Yoshihiro Matsumoto (M'74-SM'78-F'82) graduated from the Department of Electrical Engineering and received the Dr.Eng. degree from the University of Tokyo, Tokyo, Japan, in 1954 and 1974, respectively.

He is a Principal Engineer at the Heavy Apparatus Engineering Laboratory, Toshiba Corporation, Tokyo, where he has worked since 1954. He is now interested in the development of advanced computer control system, methodologies, and tool systems to improve manufacturing productivity in large systems factories. He is a Visiting Lecturer at Keio University.

Dr. Matsumoto has served on various IEEE chapters, committees, and conference boards since 1975. He was awarded the "National Invention Award" in 1981 and "Excellent National Researcher Award" in 1982, both from the Japanese government. In 1982, he was elected as an IEEE Fellow for his leadership in the application of computers in industrial control systems.

The Unix System and Software Reusability

BRIAN W. KERNIGHAN, MEMBER, IEEE

Abstract—**The Unix system contains a variety of facilities that enhance the reuse of software. These vary from the utterly conventional, such as function libraries, to basic architectural mechanisms, such as the Unix pipe. The Unix pipe, which makes whole programs building blocks of larger computational structures, has been the primary reason for the development of a literature of useful, but specialized programs—programs that would be too costly to write in a conventional programming language such as C. It has led to high levels of program reuse both by the nature of its operation and through its effect on programming conventions (e.g., programs structured as simple filters).**

Another facility enhancing reuse on Unix is the on-line C source code for Unix system programs. This has led to a shared style of programming in which existing programs are used as models for new programs, allowing the reuse of ideas, algorithms and source code. Finally, the Unix system contains many other reuse enhancing facilities, such as generic facilities for screen management (*curses* and *termcap*) and program generators (*lex* and *yacc*).

Index Terms—**Filter, generator, library, I/O redirection, pipe, Unix.**

Introduction

WHAT is sofware reusability? Although I have a vaguely formed personal opinion, scholarship demands a dictionary definition. Not surprisingly, however, "reusability" does not occur in any of the dictionaries I consulted. Accordingly, I feel free to make my own definition, or at least my own interpretation of the term.

If reusability is defined to include any way in which previously written software can be used for a new purpose, or to avoid writing more software, then it is clear that the Unix[1] operating system permits, and indeed encourages, the reuse of software, in many different ways. Some of these are utterly conventional, such as libraries of functions. Some were novel at the time that they were first used, although by now they have become a commonplace. And some are still new, or at least worth wider circulation.

In this paper, I will discuss these more or less from the bottom up, beginning with subroutine libraries. As will become apparent, this paper is not a learned treatise, but a statement of some personal observations on one fairly large Unix community.

Library Level

Let us dispose of some of the obvious topics first. A standard library mechanism permits compiled code to be stored in a file for subsequent access and binding by a loader. The most frequently used library is the standard I/O library, which provides services like file access and format conversion for programs written in the C programming language [1]. (Unlike most high-level languages, C itself does not define any I/O facilities.) The standard I/O routines are sufficiently well defined that

Manuscript received August 1, 1983.
The author is with Bell Laboratories, Murray Hill, NJ 07974.
[1] Unix is a trademark of Bell Laboratories.

0098-5589/84/0900-0513$01.00

they can be implemented on most operating systems, and so C programs that confine their system interactions to the facilities of this library can be moved without change from one system to another. They are also efficient enough that there is not too much tendency for people to reimplement.

The standard I/O library also includes routines for string handling, since string operators are not part of C, and for free-store management. Other traditional libraries include numerical routines (accessible from C, though primarily for Fortran), various graphics packages, and routines for accessing big files with B-trees or extensible hashing.

One other area that permits a major re-use of software is code that defines a parameterized interface to ASCII terminals. A (large) table defines the characteristics of hundreds of terminals; programs that perform screen management in terms of the generic facilities encapsulated in this table and a matching library [2] will run properly on any terminal without change. This library goes a long way towards making programs independent of the idiosyncrasies of individual terminals.

