PR:QA White paper Series: WP5.1

Why Coding Standards?

Abstract

Software engineering has a reputation for poor quality, late delivery and budget overrun. On past performance this reputation is well deserved. One of the most significant problems is that it errors tend not to be detected until it is too late to correct them within deadline and cost constraints.

The elimination of *all* errors is, at least in current industrial environments, well beyond the capability of available software technology. Nevertheless experience suggests that density of errors in released code can be halved by in-process checks supported by suitable analytical tools. This does not entail change of paradigms or languages. All it needs is the will to adopt proven techniques.

Of all forms of software quality control, code inspection is by far the most cost effective. When supported by good tools, the unit cost of static error detection is typically one or two orders of magnitude less than detection by dynamic methods. To make inspections work you need a coding standard setting out requirements that inspected code must satisfy. This paper explains the role of coding standards in code quality control.

PR:QA White papers provide short discussions of a range of topics in software engineering. For a current list contact

Programming Research Ltd.
Glenbrook House
1-11 Molesey Road
HERSHAM
Surrey KT12 4RH

Tel: 01932 888080
Fax 01932 888081
E-mail: support@prqa.co.uk
WWW: http://www.prqa.co.uk

Languages, compilers and errors

Of all forms of software quality control, code inspection is by far the most cost effective. When supported by good tools, the unit cost of static error detection is typically one or two orders of magnitude less than detection by dynamic methods. To make inspections work you need a coding standard stating what requirements code must satisfy.

In deciding what a coding standard must say many issues must be considered but most matters of substance can be traced ultimately to either the programming language or the compiler. To write a good coding standard you need a thorough understanding of how language and compiler affect the incidence of errors.

The language and its standard

Programming languages have been in existence for a little over fifty years. The first languages were no more than mnemomic representations of absolute machine code. Over the years these evolved through symbolic assembly codes to autocodes and the now familiar algorithmic languages. Recent developments, notably object-orientation, have produced languages that seem far removed from early developments.

Yet appearances are deceptive. Linking the earliest languages with the most recent is the underlying paradigm of the von Neumann machine architecture. This is characterised by a processor, a store (RAM) and between the two a pathway that can carry only a few bytes at a time. Backus has called this pathway *the von Neumann bottleneck*. It has dominated the design of programming languages for half a century.

A modern programming language provides data typing, data structuring, procedure structuring and means to associate particular procedures with particular data structures. You can accomplish much more in fifty lines of C than you can in assembler. Yet is not necessarily any easier to get those fifty lines correct for in essence the constructs of high-level languages are no more than convenient packaging of machine code.

Over thirty years ago computer scientists proved that the things you really need to check about machine code are either undecidable or not tractably verifiable. Reasonable general algorithms to check termination or correctness just do not exist. Since a high-level language is merely an ergonomically improved machine code, these limitations apply exactly as for low-level manipulation.

Neither termination nor correctness is a compile-time property.

If this is not bad enough, many language standards make the problem worse. In all but a few standards the quality of definition falls short of adequacy. Typical problems include:

- Lax compliance clauses,
- Wide latitude for implementation dependency,
- Lack of requirements for compiler diagnostics and run-time checks,
- Lack of stringency in the type system.
- Poor definition of integer and real arithmetic,
- Lack of precision in semantic definition.

C typifies all these weaknesses. The compliance clause permits a conforming program to contain extensions. The scope for implementation-dependency is so wide that portability is hard to define, let alone achieve. A compiler need produce only one diagnostic for a program that violates language constraints but beyond that there are no requirements to diagnose errors. Use of function prototypes supports strongish typing, but prototypes are not mandatory. Programmers often misunderstand arithmetic. The standard contains no formal semantics. The consequences are plain to see:

Most languages are not tightly defined.

Poor language design makes things worse.

Compiler writers have wide latitude.

Standards may confuse programmers.

The compiler

Where a standard exists, most compiler writers will claim to conform to it. Experience shows, however, that such claims must be examined carefully. In most cases the compiler is claimed to be conformant only under particular compile-time options. Change the options, especially those governing optimisation, and you change the program.

