C for Beginners

In the last few years, C has grown from a relatively obscure language used by a handful of programmers at universities, to a "must know" language throughout the computer industry. C has become known as a language that is powerful, fast, and efficient.

This chapter briefly introduces C. It is in two parts. Part 1 describes what a programming language is, and gives the history of the C programming language. This section also introduces some concepts basic to C, such as *structured programming*, *pointer*, and *operator*. Part 2 walks through a C programming session. It emphasizes how a C programmer deals with a real problem, and demonstrates some of the aspects of the language.

This chapter is not designed to teach you the entire C language. It will introduce you to C, so you can read the rest of this manual with some understanding. We urge you to look up individual topics of C programming in the Lexicon, and especially to study the example programs given there.

Programming languages and C

Before beginning with C, it is worthwhile to review how a microprocessor and a computer language work.

A *microprocessor* is the part of your computer that actually computes. Built into it is a group of *instructions*. Each instruction tells the microprocessor to perform a task; for example, one instruction adds two numbers together, another stores the result of an arithmetic operation in memory, and a third copies data from one point in memory to another.

Together, a microprocessor's instructions form its *instruction set*. The instruction set is, in effect, the microprocessor's "native language".

A microprocessor also contains areas of very fast storage, called *registers*. The registers are essential to arithmetic and data handling within the microprocessor. How many registers a microprocessor has, and how they are designed, help to determine how much memory the microprocessor can read and write, or *address*, and how the microprocessor handles data.

A *computer language*, as the name implies, lets a human being use the microprocessor's instruction set. The lowest level language is called "assembly language". In assembly language, the programmer calls instructions directly from the microcomputer's instruction set, and manipulates the registers within the microprocessor. To write programs in assembly language, a programmer must know both the microprocessor's instruction set and the configuration of its registers.

Assembly and high-level languages

With assembly language, the programmer can tailor the program specifically to the microprocessor. However, because each microprocessor has a unique instruction set and configuration of registers, a program written in one microprocessor's assembly language cannot be run on another microprocessor. For example, no program written in the assembly language for the Motorola 68000 microprocessor can be run on the IBM PC or any PC-compatible computer. The program must be entirely rewritten in the assembly language for the Intel 8086 microprocessor, which is difficult and time consuming.

A *high-level language* helps programmers to avoid these problems. The programmer does not need to know the microprocessor in detail; instead of specific microprocessor instructions, he writes a set of logical constructions. These constructions are then handed to another program, which translates them into the instructions and registers calls used by a specific microprocessor. In theory, a

program written in a high-level language can be run on any microprocessor for which someone has written a translation program.

A high-level language allows the programmer to concentrate on the task being executed, rather than on the details of registers and instructions. This means that programs can be written more quickly than in assembly language, and can be maintained more easily.

So, what is C?

C was invented in the mid-1970s by Dennis Ritchie, a programmer at Bell Laboratories. Ritchie created C specifically to re-write the UNIX operating system from PDP-11 assembly language. Ritchie designed C to have the power, speed, and flexibility of assembly language, but the portability of high-level languages.

In 1978, Ritchie and Brian W. Kernighan published *The C Programming Language*, which describes and defines the C language. *The C Programming Language* is the "bible" of C, a standard work to which all programmers can refer when writing their programs.

Because C is modeled after assembly language, it has been called a "medium-level" language. The programmer doesn't have to worry about specific registers or specific instructions, but he can use all of the power of the computer almost as directly as he can with assembly language. The price is that a C program often can be terse and difficult to understand.

Also, because C was written by experienced programmers for experienced programmers, it makes little effort to protect a programmer from himself. A programmer can easily write a C program that is legal and compiles correctly but crashes the system. Also, C's punctuation marks, or "operators", closely resemble each other. Thus, a mistake in typing can create a legal program that compiles correctly but behaves very differently from what you expect.

Structured programming

C is a *structured language*. This means that a C program is assembled from a number of sub-programs, or *functions*, each of which performs a discrete task. If this concept is difficult to grasp, consider the following example.

