Notes On Writing Portable Programs In C

(June 1990, 5th Revision)

A. Dolenc
A. Lemmke *
and
D. Keppel †

September 7, 2000

Contents

*Helsinki University of Technology, Laboratory of Information Processing Sciences, SF-02150 Espoo, Finland. This document is in the public domain. Email address (Internet) are ado@sauna.hut.fi (preferred contact) and arl@sauna.hut.fi, respectively.

†CS&E, University of Washington. Email address (Internet) is pardo@cs.washington.edu.

1 Foreword

A few words about the intended audience before we begin. 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 done some porting before you may not find the information herein very useful.

We suggest that [Can89] be read in conjunction with this document[1]. Submitters to the News group **comp.lang.c** have repeatedly recommended [Hor90, Koe89][2].

Disclaimer: The code fragments presented herein are intended to make applications "more" portable, meaning that they may fail with some compilers and/or environments.

This file can be obtained via anonymous ftp from *sauna.hut.fi [130.233.251.253]* in ~ *ftp/pub/CompSciLab/doc*. The files *portableC.tex*, *portableC.bib* and *portableC.ps.Z* are the LaTeX, BibTeX and the compressed PostScript, respectively.

2 Introduction

The aim of this document is to collect the experience of several people who have had to write and/or port programs 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 which have been written to run on several platforms. This is the case of publicly available software such as developed by the Free Software Foundation and the MIT X Consortium.

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 of 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. System programming[3] and twisted code associated with bizarre interpretations of [X3J88] – henceforth refered to as the Standard – will not be extensively covered in this document[4].

[1] It can be obtained via anonymous *ftp* from *cs.washington.edu* in ~*ftp/pub/cstyle.tar.Z*.

[2] We note here that none of the information herein as been taken from those two references.

[3] We include raw I/O, e.g. from terminals in this category.

[4] 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.

3 Standardization Efforts

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

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 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 attention the fact that the Standard states some environmental limits. These limits are *lower bounds*, meaning that a correct (compliant) compiler may refuse to compile an otherwise correct program which exceeds one of those limits[5].

Below are the limits which 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* `#included` *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

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

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 em 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 which 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 which *generate* C code such as compiler-compilers and many C++ compilers.

3.1.2 Unspecified and undefined behaviour

The following are examples of unspecified and undefined behaviour:

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 types.
4. An identifier is used that is not visible in the current scope.
5. A pointer is converted to 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 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 behaviour of some library functions but it falls short of defining a usefull 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 may present different behaviour in the following:

1. 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.

2. 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 section 3.8.3 §20 in [X3J88]).

3. 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` directive.

   The `#pragma` directive should pose no problems even to old preprocessors *if it comes indented*[6]. 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
   ```

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

4. Concatenation of symbols has two variants. One is the old K&R style which 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[7]:

```
#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 [8].

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

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

There are good publicly available preprocessors which are ANSI C compliant. One such preprocessor is the one distributed with the X Window System developed by the MIT X Consortium.

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

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.

[7] Some have suggeested using `#if __STDC__ == 1` instead of simply `#ifdef __STDC__` to test if the compiler is ANSI-compliant.

[8] `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. It follows that if one restricts oneself to the former there should be no problems with an ANSI C compliant compiler. The Standard extends the syntax with the following:

1. The inclusion of the keywords **const** and **volatile**.
2. The ellipsis ("...") notation to indicate a variable number of arguments.
3. Function prototypes.
4. Trigraph notation for specifying "wide" character strings.

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 an non-conforming compiler as well:

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

However, one must then make sure that the behaviour of the application does not depend on the presence of such keywords.

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 [Can89].

As stated in the Introduction we would like to exclude such topics. The reader is instead directed to the USENET news group **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*[9] 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

6 Unix flavours: 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-11. 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 for each one to develop their own dialect. In particular, the University of Berkeley at California distribute the so-called BSD[10] Unix whereas AT&T distribute (sell) System V Unix. All other vendors are descendants of one of these major dialects.

The differences between these two major flavours 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 flavours differ significantly in their raw i/o interface (that is, **ioctl** system call) which 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 an Unix-like system with header files which comply to the Standard.

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

In the following sections, we show how to handle the most simple cases which arise in practice. Some of the code which 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 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`

They provide the same functionality in all systems except that some symbols must be renamed.

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

Note however that the definitions in `<ctype.h>` are not portable across character sets.

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

Many files which a BSD systems expects to find in the `sys` directory are placed in `/usr/include` in System V. Other systems, like VMS, do not even have a `sys` directory[11].

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

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

[11] 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`.

7.3 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 available in most systems, although the Standard requires that such a symbol be defined (see section 4.1.3 of [X3J88]).

