The C Language

C is a computer language invented by Dennis Ritchie and Ken Thompson at AT&T Bell Laboratories in the early 1970s. In the approximately 25 years since its creation, C has become one of the most popular computer languages in the world. C is powerful and flexible, and it is highly portable. It has been implemented on practically every computer, and under practically every operating system, in the world.

C is the "native language" of the COHERENT system. COHERENT is written in C, and it includes a powerful C compiler among its suite of language tools for your use. You do not need to know C to use COHERENT to great advantage; however, if you plan to program under COHERENT, you would be well advised to become at least passably acquainted with it.

This tutorial is an introduction to the COHERENT C compiler and to the C language itself. The first part of this section describes how to compile programs under COHERENT. The second part is a brief tutorial in the C language.

Compiling C Programs under COHERENT

A C compiler is a program that transforms files of C source code into machine code. Compilation is a complex process that involves several steps; however, COHERENT simplifies it with the command **cc**, which controls all the actions of the compiler.

Try the Compiler

Before we launch into a lengthy explanation of what **cc** is and what it does, you can get a feel for it by trying it with a simple example. To begin, type the following to create a simple C program:

```
cat >hello.c
main() {
        printf("Hello, world\n");
}
<ctrl-D>
```

This creates a simple C program called **hello.c**. Now, compile your program by typing the following command:

```
cc -V hello.c
```

If you typed the program correctly, **cc** prints something like the following on your screen:

```
cc0 D2B000000201 hello.c  0x418CB8
cc1 D2B000000201 0x418CB8 0x408CB4
cc2a D2B000000201 0x408CB4 0x418CB8
cc2b D2B000000201 0x418CB8 hello.o
/bin/ld -X -o hello /lib/crts0.o hello.o /lib/libc.a -Z hello.o
```

What each of these messages means will be described below. If you receive an error message, try re-typing the program, and then re-compile it. When compilation is successfully completed, you will now have an executable program called **hello**. To invoke it, type:

```
hello
```

It should print the following on your screen:

```
Hello, world
```

As you can see, **cc** makes it easy to transform a file of C code into an executable program.

Phases of Compilation

As you noticed, **cc** printed a number of messages on your screen as it compiled **hello.c**. The reason you saw the messages was that compilation was performed with the **-V** option to **cc**; this tells **cc** to print a verbose output that describes each of its actions. **cc** prints numerous messages because the COHERENT C compiler is not just one program, but a number of different programs that work together. Each program performs a *phase* of compilation. The following summarizes each phase:

cpp The C preprocessor. This processes any of the '#' directives, such as **#include** or **#ifdef**, and expands macros.

cc0 The parser. This phase parses programs. It translates the program into a parse-tree format, which is independent of both the language of the source code and the microprocessor for which code will be generated.

cc1 The code generator. This phase reads the parse tree generated by **cc0** and translates it into machine code. The code generation is table driven, with entries for each operator and addressing mode.

cc2a The optimizer generator. This phase optimizes the generated code.

cc2b The optimizer generator. This phase writes the object module.

cc3 COHERENT also includes a fifth phase, called **cc3**, which can be run after the object generator, **cc2**. **cc3** generates a file of assembly language instead of a relocatable object module. **cc3** allows you to examine the code generated by the compiler. You did not see this phase when you compiled **hello.c** because this phase is optional and you did not request it. If you want COHERENT to generate assembly language, use the **-S** option on the **cc** command line.

Unless you specify the **-S** option, **cc** creates an *object module* that is named after the source file being compiled. This module has the suffix **.o**. An object module is *not* executable; it contains only the code generated by compiling a C source file, plus information needed to link the module with other program modules and with the library functions.

As the final step in its execution, **cc** calls the linker **ld** to produce an executable program.

Renaming Executable Files

When **cc** compiles a source file, by default it names the executable program after the *first* source file named on the **cc** command line. If you wish, you can give the executable file a different name. Use the **-o** (output) option, followed by the desired name.

Floating-Point Numbers

Often, you will need to use floating-point numbers in your programs. If you are unsure what a floating-point number is, see the Lexicon entry for **float**.