Suppose you want to turn a file of text into upper-case letters and print it on the screen. This job seems simple, but a program to do it must perform five tasks:

1. Accept the name of the file to open.
2. Open the file so it can be read, in much the same way that you must open a book before you can read it.
3. Read the text from the file.
4. Turn what is read into upper-case letters.
5. Finally, print the transformed text onto the screen.

A good program will also perform the following tasks:

1. Check that the file requested actually exists.
2. Check that the file requested is actually a text file rather than a file of binary information; the latter makes very little sense when printed on the screen.
3. Close the program neatly when the work is finished.
4. Stop processing and print an error message if a problem occurs.

Let's C

A structured language like C allows you to write a separate function for each of these tasks.

A structured programming language offers two major advantages over a non-structured language. First, it is easier to debug a function than an entire program because the function can be unplugged from the program as a whole, made to work correctly, and then plugged back in again. Second, once a function works, it can be used again and again in different programs. This allows you to create *libraries* of reliable functions that you can pull off the shelf whenever you need them.

The functions within a program communicate by passing values to each other. The value being passed can be an integer, a character, or—most commonly—an address within memory where a function can find data to manipulate. This passing of addresses, or *pointers*, is the most efficient way to manipulate data because by receiving one number, a function can find its way to a large amount of data. This speeds up a program's execution.

C adds some extra tools to help you construct programs. To begin, C allows you to store functions in compiled form. These precompiled functions are added only when the program is finally loaded into memory; this spares you the trouble of having to recompile the same code again and again. Second, C adds a preprocessor that expands definitions, or *macros*, and pulls in special material stored in *header files*. This allows you to store often-used definitions in one file and use them just by adding one line to your program.

Compiling a C program

When **Let's C** compiles a C program, it invokes a number of sub-programs, or *phases*, each of which performs part of the work of turning your file of C code into an executable program. The phases are as follows:

cpp The preprocessor. This reads the file of source code, adds any header files that you have requested, and expands any user-defined macros in the program.

cc0 The preprocessed file is then handed to **cc0**, the parser, which examines the program to see that it is written in legal C and translates it into a logical structure, or *tree*.

cc1 The output of the parser is then handed to **cc1**, the code generator, which translates the logical structure created by the parser into machine instructions.

cc2 The output of the code generator is then handed to **cc2**, the optimizer, which examines the code, eliminates redundant instructions, and then writes the object module file. The output of **cc2** is the relocatable object module, which always has the suffix **.obj**.

The relocatable object module is handed to MS-LINK, the linker, which opens the libraries and adds the library functions to create the executable program. What the linker does will be explained in more detail below.

This sounds complicated, and it is; for that reason, **Let's C** includes a command, called **cc**, that guides a program through the compilation process automatically. For example, to compile the program **test.c** with **Let's C**, all you have to do is type:

```
cc test.c
```

or use the MWS display interface, as described in section 1, *Tutorial Introduction*. **cc** takes care of the rest.

Writing a C program

As noted above, a C program consists of a bundle of sub-programs, or *functions*, which link together to perform the task you want done. Every C program must have at least one function that is called **main**. This is the main function; when the computer reads this, it knows that it must begin to execute the program. All other functions are subordinate to **main**. When the **main** function is finished, the program is over.

Here is a simple C program; all it does is print the message "Hello, world!" on the screen:

```
main()
{
        printf("Hello, world!\n");
}
```

As you can see, this program begins with the word **main**. The program begins to work at this point. The parentheses after **main** enclose all of the *arguments* to **main** — or would, if this program's **main** took any. An argument is an item of information that a function uses in its work.

The braces '{' and '}' enclose all the material that is subsidiary to **main**.

The word "printf" *calls* a function called **printf**. This function performs formatted printing. The line of characters (or "string") *Hello, world!* is the argument to **printf**: this argument is what **printf** is to print.

