C++ in Safety-Critical Applications: The JSF++ Coding Standard

Bjarne Stroustrup, Texas A&M University

Kevin Carroll, Lockheed-Martin Aeronautics Co.

Overview

- **Why C++?**
- **Design philosophy of JSF++**
- **Examples of rules**
- **Summary**

Language Selection on JSF

- **Primary language selection criteria:**
  - ***Object-oriented design methodology employed.***
  - ***Did not want to translate OO design into language that does not support OO capabilities.***
  - ***Tool availability and support on latest hardware platforms.***
  - ***Attract bright, young, ambitious engineers.***

- **Constraint**
  - ***Usable for avionics software***
    - Safety
    - Hard real time
    - Performance (time and space)

Language Selection: C++ or Ada95?

- **Historical perspective:**
  - ***Language choice made during the late 1990's in the midst of the "dot com" boom.***
  - ***Prospective engineers expressed very little interest in Ada.***
    - Ada tool chains were in decline
  - ***C++ was attractive to prospective engineers.***
    - C++ tools were improving

- **C++ satisfied language selection criteria as well as staffing concerns**

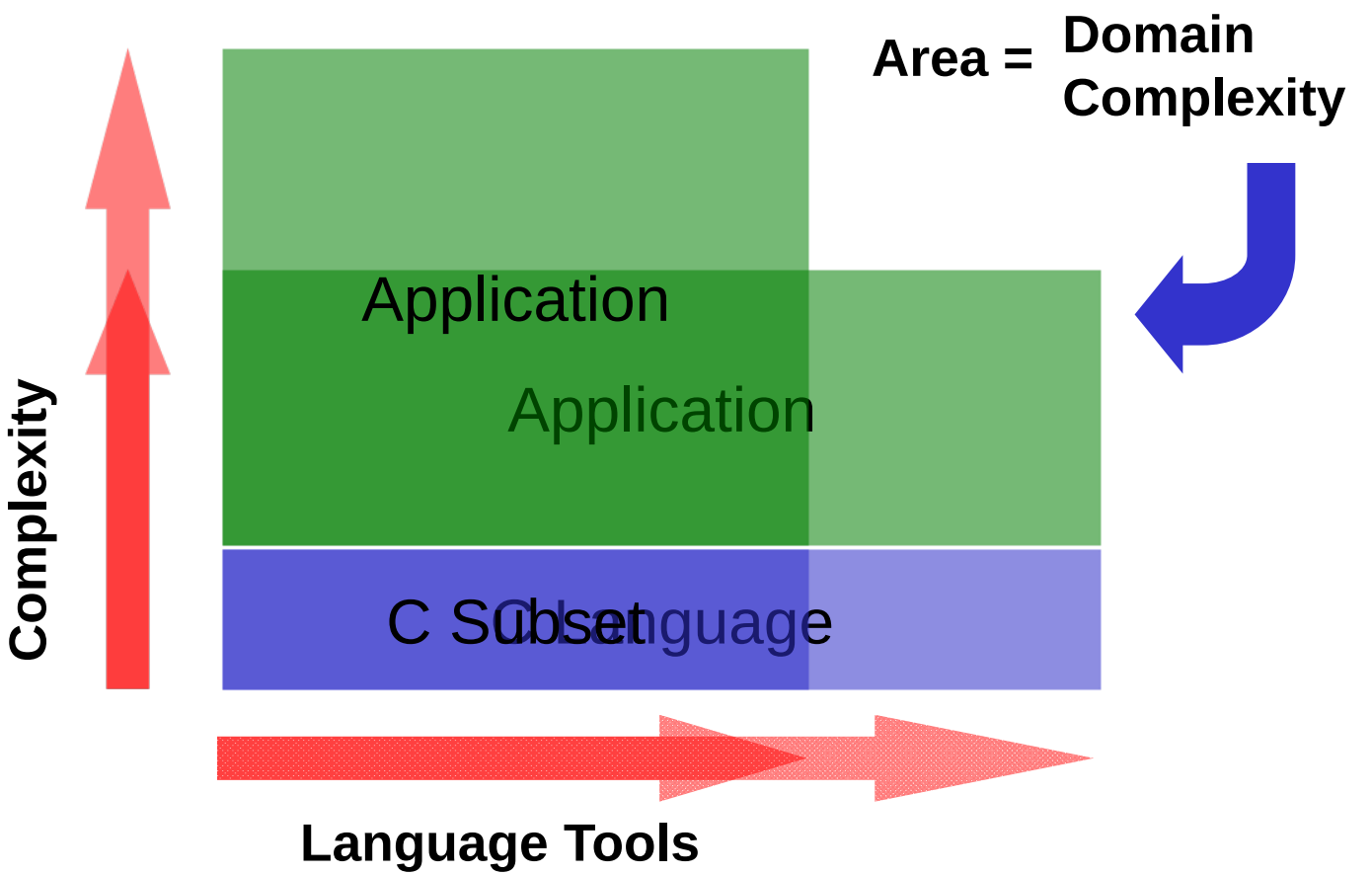
Use of C++ on JSF Program

- **LM decided to use C++**
- **UK skeptical of LM's choice to use C++**
  - ***Commissioned QinetiQ to perform a comprehensive safety review of C++.***
    - 325 "issues" raised
    - QinetiQ unaware of the JSF C++ Philosophy addressing "C++ issues"
  - ***Formed technical committee (LM, JPO, UK)***
    - 325 issues => 8 new rules and 9 rule modifications
- **Achieved full US JPO and UK JCA agreement that C++ can be used in safety-critical software.**

