EC–
A measurement based safer subset of ISO C suitable for embedded system development

Les Hatton
Computing Laboratory, University of Kent*

November 5, 2003

Abstract

With the explosive growth of embedded systems, there is a major need for a standardised code of practice in the use of C. Although this area has been explored before, progress has been ad hoc and most importantly, carried out in the general absence of any measurement support. This paper attempts to define a relatively small number of rules which avoid known fault modes in the language which have published occurrence rates. It will studiously avoid any rules for which no measurement support is available. It is anticipated that the base subset may be extended as time goes by as further data becomes available. Much of the subset is equally relevant to ISO C++, although very little data exists to guide similar initiatives for the considerable part of C++ which lies outside C.

$Date: 2003/11/02 16:12:59 $

1 Overview

Common failure modes occur across both programming languages and time. For example, use of the value of an object which has not been explicitly initialised by the programmer is allowed in most if not all programming languages and of course is equally wrong in all of them. In languages like Fortran which evolved from a base with a static memory model only, this was often not too serious in that all non-explicitly initialised objects frequently received a default value of zero by the compiler. However in expression based languages like C and C++ where local variables are allocated on the stack, the content of non-explicitly initialised objects is essentially random. A simple example of a resulting failure mode in C and C++ follows.

```
...
{
     struct_t *ps;
     ...
```

*L.Hatton@kent.ac.uk, lesh@oakcomp.co.uk

```
if ( ps && (ps->val == 5) )
{
     ...
```

The built-in guard against dereferencing a NULL value of ps exploiting the left to right evaluation order of the && operator is defeated by the fact that the pointer ps can contain anything and is usually not zero. (For example, on the author's machine, it contained the 32 bit hexadecimal value 00000003.) The program immediately stops. This then is an example of an unequivocal failure mode. No engineer would disagree and its frequency of appearance can be measured.

There have been sporadic attempts to document these deficiencies over the years starting with the fine example of [6] and continuing with [12], [11] and [1]. Organisations individually have attempted at many sites to produce a code of standardised use for the language but it was only as recently as 1998 when an industrial body attempted to produce a pseudo-standard document with a publicly available set of rules, [7]. This is a laudable attempt but fails in part because of its mixture of rules.

As is discussed in detail by [3], there are three kinds of rule commonly found in programming language standards: category A (stylistic only), category B.1 (probably related to failure but no measurement data exists) and category B.2, (measurement data for failure exists). The article makes it clear why safer subsets should be constructed from category B.2 rules as mandatory and category B.1 rules as recommendations. Getting this mixture wrong or wording the rules inappropriately leads to a well-known signal-to-noise problem whereby there are far too many false positives to see the real fault modes easily, (i.e. those supported by measurement data). In the case of MISRA, this is shown to be about a factor of 50. The purpose of this present paper then is to present a set of rules which are rooted in measurement in the hope that these can be supplemented as more data becomes available and that these can therefore represent a nucleus of what is needed. Read in conjunction with MISRA, it will assist companies in defining a deviation policy (i.e. a subset of MISRA) actually based on failure if conformance with the full standard proves too difficult because of signal to noise problems as commonly happens with legacy code for example.

2 The measurement background

It is worthwhile saying a few things about the measurement approach. Software development is unlike other areas of engineering in that its basic measurement system is astonishingly crude. Classic reliability measures in engineering categorise reliability use concepts such as MTBF, (Mean time between failures), MTTF (Mean time to fail) and PFD (Probability of Failure on Demand). The former two are applicable to a continuously running system and the latter to a system which only runs under special circumstances and the probability of it failing to run when needed is therefore appropriate. Focusing on the first two, they are obviously temporal measures.

In software development however, the most commonly used measurement is fault density. The idea here is to accumulate the current total number of faults which have been found and to divide by the number of lines of code. In other words, it is a volumetric measure and not a temporal measure, (although it is time dependent). Worse, neither of the concepts of fault nor line of code are well-defined. Although there are definitions in standards such ANSI/IEEE 729-1983 and IEC 61508, the definitions are different. For this paper, the following definitions will be used.

