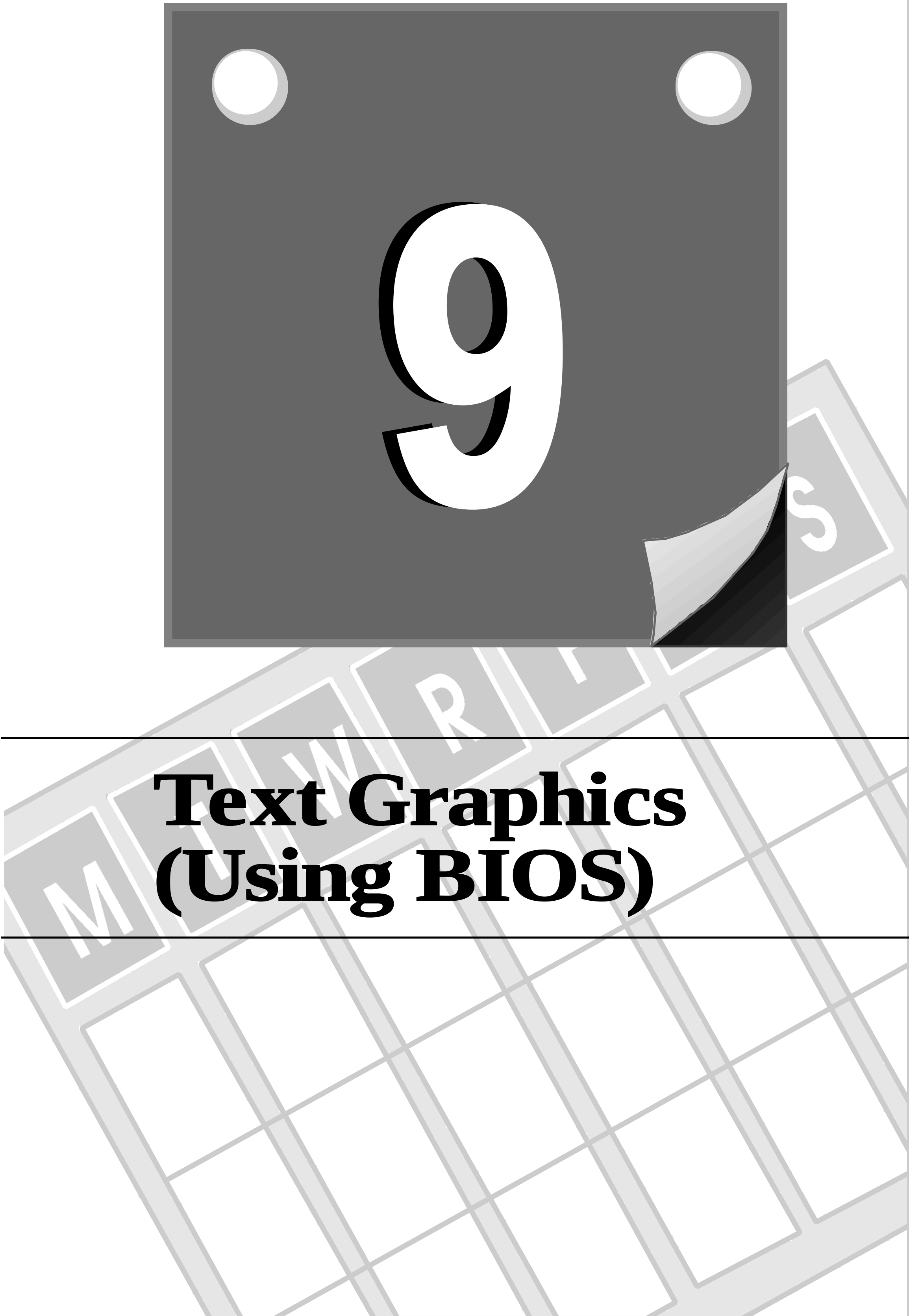


9

Text Graphics (Using BIOS)

WEEK
2

On Day 8, you were introduced to BIOS functions, which will be useful in the programs you will develop. Day 8, however, barely scratched the surface of the functions you'll need when you create a full-scale application later in this book. Many of the functions you'll need relate to how you set up your screen and text graphics. Today you will learn:

- ☐ What is meant by text graphics versus non-text graphics.
- ☐ How to combine Day 8's functions in a more organized manner.
- ☐ How to enhance your cursor functions from Day 8.
- ☐ How to create several useful screen and text functions to add to your library.
- ☐ How to add several new functions to your TYAC library.

Different Levels of Graphics

The level of graphics that can be used on a computer vary. Generally, graphics are broken down according to their levels of complexity. The complexity of each level is directly proportional to the portability of the system created using them. The different levels include:

- ☐ Monochrome text
- ☐ Colored text
- ☐ Pixel graphics
- ☐ Character graphics

Monochrome text is the least graphical and the most portable. This uses the characters that are in the ASCII chart with values from 0 to 127 or a similar set of characters. Because color is not considered and no characters other than those in the table are used, this monochrome text is sometimes not considered graphical.

Colored text is a little more graphical and less portable because not all systems support color. Colored text still uses the same character set as the monochrome text; however, the text can be displayed in a variety of colors. Virtually all personal computer systems today support colored text. Even some monochrome (single-colored) monitors support colored text by using gray-scale. Monochrome monitors that support gray-scale display colors in different shades or intensities of a single color. Most monochrome notebook computers support gray-scales of 16, 64, or 256 colors. You should understand, however, that there are some monochrome monitors that only support a single color.

Pixel graphics use the individual pixels—or dots—within a computer monitor. Because they work at a pixel level, any character can be displayed. The resolution, or clarity, of a displayed character depends on the computer, number of pixels in the computer's screen, and the number of colors that the screen can support. This includes CGA, EGA, VGA, Super VGA and more. Because the level of graphics that can be done depends on the computer system, this is much less portable than the other graphics methods. What is gained is the displayed graphics are not limited to the ASCII character set. Instead, any picture or graphic can be displayed.

Note: Most computer systems today support pixel graphics at VGA level resolutions (640×480 pixels).