Coding Standard Philosophy

- **Problem**
  - ***General purpose languages are "General Purpose"***
- **Conventional solution**
  - ***Subset language***
    - Eliminate "unnecessary" features
    - Eliminate "dangerous" features
- **Example**
  - ***MISRA C***
  - ***Well-known subset of C developed for safety-related software in the motor industry***

Problems with Language Subsets



Problems with Language Subsets

- **Subsetting alone fails to resolve some classes of problems**
  - ***Complexity is pushed out of the language and into the application code***
    - The semantics of language features are far better specified than the typical application code
  - ***Errors are simply disguised***
    - (i.e. the problem is moved, not solved)
  - ***Productivity is lowered***
    - Programmers must write significantly more lines of code
  - ***Maintenance is made more difficult***
    - More code
    - Worse localization of design decisions
    - Intent of code less obvious

JSF Philosophy (JSF++)

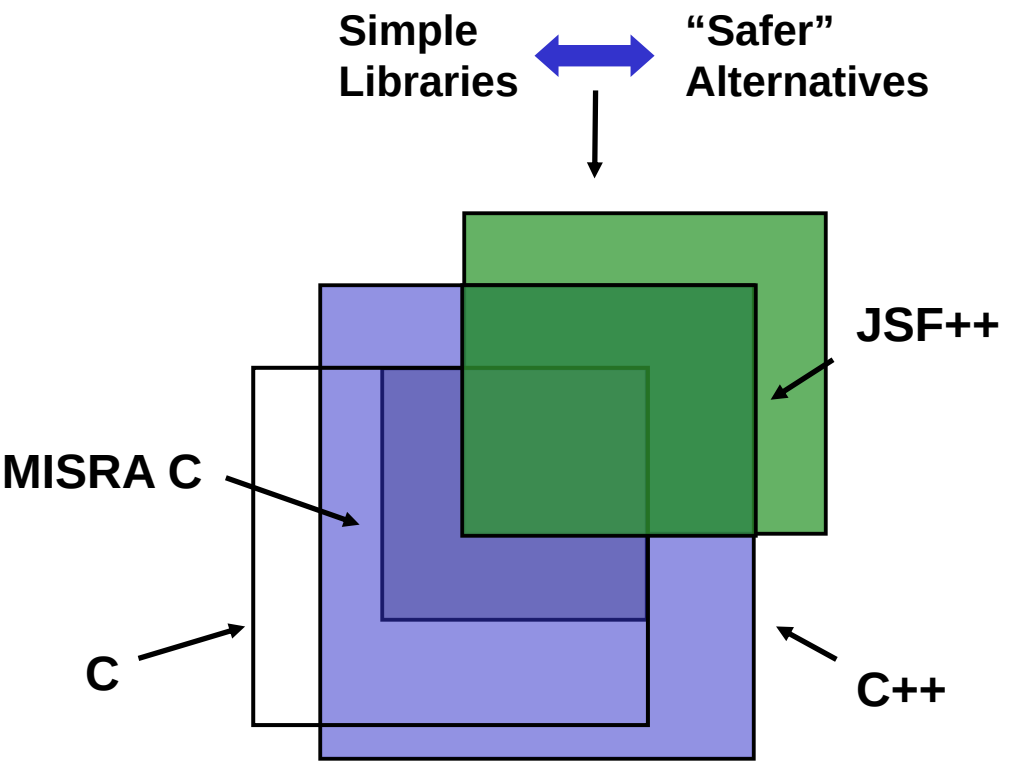
- Provide “safer” alternatives to known “unsafe” facilities
  - *Cannot be accomplished in C*
  - *Cannot be accomplished via subsetting alone*
- Craft rule-set to specifically address undefined behavior
  - *Restrict programmers to a better specified, more analyzable, and easier to read (and write) subset of C++*
  - *Eliminates large groups of problems by attacking their root causes (e.g. passing arrays between functions as pointers)*
- Ban features with behaviors that are not 100% predictable (from a performance perspective)
  - *Free store allocation (operators new and delete)*
  - *Exception handling (operator throw)*