Even under appropriate compile-time options, the claim to conformance may need to be taken with caution. Where a compiler has been tested under a formal validation scheme, a certificate of validation offers some assurance of conformity. Nevertheless validation testing varies from language to language. Testing of error handling may be confined only to compile-time errors and even then may be quite limited. Testing of implementation-dependency may be non-existent and few validation suites provide a searching test of floating-point arithmetic. Finally no current validation scheme includes systematic stress testing in which robustness of implementation is tested by giving it deliberately complex or convoluted programs to compile.

Problems with compilers may include:

poor compile-time diagnostics,

limited run-time checking,

bugs in handling poorly define constructs,

differences in implementation characteristics under different compile-time options.

lack of robustness with complex code.

poor implementation of floating-point arithmetic.

Compilers for complex languages, notably C++, can be particularly problematic. Issues in the use of C++ include whether it can act as a conforming C compiler and how it interprets the more idiosyncratic parts of the C++ draft standard. Many developers of C++ tools regard C++ not as a single language but rather as a family of syntactically similar languages - hardly the most helpful approach to implementation.

The programmer

Most programmers struggle if not with languages, then with compilers. Few programmers have eve read the standards for the languages they use. Quite often they may not have ready access to the documentation for the compilers they use. Many are not specifically trained in computing and tend to reinvent wheels rather than use known algorithms. Most seriously of all a majority do not understand the principles of good design.

Programmers may:

not read language standards,

not read compiler documentation,

lack knowledge of algorithms,

lack understanding of good design.

Despite these weaknesses programmers may have a fancifully misplaced faith in tier own abilities, often to the extent of opposing basic quality control measures.

SHORT WONDER, SO MANY PROGRAMS HAVE SO MANY ERRORS.

Ten examples

Scaremongering is easy but the extent of the problem is well illustrated by examples:

Example 1: Poor language definition

The C standard [1] permits an implementation to omit evaluation of sub-expressions if "it can deduce that its value is not used and that no needed side effects are produced". Unfortunately the standard does not define what any particular side effect is needed. Now consider the fragment:

```
volatile int v;
(void)(&v);
```

In some environments even taking the address of a volatile may change its value. Does the last statement entail use of the value? If so does it produce a needed side effect? The standard does not tell us.

Example 2: Poor language design

The C switch statement allows structured selection. The case statement does the same in Pascal. In C, however, a switch statement need not be structured as a selection. Hence the following is possible:

```
switch (x)
{
case 0:     if (y > z)
            {
                        z++
            case 2:     y = z;
                        y--
            }
            else
            {
            case 4:     y++;
            }
}
```

How do *you* think it behaves?

Example 3; Implementor latitude

In its definition of constant expressions [1] (clause 6.4), the C standard defines certain forms of constant expression then goes on to say, " … an implementation may accept other forms of constant expressions." Consider therefore the declaration:

```
static double root_two = sqrt(2.0);
```

This fragment is conforming but not strictly conforming (yes, there is a difference). Some conforming compilers accept it, some do not.

As a further example consider:

```
char ch;
…
if (ch < 0) { ch++; }
…
```

If plain char is signed then the increment statement **ch++** is potentially reachable; if unsigned it is unreachable. Worse still, many compilers have a compile time option to determine the default type of plain char. Change the option and you can crash the program.

Example 4: Programmer misunderstanding

In C even simple arithmetic can confuse the programmer. Consider the following:

```
#include<limits.h>

long x = INT_MAX + INT_MAX;
```

In an implementation where long is twice the length of int, many programmers think that the assignment to **x** will give it the value twice that of **INT_MAX**. This is not true for the result type of the addition is int and there will be loss of precision. Such errors are responsible for many bugs in C and C++ code.

Example 5: Poor diagnostics

In the C declaration:

```
unsigned int i = 1234567;
```

the integral constant 1234567 exceeds 16 bits in width yet many C and C++ compilers will give no warning of this even if int is only 16 bits wide.

Example 6: Limited run-time checking

Few language standards require much run-time error checking. The following will cause floating-point overflow in C:

```
#include <float.h>
int main(void)
{
      double over = DBL_MAX / DBL_MIN;
      return 0;
}
```

Many implementations fail to trap the error at run-time even though it is detectable at compile-time!

