Notes on Writing Portable Programs in C

(Nov 1990, 8th Revision)

A. Dolenc*
A. Lemmke
Helsinki University of Technology
D. Keppel†
CS&E, University of Washington
and
G. V. Reilly‡
Dept. of Computer Science, Brown University

February 13, 1995

Abstract

This documents describes the features and non-features of different C preprocessors, compilers, and environments. As such, it is an incomplete document, growing as information is gathered. It contains some material concerning ANSI C but it is not a substitute for the Standard itself; neither are related textbooks. We assume the reader is familiar with the C programming language.

---

*Internet: `ado@sauna.hut.fi`.
†Internet: `pardo@cs.washington.edu`.
‡Internet: `gvr@cs.brown.edu`.

Contents

1 Foreword

We will call a program *portable* if adapting it to a new environment is easier than rewriting it for that environment. This document is mainly for those who have *never* ported a program to another platform — a specific hardware and software environment — and, evidently, for those who plan to write large systems which must be used across different vendor machines. If you have already done some porting, you may not find the information herein very useful.

We suggest that [CEK+90] be read in conjunction with this document.[1] Posters to the newsgroup **comp.lang.c** have repeatedly recommended [Hor90] and [Koe89] (none of the information herein has been taken from those two references).

Disclaimer: We will attempt to keep the information herein updated, but it can happen that some of it may be incorrect at the time of reading. The code fragments presented are intended to make applications "more" portable, meaning that they may fail with some compilers and/or environments.

This document can be obtained via anonymous FTP from `sauna.hut.fi` [130.233.251.253] in '`~ftp/pub/CompSciLab/doc`'. The files '`portableC.tex`', '`portableC.sty`', '`portableC.bib`', and '`portableC.ps.Z`' are the LaTeX source and style files, BibTeX and the compressed PostScript, respectively. Alternatively, there is a site in the US from which one can obtain all four files, `cs.washington.edu` [128.95.1.4] in '`~ftp/pub/cport.tar.Z`'. All files are in the public domain. Comments, suggestions, flames, eggs, and requests for copies via e-mail should be directed to `ado@sauna.hut.fi`.

2 Introduction

The aim of this document is to collect the experience of several people who have had to write and/or port programs written in C to more than one platform.

In order to keep this document within reasonable bounds, we must restrict ourselves to programs which must execute under Unix-like operating systems and those which implement a reasonable Unix-like environment. The only exception we will consider is VMS.

A wealth of information can be obtained from programs that have been written to run on several platforms. This is the case of publicly available software such as that developed by the Free Software Foundation and the MIT X Consortium.

[1] [CEK+90] can be obtained via *anonymous FTP* from `cs.washington.edu` in '`~ftp/pub/cstyle.tar.Z`'.

When discussing portability, one focuses on two issues:

The language, which includes the preprocessor and the syntax and the semantics of the language.

The environment, which includes the location and contents of header files and the run-time library.

We include in our discussions the standardization efforts upon the language and the environment. Special attention will be given to floating-point representations and arithmetic, to limitations of specific compilers, and to VMS.

Our main focus will be *boiler-plate* problems. Systems programming, *e.g.*, raw I/O from terminals, and twisted code associated with bizarre interpretations of [X3J88] — henceforth referred to as the Standard — are not extensively covered in this document.[2]

3 Standardization Efforts

All standards have a good side and an evil side. Due to the nature of this document, we are forced to focus our attention on the latter.

The American National Standards Institute (ANSI) has recently approved of a standard for the C programming language [X3J88]. The Standard concentrates on the syntax and semantics of the language and specifies a minimum environment (the name and contents of some header files and the specification of some run-time library functions).

Copies of the ANSI C Standard (ANSI X3.159–1989) can be obtained from the following address:

American National Standards Institute
Sales Department
1430 Broadway
New York, NY 10018
(Voice) (212) 642–4900
(Fax) (212) 302–1286

3.1 ANSI C

3.1.1 Translation Limits

We first bring to the reader's attention the fact that the Standard states some environmental limits. These limits are *lower bounds*, meaning that a correct

[2] We regard this document as a living entity growing as needed and as information is gathered. Future versions of this document may contain a lot of such information.

(compliant) compiler may refuse to compile an otherwise-correct program that exceeds one of those limits.[3]

Below are the limits that we judge to be the most important. The ones related to the preprocessor are listed first.

- *8 nesting levels of conditional inclusion.*
- *8 nesting levels for* `#include`*d files.*
- *32 nesting levels of parenthesized expressions within a full expression.* This will probably occur when using macros.
- *1024 macro identifiers simultaneously.* Can happen if one includes too many header files.
- *509 characters in a logical source line.* This is a serious restriction if it applies *after* preprocessing. Since a macro expansion always results in one line, this affects the maximum size of a macro. It is unclear what the Standard means by a logical source line in this context and in most implementations this limit will probably apply *before* macro expansion.
- *6 significant initial characters in an external identifier.* Usually this constraint is imposed by the environment, *e.g.*, the linker, and not by the compiler.
- *127 members in a single structure or union.*
- *31 parameters in one function call.* This may cause trouble with functions that accept a variable number of arguments. Therefore, it is advisable that when designing such functions that either the number of parameters be kept within reasonable bounds or that alternative interfaces be supplied, *e.g.*, using arrays.

It is really unfortunate that some of these limits may force a programmer to code in a less elegant way. We are of the opinion that the remaining limits stated in the Standard can usually be obeyed if one follows "good" programming practices.

However, these limits may break programs that *generate* C code such as compiler-compilers and many C++ compilers.

[3] Maybe there *are* people out there who still write compilers in FORTRAN after all. . . .

3.1.2 Unspecified and Undefined Behavior

The following are examples of unspecified and undefined behavior:

1. The order in which the function designator and the arguments in a function call are evaluated.
2. The order in which the preprocessor concatenation operators `#` and `##` are evaluated during macro substitution.
3. The representation of floating-point types.
4. An identifier is used that is not visible in the current scope.
5. A pointer is converted to something other than an integral or pointer type.

The list is long. One of the main reasons for explicitly defining what is *not* covered by the Standard is to allow the implementor of the C environment to make use of the most efficient alternative.

3.2 POSIX

The objective of the POSIX working group P1003.1 is to define a common interface for Unix. Granted, the ANSI C standard does specify the contents of some header files and the behavior of some library functions but it falls short of defining a useful environment. This is the task of P1003.1.

