CHAPTER XVII

User Interface—Screen and Keyboard

For a program to be useful, it must have some way of communicating the results, or its needs, to the user. To accomplish communication with the user of the program, the C language comes with a robust library of routines known collectively as the standard input/output library. This chapter looks closely at these routines and answers some frequently asked questions about them.

XVII.1: Why don't I see my screen output until the program ends?

Answer:

Sometimes, depending on the compiler and operating system you are working on, output will be buffered. "Buffered" means that any output you send to a device, whether the screen, the disk, or the printer, is saved until there is a large enough amount to be written efficiently. When enough output has been saved, it is written to the device as one block.

This process can cause two problems for the programmer unaware of its effects. First, output might not be displayed on-screen until sometime after your program sends it. This effect might be a problem if the programmer is trying to keep track of exactly what his program is doing, and when.

FREE EBOOKS, NOTES , VIDEOS & PLACEMENT MATERIAL

For All Companies placement Material

@placementclasses

For CAT Exam Preparation Material

@cat_classes

For GATE Exam Preparation Material

@gate_classes

For Engineering Books & Material

@cs_ebooks

Codes of Following Programming Languages

C

@c_examples

C++

@cpp_examples

Java

@java_examples0

Python

@python_examples

The second, and more insidious, problem can occur when your program prints some prompt for the user and waits for input. There is no guarantee in the ANSI C standard that attempting to get input from the user will flush the output buffer. Therefore, the message you sent to the screen might not be displayed before your program tries to get input. The user thus does not know that he is being prompted for input; as far as he can tell, your lovely program has just suddenly stopped working.

How can this problem be fixed? There are two ways. The first way to fix the problem is to insert the line

```
setvbuf( stdout , NULL , _IONBF , 0 );
```

at the start of the program, before any output has been printed. This code has the effect of completely unbuffering output to the screen. After this command is executed, every character that is sent to the screen is printed immediately as it is sent.

This method is a convenient way to solve the problem, but it is not ideal. Without getting into a technical discussion of screen input and output, I'll just state that there are good reasons why screen output is buffered, and that you might want to leave it so.

This brings us to the other way of solving the output-buffering problem. The command `fflush()`, when invoked on an output buffer, causes it to empty itself even if it is not full. Therefore, to solve the screen buffering problem, you can simply insert the command

```
fflush( stdout );
```

whenever you want the output buffer to be flushed. It would be appropriate to flush the output buffer before requesting input from the user, or before the program goes into an extensive computation that will delay it for a time. This way, whenever the program pauses, you'll know why.

Cross Reference:

None.

XVII.2: How do I position the cursor on the screen?

Answer:

There is no method in the C standard for positioning the cursor on the screen. There are many reasons for this omission. C is designed to work across a broad range of computers, many of which have different screen types. On a line terminal, for example, it is impossible to move the cursor up. An embedded system might even be written in C, in which case there might not be a screen at all!

That being said, there is still a use for positioning the cursor on the screen in your own programs. You might want to give the user an attractive visual that is possible to display only by moving the cursor around. Or you might even want to attempt a little animation using the print commands. Despite the lack of standards for this action, there are several ways the problem can be addressed.

First, the writer of the compiler can supply a library of routines that handle screen output specific to that compiler. One of these routines will certainly be the positioning of the cursor. This is arguably the worst solution, because every manufacturer is free to make his own implementation. Therefore, a program written

with one compiler will almost certainly need to be rewritten if it moves to another compiler, much less another machine.

Second, a standard set of library functions can be defined that the compiler writer can implement for his compiler. This is the root of the popular curses package. Curses is available for most machines and compilers. Therefore, a program written to use curses for screen output works on most other computers and compilers.

Third, you can use the fact that the device to which you are printing can interpret the characters you are sending in a particular way. There is a standard way in which terminals (or screens) should be made to interpret characters sent to them, the ANSI standard. If you assume that your computer is ANSI compliant, you can print the right characters to manipulate your screen into positioning the cursor in the places you want, among other actions.

Cross Reference:

None.

XVII.3: What is the easiest way to write data to the screen?

Answer:

The C programming language contains literally hundreds of functions designed to write data to the screen. It can be difficult to decide which of them might be "best" for writing to the screen at a particular time. Many programmers simply pick one or two of the printing functions and use them exclusively. This is an acceptable practice, although it means that the programmer might not always produce the best possible code.

