8

Character Input/Output and Input Validation

You will learn about the following in this chapter:

- More about input, output, and the differences between buffered and unbuffered input
- How to simulate the end-of-file condition from the keyboard
- How to use redirection to connect your programs to files
- Making the user interface friendlier

In the computing world, we use the words *input* and *output* in several ways. We speak of input and output devices, such as keyboards, USB drives, scanners, and laser printers. We talk about the data used for input and output. We discuss the functions that perform input and output. This chapter concentrates on the functions used for input and output (or *I/O*, for short).

I/O functions transport information to and from your program; `printf()`, `scanf()`, `getchar()`, and `putchar()` are examples. You've seen these functions in previous chapters, and now you'll be able to look at their conceptual basis. Along the way, you'll see how to improve the program-user interface.

Originally, input/output functions were not part of the definition of C. Their development was left to C implementations. In practice, the Unix implementation of C has served as a model for these functions. The ANSI C library, recognizing past practice, contains a large number of these Unix I/O functions, including the ones we've used. Because such standard functions must work in a wide variety of computer environments, they seldom take advantage of features peculiar to a particular system. Therefore, many C vendors supply additional I/O functions that do make use of special features of the hardware. Other functions or families of functions tap into particular operating systems that support, for example, specific graphical interfaces, such as those provided by Windows or Macintosh OS. These specialized, nonstandard functions enable you to write programs that use a particular computer more effectively. Unfortunately, they often can't be used on other computer systems. Consequently, we'll concentrate on the standard I/O functions available on all systems, because they enable you to write portable programs that can

be moved easily from one system to another. They also generalize to programs using files for input and output.

One important task many programs face is that of validating input; that is, determining whether the user has entered input that matches the expectations of a program. This chapter illustrates some of the problems and solutions associated with input validation.

Single-Character I/O: `getchar()` and `putchar()`

As you saw in Chapter 7, "C Control Statements: Branching and Jumps," `getchar()` and `putchar()` perform input and output one character at a time. That method might strike you as a rather silly way of doing things. After all, you can easily read groupings larger than a single character, but this method does suit the capability of a computer. Furthermore, this approach is the heart of most programs that deal with text—that is, with ordinary words. To remind yourself of how these functions work, examine Listing 8.1, a very simple example. All it does is fetch characters from keyboard input and send them to the screen. This process is called *echoing the input*. It uses a `while` loop that terminates when the # character is encountered.

Listing 8.1 **The `echo.c` Program**

```
/* echo.c -- repeats input */
#include <stdio.h>
int main(void)
{
    char ch;

    while ((ch = getchar()) != '#')
        putchar(ch);

    return 0;
}
```

Since the ANSI standard, C associates the `stdio.h` header file with using `getchar()` and `putchar()`, which is why we have included that file in the program. (Typically, `getchar()` and `putchar()` are not true functions, but are defined using preprocessor macros, a topic we'll cover in Chapter 16, "The C Preprocessor and the C Library.") Using this program produces exchanges like this:

```
Hello, there. I would[enter]
Hello, there. I would
like a #3 bag of potatoes.[enter]
like a
```

After watching this program run, you might wonder why you must type a whole line before the input is echoed. You might also wonder if there is a better way to terminate input. Using a particular character, such as #, to terminate input prevents you from using that character in the text. To answer these questions, let's look at how C programs handle keyboard input. In particular, let's examine buffering and the concept of a standard input file.

Buffers

If you ran the previous program on some older systems, the text you input would be echoed immediately. That is, a sample run would look like this:

```
HHeellllloo,,  tthheerree..  II  wwoouulldd[enter]

lliikkee  aa  #
```

The preceding behavior is the exception. On most systems, nothing happens until you press Enter, as in the first example. The immediate echoing of input characters is an instance of *unbuffered* (or *direct*) input, meaning that the characters you type are immediately made available to the waiting program. The delayed echoing, on the other hand, illustrates *buffered* input, in which the characters you type are collected and stored in an area of temporary storage called a *buffer*. Pressing Enter causes the block of characters you typed to be made available to your program. Figure 8.1 compares these two kinds of input.

Why have buffers? First, it is less time-consuming to transmit several characters as a block than to send them one by one. Second, if you mistype, you can use your keyboard correction features to fix your mistake. When you finally press Enter, you can transmit the corrected version.

Unbuffered input, on the other hand, is desirable for some interactive programs. In a game, for instance, you would like each command to take place as soon as you press a key. Therefore, both buffered and unbuffered input have their uses.

Buffering comes in two varieties: *fully buffered* I/O and *line-buffered* I/O. For fully buffered input, the buffer is flushed (the contents are sent to their destination) when it is full. This kind of buffering usually occurs with file input. The buffer size depends on the system, but 512 bytes and 4096 bytes are common values. With line-buffered I/O, the buffer is flushed whenever a newline character shows up. Keyboard input is normally line buffered, so that pressing Enter flushes the buffer.

Which kind of input do you have: buffered or unbuffered? ANSI C and subsequent C standards specify that input should be buffered, but K&R originally left the choice open to the compiler writer. You can find out by running the `echo.c` program and seeing which behavior results.

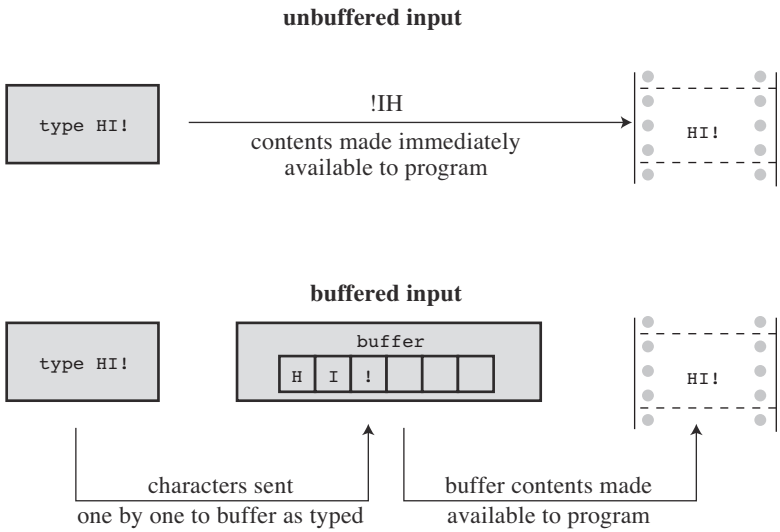

Figure 8.1 Buffered versus unbuffered input.

The reason ANSI C settled on buffered input as the standard is that some computer designs don't permit unbuffered input. If your particular computer does allow unbuffered input, most likely your C compiler offers unbuffered input as an option. Many compilers for IBM PC compatibles, for example, supply a special family of functions, supported by the `conio.h` header file, for unbuffered input. These functions include `getche()` for echoed unbuffered input and `getch()` for unechoed unbuffered input. (*Echoed input* means the character you type shows onscreen, and *unechoed input* means the keystrokes don't show.) Unix systems use a different approach, for Unix itself controls buffering. With Unix, you use the `ioctl()` function (part of the Unix library but not part of standard C) to specify the type of input you want, and `getchar()` behaves accordingly. In ANSI C, the `setbuf()` and `setvbuf()` functions (see Chapter 13, "File Input/Output") supply some control over buffering, but the inherent limitations of some systems can restrict the effectiveness of these functions. In short, there is no standard ANSI way of invoking unbuffered input; the means depend on the computer system. In this book, with apologies to our unbuffered friends, we assume you are using buffered input.