We do not know how far P1003.1 addresses the problems presented in this document as at the moment we lack proper documentation. Hopefully, this will be corrected in a future release of this document.

4 Preprocessors

Preprocessors can behave differently in several ways. For those who need them, there are good publicly available preprocessors that are ANSI C–compliant. One such preprocessor is the one distributed with the X Window System developed by the MIT X Consortium.

4.1 Command Options

The interpretation of the `-I` command option can differ from one system to another. Besides, it is not covered by the Standard. For example, the directive `#include "dir/file.h"` in conjunction with `-I..` would cause most preprocessors in a Unix-like environment to search for '`file.h`' in '`../dir`', but under VMS, '`file.h`' is only searched for in the subdirectory '`dir`' in the current working directory.

4.2 `#pragma` and `#elif`

Directives are very much the same in all preprocessors, except that some preprocessors may not know about the `defined` operator in a `#if` directive nor about the `#pragma` and `#elif` directives.

The `#pragma` directive should pose no problems even to old preprocessors *if it comes indented*.[4] Furthermore, it is advisable to enclose them with `#ifdef`s in order to document under which platform they make sense:

```
#ifdef <platform-specific-symbol>
   #pragma ...
#endif
```

Beware of `#pragma` directives that alter the semantics of the program and consider the case when they are not recognized by a particular compiler. Evidently, if the behavior of the program relies on their correct interpretation then, in order for the program to be portable, all target platforms must recognize them properly.

4.3 Concatenation

Concatenation of symbols has two variants. One is the old K&R [KR78] style that simply relied on the fact that the preprocessor substituted comments such as `/**/` for nothing. Obviously, that does not result in concatenation if the preprocessor includes a space in the output. The ANSI C Standard defines the operators `##` and (implicit) concatenation of adjacent strings. Since both styles are a fact of life it is useful to include the following in one's header files:[5]

[4] Old preprocessors only take directives that begin with `#` in the first column.

[5] Some have suggested using `#if __STDC__` instead of simply `#ifdef __STDC__` to test if the compiler is ANSI-compliant because of compilers that are *not*, but define `__STDC__` equal to zero.

```
#ifdef  __STDC__
#  define  GLUE(a,b)  a##b
#else
#  define  GLUE(a,b)  a/**/b
#endif
```

If needed, one could define similar macros to `GLUE` several arguments.[6]

4.4 Token Substitution

Some preprocessors perform token substitution within quotes while others do not. Therefore, this is intrinsically non-portable. The Standard disallows it but provides a mechanism to obtain the same results. The following should work with ANSI-compliant preprocessors or with the ones that perform token substitution within quotes:

```
#ifdef  __STDC__
#  define  MAKESTRING(s)  # s
#else
#  define  MAKESTRING(s)  "s"
#endif
```

4.5 Miscellaneous

- We would *not* trust the following to work on *all* preprocessors:

  ```
  #define  D  define
  #D this that
  ```

  The Standard does not allow such a syntax (see §3.8.3 ¶20 in [X3J88]).

- Many preprocessors ignored, or still ignore, text after the `#else`, `#elif`, and `#endif` directives. However, the Standard forbids anything but comments after these directives.

- Some preprocessors will consider it an error to `#undef` something that has not been `#define`d, although it is allowed to do so.

- Finally, we must add that the Standard has fortunately included a `#error` directive with obvious semantics. Indent the `#error` since old preprocessors do not recognize it.

[6] `GLUE(a,GLUE(b,c))` would not result in the concatenation of `a`, `b`, and `c`.

5 The Language

5.1 The Syntax

The syntax defined in the Standard is a *superset* of the one defined in K&R [KR78]. It follows that if one restricts oneself to the former, there should be no problems with an ANSI C–compliant compiler *with respect to syntax*. The *semantics* are, however, another problem altogether and is covered superficially in the next section.

The Standard extends the syntax with the following:

1. The inclusion of the keywords `const`, `enum`, `signed`, `void`, and `volatile`.
2. The inclusion of additional constant suffixes to indicate their type.
3. The ellipsis ("...") notation to indicate a variable number of arguments.
4. Function prototypes.
5. Trigraph notation for specifying otherwise-unobtainable characters in restricted character sets.

We encourage the use of the reserved words `const` and `volatile` since they aid in documenting the code. It is useful to add the following to one's header files if the code must be compiled by a non-conforming compiler as well:

```
#ifndef __STDC__
#  define const
#  define volatile
#endif
```

However, one must then make sure that the behavior of the application does not depend on the presence of such keywords. (Evidently, programs that contain identifiers with those names must be modified to conform to the Standard.)

The trigraph notation can bring unexpected results when a program is compiled by an ANSI-compliant compiler, *e.g.*, strings such as `"??!"` will produce `"|"`. Watch out!

5.2 The Semantics

The syntax does not pose any problem with regard to interpretation because it can be defined precisely. However, programming languages are always described using a natural language, *e.g.*, English, and this can lead to different interpretations of the same text.

Evidently, [KR78] does not provide an unambiguous definition of the C language otherwise there would have been no need for a standard. Although the Standard is much more precise, there is still room for different interpretations in situations such as `f(p=&a, p=&b, p=&c)`. Does this mean `f(&a,&b,&c)` or `f(&c,&c,&c)`? Even "simple" cases such as `a[i] = b[i++]` are compiler-dependent [CEK+90].

As stated in the Introduction, we would like to exclude such topics. The reader is instead directed to the Usenet newsgroups **comp.std.c** or **comp.lang.c** where such discussions take place and from where the above example was taken. *The Journal of C Language Translation*[7] could, perhaps, be a good reference. Another possibility is to obtain a clarification from the Standards Committee and the address is:

X3 Secretariat, CBEMA
311 1st St NW Ste 500
Washington DC, USA

Finally, we mention that a complete list of the differences between "ordinary" C and ANSI C can be found in the Second Edition of K&R [KR88]. A slightly less up-to-date list can also be found in [HS87].

6 Unix Flavors: System V and BSD

A long time ago (1969), Unix said "`papa`" for the first time at AT&T (then called Bell Laboratories, or Ma Bell for the intimate) on a PDP-7. Everyone liked Unix very much and its widespread use we see today is probably due to the relative simplicity of its design and of its implementation. (It is written, of course, mostly in C.)