Note: If you run a program with a higher graphics resolution and a monitor that supports only a lower graphics resolution, you can get unpredictable results. For example, if a program that supports VGA-level pixel graphics is run on a monitor that only supports colored text, the outcome may be a blank screen. Worse, the output may be garbled colors on the screen.

Character Graphics

Character graphics are similar to the color text. However, instead of being limited to the ASCII characters 0 to 127, character graphics also include the characters from 128 to 255. These additional characters are referred to as the *extended character set*. This extended character set provides many additional characters that can help give an application a more graphical look and feel without losing the portability of going to full pixel graphics. Appendix B shows the entire ASCII character set. Those characters with the values from 128 to 255 are considered the extended ASCII character set.

By using many of the characters provided in the extended set, you can create lines, boxes, grids, and more. In addition, character graphics include color. The number of colors used is generally left at the same level as the lowest pixel graphics—CGA. Figure 9.1 shows some of the character graphics in use.

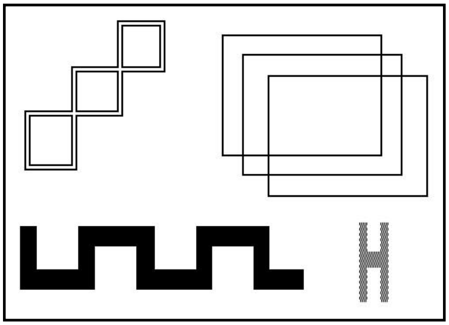


Figure 9.1. *Character graphics in use.*

> **Note:** Most DOS-based applications use character graphics. Packages that use character graphics are WordPerfect 5.2 for DOS, Paradox for DOS, DBASE IV, and Lotus 123 for DOS. Most of these packages also allow for some pixel graphics.

Organizing Your Functions

Starting on Day 12, this book will begin guiding you through the development of a complete application. In developing this application, you'll use several of the functions that you have developed on earlier days. In addition, you'll also use many new functions. Most of the functions will be developed over the next few days. Today, you'll develop several functions, including a line function and a box function, that use the character graphics.

Before learning new functions, you should ensure that your current functions are organized. In one of the exercises on Day 8, you created a library containing your functions and a header file. The header file should contain function prototypes for your functions along with any additional information that may be needed by your library's functions. Listing 9.1 is an updated version of the TYAC.H header file that you created on Day 8.

Type **Listing 9.1. The enhanced TYAC.H header file.**

```
/* Program: TYAC.H
 *          (Teach Yourself Advanced C)
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Header file for TYAC library functions
 *=======================================================*/

#ifndef _TYAC_H_
#define _TYAC_H_

/* DOS and BIOS Interrupts */
#define BIOS_VIDEO       0x10
#define DOS_FUNCTION     0x21

/* BIOS function calls */
#define SET_VIDEO        0x00
#define SET_CURSOR_SIZE  0x01
#define SET_CURSOR_POS   0x02
#define GET_CURSOR_INFO  0x03
#define WRITE_CHAR       0x09
#define SET_COLOR        0x0B
#define GET_VIDEO        0x0F
#define WRITE_STRING     0x13

/*  Types of Boxes  */
#define DOUBLE_BOX       2
#define SINGLE_BOX       1
#define BLANK_BOX        0

/*  Box fill flags  */
#define BORDER_ONLY      0
#define FILL_BOX         1

/* Colors */
#define   BLACK          0
#define   BLUE           1
#define   GREEN          2
#define   CYAN           3
#define   RED            4
#define   MAGENTA        5
#define   BROWN          6
#define   WHITE          7
#define   GRAY           8
#define   LIGHTBLUE      9
#define   LIGHTGREEN     10
#define   LIGHTCYAN      11
#define   LIGHTRED       12
#define   LIGHTMAGENTA   13
#define   YELLOW         14
#define   BRIGHTWHITE    15

```

continues

Listing 9.1. continued

```
52: /* used to set scrolling direction */
53: #define SCROLL_UP    0x07
54: #define SCROLL_DOWN  0x06
55:
56: /*------------------------*
57:     Function Prototypes
58:  *------------------------*/
59:
60:      /* Gets the current date */
61: void current_date(int *, int *, int *);
62:
63:      /* Positions the cursor to row/col */
64: void cursor(int, int);
65:      /* Returns info about cursor */
66: void get_cursor(int *, int *, int *, int *, int *);
67:      /* Sets the size of the cursor */
68: void set_cursor_size(int, int);
69:
70:      /* clear the keyboard buffer */
71: void kbclear( void );
72:      /* determine keyboard hit */
73: int  kbhit( void );
74:
75:      /* scroll the screen */
76: void scroll( int row,   int col,
77:              int width, int height,
78:              int nbr,   int direction);
79:
80: #endif
```

Analysis As you can see, this header file contains a function prototype in lines 58 to 74 for each of the functions that should be in your library. By including this header file in any of the programs you are using, you'll be sure to have all the prototypes that you need for the library functions. You'll also be able to keep any other information in this header file that is needed by your functions.

This version of the TYAC.H header file includes several other items. Lines 11 and 12 include defined constants for BIOS interrupt numbers. Lines 14 to 22 include defined constants for BIOS function numbers. These defined constants will be used in many of the functions created today. By using the defined constants instead of the actual numbers, your individual functions will be more readable.

Lines 28 to 30 declare a few additional defined constants, which will be used in creating a border on a box function later today. Lines 32 to 48 define the basic colors. This is similar to the defined constants that were created for the ANSI functions on

Day 4. A final set of defined constants are declared in lines 51 and 52. These were used in the control break function presented in the Day 8 exercises.

The TYAC.H header file will by dynamic. As you create each of your new functions, you should add the prototype into TYAC.H. By doing this and by adding each function to your TYAC.LIB library, you'll ensure that you'll have a complete library to use.

Tip: You should give your library the same name as the header file—TYAC. This will help you keep the two related together. (TYAC stands for Teach Yourself Advanced C.)

