Can’t Happen
or
/* NOTREACHED */
or
Real Programs Dump Core*

© 1984, 1985

Ian Darwin

SoftQuad, Inc.,
339 Bloor Street West, # 217,
Toronto, Ontario, Canada
M5S 1W7
utzoo!sq!ian
ian@sq.com

Geoff Collyer

University of Toronto Computing Services
255 Huron Street – UTCS
Toronto, Ontario, Canada
M5S 1A1
geoff@stat.toronto.edu (utzoo!utstat!geoff)

ABSTRACT

UNIX† programmers too often fail to check for failure of system calls or functions, taking the familiar teen-age attitude that ‘‘it can’t happen to me (or my program)’’. This paper will attempt to convince its audience to take prophylactic measures. Those who take such measures will be healthier – and less prone to surprises – than those who don’t take such measures.

In the tradition of the classic *Elements of Programming Style* some real-world programs will be criticised publicly. Actual production (or *soi-disant* production) UNIX programs and subsystems will be examined. Each of these ‘‘provides one or more lessons in style.’’ We present both before and after versions of most code fragments.

Come on out and see if we abuse one of your programs – or your programming style!

Introduction

Many people think that errors can only happen to ‘‘the other guy.’’ Unfortunately, current versions of UNIX do not provide a system call to tell if you’re running as the other guy... But I hear that Dennis is working on it!

Why is net.bugs such an active newgroup? Why is the USENIX 84.2 tape filled with nothing but 4.2 bugs and fixes? Why does *each* release of System N claim to *fix* hundreds of new bugs? They never, of course, claim to *introduce* new bugs. Clearly something is rotten in the state of software. One thing that

---

* Paper delivered at the Dallas USENIX Conference, January 21 1985
† UNIX is a trademark of Bell Laboratories.

we see wrong is the attitude that ‘‘errors won’t happen to me, so I don’t need to check for them.’’

So by now you’ve gathered that our paper is about UNIX *programming style.* And I can already hear you asking: Who cares about style? Who has *time*? Why should I care? I’ll tell you why. Because you – there in the fifteenth row – your database is being corrupted by null pointers right this moment. And you – in the three-piece suit at the back – your kernel will die a horrible death during your next vacation, and you’ll have to come back early to fix it. And you people milling about near the exit so you can sneak out if this is one of those dull papers – you won’t find out what awaits your code unless you stay for the whole talk!

Let me tell you briefly (I promise!) A Tale of Two Systems, the UNIX and the Emveeous. ‘‘There dwelt in the land of New Jersey the UNIX, a fair maid whom savants travelled far to admire.’’ The Emveeous was envious of the UNIX for her natural grace; the UNIX however, spurned the Emveeous, thinking him a course, vulgar fellow. One day her suspicions were confirmed when she saw that he had more manuals listing his errors and sins than she had manuals describing her entire life. But as the UNIX grew into middle age and got busy, she became careless, and made many mistakes, and forgot to check these errors. And the scribes duly observed these errors, and duly recorded them. And the UNIX died old and unhappy, for she saw in her final hour that her error messages manual had grown malignantly to become large, larger than that of that old simpleton, the Emveeous.

Is there a moral in this sad tale? If there is, I believe it is this: If you want a system that forces you to do everything its way, that handholds and spoonfeeds you, that spends a third of its resources checking for errors you might have made, that spews myriad messages on your terminal at random intervals, that sings you a sad song when you leave out a comma, in short, ‘‘If you want MVS, you know where to find it.’’

Our plan is to present some guidelines for safe, surprise-free programming. Taking after *The Elements of Programming Style*, we’ll present real-world examples, suggest improvements and draw conclusions. The examples are drawn from our own experience in maintaining code on the dozen UNIX systems we currently maintain on four different computer architectures. They come from several UNIXes to which we have access (some source and some binary), and from public domain code. Many of these real-world examples have been in use for years, which shows that it’s possible for latent bugs to go undetected for long periods of time. Scratch any large production project and you’ll find latent bugs; we picked these examples because they were at hand.

The remainder of the paper consists of sections on planning (‘The art of thinking’), read the manual (‘rtfm’), not reinventing the wheel, Style, and coding blunders. This paper is concerned with the C programming language; discussion of programming style in *awk*, *lex/yacc*, and in *sh* is left for a future project.

We hope that those whose code we have criticised will take it as constructive criticism rather than as personal criticism. Our aim is to improve, not to insult.

1. Examples

Pontification without proof is pointless. Here are some examples of ineffective code and how it can be improved.

1.1. The art of thinking

The art of thinking (before you code) often seems a lost art. Such safeguards as validating the input before you read it and keeping the user interface constant from one command to the next, are good things whose time has not (we hope) truly passed.