“Safer” Alternatives to “Unsafe” Facilities

- **Extension of C++’s philosophy with respect to C**
- **Examples**
  - ***passing arrays as pointers***
  - ***use of macros***
  - ***Use of (C-style) casts***
- **Many well-known dangerous aspects of C were simply “designed out” of C++.**
  - ***Many MISRA rules are simply unneeded in C++:***
    - 20, 25, 26, 71, 72, 75, 77, 78, 80, 84, 105, and 108
- **C++ provides safer alternatives to many dangerous C constructs**
  - ***E.g. polymorphism rather than switch statements***
- **Conclusion: C++ can provide a “safer” subset of a superset.**

JSF++

- MISRA is a subset of C
- C Allows “unsafe” code that C++ rejects
- JSF++ is a subset of MISRA (with respect to C)
- JSF++ is a subset of ISO C++
- C++ provides facilities that allow the banning of, or isolation of, dangerous C/C++ features
  - *Libraries, primarily relying on simple templates, are used to provide cleaner, “safer” alternatives to known problem areas of C and C++*



Examples:
"Safer" Subset of a Superset

• Variable macros	→	• Constants
• Function macros	→	• Inline functions
• C-Style casts	→	• C++-style casts
• Arrays	→	• Array class
• Dynamic memory	→	• Allocators (static)

- Note: C++ facilities such as templates and virtual functions can be used to eliminate most casts (explicit type conversions)
  - *JSF++ strongly encourages elimination of casts*

Feedback

- No standard is perfect
- We expect to refine JSF++ based on feedback
  - *Lockheed_martin developers*
  - *The embedded systems community*
  - *The C++ community*
  - *Tool builders*
- Please comment!

Coding Standard Enforcement

- Automated
  - *Where possible tools will be used to automate coding standard enforcement.*
  - *Quick, objective, and accurate.*
  - *Presently working with tool vendors to automate enforcement of additional rules.*
- Manual
  - *For those rules that cannot be automated, checklists are provided for code inspections.*

Enforcement and understanding

- Developers don't like to follow rules they don't understand
- Developers find it hard to follow rules the don't understand
  - developers should obey the spirit of the rules, not just the words
  - They can do that only if they understand the general philosophy and the rationale for individual rules
- Where possible rules are prescriptive ("do this") rather than prohibitive ("don't do that")
- Every rule has a rationale
  - Some rationales are extensive
- Many rules have examples

JSF++ overview

- 231 rules
- 11 pages of “front matter”
  - *table of contents, terminology, references, etc.*
- 58 pages of rules
- 76 page “Appendix A”
  - *with more extensive rationale and examples*

Rules

- Each rule contains either a “should”, “will” or a “shall” in bold letters indicating its type.
  - ***Should*** *rules are advisory rules. They strongly suggest the recommended way of doing things.*
  - ***Will*** *rules are intended to be mandatory requirements. It is expected that they will be followed, but they do not require verification. They are limited to non-safety-critical requirements that cannot be easily verified (e.g., naming conventions).*
  - ***Shall*** *rules are mandatory requirements. They must be followed and they require verification (either automatic or manual).*

- Breaking a **Should** rule requires one level of management approval
- Breaking a **Will** or **Shall** rule requires two levels of management approval (and documentation in code for **Shall** rules)

Example (“no macros”)

AV Rule 29

The #define pre-processor directive **shall not** be used to create inline macros. Inline functions shall be used instead.

