Bash

by Daniel & 4.6

August 22, 2026

> *We should have some ways of coupling programs like garden hose—screw in another segment when it becomes necessary to massage data in another way.* —Doug McIlroy, 1964

My favorite programming language is Bash. It is certainly not the best tool for every job, but it is a perfectly fine tool for many jobs, and it is the tool I reach for first, and the tool I return to when the others have made a mess, and the tool I have used almost every day for twenty years without ever once wishing it were something else.

Most programmers do not understand how to use Bash to write programs. They use it every day to run

simple commands in the terminal, but have no idea how to use it to solve even the simplest of programming tasks. They think of it as the thing you type commands into, not as a language you write programs in. This is understandable, because nobody taught them otherwise, and because the syntax is strange, and because the documentation is a reference manual written for people who already know.

This is my attempt at the tutorial that should have existed thirty years ago. It covers the things you need to know in order to write real programs in Bash, starting from the assumption that you already know how to program in at least one other language but that you have never thought of Bash as a language at all.

The most classic example of what makes Bash special is the pipeline: running two programs in parallel and plugging the output of one into the input of the other. Almost everyone understands this concept. Most people have used it many times in the past and perhaps are even using it on a daily basis.

I am going to guess you probably agree that running two programs and piping them together is an extremely useful and quite basic idea. Yet I challenge you to set up even the simplest, most basic two-part pipeline in your favorite programming language. Go on, I'll wait.

Are you back? Did you figure out how to even start two programs? If you managed to start them, were you able to plug them together?

Have you ever thought about how strange it is that most professional programmers have no idea whatsoever how to go about running programs? Programs are for using and for building other programs on top of. The only programming language that expects you to do this is Bash.

Other languages are like jealous husbands, always expecting you to write everything in them and never to use any program or library written in another language—to the point where it would not even occur to most programmers that while using language X it might be possible to utilize an existing tool that was written in language Y.

We as programmers have ended up in a situation very similar to the biblical story about the Tower of Babel. Each language community is off in its own little corner working on reimplementing every program or library that has ever been invented but in their favorite language. Very little thought or credence is given to the possibility of synergy between programs written in different languages. It is as if each and every programming language community is its own little North Korea.

But there is no reason for this. There is no geographical distance between programs in a computer and therefore no need for logistics.

If this piqued your interest you might be thinking: okay, I can run other programs and I can pipe them together, but what about the other programming tasks I need to do? What about loops and if statements, what about functions and data structures—what about abstraction?

Well, you have come to the right place. I am writing this to try to show you that while Bash has its limitations, at the end of the day it is definitely a real programming language and a quite nice one at that. My hope is to convince you that it is the right tool for many jobs, and that it can be quite delightful once you understand how to use it.

※

Bash essentially has two data types: strings and lists of strings. Because of this, it is very important to understand how quoting works.

In most programming languages, the presence or absence of quotation marks serves to indicate whether something is or is not a string. This distinction between

strings and other values such as numbers is completely meaningless in Bash since everything is always a string.

So the first thing to understand about Bash is that quotation marks have a completely different meaning compared to most other languages. At the same time, it is fair to say that quoting and quotation marks is one of the most important concepts and syntactical features of Bash.

What are quotation marks used for in Bash? The answer is very simple: they help you escape special characters—most importantly whitespace. Quotation marks serve the same purpose as the backslash commonly does. In fact, you could avoid using them at all and just use the backslash.

Let's look at a few examples. These are all completely equivalent:

```
hello world 123
"hello" "world" "123"
"hello" world 123
"hell"o wor"ld" "1"23
"h""e""l""l""o" """"wo""rld"""" 1""23""
```

It makes no difference at all whether or not you quote these simple alphanumerics because there is no need to escape them to begin with. It is worth emphasizing that `123` and `"123"` are totally equivalent.

Another interesting property of Bash syntax is also demonstrated here: it is perfectly fine and changes nothing to quote only part of a word. Again, the expressions `world` and `wor"ld"` are totally equivalent.

However, these are all different from the above and from each other:

```
"hello world 123"
"hello world" 123
""hello "world 1"23""
```

Can you guess why? What is different about these three expressions? Here are three more which are equivalent to the above, respectively:

```
"hello wor""ld 123"
hello" "world 123
""hello "wor""ld 1"23""
```

Of course these are also not equivalent to any of the first examples.

Did you figure it out? If not, I suggest you take a minute to think about it and we will consider this to be the first exercise of the book.