1.1.1. Check the input?

There are times when it’s easy to check the input for certain obvious errors. Many programs now check that their input file is not a directory; this is probably a good thing. Much work remains to be done in the area of input validation. Here’s a simple example from a binary-only system (it happens to be UniSoft System V, but most any UNIX will exhibit this behaviour):

```
$ tc /tmp file1
t8d{ @T@t@T@T@mm@lg@7˜@Ho@M@L@L@H@la@6v@h@‘@L@N@V@H@s5r@g@4y@q@G@M@M@R@z
c@3d˜@u@f@M@L@L@H@n2x@m@1@w@I@I@C@Hj@0{@I@P@H@P@;;lu@Nf@H@I@I@L@
        H@;;da@M/r@H
@I@I@L@Hd@.u@lk@9mk?RfRGRGRIRM7l @;
$
```

On a Tek terminal, the results would be less spectacular but no less erroneous. Consider another example:

```
$ tc pascalprog.p
```

The results would be similar to that shown.

The only valid input to this program is a file created by the old (non-di) *troff*. These files invariably start with an initialise command, which has the octal value 0100.

Note also that the second argument is ignored with no error message. The program behaves as if it ‘thinks’ that all is well, but produces voluminous trash, in the presence of a single typographical error.

How it might have looked:

```
$ tc /tmp file1
tc: /tmp: not troff output
tc: file1: cannot open (no such file or directory)
$
```

How little work it would be to check for this error, and how much more pleasant it would make life, is something to cogitate on.

Test input for plausibility and validity

1.1.2. Directories are fun

Directories can be a lot of fun when you read them into a program which expects a data file. Many places check, but many more do not. Here is how 4.2 Berkmail works.

```
$ Mail -f /usr/spool/mail
"/usr/spool/mail": 0 messages [Read only]
& h
No applicable messages
& x
$
```

How it might behave:

```
$ Mail -f /usr/spool/mail
Mail: "/usr/spool/mail" is a directory!
$
```

1.1.3. Don’t change the interface

USG systems (PWB, System III, System V) come with *labelit* and its brethren.

```
$ labelit /dev/rgmc0a gmc0a root
Current fsname: ROOT, Current volname: gmc0a, \
        Blocks: 13566, Inodes: 1904
FS Units: 1Kb, Date last mounted: Thu Jan 3 20:09:25
NEW fsname = gmc0a, NEW volname\
         = root -- DEL if wrong!! <type DEL>

$
```

There are a couple of problems with the example above: a strange interface, and no feedback where feedback is called for.

There is something quite backwards about this program's behaviour. The program tells you what it's going to do, says 'DEL if wrong!!', waits 10 seconds, then goes ahead and does it!

Everything else in UNIXdom either assumes that you know what you're doing, or asks with some user-friendly prompt like

```
last chance before scribbling on /dev/....
```

What logic can there be for the decision to make this program use a whole new method of interaction? The rationale may be that USG systems are designed for large DP shops, with COBOL and Operators, and that Operators are somehow a lower class of human than normal UNIX users. I don't think many operators would like this line of reasoning.

Scenario: you are converting from v7 to System III. You've just typed a command

```
volcopy /dev/hp1b /dev/hp0c
```

that will mistakenly copy the distribution over top of what you've been working on all night, instead of vice versa. The phone rings or, worse, your manager walks in and just *has* to talk right this instant.

> ''Tom, I need to talk to you about those...''
> ''Not now, boss, this is important!''
> ''It'll only take a few seconds''

It did - about ten seconds, in fact. Any other UNIX utility would wait until you get off the phone. Volcopy, however, will wait ten seconds. Well, I hope you got it right.

Just as a person who holds another at gunpoint assumes full moral responsibility for the actions of his victim, the programmer who *forces* the user to interact with the program (as opposed to typing a command and having it done) takes on responsibility for the user's actions. The least this program could do, having given me three lines of dull, boring information would be to give me some important information, like whether it went ahead and did the change or not! The system I was on was quite busy, and several seconds went by before I typed the DEL and several seconds more before the prompt came back. Was the change done? Quite honestly, at the time of writing, I do not know. Glad it was a labeling job, not a disk-to-disk volume copy!

What does the program do if I have my INTR key set to CTRL/C and my erase key is DEL? Many Operators in UNIX shops are cross-trained to VMS, where the *stty* command is implemented globally by patching the system image. Should this be taken into account?

Here's how it could have been:

```
$ labelit /dev/rgmc0a gmc0a root
Current fsname: ROOT, Current volname: gmc0a, \
        Blocks: 13566, Inodes: 1904
About to change fsname to gmc0a, volname to\
        root - type a 'y' to continue:y
$
```

In this case no confirmation is necessary; if I type y it will do it, if not, not. This is repeatable and predictable, so no feedback is needed, although it would not be out of line (given the importance of the operation) to printf ‘‘done’’ after the write.

