Bonus Lessons

Supplemental Lessons

Main
About
News
FAQ

Supplemental Lessons

Other Information
Source Code Files
On-line Book Ordering
Related Web Pages

Just for being nice, I'm presenting you with some Supplemental Lessons designed to help further your conquest of the C language. Some of these were promised in the book, others are done for the heck of it, still others are prompted as solutions to reader's questions (the Bonus lessons).

1. If you're looking for the Lessons promised in Chapter 11 (stuff on linked lists), see Chapter 17 below.

2. If you're looking for Chapter 13, see Chapter 13 below. (It's still under construction.)

This information here is updated at least once a month. Check back often!

The Rules

To view a new Lesson, click its link. Then use your

browser's print gizmo to print the Lesson out on your printer. Staple the pages together, or use a three-ring binder to create your own Supplemental Lesson Folder. I believe you'll find that printing things out works better than trying to read and learn from a computer screen.

The Supplemental Lessons

Bonus C For Dummies Lesson 13.1Lesson 13.1 – Does Anyone Have the Time?

C has a host of time-related routines, none of which I ever talk about in the book. This stinks because getting the time or knowing the time or even displaying the current time is often an important part of most programs. I've gone by for too long!

TIMER.H

The time functions in C are defined in the TIMER.H header file for the most part and — stand back! — they're UNIX time functions. Yech! You would think that a programming language as nice as C would have it better, but no. (Compiler and operating system specific time functions are available, however.)

TIMER.H contains many functions, but the one I want to show you is time. You might guess that time displays the current time. But no. Or that it displays perhaps the date and time. But no. No! No! No!

The time function returns the number of seconds that have elapsed since midnight, January 1, 1970. GMT. Uh-huh.

Further, the value is returned in a special time_t type of pointer, which you must declare in your program:

```
time_t *timepointer;
```

Even so, the number of seconds that have passed since you were 12 (or maybe not even yet born!) is useless. I mean, can you imagine all the math required to divvy that up into years, months, dates, hours and seconds? Egads!

Fortunately, there is a companion TIME.H function called ctime, which converts the time_t value into handy and veryprintable string. Time for a program!

Name: TODAY.C

```
#include <stdio.h>
#include <time.h>

int main()
{
   time_t now;

   time(&now);
   printf("It's now %s\n",ctime(&now));
   return 0;
}
```

Shift+Click here to download a copy of the TODAY.C source code. This program is almost utterly naked C, so it runs anywhere. I just re-compiled it under gcc in Linux and it worked, so everyone should be happy here.

Compile. Link. Run!

It's now Sat Sep 02 17:05:15 2000

Here's what's going on:

The time_t now; statement creates the time_t pointer variable, into which that huge number-of-seconds variable is stored. The variable is used by the time function, time(&now) to create and store the current time — I mean, number of seconds since Nixon was in the Whitehouse.

The killer is the ctime function inside the printf statement. That's what converts the number of seconds into a string you can read on the display.

There. Nothing to it.

Well, unless you just want to display the time. Or maybe you just want to display the date. If so, you have to look elsewhere for your time or date functions. Alas.

Better DOS Functions

Now the rest of the programs in this lesson require use of the DOS.H header file, which accesses special DOS routines to display the date and time. If you're using Microsoft Visual C++ versions 4.0 or later, you cannot compile and run these programs. Sorry.

These programs do compile under DJGPP as well as Borland C++, providing you set the target as a 16-bit DOS project.

The DOS.H header defines two functions, getdate and gettime, which fill special structures with values representing the current date or time.

Nifty.

getdate requires you to create a date structure, into which it puts values as follows:

```
struct date

{
   int da_year;    /* current year from 1980 to 2099 */
   char da_day;    /* day of the month, 1 through 31 */
   char da_mon;    /* month, 1 through 12 */
};
```

gettime had you set up a time structure into which it places values in this manner:

```
struct time

{
   unsigned char ti_min;     /* minutes, 0 to 59 */
   unsigned char ti_hour;    /* hours, 0 to 23 */
   unsigned char ti_hund;    /* hunrdredths of seconds, 0 to 99 */
   unsigned char ti_sec;     /* seconds, 0 to 59 */
};
```

The following program, NOW.C, demonstrates how to put these functions together:

Name: NOW.C

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

int main()
{
   struct date date;
   struct time time;

   getdate(&date);
   gettime(&time);

   printf("Today is %d/%d/%d, it's %d:%02d",
      date.da_mon,
      date.da_day,
      date.da_year,
      time.ti_hour,
      time.ti_min);
   return 0;
}
```

Type in the above program, or just relent and shift-click here to download yourself a copy. Compile it. Run it.

Today is 9/2/2000, it's 17:06

What you see on your screen will, of course, reflect the current date and

time (according to the computer, at least). A few things to point out:

```
struct date date;
struct time time;
```

These two statements create the date and time structures into which getdate and gettime place the current date and time values. I used the names date and time for the variables, which could be confusing to some, but isn't to me!

The printf statement is pretty straightforward. Remember that it's the backslash, \, that needs to be specified twice if used in a printf formatting string. Also, see the %02d placeholder? That ensures that the seconds value always displays with a leading zero. Otherwise a time of 12:5 looks odd, when you're used to seeing 12:05 instead.

I split the variables in the printf statement onto separate lines so you can better see them. Those are merely the date and time structure values, though I neglected to put in the seconds and hundredths of seconds values.

Now . . . room for improvement. You have a homework assignment!

I want you to modify the NOW.C program. I would like it to display the time in a 12-hour format, from 12:00 a.m. on through 12:00 p.m. (noon), and then starting with 1:00 p.m. for one o'clock in the afternoon on up to 11:00 at night. So all you're doing is applying some logic and programming to get the display to read "right."

Please work on the modifications on your own. There is no right or wrong way to do it, though there are better and worse ways! When you're done, or

if you're stuck, you can click here to see my solution, which is only one of many potential ways to do it. Good luck!

Bonus C For Dummies Lesson 13.3Lesson 13.3 – Say Hello to Mr. Bit

Remember this guy?

Well, I do. He was the Little Man with the Stupid Hat who taught me how the decimal system worked.

The Little Man with the Stupid Hat (LMSH) had ten fingers – just like you do, boys and girls! Easy enough. But the point behind LMSH was to get everyone to understand the "10s position" and "100s position" and so on, the way a big decimal number stacks up. Such as:

That's 3 hundreds, 7 tens and 9 ones, which translates into the number 379. Remember how that works? Of course you do. (And if you don't, then at least you're nodding your head.) Well now I'd like to introduce you to Binary Man:

Poor Binary Man has only one finger. So he can only count to one. Well, one and zero. LMSH can use zero fingers to show a zero, and Binary Man can do so too. Not only that, Binary Man can count to larger numbers just as LMSH can: you just need more than one of him. Thus:

Because Binary Man has one finger, he counts by twos and not by tens. So the first place is the 1's, but the second place is the 2's, then 4's, 8's, 16's, 32's and on up, each time double the amount of the previous Binary Man.

Yes, this is weird.

And it's weird primarily because it's not the way we count. We count in tens, probably because we have ten fingures. But computers have no fingers. They have only themselves, so they count by twos. "Base two," is what it's called. Also known as binary.

Some Binary Numbers

(But not so many as to bore you)

All numbers are really symbols. For example:

That's not ten at all. It's the symbol "1" and "0," which people using such numbering systems refer to as "ten." There really isn't ten of anything up there; just symbols. In fact, not all humans use "10" to mean ten. The Chinese use the following symbol:

Again, that ain't ten of anything. So why not the following:

Your decimal-influenced mind will believe that to be the number 1,010 at first (and even if it were, it's not one thousand ten of anything). The number in binary, however, is the value 10. Here's Binary Man again:

So, just like you learned when you were young, you have 1 in the 8s place and 1 in the 2s place. Add 8 and 2 and you get . . what? Anyone? Anyone . . . ?

Of course, you get ten. That's how binary represents numbers. yes, it's weird. Here is your boring example:

Above you see the value 86. You have:

1 x 64

0 x 32

1 x 16

0 x 8

1 x 4

1 x 2

0 x 1

That's 64 + 16 + 4 + 2. Add it up and you get 86.

Now isn't this a pain? Sure it is. But you shouldn't worry about it since it's the computer that figures things in binary. With your programming skills, you can display that value as decimal or even hexadecimal. So the binary part is really pointless, though it helps to understand what's going on. (More on this in a few paragraphs.)

Hexadecimal is actually a shortcut for binary. Most programmers keep a table like this one handy, so they can easly convert between hex and binary.

Remember that binary is base 2. Computers are obsessed with base 2. The printf function lacks a method for displaying binary values. (You can display Hex and decimal just fine.) But don't panic! There's a binary value display function at the end of this lesson.

Bit Twiddling

C has a few operators that let you manipulate integer values at the bit level, what's known in programming circles as bit-twiddling. Here's the mysterious list for you:

<< Shift bits left

>> Shift bits right

& Boolean AND operation

| Boolean OR operation

^ Boolean XOR operation

~ One's compliment (unary)

Okay. You've seen them. I'll save the details for the next lesson. But I can't let you down here without giving you at least one program. The following program contains the infamous binString function, which returns a string representing a binary value. It actually uses the & and << operators shown above, so you'll have to wait to find out exactly how it works.

Name: BINBIN.C

```
#include <stdio.h>
#include <stdlib.h>

char *binString(int value);

void main()
{
    char value[8];
    int v;

    printf("Enter a value, 0 to 32000:");
    gets(value);
    v = atoi(value);

    printf("Decimal value is     %i\n",v);
    printf("Hexadecimal value is %4X\n",v);
    printf("Binary value is      %s\n",binString(v));
}

char *binString(int value)
{
    static char bin[17];
```

```
    int index;


    for(index=0;index<16;index++)
    {
        if(value & 0x8000)
            bin[index] = '1';
        else
            bin[index] = '0';


        value = value << 1;
    }
    bin[16] = 0x00;


    return(bin);
}
```

Shift+Click here to download yourself a copy of the BINBIN.C source code.

Shove it into your editor. Compile. Link. And . . . run!

Enter a value, 0 to 32000:

Type in your favorite number, such as 12345 (no commas). Press the Enter key and voila:

```
Decimal value is     12345
Hexadecimal value is 3039
Binary value is      0011000000111001
```

The program displays the value you input three different ways: decimal, hexadecimal and in binary. If you add up the binary digits, you'll get the proper decimal value (which you can verify using Windows' Calculator program in the Scientific mode).

A description of how BINBIN.C works, specifically the binString function, will be offered in the next lesson. For now, if you like, feel free to use binString in any program that requires a binary number string. The function works with any integer value.

Try running the program a few more times with some additional numbers.

Try some negative numbers (-1 through -32000) to see what happens. (I'll explain it in the next lesson).

Try entering some "holy" computer numbers: 255, 16384, 1024, 4096, 32767, etc.

The printf function automatically displays decimal values. This is okay since you, as a human, expect that.

You can use the %X (or %x) placeholder in printf to display a hexadecimal value. No sweat.

Binary values require their own function to display, which is what binString does in the program. Even so, some comiplers may have a "bin" placeholder character in their printf function.

Note how the binary and hexadecimal values relate. See how hex is a convenient shortcut for binary?

Bonus C For Dummies Lesson 13.5Lesson 13.5 – Color Text

Windows may have it all, but DOS could always print in colored text. The only PC that can't print in colored text was the old monochrome system. Since then, and since the color displays have gotten better and better, printing DOS stuff in color has been a snap.

The bad news is that there are no native C language routines or functions for writing colored text on the screen. (Well, they have 'em in Borland C++, which I'll get into later.) The good news is that, thanks to the versatility of the C language, you can write your own routines. This shouldn't be hard, providing you know your BIOS calls as described in Volume I.

But there's more good news and bad news!

The bad news is that writing colored text to the screen requires that you understand the color codes used by the PC. Even so, the good news is that it's not that hard and there's a table shown later in this Lesson that describes everything you need.

Enough news!

Writing colored text

The putchar and printf functions in STDIO.H are actually shortcuts to the secret, inner BIOS routines in the PC's guts. They use Interrupt 0x10 (the Video interrupt) function 0x0A, "Write attribute and character at cursor." This function has the following parameters, which are sent to the BIOS in various microprocessor registers:

Parameter Value Register

Function number 0x09 AH

Character code ? AL

Display page 0x00 BH

Text color ? BL

Character count ? CX

The Character code is the ASCII value of the character you're displaying, which is just like the character you'd specify in a putchar function call.

The Display page is usually zero. (The PC has several text display pages, though I've rarely seen any program use anything other than page 0. (If you wanted to, you could write a program to display text to different display pages and then "flip" the pages to rapidly display information – another waste of time!) Click here for the waste-of-time info.)

The text color tells the BIOS which foreground and background colors to set for the text, whether the text is blinking and whether or not the "high intensity" colors are used. More on that in a second.

Finally, the rarely used Character count value tells the BIOS how many characters to write to the screen. Normally you write only one, but you could write up to 65,000 characters providing you send the proper value to the Video interrupt. It boggles the mind!

Always use display page zero. That's the one your PC is probably using right now (in the text mode, anyway).

You must manually move the cursor after making this call. It does not move the cursor for you.

To move the cursor, use the locate function introduced in Lesson 5-3.

You'll also need to use another function to read the cursor so that you're sure you move it properly.

(It's a bother, but you wanted to write in color!)

The color numbers are listed later in this Lesson.

And those color numbers should be unsigned char types. Anything else and you'll peeve the compiler.

Feeling blue?

When Mr. Norton introduced his Norton Utilities version 2, he added a tool that changed the color of the DOS prompt and all text displayed at the DOS prompt. I have no idea how he did that, but you might get a hint of what's going on by trying the following program.

Name: BLUE.C

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

#define VIDEO 0x10
#define BLUE 0x1F

void dcolor(char ch,unsigned char color);
```

```
void main()
{
   char *text = "Am I blue?";
   unsigned char x;

   while(*text)
   {
      dcolor(*text,BLUE);
      text++;
   }

   putchar('\n');
}

void dcolor(char ch,unsigned char color)
{
   union REGS regs;
   int x,y;

/* First, read the cursor */

   regs.h.ah = 0x03;       //Read cursor
```

```
    regs.h.bh = 0x00;       //"page"
    int86(VIDEO,&regs,&regs);
    y = regs.h.dl;          //save Y pos.
    x = regs.h.dh;          //save X pos.

/* Now, write the color char */

    regs.h.ah = 0x09;       //Write color
    regs.h.al = ch;         //character
    regs.h.bh = 0x00;       //"page"
    regs.h.bl = color;      //color
    regs.x.cx = 1;          //count
    int86(VIDEO,&regs,&regs);

/* Move the cursor forward one notch */

    y++;

/* Reset the cursor's position (locate()) */

    regs.h.ah=0x02;         //Move cursor
    regs.h.bh=0x00;         //"page"
    regs.h.dh=x;            //row
```

```
    regs.h.dl=y;            //column
    int86(VIDEO,&regs,&regs);
}
```

Carefully type the above source code into your editor. Actually, since this is on a Web page, you can copy and paste the source code into your editor, or Shift+click on this link to download the source code directly.

Compile and run. You'll see the following text displayed:

Am I blue?

The background color is blue and the text is bright white. I'll explain how that worked in the next section.

This program was created on the Borland C++ compiler.

Older versions of Microsoft C can compile this program just fine. In some cases you may need to specify _REGS instead of REGS in the dcolor function. (Just put an underline before REGS.)

The while(*text,BLUE) construction is explained in Lesson 10-7.

The cursor position must be read before the character is displayed. (You could read it afterward, it doesn't matter.) The row and column positions are saved in variables y and x. Then the cursor's position is reset by incrementing the y (column) value. This is necessary because function 0x09 does not move the cursor by itself.