Lest it be thought that all is perfect in the Unix world, however, there are some areas where common sense cries out for a library routine, yet none is in widespread use. One particularly obvious case is code to handle optional arguments for commands. The style that has evolved is that programs specify optional arguments in the form

```
program -option -option ... filenames
```

But there are myraid variations on this theme, each determined solely by the whim of individual programmers. Here are just a few

ls -lt	*sets* l *and* t *options*
tee -a -i	*sets* a *and* i *options*
troff -o1	*sets* o *option to* 1
graph -m 1	*sets* m *option to* 1
tar r1	*sets* r *option to* 1

(We pass over the fact that '**r**' means 'write' in this last example.) Some versions of Unix do provide an argument-parsing routine—for example, see **getopt** in System V—but no standard has emerged, perhaps because it's so easy to write an *ad hoc* argument parser.

Programming Language Level

Most programming on Unix systems is done in C. Although C has been castigated for syntactic problems, inadequate type-checking, and the like, the advantages it provides—expressiveness, efficiency, compactness, freedom from restriction, portability—have proven so compelling that it is widely used. The operating system and essentially all applications software are written in C.

As a programming language, C is sufficiently well defined that with some care it is possible to write programs that will move from one machine to another without change. (This is best done in cooperation with the standard I/O library mentioned above.) Since C itself permits some highly nonportable constructions, a program called **lint** is used to detect various unsafe constructions and portability problems. **lint** was created by reusing code: it shares its front end with the other compilers, but its second pass does careful checking instead of code generation.

It is hard to overstate the importance of using a high level language. Many Unix systems have the source code for commands, libraries, etc., on-line and accessible to all users. Accordingly, it is straightforward to find a program, read its code, and use that as a model or a starting point for a new program. Sharing assembly language code is far harder; the set of people who can read and profit from it is much smaller. The use of C and the public accessibility of the source code for programs has led to a shared style of programming, a set of conventions, and the reuse of ideas, algorithms, and source code. The reuse of source code in this way supplements the usual object library mechanism.

The only negative aspect of being able to read and understand programs is that people do read and understand, and then feel free to modify. The result is sometimes an improvement in an old program, but is all too often merely a difference, leading to a proliferation of variants. Few programs are unscathed by gratuitous tinkering. Far more serious, however, is that there are even within Bell Labs three or four major versions of the operating system in widespread use, sufficiently different that some programs need conditional code to compile and run properly on all of them. The increasing number of microprocessor Unix systems threatens to make the situation worse.

It is believed (or at least hoped) in some circles that abstract data types, as exemplified in Ada packages, will provide a better mechanism for code sharing. In its current form, C does not have an analogous facility, but some extensions by B. Stroustrup provide capabilities for encapsulating data structures and functions in a style based on Simula classes. The first implementation of this facility was a preprocessor that outputs C [3], it has been used in several projects at Bell Labs. Stroustrup's current work extends the class notion to provide generic functions and operator overloading as well as encapsulation. It is implemented as a compiler rather than a preprocessor, so it does a better job on things like error detection.

One advantage of Stroustrup's approach, as compared to Ada or CLU, is that it builds on a successful existing language instead of defining an entirely new one. This retains the assets of the base language, including its efficiency, portability, and established user population. Users can work into the class system gradually, beginning with its better checking of vanilla C, before learning the new capabilities.

Program Level

Most of the reuse of software in Unix occurs at the level of individual user-invoked programs where a style has evolved that employs existing programs in novel ways and in combinations with others in preference to writing new ones. In effect, entire programs become building blocks analogous to functions in a programming language. Most of the notions of this section have been described elsewhere—for example, see [4]—but it is worth a brief overview here.

The basic idea is to combine programs using the capabilities of the command interpreter (the "shell"). The shell runs individual programs, setting up the environment of open files,

etc., for each before it begins. Normally, each program begins with two open files known as the *standard input* and *standard output*, both by default connected to the user's terminal. By a command to the shell, however, either or both of these default assignments can be changed, without the program being aware of it. For example, the command **who** produces a list of logged-in users on the terminal, one per line:

```
who                          input
bwk    tty4   Jun 5 19:44    output . . .
cvw    tty6   Jun 5 18:21
```

To redirect the output of **who** to the file **temp**, this suffices:

```
who >temp
```

The program **lc** counts lines of input, from its standard input or from a list of files named as arguments. The standard input of **lc** can be directed to a file with the shell operator<, which is analogous to >:

```
lc <temp
```

The combination of these two programs, communicating through the file **temp**, counts the number of users. (It is a better demonstration on a system with 20 users instead of 2.)