Rationale: Inline functions do not require text substitutions and behave well when called with arguments (e.g. type checking is performed). See AV Rule 29 in Appendix A for an example.

See section 4.13.6 for rules pertaining to inline functions.

Further rationale for AV 29

Inline functions do not require text substitutions and are well-behaved when called with arguments (e.g. type-checking is performed).

Example: Compute the maximum of two integers.

```
#define max (a,b) ((a > b) ? a : b)        // Wrong: macro

inline int32   maxf (int32 a, int32 b)     // Correct: inline function
{
   return (a > b) ? a : b;
}

y = max (++p,q);                           // Wrong: ++p evaluated twice
y=maxf (++p,q)                             // Correct: ++p evaluated once and type
                                           // checking performed. (q is const)
```

Example (“avoid stupid names”)

AV Rule 48

Identifiers **will** not differ by:

- Only a mixture of case
- The presence/absence of the underscore character
- The interchange of the letter ‘O’, with the number ‘0’ or the letter ‘D’
- The interchange of the letter ‘I’, with the number ‘1’ or the letter ‘l’
- The interchange of the letter ‘S’ with the number ‘5’
- The interchange of the letter ‘Z’ with the number 2
- The interchange of the letter ‘n’ with the letter ‘h’.
- Rationale: Readability.

Example (“use classes well”)

AV Rule 65

A structure **should** be used to model an entity that does not require an invariant.

AV Rule 66

A class **should** be used to model an entity that maintains an invariant.

AV Rule 67

Public and protected data should only be used in **struct**s—not classes.

Rationale: A class is able to maintain its invariant by controlling access to its data. However, a class cannot control access to its members if those members are non-private. Hence all data in a class should be private.

Exception: Protected members may be used in a class as long as that class does not participate in a client interface. See AV Rule 88.

Example
("use operators conventionally")

AV Rule 84

Operator overloading **will** be used sparingly and in a conventional manner.

Rationale: Since unconventional or inconsistent uses of operator overloading can easily lead to confusion, operator overloads should only be used to enhance clarity and should follow the natural meanings and conventions of the language. For instance, a C++ operator "+=" shall have the same meaning as "+" and "=".

Example
(“use operators conventionally”)

```
Array<int,4> a(0);      // array of 4 ints initializer to 0
a[2] = 7;               // conventional use of subscripting: [ ]

Array<int,4> b(0);
b = a;                  // conventional use of assignment: =

*b = 1;                 // unconventional use of *: banned
```

Example (“avoid arrays”)

AV Rule 97

Arrays **shall not** be used in interfaces. Instead, the **Array** class should be used.

Rationale: Arrays degenerate to pointers when passed as parameters. This “array decay” problem has long been known to be a source of errors.

Note: See Array.doc for guidance concerning the proper use of the Array class, including its interaction with memory management and error handling facilities.

Example (“avoid arrays”)

- **Array code vs array code**

Correct Size?

```
void f(Point_3d* p, uint32 n)
{
    for (uint32 i=0 ; i<n ; ++i)
    {
        // process elements
    }
}
```

```
void f(Array<Point_3d>& a)
{
    const uint32 n = a.size();
    for (uint32 i=0 ; i<n ; ++i)
    {
        // process elements
    }
}
```

Size encapsulated

- **Declaration and Invocation**

```
Point_3d a1[size];
…
f(a1,size);
```

```
Fixed_array<Point_3d,size> a1(Point_3d());
…
f(a1);
```

Example (“avoid arrays”)

- **Declaration and Invocation**
  - ***(size unknown until run-time)***

```
Point_3d* a2 = new Point_3d[size];
…
f(a2,size);
```

```
Dynamic_array<Point_3d> a2(alloc,size);
…
f(a2);
```

Uses same interface

Example (“templates should be simple”)

AV Rule 101

Templates shall be reviewed as follows:

1. *with respect to the template in isolation considering assumptions or requirements placed on its arguments.*
2. *with respect to all functions instantiated by actual arguments.*

Note: The compiler should be configured to generate the list of actual template instantiations. See AV Rule 101 in Appendix A for an example.

