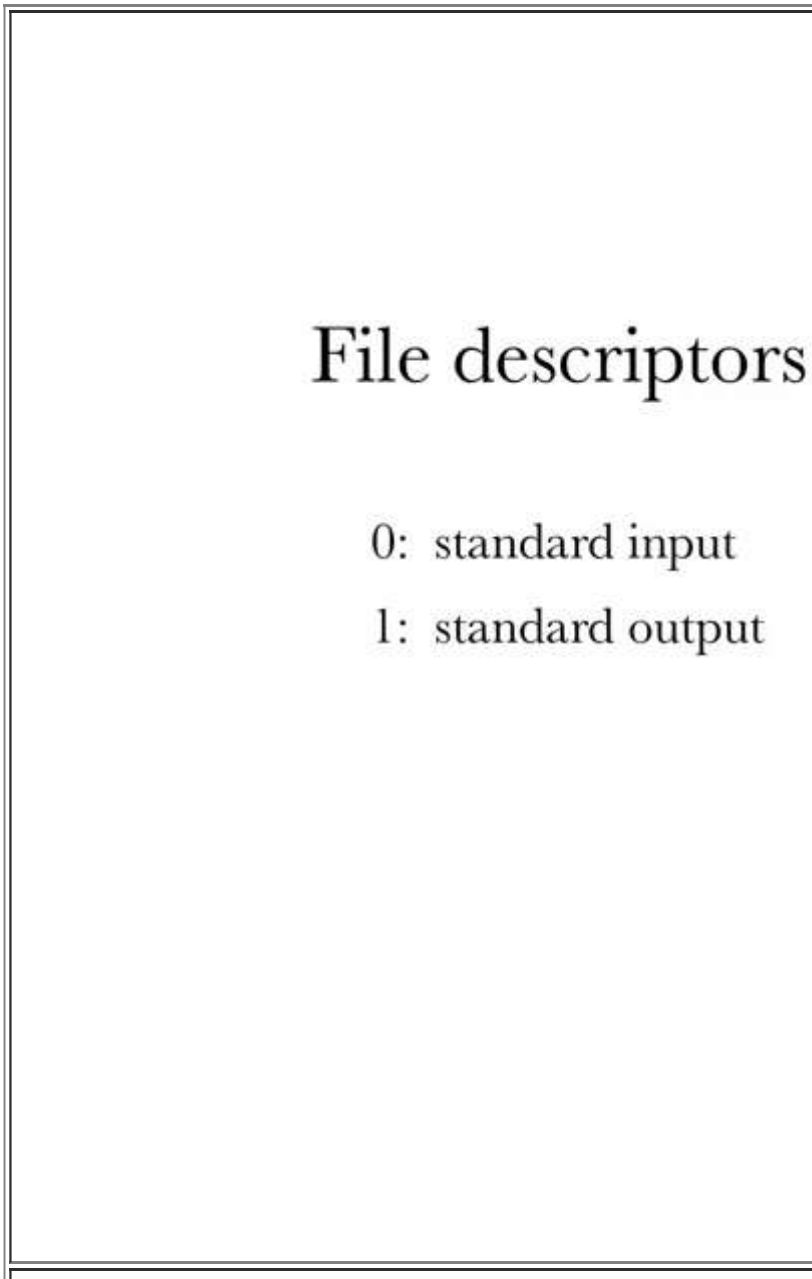
Shell programming 4 of 6

Hover over the image to see the time range in the original video. Click to play that excerpt.



A running program in unix has a certain set of open files at any one time. Open files are identified by small integers called "file descriptors".

Normally when a program is started it has three files already open. Most programs read from file descriptor 0, which we therefore call the "standard input", and write to file descriptor 1, which we therefore call the "standard output".

Your terminal in unix is also a file, and by default these two open files are both the terminal. So when a program reads input from the standard input it reads from the terminal, and when it outputs on the standard output it writes to the terminal.

But as you know, the input and output can be "redirected".