Terminating Keyboard Input

The `echo.c` program halts when # is entered, which is convenient as long as you exclude that character from normal input. As you've seen, however, # can show up in normal input. Ideally, you'd like a terminating character that normally does not show up in text. Such a character won't pop up accidentally in the middle of some input, stopping the program before you want it to stop. C has an answer to this need, but, to understand it, you need to know how C handles files.

Files, Streams, and Keyboard Input

A *file* is an area of memory in which information is stored. Normally, a file is kept in some sort of permanent memory, such as a hard disk, USB flash drive, or optical disc, such as a DVD. You are doubtless aware of the importance of files to computer systems. For example, your C programs are kept in files, and the programs used to compile your programs are kept in files. This last example points out that some programs need to be able to access particular files. When you compile a program stored in a file called `echo.c`, the compiler opens the `echo.c` file and reads its contents. When the compiler finishes, it closes the file. Other programs, such as word processors, not only open, read, and close files, they also write to them.

C, being powerful, flexible, and so on, has many library functions for opening, reading, writing, and closing files. On one level, it can deal with files by using the basic file tools of the host operating system. This is called *low-level I/O*. Because of the many differences among computer systems, it is impossible to create a standard library of universal low-level I/O functions, and ANSI C does not attempt to do so; however, C also deals with files on a second level called the *standard I/O package*. This involves creating a standard model and a standard set of I/O functions for dealing with files. At this higher level, differences between systems are handled by specific C implementations so that you deal with a uniform interface.

What sort of differences are we talking about? Different systems, for example, store files differently. Some store the file contents in one place and information about the file elsewhere. Some build a description of the file into the file itself. In dealing with text, some systems use a single newline character to mark the end of a line. Others might use the combination of the carriage return and linefeed characters to represent the end of a line. Some systems measure file sizes to the nearest byte; some measure in blocks of bytes.

When you use the standard I/O package, you are shielded from these differences. Therefore, to check for a newline, you can use `if (ch == '\n')`. If the system actually uses the carriage-return/linefeed combination, the I/O functions automatically translate back and forth between the two representations.

Conceptually, the C program deals with a stream instead of directly with a file. A *stream* is an idealized flow of data to which the actual input or output is mapped. That means various kinds of input with differing properties are represented by streams with more uniform properties. The process of opening a file then becomes one of associating a stream with the file, and reading and writing take place via the stream.

Chapter 13 discusses files in greater detail. For this chapter, simply note that C treats input and output devices the same as it treats regular files on storage devices. In particular, the keyboard and the display device are treated as files opened automatically by every C program. Keyboard input is represented by a stream called `stdin`, and output to the screen (or teletype or other output device) is represented by a stream called `stdout`. The `getchar()`, `putchar()`, `printf()`, and `scanf()` functions are all members of the standard I/O package, and they deal with these two streams.

One implication of all this is that you can use the same techniques with keyboard input as you do with files. For example, a program reading a file needs a way to detect the end of the file so

that it knows where to stop reading. Therefore, C input functions come equipped with a built-in, end-of-file detector. Because keyboard input is treated like a file, you should be able to use that end-of-file detector to terminate keyboard input, too. Let's see how this is done, beginning with files.

The End of File

A computer operating system needs some way to tell where each file begins and ends. One method to detect the end of a file is to place a special character in the file to mark the end. This is the method once used, for example, in CP/M, IBM-DOS, and MS-DOS text files. Today, these operating systems may use an embedded Ctrl+Z character to mark the ends of files. At one time, this was the sole means these operating systems used, but there are other options now, such as keeping track of the file size. So a modern text file may or may not have an embedded Ctrl+Z, but if it does, the operating system will treat it as an end-of-file marker. Figure 8.2 illustrates this approach.

prose:

Ishphat the robot
slid open the hatch
and shouted his challenge.

prose in a file:

```
Ishphat the robot\n slid open the hatch\n and shouted his challenge.\n^Z
```

Figure 8.2 A file with an end-of-file marker.

A second approach is for the operating system to store information on the size of the file. If a file has 3000 bytes and a program has read 3000 bytes, the program has reached the end. MS-DOS and its relatives use this approach for binary files because this method allows the files to hold all characters, including Ctrl+Z. Newer versions of DOS also use this approach for text files. Unix uses this approach for all files.

C handles this variety of methods by having the `getchar()` function return a special value when the end of a file is reached, regardless of how the operating system actually detects the end of file. The name given to this value is `EOF` (end of file). Therefore, the return value for `getchar()` when it detects an end of file is `EOF`. The `scanf()` function also returns `EOF` on detecting the end of a file. Typically, `EOF` is defined in the `stdio.h` file as follows:

```
#define EOF (-1)
```

Why `-1`? Normally, `getchar()` returns a value in the range `0` through `127`, because those are values corresponding to the standard character set, but it might return values from `0` through `255` if the system recognizes an extended character set. In either case, the value `-1` does not correspond to any character, so it can be used to signal the end of a file.

Some systems may define EOF to be a value other than –1, but the definition is always different from a return value produced by a legitimate input character. If you include the stdio.h file and use the EOF symbol, you don't have to worry about the numeric definition. The important point is that EOF represents a value that signals the end of a file was detected; it is not a symbol actually found in the file.

Okay, how can you use EOF in a program? Compare the return value of getchar() with EOF. If they are different, you have not yet reached the end of a file. In other words, you can use an expression like this:

```
while ((ch = getchar()) != EOF)
```

What if you are reading keyboard input and not a file? Most systems (but not all) have a way to simulate an end-of-file condition from the keyboard. Knowing that, you can rewrite the basic read and echo program, as shown in Listing 8.2.

Listing 8.2 **The** `echo_eof.c` **Program**

```
/* echo_eof.c -- repeats input to end of file */
#include <stdio.h>
int main(void)
{
    int ch;

    while ((ch = getchar()) != EOF)
        putchar(ch);

    return 0;
}
```

Note these points:

- You don't have to define EOF because stdio.h takes care of that.
- You don't have to worry about the actual value of EOF, because the #define statement in stdio.h enables you to use the symbolic representation EOF. You shouldn't write code that assumes EOF has a particular value.
- The variable ch is changed from type char to type int because char variables may be represented by unsigned integers in the range 0 to 255, but EOF may have the numeric value –1. That is an impossible value for an unsigned char variable, but not for an int. Fortunately, getchar() is actually type int itself, so it can read the EOF character. Implementations that use a signed char type may get by with declaring ch as type char, but it is better to use the more general form.
- The fact that getchar() is type int is why some compilers warn of possible data loss if you assign the getchar() return value to a type char variable.

- The fact that `ch` is an integer doesn't faze `putchar()`. It still prints the character equivalent.
- To use this program on keyboard input, you need a way to type the `EOF` character. No, you can't just type the letters *E O F*, and you can't just type `–1`. (Typing `-1` would transmit two characters: a hyphen and the digit 1.) Instead, you have to find out what your system requires. On most Unix and Linux systems, for example, pressing Ctrl+D at the *beginning* of a line causes the end-of-file signal to be transmitted. Many microcomputing systems recognize Ctrl+Z at the beginning of a line as an end-of-file signal; some interpret a Ctrl+Z anywhere as an end-of-file signal.

