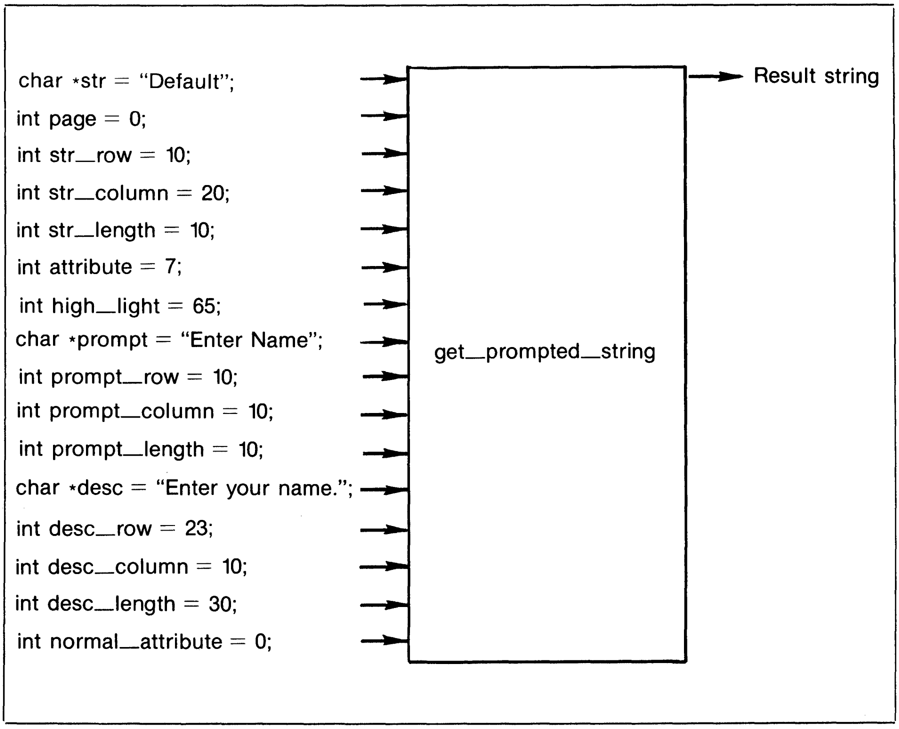
Upon invocation, this program displays the following:

```
Enter Name: Tom Burns




Enter your name.
```

The following helps you to relate the parameters of get_prompted_string to the actual display output:



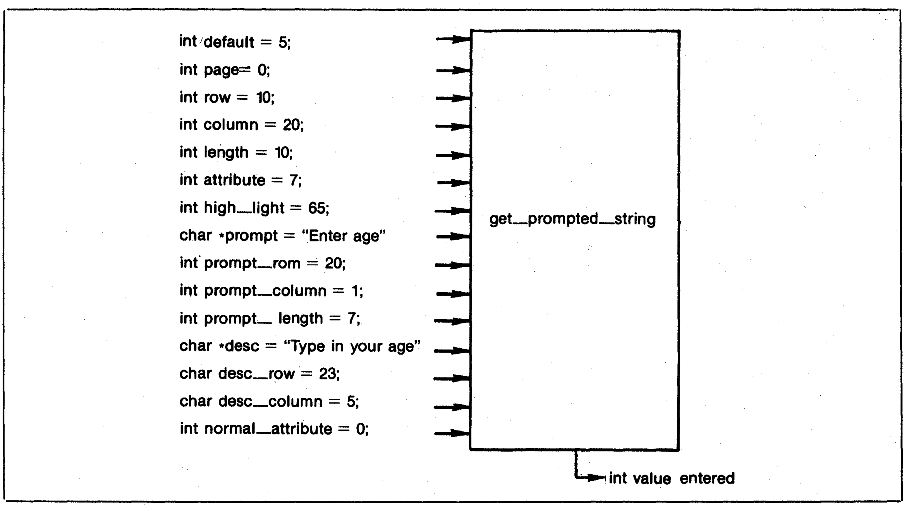
This routine implements get_prompted_string:




```
/*
 * void get_prompted_string (str, page, str_row, str_column,
 *   str_length, attribute, high_light, prompt, prompt_row,
 *   prompt_column, prompt_length, desc, desc_row, desc_column,
 *   desc_length, normal_attribute)
 *
 * Prompt the user for a character string providing a default
 * for editing purposes.  Display help text as specified which
 * is erased from the screen once the data entry is complete.
 *
 * str (in/out): Default string which once edited becomes result.
 * page (in): Desired video display page.
 * str_row (in): Screen row for string display.
 * str_column (in): Screen column for string display.
 * str_length (in): Maximum number of characters in the string.
 * attribute (in): Display attribute of string.
 * high_light (in): Display attribute of empty field space.
 * prompt (in): Desired user prompt.
 * prompt_row (in): Screen row for user prompt.
 * prompt_column (in): Screen column for user prompt.
 * prompt_length (in): Maximum number of characters in prompt.
 * desc (in): Help text.
 * desc_row (in): Screen row of the help text.
 * desc_column (in): Screen column of the help text.
 * desc_length (in): Maximum number of characters in help text.
```

```
 * normal_attribute (in): Current background color attribute.
 *
 * get_prompted_string (str, 0, 10, 20, 30, 7, 64, "Enter Name:",
 *            10, 7, 13, "Enter your first name.", 23, 7, 40, 7);
 *
 */

void get_prompted_string (char *str, int page, int str_row,
     int str_column, int str_length, int attribute, int high_light,
     char *prompt, int prompt_row, int prompt_column, int prompt_length,
     char *desc, int desc_row, int desc_column, int desc_length,
     int normal_attribute)
 {
  void get_string (char *, int, int, int, int, int, int);
  void put_string (char *, int, int, int, int, int);
  void put_prompt (char *, int, int, int, int, int);

  put_prompt (prompt, page, prompt_row, prompt_column,
              high_light, prompt_length);

  put_string (desc, page, desc_row, desc_column, high_light,
              desc_length);

  get_string (str, page, str_row, str_column, attribute,
              high_light, str_length);

  put_string (prompt, page, prompt_row, prompt_column,
              normal_attribute, prompt_length);

  put_string (desc, page, desc_row, desc_column, 0, desc_length);
 }
```

This routine implements get_prompted_int:



```
/*
 * int get_prompted_int (value, page, row, column,
 *   length, attribute, high_light, prompt, prompt_row,
 *   prompt_column, prompt_length, desc, desc_row, desc_column,
 *   desc_length, normal_attribute)
 *
 * Prompt the user for an integer value  providing a default
 * for editing purposes.  Display help text as specified which
 * is erased from the screen once the data entry is complete.
 *
 * value (in): Default value for editing.
 * page (in): Desired video display page.
 * row (in): Screen row for value display.
 * column (in): Screen column for value display.
 * length (in): Maximum number of characters in the value.
 * attribute (in): Display attribute of string.
 * high_light (in): Display attribute of empty field space.
 * prompt (in): Desired user prompt.
 * prompt_row (in): Screen row for user prompt.
 * prompt_column (in): Screen column for user prompt.
 * prompt_length (in): Maximum number of characters in prompt.
 * desc (in): Help text.
 * desc_row (in): Screen row of the help text.
 * desc_column (in): Screen column of the help text.
 * desc_length (in): Maximum number of characters in help text.
 * normal_attribute (in): Current background color attribute.
 *
 * age = get_prompted_int (27, 0, 10, 20, 30, 7, 64, "Enter Age:",
 *         10, 7, 13,"Enter age and press Enter.", 23, 7, 40, 7);
 *
 */

int get_prompted_int (int value, int page, int row,
       int column, int length, int attribute, int high_light,
       char *prompt, int prompt_row, int prompt_column,
       int prompt_length, char *desc, int desc_row, int desc_column,
       int desc_length, int normal_attribute)
 {
  void get_string (char *, int, int, int, int, int, int);
  void put_string (char *, int, int, int, int, int);
  void put_prompt (char *, int, int, int, int, int);

  int get_int (int, int, int, int, int, int, int);

  put_prompt (prompt, page, prompt_row, prompt_column,
              high_light, prompt_length);

  put_string (desc, page, desc_row, desc_column, high_light,
              desc_length);

  value = get_int (value, page, row, column, attribute,
                   high_light, length);

  put_string (prompt, page, prompt_row, prompt_column,
              normal_attribute, prompt_length);

  put_string (desc, page, desc_row, desc_column, 0, desc_length);

  return (value);
 }
```

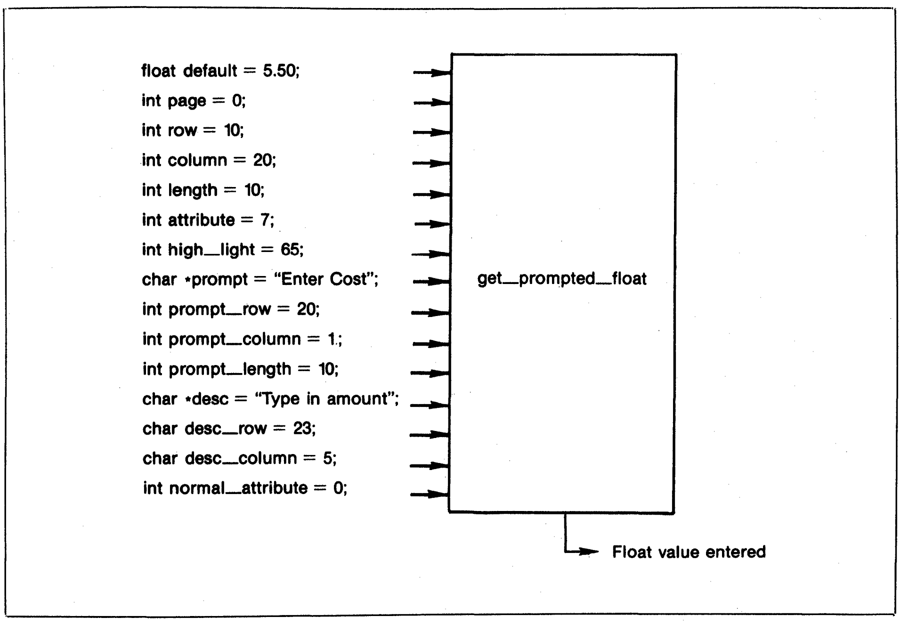
The routine get_prompted_float allows you to obtain a floating-point value from the user, as shown here:

```
float value;

value = get_prompted_float (45000.00, 0, 10, 20, 10, 14, 64,
       "Enter Salary:", 10, 5, 13, "Enter your current salary.",
        23, 7, 40, 7);
```

Upon invocation, this program will display the following:

```
      Enter Salary: 45000.00





      Enter your current salary.
```

This code implements get_prompted_float:



```
/*
 * float get_prompted_float (value, page, row, column,
 *   length, attribute, high_light, prompt, prompt_row,
 *   prompt_column, prompt_length, desc, desc_row, desc_column,
 *   desc_length, normal_attribute)
 *
 * Prompt the user for a floating point value providing a default
 * for editing purposes.  Display help text as specified which
 * is erased from the screen once the data entry is complete.
 *
 * value (in): Default value for editing.
 * page (in): Desired video display page.
 * row (in): Screen row for value display.
 * column (in): Screen column for value display.
 * length (in): Maximum number of characters in the value.
 * attribute (in): Display attribute of string.
 * high_light (in): Display attribute of empty field space.
 * prompt (in): Desired user prompt.
 * prompt_row (in): Screen row for user prompt.
 * prompt_column (in): Screen column for user prompt.
 * prompt_length (in): Maximum number of characters in prompt.
 * desc (in): Help text.
 * desc_row (in): Screen row of the help text.
 * desc_column (in): Screen column of the help text.
 * desc_length (in): Maximum number of characters in help text.
 * normal_attribute (in): Current background color attribute.
 *
 * salary = get_prompted_float (27, 0, 10, 20, 30, 7, 64,
 *     "Enter Salary:", 10, 7, 13,"Enter your current salary.",
 *      23, 7, 40, 7);
 *
 */

float get_prompted_float (float value, int page, int row,
       int column, int length, int attribute, int high_light,
       char *prompt, int prompt_row, int prompt_column, int prompt_length
       char *desc, int desc_row, int desc_column, int desc_length,
       int normal_attribute)
 {
  void get_string (char *, int, int, int, int, int, int);
  void put_string (char *, int, int, int, int, int);
  void put_prompt (char *, int, int, int, int, int);

  float get_float (float, int, int, int, int, int, int);

  put_prompt (prompt, page, prompt_row, prompt_column,
              high_light, prompt_length);

  put_string (desc, page, desc_row, desc_column, high_light,
              desc_length);

  value = get_float (value, page, row, column, attribute,
                     high_light, length);

  put_string (prompt, page, prompt_row, prompt_column,
              normal_attribute, prompt_length);

  put_string (desc, page, desc_row, desc_column, 0, desc_length);

  return (value);
  }
```

Probably the most important collection of routines that you can place into a library are those that perform I/O. Build on the routines presented in this chapter and your I/O processing should become much easier.

CHAPTER

13

Dynamic Memory

Chapter 10 examined arrays and the manipulation of arrays within Turbo C. Chapter 11 extended array manipulation to sorting and searching operations. Although you can often use arrays effectively within most applications, many instances occur where fixed array sizes cause programs to be restrictive.

Consider this example. A program tracks account balances for 100 clients by using the arrays shown here:

```
main ()
 {
  int employee_id [100];

  float account_balances [100];

  float accounts_rec [100];

  /* program code here */
 }
```

Each time the program must update an account balance, it searches the array customer_id for the corresponding customer index. Once it finds the array, the program uses the index to update the accounts receivable array. Although this algorithm seems effective, it encounters problems when it gets to customer 101. The arrays no longer provide adequate storage space. You must now modify the program.

If you have based your array declarations and program loops on macro constants,

```
main ()
 {
  int employee_id [MAX_CUSTOMER];

  float account_balances [MAX_CUSTOMER];

  float accounts_rec [MAX_CUSTOMER];

  /* program code here */
 }
```

you can modify a single statement in order to increase the size of your arrays.

```
#define MAX_CUSTOMER 150
```

Although this simple change will successfully modify the program, the code must still be recompiled. This step is relatively easy if you are a programmer. However, if you have distributed the executable files to many end users, you must now build a new executable file each time customer bases change.

A possible solution is to allocate a large amount of space for more array entries than could ever possibly exist, as shown here:

```
#define MAX_CUSTOMER 10000
```

Although this solves one problem, it unfortunately creates several additional ones. In this case, you have wasted considerable memory on empty array elements. If you have multiple arrays that must be maintained, you will soon run out of memory.

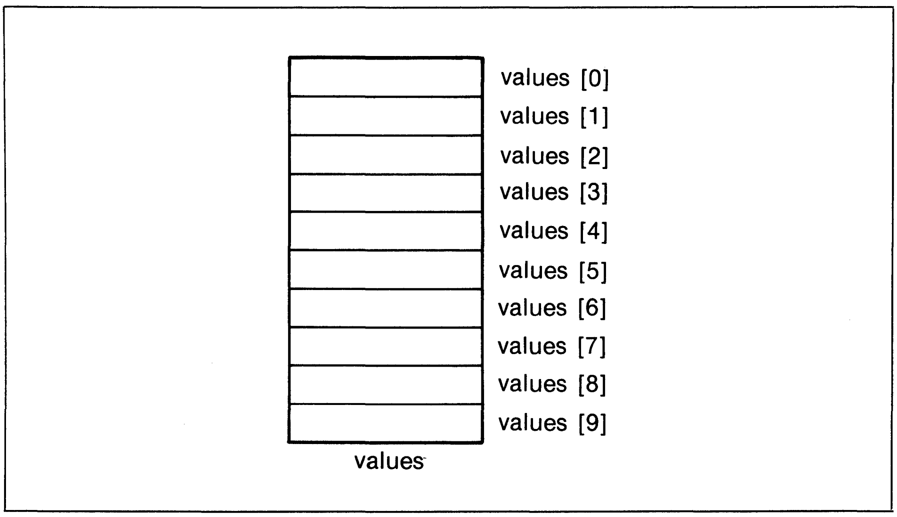
Dynamic variables provide a data structure that can actually grow or shrink by itself as your needs require. This reduces wasted memory space and prevents a program's storage requirements from being restricted. If you place elements into the dynamic lists in a specific manner, you can greatly reduce the number of sorting and searching algorithms that you may later have to perform. This chapter presents several algorithms that are commonly used in dynamic-list manipulation. Although in most cases you must modify these routines to suit your specific needs, the routines will provide a foundation on which you can build.

Dynamic Lists

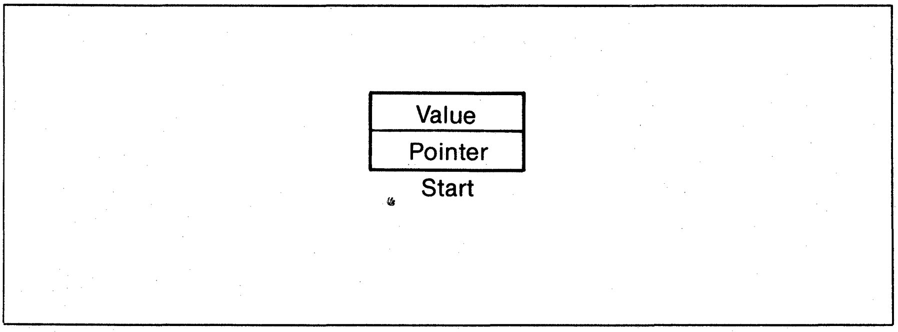
Each time you use arrays, you must specify the storage requirements during program development. The array declaration

```
int values[10];
```

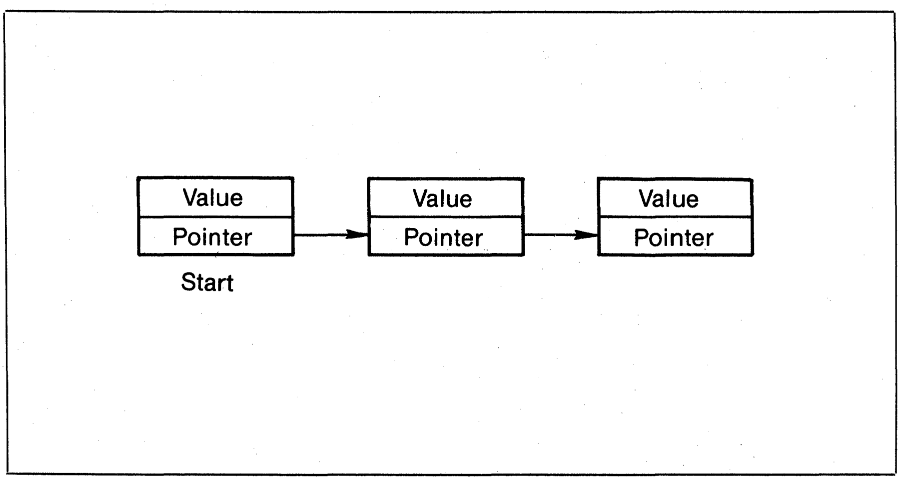
creates an array with space for ten values, as shown here:



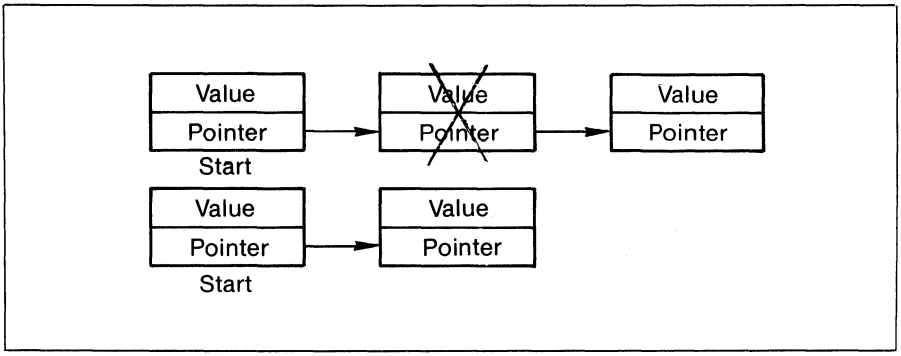
A dynamic list starts with a single element called a *node*. This element serves as the start (or head) of the list, as shown here:



As you place values into the list, you simply create and connect additional nodes; as shown here:



If an element in the list goes away (is no longer needed), you shrink the list, as shown next:



To create a linked list, you must first define a structure that provides storage space not only for the value to store, but also for a pointer to the next entry in the list. If you want to store customer account balances, your structure might contain the following:

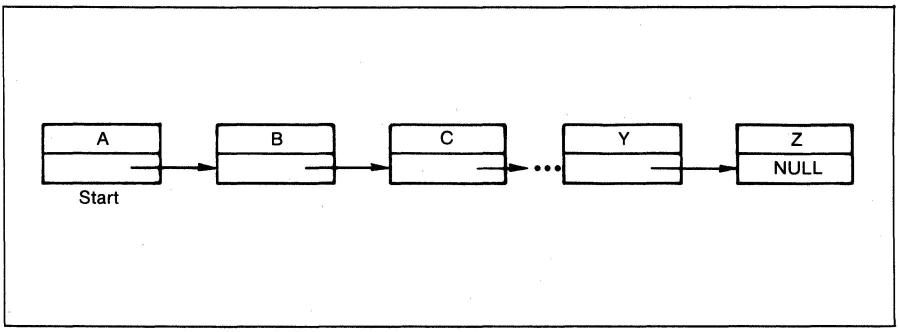
```
struct customers {
  char name[32];
  char phone[11];
  char address[32];
  char city[15];
  char state[3];
  char zip[10];
  float balance;
  struct customers *next;
 };
```

Note that the pointer within the structure is a pointer to a structure of the same type. This should make sense because all of the elements in the list are the same. A structure containing a pointer that points to a structure of the same type is called a *self-referential structure.*

To use a linked list, you normally perform the following steps:

1. Define a structure to contain the desired values and pointer to the next node in the list.
2. Create the first node in the list by allocating memory through the use of calloc or malloc (Turbo C run-time library routines). Assign the memory to the first pointer in the list (called start or head).
3. Create nodes for the remaining entries, again by using calloc or malloc. Assign the pointer of each node to point to the next node in the list. The last node in the list should point to NULL.
4. Use the list of entries as required.
5. Once the nodes are no longer required, use the routine free (Turbo C run-time library) to release allocated memory.

The following program creates a linked list that contains the uppercase letters of the alphabet:



If you follow the steps previously listed, your processing is as follows:

1. Define a structure to contain the desired values and pointer to the next node in the list.

```
struct list_entry {
  char letter;
  struct list_entry *next;
 } *start, *node, *next;
```

2. Create the first node in the list by allocating memory through the use of calloc or malloc (Turbo C run-time library routines). Assign the memory to the first pointer in the list (called start or head).

```
if ((start = (struct list_entry *)
        calloc(1, sizeof(struct list_entry))) == NULL)
  {
    printf ("Unable to allocate memory for the list\n");
    exit (1);
  }
```

Note the type coercion of the type returned by the call to calloc.

```
if ((start = (struct list_entry *)
        calloc(1, sizeof(struct list_entry))) == NULL)
```

By default, calloc returns a pointer to the type void. Since the structure is not type void, we will use the cast. Also note that if calloc cannot allocate the desired memory, it returns the value NULL.

3. Create nodes for the remaining entries again by using calloc or malloc. Assign the pointer of each node to point to the next node in the list. The last node in the list should point to NULL.

```
for (letter = 'A'; letter <= 'Z'; letter++)
  {
   if ((node->next = (struct list_entry *)
            calloc(1, sizeof(struct list_entry))) == NULL)
     {
       printf ("Unable to allocate memory for the list\n");
       exit (1);
     }
   node = node->next;
   node->letter = letter;
   node->next = NULL;
  }
```

Once you have created the list, you want to print out the values that it contains. This becomes step 4.

4. Use the list of entries as required.

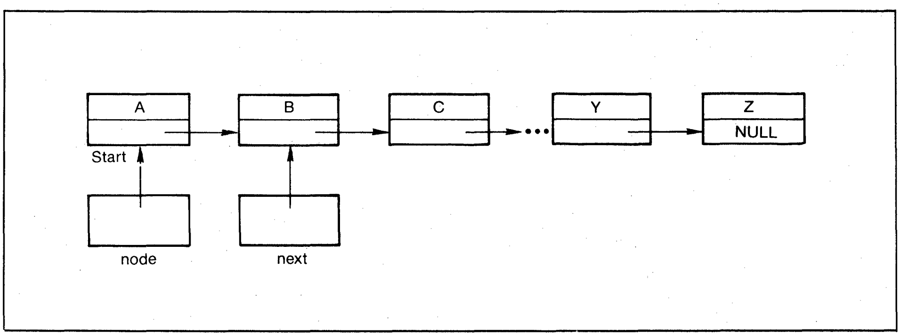
```
for (node = start->next; node != NULL; node = node->next)
   printf ("%c\n", node->letter);
```

This loop begins by examining the first element in the list (pointed to by start). If that element exists, its value is displayed (the letter "A"). Next, the loop assigns the current node to point to the next element in the list (the letter "B"). This process continues until you reach the letter "Z". Once "Z" is printed, the current node is assigned the value NULL, which is the ending condition.

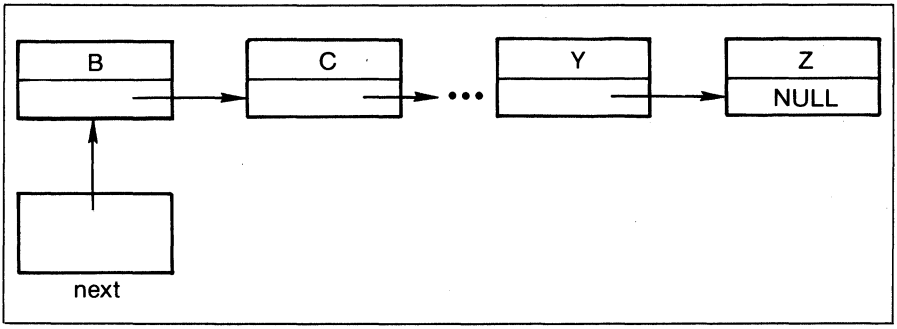
5. Once the nodes are no longer required, use the routine free (Turbo C run-time library) to release allocated memory.

```
for (node = start; node != NULL; node = next)
  {
   next = node->next;
   free (node);
  }
```

To remove nodes from the list, you use the third pointer (next). Begin by assigning node to point to start, and next to point to node—>next.



Once this processing is finished, you can free the memory pointed to by node.



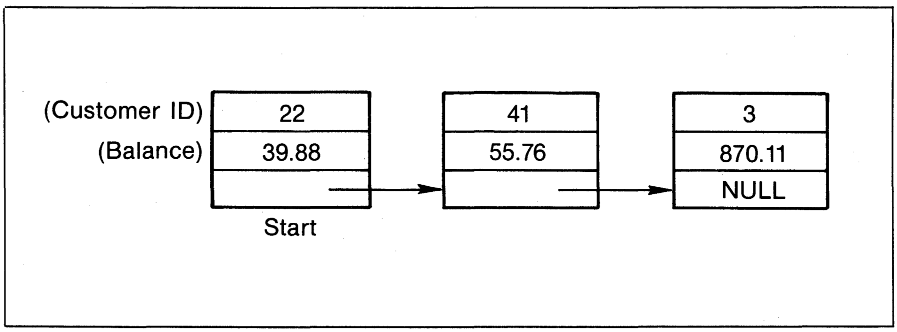
The pointer node is now assigned the value contained in next, and next is assigned node—>next. This process repeats until all of the memory has been released.

Putting all of the pieces together, the complete program is as follows:

```
#include <stdio.h>

main ()
 {
  void *calloc (unsigned, unsigned);

  struct list_entry {
    char letter;
    struct list_entry *next;
   } *start, *node, *next;

  char letter;

  if ((start = (struct list_entry *)
          calloc(1, sizeof(struct list_entry))) == NULL)
    {
      printf ("Unable to allocate memory for the list\n");
      exit (1);
    }

  node = start;

  for (letter = 'A'; letter <= 'Z'; letter++)
    {
     if ((node->next = (struct list_entry *)
              calloc(1, sizeof(struct list_entry))) == NULL)
      {
        printf ("Unable to allocate memory for the list\n");
        exit (1);
```

```
      }
     node = node->next;
     node->letter = letter;
     node->next = NULL;
    }

   for (node = start->next; node != NULL; node = node->next)
      printf ("%c\n", node->letter);

   for (node = start; node != NULL; node = next)
     {
      next = node->next;
      free (node);
     }
}
```

In this example, the nodes contained simple data. However, you could have been creating a list of customer information, as shown here:



Although the structure is different, the steps required to create the list are the same. The structure in this case could be as follows:

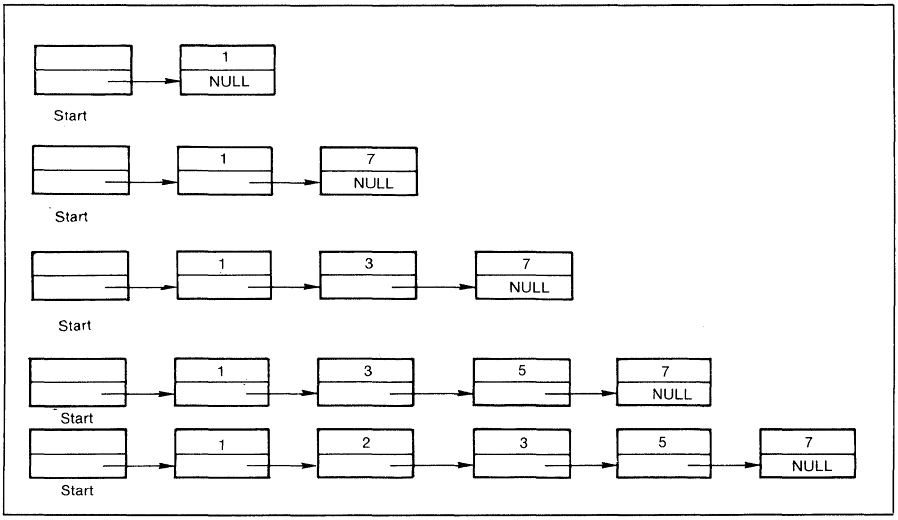
```
struct customers {
  char name[30];
  char address[30];
  char city[15];
  char state[3];
  char zip[11];
  char phone[11];
  float balance;
  struct customers *next;
 } ;
```

Maintaining a Sorted List

If you place items in your list in a specified manner, you can often reduce the amount of sorting you may later need to perform. For example, if you want to create a list of numbers entered by the user, and the user types these numbers,

1 7 3 5 2

you can build the list in sorted order, as shown here:



In this case, use a routine called list_insert, which places an element into the list based upon the value in the member value:

```
/*
 * list_insert (value, start)
 *
 * Place a value into a linked list in sorted order.
 *
 * value (in): Value to add to the list.
```

```
 * start (in): First node in the list.
 *
 * status = list_insert (5, start);
 *
 * If list_insert cannot allocate the memory required, it returns
 * the value -1.  If successful, this routine returns the value 0.
 *
 */

int list_insert (int value, struct list_entry *start)
 {
  struct list_entry *new_node, *node, *previous;

  void *calloc(unsigned, unsigned);

  node=start->next;
  previous = start;

  while ((node != NULL) && (node->value < value))
    {
     previous = node;
     node = node->next;
    }

  if ((new_node = (struct list_entry *)
             calloc(1, sizeof(struct list_entry))) == NULL)
       return (-1);

  new_node->value = value;
  new_node->next = node;
  previous->next = new_node;

  return (0);
 }
```

Modifying the previous program slightly to use this routine, the code now becomes

```
#include <stdio.h>

struct list_entry {
  char value;
  struct list_entry *next;
 } ;

main ()
 {
  void *calloc (unsigned, unsigned);

  struct list_entry *start, *node, *next;

  char letter;

  if ((start = (struct list_entry *)
          calloc(1, sizeof(struct list_entry))) == NULL)
    {
      printf ("Unable to allocate memory for the list\n");
      exit (1);
    }
```

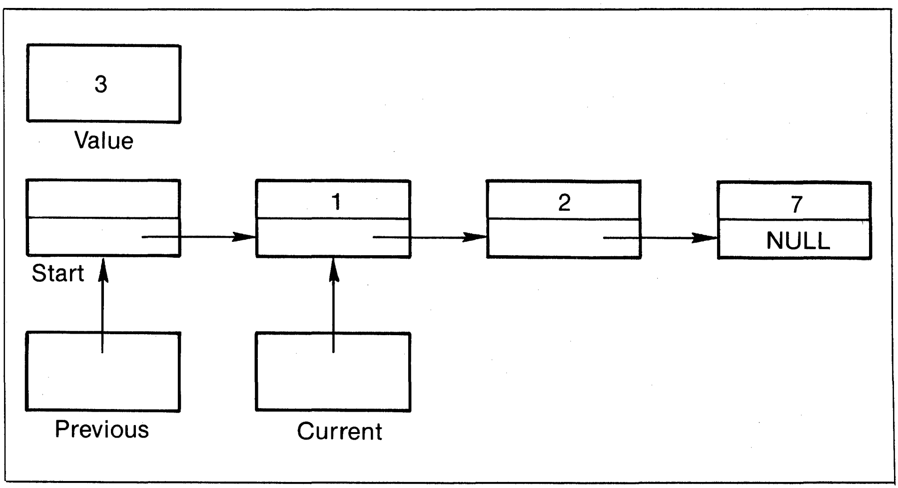
```
  node = start;
  node->next = NULL;

  for (letter = 'A'; letter <= 'Z'; letter++)
    if (list_insert (letter, start) == -1)
      {
       printf ("Unable to allocate memory for the list\n");
       exit (1);
      }

   for (node = start->next; node != NULL; node = node->next)
      printf ("%c\n", node->value);

   for (node = start; node != NULL; node = next)
     {
      next = node->next;
      free (node);
     }

 }
```

By using this routine, you can place ten integer values into the list, as shown here:

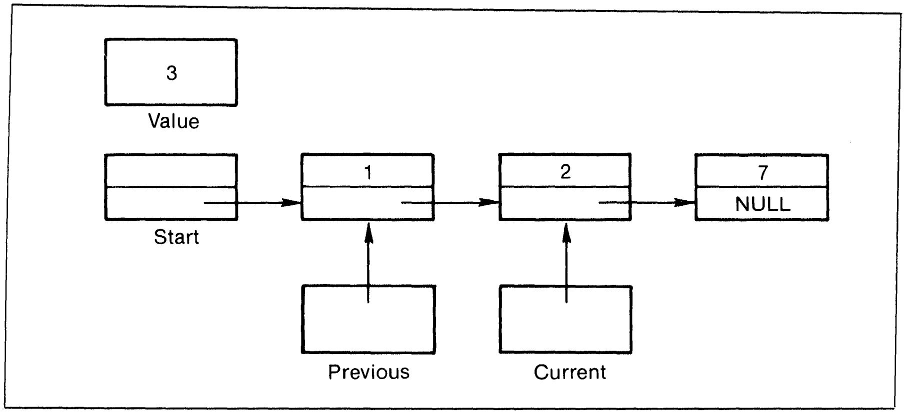
```
#include <stdio.h>

struct list_entry {
  int value;
  struct list_entry *next;
 } ;

main ()
 {
  void *calloc (unsigned, unsigned);

  struct list_entry *start, *node, *next;

  int i, value;

  int get_prompted_int (int, int, int, int, int, int, int,
                        char *, int, int, int, char *, int,
                        int, int, int);

  if ((start = (struct list_entry *)
          calloc(1, sizeof(struct list_entry))) == NULL)
    {
      printf ("Unable to allocate memory for the list\n");
      exit (1);
    }

  node = start;
  node->next = NULL;

  for (i = 0; i < 10; i++)
   {
    value = get_prompted_int (0, 0, 10, 20, 30, 7, 64, "Value:",
       10, 7, 13,"Enter an integer then press Enter.", 23, 7, 40, 7);

    if (list_insert (value, start) == -1)
```

```
         {
          printf ("Unable to allocate memory for the list\n");
          exit (1);
         }
     }

     for (node = start->next; node != NULL; node = node->next)
        printf ("%d\n", node->value);

     for (node = start; node != NULL; node = next)
       {
        next = node->next;
        free (node);
       }
   }
```