Rationale: Since many instantiations of a template can be generated, any review should consider all actual instantiations as well as any assumptions or requirements placed on arguments of instantiations.

Example (“templates should be simple”)

```
//definition:
Template<typename T, int dims> class Matrix { /* … */ };
    // dims must be a positive integer < 7
    // T must have ordinary copy semantics
    // T must provide the usual arithmetic operations (+ - * %)
    // T most provide the usual comparisons (< <= > >=)

//uses:
    Matrix<int,2> a(100,200);
    Matrix<complex,3> b(100,200,300);     // error: complex has no <
    Matrix<double,-2> b(100);             // error: negative #dimensions
```

- **C++98 catches most violations at compile time**
- **C++0x can express and enforce such requirements (“concepts”)**

Example (“always initialize”)

AV Rule 142 (MISRA Rule 30, Revised)

All variables shall be initialized before use. (See also AV Rule 136, AV Rule 71, and AV Rule 73, and AV Rule 143 concerning declaration scope, object construction, default constructors, and the point of variable introduction respectively.)

Rationale: Prevent the use of variables before they have been properly initialized. See AV Rule 142 in Appendix A for additional information.

Exception: Exceptions are allowed where a name must be introduced before it can be initialized (e.g. value received via an input stream).

Example (“always initialize”)

```
int a;              // uninitialized: banned
// …
int b = a;          // this is why: likely use before set bug
// …
a = 7;              // “initialize” a

Const int max_buf = 256;
// …
Buffer<char,max_buf> buf;     // uninitialized
Buf.fill(in2);                // fill buf from input source in2
```

Summary

- Provide “safer” alternatives to known “unsafe” facilities
  - *Note: cannot be accomplished via subsetting alone*
  - *Simple template-based libraries created to provide cleaner, “safer” alternatives to known problem areas of C and C++*
- Rule-set crafted to specifically address undefined behavior
  - *Restricts programmers to a better specified, more analyzable, and easier to read (and write) subset of C++*
  - *Eliminates large groups of problems by attacking their root causes*

Summary

- Banned features with behaviors that are not 100% predictable
  - *Free store allocation*
  - *Exception handling*
- Automated enforcement mechanisms used whenever possible
- Achieved full US JPO and UK JCA agreement

Best Practices

- Coding Standard includes guidance on topics including
  - *Arrays and pointers*
  - *Constructors/destructors*
  - *Object initialization*
  - *Inheritance hierarchies*
  - *Templates*
  - *C++-style casts*
  - *Namespaces*
  - *Statement complexity*

Implementation-Defined Aspects of C++

- Strategic approach for managing implementation-defined, undefined, and unspecified aspects of C++. (attack the root causes, not the symptoms)
- Rules that
  - *prohibit dependence on evaluation order and side-effects.*
  - *manage memory layout issues (unions, bit-fields, casts, etc.)*
  - *address overflow issues*
  - *minimize the use of casts*
  - *limit the use of pointers and arrays*
  - *prohibit mixed-mode arithmetic and comparisons*

Contributors

- The design of JSF++ involved many people
  - *Internal and external reviews*
- Key contributors
  - *Bjarne Stroustrup*
  - *Kevin Carroll*
  - *Mike Bossert*
  - *Randy Ethridge*
  - *Greg Hickman*
  - *Michael Gibbs*
  - *Mike Cottrill*
  - *Tommy Gitchell*
  - *John Colotta*
  - *John Robb*
  - *Ian Hennell*

Additional Information

1. *JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM*. Document Number 2RDU00001 Rev C. December 2005. “JSF++” http://www.research.att.com/~bs/JSF-AV-rules.pdf
2. ISO/IEC 14882:2003(E), *Programming Languages – C++.* American National Standards Institute, New York, New York 10036, 2003.
3. Bjarne Stroustrup: *Abstraction and the C++ machine model*. Proc. ICESS'04. December 2004. Also in Springer LNCS 3605. Enbedded software and systems. 2005. Bjarne Stroustrup. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 2000.
4. Lois Goldthwaite (editor): *Technical Report on C++ Performance*. WG21 N1487=03-0070. 2003-08-11.
5. Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, April 1998. MISRA C (“old think”) .
6. Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Design, 2nd Edition.* Addison-Wesley, 1998.

- More references in [1]