What a programmer should do is review what each printing function is designed for, and what it does best. Thereafter, whenever he needs to print something to the screen, he can use the function that best suits his needs. He might even create some printing functions of his own.

Learning to correctly use the printing functions contained in the standard C library is part of the first step to becoming a truly proficient programmer. Let's examine some of the functions in detail.

```
printf( <format string> , variables );
```

`printf` is the most widely used printing function. Some programmers use it exclusively to send text to the screen. Despite this fact, the function was designed only to print formatted text to the screen. In fact, `printf` is short for "print formatted." Formatted text is text that contains not just the character string that you placed into your code, but also numbers, characters, and other data dynamically created by your program. Additionally, it can make these appear in a particular way. For instance, it can make real numbers appear with a specific number of digits on either side of the decimal point. For this purpose, the `printf` function simply cannot be beat!

Why, then, might one choose not to use `printf`? There are several reasons.

The first reason is that the programmer might want to make what he is doing more clear. Perhaps the programmer is interested only in performing a small subset of the actions provided by the `printf` function. In this case, he might want to use a specific function that provides just that subset, such as

```
putchar( char );
```

This function is designed to send one character to the screen. It is great if this is what you want to do, but it's not really good for anything else. However, by using this function, you are making it exceedingly clear that what this section of code is doing is sending single characters to the screen.

```
puts( char * );
```

This function writes a string of characters to the screen. It does not attempt to accept extra data, as `printf` does, and it does not process the string that has been passed to it. Again, by using this function, you make it abundantly clear what your code is doing.

The second reason the programmer might choose not to use `printf` is that he might want to make his code more efficient. The `printf` function has a lot of overhead; what this means is that it needs to do a great deal of work to perform even a simple operation. It needs to search the string that has been passed to it for format specifiers, it needs to check how many arguments were passed to it, and so forth. The two functions already presented here do not have such overhead. They have the potential for being substantially faster. This fact is not very important for most programs that write data to the screen. It can become an important issue, however, if you are handling large amounts of data from a disk file.

The third reason not to use `printf` is that the programmer wants to reduce the size of his executable. When you use a standard C function in your program, it must be "linked in." This means that it must be included into the executable file you are producing. Whereas the code for the simple printing functions, such as `putch` or `puts`, is quite small, the code for `printf` is substantially larger—especially because it might include the other two as a matter of course!

This consideration is probably the least important of those presented so far. Still, if you are using a static linker and you want to keep your executable files small, this can be an important trick. For example, it is very desirable to keep the size of TSRs and some other programs to a minimum.

In any case, the programmer should decide which functions he needs to use based on his purposes.

Cross Reference:

None.

XVII.4: What is the fastest way to write text to the screen?

Answer:

Usually, you are not overly concerned with the speed with which your program writes to the screen. In some applications, however, you need to be able to write to the screen as quickly as possible. Such programs might include these:

- Text editors. If you cannot draw to the screen very quickly, scrolling of the screen due to the user entering text, as well as other actions, might be too slow.
- Animated text. It is common to print characters quickly over the same area to achieve animation. If you cannot print text to the screen very quickly, this animation will be too slow and will not look very good.

- Monitor programs. Such a program might continually monitor the system, another program, or some hardware device. It might need to print status updates to the screen many times a second. It is quite possible that the printing to the screen allowed by the standard C library might be too slow for such a program.

What are you to do in such a case? There are three ways you might try to increase the speed with which your program writes to the screen: by choosing print functions with a lower overhead, by using a package or library with faster print features, and by bypassing the operating system and writing directly to the screen. These methods will be examined from the least involved solution to the most complex.

Choosing Print Functions with a Lower Overhead

Some print functions have more overhead than others. "Overhead" refers to extra work that function must do compared to other functions. For example, `printf` has much more overhead than a function such as `puts`. Why is that?

The `puts` function is simple. It accepts a string of characters and writes them to the display. The `printf` function, of course, will do the same thing, but it does a lot more. It examines the string of characters you have sent to it, looking for special codes which indicate that internal data is to be printed.

Perhaps your code doesn't have internal characters, and you aren't passing anything to it. Unfortunately, the function has no way of knowing that, and it must scan the string for special characters every time.