2.1 Nomenclature

- *Fault* A fault is a static property of a system, either of its documentation or its code. It can be seen as a point of potential failure. For example fault density is the normal output of a code inspection. The code is not being executed. *Note that not all faults fail.* Even something as alarming as a divide by zero might never fail if the function which contains it is never called for example. This might sound somewhat contrived but in some complex systems, the author has found as many as 10% of all functions to be accidentally unreachable or inadvertently present but not used.

- *Failure* A failure is a dynamic property of a system and will be defined here as any difference between the actual and the expected behaviour of a system at run-time. *Every failure is caused by at least one fault.*

- *Line of code* A line of code is particularly susceptible to variations in definition. In C, there are three relatively widely used measures varying by about a factor of two.

  - Count of newlines. This is just a count of the newline characters, (the newline 'character' will differ on different architectures but the count will not.)
  - Pre-processed lines excluding blank lines. This is an attempt to remove the effects of comment by running the pre-processor before counting the non-blank lines. This is to a certain extent counterbalanced by the fact that all macros are expanded in place.
  - Executable lines. This is the most reliable measure in that at least to first order, it tends to be relatively independent of programming language. Where possible this will be used in what follows.

  In a sense it is not too important which of these are used as they tend to be highly correlated. It is however important to know which one has been used so they can be suitably normalised if necessary. The reader should note the common acronyms, KLOC (thousand lines of code, usually synonymous with KSLOC), KSLOC (thousand source lines of code, usually a count of newlines) and KXLOC (thousand executable lines of code). Of these the latter has the property of being somewhat insensitive to the programming language.

  There is one further factor which has to be taken into account in fault density and that is whether the fault has actually failed or not. The fault

density during code inspection is of course the total number of faults found divided by the total number of lines of code. At this stage it is not known in general whether they will fail or not. After testing starts, a separate density can be measured, the density of *faults that have failed.*

2.2 Relating fault to failure

The central idea of this paper is to relate faults or apparently failure-prone features of the language to the frequency with which they actually fail. As can be seen from the above discussion, it is of course relatively easy to measure their density in released systems either by manual counting or the use of parsing tools and corresponding occurrence rates are reported as of 1995 by [1] and the same population 4 years later by [4]. In this four year interval, some faults had decreased in frequency of occurrence but others had increased to give on average about the same overall occurrence rate. These two surveys will be referred to throughout the following.

To prove that this population of faults fail, this paper will simply quote a result reported in [8] which relates the two, albeit approximately. In a sense, it doesn't really matter how crude the relationship is provided the fault has actually failed. This is sufficient to warrant its inclusion in the safer subset presented here. If it has failed once, it can fail again.

3 The underlying language standard

Before describing a safer subset, it is worthwhile spending a little space discussing the underlying language standard. To define a safer subset based on measurement needs some kind of constant background. If the underlying language changes quickly, measurements may quickly age to the point of irrelevance. The MISRA standard [7] attempts to resolve this by using as its Rule 1 the requirement to adhere to ISO C 9899: 1990, also known as C90. This is a solid step but also problematic as even standardised languages move with time, indeed they are supposed to evolve. The C language is no exception with standardisation in 1990, defect corrections through a normative addendum and two technical corrigenda in the period 1993-1995 (which will be called C94 here) and re-standardisation in 1999, which will be referred to as C99. One other property of ISO standardisation is that a formal standard replaces the previous standard so in theory C90 no longer even exists as a standard.

The MISRA standard refers explicitly to C90 which of course is now officially defunct. The author's personal opinion is that in spite of a large number of able people spending a lot of time on it, C99 is a technical and commercial failure with no commercial compilers after 4 years and until recently, complete disregard for arguably its main customer, the embedded systems community. It will hopefully recover but there seems little point in using a base standard for which there are no implementations as yet. As a result, this safer subset will use C94 as its base standard and where there are semantics in common with the C99 standard, the C99 wording will be taken.