The conventional UNIX interface is widely used and understood. The next major interface will probably be something like what the Blit (I’m sorry, the 5620 DMD) provides. Let’s not go half-way in the interim.

Use a consistent dialogue.

1.2. Read the manual (rtfm)

The UNIX manual set is not yet delivered in a moving van (although I hear a group in ATT is working on it), so there’s really no excuse for writing reams of C code before you’ve read most of the manual set. But people do it.

1.2.1. Signals – to catch or not to catch

Some UNIX programmers have still not read *UNIX Programming* by Kernighan and Ritchie in Volume 2 of the UNIX Programmer’s Manual. Their programs catch signals such as interrupt and quit like this:

```
#include <signal.h>
extern int onintr();

signal(SIGINT, onintr);
```

So you write the program, and test it once, and test it a second time hitting your INTR key, and it does the right thing. So you immediately declare it ‘in production’, and post it to net.sources. And 5,000 people save copies of it − 200 of them on our machines alone − thank you very much! And a few of the 5,000 eventually get around to looking at what they’ve saved in their *src* subtrees, and a few of these actually compile the program. And it seems to do the right thing. And then one bright sunny day they background it, and then interrupt a

*rm **

that they accidentally typed afterwards. And your program wakes up and says ‘‘Hello, I’m Fred. You hit interrupt. What do you want to do now?’’ and confuses the heck out of someone, who doesn’t know what his erroneous remove has done. Or worse, it’s a long-running program, and they background it with the output piped to *lpr*, and after they interrupt the faulty *rm* command and an hour later find they got no output, they come to me and say ‘‘Your line printer spooler is busted.’’ and I waste half an hour tracking it down. Again, thank you!

It’s not really that hard to catch signals correctly, although it does add an extra 001 line(s) of code. Here’s a sample:

```
#include <signal.h>
extern int onintr();

if (signal(SIGINT, SIG_IGN) != SIG_IGN) /* iff not ignoring interrupts */
        (void) signal(SIGINT, onintr); /* then catch them */
```

Programs which catch keyboard signals even in the background can make their users wary of ever typing interrupt or quit (or hanging up their connection), since the users aren’t sure whether or not some background program will spring to life and erroneously catch the resulting signal. This is arguably a design flaw in the UNIX signal mechanism. Perhaps UNIX should ignore attempts to catch signals which were being ignored when the current program was *exec* ed. This would make the use of backgrounding with **&**

and *nohup* fool-proof. However, UNIX is what it is.

Don't blindly catch signals

A Digression on Lint

The previous example illustrates one of many common errors which can be easily caught if you are developing code on a real UNIX system. The *lint* program contains much of the debugging code that is left out of the C compiler, although there is a tendency in newer compilers to reinsert some of this checking. We strongly advise that all programs be checked for lint, and that you check the output carefully. It's well-nigh impossible to silence *lint* about some few functions (such as *malloc()*) but in general *lint* will give you good advise.

Lint is a scroll of forgiveness
for many sins of programming.
Read it wisely, and you will prosper.
Fail to read it and you will
hear maniacal laughter.

Thou shalt use lint.

1.2.2. System Calls

In what program do we find the following code sequence:

```
open("/", 0);
dup(0);
dup(0);
```

V7 *init*, alas. There are two other occurrences, in which the opens are ''open(tty, 2);'' and ''open(ctty, 2);''. The author *knew*, by god, that those system calls could *never* fail. As a result, when the kernel file or inode tables fill, *init* fails to re-populate some terminals with *init* children and thus *getty*'s. Thus those terminals will never inherit *init*'s until the next crash or reboot. We know: the file and inode tables aren't supposed to fill, thus it **can't happen**. A common counter-argument is that in such a case there is nothing sensible to be done. Yet a moment's thought often reveals a better alternative than failing to check. In this case, since the code in question is running in a child of *init*, *init* can sleep briefly and try again if a system call such as *dup* fails or if an *open* fails due to resource exhaustion.

1.2.3. /dev/kmem - open sewer or open sore?

A common disease in programs written at Berkeley is to open */dev/kmem* and grub the load averages out by the dirtiest means possible. The following is from the 4.2BSD *sendmail* source, conf.c, slightly reformatted for brevity.

```
#include <nlist.h>
struct nlist Nl[] = {
#define     X_AVENRUN     0
        { "_avenrun" },     { 0 },
};

getla()
{
        static int kmem = -1;
        double avenrun[3];

        /*
         * kmem opened here and nlist
         * called for /vmunix with Nl
         */
        (void) lseek(kmem, (long) Nl[X_AVENRUN].n_value, 0);
        (void) read(kmem, avenrun, sizeof(avenrun));
        return ((int) (avenrun[0] + 0.5));
}
```

(void) read should be avoided. The running kernel may not be /vmunix and the read may not return as many bytes as expected, leaving trash in avenrun.

```
/* same declarations */

getla()
{
        static int kmem = -1;
        double avenrun[3];
        extern off_t lseek();

        /*
         * kmem opened here and nlist
         * called for /vmunix with Nl
         */
        if (lseek(kmem, (long) Nl[X_AVENRUN].n_value, 0) < 0 ||
            read(kmem, avenrun, sizeof avenrun) != sizeof avenrun)
                return -1;              /* can't be a valid load average */
        return ((int) (avenrun[0] + 0.5));
}
```

Data that you ‘‘know’’ are there, won’t be.