How Color Text works on your PC

There is room for up to 2000 characters on the standard PC text screen. That's 25 rows of 80 columns of characters. Wow.

Knowing that a character is a byte, you may think that the screen's "memory" can hold 2000 bytes. But that's not so. The screen is actually 4048 bytes big. That's because each character on the screen has a companion byte – an attribute byte that tells the monitor which color to display the character, both background and foreground.

The character and attribute bytes go hand in hand. Normally you never mess with the attribute bytes. You merely tell your program to display characters and it does so, placing them on the screen one after the other or in a specific spot. But Video interrupt function 0x09 allows you to also set the attribute byte and thereby control the foreground and background text color.

The PC produces color text using three colors: Red, Green and Blue. If you've been around long enough, you may remember the old RGB monitors. That's Red, Green and Blue all over again; the three primary colors used to create just about any other color you see on your screen.

In the attribute byte, there are three bits for the foreground color: one bit for Red, another for Green and another for Blue. Here's how that looks in the handy eight-bits-in-a-byte graphic thing:

RGB

The byte above is shown with eight bits, 7 through 0 reading left to

right. Bit 0 is the Blue bit; bit 1 is the Green bit and bit 2 is the Red bit.

Using binary math you haven't yet been properly introduced to (which is coming in one of those "missing Lesson 13 lessons"), you can substitute values for R, G and B above:

0000-0001 is 1, or 0x01, for Blue colored text

0000-0010 is 2, or 0x02, for Green colored text

0000-0100 is 4, or 0x04, for Red colored text

The numbers shown above (0000-0001) are binary; not decimal. This is base 2 counting stuff; so 100 is not "one hundred," is the binary number 100, which is four. (This all makes total sense after I get off my butt and write Lesson 13-3.) Anyway, the values aren't important.

Instead of quibbling about binary here, change the BLUE.C program. Change line 5 to read:

```
#define BLUE 0x01
```

This sets the attribute to 0x01, which is the Blue bit for blue text.

Save to disk! Compile and run! You should see the text as blue with a black background.

Now go back to the source code and change the same line to the values 0x02 and then 0x04. Compile and run after making each change. You'll see Green text and then Red text when you run each of the programs.

Musical interlude . . .

If you've been using a PC since the dark days, then you might recall that

there are actually seven colors you can make text on the screen. In addition to Red, Green and Blue, there is Cyan, Magenta, Brown and Gray. These colors are achieved by combining the primary three. On a binary, bit-by-bit level, it looks like this:

0000-0011 is 3, or 0x03, for Green+Blue colored text, Cyan

0000-0101 is 5, or 0x05, for Red+Blue colored text, Magenta

0000-0110 is 6, or 0x06, for Red+Green colored text, Brown

0000-0111 is 7, or 0x07, for Red+Green+Blue colored text, Gray

Finally, there is an eighth color available: Black:

0000-0000 is 0, or 0x00, for black

Black simply means that none of the colors are used. The PC really does display the text, but since it's black-on-black, you can't see it. (A black foreground is used primarily with a colored background.)

Rather than have you re-edit the BLUE.C source code to try all of the color values shown above, just modify BLUE.C as follows:

First, remove the #define BLUE statement. Then edit the main function to read as follows:

```
void main()
{
   char *text = "Sample Text";
   char *t;
   unsigned char fc;
```

```
    for(fc=0x01;fc<0x08;fc++)
    {
       printf("%2X - ",fc);

       t = text;          //reset pointer

       while(*t)
       {
          dcolor(*t,fc);
          t++;
       }
       putchar('\n');
    }
}
```

Save the changed file to disk as CLRTEXT.C and compile and run. You'll see something like this displayed:

1 – Sample text

2 – Sample text

3 – Sample text

4 – Sample text

5 – Sample text

6 – Sample text

7 – Sample text

Each of the "Sample text" lines appears in a different foreground color.

The number and hyphen part of the output (displayed by printf) is not affected by your color changes. Why? Because printf uses a different display routine. That text shows up as gray on white (like number 7).

The %2X part of the printf statement displays the color number as a hexadecimal value on the screen. This comes in handy later when you're trying to pick a specific color on your screen.

The pointer needs to be reset to allow the while loop to scan the same string of text for each iteration of the for loop. This is advanced pointer stuff mulled over in Chapter 10 of Volume II.

Turn on your brights!

The fourth bit in the character attribute byte is for intensity. It's not a color. Instead, the intensity byte tells the display whether or not to display the foreground color as bright or dim.

IRGB

The colors you've seen so far as dim, actually. To see the whole 16 course meal in both dim and bright modes, modify the CLRTEXT.C source code so that the for loop reads as follows:

```
for(fc=0x00;fc<0x10;fc++)
```

Basically, change the first value to 0x00 and then the 0x08 into an 0x10. That tells the program to cycle through all 16 color combinations, from everything off to everything on (bright white!).

Save the changes to disk. Compile and Run.

You'll see the sixteen possible foreground colors displayed on your screen.

Color code 0x00 is black – no colors at all.

Color code 0x0F is all white: 0000-1111 in binary. Looking at the above binary table, you see that the Intensity, Red, Green and Blue bits are all on. Everything's on!

The for loop cycles color codes from 0x00 through 0x0F. The code 0x10, specified in the for loop, isn't used as a text color value since that's what tells the loop when to stop. (Review your for loop info if this confuses you.)

Moving on to background colors

If things weren't confusing enough, the PC also allows you to choose up to 8 different colors for the text background. Yup, these are the same 8 colors used for the foreground text, codes 0 through 7 (see above). There is no "intensity" bit for the background color attribute, just Red, Green and Blue bits. And they live in the attribute byte thusly:

RGBIRGB

So for, say, a red background you would switch on the Red background bit: 0100-0000, or 0x40 for a Red background

Actually, the above attribute byte also sets a black foreground as well (since none of the Red, Green or Blue foreground bits are set). So this is where things get a bit tricky because you must set both foreground and background attributes at once to get the text color you want.

But don't panic! Fortunately, the nature of hexadecimal numbers is such that you can easily specify foreground and background colors without having to dig through a binary reference chart. Consider this:

0 = Black 4 = Red

1 = Blue 5 = Magenta

2 = Green 6 = Brown

3 = Cyan 7 = Gray

The hexadecimal digits above work for both foreground and background colors. So if you want Blue text on a Red background, you would use: 0x41 with 4 as the Red background color and 1 as the Blue foreground color. (The table at the end of this Lesson contains more detail, but the above information should sate you for now.)

Time for another sample program!

Modify CLRTEXT.C so that the main function now looks like this:

```
void main()
{
   char *text = "Hi!";
```

```
    char *t;
    unsigned char fc,bc;

    for(bc=0x00;bc<0x80;bc+=0x10)
    {
        for(fc=0x00;fc<0x10;fc++)
        {
            printf("%2X",fc+bc);

            t = text;           //reset pointer

            while(*t)
            {
                dcolor(*t,fc+bc);
                t++;
            }
        }
        putchar('\n');
    }
}
```

Essentially there is now a "nested loop" in the function. It loops through all the foreground and background colors one at a time, displaying each

along with its hexadecimal values.

Save! Compile and Run!

You'll see a grid displayed, detailing all the colors possible for the PC's screen.

Note the differences between the high-intensity and regular foreground colors.

All the colors with the same foreground and background values (0x11, 0x22, 0x33 and so on) show up as solid on the screen. Better not use those.

The foreground and background values are added in the program to get the final text attribute (fc+bc in the program). This works since the text color values don't overlap each other in a byte – again, this is advanced binary stuff you'll have to read about in Chapter 13. Eventually.

By the way, Chapter 13 uses text attributes in DOS to explain binary math and logic – really weird ass stuff.

The PC's text screen normally uses the 0x07 attribute for all characters. That is, 0 for black background and 7 for a gray foreground.

Finally, the annoying blink bit

There are eight bits in a byte (on the PC at least), so there is one more bit left to explain in the PC's text attribute byte: the blinking blinking attribute. By far the most annoying attribute available, which is why I

saved it for last:

BLRGBIRGB

When you set the blinking attribute on, the foreground text blinks. Very annoying. To see this in action, change the source code for CRLTEXT. You only need to change on line. Modify the call to the dcolor function so that it looks like this:

dcolor(*t,fc+bc|0x80);

You're adding the pipe character (Shift+\) and then 0x80 to the math function that sets the color attribute. This is a logical binary operation that sets the 7th bit in the byte without affecting the other bits. So it's fc plus bc OR 0x80.

Save the file to disk. Compile and Run!

You should see the same grid appear on the screen, but with all blinking foreground text.

Now there is an off-chance you may see the grid displayed with bright background text. For example, if you're using a DOS window in Windows, then the video driver may be set to prevent blinking text. If so, press Alt+Enter to switch to the full screen mode. Then type the following DOS command:

mode 80

This switches to the regular text screen. Now run the CLRTEXT.C program again. The foreground text should blink in an annoying manner. (Press Alt+Enter to return to Windows.)

And now, finally after all this work, you have the Massive Text Attribute Table. Simply combine the foreground and background hex values to get the type of colored text you want:

High bitsLow bits

CodeBackground colorCodeForeground color

00Black00Black

10Blue01Blue

20Green02Green

30Cyan03Cyan

40Red04Red

50Magenta05Magenta

60Brown06Brown

70Gray07Gray

80Black (blinking)08Dark gray

90Blue (blinking)09Bright Blue

A0Green (blinking)0ABright Green

B0Cyan (blinking)0BBright Cyan

C0Red (blinking)0CBright Red

D0Magenta (blinking)0DBright Magenta

E0Brown (blinking)0EYellow

F0White (blinking)0FBright White

The display color string routine

The dcolor function used throughout this lesson presents a handy way to

display one text character in color on the screen. A nice companion routine would be something that displays a whole line of text in a specific color. That would be the dscolor function, which is shown below in the final, final rendition of the CLRTEXT.C program, which you can download by Shift+clinking this link.

Name: CLRTEXT.C

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

#define VIDEO 0x10

void dscolor(char *string,unsigned char color);
void dcolor(char ch,unsigned char color);

void main()
{
   char *text = "Hi!";
   char *t;
   unsigned char fc,bc;

   for(bc=0x00;bc<0x80;bc+=0x10)
```

```
    {
        for(fc=0x00;fc<0x10;fc++)
        {
            printf("%2X",fc+bc);
            dscolor(text,fc+bc);
        }
        putchar('\n');
    }
}

void dscolor(char *string,unsigned char color)
{
    while(*string)
    {
        dcolor(*string,color);
        string++;
    }
}

void dcolor(char ch,unsigned char color)
{
    union REGS regs;
    int x,y;
```

```
/* First, read the cursor */

    regs.h.ah = 0x03;       //Read cursor
    regs.h.bh = 0x00;       //"page"
    int86(VIDEO,&regs,&regs);
    y = regs.h.dl;          //save Y pos.
    x = regs.h.dh;          //save X pos.

/* Now, write the color char */

    regs.h.ah = 0x09;       //Write color
    regs.h.al = ch;         //character
    regs.h.bh = 0x00;       //"page"
    regs.h.bl = color;      //color
    regs.x.cx = 1;          //count
    int86(VIDEO,&regs,&regs);

/* Move the cursor forward one notch */

    y++;

/* Reset the cursor's position (locate()) */
```

```
    regs.h.ah=0x02;         //Move cursor
    regs.h.bh=0x00;         //"page"
    regs.h.dh=x;            //row
    regs.h.dl=y;            //column
int86(VIDEO,&regs,&regs);
}
```

If you get into this color stuff, you may consider creating your own set of #define routines to set the color values. For example:

```
#define FG_Black 0x00
#define FG_Blue 0x01
#define FG_Green 0x02
#define FG_Cyan 0x03
#define FG_Red 0x04
#define FG_Magenta 0x05
#define FG_Brown 0x06
#define FG_Gray 0x07
#define BG_Black 0x00
#define BG_Blue 0x10
#define BG_Green 0x20
#define BG_Cyan 0x30
#define BG_Red 0x40
```

```
#define BG_Magenta 0x50
#define BG_Brown 0x60
#define BG_Gray 0x70
#define Bright 0x08
#define Blinking 0x80
```

Here is how the defines work: To create a certain color combination, specify the proper keyword above and separate it by the | (pipe), which is the logical OR. So to have bright white text on a blue background, you would use the following:

```
dscolor("Bright White on Blue",FG_Gray|Bright|BG Blue);
```

That's FG_Gray (foreground text color gray) OR Bright (for high intensity) OR BG_Blue (background text color blue). Again, there will be more of this binary logic junk in Chapter 13 – when I get around to writing it . . .

The Borland Solution

The Borland C compiler has a whole slew of color text commands, plus versions of common C routines that display text in color. This is a veritable treasure of fun things, which is never really highlighted anywhere since very few people bother with DOS programs any more.

Two different commands are used to set the text and background color. Text color is set with the textcolor function:

```
textcolor(color)
```

And background color is set with the textbackground function:

textbackground(color)

Each command requires an integer value, equal to the values described earlier in this Lesson for various colors to display on the screen. Or you can use the color macros predefined in the CONIO.H header file:

BLACK DARKGRAY

BLUE LIGHTBLUE

GREEN LIGHTGREEN

CYAN LIGHTCYAN

RED LIGHTRED

MAGENTA LIGHTMAGENTA

BROWN YELLOW

LIGHTGRAY WHITE

The colors you set are then used by any subsequent calls to the cputs or cprintf functions. Both of those functions work just like the regular C counterparts, puts and printf, though the output is in the color specified by the most recent textcolor and textbackground functions.

You must specify the CONIO.H header file in your source code for these functions to work.

Bonus Information on Display Pages

Bonus C For Dummies Lesson 13.6Lesson 13.6 – Introduction to Recursion

It's madness! Madness!

Recursion is one of the most frustrating topics in C. No one gets, at least not right away. And if you do get it, then you'll never understand it – at least until you need it.

I wasn't originally going to discuss recursion anywhere. I feel it's beyond the scope of the books and this Web page. But I've gotten some questions and I feel it's a neat trick. So I thought I'd toss it up here in the Missing Chapter since recursion is a subject often missing from many C language tomes.

Brace yourself

I am Not An Endless Loop

I am Not An Endless Loop

I am Not An Endless Loop

. . .

Recursion is the art of calling a function within that same function. Now the guy who thought this up had to be totally nuts. Or a genius. Because, honestly, why would you call a function from within a function? Let me drag you through the obvious here.

Name: RECURSE.C

```
#include <stdio.h>


void repeat(void)
{
```

```
    puts("Recursion can drive you mad!");
    repeat();
}

void main()
{
    repeat();
}
```

The above program, RECURSE.C, is stupid. I admit it. But it shows you recursion on the very basic level (albeit stupid recursion). The function repeat is called from within itself. Anyone who's reading this and who's been through Volume I of my C book will immediate know what's going to happen when this program is compiled and run.

Download the source code for RECURSE.C by Shift+clicking here, or you can type the thing in yourself. By the way, notice that I wrote it "upside down" so I didn't have to prototype the repeat function.

Compile. Yes it will compile. Remember that C compiles stupid things all the time.

Run.

