appliedinformatics

C++ for Safety-Critical Systems

Günter Obiltschnig
Applied Informatics Software Engineering GmbH
St. Peter 33
9184 St. Jakob im Rosental
Austria
guenter.obiltschnig@appinf.com

Abstract. C++ is now widely used in the development of software for embedded systems, even safety-critical and hard-real-time systems. Even if, due to their design, other programming languages may be better suited for the development of safety-critical systems, there are other relevant factors in favor of C++. Examples are availability of skilled developers and tool support. The use of C++ in the development of air vehicle software for the Joint Strike Fighter (JSF), and the public release of the C++ coding standard used in that project (JSF C++), has certainly increased the interest in using C++ for safety-critical systems. In June 2008 the MISRA C++ standard "Guidelines for the use of the C++ language in critical systems" has been released by the Motor Industry Software Reliability Association. Similar to the JSF C++ standard, the MISRA C++ standard defines rules, as well as a "safe" subset of the C++ language for the development of safety-critical systems. This paper gives an overview of both the JSF C++ and MISRA C++ standards and also looks in detail at some of their rules and the rationale behind them. An interesting aspect that is covered is also where both standard differ. For example, JSF C++ does not allow the use of C++ exceptions at all, whereas MISRA C++ specifies detailed rules for their use.

1 Introduction

C++ [1] is now widely used in the development of software for embedded systems, even safety-critical and hard-real-time systems. It must be stated, though, that C++ (as well as C) is, by its design, not really a suitable language for writing high integrity or safety-critical software. C++ gives a programmer a great deal of freedom. With freedom comes responsibility, though, and, in the case of C++, a whole lot of responsibility. Nevertheless, there are good reasons for using C++ in a safety-critical system:

- Higher abstraction level. C++ allows for a higher level of abstraction than C, making it more appropriate for increasingly complex applications. At the same time, C++ allows low-level programming required for hardware access or high-performance code.
- Support for object-oriented programming. C++ directly supports object-oriented programming, which is the paradigm of choice in today's software development projects.
- Portability. Wide availability of C++ compilers for almost all hardware platforms enables easier porting of applications to new or lower-cost processors at any stage in a project.
- Efficiency. C++ compilers can generate code that performs as efficient, or even more efficient, than C code.

- Availability of skilled developers. C++ is one of the most widely used programming languages, and a lot of skilled C++ developers are available.
- Tool support. Many tools that support model-driven development can automatically generate C++ code.
- Maturity. C++ is mature and consequently well-analyzed and tried in practice.

On the other hand, C++ has a lot of issues that limit its suitability for safety-critical systems:

- C++ is a highly complex language. Learning and fully understanding C++ requires a huge learning effort. C++ code does not always do what one would 'intuitively' expect from looking at the source code. At the same time, the high complexity of C++ increases the probability of compiler errors. C++ compiler writers are only humans, after all.
- The semantics of some C++ constructs are not fully specified, leaving room for portability issues – the behavior of a program may vary depending on what compiler was used (undefined, unspecified and implementation-defined behavior).
- C++ makes it very easy for a programmer to make mistakes that cannot be diagnosed by a compiler (e.g., the use of the assignment operator '=' instead of the comparison operator '==').
- C++ does not provide any built-in run-time checking, e.g. for arithmetic overflows or array bound errors.
- While being a strongly-typed language, C++ leaves too many holes to circumvent the type system, deliberately or unintentionally.

When using C++ for safety-critical systems, great care must be taken to avoid any language constructs and code that can potentially lead to unintended program behavior. C has most of these issues as well, though, and this hasn't stopped C becoming one of the most widely used languages in safety-critical systems. This has only been possible through the introduction of coding standards that limit language features to a safe subset that can be used without giving rise to concerns.

2 Coding Standards

The most popular coding standard for using C in safety-critical systems is MISRA-C: Guidelines for the use of the C language in critical systems [2]. First published in 1998 and revised in 2004, MISRA-C specifies a "safe" subset of the C language in the form of 121 required and 20 advisory rules. MISRA-C has enjoyed great adoption among developers of safety-critical applications, not just in the automotive industry, at which it was initially targeted. MISRA-C has also had a great influence on another coding standard for using C++ in safety-critical systems.