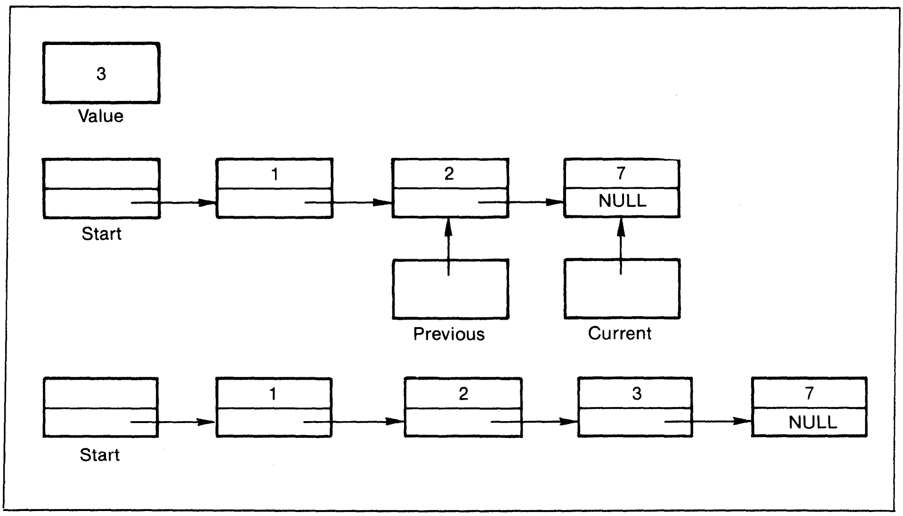
The routine works by tracking two nodes (the previous and current), as shown here:



As you traverse the list in search of the correct location at which to insert the value, you must update both pointers, as shown next:



Once you find the correct location, insert the value by using the two nodes, as illustrated here:



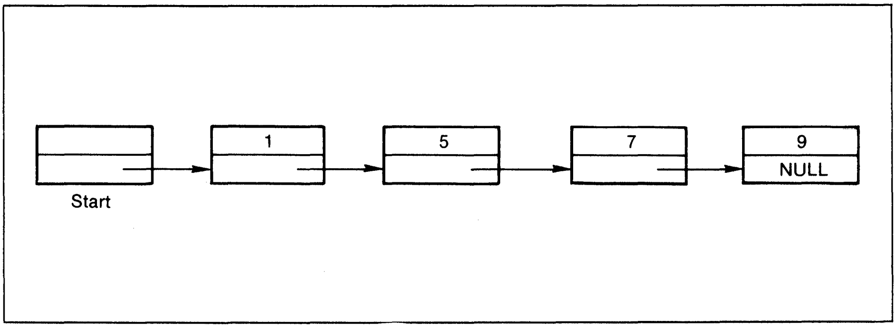
Note the processing that is required to free the memory allocated by list.

```
for (node = start; node != NULL; node = next)
  {
   next = node->next;
   free (node);
  }
```

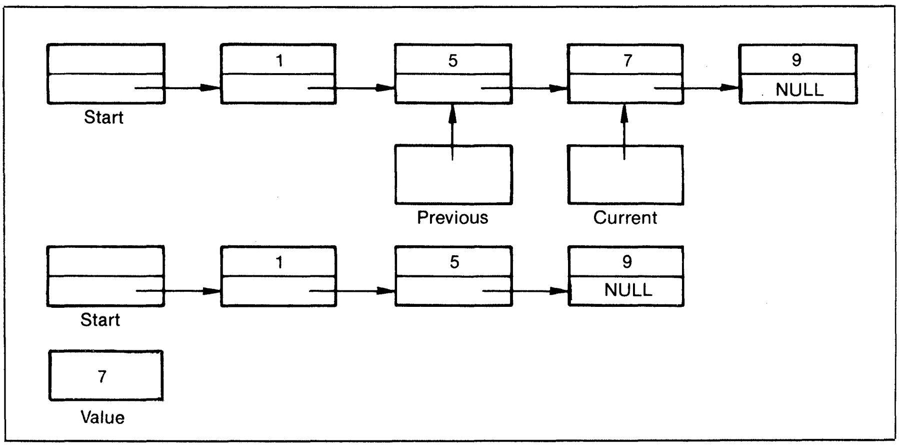
You can instead use the routine free_list, as illustrated here:

```
/*
 * void free_list (start)
 *
 * Release memory previous allocated for a dynamic structure.
 *
 * start (in): First node in the list.
 *
 * free_list (start);
 *
 */

void free_list (struct list_entry *start)
 {
  struct list_entry *node, *next;

  for (node = start; node != NULL; node = next)
   {
     next = node->next;
     free (node);
   }
 }
```

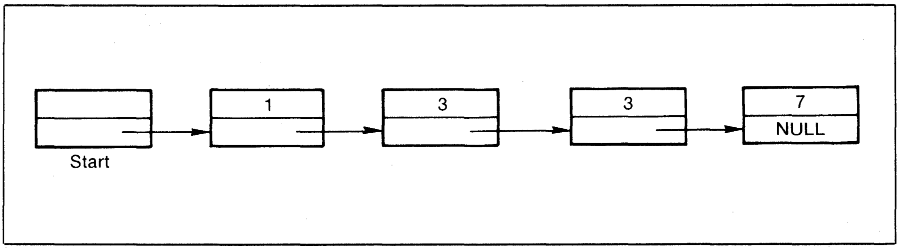
Deleting an element from a linked list is similar to inserting a node into the list. You again use two nodes to track the previous and current nodes. Given the list,



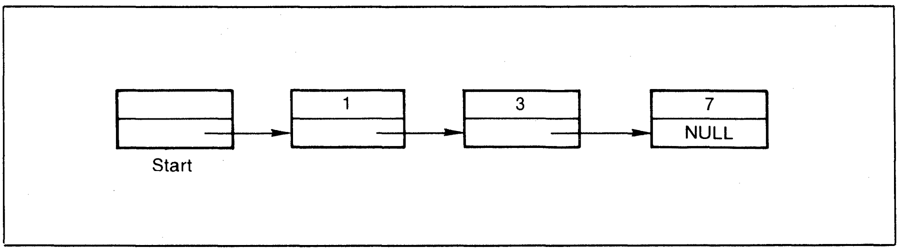
the processing to delete the node containing the value 7 becomes as follows:



The routine list_delete deletes a node from the linked list. Again, it is based upon the contents of the member value. If multiple nodes contain the same value,



only the first node is deleted.

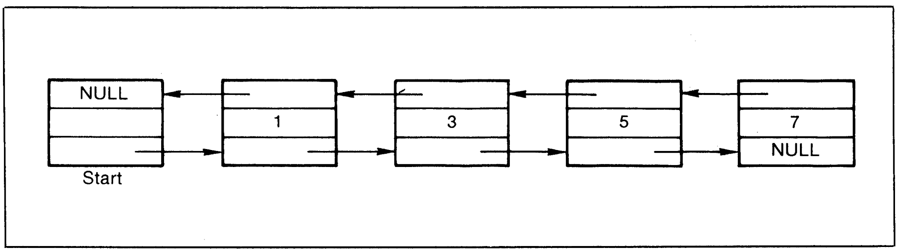



```
/*
 * list_delete (value, start)
 *
 * Remove a node containing the value specified from a linked list.
 *
 * value (in): Value to remove from the list.
 * start (in): First node in the list.
 *
 * status = list_delete (5, start);
 *
 * If list_delete is successful, it returns the value 0, otherwise
 * it returns the value -1.
 *
 */

int list_delete (int value, struct list_entry *start)
 {
  struct list_entry *node, *previous;

  node=start->next;
  previous = start;
```

```
  while ((node != NULL) && (node->value != value))
    {
     previous = node;
     node = node->next;
    }

  if (node)
   {
     previous->next = node->next;
     free (node);
     return (0);
   }
  else
    return (-1);
 }
```

Doubly Linked Lists

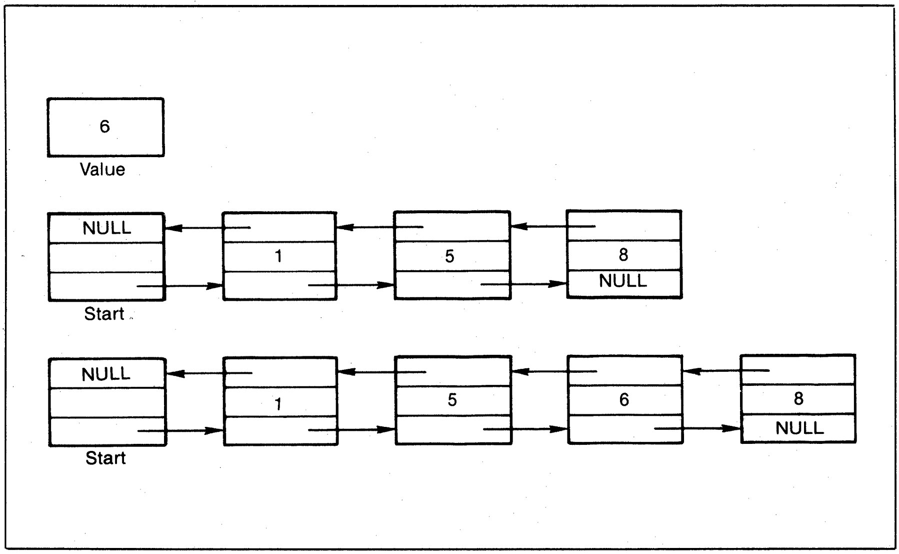
By now, you should understand that a linked list can provide considerable flexibility. The difficulty with linked lists at this point may be the processing required to insert and delete a node. You can create an even more flexible dynamic structure called a *doubly linked list.* Unlike a *singly linked list* (which only contains a pointer to the next element), a doubly linked list contains a pointer to the next and to the previous elements, as shown here:



The structure now contains two pointers, as follows:

```
struct double_link {
  int value;
  struct double_link *previous;
  struct double_link *next;
 } ;
```

Since you now have a pointer to the previous and next elements in the list, you can eliminate the need for tracking two pointers during insert and delete operations.




```
/*
 * doubly_list_insert (value, start)
 *
 * Insert a value in a doubly linked list in sorted order.
 *
 * value (in): Value to placed into the list.
 * start (in): First node in the list.
 *
 * status = doubly_list_insert (5, start);
 *
 * If doubly_list_insert is successful, it returns the value 0,
 * otherwise, this routine returns the value -1.
 *
 */

int doubly_list_insert (int value, struct list_entry *start)
 {
  struct list_entry *new_node, *node;

  void *calloc(unsigned, unsigned);

  node=start->next;
```

```
  /* locate insertion point */
  while ((node != NULL) && (node->value < value))
   if (node->next != NULL)
     node = node->next;
   else
     break;

  if ((new_node = (struct list_entry *)
              calloc(1, sizeof(struct list_entry))) == NULL)
        return (-1);

  new_node->value = value;

  if ((node != NULL) && (node->next != NULL)) /* not last entry */
   {
     new_node->next = node;
     new_node->previous = node->previous;
     (node->previous)->next = new_node;
     node->previous = new_node;
   }
  else          /* first, last or next to last entry */
    {
      if ((start->next == node) && (node == NULL))
       {
        new_node->previous = start;   /* first list entry */
        start->next = new_node;
        new_node->next = NULL;
       }
      else if (value < node->value)   /* next to last */
       {
        new_node->next = node;
        new_node->previous = node->previous;
        (node->previous)->next = new_node;
       }
      else
       {                              /* last entry */
        new_node->next = NULL;
        new_node->previous = node;
        node->next = new_node;
       }
    }

  return (0);                     /* successful insertion */
}
```

In a similar manner, this routine deletes an element from a doubly linked list. Once again, if two elements have the same value, only the first is deleted.

```
/*
 * doubly_list_delete (value, start)
 *
 * Remove a node containing a specified value in a doubly linked
 * list.
 *
```

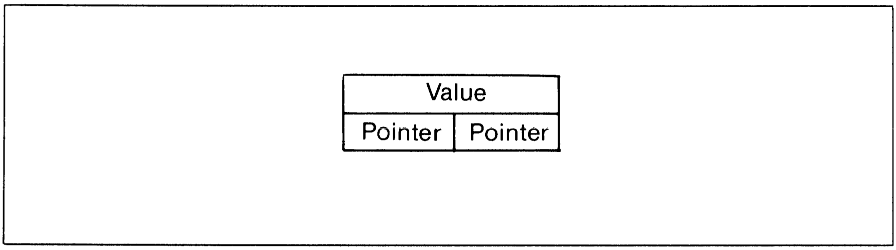
```
 * value (in): Value to be removed.
 * start (in): First node in the list.
 *
 * status = doubly_list_delete (5, start);
 *
 * If successful, this routine returns the value 0, otherwise it
 * returns the value -1.  If multiple occurrences of the value
 * exist in the list, only the first is deleted.
 *
 */

int doubly_list_delete (int value, struct list_entry *start)
 {
  struct list_entry *node;

  node=start->next;

  /* find the value */
  while ((node != NULL) && (node->value != value))
     node = node->next;

  if (node)
   {
     (node->previous)->next = node->next;
     if (node->next != NULL)
       (node->next)->previous = node->previous;
     free (node);
     return (0);
   }
  else
    return (-1);
 }
```

Binary Trees

Linked lists add considerable flexibility to your programs. By using a structure similar to that of a doubly linked list,

```
struct binary_tree {
  int value;
  struct binary_tree *right;
  struct binary_tree *left;
 } ;
```

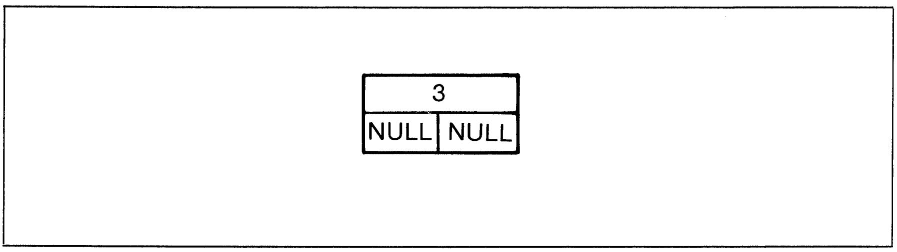
you can create a data structure in which all of the elements are automatically placed into a presorted order. When you later need to locate a value, you can do so with the same performance as that associated with a binary search. The new structure is called a *binary tree.* The structure appears as follows:



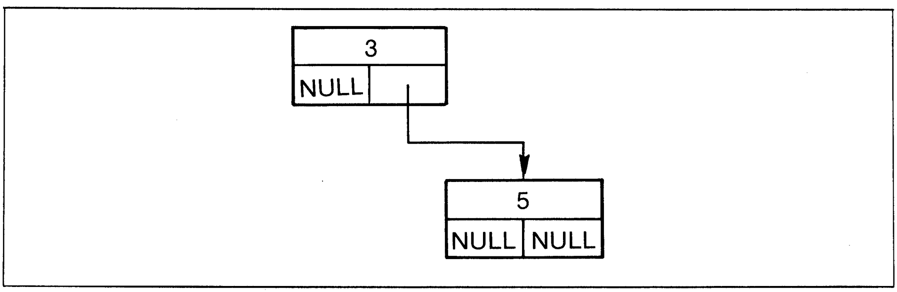
Each time you add a value to the binary tree, you begin by examining the first node (or root) of the tree. If the value is less than that of the root, you traverse the left side of the tree. If the value is greater than or equal to that of the root, you traverse the right side of the tree. Given the following numbers,

3 5 1 7 2 9 8

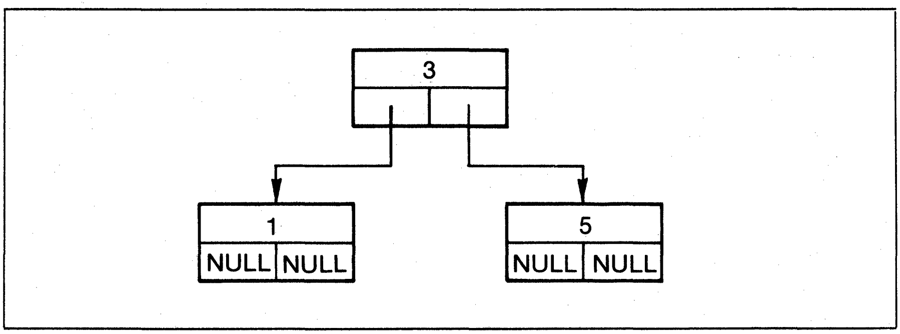
you would construct the tree as follows. The value 3 will be placed in the root node of the tree.



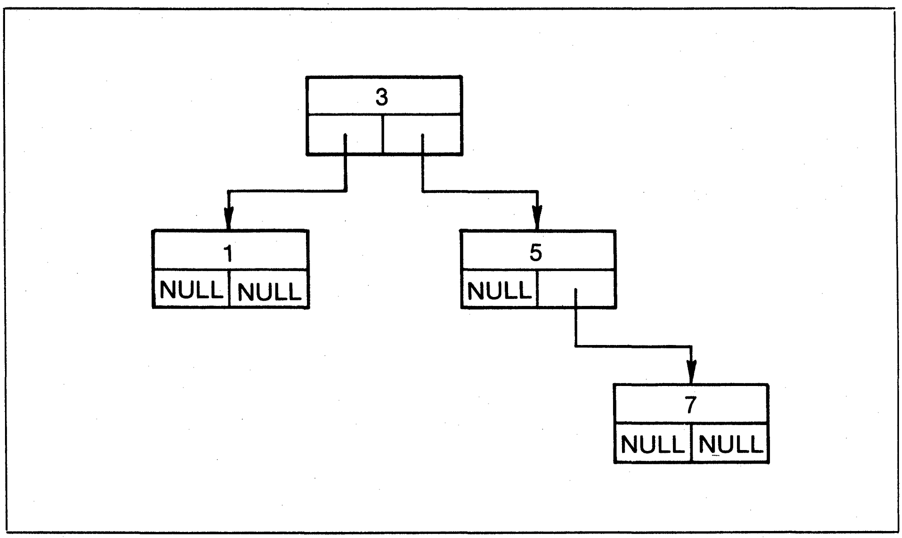
Since the value 5 is larger than 3, it becomes a right node.



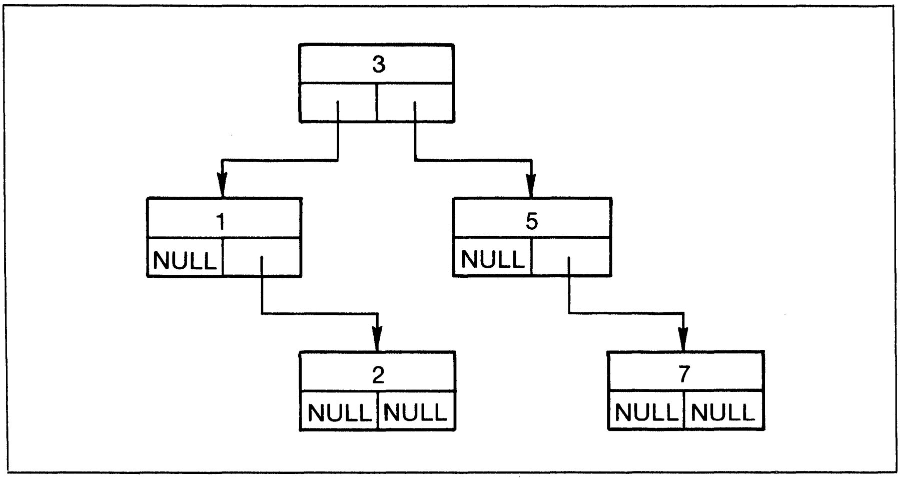
Likewise, the value 1 is less than 3, so it becomes a left node.



Since 7 is greater than 3, you traverse the right side of the tree. Since it is also greater than 5, it becomes a right node.



Since 2 is less than 3 but greater than 1, it is inserted in the tree, as shown next:



Finally, the values 9 and 8 are added as shown in Figure 13-1.

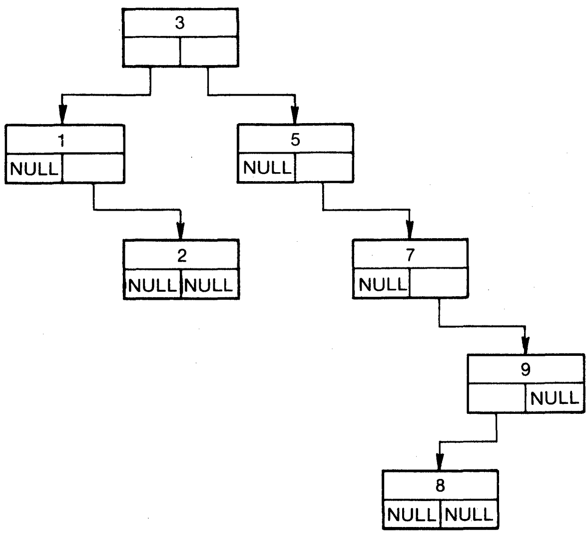
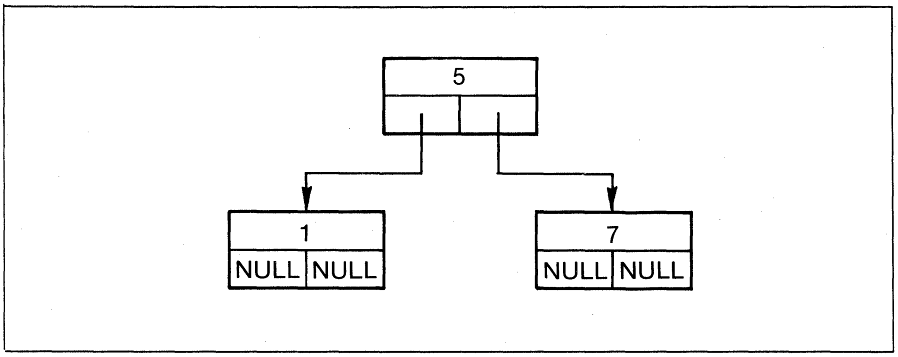


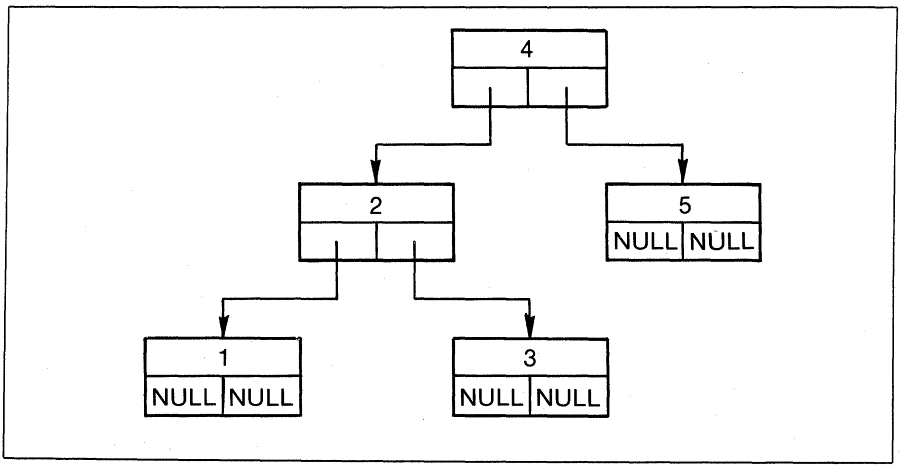
Figure 13-1. *Binary tree after adding values 8 and 9*

To list the elements in the tree, follow similar steps. First, you begin at the root and traverse the left side of the tree. You traverse nodes to the left until no nodes remain. At that point you print the value of the current node, move up one node and print its value, and then begin traversing the right nodes, if they exist. Given the tree,



you would display the values as shown in Figure 13-2.

Likewise, given the tree,



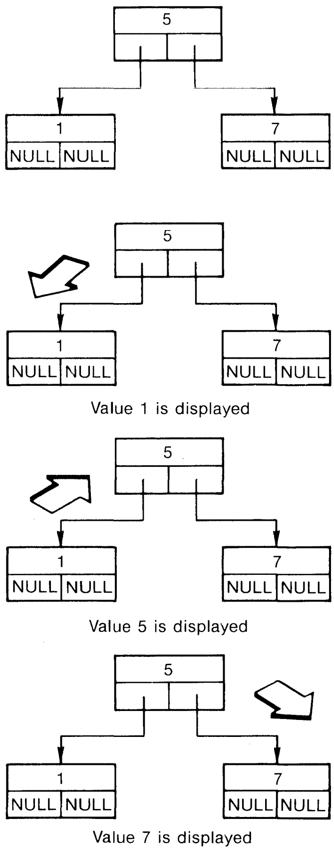


Figure 13-2. *Values displayed while traversing nodes*

the values would be printed as shown in Figure 13-3.

The following routine places a node into a binary tree:

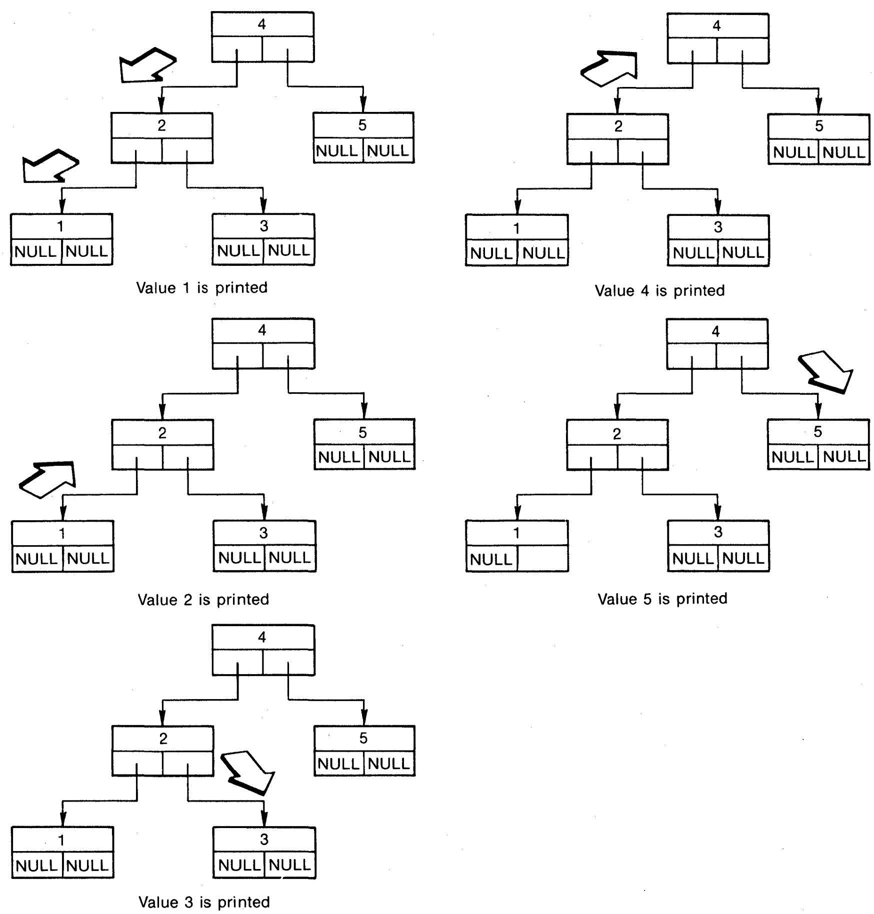


Figure 13-3. *Values printed from binary tree*

```
/*
 * tree_insert (value, node)
 *
 * Place a value into a binary tree.
 *
 * value (in): Value to placed into the tree.
 * node (in): Starting node in the "current" binary tree.
 *
 * status = tree_insert (5, node);
 *
 * If successful, tree_insert returns the value 0, otherwise
 * it returns the value -1.
 *
 */

int tree_insert (int value, struct list_entry *node)
 {
  void *calloc(unsigned, unsigned);

  if (value < node->value)
    if (node->left != NULL)
      tree_insert (value, node->left);
    else
     {
      if ((node->left = (struct list_entry *)
              calloc(1, sizeof(struct list_entry))) == NULL)
         return (-1);

      (node->left)->right = NULL;
      (node->left)->left = NULL;
      (node->left)->value = value;
     }
  else
    if (node->right != NULL)
      tree_insert (value, node->right);
    else
     {
      if ((node->right = (struct list_entry *)
              calloc(1, sizeof(struct list_entry))) == NULL)
         return (-1);

      (node->right)->right = NULL;
      (node->right)->left = NULL;
      (node->right)->value = value;
     }

  return (0);
 }
```

Note that the routine is recursive. This is because of the recursive definition of a binary tree. If you consider each subtree as its own binary tree, your processing is identical at each node.

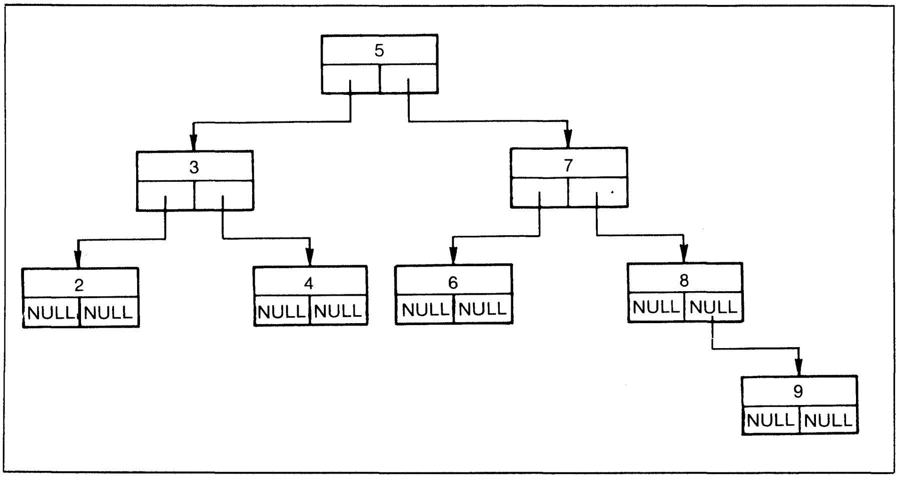
In a similar manner, the following routine displays the contents of a binary tree by using the algorithm previously discussed:

```
/*
 * void show_tree (node)
 *
 * node (in): Start of the "current" binary tree.
 *
 * show_tree (root);
 *
 * You must change the printf control sequence based upon
 * the type of value to display.
 *
 */

void show_tree (struct list_entry *node)
 {
  if (node->left)
    show_tree (node->left);

  printf ("%d\n", node->value);

  if (node->right)
    show_tree (node->right);
 }
```

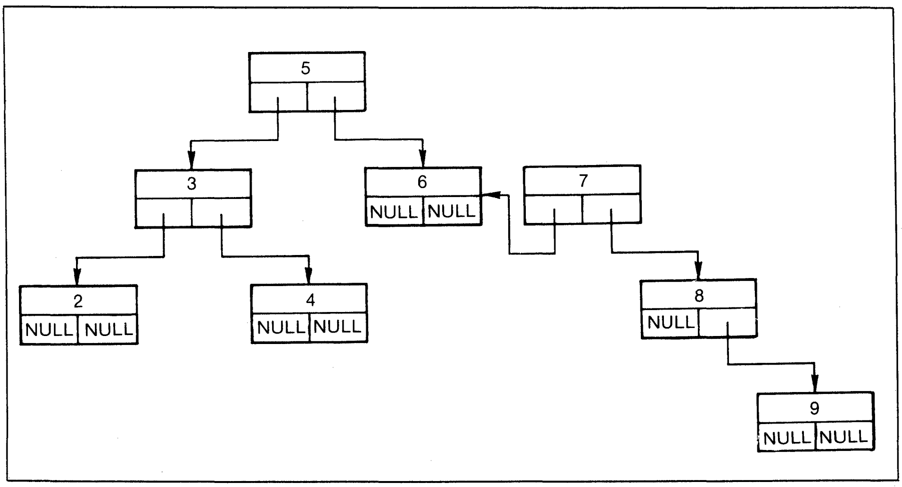
In a similar manner, the following routine releases memory prevously allocated for a binary tree:

```
/*
 * void free_tree (node)
 *
 * Release the memory previously allocated for a binary tree.
 *
 * node (in): Starting node in the "current" node.
 *
 * free_tree (root);
 *
 */

void free_tree (struct list_entry *node)
 {
  if (node->left)
    free_tree (node->left);

  if (node->right)
    free_tree (node->right);

  free (node);
 }
```

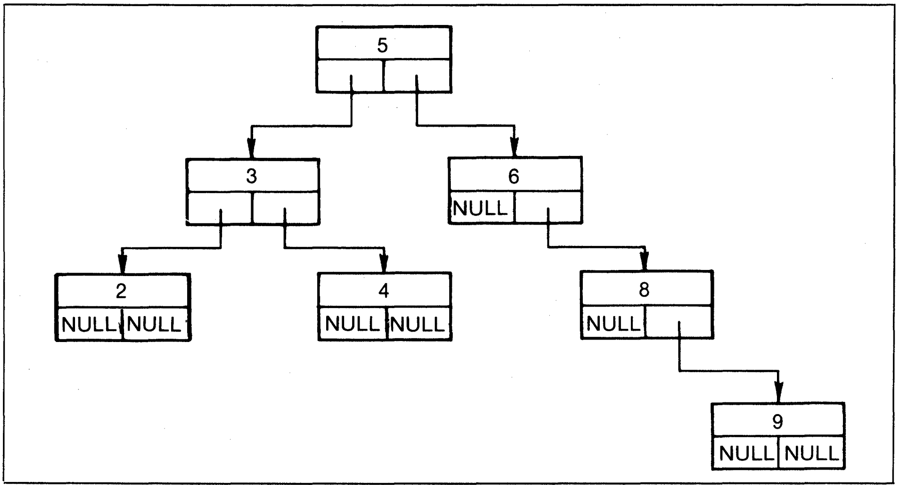
Just as you must be able to insert items into the binary tree, you must also be able to delete them. Given the following tree,



the processing to delete the node containing the value 7 requires two steps. First, you must assign the right pointer of the previous node to point to the left node of the node to delete (assuming that it exists).



Second, you must assign the right node of the node to delete to the first node in the new chain that points to NULL.