Assume that system calls will fail capriciously.

Once this routine is correct, it should be put in a system library so that programmers will not keep reinventing it. Apparently it was in 2.9BSD but not in 4.2BSD; perhaps this will be included in the next release of 4.2.

1.2.4. Files will always open and they never need to be closed

The USENET ‘B’ News system in its present (late 1984) state is a fruitful source of examples for a paper on our topic. Here is one small example: inews.c often fails to check that fopen succeeded and fails to close open FILEs.

```
        actfp = fopen(ACTIVE, "r+");
        for(;;) {
                fpos = ftell(actfp);
                if (fgets(afline, sizeof afline, actfp) == NULL) {
                        unlock();
                        return FALSE;           /* No such newsgroup locally */
                }
        ...
        fclose(actfp);
```

Stdio tends to be unamused when handed null pointers, as it would be if ACTIVE could not be opened for reading and writing for any number of reasons: no such file, no permission, full i-node table, full file table, etc. If fgets encounters end of file, this code will return, leaving ACTIVE open.

```
        actfp = fopen(ACTIVE, "r+");
        if (actfp == NULL)
                xerror("Cannot update %s\n", ACTIVE);
        for(;;) {
                fpos = ftell(actfp);
                if (fgets(afline, sizeof afline, actfp) == NULL) {
                        unlock();
                        fclose(actfp);
                        return FALSE;           /* No such newsgroup locally */
                }
        ...
        fclose(actfp);
```

Files will sometimes be unopenable for reasons you don't foresee when coding; it is as well to be prepared for such possibilities, however unlikely you (erroneously) believe them to be.

Don't assume God likes you: open and fopen will fail.

1.2.5. System calls never fail in my programs

The ultimate arrogance is to assume that the system calls and functions invoked by one's program can never fail. The following is from 4.2BSD, **/usr/src/lib/libc/net/ruserpass.c**. This module is a fruitful source of examples, so we shall return to it later.

```
ruserpass(host, aname, apass)
        char *host, **aname, **apass;
{

        renv(host, aname, apass);
        if (*aname == 0 || *apass == 0)
                rnetrc(host, aname, apass);
        if (*aname == 0) {
                char *myname = getlogin();
                *aname = malloc(16);
                printf("Name (%s:%s): ", host, myname);
                fflush(stdout);
                if (read(2, *aname, 16) <= 0)
                ...
```

This code knows that *getlogin* and *malloc* can never, ever fail. Unfortunately it is wrong. Redirecting all three standard file descriptors away from any terminal (e.g. as *nohup* (1) does) will make *getlogin* fail consistently. When this happens, *ruserpass* will cheerfully dereference *myname*, which is now a null pointer, possibly causing a core dump.

Malloc seldom fails on working programs in 4.2BSD on a VAX. As a result, programmers develop the nasty habit of failing to check for *malloc* failing. In addition to making their own debugging harder (since *malloc* can fail when you are debugging a program, even on 4.2), this causes endless grief for those using non-paging UNIXes.

All the world's NOT a VAX.

```
ruserpass(host, aname, apass)
char *host, **aname, **apass;
{
        renv(host, aname, apass);
        if (*aname == 0 || *apass == 0)
                rnetrc(host, aname, apass);
        if (*aname == 0) {
                char *myname = getlogin();

                if (myname == NULL)
                        myname = "unknown";
                *aname = malloc(16);
                if (*aname == NULL)
                        error("can't allocate memory for password",
                                 (char *)NULL);
                printf("Name (%s:%s): ", host, myname);
                fflush(stdout);
                if (read(2, *aname, 16) <= 0)
                ...
```

This version will behave sanely when functions or system calls fail, unlike the original.

Do something sensible when system calls or functions fail

1.3. Don't Reinvent the Wheel

The C libraries contain many, many valuable routines. So does /bin. Why reinvent them? I know not, but people continue to reinvent.