Here is a buffered example of running `echo_eof.c` on a Unix system:

```
She walks in beauty, like the night
She walks in beauty, like the night
  Of cloudless climes and starry skies...
  Of cloudless climes and starry skies...
                      Lord Byron
                      Lord Byron
[Ctrl+D]
```

Each time you press Enter, the characters stored in the buffer are processed, and a copy of the line is printed. This continues until you simulate the end of file, Unix-style. On a PC, you would press Ctrl+Z instead.

Let's stop for a moment and think about the possibilities for `echo_eof.c`. It copies onto the screen whatever input you feed it. Suppose you could somehow feed a file to it. Then it would print the contents of the file onscreen, stopping when it reached the end of the file, on finding an `EOF` signal. Suppose, instead, that you could find a way to direct the program's output to a file. Then you could enter data from the keyboard and use `echo_eof.c` to store what you type in a file. Suppose you could do both simultaneously: Direct input from one file into `echo_eof.c` and send the output to another file. Then you could use `echo_eof.c` to copy files. This little program has the potential to look at the contents of files, to create new files, and to make copies of files—pretty good for such a short program! The key is to control the flow of input and output, and that is the next topic.

Note Simulated `EOF` and Graphical Interfaces

The concept of simulated `EOF` arose in a command-line environment using a text interface. In such an environment, the user interacts with a program through keystrokes, and the operating system generates the `EOF` signal. Some practices don't translate particularly well to graphical interfaces, such as Windows and the Macintosh, with more complex user interfaces that incorporate mouse movement and button clicks. The program behavior on encountering a simulated EOF depends on the compiler and project type. For example, a Ctrl+Z may terminate input or it may terminate the entire program, depending on the particular settings.

Redirection and Files

Input and output involve functions, data, and devices. Consider, for instance, the `echo_eof.c` program. It uses the input function `getchar()`. The input device (we have assumed) is a keyboard, and the input data stream consists of individual characters. Suppose you want to keep the same input function and the same kind of data, but want to change where the program looks for data. A good question to ask is, "How does a program know where to look for its input?"

By default, a C program using the standard I/O package looks to the standard input as its source for input. This is the stream identified earlier as `stdin`. It is whatever has been set up as the usual way for reading data into the computer. It could be an old-fashioned device, such as magnetic tape, punched cards, or a teletype, or (as we will continue to assume) your keyboard, or some upcoming technology, such as voice input. A modern computer is a suggestible tool, however, and you can influence it to look elsewhere for input. In particular, you can tell a program to seek its input from a file instead of from a keyboard.

There are two ways to get a program to work with files. One way is to explicitly use special functions that open files, close files, read files, write in files, and so forth. That method we'll save for Chapter 13. The second way is to use a program designed to work with a keyboard and screen, but to *redirect* input and output along different channels—to and from files, for example. In other words, you reassign the `stdin` stream to a file. The `getchar()` program continues to get its data from the stream, not really caring from where the stream gets its data. This approach (redirection) is more limited in some respects than the first, but it is much simpler to use, and it allows you to gain familiarity with common file-processing techniques.

One major problem with redirection is that it is associated with the operating system, not C. However, the many C environments, including Unix, Linux, and the Windows Command-Prompt mode, feature redirection, and some C implementations simulate it on systems lacking the feature. Apple OS X runs on top of Unix, and you can use the Unix command-line mode by starting the Terminal application. We'll look at the Unix, Linux, and Windows versions or redirection.

Unix, Linux, and Windows Command Prompt Redirection

Unix (when run in command-line mode), Linux (ditto), and the Windows Command Prompt (which mimics the old DOS command-line environment) enable you to redirect both input and output. Redirecting input enables your program to use a file instead of the keyboard for input, and redirecting output enables it to use a file instead of the screen for output.

Redirecting Input

Suppose you have compiled the `echo_eof.c` program and placed the executable version in a file called `echo_eof` (or `echo_eof.exe` on a Windows system). To run the program, type the executable file's name:

```
echo_eof
```

The program runs as described earlier, taking its input from the keyboard. Now suppose you want to use the program on a text file called `words`. A *text file* is one containing text—that is, data stored as human-readable characters. It could be an essay or a program in C, for example. A file containing machine language instructions, such as the file holding the executable version of a program, is not a text file. Because the program works with characters, it should be used with text files. All you need to do is enter this command instead of the previous one:

```
echo_eof < words
```

The < symbol is a Unix and Linux and DOS/Windows redirection operator. It causes the `words` file to be associated with the `stdin` stream, channeling the file contents into the `echo_eof` program. The `echo_eof` program itself doesn't know (or care) that the input is coming from a file instead of the keyboard. All it knows is that a stream of characters is being fed to it, so it reads them and prints them one character at a time until the end of file shows up. Because C puts files and I/O devices on the same footing, the file is now the I/O *device*. Try it!

> **Note Redirection Sidelights**
>
> With Unix, Linux, and Windows Command Prompt, the spaces on either side of the < are optional. Some systems, such as AmigaDOS (for those who still play in the good old days), support redirection but don't allow a space between the redirection symbol and the filename.

Here is a sample run for one particular `words` file; the `$` is one of the standard Unix and Linux prompts. On a Windows/DOS system, you would see the DOS prompt, perhaps an `A>` or `C>`.

```
$ echo_eof < words
The world is too much with us: late and soon,
Getting and spending, we lay waste our powers:
Little we see in Nature that is ours;
We have given our hearts away, a sordid boon!
$
```

Well, that time we got our words' worth.

Redirecting Output

Now suppose you want to have `echo_eof` send your keyboard input to a file called `mywords`. Then you can enter the following and begin typing:

```
echo_eof > mywords
```

The > is a second redirection operator. It causes a new file called `mywords` to be created for your use, and then it redirects the output of `echo_eof` (that is, a copy of the characters you type) to that file. The redirection reassigns `stdout` from the display device (your screen) to the `mywords` file instead. If you already have a file with the name `mywords`, normally it would be erased and then replaced by the new one. (Many operating systems, however, give you the option of protecting existing files by making them read-only.) All that appears on your screen are the letters as you type them, and the copies go to the file instead. To end the program, press Ctrl+D

(Unix) or Ctrl+Z (DOS) at the beginning of a line. Try it. If you can't think of anything to type, just imitate the next example. In it, we use the `$` Unix prompt. Remember to end each line by pressing Enter to send the buffer contents to the program.

```
$ echo_eof > mywords
You should have no problem recalling which redirection
operator does what. Just remember that each operator points
in the direction the information flows. Think of it as
a funnel.
[Ctrl+D]
$
```

After the Ctrl+D or Ctrl+Z is processed, the program terminates and your system prompt returns. Did the program work? The Unix `ls` command or Windows Command Prompt `dir` command, which lists filenames, should show you that the file `mywords` now exists. You can use the Unix and Linux `cat` or DOS `type` command to check the contents, or you can use `echo_eof` again, this time redirecting the file to the program:

```
$ echo_eof < mywords
You should have no problem recalling which redirection
operator does what. Just remember that each operator points
in the direction the information flows. Think of it as a
funnel.
$
```

Combined Redirection

Now suppose you want to make a copy of the file `mywords` and call it `savewords`. Just issue this next command,

```
echo_eof < mywords > savewords
```

and the deed is done. The following command would have worked as well, because the order of redirection operations doesn't matter:

```
echo_eof > savewords < mywords
```

Beware: Don't use the same file for both input and output to the same command.

```
echo_eof < mywords > mywords....<--WRONG
```

The reason is that `> mywords` causes the original `mywords` to be truncated to zero length before it is ever used as input.

