How To Steal Code
or
Inventing The Wheel Only Once

Henry Spencer

Zoology Computer Systems
25 Harbord St.
University of Toronto
Toronto, Ont. M5S 1A1 Canada
{allegra,ihnp4,decvax,utai}!utzoo!henry

ABSTRACT

Much is said about ‘‘standing on other people’s shoulders, not their toes’’, but in fact the wheel is re-invented every day in the Unix/C community. Worse, often it is re-invented badly, with bumps, corners, and cracks. There are ways of avoiding this: some of them bad, some of them good, most of them under-appreciated and under-used.

Introduction

‘‘Everyone knows’’ that that the UNIX/C† community and its programmers are the very paragons of re-use of software. In some ways this is true. Brian Kernighan [1] and others have waxed eloquent about how outstanding UNIX is as an environment for software re-use. Pipes, the shell, and the design of programs as ‘filters’ do much to encourage programmers to build on others’ work rather than starting from scratch. Major applications can be, and often are, written without a line of C. Of course, there are always people who insist on doing everything themselves, often citing ‘efficiency’ as the compelling reason why they can’t possibly build on the work of others (see [2] for some commentary on this). But surely these are the lamentable exceptions, rather than the rule?

Well, in a word, no.

At the level of shell programming, yes, software re-use is widespread in the UNIX/C community. Not quite as widespread or as effective as it might be, but definitely common. When the time comes to write programs in C, however, the situation changes. It took a radical change in directory format to make people use a library to read directories. Many new programs still contain hand-crafted code to analyze their arguments, even though prefabricated help for this has been available for years. C programmers tend to think that ‘‘re-using software’’ means being able to take the source for an existing program and edit it to produce the source for a new one. While that *is* a useful technique, there are better ways.

Why does it matter that re-invention is rampant? Apart from the obvious, that programmers have more work to do, I mean? Well, extra work for the programmers is not exactly an unmixed blessing, even from the programmers’ viewpoint! Time spent re-inventing facilities that are already available is time that is *not* available to improve user interfaces, or to make the program run faster, or to chase down the proverbial Last Bug. Or, to get really picky, to make the code readable and clear so that our successors can *understand* it.

Even more seriously, re-invented wheels are often square. Every time that a line of code is re-typed is a new chance for bugs to be introduced. There will always be the temptation to take shortcuts based on how the code will be used—shortcuts that may turn around and bite the programmer when the program is

† UNIX is a trademark of Bell Laboratories.

modified or used for something unexpected. An inferior algorithm may be used because it's ‘‘good enough’’ and the better algorithms are too difficult to reproduce on the spur of the moment... but the definition of ‘‘good enough’’ may change later. And unless the program is well-commented [here we pause for laughter], the next person who works on it will have to study the code at length to dispel the suspicion that there is some subtle reason for the seeming re-invention. Finally, to quote [2], *if you re-invent the square wheel, you will not benefit when somebody else rounds off the corners*.

In short, re-inventing the wheel ought to be a rare event, occurring only for the most compelling reasons. Using an existing wheel, or improving an existing one, is usually superior in a variety of ways. There is nothing dishonorable about stealing code* to make life easier and better.

Theft via the Editor

UNIX historically has flourished in environments in which full sources for the system are available. This led to the most obvious and crudest way of stealing code: copy the source of an existing program and edit it to do something new.

This approach does have its advantages. By its nature, it is the most flexible method of stealing code. It may be the only viable approach when what is desired is some variant of a complex algorithm that exists only within an existing program; a good example was V7 *dumpdir* (which printed a table of contents of a backup tape), visibly a modified copy of V7 *restor* (the only other program that understood the obscure format of backup tapes). And it certainly is easy.

On the other hand, this approach also has its problems. It creates two subtly-different copies of the same code, which have to be maintained separately. Worse, they often have to be maintained ‘‘separately but simultaneously’’, because the new program inherits all the mistakes of the original. Fixing the same bug repeatedly is so mind-deadening that there is great temptation to fix it in only the program that is actually giving trouble... which means that when the other gives trouble, re-doing the cure must be preceded by re-doing the investigation and diagnosis. Still worse, such non-simultaneous bug fixes cause the variants of the code to diverge steadily. This is also true of improvements and cleanup work.