Although it is often convenient or even essential to capture the output of a program and reuse it later, in this specific example the temporary file is clumsy. What is needed is a way to connect the output of one program like **who** to the input of another program like **lc** without a temporary file.

This is exactly what is done by a unique contribution of Unix, the *pipe*. The following command does the same job as above, but without any temporary file:

```
who ¦ lc
```

In a pipeline, the system schedules the programs, synchronizing them so that the producer waits if it gets too far ahead of the consumer, and vice versa.

In microcosm, this trivial example is typical of Unix use: two programs are connected transiently to do a job that is worth mechanizing but not worth writing a special program for. It works because each program does its job straightforwardly (i.e., without extra features that would prevent cooperation); programs share conventions like the standard input and output files, representation of text, etc., that permit the output of one to be meaningful as input to another; and the system provides an elegant and efficient way to arrange the interconnection. It is also absolutely true that this construction was never thought of by the implementors of Unix or the programs involved; their programs are certainly being "reused."

Similiar examples abound; we have room for only a few. For example, the program **grep** prints each line of its input that matches a pattern given as a command-line argument. So the command

```
who ¦ grep joe
```

answers the question "Is Joe logged in?" and

```
who ¦ grep joe ¦ lc
```

answers "How many times is Joe logged in?"

The shell is actually a programming language, with control flow, variables, subroutine calls, interrupt handling, etc. Those additional capabilities can be used to build more complicated programs out of the basic ones. For example, we can set up a loop to watch for Joe to log in.

```
until who ¦ grep joe
do
   sleep 60
done
```

The loop **until** *cmd* **do** . . . **done** exits when *cmd* succeeds. The pipeline **who ¦ grep** returns the status returned by its last element, **grep**. If **grep** finds Joe in the output of **who**, the line is printed, **grep** returns true, and the loop terminates. Otherwise, **grep** returns false and a 60 second nap is taken before another iteration.

The shell is just an ordinary user program, not part of the operating system, so it can be invoked with its input coming from a file containing commands. A shell file can be invoked merely by naming it (after it has once been marked executable); there is no difference between invoking a shell file program and one written in a language like C. It is possible to pass arguments into such a shell file, to be substituted when it is run. So we can package our "watch for Joe" program as a shell program that will watch for anyone by placing these lines in a file called **watchfor**.

```
until who ¦ grep $1
do
   sleep 60
done
```

$1 will be replaced by the first argument of **watchfor** when it is executed.

When told to execute a program, the shell searches for it in a sequence of directories specified by the user; most users have their own directory of personal commands like **watchfor** that is searched before the standard command directories. My directory has 48 such private commands (counted by piping the directory lister into **lc**, of course); all but five are shell files, and most are only a few lines long. For example, the program **lc** mentioned above is actually a call to a more general program with parameters set so that only the line count is printed. This is quite typical: people routinely use the capabilities of the shell to cover up defects in existing programs or to combine them into new ones; it is much easier than writing a new program from scratch. Sometimes such programs have enough appeal that they are moved into the standard system-wide command directory and thus become available to everyone by default.

As a somewhat larger example of the same sort of thing, consider the program **cal**, which prints a calendar for a specified month and year. It's a program of slightly more than 200 lines of C that encapsulates a messy, technical computation, and provides a standard, documented behavior. The only problem is its user interface. For example, with no arguments, **cal** complains; it will not accept a month name, only a number; and it assumes that a single numeric argument is a year. In the following dialog, '$' is the shell's prompt.

```
$ cal
usage: cal [month] year
$ cal sept 1983
Bad argument
$ cal 9 1983
      September 1983
 S  M Tu  W Th  F  S
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

It surely would be better to have **cal** produce the calendar for the current month by default, and to recognize month names like **sept** instead of requiring a number.

With a simple shell program, we can cover up the defects in **cal** without touching or even examining its internals. The shell program is shown here so that Unix users can read it for details (explained in [5]) while others can verify that it isn't very large or complicated.