The following routine performs a binary tree deletion:

```
/*
 * delete_tree (value, start)
 *
 * Remove a node containing the value specified from a binary tree.
 *
 * value (in): Value to delete.
 * start (in): Pointer to the first node in the list.
 *
 * status = delete (5, &root);
 *
 * If multiple nodes contain the value specified, only the first
 * node found is deleted.  This routine uses the routine
 * delete_tree_entry if the the root does not point to the desired
 * value.
 *
 */

delete_tree (int value, struct list_entry **start)
 {
  struct list_entry *current_node;

  if (value == (*start)->value)
   {
```

```
      if ((*start)->left)
        current_node = (*start)->left;
      else if ((*start)->right)
        current_node = (*start)->right;
      else
        return (1);  /* only one node in the tree */

      while (current_node->right)
        current_node = current_node->right;

      if (current_node != (*start)->right)
        current_node->right = (*start)->right;

      current_node = *start;

      if ((*start)->left)
        *start = (*start)->left;
      else if ((*start)->right)
        *start = (*start)->right;

      free (current_node);

      return (0);
     }
    else if (value < (*start)->value)
      return (delete_tree_entry (value, *start, (*start)->left));
    else
      return (delete_tree_entry (value, *start, (*start)->right));
   }

/*
 * delete_tree_entry (value, previous, node)
 *
 * Remove a node containing the value specified from a binary tree.
 *
 * value (in): Value to delete.
 * previous (in): Pointer to the node preceeding the current node.
 * node (in): Current node in the binary tree.
 *
 * status = delete_tree_entry (5, root, root->next);
 *
 * This routine is called by delete_tree when the root does not
 * contain the value desired.
 *
 */

delete_tree_entry (int value, struct list_entry *previous,
                   struct list_entry *node)
 {
  struct list_entry *current_node;

  int result = -1;

  if (node == NULL)
    return (-1);

  if (node->value == value)
   {
    if (previous->left == node)
      previous->left = NULL;
```

```
    if (previous->value < value)
     {
       previous->right = node->left;
       current_node = previous;

       while (current_node->right)
         current_node = current_node->right;

       current_node->right = node->right;
     }
    else
     {
      previous->left = node->left;
      current_node = previous;

      if (current_node->left)
       {
        current_node = current_node->left;

        while (current_node->right)
          current_node = current_node->right;

        current_node->right = node->right;
       }
      else
        current_node->left = node->right;
     }

    free (node);
    result = 0;
   }
  else if (node->value > value)
   {
    if (node->right)
     result = delete_tree_entry (value, node, node->right);

    if ((node->left) && (result == -1))
      result = delete_tree_entry (value, node, node->left);
    else
      return (-1);
   }

  else
   {
    if (node->left)
      result = delete_tree_entry (value, node, node->left);

    if ((node->right) && (result == -1))
      result = delete_tree_entry (value, node, node->right);

    else
      return (-1);
   }
  return (result);
}
```

The following program uses a binary tree to display the contents of a small file in sorted order:

```
#include <stdio.h>

struct list_entry {
  char value[132];
  struct list_entry *right;
  struct list_entry *left;
 } ;

main (int argc, char *argv[])
 {
  void *calloc (unsigned, unsigned), free_tree ();

  struct list_entry *start, *node, *new_node;

  FILE *fopen (), *fp;

  char line[132];

  if (argc < 2)
    {
      printf ("SORT: Invalid usage: SORT FILENAME.EXT\n");
      exit (1);
    }
  else if (! (fp = fopen (argv[1], "r")))
    {
      printf ("SORT: Unable to open the file %s\n", argv[1]);
      exit (1);
    }

  else if ((start = (struct list_entry *)
          calloc(1, sizeof(struct list_entry))) == NULL)
    {
      printf ("Unable to allocate memory for the list\n");
      exit (1);
    }
  else
   {
     node = start;
     node->right = NULL;
     node->left = NULL;
     fgets (start->value, sizeof(line), fp);
   }

  while (fgets (line, sizeof(line), fp))
   {
    if (tree_insert (line, start) == -1)
      {
       printf ("Unable to allocate memory for the list\n");
       exit (1);
      }
   }

  fclose (fp);
  show_tree (start);
  free_tree (start);
 }

int tree_insert (char *value, struct list_entry *node)
 {
  void calloc();

  if (string_comp (value, node->value, 0) == 2)
```

```
    if (node->left != NULL)
      tree_insert (value, node->left);
    else
     {
      if ((node->left = (struct list_entry *)
              calloc(1, sizeof(struct list_entry))) == NULL)
         return (-1);

      (node->left)->right = NULL;
      (node->left)->left = NULL;
      copy_string (value, (node->left)->value, 132);
     }
  else
    if (node->right != NULL)
      tree_insert (value, node->right);
    else
     {
      if ((node->right = (struct list_entry *)
              calloc(1, sizeof(struct list_entry))) == NULL)
         return (-1);

      (node->right)->right = NULL;
      (node->right)->left = NULL;
      copy_string (value, (node->right)->value, 132);
     }

  return (0);
 }

show_tree (struct list_entry *node)
 {
  if (node->left)
    show_tree (node->left);

  printf ("%s\n", node->value);

  if (node->right)
    show_tree (node->right);
 }
```

The code fragments in this chapter are intended to provide foundations on which you can build your programs.

CHAPTER

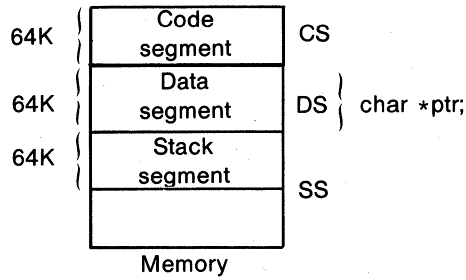
14

Memory Mapping

Chapter 3 examined the use of pointers in Turbo C string manipulation. That discussion showed that a pointer is a value that "points to" (or references) a specific location in memory and that most pointers reference memory locations contained within a 64K data segment.

```
char *ptr;
```

Although the following pointers could easily access all of the memory locations within the example region, the pointers cannot access memory locations beyond the region.



Pointers of this type are often called near pointers because the locations they reference must be within a 64K region. Memory locations within the IBM PC and PC compatibles are addressed by way of a segment and offset combination. The *segment address* defines the location of a specific 64K region. The *offset address* is used to access individual memory locations with the segment, as shown in Figure 14-1.

A program that performs memory mapping uses a segment and offset address to access specific memory locations. The most common use of memory mapping is for input and output.

Chapters 6 and 7 examined several DOS and BIOS services that perform sophisticated I/O operations. However, in some cases these services are simply too slow. As a result, many programmers instead place output characters directly into the video display memory.

The IBM PC and PC compatibles set aside a region of memory called the *video display memory.* Before a letter can appear on your screen, it must reside in the video display memory. Depending on your monitor type, the following memory locations are used:

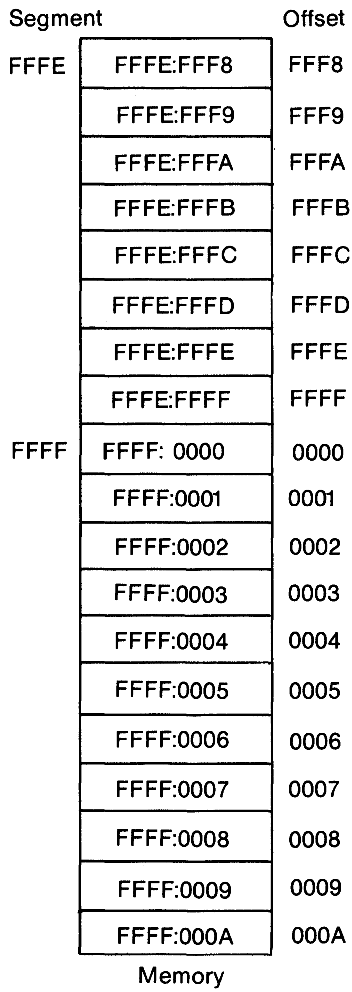
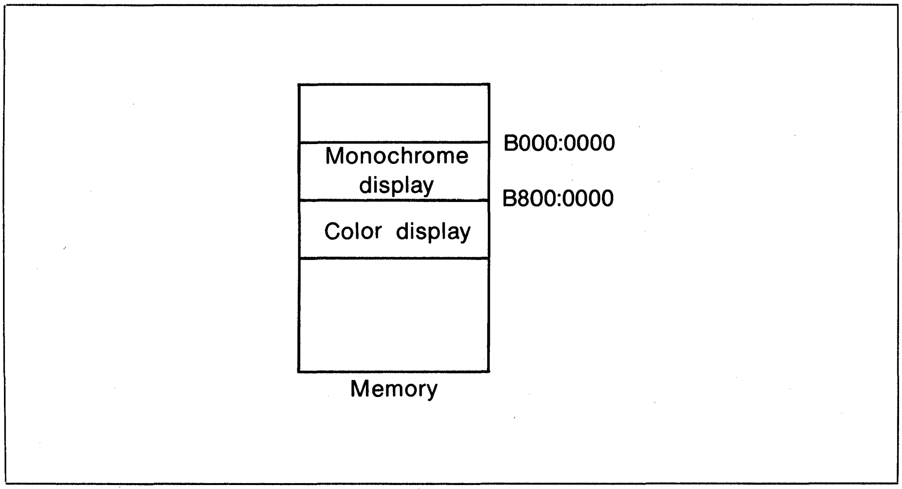


Figure 14-1. *Adding offsets to access memory locations*



Picture the video display memory as a two-dimensional array with 25 rows and 160 columns (see Figure 14-2). The 25 rows correspond to the 25 rows on your screen. Remember, each character displayed on the screen has an attribute byte with which it is associated. Thus, the value 160 is calculated by the following equation:

80 screen columns * 2 bytes (character and attribute)

To place a letter in the upper-left corner of the screen, you would reference the following memory locations:

0xB0000000	(Monochrome display)
0xB8000000	(Color display)

The attribute for the letter must reside in the memory location immediately following the character, as shown here:

0xB0000001	(Monochrome display)
0xB8000001	(Color display)

row 0	row 0, column 0 character	row 0, column 0 attribute	row 0, column 1 character	row 0, column 1 attribute	...	row 0, column 79 character	row 0, column 79 attribute
	row 1, column 0 character	row 1, column 0 attribute	row 1, column 1 character	row 1, column 1 attribute	...	row 1, column 79 character	row 1, column 79 attribute
	. . . .						
	row 24, column 0 character	row 24, column 0 attribute	row 24, column 1 character	row 24, column 1 attribute	...	row 24, column 79 character	row 24, column 79 attribute

Figure 14-2. *Two-dimensional array of video display memory*

The following program displays the letter "A" in the upper-left corner of a color screen display. Unlike previous programs that used a 16-bit (offset) pointer to a location in the data segment, this program uses a far pointer that allows you to define a segment and offset combination.

```
main ()
 {
  char far *ptr = 0xB8000000;

  *ptr = 'A';
 }
```

With the previous addressing scheme in mind, you can determine the memory location at which to display a character, as shown here:

```
char far *ptr;

ptr = (char far *) (0xB8000000 + (row * 160) + (column * 2))
```

The following program displays the letters "A-Z" in the middle of your screen and constantly changes their display attributes:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
main ()
 {
  char far *ptr, letter;

  int i, j, row = 10, column = 25, attr;

  ptr = (char far *) (0xB8000000 + (row * 160) + (column * 2));

  for (i = 0, letter = 'A'; letter <= 'Z'; i += 2, letter++)
     *(ptr + i) = letter;

  for (i = 1; i < 100; i++)
    for (attr = 1; attr <= 255; attr++)
      for (j = 1, letter = 'A'; letter <= 'Z'; j += 2, letter++)
         *(ptr + j) = attr;
 }
```

Depending on your type of monitor, you may begin to experience snow on your screen display as the program executes. This is because of the manner in which the IBM PC updates the display. Every one-eighteenth of a second, the PC performs a horizontal retrace of the contents of the screen to refresh the screen display. If you access the video display memory during this retrace cycle, snow is likely to occur.

To prevent snow from appearing, you must determine when the retrace is in effect and coordinate your video memory accesses with the retrace. The IBM PC and PC compatibles use the first bit in the value contained in port 0x3DA to specify when the retrace is active. Knowing this, you can obtain this value and determine when to perform your I/O operations (see Figure 14-3).

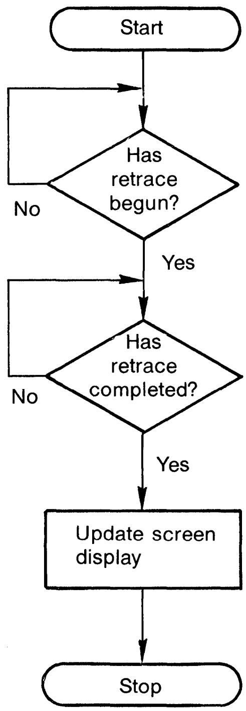


Figure 14-3. *Processing to determine when to perform I/O operations*

Turbo C provides a run-time library routine called inportb that returns the byte value from a port. As such, you would assume that you could use the following code fragment to control your output:

```
while ((inportb(0x3DA) & 1) != 1)
  ;
while ((inportb(0x3DA) & 1) == 1)
  ;
```

Unfortunately, the horizontal retrace occurs so fast that by the time this fragment completes, you do not have time to output the character. You can use the following assembly language routine instead:

```
/*
 * void memory_map_put (segment, offset, value)
 *
 * Map the given value into a memory location within the video
 * display memory.  Insure that the memory reference is in
 * sync with the horizontal retrace.
 *
 * segment (in): Segment address of the video display memory.
 * offset (in): Offset address within the video display memory.
 * value (in): Value to place into the memory.
 *
 * memory_map_put (0xB800, 0, 65);
 *
 */

void memory_map_put (int segment, int offset, int value)
 {
#pragma inline

        asm     push            DX
        asm     push            ES
        asm     push            DI
        asm     push            BX
        asm     push            AX

        asm     MOV             ES, segment
        asm     MOV             DI, offset
        asm     MOV             BX, value

        asm     MOV             DX, 03DAH
A:
        asm     IN              AL, DX
        asm     TEST            AL, 1
        asm     JNZ             A
        asm     CLI
B:
        asm     IN              AL, DX
        asm     TEST            AL, 1
        asm     JZ              B
        asm     MOV             BYTE PTR ES:[DI], BL
        asm     STI
        asm     pop             AX
        asm     pop             BX
        asm     pop             DI
        asm     pop             ES
        asm     pop             DX
 }
```

This routine receives a segment and offset address along with the value to be placed in the memory location. Since the routine does not have the same overhead as the previous Turbo C code fragment, you have time to output the desired value.

To use inline code within Turbo C programs, you must use the Microsoft macro assembler to assemble the code. Invoke the Turbo C compiler and linker from the command line, as shown here:

```
C> TCC FILENAME.C
```

Turbo C will take care of invoking the macro assembler for you.

Turbo C Inline Code
To use inline code from within a Turbo C program, you must have the Microsoft macro assembler. Next, invoke the Turbo C compiler from the command line, as shown here:

C> TCC FILENAME.C

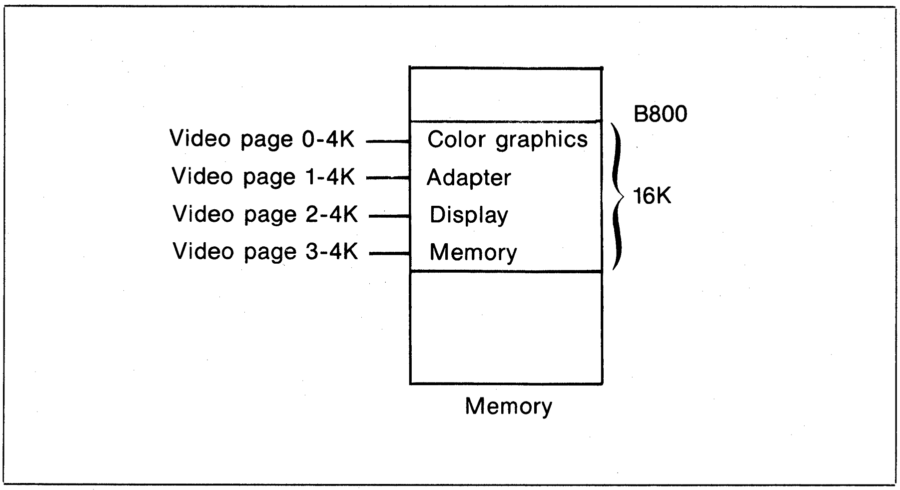
You can now use this code fragment to place the letters "A-Z" on the screen and constantly update the attributes of the letters, as shown here:

```
main ()
 {
  void memory_map_put (int, int, int);

  int letter, i, j, offset, row = 10, column = 25, attr;

  offset = (row * 160) + (column * 2);

  for (i = 0, letter = 'A'; letter <= 'Z'; i += 2, letter++)
     memory_map_put (0xB800, offset + i, letter);

  for (i = 1; i < 100; i++)
    for (attr = 1; attr <= 255; attr++)
      for (j = 1, letter = 'A'; letter <= 'Z'; j += 2, letter++)
        memory_map_put (0xB800, offset + j, attr);
 }
```

No snow will appear this time.

Video Display Pages

Depending on your type of monitor, you may save a considerable amount of memory set aside for video display output. Since a screenful of information requires only 4000 bytes (25 * 160) of storage, the additional memory can be divided up into additional video display buffers called *pages*. For example, in 80 mode the color graphics adapter (CGA) has space for four video display pages, as shown here:



Each video display page is capable of storing a screenful of information. Many of the routines in Chapter 7 allowed you to output to a specific display page. In most cases, you will use display page 0. However, by utilizing video display pages, you can often increase the flair of your applications by making output appear instantaneously. For example, the following program writes a screenful of letters to page 1 and then selects that page as the active display page:

```
#include <stdio.h>

main ()
 {
   int i;

   void set_cursor_positon ();
   void set_active_display_page ();

   for (i = 0; i < 25; i++)
    {
      set_cursor_position (1, i, 0);
      write_char_and_attr (1, 'A', 7, 80);
    }

   set_active_display_page (1);
   getchar();
   set_active_display_page (0);
 }
```

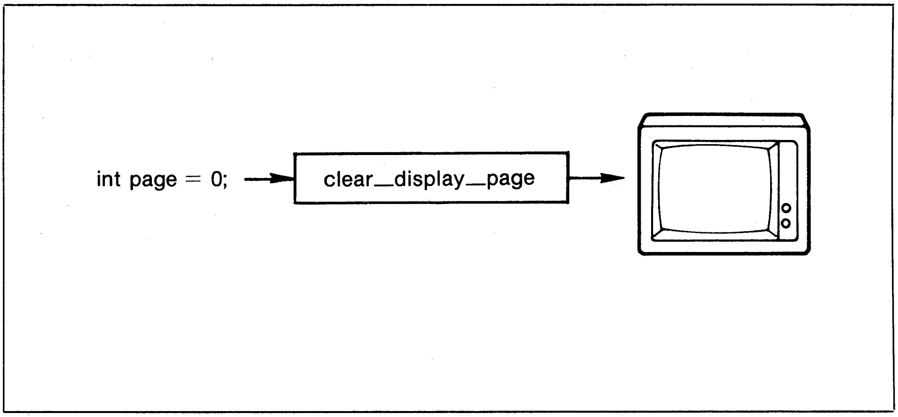
In so doing, the page of letters appears instantaneously.

Note that the previous program resets the video display back to 0 before terminating. Many applications that perform memory-mapped I/O fail to check which display page is active. As such, these applications only display output to page 0. If you do not reset the active display page to 0 before terminating, other applications may not work. Also note that the application simply exits if the display is monochrome. This is because the monochrome system does not support video display pages.

Video display pages can be quite convenient. You can use memory-mapped I/O to increase the speed of applications that use video display pages. The PC sets aside 4096 bytes for each video display page. Knowing that the starting memory location is 0xB8000000, you can compute the location of each video display page as shown here:

page_location = 0xB8000000 + (page * 4096);

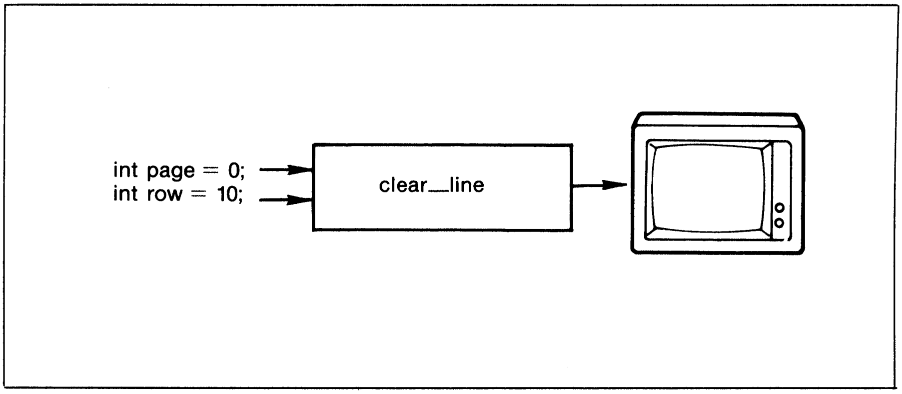
With this in mind, the following routines implement several convenient I/O routines. The first, clear_display_page, clears the contents of the specified video display page.




```
/*
 * void clear_display_page (page)
 *
 * Clear the screen display by setting each character on the
 * screen to an ASCII 32 (space character).
 *
 * page (in): Video display page to clear.
 *
 * clear_display_page (0);
 *
 * This routine only supports 80 column mode.
 *
 */

void clear_display_page (int page)
 {
  int width, mode, current_page, i, j, segment, offset;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);

  get_video_mode (&width, &mode, &current_page);

  /* only support 80 column text mode */
  if ((width != 80) || (page > 3))
    return ;

  if (mode == 7)
   {
    page = 0;
    segment = 0xB000;
   {
  else
    segment = 0xB800;
```

```
  for (i = 0; i <= 25; i++)
   {
    offset = (page * 4096) + (i * 160);

    for (j = 0; j <= 79; j++)
      memory_map_put (segment, offset + (j * 2), 32);
   }
}
```

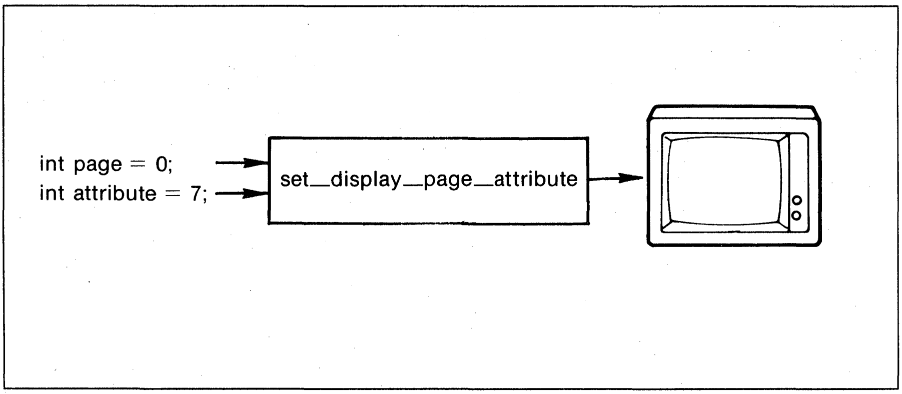
Next, clear_line erases the contents of the specified line from the desired video page.




```
/*
 * void clear_line (page, row)
 *
 * Clear a line on the screen display by setting each character
 * on the line to an ASCII 32 (space character).
 *
 * page (in): Video display page desired.
 * row (in): Display row to clear.
 *
 * clear_line (0, 10);
 *
 * This routine only supports 80 column mode.
 *
 */

void clear_line (int page, int line)
 {
  int width, mode, current_page, i, segment, offset;
```

```
void get_video_mode (int *, int *, int *);
void memory_map_put (int, int, int);

get_video_mode (&width, &mode, &current_page);

if ((width != 80) || (page > 3))
  return ;

if (mode == 7)
 {
  page = 0;
  segment = 0xB000;
 }
else
  segment = 0xB800;

offset = (page * 4096) + (line * 160);

for (i = 0; i <= 79; i++)
  memory_map_put (segment, offset + (i * 2), 32);
}
```

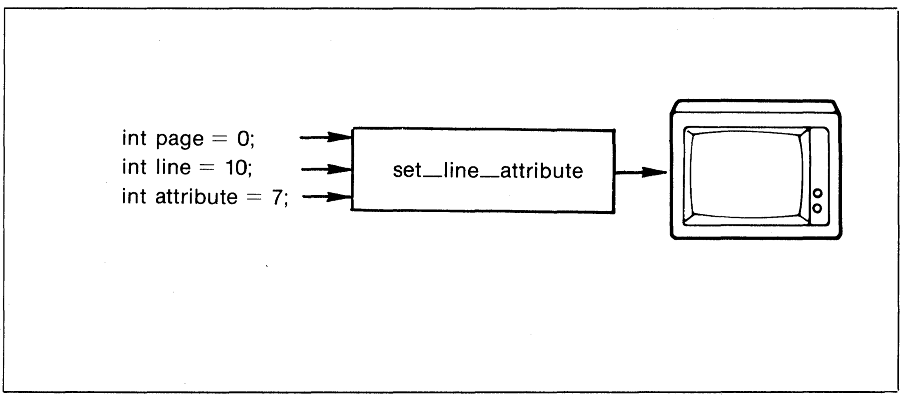
The routine set_display_page_attribute sets the attribute byte for all of the characters on a specific video display page.




```
/*
 * void set_display_page_attribute (page, attribute)
 *
 * Set the character display attribute for the video display
 * page specified.
 *
 * page (in): Video display page desired.
 * attribute (in): Desired video attribute.
 *
```

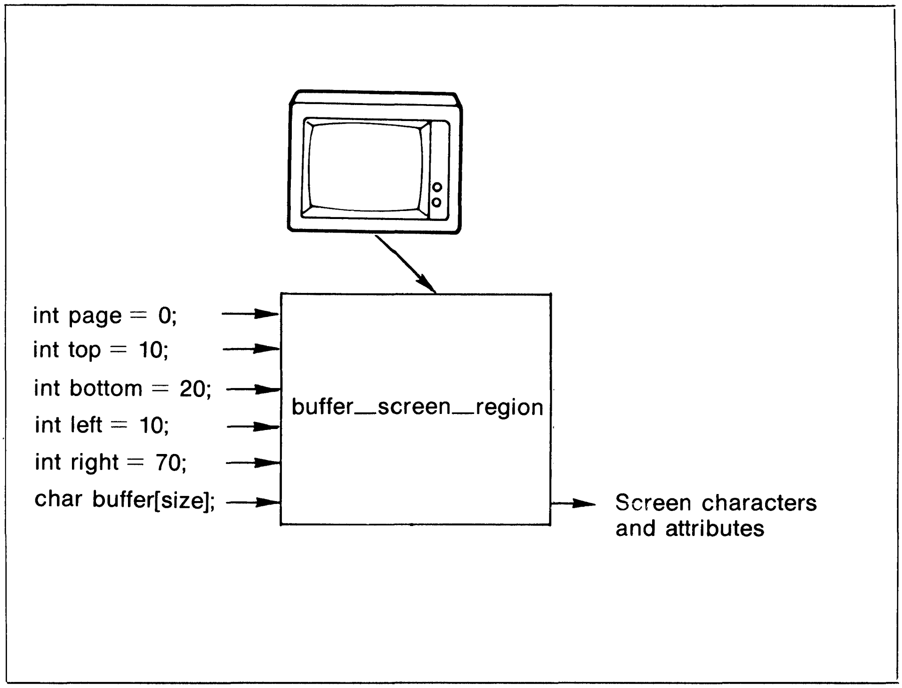
```
 * set_display_page_attribute (0, 7);
 *
 * This routine only supports 80 column mode.
 *
 */

void set_display_page_attribute (int page, int attribute)
 {
  int width, mode, current_page, i, j, segment, offset;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);

  get_video_mode (&width, &mode, &current_page);

  if ((width != 80) || (page > 3))
    return ;

  if (mode == 7)
   {
    segment = 0xB000;
    page = 80;
   }
  else
    segment = 0xB800;

  for (i = 0; i <= 25; i++)
   {
    offset = (page * 4096) + (i * 160);

    for (j = 0; j <= 79; j++)
      memory_map_put (segment, offset + (j * 2) + 1, attribute);
   }
 }
```

In a similar manner, the routine set_line_attribute sets the video display attribute for a specific row on the screen display.



```
/*
 * void set_line_attribute (page, line, attribute)
 *
 * Set the character display attribute for a line on the video
 * display page specified.
 *
 * page (in): Video display page desired.
 * line (in): Row to set the display attribute for.
 * attribute (in): Desired video attribute.
 *
 * set_line_attribute (0, 10, 7);
 *
 * This routine only supports 80 column mode.
 *
 */

void set_line_attribute (int page, int line, int attribute)
 {
  int width, mode, current_page, i, segment, offset;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);

  get_video_mode (&width, &mode, &current_page);

  if ((width != 80) || (page > 3))
    return ;

  if (mode == 7)
   {
    segment = 0xB000;
    page = 0;
   }
  else
    segment = 0xB800;

  offset = (page * 4096) + (line * 160);

  for (i = 0; i <= 79; i++)
    memory_map_put (segment, offset + (i * 2) + 1, attribute);
 }
```

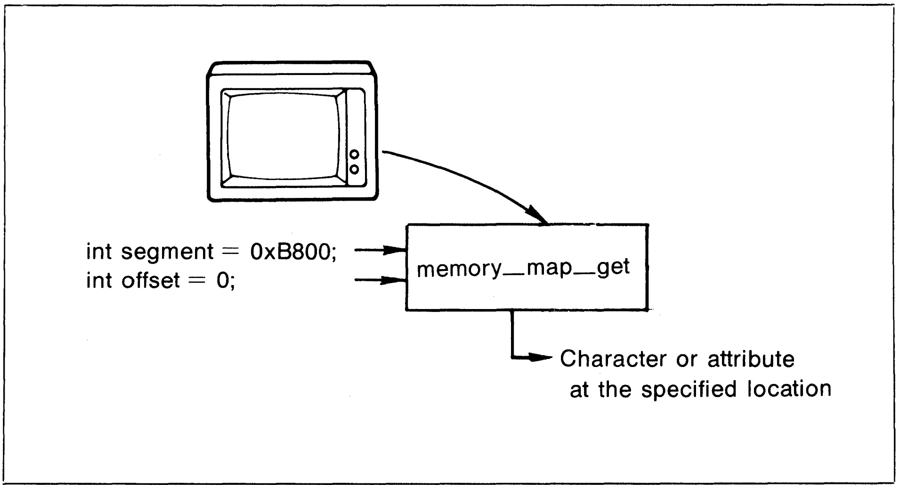
Next, buffer_screen_region saves the contents of a video display page (or region) on the display page.




```
/*
 * void buffer_screen_region (page, top, bottom, left,
 *                            right, buffer)
 *
 * Store a region of the screen display into the buffer provided.
 *
 * page (in): Video display page desired.
 * top (in): Top row of the region to save.
 * bottom (in): Bottom row of the region to save.
 * left (in): Leftmost column of the region to save.
 * right (in): Rightmost column of the region to save.
 * buffer (out): Buffer containing the region to store.
```

```
 *
 * buffer_screen_region (0, 10, 20, 0, 79, buffer);
 *
 * Remember that you are buffering not only the characters, but
 * also the attributes.  As such, your buffer size needs to be:
 *
 *      2 * (bottom - top) * (right - left)
 *
 */

void buffer_screen_region (int page, int top, int bottom,
                           int left, int right, char *buffer)
 {
  int i, j, k = 0;

  int offset, segment, mode, width, current_page;

  void get_video_mode (int *, int *, int *);

  get_video_mode (&width, &mode, &current_page);

  if ((width != 80) || (page > 3))
    return ;

  if (mode == 7)
   {
    page = 0;
    segment = 0xB000;
   }
  else
    segment = 0xB800;

  for (i = top; i <= bottom; i++)
   {
    offset = (page * 4096) + (i * 160);

    for (j = left; j <= right; j++)
     {
       buffer[k++] = memory_map_get (segment, offset + (j * 2));
       buffer[k++] = memory_map_get (segment, offset + (j * 2) + 1)
     }
   }
 }
```

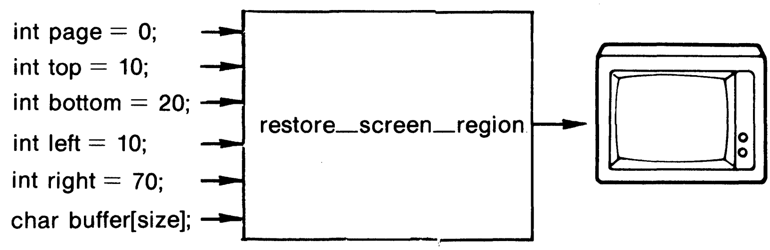
Just as an assembly language routine was required to place values into the video display memory, the following routine reads a byte from the specified segment and offset location.




```
/*
 * memory_map_get (segment, offset)
 *
 * Return the value contained in a memory location within the video
 * display memory.  Insure that the memory reference is in  sync with
 * the horizontal retrace.
 *
 * segment (in): Segment address of the video display memory.
 * offset (in): Offset address within the video display memory.
```

```
 *
 * value = memory_map_get (0xB800, 0);
 *
 */

memory_map_get (int segment, int offset)
 {
   char value;

#pragma inline

        asm     push            DX
        asm     push            ES
        asm     push            DI
        asm     push            AX

        asm     MOV             ES, segment
        asm     MOV             DI, offset

        asm     MOV             DX, 03DAH
A:
        asm     IN              AL, DX
        asm     TEST            AL, 1
        asm     JNZ             A
        asm     CLI
B:
        asm     IN              AL, DX
        asm     TEST            AL, 1
        asm     JZ              B
        asm     MOV             al, BYTE PTR ES:[DI]
        asm     MOV             value, al
        asm     STI
        asm     pop             AX
        asm     pop             DI
        asm     pop             ES
        asm     pop             DX

        return (value);
 }
```

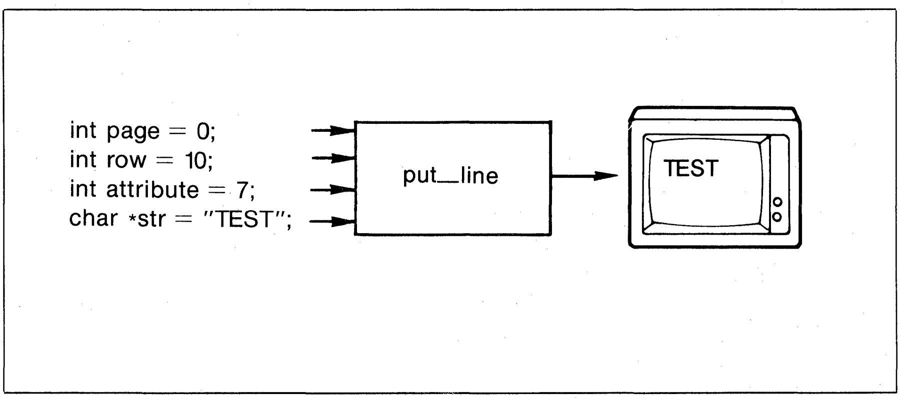
In a similar manner, the routine restore_screen_region restores a previously buffered region of the video display.