However, these facts also contributed to everyone developing their own dialect. In particular, the University of Berkeley at California distribute the so-called BSD[8] Unix whereas AT&T now distribute (sell) System V Unix. All other versions of Unix are descendants of one of these major dialects.

[7] Address is 2051, Swans Neck Way, Reston, Virginia 22091, USA.
[8] Berkeley Software Distribution

The differences between these two major flavors should not upset most application programs. In fact, we would even say that most differences are just annoying.

BSD Unix has an enhanced signal handling capability and implements sockets. However, *all* Unix flavors differ significantly in their raw I/O interface (that is, the `ioctl` system call), and this should be avoided if possible.

The reader interested in knowing more about the past and future of Unix can consult [Man89, Int90].

7 Header Files

Many useful system header files are in different places in different systems, or they define different symbols. We will assume henceforth that the application has been developed on a BSD-like Unix and must be ported to a System V-like Unix or VMS or a Unix-like system with header files that comply with the Standard.

In the following sections, we show how to handle the most simple cases that arise in practice. Some of the code that appears below was derived from the header file '`Xos.h`' which is part of the X Window System distributed by MIT. We have added changes, *e.g.*, to support VMS.

Many header files are unprotected in many systems, notably those derived from BSD version 4.2 and earlier. By "unprotected" we mean that an attempt to include a header file more than once will either cause compilation errors (*e.g.*, due to recursive or nested includes) or, in some implementations, warnings from the preprocessor stating that symbols are being redefined. It is good practice to protect header files.

7.1 '`ctype.h`'

'`ctype.h`' provides *almost* the same functionality on all systems, except that some symbols must be renamed.

```
#ifdef SYSV
#  define _ctype_  _ctype
#  define toupper  _toupper
#  define tolower  _tolower
#endif
```

Under Sys V, `toupper` and `tolower` are also defined and will check the validity of their arguments and perform the conversion only if necessary. Under

BSD-derived systems, one must normally remember to check the validity of the arguments. The following solution might be acceptable to most:

```
#ifdef SYSV
#  define  TOUPPER(c)  toupper(c)
#else /* !SYSV */
#  define  TOUPPER(c)  (islower(c)?toupper(c):(c))
#endif
```

The definitions in '<ctype.h>' *are not portable across character sets.*

7.2 'fcntl.h' and 'sys/file.h'

Many files that a BSD-like system expects to find in the 'sys' directory are placed in '/usr/include' in System V. Other systems, such as VMS, do not even have a 'sys' directory.[9]

The symbols used in the open function call are defined in different header files in the two types of systems:

```
#ifdef  SYSV
#  include <fcntl.h>
#else
#  include <sys/file.h>
#endif
```

In some systems, *e.g.*, BSD 4.3 and SunOS, it does not make a difference which one is used because both define the O_xxxx symbols.

7.3 'errno.h'

The semantics of the error number may differ from one system to another and the list may differ as well (*e.g.*, BSD systems have more error numbers than System V). Some systems, *e.g.*, SunOS, define the global symbol errno which will hold the last error detected by the run-time library. This symbol is not *declared* in most systems, although it is required by the Standard that such a symbol be defined (see §4.1.3 of [X3J88]). It is, of course, available in all Unix implementations.

The most portable way to print error messages is to use perror.

[9] Under VMS, since a path such as '<sys/file.h>' will evaluate to 'sys:file.h', it is sufficient to equate the logical name 'sys' to 'sys$library'.

7.4 'math.h'

System V has more definitions in this header file than BSD-like systems. The corresponding library has more functions as well. This header file is unprotected under VMS and Cray, and in that case we must do it ourselves:

```
#if defined(CRAY) || defined(VMS)
#  ifndef  __MATH__
#    define  __MATH__
#    include <math.h>
#  endif
#endif
```

7.5 'strings.h' *vs.* 'string.h'

Some systems cannot be treated as System V or BSD, but are really special cases, as one can see in the following:

```
#ifdef  SYSV
#  ifndef SYSV_STRINGS
#    define  SYSV_STRINGS
#  endif
#endif

#ifdef  _STDH_  /* ANSI C Standard header files */
#  ifndef SYSV_STRINGS
#    define  SYSV_STRINGS
#  endif
#endif

#ifdef  macII
#  ifndef SYSV_STRINGS
#    define  SYSV_STRINGS
#  endif
#endif

#ifdef  vms
#  ifndef SYSV_STRINGS
#    define  SYSV_STRINGS
#  endif
#endif
```

```
#ifdef  SYSV_STRINGS
#  include <string.h>
#  define  index   strchr
#  define  rindex  strrchr
#else
#  include <strings.h>
#endif
```

As one can easily observe, System V-like Unix systems use different names for `index` and `rindex` and place them in different header files. Although VMS supports better System V features, it must be treated as a special case.

7.6 'time.h' and 'types.h'

When using '`time.h`', one must also include '`types.h`'. The following code does the trick:

```
#ifdef macII
#  include <time.h>   /* on a Mac II we need this one as well */
#endif

#ifdef  SYSV
#  include <time.h>
#else
#  ifdef vms
#    include <time.h>
#  else
#    ifdef CRAY
#      ifndef __TYPES__   /* it is not protected under CRAY */
#        define __TYPES__
#        include <sys/types.h>
#      endif
#    else
#      include <sys/types.h>
#    endif /* of ifdef CRAY */
#    include <sys/time.h>
#  endif  /* of ifdef vms  */
#endif
```

The above is not sufficient in order for the code to be portable since the structure that defines time values is not the same in all systems. Different systems have vary in the way `time_t` values are represented. The Standard, for instance, only requires that it be an arithmetic type. Recognizing this difficulty, the Standard defines a function called `difftime` to compute the difference between two time values of type `time_t`, and `mktime` which takes a string and produces a value of type `time_t`.

7.7 '`varargs.h`' *vs.* '`stdarg.h`'

In some systems the definitions in both header files are contradictory. For instance, the following will produce compilation errors, *e.g.*, under VMS:

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

This is because '`<stdio.h>`' includes '`<stdarg.h>`' which in turn redefines all the symbols (`va_start`, `va_end`, etc.) in '`<varargs.h>`'. This is incorrect behavior because Standard header files should not include other Standard header files. Furthermore, the method used in '`<varargs.h>`' for defining variadic functions is incompatible with the Standard (see §11.2.3 for more information on variadic functions).