The "Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program" [3], or "JSF C++" in short, was kind of revolutionary, as it signaled a move away from Ada as the mandated programming language for avionics software by the US Department of Defense. JSF C++, while taking many rules from MISRA-C, is a bit different in concept from MISRA-C (and the new MISRA-C++) as it also defines coding style and metric guidelines, which the MISRA standards don't have. For example, AV Rule 1 requires that *"[a]ny one function (or method) will contain no more than 200 logical source lines of code*", and AV Rule 50 requires that *"[t]he first word of the name of a class, structure, namespace, enumeration, or type created with typedef will begin with an uppercase letter. All others letters will be lowercase.*" In total, JSF C++ defines 221 rules.

The current trend to move from C to C++ in the development of critical systems has lead to the publication of MISRA-C++ [4] in the summer of 2008. MISRA-C++ takes many rules from MISRA-C: 2004 and adds many more C++ specific rules, bringing it to a total of 228 rules.

3 Issues with C++ in Safety-Critical Systems

An analysis of MISRA-C++ and JSF C++ reveals many issues that must be taken great care of when using C++ in a safety-critical system. Some of these issues are briefly discussed in the following.

Preprocessor

Use of the preprocessor is strongly restricted by coding standards. Basically the only allowed use of the preprocessor is to implement include guards, preventing multiple inclusion of the same header file.[1] Other uses of macros, e.g. to define constants or to define inline macros are forbidden. Generally, C++ provides better alternatives for most uses of preprocessor macros. For defining constants, C++ supports the `const` keyword, as well as enumerations. Inline functions are a much better alternative than inline macros, avoiding all of the problems associated with functional macros and macro arguments.

Implementation-defined, unspecified and undefined behavior

The C++ standard specifies the exact (required) behavior for most, but not for all language elements. Certain language constructs are described as implementation-defined, unspecified or undefined.

Implementation-defined means that the compiler writer is free to implement a certain language construct in any way he sees appropriate, as long as the exact behavior is consistent, documented, and the compilation succeeds. The standard may specify a number of allowable behaviors from which to choose one, or it may leave it entirely up to the compiler writer.

Unspecified is similar, except that the behavior of the implementation need not be documented and need not even be consistent.

Undefined behavior means that the standards does not place any requirements whatsoever on the implementation. The compiler may even fail when compiling such a language construct, or the program may silently produce incorrect results. MISRA-C++ also defines the notion *indeterminate*, which specifies undefined behavior arising due to an omission in the C++ standard.

Code that relies on implementation-defined, unspecified or undefined behavior is not portable, and thus forbidden by coding standards. The same applied to code using proprietary compiler extensions.

Error-prone language constructs

C++ allows many constructs that can easily lead to erroneous code. Examples are `switch` statements, where coding standards restrict the placement of `case` labels and require a `break` (or `throw`) to terminate every non-empty switch-clause, as well as a `default` case.

For `if`, `else`, `switch`, `which`, `for` and `do ... while` statements, the body must be a compound statement, with the exception of `else` being followed immediately by another `if`. All `if ... else` constructs shall be terminated with an `else` clause. Restrictions are placed on the use of null statements (single semicolons). The use of `for` loops is also guarded by rules covering the use of loop counters, control variables, etc. Use of the `goto` statement is strongly restricted; `setjmp`/`longjmp` is forbidden at all.

[1] Some compilers also provide proprietary mechanisms to prevent multiple inclusion of a header file (in the form of #pragma directives), but these are not allowed by MISRA-C and JSF C++.

Type System

The type system in C++ has many loopholes that make it easy to circumvent it. Coding standards severely restrict the use of implicit or explicit type conversions (casts), as these can easily lead to a loss of information. The C++ standard does not define fixed sizes for built-in types, making it harder to write portable code requiring fixed-size integers, for example. In addition, C++ (as well as C) does all arithmetic operations in either `int` or `long`, depending on the original operand types. Prior to an arithmetic operation, integral types (`signed/unsigned short` and `signed/unsigned char`) are promoted to int. This can easily lead to non-portable code, as the following example shows:

```
short i = 10000;
short j = 8;
int32 result = i * j;
```

On a system where `int` is 32 bits (and `short` is 16 bits), the result will be as 80000, as one would expect considering that the multiplication is carried out with both operands promoted to `int`. However, on a system where `int` is just 16 bits, the result will be undefined, as the multiplication will be carried out in 16-bit, will thus overflow, and only the (wrong) result will then be converted to `int32`.