Done scratching your head? Okay, let me show you something that will make it a bit clearer what is going on.

There is a program called `printf` which can be used like this:

```
$ printf %s, a b c
a,b,c,

$ printf ++%s- 1 2 3 4 5
++1-++2-++3-++4-++5-

$ printf //%s x y z
//x//y//z

$ printf {%s} hello world
{hello}{world}
```

Get it? The first argument is a template that gets repeated once for each of the subsequent arguments, and `%s` says where to plug them in.

Let's try it with our earlier examples in order to shed some light on the difference between them all:

```
$ printf {%s} hello world 123
{hello}{world}{123}

$ printf {%s} "hello" "world" "123"
{hello}{world}{123}

$ printf {%s} "hello" world 123
{hello}{world}{123}
```

```
$ printf {%s} "hell"o wor"ld" "1"23
{hello}{world}{123}
```

See? They are all equivalent. The quotation marks make no difference.

But now look at the other examples we had:

```
$ printf {%s} "hello world 123"
{hello world 123}

$ printf {%s} "hello world" 123
{hello world}{123}

$ printf {%s} ""hello "world 1"23""
{hello}{world 123}
```

All different! And the alternative versions:

```
$ printf {%s} "hello wor""ld 123"
{hello world 123}

$ printf {%s} hello" "world 123
{hello world}{123}

$ printf {%s} ""hello "wor""ld 1"23""
{hello}{world 123}
```

What is happening here is that when we put the spaces inside quotation marks they stop being treated as word separators and instead the space characters become part of the so-called words.

Without quotes, `hello world` is two words. With quotes, `"hello world"` is one.

It may sound confusing to use the term word for something that is clearly more than one word, but this is the terminology normally used so I am trying to make you comfortable with talking about it like this.

You can also say that `printf` gets a different number of arguments depending on whether or not we put the spaces inside quotation marks.

Backslashes also achieve the same effect:

```
$ printf {{%s}} a b c
{{a}}{{b}}{{c}}

$ printf {{%s}} "a b" c
{{a b}}{{c}}

$ printf {{%s}} a b\ c
{{a}}{{b c}}

$ printf {{%s}} a\ b\ c
{{a b c}}

$ printf {{%s}} a\ b" "c
{{a b c}}
```

I will probably give even more examples of this later since it is so important to understand the quoting rules

and this concept of words, but for now I think you should be able to understand the gist of it.

※

Let's talk about variables. Like in most languages, a variable name is one or more alphanumerics plus underscores with no leading digits.

A variable name directly followed by an equals sign is an assignment:

```
$ my_variable=hello
```

You probably already know that a variable can be used like this:

```
$ echo $my_variable world
hello world
```

When assigning to a variable, the quoting rules we have already gone through apply in the same way to the word after the equals sign:

```
$ greeting="hello world"
$ other_greeting=hello\ world
$ echo $greeting, $other_greeting
hello world, hello world
```

You can set multiple variables on the same line:

```
$ my_variable=hello other_variable="hello world"
```

There is an optional distinction between empty and unbound variables. An empty variable is assigned the empty string:

```
$ my_empty_variable=
```

For the most part, you can treat unbound variables as empty variables, and this is often convenient as the distinction is seldom meaningful. In particular, using an unbound variable results in the empty string:

```
$ my_empty_variable=
$ echo .$my_empty_variable. :$nonexistent:
.. ::
```

Now, about whitespace. In most languages, whitespace is mostly insignificant. Assignments are usually written like this:

```
my_variable = "hello world"
```

What happens if we do this in Bash? A basic feature of Bash syntax is that the first word of each line determines the meaning of that line. In this case, there is nothing special about the first word, so it is simply taken to be the name of a program to run:

```
$ my_variable = "hello world"
-bash: my_variable: command not found
```

The equals sign has no special meaning by itself, so it just becomes the first argument to the command. We can see that this is the case by trying it with the `echo` command:

```
$ echo = "hello world"
= hello world
```

This is one of those things that trips up every programmer coming to Bash from another language. In Bash, `x=1` is an assignment. `x = 1` tries to run a program called `x` with the arguments `=` and `1`. The space changes everything. You will internalize this quickly, but the first few times you hit it you will stare at an error message for longer than you would like to admit.

※

There is a lot of nuance to how variable interpolation works in Bash. In fact, it might be the single most counterintuitive aspect of Bash to people who are already familiar with other programming languages.