```
Recursion can drive you mad!
Recursion can drive you mad!
Recursion can drive you mad!
Recursion can drive you mad!
```

Recursion can drive you mad!

Press Ctrl+C to stop it. (And please so stop it before you computer crashes, which it will do if you let the program run too long.)

The program calls the repeat function, which displays the string

Recursion can drive you mad!.

But the repeat function then calls itself again, which displays the same string.

Over and over.

The reason your computer crashes if the program runs too long is that you run out of stack space, the infamous "Stack Overflow" error message that drives DOS users nuts. This is explained in the next section.

Recursing Yourself to Death

Now, obviously, recursion is a useful tool. The program RECURSE.C does not show that. It merely illustrates, on a stupid level, how recursion works.

But the program doesn't work because there is no exit from the repeat function. The program is really an endless loop since repeat offers no chance to return to the main function. A program that really uses recursion would have that return.

So why does it crash? Because eventually the microprocessor runs out of stack storage space. Calling the function over and over without ever returning builds the stack to massive proportions, which eventually crashes the computer.

The stack itself is a storage place in memory, typically at the top of free memory. A special microprocessor register, the Stack Pointer, holds the stack's address. Values are pushed onto the stack and can then be popped off the stack.

Every time you call a function in C, the stack pointer holds the return address. For example.

In the figure above, the intro function is being called. The memory address where that function was called, 0x820 in the figure, is saved (pushed) onto the stack for storage. The stack pointer then moves down to the next available location on the stack:

So SP, the Stack Pointer in the microprocessor, now points to another address while the intro function runs.

After intro is done, control returns to the main function. The microprocessor finds that exactly location by popping the address stored on the stack pointer:

What happens with the RECURSE.C program is that the return address keeps getting pushed onto the stack. The stack keeps moving to a lower address in memory until the stack eventually destroys other data. The computer crashes.

To see how this works you can use the _SP macro in Turbo C++/Borland C++

(sorry Microsoft users!). Modify the RECURSE.C program by sticking the following statement into the repeat function:

printf("%04X - ",_SP);

The final program should look like this:

```
Name: RECURSE.C
#include <stdio.h>

void repeat(void)
{
   printf("%04X - ",_SP);
   puts("Recursion can drive you mad!");
   repeat();
}

void main()
{
   repeat();
}
```

Compile and run. This time you'll see the stack pointer's address displayed in the output. Something like this:

FBAA - Recursion can drive you mad!

FBA6 - Recursion can drive you mad!

FBA2 - Recursion can drive you mad!

FB9E - Recursion can drive you mad!

FB9A - Recursion can drive you mad!

FB96 - Recursion can drive you mad!

FB92 - Recursion can drive you mad!

FB8E - Recursion can ^C

Press Ctrl+C to stop so you can get a better view. That four-digit hex number gets smaller and smaller as the stack keeps pushing down into low memory. Eventually, as it reaches zero, the computer will crash.

To avoid crashing, your recursion routine must have an exit. Otherwise, it's just not recursion (and it messes up the stack).

A Way Out

When I first read about recursion I thought, "There blows the stack!" As an old-line assembly language programmer, I knew that calling a routine from within itself ran the risk of pushing the stack out the back of the computer. Fortunately, my friend Wally Wang showed me how to write a recursive routine that eventually bails itself out.

The following program is X5.C, which contains a modification of the repeat function in the RECURSE.C program. This time, the repeat function has a way out, but notice how recursion is still used to call the same routine from within itself.

Name: X5.C

```
#include <stdio.h>


void repeat(int x)
{
   if(x)
   {
      printf("%04X - ",_SP);
      puts("Recursion can drive you mad!");
      x--;
      repeat(x);
   }
}


void main()
{
   repeat(5);
}
```

Download this source code by Shift+clicking here, or you can just modify the existing source for RECURSE.C in your editor. Remember to save the file as X5.C.

Note how the repeat function has been re-typed. It now requires an integer

as input. Also, an if test is used, which is the key to making the recursion work.

Microsoft Compiler People: Do not include the printf statement that displays the stack pointer (_SP) value!

Compile and run.

No need to press Ctrl+C to stop it this time. You should see something similar to this as output:

FFEE - Recursion can drive you mad!

FFE6 - Recursion can drive you mad!

FFDE - Recursion can drive you mad!

FFD6 - Recursion can drive you mad!

FFCE - Recursion can drive you mad!

The repeat function worked through only five times, which is the value originally passed to the function. Recursion works! Here's how:

First, the repeat function requires an integer value. That value tells the function whether or not to repeat itself:

```
if(x)
```

Any positive value for x causes the loop to repeat. If x is zero, then the repeat function doesn't do anything but return. (Ah-ha! It can return!)

If x is greater than zero, then the if statements run:

```
printf("%04X - ",_SP);
puts("Recursion can drive you mad!");
```

x--;

repeat(x);

The stack pointer address is printed, along with the catchy phrase. But then x is decremented. The repeat function is called again, but this time with a new, smaller value for x. Then the whole thing happens all over. Eventually, X will equal zero. When it does, the if statements don't execute and the repeat function merely returns. At that point, the return addresses start popping off the stack: Pop! Pop! Pop! Pop! The repeat function just keeps returning to itself until eventually the big knot totally unwinds and control returns to the original calling function, main.

Amen!

Now you can sit and stare and try to figure it out.

Borland Compiler People: Before you ruminate, you can add one more line of code to the program to visually inspect the popping off of the return addresses from the stack. Modify the if statements to read as follows:

printf("%04X - ",_SP);

puts("Recursion can drive you mad!");

x--;

repeat(x);

printf("%04X popped off!\n",_SP);

The last statement is executed as the repeat function returns (when the

recursion starts unwinding). Now, when x equals zero, and the repeat function returns, you'll see the return addresses as they're popped off the stack.

FFEE - Recursion can drive you mad!

FFE6 - Recursion can drive you mad!

FFDE - Recursion can drive you mad!

FFD6 - Recursion can drive you mad!

FFCE - Recursion can drive you mad!

FFCE popped off!

FFD6 popped off!

FFDE popped off!

FFE6 popped off!

FFEE popped off!

Cute.

Something Remotely Useful

There was a great article on recursion in the recent Dr. Dobb's Journal, though most of it was above my head. But that, and a flurry of recent letters from readers, got me on the subject. I'll show you a really nifty example of recursion in the next lesson, but before then I thought I'd pass the following gem on to you.

I've been messing a lot recently with strings and arrays and needed some way to count the number of characters in a string. There is an strlen

function, which is fine, but like most people who teach C programming I just wouldn't settle for that. No, I wanted to count the number of characters in a string without using strlen or subtraction or even a while loop. No, no, no. I wanted recursion.

Name: CCC.C

```
#include <stdio.h>

int string_count(char *s)
{
   int count = 1;

   if(*s++)
      count += string_count(s);
   else
      count--;

   return(count);
}

void main()
{
   char *string="A sailor went to cc, see?";
```

```
    int x = 0;


    x = string_count(string);
    printf("The string \"%s\"\nis %i characters long\n",string,x);
}
```

Type this monster into your editor, or Shift+click here to download a copy.

Compile and run.

```
The string "A sailor went to cc, see?"
is 25 characters long
```

Now this one will probably baffle you. You can see how the count variable is incremented (actually added to itself with the += operator), but doesn't count get reset to 1 every time the string_count function is called?

Yes, it does. But who cares! The string_count function does not count the length of the string each time it's called. No, it counts the number of times the function returns.

You see, string_count returns an integer value that is added to the count variable, but that doesn't happen until the function returns.

For example, for every character in the string, the function is called and only the following chunk is executed:

```
if(*s++)
count += string_count(s);
```

The if statement checks to see if the NULL (end of string) character has been encountered. If not, then the pointer s is incremented and string_count is called again. Nothing happens to the count variable because the string_count function hasn't yet returned from itself. (Think about that one for a while.)

When the NULL character is encountered, then string_count starts to return. Only then is the count variable added to itself. The value of count grows by one every time string_count returns.

The else part of the if test decrements the value of count, which happens only once to account for the NULL at the end of the string. (Otherwise the size returned would always be one character too long.)

So, to wrap it all up, the string_count function uses recursion to call itself once for every character in the string. Then, it counts the number of returns, incrementing the value each time to get the size of the string.

Isn't this utterly insane?

Bonus C For Dummies Lesson 15-1Lesson 15-1 — What Files Lurk on Disk

Programmers do not live by simple file access alone. No, there must be directory access as well!

I get these questions all the time: So how can I find a file on disk in C? How can I change directories? How can I find out which disk or directory a file is using? They can all be answered when you unwrap the mystery of directory access in C.

Of course, there really is not directory access in C. It's all a myth! The actual routines used to access directory information are found in customized libraries for whichever operating system and compiler you're using. (Can you tell where this is leading?) For DOS, the library is DIR.H, which means these programs compile well in Borland C++ and DJGPP, but not at all in Microsoft Visual C++ 4.2 or later, or in Linux. Sorry.

Finding Files on Disk

There are two companion functions used to find files on disk, both of which are defined in the DIR.H header: findfirst and findnext.

The findfirst function is used first. It sets DOS up to look for files that match a specific pattern, which would be a matching scheme using wildcards or a specific filename itself. Esentially, the pattern is the option you would use with the DIR command:

```
DIR *.*
DIR *.C
DIR HELLO.EXE
```

., *.C and HELLO.EXE are all patterns findfirst would use.

Next findfirst uses a special file block structure, also defined in the DIR.H header. This structure is filled with information about the files

found: the file's name, size in bytes, date and time, attributes and other stuff they don't tell you about.

Finally, findfirst is passed the attributes of files to find. That way you can search for archive, read-only or even directory files. Searching for directory files, by the way, is the method by which you can scour the entire hard drive for files. (More on that in a later lesson.)

The following program shows you how findfirst can be set up to locate the first file in the current directory.

Name: D.C

```
#include <stdio.h>
#include <dir.h>

/* File attribute definitions */

#define NORMAL      0x00
#define READONLY    0x01
#define HIDDEN      0x02
#define SYSTEM      0x04
#define VOLLABEL    0x08
#define DIR         0x10
#define ARCHIVE     0x20
```

```
int main()
{
   struct ffblk fblock;          /* Create file info block */

   if( findfirst("*.*",&fblock,NORMAL) != 0 )
   {
      puts("Oops! Some kind of error!");
      return 1;
   }
   printf("I have found the file %s\n",fblock.ff_name);
   printf("It is %d bytes in size.\n",fblock.ff_fsize);

   return 0;
}
```

Hunt and peck this source code, or just click here to download. Compile it. Run.

You might see something like this as the output:

I have found the file D.EXE

It is 31792 bytes in size.

The findfirst function ventures out to disk and locates the first file it finds, which above was D.EXE (the program file itself). That file rang in at 31,692 bytes on my computer (compiled with BCC 5.0).

Here's what went on:

The first struct statement defines the file block structure used by the findfirst (and findnext) functions. The ffblk structure is defined in DIR.H. Here's what it looks like:

```
struct ffblk
{
  char     ff_reserved[22]; /* Internal use and secret stuff */
  char      ff_attrib;       /* 8-bit file attribute */
  unsigned ff_ftime;        /* Modification time (encoded) */
  unsigned ff_fdate;        /* Modification date */
  long     ff_fsize;        /* file size in bytes */
  char     ff_fname[13]     /* filename string */
}
```

This is all standard information, all of which is stored in an MS-DOS directory entry. And this is old DOS stuff too, since the creation date/time, access date/time as well as the long filename information is not present. But so much for that.

The structure is declared in D.C using struct ffblk fblock. Then the findfirst statement can do its magic:

```
findfirst("*.*",&fblock,NORMAL);
```

The first argument is the pattern to match. In this case, it's *.* (star-dot-star), to match all files in the directory. The second argument is the address of the file block, where information about the found file

is placed by findfirst. Then comes the attribute to match, which is defined as NORMAL in the program.

The whole findfirst function is placed inside an if statement to test for an error. If the function returns a non-zero value, then an error has occurred and the program bails out. Otherwise, the information stored in the structure is displayed using two printf statements.

Finding the Rest of the Files

You find one, you can find them all! After the initial findfirst function, subsequent calls to the findnext function (using the same fileblock structure), yields you all the files in the directory. Well, eventually.

And one at a time, as the following program demonstrates:

```
Name: DR.C
#include <stdio.h>
#include <dir.h>

/* File attribute definitions */

#define NORMAL      0x00
#define READONLY    0x01
#define HIDDEN      0x02
#define SYSTEM      0x04
#define VOLLABEL    0x08
```

```
#define DIR         0x10
#define ARCHIVE     0x20


int main()
{
   struct ffblk fblock;           /* Create file info block */


   if( findfirst("*.*",&fblock,NORMAL) != 0 )
   {
      puts("Oops! Some kind of error!");
      return 1;
   }
   printf("%s\t%d\n",fblock.ff_name,fblock.ff_fsize);


   while( findnext(&fblock) == 0)
      printf("%s\t%d\n",fblock.ff_name,fblock.ff_fsize);


   return 0;
}
```

Enter this program into your editor, or shift-click here to download it.

You can also create the program by simply modifying D.C as shown above.

Essentially there is just one extra line and the modified printf statement

is used twice.

Compile and run.

My output looked something like this:

```
D.EXE            31792
D.C        605
D.OBJ  732
DR.C     643
DR.EXE         31770
DR.OBJ         737
```

You see filenames and file sizes. Interesting. A few more moves and you could recreate the entire DIR command!

Nothing much changes between DR.C and D.C. The printf statement is modified to display only the filename and size. And it's used twice.

The second time printf is used is inside a while loop. The loop uses the findnext function as its test:

findnext(&fblock)

Using the findnext function with the same file block used by a previous findfirst function yields the next matching file from the directory.

That's it! The file block structure is refilled with information about the new file. And as long as the value returned is not zero, the while loop keeps searching for new files in the directory.

The while loop ends when the last file from the directory has been read.

You can change the wildcard specification to locate specific files, but you cannot (usually) specify a full pathname. In C you need to use the chdir function to change to another directory and search there. (More on that in a later lesson.)

If you need to build a database of files in the directory, then you'll need to create a linked list using structures similar to the file block, but with the linked list pointer added. (See Chapter 17.)

Changing the file attribute locates only files of that specific type.

For example, before Windows came along I wrote a DOS utility to locate hidden files in a directory, which was basically the same routine shown above but with the HIDDEN attribute specified.

Bonus C For Dummies Lesson 16-1Lesson 16.1 – Welcome to the Lower Level

The C language is considered a mid-level programming language. Mid-level because it has some parts that are considered high-level — the English-like words used for file access for example — and it has the grunts and squawks of low-level languages, which includes its abilities to twiddle bits and tweak memory with pointers and all that. But C also has another leg up because most C compilers allow you to embed the one of the lowest form of computer languages possible right there into your source code. It's called in-line assembly.

Both Borland C++ and Microsoft C++ have the built-in assembly language

feature. Even so, your compiler may implement it differently from the way documented here. Check your Help file for the details.

A keen advantage of in-line assembly is that it allows all compilers to access the computer's BIOS and DOS functions.

If you're using DJGPP, then you'll need to use the ATT nottaion, not the Intel Assembly notation shown here. Click here for more information. (Sorry!)

Need I mention that this information here is specific to a PC. You can use in-line assembly on non-PC computers, but the results will be different.

A Program Before You Get Too Bored

The following is a C program in name only. It's actually an Assembly language program placed inside a C program. Normally the whole dang doodle program wouldn't be assembly, only a small part would. But this is one of those demos, you know.

Name: HELLOW.C

```
void main()
{
        char *string = "Hello, Weirdo!$";