3.1 Poorly defined behaviour in the underlying standard

It should be noted that C is relatively unusual amongst programming languages in that the committee took unusual steps to document those areas where there was insufficient agreement to obtain an unambiguous definition. Appendix G (C90, 201 items) and J (C99, 366 items) simply list the items broken down into 4 categories:-

- Unspecified behaviour: Legal behaviour for which the committee does not provide a full specification
- Undefined behaviour: Illegal behaviour for which the committee does not provide a full specification
- Implementation-defined behaviour: Legal behaviour for which the committee does not provide a full specification but the implementor is supposed to provide one
- Locale-specific behaviour: Legal behaviour for which the committee provides a mechanism where it can be adjusted.

In practice, the undefined behaviour is by far the most prone to failure in real systems and dependence on this will be explicitly forbidden as the first rule. Dependence on implementation-defined behaviour is allowed provided this dependence is clearly stated, the implication being that the system behaviour would need to be re-verified if moved to another compiler implementation.

There seems no other natural choice at this stage. Note that this will not be implemented as a rule but is a separate requirement rather like a legal contract assumes statutory law as its background but does not state it explicitly. If there is any conflict, the statute applies. The safer subset plays the part of the contract and the underlying standard plays the part of the statute.

3.2 Permitted syntax extensions from C99

It is a little too early to judge what if any of the many new features in C99 might be useful to the embedded system developer but it seems clear that some at least will be useful. This section currently defines those which are permitted in this safer subset.

3.2.1 // comments

The // comment syntax of C99 and C++99 is permitted on the grounds that many compilers have implemented it for some time and it has been a standard part of C++ for a very long time. There is no measurement evidence for unpleasant side-effects.

4 A proposed safer subset based on measurement

The intention of this subset is to define a set of rules with the following properties:-

- *Every* rule is associated with faults which appeared in the quoted surveys and failed in the sense described above.
- Each rule covers as many of the fault modes as possible to reduce the total number of rules
- Each rule is easy to understand and as unambiguous as possible
- Each rule is as non-contentious as possible to ease acceptance
- Taken together, the rules cover the vast majority of the faults described in the earlier surveys and should therefore have a high signal to noise ratio with detection rates of around 8 per 1 KXLOC expected.

Note that the numbering of the rules is irrelevant as this may change in future releases. Each rule is identified however by a unique identifer, for example E_PREC, which is not expected to change. Note also that some of the faults associated with the referenced surveys are ignored here, for example, casting integers to pointers which is a common paradigm with embedded control system development. *The aim is to achieve a sensible balance between the risk of missing a fault mode and the benefit of presenting the programmer with high signal to noise ratio fault warning.*

Adherence to these rules does not preclude faults but it certainly removes a large percentage of the static faults *which are known to fail* and complying to it can contribute to impressive results such as quoted on [10] where after 3 years an asymptotic *fault that failed* rate of about 0.24 per KSLOC in a 130 KSLOC application has been achieved, a level comparable with the best asymptotic rates ever recorded. It should however be emphasised that safer subsets are necessary but not sufficient for low failure rates.

4.1 Underlying requirements

Where possible code features shall comply with the syntax of C94 with C99 as the over-riding semantics if the same syntactic feature is present in both standards. The object of this is to restrict programmers to those parts of ISO C which have been widely implemented (C94) whilst allowing for a number of clarifications in relevant wording which have taken place in C99. It is widely recognised that ISO C does not cater well currently for embedded control systems, so necessary and frequently numerous extensions such as the need to control interrupts are permitted where necessary.

It is strongly recommended that extensions generally are localised wherever possible such that a system contains as many conformant files as possible, maximising the ability to use tools, and also maximising the life of the system, (extensions being outside the standards community and tied to the success of the originating company tend to be much more volatile than standardised features). For example, assembler code is permitted but it is highly preferable if this is confined to an assembler function and called from a conformant C function rather than being specifically embedded in C using the local implementation facilities. Finally, where there is any conflict in the subset which follows and the underlying base standard, the underlying base standard must be honoured.

4.2 Assessment of quality of environment

4.2.1 Quality of compiler

The standard way of assessing the quality of a compiler is to use a validation suite such as FIPS160, the standard validation suite for ISO C. Regrettably, this service is no longer supplied by national standards bodies so the developer is responsible for assessing the quality of a compiler unless the supplier of the compiler can make suitable assurances.