The solution we adopt is to always include '`<varargs.h>`' last and not to define in the same module both functions that use '`<varargs.h>`' and functions that use the ellipsis notation.

7.8 '`sys/wait.h`'

This one is lacking in some systems (*e.g.*, Altos and Xenix). HP-UX does define it but one must use macros to access the fields of the `wait struct`, instead of using the names of the fields. The `wait struct` uses bit-fields and if the platform does not define it one must do it oneself and care must be taken with respect to byte ordering (see **Byte ordering** in §11.1).

8 Run-time Library

This section admittedly contains very little information if compared to [Hor90]. We direct the reader to that reference for more information.

Time and time again, it happens that the target platform does not have all the library functions needed by a given application. This is particularly true with mathematical functions. We would like to remind the reader that the sources to 4.3BSD are publicly available, and may be obtained at several sites, *e.g.*, `funic.funet.fi` [128.214.6.100] in '`~ftp/pub/bsd-sources`', the contents of which are cloned from `uunet.uu.net`. Read the copyright notices before using them.

8.1 Mathematical Functions

8.1.1 `cbrt` and `pow`

`cbrt(x)` evaluates the cube root of its argument, that is, $x^{1/3}$. `pow(x,y)` evaluates x^y. Some systems implement neither of these, or just the latter. In that case, one can define `pow` as a function of `exp` and `log`, and if one has `pow` but not `cbrt`, one can write the latter as a function of the former:

```
#define  pow(x,y)  (exp(log(x)*(y)))
#define  cbrt(x)   (pow((x),1./3.))
```

Thus defined, `pow` only admits strictly positive arguments. If the argument `x` is negative, then a result can be evaluated if `y` is an integer and one must implement such a function oneself (a predicate which determines if `y` is an integer is usually not available).

The definitions given above are a "poor man's" solution to the problem but acceptable in many situations. In order to obtain numerically robust and accurate results one must investigate other alternatives such as obtaining the source code for the 4.3BSD implementation via anonymous FTP as mentioned at the beginning of this Section.

It should be mentioned that if the argument `y` is zero then implementations differ on the result. The 4.3BSD implementation returns always 1.0; others may return undefined values, flag an error, or return not-a-number.

8.1.2 `rand`

`rand` returns a pseudo-random integer in the range 0 to `RAND_MAX`, which is guaranteed only to be at least 32,767. Do not rely on `rand` returning results over a much wider range.

8.2 Memory allocation and initialization

8.2.1 `alloca`

`alloca(n)` allocates the amount of bytes specified by `n` and returns a pointer to the allocated memory. This space is — for all practical purposes — automatically deallocated (freed) when the block scope is exited. More specifically, the storage is deallocated *no sooner* than the exit from the block scope; the implementation is allowed to do the freeing at function exit, upon the next call to `alloca`, or at any other moment deemed appropriate. The example below illustrates *incorrect* usage of `alloca`:

```
foo ()
{
 char *sto;
    {
      sto = alloca (10);
      use (sto);  /* Correct. */
    }
    use (sto); /* Error: storage may have been freed. */
}
```

Conceptually, the space is allocated on a stack, so allocation can be as fast as just adjusting the stack pointer if the machine has one, and several regions can be freed at once by simply readjusting the stack pointer. However, it is hard to implement `alloca` both portably and efficiently. `alloca` is not available on all platforms and as such is not required by the Standard. However, there are public domain implementations that work in a wide variety of cases, but which can be slow and which can delay freeing arbitrarily[10].

Thus, while it is very desirable to use `alloca` when it is available, because of efficiency considerations, it is highly recommended that the code be written so that `malloc` and `free` can easily replace it, if and when necessary.

8.2.2 `bcopy` *vs.* `memcpy` and `memmove`

`bcopy(s1,s2,n)` copies the string `s1` into `s2`, whereas `memcpy(s1,s2,n)` copies `s2` into `s1`. `bcopy` can be found in BSD-like systems, and some implementations handle overlapping strings, while others do not. `memcpy` and `memmove` are implemented in the other camp (System V); `memcpy` does not handle overlapping strings, whereas `memmove` does.

The normal solution is to use macros.

8.2.3 `bzero` *vs.* `memset`

`bzero(s,n)` is equivalent to `memset(s,0,n)`. The former is implemented in BSD-like systems, whereas the latter is implemented in System V-like systems and is required by the Standard.

See also **Initialization** in §11.2.4.

[10]A public domain implementation of `alloca` can be obtained from the Free Software Foundation (GNU); try `prep.ai.mit.edu` in '`~ftp/pub/gnu`'.

8.2.4 `malloc` and `free`

`malloc` is available in all C implementations and its behavior is very well defined except in boundary conditions. Not all implementations accept a zero-sized request. There are other minor differences such as the return type being `char *` in some implementations and `void *` in others.

In a similar vein, some implementations of `free` do not accept `NULL` as an argument. Worse, though, is that some implementations allowed the caller to use the pointer even *after* it had been `free`d so long as no other call to `malloc` was performed. Relying on such behavior is bad.

8.2.5 `realloc`

`realloc(sto,n)` takes a pointer to a region allocated with `malloc` and grows or shrinks the region so that it is of size `n`. The return value from `realloc` is a pointer to the resized storage; if the storage was grown "in place", the return value is the same as `sto`. If the region was moved, then the old contents are copied to the new storage (if `n` is smaller than the old size, then only the first `n` units are copied). If the region is grown, the new storage at the end is uninitialized and may contain garbage.

Under ANSI C:

- If `sto == NULL`, then `realloc` acts like `malloc`.
- If `n == 0`, then `realloc` acts like `free`.
- If `sto == NULL` *and* `n == 0`, the results are undefined.

For non-ANSI versions of `realloc`, specifying `NULL` as the storage or `0` as the new size causes undefined behavior. Thus, it is recommended that portable programs, *even those written in ANSI C*, not use these features. If it is necessary to rely on those features, use a macro or write a function that can be configured to check for those cases explicitly.

8.3 Miscellaneous

8.3.1 `scanf`

`scanf` can behave differently on different platforms because its descriptions, including the one in the Standard, allows for different interpretations under some circumstances. The most portable input parser is the one you write yourself.

Some versions of the `scanf` family modify and then restore arguments which are string constants. These implementations cause problems when string constants are placed in read-only memory (see “String constants” in §11.2.4). If the string is actually a constant, then some workaround is needed; usually a compiler flag may be used to indicate that such constants should be placed in writable memory instead. If such a flag is not available then the code must be modified.