Example 7: Compiler bugs

The **<ctype.h>** header is buggy in some well-known implementations in which it produces incorrect results for the **tolower()** and **toupper()** functions - *even though the implementation is by table lookup!*

Example 8: Implementation dependency

There are many implementation-dependent aspects of data representation. To take a simple example:

```
int x = 0;
```

Does not necessarily set all zeroes in x. In a one's complement system you might get all ones. If you write:

```
int x = ~(0^0);
```

you will get all zeroes, but it's not intuitively obvious that this may be necessary.

Example 9: Lack of robustness

Few language implementations are properly stress tested. It is possible for simple bugs to pass undetected for years. In one Ada compiler an array indexing error was detected by using a random stress test generator. The fix took a couple of hours but the bug had been there for nearly a decade!

Example 10: Floating-point arithmetic

Widespread use of floating-point co-processors has gone a long way to eliminating faulty implementations of floating-point arithmetic. Nevertheless software implementations, notably those provided for embedded processors, have been known to have astonishing bugs. Examples quoted by NAG include:

- Multiplication, division and negation may yield unnormalised results.
- In one FORTRAN double-precision system the numbers **1.0** and **1.0-2.0**(-47)** compared equal though they were clearly different even under the same system's single-precision representation.
- In another system **x*y** differed from **(-x)*(-y)** in as much as the last 3 bits.
- In another case negation and absolute value did not yield exact results even though exact results could be obtained simply by inverting a sign bit.
- One perverse implementation could take two numbers each greater than one, multiply them and give a result less than one.

Standards and compilers leave wide scope for confusion and error.

So what can we do?

Know the standard

Whatever languages we use, we are saddled with their standards, however good or bad they are. Unless you are prepared to work on a standards committee you cannot change a standard. Sometimes even influencing it for the better can seem more wearing than it is worth. The best course is to make sure you know what the standard says and, more importantly, what it does not say. This will not in itself reduce coding errors, but it gives a clearer understanding of the problems that may arise.

Read your language standard.

Keep reading it until you understand it.

Refer to it regularly

Know the compiler

You may have a choice of compiler. If there is a validation scheme, then wherever possible you should use a validated compiler under the options for which it has been tested for conformance. You should also find out, if necessary by doing your own tests, what the nature of implementation-defined characteristics is. If necessary, for critical applications consider stress testing. If necessary, get expert help with such testing for it is a very specialised field.

Use a validated compiler.

Use it under the options for conformance.

Test the implementation characteristics.

Stress test compilers used in critical applications.

Plug the gaps

If you know the language and your compiler well, then the need for a coding standard should be apparent but in any case the reasons for a coding standard can be stated succinctly.

A coding standard is a set of rules governing your use of a language. It supplements the language standard by defining acceptable and unacceptable usage. Unacceptable usage is usage that is conducive to error or misunderstanding. Acceptable usage is usage that avoids troublesome situations. So a coding standard helps you steer clear of all the problems we have discussed in this paper. Of course, it will not guarantee that your code is free from errors for you can always achieve an immaculate implementation of the wrong thing.

A coding standard helps you:

- avoid undefined usage,
- avoid unspecified usage,
- avoid implementation-defined usage,
- guard against compiler errors,
- guard against common programmer errors
- limit program complexity
- establish and objective basis for code review.

By avoiding undefined, unspecified and implementation-defined usage you effectively confine your use of language to the subset that is unambiguously defined. This gets round the weaknesses in the language standard. It is not foolproof but it will help prevent errors.

If you examine compiler bug reports, not just your own, but also others for the same language, you will get a reasonable idea of the kinds of things that may be badly implemented. You can then avoid those features of the language, or at least make the use subject to justification.

Also, using simple metrics, you can avoid the kind of complexity that leads to errors and makes code hard to maintain. A coding standard can define appropriate metric thresholds.

Finally, by getting your coding standard down in writing, you provide a set of rules that can be checked in code reviews. Code review is on of the most effective methods of quality control.

Quality system requirements

Software developers face increasing demands to demonstrate that development practices meet agreed standards. This is essential for companies developing safety-critical software as well as those seeking CMM, ISO 9001, UKAS or defence-related certification. Enforcement of coding standards is an essential element of properly controlled software development.