In brief, here are the rules governing the use of the two redirection operators (< and >) with Unix, Linux, or Windows/DOS:

- A redirection operator connects an *executable* program (including standard operating system commands) with a data file. It cannot be used to connect one data file to another, nor can it be used to connect one program to another program.

- Input cannot be taken from more than one file, nor can output be directed to more than one file by using these operators.
- Normally, spaces between the names and operators are optional, except occasionally when some characters with special meaning to the Unix shell or Linux shell or the Windows Command Prompt mode are used. We could, for example, have used `echo_eof<words`.

You have already seen several proper examples. Here are some wrong examples, with `addup` and `count` as executable programs and `fish` and `beets` as text files:

```
fish > beets              ← Violates the first rule
addup < count             ← Violates the first rule
addup < fish < beets      ← Violates the second rule
count > beets fish        ← Violates the second rule
```

Unix, Linux, and Windows/DOS also feature the >> operator, which enables you to add data to the end of an existing file, and the pipe operator (|), which enables you to connect the output of one program to the input of a second program. See a Unix book, such as *UNIX Primer Plus, Third Edition* (Wilson, Pierce, and Wessler; Sams Publishing), for more information on all these operators.

Comments

Redirection enables you to use keyboard-input programs with files. For this to work, the program has to test for the end of file. For example, Chapter 7 presents a word-counting program that counts words up to the first | character. Change `ch` from type `char` to type `int`, and replace `'|'` with `EOF` in the loop test, and you can use the program to count words in text files.

Redirection is a command-line concept, because you indicate it by typing special symbols on the command line. If you are not using a command-line environment, you might still be able to try the technique. First, some integrated environments have menu options that let you indicate redirection. Second, for Windows systems, you can open the Command Prompt window and run the executable file from the command line. Microsoft Visual Studio, by default, puts the executable file in a subfolder, called `Debug`, of the project folder. The filename will have the same base name as the project name and use the `.exe` extension. By default Xcode also names the executable file after the project name and places it in a Debug folder. You can run the executable from the Terminal utility, which runs a version of Unix. However, if you use Terminal, it's probably simpler to use one of the command-line compilers (GCC or Clang) that can be downloaded from Apple.

If redirection doesn't work for you, you can try having the program open a file directly. Listing 8.3 shows an example with minimal explanation. You'll have to wait until Chapter 13 for the details. The file to be read should be in the same directory as the executable file.

Listing 8.3 **The `file_eof.c` Program**

```
// file_eof.c --open a file and display it
#include <stdio.h>
#include <stdlib.h>  // for exit()
int main()
{
    int ch;
    FILE * fp;
    char fname[50];         // to hold the file name

    printf("Enter the name of the file: ");
    scanf("%s", fname);
    fp = fopen(fname, "r"); // open file for reading
    if (fp == NULL)         // attempt failed
    {
        printf("Failed to open file. Bye\n");
        exit(1);            // quit program
    }
// getc(fp) gets a character from the open file
    while ((ch = getc(fp)) != EOF)
        putchar(ch);
    fclose(fp);             // close the file

    return 0;
}
```

Summary: How to Redirect Input and Output

With most C systems, you can use redirection, either for all programs through the operating system or else just for C programs, courtesy of the C compiler. In the following, let `prog` be the name of the executable program and let `file1` and `file2` be names of files.

Redirecting Output to a File: >

```
prog >file1
```

Redirecting Input from a File: <

```
prog <file2
```

Combined Redirection:

```
prog <file2 >file1
prog >file1 <file2
```

Both forms use `file2` for input and `file1` for output.

Spacing:

Some systems require a space to the left of the redirection operator and no space to the right. Other systems (Unix, for example) accept either spaces or no spaces on either side.

Creating a Friendlier User Interface

Most of us have on occasion written programs that are awkward to use. Fortunately, C gives you the tools to make input a smoother, more pleasant process. Unfortunately, learning these tools could, at first, lead to new problems. The goal in this section is to guide you through some of these problems to a friendlier user interface, one that eases interactive data entry and smoothes over the effects of faulty input.

Working with Buffered Input

Buffered input is often a convenience to the user, providing an opportunity to edit input before sending it on to a program, but it can be bothersome to the programmer when character input is used. The problem, as you've seen in some earlier examples, is that buffered input requires you to press the Enter key to transmit your input. This act also transmits a newline character that the program must handle. Let's examine this and other problems with a guessing program. You pick a number, and the program tries to guess it. The program uses a plodding method, but we are concentrating on I/O, not algorithms. See Listing 8.4 for the starting version of the program, one that will need further work.

Listing 8.4 **The `guess.c` Program**

```
/* guess.c -- an inefficient and faulty number-guesser */
#include <stdio.h>
int main(void)
{
    int guess = 1;

    printf("Pick an integer from 1 to 100. I will try to guess ");
    printf("it.\nRespond with a y if my guess is right and with");
    printf("\nan n if it is wrong.\n");
    printf("Uh...is your number %d?\n", guess);
    while (getchar() != 'y')      /* get response, compare to y */
        printf("Well, then, is it %d?\n", ++guess);
    printf("I knew I could do it!\n");

    return 0;
}
```

Here's a sample run:

```
Pick an integer from 1 to 100. I will try to guess it.
Respond with a y if my guess is right and with
an n if it is wrong.
Uh...is your number 1?
n
Well, then, is it 2?
```

```
Well, then, is it 3?
n
Well, then, is it 4?
Well, then, is it 5?
y
I knew I could do it!
```

Out of consideration for the program's pathetic guessing algorithm, we chose a small number. Note that the program makes two guesses every time you enter `n`. What's happening is that the program reads the `n` response as a denial that the number is 1 and then reads the newline character as a denial that the number is 2.