1.3.1. Use existing tools

What does this do?

```
if (freopen("/usr/lib/whatis", "r", stdin) == NULL) {
        perror("/usr/lib/whatis");
        exit (1);
}
gotit = calloc(1, (unsigned) blklen((int *)argv));
while (fgets(buf, sizeof buf, stdin) != NULL)
        for (vp = argv; *vp; vp++)
                if (match(buf, *vp)) {
                        printf("%s", buf);
                        gotit[vp - argv] = 1;
                        for (vp++; *vp; vp++)
                                if (match(buf, *vp))
                                        gotit[vp - argv] = 1;
                        break;
                }
for (vp = argv; *vp; vp++)
        if (gotit[vp - argv] == 0)
                printf("%s: nothing apropriate\n", *vp);
```

What would happen if the *calloc* call failed?

The Berkeley system contains the command *apropos* used to find manual keywords. While its name might more appropriately have been *findman*, the program *knows* that it is to be called as ‘apropos’, because it looks at argv[0]. It is glued into the source for the *man* command. At any rate, the code attempts to reimplement *grep*, and does so in a way that is possibly correct but inarguably slow. Here is our *findman*, which runs about three times as fast as *apropos*.

```
#! /bin/sh

# findman - find manual pages given topic(s)

PATH=/bin:/usr/bin:/usr/ucb ; export PATH
INDEX=/usr/lib/whatis

for f
do
        grep $f $INDEX || echo ‘basename $0‘:\
                 nothing appropriate for $f
done
```

This code could be further speeded up by using *egrep* or *fgrep*, or by using the -y/-i option for case insensitivity, but our change will work as shown on almost any conceivable UNIX system.

There has been some debate on USENET recently (and three months ago, and six months ago, and ...) about replacing C programs with shell files and *vice versa*. Our criteria state that a program must do one function that is unique, and do it well, to be a C program. When you reinvent a shell file to be a C program, you lose the benefit of years of tuning (‘hacking’?) which has gone into the underlying tool, in this case *grep*. You also lose the generality of the well-formed tool, in this case the ability to grep for regular expressions, do case-insensitive searches, etc.

Don’t reinvent the flat tire

This is also a technical objection to the (probably inevitable) process of ‘unbundling’ UNIX and the rise of the *cat(1) Reference Manual*. When the standard UNIX tools become options, as they will in the next few years due to marketing pressures to conform to the *lowest* common denominator (and because of all those people who tried to buy UNIX systems with 10MB hard-floppy winchesters), then the use of the standard tools may become a lost art in new programmers.

1.3.2. Parsing Program Arguments (argv scanning)

What does this do on your 4.2 system?

/bin/mail -r

There are billions of different ways of parsing *argv*, the list of command line arguments passed to a C program. The problem is that they are all different. The USG long ago recognised this as a serious problem, and implemented *getopt*(3) to handle the problem. Several public domain versions of this routine have been posted to *net.sources*. At the Dallas UniForum in January, 1985, AT&T *published* the source for the latest version of System V *getopt*(1) and *getopt*(3) in hopes that people will use these functions. The library routine *getopt* really should be in every C library on every UNIX system in the world. If it's not on your system, add it. And use it.

Use of *getopt*(3) is a standard at our installation. We have in a file called */usr/pub/template.c* the skeleton of a complete C program with all the argument checking and basic declarations already built in. Copies of this file will be posted to net.sources or may be had by electronic mail request to either author.

People are always trying to build more complicated 'argv crackers'. To our mind, *getopt*(3) has proven satisfactory in the construction of dozens of small programs and the reshaping of dozens of others. *Getopt*(3) isn't IBM TSO's IKJPARSE, but then again, this is UNIX.

Here is my (Darwin's) current version of */usr/pub/template.c*, which I hack from with an editor to create almost any new C program or to clean up an existing one. To make a new program, I need only edit all the 'xxx' strings, so I can concentrate on writing a *process()* that gets the job done instead of typing argc and argv. How many times have you typed 'main(argc, argv)' in your life?

```
/*
 * name - purpose xxx
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#define	MAXSTR	500

int	debug;
char	*progname;
struct stat statbuf;
void	error(), exit();

/*
 * main - parse arguments and handle options
 */
main(argc, argv)
int argc;
char *argv[];
{
	int c;
	int errflg = 0;
	FILE *in;
	extern int optind;
	extern char *optarg;
	extern FILE *efopen();

	progname = argv[0];

	while ((c = getopt(argc, argv, "dxxx")) != EOF)
		switch (c) {
		case 'xxx':
			xxx
			break;
		case 'd':
			++debug;
			break;
		case '?':
		default:
			errflg++;
			break;
		}
	if (errflg) {
		(void) fprintf(stderr, "usage: %s xxx [file] ...\n", progname);
		exit(2);
	}

	if (optind >= argc)
		process(stdin, "stdin");
	else
		for (; optind < argc; optind++)
			if (strcmp(argv[optind], "-") == 0)
```

```
				process(stdin, "-");
			else {
				in = efopen(argv[optind], "r");
				if (fstat(fileno(in), &statbuf) != 0)
					error("can't fstat %s", argv[optind]);
				if (statbuf.st_mode & S_IFDIR)
					error("%s is directory!", argv[optind]);
				process(in, argv[optind]);
				(void) fclose(in);
			}
	exit(0);
}

/*
 * process - process input file
 */
process(in, inname)
FILE *in;
char *inname;
{
	xxx
}
```

Scan command line arguments in a standard way

1.3.3. csh - the self-contained shell

The Berkeley shell *csh* is reasonably well known and widely used and its externals have been subject to some criticism. The internals of *csh* are at least as bad, but lesser known.