A program created in this way may also be inferior, in some ways, to one created from scratch. Often there will be vestigial code left over from the program's evolutionary ancestors. Apart from consuming resources (and possibly harboring bugs) without a useful purpose, such vestigial code greatly complicates understanding the new program in isolation.

There is also the possibility that the new program has inherited a poor algorithm from the old one. This is actually a universal problem with stealing code, but it is especially troublesome with this technique because the original program probably was not built with such re-use in mind. Even if its algorithms were good for *its* intended purpose, they may not be versatile enough to do a good job in their new role.

One relatively clean form of theft via editing is to alter the original program's source to generate either desired program by conditional compilation. This eliminates most of the problems. Unfortunately, it does so only if the two programs are sufficiently similar that they can share most of the source. When they diverge significantly, the result can be a maintenance nightmare, actually worse than two separate sources. Given a close similarity, though, this method can work well.

Theft via Libraries

The obvious way of using somebody else's code is to call a library function. Here, UNIX has had some success stories. Almost everybody uses the *stdio* library rather than inventing their own buffered-I/O package. (That may sound trivial to those who never programmed on a V6 or earlier UNIX, but in fact it's a great improvement on the earlier state of affairs.) The simpler sorts of string manipulations are usually done with the *strxxx* functions rather than by hand-coding them, although efficiency issues and the wide diversity of requirements have limited these functions to less complete success. Nobody who knows about *qsort* bothers to write his own sorting function.

* Assuming no software licences, copyrights, patents, etc. are violated!

However, these success stories are pleasant islands in an ocean of mud. The fact is that UNIX's libraries are a disgrace. They are well enough implemented, and their design flaws are seldom more than nuisances, but there aren't *enough* of them! Ironically, UNIX's ''poor cousin'', the Software Tools community [3,4], has done much better at this. Faced with a wild diversity of different operating systems, they were forced to put much more emphasis on identifying clean abstractions for system services.

For example, the Software Tools version of *ls* runs unchanged, *without* conditional compilation, on dozens of different operating systems [4]. By contrast, UNIX programs that read directories invariably dealt with the raw system data structures, until Berkeley turned this cozy little world upside-down with a change to those data structures. The Berkeley implementors were wise enough to provide a library for directory access, rather than just documenting the new underlying structure. However, true to the UNIX pattern, they designed a library which quietly assumed (in some of its naming conventions) that the underlying system used *their* structures! This particular nettle has finally been grasped firmly by the IEEE POSIX project [5], at the cost of yet another slightly-incompatible interface.

The adoption of the new directory libraries is not just a matter of convenience and portability: in general the libraries are faster than the hand-cooked code they replace. Nevertheless, Berkeley's original announcement of the change was greeted with a storm of outraged protest.

Directories, alas, are not an isolated example. The UNIX/C community simply hasn't made much of an effort to identify common code and package it for re-use. One of the two major variants of UNIX still lacks a library function for binary search, an algorithm which is notorious for both the performance boost it can produce and the difficulty of coding a fully-correct version from scratch. No major variant of UNIX has a library function for either one of the following code fragments, both omnipresent (or at least, they *should* be omnipresent [6]) in simple* programs that use the relevant facilities:

```
if ((f = fopen(filename, mode)) == NULL)
        print error message with filename, mode, and specific
        reason for failure, and then exit

if ((p = malloc(amount)) == NULL)
        print error message and exit
```

These may sound utterly trivial, but in fact programmers almost never produce as good an error message for *fopen* as ten lines of library code can, and half the time the return value from *malloc* isn't checked at all!

These examples illustrate a general principle, a side benefit of stealing code: the way to encourage standardization† and quality is to make it easier to be careful and standard than to be sloppy and non-standard. On systems with library functions for error-checked *fopen* and *malloc*, it is easier to use the system functions—which take some care to do ''the right thing''—than to kludge it yourself. This makes converts very quickly.

These are not isolated examples. Studying the libraries of most any UNIX system will yield other ideas for useful library functions (as well as a lot of silly nonsense that UNIX doesn't need, usually!). A few years of UNIX systems programming also leads to recognition of repeated needs. Does *your** UNIX have library functions to:

- decide whether a filename is well-formed (contains no control characters, shell metacharacters, or white space, and is within any name-length limits your system sets)?

* I include the qualification ''simple'' because complex programs often want to do more intelligent error recovery than these code fragments suggest. However, *most* of the programs that use these functions *don't* need fancy error recovery, and the error responses indicated are *better* than the ones those programs usually have now!