If we run a command like this...
then the shell opens the file "file1" for input on file descriptor zero, and opens the file "file2" for output on file descriptor one, all *before* it actually executes the program named "foo".
When foo reads from file descriptor zero, as most

I/O redirection

```
foo <file1 >file2

bar >>file2
```

programs do, it will then be reading from the file "file1" instead of from the terminal. And when foo writes to file descriptor one, it will be writing to the file "file2" instead of to the terminal.
The operators we use for i/o redirection are less-than and greater-than signs, but you'll do better to think of them as arrows, and we normally pronounce them "direct in" and "direct out" instead of "less than" and "greater than".

If we double the greater-than sign, this opens the file for append instead of overwrite. So this command runs "bar" with its output redirected to file2, but appending to what's already there in file2.

```
wc -l <<EOF
"It's impossible to f
 of being clever."
                -- Ch

"If there are several
 thing, choose one."
                -- RF
EOF
```

```
wc -l <<\EOF
"It's impossible to f
 of being clever."
                -- Ch

"If there are several
 thing, choose one."
                -- RF
EOF
```

There is also a double less-than sign, which says to take input right there from your shell script text. For example, we can run this as a shell command...
and this will output seven -- seven lines as counted by the wc −l command. The double-less-than sign is followed by an arbitrary token, which when it appears *later*, alone on a line, terminates the input. This is called "here-is" text or "here" text.
With *this* syntax, dollar signs and backquotes inside the "here-is text" are *interpreted*, just like inside double quotes.

To get a behaviour more like single-quotes, in which no special characters inside the here-is text are interpreted, strangely the syntax is to quote the *original* mention of the end-of-file word, like this:
In the example on the right, any dollar signs or backslashes or backquotes inside the here-is text are taken literally and don't have their special meaning.

The *bash* shell also has triple-character redirection operators, but those should not be used because the resulting program will then not work under other versions of *sh*.

File descriptors

0: standard input

1: standard output

2: standard error

```
cat file | grep blof
```

I said that programs in unix are normally started with *three* open files. The third file is open on file descriptor 2, but like the file on file descriptor 1 it's open for output, and it is also connected to the terminal by default.
So why have two standard file descriptors for output?
The third file is called the "standard error", and it's used for error messages. Consider a command like this:
If the file exists but does not contain the string "blof", there will be no output. If the file does not exist or is unreadable, cat will give an error message.
We don't want this error message to go down the

	pipe, where 'grep' will read it and discard it because it doesn't contain the string "blof". We need to see this error message on the terminal. So we want this error message to bypass the pipe. So, we ask all unix programmers to write their error messages to file descriptor 2, the standard error channel, instead of writing them to file descriptor 1 mixed in with the normal output; and when you redirect the standard output, whether you do it like this with a pipe or you redirect into a file with a greater-than sign, messages to the standard error will still be seen on the screen, rather than going into the pipe or into the file on disk.

`cat >file`
 	
`cat 2>file`
 	
`cmd >file1 2>file2`
 	
`foo >&2`
 	
`echo usage: diff file1 file2 >&2`
	We can manage file descriptors fully in the shell. A command like "`cat >file`" redirects file descriptor 1 into file. If we want instead to redirect file descriptor 2, we can do "`cat 2>file`". There must be no space between the 2 and the greater-than sign; so it's still possible to do "`cat 2 >file`", which cats a file named 2 and redirects the standard output to file. Whereas "`cat 2>file`" without a space runs cat with no arguments but with file descriptor 2 redirected into the file, and file descriptor 1 left as is.
	
	And we can do them both for the same command.

We can also redirect file descriptors to each other. As we've seen here, to specify which file descriptor to redirect, we put the number on the left of the symbol -- the greater-than sign redirects file descriptor 1 by default. To redirect *to* a file descriptor, we put the number on the right of the symbol, prefaced by an ampersand, like this. ">&2" means to redirect the standard output to the standard error. So for example we can output error messages to the standard error with commands like this:

```
$ sh s1 foo bar
argument 1 is foo
argument 2 is bar
$ cat s1
echo argument 1 is $1
echo argument 2 is $2
$
```

A lot of software tools process command-line arguments. For example, when you say "cat file", that file name is an argument to cat. The cat program has to be able to get that string to be able to open the appropriate file.

