1.10 Operating System Considerations

In this section, we discuss a number of topics that are system-dependent. We begin with the mechanics of writing and running a C program.

Writing and Running a C Program

The precise steps that have to be followed to create a file containing C code and to compile and execute it depend on three things: the operating system, the text editor, and the compiler. However, in all cases the general procedure is the same. We first describe in some detail how it is done in a UNIX environment. Then we discuss how it is done in an MS-DOS environment.

In the discussion that follows, we will be using the *cc* command to invoke the C compiler. In reality, however, the command depends on the compiler that is being used. For example, if we were using the command line version of the Turbo C compiler from Borland, then we would use the command *tcc* instead of *cc*. (For a list of C compilers, see the table in Section 11.13, “The C Compiler,” on page 522.)

Steps to be followed in writing and running a C program

1 Using an editor, create a text file, say *pgm.c*, that contains a C program. The name of the file must end with .c, indicating that the file contains C source code. For example, to use the *vi* editor on a UNIX system, we would give the command

   vi pgm.c

   To use an editor, the programmer must know the appropriate commands for inserting and modifying text.

2 Compile the program. This can be done with the command

   cc pgm.c

   The cc command invokes in turn the preprocessor, the compiler, and the loader. The preprocessor modifies a copy of the source code according to the preprocessing directives and produces what is called a *translation unit*. The compiler translates the translation unit into object code. If there are errors, then the programmer must start again at step 1 with the editing of the source file. Errors that occur at this stage are called *syntax errors* or *compile-time errors*. If there are no errors, then the loader uses the object code produced by the compiler, along with object code obtained from various libraries provided by the system, to create the executable file *a.out*. The program is now ready to be executed.

3 Execute the program. This is done with the command

   a.out

   Typically, the program will complete execution, and a system prompt will reappear on the screen. Any errors that occur during execution are called *run-time errors*. If for some reason the program needs to be changed, the programmer must start again at step 1.

If we compile a different program, then the file *a.out* will be overwritten, and its previous contents lost. If the contents of the executable file *a.out* are to be saved, then the file must be moved, or renamed. Suppose that we give the command

cc sea.c

This causes executable code to be written automatically into *a.out*. To save this file, we can give the command

mv a.out sea

This causes *a.out* to be moved to *sea*. Now the program can be executed by giving the command

sea

In UNIX, it is common practice to give the executable file the same name as the corresponding source file, except to drop the .c suffix. If we wish, we can use the *–o* option to direct the output of the *cc* command. For example, the command

cc –o sea sea.c

causes the executable output from *cc* to be written directly into *sea*, leaving intact whatever is in *a.out*.

Different kinds of errors can occur in a program. Syntax errors are caught by the compiler, whereas run-time errors manifest themselves only during program execution. For example, if an attempt to divide by zero is encoded into a program, a run-time error may occur when the program is executed. (See exercise 5, on page 59, and exercise 6, on page 59.) Usually, an error message produced by a run-time error is not very helpful in finding the trouble.

Let us now consider an MS-DOS environment. Here, some other text editor would most likely be used. Some C systems, such as Turbo C, have both a command line environment and an integrated environment. The integrated environment includes both the text editor and the compiler. (Consult Turbo C manuals for details.) In both MS-DOS and UNIX, the command that invokes the C compiler depends on which C compiler is being used. In MS-DOS, the executable output produced by a C compiler is written to a file having the same name as the source file, but with the extension .exe instead of .c. Suppose, for example, that we are using the command line environment in Turbo C. If we give the command

tcc sea.c

then the executable code will be written to *sea.exe*. To execute the program, we give the command

sea.exe or equivalently *sea*

To invoke the program, we do not need to type the .exe extension. If we wish to rename this file, we can use the *rename* command.

Interrupting a Program

When running a program, the user may want to interrupt, or kill, the program. For example, the program may be in an infinite loop. (In an interactive environment it is not necessarily wrong to use an infinite loop in a program.) Throughout this text we assume that the user knows how to interrupt a program. In MS-DOS and in UNIX, a control-c is commonly used to effect an interrupt. On some systems a special key, such as *delete* or *rubout* is used. Make sure that you know how to interrupt a program on your system.

Typing an End-of-file Signal

When a program is taking its input from the keyboard, it may be necessary to generate an end-of-file signal for the program to work properly. In UNIX, a carriage return followed by a control-d is the typical way to effect an end-of-file signal. (See exercise 26, on page 66, for further discussion.)

Redirection of the Input and the Output

Many operating systems, including MS-DOS and UNIX, can redirect the input and the output. To understand how this works, first consider the UNIX command

```
ls
```

This command causes a list of files and directories to be written to the screen. (The comparable command in MS-DOS is *dir*.) Now consider the command

```
ls > tmp
```

The symbol > causes the operating system to redirect the output of the command to the file *tmp*. What was written to the screen before is now written to the file *tmp*.

Our next program is called *dbl_out*. It can be used with redirection of both the input and the output. The program reads characters from the standard input file, which is normally connected to the keyboard, and writes each character twice to the standard output file, which is normally connected to the screen.

In file dbl_out.c

```
#include <stdio.h>

int main(void)
{
  char   c;

  while (scanf("%c", &c) == 1) {
    printf("%c", c);
    printf("%c", c);
  }
  return 0;
}
```

If we compile the program and put the executable code in the file *dbl_out*, then, using redirection, we can invoke the program in four ways:

```
dbl_out
dbl_out < infile
dbl_out > outfile
dbl_out < infile > outfile
```

Used in this context, the symbols < and > can be thought of as arrows. (See exercise 26, on page 66, for further discussion.)

Some commands are not meant to be used with redirection. For example, the *ls* command does not read characters from the keyboard. Therefore, it makes no sense to redirect the input to the *ls* command; because it does not take keyboard input, there is nothing to redirect.