MISRA C++

User Experience of Tools for Safety-Critical Systems
5th June 2008

Chris Tapp

Chairman, MISRA C++ Working Group

chairman@misra-cpp.org

Introduction

- The need for MISRA C++
- Development
- Sample Rules
- *goto*
- Conclusions
- Future Work
- Questions

The need for MISRA C++

- MoD use for Safety Related / Critical Systems
  - Used for ground-based safety related applications with very little control on use (no subset)
  - JSF use for safety related avionics (using JSF++)
  - The Avionics Systems Standardisation Committee (ASSC) was approached to provide the focus for an avionics industry led standard

The need for MISRA C++

- Existing use in other safety related systems
  - Jet engine controllers
  - Medical systems
  - Nuclear
- An automotive requirement meant MISRA became interested in C++
  - MISRA C++ Working Group formed
  - In order to avoid competing standards, the fledgling ASSC led team was absorbed into a MISRA C++ working group

Development – Objectives

- 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 – Language

- C++, like all other languages, has issues which may lead to insecurities
  - Unspecified behaviour
  - Undefined behaviour
  - Implementation-defined behaviour
  - Behaviour that requires no diagnostic
- C lists these issues in Annex G (or J for C99)
- This is not the case for C++, and they had to be teased-out of ISO/IEC 14882:2003
  - Luckily, QinetiQ (a member of the Working Group) had already enumerated these for a previous project

Development – Rule Formation

- Given the similarities with ‘C’, many issues were already covered by MISRA C rules (sometimes with changes)
- Existing sources used as the basis for many other rules
  - Scott Meyers
  - Stephen Dewhurst
  - Other coding standards, including HICPP, JSF++
- Several areas of the language were targeted for major work
  - Templates
  - Inheritance
  - Exceptions
  - Unnecessary constructs

Development – Rule Structure

- Rule Number (xx.yy.zz)
  - xx.yy gives the related section in the standard
- Rule Category
  - Required
  - Advisory
  - Document
- Headline text – the rule itself
- Issue Reference – location within the standard for any language issue(s) covered by the rule
- Rationale – justification and/or explanation of rule
- Exception – any exceptions to the rule?
- Examples

Sample Rule # 1

Rule 0-1-7	**(Required)**	**The value returned by a function having a non-*void* return type that is not an overloaded operator shall always be *used*.**

Rationale

In C++ it is possible to call a function without *using* the return value, which may be an error. The return value of a function shall always be *used*.

Overloaded operators are excluded, as they should behave in the same way as built-in operators.

Exception

The return value of a function may be discarded by use of a `(void)` cast.

Example

```
uint16_t func ( uint16_t para1 )
{
   return para1;
}

void discarded ( uint16_t para2 )
{
   func ( para2 );          // value discarded – Non-compliant
   (void)func ( para2 );    // Compliant
}
```

See also

Rule 5-2-4

Sample Rule # 2

Rule 15-4-1	**(Required)**	**If a function is declared with an *exception-specification*, then all declarations of the same function (in other translation units) shall be declared with the same set of *type-ids*.**

[NDR 15.4(2)]

Rationale

It is *undefined behaviour* if a function has different *exception-specifications* in different translation units.

Example

```
// Translation unit A
void foo( ) throw ( const char_t * )
{
   throw "Hello World!";
}

// Translation unit B
// foo declared in this translation unit with a different exception
// specification
extern void foo ( ) throw ( int32_t );   // Non-compliant
                                         //    – different specifier

void b ( ) throw ( int32_t )
{
   foo ( );   // The behaviour here is undefined.
}
```

goto

- Appropriate use can make code easier to understand and may improve safety
- Inappropriate use can lead to “spaghetti-code”
- MISRA C++ permits restricted use of “*goto*”
  - No jumps in to nested scopes
  - No “back” jumps
- Note – just because MISRA C++ permits this restricted used, it is perfectly acceptable for local policy to say otherwise!

Future Work

- Proposed future work items include
  - Exemplar suite – note not a compliance suite
  - Increase coverage
    - Identified issues
    - Library, possibly as a separate document
  - Update when the next version of ISO/IEC 14882 is released (and in use)
    - BOOST?

Conclusions

- The MISRA C++ subset is now available
  - Existing sources have been pulled into a single document
  - New guidance has added significantly to enhance the state-of-the-art
- Wide adoption and establishment as best-practice will mean
  - Skills will be readily available
  - General C++ programming competence will be improved

MISRA C++

Any questions?