8.3.2 `setjmp` and `longjmp`

Quoting anonymously from **comp.std.c**, “pre-X3.159 implementations of `setjmp` and `longjmp` often did not meet the requirements of the Standard. Often they didn’t even meet their own documented specs. And the specs varied from system to system. Thus it is wise not to depend too heavily on the exact standard semantics for this facility…”.

In other words, it is not that you should *not* use them but be careful if you do. Furthermore, the behavior of a `longjmp` invoked from a nested signal handler[11] is undefined.

Finally, the symbols `_setjmp` and `_longjmp` are only defined under SunOS, BSD, and HP-UX. Some systems do not implement `setjmp` and friends at all.

8.3.3 Signal Handling

We would like to point out one problem when handling signals generated by hardware, such as `SIGFPE` and `SIGSEGV`. There are two possibilities on a normal exit from the signal handler: (i) the offending instruction is re-executed, or (ii) it is not.

The first possibility may cause an infinite loop, and the only portable solution is to `longjmp` out of the signal handler.

9 Using Floating-Point Numbers

To say that the implementation of numerical algorithms that exhibit the same behavior across a wide variety of platforms is difficult, is an understatement. This section provides very little help but we hope it is worth reading. Any additional suggestions and information are *very much* appreciated as we would like to expand this section.

[11] That is, a function invoked as a result of a signal raised during the handling of another signal. See §4.6.2.1 ¶15 in [X3J88].

9.1 Machine Constants

One problem when writing numerical algorithms is obtaining machine constants. Typical values one needs are:

- The radix of the floating-point representation.
- The number of digits in the floating-point significand expressed in terms of the radix of the representation.
- The number of bits reserved for the representation of the exponent.
- The smallest positive floating-point number ϵ such that $1.0 + \epsilon \neq 1.0$.
- The smallest non-vanishing normalized floating-point power of the radix.
- The largest finite[12] floating-point number.

On Suns, they can be obtained in '`<values.h>`'. The ANSI C Standard recommends that such constants be defined in the header file '`<float.h>`'.

Suns and standards apart, these values are not always readily available, *e.g.*, in Tektronix workstations running UTek. One solution is to use a modified version of a program that can be obtained from the network which is called `machar`. `Machar` is described in [Cod88] and can obtained by anonymous FTP from the `netlib`.[13]

It is straightforward to modify the C version of `machar` to generate a C preprocessor file that can be included directly by C programs.

There is also a publicly available program called '`config.c`' that attempts to determine many properties of the C compiler and machine that it is run on. It can generate the ANSI C header files '`<float.h>`' and '`<limits.h>`' among other useful features. This program was submitted to **comp.sources.misc**.[14] The latest version, 4.2, is available by FTP from `mcsun.eu.net` in directory '`misc`' and is called '`config42.c`' (the next version, 4.3, will be called '`enquire.c`'). Version 4.2 is also distributed with `gcc`, where it is called '`hard-params.c`'.

---

[12] Some representations have reserved values for $+inf$ and $-inf$.

[13] Email (Internet) address is `netlib@ornl.gov`. For more information, send a message containing the line `send index` to that address.

[14] The archive site of **comp.sources.misc** is `uunet.uu.net`.

9.2 Floating-Point Arguments

In the days of K&R [KR78] one was "encouraged" to use `float` and `double` interchangeably[15] since all expressions with such data types where always evaluated using the `double` representation — a real nightmare for those implementing efficient numerical algorithms in C. This rule applied, in particular, to floating-point arguments and for most compilers around, it does not matter whether one defines the argument as `float` or `double`.

According to the ANSI C Standard, such programs will continue to exhibit the same behavior *as long as one does not prototype the function.* Therefore, when prototyping functions, make sure that the prototype is included when the function definition is compiled so the compiler can check if the arguments match.

9.3 Floating-Point Arithmetic

Be careful when using the `==` and `!=` operators to compare floating-point types. Expressions such as

`if (`*float_expr1* `==` *float_expr2*`)`

will seldom be satisfied due to *rounding errors.* To get a feeling about rounding errors, try evaluating the following expression using your favorite C compiler [KM86]:

$$10^{50} + 812 - 10^{50} + 10^{55} + 511 - 10^{55} = 812 + 511 = 1323$$

Most computers will produce zero regardless of whether one uses `float` or `double`. Although the *absolute error* is large, the *relative error* is quite small and probably acceptable for many applications.

It is rather better to use expressions such as $|\mathit{float\_expr1} - \mathit{float\_expr2}| \le K$ or $\Big||\mathit{float\_expr1}/\mathit{float\_expr2}| - 1.0\Big| \le K$ (if $\mathit{float\_expr2} \neq 0.0$), where $0 < K < 1$ is a function of:

1. The floating type, *e.g.*, `float` or `double`,
2. the machine architecture (the machine constants defined in the previous section), and
3. the precision of the input values and the rounding errors introduced by the numerical method used.

[15] In fact one wonders why they even bothered to define two representations for floating-point numbers considering the rules applied to them.

Other possibilities exist and the choice depends on the application.

The development of reliable and robust numerical algorithms is a very difficult undertaking. Methods for certifying that the results are correct within reasonable bounds must usually be implemented. A reference such as [PFTV88] is always useful.

- Keep in mind that the `double` representation does not necessarily increase the *precision*. Actually, in some implementations the precision decreases, but the *range* increases.

- Do not use `double` unnecessarily, since in many cases there is a large performance penalty. Furthermore, there is no point in using higher precision, if the additional bits that would be computed are garbage anyway. The precision one needs depends mostly on the precision of the input data and the numerical method used.

9.4 Exceptions

Floating-point exceptions (overflow, underflow, division by zero, etc) are not signaled automatically in some systems. In that case, they must be explicitly enabled.

Always enable floating-point exceptions, since they may be an indication that the method is unstable. Otherwise, one must be sure that such events do not affect the output.

10 VMS

In this section, we will report some common problems encountered when porting a C program to a VMS environment and which we have not mentioned previously.

10.1 File Specifications

Under VMS, one can use two flavors of command interpreters: DCL and DEC/Shell. The syntax of file specifications under DCL differs significantly from the Unix syntax.