        _asm mov dx,string
        _asm mov ah,9
```

```
    _asm int 21h
}
```

Type the above program into your editor, or Shift+click on this link to download it.

Compile and run! You'll see:

Hello, Weirdo!

Honestly, there is nothing there to indicate that any string would display. What you did was to use in-line assembly language — a second programming language embedded in your source code — to cause the computer to do something. In this case, you used a DOS function called an "interrupt" to display text on the screen.

All the lines starting with _asm are assembly language directives.

Some versions of Microsoft C++ may require you to use two underlines before asm.

Borland C++ may require the presence of the TASM assembler to compile this program.

More detail on how the program works

The Miracle of Assembly Language

The microprocessor speaks only one language. It's called machine language. Like the name says, it's a language spoken primarily by machines, not by soft hairy beings like you or I.

Machine language consists of codes, often one- or two-byte instructions that tell the microprocessor to do something: access memory, compare

values, jump to another piece of code, and so on. It's all basic (very basic) programming.

As an example of machine language, consider this:

B4 09

The B4 is an 80x86 instruction to place a value into the AH microprocessor register — a storage place in the microprocessor for values. The value placed into the AH register is 9. From this you can deduct the following:

You would have to be nuts to write a program in machine language!

The truth is, no one writes programs in machine language. Instead, they use a programming language called Assembly. The Assembly language is merely a shortcut way to program a microprocessor in machine language. Though, instead of hexadecimal values, short mnemonics are used. Instead of B4 09 you would write:

mov ah,09h

The "mov ah" is assembly code for "put the following value into the microprocessor's AH register." And the 09h is assembly notation for the value 9 hexadecimal, what you would write a 0x09 in C. This is much easier than remembering B409. But to the computer, it's the same deal.

So the question begs, why bother?

Machine language is fast. Very fast. If Microsoft re-wrote all of Excel directly in machine language instead of C, the program would smoke. Unfortunately, it takes a lot longer to write programs in assembly than it does C. In C you have functions already defined for you. In assembly, you

have to create them all using the grunts and snorts of machine language. That takes a l-o-n-g time.

The solution is then to use C where C is most efficient: functions and other high-level language stuff. Then use assembly language for the stuff you need to run veryfast. Hence, in-line Assembly was born.

For this Lesson and the rest of this Chapter, I'll use in-line assembly merely to show you some tricks. In real life you'll probably use it only when you have some clumsy chunk of code you want to run veryfast.

More good news: Since in-line assembly works the same in both Borland and Microsoft C, I can show you some common routines between the two. No more "Borland-only" stuff.

Future lessons explain in detail about the microprocessor registers and how they work.

Most of this stuff is basic 8086 Assembly language, which still works on today's monster 32-bit Pentium systems. In fact, I would say very little assembly coding is done in the full 32-bit mode. (Mostly because your C compiler creates the 32-bit code for you much more efficiently.)

An Example From the Past

The following jewel comes from Volume I of C for Dummies, back when this whole BIOS and DOS call stuff was alien. Hopefully this will make it even more alien!

Name: DOSVER2.C

```
#include <stdio.h>


void main()
{
        char major,minor;


        _asm mov ah,30h
        _asm int 21h
        _asm mov major,al
        _asm mov minor,ah


        printf("This is DOS version %i, release %i\n",major,minor);
}
```

Type the above source code into your editor, or Shift+click this link to download the source code. This is a subtle modification of the original DOSVER.C program (which you can download if you Shift+Click here). Gone are the DOS.H header file, REGS declarations and the int86() function. Those aren't needed since you're accessing the microprocessor directly through in-line Assembly language.

Compile and Run.

The results you see on the screen may vary, depending on which version of DOS or Windows you're using. For most versions of Windows 95 you'll see either one of the following:

This is DOS version 7, release 0

Or, for later versions of Windows 95:

This is DOS version 7, release 10

This is all done using Assembly language right inside the C source code.

And you'll notice how easily the variables are echanged between the two formats. (Which is something I'll get into later.)

The DOS.H header file is no longer needed in DOSVER2.C because you're using Assembly language and not the int86() function.

The major and minor variables are declared as chars here instead of ints. The reason is that the AL and AH registers are byte-sized (8-bits). If the variables were integers, then the 8-bit values would not be stored properly.

This program may crash with some later versions of Microsoft C++ on Windows 95. There is a way to compile the program, but you must convince the Developer Studio how to create DOS programs. (If you know how to do this, please email me the solution. Thanks!)

Obligatory blow-by-blow of DOSVER2.C.

The next lesson covers the microprocessor itself, its registers and how they relate to the REGS & REGS stuff in C.

Bonus C For Dummies Lesson 17-1Lesson 17-1 – The Birth of Linked Lists

Structures are just about the best way to store database-like

information. Well, short of buying a database (which is what I would recommend to a beginner). You could probably prepare a heck of a little database using the simple structure programs demonstrated throughout Chapter 11 in Volume II. Unfortunately, there are some dreadful limitations.

For example, the program BDAYS.C attempted to create a database of birthdays. Declaring an array of structures did this:

```
struct family myfamily[10];
```

At first glance that seems perfectly fine. In fact, if you set out to build a database program, you could just expand upon the rudiments of the BDAYS.C program: just change the size of the array to match the maximum possible input you could imagine. No one would ever suspect a thing! But you would be wasting memory. And you could potentially run out of memory, for example, when the Osmond family started using your birthday database program.

The solution (or, more properly, the elegant solution) is to employ a technique known as a linked list. That's the subject of this Lesson.

A linked list is actually considered an advanced programming concept, which is why it's not covered in the book. (Well, that and I was running out of space.) It's advanced because it deals with pointers in a way most people wouldn't begin to guess at.

Just like most advanced concepts, someone smarter than you and I has already worked out the details for doing linked lists. You merely copy

the way that superbrain at MIT developed linked lists in the '70s, just like everyone else does. This isn't called cheating. No, it's "using an algorithm."

There is no requirement that you make a linked list when working with structures. Arrays are just fine most of the time. In fact, linked lists drive many programmers bonkers.

Feel free to skip this stuff if you like.

Before moving on, you must understand structures through Lesson 11-5 in the book. Further, a good grasp of pointers and the malloc() function is necessary here. You've been warned!

The Whole Big Fraggin' Idea Behind Linked Lists

Know these tidbits and you'll begin to understand linked lists:

A linked list is a bunch of structures containing pointers.

The pointers are of the structure type -- a pointer to the same structure.

The pointer contains the address of the next structure in the list.

It all works like following clues to get to a hidden treasure; one map takes you to another map, which takes you to another map and so on. In a linked list, each structure contains an extra item: a pointer (of the structure's type) which contains the address of the next structure in the list.

The pointer is the link.

The group of structures is the list.

The following figure attempts to illustrate this.

Each structure contains the address (pointer) of the next structure in the list. The first structure contains "George Washington" and then the address of the next structure in the list. This keeps going down to the last item in the list, which contains 000 or the NULL pointer.

A Program to Make You Feel Better

The problem with other C books that try to teach linked lists is that they give you the whole friggin' linked list program at once. I'll be different. Below is the silly source code for PREZEZ1.C, which starts to create a linked list-like structure as shown in the above illustration.

Name: PREZEZ1.C

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

void main()
{
   struct pres {
      char name[25];
      struct pres *next;
```

```
        };

    struct pres *president;

/* Create the first item in the list */

    president = (struct pres *)malloc(sizeof(struct pres));

    strcpy(president->name,"George Washington");
    president->next = (struct pres *)malloc(sizeof(struct pres));

/* Stop here and display the results */

    printf("The first structure has been created:\n");
    printf("president->name = %s\n",president->name);
    printf("next structure address = %i\n",president->next);

}
```

Type in the above source code, or Shift+click on this link to download a copy.

Compile and Run.

Here is what the output should look like:

The first structure has been created:

president-> = George Washington

next structure address = 3672

The structure was created and George Washington was stored in its string variable. Then the address of the next structure for which malloc() has allocated space is also included (though normally that value wouldn't be displayed). This sample holds the rudiments for all linked lists, but it's only showing you part of the Big Picture.

Refer back to Chapter 11 to discover what the -> symbol does.

Of course, the value you see on your screen may or may not be 3672 for the address of the next structure.

This statement may toss you for a loop:

president->next = (struct pres *)malloc(sizeof(struct pres));

This should make sense to you: the next variable requires a pointer value -- an address. To get that address, you simply use the malloc() function as shown above. Malloc returns the address of a chunk of memory equal to the size of the pres structure. (This was all explained in Chapter 11.)

Adding the Next Structure to the List (First Attempt)

The PREZEZ1.C program makes room for another structure, but it doesn't do anything with that room. This doesn't displease the C gods any; it's just that the program takes only half a step and never completes the process, which is to fill in the new structure created. This is really cinchy to do, but requires a bit of modification to the program.

First, a new structure variable should be created:

struct pres *new;

Second, that variable should be used to hold the value returned from the malloc() function. So that line should read:

new = (struct pres *)malloc(sizeof(struct pres));

The value returned from the malloc() function -- the address of the new structure in memory -- is saved in the new pointer variable. That value also needs to be stored in the first structure, so the following line is needed:

president->next = new;

Now the program still runs the same, but the new variable holds a value "that is never used" blah-blah-blah. So code needs to be added to fill in the new structure. Something like:

strcpy(new->name,"John Adams");

That fills in the name part of the new structure, but to fill in the pointer part you'll need something like this:

new = (struct pres *)malloc(sizeof(struct pres));

Oops! That changes the value of the new variable. Which means you can't use the following code:

new->next = new;

Ugh! Looks like another structure pointer variable is needed. You must keep the value of the current structure's address but also have a variable to store the next structure's address. Will the madness end?

The president variable holds the address of the first structure in the list

The new variable holds the value of the second structure in the list.

But . . .

You cannot use the new variable to hold the address of the third structure in the list unless you first save that address in yet another pointer variable.

Yes, there is a solution.

No, the solution does not involve an array of pointer variables, though you should pat yourself on the back for thinking of it.

Adding the Next Structure to the List (Second Attempt)

What you need to work all the structures and their addresses are only three pointer variables:

One pointer variable holds the address of the first item in the list, *first

One pointer variable holds the address of the current item in the list, *current

One pointer variable is used to hold the address of the next item in the list, *new

Of course there are problems with this, but don't rush ahead of yourself.

You still need to fix up the PREZEZ.C program so that you can add another president's name to the list.

Name: PREZEZ2.C

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

void main()
{
    struct pres {
        char name[25];
        struct pres *next;
        };

    struct pres *first;
    struct pres *current;
    struct pres *new;

/* Create the first item in the list */

    first = (struct pres *)malloc(sizeof(struct pres));

    strcpy(first->name,"George Washington");
    new = (struct pres *)malloc(sizeof(struct pres));
```

```
    first->next = new;

/* Fill in the next item: */

    current = new;

    strcpy(current->name,"John Adams");
    new = (struct pres *)malloc(sizeof(struct pres));
    current->next = new;

/* Stop here and display the results */

    printf("The first structure has been created:\n");
    printf("first->name = %s\n",first->name);
    printf("next structure address = %i\n",first->next);

    printf("The second structure has been created:\n");
    printf("current->name = %s\n",current->name);
    printf("next structure address = %i\n",current->next);

}
```

Type the above source code into your editor, or Shift+click here to download the PREZEZ2.C file. Some things have changed: The old president

variable has been replaced by the first variable, and the new and current pointers have been added.

Compile and Run!

Here's a sample of the output you might see:

```
The first structure has been created:
first->name = George Washington
next structure address = 3768
The second structure has been created:
current->name = John Adams
next structure address = 3800
```

Of course, this program is still limited in its scope. However, it provides a good foundation for linked lists.

There are some limitations: For example, a loop would be better suited to add new items to the list. Also, there must be some way to tell when you've reached the end of the list. These issues will be addressed in the next lesson.

Bonus C For Dummies Lesson 17-2Lesson 17-2 – The Adolescence of Linked Lists

I think I could rattle off about a dozen reasons why so many people find linked lists confusing. Above all, it has to be that most programming books present you with a Linked List Program that's all complete. There's nothing to really learn since you don't ever discover the painful process

-- the journey. You're merely presented with the reward and implored to understand it. This is why linked lists come in second only to pointers on the Eyeballs Bugging Out of Your Head meter.

Meanwhile, back at the White House . . .

From Lesson 17.1 you've seen a vague attempt at creating a linked list using the PREZEZ.C series of programs. Generally, it's rather cinchy to go ahead and create a structure with pointers (this was covered in Lesson 11.5):

Define the structure.

Define pointer-structure variables.

Use malloc() to create space for the pointers.

Fill the space for the new structure using the -> thing.

In fact, aside from using malloc() and the -> thing, structure-pointers are no different than structures themselves. You just create them on the fly, as you need them.

With a linked list, you not only create the structures, but you store the address of each structure in the list; the current structure holds the address of the next structure in the list. That way you can find things; the list is "linked" together by the address in memory of the next structure. Nothing gets lost!

So far, the PREZEZ2.C program has it almost right. The bottom line is that you don't need a whole basket full of structure pointer variables to make a linked list. You need only three: first, current and new. Working those

into a loop is the problem. (And then there's the problem of starting and stopping the linked list, but don't rush yourself!)

First Detour: Filling the linked list

A linked list has to be a database of sorts. Somewhere low in memory in your PC is a linked list describing all the disk drives in your system. The operating system creates it when it boots: the BIOS tells the operating system how many drives are physically present and the operating system creates a linked list describing each drive. If any other drives are added during the boot process (CD-ROMs, removable drives, network drives), they're added to the linked list. When the thing is done, the operating system gives each drive a letter depending on its position in the list. Oh, but that's all trivial.

I've set up the PREZEZ.C series of programs to use static data -- strings in double quotes -- instead of input from the keyboard:

```
strcpy(first->name,"George Washington");
```

and

```
strcpy(current->name,"John Adams");
```

This is necessary partly because of the database-like nature of linked lists but also because writing an input routine from the keyboard would make the program bigger. (That type of linked list program is coming, by the way.) So the first problem to tackle with spiffing up the PREZEZ.C program is to handle inputting the strings into the list. And the best way to do that is with an array and a loop.

The following program is POTUS.C. Potus is the code word used for the President Of The United States. (Kind of James Bondish, huh?) It shows how to display an array of strings using a while loop. A null string ends the loop, as opposed to some counter or number.

```
Name: POTUS.C
#include <stdio.h>

void main()
{

   char *presidents[] = {
      "George Washington",
      "John Adams",
      "Thomas Jefferson",
      "James Madison",
      "James Monroe",
      ""
      };

   int index = 0;

   while(*presidents[index])
```

```
    {
        printf("%s\n",presidents[index]);
        index++;
    }
}
```

Type in the above source code, or Shift+click on this link to download a copy.

Compile and Run -- you know the drill. When you run it, you should see the following:

George Washington
John Adams
Thomas Jefferson
James Madison
James Monroe

The array of Presidents works like any string array: It's an array of pointers, but in this case the pointers contain the address of a set of strings in memory. (Don't let that "array of pointers" thing get to you; it's an array o' strings.)

The last item in the array is a NULL string. This is similar to making the last item in a list of integers zero; it marks the end of a long list. The array isn't even defined with a number in the brackets. That way you can add new names without having to re-count anything.

To count the items in the list, an index variable is used. It's initiazlied to zero, the first element in the array.

A while loop ticks through each string in the list, displaying each string and incrementing the index variable. Only when the NULL string is encountered does the loop stop.

And that, ladies and gentlemen, is exactly how you will fill the names in your linked list of presidents. End of side trip.

See Chapter 9 for more information on arrays and strings.

Lesson 10.7 discusses the relationship between pointers and strings.

If you dare to look at it, Lesson 10.10 covers the thorny ground of arrays of pointers.

Second Detour: Looping the Linked List

To create the linked list you need a loop, a loop with a single if test in it to be sure you handle the first structure in the linked list properly.

After that's done, filling in the remainder of the structures works similarly to the way the POTUS.C program displayed the presidents' names.

The following program turns the trick:

Name: PREZEZ3.C

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
```

```
void main()
{
    struct pres {
        char name[25];
        struct pres *next;
        };

    char *presidents[] = {
        "George Washington",
        "John Adams",
        "Thomas Jefferson",
        "James Madison",
        "James Monroe",
        ""
        };

    struct pres *first;
    struct pres *current;
    struct pres *new;
    int index = 0;

/* Initialize all the pointers */
```

```
    first = (struct pres *)NULL;
    current = (struct pres *)NULL;
    new = (struct pres *)NULL;

/* Create the first item in the list */

    first = (struct pres *)malloc(sizeof(struct pres));
    current = first;

/* Create each item in the list */

    while(*presidents[index])
    {
        strcpy(current->name,presidents[index]);
        new = (struct pres *)malloc(sizeof(struct pres));
        current->next = new;

        current = new;
        index++;
    }

/* display the results...
 * (this section to come)
```

```
 * Just display first item in the list
 */

    printf("The first structure has been created:\n");
    printf("first->name = %s\n",first->name);
    printf("next structure address = %i\n",first->next);

}
```

Type in the above source code or Shift+click here to download a copy. The program is explained in detail below.