Note: The TYAC.LIB library and TYAC.H header file will be used from this point on. Each function you create on the following days should be added to your library. In addition, you should add the new function prototypes to TYAC.H.

Enhancing Your Cursor Functions

The cursor functions on Day 8 contained only the basic row and column parameters. The BIOS calls that work with the cursor will provide much more information than just the row and column of the cursor. While you may not always need the additional information, it's better to go ahead and make your functions a little more functional. The `get_cursor()` and `cursor()` functions presented originally in Listings 8.3 and 8.4 should be enhanced. Listings 9.2 and 9.3 contain new cursor functions.

Type

Listing 9.2. PCURSOR.C. A function to put the cursor on screen.

```
/* Program: PCURSOR.C
 * Authors: Bradley L. Jones
```

continues

Listing 9.2. continued

```
 *          Gregory L. Guntle
 * Purpose: Demonstrates a BIOS function to position the
 *          cursor on the screen.
 * Note:    This function places the cursor at a given
 *          location on the screen. The upper left position
 *          of the screen is considered (0,0) not (1,1)
 *=======================================================*/

#include <dos.h>
#include "tyac.h"

void cursor(int row, int column)
{
    union REGS inregs;

    inregs.h.ah = SET_CURSOR_POS;
    inregs.h.bh = 0;
    inregs.h.dh = row;
    inregs.h.dl = column;
    int86(BIOS_VIDEO, &inregs, &inregs);
}
```

Listing 9.3. GCURSOR.C. A function to get cursor from screen.

```
/* Program: GCURSOR.C
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Demonstrates a BIOS function to get the position
 *          of the cursor on the screen.
 * Note:    This function considers the upper left position
 *          of the screen to be (0,0) not (1,1)
 *=======================================================*/

#include <dos.h>
#include "tyac.h"

void get_cursor(int *row, int *column, int *page, int *start,
                int *end)
{
    union REGS inregs, outregs;

    inregs.h.ah = GET_CURSOR_INFO;
    inregs.h.bh = 0;
    int86(BIOS_VIDEO, &inregs, &outregs);
    *row    = (int) outregs.h.dh;
21:      *column = (int) outregs.h.dl;
22:      *page   = (int) outregs.h.bh;
23:      *start  = (int) outregs.h.ch;
24:      *end    = (int) outregs.h.cl;
25:  }
```

Analysis The PCURSOR.C listing operates exactly as it did before. Only minor cosmetic changes have been made. Notice that in line 12, the TYAC.H header file is now being included. This enables you to use any of the defined constants in this function. In addition, it ensures that the function's prototype is included. In lines 18 and 22, two defined constants are used from the TYAC.H header file: `GET_CURSOR_INFO` and `BIOS_VIDEO`. This makes the `cursor()` function's code a little easier to read.

The GCURSOR function in Listing 9.3 is different from the function presented on Day 8. Like the `cursor()` function in Listing 9.2, the TYAC.H header file has been included, and the defined constants have been used. In line 13, you'll notice a much larger change. The `get_cursor()` function now has three additional parameters, `page`, `start`, and `end`. These three parameters give the function the capability to pass additional information back to the calling program. Now when you use `get_cursor()`, you won't only get the row and column location of the cursor, but you'll also get the video page that the cursor is on and the start and end scan lines that make up the cursor's shape. (See the `set_cursor_size()` in Day 8 for information on the cursor.) This additional information makes the function much more useful.

Warning: When you change a function, you need to recompile it and update your library. If you change the parameters passed to a function, you need to change the prototype within the header file also. In addition, if you change a header file, you may need to recompile all the programs that use the header file to ensure that you don't cause problems in your other functions.

Note: An explanation of a video page will be covered later today.

You might be wondering why Day 8 did not present the `get_cursor()` functions with the additional parameters. You'll generally write a function to serve a specific need.

As time goes on, you'll find that you need to update or enhance the function. The cursor functions are a prime example of such changes. You should update your library and header file with these newer versions.

DO **DON'T**

DO continue to add your functions to (or update) your TYAC.LIB library.

DON'T forget to update your library and header file when you change a function.

DO use defined constants to make your programs and functions easier to read—and debug.

Creating New Functions

In the following sections, several new functions will be presented that will help you create applications. You'll learn a few functions that help in your use of text graphics and a function that enables you to pause until a key is entered.

Text Graphics Functions

Several text graphics functions will be presented. Many of these functions can be used with the functions you have created on previous days. These include the following:

- ☐ Setting the video mode.
- ☐ Getting the video mode.
- ☐ Setting the border color.
- ☐ Writing a character.
- ☐ Writing a character multiple times.
- ☐ Drawing a line.
- ☐ Drawing a box.
- ☐ Drawing a box with borders.

You'll want to add several of these functions to your TYAC library.

Working with the Video Mode

When you begin to work with character graphics and character graphic functions, you must have a little background information. Most computer monitors display 25 rows, each with 80 characters; however, you should never assume this. Two functions will be extremely useful when using character graphics. These are a function to set the video mode and to get the video mode (Listings 9.4 and 9.5).

Type

Listing 9.4. GVIDEO.C gets the video mode.

```
/* Program: GVIDEO.C
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Demonstrates a BIOS function which gets the
 *          current video mode.
 *=======================================================
 *
 *  Display modes (partial list):
 *
 *      0  40 by 25 black and white text
 *      1  40 by 25 16-color text
 *      2  80 by 25 black and white text
 *      3  80 by 25 16-color
 *
 *      4  320 by 200 4-color
 *      5  320 by 200 4-color
 *      6  640 by 200 2 color
 *
 *      7  80 column  mono text
 *
 *     64  80 by 43 (EGA)
 *         80 by 50 (VGA)
 *==========================================================*/

#include <dos.h>
#include “tyac.h”

void get_video(int *columns, int *display_mode, int *display_page)
{
     union REGS inregs, outregs;

     inregs.h.ah = GET_VIDEO;
     int86(BIOS_VIDEO, &inregs, &outregs);
     *columns = (int) outregs.h.ah;
     *display_mode = (int) outregs.h.al;
     *display_page = (int) outregs.h.bh;
}
```

This program gets information on the video mode. This will fill in the values of three different variables that have pointers passed on to the `get_video()` function. The first parameter, `column`, will be filled in with the number of columns available. The second parameter, `display_mode`, will be filled in with a numeric value. The value will signal which mode the video is set to. The comments in lines 8 to 22 detail the different values that the video mode could be. The third parameter, `display_page`, will be filled with the display page number. There can be more than one video page; however, only one can be active at a time. *Video pages* are areas reserved to set up screen information; however, they are not visible until they are made active. Most programs ignore video paging and simply use the current page. As shown earlier, one of the additional parameters for the cursor functions was for the video page number.