Some C run-time library functions in VMS that take file specifications as arguments or return file specifications to the caller, will accept an additional argument indicating which syntax is preferred. It is useful to use these run-time library functions via macros as follows:

```
#ifdef  VMS
#  ifndef VMS_CI         /* Which Command Interpreter to use */
#    define VMS_CI  0   /* 0 for DEC/Shell, 1 for DCL */
#  endif

#  define  Getcwd(buff,siz)   getcwd((buff),(siz),VMS_CI)
#  define  Getname(fd,buff)   getname((fd),(buff),VMS_CI)
#  define  Fgetname(fp,buff)  fgetname((fp),(buff),VMS_CI)

#else  /* !VMS */
#  define  Getcwd(buff,siz)   getcwd((buff),(siz))
#  define  Getname(fd,buff)   getname((fd),(buff))
#  define  Fgetname(fp,buff)  fgetname((fp),(buff))

#endif /* !VMS */
```

More pitfalls await the unaware who accept file specifications from the user or take them from environment values (*e.g.*, using the `getenv` function).

10.2 Miscellaneous

`end`, `etext`, `edata`: these global symbols are not available under VMS.

`struct` assignments: VAX C allows assignment of different types of `struct`s if both types have the same size. *This is not a portable feature.*

The system function: the `system` function under VMS has the same *functionality* as the Unix version, except that one must take care that the command interpreter also provides the same functionality. If the user is using DCL, then the application must send a DCL-like command.

The linker: what follows applies only to modules stored in libraries.[16] If none of the global *functions* are explicitly used (referenced by another module), then the module is not linked *at all.* It does not matter whether one of the global *variables* is used. As a side effect, the initialization of variables is not done.

The easiest solution is to force the linker to add the module using the `/INCLUDE` command modifier. Of course, there is the possibility that the command line may exceed 256 characters... (*sigh*).

[16] This does not really belong in this document, but whenever one is porting a program to a VMS environment one is bound to come across this strange behavior which can result in a lot of wasted time.

11 General Guidelines

11.1 Types and Pointers

Type sizes: *Never* make any assumptions about the size of a given type, especially pointers [CEK+90]. Statements such as `x &= 0177770` make implicit use of the size of `x`. If the intention is to clear the lowest three bits, then it is best to use `x &= ~07`. The first alternative will also clear the high-order 16 bits if `x` is 32 bits wide.

Byte ordering: There are two possibilities for byte ordering: *little-endian* and *big-endian* architectures. This problem is illustrated by the code below:

```
long int str[2] = {0x41424344, 0x0}; /* ASCII "ABCD" */
printf ("%s\n", (char *)&str);
```

A little-endian (*e.g.*, VAX) will print "`DCBA`" whereas a big-endian (*e.g.*, MC68000 microprocessors) will print "`ABCD`". (As a side note, there is also *PDP-endian* that would print "`BADC`", followed by many smileys.)

Note: The example will only function correctly if `sizeof(long int)` is 32 bits. Although not portable, it serves well as an example for the given problem.

Alignment constraints: Beware of alignment constraints when allocating memory and using pointers. Some architectures restrict the addresses that certain operands may be assigned to (that is, addresses of the form $2^k E$, where $k > 0$). Code such as

```
char *s = "bla"; /* allocated by compiler */
int  *v = (int *)s;
```

would most probably fail if the alignment constraints of `int` types are more strict than those of `char` types (the usual case for RISC architectures). The code would not fail due to alignment constraints if the memory indicated by `s` had been allocated by `malloc` and friends.

Pointer formats: [CEK+90] Pointers to objects may have the same size but different formats. This is illustrated by the code below:

```
int *p = (int *) malloc(...); ... free(p);
```

This code may malfunction in architectures where `int *` and `char *` have different representations because `free` expects a pointer of the latter type.

Pointers to different types of objects may have different sizes as well. For instance, there are platforms where a `char *` is larger than an `int *` or where a pointer to a function will not fit in, *e.g.*, `char *` or `void *` (although such cross-assignments work on many platforms, `void *` is only guaranteed to be large enough to hold a pointer to any *data* object). Therefore, it is not portable to assign to an object of type `void *` a pointer to a function. Pointers to functions are further discussed below.

Pointers to functions If you need a generic function pointer, then use `void(*)(void)`. Be sure to cast the pointer back to the original type before using it. That is, the type signature of the function pointer at the point that the function is called must *exactly* match the type signature at the point at which the function is defined.

For example, it is not possible to (portably) use `varargs` functions[17] (that is, functions that take a variable number of arguments) and fixed-argument functions interchangeably, even if the overlapping types match (that is, even if the first n arguments to the fixed-argument function are the same as the first n arguments to the `varargs` function). For instance, a function that is declared as having an integer as the first argument and an optional (integer) second argument cannot be called as a function that takes two integer arguments. Similarly, `varargs` functions of various type signatures cannot be interchanged. Such type cheating will break on systems that use different conventions for calling fixed-argument and `varargs` functions and on systems that use different conventions for passing the fixed and `varargs` parts of the argument lists.

As a corollary, it is necessary that the definitions of external variadic functions be available at the point of their usage, *e.g.*, library functions such as `printf`.

Pointer operators: [CEK+90] Only the operators `==` and `!=` are defined for all pointers of a given type. The remaining comparison operators (`<`, `<=`, `>`, and `>=`) can only be used when both operands point into the same array or to the first element after the array. The same applies to arithmetic operators on pointers.[18]

[17] There is a difference between variadic functions defined by the Standard and the pre-Standard `varargs` as defined by '`varargs.h`' which is still widely used. Here we are referring to the former, and the differences between both are explored in §11.2.3.

[18] One of the reasons for these rules is that in some architectures, pointers are represented

`NULL` **pointer:** *Never* redefine the `NULL` symbol. The `NULL` symbol should always be the *constant* zero. A null pointer of a given type will always compare equal to the *constant* zero, whereas comparison with a *variable* with value zero or to some non-zero constant has implementation-defined behavior. (In other words, the constant zero has two meanings.)

A null pointer of a given type will always convert to a null pointer of another type if implicit or explicit conversion is performed. (See 'Pointer Operators' above.)

The contents of a null pointer may be anything the implementor wishes, and dereferencing it may cause strange things to happen....

11.2 Compiler Differences

11.2.1 Conversion Rules

In arithmetic expressions, integral types may be converted in two ways: *unsigned-preserving* or *value-preserving*. In the unsigned-preserving model, `char`s, `short`s, and bit-fields are converted to `unsigned int` or `signed int` if the original types have the modifiers `unsigned` or `signed`, respectively.