The characters '\n' stand for a carriage return; this ensures that when the program is finished, the cursor is not left fixed in the middle of the screen. Finally, the semicolon ';' at the end of the command indicates that the command is finished.

One point to remember is that **printf** is *not* part of the C language. Rather, it is a *function* that was written by Mark Williams Company, then compiled and stored in a library for later use. This means that you do not have to re-invent a formatted printing function to perform this simple task: all you have to do is *call* the one that Mark Williams Company has written for you.

Although most C programs are more complicated than this example, every C program has the same elements: a function called **main**, which marks where exection begins and ends; braces that fence off blocks of code; functions that are called from libraries; and data passed to functions in the form of arguments.

A sample C programming session

This section walks you through a C programming session. It shows how you can go about planning and writing a program in C.

C allows you to be precise in your programming, which should make you a stronger programmer. Be careful, however, because C does exactly what you tell it to do: if you make a mistake, you can produce a legal C program that does very unexpected things.

Designing a program

Most programmers prefer to work on a program that does something fun or useful. Therefore, we will write something useful: a version of the UNIX utility **more**. It will do the following:

1. Open a text file on disk.
2. Display its contents in 23-line portions (one full screen).
3. After a portion is displayed, wait to see if the user wants to see another portion. If the user presses the space bar, display another portion; if he types anything else, exit.
4. Exit automatically when the end of file is reached.

As you can see, the first step in writing a program is to write down what the program is to do, in as much detail as you can manage, and in complete sentences.

Now, invoke the MicroEMACS editor and get ready to type in the program. Use the command

```
me more.c
```

or use the MWS display interface as described in section 1 of this manual. Note that the suffix **.c** on the file name indicates that this is a file of C code. If you do not use this suffix, **Let's C** will not

recognize that this is a file of C code, and will refuse to compile it.

Begin by inserting a description of the program into the top of the file in the form of a *comment*. When a C compiler sees the symbol '/*', it throws away everything it reads until it sees the symbol '*/'. This lets you insert text into your program to explain what the program does.

Now, type the following:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */
```

Save what you have typed by pressing **<ctrl-X>** and then **<ctrl-S>**. Now, anyone, including you, who looks at this program will know exactly what it is meant to do.

The main function

As described earlier, the C language permits *structured programming*. This means that you can break your program into a group of discrete functions, each of which performs one task. Each function can be perfected by itself, and then used again and again when you need to execute its task. C requires, however, that you signal which function is the *main* function, the one that controls the operation of the other functions; thus, each C program must have a function called **main()**.

Now, add **main()** to your program. Type the code that is shaded, below:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

main()
{
}
```

The parentheses "()" show that **main** is a function; if **main** were to take any arguments, they would be named between the parentheses. The braces "{}" delimit all code that is subordinate to **main**; this will be explained in more detail below.

Note that the shortest legal C program is **main(){}**. This program doesn't do anything when you run it, but it will compile correctly and generate an executable file.

Now, try compiling the program. Save your text by typing **<ctrl-X><ctrl-S>**, and then exit from the editor by typing **<ctrl-X><ctrl-C>**. Compile the program by typing:

```
cc more.c
```

or use the MWS display interface, as described in section 1. When compilation is finished, type **more**. MS-DOS pauses for a moment, and then returns the prompt to your screen. As you can see, you now have a legal, compilable C program, but one that does nothing.

Opening a file and showing text

The next step is to install routines that open a file and print its contents. For the moment, the program will read only a file called **tester**, and not break it into 23-line portions.