There is a smaller difference between the functions `putch` and `puts`, with `putch` being better (having less overhead) if you plan to write only a single character.

Unfortunately, the overhead incurred by these C functions is minuscule compared to the overhead of actually drawing the characters onto your display. Thus, this method will probably not gain the programmer very much, except in peculiar circumstances.

Using a Package or Library with Faster Print Features

This is probably the easiest option that will result in real speed gains. You can get a package that will either replace the built-in printing functions in your compiler with faster versions or provide you with faster alternatives.

This option makes life pretty easy on the programmer because he will have to change his code very little, and he can use code that someone else has already spent a great deal of time optimizing. The downside is that the code might be owned by another programmer, and including it in your code might be expensive; or, if you decide to move your code to another platform, it might not exist for that machine.

Nonetheless, this can be a very practical and workable decision for the programmer to make.

Bypassing the Operating System and Writing Directly to the Screen

This action is somewhat frowned on, for many reasons. In fact, it is impossible to perform on some machines and under some operating systems. Furthermore, it is likely to be different for different machines, or even between different compilers on the same computer!

Nonetheless, for speed of video output, you simply cannot beat writing bytes directly to the screen. With full-screen text, you might be able to write hundreds of screens per second. If you need this kind of performance (perhaps for a video game), this method of output is probably worth the effort.

Because each computer and operating system handles this concept differently, it is impractical to give code for every operating system here. Instead, you shall see exactly how this concept is carried out under MS-DOS with Borland C. Even if you are not using these systems, you should be able to learn the correct methods from the following text, enabling you to write similar routines for your computer and operating system.

First of all, you need some method to write data to the screen. You can do this by creating a pointer that "points" to the screen memory. Under Borland C for MS-DOS, this task can be accomplished with this line of code:

```
char far *Screen = MK_FP( 0xb8000 , 0x0000 );
```

A "far" pointer is one that is not limited to the small data segment that has been reserved for your program; it can point anywhere in memory. MK_FP generates a far pointer to a specific location. Some other compilers and computers will not require the pointer type differentiation, or they might not have a similar function. You should look in your compiler's manual for the appropriate information.

Now, you have a pointer that points to the upper-left corner of the screen. You can write bytes to the location of this pointer, and you will see the characters you are writing appear there, as in the following program:

```
#include <dos.h>

main()
{
   int a;
   char far *Screen = MK_FP( 0xb800 , 0x0000 );
   for( a = 0 ; a < 26 ; ++ a )
        Screen[ a * 2 ] = 'a' + a;
   return( 0 );
}
```

After running this program, you should see the alphabet printed across the top of your monitor, in lowercase letters.

You will notice that instead of being written to consecutive locations in video memory, the characters were written only to every other byte of screen memory. Why is that? It is because even though a character occupies only a single byte, a byte is stored immediately after it to hold its color value. Therefore, each character as displayed on-screen is represented by two bytes in the computer's memory: one byte for the character itself and another byte for its color value.

This means two things: First, you must write characters only into every other byte of memory, or else you will see only every other character, as well as having bizarrely colored text. Second, you need to write the color bytes yourself if you plan to have colored text or overwrite the color that already exists at a location. Unless you do this, your text will be written with the color of what was already there. Every color byte must describe not only the color of the character, but also the color of the background it is written over. There are 16 foreground colors and 16 background colors. The lower four bits of the byte are reserved for the foreground color, and the high four bits are reserved for the background color.

This topic might seem a little complex for the inexperienced programmer, but it is actually pretty easy to understand. Just remember that there are 16 colors, ranging from 0 to 16, and that to get the screen byte value,

you add the foreground color to the value of the background color times 16. This is shown by the following program:

```
#include <stdio.h>

main()
{
   int fc , bc , c;
   scanf( " %d %d" , &fc , &bc );
   printf( " Foreground = %d , Background = %d , Color = %d\n" ,
           fc , bc , fc + bc * 16 );
   return( 0 );
}
```

I think the reader will agree that it is impractical in most cases for the programmer to have to explicitly write bytes to the screen throughout his program. Instead, it is better to write a routine for writing text to the display quickly, and reuse it frequently. Let's examine the construction of such a routine.