The Standard determines that the value-preserving model must be used, meaning that `unsigned` values are promoted to `signed int`, or simply `int`, if it can represent all the values of the original type; otherwise it is converted to `unsigned int`. (See §3.2 of the Standard.)

The following example illustrates the problem. On a machine with a 16-bit `short int`, and 32-bit `int`, the code fragment

```
unsigned short int x = 1;
if (x < -1) printf ("unsigned-preserving");
else printf ("value-preserving");
```

prints `unsigned-` or `value-preserving` accordingly. Plenty of other examples can be derived, such as initializing `x` with 2^{15} and using the predicate `(x*x*2 > 0)`. The expression `x*x*2` would probably result in the same bit pattern in both models but would cause arithmetic overflow in the value-preserving model.

---

as a pair of values and only equality is a well-defined operator for arbitrary pairs of values. The other operators are only well-defined when one of the values of both pairs is guaranteed to match, in which case the situation is analogous to "ordinary" architectures.

11.2.2 Compiler Limitations

In practice, much too frequently one runs into several, unstated compiler limitations:

- Some of these *limitations* are *bugs*. Many of these bugs are in the optimizer and therefore when dealing with a new environment it is best to explicitly disable optimization until one gets the application "going".
- Some compilers cannot handle large modules or "large" statements.[19] Therefore, it is advisable to keep the size of modules within reasonable bounds. Besides, large modules are more cumbersome to edit and understand.

11.2.3 ANSI C

The Standard has introduced and officialized current practice, but as we all know not many compilers conform to the Standard. Among the features that are not yet widely supported, we mention here only a few:

Constant suffixes: Many compilers allow for suffixes to be appended to constants, such as `10L` to indicate a `long` constant. The Standard allows further typing of constants, such as `10UL` to indicate an `unsigned long` constant. However, multiple suffixes are not supported by many compilers.

New types: Besides the type `void *` which is mentioned in the next section, the Standard has introduced the type `long double`.

Variadic functions: Variadic functions, as defined by the Standard, differ significantly from '`<varargs.h>`'. Besides the ellipsis notation, it is required by the Standard that the first argument be identified and that '`<stdarg.h>`' be used instead (see §7.7). Therefore, it is not possible to define a variadic function which takes no arguments.

11.2.4 Miscellaneous

`char` **types:** When `char` types are used in expressions, most implementations will treat them as `unsigned` *but there are many others that treat them as* `signed` (*e.g.*, VAX C and HP-UX). It is advisable to always cast `char`s when they are used in arithmetic expressions.

[19] Programs that generate other programs, *e.g.*, `yacc`, can generate, for instance, very large `switch` statements.

Initialization: Do not rely on the initialization of `auto` variables and of memory returned by `malloc`. In particular, since not all `NULL` pointers are represented by a bit pattern of all-zeroes, it is good practice to always initialize pointers appropriately.

The `calloc` library function returns an area of memory that has been cleared to zero. Although this can be used to initialize arrays and `structs` on many architectures, not all architectures represent `NULL` pointers internally with a zero bit-pattern. Similarly, it is not safe to assume that all architectures represent the floating-point constant `0.0` using a zero bit-pattern.

The semantics of many library functions differ from system to system. Also, the specifications of some library functions have been changed in the ANSI C Standard. For example, `realloc` is now required to behave like `malloc` when called with a `NULL` argument; formerly, many implementations would dump core if handed `NULL`.

Bit fields: Some compilers, *e.g.*, VAX C, require that bit fields within `struct`s be of type `int` or `unsigned`. Furthermore, the upper bound on the length of the bit field may differ among different implementations.

`sizeof`:
1. The result of `sizeof` may be `unsigned` or `signed`.
2. If `p` is a pointer, then `sizeof(*p)` is allowed by the Standard and many compilers even if `p` does not contain a valid address such as `NULL`. However, some compilers dereference the pointer causing programs to crash.

`void` and `void *`: Some very old compilers do not recognize `void` [*sic*]. Although required by the Standard, some compilers recognize `void` but fail to recognize `void *`. The following code might prove useful:

```
#if __STDC__
#  define  HAS_VOIDP
#endif
#ifdef HAS_VOIDP
   typedef void *voidp;
#else
   typedef char *voidp;
#endif
```

Functions as arguments: When calling functions passed as arguments, always dereference the pointer. In other words, if `f` is a pointer to a function, use `(*f)()` instead of simply `(f)()`, because some compilers may not recognize the latter.

String constants: Do not modify string constants since many implementations place them in read-only memory. Furthermore, that is what the Standard requires — and that is how a constant should behave!

Note: In statements such as "`char *s = "string"`", `"string"` is a string constant, whereas in "`char s[] = "string"` it is not and it is legal to modify `s`.

`struct` comparisons: Some compilers might allow for `struct`s to be compared for equality or inequality. Such an extension is not included in the Standard (meaning it is not portable).

Initialization of aggregates: Some compilers cannot initialize `auto` aggregate types. Statements such as:

```
{
   typedef struct {double x,y} Interval;
   Interval range = {0.0,0.0};
   ...
}
```

are not allowed by some compilers unless the modifier `static` is used or if `range` has file scope. Although declaring all such variables `static` would handle most situations, the most portable solution is to add code that performs the initialization.

Nested comments: Nested comments were never allowed in the C language, but they are allowed by some compilers. Nested comments are used by some to comment out source code containing comments. However, the same effect can be obtained using an `#if 0` and `#endif` pair.

Shift operators: When shifting `signed int`s right, the vacated bits might be filled with zeroes or with copies of the sign bit. `unsigned int`s will be filled with zeroes.

Division and remainder: When both operands are non-negative, then the remainder is non-negative and smaller than the divisor; if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor.

11.3 Files

11.3.1 General Guidelines

Remember that not all operating systems share Unix's simple notion of a file as a stream of bytes. MS-DOS, for instance, has text files and binary files; it is important to open files in the correct mode. VMS has many different file types and each file is viewed as being a collection of structured records.

MS-DOS provides a "poor man's" implementation of pipes and redirection. It does not expand wildcards, however. The user must do the wildcard expansion using `findfirst` and `findnext`. Under VMS, the user must also expand wildcards, and parse `argv` for redirection directives manually.

