#ifdef Considered Harmful, or Portability Experience With C News

Henry Spencer – Zoology Computer Systems, University of Toronto
Geoff Collyer – Software Tool & Die

ABSTRACT

We believe that a C programmer’s impulse to use #ifdef in an attempt at portability is usually a mistake. Portability is generally the result of advance planning rather than trench warfare involving #ifdef. In the course of developing C News on different systems, we evolved various tactics for dealing with differences among systems without producing a welter of #ifdefs at points of difference. We discuss the alternatives to, and occasional proper use of, #ifdef.

Introduction

With UNIX running on many different computers, vaguely UNIX-like systems running on still more, and C running on practically everything, many people are suddenly finding it necessary to port C software from one machine to another. When differences among systems cause trouble, the usual first impulse is to write two different versions of the code—one per system—and use #ifdef to choose the appropriate one. This is usually a mistake.

Simple use of #ifdef works acceptably well when differences are localized and only two versions are present. Unfortunately, as software using this approach is ported to more and more systems, the #ifdefs proliferate, nest, and interlock. After a while, the result is usually an unreadable, unmaintainable mess. Portability without tears requires better advance planning.

When we wrote C News [Coll87a], we put a high priority on portability, since we ran several different systems ourselves, and expected that the software would eventually be used on many more. Planning for future adaptations saved us (and others) from trying to force changes into an uncooperative structure when we later encountered new systems. Porting C News generally involves writing a few small primitives. There have been surprises, but in the course of maintaining and improving the code and its portability, we insisted that the software remain readable and fixable. And we were not prepared to sacrifice performance, since one of C News’s major virtues is that it is far faster than older news software. We evolved several tactics that should be widely applicable.

The Nature of the Problem

Consider what happens when #ifdef is used carelessly. The first #ifdef probably doesn’t cause much trouble. Unfortunately, they breed. Worse, they nest, and tend to become more deeply nested with time. #ifdefs pile on top of #ifdefs as portability problems are repeatedly worked around rather than solved. The result is a tangled and often impenetrable web. Here’s a noteworthy example from a popular newsreader.²

²To quote from the old UNIX kernel: “you are not expected to understand this”.

Further, given worst case elaboration and nesting (each #ifdef always has a matching #else), the number of alternative code paths doubles with each extra level of #ifdef. By the time the depth reaches 5 (not at all rare in the work of #ifdef enthusiasts), there are potentially 32 alternate code paths to consider. How many of those paths have been tested? Probably two or three. How many of the possible combinations even make sense? Often not very many. Figure 2 is another wonderful example, the Leaning Tower Of Hostnames. It’s most unlikely that anyone understands this code any more.

Furthermore, given worst case elaboration and nesting (each #ifdef always has a matching #else), the number of alternative code paths doubles with each extra level of #ifdef. By the time the depth reaches 5 (not at all rare in the work of #ifdef enthusiasts), there are potentially 32 alternate code paths to consider. How many of those paths have been tested? Probably two or three. How many of the possible combinations even make sense? Often not very many. Figure 2 is another wonderful example, the Leaning Tower Of Hostnames. It’s most unlikely that anyone understands this code any more. In such situations, maintenance is reduced to hit-or-miss patching. If you find and fix a bug, how many other branches does it need to be fixed on? If you discover a performance bottleneck and work out a way to fix it, will you have to apply the fix separately to each branch? Now envision what happens when hurried or careless maintainers don’t apply their fixes in all the places where they are relevant.

Philosophical Aspects

The key step in avoiding such messes is to realize that portability requires planning. There is an abundance of bad examples to show that portability cannot be added onto or patched into unportable software. Many of the problems we discuss stem from the “never mind good, we want it next week”
Even the best planning cannot anticipate all problems, but it is important to retain the emphasis on planning even into ongoing maintenance. When a new portability problem surfaces, it is important to step back and think about the problem and its solution. Is this a unique problem, or the harbinger of a whole new class of them? Usually it’s the latter, which makes planning all the more crucial: how can the solution deal with all of them, not just the current one? Failure to think leads to the patch-upon-patch approach to portability, rapidly producing unreadable and unmaintainable code.

Once the problem (class) and the solution are understood, then and only then it is time to start work on the code. Typically this will mean re-implementing parts of it, not just hacking up the old code to work somehow. This highlights another issue: to revise the code, you must understand it... and that means not making an incomprehensible mess this time to interfere with maintenance next time.

All of this is typically more work than just hacking in a quick fix. Sometimes a quick fix may be necessary, or later thought may show that an earlier ‘solution’ was really a quick fix and needs generalizing. In such cases, it is important to go back and fix the kludges. The time is not wasted; it is an investment in the future.

More generally, portability requires time and thought. Nobody gets everything right the first time; getting the code right means taking the time to think about what went wrong, decide what the mistakes were, and go back and fix them.

The alert reader may notice that almost all the remarks in this section could also be applied to achieving high performance, high reliability, etc., and that no specific boundary between development and maintenance was mentioned. We’ve really discussed how to achieve high-quality software. In our experience, this approach works; we can’t imagine any other that would.

