# Goodbye World!

## The perils of relying on output streams in C

Jim Meyering
meyering@redhat.com
maintainer of coreutils, gnulib, etc.

# Outline:

- There are many problems with stdout
- they make it hard to write truly robust code
- one problem (the tip of the iceberg) and a 95% solution
- offenders, then and now
- oops, a QoI problem with glibc’s snprintf

# All users of C recognize this program:

```
#include <stdio.h>
int
main ()
{
  printf ("Hello, world!\n");
  return 0;
}
```

Is it robust?

# No.  It fails to diagnose a write error:

```
$ gcc hello.c && ./a.out > /dev/full; echo $?
0
```

# What would it take to make it robust?

If you think a nonzero exit code is enough
of an error indication, then maybe this:

```
#include <stdio.h>
int
main ()
{
  if (printf ("Hello, world!\n") != 14)
    return 1;
  return 0;
}
```

# or even this, if you like brevity:

```
#include <stdio.h>
int
main ()
{
  return ! (printf ("Hello, world!\n") == 14);
}
```

# Unfortunately there are two problems with this:

Most people *do* want diagnostics.  They help distinguish a disk full error from permission denied or EIO.

Testing each and every use of printf, fprintf, fputs, fseek, putc, putchar, etc., is often counter-productive and usually unmaintainable.

# Improvement: detect fclose failure:

Here we call fclose explicitly rather than relying on exit doing it for us, and give a diagnostic upon failure:

```
#include <stdio.h>
#include <stdlib.h>
int
main ()
{
  printf ("Hello, world!\n");

  if (fclose (stdout) != 0)
    {
      perror ("hello: write error");
      return EXIT_FAILURE;
    }
  return EXIT_SUCCESS;
}
```

# Done?

Now we see what we expected:

```
$ ./a.out > /dev/full; echo $?
hello: write error: No space left on device
1
```

We can even provoke a different diagnostic like this:

```
$ ./a.out >&-; echo $?
hello: write error: Bad file descriptor
1
```

The use of ">&-" above tells the shell to run the command with closed stdout.

# Are we done?  No.

For that small example, it might be ok, but in general, no.
First of all, what if there are two or more exit points?
We don’t want to duplicate even that small amount of code, so...

# First, factor out the stream-closing function

```
#include <stdio.h>
#include <stdlib.h>

static void
close_stdout (void)
{
  if (fclose (stdout) != 0)
    {
      perror ("hello: write error");
      exit (EXIT_FAILURE);
    }
}

int
main ()
{
  atexit (close_stdout);
  printf ("Hello, world!\n");
  return EXIT_SUCCESS;
}
```

"atexit" arranges to call the named function at exit time.

# Better still, but we’re not there yet.

Here is a counterexample:

```
$ stdbuf --output=0 ./a.out > /dev/full; echo $?
0
```

Invoking stdbuf like that disables buffering on a.out’s stdout.
That makes printf perform the write syscall, which fails.
Then, when it comes time to close, there is no buffered data,
and so the fclose succeeds.

# ferror saves us, but at a price

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

static void
close_stdout (void)
{
  bool prev_fail = ferror (stdout);
  bool fclose_fail = fclose (stdout);
  if (prev_fail || fclose_fail)
    {
      if (fclose_fail)
        perror ("hello: write error");
      else
        fprintf (stderr, "hello: write error\n");
      exit (EXIT_FAILURE);
    }
}
```

# Using ferror like that comes with a small cost:

In the unusual event that fclose succeeds when ferror returns nonzero, the diagnostic will not include errno information.

# A corner case: closed stdout

There’s still one problem remaining.  What if your program is like touch, mv and cp in that it rarely uses stdout?

Since it doesn’t use stdout for normal operation it should not mind if you run it with stdout already closed. If touch were to use the close_stdout function, it would fail:

- $ touch foo >&-
- touch: write error: Bad file descriptor

While using ">&-" may seem contrived (it *is* often wrong), it is not uncommon for a daemon to run with stdout closed. If that daemon may exec your program, your program should work with closed stdout.

# Another fix: use __fpending

```
void
close_stdout (void)
{
  bool prev_fail = (ferror (stdout) == 0);
  bool none_pending = (__fpending (stdout) == 0);
  bool fclose_fail = (fclose (stdout) == 0);

  if (prev_fail || fclose_fail)
    {
      int e = fclose_fail ? errno : 0;

      if (!prev_fail && none_pending && e == EBADF)
        return;

      error (EXIT_FAILURE, e, "write error");
    }
}
```

The current version of close_stdout handles even more corner cases:

http://git.sv.gnu.org/cgit/gnulib.git/tree/lib/closeout.c
http://git.sv.gnu.org/cgit/gnulib.git/tree/lib/close-stream.c

# TL;DR: fix many programs by adding two lines

Yes, using stdout in code that is supposed to be robust is a pain, but in the vast majority of cases, adding only two lines is enough to solve this problem:

- #include "closeout.h"
- atexit (close_stdout); # call very early in main

Also include <stdlib.h>, for atexit, if it is not already done.

Assuming you are using gnulib, you would also add "closeout" to your list of modules.

# Offenders, (all fixed in 2005):

- perl
  - http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/2004-12/msg00072.html
- python
  - http://mail.python.org/pipermail/python-bugs-list/2004-December/026600.html
- even rsync
  - https://bugzilla.samba.org/show_bug.cgi?id=2116

# Offenders, now:

- emacs --batch --eval ’(print "oops!")’ > /dev/full
- guile --help > /dev/full
- printf ’(display "x")’|guile > /dev/full

# Summary:

Always detect and diagnose write failure. In C, check both ferror and fclose return values. Every program that writes to an output stream should be careful to close it explicitly and to detect and report any error. Simply calling fclose and checking its return value is not always enough: if the program generates output via an unchecked call to a function like printf, fwrite or fputs, then it must call ferror just before fclose to detect a prior failure.

Always check the return value of fclose. Even if all stream output functions and a final fflush have succeeded, fclose may still fail when the output file is on a networked (e.g., NFS) or distributed (e.g., CODA) file system.

When it comes time to close standard output, don't report a failure just because that stream happened to be closed at start-up. Do report the failure if there has been any attempt to write to a closed stream.

Upon failure, always give an accurate diagnostic and exit with nonzero status.

# Thank You

# More problems:

- snprintf (NULL, 0, fmt, 0) may allocate memory

Exercise with the printf command from coreutils 6.9 or earlier:

```
(ulimit -v 10000
 env printf %.20000000f 0)
$ echo $?
$ 0
```