It is also worthwhile noting that computer arithmetic, itself a combination of the chip functionality and the compiler run-time library may not be of the standard expected. A widely used method of testing this was introduced by [5] as a modification of the venerable but highly effective *paranoia*. No embedded system compiler the author has had access to has passed this without some problems as yet.

4.2.2 Quality of tools

Static analysers should be subject to the same tests as compilers. For example, a static analyser should parse FIPS160 correctly and if possible detect features which are poorly-defined in ISO C for which there are tests defined in FIPS160. Due to the significant costs of acquiring FIPS160 for inhouse testing, the author is aware of only one toolset which actually does this, [10].

4.3 Measurement based rule set for ISO C

The total number of rules necessary to achieve the stated goals is agreeably small although they vary in scope considerably.

4.3.1 S_GLOB: There shall be no dependence on any of the undefined features of ISO C

The word 'undefined' is used here in the sense described by ISO C and refers specifically to the 97 items in Appendix G of C94 (and the 191 items in Appendix J of C99 wherever there is overlap). This rule is very important and removes the need for many rules normally present in C and C++ standards such as 'all automatics should be initialised before use', 'do not divide by zero', 'expressions shall not depend on evaluation order' and so on. The ISO C committee has always been aware of these dangers and provides well for their documentation. It seems pointless to repeat subsets of them in a safer subset itself - they are all dangerous and should all be excluded.

Note that this rule is not statically enforceable in full. A number of the items detailed as undefined cannot be detected statically although a good static analyser should be able to make a reasonable attempt at the section as a whole.

4.3.2 F_PROT: All function calls and definitions shall be preceded by a new-style function prototype

4.3.3 F_COMP: All function call arguments and function definition parameters shall be compatible with the corresponding arguments in the corresponding function declaration

Taken together the previous three rules make the largest single contribution to the reduction of known fault in the C language. The first one accounts for 25.8% of all faults, ([1]), and the second guarantees that information cannot be lost across an interface. It should be noted that the second rule is actually stronger than the ISO standard which only requires that arguments be *assignment* compatible between declaration and use. All arithmetic types are assignment compatible so ISO C permits a *long* to be passed to a *short* or even to an *unsigned short*. Requiring full compatibility stops this dangerous practice. Some of the older fault modes such as 'return ' and 'return expr ' in the same function are rejected by C99 semantics.

The reader could quite reasonably ask why it is necessary to specify this 13 years after the facility appeared. Unfortunately, the facility is optional and in the surveys quoted earlier, approximately 1 interface in every 3 was not protected and only 12% of all systems checked had 100% protection.

Note that the first of these two rules is redundant in C++ as it is the only facility provided in that language. The second rule is however worthwhile in C++ also.

4.3.4 E_PREC: A boolean-valued sub-expression shall not appear next to a bit operator, an assignment operator or a relational operator in a if or while expression

This is a far more precise rule than the one normally used which is to prohibit operator precedence by forcing developers to use parentheses in all cases. This is an extraordinarily noisy rule, [3] as it would require code fragments like this:-

```
x = x + 1;      /* Wrong */
q = *p;         /* Wrong */
```

to be replaced by:-

```
x = (x + 1);    /* Right */
q = (*p);       /* Right */
```

It is hard to imagine that enforcing this has any effect other than to irritate programmers. In contrast the rule as given accurately picks up the examples of precedence failure reported in [6] and others with relatively few misfires, for example:-

```
if ( flags & FLAG != 0 )
/* Wrong - the programmer had meant if ( (flags & FLAG) != 0 ) */
```

4.3.5 E_SIDE: Every expression statement shall have at least one side-effect for any execution path

An expression statement is a defined concept in C. If it has no side-effects, (modifying a file or object or accessing a volatile), the statement doesn't do anything. Compilers will frequently remove it without warning. The failure is indirect in that the programmer clearly meant to do something but not this. The classic example is:-

```
i == j;          /* i = j; was meant. i == j; has no effect */
```

The reference to execution path is to prevent the following statement being used

```
(i > j) ? k++ : k ;
```

4.3.6 E_COMM: The expression on the left hand side of a comma operator shall have at least one side-effect

Many standards panic over the comma operator. It is perfectly well defined but programmers should perhaps take a little more trouble to understand its use. Consider for example this extra-ordinary sample of code from an air-traffic control system:-

```
z = i++ + (i,y++);
```

It is perfectly well-defined in terms of evaluation order but the left hand side of the comma operator does absolutely nothing and the code has exactly the same action as:-

```
z = i++ + y++;
```

4.3.7 E_UNEG: An unsigned expression shall not be compared for negativity

An unsigned expression can't be negative but this use appeared on average every 16 KXLOC in the surveys quoted earlier.

