8.2. Catching user input

8.2.1. Using the read built-in command

The **read** built-in command is the counterpart of the **echo** and **printf** commands. The syntax of the **read** command is as follows:

read `[options] NAME1 NAME2 ... NAMEN`

One line is read from the standard input, or from the file descriptor supplied as an argument to the `-u` option. The first word of the line is assigned to the first name, `NAME1`, the second word to the second name, and so on, with leftover words and their intervening separators assigned to the last name, `NAMEN`. If there are fewer words read from the input stream than there are names, the remaining names are assigned empty values.

The characters in the value of the `IFS` variable are used to split the input line into words or tokens; see Section 3.4.8. The backslash character may be used to remove any special meaning for the next character read and for line continuation.

If no names are supplied, the line read is assigned to the variable `REPLY`.

The return code of the **read** command is zero, unless an end-of-file character is encountered, if **read** times out or if an invalid file descriptor is supplied as the argument to the `-u` option.

The following options are supported by the Bash **read** built-in:

Table 8-2. Options to the read built-in

Option	Meaning
-a `ANAME`	The words are assigned to sequential indexes of the array variable `ANAME`, starting at 0. All elements are removed from `ANAME` before the assignment. Other `NAME` arguments are ignored.
-d `DELIM`	The first character of `DELIM` is used to terminate the input line, rather than newline.
-e	**readline** is used to obtain the line.
-n `NCHARS`	**read** returns after reading `NCHARS` characters rather than waiting for a complete line of input.
-p `PROMPT`	Display `PROMPT`, without a trailing newline, before attempting to read any input. The prompt is displayed only if input is coming from a terminal.
-r	If this option is given, backslash does not act as an escape character. The backslash is considered to be part of the line. In particular, a backslash-newline pair may not be used as a line continuation.
-s	Silent mode. If input is coming from a terminal, characters are not echoed.
-t `TIMEOUT`	Cause **read** to time out and return failure if a complete line of input is not read within `TIMEOUT` seconds. This option has no effect if **read** is not reading input from the terminal or from a pipe.
-u `FD`	Read input from file descriptor `FD`.

This is a straightforward example, improving on the `leaptest.sh` script from the previous chapter:

```
michel ~/test> cat leaptest.sh
#!/bin/bash
# This script will test if you have given a leap year or not.

echo "Type the year that you want to check (4 digits), followed by [ENTER]:"

read year

if (( ("$year" % 400) == "0" )) || (( ("$year" % 4 == "0") && ("$year" % 100 !=
"0") )); then
  echo "$year is a leap year."
else
  echo "This is not a leap year."
fi

michel ~/test> leaptest.sh
Type the year that you want to check (4 digits), followed by [ENTER]:
2000
2000 is a leap year.
```

8.2.2. Prompting for user input

The following example shows how you can use prompts to explain what the user should enter.

```
michel ~/test> cat friends.sh
#!/bin/bash

# This is a program that keeps your address book up to date.

friends="/var/tmp/michel/friends"

echo "Hello, "$USER".  This script will register you in Michel's friends database."

echo -n "Enter your name and press [ENTER]: "
read name
echo -n "Enter your gender and press [ENTER]: "
read -n 1 gender
echo

grep -i "$name" "$friends"

if  [ $? == 0 ]; then
  echo "You are already registered, quitting."
  exit 1
elif [ "$gender" == "m" ]; then
  echo "You are added to Michel's friends list."
  exit 1
else
  echo -n "How old are you? "
  read age
  if [ $age -lt 25 ]; then
    echo -n "Which colour of hair do you have? "
    read colour
    echo "$name $age $colour" >> "$friends"
    echo "You are added to Michel's friends list.  Thank you so much!"
  else
    echo "You are added to Michel's friends list."
    exit 1
  fi
fi

michel ~/test> cp friends.sh /var/tmp; cd /var/tmp
michel ~/test> touch friends; chmod a+w friends

michel ~/test> friends.sh
Hello, michel.  This script will register you in Michel's friends database.
Enter your name and press [ENTER]: michel
Enter your gender and press [ENTER] :m
You are added to Michel's friends list.

michel ~/test> cat friends
```

Note that no output is omitted here. The script only stores information about the people Michel is interested in, but it will always say you are added to the list, unless you are already in it.

