Chapter 9: Standard Input / Output

THE stdio.h HEADER FILE

Examine the file simpleio.c for our first look at a file with standard I/O. Standard I/O refers to the most usual places where data is either read from, the keyboard, or written to, the video monitor. Since they are used so much, they are used as the default I/O devices and do not need to be named in the Input/Output instructions. This will make more sense when we actually start to use them so let's look at the file in front of you.

The first thing you should take notice of is the second line of the example file, the line with `#include "stdio.h"`. This is very much like the **#define** we have already studied, except that instead of a simple substitution, an entire file is read in at this point. The system will find the file named **stdio.h** and read its entire contents in, replacing this statement. Obviously then, the file named **stdio.h** must contain valid C source statements that can be compiled as part of a program. You will recall that we stated earlier that the preprocessor does textual substitution. This particular file is composed of several standard #defines to define some of the standard I/O operations. The file is called a header file and you will find several different header files on the source disks that came with your C compiler. Each of the header files has a specific purpose and any or all of them can be included in any program. Most header files contain definitions of a few types, function prototypes for the functions in its group, and some macros.

Your C compiler uses the double quote marks to indicate that the search for the include file will begin in the current directory, and if it not found there, the search will continue in the include directory as set up in the environment for your compiler. It also uses the "less than" and "greater than" signs to indicate that the file search should begin in the directory specified in the environment. Most of the programs in this tutorial have the double quotes in the include statements. The next program uses the "<" and ">" to illustrate the usage. Note that this will result is a slightly faster (but probably unnoticeable) compilation because the system will not bother to search the current directory first. If you know the include file is not in the current directory, it is best to use the "<" and ">" with the filename.

As many includes can be used as necessary, and it is perfectly all right for one header file to include one or more additional header files. It is very common to include four or five header files in a program.

INPUT/OUTPUT OPERATIONS IN C

Actually the C programming language has no input or output operations defined as part of the language, they must be user defined. Since everybody does not want to reinvent his own input and output operations, the compiler writers have done a lot of this for us and supplied us with several input functions and several output functions to aid in our program development. The functions have become a standard, and you will find the same functions available in every compiler. In fact, the industry standard of the C language definition has become the book written by Kernigan and Ritchie, and they have included these functions in their definition. Occasionally, when reading literature about C, you will find an author refer to K & R. This refers to the book, "The C Programming Language", written by Kernigan and Ritchie. You would be advised to purchase a copy for reference. As of this writing, a second edition of this book is available and is definitely the preferred edition.

You should print out the file named **stdio.h** and spend some time studying it. There will be a lot that you will not understand about it, but parts of it will look familiar. The name **stdio.h** is sort of cryptic for "standard input/output header", because that is exactly what it is. It defines the standard input and output functions in the form of #defines, macros, and prototypes for the functions. Don't worry too much about the details of this now. You can always return to this topic later for more study if it interests you, but you will really have no need to completely understand the **stdio.h** file. You will have a tremendous need to use it however, so these comments on its use and purpose are necessary.

OTHER INCLUDE FILES

When you begin writing larger programs and splitting them up into separately compiled portions, you will have occasion to use some definitions common to each of the portions. It would be to your advantage to make a separate file containing the definitions and use the **#include** to insert it into each of the files. If you want to change any of the common statements, you will only need to change one file and you will be assured of having all of the common statements agree. This is getting a little ahead of ourselves but you now have an idea how the **#include** directive can be used.

BACK TO THE FILE NAMED simpleio.c

Let's continue our tour of the file in question. The one variable named **c** is defined and a message is printed out with the familiar **printf()** function. We then find ourselves in a continuous loop as long as the value of **c** is not equal to capital X. If there is any question in your mind about the loop control, you should review chapter 3 before continuing. The two new functions within the loop are of paramount interest in this program since they are the new functions. These are functions to read a character from the keyboard and display it on the monitor one character at a time.

The function **getchar()** reads a single character from the standard input device, the keyboard being assumed because that is the standard input device, and assigns it to the variable named **c**. The next function **putchar()**, uses the standard output device, the video monitor, and outputs the character contained in the variable named **c**. The character is output at the current cursor location and the cursor is advanced one space for the next character. The system is therefore taking care of a lot of the overhead for us. The loop continues reading and displaying characters until we type a capital X which terminates the loop.

Compile and run this program for a few surprises. When you type on the keyboard, you will notice that what you type is displayed faithfully on the screen, and when you hit the return key, the entire line is repeated. We only told it to output each character once but it seems to be saving the characters up and redisplaying them. A short explanation is in order.

DOS IS HELPING US OUT

We need to understand a little bit about how DOS works to understand what is happening here. When data is read from the keyboard, under DOS control, the characters are stored in a buffer until a carriage return is entered at which time the entire string of characters is given to the program. When the characters are being typed, however, the characters are displayed one at a time on the monitor. This is called echo, and happens in many of the applications you run.

