

"C and Portability "

by

Les Hatton¹

Version #1.2, 25th. November, 1996

¹ Oakwood Computing, Oakwood, 11 Carlton Road, New Malden, Surrey KT3 3AJ, U.K.
lesh@oakcomp.demon.co.uk

A few asides on portability

Before getting down to the real business of linguistic polemics, I thought I would like to talk a little about portability in general. Portability has been one of the primary goals of every system I have been involved with in the last 15 years from million line Fortran packages to rather more humble attempts in C and C++, so perhaps the first question is “what is portability?”. There are several facets to this question:

- Portability of language with regard to unambiguous compiled behaviour in the compiler.
- Portability of environment. This covers operating system issues such as those addressed by Posix for example.
- Portability of external devices. Graphics and other devices each have hardware specific commands as a rule.

Each of these areas contains different types of portability problem.

I suspect different people may have different ideas. For example, I have witnessed a Fortune 500 company discuss the entire problem of portability as being that of the application environment only, (a planned corporate move from VMS to Unix), without even considering the other two, which in practice can present far worse portability problems. To me, in its purest form, portability represents the ability to recompile and relink a software package on different systems to the one on which it was designed in the full expectation that there will be neither compiler nor linker diagnostics and that the run-time behaviour of the application will be functionally identical. That’s good enough for me as a commercial systems engineer and I don’t need the kind of portability which is expected for Java for example, (although it would be nice). However, if an application has interfaces to external devices or to operating system facilities, this complicates the issue and maximal portability then dictates isolating the code for these interfaces from the rest of the (in theory fully portable) system by appropriate design.

The real question of course is, to what extent even these relatively limited goals are achievable with modern, (or even ancient) programming languages. My experience is that exceptional portability is far from easy and has to be continually in the minds of the developers from the design onwards. Back in the early 1980's, I designed and led the implementation team for a million line Fortran 77 seismic processing system, one of whose main goals was to exploit the increasingly rapidly changing face of numerical computational technology by being portable, [Hatton, 1988]. Being an engineer, I did not expect perfection, I was actually aiming for 99.5% portability as the systems we were designing then were right at the bleeding edge of graphical and vector processing technology and not only did we have to cater for some extraordinarily bizarre pre- Postscript graphics devices, but also floating point arithmetic accelerators of various persuasions. To quantify portability, I used the following functional definition of the portability p :

$$p = 100 \left(1 - \frac{t_{port}}{t_{dev}} \right)$$

where t_{dev} is the time taken to develop the system and t_{port} is the *porting time* or time taken to get it working on a different system. A million line commercial system of relatively modest application complexity represents around 50- 100 programmer years work, so I was expecting the system to be portable within a time period of 3- 6 programmer months. In fact, the longest port was the first as you might expect, and took about 2 months. Subsequently, this dropped to about two weeks in the 10 or so ports which took place later, representing a portability ratio of about 99.9%. The point I would like to make though is that this was achieved only by the continuously monitored and parsimonious use of a relatively simple and well understood language, Fortran 77 and a careful design which isolated the environmental and external device support issues.

Another question I would like to address before returning to the subject of the paper is, "How do you know when you have finished porting a system ?". The simple answer of course is that you never

do. Successful compilation and linking is certainly not a sufficient condition, since as I will discuss shortly, there is sufficient leeway of interpretation in any modern programming language that the compiler writer may make different decisions based on the same code fragment, leading to unexpected behaviour. In practice, any porting exercise must be accompanied by a regression suite, the sophistication of which is directly proportional to the degree of confidence which is being sought. This is a necessary overhead of the first attempt to port a system. Even systems with detailed regression suites have in my experience exhibited non-portable behaviour after relatively lengthy periods. One such example cost my company at the time a significant amount of money to resolve, when a single branch on a relational comparison of two floating point numbers in 70,000 lines of Fortran 77 source code, behaved differently only on the fifth or sixth port. The problem took several weeks to trace and is still a potential failure point in most modern programming languages. This occurred about a year after we thought the system was ‘fully’ portable.

Portability and language

It is not my intention to discuss environmental or external device portability here, although I will note in passing that support for such issues is exceptionally good in C. Rather I will confine myself to discussing the heart of Figure 1, the programming language itself. Elsewhere in this special edition you can read about portability experiences and claims for various other languages. So what features make a language portable or non-portable ?

Portability inducing properties

Portability is promoted by common practice and the long, stable and distinguished history of the dialect known as Kernighan & Ritchie C followed by a sympathetic and timely standardisation of the language in the form of ISO C 9899:1990, referred to here as Standard C. Even competitions such as the wonderfully anarchic Obfuscated C competition, which has as one of its goals, “to stress compilers by feeding them unusual code”, have contributed considerably to understanding the dark corners of the language.