The issue revolves around the distinction between strings and lists of strings—the only two data types recognized by Bash, remember?

So, what am I even talking about? Let's look at this example:

```
$ printf {%s} hello world
{hello}{world}

$ printf {%s} "hello world"
{hello world}
```

This should make a lot of sense to anybody. But now consider this:

```
$ my_variable="hello world"
$ printf {%s} $my_variable
```

What will the output of this command be? I hope you agree that the answer depends on how many arguments the `printf` command is getting.

But this is not obvious. On one hand, we are passing it the value of one variable. On the other hand, that value consists of two words.

Or, to make it sound really confusing on purpose, while the value of the variable is a single word, that word itself consists of two words.

You should be able to guess what the output will be just from the fact that I am making a big deal out of how confusing it is.

```
$ my_variable="hello world"
$ printf {%s} $my_variable
{hello}{world}
```

We are only passing one variable, but `printf` receives two arguments.

What happens is that Bash first expands the value of the variable and only then invokes `printf` with the resulting argument list.

This is called word splitting and understanding where and when this takes place is essential to being able to use Bash properly.

In most cases, you absolutely do not want word splitting to occur, especially when the value of the variable is not known in advance. For example, if the value comes from user input, the user is now able to pass arbitrary arguments to whatever program is being invoked.

Depending on what program we are talking about, it should be easy to see how quickly this could turn into a horrible disaster. It is very similar to the problem of SQL injection when dealing with user input from a web app—something many programmers will be familiar with.

Luckily, it is not quite as bad as directly splicing a user-supplied string into an SQL query, which enables the user to essentially do anything they want to the

database by simply finishing whatever the surrounding SQL syntax is, and then adding a semicolon followed by `DROP TABLE` or whatever. The equivalent would not work in Bash:

```
$ evil_user_input="; rm -rf /"
$ printf {%s} $evil_user_input
{;}{rm}{-rf}{/}
```

Phew.

But the issue is scary enough that I have to admit—I hope you will forgive me—I used another example to double check this on my laptop.

The fix is simple, and it is the single most important habit in all of Bash: put double quotes around your variables. Always. Every time. Until you have a specific reason not to, and you understand what that reason is.

```
$ my_variable="hello world"
$ printf {%s} "$my_variable"
{hello world}
```

When you write `"$my_variable"`, Bash expands the variable inside the double quotes but does not perform word splitting on the result. The value stays as one word, regardless of whether it contains spaces or tabs or anything else. This is almost always what you want.

The rule is: if you are using a variable and you want its value to arrive as a single argument to whatever command you are running, put double quotes around it. If you forget, Bash will silently split your value into pieces and hand those pieces to the command as separate arguments, and the command will do something you did not intend, and you will spend twenty minutes figuring out why.

There is exactly one common case where you want to leave the quotes off on purpose: when the variable contains a list of arguments that you actually want to be split into separate words. This is rare and you will know when you need it. Until then, quote everything.

I cannot overstate how many bugs in Bash scripts come down to a missing pair of double quotes around a variable. If you take away one thing from this entire tutorial, let it be this: `"$variable"`, not `$variable`. Always.

※

Every command in Bash returns a number when it finishes. The number is called the exit code. Zero means success. Anything else means failure.

This is the opposite of what most programming languages do. In most languages, zero is false and nonzero

is true. In Bash, zero is success and nonzero is failure. This seems backwards until you realize the logic: there is only one way to succeed, but many ways to fail, so you want the single success case to have the single value zero and the various failure cases to have different nonzero values that can tell you what went wrong.

You can see the exit code of the last command by looking at the special variable `$?`:

```
$ echo hello
hello
$ echo $?
0

$ ls /nonexistent
ls: /nonexistent: No such file or directory
$ echo $?
2
```

The `echo` command succeeded and returned zero. The `ls` command failed because the directory does not exist and returned two. Different programs use different nonzero values for different kinds of failure, but the zero-means-success convention is universal.

This is the foundation of conditionals in Bash. The `if` statement does not check whether something is true or false in any abstract sense. It runs a command and checks whether that command returned zero.

```
$ if ls /nonexistent; then
>   echo it exists
> else
>   echo it does not exist
> fi
ls: /nonexistent: No such file or directory
it does not exist
```

The `>` at the beginning of those continuation lines is just the shell telling you it is waiting for more input. The `fi` at the end closes the `if` block. Yes, it is `if` spelled backwards. No, I did not design this.