With the above paragraph in mind, it should be clear that when you are typing a line of data into SIMPLEIO, the characters are being echoed by DOS, and when you return the carriage by hitting return or enter, the characters are given to the program. As each character is given to the program, it displays it on the screen resulting in a repeat of the line typed in. To better illustrate this, type a line with a capital X somewhere in the middle of the line. You can type as many characters as you like following the X and they will all display because the characters are being read in under DOS, echoed to the monitor, and placed in the DOS input buffer. DOS doesn't think there is anything special about a capital X. When the string is given to the program, however, the characters are accepted by the program one at a time and sent to the monitor one at a time, until a capital X is encountered. After the capital X is displayed, the loop is terminated, and the program is terminated. The characters on the input line following the capital X are not displayed because the capital X signalled program termination.

Compile and run simpleio.c. After running the program several times and feeling confident that you understand the above explanation, we will go on to another program.

Don't get discouraged by the above seemingly weird behavior of the I/O system. It is strange, but there are other ways to get data into the computer. You will actually find the above method useful for many applications, and you will probably find some of the following useful also.

ANOTHER STRANGE I/O METHOD

Load the file named singleio.c and display it on your monitor for another method of character I/O. Once again, we start with the standard I/O header file using the "<" and ">" method of defining it. Then we define a variable named **c**, and we print a welcoming message. Like the last program, we are in a loop that will continue to execute until we type a capital X, but the action is a little different here.

The function named **_getch()** is a get character function. It differs from the function named **getchar()** in that it does not get tied up in DOS. It reads the character in without echo, and puts it directly into the program where it is operated on immediately. This function therefore reads a character, immediately displays it on the screen, and continues the operation until a capital X is typed. Note that although **_getch()** is available with most popular microcomputer C compilers, it is not included in the ANSI standard and may not be available with all C compilers. It's use may therefore make a program nonportable. If your compiler does not support the **_getch()** function, use the **getchar()** function instead.

When you compile and run this program, you will find that there is no repeat of the lines when you hit a carriage return, and when you hit the capital X, the program terminates immediately. No carriage return is needed to get it to accept the line with the X in it, so this program operates a little differently from the last one. However, we do have another problem here, since there is no linefeed with the carriage return.

NOW WE NEED A LINE FEED

It is not apparent to you in most application programs but when you hit the enter key, the program supplies a linefeed to go with the carriage return. You need to return to the left side of the monitor and you also need to drop down a line. The linefeed is not automatic. We need to improve our program to do this also. If you will load and display the program named betterin.c, you will find a change to incorporate this feature.

In betterin.c, we have two additional statements at the beginning that will define the character codes for the linefeed (LF), and the carriage return (CR). If you look at any ASCII table you will find that the codes 10 and 13 are exactly as defined here. In the main program, after outputting the character in line 15, we compare it to CR, and if it is equal to CR, we also output a linefeed which is the LF. We could have completely omitted the two **#define** statements and used the statement `if (c == 13) putchar(10);` but it would not be very descriptive of what we are doing here. The method used in this program represents better programming practice.

You will notice that line 16 deviates from the usual style for an if statement, but we have a choice.

We can format the code anyway we desire to improve the readability. It is strictly a programmer's choice.

Compile and run betterin.c to see if it does what we have said it should do. It should display exactly what you type in, including a linefeed with each carriage return, and should stop immediately when you type a capital X. Once again, if your compiler does not support **_getch()**, use the **getchar()** function.

WHICH METHOD IS BEST ?

We have examined two methods of reading characters into a C program, and are faced with a choice of which one we should use. It really depends on the application because each method has advantages and disadvantages.

When using the first method, DOS is actually doing all of the work for us by storing the characters in an input buffer and signaling us when a full line has been entered. We could write a program that, for example, did a lot of calculations, then went to get some input. While we were doing the calculations, DOS would be accumulating a line of characters for us, and they would be there when we were ready for them. However, we could not read in single keystrokes because DOS would not report a buffer of characters to us until it recognized a carriage return.

The second method, used in betterin.c, allows us to get a single character, and act on it immediately. We do not have to wait until DOS decides we can have a line of characters. We cannot do anything else while we are waiting for a character because we are waiting for the input keystroke and tying up the entire machine. This method is useful for highly interactive types of program interfaces. It is up to you as the programmer to decide which is best for your needs.

I should mention at this point that there is also an **_ungetch()** function that works with the **_getch()** function. If you **_getch()** a character and find that you have gone one too far, you can **_ungetch()** it back to the input device. This simplifies some programs because you don't know that you don't want the character until you get it. You can only **_ungetch()** one character back to the input device, but that is sufficient to accomplish the task this function was designed for. It is difficult to demonstrate this function in a simple program so its use will be up to you to study when you need it. Another function that may be available with your compiler, but is not part of the ANSI standard, is the **_getche()** function which is identical to the **_getch()** function except that it echoes the character to the monitor for you.