† Speaking of encouraging standardization: we use the names *efopen* and *emalloc* for the checked versions of *fopen* and *malloc*, and arguments and returned values are the same as the unchecked versions except that the returned value is guaranteed non-NULL if the function returns at all.

* As you might guess, my system has all of these. Most of them are trivial to write, or are available in public-domain forms.

- close all file descriptors except the standard ones?
- compute a standard CRC (Cyclic Redundancy Check ‘‘checksum’’)?
- operate on *malloc*ed unlimited-length strings?
- do what *access*(2) does but using the effective userid?
- expand metacharacters in a filename the same way the shell does? (the simplest way to make sure that the two agree is to use *popen* and *echo* for anything complicated)
- convert integer baud rates to and from the speed codes used by your system’s serial-line *ioctl*s?
- convert integer file modes to and from the *rwx* strings used† to present such modes to humans?
- do a binary search through a file the way *look*(1) does?

The above are fairly trivial examples of the sort of things that *ought* to be in UNIX libraries. More sophisticated libraries can also be useful, especially if the language provides better support for them than C does; C++ is an example [7]. Even in C, though, there is much room for improvement.

Adding library functions does have its disadvantages. The interface to a library function is important, and getting it right is hard. Worse, once users have started using one version of an interface, changing it is very difficult even when hindsight clearly shows mistakes; the near-useless return values of some of the common UNIX library functions are obvious examples. Satisfactory handling of error conditions can be difficult. (For example, the error-checking *malloc* mentioned earlier is very handy for programmers, but invoking it from a library function would be a serious mistake, removing any possibility of more intelligent response to that error.) And there is the perennial headache of trying to get others to adopt your pet function, so that programs using it can be portable without having to drag the source of the function around too. For all this, though, libraries are in many ways the most satisfactory way of encouraging code theft.

Alas, encouraging code theft does not guarantee it. Even widely-available library functions often are not used nearly as much as they should be. A conspicuous example is *getopt*, for command-line argument parsing. *Getopt* supplies only quite modest help in parsing the command line, but the standardization and consistency that its use produces is still quite valuable; there are far too many pointless variations in command syntax in the hand-cooked argument parsers in most UNIX programs. Public-domain implementations of *getopt* have been available for years, and AT&T has published (!) the source for the System V implementation. Yet people continue to write their own argument parsers. There is one valid reason for this, to be discussed in the next section. There are also a number of excuses, mostly the standard ones for not using library functions:

- ‘‘It doesn’t do quite what I want.’’ *But often it is close enough to serve, and the combined benefits of code theft and standardization outweigh the minor mismatches.*
- ‘‘Calling a library function is too inefficient.’’ *This is mostly heard from people who have never profiled their programs and hence have no* reliable *information about what their code’s efficiency problems are [2].*
- ‘‘I didn’t know about it.’’ *Competent programmers know the contents of their toolboxes.*
- ‘‘That whole concept is ugly, and should be redesigned.’’ (Often said of *getopt*, since the usual UNIX single-letter-option syntax that *getopt* implements is widely criticized as user-hostile.) *How likely is it that the rest of the world will go along with your redesign (assuming you ever finish it)? Consistency and a high-quality implementation are valuable even if the standard being implemented is suboptimal.*
- ‘‘I would have done it differently.’’ *The triumph of personal taste over professional programming.*

† If you think only *ls* uses these, consider that *rm* and some similar programs *ought* to use *rwx* strings, not octal modes, when requesting confirmation!

Theft via Templates

Templates are a major and much-neglected approach to code sharing: ‘‘boilerplate’’ programs which contain a carefully-written skeleton for some moderately stereotyped task, which can then be adapted and filled in as needed. This method has some of the vices of modifying existing programs, but the template can be designed for the purpose, with attention to quality and versatility.

Templates can be particularly useful when library functions are used in a stereotyped way that is a little complicated to write from scratch; *getopt* is an excellent example. The one really valid objection to *getopt* is that its invocation is not trivial, and typing in the correct sequence from scratch is a real test of memory. The usual *getopt* manual page contains a lengthy example which is essentially a template for a *getopt*-using program.