### Portable Interfaces

Systems do, unfortunately, differ. It’s often possible to avoid system-dependent areas well enough that the same code will run on all systems; we’ll discuss that later. But sometimes multiple variants are inevitable. Even within the UNIX family, there are significant variations between systems.

```
void cleanup_cni()
{
    register SigUIM ops;
    register SigUIM boposity = 0;
#endif VENUSES
    if (verbose)
        fprintf(stderr,"---hang on a second...",stdin); FLUSH;
    ELSE
#endif TESTS
    fprintf(stderr,"---hang on...",stdin); FLUSH;
#endif
    syspath("Check out your .newrc--hang on a second...",stdin); FLUSH;
    ELSE
#endif TESTS
    fprintf(stderr,"---hang on...",stdin); FLUSH;
#endif
    "It looks like the active file is messrd up. Contact your sys admin,",stdin);
    
    "leave the \'\'bogus\'\' groups alone, and they may come back to normal. Maybe.",stdin); FLUSH;
    }
#endif RELEASES
    ELSE if (boposity) {
#endif VENUSES
    if (verbose)
        fprintf(stderr,"Moving bogus newsgroups to the end of your .newrc",stdin); FLUSH;
    ELSE
#endif TESTS
    fprintf(stderr,"Moving bogus newsgroups to the end of your .newrc",stdin); FLUSH;
#endif
    "You should edit bogus newsgroups out of your .newrc",stdin); FLUSH;
    }
#endif TESTS
    ELSE
#endif VENUSES
    if (verbose)
        fprintf(stderr,"You should edit bogus newsgroups out of your .newrc",stdin); FLUSH;
    ELSE
#endif TESTS
    fprintf(stderr,"You should edit bogus newsgroups out of your .newrc",stdin); FLUSH;
    }
```

Figure 1: Example of overuse of #ifdef

Spencer, & Collyer
#ifdef, or something similar, ultimately is unavoidable. It can be managed, however, to minimize problems.

Among the basic principles of good software engineering are clean interfaces and information hiding: when faced with a decision that might change, hide it in one module, with a simple outside-world interface defined independently of exactly how the decision is made inside. One would think that well-educated modern programmers would not need to be taught the virtues of this technique. Unfortunately, #ifdef doesn't hide anything, and the interface it creates is arbitrarily complex and almost never documented.

The best method of managing system-specific variants is to follow those same basic principles: define a portable interface to suitably-chosen primitives, and then implement different variants of the primitives for different systems. The well-defined interface is the important part: the bulk of the software, including most of the complexity, can be written as a single version using that interface, and can be read and understood in portable terms. It is common wisdom\(^2\) that localizing system dependencies in this way eases porting in cases where the code must actually be rewritten. Our point is that it makes the code simpler, cleaner, and more manageable even when no rewrite is expected.

As a small case in point, when part of C News wishes to arrange that a file descriptor associated with a stdin stream be closed at exec time, to avoid passing it to unprepared children, this is done by

\[
\text{fclsexec}(fp);
\]

(where \(fp\) is the stdin structure pointer) rather than by some complex invocation of ioctl or something similar. Only the implementation of fclsexec needs to be cluttered with the details. (As others have noted in the past [ODel87a, Spen88a] in other contexts, one paradoxical problem of UNIX's not-too-complex system interfaces is that they have discouraged the development of libraries with cleaner, higher-level interfaces.)

This confines #ifdef, but at first glance doesn't seem to eliminate it. Sometimes several system-specific primitives will be compiled from the same source, with portions selected by #ifdef. Note that even limiting the damage can be very important. However, in our experience, it's much more usual for the different variants to be completely different code, compiled from different source files—in essence, the parts outside the #ifdef disappear. The individual source files are generally small and comprehensible, since they implement only the primitives and are uncluttered with the complexities of the main-line logic. Out of 50 such source files in C

\[\text{/* name of this site */}
\]

\[
\text{#ifdef GETHOSTNAME}
\text{char *hostname;}
\#
\text{undef SITENAME}
\#
\text{define SITENAME hostname}
\text{else /* !GETHOSTNAME */}
\#
\text{ifdef DOUNAME}
\#
\text{include <sys/utsname.h>}
\text{struct utsname utsn;}
\#
\text{undef SITENAME}
\#
\text{define SITENAME utsn.nodename}
\text{else /* !DOUNAME */}
\#
\text{ifdef PHOSTNAME}
\text{char *hostname;}
\#
\text{undef SITENAME}
\#
\text{define SITENAME hostname}
\text{else /* !PHOSTNAME */}
\#
\text{ifdef WHOAMI}
\#
\text{undef SITENAME}
\#
\text{define SITENAME sysname}
\#
\text{endif /* WHOAMI */}
\#
\text{endif /* PHOSTNAME */}
\#
\text{endif /* DOUNAME */}
\#
\text{endif /* GETHOSTNAME */}
\]

\[\text{Figure 2: The Leaning Tower of Hostnames}\]
News, half are less than 25 lines, most are under 50, and only a few are over 100. As an example, Figure
3 and Figure 4 are two implementations of fclsexec. There is hardly anything to be gained by trying to
combine these two files into one file with #ifdefs every second line.

There are, of course, things that cannot conveniently be encapsulated as functions, for reasons
of either interface or efficiency. But a "primitive" is not necessarily a function. Types and macros
defined in a header file are also useful ways of hiding system-specific detail. Programmers often use
such facilities on a small scale, e.g. the use of off_t as the system-supplied type for a size of a file or an
offset within it, but they don't write such header

files nearly as often as they should.