One solution is to use a `while` loop to discard the rest of the input line, including the newline character. This has the additional merit of treating responses such as `no` and `no way` the same as a simple `n`. The version in Listing 8.4 treats `no` as two responses. Here is a revised loop that fixes the problem:

```
while (getchar() != 'y')   /* get response, compare to y */
{
    printf("Well, then, is it %d?\n", ++guess);
    while (getchar() != '\n')
        continue;          /* skip rest of input line    */
}
```

Using this loop produces responses such as the following:

```
Pick an integer from 1 to 100. I will try to guess it.
Respond with a y if my guess is right and with
an n if it is wrong.
Uh...is your number 1?
n
Well, then, is it 2?
no
Well, then, is it 3?
no sir
Well, then, is it 4?
forget it
Well, then, is it 5?
y
I knew I could do it!
```

That takes care of the problems with the newline character. However, as a purist, you might not like `f` being treated as meaning the same as `n`. To eliminate that defect, you can use an `if` statement to screen out other responses. First, add a `char` variable to store the response:

```
char response;
```

Then change the loop to this:

```
while ((response = getchar()) != 'y')     /* get response */
{
   if (response == 'n')
      printf("Well, then, is it %d?\n", ++guess);
   else
      printf("Sorry, I understand only y or n.\n");
   while (getchar() != '\n')
      continue;                /* skip rest of input line */
}
```

Now the program's response looks like this:

```
Pick an integer from 1 to 100. I will try to guess it.
Respond with a y if my guess is right and with
an n if it is wrong.
Uh...is your number 1?
n
Well, then, is it 2?
no
Well, then, is it 3?
no sir
Well, then, is it 4?
forget it
Sorry, I understand only y or n.
n
Well, then, is it 5?
y
I knew I could do it!
```

When you write interactive programs, you should try to anticipate ways in which users might fail to follow instructions. Then you should design your program to handle user failures gracefully. Tell them when they are wrong, and give them another chance.

You should, of course, provide clear instructions to the user, but no matter how clear you make them, someone will always misinterpret them and then blame you for poor instructions.

Mixing Numeric and Character Input

Suppose your program requires both character input using `getchar()` and numeric input using `scanf()`. Each of these functions does its job well, but the two don't mix together well. That's because `getchar()` reads every character, including spaces, tabs, and newlines, whereas `scanf()`, when reading numbers, skips over spaces, tabs, and newlines.

To illustrate the sort of problem this causes, Listing 8.5 presents a program that reads in a character and two numbers as input. It then prints the character using the number of rows and columns specified in the input.

Listing 8.5 **The `showchar1.c` Program**

```
/* showchar1.c -- program with a BIG I/O problem */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
    int ch;             /* character to be printed    */
    int rows, cols;     /* number of rows and columns */
    printf("Enter a character and two integers:\n");
    while ((ch = getchar()) != '\n')
    {
        scanf("%d %d", &rows, &cols);
        display(ch, rows, cols);
        printf("Enter another character and two integers;\n");
        printf("Enter a newline to quit.\n");
    }
    printf("Bye.\n");

    return 0;
}

void display(char cr, int lines, int width)
{
    int row, col;

    for (row = 1; row <= lines; row++)
    {
        for (col = 1; col <= width; col++)
            putchar(cr);
        putchar('\n');  /* end line and start a new one */
    }
}
```

Note that the program reads a character as type `int` to enable the `EOF` test. However, it passes the character as type `char` to the `display()` function. Because `char` is smaller than `int`, some compilers will warn about the conversion. In this case, you can ignore the warning. Or you can eliminate the warning by using a typecast:

```
display(char(ch), rows, cols);
```

The program is set up so that `main()` gets the data and the `display()` function does the printing. Let's look at a sample run to see what the problem is:

```
Enter a character and two integers:
c 2 3
ccc
ccc
```

```
Enter another character and two integers;
Enter a newline to quit.
Bye.
```

The program starts off fine. Enter `c 2 3`, and it prints two rows of three *c* characters, as expected. Then the program prompts you to enter a second set of data and quits before you have a chance to respond! What's wrong? It's that newline character again, this time the one immediately following the `3` on the first input line. The `scanf()` function leaves it in the input queue. Unlike `scanf()`, `getchar()` doesn't skip over newline characters, so this newline character is read by `getchar()` during the next cycle of the loop before you have a chance to enter anything else. Then it's assigned to `ch`, and `ch` being the newline character is the condition that terminates the loop.

To clear up this problem, the program has to skip over any newlines or spaces between the last number typed for one cycle of input and the character typed at the beginning of the next line. Also, it would be nice if the program could be terminated at the `scanf()` stage in addition to the `getchar()` test. The next version, shown in Listing 8.6, accomplishes this.

Listing 8.6 **The `showchar2.c` Program**

```
/* showchar2.c -- prints characters in rows and columns */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
    int ch;             /* character to be printed      */
    int rows, cols;     /* number of rows and columns   */

    printf("Enter a character and two integers:\n");
    while ((ch = getchar()) != '\n')
    {
        if (scanf("%d %d",&rows, &cols) != 2)
            break;
        display(ch, rows, cols);
        while (getchar() !=  '\n')
            continue;
        printf("Enter another character and two integers;\n");
        printf("Enter a newline to quit.\n");
    }
    printf("Bye.\n");

    return 0;
}

void display(char cr, int lines, int width)
{
```

```
    int row, col;

    for (row = 1; row <= lines; row++)
    {
        for (col = 1; col <= width; col++)
            putchar(cr);
        putchar('\n');  /* end line and start a new one */
    }
}
```

The `while` statement causes the program to dispose of all characters following the `scanf()` input, including the newline. This prepares the loop to read the first character at the beginning of the next line. This means you can enter data fairly freely:

```
Enter a character and two integers:
c 1 2
cc
Enter another character and two integers;
Enter a newline to quit.
! 3 6
!!!!!!
!!!!!!
!!!!!!
Enter another character and two integers;
Enter a newline to quit.

Bye.
```

By using an `if` statement with a `break`, we terminate the program if the return value of `scanf()` is not `2`. This occurs if one or both input values are not integers or if end-of-file is encountered.

Input Validation

In practice, program users don't always follow instructions, and you can get a mismatch between what a program expects as input and what it actually gets. Such conditions can cause a program to fail. However, often you can anticipate likely input errors, and, with some extra programming effort, have a program detect and work around them.

Suppose, for instance, that you had a loop that processes nonnegative numbers. One kind of error the user can make is to enter a negative number. You can use a relational expression to test for that:

```
long n;
scanf("%ld", &n);      // get first value
while (n >= 0)        // detect out-of-range value
```

```
{
    // process n
    scanf("%ld", &n);  // get next value
}
```

Another potential pitfall is that the user might enter the wrong type of value, such as the character `q`. One way to detect this kind of misuse is to check the return value of `scanf()`. This function, as you'll recall, returns the number of items it successfully reads; therefore, the expression

```
scanf("%ld", &n) == 1
```

is true only if the user inputs an integer. This suggests the following revision of the code:

```
long n;
while (scanf("%ld", &n) == 1 && n >= 0)
{
    // process n
}
```

In words, the `while` loop condition is "while input is an integer and the integer is positive."

The last example terminates input if the user enters the wrong type of value. You can, however, choose to make the program a little more user friendly and give the user the opportunity to try to enter the correct type of value. In that case, you need to dispose of the input that caused `scanf()` to fail in the first place, for `scanf()` leaves the bad input in the input queue. Here, the fact that input really is a stream of characters comes in handy, because you can use `getchar()` to read the input character-by-character. You could even incorporate all these ideas into a function such as the following:

```
long get_long(void)
{
    long input;
    char ch;

    while (scanf("%ld", &input) != 1)
    {
        while ((ch = getchar()) != '\n')
            putchar(ch);  // dispose of bad input
        printf(" is not an integer.\nPlease enter an ");
        printf("integer value, such as 25, -178, or 3: ");
    }

    return input;
}
```

This function attempts to read an `int` value into the variable `input`. If it fails to do so, the function enters the body of the outer `while` loop. The inner `while` loop then reads the

offending input character-by-character. Note that this function chooses to discard all the remaining input on the line. Other possible choices are to discard just the next character or word. Then the function prompts the user to try again. The outer loop keeps going until the user successfully enters an integer, causing `scanf()` to return the value `1`.