Type the shaded lines into your program, as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>

main()
{
        char string[128];
        FILE *fileptr;

/* Open file */
        fileptr = fopen("tester", "r");

/* Read material and display it */
        for (;;)
        {
                fgets(string, 128, fileptr);
                printf("%s\n", string);
        }
}
```

Note first how comments are inserted into the text, to guide the reader.

Now, note the lines

```
        char string[128];
        FILE *fileptr;
```

These *declare* two data structures. That is, they tell **Let's C** to set aside a specific amount of memory for them.

The first declaration, **char string[128];**, declares an array of 128 **char**s. A **char** is a data entity that is exactly one byte long; this is enough space to store exactly one alphanumeric character in memory, hence its name. An *array* is a set of data elements that are recorded together in memory. In this instance, the declaration sets aside 128 **char**s-worth of memory. This declaration reserves space in memory to hold the data that your program reads.

The second declaration, **FILE *fileptr**, declares a *pointer* to a **FILE** structure. The asterisk shows that the data element points to something, rather than being the thing itself. When a variable is declared to be a pointer, **Let's C** sets aside enough space in memory to hold an *address*. When your program reads that address, it then knows where the actual data are residing, and looks for them there. C uses pointers extensively, because it is much more efficient to pass the address of data than to pass the data themselves. You may find the concept of pointers to be a little difficult to grasp; however, as you gain experience with C, you will find that they become easy to use.

The **FILE** structure is the data entity that holds all the information your program needs to read information from or write information to a file on the disk. For now, all you need to remember is that this declaration sets aside a place to hold a pointer to such a structure, and the structure itself holds all of the information your program needs to manipulate a file on disk. In effect, the variable **fileptr** is used within your program as a synonym for the file itself.

Now, the line

```
fileptr = fopen("tester", "r");
```

opens the file to be read. The function **fopen** *opens* the file, fills the **FILE** structure, and fills the variable **fileptr** with the address of where that structure resides in memory.

fopen takes two arguments. The first is the name of the file to be opened, within quotation marks. The second argument indicates the *mode* in which to open the file; **r** indicates that the file will be read only.

The lines

```
for(;;)
{
```

begin a *loop*. A loop is a section of code that is executed repeatedly until a condition that you set is met. For example, you may define a loop that executes until the value of a particular variable becomes greater than zero.

for is built into the C language. Note that it has braces, just like **main()** does; these braces mean that the following lines, up to the next right brace (}) are part of this loop. You can set conditions that control how a **for** loop operates; in its present form, it will loop forever. This will be explained in more detail shortly.

Two library functions are executed within the loop. The first,

```
fgets(string, 128, fileptr);
```

reads a line from the file named in the **fileptr** variable, and writes it into the character array called **string**. The middle argument ensures that no more than 128 characters will be read at a time. The second line within this loop,

```
printf("%s\n", string);
```

prints out the line. **printf** is a powerful and subtle function; in its present form, it prints on the screen the string named in the variable *string*.

Finally, the line at the top of the program

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

tells **Let's C** to read a *header file* called **stdio.h**. The term "STDIO" stands for "standard input and output"; **stdio.h** declares and defines a number of routines that will be used to read data from a file and write them onto the screen.

When you have finished typing in this code, again compile the program as you did earlier. If an error occurs, check what you have typed and make sure that it *exactly* matches the code shown on the previous page. If you find any errors, fix them and then recompile. If errors persist, see the sections *Error Messages* and *Questions and Answers* for help.

When compilation is finished, execute **more** as you did earlier. The file **tester** is included with **Let's C**. You will see the text from **tester** scroll across the screen. When the text is finished, however, the DOS prompt does not return; you have not yet inserted code that tells the program to recognize that the file is finished. Type **<ctrl-C>** to break the program and return to DOS.

Accepting file names

Of course, you will want **more** to be able to display the contents of any file, not just files named **tester**. The next step is to add code that lets you pass arguments to the program through its command line. This task requires that you give the **main()** function two arguments; by tradition, these are always called **argc** and **argv**. How they work will be described in a moment.

The enhanced program now appears as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
/* Declare arguments to main() */
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;

/* Open file */
        fileptr = fopen(argv[1], "r");

/* Read material and display it */
        for (;;)
        {
                fgets(string, MAXCHAR, fileptr);
                printf("%s\n", string);
        }
}
```

First, a small change has been added: the line

```
        #define MAXCHAR 128
```

defines the *manifest constant* **MAXCHAR** to be equivalent to 128. This is done because the "magic number" 128 is used throughout the program. If you decide to change the number of characters that this program can handle at once, all you would have to do is to change this one line to alter the entire program. This cuts down on mistakes in altering and updating the program. If you look lower in the program, you will see that the declaration

```
        char string[128]
```

has been changed to read

```
        char string[MAXCHAR]
```

The two forms are equivalent; the only difference is that the latter is easier to use. It is a good idea to use manifest constants wherever possible, to streamline changes to your program.