Although C's limited macro facilities hamper large-scale use of header-file encapsulation, more ambitious applications can be useful despite occasional clumsiness. As an example, consider our
STRCHR primitive, which generates in-line code except on machines with compilers clever enough to
do so automatically (see Figure 5). This is a bit awkward: what is being defined here is not exactly
a function, but C preprocessor macros nevertheless force it to look like one. In the absence of a stan-
dard way to force inline expansion of normal functions, it remains a powerful technique for portable
performance engineering despite its flaws: this and similar portable optimizations sped up major

/*
 * set close on exec (on UNIX)
 */
#include <stdio.h>
#include <sgtty.h>
void
fclsexec(fp)
FILE *fp;
{
    (void) ioctl(fileno(fp), FIOCLEX, (struct sgttyb *)NULL);
}

Figure 3: One implementation of fclsexec

/*
 * set close on exec (on System V)
 */
#include <stdio.h>
#include <fcntl.h>
void
fclsexec(fp)
FILE *fp;
{
    (void) fcntl(fileno(fp), F_SETFD, 1);
}

Figure 4: Another implementation of fclsexec

#define FASTSTRCHR
#define STRCHR(src, chr, dest) ((dest) = strchr(src, chr))
else
#define STRCHR(src, chr, dest) \\    for ((dest) = (src); *(dest) == '0' && *(dest) != (chr); ++(dest)) \\    ;
    if (*(dest) == '0') \\        (dest) = NULL /* N.B.: missing semi-colon */
#endif

Figure 5: To inline or not to inline

188 Summer '92 USENIX – June 8–June 12, 1992 – San Antonio, TX
If one must use `#ifdef`, and it cannot be confined to header files and the like, one good rule of thumb is use `#ifdef only in declarations` (where "declarations" is understood to include macro definitions). This at least encourages some thought about defining an interface, rather than just hacking in something that somehow seems to work.

Finally, when defining interfaces, it is important to document them. The biggest reason for doing this is that it is important discipline that forces you to think about the issues and fill in fuzzy spots. The resulting documentation is also very valuable for maintenance. Perhaps somewhat surprisingly, it's also valuable for development, even if the project is not an army-of-ants operation using buildings full of people. We found it very important to document crucial interfaces like our configuration primitives, even though only two people were involved, to make sure things were being done consistently and we understood each other.

### Standard Interfaces

Of course, good interface design is not simple, especially given the limitations of existing programming languages. Often the best way to solve this problem is to avoid it instead. If an interface is needed, there is much to be said for choosing one that is already standard.

---

3Indeed, places where internal interfaces weren't completely documented were fruitful sources of misunderstandings, bugs, and a certain amount of snarling at each other.

```c
/*
 * strerror — map error number to descriptive string
 *
 * This version is obviously somewhat UNIX-specific.
 */
char *
strerror(errnum)
int errnum;
{
    extern int sys_nerr;
    extern char *sys_errlist[];
    if (errnum > 0 && errnum < sys_nerr)
        return(sys_errlist[errnum]);
    else if (errnum == 0)
        return("unknown error");
    else
        return("no details given");
}
```

Figure 6: `strerror`

There are several sources of reasonably decent standard interfaces, notably ANSI C [Inst89a] and POSIX 1003.1 [Engl90a]. Since these standards are quite recent, many of the systems of interest do not implement them fully. This doesn't preclude using the interfaces, however: you can supply your own implementation(s) for use on outdated systems. An example is the ANSI function `strerror` (shown in Figure 6).

This approach does impose a few constraints, since the standard interfaces sometimes are a bit ugly, and often aren't ideal for every program. It's tempting to come up with customized ones instead. But the standard ones have major advantages. For one thing, people understand (or will understand) them without having to decipher your code. For another, on systems which do implement the standard interfaces, the system-provided ones can be used. (This is particularly significant for primitives like `memcpy`, where system-specific tuning can produce major improvements in efficiency [Spen88a]. If you define your own customized interface, you must do your own customized implementation, which denies you the opportunity to benefit from the work of others.) For a third, while the standard interfaces may not be ideal, by and large they contain no grievous mistakes, and avoiding disasters is usually more important than achieving a precisely optimal solution. Finally, a standard interface saves endless puzzling, not to mention uncomplimentary speculation, by later maintainers: "did he have some deep subtle reason for using a non-standard interface, or was he just stupid?".

Reimplementing a standard interface can be a useful tactic when the standard interface does the right thing but the usual implementations perform
poorly. A version which is faster but compatible can solve performance problems while leaving the door open to the possibility that the system implementations will improve someday. The stdio library is a particular case in point: old implementations of functions like fgets and fread are extremely inefficient, and even modern ones often can be improved on. This particular case gets tricky, because doing better means relying on ill-documented and somewhat variable internal interfaces, but the performance wins for C News are so massive that we nevertheless did it.

Pitfalls that need careful attention when using standard interfaces are error checking and boundary conditions. It is important not to make assumptions that aren’t in the standard. For example, a depressing amount of UNIX software assumes that close never returns any interesting status. Unfortunately, as networked file systems get more common and other complications are introduced, it is not at all unthinkable for an I/O error to be discovered only at close time. Meticulous error checking is important [Darw85a]. For example, see Figure 7.