The following table shows code quality practices cited by various de facto and international standards:

Table 1: Code quality requirements

Code quality practice	Standard reference(s)
Reviews against coding standards	ISO 9000-3 (maximum) 4.1.1.2.1.5.6.4. poss. 6.4) TickIT Auditor's course (15.2) CMM (level 3) Def Stan 00.55 (31.1.1, 31.2.2) Draft IEC 1508: Part 3 (12.2.13)
Coding standards	ISO 9000-3 (5.4.2.3, 5.4.5c, 5.6.3) TickIT suppliers guide (4.5.7) CMM (level 2) Def Stand 00.55 (31.1.1) Draft IEC 1508: Part 3 (11.2.18, 19,20)
Safe subsets enforcement	Def Stan 00.55 (31.2.2) IEC 65A (11.2.15)

An appropriate coding standard meets the code quality requirements of all current industry and international standards for software development.

Developers with stringent quality requirements have been quick to adopt coding standards. The results have, in some cases, astonished them. PR:QA clients have reported the following experiences:

- A European telecommunications multinational has achieved a 30% reduction in the development life cycle for mission-critical systems using a coding standard supported by automated tools.

- By adopting and checking adherence to a coding standard, a large UK developer found and corrected a fault that would have cost in excess of £6 million to correct in the field.

Evidence suggests that a coding standard pays for itself in months if not weeks. For more details see the PR:QA White Paper WP-3.1 Configured Programming Standards.

A coding standard pays.

Automatic enforcement

A major concern of software development managers is usually to meet deadlines because of the protracted cycle of fixing and testing needed to remove errors. The earlier errors are caught, the shorter the lead-time to delivery. Sticking to a coding standard helps you avoid errors. Reviewing code against your coding standard helps you detect any errors you have made.

This may sound odd. If you have a coding standard, you need code reviews to make sure you are sticking it. Reviews take time and cost money. How can they help you meet deadlines?

It is beyond dispute that code review catches errors that would otherwise pass into production code. Nevertheless manual code review is time consuming. An effective manual review can get through, say 250 lines per hour - if it does more it is probably missing things.

But code review does not have to be manual.

A coding standard can be enforced automatically by static checking tools. PR:QA's world-leading static checkers ***QA Fortran***. ***QA C*** and ***QA C++*** provide effective automation of code inspection for Fortran, C and C++ respectively. Key benefits of automating checking are:

- Much faster code review - entire source files can be checked in minutes - several orders of magnitude faster than by manual means.
- Objective metric analysis - by taking appropriate measurements you get an objective quality profile and can apply statistical process control as occurs in manufacturing.

A good static analysis tool can be configured to suit **your** coding standard. Many PR:QA clients enforce their coding standards by configuring PR:QA tools to check to the particular rules that they want to enforce.

PR:QA can help you

PR:QA is a world leader in code quality. We offer clients the package of products and services they need to meet their delivery and quality targets.

Products

PR:QA's static checkers, QA C, QA C++ and QA Fortran are the most comprehensive available. All of them are configurable to enforce customised coding standards. PR:QA staff are active members of the BSI, ANSI and ISO standards committees for C and C++.

We help develop the standards and we keep our products up-to-date with them. PR:QA's own software quality assurance achieves very low defect levels comparable with CMM level 5 organisations.

Consultancy

PR:QA consultants are respected practitioners in their field. All of them are scientifically educated and have extensive experience in commercial, industrial and research fields. We serve clients world-wide and have an extensive track record in process, productivity and quality improvement in the most demanding applications and environments. Our consultants advise diverse clients in the UK and overseas. Typical assignments include:

- Development of coding standards for non safety-related software.
- Development of coding standards for safety-related software development to IEC 1508: Part 3.
- Development of language justifications for safety-related software development to IEC 1508: Part 3,
- Configuration of PR:QA tools to enforce coding standards.
- Managed programmes for process improvement to comply with ISO 9001, CMM, UKAS and defence-related quality standards.

PR:QA can develop a coding standard in two-to four weeks and supply tools for automatic enforcement.

If you want a coding standard, PR:QA has the expertise and the technology you need.

References

[1] ISO/IEC 9890:1990 - Programming languages -C

[2] Hatton, L., Safer C Developing Software for High-integrity and Safety-critical Systems

For more information contact:

Programming Research Ltd.
Glenbrook House
1-11 Molesey Road
HERSHAM
Surrey
KT12 4RH

Tel: **01932 888080**
Fax **01932 888081**
E-mail: **support@prqa.co.uk**
WWW: **http://www.prqa.co.uk**