Csh contains its own versions of *malloc* and *_doprnt*. It is thus unable to use *stdio* or some other functions in the C library. When we attempted to link *csh* with a faster version of *getpwent* that uses *stdio* and an *mdbm* data base for the password file, we found that *csh* failed mysteriously.

Our fix was to make *csh* use the old, slow *getpwent*. Such is the price of bad coding. (Neither of us uses *csh*, so we don't much care about its performance.)

The reason that *csh* was written this way is that it contains so many features that it wouldn't fit on PDP-11 UNIXes unless the C library was hacked out. In our view, a better approach would have been to eliminate a few features – this shell has enough bells and whistles that one or two could profitably be dropped.

Don't provide private versions of C library functions unless they are declared *static* (i.e. invisible to other object files).

Don't fight the C library; use it.

1.4. C Style and Portability

Coding style is in some ways a matter of personal preference. But there are some things that just don't work, or which are indicative of underlying sloppiness, or are just poor practice. Portability at a working level means the refusal to include code that is bound up in the shape of the particular digital computer we happen to be running on.

1.4.1. Magic numbers

Using a literal numeric constant in-line is almost always a bad idea, since the reader of such code is given little clue how the number was derived or what it represents. We return to 4.2BSD's **ruserpass**.

```
ruserpass(host, aname, apass)
	char *host, **aname, **apass;
{

	renv(host, aname, apass);
	if (*aname == 0 || *apass == 0)
		rnetrc(host, aname, apass);
	if (*aname == 0) {
		char *myname = getlogin();
		*aname = malloc(16);
		printf("Name (%s:%s): ", host, myname);
		fflush(stdout);
		if (read(2, *aname, 16) <= 0)
		...
```

Note the magic numbers: 0, 2 and 16. The first three zeroes are null pointers. The last zero is used to test for an error while reading the password *or* end of file (a zero count). The 2 is the file descriptor of the standard error output. 16 is the (arbitrary) maximum length of a password, including the null byte at the end.

```
#define MAXPWLEN 15
#define STDERR 2
ruserpass(host, aname, apass)
char *host, **aname, **apass;
{
	renv(host, aname, apass);
	if (*aname == NULL || *apass == NULL)
		rnetrc(host, aname, apass);
	if (*aname == NULL) {
		char *myname = getlogin();

		*aname = malloc(MAXPWLEN+1); /* 1 is for\
			the null byte */
		printf("Name (%s:%s): ", host, myname);
		fflush(stdout);
		if (read(STDERR, *aname, MAXPWLEN) <= 0) /* \
			error or EOF */
		...
```

These may seem trivial, but in larger programs it isn't always obvious that the many occurrences of some magic number are (or are not) related. #define'ing these numbers also makes changing them fairly easy.

Use #define to give numbers explanatory names.

On the other hand, grabbing a random #define'd symbol that happens to have the right value, and using it, is no better. In particular, *stdio.h*'s BUFSIZ is often used incorrectly to mean ''a bunch of characters''. This practice originates on Version 7, where BUFSIZ was 512, a convenient size for holding strings, names, etc. But this is unportable, since some *stdio* implementation might use 48 as its BUFSIZ.

Better to use a name such as MAXSTR for maximum string length, and use your own criteria for what's reasonable.

As an exercise, consider what the value for MAXSTR should be. Is 100 enough? 256? 512? How can you prevent overrunning strings, no matter how big you make them? Remember that users are creative, and somebody will find a way to try to exceed whatever you specify. Remember that some machines still have limited memory, so you can't make your strings 5120 bytes each. Think about *strncmp* (3); is its design useful here?