After the user clears the hurdle of entering integers, the program can check to see whether the values are valid. Consider an example that requires the user to enter a lower limit and an upper limit defining a range of values. In this case, you probably would want the program to check that the first value isn't greater than the second (usually ranges assume that the first value is the smaller one). It may also need to check that the values are within acceptable limits. For example, the archive search may not work with year values less than 1958 or greater than 2014. This checking, too, can be accomplished with a function.

Here's one possibility; the following function assumes that the `stdbool.h` header file has been included. If you don't have `_Bool` on your system, you can substitute `int` for `bool`, `1` for `true`, and `0` for `false`. Note that the function returns `true` if the input is invalid; hence the name `bad_limits()`:

```
bool bad_limits(long begin, long end,
                long low, long high)
{
    bool not_good = false;

    if (begin > end)
    {
        printf("%ld isn't smaller than %ld.\n", begin, end);
        not_good = true;
    }
    if (begin < low || end < low)
    {
        printf("Values must be %ld or greater.\n", low);
        not_good = true;
    }
    if (begin > high || end > high)
    {
        printf("Values must be %ld or less.\n", high);
        not_good = true;
    }

    return not_good;
}
```

Listing 8.7 uses these two functions to feed integers to an arithmetic function that calculates the sum of the squares of all the integers in a specified range. The program limits the upper and lower bounds of the range to 1000 and –1000, respectively.

Listing 8.7 **The `checking.c` Program**

```
// checking.c -- validating input
#include <stdio.h>
#include <stdbool.h>
// validate that input is an integer
long get_long(void);
// validate that range limits are valid
bool bad_limits(long begin, long end,
                long low, long high);
// calculate the sum of the squares of the integers
// a through b
double sum_squares(long a, long b);
int main(void)
{
    const long MIN = -10000000L;  // lower limit to range
    const long MAX = +10000000L;  // upper limit to range
    long start;                   // start of range
    long stop;                    // end of range
    double answer;

    printf("This program computes the sum of the squares of "
           "integers in a range.\nThe lower bound should not "
           "be less than -10000000 and\nthe upper bound "
           "should not be more than +10000000.\nEnter the "
           "limits (enter 0 for both limits to quit):\n"
           "lower limit: ");
    start = get_long();
    printf("upper limit: ");
    stop = get_long();
    while (start !=0 || stop != 0)
    {
        if (bad_limits(start, stop, MIN, MAX))
            printf("Please try again.\n");
        else
        {
            answer = sum_squares(start, stop);
            printf("The sum of the squares of the integers ");
            printf("from %ld to %ld is %g\n",
                    start, stop, answer);
        }
        printf("Enter the limits (enter 0 for both "
               "limits to quit):\n");
        printf("lower limit: ");
        start = get_long();
        printf("upper limit: ");
        stop = get_long();
```

```
    }
    printf("Done.\n");

    return 0;
}

long get_long(void)
{
    long input;
    char ch;

    while (scanf("%ld", &input) != 1)
    {
        while ((ch = getchar()) != '\n')
            putchar(ch);  // dispose of bad input
        printf(" is not an integer.\nPlease enter an ");
        printf("integer value, such as 25, -178, or 3: ");
    }

    return input;
}

double sum_squares(long a, long b)
{
    double total = 0;
    long i;

    for (i = a; i <= b; i++)
        total += (double)i * (double)i;

    return total;
}

bool bad_limits(long begin, long end,
                long low, long high)
{
    bool not_good = false;

    if (begin > end)
    {
        printf("%ld isn't smaller than %ld.\n", begin, end);
        not_good = true;
    }
    if (begin < low || end < low)
    {
        printf("Values must be %ld or greater.\n", low);
        not_good = true;
```

```
    }
    if (begin > high || end > high)
    {
        printf("Values must be %ld or less.\n", high);
        not_good = true;
    }

    return not_good;
}
```

Here's a sample run:

```
This program computes the sum of the squares of integers in a range.
The lower bound should not be less than -10000000 and
the upper bound should not be more than +10000000.
Enter the limits (enter 0 for both limits to quit):
lower limit: low
low is not an integer.
Please enter an integer value, such as 25, -178, or 3: 3
upper limit: a big number
a big number is not an integer.
Please enter an integer value, such as 25, -178, or 3: 12
The sum of the squares of the integers from 3 to 12 is 645
Enter the limits (enter 0 for both limits to quit):
lower limit: 80
upper limit: 10
80 isn't smaller than 10.
Please try again.
Enter the limits (enter 0 for both limits to quit):
lower limit: 0
upper limit: 0
Done.
```

Analyzing the Program

The computational core (the function `sum_squares()`) of the `checking.c` program is short, but the input validation support makes it more involved than the examples we have given before. Let's look at some of its elements, first focusing on overall program structure.

We've followed a modular approach, using separate functions (modules) to verify input and to manage the display. The larger a program is, the more vital it is to use modular programming.

The `main()` function manages the flow, delegating tasks to the other functions. It uses `get_long()` to obtain values, a `while` loop to process them, the `badlimits()` function to check for valid values, and the `sum_squares()` function to do the actual calculation:

```
start = get_long();
printf("upper limit: ");
```

```
stop = get_long();
while (start !=0 || stop != 0)
{
    if (bad_limits(start, stop, MIN, MAX))
        printf("Please try again.\n");
    else
    {
        answer = sum_squares(start, stop);
        printf("The sum of the squares of the integers ");
        printf("from %ld to %ld is %g\n", start, stop, answer);
    }
    printf("Enter the limits (enter 0 for both "
           "limits to quit):\n");
    printf("lower limit: ");
    start = get_long();
    printf("upper limit: ");
    stop = get_long();
}
```

The Input Stream and Numbers

When writing code to handle bad input, such as that used in Listing 8.7, you should have a clear picture of how C input works. Consider a line of input like the following:

```
is  28 12.4
```

To our eyes, it looks like a string of characters followed by an integer followed by a floating-point value. To a C program it looks like a stream of bytes. The first byte is the character code for the letter `i`, the second is the character code for the letter `s`, the third is the character code for the space character, the fourth is the character code for the digit `2`, and so on. So if `get_long()` encounters this line, which begins with a nondigit, the following code reads and discards the entire line, including the numbers, which just are other characters on the line:

```
while ((ch = getchar()) != '\n')
   putchar(ch);  // dispose of bad input
```

Although the input stream consists of characters, the `scanf()` function can convert them to a numeric value if you tell it to. For example, consider the following input:

```
42
```

If you use `scanf()` with a `%c` specifier, it will just read the `4` character and store it in a `char` variable. If you use the `%s` specifier, it will read two characters, the `4` character and the `2` character, and store them in a character string. If you use the `%d` specifier, `scanf()` reads the same two characters, but then proceeds to calculate that the integer value corresponding to them is 4 × 10 + 2, or 42. It then stores the integer binary representation of that value in an `int` variable. If you use an `%f` specifier, `scanf()` reads the two characters, calculates that they correspond to

the numeric value 42.0, expresses that value in the internal floating-point representation, and stores the result in a `float` variable.