Now, look at the line that declares **main()**. You will see that **main()** has two arguments: **argc** and **argv**.

The first is an **int**, or integer, as shown by its declaration — **int argc;**. **argc** gives the *number* of entries typed on a command line. For example, when you typed

```
more filename
```

the value of **argc** was set to two: one for the command name itself, and one for the file-name argument. **argc** and its value are set by **Let's C**. You do not have to do anything to ensure that this value is set correctly.

argv, on the other hand, is an array of pointers to the command line's elements. In this instance, **argv[1]** points to name of the file that you want **more** to read. This, too, is set by **Let's C**, and works automatically.

If you look below at the line that delares **fopen()**, you will see that **tester** has been replaced with **argv[1]**; this means that you want **fopen()** to open the file named in the first argument to the **more** command.

Now, try running the program by typing

```
more tester
```

more will open **tester** and display its contents on the screen. You still need to type **<ctrl-C>** when the file is finished; the code to recognize the end of the file will be inserted later.

Also, be sure that you give the command only one file name as an argument, no more and no less. Code that checks against errors has not yet been inserted, and handing it the wrong number of arguments could cause MS-DOS to crash.

Error checking

Obviously, the program runs at this stage, but is still fragile, and could cause problems for you. The next step is to stabilize the program by writing code to check for errors. To do so, a programmer must first write code to capture error conditions, and then write a routine to react appropriately to an error.

Our edited program now appears as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
/* define arguments to main() */
int argc;
char *argv[];
{
       char string[MAXCHAR];
       FILE *fileptr;
```

```
/* Check if right number of arguments was passed */
        if ((argc-1) != 1)
                error("Usage: more filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Read material and display it */
        for (;;)
        {
                fgets(string, MAXCHAR, fileptr);
                printf("%s\n", string);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

The additions to the program are introduced by comments.

The first addition

```
        if ((argc-1) != 1)
                error("Usage:  more filename");
```

checks to see if the correct number of arguments was passed on the command line; that is to say, it checks to make sure that you named a file when you typed the **more** command.

As noted above, **argc** is the number of arguments on the command line, or rather, the number of arguments plus one, because the command name itself is always considered to be an argument. The statement **if((argc-1) != 1)** will check this. The **if** statement is built into C. If the condition defined between its parentheses is true, then do something, but if it is not true, do nothing at all. The operator **!=** means "does not equal". Therefore, our statement means that if **argc** minus one is not equal to one (in other words, if there is not one and only one argument to the **more** command), execute the function **error**. **error** is defined below.

Our **fopen** function also has some error checking added (which will be described in a moment):

```
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");
```

fopen will return a value called "NULL" if, for any reason, it cannot open the file you requested. Thus, our new **if** statement says that if **fopen** cannot open the file named on the first argument to the command line (that is, **argv[1]**), it should invoke the **error** function.

C always executes nested functions from the "inside out". That means that the innermost function (that is, the function that is enclosed most deeply within the pairs of parentheses) is executed first. Its result, or what it *returns*, is then passed to next outermost function as an argument; that function is then executed and what it returns is, in turn, passed to the function that encloses it, and so on. In this instance, the innermost function is

```
        fileptr = fopen(argv[1], "r")
```

fopen is executed and what it returns is written into **fileptr**. What **fopen** returned is then passed to

the next outer operation; in this case, it is compared with NULL, as follows:

```
(fileptr = fopen(argv[1], "r") == NULL
```

What that operation returns is then passed to the outermost function, in this case the **if** statement, which evaluates what it is passed, and acts accordingly. If **fileptr** is NULL (that is, if **fopen** couldn't open the file), the **if** statement will be true and the **error** function will be called. If, however, the file was opened, **fileptr** will not equal NULL and the program will proceed.