Lines 30 to 36 contain the bulk of this program. As you can see, the `get_video()` function looks like many of the other BIOS functions. There has been one subtle change. Instead of using an interrupt and a function number, defined constants are used. The `GET_VIDEO` and `BIOS_VIDEO` constants should make this function easier to understand.

The ability to get the video mode is important; however, sometimes you'll want to set it. The `set_video()` function does just that.

Listing 9.5. SVIDEO.C sets the video mode.

```
/* Program: SVIDEO.C
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Demonstrates a BIOS function to set the video
 *          mode.
 *=======================================================*
 *
 *  Display modes (partial list):
 *
 *      0  40 by 25 black and white text
 *      1  40 by 25 16-color text
 *      2  80 by 25 black and white text
 *      3  80 by 25 16-color
 *
 *      4  320 by 200 4-color
 *      5  320 by 200 4-color
 *      6  640 by 200 2 color
 *
 *      7  80 column  mono text
 *
 *     64  80 by 43 (EGA)
 *          80 by 50 (VGA)
 *============================================================*/

#include <dos.h>
#include “tyac.h”

void set_video(int display_mode)
{
     union REGS inregs;

     inregs.h.ah = SET_VIDEO;
     inregs.h.al = display_mode;
     int86(BIOS_VIDEO, &inregs, &inregs);
}
```

Analysis The `set_video()` function accepts a numeric parameter for the mode. These are the same numbers that are returned by the `get_video()` function. You should notice that this function attempts to set the mode even if an invalid mode is passed. By not editing the display mode that is to be set, this function will be capable of setting additional modes that may be supported in the future. You may want to consider adding logic to prevent any invalid display modes from being set. In an exercise at the end of the day, you’ll be asked to use these functions to write a program.

Setting the Border Color

Once you are able to work with the video mode, you are ready to forge into text graphics and color functions. Most of the functions work on the screen; however, there is also a border to the screen that can be colored. Listing 9.6 contains a function to set the border color.

Type **Listing 9.6. SBRDCLR.C sets the border color.**

```
/* Program: SBRDCLR.C
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Sets the border color.
 *=======================================================*/

#include <dos.h>
#include “tyac.h”

void set_border_color(int color)
{
    union REGS inregs;

```

continues

Listing 9.6. continued

```
14:      inregs.h.ah = SET_COLOR;   /* Set Color Palette */
15:      inregs.h.bh = 0;           /* BL contains background and border
                                       color */
16:      inregs.h.bl = color;       /* New color */
17:      int86(BIOS_VIDEO, &inregs, &inregs);
18:  }
```

Analysis

This listing contains the `set_border_color()` function, which sets the color of the border to the value passed in `color`. This color should be one of the values contained in TYAC.H that is included in line 8. As you can determine from this listing, the border color is set using the `BIOS_VIDEO` interrupt (0×10) and the `SET_COLOR` function (0×0B). In addition, the `bh` register needs to be set to 0. The actual color number is set in line 16 to the `bl` register.

Warning: The `set_border_color()` function in Listing 9.5 does not contain any error trapping. You may wish to add edit checks limiting the acceptable values to this function. You should consider adding error trapping to all the functions that are presented.

Writing a Character in Color

While setting the border's color is fun, it's often not done. What is often done is writing a character to the screen. Although the ANSI `putchar()` function does a splendid job of putting a character on the screen, it doesn't do it in color. Listing 9.7 presents a new function for writing a character in color.

Type

Listing 9.7. WRITECH.C. A function to write a character in color.

```
/* Program: WRITECH.C
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Writes a character at a the current cursor
 *          location.
 *=======================================================*/

#include <dos.h>
#include “tyac.h”
```

```
10:
11:  void write_char(char ch, int fcolor, int bcolor)
12:  {
13:      union REGS inregs;
14:
15:      inregs.h.ah = WRITE_CHAR;
16:      inregs.h.al = ch;             /* Character to write */
17:      inregs.h.bh = 0;              /* Display page goes here */
18:      inregs.h.bl = (bcolor << 4 ) | fcolor;
19:      inregs.x.cx = 1;
20:      int86(BIOS_VIDEO, &inregs, &inregs);
21:  }
```

Analysis This function writes a character at the cursor's current location. For this reason, the function is good to use in conjunction with the `cursor()` function learned earlier. As you can see by line 11, the function takes three parameters. The first, `ch`, is the character to be printed. The second two are the foreground color, `fcolor`, and the background color, `bcolor`. These are used in line 18 to set the color of the character to be printed.

This setting of the color might seem confusing, but it works. Because the numbers for the colors are small, they don't require an entire byte for each of the background and foreground colors. Instead, the background color is placed in the high order bits of an individual character (the top half). The foreground colors are stored in the lower bits (or the bottom half). To accomplish this, the background color, `bcolor`, is shifted four positions and then "OR"ed into the same register as the foreground color. This isn't an uncommon practice for setting a foreground/background color combination into a single field.

A few other registers are also set in this function. The first is the `ah` register, which is set to function `WRITE_CHAR` (0×09 defined in your TYAC.H header file). The `x.cx` register is set to 1, the `bh` register to 0, and the `al` register to `ch`, which is the character that is going to be printed. (The `cx` register will be covered in the next section.) The `bh` register is set to the video page number. We are assuming zero here; however, if you decide to work with video paging, this function can be modified to assign the video page to the `bh` register. In line 20, writing the character is accomplished by passing these register values with an interrupt 0×10—the defined value of `BIOS_VIDEO` in your TYAC.H file.