In short, input consists of characters, but `scanf()` can convert that input to an integer or floating-point value. Using a specifier such as `%d` or `%f` restricts the types of characters that are acceptable input, but `getchar()` and `scanf()` using `%c` accept any character.

Menu Browsing

Many computer programs use menus as part of the user interface. Menus make programs easier for the user, but they do pose some problems for the programmer. Let's see what's involved.

A menu offers the user a choice of responses. Here's a hypothetical example:

```
Enter the letter of your choice:
a. advice           b. bell
c. count            q. quit
```

Ideally, the user then enters one of these choices, and the program acts on that choice. As a programmer, you want to make this process go smoothly. The first goal is for the program to work smoothly when the user follows instructions. The second goal is for the program to work smoothly when the user fails to follow instructions. As you might expect, the second goal is the more difficult because it's hard to anticipate all the possible mistreatment that might come your program's way.

Modern applications typically use graphical interfaces—buttons to click, boxes to check, icons to touch—instead of the command-line approach of our examples, but the general process remains much the same: Offer the user choices, detect and act upon the user's response, and protect against possible misuse. The underlying program structure would be much the same for these different interfaces. However, using a graphical interface can make it easier to control input by limiting choices.

Tasks

Let's get more specific and look at the tasks a menu program needs to perform. It needs to get the user's response, and it needs to select a course of action based on the response. Also, the program should provide a way to return to the menu for further choices. C's `switch` statement is a natural vehicle for choosing actions because each user choice can be made to correspond to a particular `case` label. You can use a `while` statement to provide repeated access to the menu. In pseudocode, you can describe the process this way:

```
get choice
while choice is not 'q'
    switch to desired choice and execute it
    get next choice
```

Toward a Smoother Execution

The goals of program smoothness (smoothness when processing correct input and smoothness when handling incorrect input) come into play when you decide how to implement this plan. One thing you can do, for example, is have the "get choice" part of the code screen out inappropriate responses so that only correct responses are passed on to the `switch`. That suggests representing the input process with a function that can return only correct responses. Combining that with a `while` loop and a `switch` leads to the following program structure:

```
#include <stdio.h>
char get_choice(void);
void count(void);
int main(void)
{
    int choice;

    while ( (choice = get_choice()) != 'q')
    {
        switch (choice)
        {
            case 'a' :  printf("Buy low, sell high.\n");
                        break;
            case 'b' :  putchar('\a');  /* ANSI */
                        break;
            case 'c' :  count();
                        break;
            default  :  printf("Program error!\n");
                        break;
        }
    }
    return 0;
}
```

The `get_choice()` function is defined so that it can return only the values `'a'`, `'b'`, `'c'`, and `'q'`. You use it much as you use `getchar()`—getting a value and comparing it to a termination value (`'q'`, in this case). We've kept the actual menu choices simple so that you can concentrate on the program structure; we'll get to the `count()` function soon. The `default` case is handy for debugging. If the `get_choice()` function fails to limit its return value to the intended values, the `default` case lets you know something fishy is going on.

The `get_choice()` Function

Here, in pseudocode, is one possible design for this function:

```
show choices
get response
while response is not acceptable
    prompt for more response
    get response
```

And here is a simple, but awkward, implementation:

```
char get_choice(void)
{
    int ch;

    printf("Enter the letter of your choice:\n");
    printf("a. advice           b. bell\n");
    printf("c. count            q. quit\n");
    ch = getchar();
    while (  (ch < 'a' || ch > 'c') && ch != 'q')
    {
        printf("Please respond with a, b, c, or q.\n");
        ch = getchar();
    }
    return ch;
}
```

The problem is that with buffered input, every newline generated by the Return key is treated as an erroneous response. To make the program interface smoother, the function should skip over newlines.

There are several ways to do that. One is to replace `getchar()` with a new function called `get_first()` that reads the first character on a line and discards the rest. This method also has the advantage of treating an input line consisting of, say, `act`, as being the same as a simple `a`, instead of treating it as one good response followed by `c` for `count`. With this goal in mind, we can rewrite the input function as follows:

```
char get_choice(void)
{
    int ch;

    printf("Enter the letter of your choice:\n");
    printf("a. advice           b. bell\n");
    printf("c. count            q. quit\n");
    ch = get_first();
    while (  (ch < 'a' || ch > 'c') && ch != 'q')
    {
        printf("Please respond with a, b, c, or q.\n");
        ch = getfirst();
    }
    return ch;
}

char get_first(void)
{
    int ch;
```

```
    ch = getchar();           /* read next character */
    while (getchar() != '\n')
        continue;             /* skip rest of line */
    return ch;
}
```

Mixing Character and Numeric Input

Creating menus provides another illustration of how mixing character input with numeric input can cause problems. Suppose, for example, the `count()` function (choice `c`) were to look like this:

```
void count(void)
{
    int n,i;

    printf("Count how far? Enter an integer:\n");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
}
```

If you then responded by entering `3`, `scanf()` would read the `3` and leave a newline character as the next character in the input queue. The next call to `get_choice()` would result in `get_first()` returning this newline character, leading to undesirable behavior.

One way to fix that problem is to rewrite `get_first()` so that it returns the next non-whitespace character rather than just the next character encountered. We leave that as an exercise for the reader. A second approach is having the `count()` function tidy up and clear the newline itself. This is the approach this example takes:

```
void count(void)
{
    int n,i;

    printf("Count how far? Enter an integer:\n");
    n = get_int();
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
    while ( getchar() != '\n')
        continue;
}
```

This function also uses the `get_long()` function from Listing 8.7, but changes it to `get_int()` to fetch type `int` instead of type `long`; recall that the original checks for valid input and gives the user a chance to try again. Listing 8.8 shows the final menu program.

Listing 8.8 **The `menuette.c` Program**

```
/* menuette.c -- menu techniques */
#include <stdio.h>
char get_choice(void);
char get_first(void);
int get_int(void);
void count(void);
int main(void)
{
    int choice;
    void count(void);

    while ( (choice = get_choice()) != 'q')
    {
        switch (choice)
        {
            case 'a' :  printf("Buy low, sell high.\n");
                        break;
            case 'b' :  putchar('\a');  /* ANSI */
                        break;
            case 'c' :  count();
                        break;
            default  :  printf("Program error!\n");
                        break;
        }
    }
    printf("Bye.\n");

    return 0;
}

void count(void)
{
    int n,i;

    printf("Count how far? Enter an integer:\n");
    n = get_int();
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
    while ( getchar() != '\n')
        continue;
}

char get_choice(void)
{
    int ch;
```

```
    printf("Enter the letter of your choice:\n");
    printf("a. advice           b. bell\n");
    printf("c. count            q. quit\n");
    ch = get_first();
    while (  (ch < 'a' || ch > 'c') && ch != 'q')
    {
        printf("Please respond with a, b, c, or q.\n");
        ch = get_first();
    }

    return ch;
}

char get_first(void)
{
    int ch;

    ch = getchar();
    while (getchar() != '\n')
        continue;

    return ch;
}

int get_int(void)
{
    int input;
    char ch;

    while (scanf("%d", &input) != 1)
    {
        while ((ch = getchar()) != '\n')
            putchar(ch);  // dispose of bad input
        printf(" is not an integer.\nPlease enter an ");
        printf("integer value, such as 25, -178, or 3: ");
    }

    return input;
}
```

Here is a sample run:

```
Enter the letter of your choice:
a. advice           b. bell
c. count            q. quit
a
```

```
Buy low, sell high.
Enter the letter of your choice:
a. advice           b. bell
c. count            q. quit
count
Count how far? Enter an integer:
two
two is not an integer.
Please enter an integer value, such as 25, -178, or 3: 5

1
2
3
4
5
Enter the letter of your choice:
a. advice           b. bell
c. count            q. quit
d
Please  respond with a, b, c, or q.
q
```

It can be hard work getting a menu interface to work as smoothly as you might want, but after you develop a viable approach, you can reuse it in a variety of situations.

