

An Introduction to MISRA C++

Chris Tapp
LDRA Limited

Copyright © 2008 SAE International

ABSTRACT

The inexorable expansion of software in the automotive sector has led to a rapid expansion in the number of programming languages used in system development. C++ is one such language, and, given its wide user base, it is expected to be a major player in future automotive projects.

Given the above, MISRA has produced a set of guidelines (called MISRA C++) to assist in using C++ for critical systems. These guidelines are along the lines of those given within MISRA C.

This paper explains the background behind MISRA C++, the development process used in its production and shows examples of some of the guidelines.

INTRODUCTION

Few could have predicted the effect that MISRA C [1] would have within embedded systems engineering. Since its launch in 1998, it has become the dominant coding standard used for the development of critical systems with the C programming language.

Given the success of MISRA C, and the fact that C++ is now being used within critical systems, MISRA committed itself to the development of a similar set of guidelines for C++.

To that end, the MISRA C++ Working Group ("The Working Group") was established towards the end of September 2005. Two years later on, work is nearing completion.

It is hoped that MISRA C++ will become as successful and widely-adopted as MISRA C.

BACKGROUND

All programming languages contain aspects that are poorly defined, incompletely specified or defined in a way that permits different implementations to exhibit different behaviors. Any of these issues (called *insecurities*) may lead to unpredictable program behavior.

C++ is no exception, having a total of 198 insecurities, which can be broken down into various categories:

Unspecified Behavior – This occurs when code which passes successfully through a compiler and is then executed exhibits behavior that depends on the compiler implementation, but where the implementation is not required to document which behavior occurs. The C++ Standard [5] usually defines the range of possible behaviors that are available to the implementation.

Undefined Behavior – This occurs when a program contains an error, but there is no requirement for the implementation to detect it. Undefined behavior also arises due to omissions within the C++ Standard [5].

Implementation-Defined Behavior – This occurs when code which passes successfully through a compiler and is then executed exhibits behavior that depends on the compiler implementation, but where the implementation is required to document which behavior occurs.

Behavior That Requires No Diagnostics – These are issues that may be detected by a compiler, but for which it is explicitly stated there is no requirement to generate a diagnostic message.

Common programming errors are a second source of problems. For example, C++ programmers may inadvertently use the assignment operator (=) where the equality operator (==) was intended.

Insecurities and program errors are obviously unacceptable when developing critical systems. In order to help reduce the occurrences of these issues, the MISRA Guidelines [3] (Table 3) and IEC 61508 [4] require that "a restricted subset of a standardized structured language" be used. The rules contained within MISRA C++ constitute such a subset.

OBJECTIVES

The main objectives of MISRA C++ are:

- Produce a C++ subset suitable for use in critical systems.
- Produce a subset of C++ using techniques similar to those within MISRA C.

- Gather existing C++ guidelines from many diverse sources into a single repository.
- Add new guidance so as to significantly enhance the state-of-the-art.
- Establish a single, generic set of guidelines for the use of C++ in critical systems.
- Produce guidelines that are understandable to the majority of programmers.

DEVELOPMENT PROCESS

As the insecurities and common programming errors in C++ needed to be covered by the C++ subset, the first task was to identify them.

Identification of the language insecurities for C was easy, as the C Language Reference Manual [6] includes a definitive list in Appendix G. Unfortunately; this was not the case for C++, as the C++ Language Reference Manual (LRM) [5] does not contain a comparable list. QinetiQ (one of the members of the Working Group) had previously identified the insecurities within the LRM [5] (by searching for relevant phrases such as “it is undefined if”, “implementation-defined”, etc.) and enumerated them in a report entitled “An Investigation of the Unpredictable Features of the C++ Language” [7]. This document was used as a major source during the development of MISRA C++.

The identification of the common programming errors associated with C++ was relatively easy, as these had already been identified by a significant number of sources, including:

- Scott Meyers [8, 9]
- Stephen Dewhurst [10]
- Other coding standards [13, 14, 15]

C-related sources were also used, as many of the issues they identify are also relevant to C++:

- Andrew Koenig [11]
- Les Hatton [12]
- MISRA C [1, 2]

The issues identified above lead to rules relating to many features of C++. In addition, four features of the C++ language were given special attention:

1. Templates
2. Inheritance
3. Exceptions
4. Unnecessary Constructs

These were chosen because the Working Group felt that they were not adequately covered (if at all) by any identifiable source.

Each feature was “championed” by a member of the Working Group. This person was responsible for

producing a set of rules to control the behaviors associated with the feature. These rules were then reviewed by the Working Group and feedback provided to the champion to enable refinement of the rules. This process was repeated until the Working Group was happy with the results.

RULE STRUCTURE

Each rule consists of various items, as detailed below:

Rule Number – this uniquely identifies the rule. The rules are structured so that new rules may be added in the future without affecting the numbering of pre-existing rules.

Rule Category – this may be “Required” (a rule which requires a *formal deviation*ⁱ if it is not followed), “Advisory” (a rule for which a deviation is optional if it is not followed) or “Document” (a rule that requires supporting documentation to be produced and which may not be deviated from).

Headline text – the actual rule.

Issue References – an optional list of references to the C++ standard [5] that allow the reader to locate any insecurity(s) targeted by the rule.

Rationale – a justification for the rule and, when appropriate, an explanation of the issue(s) it targets.

Exception – an optional entry that lists any permitted exceptions to the rule. For example, the rule “*The value returned by a function having a non-void return type that is not an overloaded operator shall be used.*” has the exception “*The return value of a function may be discarded by use of a (void) cast.*” to allow a return value to be explicitly unused.

Example – a code fragment showing non-compliant code and, if appropriate, an example of how the same effect can be achieved in a compliant manner.

See Also – an optional entry containing cross-references to other (related) rules or other documents.

COLLATERAL DAMAGE

When framing the rules, it was sometimes difficult to precisely target an issue in a way that did not result in some desirable (i.e. safe) aspects of the language being excluded as a consequence. When such issues were identified, the C++ Working Group decided it was better to accept a rule whose scope was wider than desired, rather than to risk creating incomplete coverage for a particular issue.

ⁱ A process whereby a deviation is recorded in a log, given a written justification, subject to peer review and approved by a competent person.