Note: The TYAC.H header file presented in Listing 9.1 contained the colors that are available. The foreground colors can be any one of the 16 colors presented. The background colors are only the first eight.

Repeating a Character

Often you'll want to print a character several times. For example, to draw a line of asterisks, you could call the `write_char()` several times in a row. Or alternatively, you could create a function to do this for you. Listing 9.8 shows a function that you may think accomplishes the task of printing a character a given number of times.

Type

Listing 9.8. Using a `for` loop to repeat a character.

```
/* Program: LIST0907.C
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Writes a character several times.
 *=======================================================*/

#include “tyac.h”

void repeat_char(char ch, int howmany, int fcolor, int bcolor)
{
    int ctr;

    for( ctr = 0; ctr < howmany; ctr++ )
    {
      write_char( ch, fcolor, bcolor );
    }
}
```

Analysis

As you can see, this is a straightforward function. It simply calls the `write_char()` function the number of times requested. This function doesn't work the way you might expect. It writes the character the number of times stated in `howmany`; however, it will write them on top of each other! The `write_char()` function has no control of the cursor. You have to move the cursor yourself. Each time you write a character, you need to move the cursor over one column. In addition, you have to determine where the cursor originally was to know the values to increment.

While you could get a function like this to work, there is an easier—and better—alternative. Listing 9.9 presents a function that looks virtually identical to the `write_char()` function presented in Listing 9.8; however, there are a few subtle differences.

Type

Listing 9.9. REPEATCH.C. A better repeating character function.

```
1:   /* Program: REPEATCH.C
2:    * Authors: Bradley L. Jones
3:    *          Gregory L. Guntle
4:    * Purpose: Repeats a character starting at a the current
5:    *          cursor location.
6:    *=======================================================*/
7:
8:   #include <dos.h>
9:   #include “tyac.h”
10:
11:  void repeat_char(char ch, int howmany, int fcolor, int bcolor)
12:  {
13:      union REGS inregs;
14:
15:      inregs.h.ah = WRITE_CHAR;
16:      inregs.h.al = ch;             /* Character to write */
17:      inregs.h.bh = 0;              /* Display page goes here */
18:      inregs.h.bl = (bcolor << 4 ) | fcolor;
19:      inregs.x.cx = howmany;        /* Nbr of times to display */
20:      int86(BIOS_VIDEO, &inregs, &inregs);
21:  }
```

Analysis

The first difference you'll notice is that the function is called `repeat_char()` instead of `write_char()`. In line 11, you can see that an additional parameter is passed to the function. This parameter, `howmany`, specifies the number of times the character is to be repeated. In line 19, the value in `howmany` is assigned to the `x.cx` register. For the `write_char()` function, the number 1 was assigned to the `cx` register. This is because when you write a character, you are only repeating the character one time.

With these minor differences, the `repeat_char()` function is complete. You may be wondering why the `write_char()` function isn't eliminated and the `repeat_char()` function always used. If you call `repeat_char()` and pass the value of 1 in the `howmany` field, then you are essentially accomplishing the `write_char()` function. The basic reasons for having separate functions are readability and usability. When you write a character in color, generally you don't think of repeating it.

Drawing a Line

You already know how to draw a line with character graphics even though you may not be aware of it. A line is simply a set of repeated line characters. There are several line characters in the extended ASCII character set. These include characters 179 and 186 for drawing vertical lines and characters, and 196 and 204 for drawing horizontal lines. Listing 9.10 demonstrates the use of the `repeat_char()` function to print not only a line of asterisks, but also text lines using some of the extended ASCII characters.

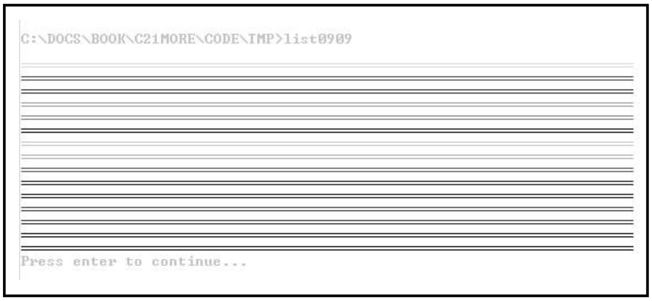
Note: When you compile this program, you should link in your TYAC library. You should add each new function you learn to your TYAC library. Many of the remaining programs (listings that create executable files) assume that you are linking the TYAC library. In addition, they assume that you have updated it with any new functions.

Listing 9.10. Using the `repeat_char()` function and drawing lines.

```
/* Program: LIST0909.C
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Use the repeat_char() function.
 *=======================================================*/

#include <stdio.h>
#include “tyac.h”

void pause(char *message);

int main(void)
{
   int fcolor, bcolor;

   for( bcolor = 0; bcolor < 8;  bcolor++ )
   {
      for( fcolor = 0; fcolor < 16; fcolor++ )
      {
         cursor( fcolor+2 , 0 );
         repeat_char( 205, 70, fcolor, bcolor );
      }
      pause(“Press enter to continue...”);
   }
   return 0;
}
```

```
27:
28:  void pause(char *message)
29:  {
30:     printf("\n%s", message);
31:     while ((getchar()) != '\n') { }
32:     fflush(stdin);
33:  }
```

Output



```
C:\DOCS\BOOK\C21MORE\CODE\TMP>list0909

════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
════════════════════════════════════════════════════════════════════════════════
Press enter to continue...
```

Note: The output will be in color; however, it isn't shown here.

Analysis

This program is fun because you are working with color. As you can see, Listing 9.10 does a little more than print a bunch of double lines. As usual, line 8 includes the TYAC.H header file. The TYAC.H header file should contain all the function prototypes for the new functions that you are creating. In addition, the TYAC.LIB file should contain all the functions. If you get a link error for `cursor()`, it could be that you didn't add this function to your library on Day 8.