4.3.8 E_CSIGN: No implicit conversion shall changed the signed nature of an object or reduce the number of bits in that object

4.3.9 E_REACH: Every non-null statement shall be reachable

This rule is slightly noisy in that it detects the following benign case:-

```
case YES:
     ...
     return;
     break;          /* Unreachable but who cares ? */
```

More importantly, note that this rule automatically includes the much wider problem of detecting entire functions which are unused in a system. This is a surprisingly common problem as embedded systems become much more complex.

4.3.10 E_HIDE: Local objects shall not hide objects with internal or external linkage

The problem here is really confusion. It is common (1 occurrence around every 1.3 KXLOC) in the population of faults which cause failure. A simple example is:-

```
int i = 4;          /* i is 4 */
...
{
     int i = 5;     /* i is 5 */
     ...
}
                    /* i is 4 */
```

4.3.11 E_EXTR: The extern keyword shall not be used in a nested block

Some uses of this were undefined until C94 and programmers often find its definition surprising. It occurs rarely but can be extraordinarily difficult to diagnose so it is worthy of inclusion.

4.3.12 E_FEQL: Floating point objects shall not be compared for equality or inequality

4.3.13 E_FLOOP: Floating point objects shall not be used in the initialisation, controlling or re-initialisation expressions of a for statement or the controlling expression of a while statement

These two rules simply pass on the recommendations of handling floating point calculation in all programming languages which support it. There is nothing wrong at all with C's handling of floating point arithmetic to the point where the author sees no point in cluttering the language with fixed point arithmetic. All that is necessary is to heed the lessons of 30 years of floating point experience in other languages. It is disconcerting to note that ISO C added the ability to loop on floating point variables some 12 years after the Fortran committee consigned it quite rightly to the waste-can of experience.

The problem in both cases is that round-off errors in the permitted differences in the implementation of floating point arithmetic can cause these statements to behave differently on different machines. By avoiding these, the author has found very impressive agreement in large floating point algorithms implemented in very different compiler environments such Borland C++Builder and the GNU C Compiler.

4.3.14 E_BITF1: Every named bitfield of size 1 shall use the unsigned or signed keyword in its declaration

The author twice saw this in medical systems leading to dangerous failure. The problem is caused by the ISO C committee failing to agree whether a bitfield

declaration should be signed or unsigned. The corresponding implementation-defined behaviour means that the signed nature must be specified by the programmer to get portable behaviour.

4.3.15 E_NARRW: No pointer shall be cast to a narrower integral type

No comment necessary except perhaps to report its occurrence rate of every 9 KXLOC on average in the surveys quoted earlier.

4.3.16 E_PCAST: Pointers shall not be cast to different pointer types

Although this is permitted in ISO C and probably always will be, de-referencing them in their cast form is undefined and probably always will be. It is usually a symptom of a much deeper malaise and its most common manifestation in embedded control systems is in alignment problems. In the surveys quoted above, alignment was by far the most common pointer fault in embedded control systems.

4.3.17 E_CASE1: A switch statement shall have at least one 'case:' and shall have exactly one 'default:'

4.3.18 E_CFALL: No 'case:' or 'default:' shall be reachable from the previous 'case:' or 'default:' except by direct fall-through

Here direct fall-through means that no expressions appear between a 'case:' or 'default:' and the following one. An example follows.

```
switch( i )
{
case 1:
case 2:     /* Allowed */
...
case 6:
     ++k;
case 7:     /* Not allowed*/
...
}
```

4.3.19 P_ARGS2: No function-like macro shall use an argument more than once

Consider

```
#define    SQR(x)   ((x)*(x))
...
z = SQR(y++);
```

The programmer gets not one but two surprises. First of all y is incremented by 2 by this statement and second, it depends on evaluation order. The C library by design seeks to avoid this and so should every user.