```
case $# in  # how many arguments were given?
0) set 'date'; m=$2; y=$6 ; ; # no args: use today
1) m=$1; set 'date'; y=$6 ; ; # 1 arg; use this year
*) m=$1; y=$2 ; ;  # 2 args: take them literally
esac

case $m in     # what month?
jan*|Jan*)   m=1 ; ;
feb*|Feb*)   m=2 ; ;
mar*|Mar*)   m=3 ; ;
apr*|Apr*)   m=4 ; ;
may*|May*)   m=5 ; ;
jun*|Jun*)   m=6 ; ;
jul*|Jul*)   m=7 ; ;
aug*|Aug*)   m=8 ; ;
sep*|Sep*)   m=9 ; ;
oct*|Oct*)   m=10 ; ;
nov*|Nov*)   m=11 ; ;
dec*|Dec*)   m=12 ; ;
[1-9]|10|11|12) ; ;    # numeric month
*)  y=$m; m="" ; ;  # plain year
esac

/usr/bin/cal $m $y   # run real cal
```

This determines the month and year from the **date** command if the user doesn't provide them, then converts the information into the right form for the real **cal** program, which is invoked as the very last step. This file is placed in one's own command directory under the name **cal**.

In effect, the approach is to treat an existing program as a black box with known but in some way inadequate properties. Rather than rewrite it, one wraps it up in a (much smaller) program that provides more suitable properties, or that permits the job to be factored into easy pieces.

It is often possible to write a program from scratch in the shell more easily than in C. One example is the **news** program on our system. **news** searches a standard directory for files that are newer than a particular file in the user's home directory, and prints those that are newer, with a heading. The user's file is then updated to reflect when news was last read. The C version of this program is 360 lines long; the shell version is less than 20 lines.

The use of the shell as a programming tool has evolved over the years. It began a decade ago with file redirection, but the real revolution occurred with the invention of pipes, for these made program connections syntactically clean, and eliminated the temporary files. The programmability of the shell itself has increased, reaching a plateau with the 7th Edition shell written by S. R. Bourne [6]. There has been a steady increase in the use of shell procedures instead of programs written in conventional languages; some of this growth is documented in [4].

One interesting sidelight has been the parallel evolution of programs that, like the shell, facilitate arbitrary connections of other programs as well as being useful in their right. There are two major examples of these "programmable filters."

The first is **sed**, a stream editor. **sed** applies a sequence of editor commands to each line of its input before writing the transformed line to the output. For example, to change UNIX to Unix on each line, this command is used:

```
sed 's/UNIX/Unix/g' filenames. . . >output
```

sed was developed originally for editing files that were too big for the standard editor, but it is now frequently used to match the output format from one program to the input format for another by making some simple editing transformation as the data flows through a pipeline.

The second general-purpose filter is **awk**, which was intended for writing data validation and report generation programs. Although it is still used primarily for those purposes, it is also increasingly a component in shell programs since it is better at handling numbers and storing data than either the shell or **sed**. As an illustration, again more to show scale than for details, the following program watches all users who log in or log out, reporting the changes every minute:

```
new=/tmp/n$$  # unique temp file names
old=/tmp/o$$
who >$old  # initialize old one

while true
do
   who >$new
   diff $old $new
   mv $new $old
   sleep 60
done | awk '$1 == ">" { $1 = "in:    "; print}
            $1 == "<" { $1 = "out:  "; print}'
```

The first two lines name temporary files that will not collide with others that might already be in use. The program **diff** compares two files and prints a list of differences; for example, the output of **who** is now

```
bwk     tty4    Jun 5 19:44
jlb     tty5    Jun 5 21:46
```

and the difference between this and the previous **who** output, as reported by **diff**, is

