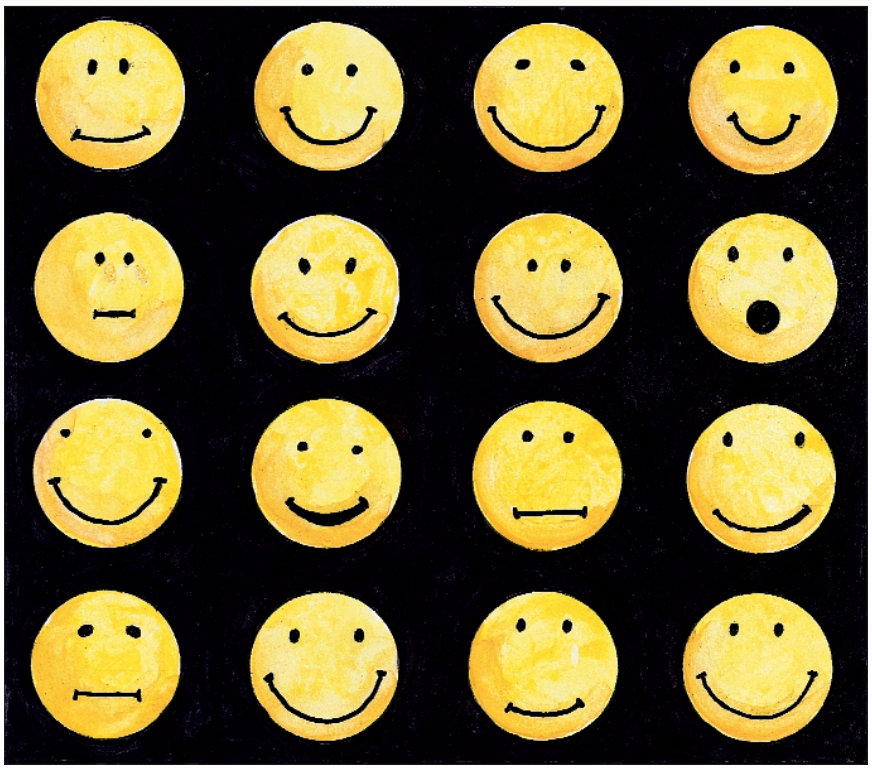
by Peter Collinson, **Hillside Systems**



DAVID FAUST

Differences Between Files

I spend my life creating and modifying files and I am sure that you do too. If I have a new undertaking, I've usually done something similar in the past and can use that work as a boost to start on the new task. It's rare for me to start completely from scratch with an empty file.

It's often much easier to grab some old work and bend it to the new requirements. Sometimes I plan for this eventuality and spend time constructing template files that act as the leaping off point for some work or other. For example, I set up `troff` templates for business and personal letters I occasionally have to write. The business templates include my company information with a space to place the destination address into window envelopes. I've used the templates ever since. The time spent in generating the templates has been more than offset in the long run by the time saved every time that I create a letter.

If the work I am grabbing to reuse is a program, then a real effort of will is needed to recognize that the code should be placed in distinct modules and shared between the old and new application. Of course, sharing makes sense–most of the time. For example, if a bug appears, then it can be fixed in one place and all the applications that use the file will benefit.

Sometimes, "getting the job done" takes precedence over "getting the code structure right." It can be quicker to take old source and hack it to make it work for the new requirements. Also, there are often good engineering reasons for generating a completely fresh source for the new job starting with the source for the previous one.

For the past 10 years, I've been largely responsible for the Web booking forms you use when planning to attend one of the wide range of conferences run by the USENIX Association. The booking forms are complex–with several pages of data entry–but the requirements can change dramatically from conference to conference.

The source for these systems is in Perl, and to set up a new conference I generally start with the code for the last one. Over time, the code has become better organized and considerably more modular. Some parts of the system have migrated into Perl modules, but these tend to be low level code elements that have proven to be constant.