Other people can now start executing the script:

```
[anny@octarine tmp]$ friends.sh
Hello, anny.  This script will register you in Michel's friends database.
Enter your name and press [ENTER]: anny
Enter your gender and press [ENTER] :f
How old are you? 22
Which colour of hair do you have? black
You are added to Michel's friends list.
```

After a while, the `friends` list begins to look like this:

```
tille 24 black
anny 22 black
katya 22 blonde
maria 21 black
--output omitted--
```

Of course, this situation is not ideal, since everybody can edit (but not delete) Michel's files. You can solve this problem using special access modes on the script file, see SUID and SGID in the Introduction to Linux guide.

8.2.3. Redirection and file descriptors

8.2.3.1. General

As you know from basic shell usage, input and output of a command may be redirected before it is executed, using a special notation - the redirection operators - interpreted by the shell. Redirection may also be used to open and close files for the current shell execution environment.

Redirection can also occur in a script, so that it can receive input from a file, for instance, or send output to a file. Later, the user can review the output file, or it may be used by another script as input.

File input and output are accomplished by integer handles that track all open files for a given process. These numeric values are known as file descriptors. The best known file descriptors are *stdin*, *stdout* and *stderr*, with file descriptor numbers 0, 1 and 2, respectively. These numbers and respective devices are reserved. Bash can take TCP or UDP ports on networked hosts as file descriptors as well.

The output below shows how the reserved file descriptors point to actual devices:

```
michel ~> ls -l /dev/std*
lrwxrwxrwx  1 root    root     17 Oct  2 07:46 /dev/stderr -> ../proc/self/fd/2
lrwxrwxrwx  1 root    root     17 Oct  2 07:46 /dev/stdin -> ../proc/self/fd/0
lrwxrwxrwx  1 root    root     17 Oct  2 07:46 /dev/stdout -> ../proc/self/fd/1
```

```
michel ~> ls -l /proc/self/fd/[0-2]
lrwx------  1 michel  michel   64 Jan 23 12:11 /proc/self/fd/0 -> /dev/pts/6
lrwx------  1 michel  michel   64 Jan 23 12:11 /proc/self/fd/1 -> /dev/pts/6
lrwx------  1 michel  michel   64 Jan 23 12:11 /proc/self/fd/2 -> /dev/pts/6
```

Note that each process has its own view of the files under `/proc/self`, as it is actually a symbolic link to `/proc/<process_ID>`.

You might want to check **info MAKEDEV** and **info proc** for more information about `/proc` subdirectories and the way your system handles standard file descriptors for each running process.

When excuting a given command, the following steps are excuted, in order:

- If the standard output of a previous command is being piped to the standard input of the current command, then `/proc/<current_process_ID>/fd/0` is updated to target the same anonymous pipe as `/proc/<previous_process_ID/fd/1`.
- If the standard output of the current command is being piped to the standard input of the next command, then `/proc/<current_process_ID>/fd/1` is updated to target another anonymous pipe.
- Redirection for the current command is processed from left to right.
- Redirection "N>&M" or "N<&M" after a command has the effect of creating or updating the symbolic link `/proc/self/fd/N` with the same target as the symbolic link `/proc/self/fd/M`.
- The redirections "N> file" and "N< file" have the effect of creating or updating the symbolic link `/proc/self/fd/N` with the target file.
- File descriptor closure "N>&-" has the effect of deleting the symbolic link `/proc/self/fd/N`.
- Only now is the current command executed.