```
/*
 * void restore_screen_region (page, top, bottom, left,
 *                             right, buffer)
 *
 * Restore a previously stored region of the screen display
 * from the buffer provided.
 *
 * page (in): Video display page desired.
 * top (in): Top row of the region to restore.
 * bottom (in): Bottom row of the region to restore.
 * left (in): Leftmost column of the region to restore.
 * right (in): Rightmost column of the region to restore.
 * buffer (out): Buffer containing the region to restore.
 *
 * restore_screen_region (0, 10, 20, 0, 79, buffer);
 *
 */

void restore_screen_region (int page, int top, int bottom,
                            int left, int right, char *buffer)
 {
```

```
    int i, j, k = 0;

    int offset, segment, mode, width, current_page;

    void get_video_mode (int *, int *, int *);
    void memory_map_put (int, int, int);

    get_video_mode (&width, &mode, &current_page);

    if ((width != 80) || (page > 3))
      return ;

    if (mode == 7)
     {
      page = 0;
      segment = 0xB000;
     }
    else
      segment = 0xB800;

    for (i = top; i <= bottom; i++)
     {
      offset = (page * 4096) + (i * 160);

      for (j = left; j <= right; j++)
        {
          memory_map_put (segment, offset + (j * 2), buffer[k++]);
          memory_map_put (segment, offset + (j * 2) + 1, buffer[k++]);
        }
     }
  }
```

The routine put_line writes a character string to the video memory by using the specified attribute.



```
/*
 * void put_line (page, row, attribute, line)
 *
 * Place a character string at the row specified using the
 * display attribute given.
 *
 * page (in): Video display page desired.
 * row (in): Display row to place the string at.
 * attribute (in): Desired video display attribute.
 * line (in): Character string to display.
 *
 * put_line (0, 10, 7, "This is a test");
 *
 * This routine only supports 80 column mode.
 *
 */

void put_line (int page, int line, int attribute, char *str)
 {
  int width, mode, current_page, i, segment, offset;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);

  get_video_mode (&width, &mode, &current_page);

  if ((width != 80) || (page > 3))
    return ;

  if (mode == 7)
   {
    segment = 0xB000;
    page = 0;
   }
  else
    segment = 0xB800;

  offset = (page * 4096) + (line * 160);

  for (i = 0; i <= 79; i++)
   {
    if (*str)
     {
      memory_map_put (segment, offset + (i * 2), *str++);
      memory_map_put (segment, offset + (i * 2) + 1, attribute);
     }
    else
      break ;
   }
 }
```

Experiment with the routines presented in this chapter and you should find them to be quite fast. In fact, many of the I/O manipulation routines in this book can be converted to memory-mapped output if your needs require. Many of the routines in Chapter 15 are based on memory-mapped output.

CHAPTER

15

Menus and Special I/O

Chapter 12 examined a series of I/O routines that greatly simplify your programming of the user interface. Those routines allow your programs to prompt for and obtain data from the user in a consistent manner, so users can feel more at ease with the programs they are running. However, anytime a user must interact with a program, the chance of human error increases. Many program developers (and end users) prefer to develop menu-driven systems.

Traditionally, menus have taken the following form:

```
             General Ledger

      Perform Payroll.............1
      Accounts Receivable.........2
      Print Checks................3
      Accounts Payable............4
      Past Due Accounts...........5
      Quit........................6
```

Here the user must enter the number that corresponds with a desired choice.

As more people are exposed to computers on a regular basis, they begin to expect more from the user interface. The routines in this chapter address several expectations of users concerning menu-driven applications. They display the menu in a manner that is traditional in appearance, but more flexible in functional capabilities, as shown here:

```
        General Ledger Accounting Package

   P Perform Payroll Operations
   R Update Accounts Receivable
   C Print Checks
   A Update Accounts Payable
   L Process Past Due Accounts
   Q Quit General Ledger Returning to DOS



 Press desired key or use arrow keys and press Enter
```

The routines in this chapter support two extremes of user preferences with regard to menus. The traditional "enter the corresponding letter or number" response is fully supported. In front of each menu option is a single character. If the user presses the the key that corresponds to that character, the option is selected.

The second option involves the keyboard arrow keys. The current option is always highlighted. To select a different option, the user simply presses the UP ARROW or DOWN ARROW key to highlight a different option. Once the desired option is highlighted, the user presses the ENTER key to select it.

A significant amount of code can be duplicated in menu-driven programs. In many cases, if you have several menus, you may have simply cut and pasted the required code. However, this chapter develops three standard menu-manipulation routines. Rather than duplicating code, you simply pass a structure containing the required menu information to the routine. The routine, in turn, displays the appropriate menu and entry selections.

The final topic examined in this chapter is pop-up menus. By building on routines presented in Chapter 14, pop-up menu processing becomes relatively simple.

Menu Structure

Each menu routine in this section is based on a menu structure that contains the following:

```
struct menus {
  int num_entries;
  char choices [15];
  char *entries[15];
  char *title;
  char *prompt;
 };
```

Each menu is restricted to a maximum of 14 entries. This restriction is not because of processing, but rather because of ease of use. If your menu contains too many entries, your screen becomes cluttered. Likewise, too many menu entries also become cumbersome to the end user. Given the following structure,

```
struct menus {
  int num_entries;
  char choices [15];
  char *entries[15];
  char *title;
  char *prompt;
```

```
} payroll_menu =
    { 6, "PRCALQ",
        {"Perform Payroll Operations",
         "Update Accounts Receivable",
         "Print Checks",
         "Update Accounts Payable",
         "Process Past Due Accounts",
         "Quit General Ledger Returning to DOS"},
         "General Ledger Accounting Package",
         "Press desired key or use arrow keys and press Enter"
} ;
```

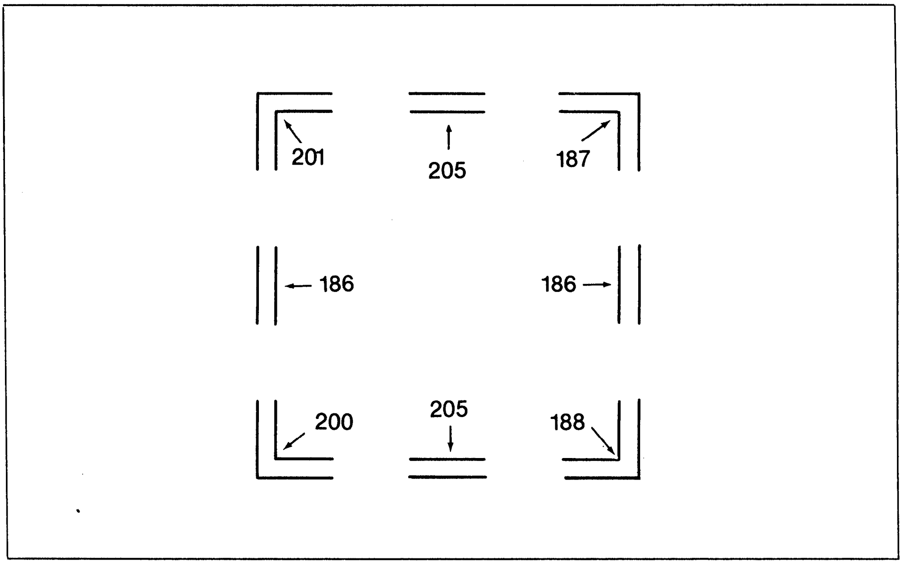
the routine display_menu (which appears later in this chapter) displays the following:

```
        General Ledger Accounting Package

    P Perform Payroll Operations
    R Update Accounts Receivable
    C Print Checks
    A Update Accounts Payable
    L Process Past Due Accounts
    Q Quit General Ledger Returning to DOS



  Press desired key or use arrow keys and press Enter
```

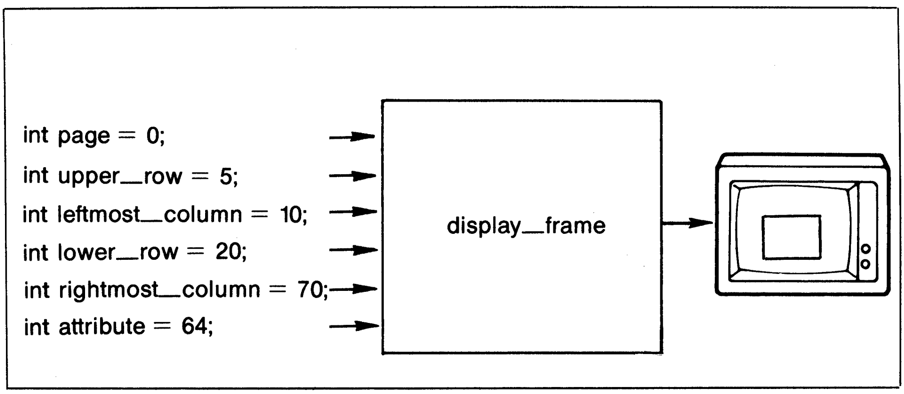
Note the use of the choices field within the menu structure. Each menu option has a character that corresponds to it. The choices field defines those characters.

Framing a Menu

Each of the routines in this chapter provides a frame around the menus it displays. This tends to draw the attention of the user to the menu options. Depending on the number of entries in the menu, the size of the frame will differ from menu to menu. The routines use the extended ASCII character set illustrated here to box the menu:



The following code implements display_frame:

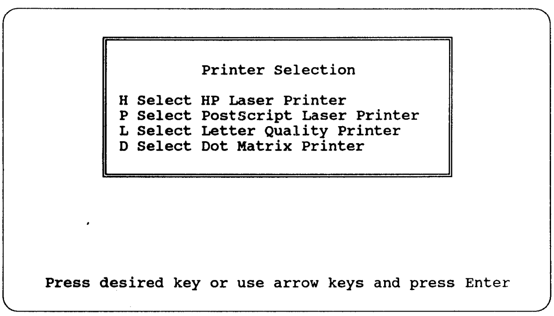


```
/*
 * void display_frame (page, upper_row, leftmost_column,
 *                     lower_row, rightmost_column, attribute)
 *
 * Using extended ASCII characters, display a box on the screen
 * that can be used as a frame for messages or menus.
 *
 * page (in): Desired video display page.
 * upper_row (in): Top row of the display frame.
 * leftmost_column (in): Leftmost column of the display frame.
 * lower_row (in): Lower row of the display frame.
 * attribute (in): Video attribute desired.
 *
 * display_frame (0, 10, 5, 20, 65, 64);
 *
 */

void display_frame (int page, int upper_row,
                    int leftmost_column, int lower_row,
                    int rightmost_column, int attribute)
 {
  int width, mode, current_page, i, segment, offset;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);

  get_video_mode (&width, &mode, &current_page);

  if ((width != 80) || (page > 3))  /* only support 80 columns */
    return ;

  if (mode == 7)
   {
    segment = 0xB000;               /* monochrome */
    page = 0;
   }
  else
    segment = 0xB800;

  /* put in upper corners */

  offset = (page * 4096) + (upper_row * 160);
  memory_map_put (segment, offset + (leftmost_column * 2), 201);
  memory_map_put (segment, offset + (leftmost_column * 2) + 1,
                  attribute);
  memory_map_put (segment, offset + (rightmost_column * 2), 187);
  memory_map_put (segment, offset + (rightmost_column * 2) + 1,
                  attribute);

  /* top row of frame */

  for (i = leftmost_column+1; i <= rightmost_column-1; i++)

 {
    memory_map_put (segment, offset + (i * 2), 205);
    memory_map_put (segment, offset + (i * 2) + 1, attribute);
 }

/* put in bottom corners */

offset = (page * 4096) + (lower_row * 160);
memory_map_put (segment, offset + (leftmost_column * 2), 200);
memory_map_put (segment, offset + (leftmost_column * 2) + 1,
                attribute);
```

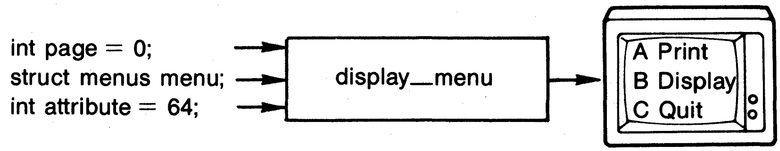
```
memory_map_put (segment, offset + (rightmost_column * 2), 188);
memory_map_put (segment, offset + (rightmost_column * 2) + 1,
                attribute);

/* bottom row of frame */

for (i = leftmost_column+1; i <= rightmost_column-1; i++)
 {
    memory_map_put (segment, offset + (i * 2), 205);
    memory_map_put (segment, offset + (i * 2) + 1, attribute);
 }

/* put in the sides */

for (i = upper_row + 1; i <= lower_row -1; i++)
 {
   offset = (page * 4096) + (i * 160);
   memory_map_put (segment, offset + (leftmost_column * 2), 186);
   memory_map_put (segment, offset + (leftmost_column * 2) + 1,
                   attribute);
   memory_map_put (segment, offset + (rightmost_column * 2), 186);
   memory_map_put (segment, offset + (rightmost_column * 2) + 1,
                   attribute);
 }
}
```

Displaying and Using a Menu

The following routine displays a menu on the screen. As previously stated, the menu is surrounded by a frame, and provides a user prompt at the bottom of the screen, as shown here:



The following routine, display_memo, displays a menu:

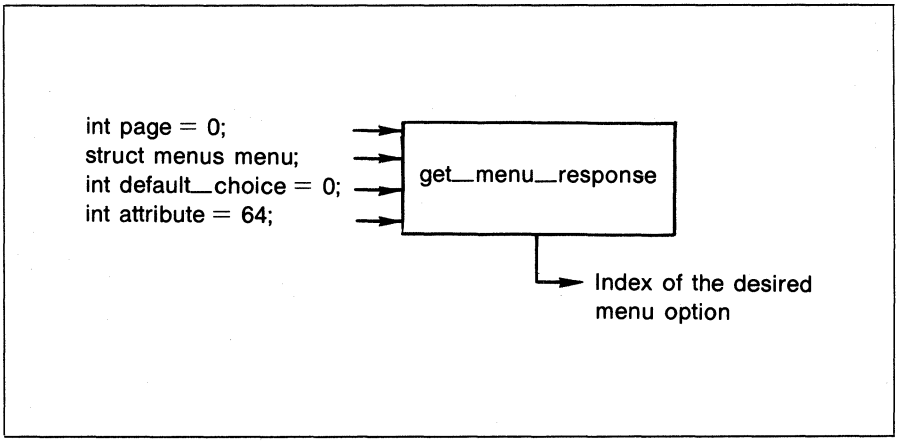



```
/*
 * void display_menu (page, menu, attribute)
 *
 * Display a menu on the video display page specified.
 *
 * page (in): Video display page desired.
 * menu (in): Structure containing desired menu.
 * attribute (in): Video display attribute for the menu.
 *
 * display_menu (0, main_menu, 7);
 *
 */

void display_menu (int page, struct menus menu, int attribute)
 {
  int upper_row, lower_row, leftmost_column, rightmost_column;
  int title_row, prompt_row, title_column, prompt_column, max_size;
  int width, mode, current_page, i, j, segment, offset;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);
  void display_frame (int, int, int, int, int, int);

  get_video_mode (&width, &mode, &current_page);

  if ((width != 80) || (page > 3))
    return ;            /* only support 80 column */

  if (mode == 7)        /* monochrome system */
   {
    segment = 0xB000;
    page = 0;
   }
  else
    segment = 0xB800;

  /* determine upper and lower frame row */

  upper_row = 13 - 4 - (menu.num_entries / 2);
  lower_row = 13 + 1 + ((menu.num_entries % 2) ?
          ((menu.num_entries + 1) / 2): (menu.num_entries / 2));
```

```
/* determine title location and display title */

title_row = upper_row + 2;
for (i = 0; menu.title[i]; i++) ;
title_column = 39 - (i / 2);
offset = (page * 4096) + (title_row * 160) + (title_column * 2);

for (i = 0; menu.title[i]; i++)
 {
  memory_map_put (segment, offset + (i * 2), menu.title[i]);
  memory_map_put (segment, offset + (i * 2) + 1, attribute);

 }

max_size = i;    /* largest string display thus far */
                 /* frame size is relative */

/* determine largest string for frame size */

for (i = 0; i < menu.num_entries; i++)
 {
  for (j = 0; menu.entries[i][j]; j++)
    ;
  if (j > max_size)
      max_size = j;
 }

/* center menu on column 39 */

leftmost_column = 39 - ((max_size + 8) /2);
rightmost_column = 39 + ((max_size + 8) /2);

display_frame (page, upper_row, leftmost_column,
               lower_row, rightmost_column, 7);

/* display the menu choices */

for (i = 0; i < menu.num_entries; i++)
 {
  offset = (page * 4096) + ((title_row + 2 + i) * 160);
  memory_map_put (segment, offset + (leftmost_column + 2) * 2,
        menu.choices[i]);
  memory_map_put (segment, offset + (leftmost_column + 2) * 2 + 1,
        attribute);

  offset = (page * 4096) + ((title_row + 2 + i) * 160) +
           (leftmost_column + 4) * 2;

  for (j = 0; menu.entries[i][j]; j++)
   {
     memory_map_put (segment, offset + (j * 2), menu.entries[i][j]);
     memory_map_put (segment, offset + (j * 2) + 1, attribute);
   }
 }

/* determine the location for the menu prompt and display it */

prompt_row = 23;
for (i = 0; menu.prompt[i]; i++) ;
prompt_column = 39 - (i / 2);
offset = (page * 4096) + (prompt_row * 160) + (prompt_column * 2);
for (i = 0; menu.prompt[i]; i++)
```

```
    {
     memory_map_put (segment, offset + (i * 2), menu.prompt[i]);
     memory_map_put (segment, offset + (i * 2) + 1, attribute);
    }
 }
```

Once the menu is displayed, the following routine obtains the user selection:




```
/*
 * get_menu_response (page, menu, default_choice, attribute)
 *
 * Get the user's response to the menu specified.
 *
 * page (in): Video display page desired.
 * menu (in): Structure containing the desired menu.
 * default_choice (in): Default menu option.
 * attribute (in): Desired video attribute.
 *
 * selection = get_menu_response (0, main_menu, 0, 64);
 *
 */

get_menu_response (int page, struct menus menu,
                    int default_choice, int attribute)
 {
  int upper_row, leftmost_column;
  int width, mode, current_page, i, j, segment, offset;
  int done = 0, row, old_row, max_size, choice, valid_key;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);
```

```
get_video_mode (&width, &mode, &current_page);

if ((width != 80) || (page > 3))
  return ;       /* only support 80 column */

if (mode == 7)  /* monochrome system */
 {
  segment = 0xB000;
  page = 0;
 }
else
  segment = 0xB800;

upper_row = 13 - 4 - (menu.num_entries / 2);

for (i = 0; menu.title[i]; i++) ;

max_size = i;    /* largest string display thus far */
                 /* frame size is relative */

/* determine the largest string in the menu */

for (i = 0; i < menu.num_entries; i++)
 {
  for (j = 0; menu.entries[i][j]; j++)
    ;
  if (j > max_size)
      max_size = j;

 }

/* determine the leftmost column based on centering */

leftmost_column = 39 - ((max_size + 8) /2);
row = default_choice;

/* get the user response */

while (! done)
 {
  /* determine and highlight the current row */

  offset = (page * 4096) + ((upper_row + 4 + row) * 160);
  memory_map_put (segment, offset + (leftmost_column + 2) * 2 + 1,
        7);
  offset = (page * 4096) + ((upper_row + 4 + row) * 160) +
           (leftmost_column + 4) * 2;

  for (j = 0; menu.entries[row][j]; j++)
     memory_map_put (segment, offset + (j * 2) + 1, 7);

  choice = no_echo_read();
  valid_key = 0;
  old_row = row;

  /* see if the user pressed a function or arrow key */

  if (choice == 0)
    {
     choice = no_echo_read();
     switch (choice) {
       case 72: if (row == 0)  /* up arrow */
```

```
                row = menu.num_entries - 1;
              else
                row--;
              valid_key = 1;
              break;

      case 80: if (row == menu.num_entries - 1)
                row = 0;      /* down arrow */
              else
                row++;
              valid_key = 1;
              break ;
     }
    if (valid_key)
     {
      /* dehighlight previous row, highlight new row */

      offset = (page * 4096) + ((upper_row + 4 + old_row) * 160);
      memory_map_put (segment, offset + (leftmost_column + 2) * 2 + 1
        attribute);
      offset = (page * 4096) + ((upper_row + 4 + old_row) * 160) +
          (leftmost_column + 4) * 2;
      for (j = 0; menu.entries[old_row][j]; j++)
          memory_map_put (segment, offset + (j * 2) + 1, attribute);

      offset = (page * 4096) + ((upper_row + 4 + row) * 160);
      memory_map_put (segment, offset + (leftmost_column + 2) * 2 + 1
         7);
      offset = (page * 4096) + ((upper_row + 4 + row) * 160) +
          (leftmost_column + 4) * 2;
      for (j = 0; menu.entries[row][j]; j++)
          memory_map_put (segment, offset + (j * 2) + 1, 7);
     }
   }
  else
   {
    if (choice == 13)  /* carriage return */
      return (row);
    else               /* test letter pressed */
      for (i = 0; i < menu.num_entries; i++)
        if ((choice == menu.choices[i]) ||
           ((choice & ~32) == menu.choices[i]))
          return (i);
   }
 }
}
```

The routines are implemented separately simply to reduce the amount of code in each function.

Pop-Up Menus

Most applications that use menus to prompt the user for information normally clear the screen display and then place the menu on a blank screen. However, in some cases, it is more convenient for the end user to leave the current display active and to place the menu in a corner of the screen display, as shown here:

```
+------------------------------+
|     Source Code Display      |tate: California       Zip: 81203
|                              |
| A Display Ada Programs       |bats right, 415 lifetime average.
| B Display BASIC Programs     |
| C Display C Programs         |with the Giants.
| D Display DBASE Programs     |
+------------------------------+

Press desired key or use arrow keys and press Enter
```

Once the user makes a selection, the menu disappears, as shown here:

```
Enter Name: Kevin Shafer

Address: 1234 - First Ave

City: San Francisco     State: California       Zip: 81203

Description: Short stop, bats right, 415 lifetime average.

             Brief stint with the Giants.
```

Such a menu is called a *pop-up menu,* since it apparently appears from nowhere and overlays the current contents of the screen. Popup menu processing is quite straightforward. First, you simply save to a buffer (containing characters and their attributes) the contents of the section of the display that you will overwrite. Next, you display and process the menu that obtains the user selection. Lastly, you must restore the screen contents that were previously buffered (see Figure 15-1).

Using the routines buffer_video_region and restore_video_region presented in Chapter 14, the processing becomes quite simple. The routines are based on the menu type previously discussed. The only difference is that the routines now allow you to pass the coordinates of the upper-left corner of the menu.

The following routine displays a video pop-up menu to the screen. Assuming that the menu structure is as follows,

```
struct menus {
  int num_entries;
  char choices [15];
  char *entries[15];
  char *title;
  char *prompt;
} prt_menu = { 4, "HPLD",
                 {"Select HP Laser Printer",
                  "Select PostScript Laser Printer",
                  "Select Letter Quality Printer",
                  "Select Dot Matrix Printer"},
                  "Printer Selection",
                  "Press desired key or use arrow keys and press Enter"
} ;
```

the invocation

```
display_popup_menu (0, 1, 2, prt_menu, 7);
```

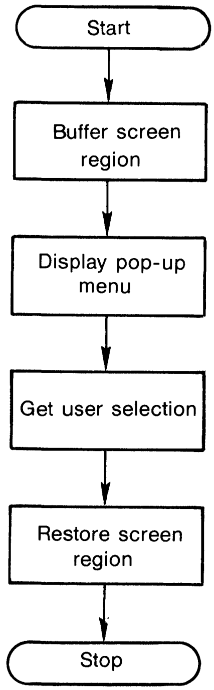
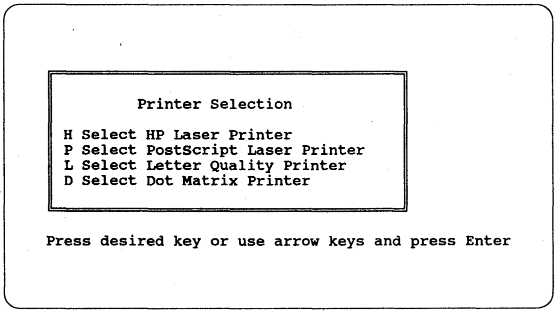
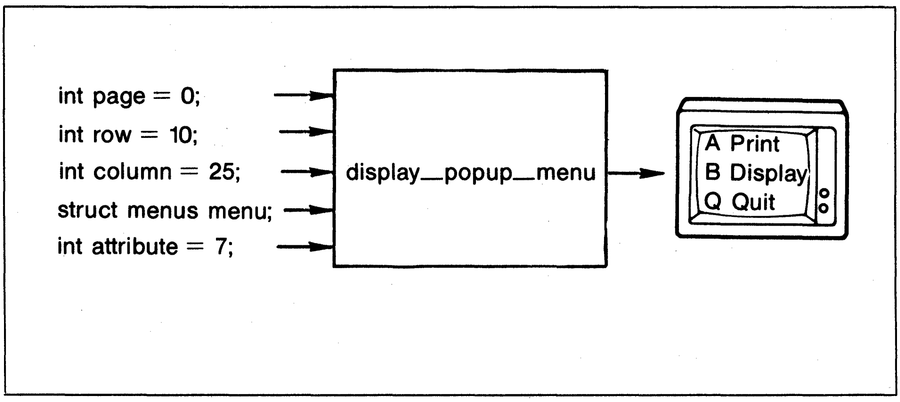


Figure 15-1. *Processing involved in pop-up menu display*

displays the following:



The following routine implements display_popup_menu.

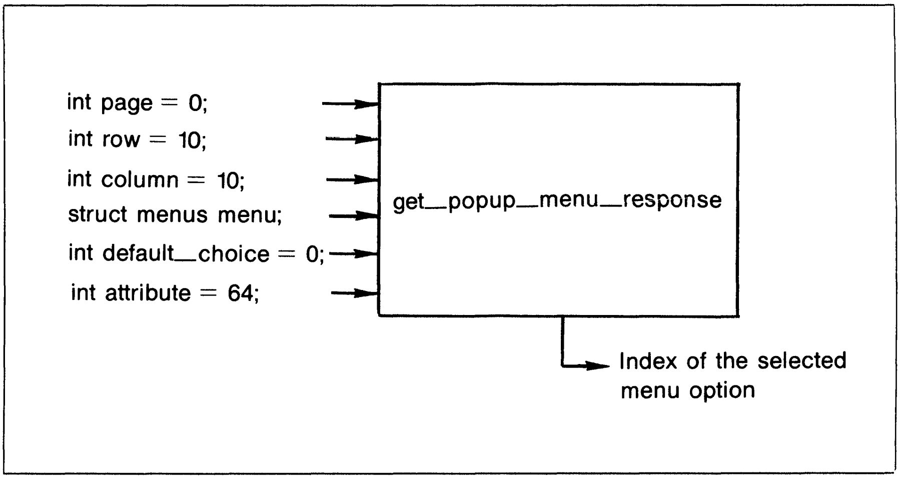



```
/*
 * void display_popup_menu (page, row, column, menu, attribute)
 *
 * Display the video popup menu specified.  Save the previous
 * screen contents restoring them once the popup is complete.
 *
 * page (in): Video display page desired.
 * row (in): Desired upper row for the popup menu.
 * column (in): Desired leftmost column for the popup menu.
 * menu (in): Structure containing the popup menu.
```

```
 * attribute: Video display attribute desired.
 *
 * display_popup_menu (0, 0, 0, printer_menu, 64);
 *
 * display_popup_menu relies on a global variable called buffer
 * that it can store the current screen contents into.  By making
 * this variable global, it is easily accessed by the routine
 * get_popup_menu_response which later restores the screen.
 *
 */

void display_popup_menu (int page, int row, int column,
           struct menus menu, int attribute)
 {
  int upper_row, lower_row, leftmost_column, rightmost_column;
  int title_row, prompt_row, title_column, prompt_column, max_size;
  int title_size, width, mode, current_page, i, j, segment, offset;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);
  void buffer_video_region (int, int, int, int, int, char *);
  void display_frame (int, int, int, int, int, int);

  get_video_mode (&width, &mode, &current_page);

  if ((width != 80) || (page > 3))
    return ;        /* only support 80 column */

  if (mode == 7)    /* monochrome system */
   {
    segment = 0xB000;
    page = 0;
   }
  else
    segment = 0xB800;

  /* save the previous screen contents */

  buffer_screen_region (page, 0, 24, 0, 79, buffer);

  upper_row = row;
  lower_row = row + 5 + menu.num_entries;
  title_row = upper_row + 2;

  for (i = 0; menu.title[i]; i++) ;

  max_size = i;   /* largest string display thus far */
                  /* frame size is relative */

  title_size = i;

  /* determine the largest string in the menu */

  for (i = 0; i < menu.num_entries; i++)
   {
    for (j = 0; menu.entries[i][j]; j++)
      ;
    if (j > max_size)
        max_size = j;
   }

  leftmost_column = column;
  rightmost_column = max_size + column + 8;
```

```
/* clear the screen region which will contain the menu */

for (i = row; i < row + 6 + menu.num_entries; i++)
 {
  offset = (page * 4096) + (i * 160);
  for (j = leftmost_column; j <= rightmost_column; j++)
    memory_map_put (segment, offset + (j * 2) + 1, 0);
 }

/* determine the location of the title and display it */

title_column = ((column + 8 + max_size) / 2) - (title_size / 2);
offset = (page * 4096) + (title_row * 160) + (title_column * 2);

for (i = 0; menu.title[i]; i++)
 {
  memory_map_put (segment, offset + (i * 2), menu.title[i]);
  memory_map_put (segment, offset + (i * 2) + 1, attribute);
 }

display_frame (page, upper_row, leftmost_column,
               lower_row, rightmost_column, 7);

/* display the menu choices */

for (i = 0; i < menu.num_entries; i++)
 {
  offset = (page * 4096) + ((title_row + 2 + i) * 160);
  memory_map_put (segment, offset + (leftmost_column + 2) * 2,
        menu.choices[i]);
  memory_map_put (segment, offset + (leftmost_column + 2) * 2 + 1,
        attribute);
  offset = (page * 4096) + ((title_row + 2 + i) * 160) +
           (leftmost_column + 4) * 2;

  for (j = 0; menu.entries[i][j]; j++)
   {
     memory_map_put (segment, offset + (j * 2), menu.entries[i][j]);
     memory_map_put (segment, offset + (j * 2) + 1, attribute);
   }
 }

/* determine the location of the prompt and display it */

prompt_row = row + 7 + menu.num_entries;
for (i = 0; menu.prompt[i]; i++) ;
prompt_column = column;
offset = (page * 4096) + (prompt_row * 160) + (prompt_column * 2);

for (i = 0; menu.prompt[i]; i++)
 {
  memory_map_put (segment, offset + (i * 2), menu.prompt[i]);
  memory_map_put (segment, offset + (i * 2) + 1, attribute);
 }
}
```

The following routine obtains a user response to the menu and then restores the previous screen contents:




```
/*
 * get_popup_menu_response (page, row, column, menu,
 *                          default_choice, attribute)
 *
 * Get the user's response to a video popup menu.  Once the
 * response is known, restore the previous screen contents.
 *
 * page (in): Video display page desired.
 * row (in): Upper row of the display frame.
 * column (in): Leftmost column of the display frame.
 * menu (in): Structure containing the popup menu.
 * default_choice (in): Default menu option.
 * attribute (in): Video display attribute desired.
 *
 * get_popup_menu_response (0, 0, 0, printer_menu, 0, 64);
 *
 * get_popup_menu_response uses a global variable called buffer
 * which contains the screen contents to restore.
 *
 */

get_popup_menu_response (int page, int row, int column,
          struct menus menu, int default_choice, int attribute)
 {
  int upper_row, leftmost_column;
  int width, mode, current_page, i, j, segment, offset;
  int done = 0, old_row, max_size, choice, valid_key;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);
  void display_frame (int, int, int, int, int, int);

  get_video_mode (&width, &mode, &current_page);

  if ((width != 80) || (page > 3))
```

```
    return ;       /* only support 80 column */

  if (mode == 7)   /* monochrome system */
   {
    segment = 0xB000;
    page = 0;
   }
  else
    segment = 0xB800;

  upper_row = row;
  row = default_choice;
  leftmost_column = column;

  while (! done)
   {

  /* highlight the current option */

  offset = (page * 4096) + ((upper_row + 4 + row) * 160);
  memory_map_put (segment, offset + (leftmost_column + 2) * 2 + 1,
        7);

  offset = (page * 4096) + ((upper_row + 4 + row) * 160) +
           (leftmost_column + 4) * 2;

  for (j = 0; menu.entries[row][j]; j++)
     memory_map_put (segment, offset + (j * 2) + 1, 7);

  choice = no_echo_read();
  valid_key = 0;
  old_row = row;

  /* test if user pressed function or arrow key */

  if (choice == 0)
    {
     choice = no_echo_read();
     switch (choice) {
       case 72: if (row == 0)   /* up arrow */
                  row = menu.num_entries - 1;
                else
                  row--;
                valid_key = 1;
                break;

       case 80: if (row == menu.num_entries - 1)
                  row = 0;       /* down arrow */
                else
                  row++;
                valid_key = 1;
                break ;
      }
     if (valid_key)
      {
       /* dehighlight previous option, hightlight new row */

       offset = (page * 4096) + ((upper_row + 4 + old_row) * 160);
       memory_map_put (segment, offset + (leftmost_column + 2) * 2 +
         attribute);
       offset = (page * 4096) + ((upper_row + 4 + old_row) * 160) +
           (leftmost_column + 4) * 2;
       for (j = 0; menu.entries[old_row][j]; j++)
           memory_map_put (segment, offset + (j * 2) + 1, attribute);
```

```
         offset = (page * 4096) + ((upper_row + 4 + row) * 160);
         memory_map_put (segment, offset + (leftmost_column + 2) * 2 +
            7);
         offset = (page * 4096) + ((upper_row + 4 + row) * 160) +
             (leftmost_column + 4) * 2;
         for (j = 0; menu.entries[row][j]; j++)
             memory_map_put (segment, offset + (j * 2) + 1, 7);
         }
       }
     else
       {
        if (choice == 13)  /* carriage return */
         {
           restore_screen_region (page, 0, 24, 0, 79, buffer);
           return (row);
         }
        else               /* test the letter entered */
          for (i = 0; i < menu.num_entries; i++)
            if ((choice == menu.choices[i]) ||
               ((choice & ~32) == menu.choices[i]))
             {
               restore_screen_region (page, 0, 24, 0, 79, buffer);
               return (i);
             }
       }
    }
  }
```

Advanced Video Pop-Up Menus

Video pop-up menus can be easily implemented. With them, your screen processing capabilities are virtually unlimited. Consider the following pop-up, which allows the user to add, subtract, multiply, or divide two numbers.