The routines that print floating-point numbers are large, and most C programs do not need to print floating-point numbers; therefore, the code to perform floating-point arithmetic is not included in a program by default. You must ask **cc** to include these routines with your program by using the **-f** option to **cc**.

To see how this works, let's modify **hello.c** to use floating-point numbers. Edit **hello.c** by typing the following commands:

```
ed hello.c
2
c
        printf("Hello, world %f\n", 123.4);
.
w
q
```

Now, compile the program with the same command line as before:

```
cc -V hello.c
```

When compilation has finished, type **hello**. You'll see the following output:

```
You must compile with the -f flag
to include printf() floating point.
Hello, world
```

COHERENT is telling you that you are using a floating-point number but that you did not compile the program to include code to process floating-point numbers. Now, recompile the program using the **-f** option to **cc**:

```
cc -V -f hello.c
```

When compilation has finished, type **hello**. If you typed the program correctly, you will see the following:

```
Hello, world 123.400000
```

As you can see, **hello** is now displaying the floating-point number **123.4** for you. For detailed information on **printf()**, see its entry in the Lexicon; **printf()** is also discussed in the tutorial section below.

Compiling Multiple Source Files

Many programs are built from more than one file of C source code. For example, the program **factor**, which is provided with COHERENT, is built from the C source files **factor.c** and **atod.c**. To produce the executable program **factor**, both source files must be compiled; the linker **ld** then joins them to form an executable file.

To compile a program that uses more than one source file, type all of the source files onto the **cc** command line. For example, to compile **factor** you would type the following:

```
cc -o factor -f factor.c atod.c -lm
```

This command compiles both C source files to create the program **factor**.

In the above example, **cc** produces the non-executable object modules **factor.o** and **atod.o**, and then links them to produce the executable file **factor**.

The argument **-lm** tells **cc** to include routines from the mathematics library when the object modules are linked. This option must come *after* the names of all of the source files, or the program will not be linked correctly.

Linking Without Compiling

When you are writing a program that consists of several source files, you will need to compile the program, test it, and then change one or more of the source files. Rather than recompile all of the source files, you can save time by recompiling only the modified files and relinking the program.

For example, if you modify the **factor** program by changing the source file **factor.c**, you can recompile **factor.c** *and* relink the entire program with the following command:

```
cc -o factor -f factor.c atod.o -lm
```

This **cc** command refers to the C source file **factor.c** and the *object module* **atod.o**. **cc** recognizes that **atod.o** is an object module and simply passes it to the linker **ld** without re-compiling it. You will find this particularly useful when your programs consist of many source files and you need to compile only a few of them.

To simplify compiling, especially if you are developing systems that use many source modules, you should consider using the **make** utility that is included with COHERENT. For more information on **make**, see its entry in the Lexicon, or see the tutorial for **make** that appears later in this manual.

Compiling Without Linking

At times, you will need to compile a source file but not link the resulting object module to the other object modules. You will do this, for example, to compile a module that you wish to insert into a library. Use **cc**'s option **-c** to tell **cc** not to link the compiled program. This option is often used to create relocatable object modules that can be archived into a library for later use.

For example, if you wanted just to compile **factor.c** without linking it, you would type:

```
cc -c factor.c
```

To link the resulting object module with the object module **atod.o** and with the appropriate libraries, type the following command:

```
cc -o factor -f factor.o atod.o -lm
```

Assembly-Language Files

C makes most assembly language programming unnecessary. However, you may wish to write small parts of your programs in assembly language for greater speed or to access processor features that C cannot use directly. COHERENT includes an assembler, named **as**, which is described in detail in the Lexicon.

To compile a program that consists of the C source file **example.c** and the assembly-language source file **example.s**, simply use the **cc** command as usual:

```
cc -o example example1.c example2.s
```

cc recognizes that the suffix **.s** indicates an assembly-language source file, and assembles it with **as**; then it links both object modules to produce an executable file.

Changing the Size of the Stack

The *stack* is the segment of memory that holds function arguments, local variables, and function return addresses. COHERENT takes advantage of the 80386 microprocessor's ability to allocate stack dynamically.