This program does a lot for the small amount of code presented. It presents the available color combinations by using two `for` loops to cycle through the foreground and background colors. In addition, double lines are drawn across the screen. Line 21 draws the double lines by repeating character 205, which is a double line. If you wanted single lines instead, you could have used character 196. For fun, try using character 179. This presents vertical, multicolored lines.

The `pause()` function is used in line 23 to provide a momentary break between each background color. Each time you press the enter key, the screen is redrawn with the next background color. The code for the pause function is in lines 28 to 33. It will be covered at the end of today.

Writing a String in Color

Writing a string in color can be accomplished in several different ways. The simplest way is to break it down into individual characters and print each one. There is an alternate way that won't be shown in this book. It involves using an extended set of registers that aren't always available. Using the `write_char()` function makes writing a string in color much easier. Listing 9.11 presents the `write_string()` function.

Type

Listing 9.11. WRITESTR.C writing a string in color.

```
1:   /* Program: WRITESTR.C
2:    * Authors: Bradley L. Jones
3:    *          Gregory L. Guntle
4:    * Purpose: Write string to the screen
5:    *          Uses   write_char
6:    *===========================================================*/
7:
8:   #include <string.h>   /* for strlen() */
9:   #include “tyac.h”
10:
11:  void write_string(char *string, int fcolor, int bcolor, int row,
                       int col)
12:  {
13:      int len = strlen(string);
14:      int i;
15:
16:      for (i=0; i < len; i++)
17:      {
18:         cursor(row, col+i);              /* Position cursor */
19:         write_char( (char)*(string+i), fcolor, bcolor);
20:      }
21:  }
```

Analysis

This isn't a perfect listing, but it is effective. As you can see, a `for` loop in lines 16 to 20 enables you to loop through each character of the string. For each iteration of the string, the cursor is placed and a character is written. This is done in line 19 with the `write_char()` function. The character passed is:

```
(char)*(string+i)
```

Don't let this confuse you. This is just the character at the `i` offset in string.

The `write_string()` function is a powerful function. It can write a string at any position on the screen. In addition, `write_string()` can write it in color. You may find this to be one of your most useful functions. Where `write_string()` has its flaw is in escape characters. In `printf()`, certain sequences of characters perform special functions. An exercise at the end of today asks you to update `write_string()` so some of the escape sequences are implemented. You should do this exercise. If you don't, you should at least update your `write_string()` function with the answer provided for the exercise.

Drawing a Box

Setting the cursor, writing characters, and manipulating color give you a lot of power in manipulating the screen. These three capabilities are what have enabled you to draw lines and write colored text. These functions also work together to help you create a box and, on later days, an entire entry screen for an application.

A box is a simple construct that can have several uses when working with output to the screen. A box can be used to create menus, screens, messages, and more. Boxes begin to get more complex when you choose to add borders to them. In addition, there can be an additional level of complexity added when you choose to have filled or unfilled boxes. Figure 9.2 presents different styles of boxes.

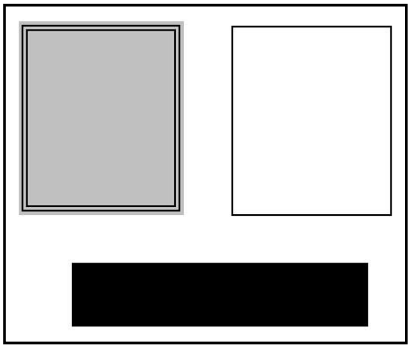


Figure 9.2. *A few examples of boxes.*

Listing 9.12 presents a function that enables you to create all of the boxes in Figure 9.2 using text graphics. Several of the added defined constants in the TYAC.H header file presented in Listing 9.1 were made specifically for the `box()` function. You should verify that these defined constants are in your TYAC.H header file. You should also add the function prototype for the box function. The defined constants that were added specifically for the box function are:

```
/*  Types of Boxes  */
#define DOUBLE_BOX      1
#define SINGLE_BOX      2
#define BLANK_BOX       3

```

```
/*  Box fill flags  */
#define BORDER_ONLY      0
#define FILL_BOX         1
```

You’ll see these constants used within the `box()` function’s listing. In addition, you may choose to use them when you use the `box()` function in your programs. Listing 9.13 is a small listing that uses the `box()` function along with several of the defined constants.

Listing 9.12. BOX.C. The `box()` function.

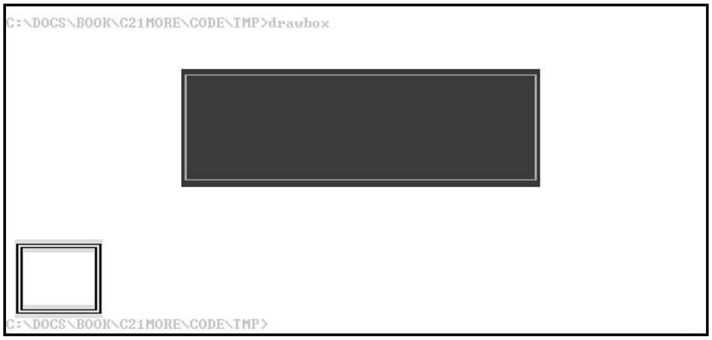
```
/* ------------------------------------------
 * Program: BOX.C
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Draws a box on the screen using other BIOS
 *          functions.
 *
 * Enter with: start_row   (0-24)
 *             end_row     (0-24)
 *             start_col   (0-79)
 *             end_col     (0-79)
 *=======================================================*/

#include “tyac.h”

void box(int start_row, int end_row,
         int start_col, int end_col,
         int box_type,  int fill_flag,
         int fcolor,    int bcolor )
{
   int row;

   /* BOX CHARACTERS */
   static unsigned char DBL_BOX[6] = “…═ª∫»º” ; /* Double-sided box
                                                  characters */
   static unsigned char SGL_BOX[6] = “⁄ƒ™╙Ÿ” ; /* Single-sided box
                                                  characters */
   static unsigned char BLK_BOX[6] = “      “ ; /* Spaces for
                                                  erasing a box */
   static unsigned char *USE_BOX;              /* Set this vari
                                                  able to the
                                                  appropriate box
                                                  characters to
                                                  use */

   /* Determine BOX Type to Draw */
   switch (box_type)
   {
     case DOUBLE_BOX:    USE_BOX = DBL_BOX;
                         break;
     case SINGLE_BOX:    USE_BOX = SGL_BOX;
                         break;
     case BLANK_BOX:     USE_BOX = BLK_BOX;
                         break;
     default:            USE_BOX = DBL_BOX;
                         break;
   }

   /* Draw the top two corner characters */
   cursor(start_row, start_col);
   write_char(USE_BOX[0], fcolor,bcolor);
   cursor(start_row, end_col);
   write_char(USE_BOX[2], fcolor,bcolor);

   /* Draw the top line */
   cursor(start_row, start_col+1);
   repeat_char(USE_BOX[1],end_col-start_col-1,fcolor,bcolor);

   /* Draw the bottom line */
   cursor(end_row,start_col+1);
   repeat_char(USE_BOX[1],end_col-start_col-1,fcolor,bcolor);

   /* Draw the sides */
   for (row=start_row+1; row < end_row; row++ )
   {
     cursor(row,start_col);
     write_char(USE_BOX[3],fcolor,bcolor);
     cursor(row,end_col);
     write_char(USE_BOX[3],fcolor,bcolor);
   }

   /* Draw the bottom corner pieces */
   cursor(end_row, start_col);
   write_char(USE_BOX[4],fcolor,bcolor);
   cursor(end_row,end_col);
   write_char(USE_BOX[5],fcolor,bcolor);

   /* fill box */

   if(fill_flag != BORDER_ONLY)
   {
      for( row= start_row+1; row < end_row; row++ )
      {
         cursor( row, start_col+1 );
         repeat_char( ‘ ‘, ((end_col-start_col)-1), fcolor,
                      bcolor);
      }
   }
}  /* end of BOX */
```