Compile and run!

The program's output is stunted, displaying only the first structure in the list:

```
The first structure has been created:
first->name = George Washington
next structure address = 3736
```

I limited output because displaying the structures opens a whole new can of worms that this program fails to address. (More on that later.)

How it Works

Here are the details -- the blow-by-blow, though I lack rights to use that icon here:

The program starts by creating the first structure in the linked list. That address marks the start of the linked list, so the *first variable

must never change:

first = (struct pres *)malloc(sizeof(struct pres));

current = first;

After the *first variable is created, it's stored in the *current variable

for use inside the loop. At this point, the first structure can be filled

in the loop just like every other structure.

The guts of the program consists of a while loop, identical to the one in

POTUS.C:

while(*presidents[index])

The loop continues to spin as long as there are strings in the presidents

array. The index variable is incremented at the end of the loop to keep

track of the position in the array:

index++;

Inside the loop, two things happen: The string is saved in the structure

and a new structure and pointer are created and stored:

strcpy(current->name,presidents[index]);

new = (struct pres *)malloc(sizeof(struct pres));

current->next = new;

This works the same as for the PREZEZ2.C program, however in this program

you've eliminated the extra pointer variables and replaced them with

*current and *new. The *current pointer references the current structure

and *new is used only to get the address of the next structure in the list.

The following statement ensures that the next structure is properly filled-in:

current = new;

This means that the next time the loop spins, it will be filling in the next structure in the list. After that, the index variable is incremented and the process continues, creating structures to fill.

And Now, the Problems . . .

The PREZEZ3.C program isn't without its problems. Before that, note what works:

- The program creates a linked list, with each structure containing the address of the next structure in memory.
- The loop fills each structure using a minimum of pointer variables
- The first structure's address is stored in the *first variable, allowing the rest of the program to locate that structure and then all the other structures in the linked list.

And now a few things that don't work:

- The program doesn't display all the items in the list. No biggie; the next program in this Lesson shows you how. (I just didn't want the thing overly bulked up.)
- The program works only in one fell swoop: it creates the linked list all at once. If you wanted to write a program that added items to the linked

list one at a time, this code wouldn't cut it. (More on that in the next lesson).

Okay. Minor gripes, but legit. The first item in the gripe list is most important. How would you display all the items in the linked list? It's possible, of course. But how?

Think about it! Go back and look at the source code. I'll wait here.

Do-be-do-be-do Dee-do-be-do-be . . .

Right. You need some kind of loop. A for loop would work, but you'd be cheating; you already know how many linked lists there are so you could hard-code the thing:

```
/* Display the results */


   current = first;

   for(index=0;index<5;index++) { printf("structure %i: ",index+1);
printf("%s\n",current->name);

      current = current->next;

   }
```

Use the *current pointer to stomp through the linked list. Initialize it to the start of the list, as shown above with current = first;. Then, because you know there are only five structures in the list, re-use the index variable in a for loop to march through the list only five times. Inside the loop, use printf to display each structure's data. Then the *current pointer is reset to the address of the next structure using the

current = current->next; statement. Nifty. Tidy. And it works.

BUT IT'S NOT GOOD PROGRAMMING!

You can Shift+click here to download the PREZEZ4.C program, which contains the above for loop to display the structures. Go ahead, even though you know it's not the real solution to the problem.

Compile the code and run the program. Output looks like this:

Structure 1: George Washington

Structure 2: John Adams

Structure 3: Thomas Jefferson

Structure 4: James Madison

Structure 5: James Monroe

If you're from the school of "If it works, it's done," then you can stop here. Otherwise, you'll need to devise a more elegant solution.

Right: it still needs a loop. But why not a while loop? After all, a while loop was used to create the structures. Likewise, a while loop can be used to display them. The drawback to that approach is that a while loop needs a condition on which to stop. Right now, there is no such condition, which is yet another problem with the PREZEZ.C series of programs: how can you tell when the linked list is done.

Now you could create a *last pointer. But that's overkill. (If you thought of that, then you're using your brain too much. This is C programming, not Pascal!) Think instead of how the first while loop works. How does it know

when to stop?

N U L L . . .

Right. To end the linked list, you need a NULL pointer. The final structure in the list should have a pointer to NULL instead of a pointer to the next structure. That makes so much sense you'd expect it to be taught in Sunday School: The last structure in the list, logically, has nothing following it. It demands a NULL pointer!

Here's another look at that linked list illustration to drive the point home:

Look at the last structure in the list. What does it end with? 000? Isn't that a NULL? Absolutely!

Now two things must be done: The loop that creates the linked list must cap it off with a NULL pointer in the last structure. After that, the loop that displays the list uses the same NULL to determine when the list is done. You could ruminate on this for a time if you like, or just peek ahead to the following source code, the last of the PREZEZ.C series.

Name: PREZEZ5.C

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
```

```
void main()
{
    struct pres {
        char name[25];
        struct pres *next;
        };

    char *presidents[] = {
        "George Washington",
        "John Adams",
        "Thomas Jefferson",
        "James Madison",
        "James Monroe",
        ""
        };

    struct pres *first;
    struct pres *current;
    struct pres *new;
    int index = 0;

/* Initialize all the pointers */
```

```
    first = (struct pres *)NULL;
    current = (struct pres *)NULL;
    new = (struct pres *)NULL;

/* Create the first item in the list */

    first = (struct pres *)malloc(sizeof(struct pres));
    current = first;

/* Create each item in the list */

    while(1)
    {
        strcpy(current->name,presidents[index]);

        index++;
        if(!*presidents[index])
        {
            current->next = (struct pres *)NULL;
            break;
        }
        else
        {
```

```
            new = (struct pres *)malloc(sizeof(struct pres));
            current->next = new;
            current = new;
        }
    }

/* Display the results */

    current = first;
    index = 1;
    while(current)
    {
        printf("Structure %i: ",index++);
        printf("%s\n",current->name);
        current = current->next;
    }
}
```

Type in the above source code or Shift+click here to download a copy.

Compile and Run. The output will be the same as for PREZEZ4.C (above) but the method is better, more flexible. Here's what's going on:

First, a while(1) ("while true") loop is used to spin indefinitely. That's because an if-else structure is used inside the loop to determine when the last item from the array has been read. (There are other ways to do this

as well.)

Inside the loop, the structure is filled in two steps. First, the president's name is fetched from the array and stuffed into the structure:

strcpy(current->name,presidents[index]);

This works the same for all the structures, whether they're the first, middle or last.

Next, the index variable is incremented:

index++;

You need to determine whether or not this is the last structure and the way to know that is to see if there is another item lurking in the presidents array. An if statement checks for you:

if(!*presidents[index])

This is a peek ahead. The value of the string there isn't used now; it will be used the next time the loop repeats (if it repeats). Otherwise, if tests to see whether anything is there at all. If not (which is the ! part of the test), then the next pointer in the structure is assigned a NULL structure-pointer value and the loop breaks:

current->next = (struct pres *)NULL;

break;

Otherwise, malloc() grabs another chunk of memory. The new address is saved, the current pointer is reset to that address so that the loop can repeat and the new structure be filled:

new = (struct pres *)malloc(sizeof(struct pres));

current->next = new;

current = new;

To display the results a simpler construction can be used. After all, the next pointer in each structure variable holds an address for every item in the list but the last one. In that case, it holds the NULL value. And all of the C language looping statements interpret NULL as a FALSE.

Before the loop starts, the current pointer is initialized to the address of the first item in the list, and the index variable is set to one (for counting purposes):

current = first;

index = 1;

After that, the loop can spin until the value of the next pointer is zero (index is not used for the loop!):

```
while(current)
{
   printf("Structure %i: ",index++);
   printf("%s\n",current->name);
   current = current->next;
}
```

Only the name part of each structure is displayed with a printf. In fact, it could all go on one line:

printf("Structure %i: %s\n",index+1,current->name);

Then, the current variable is set equal to the address of the next

structure in the linked list:

current = current->next;

When the last item in the list is encountered, the current variable will be set equal to NULL. That's the condition the while loop interprets as the end of the loop.

The program may not be utterly elegant, and it's certainly not the only way to accomplish the task, but it works and is flexible: You can add the rest of the presidents' names to the array and the sucker will still work as advertised -- which is the bottom line.

The next lesson shows a different way to add and display elements in a linked list -- more of a database-like thing than the PREZEZ.C series of programs.

Bonus C For Dummies Lesson 17-3Lesson 17-3 – The Dawn of the Database

Linked lists are really all about databases. The structure itself is like a record in a database, and the variables in the structure are fields. I can't see how it could get any more obnoxiously obvious. And if that's all true, then the linked list is really just another term for a database. So starting with this Lesson and continuing for a few more Lessons, you're about to take a bath into the world of databases.

The Ubiquitous Bank Account Program

It seems like the last few times I've gotten questions from readers about linked lists it had to do with banking programs. You know the type: They

lose track of you and your money – unless you owe them money, in which case they're on you like white on rice.

The following program is BANK1.C, which should tell you immediately that it's only the first in what will probably be several more BANK programs.

This program creates a linked list one record at a time. Unlike the previous linked list examples, this one take a giant step into the area of databases.

Name: BANK1.C

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <ctype.h>
#include <conio.h>

void addNewAccount(void);

struct account {
    int number;
    char lastname[15];
    char firstname[15];
    float balance;
```

```
    struct account *next;
    };
struct account *first,*current,*new;
int anum = 0;

void main()
{
    char ch;

/* Initialize all the pointers */

    first = (struct account *)NULL;
    current = (struct account *)NULL;
    new = (struct account *)NULL;

    do
    {
        puts("\nA - Add a new account");
        puts("Q - Quit this program\n");
        printf("\tYour choice:");
        ch = toupper(getch());
        switch(ch)
        {
```

```
            case 'A':
                puts("Add new account\n");
                addNewAccount();
                break;
            case 'Q':
                puts("Quit\n");
            default:
                break;
        }
    }
    while(ch != 'Q');
}

void addNewAccount(void)
{
    char buffer[64];

    new = (struct account *)malloc(sizeof(struct account));

/* Check to see if this is the first record
 * If so, then initialize all the pointers to this,
 * first structure in the database
 */
```

```
    if(first==(struct account *)NULL)
        first = current = new;

/* Otherwise, you must find the end of the structure list
 * (Easily spotted by the NULL pointer) and add on the
 * new structure you just allocated memory for
 */

    else
    {
        current = first;        //make the first record the current one
                                //and loop through all records:

        while(current->next != (struct account *)NULL)
            current = current->next;

                                //the last record is found
        current->next = new;    //save the address of the new record
        current = new;          //and make the current record the new one
    }

/* Now, you just fill in the new structure */
```

```
    anum++;
    printf("              Account number: %5i\n",anum);
    current->number = anum;

    printf(" Enter customer's last name: ");
    gets(current->lastname);

    printf("Enter customer's first name: ");
    gets(current->firstname);

    printf("      Enter account balance: $");
    current->balance = atof(gets(buffer));

/* Finally, cap the new record with a NULL pointer
 * so that you know it's the last record:
 */

    current->next = (struct account *)NULL;
}
```

You'd be nuts to type in that whole program by hand. Instead, Shift+Click here to download a copy for your files.

Compile the code. Run it.

A - Add a new account

Q - Quit this program

Your choice:

There are only two options at this stage. Type A to add a new record. Fill in the last name, first name and a bogus account value. After that, you're returned to the main menu.

Presently there's no way to review the records you're creating; that function will be added in the second half of this lesson. Before then, what's going on needs to be dissected.

You can press A again to add another record. In, fact you can keep adding records all day long.

The program already specifies the account number for you, starting with 1 (one). This value is stored in the anum variable.

Notice that the anum variable is global, as is the definition of the structure and the three linked-list pointers: first, current and next.

Please review Lessons 17-1 and 17-2 on linked lists if the concept is new to you.

Chapter 11 in Volume II of C for Dummies discusses structures in general.

How it All Works

The key to the program is a small snippet of code that's used to find the last structure in a linked list. This is something new; the previous

lessons all stuffed the link list in one fell swoop. This time, to add a new structure (or actually perform just about any operation on the linked list) you need to know which structure is the last. Here's the simple while loop that does that:

```
while(current->next != (struct account *)NULL)
   current = current->next;
```

The pointer variable current contains the address of the current structure.

The value of current->next is the address of the next structure in the list. If that address is NULL, it means you've found the last structure in the list. The while loop stops.

If the value of current->next is not NULL, then the statement

current = current->next

adjusts the current pointer to contain the address of the next structure in the list. The process repeats.

This while loop elegantly stomps through every structure in the list. After it's done, the value of the current pointer is the address of the last structure in the list (the one with NULL as the value of current->next. The Figure below illustrates this somewhat.

Working the whole BANK1.C program from the top down: The main function merely displays a menu and calls other functions to do the actual work. Nothing there should surprise you, other than the do-while loop instead of

a while(TRUE) loop (which I normally use). Again, this is just Yet Another Way things can be done in C.

The addNewAccount function is where the fun starts. And the fun starts with allocating space for the new structure:

```
new = (struct account *)malloc(sizeof(struct account));
```

Now, normally there should be some code there comparing new to NULL just to see if memory was available. But these structures are so small it should never happen. (In a "real" program you must do that.) (I know I'm bad for not checking. A later lesson actually does the checking. Promise!)

After creating the new structure, you need to check to see if it's the first structure in the linked list:

```
if(first==(struct account *)NULL)
```

All the structure pointers were initialized to NULL in the main function, so if first is still a NULL pointer, then you're adding the first record in the list. (Yet another reason to always initialize pointers!). The address of this, the first structure, is then stored in the first pointer variable, as well as the current pointer variable:

```
first = current = new;
```

Otherwise, if there are already structures in the list, you need to spin up to the last structure and add the new structure to the end. That happens in an else part of the program:

```
else
{
```

```
    current = first;        //make the first record the current one
                            //and loop through all records:

    while(current->next != (struct account *)NULL)
        current = current->next;

                            //the last record is found
    current->next = new;    //save the address of the new record
    current = new;          //and make the current record the new one
```

First the current pointer is initialized to the address of the first structure in the list. Then your friendly while loop is used to spin through all the structures until the last one is found (the one with NULL as its current->new value.)

After the while loop spins, the record current points at is the last record in the list. First, the current->new value is changed from NULL to the address of the new structure. Then the new structure is made the current structure, so that it's data may be filled in.

```
anum++;
printf("            Account number: %5i\n",anum);
current->number = anum;
```

The value of the global variable anum (account number) is incremented and assigned to the new account. The extra spaces in the printf string line up the colon with the next several prompts. And the %5i makes the integer

display to 5 digits.