When you run a script from the command line, nothing much changes because the child shell process will use the same file descriptors as the parent. When no such parent is available, for instance when you run a script using the *cron* facility, the standard file descriptors are pipes or other (temporary) files, unless some form of redirection is used. This is demonstrated in the example below, which shows output from a simple **at** script:

```
michel ~> date
Fri Jan 24 11:05:50 CET 2003

michel ~> at 1107
warning: commands will be executed using (in order)
a) $SHELL b) login shell c)/bin/sh
at> ls -l /proc/self/fd/ > /var/tmp/fdtest.at
at> <EOT>
job 10 at 2003-01-24 11:07

michel ~> cat /var/tmp/fdtest.at
total 0
lr-x------    1 michel michel  64 Jan 24 11:07 0 -> /var/spool/at/!0000c010959eb (deleted)
l-wx------    1 michel michel  64 Jan 24 11:07 1 -> /var/tmp/fdtest.at
l-wx------    1 michel michel  64 Jan 24 11:07 2 -> /var/spool/at/spool/a0000c010959eb
lr-x------    1 michel michel  64 Jan 24 11:07 3 -> /proc/21949/fd
```

And one with **cron**:

```
michel ~> crontab -l
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/tmp/crontab.21968 installed on Fri Jan 24 11:30:41 2003)
# (Cron version -- $Id: chap8.xml,v 1.9 2006/09/28 09:42:45 tille Exp $)
32 11 * * * ls -l /proc/self/fd/ > /var/tmp/fdtest.cron

michel ~> cat /var/tmp/fdtest.cron
total 0
lr-x------    1 michel michel  64 Jan 24 11:32 0 -> pipe:[124440]
l-wx------    1 michel michel  64 Jan 24 11:32 1 -> /var/tmp/fdtest.cron
l-wx------    1 michel michel  64 Jan 24 11:32 2 -> pipe:[124441]
lr-x------    1 michel michel  64 Jan 24 11:32 3 -> /proc/21974/fd
```

8.2.3.2. Redirection of errors

From the previous examples, it is clear that you can provide input and output files for a script (see Section 8.2.4 for more), but some tend to forget about redirecting errors - output which might be depended upon later on. Also, if you are lucky, errors will be mailed to you and eventual causes of failure might get revealed. If you are not as lucky, errors will cause your script to fail and won't be caught or sent anywhere, so that you can't start to do any worthwhile debugging.

When redirecting errors, note that the order of precedence is significant. For example, this command, issued in `/var/spool`

```
ls -l * 2> /var/tmp/unaccessible-in-spool
```

will redirect standard output of the **ls** command to the file `unaccessible-in-spool` in `/var/tmp`. The command

```
ls -l * > /var/tmp/spoollist 2>&1
```

will direct both standard input and standard error to the file `spoollist`. The command

```
ls -l * 2 >& 1 > /var/tmp/spoollist
```

directs only the standard output to the destination file, because the standard error is copied to standard output before the standard output is redirected.

For convenience, errors are often redirected to `/dev/null`, if it is sure they will not be needed. Hundreds of examples can be found in the startup scripts for your system.

Bash allows for both standard output and standard error to be redirected to the file whose name is the result of the expansion of `FILE` with this construct:

&> `FILE`

This is the equivalent of **>** `FILE` **2>&1**, the construct used in the previous set of examples. It is also often combined with redirection to `/dev/null`, for instance when you just want a command to execute, no matter what output or errors it gives.

8.2.4. File input and output

8.2.4.1. Using /dev/fd

The `/dev/fd` directory contains entries named `0`, `1`, `2`, and so on. Opening the file `/dev/fd/N` is equivalent to duplicating file descriptor *N*. If your system provides `/dev/stdin`, `/dev/stdout` and `/dev/stderr`, you will see that these are equivalent to `/dev/fd/0`, `/dev/fd/1` and `/dev/fd/2`, respectively.