Now, you probably want to do things like compare strings or check whether a file exists, not just run programs and check whether they succeeded. For this, Bash has a construct called `[[`, the double bracket. It is built into Bash and it understands conditions:

```
$ [[ 1 = 1 ]]
$ echo $?
0

$ [[ 1 = 2 ]]
$ echo $?
1

$ [[ -f /etc/passwd ]]
$ echo $?
0
```

The `-f` flag asks whether a file exists. Since `/etc/passwd` exists on every Unix system, the condition is true and the exit code is zero.

Used with `if`, it looks like this:

```
$ name=max
$ if [[ $name = max ]]; then
>   echo hello
> fi
hello
```

Notice something important: the variable `$name` is not in double quotes. Inside `[[`, you do not need to quote your variables. Bash knows not to perform word splitting inside double brackets. This is one of the main reasons to prefer `[[` over the alternatives.

The spaces around and inside the brackets are not optional. Writing `[[$name` would not work. The spaces are syntax. This is the same lesson as the variable assignment: in Bash, spaces change everything.

The semicolon before `then` is there because `then` has to be the first word on its line, or it has to be separated from the previous command by a semicolon. You could also put `then` on the next line:

```
$ if [[ $name = max ]]
> then
```

```
>   echo hello
> fi
```

Both styles work. Use whichever you find easier to read.

You can do pattern matching inside `[[`:

```
$ name="Max Weber"
$ if [[ $name = Max* ]]; then
>   echo hello Max
> fi
hello Max
```

The `*` matches anything, so `Max*` matches any string that starts with Max.

For numeric comparisons, `[[` uses abbreviations: `-lt` for less than, `-gt` for greater than, `-eq` for equals, `-le` for less than or equal, `-ge` for greater than or equal, and `-ne` for not equal:

```
$ if [[ 5 -gt 3 ]]; then
>   echo five is greater than three
> fi
five is greater than three
```

You can combine conditions with `&&` for and, and | for or:

```
$ if [[ -f /etc/passwd && $name = max ]]; then
>   echo file exists and name matches
> fi
```

There is also an older form, the single bracket `[`, which you will see in other people's scripts. It works similarly but has more pitfalls—you have to quote your variables inside it, and it does not support `&&` or patterns. If you are writing Bash, use `[[`. There is no good reason not to.

※

Loops in Bash come in two main varieties, and each one is useful for different things.

The `for` loop iterates over a list of words:

```
$ for fruit in apple banana cherry; do
>   echo "I like $fruit"
> done
I like apple
I like banana
I like cherry
```

The list of words after `in` follows the same word-splitting rules we have already discussed. Each word becomes the value of the variable for one pass through the loop. The `do` and `done` play the roles that curly braces play in most other languages.

Where this gets genuinely useful is with file globbing. The pattern `*.txt` expands to all the filenames in the

current directory that end in `.txt`, and each filename becomes one word:

```
$ for file in *.txt; do
>   echo "Processing $file"
>   wc -l "$file"
> done
```

Notice that I put `"$file"` in double quotes when passing it to `wc`. This is because filenames can contain spaces, and if they do, an unquoted `$file` would be split into multiple arguments. This is the habit. Quote your variables.

The `while` loop runs as long as a command succeeds:

```
$ count=0
$ while [[ $count -lt 5 ]]; do
>   echo $count
>   count=$((count + 1))
> done
0
1
2
3
4
```

The `$((...))` is arithmetic expansion. It is one of the few places in Bash where numbers are treated as numbers rather than strings. The `-lt` means less than.

Inside `[[`, numeric comparisons use abbreviations: `-lt`, `-gt`, `-eq`, `-le`, `-ge`, and `-ne`. Strings use `=` and `!=`.

But the most useful thing you can do with a `while` loop is read input line by line:

```
$ cat names.txt | while read line; do
>   echo "Hello, $line"
> done
```

The command `read` is a built-in that reads one line from standard input and assigns it to a variable. It returns zero when it successfully reads a line and nonzero when there is nothing left to read. So `while read line` means: keep reading lines until there are no more.

This pattern—piping the output of one command into a `while read` loop—is one of the most common and most powerful patterns in Bash. It is how you process the output of a command line by line, doing whatever you need to do with each line. We will come back to this when we talk about pipes.