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

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 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 a special case, 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 which 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 under VMS[12]:

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

8 Run-time Library

getlogin: This one is not defined, e.g. under VMS. In that case, one can always use `getenv(``HOME'')`.

scanf: Scanf can behave differently in different platforms because it's 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.

[12] We are not sure this behaviour occurs only under VMS.

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 behaviour of a **longjmp** invoked from a nested signal handler[13] is undefined.

Finally, the symbols `_setjmp` and `_longjmp` are only defined under SunOS, BSD, and HP-UX.

9 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[14]. Therefore, it is advisable to keep the size of modules within reasonable bounds. Besides, large modules are more cumbersome to edit and understand.

10 Using floating-point numbers

To say that the implementation of numerical algorithms which exhibit the same behaviour 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 is *very much* appreciated as we would like to expand this section.

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

[14] Programs which generate other programs, e.g. YACC, can generate, for instance, very large **switch** statements.

10.1 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 *eps* such that $1.0 + eps \neq 1.0$.
- The smallest non-vanishing normalized floating-point power of the radix.
- The largest finite[15] floating-point number.

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

Sun's 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 which can be obtained from the network called **machar**. **Machar** is described in [Cod88] and can obtained by anonymous *ftp* from the *netlib*[16].

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

There is also a publicly available program called *config.c* which attempts to determine many properties of the C compiler and machine that it is run on. This program was submitted to **comp.sources.misc**[17].

10.2 Floating-point arguments

In the days of K&R[KR78] one was "encouraged" to use *float* and *double* interchangeably[18] 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 compiler around it does not matter whether one defines the argument as *float* or *double*.

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

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

[17] The arquive site of **comp.sources.misc** is *uunet.uu.net*.

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

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

10.3 Floating-point arithmetic

Be careful when using the `==` and `!=` operators when comparing floating 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 favourite C compiler[KM86]:

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

Most computers will produce zero regardless if 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 $|float\_expr1 - float\_expr2| \leq K$ or $\left|\left|\frac{float\_expr1}{float\_expr2}\right| - 1.0\right| \leq K$ (if $float\_expr2 \neq 0.$), 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.

Other possibilities exist and the choice depends on the application.

The development of reliable and robust numerical algorithm 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 most implementations the precision decreases but the *range* increases.

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

10.4 Exceptions

Floating-point exceptions (overlow, 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.

11 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 in the previously.

11.1 File specifications

Under VMS one can use two flavours 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 which 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 flavour 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
# define  Getcwd(buff,siz)   getcwd((buff),(siz))
# define  Getname(fd,buff)   getname((fd),(buff))
# define  Fgetname(fp,buff)  fgetname((fp),(buff))

#endif /* of ifdef 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).

11.2 Miscellaneous

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

`Struct` **assignments:** VAX C allows assignment of structs if the types of both sides 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 provide also 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[19]. 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*).

12 General Guidelines

12.1 Machine architectures, Type compatibility, Pointers, etc.

1. **Never** make any assumptions about the size of a given type, especially pointers. [Can89] Statements such as `x &= 0177770` make implicit use of the size of `x`. If the intention is to clear the lower three bits then it is best

[19] 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 behaviour which can result in a lot of wasted time.

to use `x &= ~07`. The first alternative will also clear the high order 16 bits if `x` is 32 bits wide.

2. In some architectures the byte order is inverted; these are called *little-endian* versus *big-endian* architectures. This problem is illustrated by the code below[20]:

   ```
   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`".

3. 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 $2^k E, k > 0$).

4. [Can89] 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.

5. [Can89] 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[21].

6. **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 behaviour.

   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 item 4 above.)

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

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

[21] One of the reasons for these rules is that in some architectures pointers are represented as a pair of values and only under those circumstances are two pairs comparable.

12.2 Compiler differences

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

2. Do not rely on the initialization of `auto` variables and of memory returned by `malloc`.

3. Some compilers, e.g. VAX C, require that bit fields within `structs` be of type `int` or `unsigned`. Futhermore, the upper bound on the length of the bit field may differ among different implementations.

4. The result of `sizeof` may be `unsigned`.

12.3 Files

1. Keep files reasonably small in order not to upset some compilers.

2. 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*.

3. Do not use special characters especially multiple dots (dots have a very special meaning under VMS).

12.4 Miscellaneous

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.

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.

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.

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!

13 Acknowledgements

We are grateful for the help of Antti Louko (HTKK/Lsk) and Jari Helminen (HTKK) in commenting and correcting a previous draft of this document. We thank all the contributors of USENET News groups **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

[Can89] L.W. Cannon. Recommended C Style and Coding Standards. Technical report, November 1989.

[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.

[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., 1978.

[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.