The discussion so far in this chapter should be a good indication that, while the C programming language is very flexible, it does put a lot of responsibility on you as the programmer to keep many details in mind.

NOW TO READ IN SOME INTEGERS

Load and display the file named intin.c for an example of reading some formatted data from the keyboard. The structure of this program is very similar to the last three except that we define an int type variable and loop until the variable somehow acquires the value of 100.

Instead of reading in a character at a time, as we have in the last three example programs, we read in an entire integer value with one call using the function named **scanf()**. This function is very similar to the **printf()** that you have been using for quite some time by now except that it is used for input instead of output. Examine the line with the **scanf()** and you will notice that it does not ask for the variable **valin** directly, but gives the address of the variable since it expects to have a value returned from the function. Recall that a function must have the address of a variable in order to return a value to that variable in the calling program. Failing to supply a pointer to the parameter in the **scanf**

() function is the most common problem encountered in using this function.

The function **scanf()** scans the input line until it finds the first data field. It ignores leading blanks and in this case, it reads integer characters until it finds a blank or an invalid decimal character, at which time it stops reading and returns the value.

Remembering our discussion above about the way the DOS input buffer works, it should be clear that nothing is actually acted on until a complete line is entered and it is terminated by a carriage return. At this time, the buffer is input, and our program will search across the line reading all integer values it can find until the line is completely scanned. This is because we are in a loop and we tell it to find a value, print it, find another, print it, etc. If you enter several values on one line, it will read each one in succession and display the values. Entering the value of 100 will cause the program to terminate, and entering the value 100 with other values following, will cause termination before the following values are considered.

IT MAKES WRONG ANSWERS SOMETIMES

If you enter a number up to and including 32767, it will display correctly, but if you enter a larger number, it will appear to make an error unless your system uses a much larger range for an **int** type variable. For example, if you enter the value 32768, it will display the value of -32768, entering the value 65536 will display as a zero. These are not errors but are caused by the way an **int** variable is defined. The most significant bit of the 16 bit pattern available for the integer variable is the sign bit, so there are only 15 bits left for the value. The variable can therefore only have the values from -32768 to 32767, any other values are outside the range of integer variables. This is up to you to take care of in your programs. It is another example of the increased responsibility you must assume using C rather than another high level language such as Pascal, Modula-2, etc.

The above paragraph is true for most MS-DOS C compilers. There is a very small possibility that your compiler uses an integer value stored in a field size other than 16 bits. If that is the case, the same principles will be true but with different limits than those given above.

Compile and run this program, entering several numbers on a line to see the results, and with varying numbers of blanks between the numbers. Try entering numbers that are too big to see what happens, and finally enter some invalid characters to see what the system does with nondecimal characters.

CHARACTER STRING INPUT

Load and display the file named stringin.c for an example of reading a string variable from the keyboard. This program is identical to the last one except that instead of an integer variable, we have defined a string variable with an upper limit of 24 characters (remember that a string variable must have a null character at the end). The variable in the **scanf()** does not need an & because **big** is an array variable and by definition it is already a pointer. This program should require no additional explanation. Compile and run it to see if it works the way you expect.

You probably got a surprise when you ran it because it separated your sentence into separate words. When used in the string mode of input, **scanf()** reads characters into the string until it comes to either the end of a line or a blank character. Therefore, it reads a word, finds the blank following it, and displays the result. Since we are in a loop, this program continues to read words until it exhausts the DOS input buffer. We have written this program to stop whenever it finds a capital X in column 1, but since the sentence is split up into individual words, it will stop anytime a word begins with capital X. Try entering a 5 word sentence with a capital X as the first character in the third word. You should get the first three words displayed, and the last two simply ignored when the program stops.

Try entering more than 24 characters to see what the program does. In an actual program, it is your responsibility to count characters and stop when the input buffer is full. You may be getting the feeling that a lot of responsibility is placed on you when writing in C. Along with this responsibility you get a lot of flexibility in the bargain also.

INPUT/OUTPUT PROGRAMMING IN C

C was not designed to be used as a language for lots of input and output, but as a systems language where a lot of internal operations are required. You would do well to use another language for I/O intensive programming, but C could be used if you desire. The keyboard input is very flexible, allowing you to get at the data in a very low level way, but very little help is given you. It is therefore up to you to take care of all of the bookkeeping chores associated with your required I/O operations. This may seem like a real pain in the neck, but in any given program, you only need to define your input routines once and then use them as needed. Don't let this worry you. As you gain experience with C, you will easily handle your I/O requirements.