When the first public-domain *getopt* appeared, it quickly became clear that it would be convenient to have a template for its use handy. This template eventually grew to incorporate a number of other things: a useful macro or two, definition of *main*, opening of files in the standard UNIX filter fashion, checking for mistakes like opening a directory, filename and line-number tracking for error messages, and some odds and ends. The full current version can be found in the Appendix; actually it diverged into two distinct versions when it became clear that some filters wanted the illusion of a single input stream, while others wanted to handle each input file individually (or didn’t care).

The obvious objection to this line of development is ‘‘it’s more complicated than I need’’. In fact, it turns out to be surprisingly convenient to have all this machinery presupplied. *It is much easier to alter or delete lines of code than to add them.* If directories are legitimate input, just delete the code that catches them. If no filenames are allowed as input, or exactly one must be present, change one line of code to enforce the restriction and a few more to deal with the arguments correctly. If the arguments are not filenames at all, just delete the bits of code that assume they are. And so forth.

The job of writing an ordinary filter-like program is reduced to filling in two or three blanks* in the template, and then writing the code that actually processes the data. Even quick improvisations become good-quality programs, doing things the standard way with all the proper amenities, because even a quick improvisation is easier to do by starting from the template. *Templates are an unmixed blessing; anyone who types a non-trivial program in from scratch is wasting his time and his employer’s money.*

Templates are also useful for other stereotyped files, even ones that are not usually thought of as programs. Most versions of UNIX have a simple template for manual pages hiding somewhere (in V7 it was */usr/man/man0/xx*). Shell files that want to analyze complex argument lists have the same *getopt* problem as C programs, with the same solution. There is enough machinery in a ‘‘production-grade’’ *make* file to make a template worthwhile, although this one tends to get altered fairly heavily; our current one is in the Appendix.

Theft via Inclusion

Source inclusion (**#include**) provides a way of sharing both data structures and executable code. Header files (e.g. *stdio.h*) in particular tend to be taken for granted. Again, those who haven’t been around long enough to remember V6 UNIX may have trouble grasping what a revolution it was when V7 introduced systematic use of header files!

However, even mundane header files could be rather more useful than they normally are now. Data structures in header files are widely accepted, but there is somewhat less use of them to declare the return types of functions. One or two common header files like *stdio.h* and *math.h* do this, but programmers are still used to the idea that the type of (e.g.) *atol* has to be typed in by hand. Actually, all too often the programmer says ‘‘oh well, on my machine it works out all right if I don’t bother declaring *atol*’’, and the result is dirty and unportable code. The X3J11 draft ANSI standard for C addresses this by defining some more header files and requiring their use for portable programs, so that the header files can do all the work and do it *right*.

* All marked with the string ‘xxx’ to make them easy for a text editor to find.

In principle, source inclusion can be used for more than just header files. In practice, almost anything that can be done with source inclusion can be done, and usually done more cleanly, with header files and libraries. There are occasional specialized exceptions, such as using macro definitions and source inclusion to fake parameterized data types.

Theft via Invocation

Finally, it is often possible to steal another program's code simply by invoking that program. Invoking other programs via *system* or *popen* for things that are easily done in C is a common beginner's error. More experienced programmers can go too far the other way, however, insisting on doing everything in C, even when a leavening of other methods would give better results. The best way to sort a large file is probably to invoke *sort*(1), not to do it yourself. Even invoking a shell file can be useful, although a bit odd-seeming to most C programmers, when elaborate file manipulation is needed and efficiency is not critical.

Aside from invoking other programs at run time, it can also be useful to invoke them at compile time. Particularly when dealing with large tables, it is often better to dynamically generate the C code from some more compact and readable notation. *Yacc* and *lex* are familiar examples of this on a large scale, but simple *sed* and *awk* programs can build tables in more specialized, application-specific ways. Whether this is really theft is debatable, but it's a valuable technique all the same. It can neatly bypass a lot of objections that start with ‘‘but C won't let me write...’’.

An Excess of Invention

With all these varied methods, why is code theft not more widespread? Why are so many programs unnecessarily invented from scratch?

The most obvious answer is the hardest to counter: theft requires that there be something to steal. Use of library functions is impossible unless somebody sets up a library. Designing the interfaces for library functions is not easy. Worse, doing it *well* requires insight, which generally isn't available on demand. The same is true, to varying degrees, for the other forms of theft.