4.3.20 P_PAREN: All macro arguments shall be parenthesized unless by doing so a syntax violation would be created

Consider

```
#define    BADABS(x)   ((x > 0)? x : -x)
#define    GOODABS(x)  ((x > 0)? (x) : -(x))
...
z = BADABS(a-b);       /* a-b < 0, z = -a-b */
z = GOODABS(a-b);      /* a-b < 0, z = -(a-b) */
```

This is perfectly straightforward however there are some reasonable uses which would lead to syntax failure if the arguments were parenthesised, for example,

```
#define DO_OP(x, y, op) ((x) op (y))                    /* ??? */
#define STR(a) # a                                      /* ??? */
```

4.4 Some concluding remarks

The observant reader will notice that no rules cover use of the C standard library. The simple reason is that faults based on the standard library had a very small impact in the measurement experiments which underpin this paper. For a similar reason, a number of commonly quoted rules based on good practice such as 'all functions should have a single point of exit' or 'the goto statement is forbidden' were also excluded from this subset. The temptation to add rules is almost overwhelming when defining programming standards but unless there is published measurement support suggesting that exclusion of a feature is very highly likely to improve reliability, the temptation should be ignored. Failing to do this means that it is always possible to force a programmer into changing code for no good reason, itself a known defect injector. By avoiding this temptation, the hard-pressed programmer is presented with a compact set of rules specifically associated with failure and which can be built on as measurement experience grows.

5 Conclusions

A compact safer subset for the ISO C language and to a smaller extent ISO C++ based on measured occurrence rates of fault and indirect though compelling relationships to failure has been presented. The subset is compact and covers most of the faults included in the cited surveys. The increasing use of a standard like MISRA is to be welcomed but with caution. When signal to noise ratios are high in the sense described by [4], it is all too easy for a standard not to achieve its primary goal and simply become a weapon used against system suppliers. In these circumstances, only the tool vendors flourish. The time an engineer has to spend in sieving out the wheat from the chaff may be far better spent on techniques which are known to be highly effective such as code inspections. At the end of the day, an engineer has only so much time to spend. Where he or she spends it to greatest effect is ultimately of course a matter for measurement and not speculation.

6 Download site

As measurement feedback grows, the latest version of the subset will continue to be available for free download at http://www.leshatton.org/.

References

[1] Hatton L. (1995) *Safer C: Developing high-integrity and safety-critical systems* McGraw-Hill, ISBN 0-07-707640-0

[2] Hatton L. (1997) *The T experiments: errors in scientific software* IEEE Computational Science and Engineering, April-June 1997

[3] Hatton L. (2003) *Safer Language Subsets: an overview and a case history, MISRA C* Accepted by Information and Software Technology, September 2003

[4] Hatton L. (2004) *Software Failure: avoiding the avoidable and living with the rest* Addison-Wesley, to appear in 2004.

[5] Hatton L. (2004b) *Embedded System Paranoia: a tool for testing the quality of embedded system arithmetic* Submitted to Information and Software Technology, September 2003

[6] Koenig A. (1988) *C traps and pitfalls* Addison-Wesley, ISBN 0-201-17928-8

[7] MISRA (1998) *MISRA C: guidelines for the use of C language in vehicle based software* MISRA, ISBN 0-9524156-9-0

[8] Pfleeger.L., Hatton L. (1997) *Investigating the influence of formal methods* IEEE Computer, (30)2, February

[9] Smith D.J. (1997) *Reliability, Maintainability and Risk* Butterworth-Heinemann, ISBN 0-7506-3752-8

[10] Safer C (2000) *The Safer C toolset* http://www.oakcomp.co.uk/

[11] Spuler D.A. (1994) *C++ and C debugging, testing and reliability* Prentice-Hall, ISBN 0-13-308172-9

[12] van der Linden P. (1994) *Expert C programming: deep C secrets* Prentice-Hall, ISBN 0-13-177429-8