First, you need to ask yourself, "What information will I need to pass to my general-purpose printing function?" For starters, you want to be able to specify

The text to be written to the screen

The location of the text (two numbers)

The color of the text, as well as the background (also two numbers)

Now that you know what data you need to pass to your function, you can declare it in the following fashion:

```
void PrintAt( char *Text , int x , int y , int bc , int fc )
{
```

Now you want to calculate the byte value for the color of the text you will print:

```
int Color = fc + bc * 16;
```

You also need to calculate the starting position for the text pointer:

```
char far *Addr = &Screen[ ( x + y * 80 ) * 2 ];
```

Pay special attention to the fact that you must multiply the offset by two to write into the correct place. Also, note that this line assumes that somewhere in your code you have already defined the `Screen` variable. If you haven't, just insert the line

```
char far *Screen = MK_FP( 0xb800 , 0x0000 );
```

somewhere in your code.

Now that the preliminaries are out of the way, it's time to actually copy the bytes onto the screen. Look at the code that will carry out this task:

```
while( *Text )
{
   *( Addr++ ) = *( Text++ );
   *( Addr++ ) = Color;
}
```

This code loops while there are still characters left to copy, copying each character to the screen along with its corresponding color.

Take a look at this code in its entirety along with a corresponding test program:

```
#include <dos.h>
/* This is needed for the MK_FP function */
char far *Screen = MK_FP( 0xb800 , 0x0000 );

void PrintAt( char *Text , int x , int y , int bc , int fc )
{
        int Color = fc + bc * 16;
        char far *Addr = &Screen[ ( x + y * 80 ) * 2 ];
        while( *Text )
        {
                *( Addr++ ) = *( Text++ );
                *( Addr++ ) = Color;
        }
}

main()
{
        int a;
        for( a = 1 ; a < 16 ; ++ a )
           PrintAt( "This is a test" , a , a , a + 1  , a );
        return( 0 );
}
```

If you time this function as compared to the built-in printing functions, you should find it to be much faster. If you are using some other hardware platform, you might be able to use the concepts presented here to write a similarly quick printing function for your computer and operating system.

Cross Reference:

None.

XVII.5: How can I prevent the user from breaking my program with Ctrl-Break?

Answer:

MS-DOS, by default, enables the user of a program to stop its execution by pressing Ctrl-Break. This is, in most cases, a useful feature. It enables the user to exit in places from which a program might not allow exit, or from a program that has ceased to execute properly.

In some cases, however, this action might prove to be very dangerous. Some programs might carry out "secure" actions that, if broken, would give the user access to a private area. Furthermore, if the program is halted while updating a data file on disk, it might destroy the data file, perhaps destroying valuable data.

For these reasons, it might be useful to disable the Break key in some programs. A word of warning: delay placing this code in your program until you are 100 percent sure it will work! Otherwise, if your code

malfunctions and your program gets stuck, you might be forced to reboot the computer, perhaps destroying updates to the program.

Now I'll show you how to disable the Break key. This is something of a special operation. It can't be done on some machines, some do not have a Break key, and so on. There is therefore no special command in the C language for turning off the Break key. Furthermore, there is not even a standard way to do this on MS-DOS machines. On most machines, you must issue a special machine-language command. Here is a subroutine for turning off the Break key under MS-DOS:

```
#include <dos.h>

void StopBreak()
{
    union REGS in, out;
    in.x.ax = 0x3301;
    in.x.dx = 0;
    int86( 0x21, &in, &out );
}
```

This subroutine creates a set of registers, setting the `ax` register to hexadecimal 3301, and the `dx` register to 0. It then calls interrupt hexadecimal 21 with these registers. This calls MS-DOS and informs it that it no longer wants programs to be stopped by the Break key.