The next few questions and answers are pretty easy to figure out (providing you've been through Volume I).

Finally – and most importantly – the current->next variable is assigned the NULL value, marking the end of the list:

```
current->next = (struct account *)NULL;
```

That ties everything up nicely.

Adding the List All Routine

It's pointless to create data and shove it into memory without any chance to getting it back out. It's like Garbage-In/Garbage-Out, but without the Garbage-Out part. Fortunately, listing the contents of the structure is a snap.

In fact, it's so easy, I would like you to try it on your own. So don't peek ahead! Sit down and write a routine called listAll and stick it on the end of your BANK1.C source code file. Then modify the main function so that pressing the L key displays all the files in the list.

Do it now!

Compile? Run?

Did it work?

If not, keep in mind you can use the NULL byte at the end of the list as a test for the end of the loop.

How about this. Run the program again, and press L to list all the records

before you entered any records.

Did it crash?

Okay. Give up? The following is my code for the listAll function. This isn't the best or only way to do, just another way:

```
void listAll(void)
{
   if(first==(struct account *)NULL)
      puts("There are no records to print!");
   else
   {
      current=first;
      do
      {
         printf("%5i: %-15s %-15s $%8.2f\n",\
            current->number,\
            current->lastname,\
            current->firstname,\
            current->balance);
      }
      while((current=current->next) != (struct account *)NULL);
   }
}
```

First, an if test is done to ensure that there are records to list. If the

value of the first pointer is still NULL, then the linked list is empty.

The program can go home!

The main bulk of the structure listing routine is a do-while loop. This is one of the smoothest ways to do things without skipping over the first or last item in the linked list. The loop is driven by the following:

```
(current=current->next) != (struct account *)NULL
```

The current=current->next statement moves the current pointer up through the linked list. But the != stops it when the result of that operation is the NULL pointer. This is one way to read in the last structure in the list (other ways skip the last structure). After all, current=current->:next does assign NULL to the value of current for the last structure. When that happens, the do-while loop stops cold.

Inside the loop, a printf statement formats the output. This lines up everything in nice, neat columns and rows. Here is sample of this routine's output:

```
1: Lennon          John          $  100.00
2: McCartney       Paul          $  100.00
3: Harrison        George        $  100.00
4: Starr           Ringo         $ 1000.00
```

Don't worry about modifying your program to look like mine. Instead, just Shift+click here to download the complete source code for BANK2.C.

Compile and run BANK2.C if you like. Compare it with your own source code.

Remember, it's not the best way to do things, just another way.

The backslashes in the printf statement are to separate it onto different lines. (It is rather long.)

Numbers between the % and the i or s in a printf formatting string tell printf how many spaces to use when displaying data.

The - (minus sign) in the printf statement left-justifies the output.

The next lesson continues this series with examples of deleting records in the database. Betcha can't wait!

Bonus C For Dummies Lesson 17-4Lesson 17-4 – Decimating the Linked List

Most of us take the word "decimate" to mean "destroy," which is true but not the original meaning of the word. When the Roman army decimated a foe, they just killed every tenth guy. Ten = dec, which is where decimate came from.

Decimation is what you're about to do in your BANK.C program. But you won't be destroying the list entirely. Instead, you'll just be killing off random structures in the linked list. This is entirely workable – in a sort of insane, twisted way.

Getting rid of deadbeat records

If you have a database that's just a series of 3x5 cards, then removing old entries is cinchy: pull out the dead card. No one would know!

Now suppose your database is a series of structures in an array. How can you delete structure account[7]? If you just zero out the entries, then you have a blank record in the database. No, you'd probably have to copy all the structures account[8] and above down a notch. Yeah, that might work . . .

But arrays are boring. The truth is pointers are much easier to work with. After all, a linked list is nothing but a string of pointers to various spots in memory. Consider this illustration (not to scale):

The malloc function can really put the new structure it allocates anywhere in memory. All you get back is the address in a pointer variable. Even so, providing you link the structures together, it doesn't matter where in memory each record lives.

To delete a record, you merely remove its pointer: Take the pointer in the previous record and have it point at the next record. Lookee:

The pointer in the second structure now points to the fourth structure. The third structure is out of the list. (And, realistically, in memory the records may not be sequential anyway.) How does it work? Pointers!

Yes, it's true, the deleted structure is not removed from memory.

Well, you could use the free function to remove the structure from memory. Mostly, however, programs just re-pack or re-index, which removes the dead records.

Many programs use this technique. Even DOS/Windows: A delete file is merely "skipped over" on disk. This is how UNDELETE programs work; they find the remains of the dead file and then put it back into the list.

The fact that pointers connect these structures will be used in a later lesson to sort them all.

Adding a delete routine to the BANK.C program

To delete a structure from the BANK.C program you need to know three things:

Which structure to delete

The address stored in previous structure's *next pointer (which pointers to the structure you need to delete)

The address of the following structure in memory

So, basically, you take the address stored in the previous structure's *next pointer and store the address of the next structure in there. (Review the illustrations earlier in this Lesson to get an idea of how this works). Only one new variable is needed: a pointer to hold the address of the previous structure in memory. The following code does it all:

```
void deleteAccount(void)
{
   char ch;
   struct account *previous;
```

```
if(first==(struct account *)NULL)
   puts("There are no records to delete!");
else
{
   current=first;
   do
   {
      printf("%5i: %-15s %-15s $%8.2f\n",\
         current->number,\
         current->lastname,\
         current->firstname,\
         current->balance);
      printf("DELETE THIS RECORD?");
      ch = toupper(getch());
      if(ch=='Y')
      {
         puts("Yes!");
         if(current==first)
         {
            first=current->next;
            break;
         }
         else
```

```
            {
               previous->next = current->next;
               break;
            }
         }
         else
         {
            puts("No!");
            previous=current;
         }
      }
      while((current=current->next) != (struct account *)NULL);
   }
}
```

Copy and paste the above code into your BANK2.C program. Remember to prototype! Remember to add a Delete item to the menu! Save your source code to disk as BANK3.C. Or, just Shift+Click here to download a copy of BANK3.C

Compile and Run!

Enter about three or four different records, then try the new Delete command. Delete a record in the middle, then list 'em all. Then try deleting the first and last records to make sure that works.

So How Does It Work?

The deleteAccount function is relatively simple, with only a few new tricks. First, only two variables are required: ch to hold the Y or N key press response, and *previous to hold the address of the previous structure in memory. Most everything else is pulled from other parts of the program.

The entire function works off of a do-while loop exactly the same as the one found in the listAll function – except for the Y or N prompt:

```
printf("DELETE THIS RECORD?");
ch = toupper(getch());
```

If the user press Y, then another if-else structure is used to delete the record.

First, a test is made to see if the current record is the first record:

```
if(current==first)
{
   first=current->next;
   break;
}
```

If the current record is the first record, then all you need to do is change the value stored in the first pointer (the address of the first structure in memory). Just change the value to the address of the next

structure in memory and you're done. The new starting address is stored in the first pointer.

If the current structure is not the first structure, then the following takes place (the tricky part):

```
else
{
   previous->next = current->next;
   break;
}
```

The address of the previous structure is stored in the previous variable. The value of previous->next then points to the current structure and current->next points to the next structure. So to eliminate the current structure, you change the value of previous->next to equal the value of current->next. Get it? (Review the illustration.)

But where did previous come from? From this:

```
else
{
   puts("No!");
   previous=current;
}
```

If the user types N to not delete a record, then the address of the

current structure is stored in the previous variable. This works since the only time that wouldn't happen is when the first record is deleted – and that's already happened earlier in the loop! (Tricky, eh?)

If it's still fuzzy as to why this function works, review it again and think of the illustrations.

Remember that pointers are addresses. In this program, they hold the address of structures at various locations in memory. The next value in each structure is what makes this a linked list – and what allows you to skip over structures without really having to delete anything or shuffle stuff around.

Yes, technically you should use the free function to remove the structure from memory. I will show you how that's done in a later lesson. (It's okay at this point because the structures are small and you're typing them all in; I don't expect you to type in more than 32K of data, okay?)

Bonus C For Dummies Lesson 17-5Lesson 17-5 – Free At Last!

I've been bad. Bad author! Bad! I've done a lot of explaining how malloc works, here on the Web and in the Books, but I left out one little, teensy, weensy thing:

After it's allocated and used, memory should be freed.

The need to free memory shall become painfully apparent to you as your programs grow. All that mallocing stuff gobbles up tons of memory, never giving it back. Especially for the tiny programs you write in these

lessons, 64K of memory is about all you have. Lose it and it's gone.

Now, I haven't been totally bad; for example, the BANK.C programs you've been working with use manual entry. You must type in all the data from the keyboard. So the odds are pretty long that you'll never chew up all memory by typing something in. But if you're reading data from disk (which is the topic of the next lesson), you're in trouble.

A Memory Consuming Example

The following program uses malloc to assign 32K chunks of data. Now malloc returns the address of the chunk of memory it allocates, but in this example, the address isn't saved. I'm just trying to prove a point:

```
Name: GOBBLE.C
#include <stdio.h>
#include <stdlib.h>

void main()
{
   int x;

   for(x=0;x<10000;x++)
   {
      printf("Allocating block %i\n",x);
      if((char *)malloc(32767) == NULL)
```

```
        {
            puts("Unable to allocate memory!");
            exit(0);
        }
    }
}
```

Be a masochist and type the above source code into your editor, saving it to disk as GOBBLE.C. Or, be a smart Web user and Shift+Click here to download a copy of GOBBLE.C

Compile and Run!

You might see something like this:

```
Allocating block 0
Allocating block 1
Unable to allocate memory!
```

If you're lucky, you may get more output than what I got above. But eventually, at some point in the run, the program stops because there is no more memory to allocate. Tragic, eh?

The program uses the (char *) prototype to allocate 32,676 bytes (32K) of memory.

Malloc returns the address of the memory block allocated. That value isn't saved in this program. (No need to.)

When malloc returns NULL, it means no memory could be allocated. This

program uses an if test to determine that.

The actual number of 32K chunks the program allocates depends upon which memory module your compiler is using. For DOS programs, that's typically the "small" or "tiny" module, which gives the program only 64K of memory. If you can, try reconfiguring your compiler to the "huge" or "outrageous" module to see how much memory gets allocated before it's all gone.

Free up some memory with free

It's frustrating to have a PC with megabytes of RAM, yet you can write a silly little program that supposedly gobbles up all memory in only two 32K chunks. Now you're actually facing a conundrum many PC users have: "My computer has megabytes of RAM, how come I'm running out of memory? The truth is, memory must be allocated, used and then freed for use by something else. It's only fair. But DOS (and Windows to some extent) doesn't really, truly manage memory in a PC. For example, Microsoft Word version 6 (for Windows) would typically grab all of a computer's resources when it ran. If you ever worked with Word for a long period of time, eventually you would get the cheery error message "Not enough memory to save document." (That was just bad programming; Word's programmers were hogging up resources and not returning them when they were done.)

To make life better for yourself and your users, it's always nice to free up memory after you've used it. Say, for example, you write a program that reads a file from disk, gathers information, changes settings, and then

closes the file. It's just plain nice to free up the space used by the buffer. And to do that, you use the free function. (Click here for a definition.)

Essentially, the free function de-allocates memory allocated by malloc. And any time you allocate memory that's not used again in a program, you should employ the free function to clear up that memory. The following program, CHEW.C, is a fix-up of GOBBLE.C, which frees the memory and allows all 10,000 blocks to be allocated:

Name: CHEW.C

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
   int x;
   char *buffer;

   for(x=0;x<10000;x++)
   {
      printf("Allocating block %i\n",x);
      if((buffer=(char *)malloc(32767)) == NULL)
      {
```

```
            puts("Unable to allocate memory!");
            exit(0);
        }
        printf("Freeing block %i\n",x);
        free((void *)buffer);
    }
}
```

Type in the above source code or Shift+Click here to download yourself a copy. Compile and Run.

You'll notice that the program runs all the way through, allocating memory and freeing it up so that you really never run out.

free doesn't "erase" memory; it merely flags a chunk of memory as available for re-allocation by another malloc function.

You don't really have to use free every time you use malloc. However, any time you allocate memory that isn't used again, you should free it up.

An example of not freeing memory would be creating a buffer to store information that's used the entire time the program runs. In that case, the memory is freed by the operating system when the program quits.

The next lesson, when it's published here on the Web, deals with allocating and freeing memory using the BANK program.

Nothing puts the twinkle in a programmer's eye like graphics programming.

There's something about it that just seems naturally fun. And there's nothing wrong with that. Graphics are fun. But unlike a lot of other programming -- especially the stuff in C for Dummies -- graphics programming is burdened with math.

Oh no! Math!

Relax. Always remember that the computer does the math. Still, it's up to you to know the equations. If you've been through Algebra in high school you may know some of them. Here are two popular equations you might remember if you weren't sleeping too soundly:

The first is the equation for a line: Y and X are coordinates you'd plot on a graph (or on the screen). M is the slope of the line, up or down, and B is where the line crosses the Y (vertical) access.

The second equation is for plotting a circle, with X squared and Y squared the coordinates and R (also squared) being the circle's radius. Simple. Simple. Simple. (And I bet if you knew you'd be plotting graphics on a PCs screen instead of fumbling around with a pencil on graph paper, you'd probably have paid more attention in class!)

As long as you can work with those boogers in C, tie them into a loop, and translate the results to the giant pixel grid on your computer's monitor, you should be in great shape! (Or great shapes, depending on what your program draws.)

A Quick [Boring] PC Graphics History Lesson

The PC has perhaps the worst graphics standards in the history of computing. Oh, well, maybe the Apple II had a worse system (and God bless the few who really understood it). The Mac is far better. Best? The Amiga. But that's trivia. What's important to know is that the PC has several weird graphics "standards," some of which aren't even standards in the true sense of the word.

MDA. Originally, the PC came with the Monochrome Misplay Adapter (MDA) card. It showed nice, crisp white text on a black background. Or green text on a black background or amber text. (I liked the amber text myself. In fact, my friend Tom still has my old amber monitor.) Color? Nope. Graphics? Nope. Only text.

CGA. If you wanted color, you waited a few months until IBM released the Color Graphics Adapter (CGA). It shows ugly, fuzzy text in one of 16 colors on a background of up to 8 colors. (See Lesson 13-5 for more information on colored text.) Color? Yes, 16 of them in the text mode. Graphics? Yes. There were two graphics modes in the CGA: The low resolution 320 x 200 mode with four colors (black, white, magenta and cyan) or a high resolution 640 x 200 with only two colors, black and white. But that was enough for most PC users.

Unintended moral: The higher the graphics resolution, the fewer colors you get. The lower the resolution, the more colors. It's a nice trade-off since you can use more colors to fool the eye into thinking it sees higher resolution.

Let me be blunt: CGA sucked. The text was horrid. That was one reason I used monochrome for years before switching to VGA in 1988. But I didn't use the MDA. I used yet another graphics standard: Hercules, or monographics. The Hercules card was a clone of the old MDA, but it also had a special graphics mode. Sure, the graphics were black-and-white, but they looked fabulous and the text was much better than CGA. I even had a few games that used Hercules graphics modes. And my Hercules card even displayed italics and underlined fonts in the old DOS WordPefect. (Man, I was in heaven! Ha-ha!)

EGA. To improve upon CGA, IBM created the EGA (Enhanced Graphics Adapter) standard. EGA offered better-looking text over CGA plus a few new graphics resolutions. Oh, but this is all trivia.

VGA. EGA was quickly replaced by the VGA standard, which IBM introduced in 1987 with its PS/2 computers. Actually, there were two VGA standards. The first was MCGA, which came with IBM's PS/1 (the "home" computer). MCGA, which stands for something I can't remember, had lots of graphics

resolutions and color combinations. VGA includes the MCGA standards, so MCGA is pretty much a dud.

SVGA. Eventually, the "industry" improved upon VGA and produced Super VGA or SVGA, the graphics standard nearly all PCs have been sold with since the early '90s.

Why is any of this important?

Because you must set your PC to the proper graphics mode before you can program in graphics. Your programs do this for you automatically. When you run a PC game, it typically switches to another graphics mode on the computer. That's because the programmer has chosen a graphics mode (resolution and colors) proper for the game. When you program in graphics, it's up to you to set the proper mode.

As a preview, follow these steps in Windows:

Start an MS-DOS prompt window. In Windows, run the COMMAND.COM program, or click on the MS-DOS Prompt icon to start up a DOS window.

Switch to full-screen mode. If the DOS prompt is in a window in Windows, then press Alt+Enter so that it fills the screen -- just like the good old days!

Type the following:

```
MODE 40
```

Press Enter. Welcome to 40-column mode! This is an old CGA graphics mode, designed primarily so that people could use their PCs with a TV set as a

monitor. (I wonder if anyone ever did?) In fact, the DIR command was formatted for 40-column output primarily for this reason. (They changed the DIR format starting with DOS version 6.)

Type EXIT to return to Windows.

If you're using an SVGA video system, then your computer has 15 different modes you can use to display text or graphics. You pick the resolution and the number of colors and go from there. The following table lists the graphics modes you have to choose from. (If you don't see a mode listed in the table, then the SVGA adapter cannot produce it; these are typically MDA, PCjr and EGA graphics modes on one uses anymore anyway anyhow.)

ModeTypeResolutionColors

0x00Text40 x 2516

0x01Text40 x 2516

0x02Text80 x 2516

0x03Text80 x 2516

0x04Graphics

(alt. color)320 x 2004

0x05Graphics320 x 2004

0x06Graphics640 x 2002

0x07Graphics720 x 350

720 x 400*2

0x0dGraphics320 x 20016

0x0eGraphics640 x 20016

0x0fGraphics640 x 3502

0x10Graphics640 x 3504 or 16**

0x11Graphics640 x 4802

0x12Graphics640 x 48016

0x13Graphics320 x 200256

* The vertical resolution is set using function 0x12 of Interrupt 0x10 -- something I'll never mess with.

** Only EGA adapters with 64K of video RAM display 4 colors in mode 0x10.

My favorite mode is 0x13, which is an old MCGA mode available on all VGA adapters. It uses a rather low resolution, 320 x 200 pixels, but has a ton of colors: 256.

More modes are available than those shown above. The table lists only the standard VGA modes. I know that Super VGA has even more colors and resolutions. If you get into this, consider buying a technical manual on VGA graphics.

Set Some Dem Modes!

Wow! This chapter has way too much text. But the point's been made: Before you do doodle with graphics, you must first set the graphics mode.

Actually, you should do several things: You should discover what graphics mode the PC is currently in, save it, set the mode to what you want, then reset the mode back to what it was when you're done. That's a lot of work, but it's being nice to your user.

Borland C++ has some graphics mode setting commands, but Microsoft C and all other C language compilers probably don't. Because of that, I'm going to use in-line Assembly language to run the graphics basics for this Chapter. (See Chapter 16 for more information.)

To set the mode you use the PC's Video Interrupt, 0x10, to call the BIOS. Here's the routine:

```
void set_mode(unsigned char mode)
{
   _asm mov ah,00
   _asm mov al,mode
   _asm int 10h
}
```

The set_mode function is called with an unsigned char value, from 0 through 255, which you would pluck from the table shown earlier in this Lesson. That mode is saved into the AL register, the value zero put into the AH register and the function sped off to Interrupt 10h (which is Assembler for 0x10). Voila, ze mode is set.

Of course, fetching the current graphics mode is also important. Otherwise you may leave your user in some low-rez dorky mode and they won't have a clue as to how to get back to normal. (The MODE 80 command does the trick in DOS.) Here is the function to fetch the current graphics mode:

```
unsigned char get_mode(void)
{
```

```
    unsigned char mode;

    _asm mov ah,0fh
    _asm int 10h

    _asm mov mode,al
    return(mode);
}
```

The value mode is returned from Interrupt 0x10 in the AL register. That's saved into the mode variable and returned from the get_mode function.

And now, your program:

```
Name: GRAFMODE.C
#include <stdio.h>

#define NORMAL 0x03
#define LOWGRAF 0x04
#define HIGRAF 0x06
#define VGA256 0x13

unsigned char get_mode(void);
void set_mode(unsigned char mode);
```

```
void main()
{
    unsigned char save;

//Save the current mode (which is probably 3)

    save = get_mode();

    printf("You're in mode %i now.\n",save);
    getchar();

//Switch to low rez CGA mode

    set_mode(LOWGRAF);
    puts("Hello Low CGA Graphics Mode!");
    getchar();

//High rez CGA mode

    set_mode(HIGRAF);
    puts("Hello High CGA Graphics Mode!");
    getchar();
```

```
//Low rez MCGA colorful mode

    set_mode(VGA256);
    puts("Hello VGA 256 Graphics Mode!");
    getchar();

//Restore mode before exiting

    set_mode(save);
}

unsigned char get_mode(void)
{
    unsigned char mode;

    _asm mov ah,0fh      //Return mode value
    _asm int 10h         //Video interrupt

    _asm mov mode,al     //Save mode
    return(mode);
}

void set_mode(unsigned char mode)
```

```
{
   _asm mov ah,00       //Set mode
   _asm mov al,mode     //mode value
   _asm int 10h         //Video interrupt
}
```

Type in the above source code, or Shift+click on this link to download a copy.

Compile!

Run the program in a DOS window (if you're using Windows). And switch the window over to full-screen by pressing Alt+Enter if you need to. (Windows will change modes otherwise, which is annoying).

Now, run. You should see the following:

You're in mode 3 now

You're probably in mode 3, which is the standard 80 x 25 text mode.

Press Enter.

Clunky, huh? Press Enter.

That's the old "high" resolution mode. Ick. Press Enter

Finally you have the high color VGA image. That's the one you'll be working with in this Lesson. Press Enter and your system will be restored back to its original graphics mode.

Did you notice that the printf function works in graphics modes? Pretty much so. You can mix graphics and text any time. The cursor moves the same on a graphics screen as it does on a text screen. Just use any C language text function to display text.

Windows is actually capable of viewing all these modes in a DOS window on the screen. Just press Alt+Enter to switch between full-screen and windowed mode. I prefer to work full screen when I do graphics.

Feel free to modify the program to display other modes shown in the Table.

To see if the program really does save the graphics mode, type MODE 40 before you run it. If the program restores the 40-column mode when it's done, then you've just done your users a favor.

If you're using DJGPP, then you'll need to use the ATT nottaion, not the Intel Assembly notation shown here. Visit http://www.delorie.com/djgpp/ for more information. (Sorry!)

Bonus C For Dummies Lesson 18-2Lesson 18-2 – Hello, Pixel Fairy!

(If you have Microsoft Visual C++, click the icon below.)

The graphics screen is a big grid. Well, the text screen is a big grid, too. So if you've been plotting out text on the screen using rows and columns, you should get used to plotting graphics on its grid in no time. But with graphics, you're not plotting characters, you're plotting pixels.

All graphics, no matter how complex they look or what they do, are merely a collection of pixels on the screen. So when you start creating graphics, you start by plotting pixels on a grid. And since you're programming, you can afford to have the computer do all the plotting work for you.

Every graphics program you've ever seen -- including Windows itself -- starts with a basic routine to plot pixels on the screen.

Other routines, for drawing lines and circles and such, are all functions designed to plot pixels in a certain manner.

Yes, even animation involves plotting pixels.

It's Just a Big Grid

Pixels have three attributes associated with them: X position, Y position and color. (Again, this is like text, where each character has a row and column position as well as a character code value.)

Pixels have an X (horizontal) position and a Y (vertical) position. The pixel grid starts in the upper left corner of the screen at position 0,0. This is the same for all graphics resolutions. The maximum values for X and Y differ from graphics mode to graphics mode. In the high color VGA

mode 0x13, the maximum X value is 320 and the maximum Y value is 200. (Incidentally, that's the size of the graphic image shown above: 320 x 200 pixels.)

In addition to the X and Y locations, pixels also have a color value. Like the resolution, this value depends on the graphics mode. Color values range from 2 (on and off, or "black" and "white") on up to 256. The colors assigned to those values vary with the graphics mode.

That's the basics of all graphics. No matter what you see on the screen, it's basically a collection of pixels set at certain locations in the graphics grid, each pixel tuned to a certain color.

Pixel is short for "picture element." It's a dot on the screen.

Pixar is the name of the company that produced Toy Story.

X locations go from zero upward as you move from left to right across the screen. This is the same as you learned in Algebra or Geometry class, plotting on graph paper.

Y locations go from zero upward as you move from the top down on your screen. This is different from the graph paper you used in school (where Y started at the lower left corner of the graph).

X values correspond to column values on the text screen; Y values correspond to row values on the text screen.

Plotting Pixels

Before you can do any graphics on any computer, you must have a function that plots a single pixel on the screen. The good news is that it's simple

to do. The bad news is that there are dozens of ways to do it.

For example, one programmer I know likes to plot pixels by writing directly to video memory. Yikes! You can also plot pixels by manipulating the graphics hardware inside your computer. I have technical books full of VGA and SVGA documentation that tell über nerds how to manipulate the screen and do all sorts of twisted things to it. Again: Yikes!

Some folks opt to get graphics libraries for their programming whims. For example, you can buy a library full of high-speed graphics routines -- everything from plotting pixels to drawing lines to creating animations, all programmed for you and ready to use.

For this chapter, I'm going to show you the tried and true method for plotting pixels: using the PC's BIOS with in-line Assembly. This isn't the fastest way to do things (people into computer graphics are into speed), but it's standard. Here's the function:

```
void pset(int x,int y,unsigned char color)
{
   _asm mov ah,0ch
   _asm mov al,color
   _asm mov cx,x
   _asm mov dx,y
   _asm int 10h
}
```

The pset function is called with three values: x, the pixel's X

coordinate; y, the pixel's Y coordinate; and color, which sets the pixel's color. These values are stuffed into the proper microprocessor registers using in-line assembler. Register AH is loaded with the function number, 0x0C, which writes a pixel to the screen, and Video Interrupt 0x10 is called.

Note that the pset function doesn't check to see if value numbers are used. That's good in a way, since the BIOS function writes pixels in any mode; the maximum value for X could be anything from 320 to 400 to 640 to 1024 or more. Of course, the pixel appears only in the proper graphics mode.

I'll get into color values in a few pages, but generally speaking, zero is always black (or turning a pixel off). Values from 1 on up could be any color, though.

Time for a program to run pset through it's paces:

Name: PIXELS1.C

```
#include <stdio.h>

#define VGA256 0x13
#define XMAX 320
#define YMAX 200

void pset(int x,int y,unsigned char color);
```

```
unsigned char get_mode(void);
void set_mode(unsigned char mode);

void main()
{
    unsigned char save,c;
    int x,y;

/* Initialze things */

    save = get_mode();      //Save current screen mode
    set_mode(VGA256);

/* Main routine */

    x=XMAX/2;
    y=YMAX/2;
    c=1;
    pset(x,y,c);

    getchar();

/* Reset stuff for end */
```

```
    set_mode(save);        //Restore screen mode
}

void pset(int x,int y,unsigned char color)
{
    _asm mov ah,0ch
    _asm mov al,color
    _asm mov cx,x
    _asm mov dx,y
    _asm int 10h
}

unsigned char get_mode(void)
{
    unsigned char mode;

    _asm mov ah,0fh
    _asm int 10h

    _asm mov mode,al
    return(mode);
}
```

```
void set_mode(unsigned char mode)
{
   _asm mov ah,00
   _asm mov al,mode
   _asm int 10h
}
```

Type in the above source code or, better still, Shift+click here to download a copy.

Compile! Run! The screen goes graphical and you should see a single, blue pixel -- one dot -- in the middle of the screen. That's one tiny building block with which you can do anything you can visualize on your PC's graphics screen.

As you know, I'm a big fan of defines. In the PIXELS1.C program there are three of them: VGA256, XMAX and YMAX. VGA256 is the graphics mode and XMAX and YMAX represent the maximum horizontal and vertical resolutions for that mode, 320 and 200 respectfully.

Using the XMAX and Y MAX values in the program is a lot easier than having to remember the values every time you write a graphics equation. For example, if you properly define XMAX as the screen's horizontal resolution, then you can always use XMAX or XMAX/2 to represent the far left or center of the screen -- no matter which graphics mode you're in.

Refer to the Table in Lesson 18-1 for more information on graphics modes

and resolutions.

More Dots

Heck ALL the Dots!

The following program uses a nested for loop to paint the entire low rez VGA screen with all 256 possible color combinations. Dazzle your friends!

Name: PIXELS2.C

```
#include <stdio.h>

#define VGA256 0x13
#define XMAX 320
#define YMAX 200
#define MAXCOLORS 256

void pset(int x,int y,unsigned char color);
unsigned char get_mode(void);
void set_mode(unsigned char mode);

void main()
{
   unsigned char save,c;
   int x,y;
```

```
/* Initialze things */

    save = get_mode();      //Save current screen mode
    set_mode(VGA256);

/* Main routine */

    for(c=0;c<MAXCOLORS;c++)
    {
        for(y=0;y<YMAX;y++)
        {
            for(x=0;x<XMAX;x++)
            {
                pset(x,y,c);
            }
        }
        getchar();
    }

/* Reset stuff for end */

    set_mode(save);        //Restore screen mode
}
```

```
void pset(int x,int y,unsigned char color)
{
   _asm mov ah,0ch
   _asm mov al,color
   _asm mov cx,x
   _asm mov dx,y
   _asm int 10h
}

unsigned char get_mode(void)
{
   unsigned char mode;

   _asm mov ah,0fh
   _asm int 10h

   _asm mov mode,al
   return(mode);
}

void set_mode(unsigned char mode)
{
   _asm mov ah,00
```

```
    _asm mov al,mode
    _asm int 10h
}
```

Doggedly type in all that source code, or just Shift+Click here to download a freshly pressed copy, written by the author.

Compile! Run!

The nested for loops paint the entire screen, top to bottom. If you have a slower PC, you may actually see the pixels fill the screen. (Yes, this isn't the most efficient method for white washing a computer screen.)

Press ENTER to view the next color screen. Here's how the colors map out:

0 to 15 - Standard CGA colors

Black, Blue, Green, Cyan, Red, Magenta, Brown, Gray, Dark Gray, Bright Blue, Bright Green, Bright Cyan, Bright Red, Bright Magenta, Yellow, Bright White. (Incidentally, these are the same codes and colors for color text on the screen. See Lesson 13-5.)

16 to 31 - Grayscale

Shades of gray from almost black on up through almost white.

32 to 103 - High intensity rainbow

Shades of Blue, Red and Green in high, moderate and low saturation.

104 to 175 - Moderate intensity rainbow

Shades of Blue, Red and Green in high, moderate and low saturation.

176 to 247 - Low intensity rainbow

Shades of Blue, Red and Green in high, moderate and low saturation.

248 to 255 - Black

Black, black, black, black, black, black, black and black again.

Now you don't have to press the ENTER key 255 times to get the whole idea. If you tire of it, press Ctrl+C to get out of the loop. You'll then have to type MODE 80 to return to the normal text screen, or just keep working in 40 column mode if you enjoy it.

Actually, if you're interested in viewing the VGA colors, you can refer to a related program: click here.

The Old Randomizer Star Field Trick

Now you can get really silly with pixels. Don't worry about stringing them into lines and circles yet. Instead, take that old chestnut the random start field program out of the closet. This program combines the randomizing functions from Volume I with the pset function introduced in this Lesson. The results are predictable, but fun to watch nonetheless.

Name: PIXELS3.C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>                    //for seeding randomizer

#define VGA256 0x13
#define XMAX 320
#define YMAX 200
```

```
#define MAXCOLORS 256

void pset(int x,int y,unsigned char color);
unsigned char get_mode(void);
void set_mode(unsigned char mode);
int rnd(int range);
void seedrnd(void);

void main()
{
   unsigned char save,c;
   int loop,x,y;
   long int delay;

/* Initialze things */

   seedrnd();              //Seed the randomizer
   save = get_mode();      //Save current screen mode
   set_mode(VGA256);

/* Main routine */

//Plot a pixel in a random location with a random color
```

```
    for(loop=0;loop<1000;loop++)
    {
        x=rnd(XMAX);
        y=rnd(YMAX);
        c=rnd(MAXCOLORS);
        pset(x,y,c);
        for(delay=0;delay<50000;delay++);
    }
    getchar();

/* Reset stuff for end */

    set_mode(save);        //Restore screen mode
}

void pset(int x,int y,unsigned char color)
{
    _asm mov ah,0ch
    _asm mov al,color
    _asm mov cx,x
    _asm mov dx,y
    _asm int 10h
}
```

```
unsigned char get_mode(void)
{
   unsigned char mode;

   _asm mov ah,0fh
   _asm int 10h

   _asm mov mode,al
   return(mode);
}

void set_mode(unsigned char mode)
{
   _asm mov ah,00
   _asm mov al,mode
   _asm int 10h
}

/* Random number routines stolen from Volume I */

int rnd(int range)
{
        int r;
```

```
        r=rand()%range;                 //spit up random num.
    return(r);
}

void seedrnd(void)
{
    srand((unsigned)time(NULL));    //seed randomizer
}
```

Type the above source code into your editor, or Shift+Click here to download yourself a copy.