※

Functions in Bash look a little strange if you are used to other languages, but they are simpler than they look.

```
greet() {
  echo "Hello, $1"
}
```

That is the whole declaration. There is no keyword like `function` or `def` before the name—just the name, a pair of empty parentheses, and the body in curly braces. Bash does have an optional `function` keyword, but the parentheses style is more portable and more common.

The parentheses are always empty. This is the most confusing thing about Bash functions for people coming from other languages. You do not declare parameters. You access them by position: `$1` is the first argument, `$2` is the second, `$3` the third, and so on. `$@` is all of them.

```
greet() {
  echo "Hello, $1. You are $2 years old."
}

$ greet max 25
Hello, max. You are 25 years old.
```

Notice that you call a function the same way you call a program: the function name followed by its arguments, separated by spaces. No parentheses around the arguments. No commas between them. A function call in Bash looks exactly like a command invocation, because it is one.

This is not a coincidence. In Bash, a function is a command. It is a command that happens to be defined

in your script rather than being a separate program on disk, but it behaves exactly like one: it receives arguments the same way, it has an exit code, and it can be used anywhere a program could be used.

The exit code of a function is the exit code of the last command it ran, unless you explicitly set it with `return`:

```
is_root() {
  [[ $(whoami) = root ]]
}

file_exists() {
  [[ -f $1 ]]
}
```

These functions have no `echo` and no explicit `return`. They just run `[[` and let its exit code be their exit code. So you can use them directly in an `if`:

```
if file_exists /etc/passwd; then
  echo found it
fi
```

The distinction between return codes and output is important and often misunderstood. A function's return code is a number, zero or nonzero, signaling success or failure. A function's output is whatever it prints to standard output with `echo` or `printf` or any other command. These are two different channels.

You can capture a function's output in a variable using command substitution:

```
current_user() {
  whoami
}

$ user=$(current_user)
$ echo you are $user
you are max
```

The `$(...)` runs the command inside it and replaces itself with whatever that command printed. This is how you get a return value out of a function in the sense that most programmers mean when they say return value.

If you need local variables inside a function—variables that do not leak out into the calling scope—you use `local`:

```
greet() {
  local greeting="Hello"
  echo "$greeting, $1"
}

$ greet max
Hello, max
$ echo $greeting
```

Without `local`, every variable you assign inside a function is global. This is one of Bash's genuinely bad defaults. Get in the habit of declaring your function variables with `local`.

※

There is one more thing worth knowing before we get to pipes, and that is the here-document. It solves a problem you will hit quickly: how do you pass multiple lines of text to a command?

```
$ cat <<EOF
Hello, world.
This is multiple lines
of text.
EOF
Hello, world.
This is multiple lines
of text.
```

The `<<EOF` means: take everything from the next line until you see a line that says `EOF` by itself, and feed it to the command as its standard input. The word `EOF` is not special—you can use any word you like as the delimiter. `EOF` is just a convention.

Variables are expanded inside a here-document, just like inside double quotes:

```
$ name=max
$ cat <<EOF
Hello, $name.
Today is $(date).
EOF
Hello, max.
Today is Fri Aug 22 14:30:00 UTC 2026.
```

If you do not want variable expansion, put the delimiter in quotes:

```
$ cat <<'EOF'
Hello, $name.
This is literal text.
EOF
Hello, $name.
This is literal text.
```

This is enormously useful when you need to embed a chunk of another language inside your script—a SQL query, a block of HTML, a configuration file—without Bash trying to interpret all the dollar signs and backslashes in it.

There is also the here-string, which is the same idea but for a single line:

```
$ cat <<< "hello world"
hello world
```

It is a small convenience but a welcome one. Instead of `echo "hello" | somecommand`, you can write `somecommand <<< "hello"`, which makes the data flow more visible.

※

Now we get to the part I have been building toward since the first page.

A pipe connects the standard output of one program to the standard input of another. You already know this. You have already typed `ls | grep something` a thousand times. What I want to do now is make you understand what is actually happening, because once you do, the whole design of Unix will make sense in a way it probably did not before.

Every program in Unix has three channels open from the moment it starts: standard input, standard output, and standard error. Standard input is where it reads from. Standard output is where it writes to. Standard error is where it complains to. By default, standard input comes from the keyboard and the other two go to the screen.

A pipe replaces these defaults. When you write this:

```
$ ls | grep txt
```

what happens is that Bash starts both `ls` and `grep` at the same time, and connects the standard output of `ls` to the standard input of `grep`. The programs run in parallel. `ls` writes its output without knowing or caring that the output is going to another program rather than to the screen. `grep` reads its input without knowing or caring that the input is coming from another program rather than from the keyboard. Neither program has to know about the other. The pipe is a wire between them, and each end of the wire looks exactly like the default.