Another point to notice is how each function, when faced with doing something a bit complicated, delegated the task to another function, thus making the program much more modular.

Key Concepts

C programs see input as a stream of incoming bytes. The `getchar()` function interprets each byte as being a character code. The `scanf()` function sees input the same way, but, guided by its conversion specifiers, it can convert character input to numeric values. Many operating systems provide redirection, which allows you to substitute a file for a keyboard for input and to substitute a file for a monitor for output.

Programs often expect a particular form of input. You can make a program much more robust and user friendly by anticipating entry errors a user might make and enabling the program to cope with them.

With a small program, input validation might be the most involved part of the code. It also opens up many choices. For example, if the user enters the wrong kind of information, you can terminate the program, you can give the user a fixed number of chances to get the input right, or you give the user an unlimited number of chances.

Summary

Many programs use `getchar()` to read input character-by-character. Typically, systems use *line-buffered input,* meaning that input is transmitted to the program when you press Enter. Pressing Enter also transmits a newline character that may require programming attention. ANSI C requires buffered input as the standard.

C features a family of functions, called the *standard I/O package*, that treats different file forms on different systems in a uniform manner. The `getchar()` and `scanf()` functions belong to this family. Both functions return the value `EOF` (defined in the `stdio.h` header) when they detect the end of a file. Unix systems enable you to simulate the end-of-file condition from the keyboard by pressing Ctrl+D at the beginning of a line; DOS systems use Ctrl+Z for the same purpose.

Many operating systems, including Unix and DOS, feature *redirection,* which enables you to use files instead of the keyboard and screen for input and output. Programs that read input up to `EOF` can then be used either with keyboard input and simulated end-of-file signals or with redirected files.

Interspersing calls to `getchar()` with calls to `scanf()` can cause problems when `scanf()` leaves a newline character in the input just before a call to `getchar()`. By being aware of this problem, however, you can program around it.

When you are writing a program, plan the user interface thoughtfully. Try to anticipate the sort of errors users are likely to make and then design your program to handle them.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. `putchar(getchar())` is a valid expression; what does it do? Is `getchar(putchar())` also valid?

2. What would each of the following statements accomplish?

   a. `putchar('H');`

   b. `putchar('\007');`

   c. `putchar('\n');`

   d. `putchar('\b');`

3. Suppose you have an executable program named `count` that counts the characters in its input. Devise a command-line command using the `count` program to count the number of characters in the file `essay` and to store the result in a file named `essayct`.

4. Given the program and files in question 3, which of the following are valid commands?

   a. `essayct <essay`

   b. `count essay`

   c. `essay >count`

5. What is `EOF`?

6. What is the output of each of the following fragments for the indicated input (assume that `ch` is type `int` and that the input is buffered)?

   a. The input is as follows:

   ```
   If you quit, I will.[enter]
   ```

   The fragment is as follows:

   ```
   while ((ch = getchar()) != 'i')

       putchar(ch);
   ```

   b. The input is as follows:

   ```
   Harhar[enter]
   ```

   The fragment is as follows:

   ```
   while ((ch = getchar()) != '\n')

   {
      putchar(ch++);
      putchar(++ch);
   }
   ```

7. How does C deal with different computers systems having different file and newline conventions?

8. What potential problem do you face when intermixing numeric input with character input on a buffered system?

Programming Exercises

Several of the following programs ask for input to be terminated by `EOF`. If your operating system makes redirection awkward or impossible, use some other test for terminating input, such as reading the `&` character.

1. Devise a program that counts the number of characters in its input up to the end of file.

2. Write a program that reads input as a stream of characters until encountering `EOF`. Have the program print each input character and its ASCII decimal value. Note that characters preceding the space character in the ASCII sequence are nonprinting characters. Treat them specially. If the nonprinting character is a newline or tab, print `\n` or `\t`, respectively. Otherwise, use control-character notation. For instance, ASCII 1 is Ctrl+A, which can be displayed as `^A`. Note that the ASCII value for `A` is the value for Ctrl+A plus 64. A similar relation holds for the other nonprinting characters. Print 10 pairs per line, except start a fresh line each time a newline character is encountered. (Note: The operating system may have special interpretations for some control characters and keep them from reaching the program.)

3. Write a program that reads input as a stream of characters until encountering `EOF`. Have it report the number of uppercase letters, the number of lowercase letters, and the number of other characters in the input. You may assume that the numeric values for the lowercase letters are sequential and assume the same for uppercase. Or, more portably, you can use appropriate classification functions from the `ctype.h` library.

4. Write a program that reads input as a stream of characters until encountering `EOF`. Have it report the average number of letters per word. Don't count whitespace as being letters in a word. Actually, punctuation shouldn't be counted either, but don't worry about that now. (If you do want to worry about it, consider using the `ispunct()` function from the `ctype.h` family.)

5. Modify the guessing program of Listing 8.4 so that it uses a more intelligent guessing strategy. For example, have the program initially guess 50, and have it ask the user whether the guess is high, low, or correct. If, say, the guess is low, have the next guess be halfway between 50 and 100, that is, 75. If that guess is high, let the next guess be halfway between 75 and 50, and so on. Using this *binary search* strategy, the program quickly zeros in on the correct answer, at least if the user does not cheat.

6. Modify the `get_first()` function of Listing 8.8 so that it returns the first non-whitespace character encountered. Test it in a simple program.

7. Modify Programming Exercise 8 from Chapter 7 so that the menu choices are labeled by characters instead of by numbers; use `q` instead of `5` as the cue to terminate input.

8. Write a program that shows you a menu offering you the choice of addition, subtraction, multiplication, or division. After getting your choice, the program asks for two numbers, then performs the requested operation. The program should accept only the offered menu choices. It should use type `float` for the numbers and allow the user to try again if he or she fails to enter a number. In the case of division, the program should prompt the user to enter a new value if `0` is entered as the value for the second number. A typical program run should look like this:

```
Enter the operation of your choice:
a. add           s. subtract
m. multiply      d. divide
q. quit
a
Enter first number: 22.4
Enter second number: one
one is not an number.
Please enter a number, such as 2.5, -1.78E8, or 3: 1
22.4 + 1 = 23.4
Enter the operation of your choice:
a. add           s. subtract
m. multiply      d. divide
q. quit
d
Enter first number: 18.4
Enter second number: 0
Enter a number other than 0: 0.2
18.4 / 0.2 = 92
Enter the operation of your choice:
a. add           s. subtract
m. multiply      d. divide
q. quit
q
Bye.
```