Different operating systems use widely different syntax to specify pathnames. This is a potential source of problems. Some compilers may provide run-time pathname translation to translate between Unix syntax and the host's syntax.

11.3.2 Source Files

- Keep files reasonably small in order not to upset some compilers.
- File names should not exceed 14 characters (many System V-derived system impose this limit, whereas in BSD-derived systems a limit of 15 is usually the case). In some implementations this limit can be as low as 8 characters. These limits are often *not* imposed by the operating system but by system utilities such as `ar`.
- Do not use special characters especially multiple dots (dots have a very special meaning under VMS).

11.4 Miscellaneous

System dependencies: Isolate system-dependent code in separate modules and use conditional compilation.

Utilities: Utilities for compiling and linking such as `Make` simplify considerably the task of moving an application from one environment to another. Even better, use `Imake` since `Make` files are very unportable. `Imake` is distributed with the X Window System by MIT. One of the authors of this document has used it extensively with very good results.

Many of the tools and libraries that one takes for granted on Unix, such as `lex`, `yacc`, `curses`, `sed`, `awk`, and the various shells, are often not available on other operating systems. Public-domain versions of most of the useful tools are available at many archive sites. However, the so-called copyleft restrictions on many of these programs may prove to be problematic to some would-be porters.

Name space pollution: Minimize the number of global symbols in the application. One of the benefits is the lower probability that any conflicts will arise with system-defined functions.

Character sets: Do not assume that the character set is ASCII. If the character set in question is not [American] English, then other characters will also be alphabetic, and their lexicographic ordering will not necessarily have any relationship to their positions within the character set. If the character set is Asian, then "characters" may be of type `wchar_t`, not `char`, and will, in general, require two or more bytes of storage each. The library string functions should be capable of handling these correctly. Code that iterates through arrays of `char`s may need to be changed to handle multibyte characters correctly.

If the program's messages are likely to be translated into other languages, take care to modularize the code for easy translation. Consider keeping all text in a "language" file. Be aware that carefully formatted reports and printing routines may need major surgery.

Binary Data: Great care must be taken when reading and writing binary data. For example, a file of floating-point numbers in binary format written by machine A is unlikely to be usable on machine B.

11.5 Writing Portable Code

Write code under the assumption that it will be ported to many strange machines. It is considerably easier to port code to a new environment when the code has been written with porting in mind, than it is to "retrofit" portability.

One school of thought advocates "Port early, port often." That is, whenever the code reaches a certain level of stability on the development system, port it to other systems. This method has the advantage that portability problems are discovered early, and the possible disadvantage that potentially far more time could be spent in porting than would be the case if the code were just ported once, when complete.

Code in ANSI C whenever possible. Many of the extensions — prototypes, stronger type-checking, etc. — enhance portability. The more widely ANSI C is used, the quicker it will gain acceptance. Of course, this may not be an option if the code must be ported to platforms without ANSI C compilers. The short-term solution is to use the various tricks discussed in [CEK+90] and elsewhere; the long-term solution is to force vendors to release ANSI C compilers for their systems. Alternatively, a converter such as `protoize` (available via anonymous FTP from `prep.ai.mit.edu`) can convert between ANSI and non-ANSI programs.

Make complete, correct declarations; don't let parameters default to `int`. Include all of the necessary header files. Declare functions with no return value as `void`. Check the results of system calls.

Use `lint`. Programs that fail to pass `lint` quietly will undoubtedly be difficult to port. Compile code with as many different compilers as possible with all warnings enabled.

[CEK+90] has more to say about this.

12 Further Reading

One can argue that portability and "well-written" code go hand-in-hand. Loosely defined, well-written code is one that is "easy" to understand *and* "easy" to maintain, and there are several style guides in the public domain expressing various views on the subject.

Besides the style guide mentioned in the foreword, there are a few more that can be obtained in `cs.toronto.edu` [128.100.1.65] in '`~ftp/doc/programming`'. We also recommend '`standards.text`' from the Free Software Foundation which can be found in various sites, *e.g.*, `prep.ai.mit.edu` [18.71.0.38] in '`~ftp/pub/gnu`'.

For those who have access to the Usenet newsgroup **comp.lang.c**, we highly recommend reading the Frequently Asked Questions List (known as the *FAQL*) which is posted at the beginning of every month.

13 Acknowledgements

We are grateful for the early help of A. Louko (HTKK/Lsk) and J. Helminen (HTKK). The following persons have commented on and corrected previous revisions of this document: Geoffrey H. Cooper and Guy Harris. Special thanks go to Steven Pemberton, the main author of '`config.c`', for making available such a useful tool. We thank all the contributors to the Usenet newsgroups **comp.std.c** and **comp.lang.c** from where we have taken a lot of information. Some information within was obtained from [Hew88].

14 Trademarks

References

[CEK+90] L. W. Cannon, R. A. Elliot, L. W. Kirchoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittinton, Henry Spencer, David Keppel, and Mark Brader. Recommended C Style and Coding Standards. Technical report, in the public domain, June 1990.

[Cod88] W. J. Cody. Algorithm 665, MACHAR: A Subroutine to Dynamically Determine Machine Parameters. *ACM Transactions on Mathematical Software*, 14(4):303–311, December 1988.

[Hew88] Hewlett-Packard Company. *HP-UX Portability Guide*, 1988.

[Hor90] Mark Horton. *Portable C Software*. Prentice-Hall, 1990.

[HS87] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Prentice-Hall, Inc., second edition, 1987.

[Int90] Interviews. Interview With Five Technologists. *UNIX Review*, 8(1):41–89, January 1990.

[KM86] U. W. Kulish and W. L. Miranker. The Arithmetic of the Digital Computer: A New Approach. *SIAM Review*, 28(1):1–40, March 1986.

[Koe89] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley Publishing Co., Reading, Massachusetts, 1989.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., first edition, 1978.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., second edition, 1988.

[Man89] Tom Manuel. A Single Standard Emerges from the UNIX Tug-Of-War. *Electronics*, pages 141–143, January 1989.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *NUMERICAL RECIPES in C: The Art of Scientific Computing*. Cambridge University Press, 1988.

[X3J88] X3J11. Draft Proposed American National Standard for Information Systems — Programming Language C. Technical Report X3J11/88–158, ANSI Accredited Standards Committee, X3 Information Processing Systems, December 1988.