We can process command-line arguments in our shell scripts, too. If we run "sh s1 foo bar", then the "s1" shell script gets those two command-line arguments "foo" and "bar".
How does it get them?

These variable interpolations are like other special variable names in that the dollar sign is followed by just a single character for the special variable name. So we can only go up to argument number 9 this

way, at least in some versions of sh. But if we're going that high we'd want to be doing this in a loop, anyway. We'll get to that shortly.

```
$ sh add 2 3
5
$ sh add 2
usage: add x y
$ cat add
if test $# -ne 2
then
    echo usage: add x y >&2
    exit 1
fi
expr $1 + $2
$
```

You are going to want to check that the user supplied enough arguments before you start accessing them. Let's write a shell script like this:

That error message went to the standard error, of course!

The special variable $# gives you the number of command-line arguments. We use 'test' and an 'if' to require it to be two, else we output an error message to the standard error and we exit.
The "exit" command terminates the shell program at this point, and you can specify an exit status for this program, which here is 1 because it did not succeed.
But if the 'if' statement doesn't object to the argument count, then we can go ahead and add them.
You might be inclined to use an 'echo' command here, in the last line here, because you're thinking of output, but remember that software tools generally produce their results on the standard output -- expr is already outputting to the standard output. You don't need to do anything further than just running

expr to output the sum on the standard output.

```
~
~
~
~
~
~
"add" 6L, 76C written
$ cat add
if test $# -ne 2
then
    echo usage: $0 x y >&2
    exit 1
fi
expr $1 + $2
$ sh add 3 4 5 6
usage: add x y
$ mv add glop
$ sh glop 3 4 5 6
usage: glop x y
$
```

In C you can access the name of the program itself by using argument zero, for example to use in this error message; and you can do that in sh too, with $0.

```
$ cat s3
echo now '$*' is $*
$ sh s3 3 4 5
now $* is 3 4 5
$ sh s3 6
now $* is 6
$ sh s3
now $* is
$
```

Ok. These are obviously toy shell programs. How do we access *all* arguments?

Well, for one, there's a special variable $* which expands to all arguments.

Using this can be tricky sometimes, though.

Ok, that's three files, one of which has space in the file name:
... Obviously, we need single quotes to be able to access that file...

```
$ echo contents of file1 >file1
$ echo contents of file2 >file2
$ echo contents of file three >'file three'
$ cat file three
cat: file: No such file or directory
cat: three: No such file or directory
$ cat 'file three'
contents of file three
$ cat show1
cat $*
$ sh show1 file1 file2
contents of file1
contents of file2
$ sh show1 file three
cat: file: No such file or directory
cat: three: No such file or directory
$ sh show1 'file three'
cat: file: No such file or directory
cat: three: No such file or directory
$
```

Ok, with that setup, let's write a simple shell script to output files named in its arguments.
Well, it's just going to be a cat.
... Obviously we need to quote that space.

```
$ sh show1 'file three'
cat: file: No such file or directory
cat: three: No such file or directory
$ cat show2
cat '$*'
$ sh show2 'file three'
cat: $*: No such file or directory
$ cat show3
cat "$*"
$ sh show3 'file three'
contents of file three
$ sh show3 file1 file2
cat: file1 file2: No such file or directory
$ sh show1 file1 'file three'
contents of file1
cat: file: No such file or directory
cat: three: No such file or directory
$ sh show3 file1 'file three'
cat: file1 file three: No such file or directory
$
```

It doesn't suffice to quote the space on the command-line, because when the argument is interpolated with $*, there's no quoting and the space still separates the words.
Obviously, we don't want single quotes.
Double quotes allow the interpretation of dollar sign while suppressing the interpretation of space:
So in this last case, we need to avoid the quotes, but in the file three case, we need the quotes.
And we're not really heading towards a solution here, because we can always combine the situations:

What we need is to be in double quotes, but somehow magically to cease the double quotes inbetween arguments, while keeping the double quotes around a single argument.
So there's a special

```
$ sh show3 file1 file2
cat: file1 file2: No such file or directory
$ sh show1 file1 'file three'
contents of file1
cat: file: No such file or directory
cat: three: No such file or directory
$ sh show3 file1 'file three'
cat: file1 file three: No such file or directory
$ cat show4
cat "$@"
$ sh show4 file1
contents of file1
$ sh show4 'file three'
contents of file three
$ sh show4 file1 'file three'
contents of file1
contents of file three
$ cat show4
cat "$@"
$
```

syntax for just this situation. You could call it a kludge. But without it, there's no way to solve this problem.
The special syntax is "$@".
Outside of double quotes, "$@" is exactly the same as $*.
Inside double quotes, the double quotes are considered to cease to operate inbetween arguments, while still operating within a given argument.
So normally we use $@, inside double quotes, when we want to pass all command-line arguments to another program.

```
cat "$@" | ...
```

Our inclination as programmers in processing all command-line arguments is probably to use a loop. But that's not always needed in shell programs.
In general, in writing shell programs you should think of what processing can be done by the software tools you use as primitives in your program, so that you don't have to do it yourself.
Many shell scripts take the form of cat "$@" piped to something -- this will go through zero or more filename arguments, with no loops, and it even gets the zero-file-name case right, where it's supposed to read the standard input -- because cat does all that.

If you want to loop

```
for i
do
    ... something exciting here ...
done
```

through all command-line arguments and do something more complicated, remember the 'for' syntax where you just say "for variable name", no "in" keyword, and it loops through all command-line arguments.

```
while test $# -gt 0
do
    ... various stuff, including a "shift" at some point ...
done
```

shift:

$3 → $2 → $1 → discarded

For more complicated processing of command-line arguments, sometimes we want to write a loop of this form...
where we "consume" one or even sometimes more of the command-line arguments in our loop. The *shift* command moves all of the command-line arguments down one place. If there were three command-line arguments, $3 becomes $2; $2 becomes $1; $1 is discarded.

Here's another kind of use of shift: Suppose we want to turn something of the form, running a shell script with a special category name and then zero or more files, into this, where the category name forms part of the command, and the zero or more filenames come at the end. An arbitrary category name, and then

```
sh q category file1 file2
      ⟶ /u/sally/category/process file1 file2
```

```
category="$1"
shift
/u/sally/"$category"/process "$@"
```

processing zero or more filename arguments in the usual way.
We could write this: because the "$@" in the last line will then be the remaining command-line arguments after the shift has removed argument one, the category name.

${x}

```
sed -n 3p file
```

~~sed -n $ip file~~

~~sed -n $i p file~~

```
sed -n ${i}p file
```

There are a few "special" variable interpolation syntaxes in *sh*. I'd like to mention just one.
Suppose we want to run a command like "sed −n 3p file -- output line 3 of the file named "file". This command is fine, but suppose that that number 3 is a variable?
"sed −n $ip file" is not going to work, because this is a reference to a variable named "ip".
And we can't put a space there, because then ***sed*** won't work; the 3 and the p have to be the same word, no space, because that's the syntax of sed.
Instead, we can use the syntax which looks like this: ${x}
for a variable named "x", with the variable name in braces. This is syntactically unambiguous because the left brace immediately follows the dollar sign.

And... it means the same thing as $x!
But if we use that syntax with our sed command, then we get the variable i's value interpolated right there, it goes right next to the 'p' with no space; and this is the solution for this problem.

```
$ cat s6
echo one
# echo two three
echo four five # six seven
echo eight
$ sh s6
one
four five
eight
$
```

At around this point we start wanting to be able to write "comments" in our shell programs.
The comment character in *sh* is the number sign, which introduces a comment to the end of the line.

This ends the core part of this shell programming video series. But there are two more videos.

Video five discusses some details of shell programming which you need to get right to make your shell program behave as a normal program in every way. It's extra-reading material which is not covered in this course.

Video six discusses some additional aspects of shell programming having to do with handling processes,

	which we'll discuss later in the course. So we'll get back to video six substantially later.