Despite its reputation as a hotbed of software re-use, UNIX is actually hostile to some of these activities. If UNIX directories had been complex and obscure, directory-reading libraries would have been present from the beginning. As it is, it was simply *too easy* to do things ‘‘the hard way’’. There *still* is no portable set of functions to perform the dozen or so useful manipulations of terminal modes that a user program might want to do, a major nuisance because changing those modes ‘‘in the raw’’ is simple but highly unportable.

Finally, there is the Not Invented Here syndrome, and its relatives, Not Good Enough and Not Understood Here. How else to explain AT&T UNIX's persistent lack of the *dbm* library for hashed databases (even though it was developed at Bell Labs and hence is available to AT&T), and Berkeley UNIX's persistent lack of the full set of *strxxx* functions (even though a public-domain implementation has existed for years)? The X3J11 and POSIX efforts are making some progress at developing a common nucleus of functionality, but they are aiming at a common subset of current systems, when what is really wanted is a common superset.

Conclusion

In short, never build what you can (legally) steal! Done right, it yields better programs for less work.

References

[1] Brian W. Kernighan, *The Unix System and Software Reusability*, IEEE Transactions on Software Engineering, Vol SE-10, No. 5, Sept. 1984, pp. 513-8.

[2] Geoff Collyer and Henry Spencer, *News Need Not Be Slow*, Usenix Winter 1987 Technical Conference, pp. 181-190.

[3] Brian W. Kernighan and P.J. Plauger, *Software Tools*, Addison-Wesley, Reading, Mass. 1976.

[4] Mike O'Dell, *UNIX: The World View*, Usenix Winter 1987 Technical Conference, pp. 35-45.

[5] IEEE, *IEEE Trial-Use Standard 1003.1 (April 1986): Portable Operating System for Computer Environments*, IEEE and Wiley-Interscience, New York, 1986.

[6] Ian Darwin and Geoff Collyer, *Can't Happen or /* NOTREACHED */ or Real Programs Dump Core*, Usenix Winter 1985 Technical Conference, pp. 136-151.

[7] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass. 1986.

Appendix

Warning: these templates have been in use for varying lengths of time, and are not necessarily all entirely bug-free.

C program, single stream of input

```
/*
 * name - purpose xxx
 *
 * $Log$
 */

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

#define	MAXSTR	500		/* For sizing strings -- DON'T use BUFSIZ! */
#define	STREQ(a, b)	(*(a) == *(b) && strcmp((a), (b)) == 0)

#ifndef lint
static char RCSid[] = "$Header$";
#endif

int debug = 0;
char *progname;

char **argvp;				/* scan pointer for nextfile() */
char *nullargv[] = { "-", NULL };	/* dummy argv for case of no args */
char *inname;				/* filename for messages etc. */
long lineno;			/* line number for messages etc. */
FILE *in = NULL;			/* current input file */

extern void error(), exit();
#ifdef UTZOOERR
extern char *mkprogname();
#else
#define	mkprogname(a)	(a)
#endif

char *nextfile();
void fail();

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

	progname = mkprogname(argv[0]);

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

	if (optind >= argc)
		argvp = nullargv;
	else
		argvp = &argv[optind];
	inname = nextfile();
	if (inname != NULL)
		process();
	exit(0);
}

/*
 - getline - get next line (internal version of fgets)
 */
char *
getline(ptr, size)
char *ptr;
int size;
{
	register char *namep;

	while (fgets(ptr, size, in) == NULL) {
		namep = nextfile();
		if (namep == NULL)
			return(NULL);
		inname = namep;		/* only after we know it’s good */
	}
	lineno++;
	return(ptr);
}

/*
 - nextfile - switch files
 */
char *				/* filename */
nextfile()
{
	register char *namep;
	struct stat statbuf;
	extern FILE *efopen();

	if (in != NULL)
		(void) fclose(in);

	namep = *argvp;
	if (namep == NULL)	/* no more files */
		return(NULL);
	argvp++;

	if (STREQ(namep, "-")) {
		in = stdin;
		namep = "stdin";
	} else {
		in = efopen(namep, "r");
		if (fstat(fileno(in), &statbuf) < 0)
			error("can’t fstat ‘%s’", namep);
		if ((statbuf.st_mode & S_IFMT) == S_IFDIR)
			error("‘%s’ is directory!", namep);
	}
	lineno = 0;
	return(namep);
}

/*
 - fail - complain and die
 */
void
fail(s1, s2)
char *s1;
char *s2;
{
	fprintf(stderr, "%s: (file ‘%s’, line %ld) ", progname, inname, lineno);
	fprintf(stderr, s1, s2);
	fprintf(stderr, "\n");
	exit(1);
}

/*
 - process - process input data
 */
void
process()
{
	char line[MAXSTR];

	while (getline(line, (int)sizeof(line)) != NULL) {
		xxx
	}
}
```