As this example shows, C allows a programmer to nest functions quite deeply. Although nested functions are sometimes difficult to untangle when you read them, they make programming much more convenient.

Finally, at the bottom of the file is a new function, called **error**:

```
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

This function stands outside of **main**, as you can tell because it appears outside of **main**'s closing brace. This function is called only when your program needs it. If there are no errors, the program progresses only until the closing brace and the **error** function is never called.

error takes one argument, the message that is to be printed on the screen. This message is defined by the routine that calls **error**. **error** uses the function **printf** to print the message, then calls the **exit** function; this, as its name implies, causes the program to stop. The argument **1** is a special signal that tells MS-DOS that something went wrong with your program.

When the error checking code is inserted, recompile the program without an argument. Previously, this would crash MS-DOS; now, all it does is print the message

```
Usage: more filename
```

and terminate the program.

Print a portion of a file

So far, our utility just opens a file and streams its contents over the screen. Now, you must insert code to print a 23-line portion of the file. At present, it will only print the first 23 lines, and then exit.

To do so, you must insert another **for** loop. Unlike our first loop, which ran forever, this one will cycle only 23 times, and then stop. Our updated program appears as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128
```

```
main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if ((argc-1) != 1)
                error("Usage: more filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines */
        for (;;)
        {
                for (ctr = 0; ctr < 23; ctr++)
                {
                        fgets(string, MAXCHAR, fileptr);
                        printf("%s\n", string);
                }
                exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

The new **for** loop is nested inside the loop governed by **for(;;)**. The program also declares a new variable, **ctr**, at the beginning of the program. **ctr** keeps track of how many times the loop has executed. Now, look at the line:

```
        for (ctr = 0; ctr < 23; ctr++)
```

It has three sub-statements, which are separated by semicolons. The first sub-statement sets **ctr** to zero; the second says that execution is to continue as long as **ctr** is less than 23; and the third says that **ctr** is to be increased by one every time the loop executes (this is indicated by the **++** appended to **ctr**). With each iteration of this loop, **fgets** reads a line from the file named on the **more** command line, and **printf** prints it on the screen.

Also, an **exit** call has been set after this new loop; this ensures that the program will exit automatically after the loop has finished executing. This is a temporary measure, to make sure that you no longer have to type **<ctrl-C>** to return to MS-DOS.

When you have updated the program, recompile it in the usual way. When you run it, **more** will show the first 23 lines of the file, and then the MS-DOS prompt will return.

The program is now approaching its final form.

Checking for the end of file

The next-to-last step in preparing the program is teaching it to recognize the end of a file when it sees it. This does not appear to be needed now because the program exits automatically after 23 lines or fewer, but it will be quite necessary when the program begins to display more than one 23-line portion of text.

The libraries included with **Let's C** include a function that checks for the end of file (or EOF); it is called **feof()**. Before the program attempts to print out a line of text, it should check if the end of the file has been reached. This means placing **feof** in an **if** statement; the statement will take advantage of the fact that **feof** outputs, or *returns*, a zero if the end of file has not been reached, and returns a number other than zero if the end of file has been reached. The **if** statement will capture what **feof** returns, and continue execution as long as the value of the number returned is zero.

The updated program now appears as follows:

```
/*
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if ((argc-1) != 1)
                error("Usage: more filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");
```

```
/* Output 23 lines, while checking for EOF */
        for (;;)
        {
                for (ctr = 0; ctr < 23; ctr++)
                {
                        if (feof(fileptr) == 0)
                        {
                                fgets(string, MAXCHAR, fileptr);
                                printf("%s\n", string);
                        }
                        else
                                exit(0);
                }
                exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

First, note that the comment that describes the program's output has been changed to reflect our changes to the program. It is important for a programmer to ensure that the comments and the code are in step with each other.