Here's a program to test this function:

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

void StopBreak()
{
    union REGS in, out;
    in.x.ax = 0x3301;
    in.x.dx = 0;
    int86( 0x21, &in, &out );
}

int main()
{
    int a;
    long b;
    StopBreak();
    for( a = 0; a < 100; ++a )
    {
        StopBreak();
        printf( "Line %d.\n", a );
        for( b = 0; b < 500000L; ++b );
    }
    return 0;
}
```

Cross Reference:

None.

XVII.6: How can you get data of only a certain type, for example, only characters?

Answer:

As with almost all computer science questions, the answer is, it depends on exactly what you're doing. If, for example, you are trying to read characters from the keyboard, you can use `scanf`:

```
scanf( " %c" , &c );
```

Alternatively, you can do this with some of the built-in C library functions:

```
c = getchar();
```

These options will produce basically the same results, with the use of `scanf` providing more safety checking for the programmer.

If you want to receive data of other types, you can use two methods. You can get the data character by character, always making sure that the correct thing is being entered. The other method is to use `scanf`, checking its return value to make sure that all fields were entered correctly.

You can use the second method to simply and efficiently extract a stream of records, verifying all of them to be correct. Here is an example program that carries out this maneuver:

```
#include <stdio.h>

main()
{
   int i,a,b;
   char c;
   void ProcessRecord( int, int, char );

   for( i = 0 ; i < 100 ; ++ a ) /* Read 100 records */
   {
        if ( scanf( " %d %d %c" , &a , &b , &c ) != 3 )
             printf( "data line %d is in error.\n" );
        else
             ProcessRecord( a , b , c );
   }
   return( 0 );
}
```

Cross Reference:

None.

XVII.7: Why shouldn't *scanf* be used to accept data?

Answer:

Although `scanf` is generally the most-used function for keyboard input, there are times when it is best not to use `scanf`. These situations can be broken down into various cases:

- Cases in which the user's keystrokes must be processed immediately when entered.

  If you are writing a program in which keystrokes must be acted on immediately after the key is pressed, `scanf` is useless. `scanf` waits at least until Enter is pressed. You don't know whether the user will press Enter one second, one minute, or one century after the key is pressed.

  Although this use is obviously bad in a real-time program, such as a computer game, it can also be bad in common utility programs. If you have a lettered menu, the user will probably prefer to press the letter *a* by itself, rather than pressing *a* followed by the Enter key.

  Unfortunately, the standard C library has no functions designed to carry out this action. You must rely on supplementary libraries or special functions included with your compiler.

- Cases in which you need things that `scanf` might parse away.

  `scanf` is a very smart function—in some cases, too smart. It will cross lines, throw away bad data, and ignore white space to attempt to satisfy the programmer's request for input data.

  Sometimes, however, you do not need this degree of cleverness! Sometimes you want to see the input exactly as the user typed it, even if there is not enough of it, too much of it, or such. A case of a program that is not suitable for `scanf` is one that must accept textual commands from the user. You don't know ahead of time how many words will be in the sentence that the user will type, nor do you have any way of knowing when the user will press Enter if you are using `scanf`!

- Cases in which you do not know ahead of time what data type the user will be entering.

  Sometimes, you are prepared to accept input from the user, but you do not know whether he will be entering a number, a word, or some special character. In these cases, you must get the data from the user in some neutral format, such as a character string, and decide what exactly the input is before continuing.

Additionally, `scanf` has the problem of preserving bad input in the input buffer. For example, if you are attempting to read in a number, and the user enters a character string, the code might loop endlessly trying to parse the character string as a number. This point can be demonstrated by the following program:

```
#include <stdio.h>

main()
{
    int i;
    while( scanf( "%d", &i ) ==0 )
    {
        printf( "Still looping.\n" );
    }
     return( 0 );
}
```

The program works fine if you enter a number as it expects, but if you enter a character string, it loops endlessly.

Cross Reference:

None.

XVII.8: How do I use function keys and arrow keys in my programs?

Answer:

The use of function keys and arrow keys in a program can make the program much easier to use. The arrow can be allowed to move the cursor, and the function keys can enable users to do special things, or they can replace commonly typed sequences of characters.

However, as is often the case with "special" features, there is no standard way to access them from within the C language. Using `scanf` to try to access these special characters will do you no good, and `getchar` cannot be depended on for this sort of operation. You need to write a small routing to query DOS for the value of the key being pressed. This method is shown in the following code:

```
#include <dos.h>

int GetKey()
{
    union REGS in, out;
    in.h.ah = 0x8;
    int86( 0x21, &in, &out );
    return out.h.al;
}
```

This method bypasses the C input/output library and reads the next key from the key buffer. It has the advantage that special codes are not lost, and that keys can be acted on as soon as they are pressed, instead of being stored in a buffer until Enter is pressed.

Using this function, you can get the integer function of keys when they are pressed. If you write a test program like

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

int GetKey()
{
    union REGS in, out;
    in.h.ah = 0x8;
    int86( 0x21, &in, &out );
    return out.h.al;
}

int main()
{
    int c;
    while( ( c = GetKey() ) != 27 )
            /* Loop until escape is pressed */
    {
        printf( "Key = %d.\n", c );
    }
    return 0;
}
```

you might get output like this for a typed string:

```
Key = 66.
Key = 111.
Key = 98.
Key = 32.
Key = 68.
Key = 111.
Key = 98.
Key = 98.
Key = 115.
```

When you press function keys or arrows, something different will happen; you will see a zero followed by a character value. This is the way special keys are represented: as a zero value followed by another, special value.

You therefore can take two actions. First, you can watch for zeros and, whenever one is pressed, treat the next character in a special fashion. Second, in the key press function, you can check for zeros and, when one is pressed, get the next value, modify it in some way, and return it. This second option is probably the better of the two options. Here's an efficient way of getting this task done:

```
/*
 New improved key-getting function.
*/

int GetKey()
{
    union REGS in, out;
    in.h.ah = 0x8;
    int86( 0x21, &in, &out );
    if ( out.h.al == 0 )
        return GetKey()+128;
    else
        return out.h.al;

}
```

This is the most efficient and clean of the two solutions. It will save a lot of work on the programmer's part by saving him from having to check for special cases. Special keys have values over 128.

Cross Reference:

None.

XVII.9: How do I prevent the user from typing too many characters in a field?

Answer:

There are two reasons to prevent a user from typing too many characters in a field. The first reason is that you might want to deal with only a fixed number of characters in your code. The second, and perhaps more important, reason is that if the user types more characters than your buffer can handle, it will overflow your

buffer and corrupt memory. This potential danger is often overlooked in C tutoring books. For example, the following code is very dangerous if the user types more than 50 characters:

```
char buf[ 50 ];
scanf( "%s", buf );
```

The way to fix the problem is by specifying in your scanning statement the maximum size of the string you want to scan. This task is accomplished by inserting a number between the % and the s, like so:

```
"%50s"
```

This specification will accept, at most, 50 characters from the user. Any extra characters the user types will remain in the input buffer and can be retrieved by another scanning command.

It is important to note that a string also needs a null terminator. Therefore, if you want to accept 50 characters from the user, your string must be of length 51. This is 50 characters for the real string data, plus a byte for the null terminator.

The following example program tests this technique:

```
#include <stdio.h>

/*
   Program to show how to stop the
   user from typing too many characters in
   a field.
*/

int main()
{
   char str[ 50 ]; /* This is larger than you really need */

   /*
     Now, accept only TEN characters from the user.  You can test
     this by typing more than ten characters here and seeing
     what is printed.
   */

   scanf( "%10s", str );

   /*
     Print the string, verifying that it is, at most, ten characters.
   */

   printf( "The output is: %s.\n", str );
   return( 0 );
}
```

And here's a sample run of the program. With the input

```
supercalifragilisticexpialidocious
```

the output is

```
supercalif.
```

Cross Reference:

None.

XVII.10: How do you zero-pad a number?

Answer:

To zero-pad a number, insert a number, preceded by a zero, after the %in the format specifier. This matter is best explained by direct example:

```
/* Print a five-character integer, padded with zeros. */

printf( "%05d", i );

/* Print a floating point, padded left of the zero out to
   seven characters. */

printf( "%07f", f );
```

If you fail to include the zero prefix on the number, it will be padded with spaces and not zeros.

Here is a sample program demonstrating this technique:

```
#include <stdio.h>

int main()
{
   int i = 123;
   printf( "%d\n", i );
   printf( "%05d\n", i );
   printf( "%07d\n", i );
   return( 0 );
}
```

And here is its output:

```
123
00123
0000123
```

Cross Reference:

None.

XVII.11: How do you print a dollars-and-cents value?

Answer:

The C language does not have any built-in facility for printing dollars-and-cents values. However, this omission does not present the programmer who is trying to print monetary values with an insurmountable

problem. It is quite easy to create a function that prints monetary values for you. After you create such a function, you can use it in any program you want.

Such a function is short and easy to write, and it will be presented here with a short explanation of how it works. The routine is broken into small, easy-to-write segments, making it easier to understand. The reason for breaking a program into smaller segments is discussed in Chapter XI, "Debugging."