C program, separate input files

```
/*
 * name - purpose xxx
 *
 * $Log$
 */

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

#define	MAXSTR	500		/* For sizing strings -- DON’T use BUFSIZ! */
#define	STREQ(a, b)	(*(a) == *(b) && strcmp((a), (b)) == 0)

#ifndef lint
static char RCSid[] = "$Header$";
#endif

int debug = 0;
char *progname;

char *inname;				/* filename for messages etc. */
long lineno;			/* line number for messages etc. */

extern void error(), exit();
#ifdef UTZOOERR
extern char *mkprogname();
#else
#define	mkprogname(a)	(a)
#endif
void fail();

/*
 - main - parse arguments and handle options
 */
main(argc, argv)
int argc;
```

```
char *argv[];
{
        int c;
        int errflg = 0;
        FILE *in;
        struct stat statbuf;
        extern int optind;
        extern char *optarg;
        extern FILE *efopen();
        void process();

        progname = mkprogname(argv[0]);

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

        if (optind >= argc)
                process(stdin, "stdin");
        else
                for (; optind < argc; optind++)
                        if (STREQ(argv[optind], "-"))
                                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_IFMT) == S_IFDIR)
                                        error("`%s' is directory!", argv[optind]);
                                process(in, argv[optind]);
                                (void) fclose(in);
                        }
        exit(0);
}

/*
 - process - process input file
 */
void
process(in, name)
FILE *in;
char *name;
{
        char line[MAXSTR];

        inname = name;
        lineno = 0;

        while (fgets(line, sizeof(line), in) != NULL) {
                lineno++;
                xxx
        }
}

/*
 - fail - complain and die
 */
```

```
void
char *s1;
char *s2;
{
        fprintf(stderr, "%s: (file ‘%s’, line %ld) ", progname, inname, lineno);
        fprintf(stderr, s1, s2);
        fprintf(stderr, "\n");
        exit(1);
}
```

Make file

```
# Things you might want to put in ENV and LENV:
# -Dvoid=int            compiler lacks void
# -DCHARBITS=0377       compiler lacks unsigned char
# -DSTATIC=extern       compiler dislikes "static foo();" as forward decl.
# -DREGISTER=           machines with few registers for register variables
# -DUTZOOERR            have utzoo-compatible error() function and friends
ENV = -DSTATIC=extern -DREGISTER= -DUTZOOERR
LENV = -Dvoid=int -DCHARBITS=0377 -DREGISTER= -DUTZOOERR

# Things you might want to put in TEST:
# -DDEBUG               debugging hooks
# -I.           header files in current directory
TEST = -DDEBUG

# Things you might want to put in PROF:
# -Dstatic=’/* */’  make everything global so profiler can see it.
# -p                profiler
PROF =

CFLAGS = -O $(ENV) $(TEST) $(PROF)
LINTFLAGS = $(LENV) $(TEST) -ha
LDFLAGS = -i

OBJ = xxx
LSRC = xxx
DTR = README dMakefile tests tests.good xxx.c

xxx:    xxx.o
        $(CC) $(CFLAGS) $(LDFLAGS) xxx.o -o xxx

xxx.o:          xxx.h

lint:   $(LSRC)
        lint $(LINTFLAGS) $(LSRC) | tee lint

r:      xxx tests tests.good        # Regression test.
        xxx <tests >tests.new
        diff -h tests.new tests.good && rm tests.new

# Prepare good output for regression test -- name isn’t "tests.good"
# because human judgement is needed to decide when output is good.
good: xxx tests
        xxx <tests >tests.good

dtr:    r $(DTR)
        makedtr $(DTR) >dtr

dMakefile:  Makefile
        sed ’/ˆL*ENV=/s/ *-DUTZOOERR//’ Makefile >dMakefile

clean:
        rm -f *.o lint tests.new dMakefile dtr core mon.out xxx
```