```
2c2
< cvw    tty6    Jun 5 18:21
---
> jlb    tty5    Jun 5 21:46
```

The **awk** program simply reformats and prints these lines every time that **diff** identifies a change. (Naturally, all of these examples of program output were created by redirecting the output of the commands to files.)

Again, the important point is not the specific details, but the ease with which new, useful and nontrivial programs can be created out of the programs that already exist. The alternatives of doing things by hand or writing conventional programs seem equally unpalatable. I believe that Unix is unique in the degree to which it encourages this kind of productive reuse of programs.

One important aspect of software reusability is packaging sophisticated algorithms and good engineering to make them useful to less sophisticated users. The program **diff** mentioned above is a good example: it relies on a very careful implementation of a nonobvious algorithm. Once done properly, however, it is available for a variety of other uses. Two examples illustrate the point. The program **diffmark** uses **diff** output to add formatting commands to a document so that change bars will be printed in the margins to identify revised parts of the document. The Source Code Control System [7] uses **diff** to create a list of differences so it can keep a space-efficient history of versions of source programs.

Another example of reusing technology is **yacc**, a compiler-compiler developed by S. C. Johnson [8]. It provides a fast LR(1) parser-generator integrated with C in a form that is eminently usable by people who don't even know what LR(1) means.

Regular expressions for string matching is another technology that permeates Unix. The command interpreter uses regular expressions to specify patterns of filenames. Programs like the editor, **grep**, **sed**, **awk**, and others provide string matching based on regular expressions. And the program **lex** does for regular expression grammars what **yacc** does for context-free grammars—it provides a recognizer and a framework upon which code can be placed. All of these programs share a basic technique—the use of regular expressions to specify strings of characters. Many of them share code as well.

System Level

It was mentioned above that the Unix operating system and most of its applications programs are written in C. As a result, the Unix system itself can be moved (though not entirely without change) from one computer to another. In 1977 S. C. Johnson and D. M. Ritchie moved Unix from the PDP-11 to the Interdata 8-32 [9]; in an independent effort, R. Miller moved the same system to an Interdata 7-32 [10]. Since then, the system has been moved to a wide variety of machines. Of these, the DEC VAX family is probably the most popular, but Unix systems run on machines as small as the Motorola 68000 and as large as IBM and Amdahl mainframes. Roughly speaking, porting Unix takes a year of effort. A C compiler is needed for the new machine, although only about 25 percent of the 8000-line compiler is machine-dependent [11]. The operating system itself has about 1000 lines of assembly language that have to be changed; the remaining 9000 lines of C remain essentially the same. Device drivers have to written but they are written in C, not assembler. And a number of other programs that truly depend on the particular machine (for example, the assembler) have to be written or revised.

Once this effort has been made, the rewards are significant. For most users, the system on different machines is the same—they would be hard-pressed to identify any difference that matters. Users are finally free of the trauma of moving from one operating system and set of programs to another when the hardware changes. Instead, they can carry their experience and their code from one piece of hardware to the next, without having to relearn anything. It seems likely that a major component of the current success of Unix is its portability.

Most applications programs are the same down to the source code; this is true even if they don't use the standard libraries, since the systems are comparable at the level of the system calls. To tell the whole truth, however, it should be noted that there are problem areas that affect a few programs. The most serious, at least from my experience, is that the terminal interface (technically, the **ioctl** system call) is significantly different in the major variants of Unix. This means that any program that wishes to directly control terminal speed, echoing of characters, etc., may have to contain conditionally-compiled code to cope with different systems. This is a sorry state of affairs.

One relatively recent development in Unix systems, which seems to be occurring simultaneously in many places, is the connection of Unix systems into networks that to some degree efface the differences between the machines on the network. Two examples should suffice. The Newcastle connection [12] is a set of Unix machines (primarily PDP-11's) connected by a Cambridge ring network. The novel aspect is that these systems in effect share a file system that spreads across all machines in the network. The resulting distributed system is functionally indistinguishable from a conventional single-processor Unix system. The implementation is quite simple in concept. A set of routines is interpolated between user programs and the system calls. These routines detect references to remote resources, map them into remote accesses to counterpart processes on the other machines, then map the results back into local terms.

Another example is the Datakit network in our laboratory at Bell Labs, Murray Hill. This network connects a dozen VAX's and a handful of smaller machines. Each machine has a name; that name is also the name of a program that will run programs on the machine. So, for example, to run who on the machine called **alice** (machine name etymologies are irrelevant), one says