Finally, note that standard interfaces exist on more than just the C level. By including an “override” directory early in the shell’s search path, it becomes trivial to substitute reimplemented programs for standard ones that are missing or defective. We have a remarkably large—and steadily growing—list of known portability problems that arise from defective implementations of standard UNIX programs.

**Inside-Out Interfaces**

Sometimes there simply isn’t any way to provide a necessary primitive on some systems. For example, most modern UNIXes permit setting the real user ID to equal the effective user ID, but some old systems allow only root to change the real IDs... and it is necessary to change the real IDs to create directories with proper ownerships. Given that many people will be reluctant to let a large and complex program written by a stranger run as root, there doesn’t seem to be any easy way out.

In this case, there is: turn the interface inside out, and have the dirty work done by caller rather than callee. Specifically, have the complex program invoked by a simple setuid-root program which sets things up properly on uncooperative systems.

---

3The standard build procedure for C News runs a test program to check compatibility with the local stdio implementation.

4The standard build procedure for C News runs a test program to check compatibility with the local stdio implementation.

5Lest anyone think we are disparaging porters and resellers only, we should comment that AT&T is as guilty as anyone else. For example, several releases of System V make have violated the System V Interface Definition in their handling of command lines like test -s file in makefiles. (Makefile command lines are specified to be executed just as if by the shell, but if test is a shell builtin and there is no actual program by that name, make often chokes and dies on this line.) Or should be!!!

 第 7 图:必要的错误检查

```c
/*
 * nfclose(stream) - flush the stream, fsync its file descriptor and
 * fclose the stream, checking for errors at all stages. This dance
 * is needed to work around the lack of UNIX file system semantics
 * in Sun's NFS. Returns EOF on error.
 *
 *include <stdio.h>

 int
 nfclose(FILE *stream);
 { register FILE *stream;
   register int ret = 0;
   if (fflush(stream) == EOF)
     ret = EOF;
   if (fsync(fileno(stream)) < 0) /* may get delayed error here */
     ret = EOF;
   if (fclose(stream) == EOF)
     ret = EOF;
   return ret;
 }
```

Figure 7: Necessary error checking
A more mundane example is the problem of reading directories. Thanks to the lack of a library package for directory-reading in the oldest UNIX variants, there isn’t any standard way to do it. Raw reads don’t work on 4.2+BSD systems (and increasingly many others), the Berkeley directory library works well but has stupid name clashes with many old systems, and the POSIX library isn’t widespread yet. Worse, because the insides of a directory-reading library are system-specific, it’s difficult to provide a portable reimplementation of the POSIX functions.

The simplest way around this one is to move the problem out to a higher level of abstraction. The ls utility portably does the job, so wrap the invocation of the program in a shell file, with the list of names generated by ls and fed into the application as arguments or on standard input. The performance impact is rarely significant, and the alternative currently involves at least six different variants of the code, with more surfacing daily.

A less happy example of this technique is C News’s spacefor program, used to check disk space so activity can be curtailed when it runs short. Its interface is simple and clean, and it is used everywhere in C News. Making it a shell program offered the possibility of exploiting the df command, which encapsulates the ugly complications of finding out how much space is available (and, sometimes, the root privileges needed to do so). Unfortunately, df is often relatively costly to invoke; worse, the only portable way to do 32-bit arithmetic from shell scripts is to use awk, which likewise tends to have considerable startup overhead. With some care, the performance impact was tolerable, although not entirely pleasant.

What we had not anticipated was that every little UNIX variant has its own different, incompatible df output format. Even “consider it standard” System V has at least three. The importance of program output being useful as program input [Ric78a] has been disregarded completely. In the end, we found that while the df version remains useful—people with really odd systems can customize it easily—it was best to also provide C variants that use the three or four commonest space-determining system calls, improving (!) portability within a fairly large subset of UNIX variants.

Levels of Abstraction

In general, avoiding problems is better than solving them. The best way to solve portability problems is not to get involved with them. Sometimes they can’t be avoided, but often a bit of ingenuity suffices to find a way around them.

The most powerful way of avoiding problems is to choose a level of abstraction where they don’t show up. The ls example earlier was a case in point. The standard UNIX shell is a very powerful programming language, sufficiently removed from the lower levels of the system that shell programs are often highly portable. (Gratuitous differences in utility programs do get in the way, as do attempts to “improve” the shell that result in subtle or not-so-subtle incompatibilities, but this is usually a manageable problem.)

The usual objection to shell programming is the inefficiency of the result, but a careful division of labor between the shell and the programs it invokes is all that is needed. Most of the C News batching subsystem is written in shell, but it remains highly efficient, because most of its time is spent in the “batcher | compress | uux” pipeline, and those are all C programs.

Intermediate levels of abstraction, although harder to find, do exist. Substantial pieces of C News are coded in awk [Spen91a] where efficiency is not crucial and requirements permit.