```
Value: 0.00000000

Value:

Result:

F7  Addition        F8  Subtraction
F9  Multiplication  F10 Division


Enter first value.
```

The routine can be very useful in cases where the user must enter a numeric response. If your program allows the user to press the F9 key, for example, to activate the pop-up, the user can first perform calculations and then respond to the numeric prompt. Once the user selects the Quit option, the original screen contents are restored. For example, if the user needs to know the result of 625 divided by 17, the first entry would be

```
Value: 625.000000

Value:

Result:

F7  Addition        F8  Subtraction
F9  Multiplication  F10 Division


Enter first value.
```

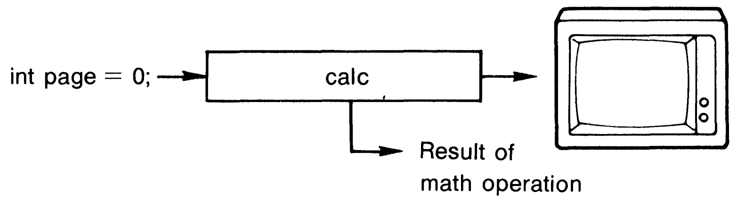
followed by

```
Value: 625.000000

Value: 17.0000000

Result:

F7  Addition        F8  Subtraction
F9  Multiplication  F10 Division


Enter second value.
```

Once the user presses the F10 key for division, the result is displayed and the user is asked to press any key to continue.

```
Value: 625.000000

Value: 17.0000000

Result: 36.64705

F7  Addition        F8  Subtraction
F9  Multiplication  F10 Division


Press any key to continue
```

The following routine implements the pop-up math processor:




```
/*
 * float calc (page)
 *
 * Display a video popup calculator on the display page specified.
 *
 * page (in): Desired video display page.
 *
 * result = calc (0);
 *
 * calc saves the current screen contents and then displays a
 * simple calculator.  Once the operation is complete, calc
 * restores the previous screen contents and returns the result
 * of the operation.
 *
 */

float calc (int page)
 {
  int upper_row, lower_row, leftmost_column, rightmost_column;
  int attribute, width, mode, current_page, i, j, segment, offset;
  int key, done = 0;

  char buffer[8000];

  float a, b, result;

  void get_video_mode (int *, int *, int *);
  void memory_map_put (int, int, int);
  void buffer_video_region (int, int, int, int, int, char *);
  void put_float (float, int, int, int, int, int, int);
  void put_string (char *, int, int, int, int, int);

  float get_prompted_float (float, int, int, int, int, int,
           int, char *, int, int, int, char *, int, int, int, int);

  attribute = 7;

  get_video_mode (&width, &mode, &current_page);
```

```
if ((width != 80) || (page > 3))
  return ;      /* only support 80 column */

if (mode == 7)  /* monochrome system */
 {
  segment = 0xB000;
  page = 0;
 }
else
  segment = 0xB800;

buffer_screen_region (page, 0, 24, 0, 79, buffer);

upper_row = 0;
lower_row = upper_row + 14;
leftmost_column = 0;
rightmost_column = 79;

/* clear the screen region to be used by the calculator */

for (i = upper_row; i < lower_row; i++)
 {
  offset = (page * 4096) + (i * 160);
  for (j = leftmost_column; j <= rightmost_column; j++)
    memory_map_put (segment, offset + (j * 2) + 1, 0);
 }

/* display the upper row of the frame */

offset = (page * 4096) + (upper_row * 160);
memory_map_put (segment, offset + (leftmost_column * 2), 201);
memory_map_put (segment, offset + (leftmost_column * 2) + 1,
                attribute);
memory_map_put (segment, offset + (rightmost_column * 2), 187);
memory_map_put (segment, offset + (rightmost_column * 2) + 1,
                attribute);

for (i = leftmost_column+1; i <= rightmost_column-1; i++)
 {
    memory_map_put (segment, offset + (i * 2), 205);
    memory_map_put (segment, offset + (i * 2) + 1, attribute);
 }

/* display the lower row of the frame */

offset = (page * 4096) + (lower_row * 160);
memory_map_put (segment, offset + (leftmost_column * 2), 200);
memory_map_put (segment, offset + (leftmost_column * 2) + 1,
                attribute);

memory_map_put (segment, offset + (rightmost_column * 2), 188);
memory_map_put (segment, offset + (rightmost_column * 2) + 1,
                attribute);

for (i = leftmost_column+1; i <= rightmost_column-1; i++)
 {
    memory_map_put (segment, offset + (i * 2), 205);
    memory_map_put (segment, offset + (i * 2) + 1, attribute);
 }
```

```
put_string ("Value:", page, 2, 4, attribute, 10);
put_string ("Value:", page, 4, 4, attribute, 10);
put_string ("Result:", page, 6, 4, attribute, 10);
put_string ("F7  Addition        F8  Subtraction", page,
      8, 4, attribute, 40);
put_string ("F9  Multiplication  F10 Divsion", page,
      9, 4, attribute, 40);
 a = get_prompted_float (0, page, 2, 12,
              10, 7, 7, "Value:", 2, 4, 10,
              "Enter first value.", 12, 4, 25, 7);

 b = get_prompted_float (0, page, 4, 12,
              10, 7, 7, "Value:", 4, 4, 10,
              "Enter second value.", 12, 4, 25, 7);

 put_string ("Select function key desired ", 0, 12,
              4, 7, 30);

 while (! done)
  {
   key = no_echo_read();

   if (key == 0)  /* need a function key */
    {
     key = no_echo_read ();

     if (key == 65)       /* F7 */
       {
        result = a + b;
        done = 1;
       }
     else if (key == 66)  /* F8 */
       {
        result = a - b;
        done = 1;
       }
     else if (key == 67)  /* F9 */
       {
        result = a * b;
        done = 1;
       }
     else if (key == 68)  /* F10 */
       {
        result = a / b;
        done = 1;
       }
    }
  }

 put_float (result, page, 6, 12, 7, 20, 5);

 put_string ("Press any key to continue    ", 0, 12,
              4, 7, 30);

 key = no_echo_read ();
 if (key == 0) no_echo_read ();  /* read second half of
                                    special or function key */

 restore_screen_region (page, 0, 24, 0, 79, buffer);

 return (result);
}
```

Menus and video pop-up menus can be used effectively to control the user interface. The routines in this chapter are meant to provide you with the foundation from which you can develop more powerful routines in the future.

A P P E N D I X

A

ASCII Codes

Table A-1 lists the ASCII codes for characters.

Table A-1. *ASCII Character Codes*

DEC	OCTAL	HEX	ASCII	DEC	OCTAL	HEX	ASCII
0	000	00	NUL	10	012	0A	LF
1	001	01	SOH	11	013	0B	VT
2	002	02	STX	12	014	0C	FF
3	003	03	ETX	13	015	0D	CR
4	004	04	EOT	14	016	0E	SO
5	005	05	ENQ	15	017	0F	SI
6	006	06	ACK	16	020	10	DLE
7	007	07	BEL	17	021	11	DC1
8	010	08	BS	18	022	12	DC2
9	011	09	HT	19	023	13	DC3

Table A-1. ASCII Character Codes (continued)

DEC	OCTAL	HEX	ASCII	DEC	OCTAL	HEX	ASCII
20	024	14	DC4	64	100	40	@
21	025	15	NAK	65	101	41	A
22	026	16	SYN	66	102	42	B
23	027	17	ETB	67	103	43	C
24	030	18	CAN	68	104	44	D
25	031	19	EM	69	105	45	E
26	032	1A	SUB	70	106	46	F
27	033	1B	ESC	71	107	47	G
28	034	1C	FS	72	110	48	H
29	035	1D	GS	73	111	49	I
30	036	1E	RS	74	112	4A	J
31	037	1F	US	75	113	4B	K
32	040	20	SPACE	76	114	4C	L
33	041	2i	!	77	115	4D	M
34	042	22	”	78	116	4E	N
35	043	23	#	79	117	4F	O
36	044	24	$	80	120	50	P
37	045	25	%	81	121	51	Q
38	046	26	&	82	122	52	R
39	047	27	’	83	123	53	S
40	050	28	(	84	124	54	T
41	051	29	)	85	125	55	U
42	052	2A	*	86	126	56	V
43	053	2B	+	87	127	57	W
44	054	2C	,	88	130	58	X
45	055	2D	—	89	131	59	Y
46	056	2E	.	90	132	5A	Z
47	057	2F	/	91	133	5B	[
48	060	30	0	92	134	5C	\
49	061	31	1	93	135	5D	]
50	062	32	2	94	136	5E	^
51	063	33	3	95	137	5F	—
52	064	34	4	96	140	60	‘
53	065	35	5	97	141	61	a
54	066	36	6	98	142	62	b
55	067	37	7	99	143	63	c
56	070	38	8	100	144	64	d
57	071	39	9	101	145	65	e
58	072	3A	:	102	146	66	f
59	073	3B	;	103	147	67	g
60	074	3C	<	104	150	68	h
61	075	3D	=	105	151	69	i
62	076	3E	>	106	152	6A	j
63	077	3F	?	107	153	6B	k

Table A-1. *ASCII Character Codes (continued)*

DEC	OCTAL	HEX	ASCII	DEC	OCTAL	HEX	ASCII
108	154	6C	l	118	166	76	v
109	155	6D	m	119	167	77	w
110	156	6E	n	120	170	78	x
111	157	6F	o	121	171	79	y
112	160	70	p	122	172	7A	z
113	161	71	q	123	173	7B	{
114	162	72	r	124	174	7C	|
115	163	73	s	125	175	7D	}
116	164	74	t	126	176	7E	~
117	165	75	u	127	177	7F	DEL

APPENDIX

B

Turbo C Run-Time Library

This appendix provides you with the calling sequence and function of each routine in the Turbo C run-time library. As stated earlier in this text, the goal of developing a complete library of routines is to reduce duplication of effort. As such, it is very important that you become familar with the Turbo C run-time library.

As you will find, Borland provides you with a myriad of routines that you can use extensively within your applications. Take some time now to examine the Borland Turbo C run-time library.

void abort (void);

Function Writes a termination message to stderr, aborting the current application by invoking _exit(3).

Include File <stdlib.h>

Example

```
abort();
```

int abs (int);

Function Returns the absolute value of the specified integer value.

Include File <stdlib.h>

Example

```
result = abs (a * b);
```

Note If you do not include stdlib.h, Turbo C will invoke abs as a macro, as opposed to a function. The abs routine will return a value in the range 0 to 32,767.

*int absread (int disk, int num_sectors, int first_sector, void *buffer);*

Function Reads a physical sector or sectors from disk into the specified buffer.

Include File <dos.h>

disk (in):	Disk drive desired (A = 0, B = 1, C = 2)
num_sectors (in):	Number of sectors to read
first_sector (in):	Starting sector number for read
buffer (out):	Buffer to read disk information into

Example

status = absread (0, 1, 0, boot_record);

Note If successful, absread returns 0; otherwise, it returns −1.

*int abswrite (int disk, int num_sectors, int first_sector, void *buffer);*

Function Writes a physical sector or sectors from disk to the specified buffer.

Include File <dos.h>

disk (in):	Disk drive desired (A = 0, B = 1, C = 2)
num_sectors (in):	Number of sectors to write
first_sector (in):	Starting sector number for write
buffer (out):	Buffer to write disk information from

Example

status = abswrite (0, 1, 0, boot_record);

Note If successful, abswrite returns 0; otherwise, it returns −1.

*int access (char *filename, int access_mode);*

Function Determines if the specified file exists, and if so, how the file can be accessed.

Include File <io.h>

filename (in):	File desired	
access_mode (in):	Bit pattern specifying the mode of access desired:	
	0 File existence	1 Executable
	2 Writeable	4 Readable
	6 Read/write access	

Example

status = access ("TURBO.NTS", 0);

Note If the access mode is valid for the specified file, access returns the value 0; otherwise, it returns −1.

double acos (double);

Function Returns the arc cosine of the specified expression.

Include File <math.h>

Example

result = acos (pi);

*int allocmem (unsigned paragraphs, unsigned *segment_address);*

Function Allocates a DOS memory segment.

Include File <dos.h>

paragraphs (in):	Number of 16-byte paragraphs desired
segment_address (in):	Pointer to the segment address returned

Example

```
status = allocmem (1000, &seg_addr);
```

Note If successful, allocmem returns the value −1; otherwise, it returns the size of the largest available block. You should not use this function with malloc or calloc, farmalloc or faralloc.

void far arc (int xloc, int yloc, int start_angle, int end_angle, int radius);

Function Draws a circular arc at the specified x,y location, using the starting and ending angles provided.

Include File <graphics.h>

xloc, yloc (in):	Specifies the center of the arc
start_angle (in):	Starting angle for the arc (0-360)
end_angle (in):	Ending angle for the arc (0-360)
radius (in):	Radius of the arc

Example

arc (100, 200, 0, 360, 10);

Note The arc routine uses the current drawing color.

*char *asctime (struct tm *time);*

Function Converts a date and time to its ASCII representation.

Include File <time.h>

Example

time_string = asctime (&current_datetime);

double asin (double);

Function Returns the arc sine of the specified expression.

Include File <math.h>

Example

result = asin

void assert (int condition);

Function Tests the specified condition. If the test fails, assert terminates the current program and displays the message

Assertion failed: file PROGRAM, line LINE_NUMBER

Include File <assert.h>

condition (in): Boolean expression to test

Example

assert(argc > 1);

Note If you set the NDEBUG directive to no debugging prior to an assert, Turbo C will ignore the assert.

double atan (double);

Function Returns the arc tangent of the specified expression.

Include File <math.h>

Example

result = atan (pi)

double atan2 (double, double);

Function Returns the arc tangent of x and y expressions.

Include File <math.h>

Example

result = atan2 (y, x);

int atexit (atexit_t function_name);

Function Defines a function that Turbo C will invoke (without arguments) at program termination prior to returning control to the operating system.

Include File <stdlib.h>

function_name (in):	Entry point of function to add to exit list

Example

```
  void test ()
{
```

```
    printf ("Test called \n");
  }
main ()
  {
    status = atexit (test);
  }
```

Note If successful, atexit returns the value 0. If an error occurs, atexit returns a nonzero value.

*double atof (char *str);*

Function Converts an ASCII string representation of a value to a floating-point value.

Include File <math.h>

str (in):	ASCII representation of value

Example

```
salary = atof ("77500.34");
```

Note If the string cannot be converted, atof returns 0.

*int atoi (char *str);*

Function Converts a string representation of an integer value to a value of type int.

Include File <stdlib.h>

str (in): ASCII representation of the value

Example

age = atoi ("59");

Note If the string cannot be converted, atoi returns 0.

*long atol (char *str);*

Function Converts an ASCII representation of a value to a value of type long int.

Include File <stdlib.h>

Example

zip_code = atol ("89126");

Note If the string cannot be converted, atol returns the value 0.

void far bar (int left_corner, int top_corner, int right_corner, int bottom_corner);

Function Draws a bar (for a bar graph) with the specified corners.

Include File <graphics.h>

left_corner (in):	Specifies the x coordinate of the upper-left corner
top_corner (in):	Specifies the y coordinate of the upper-left corner
bottom_corner (in):	Specifies the y coordinate of the lower-right corner
right_corner (in):	Specifies the x coordinate of the lower-right corner

Example

```
bar (10, 10, sales, offset);
```

Note The bar routine uses the current fill pattern and color.

void far bar3d (int left_corner, int top_corner, int right_corner, int bottom_corner, int depth, int top_flag);

Function Draws a bar (for a bar graph) with the specified corners.

Include File <graphics.h>

left_corner (in):	Specifies the x coordinate of the upper-left corner
top_corner (in):	Specifies the y coordinate of the upper-left corner
bottom_corner (in):	Specifies the y coordinate of the lower-right corner
right_corner (in):	Specifies the x coordinate of the lower-right corner
depth (in):	Depth of the bar in pixels
top_flag (in):	If 0, the bar is not drawn in 3D. If nonzero, the number of pixels specified by depth are added to the bar

Example

```
bar3d (10, 10, sales, offset,3, 1);
```

Note The bar3d routine uses the current fill pattern and color.

int bdos (int DOSfunction, unsigned dx, unsigned al);

Function Invokes a DOS function (small memory model) that only requires the DX and AX registers.

Include File <dos.h>

DOSfunction (in):	DOS service to be performed
dx (in):	DX register contents for service
al (in):	AL register contents for service

Example

```
current_drive = bdos (0×19, 0, 0);
```

Note The bdos routine returns the contents of the AX register or the value −1 if an error occurs.

*int bdosptr (int DOSfunction, void *parameter, unsigned al);*

Function Invokes a DOS function that requires a pointer to an argument and AX registers.

Include File <dos.h>

DOSfunction (in):	DOS service to perform
parameter (in):	Parameter required for the service
al (in):	AL register contents for service

Example

```
current_drive = bdosptr (0×19, 0, 0);
```

Note The bdosptr routine returns the contents of the AX register or the value −1 if an error occurs.

int bioscom (int command, char byte, int port_id);

Function Performs serial I/O through the specified port.

Include File <bios.h>

command (in):	Operation to perform: 0 Set port 1 Send character 2 Receive char 3 Return status
byte (in):	Port settings or char to output
port_id (in):	Serial port ID (0-com1, 1-com2)

Example

```
status = bioscom (1, 65, 0);
```

Note Refer to bios.h for more specifics on port settings.

*int biosdisk (int command, int disk, int side, int track, int sector, int numsectors, void *buffer);*

Function Performs disk operations by means of BIOS interrupt 13H.

Include File <bios.h>

command (in):	Disk operation to perform: 0 Resets disk system 1 Returns last operation status 2 Reads sector(s) 3 Writes sector(s) 4 Verifies sector(s) 5 Formats track (AT and XT Services)

	6	Formats track set bad sector flags
	7	Formats drive
	8	Returns disk drive parameters
	9	Initializes drive parameters
	0xA	Long read operation
	0xB	Long write operation
	0xC	Disk seek
	0xD	Alternate disk reset
	0xE	Reads sector buffer
	0xF	Writes sector buffer
	0x10	Test drive ready
	0x11	Recalibrates drive
	0x12	Controller diagnostic
	0x13	Drive diagnostic
	0x14	Controller internal diagnostic
disk (in):	Disk drive desired (A = 0, B = 1, C = 2)	
side (in):	Side of disk desired (0 or 1)	
track (in):	Track desired	
sector (in):	Starting sector for operations	
numsectors (in):	Number of sectors to manipulate	
buffer (in/out):	Data buffer for operations	

Example

status = biosdisk (2, 0, 0, 0, 0, 1, bootrecord);

Note If the operation is successful, 0 is returned. Otherwise, biosdisk returns an error status value.

int biosequip (void);

Function Returns a value specifying the current equipment connections.

Include File <bios.h>

Example

equipment = biosequip ();

Note The integer value returned specifies the following:

Bit 1	Math coprocessor present
Bits 2–3	Motherboard RAM (0=16K, 1=32K, 2=48K, 3=64K)
Bits 4–5	Video mode (0=n/a, 1=40×25 color, 2=80×25 color, 3=80×25 bw)
Bits 6–7	Number of diskettes (0=1 drive, 1=2 drives, 2=3 drives, 3=4 drives)
Bit 11	Number of serial ports
Bit 12	Game adapter present
Bits 14–15	Number of printers present

int bioskey (int command);

Function Provides an interface for BIOS keyboard services.

Include File <bios.h>

command (in): Operation to perform:

- 0 returns the next key pressed. If the lower byte is 0, the upper byte contains the scan code for the key pressed
- 1 returns the next key in the buffer if a keystroke is available; otherwise, returns the value 0
- 2 returns the current keyboard status:
  - 0x1 Right SHIFT key pressed
  - 0x2 Left SHIFT key pressed
  - 0x4 CTRL key pressed
  - 0x8 ALT key pressed

0x10	SCROLL LOCK on
0x20	NUM LOCK on
0x40	CAPS LOCK on
0x80	INS on

Example

status = bioskey (2);

int biosmemory (void);

Function Returns the amount of system memory in kilobytes.

Include File <bios.h>

Example

Kbytes = biosmemory ();

int biosprint (inc command, int byte, int port_id);

Function BIOS printer interface routine.

Include File <bios.h>

command (in):	Command to perform: 0 prints the character in bytes 1 initializes the specified printer port 2 returns the printer status
byte (in):	Character to be printed for command 1
port_id (in):	Port number affected (0-LPT1, 1-LPT2)

Example

status = biosprint (1, 65, 0);

Note Valid return status values include the following:

0x01 Device time out	0x08 Output error
0x10 Printer selected	0x20 Printer out of paper
0x40 Ack	0x80 Printer not busy

long biostime (int command, long new_realtime);

Function Sets or returns the current system real-time clock counts.

Include File <bios.h>

command (in):	Command to perform (0 returns current count, 1 sets current count)
new_realtime (in):	Clock ticks since midnight

Example

count = biostime (0, dummy);

Note The real-time clock on the IBM PC and PC compatibles ticks 18.2 times per second. This routine returns the number of clock ticks since midnight.

*int brk (void *end_datasegment);*

Function Modifies the data segment space allocation.

Include File <alloc.h>

end_datasegment (in):	The new desired end of the data segment

Example

```
status = brk (endds);
```

Note The memory location that immediately follows the data segment is called the *break value.* By modifying this value you can resize the application's data segment size.

*void *bsearch (void *key, void *base, int number_of_elements, int width, int (*compare_function)());*

Function Performs a binary search to locate a specific element in an array.

Include File <stdlib.h>

key (in):	The search key of the desired item

base (in):	Pointer to element 0 of the array
number_of_elements (in):	Number of elements in the array
width (in):	Size of each entry in bytes
compare_function (in): Return a value < 0 if a < b Return a value = 0 a = b Return a value > 0 if a > b	Function to perform element comparisons

Example

index = bsearch (ssan, social, 100, 2, comp);

Note If the element is not found, bsearch returns the value 0.

double cabs (struct complex number);

Function Returns the absolute value of a complex number.

Include File <math.h>

number (in):	Complex number desired

Example

result = cabs (complex);

*void *calloc (size_t number_of_elements, size_t element_size);*

Function Allocates a contiguous block of memory and initializes it to zero.

Include File <stdlib.h>, <alloc.h>

number_of_elements (in):	Number of elements to allocate space for
element_size (in):	Size of each element in bytes

Example

```
pointer = (char *) calloc (1, 255);
```

double ceil (double);

Function Rounds the value of a double expression up.

Include File <math.h>

Example

```
max = ceil (value);
```

*char *cgets (char *string);*

Function Reads a character string from the console device.

Include File <conio.h>

string (in/out):	String to be read. Upon input, string[0] should contain the number of characters to read. Upon completion, the string will contain string[1], the number of characters read, or string[2], the first character read

Example

```
cgets (string);
```

Note The cgets routine replaces the newline character with the null character.

*int chdir (char *pathname);*

Function Selects the specified current directory.

Include File <dir.h>

Example

```
status = chdir ("\\TURBOC");
```

Note If successful, chdir returns the value 0. If an error occurs, it returns the value −1.

*int _chmod (char *filename, int function[, int attribute]);*

Function Sets or returns the attributes for a file.

Include File <io.h>

filename (in):	Name of the desired file
function (in):	If 0, the current attribute is returned; if 1, the current attribute is set
attribute (in):	Desired file attribute: FA_RDONLY FA_HIDDEN FA_SYSTEM

Example

```
attribute = _chmod ("ALLOC.H", 0);
```

Note If successful, _chmod returns the value 0. If an error occurs, the routine returns −1.

*int chmod (char *filename, int access);*

Function Modifies the file access restrictions for the specified file.

Include File <io.h>

filename (in):	Name of the desired file	
access (in):	File access desired:	
	S_IWRITE	Write access
	S_IREAD	Read access
	S_IREAD | S_IWRITE	Read/write access

Example

result = chmod ("TEST.C", S_IWRITE);

Note If the routine is successful, chmod returns the value 0. If an error occurs, the routine returns −1.

int chsize (int file_handle, long new_size);

Function Sets the size attribute for a file as specified.

Include File <io.h>

handle (in):	File handle associated with the file whose size is being set
new_size (in):	Desired size of the file in bytes

Example

result = chsize (file_handle, 32000);

Note The file must be opened in either write or read/write mode.

void far circle (int xloc, int yloc, int radius);

Function Draws a circle at the specified x and y location.

Include File <graphics.h>

xloc, yloc (in):	The x and y locations of the center of the circle
radius (in):	Desired size of the circle's radius

Example

```
circle (100, 100, 10);
```

Note The circle routine uses the current drawing color.

unsigned int _clear87 (void);

Function Clears the math coprocessor floating-point status word.

Include File <float.h>

Example

```
status = _clear87();
```

Note The value returned by _clear87 contains the previous status word.

void far cleardevice (void);

Function Clears the screen display in graphics mode.

Include File <graphics.h>

Example

```
cleardevice( );
```

Note The cleardevice routine erases the entire graphics screen and updates the current position to 0,0.

*void clearerr (FILE *file_pointer);*

Function Clears a file's (stream's) error status indicator.

Include File <stdio.h>

file_pointer (in): Data stream desired

Example

```
clearerr (file_pointer);
```

Note This service is closely related to ferror.

void far clearviewport (void);

Function Clears the current viewport in graphics mode.

Example

clearviewport ();

Note The clearviewport routine erases the current viewport and updates the current position to 0,0.

int _close (int file_handle);

Function Closes the file associated with the given file handle.

Include File <io.h>

file_handle (in): File handle of the file to close

Example

status = _close (file_handle);

Note If successful, _close returns the value 0; otherwise, it returns the value −1. Unlike close, _close does not place an end-of-file marker (^Z) at the end of the file.

int close (int file_handle);

Function Closes the file associated with the given file handle.

Include File <io.h>

file_handle (in): File handle of the file to close

Example

status = close (file_handle);

Note If successful, _close returns the value 0; otherwise, it returns the value −1. The _close routine is a text file manipulation routine. Upon invocation, this routine places a ^Z end-of-file marker at the end of the file.

void far closegraph (void);

Function Turns off graphics, returning you to text mode.

Include File <graphics.h>

Note The closegraph routine performs the inverse function of initgraph.

void clreol (void);

Function Clears text from the current cursor position to the end of the current line for the current text window.

Include File <conio.h>

Example

clreol();

Note The clreol routine does not move the current cursor position.

void clrscr();

Function Clears the current text window and places the cursor in the upper-left corner of the window (1,1).

Include File <conio.h>

Example

clrscr();

unsigned coreleft (void);

Function Returns the number of bytes of core memory that are currently unused.

Include File <alloc.h>

Example

bytes = coreleft ();

Note For the compact, large, and huge memory models, use a return type of unsigned long.

double cos (double);

Function Returns the cosine of the specified double expression.

Include File <math.h>

Example

result = double (pi);

double cosh (double);

Function Returns the hyperbolic cosine of the given double expression.

Include File <math.h>

Example

result = double (pi);

*struct country *country_info (int country_code, struct country *country_info);*

Function Returns country-specific information.

Include File <dos.h>

country_code (in):	Country code number of the desired country
country_info (out):	Structure containing country information

Note The DOS.H file defines the country structure.

*int cprintf (char *format_string [, parameter[, . . .]]);*

Function Sends formatted output to the BIOS or video RAM.

Include File <conio.h>

format_string (in):	String specifying the output format
parameter (in):	Data to be output

Example

cprintf ("String %s Number %d \n", str, 10);

Note This routine does not expand newline characters into a carriage return/linefeed. This routine writes its output to the current window.

*void cputs (char *string);*

Function Writes a character string to BIOS or video RAM.

Include File <conio.h>

string (in):	Character string to display

Example

cputs ("This is a test string \n");

Note The cputs routine does not append a newline character. This routine writes its output to the current window.

*int _creat (char *filename, int attribute);*

Function Creates a file with the specified name and attribute.

Include File <dos.h>

filename (in):	Filename to create
attribute (in):	Desired file attribute

Example

result = _creat ("TEST.DAT", 0);

Note If a file with the specified name exists, _creat overwrites it if the write attribute is set. If successful, _creat returns a file handle to the desired file; otherwise, it returns the value −1.

*int creat (char *filename, int access);*

Function Creates a file with the specified name and access.

Include File <sys\stat.h>

filename (in):	Filename to create	
access (in):	File access:	
	S_IWRITE	Write access
	S_IREAD	Read access
	S_IWRITE | S_IREAD	Read/write access

Example

result = creat ("TEST.DAT", S_IWRITE);

Note If a file with the specified name exists, creat overwrites it. If successful, creat returns a file handle to the desired file; otherwise, it returns the value −1.

*int creatnew (char *filename, int attribute);*

Function Creates a file with the specified name and attribute.

Include File <io.h>

filename (in):	Filename to create
attribute (in):	Desired file attribute

Example

```
result = creatnew ("TEST.DAT", 0);
```

Note If a file with the specified name exists, creatnew overwrites it. If successful, creatnew returns a file handle to the desired file; otherwise, it returns the value −1.

*int creattemp (char *filename, int attribute);*

Function Creates a temporary file with the specified path given in filename and attribute.

Include File <io.h>

filename (in/out):	Path in which to create temporary file
attribute (in):	Desired file attribute

Example

result = creattemp ("TEST.DAT", 0);

Note If successful, creattemp returns a file handle to the desired file; otherwise, it returns the value −1.

*int cscanf (char *format_sequence [, arguments]);*

Function Performs formatted input to the console device in a manner similar to scanf.

format_sequence (in):	Specifies the input format desired
arguments (in):	Pointers to the variables to be input

Example

num_fields = cscanf ("%d %d", &value1, &value2);

Note The cscanf routine returns the number of input fields successfully scanned and stored.

*char *ctime (long *seconds_since_01_01_1970);*

Function Returns a string that corresponds to the specified date.

Include File <time.h>

seconds_since_01_01_1970 (in): Number of seconds since 00:00 Jan 1, 1970

Example

```
printf ("Date %s \n", ctime (&seconds));
```

Note The Turbo C routine time returns the number of seconds since 01/01/1970 for the current date.

*void ctrlbrk (int (*function)(void));*

Function Defines a control-break handler.

Include File <dos.h>

function (in): Address of the function DOS will execute each time interrupt 23H occurs. This interrupt occurs whenever the user presses CTRL-C or CTRL-BREAK.

Example

```
int my_handler ();
ctrlbrk (my_handler);
```

Note Upon program termination, DOS resets the interrupt 23H handler to its original value.

void delay (unsigned milliseconds);

Function Temporarily suspends processing for the specified duration.

Include File <dos.h>

milliseconds (in): Number of milliseconds to delay for

Example

```
delay (1000); /* delay 1 second */
```

void delline (void);

Function Deletes the line containing the cursor in the current window, and scrolls all of the lines below the current line up one line.

Include File <conio.h>

Example

delline();

*void far detectgraph (int far *graph_driver, int far *graph_mode);*

Function Returns the graphics driver and mode to be used with the current hardware.

Include File <graphics.h>

graph_driver (out):	Graphics driver to use for current hardware
graph_mode (out):	Graphics mode to use for current hardware

Example

detectgraph (&graph_driver, &graph_mode);

double difftime (time_t time2, time_t time1);

Function Returns the number of seconds by which the two specified times differ.

Include File <time.h>

time2 (in):	Time to subtract time1 from
time1 (in):	Time subtracted from time2 to yield the difference in seconds

Example

```
result = difftime (today, yesterday);
```

void disable (void);

Function Disables hardware interrupts (with the exception of unmaskable interrupts).

Include File <dos.h>

Example

```
disable();
```

Note To reenable interrupts, you must use the enable routine.

div_t div (int numerator, int denominator);

Function Performs integer division, returning both a quotient and a remainder.

Include File <stdlib.h>

numerator (in):	Number to be divided
denominator (in):	Number divided into the numerator

Example

result = div (16, 3);

Note div_t is a structure containing

```
typedef struct {
    int quot;
    int rem;
} div_t;
```

*int dosexterr (struct DOSERR *error_info);*

Function Fills the structure pointed to by error_info with the extended error information for the last failing DOS system service.

Include File <dos.h>

error_info (out):	Structure to contain extended error information

Example

result = dosexterr (&error_info);

Note If dosexterr returns the value 0, the previous DOS system service did not experience an error.

*long dostounix (struct date *date_ptr, struct time *time_ptr);*

Function Converts the DOS date and time format into a UNIX date and time format.

Include File <dos.h>

date_ptr (in):	Structure containing the current DOS date
time_ptr (in):	Structure containing the current DOS time

Example

```
unix_time = dostounix (&date_var, &time_var);
```

*void far drawpoly (int number_of_points, int far *points);*

Function Draws the outline of the polygon contained in the points array.

Include File <graphics.h>

number_of_points (in):	Number of points in the polygon
points (in):	Array containing the x,y coordinates of the polygon

Example

```
drawpoly (4, triangle);
```

Note If an error occurs in drawpoly, graphresult will contain −6.

int dup (int file_handle);

Function Duplicates a DOS file handle.

Include File <io.h>

file_handle (in):	File handle to duplicate

Example

```
new_handle = dup (file_handle);
```

Note If dup is successful, it returns a positive file handle. Otherwise, dup returns a negative value.

int dup2 (int old_file_handle, int new_file_handle);

Function Duplicates a DOS file handle.

Include File <io.h>

old_file_handle (in):	File handle to duplicate
new_file_handle (in):	New copy of file handle

Example

dup (oldfile, filehandle);

Note This routine is provided for compatibility with UNIX.

*char *ecvt (double value, int number_of_digits, int *decimal_loc, int *sign);*

Function Converts a floating-point number into a character string.

Include File <stdlib.h>

value (in):	Floating-point value to convert
number_of_digits (in):	Number of digits in the string representation of the value

decimal_loc (out):	Location of the decimal point
sign (out):	Negative value if the value is nonzero; 0 otherwise

Example

str = ecvt (664.333, 7, &loc, &sign);

Note This routine does not place the decimal into the string or place a negative sign at the front of the string.

void far ellipse (int x_loc, int y_loc, int start_angle, int end_angle, int x_radius, int y_radius);

Function Draws an ellipse, at the given location, with the aspect ratio specified by x_radius and y_radius.

Include File <graphics.h>

x_loc, y_loc (in):	The x and y locations of the center of the ellipse
start_angle (in):	Starting angle for the ellipse
end_angle (in):	End angle for the ellipse
x_radius (in):	Radius of the ellipse along the x axis
y_radius (in):	Radius of the ellipse along the y axis

Example

ellipse (100, 100, 0, 360, 10, 5);

Note The ellipse routine uses the current drawing color.

void enable (void);

Function Enables interrupts previously disabled by the run-time library disable routine.

Include File <dos.h>

Example

```
enable( );
```

Note The disable routine can only disable unmaskable interrupts.

int eof (int file_handle);

Function Returns true if the file associated with the given file handle has reached end of file; otherwise, eof returns the value 0.

Include File <io.h>

Example

```
while (! eof (file_handle))
```

Note If eof experiences an error, it returns the value −1.

*int exec . . . (char *path, char *arg0, char *arg1 . . . , NULL);*

Function Spawns a DOS command as a child process.

Include File <process.h>

path (in):	Name of the command to spawn
arg0-arg*n*:	Command-line parameters for the spawned program

Example

status = execl ("TEST.EXE", "TEST.EXE", "A1", NULL);

Note The execl routine is similar to exec but only searches the root or current directory for the child. If you add the suffix "p," it will search for the child program in the directories contained in the DOS path. If you add the suffix "l," you pass the command-line parameters as individual values. If you add "v," you are passing the command-line parameters as an array of pointers. Lastly, the "e" suffix allows you to pass an environment to the child process. If no environment is specified, the child inherits the current environment.

void _exit (int status);

Function Terminates the current program, returning the specified status value.

Include File <process.h>

status (in):	Status value to return to the parent process or DOS

Example