The plain `char` type is problematic, as it can be either signed or unsigned. In C++, `char`, `unsigned char` and `signed char` are all distinctive types. Coding standards only allow the use of the plain `char` type for holding character values and forbid the use of the plain `char` type for arithmetic operations, as this leads to non-portable code.

Classes

Coding standard put a strong focus on encapsulation, meaning that member variables of a class must generally be private, and any undesired access to member variables (e.g., though a member function returning a non-const pointer a reference to it) must be prevented. Multiple inheritance should be avoided, with the exception of using multiple interface classes (classes only having pure virtual member functions as members) as base classes. Special care is required when dealing with compiler-generated default constructors and assignment operators.

Dynamic Memory

Use of dynamic or heap memory (`new`, `delete`) is generally strongly restricted in safety-critical systems.

4 A Comparison of MISRA-C++ and JSF C++

From a C++ developer's perspective, a comparison of MISRA-C++ and JSF C++ is a rewarding exercise, as it shows some interesting similarities, differences and even conflicts between the two standards and sharpens the eye for any potential issues arising out of the use of C++ in safety-critical systems.

MISRA-C++ and JSF C++ have common roots in MISRA-C. MISRA-C++ takes many rules from MISRA-C, as does JSF C++. Another thing that both standards have in common is their dislike for macros, basically only allowing the use of macros for include guards, while at the same time making the use of include guards in header files mandatory. Both standards require the use of defensive programming techniques (run-time checks) and recommend the use of static analysis tools (MISRA-C++ Rule 0-3-1, AV Rule 15). There are a lot of "common sense" rules (e.g., avoiding "stupid" names, always initialize variables, etc.), where both standards agree as well.

MISRA-C++ does not have any rules or guidelines regarding coding style or metrics. In the introductory text however, the authors of MISRA-C++ recommend the selection of an

appropriate coding styleguide. In contrast, JSF C++ has specific rules for coding style and software metrics (e.g., AV Rules 1, 3, 41 – 45, etc.).

MISRA-C++ and JSF C++ disagree in a number of issues. For example, JSF C++ does not allow the use of C++ exceptions[2], while MISRA-C++ allows it. When dealing with null pointers, MISRA-C++ enforces the use of a `NULL` macro, whereas JSF C++ discourages use of the `NULL` macro and suggests the use of literal zero (0) instead.

5 Other Recommendations

C++ provides many facilities that make writing correct and robust code easier. Although coding standards do not explicitly enforce or recommend the use of these facilities, their use can generally be recommended.

For example, the RAII (Resource Acquisition Is Initialization)[3] idiom helps in implementing correct management of resources. RAII enables the implementation of smart pointers, which help to avoid many pointer-related errors[4]. RAII also enables the implementation of scoped locks that greatly simplify the correct implementation of critical sections.

Another important facility for writing robust C++ code are templates, as they enable the implementation of bounds-checked arrays. An interesting use for templates in safety-critical systems is the implementation of fixed-point arithmetic.

When using exceptions, a C++ developer should be familiar with the concept of exception safety [5]. Exceptions greatly simplify error handling in programs. However, an exception in the wrong place can easily lead to objects or data structures left in an invalid state, causing trouble at a later time. Again, RAII is the most important tool for writing exception safe code.

6 Conclusion

The use of C++ for safety-critical systems is possible, provided that an appropriate coding standard such as MISRA-C++ or JSF C++ is followed, and appropriate tools that automatically check the conformance of all source code are used. A comparison of two C++ coding standards for safety-critical systems, MISRA-C++ and JSF C++, shows some interesting similarities, differences and also conflicts. Any programmer intending to write safety-critical code in C++ should familiarize himself with both coding standards, as they give a great insight into the potential hazards one has to deal with when using C++ in critical code.

References

[1] ISO/IEC 14882 International Standard: Programming Languages – C++, Second Edition, ISO/IEC, 2003

[2] MISRA-C: 2004 – Guidelines for the use of the C language in critical systems, MIRA Limited, 2004.

[3] Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, Lockheed Martin Corporation, 2005

[4] MISRA-C: 2008 – Guidelines for the use of the C++ language in critical systems, MIRA Limited, 2008.

[5] David Abrahams, Exception-Safety in Generic Components, http://www.boost.org/more/generic_exception_safety.html

[2] This has mostly to do with missing support for exceptions in the compiler used in the JSF project.
[3] http://en.wikipedia.org/wiki/Resource_acquisition_is_initialization
[4] Smart pointers have limited use in safety-critical code, as the use of dynamic memory is severely restricted.