One situation where high-level abstractions are particularly beneficial is when one must step outside “common base UNIX.” Common base UNIX is essentially Version 7 [Lab82a], although the later V7 innovations have taken a while to find their way into System V (and some have never done so). POSIX 1003.1 [Eng90a] is mostly an attempt to codify common base UNIX. Unfortunately, common base UNIX did not address some issues at all, notably dealing with real-time networks like the Internet. Attempts to define interfaces for real-time networks [Div83a, ATT86a] have generally resulted in complex and ugly messes. Where, there is no consensus on which one to use, and the quality of the designs can be judged by the rate at which they are being redesigned to deal with unexpected problems. Although higher-level abstractions for networking are not as common or as well-designed as they should be, networked file systems and shell invocation of programs like rsh can provide limited networking functionality without having to deal with the underlying mess.

A side benefit of high-level abstractions is that the resulting programs are generally far easier to modify and customize. This is a particularly important consideration for software intended to be run on many systems with varying administrative policies. Many system administrators who are not up to deciphering a 5000-line C program can cope quite well with modifying a 50-line shell script. We have made a conscious effort to put policy decisions in shell scripts, not in C code, wherever possible, and have had extensive and loud positive feedback on this.

#ifdef Considered Harmful...

There are occasional exceptions like V10 Research UNIX [Cent90a] that are useful sources of interface ideas.
There is one negative aspect to moving to a higher level of abstraction: the resulting programs depend on a larger and perhaps more fragile set of underlying abstractions. Porting C News to a radically non-UNIX-like operating system reportedly typically involves little change to the C code, since the

```c
#ifdef SYSLOG
    syslog(LOG_ERR, "cannot open history file: %m");
#else
    perror("ntpxfer: cannot open history file");
#endif
exit(1);
#endif
#endif
#endif

if ((server = get_tcp_conn(argv[1], "ntpx") < 0)
{
#ifdef SYSLOG
    syslog(LOG_ERR, "cannot open socket: %m");
#else
    perror("ntpxfer: cannot open socket");
#endif
exit(1);
}
if ((rd_fp = fdopen(server, "r")) == (FILE *) 0) {
#ifdef SYSLOG
    syslog(LOG_ERR, "cannot fdopen socket: %m");
#else
    perror("ntpxfer: cannot fdopen socket");
#endif
exit(1);
}

syslog(LOG_DEBUG, "connected to nntp server at %s", argv[1]);
#endif
#endif
else
    printf("connected to nntp server at %s\n", argv[1]);
#endif
/*
 * ok, at this point we're connected to the nntp daemon
 * at the distant host.
*/
```

Figure 8: A truly awful style
UNIX and C programming interfaces are widespread even on non-UNIX systems, but substantial shell files relying on dozens of major UNIX utilities are more of a challenge. There is also the problem, mentioned earlier, of UNIX suppliers breaking formerly-working utilities.

Low-Level Portability

We assume that everyone reading this has had exposure to elementary notions of portability like using typedef names, avoiding stupid assumptions about the sizes of integers and/or pointers, being careful about byte order in interchange formats, etc. There are nevertheless a good many fine points that deserve some illumination, particularly in the area of how to use #ifdef safely.

As mentioned earlier, if #ifdef is needed at all, it is best confined to declarations, to try to preserve some explicit notion of interfaces. Such declarations, in turn, preferably should be confined to header (.h) files, to minimize the temptation to introduce #ifdef into main-line code.

An optional feature such as debugging assistance or logging can be defined as a macro or function that does nothing when not needed, else the full-blown function can be defined (perhaps in one of several system-specific ways, e.g. using a syslog daemon or not). At worst, this requires one #ifdef per such feature rather than the now-notorious style, seen in various bits of popular software, of clustering #ifdefs at the site of each call of said function(s), see Figure 8.

One awkward area is functions with variable numbers of arguments. There is no way to write a C macro that can take a variable number of arguments, which makes it awkward to provide such an interface while still being able to hide the innards. Various tricks are in use, none of them entirely satisfactory; perhaps the least objectionable is an extra level of parentheses:

```c
DEBUG( ("oops: @s \dn", b, c));
```

which lets a header file decide to either pass or discard the whole argument list:

```c
#ifdef NDEBUG
#define DEBUG(list) /* nothing */
#else
#define DEBUG(list) printf list
#endif
```

A related problem is that definition of a variable-arguments function pretty well invariably involves some #ifdefing to cope with the unfortunate differences between ANSI C stdarg.h and the traditional (although less portable) varargs.h.

Although macros cannot take variable numbers of arguments, it is still possible to have them pick and choose among a fixed number of arguments. For example, the VERBOSE-TERSE business in one of our first exhibits, an attempt to avoid compiling in unneeded strings, can be handled with a macro:

```c
MSG(short_form, long_form, iostream)
```

A short-form-only definition of the macro simply doesn’t use the long_form argument. The choice can even be made at run time using if or the "?" operator, all by changing only the definition of the macro.

One valid use of #ifdef, particularly in header files, is the idiom

```c
#ifdef NDEBUG
#define DEBUG(list) /* nothing */
#else
#define DEBUG(list) printf list
#endif
```

to supply a default value that can be overridden at compilation time (with cc -DCOPYSIZE=4096). One could wish for a shorter form (e.g., #ifndefdef), or even a compiler option allowing one to specify a value that overrides the first one defined in the program, since this idiom is common and very useful.