In general, I don't share code between conferences because several conferences are taking bookings at any one time and a necessary change for one conference may conflict with the needs of another. Any change requires testing, so it's a bad idea to interfere with live forms on the Web, which could possibly break the system while an unsuspecting person is attempting to use it. It's easier and considerably better engineering to create a completely new source set for each conference. I clone the old source and the new code is then adapted to the needs of the conference. The entire system can be tested before it is placed live.

Taking old work and bending it to the needs of a new job is a very common activity, and I am sure you recognize your

activities in this discussion. However, as our collective file systems fill with copies of files that are related or possibly identical, we need to find some way of asking questions about their contents. It's common to want to know if the contents of files are the same, or to ask what changes have been made to a file since a copy was made. UNIX has several tools that can be used to answer these questions. The tools form part of the basic UNIX toolkit and have been around relatively unchanged for some time.

Comparing Files

We can get an idea if files differ by using the `ls` command and seeing if the files differ in size.

```
$ ls -l
-rw-r--r--   1 pc    27 Oct  8 11:11 f_one
-rw-r--r--   1 pc    39 Oct  8 11:27 f_two
-rwxr-xr-x   1 pc  6052 Oct  7 11:09 x_one
-rwxr-xr-x   1 pc  6052 Oct  7 11:16 x_two
```

I've abbreviated the `ls` listing for publication purposes. If `f_two` was derived from `f_one`, then we know that it's now different from the original because they have different sizes. However, what about `x_one` and `x_two`? These files are the output from a C compiler and may differ even though their sizes are identical.

The `cmp` command performs a byte-by-byte comparison of two files. Nothing will be printed if the files are the same, but a message is output if they differ.

```
$ cmp x_one x_two
x_one x_two differ: char 2292, line 6
```

The command compares the bytes in the files and prints a message when the first difference is found. The files are different, but how do they differ? The `-l` (*ell* not *one*) option lists the differences.

```
$ cmp -l x_one x_two
  2292 233 173
  2515 144 104
```

The listing here betrays the antiquity of the `cmp` command. The output is a line for each differing byte. Each line shows the offset in decimal from the start of the file where the difference is found and then lists the differing bytes in octal. The output from the command can be hard to interpret, depending on the source. For this example, I changed a lower-case "d" (octal 144) to an upper-case "D" (octal 104) and recompiled the source program to generate `x_two`, but it isn't clear to me why this also resulted in a change to byte number 2292.

The `cmp` command is also useful when shell scripting. It returns a status value that indicates what happened when the files were compared. Zero is returned if the files are identical, one if the files differ, and a value greater than one if an error occurred. If you are copying files in a script, and only want to copy if the files are different, code such as

```
if cmp -s srcfile dstfile
then
   echo 'Files OK'
else
   cp srcfile dstfile
   echo 'Files updated'
fi
```

will do the trick for you. The `if` command tests the result of the `cmp` command and executes the first branch if the result is zero and the second branch if not. The `-s` flag to the `cmp` command makes it work "silently" so nothing is printed, even if there is an error. If the files are the same, then the first arm of the `if` statement is executed and the comforting message is printed. If the files differ, or the target file doesn't exist, the `cmp` command will print nothing and returns a non-zero status. The second arm of the `if` statement will be executed, copying the file and printing the update message. I use this type of coding considerably when archiving files so that the dates on the stored files are only altered when the source file has changed.

Incidentally, if you want the script to work silently, then you can invert the sense of the test with the exclamation mark (which means "not"):

```
if ! cmp -s srcfile dstfile
then
   cp srcfile dstfile
fi
```

I don't like using the exclamation mark since I find it easy to miss later when reading complex code. Also, I think the syntax can seem counter-intuitive. For clarity, I prefer to use the null statement `(:)`, writing:

```
if cmp -s srcfile dstfile
then
   :
else
   cp srcfile dstfile
fi
```

The `cmp` command has a number of uses but is mostly used to compare binary files. It can tell you whether a graphic image in one file is the same as another. It can be used to compare command binaries, although this isn't a guarantee that the programs are different. Different compilers, or different compiler options, can generate different binaries for the same functioning program. Finally, it can be used to verify that file copies have worked if other means, like obtaining a checksum for the files, are unavailable to you.