Where To Go From Here

This discussion of the **cc** command is by no means complete, but it includes enough information for you to begin to compile your programs. The Lexicon's entry for **cc** gives all of the command-line options available with **cc**. The Lexicon also has entries for **cpp**, the compiler phases, and for the linker **ld**, and describes them at greater length. All error messages generated by **cc** and by the assembler **as** appear in the appendix to this manual.

The next section in this tutorial introduces the C programming language.

C for Beginners

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

This chapter is not designed to teach you the entire C language. It introduces 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 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 register 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?

As noted earlier, C was invented at AT&T Bell Laboratories by Dennis Ritchie and Ken Thompson. They 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 described and defined the C language. In 1988, the American National Standards Institute (ANSI) published its standard for the C language. This standard has, on the whole, become the basis for current implementations of C.

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.

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 when run. 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. Read 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. 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.

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 a *library* 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.

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 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.

To see how these elements work, review the program **hello.c**, which you worked with earlier in this tutorial:

```
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 newline character. This character "tosses the carriage", or moves the cursor to a new line and returns it to the leftmost column on your screen. Using this character ensures that when printing 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 function call 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 your 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 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 execution 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, nothing more and nothing less. 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 COHERENT utility **scat**, that we'll call **display**. It will do the following:

1. Open a text file on disk.
2. Display its contents in 23-line chunks (one full screen).
3. After displaying a chunk, wait to see if the user wants to see another chunk. If the user presses the **<return>** key alone, display another chunk; if the user types any other key before pressing the **<return>** key, 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 preferably in complete sentences.

Now, invoke **ed** or MicroEMACS and get ready to type in the program:

```
ed display.c
```

or:

```
me display.c
```

We suggest that you use the MicroEMACS editor, because this tutorial will make numerous changes to the program as it progresses and it will be easier to see these changes in context if you use a screen editor rather than a line editor. The rest of this tutorial assumes that you are using MicroEMACS. If you are not familiar with MicroEMACS, it is briefly described in *Using the COHERENT System*. A tutorial for MicroEMACS also appears in this manual, or you may wish to see the entry for **me** in the Lexicon.

In the above commands, the suffix **.c** on the file name indicates that this is a file of C code. If you do not use this suffix, the **cc** command 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.

Type the following:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, 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 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, 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 display.c
```

When compilation is finished, type **display**. The shell will pause briefly, then return the prompt to your screen. As you can see, you now have a legal, compilable C program, but one that does nothing.

Open a File and Show 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 **display.c**, and not break it into 23-line portions.

Type the shaded lines into your program, as follows:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, 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("display.c", "r");

/* Read material and display it */
	for (;;) {
		fgets(string, 128, fileptr);
		printf("%s", 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 COHERENT 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, the C compiler 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 a detailed discussion of the **FILE** structure, see its entry in the Lexicon. 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("display.c", "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 rather than written into.

The lines

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

begin a *loop*. A loop is a section of code that is executed repeatedly until a condition that you set is fulfilled. 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", string);
```

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

Finally, the line at the top of the program:

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

tells the C preprocessor **cpp** to read the *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, check it in the table of error messages that appear at the end of this tutorial.

When compilation is finished, execute **display** as you did earlier. You will see the text from **display.c** scroll across the screen. When the text is finished, however, the COHERENT 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 COHERENT.

Accepting File Names

Of course, you will want **display** to be able to display the contents of any file, not just files named **display.c**. 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 appears as follows. You should change or insert the lines that are shaded:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, 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", 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()** now 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

```
display 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 the compiler. 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 arguments. In this instance, **argv[1]** points to name of the file that you want **display** to read. This, too, is set by COHERENT, and works automatically.

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

Now, try running the program by typing

```
display display.c
```

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 problems for you.

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

Error Checking

Obviously, the program runs at this stage, but is still fragile, and could cause problems. 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 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, 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 != 2)
		error("Usage: display 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", string);
	}
}

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

The additions to the program are introduced by comments.