The main use of the `/dev/fd` files is from the shell. This mechanism allows for programs that use pathname arguments to handle standard input and standard output in the same way as other pathnames. If `/dev/fd` is not available on a system, you'll have to find a way to bypass the problem. This can be done for instance using a hyphen (-) to indicate that a program should read from a pipe. An example:

```
michel ~> filter body.txt.gz | cat header.txt - footer.txt
This text is printed at the beginning of each print job and thanks the sysadmin
for setting us up such a great printing infrastructure.

Text to be filtered.

This text is printed at the end of each print job.
```

The **cat** command first reads the file `header.txt`, next its standard input which is the output of the **filter** command, and last the `footer.txt` file. The special meaning of the hyphen as a command-line argument to refer to the standard input or standard output is a misconception that has crept into many programs. There might also be problems when specifying hyphen as the first argument, since it might be interpreted as an option to the preceding command. Using `/dev/fd` allows for uniformity and prevents confusion:

```
michel ~> filter body.txt | cat header.txt /dev/fd/0 footer.txt | lp
```

In this clean example, all output is additionally piped through **lp** to send it to the default printer.

8.2.4.2. Read and exec

8.2.4.2.1. Assigning file descriptors to files

Another way of looking at file descriptors is thinking of them as a way to assign a numeric value to a file. Instead of using the file name, you can use the file descriptor number. The **exec** built-in command can be used to replace the shell of the current process or to alter the file descriptors of the current shell. For example, it can be used to assign a file descriptor to a file. Use

exec fdN> `file`

for assigning file descriptor N to `file` for output, and

exec fdN< `file`

for assigning file descriptor N to `file` for input. After a file descriptor has been assigned to a file, it can be used with the shell redirection operators, as is demonstrated in the following example:

```
michel ~> exec 4> result.txt

michel ~> filter body.txt | cat header.txt /dev/fd/0 footer.txt >& 4

michel ~> cat result.txt
This text is printed at the beginning of each print job and thanks the sysadmin
for setting us up such a great printing infrastructure.

Text to be filtered.

This text is printed at the end of each print job.
```

File descriptor 5

Using this file descriptor might cause problems, see the Advanced Bash-Scripting Guide, chapter 16. You are strongly advised not to use it.

8.2.4.2.2. Read in scripts

The following is an example that shows how you can alternate between file input and command line input:

```
michel ~/testdir> cat sysnotes.sh
#!/bin/bash

# This script makes an index of important config files, puts them together in
# a backup file and allows for adding comment for each file.

CONFIG=/var/tmp/sysconfig.out
rm "$CONFIG" 2>/dev/null

echo "Output will be saved in $CONFIG."

# create fd 7 with same target as fd 0 (save stdin "value")
exec 7<&0

# update fd 0 to target file /etc/passwd
exec < /etc/passwd

# Read the first line of /etc/passwd
read rootpasswd

echo "Saving root account info..."
echo "Your root account info:" >> "$CONFIG"
echo $rootpasswd >> "$CONFIG"

# update fd 0 to target fd 7 target (old fd 0 target); delete fd 7
exec 0<&7 7<&-

echo -n "Enter comment or [ENTER] for no comment: "
read comment; echo $comment >> "$CONFIG"

echo "Saving hosts information..."

# first prepare a hosts file not containing any comments
TEMP="/var/tmp/hosts.tmp"
cat /etc/hosts | grep -v "^#" > "$TEMP"

exec 7<&0
exec < "$TEMP"

read ip1 name1 alias1
read ip2 name2 alias2

echo "Your local host configuration:" >> "$CONFIG"

echo "$ip1 $name1 $alias1" >> "$CONFIG"
echo "$ip2 $name2 $alias2" >> "$CONFIG"

exec 0<&7 7<&-

echo -n "Enter comment or [ENTER] for no comment: "
read comment; echo $comment >> "$CONFIG"
rm "$TEMP"

michel ~/testdir> sysnotes.sh
Output will be saved in /var/tmp/sysconfig.out.
Saving root account info...
Enter comment or [ENTER] for no comment: hint for password: blue lagoon
Saving hosts information...
Enter comment or [ENTER] for no comment: in central DNS
```

```
michel ~/testdir> cat /var/tmp/sysconfig.out
Your root account info:
root:x:0:0:root:/root:/bin/bash
hint for password: blue lagoon
Your local host configuration:
127.0.0.1 localhost.localdomain localhost
192.168.42.1 tintagel.kingarthur.com tintagel
in central DNS
```