Comparing Text Files

The `cmp` command is a useful tool in your personal UNIX toolkit. It answers the question "Do these files differ?" and is usable with any file you can find on the system. When dealing with text files, however, we often want to find out how the files differ by asking "What changes have been made to the file?" The `diff` command is the tool designed for this job.

```
$ diff f_one f_two
2c2
< line b
---
> line two
3a4
> line four
```

The aim of the default output from the `diff` command tells you the changes that are needed to alter the first file into the second. These two tiny example files have two differences. Line 2 of each file is different with `f_one` having "line b" and `f_two` having "line two." Second, `f_two` has an additional line: "line four." The output is reminiscent of commands used to drive the `ed` (or `ex`) editor. The line specification that tells you where the differences are contains `ed` commands: "c" means "change the line" and "a" means "append." So `2c2` means that line 2 in the first file has been "changed" which matches line 2 in the second file; `3a4` means that the line that follows is line 4 of `f_two` and should be appended after line 3 of `f_one`.

If we call `diff` and reverse the order of the files, we can see that the command is telling us something different.

```
diff f_two f_one
2c2
< line two
---
> line b
4d3
< line four
```

Now to get from `f_two` to `f_one` we delete ("d") line 4. The example underlines the thinking behind the output and is telling us how to convert one file into another.

White space in files can be a problem. Lines can contain the same information but can be padded with tabs or space characters that humans wish to ignore. Some text editors add white space to the end of the lines without you being particularly aware of it. The `diff` command has an option (`-b`) that ignores any trailing white space when making a comparison. You can also force a case independent comparison by supplying the `-i` option.

Although the `diff` output shown looks comprehensible, it can sometimes be difficult to understand when the files you are comparing are large and contain a considerable number of repeated lines. The most difficult task for `diff` is matching up repeated sections in files so that it doesn't report differences where no differences exist. The early version of `diff` was prone to panicking and printing `Jackpot` when it came across a pair of files whose differences caused its matching algorithm to loop endlessly.

Modern versions of `diff` do a much better job, but the output can sometimes be counter-intuitive. It's common to see what appears to be odd groupings of lines when comparing program source that contains many lines having similar content. These odd groupings are usually only odd to you, because you know about the structure of the files, and `diff` is only looking at the patterns caused by the contents.

Although `diff` is intended to be used to compare text files, modern versions will work on binary files. The code will not attempt to apply the text comparison algorithm if a binary file is encountered. So

```
$ diff x_one x_two
Binary files x_one and x_two differ
```

The command has noticed that the files are "binary" and just tells you that they differ.

Context *diff*

Quite often you want to do a `diff` on a pair of files so you can apply the changes you have made to a third file, such as porting bug fixes you made to one file into a file that was derived from the same source. When the source is complex and terse, you want to see the lines around the difference to be able to locate the correct place in the source file you are editing.

The "Context diff" option, `-c` allows you to do this.

```
$ diff -c f_one f_two
*** f_one  Mon Oct  8 11:11:15 2001
--- f_two  Mon Oct  8 11:27:07 2001
***************
*** 1,3 ****
  line one
! line b
  line three
--- 1,4 ----
  line one
! line two
  line three
+ line four
```

The listing starts with the names of the files and their last modified dates. The first file has `***` before it and the second `---`. These symbols are used to mark the sections of the files that follow. Different lines are then shown with an exclamation mark before them; lines that are added are prefixed with a `+`, and lines that have been removed are marked with a `-`.

It's much easier to place this type of output into an editor and use the file contents to ensure that you are making the correct changes to that third file.