The first addition

```
	if (argc != 2)
		error("Usage:  display 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 **display** 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 != 2)** checks 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** is not equal to two (in other words, if there are not two and only two arguments to the **display** command — the command name itself plus a file name), 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() returns 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 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 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", 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 in **main** 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 COHERENT that something went wrong with your program.

When the error checking code is inserted, recompile the program and execute it without an argument. Previously, this would cause the program to crash; now, all it does is print the message

```
Usage: display 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 ’scat’ utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, 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 != 2)
		error("Usage: display 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", string);
		}
		exit(0);
	}
}

/* Process error messages */
error(message)
char *message;
{
	printf("%s", 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 **display** 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 the shell.

When you have updated the program, recompile it in the usual way. When you run it with an appropriate file of an appropriate length, e.g., **display.c** itself, **display** will show the first 23 lines of the file, and then the shell's 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 function **fgets()** checks to see if it has arrived at the end of a file, and returns a special value if it has. **fgets()** normally returns the address of the string into which it writes its output; however, if it runs into the end of a file (or if any other error occurs), it returns the special value NULL. By reading the value of what **fgets()** returns, **display** can detect if the end of the file has been encountered, and stop reading. To do so, the **fgets()** statement must be set within an **if** statement. The **if** statement will capture what **fgets()** returns, and continue execution as long as the value of the number returned is not NULL.

The updated program now appears as follows:

```
/*
 * Truncated version of the ’scat’ utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, 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 != 2)
                error("Usage: display 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 (fgets(string, MAXCHAR, fileptr) != NULL)
                                printf("%s", string);
                        else
                                exit(0);
                }
                exit(0);
        }
}

/* Process error messages */
error(message)
char *message;
{
        printf("%s", 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 (fgets(string, MAXCHAR, fileptr) != NULL)
```

checks what **fgets()** returns: if it does not return NULL, the end of the file has not been reached, the **if** statement is true and the program prints out the next line. (The operator **!=** indicates "not equal".) If it returns NULL, however, the end of file has been reached, the **if** statement is false so the **else** statement is executed, which causes **display** to exit.

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 has been read from the file and should be 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 whenever the argument contains more than 23 lines. The program should write another portion if you press the **<return>** key alone; if you type any other key before you press **<return>**, then it should exit.

To do so, we will print a query on the screen, then read what the user has typed and interpret it. When these changes are inserted, the program is complete:

```
/*
 * Truncated version of the 'scat' utility.
 * Open a file, print out 23 lines, wait.
 * If user types <return>, 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 != 2)
                error("Usage: display 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 (fgets(string, MAXCHAR, fileptr) != NULL)
                                printf("%s", string);
                        else
                                exit(0);
```

```
                }
/* Query if user wishes to continue */
                printf("Continue? ");
                fflush(stdout);
                fgets(string, MAXCHAR, stdin);

                if (string[0] != '\n')
                        exit(0);
        }
}

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

These new lines introduce a few new twists. The lines

```
        printf("Continue? ");
        fflush(stdout);
```

print the prompt **Continue?** on the screen. Note that no '\n' appears after the prompt; this ensures that the cursor does *not* jump to the next line, but stays next to the prompt. Because no '\n' appears after the line, however, you have to force it to appear on the screen; this is accomplished with the statement:

```
        fflush(stdout);
```

fflush() flushes matter to an output device. **stdout** points to a file stream, just like the stream that you opened with the call to **fopen()**, earlier in the program. **stdout** is opened in the header file **stdio.h**, which was read at the beginning of the program; it always points to the user's screen.

The next line reads the user's keyboard:

```
        fgets(string, MAXCHAR, stdin);
```

This version of **fgets** reads matter into our array **string**; however, instead of reading the file pointed to by **fileptr**, it reads what is pointed to by **stdin**. **stdin** is a stream that is also defined in **stdio.h**; it always points to the user's keyboard.

Finally, the statement

```
        if (string[0] != '\n')
```

checks what the user typed by reading the first (that is, the zero-th) character written in the array **string** by the preceding call to **fgets()**. (Note that with C, counting always begins with zero rather than one.) If the user just types **<return>**, then **string[0]** will hold '\n'; and the **if** statement will *not* be true, the program jumps to the preceding **for** statement, and more text is written to the screen. However, if the user types anything before typing **<return>**, the **if** statement will succeed and the program will exit. This may seem a little convoluted, but it actually is a straightforward and efficient way to receive information from the user.

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

When compilation is finished, try typing