However, the first question to ask about such numeric parameters is whether they should be there at all. Consider:

```c
#ifdef pdpl1
#define LBUFLEN 512
#define LBUFLEN 1024
#else
#define LBUFLEN 1024
#endif
```

This code presumes that people on small machines (or at least PDP-11s) prefer their programs to crash earlier than people on large machines. Any code using such (unchecked) fixed-sized buffers is prone to falling over and dying (or at best mysteriously truncating or wrapping long lines) anyway; the #ifdefs tip us off that these limits should be abolished and replaced with code that deals with dynamically-sized strings.

Another legitimate use of #ifdef, in fact required by the ANSI C standard in standard header files, is in protecting header files against multiple inclusion. In complex programs it can be quite difficult to ensure that a needed header file is included once and only once, and including it more than once typically causes problems with duplicate typedefs, structure tags, etc. Ignoring some issues of name-space control, the usual idiom for defending header files against multiple inclusion goes something like this:

```c
#ifdef FOO_H
#define FOO_H 1
/* interface to the foo module */
```

8Actually, it's awkward in a great many ways, this being only one.
typedef struct {
    char *foo_a;
    char *foo_b;
} foo;
extern foo *mkfoo();
extern int rmfoo();
#endif

(Some compiler implementors have invented bizarre special-purpose constructs, typically using ANSI C's #pragma, to avoid having the compiler re-scan the header file on later inclusions. That is not necessary. It suffices to have the compiler remember that the entire text of the file is inside the #ifndef, and hence need not be rescanned if FOO_H is still defined.)

#ifdef vax
    f(*ptr);
#endif
#ifdef pyr
    /* darned Pyramid is so picky */
    if (ptr != NULL)
        f(*ptr);
#endif
#ifdef sparc
    /* the Sun 4 is just as bad! */
    if (ptr != NULL)
        f(*ptr);
#endif
#ifdef cray
    /* avoid dereferencing null */
    if (ptr != NULL)
        f(*ptr);
#endif

Figure 9: Protecting broken code

A related point, also illustrated by that example, is that if one must use #ifdef, one should test for specific features or characteristics (typically indicated to the compiler by symbols defined in a header file or on a command line), not for specific machines. There will almost always be another machine with the same problem. Consider the interesting bit of code shown in Figure 10. Rather mysterious, isn’t it? What is so odd about Crays, and is it only Crays that are affected?

If testing for particular machines is unavoidable, perhaps because of some highly machinespecific operation, consider what happens if no machine is specified (or if the machine is one you’ve never heard of and hence didn’t bother to list). Don’t assume there is a default machine. It is much kinder to produce a syntax error than silently inappropriate code.

Occurrences of #include inside #ifdef should always be viewed with suspicion. There are better ways. Consider:

```c
#ifdef NDIR
    #ifdef M_XENIX
        #include <sys/ndir.h>
    #else
        #include <ndir.h>
    #endif
#else
    #include <sys/dir.h>
#endif
#ifdef USG
    #include <time.h>
#else
    #include <sys/time.h>
#endif
```

Arranging, typically via a makefile, to put an “override” directory in the search path for header files is a tremendously powerful way of fixing botches in a site’s header files without #ifdef.

When one uses #ifdef, one should base the tests on individual features:

```c
#include <signal.h>
/* may define SIGTSTP */
#ifdef SIGTSTP
    (void) signal(SIGTSTP, SIG_IGN);
    /* no suspension, thanks */
#endif
```

and not on (inaccurate) generalisations:

```c
while (*s != '\n')
#else
    while (*s != '\r')
#endif
/* till a newline (not echoed) */
```

Figure 10: Mysterious code
Pragmatic Aspects of Portability

In practice, one encounters all manner of breakage in vendor-supplied system software: compilers, utilities (notably the shell and awk), libraries, kernels. Optimizers may need to be turned off if they are broken. Installers may have to pick up working commands from other sources (e.g. the Usenet group comp.sources.unix or the GNU [Fonna] project). Sometimes it is worth supplying simple but correct versions of small things (e.g. library functions) when a large class of machines is known to have broken ones. We ultimately decided that we could not provide complete replacements, or even workarounds, for all potentially-broken system software. Sometimes the problems are horrific enough that the right response is not to contort one's code but to get the customers to complain about the breakage until it is fixed.

Given all these potential problems, it is important to detect breakage as well as avoiding it or coping with it. We think very highly of regression tests, prepackaged tests that exercise the basic functionality of the software and check that the results are correct. They are very useful during development, both for bug-hunting in new code and for confidence testing before release. Of equal importance, though, is that they give the installer reasonable confidence that the software is actually working on his system, and that no glaring portability problems have escaped his notice.

Similarly, internal consistency checks, such as validated magic numbers in structures passed between user code and libraries, can save one's sanity by detecting breakage in system software early, before corruption spreads everywhere. Trying to debug a core dump by mail on an unfamiliar machine is not fun.

To a greater extent than we had anticipated, one learns about portability by porting. The system call variations among UNIX systems are fairly well documented and understood. The variations in commands were less well understood, at least by us, and the variations in programming environments were still more surprising. There is no substitute for trying your software on several seriously-different machines before release. It's also worth making an effort to pick your beta-testers for maximum diversity of environments: we found a lot of unexpected problems that way.

Finally, a plea: if you find portability problems, document them. You can't expect everyone to actually read the documentation—we frequently respond to queries with "please read section such-and-such, it'll tell you all about it"—but the more careful and conscientious installers benefit greatly from an advance look at known problems, especially when a truly weird system is involved.