```
_exit (0);
```

Note The _exit routine does not close open files.

void exit (int status);

Function Terminates the current program, returning the specified status value.

Include File <process.h>

status (in):	Status value to return to the parent process or DOS

Example

```
exit (0);
```

Note The exit routine flushes file buffers and appropriately updates files.

double exp (double value);

Function Returns the exponential of the specified value.

Include File <math.h>

value (in): Value to return the exponential of

Example

```
result = exp (x);
```

double fabs (double value);

Function Returns the absolute value of a double-precision expression.

Include File <math.h>

value (in): Value to return the absolute value of

Example

```
result = fabs (−44.3332);
```

*void far *farcalloc (unsigned long number_of_entries, unsigned long entry_size);*

Function Allocates memory from the far heap and clears it.

Include File <alloc.h>

number_of_entries (in):	Number of elements to allocate space for
entry_size (in):	Number of bytes in each element

Example

chunk = faralloc (1, 65000);

Note If faralloc cannot allocate the specified space, it returns the value NULL.

long farcoreleft (void);

Function Returns the number of bytes available in the far heap.

Include File <alloc.h>

Example

long_var = farcoreleft ();

*void farfree (void far *ptr);*

Function Returns previously allocated memory to the far heap.

Include File <alloc.h>

Example

farfree (chunk);

*void far *farmalloc (unsigned long number_of_bytes);*

Function Allocates space from the far heap.

Include File <alloc.h>

number_of_bytes (in):	Number of bytes of memory to allocate

Example

buffer = farmalloc (65000);

Note If farmalloc cannot satisfy the request, it returns the value NULL; otherwise, it returns a pointer to the desired memory.

*void far *farrealloc (void far *ptr, unsigned long num_bytes);*

Function Reallocates memory for a previously allocated segment of memory from the far heap.

Include File <alloc.h>

ptr (in):	Pointer to the previously allocated memory
num_bytes (in):	Number of bytes of memory desired

Example

block = farrealloc (block, 65001);

*int fclose (FILE *stream);*

Function Flushes all of the buffers associated with a file and updates the file on disk.

Include File <stdio.h>

Example

status = fclose (file);

Note If successful, fclose returns the value 0.

int fcloseall (void);

Function Flushes all of the buffers associated with open files and updates each file on disk as it closes it.

Include File <stdio.h>

Example

status = fcloseall ();

Note This routine returns the number of file streams closed.

*char *fcvt (double value, int number_of_digits, int *decimal_loc, int *sign);*

Function Converts a floating-point number into a character string.

Include File <stdlib.h>

value (in):	Floating-point value to convert
number_of_digits (in):	Number of digits in the string representation of the value
decimal_loc (out):	Location of the decimal point
sign (out):	Negative value if the value is negative; 0 otherwise

Example

str = ecvt (664.333, 7, &loc, &sign);

Note This routine does not place the decimal into the string or place a negative sign at the front of the string. This routine differs from ecvt in that it rounds to FORTRAN F format the number of digits specified.

*FILE *fdopen (int handle, char *open_type);*

Function Associates a stream with the file handle returned by creat, dup, dup2, or open.

Include File <stdio.h>

handle (in):	File handle to associate with the stream				
open_type (in):	Specifies how the file can be accessed:				
	r	Read-only	w	Write access	
	a	Append	r+	Read/write	
	w+	Create new	a+	Create for append if no file	

Example

```
file_pointer = fdopen (handle, "r");
```

Note If an error occurs, fdopen returns the value NULL.

*int feof (FILE *stream);*

Function Returns true (1) if the specified file has reached end of file; otherwise, returns 0.

Include File <stdio.h>

stream (in):	File stream to examine for end of file

Example

```
while (! feof (in_file))
```

*int ferror (FILE *stream);*

Function Returns true (1) if the specified file experienced a read or write error; otherwise, returns 0.

Include File <stdio.h>

stream (in): File stream to examine for error

Example

status = ferror (in_file);

*int fflush (FILE *stream);*

Function Flushes the file buffers for the specified file.

Include File <stdio.h>

Example

status = fflush (in_file);

Note If the file is an input file, fflush flushes the input stream.

*int fgetc (FILE *stream);*

Function Reads the next character from the specified file stream.

Include File <stdio.h>

Example

ltr = fgetc(in_file);

Note If fgetc obtains EOF or an error occurs, it will return the value EOF.

int fgetchar (void);

Function Reads the next character from stdin.

Include File <stdio.h>

Example

ltr = fgetchar;

Note If fgetchar obtains EOF or an error occurs, it will return the value EOF. Unlike getchar, fgetchar is a function.

*int fgetpos (FILE *file_stream, fpos_t *file_position);*

Function Stores the current file position of the file associated with file_stream into the location pointed to by file_position.

Include File <stdio.h>

file_stream (in):	File pointer associated with the file of interest
file_position (out):	Location at which current file position is stored

Example

status = fgetpos (fp, &file_position);

Note If successful, fgetpos returns 0; otherwise, it returns a nonzero value.

*char *fgets (char *string, int num_bytes, FILE *stream);*

Function Reads a character string from an input stream.

Include File <stdio.h>

string (out):	Character string of data read
num_bytes (in):	Maximum number of characters in the string
stream (in):	File stream to read the characters from

Example

status = fgets (str, 255, fp);

Note Upon end of file, fgets returns NULL. It leaves the newline character at the end of each string.

long filelength (int file_handle);

Function Returns the number of bytes in the file associated with the given file handle.

Include File <io.h>

file_handle (in):	File handle returned from open or creat

Example

long_var = filelength (handle);

Note If filelength encounters an error, it returns the value −1L.

*int fileno (FILE *stream);*

Function Obtains a file handle for a given file stream.

Include File <stdio.h>

stream (in):	File stream to return a file handle to

Example

file_handle = fileno (fp);

*void far fillpoly (int number_of_points, int far *points);*

Function Draws the outline for, and fills in, the polygon specified in the points array.

Include File <graphics.h>

number_of_points (in):	Number of points in the polygon
points (in):	Array of points specifying the polygon's shape

Example

fillpoly (10, points);

Note The fillpoly routine uses the current drawing and fill colors and fill pattern.

*int findfirst (char *path, struct ffblk *fileblock, int attribute);*

Function Searches a directory of files for a file that matches the given description (filename or wildcard characters).

Include File <dir.h>

path (in):	DOS pathname (including optional drive) of the path to examine in search of the files

fileblock (out):	File block structure containing
	struct ffblk {
	char ff_reserved[21];
	char ff_attrib;
	int ff_ftime;
	int ff_fdate;
	long ff_fsize;
	char ff_name[13];
	};

attribute (in):	File attribute to be used in matching files

Example

```
status = findfirst ("*.*", &file_block, 0);
```

Note Use findfirst to locate the first matching file and the findnext routine to locate subsequent files. If findfirst is successful, it returns the value 0.

*int findnext (struct ffblk *fileblock);*

Function Searches a directory of files for a file that matches the description (filename or wildcard characters) given in a call to findfirst.

Include File <dir.h>

fileblock (out):	File block structure as defined in findfirst

Example

status = findnext (&file_block);

Note Use findfirst to locate the first matching file and the findnext routine to locate subsequent files. If successful, the findfirst routine returns the value 0.

void far floodfill (int x_loc, int y_loc, int border_color);

Function Fills a region bounded by the color specified in border_color with the current fill pattern and color.

Include File <graphics.h>

x_loc (in):	The x point that resides within the region to fill
y_loc (in):	The y point that resides within the region to fill
border_color:	Color surrounding the region to fill

Example

floodfill (100, 100, red);

Note If floodfill encounters an error, graphresult will contain −7.

double floor (double value);

Function Rounds a double-precision value down to the largest integer that is not greater than the value.

Include File <math.h>

value (in):	Double-precision value to round

Example

```
approx_cost = floor (purchase_price * tax);
```

int flushall (void);

Function Flushes all of the disk buffers for open file streams.

Include File <stdio.h>

Note The flushall routine returns 0 upon success.

double fmod (double x, double y);

Function Returns the remainder of the division of two double-precision values.

Include File <math.h>

x (in):	Value y is divided into
y (in):	The divisor

Example

```
result = fmod (total_sales, employees);
```

Note See also the modf routine.

*void fnmerge (char *path, char *drive, char *directory, char *filename, char *extension);*

Function Builds a complete DOS pathname from all of the component parts.

Include File <dir.h>

path (out):	Complete DOS pathname
drive (in):	Desired drive letter, including colon
directory (in):	DOS directory path desired
filename (in):	8-character DOS filename
extension (in):	3-character DOS file extension

Example

```
fnmerge (path, "A:", "\\TURBOC\\", "TEST", "C");
```

Note See also the fnsplit routine.

void fnsplit (char *path, char *drive, char *directory, char *filename, char *extension);

Function Breaks down a complete DOS pathname into all of its component parts.

Include File <dir.h>

path (out):	Complete DOS pathname
drive (in):	Desired drive letter, including colon
directory (in):	DOS directory path
filename (in):	8-character DOS filename
extension (in):	3-character DOS file extension

Example

```
fnsplit ("A:\\TBO\\FILENAME.C", drive, path, file, ext);
```

Note See also the fnmerge routine.

FILE *fopen (char *filename, char *access_type);

Function Opens a DOS file stream.

Include File <stdio.h>

filename (in):	Name of the file to open
access_type (in):	Defines how the file will be accessed: r Read-only w Write a Append r+ Read/write w+ Create write a+ Append create if new file

Example

```
fp = fopen (argv[1], "r");
```

Note If unsuccessful, fopen returns NULL. If you need to open a file in binary mode, simply attach a b to the access type, as in rb or wb. For text mode, you can append the letter t.

*unsigned FP_OFF(void far *far_pointer);*

Function Returns the offset portion of a far pointer.

Include File <dos.h>

far_pointer (in):	Far pointer to return the offset portion of

Example

```
offset = FP_OFF (far_pointer);
```

void _fpreset (void);

Function Reinitializes the floating-point math library.

Include File <float.h>

Note Early DOS versions (before version 2.x) allowed child processes to leave the 8087 in an inconsistent state. This routine resets the math coprocessor to a known state.

*int fprintf (FILE *stream, char *format_sequence [, arguments . . .]);*

Function Performs formatted output to a file stream.

Include File <stdio.h>

stream (in):	File stream to be written to
format_sequence (in):	Control sequence specifying the output format
arguments (in):	Data to be written to the file

Example

```
num_bytes = fprintf (fp, "%d %f \n", days, salary);
```

Note The fprintf routine returns the number of bytes written to the data stream.

*unsigned FP_SEG(void far *far_pointer);*

Function Returns the segment portion of a far pointer.

Include File <dos.h>

far_pointer (in):	Far pointer to return the segment portion of

Example

```
segment = FP_SEG (far_pointer);
```

*int fputc (int character, FILE *stream);*

Function Outputs a single character to a file stream.

Include File <stdio.h>

character (in):	Character to be written to the file stream
stream (in):	File stream to be written to

Example

```
result = fputc ('a', fp);
```

Note If successful, fputc returns the character written. If an error occurs, fputc returns EOF.

*int fputs (char *str, FILE *stream);*

Function Writes a character string to a file stream.

str (in):	Character string to be written to the data stream
stream (in):	File string to be written to

Example

last_char = fputs ("This is a test\n", fp);

Note If successful, fputs returns the last character written. If an error occurs, fputs returns EOF.

*int fread (void *pointer, int num_bytes, int num_items, FILE *stream);*

Function Reads the specified number of bytes from a data stream.

Include File <stdio.h>

pointer (in):	Pointer to the data buffer
num_bytes (in):	Number of bytes in each entry
num_items (in):	Number of items of num_bytes length to read

Example

num_items_read = fread(buffer, 255, 5, fp);

Note If successful, fread returns the number of items read. If an error occurs, fread returns an invalid count.

*void free (void *pointer);*

Function Releases a section of previously allocated memory.

Include File <stdlib.h>

pointer (in):	Pointer to the previously allocated memory

Example

free (list_node);

int freemem (unsigned segment);

Function Frees a previously allocated DOS segment.

Include File <dos.h>

segment (in):	Segment address of the memory block to release

Example

result = freemem (segment_addr);

Note If successful, freemem returns 0. If an error occurs, it returns −1.

*FILE *freopen (char *filename, char *access_type, FILE *stream);*

Function Substitutes a named file in place of a file stream.

Include File <stdio.h>

filename (in):	Name of the file to open
access_type (in):	Specifies how the file is to be opened: r Read-only w Write a Append r+ Read/write w+ Create write a+ Append create if new file
stream (in):	File pointer to be associated with the file

Example

fp = freopen ("TEMP.DAT", "w", stdout);

Note If successful, freopen returns the value of the file pointer. If an error occurs, it returns the value NULL.

*double frexp (double value, int *exponent);*

Function Splits a double-precision value into an exponent and mantissa.

Include File <math.h>

value (in):	Value to be split
exponent (out):	Exponent of the value

Example

mantissa = frexp (value, &exponent);

Note The value returned by frexp is the mantissa.

*int fscanf (FILE *stream, char format_sequence [, argument . . .]);*

Function Writes formatted output to a file stream.

Include File <stdio.h>

stream (in):	File stream to write to
format_sequence (in):	Control sequence specifying the output format
arguments (in):	Data to be written to the file

Example

num_fields = fscanf (fp, "%d %d %fn", &a, &b, &c);

Note The fscanf routine returns the number of fields filled successfully.

*int fseek (FILE *stream, long offset, int location);*

Function Moves the file pointer in a file stream.

Include File <stdio.h>

stream (in):	File stream desired	
offset (in):	Desired byte offset in the file	
location (in):	Location to offset from:	
	SEEK_SET (0)	Start of file
	SEEK_CUR (1)	Current file position
	SEEK_END (2)	End of file

Example

```
status = fseek (fp, 128, SEEK_SET);
```

Note If successful, fseek returns the value 0. If an error occurs, it returns a nonzero value.

*int fsetpos (FILE *file_stream, const fpos_t *file_position);*

Function Sets the current file position for the specified file to the value last saved by fgetpos.

Include File <stdio.h>

file_stream (in):	File pointer associated with the desired file
file_position (in):	File position to be selected for the file

Example

```
result = fsetpos (fp, &file_position);
```

Note If successful, fsetpos returns 0; otherwise, it returns a nonzero value.

*int fstat (char *handle, struct stat *stat_info);*

Function Returns information about the file associated with a file handle.

Include File <stat.h>

handle (in):	File handle associated with the desired file
stat_info (out):	Structure containing the file information

Example

```
result = fstat (file_handle, &stat_info);
```

Note If successful, fstat returns the value 0; otherwise, it returns the value −1.

*long ftell (FILE *stream);*

Function Returns the current file-pointer location.

Include File <stdio.h>

stream (in):	Data file stream desired

Example

```
loc = ftell (fp);
```

*int fwrite (void *buffer, int num_bytes, int num_items, FILE *stream);*

Function Writes the specified number of bytes to a data stream.

Include File <stdio.h>

pointer (in):	Pointer to the data buffer
num_bytes (in):	Number of bytes in each entry
num_items (in):	Number of items of num_bytes length to read

Example

```
num_items_written = fwrite(buffer, 255, 5, fp);
```

Note If successful, fwrite returns the number of items written. If an error occurs, it returns an invalid count.

*char *gcvt (double value, int num_digits, char *str);*

Function Converts a double-precision value into its character string representation.

Include File <stdlib.h>

value (in):	Double-precision value
num_digits (in):	Number of digits in the string
str (out):	ASCII representation of the floating-point value

Example

status = gcvt (334.33, 10, str);

Note gcvt returns a value of type string.

void geninterrupt (int interrupt_number);

Function Generates the desired software interrupt.

Include File <dos.h>

interrupt_number (in):	Desired software interrupt number

Example

```
geninterrupt (0x21);
```

*void far getarccoords (struct arccoordstype far *arc_coord);*

Function Returns the coordinates of the last call to arc.

Include File <graphics.h>

arc_coord (out):	Structure containing the arc coordinates

Example

```
getarccoords (&arc_coords);
```

Note The structure type is

```
struct arccoordstype {
   int x, y;
   int xstart, ystart, xend, yend;
};
```

*void far getaspectratio (int far *x_aspect, int far *y_aspect);*

Function Returns the aspect ratio for the current graphics mode.

Include File <graphics.h>

x_aspect (out):	The aspect ratio for the current graphics mode
y_aspect (out):	The aspect ratio (normalized to 10000) for the current mode

Example

```
getaspectratio (&x_aspect, &y_aspect);
```

int far getbkcolor (void);

Function Returns the current graphics mode background color.

Include File <graphics.h>

Example

```
background_color = getbkcolor ();
```

Note The setbkcolor routine sets the current background color.

*int getc(FILE *stream);*

Function Gets the next character in a file stream.

Include File <stdio.h>

stream (in): File stream to read a character from

Note If getc encounters an end of file, it returns EOF.

int getcbrk (void);

Function Returns the current state of control-break checking, on (1) or off (0).

Include File <dos.h>

Example

```
state = getcbrk();
```

Note The setcbrk routine enables or disables control-break checking.

int getch(void);

Function Gets a character from the console device without echoing that character.

Include File <conio.h>

Example

letter = getch();

Note If getch encounters an end of file or an error, it returns the value EOF.

int getchar(void);

Function Gets the next character from the stdin file stream.

Include File <stdio.h>

Example

letter = getchar();

Note If getchar encounters an end of file or an error, it returns the value EOF.

int getche(void);

Function Gets a character from the console device echoing that character.

Include File <conio.h>

Example

letter = getche( );

Note If getche encounters an end of file or an error, it returns the value EOF. The routine echoes the character to the current window.

int far getcolor (void);

Function Returns the current graphics color.

Include File <graphics.h>

Example

foreground_color = getcolor ( );

Note The setcolor routine specifies the current color.

*int getcurdir (int drive, char *directory);*

Function Returns the current directory for the specified disk drive.

Include File <dir.h>

drive (in):	Disk drive ID desired (0 = current, 1 = A, 2 = B)
directory (out):	DOS pathname of the directory

Example

status = getcurdir (1, directory);

Note If getcurdir encounters an error, the value −1 is returned.

*char *getcwd (char *directory, int num_bytes);*

Function Returns the current working directory.

Include File <dir.h>

directory (in):	Buffer containing the current directory
num_bytes (in):	Number of bytes malloc should allocate to store the current directory. DOS directory names do not exceed 64 characters

Example

status = getcwd (directory, 64);

Note If getcwd encounters an error, it returns −1.

*void getdate (struct date *current_date);*

Function Returns the current DOS system date.

Include File <dos.h>

current_date (out): Structure containing the current system date:

```
struct date {
  int da_year;
  char da_day;
  char da_mon;
} ;
```

Example

```
getdate (&current_date);
```

*void getdfree (int drive, struct dfree *disk_info);*

Function Returns the amount of free space on the specified drive.

Include File <dos.h>

drive (in): Disk drive desired (0 = A, 1 = B, 2 = C)

disk_info (out): Structure containing the disk space information:

```
struct dfree {
  unsigned df_avail; /* clusters available */
```

```
        unsigned df_total; /* total clusters */
        unsigned df_bsec; /* bytes per
        sector */

        unsigned df_sclus; /* sectors per
        cluster */
};
```

Example

status = getdfree (0, &disk_info);

Note If getdfree encounters an error, it returns −1.

int getdisk (void);

Function Returns the current disk drive.

Include File <dir.h>

Example

drive = getdisk( );

Note Disk drives are identified as 0 = A, 1 = B, 2 = C.

*char *far getdta (void);*

Function Returns the address of the disk transfer.

Include File <dos.h>

Example

far_address = getdta();

Note By default, DOS places the disk transfer's address at offset 0x80 of the program segment prefix.

*char *getenv(char *environment_variable);*

Function Returns the value assigned to an environment variable.

Include File <stdlib.h>

environment_variable (in):	Environment variable to return the value of

Example

str = getenv ("INCLUDE");

Note If getenv cannot find a matching entry, it returns a NULL string.

*void getfat (int drive, struct fatinfo *fat_info);*

Function Returns file allocation table information for the specified disk drive.

Include File <dos.h>

drive (in): Disk drive desired (0 = current, 1 = A, 2 = B, 3 = C)

fat_info (in): Structure containing the FAT information:

```
struct fatinfo {
  char fi_sclus; /* sectors/cluster */
  char fi_fatid; /* fat ID byte */
  int fi_nclus; /* number of clusters */
  int fi_bysec; /* bytes/sector */
};
```

Example

```
getfat (1, &fat_info);
```

*void getfatd (struct fatinfo *fat_info);*

Function Returns file allocation table information for the default disk drive.

Include File <dos.h>

fat_info (in): Structure containing the FAT information:

```
struct fatinfo {
  char fi_sclus;   /* sectors/cluster */
  char fi_fatid;   /* fat ID byte */
  int fi_nclus;   /* number of clusters */
  int fi_bysec;   /* bytes/sector */
};
```

Example

```
getfatd (&fat_info);
```

*void far getfillpattern (char far *fill_pattern);*

Function Copies a user-defined fill pattern into memory for fill operations in graphics mode.

Include File <graphics.h>

fill_pattern (in): An 8-byte array, where each byte represents 8 pixels; thus, an 8 × 8 pattern can be specified

Example

```
get_fillpattern (my_pattern);
```

*void far getfillsettings (struct fillsettingstype far *fill_info);*

Function Returns the current graphics mode fill pattern and color.

Include File <graphics.h>

fill_info (out): Structure containing the current fill pattern and color:

```
struct fillsettingstype {
  int pattern;
  int color;
} ;
```

Example

getfillsettings (&fill_info);

Note If the pattern returned is 12, a user-defined pattern is in effect. Predefined patterns include

0	Empty fill
1	Solid fill
2	Line fill -
3	Left-slash fill /
4	Thick left-slash fill
5	Backslash fill \
6	Thick backslash fill
7	Light hatch fill
8	Heavy crosshatch fill
9	Interleaving line fill
10	Wide-spaced dot fill
11	Close-spaced dot fill

*int getftime (int file_handle, struct ftime *file_stamp);*

Function Gets a file's date and time stamp.

Include File <dos.h>

file_handle (in): File handle associated with the desired file

file_stamp (out): Structure containing the file's date and time:

```
struct ftime {
  unsigned ft_tsec:5;
  unsigned ft_min:6;
  unsigned ft_hour:5;
  unsigned ft_day: 5;
  unsigned ft_month:4;
  unsigned ft_year: 7;
};
```

Example

status = getftime (filehandle, &file_stamp);

Note If successful, getftime returns the value 0.

int far getgraphmode ();

Function Returns the current graphics mode.

Include File <graphics.h>

Example

save_mode = getgraphmode ();

Note See graphics.h for definitions of graphics modes.

*void far getimage (int left_corner, int top_corner, int right_corner, int bottom_corner, void far *image);*

Function Saves a bit image from the specified screen coordinates.

Include File <graphics.h>

left_corner (in):	Leftmost corner of the image to save
top_corner (in):	Topmost corner of the image to save
right_corner (in):	Rightmost corner of the image to save
bottom_corner (in):	Bottommost corner of the image to save
image (out):	Buffer containing the bit image

Example

getimage (10, 20, 20, 30, buffer);

Note The image size cannot exceed 64K.

int far getmaxcolor (void);

Function Returns the number associated with the last color value you can specify in graphics mode.

Include File <graphics.h>

Example

max_color = getmaxcolor ();

int far getmaxx (void);

Function Returns the maximum x screen coordinate.

Include File <graphics.h>

Example

```
max_x = getmaxx ();
```

int far getmaxy (void);

Function Returns the maximum y screen coordinate.

Include File <graphics.h>

Example

```
max_y = getmaxy ();
```

*void far getmoderange (int graph_driver, int far *lowest_mode, int far *highest_mode);*

Function Returns the lowest and highest graphics mode values that you can specify for the given graph driver.

Include File <graphics.h>

Example

getmoderange (graph_driver, &low, &high);

Note If the graphics driver specified is invalid, both low and high are set to −1.

*void far getpalette (struct palettetype far *palette);*

Function Returns information about the current available colors.

Include File <graphics.h>

palette (out): Structure containing palette information:

```
struct palettetype {
  unsigned char size;
  signed char colors [MAX_COLORS +1];
};
```

Example

getpalette (&palette);

Note See graphics.h for color definitions.

*char *getpass (char *prompt);*

Function Prompts the user to enter a password and returns the password entered.

Include File <conio.h>

prompt (in): Character string prompt

Example

```
pass = getpass ("Enter your secret password");
```

Note The password can contain up to eight characters.

int far getpixel (int x_loc, int y_loc);

Function Returns the color of the pixel at the specified x,y location.

Include File <graphics.h>

x_loc, y_loc (in): The x and y location of the desired pixel

Example

```
color = getpixel (10, 20);
```

unsigned getpsp (void);

Function Returns the program segment prefix address.

Include File <dos.h>

Example

```
segment_addr = getpsp ();
```

Note getpsp only works under DOS 3.x.

*char *gets (char *string)*

Function Returns a character string from the stdin stream.

Include File <stdio.h>

string (out): Character string read

Example

```
status = gets (str);
```

Note If gets encounters an error or end of file, it returns EOF. It replaces a newline character with the NULL character.

*int gettext (int left_corner, int top_corner, int right_corner, int bottom_corner, void *buffer);*

Function Copies text from your screen display into a buffer.

left_corner (in):	Specifies the x coordinate of the upper-left corner of the region to copy
top_corner (in):	Specifies the y coordinate of the upper-left corner of the region to copy
bottom_corner (in):	Specifies the y coordinate of the lower-right corner of the region to copy
right_corner (in):	Specifies the x coordinate of the lower-right corner of the region to copy
buffer (out):	Buffer in memory that the text is copied to

Example

```
gettext (0, 10, 20, 79, buffer);
```

Note All coordinates are screen coordinates, as opposed to window coordinates. Calculate your buffer size as

```
size = (rows) * (columns) * 2
```

*gettextinfo (struct text_info *text_record);*

Function Returns specifics about text mode.

Include File <conio.h>

text_record (out): Structure containing text mode information:

```
struct text_info {
  unsigned char winleft;
  unsigned char wintop;
  unsigned char winright;
  unsigned char winbottom;
  unsigned char attribute;
  unsigned char normattr;
  unsigned char currmode;
  unsigned char screenheight;
  unsigned char screenwidth;
  unsigned char curx;
  unsigned char cury;
};
```

Example

```
gettextinfo (&text_record);
```

void far gettextsettings (struct textsettingstype far text_record);

Function Returns information about graphics mode text settings.

Include File <graphics.h>

text_record (out): Structure containing graphics text information:

```
struct textsettingstype {
  int font;
  int direction;
  int charsize;
  int horiz;
  int vert;
};
```

Example

```
gettextsettings (&text_record);
```

*void gettime (struct time *system_time);*

Function Returns the current system time.

Include File <dos.h>

system_time (out): Structure assigned the current system time:

```
struct time {
  unsigned char ti_min;
  unsigned char ti_hour;
  unsigned char ti_hund;
  unsigned char ti_sec;
};
```

Example

```
gettime (&current_time);
```

*unsigned interrupt (*getvect(int interrupt_number))();*

Function Returns the interrupt vector address for the specified interrupt.

Include File <dos.h>

interrupt_ number (in): Interrupt number to return the vector for

Example

vector_address = getvect (5);

*getviewsettings (struct viewporttype far *view_port);*

Function Returns specifics about the current viewport.

Include File <graphics.h>

viewport (out): Structure containing specifics about the current viewport:

```
struct viewporttype {
  int left, top, right, bottom;
  int clipflag;
};
```

Example

getviewsettings (&view_port);

int getverify (void);

Function Returns the current state of disk verification.

Include File <dos.h>

Example

status = getverify ();

Note If disk verification is on, getverify returns the value 1. If disk verification is off, getverify returns 0.

*int getw (FILE *stream);*

Function Gets an integer value from a data stream.

Include File <stdio.h>

stream (in): Data file stream

Example

value = getw (fp);

int far getx (void);

Function Returns the current position's x coordinate in graphics mode.

Include File <graphics.h>

Example

```
xloc = getx();
```

Note Coordinates are viewport relative.

int far gety (void);

Function Returns the current position's y coordinate in graphics mode.

Include File <graphics.h>

Example

```
yloc = gety();
```

Note Coordinates are viewport relative.

*struct tm *gmtime(long *clock);*

Function Converts a date and time to Greenwich mean time.

Include File <time.h>

clock (in):	Structure containing the time to convert

Example

```
gmt_time = gmtime (&current_time);
```

void gotoxy (int x_loc, int y_loc);

Function Sets the cursor position (column,row) in text mode.

Include File <conio.h>

x_loc (in):	Desired column for the cursor
y_loc (in):	Desired row for the cursor

Example

```
gotoxy (10, 10);
```

Note The cursor is positioned within the current window.

*char far *grapherrormsg (int error_code);*

Function Returns an error message string for the specified graphresult.

Include File <graphics.h>

error_code (in):	Error code value contained in graphresult

Example

```
msg = grapherrormsg (-8);
```

*void far _graphfreemem (void far *pointer, unsigned bytes);*

Function Releases memory allocated for graphics by _graphgetmem.

Include File <graphics.h>

pointer (in):	Pointer to the allocated memory
bytes (in):	Number of bytes to release

Example

```
_graphfreemem (buffer, 1024);
```

*void far *far _graphgetmem (unsigned size);*

Function Allocates memory for graphics manipulation.

Include File <graphics.h>

size (in): Number of bytes of memory to allocate

Example

```
buffer = _graphgetmem (1024);
```

int far graphresult (void);

Function Returns the error code for the last unsuccessful graphics operation.

Include File <graphics.h>

Example

```
status = graphresult ();
```

Note Once you invoke graphresult, Turbo C resets its value to 0. Common error status codes include

0	No error
−1	Graphics not installed; use initgraph
−2	Graphics hardware not found

−3	Device driver not found
−4	Invalid device driver file
−5	Insufficient memory to load driver
−6	Out of memory in scan fill
−7	Out of memory in flood fill
−8	Font file not found
−9	Insufficient memory to load font
−10	Invalid graphics mode for driver selected
−11	Graphics error
−12	Graphics I/O error
−13	Invalid font file
−14	Invalid font number
−15	Invalid device number

int gsignal (int signal);

Function Raises the specified signal, and executes the action routine.

Include File <signal.h>

signal (in): Software signal, ranging from 1 to 15

Example

result = gsignal (2);

Note The gsignal routine returns the value returned by the action defined or the constant SIG_DFL if the signal is invalid.

*void harderr (int (*function_ptr) ());*

Function Defines a hardware error handler.

Include File <dos.h>

function_ptr (in):	Address of the function to serve as the hard error handler

Example

harderr (my_handler);

Note Hard errors occur when interrupt 0x24 is invoked. The most common occurrence of this is an open disk drive. You can define your own error-handling routine. When interrupt 0x24 occurs, your routine will receive these parameters:

(int error_value, int ax, int bp, int si);

See the Turbo C manual for more specifics on the information contained in these parameters.

void hardresume (int resume_code);

Function Returns 2 (abort), 1 (retry), or 0 (ignore) based upon a hard error-handling routine to DOS.

Include File <dos.h>

resume_code (in):	Return status value (2=abort, 1=retry, 0=ignore)

Example

hardresume (0);

void hardretn (int error_code);

Function Returns an error status code to the application based upon a hard error handler.

Include File <dos.h>

error_code (in):	Value returned to the application program

Example

hardretn (0);

void highvideo (void);

Function Selects high-intensity attributes for text display.

Include File <conio.h>

Example

highvideo ();

Note This routine allows you to make some text appear in a heavier boldface on your screen display.

double hypot (double x, double y);

Function Returns the hypotenuse of a right triangle.

Include File <math.h>

x (in): x side of the triangle
y (in): y side of the triangle

Example

z = hypot (x, y);

unsigned far imagesize (int left_corner, int top_corner, int right_corner, int bottom_corner);

Function Returns the number of bytes required to store the specified graphics image.

Include File <graphics.h>

left_corner (in):	Specifies the x coordinate of the upper-left corner of the region to copy
top_corner (in):	Specifies the y coordinate of the upper-left corner of the region to copy
bottom_corner (in):	Specifies the y coordinate of the lower-right corner of the region to copy
right_corner (in):	Specifies the x coordinate of the lower-right corner of the region to copy

Example

```
num_bytes = imagesize (10, 10, 20, 20);
```

*void far initgraph (int far *graph_driver, int far *graph_mode, char far *driver_path);*

Function Initializes graphics by loading a graphics driver from disk, validating the driver, and placing the system into graphics mode.

Include File <graphics.h>

graph_driver (in):	Graphics driver for system
graph_mode (in):	Desired graphics mode
driver_path (in):	DOS subdirectory that contains graphics device driver files

Example

initgraph (&graphics_driver, &graphics_mode, "");

Note If driver_path is NULL, BGI files must be in the current directory. Use the following values for graphics drivers:

0	DETECT Autodetect correct driver for hardware
1	CGA monitor
2	MCGA monitor
3	EGA monitor
4	EGA64 monitor
5	EGAMONO monitor
6	RESERVED
7	HERCMONO Hercules monitor
8	ATT400 monitor
9	VGA monitor
10	PC3270 monitor

See the file graphics.h for graphics modes.

int inport (int port_number);

Function Inputs a word from the specified hardware port.

Include File <dos.h>

port_number (in): Desired hardware port number

Example

status = inport (0x3da);

int inportb (int port_number);

Function Inputs a byte from the specified hardware port.

Include File <dos.h>

port_number (in): Desired hardware port number

Example

```
status = inportb (0x3da);
```

void insline (void);

Function Inserts a blank line at the current cursor position in the current text window.

Include File <conio.h>

Example

```
insline();
```

Note All lines (including the current line and below) are moved down one line.

*int int86x (int interrupt_number, union REGS *inregs, union REGS *outregs);*

Function Invoke the specified 8086 interrupt and assign the 8086 registers the values contained in the structure inregs.

Include File <dos.h>

interrupt_number (in):	Desired 8086 interrupt
inregs (in):	Structure containing the values to assign to the 8086 registers (see Chapter 6)
outregs (out):	Structure containing the values contained in the 8086 registers following the interrupt service

Example

```
status = int86 (0x10, inregs, outregs);
```

Note int86 returns the value of the AX register upon completion of the interrupt routine.

*int int86 (int interrupt_number, union REGS *inregs, union REGS *outregs, struct SREGS *segregs);*

Function Invokes the specified 8086 interrupt and assigns the 8086 registers the values contained in the structures inregs and sregs.

Include File <dos.h>

interrupt_number (in):	Desired 8086 interrupt
inregs (in):	Structure containing the values to assign to the 8086 registers (see Chapter 6)
outregs (out):	Structure containing the values contained in the 8086 registers following the interrupt service
sregs (in/out):	Structure containing the 8086 segment registers

Example