One final point must be made about these I/O functions. It is perfectly permissible to intermix **scanf ()** and **getchar()** functions during read operations. In the same manner, it is also fine to intermix the output functions, **printf()** and **putchar()** in any way you desire.

IN MEMORY I/O

The next operation may seem a little strange at first, but you will probably see lots of uses for it as you gain experience. Load the file named inmem.c and display it for another type of I/O, one that never accesses the outside world, but stays in the computer.

In inmem.c, we define a few variables, then assign some values to the ones named **numbers** for illustrative purposes and then use an **sprintf()** function. The function acts just like a normal **printf()** function except that instead of printing the line of output to a device, it prints the line of formatted output to a character string in memory. In this case the string goes to the string variable named **line**, because that is the string name we inserted as the first argument in the **sprintf()** function. The spaces after the 2nd **%d** were put there to illustrate that the next function will search properly across the line. We print the resulting string and find that the output is identical to what it would have been by using a **printf()** instead of the **sprintf()** in the first place. You will see that when you compile and run the program shortly.

Since the generated string is still in memory, we can now read it with the function **sscanf()**. We tell the function in its first argument that **line** is the string to use for its input, and the remaining parts of the line are exactly what we would use if we were going to use the **scanf()** function and read data from outside the computer. Note that it is essential that we use pointers to the data because we want to return data from a function. Just to illustrate that there are many ways to declare a pointer several methods are used, but all are ultimately pointers. The first two simply declare the address of the elements of the array, while the last three use the fact that **result**, without the accompanying subscript, is a pointer. Just to keep it interesting, the values are read back in reverse order. Finally the values are displayed on the monitor.

IS THAT REALLY USEFUL ?

It seems sort of silly to read input data from within the computer but it does have a real purpose. It is possible to read data from an input device using any of the standard functions and then do a format conversion in memory. You could read in a line of data, look at a few significant characters, then use these formatted input routines to reduce the line of data to internal representation. That would sure beat writing your own data formatting routines.

STANDARD ERROR OUTPUT

Sometimes it is desirable to redirect the output from the standard output device to a file. However, you may still want the error messages to go to the standard output device, in our case the monitor. This next function allows you to do that. Load and display special.c for an example of this new function.

The program consists of a loop with two messages output, one to the standard output device and the other to the standard error device. The message to the standard error device is output with the function **fprintf()** and includes the device name **stderr** as the first argument. Other than those two small changes, it is the same as our standard **printf()** function. (You will see more of the **fprintf()** function in the next chapter, but its operation fit in better as a part of this chapter.) Ignore the line with the exit for the moment, we will return to it.

Compile and run this program, and you will find 12 lines of output on the monitor. To see the difference, run the program again with redirected output to a file named STUFF by entering the following line at the DOS prompt;

```
C:> special >stuff
```

This time you will only get the 6 lines output to the standard error device, and if you look in your directory, you will find that the file named STUFF contains the other 6 lines, those to the standard output device. You can use I/O redirection with any of the programs we have run so far, and as you may guess, you can also read from a file using I/O redirection but we will study a better way to read from a file in the next chapter. More information about I/O redirection can be found in your DOS manual.

WHAT ABOUT THE exit(4) STATEMENT ?

Now to keep our promise about the **exit(4)** statement. Redisplay the file named special.c on your monitor. The last statement exits the program and returns the value of 4 to DOS. Any number from 0 to 19 can be used in the parentheses for DOS communication. If you are operating in a BATCH file, this number can be tested with the ERRORLEVEL command.

Most compilers that operate in several passes return a 1 with this mechanism to indicate that a fatal error has been detected and it would be a waste of time to go on to another pass resulting in even more errors.

It is therefore wise to use a batch file for compiling programs and testing the returned value for errors. A check of the documentation for my computer, resulted in a minimal and confusing documentation of the ERRORLEVEL command, so a brief description of it is given in this file in case your documentation does not include enough information to allow you to use it.

One additional feature must be mentioned here. Since we wish to return an **int** value to the operating system, we must define the main program entry point as returning an **int** rather than a **void** as we have used in most of the example programs to this point. Refer to line 5 for an example of this extension.

Programming Exercise:

---

1. Write a program to read in a character using a loop, and display the character in

its normal **char** form. Also display it as a decimal number. Check for a dollar sign to use as the stop character. Use the **_getch()** form of input so it will print immediately. Hit some of the special keys, such as function keys, when you run the program for some surprises. You will get two inputs from the special keys, the first being a zero which is the indication to the system that a special key was hit.

2. Add a character string to singleio.c and store the input characters in the string. When the X is detected, add a terminating null to the string and print out the string with a **printf()** function call.

Index Previous Next . . . . *C Tutorial*

The Webwizard