These routines need to use some of the standard C routines. Therefore, you need to include some header files. Make sure that any program that uses this routine includes these header files at the start:

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

With the proper header files included, you can create a function that will accept a dollar value and print it with commas:

```
void PrintDollars( double Dollars )
{
   char buf[ 20 ];
   int l , a;
   sprintf( buf , "%01f" , Dollars );

   l = strchr( buf , '.' ) - buf;
   for( a = ( Dollars < 0.0 ) ; a < l ; ++ a )
   {
       printf( "%c" , buf[ a ] );
       if ( ( ( ( l - a ) % 3 ) == 1 ) && ( a != l - 1 ) )
           printf( "," );
   }
}
```

Perhaps you're used to seeing real numbers represented as floats. This is commonplace. Floats, however, are generally not suited for monetary work, because they suffer from a large degree of inaccuracy, such as rounding errors. Doubles are far more accurate than floats and therefore are much better suited for true numerical work.

You can easily test this routine yourself by writing a program that passes it integer numbers. This routine will not, however, print decimals or "change." To perform this task, you need to write another function specifically dedicated to this purpose:

```
void PrintCents( double Cents )
{
   char buf[ 10 ];
   sprintf( buf , "%.02f" , Cents );
   printf( "%s\n" , buf + 1 + ( Cents <= 0 ) );
}
```

This routine takes a decimal value and prints it correctly. Again, you can test this routine by writing a small program that passes it values.

Now you have two routines: one that prints the dollars part of a monetary value, and one that prints the cents. You certainly don't want to have to separate each number into two parts and call each function separately!

Instead, you can make one function that accepts a monetary value, divides it into its dollars and cents components, and calls the two routines you already have. Here is that function:

```
void DollarsAndCents( double Amount )
{
   double Dollars = Amount >= 0.0 ? floor( Amount ) : ceil( Amount
);
   double Cents = Amount - (double) Dollars;

   if ( Dollars < 0.0 ) printf( "-" );
   printf( "$" );
   PrintDollars( Dollars );
   PrintCents( Cents );
}
```

There you have it! The `DollarsAndCents` routine accepts a real number (a double) and prints it to the screen in dollars-and-cents format. You probably want to test the routine. To do this, you can make a `main` function that attempts to print many dollars-and-cents values. Here is such a routine:

```
int main()
{
  double num = .0123456789;
  int a;
  for( a = 0 ; a < 12 ; ++ a )
  {
       DollarsAndCents( num);
       num *= 10.0;
  }
  return( 0 );
}
```

The output of the preceding program should look like this:

```
$0.01
$0.12
$1.23
$12.35
$123.46
$1,234.57
$12,345.68
$123,456.79
$1,234,567.89
$12,345,678.90
$123,456,789.00
$1,234,567,890.00
```

If you want to print monetary values differently, it is quite easy to modify this program to print numbers in a different format.

Cross Reference:

None.

XVII.12: How do I print a number in scientific notation?

Answer:

To print a number in scientific notation, you must use the %e format specifier with the printf function, like so:

```
float f = 123456.78;
printf( " %e is in scientific\n" , f );
```

Of course, if you are to do this with integers, you must convert them to floating point first:

```
int i = 10000;
printf( " %e a scientific integer.\n" , (float) i );
```

Here is an example program demonstrating the %e format specifier:

```
#include <stdio.h>

main()
{
    double f = 1.0 / 1000000.0;
    int i;

    for( i = 0 ; i < 14 ; ++i )
    {
        printf( "%f = %e\n" , f , f );
        f *= 10.0;
    }
    return( 0 );
}
```

Cross Reference:

None.

XVII.13: What is the ANSI driver?

Answer:

Each computer has its own way of handling the screen. This is a necessary evil; if we became locked into a certain standard, the industry would stagnate. However, this difference causes great problems when you are attempting to write programs for different computers, as well as programs that must communicate over the phone line. To help alleviate this problem, the ANSI standard was introduced.

The ANSI standard attempts to lay a basic outline of how programs can cause the video terminal to perform certain standard tasks, such as printing text in different colors, moving the cursor, and clearing the screen. It does this by defining special character sequences that, when sent to the screen, affect it in specified ways.