```
status = int86x (0x10, inregs, outregs, sregs);
```

Note int86x returns the value of the AX register upon completion of the interrupt service routine.

*int intdos(union REGS *inregs, union REGS *outregs);*

Function Invokes DOS interrupt 0x21 (general-purpose interrupt) after assigning the 8086 registers the values contained in the structure inregs.

Include File <dos.h>

inregs (in):	Structure containing the values to assign to the 8086 registers (see Chapter 6)
outregs (out):	Structure containing the values contained in the 8086 registers following the interrupt service

Example

status = intdos (inregs, outregs);

Note intdos returns the value of the AX register upon completion of the interrupt service routine.

*int intdosx(union REGS *inregs, union REGS *outregs, struct SREGS sregs);*

Function Invokes DOS interrupt 0x21 (general-purpose interrupt) after assigning the 8086 registers the values contained in the structures inregs and sregs.

Include File <dos.h>

inregs (in):	Structure containing the values to assign to the 8086 registers (see Chapter 6)
outregs (out):	Structure containing the values contained in the 8086 registers following the interrupt service
sregs (in/out):	Structure containing the 8086 segment registers

Example

status = intdosx (inregs, outregs, sregs);

Note intdosx returns the value of the AX register upon completion of the interrupt service routine.

*void intr (int interrupt_number, struct REGPACK *regs);*

Function Execute 8086 interrupt service routine.

Include File <dos.h>

interrupt_number (in): Desired interrupt service routine

regs (in/out): Structure containing 8086 registers:

```
struct REGPACK {
     unsigned r_ax, r_bx, r_cx, r_dx;
     unsigned r_bp, r_si, r_di, r_ds;
     unsigned r_es, r_flags;
};
```

Example

```
intr (5, regs);
```

int ioctl (int devhandle, int command [,int argdx, int argcx]);

Function Extended control to an I/O device.

Include File <io.h>

devhandle (in): Handle to the desired device
command (in): Specific command to perform:

- 0 Get device info
- 1 Set device info into argdx
- 2 Read the number of bytes specified by argcx into the buffer pointed to by argdx
- 3 Write the number of bytes specified by argcx from the buffer pointed to by argdx
- 4 Same as command 2. Treat the device handle as a disk-drive specifier.
- 5 Same as command 3. Treat the device handle as a disk-drive specifier.
- 6 Get input status
- 7 Get output status
- 8 Test device removeability
- 11 Reset sharing conflict retry count

Example

```
status = ioctl (handle, 0, &argcx, &argdx);
```

Note This routine provides direct device-driver access.

int isalnum(int character);

Function Returns 1 if the character contained in the parameter character is alphanumeric; otherwise, returns 0.

Include File <io.h>

character (in): Character to examine

Example

```
while (isalnum (letter));
```

int isalpha (int character);

Function Returns 1 if the character contained in the parameter character is in the range A-Z or a-z; otherwise, returns 0.

Include File <io.h>

character (in): Character to examine

Example

```
while (isalpha (letter));
```

int isascii(int character);

Function Returns 1 if the character contained in the parameter character is in the range 0-127.

Include File <io.h>

character (in): Character to examine

Example

```
while (isascii (letter));
```

int isatty (int devicehandle);

Function Returns 1 if the device associated with the device handle is a tty device.

Include File <io.h>

devicehandle (in): Handle to the desired device

Example

```
status = isatty (handle);
```

Note isatty returns the value 1 if the device is a console, terminal, printer, or serial port.

int iscntrl(int character);

Function Returns 1 if the character contained in the parameter character is in the range 0-0x1F.

Include File <io.h>

character (in): Character to examine

Example

```
while (iscntrl (letter));
```

int isdigit(int character);

Function Returns 1 if the character contained in the parameter character is in the range '0'-'9'.

Include File <io.h>

character (in): Character to examine

Example

```
while (isdigit (letter));
```

int isgraph(int character);

Function Returns 1 if the character contained in the parameter character is a printable character other than a space.

Include File <io.h>

character (in): Character to examine

Example

```
while (isgraph(letter));
```

int islower(int character);

Function Returns 1 if the character contained in the parameter character is a lowercase character.

Include File <io.h>

character (in): Character to examine

Example

```
while (islower(letter));
```

int isprint(int character);

Function Returns 1 if the character contained in the parameter character is a printable character.

Include File <io.h>

character (in): Character to examine

Example

```
while (isprint(letter));
```

int ispunct(int character);

Function Returns 1 if the character contained in the parameter character is a punctuation character (iscntrl or isspace).

Include File <io.h>

character (in): Character to examine

Example

```
while (ispunct(letter));
```

int isspace(int character);

Function Returns 1 if the character contained in the parameter character is a space, carriage return, tab, form feed, newline, or vertical tab.

Include File <io.h>

character (in): Character to examine

Example

```
while (isspace(letter));
```

int isupper(int character);

Function Returns 1 if the character contained in the parameter character is an uppercase letter.

Include File <io.h>

character (in): Character to examine

Example

```
while (isupper(letter));
```

int isxdigit(int character);

Function Returns 1 if the character contained in the parameter character is a hexidecimal digit (0-9, 'A'-'F').

Include File <io.h>

character (in):	Character to examine

Example

```
while (isxdigit(letter));
```

*char *itoa (int value, char *str, int radix);*

Function Converts an integer value to its ASCII representation.

Include File <stdlib.h>

value (in):	Integer value to convert
str (out):	String to contain ASCII representation
radix (in):	Specifies the desired radix (2-36): 2 (binary), 10 (decimal), 8 (octal), 16 (hex), and so forth)

Example

```
result = itoa (3344, str, 10);
```

Note itoa does not return an error status.

int kbhit (void);

Function Returns a nonzero value if keys are available in the keyboard buffer. If no keys have been pressed, kbhit returns the value 0.

Include File <conio.h>

Example

```
while (! kbhit());
```

void keep (int status, int paragraphs);

Function Terminates the current program resident in memory.

Include File <dos.h>

status (in):	Exit status value returned to DOS
paragraphs (in):	Number of 16-byte paragraphs DOS must allocate for the memory-resident program

Example

```
keep (1, 1000);
```

Note For more information on memory-resident C programs, refer to Osborne/McGraw-Hill's *C Power User's Guide.*

long labs (long value);

Function Returns the absolute value of a long variable.

Include File <math.h>

value (in):	Value of which to return the absolute value

Example

result = labs (−3443223L);

double ldexp (double value, int exponent);

Function Returns the result of value * 2 raised to the exponent.

Include File <math.h>

value (in):	Value to be multiplied by the expression 2 to the power of exponent
exponent (in):	Power to which to raise the value 2

Example

result = ldexp (value, 10);

ldiv_t ldiv (long numerator, long denominator);

Function Returns the quotient and remainder of the integer division of two numbers.

Include File <stdlib.h>

numerator (in):	Number to be divided
denominator (in):	Number divided into the numerator

Example

result = ldiv (160000L, 56555L);

Note ldiv_t is a structure containing:

```
typedef struct {
   long quot;
   long rem;
} ldiv_t;
```

*void *lfind (void *key_desired, void *base_address, int *num_elements, int element_width, int (*compare_function)());*

Function Performs a generic search of an array for the specified key value.

Include File <stdlib.h>

key_desired (in):	Pointer to the desired value
base_address (in):	Starting address of the array to search
num_elements (in):	Number of elements in the array
element_width (in):	Number of bytes in each element
compare_function (in):	Pointer to the function to be used for element comparisons: Return a value < 0 if a < b Return a value = 0 if a = b Return a value > 0 if a > b

Example

location = lfind (name, namearray, &num_elements, sizeof(name), str_comp);

Note This is a generic sequential search routine. It will work for all types (int, float, char, and so forth). lfind returns the address of the matching element if it is found, or the value 0, otherwise.

void far line (int xstart, int ystart, int xend, int yend);

Function Draws a line between two specified points.

Include File <graphics.h>

xstart, ystart (in):	x and y start coordinates of the line
xend, yend (in):	x and y end coordinates of the line

Example

line (10, 10, 20, 20);

Note line uses the current drawing color.

void far linerel (int x_offset, int y_offset);

Function Draws a line from the current position to the position specified by the x and y offset.

Include File <graphics.h>

x_offset (in):	Relative distance along x axis
y_offset (in):	Relative distance along y axis

Example

```
linerel (10, 10);
```

Note linerel uses the current drawing color.

void far lineto (int x_loc, int y_loc);

Function Draws a line from the current position to the position xloc, yloc.

Include File <graphics.h>

x_loc (in):	Point on x axis to which to draw
y_loc (in):	Point on y axis to which to draw

Example

```
lineto (10, 10);
```

Note lineto uses the current drawing color.

*struct tm *localtime (long *seconds);*

Function Returns a structure containing the current time broken down into its individual parts.

Include File <time.h>

seconds (in): Seconds since 00:00:0 GMT 01/01/1970

Example

```
current_time = localtime (&timeinseconds);
```

Note The structure returned contains:

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

int lock (int filehandle, long offset, long length);

Function Locks a portion of a file as specified.

Include File <io.h>

filehandle (in):	File handle associated with the desired file
offset (in):	Offset to the first byte to lock
length (in):	Number of bytes to lock

Example

status = lock (file, 256, 512);

Note lock provides file locking under DOS 3.X. If successful, lock returns the value 0; otherwise, it returns −1.

double log (double value);

Function Returns the natural logarithm for the specified value.

Include File <math.h>

value (in):	Value of which to return the natural logarithm

Example

```
result = log (value);
```

double log10 (double value);

Function Returns the log to the base 10 of the specified value.

Include File <math.h>

value (in): Value of which to return the log to the base 10

Example

```
result = log10 (value);
```

void longjmp (jmp_buf task_state, int return_value);

Function Performs a long goto outside of the current block of code.

Include File <setjmp.h>

task_state (in):	Buffer storing the values of CS, DS, ES, SS, SI, DI, SP, FP, and flags
return_value (in):	Value to return from jump

Example

```
longjmp (task_state, 1);
```

Note See the routine setjmp.

void lowvideo (void);

Function Selects low-intensity attributes for text display.

Include File <conio.h>

Example

```
lowvideo ();
```

Note This routine allows you to make some text appear duller on your screen display.

unsigned long_lrotl (unsigned long long_value, int num_shifts);

Function Rotates an unsigned long value to the left by the specified number of shifts.

Include File <stdlib.h>

long_value (in):	Unsigned long value to shift left
num_shifts (in):	Number of shifts to perform

Example

long_result = _lrotl (address, 16);

unsigned long _lrotr (unsigned long long_value, int num_shifts);

Function Rotates an unsigned long value to the right the specified number of shifts.

Include File <stdlib.h>

long_value (in):	Unsigned long value to shift right
num_shifts (in):	Number of shifts to perform

Example

long_result = _lrotr (address, 16);

*void *lsearch (void *desired_key, void *base_address, int num_elements, int width, int (*compare_function)());*

Function Searches an array for a specific value. If the value is found, lsearch returns its address. Otherwise, lsearch appends it to the end of the list.

Include File <stdlib.h>

desired_key (in):	Value to search for, or append if not found
base_address (in):	Starting address of the array
num_elements (in):	Number of elements in the array
width (in):	Number of bytes in each element
compare_function (in):	Pointer to the function to be used for the element comparison: Return a value < 0 if a < b Return a value = 0 if a = b Return a value > 0 if a > b

Example

result = lsearch (&ssan, table, &num_elements, sizeof (ssan), compare_float);

Note If lsearch appends the value, it returns the value 0. Otherwise, lsearch returns the address of the desired element.

long lseek (int filehandle, long offset, int location);

Function Moves the file pointer associated with the given file handle to the specified offset.

Include File <io.h>

filehandle (in):	File handle associated with the desired file

offset (in):	Desired offset within the file
location (in):	Location from which to branch: SEEK_SET (0) Beginning of file SEEK_CUR (1) Current position SEEK_END (2) End of file

Example

result = lseek (filehandle, 1024, SEEK_SET);

Note If successful, lseek returns the new file position; otherwise, it returns the value −1.

*char *ltoa (long value, char *str, int radix);*

Function Converts a long int value to its ASCII representation.

Include File <stdlib.h>

value (in):	Long value to convert
str (out):	String to contain ASCII representation
radix (in):	Specifies the desired radix (2-36): 2 (binary), 10 (decimal), 8 (octal), 16 (hex), and so forth

Example

result = ltoa (334114L, str, 10);

Note ltoa does not return an error status.

*void *malloc (size_t num_bytes);*

Function Allocates the number of bytes specified from memory.

Include File <alloc.h>

num_bytes (in):	Number of bytes to allocate

Example

```
node = malloc (255);
```

Note If successful, malloc returns a pointer to the allocated memory. If unsuccessful, malloc returns the value NULL.

*double matherr (struct exception *except);*

Function Defines a math error exception handler.

Include File <math.h>

except (in): Structure containing information about the math exception:

```
struct exception {
   int type;
   char *name;
   double arg1, arg2, retval;
};
```

Example

matherr is not directly called by the user

Notes When Turbo C encounters an exception while performing a mathematical routine from the run-time library, it invokes the routine matherr. By default, this routine simply returns 0. However, you can develop your own matherr routine to interrogate the exception structure. The file math.h defines each of the possible exceptions.

**void memcmp (void *ptr1, void *ptr2, unsigned num_bytes);*

Function Compares the values pointed to by ptr1 to those pointed to by ptr2.

Include File <mem.h>

ptr1 (in):	Pointer to the first block of memory
ptr2 (in):	Pointer to the second block of memory
num_bytes (in):	Number of bytes to compare

Example

result = memcmp (s1, s2, 255);

Note memcmp returns one of the following:

0 if s1 == s2
> 0 if s1 > s2
< 0 if s1 < s2

*void *memcpy (void *destination, void *source, unsigned num_bytes);*

Function Copies the number of specified bytes from one memory location to another.

Include File <mem.h>

destination (in):	Location to which the bytes are copied

source (in):	Location from which the bytes are copied
num_bytes (in):	Number of bytes to copy

Example

```
memcpy (array_b, array_a, sizeof (array_a));
```

*void *memccpy (void *destination, void *source, unsigned char letter, unsigned num_bytes);*

Function Copies the number of specified bytes from one memory location to another, or until the letter contained in the variable letter is copied to the destination.

Include File <mem.h>

destination (in):	Location to which the bytes are copied
source (in):	Location from which the bytes are copied
letter (in):	Letter that, when copied, terminates the copy
num_bytes (in):	Number of bytes to copy

Example

```
memccpy (array_b, array_a, 'a', sizeof (array_a));
```

Note memccpy returns the byte immediately following letter if letter was copied. Otherwise, memccpy returns the value NULL.

*void *memchr (void *ptr, char letter, unsigned num_bytes);*

Function Searches the first num_bytes of an array for the letter contained in the variable letter.

Include File <mem.h>

ptr (in):	Pointer to the array in memory
letter (in):	Letter for which to search
num_bytes (in):	Number of bytes to search in the array

Example

```
location = memchr (str, 'A', sizeof (str));
```

Note memchr returns a pointer to the first occurrence of letter in the string, or the value NULL if letter is not found.

**void memicmp (void *ptr1, void *ptr2, unsigned num_bytes);*

Function Compares the values pointed to by ptr1 to those pointed to by ptr2 and ignores the case of each letter.

Include File <mem.h>

ptr1 (in):	Pointer to the first block of memory
ptr2 (in):	Pointer to the second block of memory
num_bytes (in):	Number of bytes to compare

Example

```
result = memicmp (s1, s2, 255);
```

Note memicmp returns one of the following:

0 if s1 = = s2
\> 0 if s1 > s2
< 0 if s1 < s2

*void *memmove (void *destination, void *source, unsigned num_bytes);*

Function Copies num_bytes from the source memory location to the destination.

Include File <string.h>

destination (in):	Pointer to the destination location in memory
source (in):	Pointer to the source location in memory that contains the data to copy
num_bytes (in):	Number of bytes to copy

Example

ptr = memmove (str1, str2, strlen (str1));

*void *memset (void *ptr, char letter, unsigned num_bytes);*

Function Sets the number of bytes specified in the array pointed to by ptr to the value in the variable letter.

Include File <mem.h>

ptr (in):	Pointer to the array in memory
letter (in):	Letter to assign to the memory locations
num_bytes (in):	Number of bytes to which to assign the value

Example

memset (str, 'A', sizeof (str));

Note memset returns a pointer to the array in memory.

*int mkdir (char *DOSpathname);*

Function Creates the specified DOS subdirectory.

Include File <dir.h>

DOSpathname (in):	DOS subdirectory name to create

Example

```
status = mkdir ("TESTDIR");
```

Note If successful, mkdir returns the value 0. Otherwise, mkdir returns the value −1.

*void far *MK_FP (unsigned segment, unsigned offset);*

Function Returns a far pointer that is a combination of the provided segment and offset.

Include File <dos.h>

segment (in):	Segment address portion of the far address
offset (in):	Offset address portion of the far address

Example

```
far_address = MK_FP (segment, offset);
```

*char *mktemp (char *template);*

Function Creates a unique filename.

Include File <dir.h>

template (in): A string containing 6 X's (XXXXXX)

Example

```
result = mktemp (template);
```

Notes The string template should be in the form "XXXXXX". The routine mktemp replaces the "X's" with a unique filename in the form AA.AAA. mktemp returns a pointer to the new filename.

*double modf (double value, double *integer_portion);*

Function Splits a double-precision value into an integer and fractional portion.

Include File <math.h>

value (in):	Value to split
integer_portion (out):	Integer portion of the value

Example

```
fractional_part = modf (value, &integer_part);
```

void movedata (int source_segment, int source_offset, int target_segment, int target_offset, unsigned num_bytes);

Function Moves the number of specified bytes from the source location to the target.

Include File <mem.h>

source_segment (in):	Segment address of the source location
source_offset (in):	Offset address of the source location
target_segment (in):	Segment address of target location
target_offset (in):	Offset address of the target location
num_bytes (in):	Number of bytes to transfer

Example

```
movedata (segment, offset, 0xB000, 0, 4000);
```

Note movedata does not return a value.

void far moverel (int x_offset, int y_offset);

Function Moves the current position to the position specified by the x and y offsets.

Include File <graphics.h>

x_offset (in):	Relative distance along x axis
y_offset (in):	Relative distance along y axis

Example

moverel (10, 10);

int movetext (int left_corner, int top_corner, int right_corner, int bottom_corner, int new_leftcorner, int new_topcorner);

Function Moves a region of text from one location on the screen to another.

left_corner (in):	Specifies the x coordinate of the upper-left corner of the region to move
top_corner (in):	Specifies the y coordinate of the upper-left corner of the region to move
bottom_corner (in):	Specifies the y coordinate of the lower-right corner of the region to move
right_corner (in):	Specifies the x coordinate of the lower-right corner of the region to move
new_leftcorner (in):	Location to which to move text
new_topcorner (in):	Location to which to move text

Example

movetext (15, 20, 21, 25, 5, 10);

Note If successful, movetext returns the value 1; otherwise, movetext returns 0.

void far moveto (int x_loc, int y_loc);

Function Moves the current position to the position xloc, yloc.

Include File <graphics.h>

x_loc (in):	Point on x axis to which to move
y_loc (in):	Point on y axis to which to move

Example

moveto (10, 10);

*void movmem (void *source, void target, unsigned num_bytes);*

Function Moves the number of bytes specified from a source to a target location.

Include File <mem.h>

source (in):	Pointer to the source location in memory

target (in):	Pointer to the target location in memory
num_bytes (in):	Number of bytes to move

Example

movmem (my_array, your_array, sizeof(my_array));

Note movmem does not return a value.

void normvideo (void);

Function Selects normal video display attributes for text following a call to either highvideo or lowvideo.

Include File <conio.h>

Example

normvideo ();

void nosound (void);

Function Turns off the IBM PC speaker.

Include File <dos.h>

Example

nosound();

Note The routine sound turns on the PC speaker.

*int _open (char *DOSpathname, int access_type);*

Function Opens a file for read or write access.

Include File <io.h>

DOSpathname (in):	String containing the filename to open	
access_type (in):	Specifies the mode of access to support:	
	O_NOINHERIT	Not passed to child process
	O_DENYALL	Only current handle can access
	O_DENYWRITE	Only current handle can write access any open can read the file
	O_DENYREAD	Only current handle can read access any other open can write
	O_DENYNONE	Shared file

Example

```
handle = _open ("TEST.DAT", O_DENYNONE);
```

Note This function is unique to DOS. If an error occurs, _open returns the value −1.

*int open (char *DOSpathname, int access_type[, int permissions]);*

Function Opens a file for read or write access.

Include File <io.h>

DOSpathname (in):	String containing the filename to open	
access_type (in):	Specifies the mode of access to support:	
	O_RDONLY	Read-only access
	O_WRONLY	Write-only access
	O_RDWR	Read/write access
	O_NDELAY	Not used
	O_APPEND	Open in append mode
	O_CREAT	Create the file if non-existent
	O_TRUNC	Truncate the file to 0 bytes if it exists
	O_EXCL	Not used
	O_BINARY	Binary mode open
	O_TEXT	Text mode open
permission (in):	Defines the file permissions:	
	S_IWRITE	Write access granted
	S_IREAD	Read access granted
	S_IREAD | S_IWRITE	Read/write access

Example

```
handle = open ("TEST.DAT", O_BINARY);
```

Note If open experiences an error, it returns the value −1.

void outport (int pord_id, int word);

Function Outputs a value to the specified hardware port.

Include File <dos.h>

port_id (in):	Desired port address
word (in):	Value to output to the port

Example

```
outport (0x3da, 0);
```

void outportb (int pord_id, char byte);

Function Outputs a byte value to the specified hardware port.

Include File <dos.h>

port_id (in):	Port address desired
byte (in):	Value to output to the port

Example

```
outportb (0x3da, 255);
```

*void far outtext (char far *string);*

Function Outputs a character string at the current position in the viewport.

Include File <graphics.h>

string (in):	Character string to display

Example

```
outtext ("TEXT");
```

*void far outtextxy (int x_loc, int y_loc, char far *string);*

Function Displays a text string within the current viewport at the specified x and y location.

Include File <graphics.h>

x_loc, y_loc (in):	Location in the viewport at which to display the string
string (in):	Character string to display

Example

```
outtextxy (5, 10, "TEXT");
```

*char *parsfnm (char *command_line, struct fcb *filecontrol_block, int al_register);*

Function Parses a string into a file-control block containing a drive, filename, and extension.

Include File <dos.h>

command_line (in):	String to parse in search of the filename
filecontrol_block (in):	Structure into which drive, filename, and extension are placed
al_registers (in):	AL register setting for DOS interrupt: 0 Scan past leading separators 2 Match FCB drive specifier with drive found in the command line 4 Match FCB filename to filename found in the command line 8 Match FCB file extension to the file extension found in the command line

Example

result = parsfnm (commline, &file_control_block):

Note If successful, parsfnm returns a pointer to the first byte following the filename. Otherwise, parsfnm returns NULL.

int peek (int segment, int offset);

Function Returns the integer value contained in the memory location pointed to by the specified segment and offset value.

Include File <dos.h>

segment (in):	Segment address desired
offset (in):	Offset address desired

Example

```
char_and_color = peek (0xb800, 0);
```

int peekb (int segment, int offset);

Function Returns the byte value contained in the memory location pointed to by the specified segment and offset value.

Include File <dos.h>

segment (in):	Desired segment address
offset (in):	Desired offset address

Example

```
color = peekb (0xb800, 1);
```

*void perror (char *string);*

Function Displays an error message to stderr and describes the error associated with the most recent system call.

Include File <stdio.h>

string (in):	The name of the program encountering the error

Example

```
perror ("FILECOPY");
```

void far pieslice (int x_loc, y_loc, int start_angle, int end_angle, int radius);

Function Draws a pie slice on your screen at the specified x and y location by using the starting and stopping angles given with a radius as provided. Fill the pie slice with the current fill pattern and color.

x_loc, y_loc (in):	Location at which to draw
start_angle (in):	Starting angle of the pie arc (0-360)
end_angle (in):	Ending angle of the pie arc (0-360)
radius (in):	Desired radius in pixels

Example

```
pieslice (100, 100, 45, 90, 30);
```

Note pieslice uses the current fill pattern and color.

void poke (int segment, int offset, int value);

Function Places the specified integer value into the memory location given by segment and offset.

Include File <dos.h>

segment (in):	Segment address of the desired memory location
offset (in):	Offset address of the desired memory location
value (in):	Value to place into the memory location

Example

```
poke (0xB800, 0, 0x4807);
```

void pokeb (int segment, int offset, int value);

Function Places the specified byte value into the memory location given by segment and offset.

Include File <dos.h>

segment (in):	Segment address of the desired memory location
offset (in):	Offset address of the desired memory location

value (in):	Byte value to place into the memory location

Example

pokeb (0xB800, 0, 'a');

*double poly (double x, int degree, double *poly_array);*

Function Generates a polynomial of degree *n* from the coefficients specified in poly_array. Evaluates the polynomial for the value specified in x and returns the result.

Include File <math.h>

x (in):	Value for which to evaluate the polynomial
degree (in):	Degree of the polynomial
poly_array (in):	Array containing the polynomial coefficients

Example

result = poly (5, 3, coeffs);

double pow (double value, double power);

Function Returns the result of value raised to the specified power.

Include File <math.h>

value (in):	Value to raise to the specified power
power (in):	Desired power

Example

result = pow (5, 2);

double pow10 (int power);

Function Returns the result of the value 10 raised to the specified power.

Include File <math.h>

power (in):	Power to raise the value of 10 to

Example

one_hundred = pow10 (2);

*int printf (char *format_sequence [, argument. . .]);*

Function Provides formatted output to stdout.

Include File <stdio.h>

format_sequence (in):	Control characters that specify the format of the data to be output
argument (in):	Data to be output

Example

```
printf ("DATA %d %f %s \n", 5, 33.44, str_var);
```

Note The printf routine supports the following control sequences:

%d	Signed integer
%i	Signed integer
%o	Octal value
%u	Unsigned integer
%x	Unsigned hexidecimal
%X	Unsigned hexidecimal
%f	Floating-point value
%e	Floating-point value in [−]d.dddd e [+/−]ddd
%g	Floating-point value in either f or e format, depending upon value or precision
%c	Character
%s	String value
%%	% character is printed
%n	Pointer to the type int
%p	Pointer value

You can also append the following input size modifiers:

l	Long value
h	Short integer
f	Far pointer
N	Near pointer

*int putc (int character, FILE *stream);*

Function Outputs the specified character to the file associated with stream.

Include File <stdio.h>

character (in):	Character to be output
stream (in):	File to which character is to be output

Example

result = putc ('A', stdout);

Note If successful, putc returns the value of the output character.

int putch (int character);

Function Outputs the specified character to BDOS or video memory.

Include File <conio.h>

character (in):	Character to be output

Example

(in): result = putch(character);

Note putch writes its output to the current window.

int putchar(int character);

Function Outputs a character to the stdout stream.

Include File <stdio.h>

character (in): Character to be output

Example

```
result = putchar('A');
```

Note The putchar routine is a C macro defined as putc (character, stdout).

*int putenv (char *environment_entry);*

Function Places an entry in the current environment.

Include File <stdlib.h>

environment_entry (in): Character string to be placed into the current environment

Example

```
result = putenv ("FILE=MYFILE");
```

Note If successful, putenv returns the value 0. If an error occurs, putenv returns −1. DOS assigns a copy of the current DOS environment to the executing program. Therefore, putenv does not place an entry in the actual DOS environment, but rather the program's copy.

*void far putimage (int x_loc, y_loc, void far *buffer, int operation);*

Function Places a graphics image previously saved by getimage back on the screen display at the specified location.

Include File <graphics.h>

x_loc, y_loc (in):	Coordinates of upper-left corner of the image
buffer (in):	Buffer containing the graphics image
operation (in):	Specifies how the pixels are placed back onto the screen: 0 Straight copy 1 Exclusive or 2 Inclusive or 3 And 4 Inverse source copy

Example

```
putimage (100, 100, box, 0);
```

void far putpixel (int x_loc, int y_loc, int pixel_color);

Function Displays a pixel of the specified color at the x and y location given.

Include File <graphics.h>

x_loc, y_loc (in):	Location at which to display the pixel
pixel_color (in):	Color of the pixel

Example

```
putpixel (100, 100, 1);
```

*int puts (char *string);*

Function Writes a string to the stream associated with stdout.

Include File <stdio.h>

string (in):	Character string to be displayed

Example

```
puts ("String to output");
```

Note If successful, puts returns the last character written. If an error occurs, puts returns EOF.

*int puttext (int left_corner, int top_corner, int right_corner, int bottom_corner, void *buffer);*

Function Copies text stored in a buffer back to the screen display.

left_corner (in):	Specifies the x coordinate of the upper-left corner of the region to restore
top_corner (in):	Specifies the y coordinate of the upper-left corner of the region to restore
bottom_corner (in):	Specifies the y coordinate of the lower-right corner of the region to restore
right_corner (in):	Specifies the x coordinate of the lower-right corner of the region to restore

Example

```
status = puttext (10, 10, 20, 20, buffer);
```

Note If successful, puttext returns 1; otherwise, it returns 0.

*int putw (int word, FILE *stream);*

Function Outputs a word (16 bit) value to the specified file stream.

Include File <stdio.h>

word (in):	16-bit value to be output
stream (in):	File stream to output to

Example

result = putw (1024, fp);

Note If successful, putw returns the integer value output.

*void qsort (void *base_address, int num_elements, int width, int (*compare_function)());*

Function Uses quick sort to sort the items in an array.

Include File <stdlib.h>

base_address (in):	Address of the first element in the array
num_elements (in):	Number of array elements
width (in):	Number of bytes in each element
compare_function (in):	Function to be used to compare array elements

Example

qsort (my_array, 10, sizeof(float), float_compare);

Note This quick sort is a generic quick sort algorithm that works for all array types. See Chapter 11 for more information on the comparison functions.

int rand(void);

Function Returns a random number.

Include File <stdlib.h>

Example

random = rand ();

Note To reseed the random number generator, use srand.

*int randbrd (struct fcb *file_control_block, int num_records);*

Function Using a file control block, reads the number of records specified by num_records.

Include File <dos.h>

file_control_block (in):	Pointer to a file control block that contains the file characteristics
num_records (in):	Number of records to read

Example

```
result = randbrd (&file_control_block, 5);
```

Note: The randbrd routine returns one of the following values:

0 All records were read
1 End of file reached (all records read)
2 Incomplete records are read
3 End of file reached (records incompletely read)

*int randbwr (struct fcb *file_control_block, int num_records);*

Function Using a file control block, writes the number of records specified by num_records.

Include File <dos.h>

file_control_block (in):	Pointer to a file control block that contains the file characteristics
num_records (in):	Number of records to write

Example

result = randbwr (&file_control_block, 5);

Note The randbwr routine returns one of the following values:

0	All records written
1	Insufficient disk space

int random (int boundary);

Function Returns a random number between 0 and the value of boundary −1.

Include File <stdlib.h>

boundary (in):	This value −1 is the highest value rand can return

Example

start = random (10);

Note The random routine is defined as rand % num.

void randomize (void);

Function Initializes the random number generator.

Include File <stdlib.h>

Example

```
randomize ();
```

Note The randomize routine initializes the random number generator with a random value.

*int _read (int file_handle, void *buffer, int num_bytes);*

Function Reads the specified number of bytes from the file associated with the file handle.

Include File <io.h>

file_handle (in):	File handle associated with the desired file
buffer (in):	Location to read the data into
num_bytes (in):	Number of bytes to read

Example

```
result = _read (filehandle, buffer, 256);
```

Note If successful, _read returns the number of bytes read. If an error occurs, _read returns −1, and on end of file, it returns the value 0. The _read routine directly calls a DOS system service. The maximum number of bytes this routine can read is 65,534.

*int read (int file_handle, void *buffer, int num_bytes);*

Function Reads the specified number of bytes from the file associated with the file handle.

Include File <io.h>

file_handle (in):	File handle associated with the desired file
buffer (in):	Location to read the data into
num_bytes (in):	Number of bytes to read

Example

```
result = read (filehandle, buffer, 256);
```

Note If successful, read returns the number of bytes read. If an error occurs, read returns −1, and on end of file, it returns the value 0. The maximum number of bytes this routine can read is 65,534.

*void *realloc (void *pointer, unsigned newsize_in_bytes);*

Function Modifies the amount of a previously allocated section of memory.

Include File <alloc.h>

pointer (in):	Pointer to the previously allocated memory
newsize_in_bytes (in):	Size desired for the memory block

Example

ptr = realloc (ptr, 1024);

Note If successful, realloc returns a pointer to the new block of allocated memory. If the request is unsuccessful, realloc returns NULL.

void far rectangle (int left_corner, int top_corner, int right_corner, int bottom_corner);

Function Draws a rectangle with the specified corners.

Include File <graphics.h>

left_corner (in):	Specifies the x coordinate of the upper-left corner of the rectangle
top_corner (in):	Specifies the y coordinate of the upper-left corner of the rectangle
bottom_corner (in):	Specifies the y coordinate of the lower-right corner of the rectangle
right_corner (in):	Specifies the x coordinate of the lower-right corner of the retangle

Example

rectangle (10, 10, 20, 30);

Note The rectangle routine uses the current line thickness and drawing color.

*int registerbgidriver (void (*driver)(void));*

Function Registers linked-in graphics driver code.

Include File <graphics.h>

Example

status = registerbgidriver (EGA_driver);

Note If registerbgidriver encounters an error, it returns the corresponding error code; otherwise, it returns the value 0.

*int registerbgifont (void(*font)(void));*

Function Registers a linked-in graphics font.

Include File <graphics.h>

Example

status = registerbgifont (big_font);

Note If registerbgifont encounters an error, it returns the corresponding error code; otherwise, it returns the value 0.

*int rename (char *oldname, char *newname);*

Function Renames an existing file as specified.

Include File <stdio.h>

oldname (in):	Current name of the file to rename
newname (in):	Desired name of the file to rename

Example

result = rename ("TEST.C", "TEST.SAV");

Note If successful, rename returns the value 0. If an error occurs, rename returns the value −1.

void far restorecrtmode (void);

Function Restores the screen mode to the setting that was in effect prior to a call to initgraph.

Include File <graphics.h>

Example

restorecrtmode ();

*int rewind (FILE *stream);*

Function Resets the file pointer of the specified stream to the beginning of a file.

Include File <stdio.h>

stream (in):	File stream associated with the file to reset

Example

result = rewind (fp);

Note If successful, rewind returns the value 0. If an error occurs, rewind will return a nonzero result.

*int rmdir (char *directory_name);*

Function Removes the specified DOS directory.

Include File <dir.h>

directory_name (in):	Name of the DOS subdirectory to remove

Example

result = rmdir ("QUICKC");

Note The rmdir routine cannot remove a directory if the directory contains files, the directory is the current directory, or the directory is the root directory. If successful, rmdir returns the value 0. Otherwise, rmdir returns the value −1.

unsigned _rotl (unsigned value, int num_shifts);

Function Rotates an unsigned value to the left the number of shifts specified.

Include File <stdlib.h>

value (in):	Unsigned value to shift left
num_shifts (in):	Number of shifts to perform

Example

result = _rotl (address, 16);

unsigned _rotr (unsigned value, int num_shifts);

Function Rotates an unsigned value to the right the number of shifts specified.

Include File <stdlib.h>

value (in):	Unsigned value to shift right
num_shifts (in):	Number of shifts to perform

Example

result = _rotr (address, 16);

*char *sbrk (int increment);*

Function Adds the number of bytes specified to the data space allocation (see brk).

Include File <alloc.h>

increment (in):	Number of bytes to add to the data space

Example

result = sbrk (1024);

Note If successful, sbrk returns the previous brk value.

int scanf (char format—sequence [,argument...]);

Function Perform formatted input from stdin.

Include File <stdio.h>

format_sequence (in):	Control sequence specifying the output format
argument (in):	Data to be read

Example

