In a similar manner, if you send standard output of **is.term** through a pipeline, test reports standard output is not associated with a terminal. In this example, cat copies standard input to standard output:

```
$ ./is.term | cat
FD 1 (stdout) is NOT going to the screen
```

optional You can also experiment with test on the command line. This technique allows you to make changes to your experimental code quickly by taking advantage of command history and editing (page 328). To better understand the following examples, first verify that test (called as [ ]) returns a value of 0 (*true*) when file descriptor 1 is associated with the screen and a value other than 0 (*false*) when file descriptor 1 is not associated with the screen. The **$?** special parameter (page 465) holds the exit status of the previous command.

```
$ [ -t 1 ]
$ echo $?
0

$ [ -t 1 ] > hold
$ echo $?
1
```

As explained on page 292, the && (AND) control operator first executes the command preceding it. Only if that command returns a value of 0 (*true*) does && execute the command following it. In the following example, if **[–t 1]** returns 0, && executes **echo "FD 1 to screen"**. Although the parentheses (page 292) are not required in this example, they are needed in the next one.

```
$ ( [ -t 1 ] && echo "FD 1 to screen" )
FD 1 to screen
```

Next, the output from the same command line is sent through a pipeline to cat, so test returns 1 (*false*) and && does not execute echo.

```
$ ( [ -t 1 ] && echo "FD 1 to screen" ) | cat
$
```

The following example is the same as the previous one, except test checks whether file descriptor 2 is associated with the screen. Because the pipeline redirects only standard output, test returns 0 (*true*) and && executes echo.

```
$ ( [ -t 2 ] && echo "FD 2 to screen" ) | cat
FD 2 to screen
```

In this example, test checks whether file descriptor 2 is associated with the screen(it is) and echo sends its output to file descriptor 1 (which goes through the pipeline to cat).

PARAMETERS

Shell parameters were introduced on page 300. This section goes into more detail about positional parameters and special parameters.

POSITIONAL PARAMETERS

Positional parameters comprise the command name and command-line arguments. These parameters are called *positional* because you refer to them by their position on the command line. You cannot use an assignment statement to change the value of a positional parameter. However, the bash set builtin (page 460) enables you to change the value of any positional parameter except the name of the calling program (the command name). The tcsh set builtin does not change the values of positional parameters.

$0: NAME OF THE CALLING PROGRAM

The shell expands **$0** to the name of the calling program (the command you used to call the program—usually the name of the program you are running). This parameter is numbered zero because it appears before the first argument on the command line:

```
$ cat abc
echo "This script was called by typing $0"
$ ./abc
This script was called by typing ./abc
$ /home/sam/abc
This script was called by typing /home/sam/abc
```

The preceding shell script uses echo to verify the way the script you are executing was called. You can use the basename utility and command substitution to extract the simple filename of the script:

```
$ cat abc2
echo "This script was called by typing $(basename $0)"
$ /home/sam/abc2
This script was called by typing abc2
```

When you call a script through a link, the shell expands **$0** to the value of the link. The busybox utility (page 729) takes advantage of this feature so it knows how it was called and which utility to run.

```
$ ln -s abc2 mylink
$ /home/sam/mylink
This script was called by typing mylink
```

When you display the value of **$0** from an interactive shell, the shell displays its name because that is the name of the calling program (the program you are running).

```
$ echo $0
bash
```