Type

Listing 9.13. DRAWBOX.C. Using the `box()` function to draw boxes.

```
1:   /* Program: DRAWBOX.C
2:    * Authors: Bradley L. Jones
3:    *          Gregory L. Guntle
4:    * Purpose: Draws boxes on the screen using BIOS
5:    *=======================================================*/
6:
7:   #include “tyac.h”
8:
9:   int main()
10:  {
11:     int start_row = 5;
12:     int start_col = 20;
13:     int end_row = 13;
14:     int end_col = 60;
15:
16:     box(start_row, end_row, start_col, end_col,
17:         SINGLE_BOX, FILL_BOX, RED, CYAN );
18:
19:     box( 18, 23, 1, 10, DOUBLE_BOX, BORDER_ONLY, YELLOW, BLUE );
20:
21:     return 0;
22:  }
```

Output



```
C:\DOCS\BOOK\C21MORE\CODE\TMP>drawbox
C:\DOCS\BOOK\C21MORE\CODE\TMP>
```

Analysis

This output is from Listing 9.12. As you can see, two different boxes are created on the screen. In addition, these boxes are in color. The first box is a filled cyan box with a red border. The red border is a single solid line. The second box is not filled and has a yellow border on a blue background. In looking at Listing 9.13, this follows the two calls to `box()` in lines 16 and 19.

As you can see, the `box()` function takes eight parameters. While this may seem like a lot, by having eight parameters, the `box()` function becomes very useful. Listing 9.13

contains the box function itself. Line 16 starts the function. Here you can see the eight parameters that are received. The first two parameters are the starting row and the ending row for the box. The third and fourth parameters are the starting and ending columns. By setting these four parameters, you define the size of the box. The four variables, `start_row`, `end_row`, `start_col`, and `end_col` will be used throughout the `box()` function.

The next parameter, `box_type`, determines the type of border on the box. One of three possible borders are available in the box, `DOUBLE_BOX`, `SINGLE_BOX`, or `BLANK_BOX`. These are defined constants that are in the TYAC.H header file. Their corresponding values, two, one, or zero, could also be used. This parameter is used in line 30 to determine which borders are going to be used in the box.

The fifth parameter, `fill_flag`, also uses defined constants from the TYAC.H header file. `fill_flag` can be set to `BORDER_ONLY` or `FILL_BOX`. This variable determines whether the inside of the box will be filled. If the `fill_flag` contains `BORDER_ONLY`, the box won't be filled.

The last two parameters, `fcolor` and `bcolor`, are the colors. These are used to set the foreground and background colors of the individual box characters.

This listing needs more explanation than those previously presented. Line 16 contains the function parameters, which have already been discussed. Lines 24 to 26 contain character arrays that contain the characters used to create the border on the box. Line 27 contains a character pointer, `USE_BOX`, that is set to point at one of these three sets of border types. Lines 30 to 40 use the `box_type` parameter to set the appropriate array to the `USE_BOX` pointer. Notice that if an inappropriate value is received in `box_type`, then the default sets `DBL_BOX`, the double-lined border.

The box is drawn in lines 42 to 69. Line 43 places the cursor at the location of the top-left corner of the box. The `write_char()` function is then used to write the top-left corner character from the array that `USE_BOX` points to. The top-right character is drawn in lines 45 and 46, the bottom two corners are drawn in lines 65 to 69. The corners are connected with the edge lines in lines 52 to 63. The `repeat_char()` function is used to draw the horizontal lines. A `for` loop in lines 57 to 63 is used to draw the vertical lines. In each case, the appropriate character is used from the array being pointed at by `USE_BOX`.

Lines 73 to 80 fill the box by printing spaces inside the box border. The spaces effectively place the background color in the box. Only the inside of the box is filled because the border has already been drawn.

This function makes the assumption that the row and column values passed in are correct. If you pass a `start_row` that is greater than an `end_row`, or a `start_col` that is greater than an `end_col`, then you won't get a box, but the function will still work. Another flaw that can cause problems is a result of the row and column values. There is no check to ensure that the values are appropriate for the screen. If you pass 1000 for a row value, you'll get unpredictable results. Both of these problems could be avoided with coding additions; however, the coding additions can cause limitations in the function. As long as you are aware of the possible problems, they can be avoided easily.