```
num_fields = scanf ("%d %d %f \n", &a, &b, &c);
```

Note The scanf routine returns the number of fields successfully filled. See printf for the control-sequence formatting characters.

*char *searchpath (char *filename);*

Function Searches the DOS PATH for a file that matches the name given, and if the file is found, returns a complete DOS pathname to the file.

Include File <dir.h>

filename (in):	Name of the DOS file to search for

Example

```
pathname = searchpath ("TURBOC.DAT");
```

Note If successful, searchpath returns the complete pathname. If the file is not found, searchpath returns the value NULL.

*void segread (struct REGS *segment_registers);*

Function Returns the current values of the segment registers.

Include File <dos.h>

segment_registers (in): Structure containing the DOS segment registers

Example

segread (&segment_registers);

void far setactivepage (int page_number);

Function Specifies the active video display page for graphics output.

Include File <graphics.h>

page_number (in): Desired video display page number

Example

setactivepage (2);

Note Only EGA, VGA, and Hercules graphics cards support multiple graphics display pages.

*void far setallpalette (struct palettetype far *palette);*

Function Defines the colors of the palette.

Include File <graphics.h>

palette (in):	Structure containing the palette colors:
	struct palettetype {
	unsigned char size;
	signed char colors
	[MAX_COLORS+1];
	};

Example

```
setallpallette (&my_palette);
```

void far setbkcolor (int background_color);

Function Sets the current graphics background color.

Include File <graphics.h>

background_color (in):	Desired color from your current palette

Example

```
setbkcolor (1);
```

int setblock (int segment, int newsize_in_bytes);

Function Modifies the size of a previously allocated DOS segment.

Include File <dos.h>

segment (in):	Previously allocated DOS segment
newsize_in_bytes (in):	Desired segment size in bytes

Note If successful, setblock returns the value −1. If an error occurs, setblock returns the value of the largest available block.

*void setbuf (FILE *stream, char *buffer);*

Function Assigns a new buffer to be used for file I/O to the specified stream.

Include File <stdio.h>

stream (in):	File stream to be buffered
buffer (in):	Memory location to be used for buffering

Example

```
setbuf (fp, char_array);
```

Note If the buffer specified is NULL, I/O to the file stream is not buffered.

int setcbrk (int status);

Function Enables/disables control-break checking.

Include File <dos.h>

status (in):	Desired control break setting: 0 Disables control-break checking 1 Enables control-break checking

Example

```
result = setcbrk (1);
```

Note The setcbrk routine returns the current state of control-break checking.

void far setcolor (int color);

Function Sets the current drawing color.

color (in):	Desired drawing color from your current palette

Example

```
setcolor (2);
```

*void setdate (struct date *current_date);*

Function Sets the current DOS system date.

Include File <dos.h>

current_date (out): Structure containing the current system date:

```
struct date {
   int da_year;
   char da_day;
   char da_mon;
};
```

Example

```
status = setdate (&current_date);
```

int setdisk (int disk_drive);

Function Sets the disk drive as specified.

Include File <dir.h>

disk_drive (in): Desired disk drive (A = 0, B = 1, C = 2...)

Example

```
result = setdisk (1);
```

Note The setdisk routine returns the number of disk drives available.

*void setdta (char far *disk_transfer_address);*

Function Defines a new disk transfer address.

Include File <dos.h>

disk_transfer_address (in): Address of the desired disk transfer

Example

```
setdta ((char far *) buffer);
```

*setfillpattern (char far *fill_pattern, int color);*

Function Selects a user-defined fill pattern for graphics mode.

fill_pattern (in):	Array containing desired fill pattern
color (in):	Desired fill color from current palette

Example

```
setfillpattern (xxx, 1);
```

void far setfillstyle (int fill_style, int color);

Function Selects a fill style and color.

fill_pattern (in):	Desired fill style: 0 Empty fill 1 Solid fill 2 Line fill - 3 Left-slash fill / 4 Thick left-slash fill 5 Backslash fill \ 6 Thick backslash fill 7 Light hatch fill 8 Heavy crosshatch fill 9 Interleaving line fill 10 Wide-spaced dot fill 11 Close-spaced dot fill
color (in):	Desired fill color from your current palette

Example

setfillstyle (3, 1);

*int setftime (int file_handle, struct ftime *file_stamp);*

Function Sets a file's date and time stamp.

Include File <dos.h>

file_handle (in):	File handle associated with the desired file
file_stamp (out):	Structure containing the file's date and time:

```
struct ftime {
   unsigned ft_tsec:5;
   unsigned ft_min:6;
   unsigned ft_hour:5;
   unsigned ft_day: 5;
   unsigned ft_month:4;
   unsigned ft_year: 7;
};
```

Example

status = setftime (filehandle, &file_stamp);

Note If successful, setftime returns the value 0.

unsigned far setgraphbufsize (unsigned buffer_size);

Function Defines the size of the internal graphics buffer.

buffer_size (in): Size, in bytes, of the buffer desired

Example

setgraphbufsize (9182);

Note You must call setgraphbufsize before initgraph.

void far setgraphmode (int graphics_mode);

Function Selects the current graphics mode.

Include File <graphics.h>

graphics_mode (in): Desired graphics mode

Example

setgraphmode (CGA);

Note See the graphics.h file for graphics modes.

int setjmp (jmp_buf task_state);

Function Marks the location for a future goto that is outside the current block of code.

Include File <setjmp.h>

task_state (in):	Buffer storing the values of CS, DS, ES, SS, SI, DI, SP, FP, and flags

Example

```
setjmp (&task_state);
```

Note See the longjmp routine.

void far setlinestyle (int line_style, unsigned pattern, int thickness);

Function Sets the current width and line style.

Include File <graphics.h>

line_style (in):	Desired line style: 0 Solid line 1 Dotted line 2 Centered line 3 Dashed line 4 User-defined line style
pattern (in):	Desired line pattern if user-defined pattern is used
thickness (in):	Desired thickness for the line: 1 One-pixel thickness 3 Three-pixel thickness

Example

setlinestyle (0, 0, 1);

*void setmem (void *address, int num_bytes, char letter);*

Function Assigns the number of occurrences of the specified letter to the address given.

Include File <mem.h>

address (in):	Memory address to assign the characters to
num_bytes (in):	Number of bytes to assign the letters to
letter (in):	Value to assign to the memory locations

Example

setmem (ptr, 255, 'A');

int setmode (int filehandle, unsigned mode);

Function Sets the file associated with the given handle to the mode (text or binary) as specified.

Include File <io.h>

filehandle (in):	File handle of the file to modify

mode (in):	Desired mode:	
	O_BINARY	Binary file
	O_TEXT	Text file

Example

setmode (fp, O_BINARY);

Note If successful, setmode returns the value 0. Otherwise, it returns the value −1.

void far setpalette (int index, int color);

Function Assigns an actual color to an index in the current palette.

Include File <graphics.h>

index (in):	Index into the palette that you are assigning a color to
color (in):	Actual color value assigned to the palette index

Example

setlinestyle (1, 4);

void far settextjustify (int horizontal, int vertical);

Function Specifies text justification.

Include File <graphics.h>

horizontal (in):	Specifies how horizontal text is to be justified: 0 Left justify 1 Center justify 2 right justify
vertical (in):	Specifies how vertical text is to be justified: 0 Bottom justify 1 Center justify 2 Top justify

Example

settextjustify (1, 1);

void far settextstyle (int font, int direction, int size);

Function Specifies the graphics mode text font, direction, and size.

Include File <graphics.h>

font (in):	Specifies desired font: 0 8 × 8 bit-mapped font 1 Triplex font 2 Small font 3 Sans serif font 4 Gothic font
direction (in):	Specifies how text is to be written: 0 Horizontal left to right 1 Vertical bottom to top

size (in):	Specifies font size (1-10): 1 8 × 8 2 16 × 16

Example

settextstyle (0, 0, 1);

*void settime (struct time *system_time);*

Function Sets the current system time.

Include File <dos.h>

system_time (out): Structure containing the current system time:

```
struct time {
    unsigned char ti_min;
    unsigned char ti_hour;
    unsigned char ti_hund;
    unsigned char ti_sec;
};
```

Example

settime (&current_time);

type (in):	Specifies the type of buffering desired: _IOFBF Full file buffer for input/output _IOLBF Line buffer the file _IONBF No buffering for the file
num_bytes (in):	Number of bytes to allocate for the buffer

Example

setvbuf (fp, char_array, _IONBF, 0);

Note If the buffer specified is NULL, I/O to the file stream is not buffered.

*void setvect (int interrupt_number, void interrupt (*service_routine) ());*

Function Defines a new interrupt handler for the specified interrupt.

Include File <dos.h>

interrupt_number (in):	Interrupt to define a new service routine for
service_routine (in):	Function serve as the new interrupt handler

void far setusercharsize (int xmult, int xdiv, int ymult, int ydiv);

Function Specifies graphics mode character magnification.

Include File <graphics.h>

xmult, xdiv (in):	Width scaling factors
ymult, ydiv (in):	Height scaling factors

Example

```
setusercharsize (2, 1, 3, 2);
```

Note These values are only active when you have called the settextstyle routine with charsize equal to 0.

*void setvbuf (FILE *stream, char *buffer, int type, unsigned num_bytes);*

Function Assigns a new buffer to be used for file I/O to the specified stream.

Include File <stdio.h>

stream (in):	File stream to be buffered
buffer (in):	Memory location to be used for buffering

Example

setvect (5, my_print_routine);

void setverify (int state);

Function Enables or disables disk verification.

Include File <dos.h>

state (in):	Desired disk verification state: 0 disables and 1 enables disk verification

Example

setverify (1);

void far setviewport (int left_corner, int top_corner, int right_corner, int bottom_corner, int clip);

Function Defines the current viewport for graphics output.

Include File <graphics.h>

left_corner (in):	Upper-left corner x viewport coordinate
top_corner (in):	Upper-left corner y viewport coordinate

right_corner (in):	Lower-right corner x viewport coordinate
bottom_corner (in):	Lower-right corner y viewport coordinate
clip (in):	Specifies whether values outside of the viewport are clipped. If clip is a nonzero value, clipping is enabled

Example

setviewport (10, 10, 200, 200, 1);

void far setvisualpage (int page_number);

Function Sets the video display page to be displayed.

Include File <graphics.h>

page_number (in):	Desired video display page number

Example

setvisualpage (2);

Note Only EGA, VGA, and Hercules graphics cards support multiple graphics display pages.

double sin (double value);

Function Returns the trigonometric sine of the specified value.

Include File <math.h>

value (in): Value to return the sine of

Example

result = sin (pi);

double sinh (double value);

Function Returns the hyperbolic sine of a value.

Include File <math.h>

value (in): Value to return the hyperbolic sine of

Example

result = sinh (value);

void sleep (unsigned seconds);

Function Suspends the current application for the interval of time specified.

Include File <dos.h>

seconds (in): Number of seconds to suspend the application for

Example

sleep (10);

void sound (unsigned frequency);

Function Turns on the IBM PC speaker at the specified frequency.

Include File <dos.h>

frequency (in):	Desired frequency for the speaker sound

Example

```
sound (14);
```

*int spawn . . . (int mode, char *command, char *arg [, . . .], NULL);*

Function Creates and executes a child process.

Include File <process.h>

mode (in):	Action taken after spawn call:	
	P_WAIT	Wait until child process completes
	P_NOWAIT	Continue to run as child process runs
	P_OVERLAY	Overlay child process in memory previously contained by the parent
command (in):	Complete DOS pathname of the command to execute	
arg (in):	Command-line argument passed to the child process	

Example

spawn (P_WAIT, "BACKUP", "C:*.*", "A:", NULL);

Note Several versions of spawn exist:

spawnl	Search only the root or current directory
spawnle	Same as spawnl; also allows environment to be passed as a parameter
spawnp	Searches the DOS PATH command
spawnv	Command-line arguments are passed as a single array of pointers
spawnlp	
spawnlpe	
spawnve	
spawnvp	
spawnvpe	

See the dos.h file for the calling sequence of each command. If successful, spawn returns the value 0.

*int sprintf (char *string, char *format_sequence[,argument . . .]);*

Function Writes formatted output to a string, as opposed to a file stream.

Include File <stdio.h>

string (out):	Character string containing the formatted output
format_sequence (in):	Control sequence that specifies how to format the data
argument (in):	Data to be output

Example

result = sprintf (str, "%d", age);

Note The sprintf routine returns the number of bytes output, not including the NULL terminal.

double sqrt (double value);

Function Returns the square root of a value.

Include File <math.h>

value (in):	Value to return the square root of

Example

five = sqrt (25.0);

void srand (unsigned seed);

Function Initializes or seeds the random number generator.

Include File <stdlib.h>

seed (in):	Desired seed for the random number generator

Example

srand (time(&current));

*int sscanf (char *string, char format_sequence [,argument...]);*

Function Performs formatted input from a string, as opposed to a file stream.

Include File <stdio.h>

string (in):	Character string to read from
format_sequence (in):	Control sequence specifying the output format
argument (in):	Data to be read

Example

num_fields = sscanf (str, "%d %d %f \n", &a, &b, &c);

Note The sscanf routine returns the number of fields successfully filled.

*int stat (char *pathname, struct stat *stat_info);*

Function Returns information about the specified file.

Include File <stat.h>

pathname(in):	Pathname of the desired file
stat_info (out):	Structure containing the file information

Example

```
result = stat ("TEST.C", &stat_info);
```

Note If successful, stat returns the value 0. Otherwise, it returns the value −1.

unsigned int _status87 (void);

Function Returns the current math coprocessor status word.

Include File <float.h>

Example

```
status = _status87();
```

Note The float.h file defines the return value of _status87.

*int stime (long *seconds);*

Function Sets the current system time to the number of seconds since 00:00 01/01/1970.

Include File <time.h>

seconds (in):	Number of seconds since 00:00 01/01/1970

Example

```
stime (&lots_of_seconds);
```

*char *stpcpy (char *destination, char *source);*

Function Copies the contents of the source string to the destination.

Include File <string.h>

destination (out):	String characters are copied to
source (in):	String characters are copied from

Example

```
result = stpcpy (destination, "STRING TO COPY");
```

Note The stpcpy routine returns destination + the number of characters copied.

*char *strcat (char *destination, char *source);*

Function Appends the contents of the source string to the destination.

Include File <string.h>

destination (out):	String characters are appended to
source (in):	String characters are copied from

Example

result = strcat (destination, "STRING TO APPEND");

Note The strcat routine returns destination + the number of characters appended.

*strchr (char *string, char letter);*

Function Searches a given string for the specified character.

Include File <string.h>

string (in):	Character string to search
letter (in):	Letter to search for

Example

loc = strchr (str, 'A');

Note The strchr routine returns a pointer to the first occurrence of the letter specified or the value NULL if the letter does not exist.

*int strcmp (char *s1, char *s2);*

Function Compares the contents of two character strings.

Include File <string.h>

s1, s2 (in): Character strings to compare

Example

result = strcmp ("STRING 1", "STRING 2");

Note The strcmp routine returns a value that is

<0 if s1 < s2
=0 if s1 == s2
>0 if s1 > s2

*char *strcpy (char *destination, char *source);*

Function Copies the contents of the source string to the destination.

Include File <string.h>

destination (out):	String characters are copied to
source (in):	String characters are copied from

Example

result = strcpy (destination, "STRING TO COPY");

Note The strcpy routine returns destination.

*int strcspn (char *s1, char *s2);*

Function Returns an index into s1 that consists entirely of characters not contained in s2.

Include File <string.h>

s1 (in):	String to return the index into
s2 (in):	String of characters to compare s1 characters to

Example

index = strcspn (str, "ABCDE");

*char *strdup (char *str);*

Function Returns a pointer to a string containing the same sequence of characters as the given string.

Include File <string.h>

str (in): Character string to duplicate

Example

result = strdup ("String to duplicate");

Note The strdup routine returns a pointer to the new string, or it returns the value NULL if space for the string could not be allocated.

*char *_strerror (const char *string);*

Function Generates customized error messages.

Include File <string.h>

string (in): Contains the most current error message

Example

result = _strerror (str);

*char *strerror (char *string);*

Function Returns a pointer to the error message string, allowing you to develop customized error messages.

Include File <string.h>

string (in): Customized error message

Example

result = strerror ("Invalid disk drive specified \n");

Note If string is NULL, the result of strerror is the error message associated with the last system error.

*int stricmp (char *s1, char *s2);*

Function Compares one string to another, ignoring the case of each letter.

Include File <string.h>

s1, s2 (in): Character strings to compare

Example

result = stricmp ("String 1", "STRING 1");

Note The stricmp routine returns a value that is

<0 if s1 < s2
=0 if s1 == s2
>0 if s1 > s2

*unsigned strlen (char *string);*

Function Returns a count of the number of characters in a string.

Include File <string.h>

string (in): Character string to return the number of characters in

Example

length = strlen ("String to count");

*char *strlwr (char *string);*

Function Converts uppercase letters in a string to lowercase.

Include File <string.h>

string (in/out):	String to convert to lowercase

Example

result = strlwr (string);

*char *strncat (char *destination, char *source, int num_bytes);*

Function Appends the contents of the source string to the destination. Do not let the resultant string exceed num_bytes characters.

Include File <string.h>

destination (out):	String characters are appended to
source (in):	String characters are copied from
num_bytes (in):	Maximum number of bytes in destination

Example

result = strcat (destination, src, sizeof (destination));

Note The strcat routine returns destination + the number of characters appended.

*int strncmp (char *s1, char *s2, int num_bytes);*

Function Compares the contents of two character strings.

Include File <string.h>

s1, s2 (in):	Character strings to compare
num_bytes (in):	Maximum number of bytes to examine

Example

result = strncmp (s1, "STRING 2", strlen (s1));

Note The strncmp routine returns a value that is

<0 if s1 < s2
=0 if s1 == s2
>0 if s1 > s2

*char *strncpy (char *destination, char *source, num_bytes);*

Function Copies the contents of the source string to the destination.

Include File <string.h>

destination (out):	String characters are copied to

source (in):	String characters are copied from
num_bytes (in):	Maximum number of bytes to copy to destination

Example

```
result = strncpy (destination, s1, sizeof (destination));
```

Note The strncpy routine returns destination.

*int strnicmp (char *s1, char *s2, int num_bytes);*

Function Compares the contents of two character strings, ignoring the case of each letter.

Include File <string.h>

s1, s2 (in):	Character strings to compare
num_bytes (in):	Maximum number of bytes to examine

Example

```
result = strnicmp (s1, "STRING 2", strlen (s1));
```

Note The strnicmp routine returns a value that is

<0 if s1 $<$ s2
$=0$ if s1 $==$ s2
>0 if s1 $>$ s2

*char *strnset (char *string, char character, int max_bytes);*

Function Assigns max_bytes occurrences of the specified character to the given string.

Include File <string.h>

string (out):	Character string to assign the characters to
character (in):	Character to assign to the string
max_bytes (in):	Number of characters to assign

Example

```
result = strnset (str, 'A', sizeof(str));
```

*char *strpbrk (char *s1, char *s2);*

Function Scans s2 for the first occurrence of a character in s1.

Include File <string.h>

s2 (in):	Search to scan
s1 (in):	Set of letters to search for

Example

result = strpbrk (s1, s2);

Note The strpbrk routine returns a pointer to the first character in s2 that occurs in s1. If no characters occur, strpbrk returns the value NULL.

*char *strrchr (char *str, char character);*

Function Searches a string for the rightmost occurrence of the specified character.

Include File <string.h>

str (in):	String to search
character (in):	Letter to search for

Example

index = strrchr (str, 'z');

Note If the letter does not occur in the string, strrchr returns the value NULL. If the letter occurs, strrchr returns a pointer to the rightmost location.

*char *strrev (char *string);*

Function Reverses the characters contained in the string.

Include File <string.h>

string (in/out):	String containing the characters to reverse

Example

result = strrev (str);

*char *strset (char *string, char character);*

Function Assigns each of the characters in a string to the character specified.

Include File <string.h>

string (out):	Character string to assign the letter to
character (in):	Character to assign to the string

Example

result = strset (str, 'A');

*char *strstr (char *s1, char *s2);*

Function Searches the s1 string for the first occurrence of the s2 string.

Include File <string.h>

s1 (in):	String to search for
s1 (in):	Character string to search

Example

```
index_ptr = strstr ("This is it", "is");
```

Note If the string is found, strstr returns a pointer to the first occurrence of the string in s2. If the string is not found, strstr returns the value NULL.

*double strtod (char *string, char **end);*

Function Converts a character string representation of a floating-point value to a value of type double.

Include File <string.h>

string (in):	Character string to convert
end (out):	Character that the conversion stopped at

Example

double_result = strtod ("133.344", &end);

Note The strtod routine stops at NULL or at the first character that cannot be converted. If *end is not equal to NULL, the string contained invalid characters.

*char *strtok (char *s1, char *s2);*

Function Searches the character string s1 for a set of tokens defined in s2.

Include File <string.h>

s1 (in):	Character string to search
s2 (in):	String of tokens

Example

result = strtok (s1, s2);

Note If a token is found, strtok returns a pointer to that location. Otherwise, strtok returns the value NULL.

*long strtol (char *string, char **end, int radix);*

Function Converts a character string representation of a long value to a value of type long.

Include File <string.h>

string (in):	Character string to convert
end (out):	Character that the conversion stopped at
radix (in):	Base of the value contained in the string

Example

long_result = strtol ("133344L", &end, 10);

Note The strtol routine stops at NULL or at the first character that cannot be converted. If *end is not equal to NULL, the string contained invalid characters.

*unsigned long strtoul (const char *str, char **end_pointer, int radix);*

Function Converts a string containing an ASCII representation of a value to an unsigned long integer.

Include File <stdlib.h>

str (in):	String containing ASCII representation of the value

end_pointer (out):	Pointer to the last character used in the conversion
int radix (in):	Base of ASCII representation (2, 8, 10, 16)

Example

result = strtoul ("56333", end_pointer, 10);

*char *strupr (char *string);*

Function Converts a character string to uppercase.

Include File <string.h>

string (in):	Character string to convert to uppercase

Example

result = strupr (s1);

*void swab (char *s1, char *s2, int num_bytes);*

Function Swaps the specified number of bytes from one string to another.

Include File <stdlib.h>

s1,s2 (in/out):	Strings containing the bytes to exchange

Example

```
swab (s1, s2, sizeof (s1));
```

*int system (char *DOScommand);*

Function Invokes a DOS command from within your program.

Include File <stdlib.h>

DOScommand (in): DOS system command to execute

Example

```
result = system ("DIR");
```

Note The value returned is that generated by COMMAND.COM.

double tan (double value);

Function Returns the trigonometric tangent of the specified value.

Include File <math.h>

value (in): Value to return the tangent of

Example

result = tan (pi * x);

double tanh (double value);

Function Returns the hyperbolic tangent of the specified value.

Include File <math.h>

value (in):	Value to return the tangent of

Example

result = tanh (pi * x);

long tell (int filehandle);

Function Returns the file pointer position for the specified file.

Include File <io.h>

filehandle (in):	File handle associated with the desired file

Example

loc = tell (filehandle);

void textattr (int attribute);

Function Sets foreground and background text mode colors and attributes.

Include File <conio.h>

attribute (in):	Specifies the foreground and background colors. First four LSBs are foreground color; next three LSBs are background color; MSB is blink attribute enable bit.

Example

```
textattr (0xA1);
```

Note The following colors are defined in conio.h:

Foreground and Background:

0	Black	5	Magenta
1	Blue	6	Brown
2	Green	7	Light gray
3	Cyan	8	Dark gray
4	Red		

Foreground Only:

9	Light blue	13	Light magenta
10	Light green	14	Yellow
11	Light cyan	15	White
12	Light red	128	Blink

void textbackground (int background—color);

Function Selects the desired background color.

Include File <conio.h>

background_color (in): Desired background color (0-7)

Example

textbackground (3);

Note See conio.h for color definitions.

void textcolor (int color);

Function Selects a new character color for text mode.

Include File <conio.h>

color (in): Desired color

Example

textcolor (1);

*int far textheight (char far *string);*

Function Returns the pixel height of a string.

Include File <graphics.h>

string (in):	String of interest

Example

```
result = textheight ("TEXT"); /* height by default is 8 */
```

void textmode (int desired_mode);

Function Selects a specific text video mode.

Include File <conio.h>

desired_mode (in):	Desired text mode:
−1	Last text mode selected
0	Black and white, 40 column
1	Color, 40 column
2	Black and white, 80 column
3	Color, 80 column
7	Monochrome, 80 column

Example

```
textmode (3);
```

*int far textwidth (char far *string);*

Function Returns the pixel width of a string.

Include File <graphics.h>

string (in): String of interest

Example

result = textwidth ("TEXT"); /* width by default is 8 * 4 */

*long time (long *seconds);*

Function Returns the number of seconds that have elapsed since 00:00 01/01/1970.

Include File <time.h>

seconds (out): Number of seconds since 00:00 01/01/1970

Example

result = time (&seconds);

Note The time routine also returns the number of seconds.

int toascii (int character);

Function Converts a given character to a value in the range 0-127.

Include File <ctype.h>

character (in): Character to convert to ASCII

Example

ltr = toascii (extended_ascii_char);

int _tolower (int character);

Function Converts an uppercase letter to lowercase.

Include File <ctype.h>

character (in): Character to convert to lowercase

Example

lower = _tolower(character);

int tolower (int character);

Function Converts an uppercase letter to lowercase.

Include File <ctype.h>

character (in): Character to convert to lowercase

Example

lower = tolower(character);

int _toupper (int character);

Function Converts a lowercase letter to uppercase.

Include File <ctype.h>

character (in): Character to convert to uppercase

Example

upper = _toupper(character);

int toupper (int character);

Function Converts a lowercase letter to uppercase.

Include File <ctype.h>

character (in): Character to convert to uppercase

Example

upper = toupper(character);

void tzet (void);

Function A UNIX-compatibility routine; no function under DOS.

Example

tzet ();

*char *ultoa (unsigned long value, char *str, int radix);*

Function Converts an unsigned long int value to its ASCII representation.

Include File <stdlib.h>

value (in):	Long value to convert
str (out):	String to contain ASCII representation
radix (in):	Specifies the desired radix (2-36): 2=binary, 10=decimal, 8=octal, 16=hex, and so on

Example

result = ultoa (334114L, str, 10);

Note The ultoa routine does not return an error status.

*int ungetc (char character, FILE *stream);*

Function Pushes a value back into the input file stream.

Include File <stdio.h>

character (in):	Value to put back into the file stream
stream (in):	File stream desired

Example

result = ungetc (letter, fp);

Note The ungetc routine returns the value put back into the file stream.

int ungetch (char character);

Function Pushes a value back into the keyboard buffer.

Include File <stdio.h>

character (in): Value to put back into the keyboard buffer

Example

result = ungetch (letter);

Note The ungetch routine returns the value put back into the keyboard buffer.

*void unixtodos (long unixtime, struct date *date_ptr, struct time *time_ptr);*

Function Converts a UNIX time to DOS format.

Include File <dos.h>

unixtime (in):	Date and time in UNIX format
date_ptr (out):	Structure containing the DOS format date
time_ptr (out):	Structure containing the DOS format time

Example

```
unixtodos (unixdatetime, &dosdate, &dostime);
```

*int unlink (char *DOSfilename);*

Function Deletes the specificed DOS file name.

Include File <dos.h>

DOSfilename (in):	DOS file to delete

Example

```
result = unlink ("TEST.BAK");
```

Note The unlink routine returns 0 if successful and −1 if an error occurs.

int unlock (int filehandle, long offset, long num_bytes);

Function Releases a file-sharing lock previously set by lock.

Include File <dos.h>

filehandle (in):	File handle of desired file
offset (in):	Location of the first byte to unlock
num_bytes (in):	Number of bytes to unlock

Example

```
result = unlock (filehandle, 1024, 255);
```

Note The unlock routine returns 0 if successful and −1 if an error occurs.

usertype va_arg (va_list param, usertype);

Function Returns the next argument in a variable argument list.

Include File <stdarg.h>

param (in):	Variable-length argument list
type (in):	Data type of the values in the argument list

Example

param = va_arg (arg_list, int);

Note The va_arg routine returns NULL after the last parameter in the list.

void va_end (va_list parameter);

Function Marks the end of a variable argument list.

Include File <stdarg.h>

parameter (in): Variable argument list

Example

va_end (arg_list);

void va_start (va_list parameter);

Function Marks the start of a variable argument list.

Include File <stdarg.h>

parameter (in): Variable argument list

Example

va_start (arg_list);

int vfprintf (FILE *stream, char format_sequence, va_list arglist);

Function Outputs formatted data to an output stream.

Include File <stdarg.h>

stream (in):	File stream to output to
format_sequence (in):	Control characters that specify how the output is to be formatted
arglist (in):	Variable-length argument list to be output

Example

result = vfprintf (fp, "%d %c", arg_list);

Note The vfprintf routine returns the number of bytes output.

int vfscanf (FILE *stream, char format_sequence, va_list arglist);

Function Inputs formatted data from an input stream.

Include File <stdarg.h>

stream (in):	File stream to input from
format_sequence (in):	Control characters that specify how the output is to be formatted
arglist (in):	Variable-length argument list to be output

Example

result = vfscanf (fp, "%d %c", arg_list);

Note The vfscanf routine returns the number of fields filled.

int vprintf (char format_sequence, va_list arglist);

Function Outputs formatted data to stdout.

Include File <stdarg.h>

format_sequence (in):	Control characters that specify how the output is to be formatted
arglist (in):	Variable-length argument list to be output

Example

result = vprintf ("%d %c", arg_list);

Note The vprintf routine returns the number of bytes output.

int vscanf (char format_sequence, va_list arglist);

Function Inputs formatted data from stdin.

Include File <stdarg.h>

format_sequence (in):	Control characters that specify how the output is to be formatted
arglist (in):	Variable-length argument list to be output

Example

result = vscanf ("%d %c", arg_list);

Note The vscanf routine returns the number of fields filled.

*int vsprintf (char *string, char format_sequence, va_list arglist);*

Function Outputs formatted data to a character string.

Include File <stdarg.h>

string (in):	Character string to output data to
format_sequence (in):	Control characters that specify how the output is to be formatted
arglist (in):	Variable-length argument list to be output

Example

result = vsprintf (str, "%d %c", arg_list);

Note The vsprintf routine returns the number of bytes output.

*int vsscanf (char *string, char format_sequence, va_list arglist);*

Function Inputs formatted data from a character string.

Include File <stdarg.h>

string (in):	Character string to input data from
format_sequence (in):	Control characters that specify how the output is to be formatted
arglist (in):	Variable-length argument list to be output

Example

```
result = vsscanf (str, "%d %c", arg_list);
```

Note The vsscanf routine returns the number of fields filled.

int wherex (void);

Function Returns the x coordinate of the cursor within the current window.

Include File <conio.h>

Example

```
x_loc = wherex ();
```

int wherey (void);

Function Returns the y coordinate of the cursor within the current window.

Include File <conio.h>

Example

y_loc = wherey ();

void window (int left_corner, int top_corner, int right_corner, int bottom_corner);

Function Defines the active text mode window.

Include File <conio.h>

left_corner (in):	Upper-left corner x window coordinate
top_corner (in):	Upper-left corner y window coordinate
right_corner (in):	Lower-right corner x window coordinate
bottom_corner (in):	Lower-right corner y window coordinate

Example

window (1, 1, 80, 25);

*int _write (filehandle, void *buffer, int num_bytes);*

Function Writes the specified number of bytes to the file associated with the given file handle.

Include File <io.h>

filehandle (in):	File handle associated with the desired file
buffer (in):	Buffer containing the data to output
num_bytes (in):	Number of bytes to write to the file

Example

bytes_wtn = _write (filehandle, buffer, sizeof(buffer));

Note If successful, _write returns the number of bytes written; otherwise, _write returns the value −1. The maximum number of bytes this routine can write is 65,534.

*int write (filehandle, void *buffer, int num_bytes);*

Function Writes the specified number of bytes to the file associated with the given file handle.

Include File <io.h>

filehandle (in):	File handle associated with the desired file
buffer (in):	Buffer containing the data to output
num_bytes (in):	Number of bytes to write to the file

Example

bytes_wtn = write (filehandle, buffer, sizeof(buffer));

Note If successful, write returns the number of bytes written; otherwise, it returns the value −1. The maximum number of bytes this routine can write is 65,534.

Trademarks

IBM®	International Business Machines Corporation
MS-DOS®	Microsoft Corporation
Turbo C®	Borland International, Inc.

Index

D

E

F

The manuscript for this book was prepared and submitted to Osborne/McGraw-Hill in electronic form. The acquisitions editor for this project was Cynthia Hudson, the technical reviewer was Charles P. Jazdzewski, and the project editor was Fran Haselsteiner.

Cover art by Bay Graphics Design Associates. Color separation by Colour Image. Cover supplier, Phoenix Color Corp. Book printed and bound by R.R. Donnelley & Sons Company, Crawfordsville, Indiana.

TURBO C®
PROGRAMMER'S LIBRARY

Whether you're just learning to use Borland's Turbo C® or you're developing professional programs, the **Turbo C® Programmer's Library** is a must for you.

Kris Jamsa, author of the bestselling **Turbo Pascal® Programmer's Library,** shows you how to develop a powerful library of hundreds of Turbo C® routines, and reveals the tricks you need to master the Turbo C run-time library. By examining the code in this text you'll gain tremendous insights into the development of the run-time library, which, in turn, will help you fully exploit the library's routines.

In **Turbo C® Programmer's Library,** you'll find coverage of

- String manipulation
- Pointers
- Recursion
- The DOS and BIOS services
- Array manipulation
- Searching and sorting
- Dynamic list manipulation
- Memory mapped I/O
- Pop-up menu processing

The best way to develop good code is to see good code, and **Turbo C® Programmer's Library** packs more good code and techniques than any other Turbo C text on the market.

Kris Jamsa is the the author of *Turbo Pascal® Programmer's Library,* now in its second edition, and Osborne/McGraw-Hill bestsellers *Using OS/2™, OS/2™: The Pocket Reference, DOS: The Complete Reference, DOS: The Pocket Reference, DOS: Power User's Guide, Turbo Pascal® 4: The Pocket Reference,* and *The C Library.* He holds a B.S. degree in computer science from the United States Air Force Academy, and an M.S. degree in computer science from the University of Nevada at Las Vegas. He is currently a VAX/VMS systems manager for the United States Air Force.

ISBN 0-07-881394-8

$22.95

BORLAND·OSBORNE McGraw-Hill