1.4.2. _doprnt considered unportable

Berkeley in 4.2BSD documented the **internal** *stdio* interface *_doprnt*. This was a mistake. Other *stdio* implementations often do not contain a *_doprnt* or anything like it. The Berkeley *curses* has used *_doprnt* for a long time, even before 4.2BSD:

```
/*
 * This routine actually executes the printf and adds it to the window
 *
 * This is really a modified version of "sprintf".  As such,
 * it assumes that sprintf interfaces with the other printf functions
 * in a certain way.  If this is not how your system works, you
 * will have to modify this routine to use the interface that your
 * "sprintf" uses.
 */
_sprintw(win, fmt, args)
WINDOW   *win;
char     *fmt;
int      *args; {

         FILE  junk;
         char  buf[512];

         junk._flag = _IOWRT + _IOSTRG;
         junk._ptr = buf;
         junk._cnt = 32767;
         _doprnt(fmt, args, &junk);
         putc('\0', &junk);
         return waddstr(win, buf);
}
```

This is coded as if it were internal to *stdio*, yet it is not. The design error is in offering to provide an interface with a variable number of arguments, something which may be unimplementable on some machines and which on others is only expressed portably using *<varargs.h>*. A better design would have been to take a single string as an argument rather than a *printf* format and arguments. The caller should format the arguments into a character buffer first using *sprintf* and pass the address of the buffer to this function. System V Release 2 includes *vsprintf()* in its stdio, but this is far from standard yet.

> No after version is offered,
> since the design is fatally flawed.

Don't assume that you can write functions that take a variable number of arguments. If you must do so, use *<varargs.h>*.

Avoid variable number of arguments in functions.

1.4.3. Nested arguments vs checking

It's all too common to see a line of nested function calls. The main problem is that it discourages proper checking of system function return calls. This code, taken from net.sources in early 1985, is typical:

```
/* Blast into a users terminal.  Great fun, \
        and sometimes useful. */

#include<sgtty.h>
#include<stdio.h>
#include<sys/file.h>

int errno;

main(argc,argv)
int argc;
char *argv[];
{
        errno = 0;
        if (argc != 2) fprintf(stderr,"blast: need\
                tty number (only).\n");
        else blast(open(argv[1],O_RDWR,0666));
}

blast(fd)
register FILE *fd;
{
        char c;

        if (errno) return;

        ioctl(fd,TIOCNXCL,0);          /* turn off exclusive use */

        while ( (c=getchar()) != EOF) ioctl(fd,TIOCSTI,&c);
}
```

Notice that the error message ('need tty number only') and the open call do not agree; the open seems to want /dev/ttyNN. But the open itself is not checked. Or is it? Notice the obscure line 'errno = 0'; the subprogram checks this and returns silently if errno !=0. Programs that exit silently on error conditions can be frustrating to use, and can almost always be improved upon with little work.

But that's not all. Note the declarations in the function. The variable 'fd' is returned by open(2), so it's a file descriptor (int). But it's declared in the subprogram as FILE *, a **stdio** stream pointer. But it's *used* in the subprogram as a file descriptor again.

Here's how some of the code might look:

```
/*
 * blast - blast text into user's terminal input buffer (4.2 only)
 */

#include <sgtty.h>
#include <stdio.h>
#include <sys/file.h>

void error();

main(argc,argv)
int	argc;
char	*argv[];
{
	int myfile;

	if (argc != 2) {
		(void) fprintf(stderr,"usage: blast /dev/ttyname\n");
		exit(1);
	}
	if ((myfile = open(argv[1], O_RDWR, 0666)) < 0)
		error("can't open terminal %s", argv[1]);
	else
		blast(myfile);
	close(myfile);
	exit(0);
}
```

There may be other problems in the subprobgram; they are left as an exercise for the reader.

1.5. Coding

Coding errors and omissions are last, but not least, on our list of suggested improvements.

1.5.1. isascii, the forgotten macro

When we moved our PDP-11/70 from PWB 1.0 to Version 7, some of our users had become dependant on the RJE software to access an attached slave processor (an IBM 3033). We had to provide support for this access, so we put up the System III RJE software. Months later we put up *sendmail*, and had a month of intermittent looping sendmails (we spoke to each other about 'harpooning sendwhales' at the time).

Programmers often seem unaware that most of the macros defined in <ctype.h> are only defined for arguments which are ASCII characters. 4.2BSD's **sendmail** sometimes uses these macros without first checking that the characters being tested are legal ASCII characters, via *isascii*. As a result, giving *sendmail* an address containing a character with the 0200 bit set will cause it to loop.

```
	if (isspace(*s))
		s++;
```

This attempts to skip spaces, but will do undefined things if *s isn't an ASCII character, possibly including referencing outside allocated memory, which may produce a core dump.

The System III RJE had a bug which occasionally produced trash in the name field, which it passed to *mail* by the *exec* system call parameter list, and causing the sendmail loops which we observed since sendmail didn't properly validate its input. Clearly a case of USG and Berkeley code attacking each other!

Here's how this code should be done:

```
        if (isascii(*s) && isspace(*s))
                s++;
```

This will stop skipping whitespace upon encountering a non-ASCII character.