Note: Writing foolproof code can be costly in the amount of time required to handle every situation. It's best to handle the problems that are most likely to occur or that will cause serious problems. The amount of time needed to add the code must be weighed against the value added by the code.

A Pausing Function

One last function is going to be presented as an added bonus. This function isn't really a text graphics function; however, it's one that you may find useful. This is the `pause()` function. It was used in Listing 9.13. It enables you to stop what is being displayed and wait for the user to press the enter key. This is a good addition to your TYAC library.

Type

Listing 9.14. PAUSE.C. A pausing function.

```
/* Program: PAUSE.C
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Causes the program to PAUSE until the ENTER
 *          key is pressed.
 * Note:    This program requires the calling routine
 *          to pass a message to display.
 *=====================================================*/
#include <stdio.h>

void pause(char *message)
{
  printf("\n%s", message);
```

```
   while ((getchar()) != '\n') { }
   fflush(stdin);
}
```

Analysis This short function receives a message that is displayed using the `printf()` function. It then uses `getchar()` to get characters until it reads a new line (enter). Once it reads an enter key, it flushes the keyboard and returns.

Updating Your TYAC Library and Header File

At this point you should ensure that your TYAC.H header file and your TYAC.LIB are both up-to-date. Exercise 1 asks you to update your library with all of today's functions. In addition, you should verify that your TYAC.H header file is similar to Listing 9.15.

Type **Listing 9.15. LIST0914.H. A new version of TYAC.H.**

```
/* Program: TYAC.H
 *          (Teach Yourself Advanced C)
 * Authors: Bradley L. Jones
 *          Gregory L. Guntle
 * Purpose: Header file for TYAC library functions
 *=======================================================*/

#ifndef _TYAC_H_
#define _TYAC_H_

/* DOS and BIOS Interrupts */
#define BIOS_VIDEO       0x10
#define DOS_FUNCTION     0x21

/* BIOS function calls */
#define SET_VIDEO        0x00
#define SET_CURSOR_SIZE  0x01
#define SET_CURSOR_POS   0x02
#define GET_CURSOR_INFO  0x03
#define WRITE_CHAR       0x09
#define SET_COLOR        0x0B
#define GET_VIDEO        0x0F
#define WRITE_STRING     0x13

/*  Types of Boxes  */
```

continues

Listing 9.15. continued

```
#define DOUBLE_BOX       1
#define SINGLE_BOX       2
#define BLANK_BOX        3

/*  Box fill flags  */
#define BORDER_ONLY      0
#define FILL_BOX         1

/* Colors */
#define   BLACK          0
#define   BLUE           1
#define   GREEN          2
#define   CYAN           3
#define   RED            4
#define   MAGENTA        5
#define   BROWN          6
#define   WHITE          7
#define   GRAY           8
#define   LIGHTBLUE      9
#define   LIGHTGREEN     10
#define   LIGHTCYAN      11
#define   LIGHTRED       12
#define   LIGHTMAGENTA   13
#define   YELLOW         14
#define   BRIGHTWHITE    15

/* used to set scrolling direction */
#define SCROLL_UP    0x07
#define SCROLL_DOWN  0x06

/*------------------------*
    Function Prototypes
 *------------------------*/

     /* Gets the current date */
void current_date(int *, int *, int *);

     /* Positions the cursor to row/col */
void cursor(int, int);
     /* Returns info about cursor */
void get_cursor(int *, int *, int *, int *, int *);
     /* Sets the size of the cursor */
void set_cursor_size(int, int);

     /* clear the keyboard buffer */
void kbclear( void );
     /* determine keyboard hit */
int  kbhit( void );

```

```
     /* sroll the screen */
void scroll( int row,   int col,
             int width, int height,
             int nbr,   int direction);

     /* pause until ENTER pressed */
void pause(char *);

     /* Video mode functions */
void set_video(int);
void get_video(int *, int *, int *);

     /* Text Graphics functions */
void write_char(char, int, int);
void repeat_char(char, int, int, int);
void write_string(char *, int, int, int, int);
void box(int, int, int, int, int, int, int, int);
void set_border_color(int);

#endif
```

DO **DON’T**

DO enter all of today’s functions and add them to your library.

DON’T over code your functions. If you spend an excessive amount of time coding edits to prevent problems, you may never be able to use the function.

DO code enough edits into your functions to avoid as many problems as is reasonable.

Summary

Today, you covered a multitude of additional functions that work with text graphics. Before jumping into these functions, you were given an overview of the different types of screen graphics. You also moved right into creating new functions. The day focused on functions that put text graphics on the screen; this included colored characters, repeating a character, drawing text lines, and creating boxes. With the functions presented in today’s material, you have the beginning building blocks for creating an application’s screens.

Q&A

Q What is the benefit of using text graphics instead of pixel graphics?

A Text graphics are much more portable and can be displayed on monitors of varying resolutions, including some monochrome monitors. In addition, text graphics are much easier to work with because you are working with a limited number of characters.

Q Can you write text graphics programs that will be portable to C++ or Windows?

A This is really two different questions. Windows programs use pixel graphics. Windows can run text graphics programs in DOS windows. In regard to portability to C++, all C constructs are portable to C++. This means that you can write text graphics characters in C++ also.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is meant by monochrome?
2. What are text graphics?
3. How many characters are in the ASCII character set?
4. What are considered to be the extended ASCII characters?
5. When you change a library function, what must you do?
6. When you change a header file, what should you do?
7. What is a reason for using defined constant?
8. What are the colors that can be used?

Exercises

1. Add all of the functions that you have created today to your TYAC.LIB library. These functions should be:

```
cursor()      (updated)
get_cursor() (updated)
get_video()
set_video()
set_border_color()
write_char()
repeat_char()
write_string()
box()
pause()
```

2. Write a program that uses the `get_video()` functions. Once the program gets the mode, it should display the values that are set.
3. Modify the program from Exercise 2 to also set the video mode to 40 columns.
4. Change the character in Listing 9.9 to character 177. What does this character do?
5. **ON YOUR OWN:** Use the functions presented today to write a program that displays information on the screen in color. Use your imagination to determine what you should display.