Simplicity

The single most important portability inducing feature is however *simplicity*. C is arguably the last standardised language which can still be effectively understood by one person. As a direct result of these factors, the standard itself has attracted relatively few bug reports, (known as Defect Reports in the parlance of the ISO), compared to languages such as Ada for example, and it remains a relatively parsimonious language. It is fortunate in the sense that the burden of including the entirely unproven benefits of object-oriented technology have fallen on a derivative language, C++, rather than the language itself, as has happened with the transitions from Ada83 to Ada95 and Fortran 77 to Fortran 90 for example. Lest the reader think that lack of OO support is a disadvantage, there is now significant data questioning whether OO achieves *any* of its promised benefits, at least with regard to C++ implementations, [Hatton, 1998].

Validation

A crucial area which enhances portability is the existence of a validation system for compilers. Ada is particularly strong in this area, with a mandatory requirement. With C however, although a detailed validation system exists, (the NIST FIPS 160 suite), its use is not mandatory and in practice many compilers are not validated. The validation system is also somewhat lightweight with only the language syntax and a set of restrictions called *constraints*, defined in Standard C, forming part of the process. Be that as it may, the framework is there and even if their compiler is not validated, users can licence the validation suite to satisfy themselves of a certain minimum level of quality. In safety-related applications, such confidence building would be explicitly expected as part of the standard of care, [Hatton, 1995].

One perhaps surprising absentee from formal validation in C is floating point behaviour. Although in many cases, implementations are now IEEE 754 compliant, there is normally no indication. Users who rely on floating point arithmetic are strongly recommended to run Kahan's splendid *paranoia.c* obtainable from netlib@research.att.com for example. This C program puts the

floating point implementation through a wide range of exercises, and has been responsible for flushing out a large number of unpleasant bugs.

Portability reducing properties

In spite of the best efforts of a language standardising committee, there is always disagreement and compromise. The C committee however performed the important public service of carefully listing all the areas of the language upon which the committee were unable to reach consensus, resulting in 197 areas of imprecise definition delineated by the standard itself in its Appendix G, and *which do not form part of any validation process*. These are categorised as follows:

- a) Unspecified behaviour. The standard describes 22 such issues as *legal* behaviour for which the compiler behaviour is not specified. These include issues such as the evaluation order of operands as in the following example:

```
x[i] = i++; /* i incremented before or after use in x[] ? */
```

Regrettably, this subtle problem is common to many programming languages.

- b) Undefined behaviour. The standard describes 97 such issues as *illegal* behaviour for which the compiler behaviour is not specified. The standard therefore allows the compiler writer the option of whether or not to diagnose certain difficult faults. In practice, this category includes some of the worst problems encountered in practice, although many of them are statically diagnosable, one of the important indicators of a competent language, [Hatton, 1995].
- c) Implementation- defined behaviour. This is behaviour which the compiler writer must specify, but they are given leeway in how it might be implemented. In other words, these features tend to vary from implementation to implementation, but they can sustain a formal argument. There are 76 of them altogether in Standard C.
- d) Locale- specific behaviour. These are issues relating to internationalisation. I did not pay much attention to these until a

recent requirement to implement a major package to be Kanji-aware changed my views on this subject for ever. This too is an important portability issue which I strongly suggest people think about. Typical Japanese use involves handling strings of characters comprising perhaps 4 different character sets simultaneously: one of the several 16 bit pictorial Kanji sets, katakana, (a phonetic 51 character set the Japanese use to express foreign words), hiragana, (a phonetic 51 character alphabet the Japanese use for local words), and Roman, the alphabets common to the West. I can strongly recommend that companies try and free themselves from the somewhat insular ASCII.

- e) Finally, we can add the defect reports themselves of which there are around 60, resolved in Technical Corrigendum 1. These bring the standard up to date as of 1994 and the standardisation process for C9X has already started.

In discussions of reliability, it would be prudent to add the following:-

- f) Empirically determined misbehaviour. Over a period of years, responsible users report problems with use of the language, allowing other users to benefit. Very frequently, these refer to aspects which are entirely well-defined, but nevertheless are observed to cause problems in practice. As such, although they tend to be problematic, the problems are portable, and so can not be considered portability issues. In C, this has led to a few hundred issues which are worth avoiding. Many of these occur in other languages also, and it is a tribute to the C community that they are relatively well-documented, for example, [Koenig, 1989], [Spuler, 1994]. Some spectacular failures have been associated with these.