Configuration

Given the senseless diversity in existing systems, some way to configure software for a new system is needed. Given that #ifdef can't do the whole job, how should we proceed? C News currently has an interactive build script that interrogates the installer about his system and then constructs a few shell scripts, which when run will use make to build the software. We intend to push most of the shell scripts into the makefiles, so that casual use of make works as people expect, but the general approach seems to be a good one: ask which emulation routines and header files are necessary, rather than trying to guess. This strategy even allows cross-compilation and some degree of cross-compilation, which autoconfiguration schemes generally don't. It is also more trustworthy than autoconfiguration schemes, which can be fooled by some new innovation.

Almost all of build's configuration questions turn into choices of files rather than values for #ifdef

#ifdef Considered Harmful ...

Similarly, internal consistency checks, such as validated magic numbers in structures passed between user code and libraries, can save one's sanity by detecting breakage in system software early, before corruption spreads everywhere. Trying to debug a core dump by mail on an unfamiliar machine is not fun.

To a greater extent than we had anticipated, one learns about portability by porting. The system call variations among UNIX systems are fairly well documented and understood. The variations in commands were less well understood, at least by us, and the variations in programming environments were still more surprising. There is no substitute for trying your software on several seriously-different machines before release. It's also worth making an effort to pick your beta-testers for maximum diversity of environments: we found a lot of unexpected problems that way.

Finally, a plea: if you find portability problems, document them. You can't expect everyone to actually read the documentation—we frequently respond to queries with "please read section such-and-such, it'll tell you all about it"—but the more careful and conscientious installers benefit greatly from an advance look at known problems, especially when a truly weird system is involved.

Configuration

Given the senseless diversity in existing systems, some way to configure software for a new system is needed. Given that #ifdef can't do the whole job, how should we proceed? C News currently has an interactive build script that interrogates the installer about his system and then constructs a few shell scripts, which when run will use make to build the software. We intend to push most of the shell scripts into the makefiles, so that casual use of make works as people expect, but the general approach seems to be a good one: ask which emulation routines and header files are necessary, rather than trying to guess. This strategy even allows cross-compilation and some degree of cross-compilation, which autoconfiguration schemes generally don't. It is also more trustworthy than autoconfiguration schemes, which can be fooled by some new innovation.

Almost all of build's configuration questions turn into choices of files rather than values for #ifdef

#ifdef Considered Harmful ...

Similarly, internal consistency checks, such as validated magic numbers in structures passed between user code and libraries, can save one's sanity by detecting breakage in system software early, before corruption spreads everywhere. Trying to debug a core dump by mail on an unfamiliar machine is not fun.

To a greater extent than we had anticipated, one learns about portability by porting. The system call variations among UNIX systems are fairly well documented and understood. The variations in commands were less well understood, at least by us, and the variations in programming environments were still more surprising. There is no substitute for trying your software on several seriously-different machines before release. It's also worth making an effort to pick your beta-testers for maximum diversity of environments: we found a lot of unexpected problems that way.

Finally, a plea: if you find portability problems, document them. You can't expect everyone to actually read the documentation—we frequently respond to queries with "please read section such-and-such, it'll tell you all about it"—but the more careful and conscientious installers benefit greatly from an advance look at known problems, especially when a truly weird system is involved.

Configuration

Given the senseless diversity in existing systems, some way to configure software for a new system is needed. Given that #ifdef can't do the whole job, how should we proceed? C News currently has an interactive build script that interrogates the installer about his system and then constructs a few shell scripts, which when run will use make to build the software. We intend to push most of the shell scripts into the makefiles, so that casual use of make works as people expect, but the general approach seems to be a good one: ask which emulation routines and header files are necessary, rather than trying to guess. This strategy even allows cross-compilation and some degree of cross-compilation, which autoconfiguration schemes generally don't. It is also more trustworthy than autoconfiguration schemes, which can be fooled by some new innovation.

Almost all of build's configuration questions turn into choices of files rather than values for #ifdef

#ifdef Considered Harmful ...

Similarly, internal consistency checks, such as validated magic numbers in structures passed between user code and libraries, can save one's sanity by detecting breakage in system software early, before corruption spreads everywhere. Trying to debug a core dump by mail on an unfamiliar machine is not fun.

To a greater extent than we had anticipated, one learns about portability by porting. The system call variations among UNIX systems are fairly well documented and understood. The variations in commands were less well understood, at least by us, and the variations in programming environments were still more surprising. There is no substitute for trying your software on several seriously-different machines before release. It's also worth making an effort to pick your beta-testers for maximum diversity of environments: we found a lot of unexpected problems that way.

Finally, a plea: if you find portability problems, document them. You can't expect everyone to actually read the documentation—we frequently respond to queries with "please read section such-and-such, it'll tell you all about it"—but the more careful and conscientious installers benefit greatly from an advance look at known problems, especially when a truly weird system is involved.

Configuration

Given the senseless diversity in existing systems, some way to configure software for a new system is needed. Given that #ifdef can't do the whole job, how should we proceed? C News currently has an interactive build script that interrogates the installer about his system and then constructs a few shell scripts, which when run will use make to build the software. We intend to push most of the shell scripts into the makefiles, so that casual use of make works as people expect, but the general approach seems to be a good one: ask which emulation routines and header files are necessary, rather than trying to guess. This strategy even allows cross-compilation and some degree of cross-compilation, which autoconfiguration schemes generally don't. It is also more trustworthy than autoconfiguration schemes, which can be fooled by some new innovation.