Use isascii before other ctype.h macros

Maybe the <ctype.h> macros should do this validation. They probably should, but they don't. If you are writing for UNIX rather than some hypothetical future system, you need to call *isascii()* if you want portable code.

1.5.2. scanf sometimes stops scanning too soon

Programmers sometimes fail to test that scanf scanned as many items as they expected. This leaves the remaining variables pointed to by scanf's arguments containing their previous contents, often trash if the variables are uninitialised. B news 2.10.1 contained such a bug in rfuncs.c, as shown below.

```
        while (fgets(buf, sizeof buf, af)) {
                sscanf(buf, "%s %ld", n, &s);
                if (strcmp(n, ng) == 0) {
```

This code will leave garbage in *n* at end of file and in *s* at end of file or if the second item in *buf* isn't numeric.

```
        while (fgets(buf, sizeof buf, af) != NULL)
                if (sscanf(buf, "%s %ld", n, &s) == 2 &&
                        strcmp(n, ng) == 0) {
```

This will only attempt to use *n* (and later *s*) if *sscanf* actually scanned two items.

Expect scanf to stop scanning inconveniently soon
----

2. ''Code it now, we'll fix it later''

Careful coding takes longer. Like careful flying, on the part of the airline captains who will fly most of you home from this conference. Both take longer, but both give you a warm feeling.

'Later' never comes.

In a 'pressure-cooker' environment there is a strong tendency to 'get the thing out the door' without concern for software quality. I perceive this as a general failing of North American management; there is almost everywhere a pressure to provide the *appearance of productivity* regardless of true costs and long-range effectiveness.

The only answer to this is to fight bottom line with bottom line. If you add up the costs of one programmer-year for each major UNIX shop, to include the time spent porting 'portable' code, you will have a starting point for the real costs. Add in all the in-the-field debugging, including costs of debugging which are transferred to the end-user by shipping undebugged code, you'll be on your way. Don't forget to translate customer debugging time into customer dissatisfaction. My first guess at a conversion factor is

One unhappy customer == Ten lost sales

If you find your management pushing to you to 'code now, fix later', just remind them that 'later' never comes.

> ‘‘Our competitors’ code is done more carefully than mine.
> I guess *their* bottom line extends to the horizon.’’

3. Conclusion

We can’t close without citing two good guys. Despite its plethora of bugs and its *crypt* (1)-output-like configuration file, *sendmail* is careful about returning mail that would otherwise fall on the floor. The Honey Danber version of *uucp* is good in the same way, and may be better coded – we’ll have to see.

A second, smaller-scale winner is the multi-key database (MDBM) posted to the net in mid- to late 1984. We built an entire password database structure on top of this package, and used it in a large student environment (several thousands of students over half a dozen UNIX systems) We ran across one obscure bug, but since the author, Chris Torek, had taken the trouble to check for ‘‘impossible’’ errors, we got the message ‘‘MDBM BUG...’’ instead of a scrambled eggs database. Thanks.

Just to sum up, we’ve presented some guidelines for good programming and shown how they can reduce bugs. Reducing bugs means reducing costs.

Some of these guidelines are so well-known that they are almost truisms; many of them appear in your *fortune* file when you log in. Others are our own invention or are paraphrases of originals.

But guidelines are guidelines, and they are useful only if they are put to work in day-to-day programming. That’s where you come in.

4. Recommended Reading

For a view similar to ours, see Kernighan and Plaugher in *The Elements of Programming Style* (2nd Edition, McGraw-Hill, 1978) and the *Software Tools* books (by the same authors, Addison-Wesley).

For a countervailing view, see some (a lot?) of the code which appears in the USENET newsgroup *net.sources* .

Acknowledgement

Thanks to Bruce Freeman for assisting with some of the examples. Our positions at the University of Toronto have afforded us the opportunity to peruse a tremendous amount of questionable code over the past ‘N’ years.

Thanks to Laura Creighton for reading several revisions of the manuscript with a critical eye.

Program fragments listed herein are copyright © by AT&T, The Regents of the University of California, and other interested parties. Public Domain code is by the USENET News Project, Mike Newton, and others. The story of the UNIX and the Emveeous was inspired by Doug McIlroy’s ‘The UNIX and the Echo’, of which our tale is but a pale echo.

The template file */usr/pub/template.c* was adapted by Henry Spencer from the example in the *getopt* (3) manual page while he was writing the public-domain implementation of *getopt* ; the version presented here has been extensively hacked over, so Henry should not be blamed for its present state.

Some of the ‘mottos’ used are excerpted from *The Elements of Programming Style* (see under ‘Recommended Reading’).

Most of the paper was typed by one or another of the authors. Read ‘we’ for ‘I’, and ‘I’ for ‘we’, throughout.

Mark Horton provided useful feedback on the paper after the presentation at Dallas; some of his suggestions are incorporated into the present version.