It can sometimes help to have the two files listed side by side so you can see the lines that differ. The `sdiff` command permits this, although a problem can be getting enough screen real estate to see the result. Here is abbreviated output from my two sample files:

```
$ sdiff f_one f_two
line one            line one
line b          |   line two
line three          line three
                >   line four
```

Multiple Files and *diff*

Although `diff` started life simply to compare two files, its ability to handle directories that contain similar file sets has grown over the years. If one or more of the arguments are directories, `diff` will "do the right thing."

```
diff file directory
```

compares `file` with `directory/file`. Supplying two directories will compare their contents.

```
$ diff -c 1 2
diff -c 1/f_two 2/f_two
*** 1/f_two     Mon Oct  8 11:43:06 2001
--- 2/f_two     Mon Oct  8 11:43:30 2001
***************
*** 2,4 ****
--- 2,5 ----
  line two
  line three
  line four
+ line five
Binary files 1/x_one and 2/x_one differ
Only in 2: xy
```

The listing is somewhat condensed, and a fake `diff` command is printed for every file that is compared. Since I've asked for a context `diff`, this is done for `f_two` in both directories. It picks up any binary files that are different and reports missing files. You can make the command recurse down a pair of hierarchies by adding the `-r` option, which makes it possible to find what files differ in trees that started life as clones.

Using *diff* Output

I've said that the output is close to that of UNIX's `ed` editor, and you can make `diff` output a script that can be fed into the editor by supplying the `-e` flag.

```
$ diff -e f_one f_two
3a
line four
.
2c
line two
.
```

Notice that the `a` (append) command is output first, because it changes the line numbers that the editor is using to address the file. It's possible to dump this output to a file and then use that file as an editing script that's fed into `ed`. You have to be careful that the line numbers to be edited by the `diff` output are the right line numbers in the file you are changing automatically.

The idea of applying the output from `diff` automatically was explored and implemented in a reliable manner by Larry Wall, who went on to create Perl. Larry addressed the problems of applying `diff` output with his `patch` program. He wanted to provide a way for software authors to maintain their programs by sending out "patch files" that were essentially just the output of the `diff` command.

Software authors would clone their distribution files in a new directory, and edit them to create the next distribution. They would then use `diff` to create a listing of all the changes that had been made to all the files. This listing could be sent over the network and applied automatically using the `patch` program to a set of original sources at some site in the world. Actually it's unlikely that programmers would clone source trees. Source management systems like SCCS or RCS have the ability to generate file differences between the current working version of a file set and previous versions, but hopefully you get the idea.

There were several reasons why updating source programs with `diff` output became popular. First, it wasn't that easy to obtain files from remote machines. Machines were connected using UUCP, and files were generally pushed out in the news system or passed around on tape or disks. New sources were often large, and it was deemed to be too much of a load to transmit huge numbers of bytes over the network. It was more efficient to transmit the differences. Second, updating source programs with `diff` made it possible to send changes to source protected by source licenses. Sending UNIX source files out over the network violated the original UNIX license agreements. You could send the changes you had made to the files, and there was no real complaint about sending the odd line of original code to act as a reference.

As we've seen, there are problems with sending out changes to files based on line numbers; `patch` understands the source file you are applying the changes to may not be exactly the same as the source that acted as a reference when the `diff` listing was made.

If I've obtained some source from you, I may have altered that source myself before you had sent me the patch file. If possible, I'd still like to be able to automatically apply your bug fixes while preserving my changes. There are cases where your changes may conflict with mine, where `patch` would object and create a file that I could investigate and merge in later. However, I've either added or deleted some code, so the line numbers attached to the changes specified in the `diff` listing need to be re-synchronized with the source before they are applied.

The `patch` program makes good use of the redundant information in the output from `diff`. It looks at the quoted source as the "original" in the output and compares it with the source that it finds in the file being patched. You'll find that it works much more reliably when supplied with context `diff` output.

I often use `patch` when updating programs that have started from a shared source. I'll use `diff -c` to create a difference file, edit the listing to just contain the needed changes and use `patch` to apply the changes automatically to the target file. You will be surprised how often this technique finds and applies changes that you had forgotten you had made. ⊸

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever ... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.