8.2.4.3. Closing file descriptors

Since child processes inherit open file descriptors, it is good practice to close a file descriptor when it is no longer needed. This is done using the

exec fd<&-

syntax. In the above example, file descriptor 7, which has been assigned to standard input, is closed each time the user needs to have access to the actual standard input device, usually the keyboard.

The following is a simple example redirecting only standard error to a pipe:

```
michel ~> cat listdirs.sh
#!/bin/bash

# This script prints standard output unchanged, while standard error is
# redirected for processing by awk.

INPUTDIR="$1"

# fd 6 targets fd 1 target (console out) in current shell
exec 6>&1

# fd 1 targets pipe, fd 2 targets fd 1 target (pipe),
# fd 1 targets fd 6 target (console out), fd 6 closed, execute ls
ls "$INPUTDIR"/* 2>&1 >&6 6>&- \
                                # Closes fd 6 for awk, but not for ls.

| awk 'BEGIN { FS=":" } { print "YOU HAVE NO ACCESS TO" $2 }' 6>&-

# fd 6 closed for current shell
exec 6>&-
```

8.2.4.4. *Here* documents

Frequently, your script might call on another program or script that requires input. The *here* document provides a way of instructing the shell to read input from the current source until a line containing only the search string is found (no trailing blanks). All of the lines read up to that point are then used as the standard input for a command.

The result is that you don't need to call on separate files; you can use shell-special characters, and it looks nicer than a bunch of **echo**'s:

```
michel ~> cat startsurf.sh
#!/bin/bash

# This script provides an easy way for users to choose between browsers.

echo "These are the web browsers on this system:"
```

```
# Start here document
cat << BROWSERS
mozilla
links
lynx
konqueror
opera
netscape
BROWSERS
# End here document

echo -n "Which is your favorite? "
read browser

echo "Starting $browser, please wait..."
$browser &
```

```
michel ~> startsurf.sh
These are the web browsers on this system:
mozilla
links
lynx
konqueror
opera
netscape
Which is your favorite? opera
Starting opera, please wait...
```

Although we talk about a *here document*, it is supposed to be a construct within the same script. This is an example that installs a package automatically, eventhough you should normally confirm:

```
#!/bin/bash

# This script installs packages automatically, using yum.

if [ $# -lt 1 ]; then
        echo "Usage: $0 package."
        exit 1
fi

yum install $1 << CONFIRM
y
CONFIRM
```

And this is how the script runs. When prompted with the "Is this ok [y/N]" string, the script answers "y" automatically:

```
[root@picon bin]# ./install.sh tuxracer
Gathering header information file(s) from server(s)
Server: Fedora Linux 2 - i386 - core
Server: Fedora Linux 2 - i386 - freshrpms
Server: JPackage 1.5 for Fedora Core 2
Server: JPackage 1.5, generic
Server: Fedora Linux 2 - i386 - updates
Finding updated packages
Downloading needed headers
Resolving dependencies
Dependencies resolved
I will do the following:
[install: tuxracer 0.61-26.i386]
Is this ok [y/N]: EnterDownloading Packages
Running test transaction:
Test transaction complete, Success!
```

```
tuxracer 100 % done 1/1
Installed:  tuxracer 0.61-26.i386
Transaction(s) Complete
```

Prev Home Next
Displaying user messages Up Summary