Compile! Run!

Are the starts out tonight?

I don't know if it's cloudy or bright . . .

Problem: the program loops only 1000 times. Can you imagine a better way to halt it on your own? Something that would cause the stars to twinkle until you pressed a key on the keyboard, say? I'll leave it to you to divine a solution to that problem.

You can adjust the value of the delay for loop to make the stars twinkle slower or faster. This depends on the speed of your computer.

Wow! Does this program remind you of the DIBBLE.C program from the end of Volume I? Same concept, but pixels instead of happy faces.

Speaking of old programs, consider modifying the old DRUNK.C program from the end of Volume I into a program that displays pixels on the

screen. You'll need to adjust the maximum X and Y values, plus change the drunk into a cursor of whatever color. Also, you'll need to overwrite the drunk with color 0 (zero), which is black. See what you come up with; in a few lessons I'll be tackling this program myself using animation techniques.

Bonus C For Dummies Lesson 18-3Lesson 18-3 – Loverly Lines

A line is defined as connecting two points. A line is also defined as that thing you wait in while you're at Disneyland. And not just Disneyland, but Universal Studios, too. My son just went to Universal Studios, Los Angeles to go on the Jurassic Park ride. The line was 3½ hours long. That line, too, was connecting two points: the point where my son was and the point where people were boarding the 10-minute long ride. Too much.

Back in geometry class, you drew line by first defining the two points on graph paper. You drew a dot at the first location and a dot at the second location. Then you used a ruler to connect the two dots. Amazing. On a computer screen, which works like graph paper, you take the starting location of the line and the ending location and then pray that there is some simple line-drawing algorithm you can use like a ruler to connect them. That's the gist of it. And don't worry -- this is still C for

Dummies. There won't be any math here.

Yes, I have the line drawing algorithm all worked out. It's explained in the next section.

The line requires two locations on the computer screen. I'll refer to the starting location as X1, Y1 and the ending location as X2, Y2.

All polygons are created with lines. So a rectangle, for example, is merely a line drawn between four sets of coordinates.

It was really my mom's fault for taking my son to Universal Studios on a Sunday in July, the last day of a promotional special. The park was jammed. The line was 5 hours long to get into the Backdraft exhibit.

The Line Drawing Equation

All lines are drawn using the simple line drawing formula you probably forget from algebra class:

You don't have to read this explanation: Y and X are the vertical and horizontal coordinates of the line; for every Y position on the line there is a corresponding X position. M is the slope of the line, or the ratio of Y over X. And B is where the line crosses the Y-axis. Enough!

The idea is to convert that simple line-drawing equation into something the C compiler can understand. After doing that, you work it out so that the function can also plot pixels for all the X and Y coordinates of the line. So, using the vast skills you honed after years of schooling, you decided to look it up in a book where some propeller-head has already

figured out the answer. But I won't let you get off that easy. At least not all at once.

Drawing a Straight Line

The simplest type on line to draw is the straight line. For a horizontal line, you would plot pixels from X1 to X2 with the same value of Y – a simple for loop. For a vertical line, you plot pixels from Y1 to Y2 with the same value of X – another simple for loop. However, a vertical line has a slope of infinity (or some other wacky, nonsensical value). So it must be treated as a special case, one that cannot follow the y=mx+b formula. (Yes, an exception.)

Name: VERTLINE.C

```
#include <stdio.h>

#define VGA256 0x13
#define XMAX 320
#define YMAX 200
#define RED 4

void pset(int x,int y,unsigned char color);
unsigned char get_mode(void);
void set_mode(unsigned char mode);
```

```
void line(int x1,int y1,int x2,int y2,unsigned char color);

void main()
{
    unsigned char save;
    int x1,y1,x2,y2;

/* Initialze things */

    save = get_mode();      //Save current screen mode
    set_mode(VGA256);

/* Main routine */

    printf("Coordinates [x1,y1,x2,y2]:");
    scanf("%i,%i,%i,%i",&x1,&y1,&x2,&y2);
    printf("(%i,%i) to (%i,%i)",x1,y1,x2,y2);
    getchar();

    line(x1,y1,x2,y2,RED);
    getchar();

/* Reset stuff for end */
```

```
    set_mode(save);        //Restore screen mode
}

void line(int x1,int y1,int x2,int y2,unsigned char color)
{
    int x,y;

    for(y=y1;y<=y2;y++)
        pset(x1,y,color);
}

void pset(int x,int y,unsigned char color)
{
    _asm mov ah,0ch
    _asm mov al,color
    _asm mov cx,x
    _asm mov dx,y
    _asm int 10h
}

unsigned char get_mode(void)
{
    unsigned char mode;
```

```
    _asm mov ah,0fh
    _asm int 10h

    _asm mov mode,al
    return(mode);
}

void set_mode(unsigned char mode)
{
    _asm mov ah,00
    _asm mov al,mode
    _asm int 10h
}
```

This program uses the pset function from Lesson 18-2 to help you plot the line. In addition to the line drawing function, I added code that lets you input the starting and ending coordinates for the line. Now don't get cocky with your values! It's easy to crash this program by typing in negative numbers. (I'll leave it up to you to bulletproof the program on your own.)

Type in the above source code or, if you're not a masochist, Shift+click here to grab a copy the author typed in himself.

Compile! Run!

Type in the four sets of coordinates, separating each by commas. Oh, heck, just type in the following:

160,0,160,200

That's 160 for both values of X, which puts the line in the center of the screen. Y1 is equal to 0 (at the top of the screen) and Y2 is equal to 200 (bottom). Press Enter to see the results. Press Enter again to quit.

Now. The program can draw any vertical line on the screen. It ignores the X2 coordinate (which your compiler may alert you to when you created it), but uses the two Y coordinates to draw the line. Alas, this isn't without its problems.

Run the program again. This time type in the following for the coordinates:

160,200,160,0

This is essentially the same thing, but with the coordinate sets reversed. But will the program draw the line up? Press Enter to find out.

Nope. And it shouldn't surprise you. Review your program's for loop:

for(y=y1;y<=y2;y++)

The loop never works, not once. If Y1 is greater than Y2, then the for loop quits before it loops even once. And it should because that's the way the C language works. But, obviously, that's not going to fit into the grand scheme of things when you draw a line. Obviously there will be times when Y1 is greater than Y2 and your program will still need to draw the line.

The solution? Reverse the coordinates. Swap them. Add the following function to the VERTLINE.C program:

```
/* Swap the values of two variables */


void swap(int *a,int *b)
{
   int temp;


   temp = *a;
   *a = *b;
   *b = temp;
}
```

Defining the variables as pointers here allows you to swap their values without having to swap names. (It also allows the function to be used elsewhere in the line function, which you'll discover later in this lesson.)

Remember to prototype the function at the start of your source code:

```
void swap(int *a,int *b);
```

And finally, you'll need to modify the line function to take care of the case when Y2 is larger than X2. Here's how that function looks:

```
void line(int x1,int y1,int x2,int y2,unsigned char color)
{
   int x,y;
```

```
    if(y1 > y2)
        swap(&y1,&y2);

    for(y=y1;y<=y2;y++)
        pset(x1,y,color);
}
```

A simple if test is all that's required to see if Y1 is greater than Y2.

If so, the values are swapped. (Refer to Chapter 10 in Volume II for more information on how the ampersands are used when passing values to a pointer function like swap.)

Make the proper adjustments to your program. Save. Compile. Run.

Now type in the following values for your coordinate sets:

160,200,160,0

The value of X is merely determines where the vertical line will slice the screen. However, Y1 is greater than Y2 in this example. Will it work?

Press the Enter key to find out.

Well, you knew it would.

Drawing Lines Hither, Thither and Yon

There are actually many line drawing algorithms, but y=mx+b is the most common. The puzzlement is how to shift the values of X1,Y1 and X2,Y2 into that equation. The core code you can probably figure out yourself:

```
for(x=x1;x<=x2;x++)
{
```

```
    y = m*x+b;
    pset(x,y,color);
}
```

A for loop whizzes the value X from X1 to X2. Then the value for Y is calculated using y=mx+b. Finally, the pset function is used to plot the actually points on the screen, X and Y. Cinchy!

Of course, where does M come from? And how do you calculate B?

M is the slope of the line, or the ratio of the change in Y over the change in X. That's calculated using the formula shown above.

From your basic C knowledge, you should remember that anytime you use division, you'll be dealing with a floating point number. So right off the bat, you should know that M is a floating point variable. And you, therefore, must typecast the results of the Y over X thing. But don't think! Just look at this handy translation into C:

```
m = (float)(y2-y1)/(float)(x2-x1);
```

To solve for B you end up with something like the following:

Looks okay to me, boss! And it translates easier into C:

```
b = y1 - (m*x1);
```

Since the equation for B involves the value of M, B should also be a floating point number. So both of them need to be declared at the start of the new line function as floating point numbers. In fact, here is the

entire new line function for you:

```
void line(int x1,int y1,int x2,int y2,unsigned char color)
{
    int x,y;
    float m,b;

// Plot straight vertical line

    if (x2==x1)
    {
        if(y1 > y2)
            swap(&y1,&y2);

        for(y=y1;y<=y2;y++)
            pset(x1,y,color);

        return;
    }

    m = (float)(y2-y1)/(float)(x2-x1);
    b = y1 - (m*x1);

    for(x=x1;x<=x2;x++)
```

```
    {
      y = m*x+b;
      pset(x,y,color);
    }
}
```

Above, you'll note that the new floating point M and B variables are defined. Then comes the code for plotting a straight vertical line. That's determined by matching the values of X1 and X2. If they're equal, then you have a vertical line. The rest of the code between the if statements' brackets is what you've already gone over in this lesson.

Next comes the new stuff. First the value of M is calculated, then the value of B. Then a for loop steps through the points in the line using the old y=mx+b chestnut. pset slaps down all the pixels in the proper spot.

Make the appropriate changes in your VERTLINE.C source code, adding in the new stuff for the line function as shown above. Save your new creation to disk under the name LINE1.C. Shift+Click here to download a copy of LINE1.C.

Compile and run!

Type in the following as your first sample set of coordinates:

0,0,320,200

Press Enter. Thwoop! A nice red line from the upper left corner down to the lower right. Press Enter to quit.

Now you can mess with drawing lines all the doo-dah-day if you like.

Unfortunately, you're going to eventually encounter the following problem.

Run the program again and type in these coordinates:

160,200,0,0

That's a line from the bottom center of the screen up and back to the upper left corner. Press Enter to draw the line.

Or not, as the case may be. You see, it's the for loop again. It expects the values of X to crawl ahead from little to big. When you try to draw a line in the other direction, the loop never loops. This isn't a major hassle, though. It simply means you need to check to see if the coordinates are "backwards" and then re-reverse them using the swap function.

The following, final rendition of the line program, LINE.C, does the job completely and utterly:

Name: LINE.C

```
#include <stdio.h>

#define VGA256 0x13
#define XMAX 320
#define YMAX 200
#define RED 4
```

```
void pset(int x,int y,unsigned char color);
unsigned char get_mode(void);
void set_mode(unsigned char mode);
void line(int x1,int y1,int x2,int y2,unsigned char color);
void swap(int *a,int *b);

void main()
{
   unsigned char save;
   int x1,y1,x2,y2;

/* Initialze things */

   save = get_mode();      //Save current screen mode
   set_mode(VGA256);

/* Main routine */

   printf("Coordinates [x1,y1,x2,y2]:");
   scanf("%i,%i,%i,%i",&x1,&y1,&x2,&y2);
   printf("(%i,%i) to (%i,%i)",x1,y1,x2,y2);
   getchar();
```

```
    line(x1,y1,x2,y2,RED);
    getchar();

/* Reset stuff for end */

    set_mode(save);        //Restore screen mode
}

void line(int x1,int y1,int x2,int y2,unsigned char color)
{
    int x,y;
    float m,b;

// Plot straight vertical line

    if (x2==x1)
    {
        if(y1 > y2)
            swap(&y1,&y2);

        for(y=y1;y<=y2;y++)
            pset(x1,y,color);
```

```
        return;
    }

// Reverse X1 and X2 if needed

    if(x1 > x2)
    {
        swap(&x1,&x2);
        swap(&y1,&y2);
    }

    m = (float)(y2-y1)/(float)(x2-x1);
    b = y1 - (m*x1);

    for(x=x1;x<=x2;x++)
    {
        y = m*x+b;
        pset(x,y,color);
    }
}

/* Swap the values of two variables */
```

```
void swap(int *a,int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

void pset(int x,int y,unsigned char color)
{
    _asm mov ah,0ch
    _asm mov al,color
    _asm mov cx,x
    _asm mov dx,y
    _asm int 10h
}

unsigned char get_mode(void)
{
    unsigned char mode;

    _asm mov ah,0fh
```

```
    _asm int 10h


    _asm mov mode,al
    return(mode);
}


void set_mode(unsigned char mode)
{
    _asm mov ah,00
    _asm mov al,mode
    _asm int 10h
}}
```

Shift+click here to download a final copy for your files. Then compile and run. The program draws lines between any two points on the screen. No problem.

The new chunk of code in the line function merely checks to see if X1 is greater than X2. If so, then both coordinates are swapped, X1,Y2 and X2,Y2. After all, it doesn't matter which direction the line goes, merely that it connects those two points. Again, the handy swap function is used to swap both coordinate sets.

The next lesson continues your graphics studies with some polygons drawn using the basic line function.

On your own, you should try to devise a program that draws lines between

random coordinates using random colors.

Another program to consider: a line that bounces around the screen (like a cheesy screen saver) in different colors. I did one of these on my own. Shift+click here to see a copy, but don't cheat! Come up with your own first before you compare it to mine. (And remember that there are many different ways to work the same program.)