This is the most important idea in all of Unix. Programs do not need to know about each other. They do not need to be written in the same language. They do not need to agree on a binary format or a function signature or a data structure. They agree on one thing: text goes in, text comes out. The pipe connects any program that writes text to any program that reads text. That is the entire protocol.

Let's build something real. Suppose you have a log file full of lines like this:

```
2026-08-22 ERROR database connection failed
2026-08-22 INFO user logged in
2026-08-22 ERROR timeout exceeded
2026-08-22 INFO page loaded
2026-08-22 ERROR database connection failed
```

And you want to know which errors occur most frequently. In most programming languages, you would write a script that reads the file, filters the error lines, extracts the message, counts occurrences, and sorts by count. That is maybe fifteen or twenty lines of code.

In Bash:

```
$ grep ERROR logfile.txt | cut -d' ' -f4- |
  sort | uniq -c | sort -rn
      2 database connection failed
      1 timeout exceeded
```

Five programs, each doing one thing. `grep` filters to only the error lines. `cut` extracts everything from the fourth field onward (the message, without the date and the word ERROR). `sort` sorts the lines alphabetically, which groups identical lines together. `uniq -c` collapses adjacent identical lines into a count. The final `sort -rn` sorts the counts in descending numeric order.

None of these programs is complicated. Each one is almost trivially simple. The pipeline is what makes them powerful. And the pipeline is just Bash connecting their channels.

Here is another one. You want to find the ten largest files in a directory tree:

```
$ find . -type f -exec du -h {} + |
  sort -rh | head -10
```

Three programs. `find` walks the directory tree and lists the files. `du -h` reports their sizes in human-readable form. `sort -rh` sorts by human-readable size in reverse order. `head -10` takes the first ten lines. Done.

This is what I meant at the very beginning when I said that many things that are nearly impossible to do in any reasonable way in popular programming languages are downright trivial in Bash. It is not that the individual operations are hard. It is that other languages make it hard to compose the operations together, because other languages do not think of programs as building blocks. They think of functions as building blocks and programs as finished products. Bash thinks of programs as functions. Any program is a function: it takes input, it produces output, and it can be composed with any other.

When Doug McIlroy wrote that memo in 1964 about coupling programs like garden hose, this is what he was imagining. He imagined it thirteen years before I was born and it is still the most productive way to work with a computer that anyone has ever come up with. The whole of the modern software industry is an elaborate and ongoing effort to avoid learning this.

※

There is one thing I have glossed over that deserves a few more words, because it will save you grief.

The `$(...)` syntax is called command substitution, and it captures the output of a command as a string. You have already seen it used to assign a value to a variable:

```
$ files=$(ls)
```

But it can be used anywhere a string can be used, including inside double quotes:

```
$ echo "There are $(ls | wc -l) files here"
There are 42 files here
```

This is how you embed the result of one pipeline inside another command. The command inside `$(...)` runs in a subshell—its own little environment—and its standard output is captured and pasted into the surrounding command.

Combined with everything else, this gives you a remarkably complete programming language. You can write functions that call other functions, capture their output, make decisions based on exit codes, loop over lines of text, and compose any number of programs

together with pipes. It is not a language for writing a web server or a video game. It is a language for getting things done with the tools that already exist on your computer, and for that purpose, nothing else comes close.

I have not covered everything. I have not talked about arrays, which Bash has and which are occasionally useful. I have not talked about traps, which let you clean up when a script exits. I have not talked about process substitution, which lets you treat the output of a command as a file. I have not talked about the many options and flags that change Bash's behavior in important ways, like `set -e` which makes a script exit on the first error, or `set -u` which makes unbound variables an error rather than an empty string.

But you now have enough to start writing real programs in Bash, and the things I have not covered are things you will learn as you need them, the way everyone does—by having a problem, looking it up, and discovering that Bash has a tool for it that you did not know about. Bash has a lot of tools you do not know about. That is part of the fun.

The other part of the fun is the one I started with. Every program on your computer is a function you can call. Every pipeline is a composition of functions. The

programs do not need to know about each other and they do not need to speak the same language. They are coupled like garden hose: screw in another segment when you need to massage the data in another way. It was a good idea in 1964, it is a good idea now, and it will still be a good idea when everything else about how we write software has changed.