```
        display display.c
```

The first 23 lines of the source code to the program now appear on your screen. Hit **<return>**; the next 23 lines appear. Now, type any other key, and then press **<return>**: the program exits.

You now have a simple but helpful **display** 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 seem 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; the following section gives a small bibliography of books on C. Also, look at the sample C programs in the Lexicon. These demonstrate how to use many of the functions available to you with COHERENT.

Bibliography

The following books may be helpful in developing your skills with C. This list also contains all books that are referenced in this manual. It is by no means exhaustive; however, it should prove helpful to both beginners and experienced programmers.

American National Standards Institute: *Draft Programming Language C (October 1986 Draft).* Washington, D.C.: X3 Secretariat, Computer and Business Equipment Manufacturers Association, 1986.

AT&T Bell Laboratories: *The C Programmer's Handbook.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985.

Bentley, J.: *Programming Pearls.* Reading, Mass.: Addison-Wesley Publishing Company, 1986. *Not, strictly speaking, about C — but belongs on every programmer's bookshelf.*

Brooks, F.P., Jr.: *The Mythical Man-Month: Essays on Software Engineering.* Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1975. *Not about programming, but should be read by every programmer.*

Chirlin, P.M.: *Introduction to C.* Beaverton, Or.: Matrix Publishers, Inc., 1984.

Derman, B. (ed.): *Applied C.* New York: Van Nostrand Reinhold Co., Inc., 1986.

Feuer, A.R.: *The C Puzzle Book.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1982.

Gehani, G.: *Advanced C: Food for the Educated Palate.* Rockville, Md.: Computer Science Press, 1985.

Hancock, L.: Krieger, M.: *The C Primer.* New York: McGraw-Hill Book Publishers, Inc., 1982.

Harbison, S.; Steele, G.: *C: A Reference Manual.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Haviland, K.F., Salama, B.: *UNIX System Programming.* Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1987.

Hogan, T.: *The C Programmer's Handbook.* Bowie, Md.: Brady Publishing, 1984.

Kelley, A.; Pohl, I.: *C by Dissection: The Essentials of C Programming.* Menlo Park, Ca.: The Benjamin/Cummings Publishing Company, Inc., 1987.

Kernighan, B.W.; Ritchie, D.M.: *The C Programming Language.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1978.

Kernighan, B.W.; Plauger, P.J.: *The Elements of Programming Style*, ed. 2. New York: McGraw-Hill Book Co., 1978.

Kochan, S.G.: *Programming in C.* Hasbrouck Heights, N.J.: Hayden Book Co., Inc., 1983.

Knuth, D.E.: *The Art of Computer Programming*, vol. 1: *Basic Algorithms.* Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 2: *Seminumerical Algorithms.* Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 3: *Sorting and Searching.* Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Lapin, J.E.: *Portable C and UNIX System Programming.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1987.

Mark Williams Company: *ANSI C: A Lexical Guide.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Plum, T.: *C Programming Guidelines.* Cardiff, N.J.: Plum Hall, Inc., 1984.

Plum, T.; Brodie, J.: *Efficient C.* Cardiff, NJ: Plum Hall, Inc., 1985.

Purdum, J.: *C Programming Guide*. Indianapolis: Que Corp., 1983.

Purdum, J.; Leslie, T.C.; Stegemoller, A.L.: *C Programmer's Library*. Indianapolis: Que Corp., 1984.

Rochkind, M.J.: *Advanced UNIX Programming.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985.

Traister, R.J.: *Going from BASIC to C*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

Traister, R.J.: *Mastering C Pointers*. New York: Academic Press, Inc., 1990.

Traister, R.J.: *Programming in C for the Microprocessor User*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

Vile, R.C., Jr.: *Programming in C with Let's C*. Glenview, IL: Scott, Foresman and Company, 1988.

Waite, M.; Prata, S.; Martin, D.: *C Primer Plus*. Indianapolis: Howard W. Sams, Inc., 1984.

Weber Systems, Inc.: *C Language User's Handbook*. New York: Ballantine Books, 1984.

Zahn, C.T.: *C Notes*. New York: Yourdan Press, 1979.