Now, when you print these character sequences to the screen normally on some computers, you see the characters themselves, not the effect they were intended to produce. To fix this problem, you need to load

FREE EBOOKS, NOTES , VIDEOS & PLACEMENT MATERIAL

For All Companies placement Material

@placementclasses

For CAT Exam Preparation Material

@cat_classes

For GATE Exam Preparation Material

@gate_classes

For Engineering Books & Material

@cs_ebooks

Codes of Following Programming Languages

C

@c_examples

C++

@cpp_examples

Java

@java_examples0

Python

@python_examples

a program that will observe every character being printed to the screen, remove any special characters from the screen (so that they do not get printed), and carry out the desired action.

On MS-DOS machines, this program is called ANSI.SYS, and it must be loaded when the machine is booted up. This can be done by adding the line

```
DRIVER=ANSI.SYS
```

to your CONFIG.SYS file. The actual ANSI.SYS driver might be somewhere else in your directory tree; if so, it must be specified explicitly (with the full path). Here's an example:

```
driver=c:\sys\dos\ansi.sys
```

Cross Reference:

None.

XVII.14: How do you clear the screen with the ANSI driver?

Answer:

This action can be accomplished with <esc>[2J. Here is a program that demonstrates this point:

```
#include <stdio.h>

main()
{
   printf( "%c[2JNice to have an empty screen.\n", 27 );
    return( 0 );
}
```

Cross Reference:

None.

XVII.15: How do you save the cursor's position with the ANSI driver?

Answer:

This maneuver can be accomplished with <esc>[s. Here is a program that demonstrates this action:

```
#include <stdio.h>

main()
{
   printf( "Cursor position is %c[s \n", 27 );
   printf( "Interrupted!\n" );
```

```
    printf( "%c[uSAVED!!\n", 27 );
    return( 0 );
}
```

Cross Reference:

None.

XVII.16: How do you restore the cursor's position with the ANSI driver?

Answer:

This action can be accomplished with <esc>[u. Refer to the preceding FAQ for an example.

Cross Reference:

None.

XVII.17: How do you change the screen color with the ANSI driver?

Answer:

The way to carry out this task is to change the current text background color, then clear the screen. The following program serves as an example:

```
#include <stdio.h>

int main()
{
    printf( "%c[43;32m%c[2JOhh, pretty colors!\n", 27, 27 );
    return( 0 );
}
```

Cross Reference:

None.

XVII.18: How do you write text in color with the ANSI driver?

Answer:

The color of text is one of the text's attributes you can change. You can change the attributes of text with <esc>[<attr>m In the case of ANSI sequences, these attributes are represented by numerical values. You can

set multiple attributes with one command by separating them with semicolons, like this: <esc>[<attr>; <attr>m
The following program demonstrates this action:

```
#include <stdio.h>

main()
{
    printf( "%c[32;44mPsychedelic, man.\n", 27 );
    return( 0 );
}
```

Here is a list of attributes supported by the ANSI driver. Your particular monitor might not support some of the options.

1. High Intensity.
2. Low Intensity.
3. Italic.
4. Underline.
5. Blinking.
6. Fast Blinking.
7. Reverse.
8. Invisible.

Foreground colors:

30. Black.
31. Red.
32. Green.
33. Yellow.
34. Blue.
35. Magenta.
36. Cyan.
37. White.

Background colors:

40. Black.
41. Red.
42. Green.
43. Yellow.
44. Blue.
45. Magenta.
46. Cyan.
47. White.

Cross Reference:

None.

XVII.19: How do I move the cursor with the ANSI driver?

Answer:

There are two ways to move the cursor, relative motion and absolute motion. Relative motion is measured from the place where the cursor currently is; for example, "Move the cursor up two spaces." Absolute

placement is measured from the upper-left corner of the screen; for example, "Move the cursor to the 10th row, column 5."

Relative motion is carried out in the following fashion:

<esc>[#a in which # is the number of spaces to move up.
<esc>[#b in which # is the number of spaces to move down.
<esc>[#c in which # is the number of spaces to move right.
<esc>[#d in which # is the number of spaces to move left.

To move the cursor to an absolute location, you do this:

<esc>[<row>;<col>H in which row and col are the row and column at which you want the cursor to be positioned.

Cross Reference:

None.