Almost all of build's configuration questions turn into choices of files rather than values for #ifdef

#ifdef Considered Harmful ...

Similarly, internal consistency checks, such as validated magic numbers in structures passed between user code and libraries, can save one's sanity by detecting breakage in system software early, before corruption spreads everywhere. Trying to debug a core dump by mail on an unfamiliar machine is not fun.

To a greater extent than we had anticipated, one learns about portability by porting. The system call variations among UNIX systems are fairly well documented and understood. The variations in commands were less well understood, at least by us, and the variations in programming environments were still more surprising. There is no substitute for trying your software on several seriously-different machines before release. It's also worth making an effort to pick your beta-testers for maximum diversity of environments: we found a lot of unexpected problems that way.

Finally, a plea: if you find portability problems, document them. You can't expect everyone to actually read the documentation—we frequently respond to queries with "please read section such-and-such, it'll tell you all about it"—but the more careful and conscientious installers benefit greatly from an advance look at known problems, especially when a truly weird system is involved.

Configuration

Given the senseless diversity in existing systems, some way to configure software for a new system is needed. Given that #ifdef can't do the whole job, how should we proceed? C News currently has an interactive build script that interrogates the installer about his system and then constructs a few shell scripts, which when run will use make to build the software. We intend to push most of the shell scripts into the makefiles, so that casual use of make works as people expect, but the general approach seems to be a good one: ask which emulation routines and header files are necessary, rather than trying to guess. This strategy even allows cross-compilation and some degree of cross-compilation, which autoconfiguration schemes generally don't. It is also more trustworthy than autoconfiguration schemes, which can be fooled by some new innovation.

Almost all of build's configuration questions turn into choices of files rather than values for #ifdef
to examine. The few exceptions are mostly historical relics, and will be revised or deleted as time permits.

Statistics

A snapshot of current C News working sources shows 955 lines of header files and 19,762 lines of C files, split between 5,640 lines from libraries (including alternate versions of primitives), and 14,122 lines of mainline C code. Here is a breakdown of the #ifdef usage in that code:

<table>
<thead>
<tr>
<th>reason</th>
<th>h</th>
<th>main .c</th>
<th>dbz</th>
<th>rna</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifndefdef</td>
<td>13</td>
<td>40</td>
<td>6</td>
<td>0</td>
<td>59</td>
</tr>
<tr>
<td>comment</td>
<td>4</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>config.</td>
<td>6</td>
<td>25</td>
<td>19</td>
<td>7</td>
<td>57</td>
</tr>
<tr>
<td>protect .h</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>pdp11</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>lint</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>scssid</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>STATS</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>other</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>34</td>
<td>97</td>
<td>28</td>
<td>7</td>
<td>166</td>
</tr>
</tbody>
</table>

The .h column represents header files. The main .c column represents all .c files other than those in the dbz and rna (Australian readnews) directories. The ifndefdef row represents the 'if not defined, define' idiom. The comment row represents uses of #ifdef to comment out obsolete, futuristic or otherwise unwanted code. The config. row represents uses of #ifdef to configure the software.

rna is presented separately because we inherited it rather than writing it. dbz is presented separately because it uses #ifdef heavily for configuration, for backward compatibility and to attempt to stand independently of C News. The main C files' use of #ifdef for "configuration" is misleading; in fact this is mostly vestigial code, superseded but not yet deleted from our current working copies.

Conclusions

Despite problems along the way, C News is outstandingly portable. It comes up easily on an amazing variety of UNIX systems. Other people have reported porting C News relatively easily to environments that we had considered too hostile, or at least too different from UNIX, to even consider as possible target systems: notably VMS, MS-DOS and Amiga DOS. The only major operating system known to present serious obstacles is VM/CMS.

---

14 Of those that affect compilation at all; some questions are decisions affecting setup of control files for the compiled software to use.

In our experience, #ifdef is usually a bad idea (although we do use it in places). Its legitimate uses are fairly narrow, and it gets abused almost as badly as the notorious goto statement. Like the goto, the #ifdef often degrades modularity and readability (intentionally or not). Given some advance planning, there are better ways to be portable.

Acknowledgements

Thanks to Rob Kolstad for helpful comments on a draft of this paper. Thanks to James Clark for grep (and the rest of groff). And thanks to the authors of our bad examples—you know who you are.

References

Spen91a. Henry Spencer, "Awk As A Major Systems Programming Language,"  

Author Information

Henry Spencer is head of Zoology Computer Systems at the University of Toronto. He is known for his regular expression and string libraries, and as a co-author of the C News netnews software. Reach him via Canada Post at Zoology Computer Systems, 25 Harbord St., University of Toronto, Toronto, Ont. M5S 1A1 Canada. His electronic mail address is utzoo henry or henry@zoo.toronto.edu.

Geoff Collyer leads C News development at Software Tool & Die. He is senior author of the C News netnews software. His interests include simple, small, fast, elegant and powerful system software. Reach him via U.S. Mail at Software Tool & Die, 1330 Beacon St. #215, Brookline, MA 02146. His electronic mail address is worldlgeoff or geoff@world.std.com.