Conversions

Every language has its *bête noir* of portability. In practice, in common with many other languages, this is probably *conversions* for

C, in that it is very commonly implicated in failures in C systems, and certainly in portability failures. Type conversions in languages are the kind of things that everybody dislikes and nobody seems to be able to live without. Much intellectual discussion arises from the comparison of the benefits of strongly-typed languages, (type conversions forbidden) as Ada is purported to be, against the weakly-typed languages, (implicit type conversions performed) such as C. In practice, no widely used language is strongly-typed as such a language simply prevents type conversions taking place. As a result, ways of breaking the rules are provided to liberate programmers of the restrictions (i.e. safety) of strong typing, usually to extract a tiny piece of unnecessary performance or avoid the need for thought on how to do it safely. One such example in Ada was responsible for spreading the European rocket Ariane 5 all over the landscape in June 1996, to the tune of several hundred million dollars, when a 64 bit floating point number was jammed into a 16 bit integer container, which regrettably could not contain it. Almost the identical problem has been responsible for a number of other well-publicised disasters.

For our purposes, we are considering portability, so although the rigours of such things as precedence levels, (5 in Ada, 15 in C and a mind-numbing 22 in C++) provide many challenges to correct use, at least they are well-defined. In contrast, in C and C++ particularly, type conversions are rife, frequently implementation-defined and generally a minefield. For example, in C and C++, the special problems of handling signed and unsigned arithmetic in consort with occasionally differing semantics and conversion rules with can silently convert between the two, can make life very difficult for would-be portable code.

C portability experiences

Ultimately, of course, one can pontificate about portability endlessly. The proof must lie within actual practice and I would like to finish with some data in this area. Prior to using C, I thought Fortran 77 was an exceptionally portable language with portability ratios, (as defined above), of 99.5% and above. My experiences with C are however very positive, (in contrast with C++ for example).

Over a period of several years, it has proven possible to achieve and sustain compile-time portability with a 70,000 LOC system providing the parsimonious subset of the language which results from excluding the portability reducing features described above is enforced, (in my company's case automatically using a specific tool for the purpose, QAC). This includes environments as disparate as Unix and Windows NT/95. In the last three years, no port has required other than cosmetic changes, (for example setting differing flags for the compiler), and the portability ratio exceeds 99.9% for this product, (it takes less than a day to port including successfully completing the regression baselines for each platform). In stark contrast, C++ seems to be a source of endless surprise, with the most recent port of one of my company's C++ products failing (inter alia) with a compiler's semantic difficulties with the interpretation of const of all things, and the portability ratio is only around 95% in practice. This is probably symptomatic of the relative immaturity and inherent enormous complexity of the C++ language definition which will probably cast a dark shadow over portability prospects for that language for some years to come.

There are of course numerous examples of big systems written in C which have achieved very high levels of portability. X11, Unix and Motif all spring to mind as outstanding examples, particularly as the former two of these place particular demands on portability by virtue of their closeness to specific hardware issues in terms of OS or graphics primitives.

To summarise, I would like somewhat tongue in cheek, to propose a language-independent portability scale rather like the long-standing F0-F5 Fujita scale for tornadoes, (which I used to study many years ago). The comparison may seem facile, but in terms of cost, a really bad portability problem can cost just as much as a really bad tornado.

The P scale	Portability ratio	Comments
P0	> 99.9%	Porting carried out in atmosphere of extreme smugness. Slightly ruffled hair waiting for regression suites to complete successfully. You have to have done lots to get this smug.

P1	99.5%	Minor inconvenience. Occasional heart- stopping moment during compile, but eventually completes with a bit of tinkering. Regression suite runs successfully.
P2	99%	Modest inconvenience. A fair amount of huffing and puffing with calls back to base and frequent but generally short bursts of activity. Eventually successful compilation and successful regression result without too many changes.
P3	95%	Severe inconvenience. A lot of hot air experienced from managers told that the software was portable. Substantial code changes usually affecting all other platforms and many unkind words said.
P4	75%	Very severe inconvenience. Only a supremely masochistic company will pursue such non- portable code to completion. Large amounts of corporate huffing and puffing eventually achieves result but nobody is quite sure if the regression works or not.
P5	50%	Incredible inconvenience. Portability time comparable with original development time. This is not unknown and usually leads to major management changes. A disaster in every sense of the word.

In short, portability should be planned not assumed for any language.

Acknowledgements

Many people over the years have shared with me their generally hard- earned experiences of achieving practical portability. I would particularly like to thank the engineers at Programming Research and also those involved with SKS in the 1980's who contributed so much.

References

Hatton L. et. al. (1988) "SKS: a large- scale study in software portability", Software Practice and Experience

Hatton L. (1995) "Safer C", McGraw- Hill, ISBN 0- 07- 707640- 0

Hatton L. (1998) "Does OO sync with the way we think", IEEE Software.

Koenig A, (1989) "C traps and pitfalls", McGraw- Hill

Spuler D, (1994) "C and C++ debugging", Prentice- Hall