Our new **if** statement

```
        if (feof(fileptr) == 0)
        {
```

checks what **feof** returns: if it returns zero, the end of the file has not been reached, the **if** statement is true, and the program prints out the next line. If it returns a number other than zero, the end of file has been reached, the **if** statement is false so the **else** statement is executed, which causes **more** to exit. **feof** takes one argument, which is the **FILE** that was defined by **fopen**.

Note, too, that a new control statement is introduced: **else**. This, like **if**, is built into the C language. An **else** statement is always paired with an **if** statement; together, they mean that if the condition for which **if** is testing is true, the program should do one thing; otherwise, it should do something else. In this case, the program says that if the end of file has not been reached, another line should be read from the file and printed on the screen; however, if it has been reached, then the program should exit. As you can imagine, **if/else** pairs are common in C programming; they are logical and useful.

One more task must be done on our program; then it is finished.

Polling the keyboard

For the program to be complete, it has to ask you if you want to see another 23-line portion of text. The program should write another portion if you press the space bar, and exit if you type anything else.

The program will use a new function, **getcnb**, to accomplish this task. **getcnb** reads what you type in an unbuffered fashion; that means that you do not have to type the carriage return key for the keystroke to be read by the program. This is placed within an **if** statement that compares what character is typed with the space character. If they are not the same (as indicated by the operator **!=**), the program will exit; otherwise, it will loop through again and show another 23 lines.

Let's C

When these changes are inserted, the program is complete:

```
 * Truncated version of the 'more' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <space>,
 * print another 23 lines,
 * if user types any other key, exit.
 * Exit when EOF is read.
 */

#include <stdio.h>
#define MAXCHAR 128

main(argc, argv)
int argc;
char *argv[];
{
        char string[MAXCHAR];
        FILE *fileptr;
        int ctr;

/* Check if right number of arguments was passed */
        if ((argc-1) != 1)
                error("Usage: more filename");

/* Open file */
        if ((fileptr = fopen(argv[1], "r")) == NULL)
                error("Cannot open file");

/* Output 23 lines, while checking for EOF */
        for (;;)
        {
                for (ctr = 0; ctr < 23; ctr++)
                {
                        if (feof(fileptr) == 0)
                        {
                                fgets(string, MAXCHAR, fileptr);
                                printf("%s\n", string);
                        }
                        else
                                exit(0);
                }
/* Read keyboard; exit if not <space> */
                if (getcnb() != ' ')
                        exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s\n", message);
        exit(1);
}
```

After you have inserted these changes, again compile the program.

When compilation is finished, try typing

```
more more.c
```

The first 23 lines of the source code to the program now appear on your screen. Hit the space bar; the next 23 lines appear. Now, type any other key: the program exits.

You now have a simple but helpful **more** utility.

For more information

This section has given you a brief, concentrated introduction to writing a C program. If you are new to programming, much of what happened must seemed strange, but we hope it helped you to appreciate the logic of how C works.

Numerous books are on the market to teach beginners how to program in C; see the bibliography at the end of section 1 of this manual for a list of them. Also, look at the sample C programs in the Lexicon. These demonstrate how to use many of the functions available to you with **Let's C**.

With patience, you should discover that programming with C is one of the greatest pleasures to be had with a computer: few feats are as satisfying as delving into the machine and having it do exactly what you want it to do.

Where to go from here

The following section, *Advanced compiling*, introduces some of the more sophisticated features of the **Let's C** compiler. You should look through this section when you feel that you are ready for advanced programming.

If you have any questions about any of the features of **Let's C**, or about any of the functions that were described in this tutorial, look in the Lexicon. For example, if you have a question about **feof** or **printf**, look them up in the Lexicon. There, you will full descriptions of how to use them, plus sample C programs that show how to use them. By typing, compiling, and running the sample programs, you will quickly learn how to use the C language.