```
alice who
```

The standard output from alice is returned to the local machine, where it may be piped to another machine like this:

```
alice who | rabbit lc
```

Without arguments, the name of the machine ("alice") causes an invocation of the shell on that machine.

We have made extensive use of these capabilities to hide details of where common resources are actually kept. For example, one machine reads the Associated Press news wire continuously, but the command **apnews** can be run from any machine; it accesses the proper program on the proper machine without the user having to know or care where the actual phone line is. Of course the implementation uses the capabilities of the shell discussed in the previous section.

In addition to these local networks, there is a huge network of Unix systems loosely connected by telephone, sharing mail, news, programs, and insights into how to get things done. This network would be far less effective if it were not based on shared programs and a common base of knowledge.

Conceptual Level

In the long run, the greatest impact and the greatest "reuse" of software comes from building on what has been learned, so that one is not forced to relearn things that should already be known. Unix provides a variety of lessons that we can build on for the future.

Some of these are specific lessons about the design, implementation and organization of systems: the importance of a clean simple file system; the notions of pipes and I/O redirection; the value of a programmable command interpreter. These are not all new with Unix, of course, but they seem to have been combined in it most effectively.

Some lessons are more general. For example, it is now amply demonstrated that high-level languages are the only sensible way to implement systems. It is also evident that systems do not have to be big to be useful; indeed, quite the contrary is often true.

Finally, the experience with Unix shows that with a good operating system and a good set of tools, it is possible to make extensive reuse of programs and technology, instead of starting over from nothing with each new task.

References

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall 1978.

[2] W. N. Joy *et al.*, *curses(3), termcap(5), Unix Programmer's Manual*, 4.1BSD June 1981.

[3] B. Stroustrup, "Adding classes to the C language: an exercise in language extension," *Software–Practice and Experience*, vol. 13, Feb. 1983.

[4] B. W. Kernighan and J. R. Mashey, "The Unix programming environment," *IEEE Comput. Magazine*, vol. 14, pp. 12-24, Apr. 1981.

[5] B. W. Kernighan and R. Pike, *The Unix Programming Environment*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[6] S. R. Bourne, "The Unix shell," *Bell Syst. Tech. J.*, vol. 57, pp. 1971-1990, July 1978.

[7] M. J. Rochkind, "The source code control system," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 364-370, Dec. 1975.

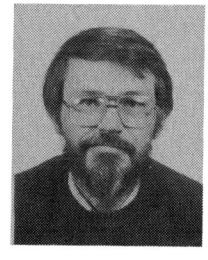
[8] S. C. Johnson, "Yacc–Yet another compiler-compiler," *UNIX Programmer's Manual*, Bell Laboratories, Murray Hill, NJ, Sec. 19, Jan. 1979.

[9] S. C. Johnson and D. M. Ritchie, "Portability of C programs and the Unix system," *Bell Syst. Tech. J.*, vol. 57, pp. 2021-2048, July, 1978.

[10] R. Miller, "UNIX–A portable operating system?" *Oper. Sys. Rev.*, vol. 12, pp. 32-37, July 1978.

[11] S. C. Johnson, "A portable compiler: Theory and practice," *Proc. 5th ACM Symp. Principles of Programming Languages*, Jan. 1978, pp. 97-104.

[12] D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle connection," *Software–Practice and Experience*, vol. 12, pp. 1147-1162, Dec. 1982.



Brian W. Kernighan (S'62-M'64-M'72) received the Ph.D. degree in electrical engineering and computer science from Princeton University, Princeton, NJ.

He has been with the Computing Science Research Center at Bell Laboratories, Murray Hill, NJ, since 1969. His current research interests include document preparation, programming languages, and software tools. He is the coauthor of several books, including *Software Tools*, *The C Programming Language*, and *The Unix Programming Environment*.

Dr. Kernighan is a member of the Association for Computing Machinery. He is on the editorial advisory boards of *Software–Practice and Experience* and *The Bell System Technical Journal*.