# C++ for Safety-Critical Systems

**DI Günter Obiltschnig**
Applied Informatics Software Engineering GmbH
guenter.obiltschnig@appinf.com

applied informatics

A life-critical system or safety-critical system is a system whose failure or malfunction may result in:

- death or serious injury to people, or
- loss or severe damage to equipment or
- environmental harm.

Wikipedia

# Is **C++** the "right" programming language for safety-critical software?

Before answering this, we may first want to ask:

# Is **C** the "right" programming language for safety-critical software?

# NO!

- C puts all responsibility on the programmer.
- C makes it too easy to make mistakes.
- C has a weak type system.
- C has no automatic runtime checks.
- C is insufficiently specified.

# So why do we keep using C for safety-critical software?

- C is available everywhere.
- C programmers are easy to find.
- C is efficient (execution speed, code size).
- C gives direct access to the bare metal.
- C is open and portable (assembler).
- *The many issues of C with regards to safety-critical systems are well understood.*

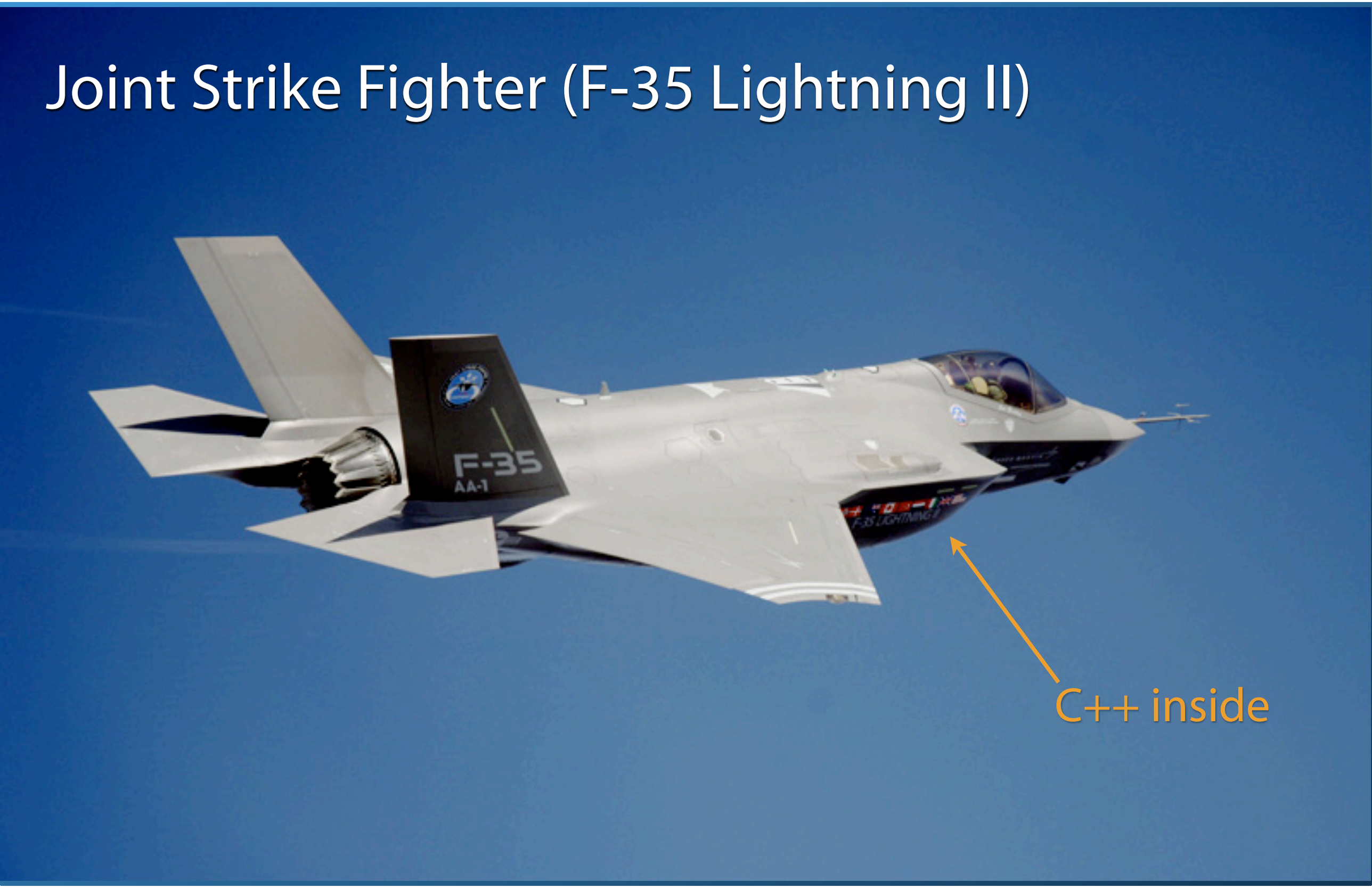
# So what about C++?

- C++ puts all responsibility on the programmer.
- C++ makes it easy to make mistakes.
- C++ has an easily circumventable type system.
- C++ has no automatic runtime checks.
- C++ is insufficiently specified.
- *C++ is highly complex.*

*"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."*

Bjarne Stroustrup (circa 1986)

# Joint Strike Fighter (F-35 Lightning II)



# Why C++?

- Higher abstraction level.
- Support for object-oriented programming and other modern paradigms.
- Portability and efficiency.
- C++ makes it far easier than C to write safe software (*if we let it to*).
- C++ *is mature and consequently well-analyzed and tried in practice.*

# Also...

- There are factors that have a much more significant influence on the quality of the resulting product than the choice of programming language:
  - the **software engineering process**,
  - the choice of **tools**, and
  - the **skills and education** of the software engineers.

# Case in Point

- The Ariane 5 Flight 501 disaster (1996) could not have been avoided by the use of a different programming language.

# Coding Standards

- General Coding Standard Idea:
  *Limit use of language features to a "safe" subset.*
- Eliminate all that's "unnecessary" and/or "dangerous".
- Example: MISRA-C/C++

- Bjarne Stroustrup on Coding Standard Philosophy:*
  - *Just sub-setting simply moves complexity from language rules to application code.*
  - *Provide a superset (safer alternatives to unsafe features) with close-to-ideal properties, then subset that superset.*
- Example: JSF C++

* Talk "C++ in Safety-Critical Systems"

- Examples for subset of superset:
  - Use an Array class template instead of plain C++ arrays (bounds checking, avoid array decay problem).
  - Use inline functions instead of function macros.
  - Use consts and/or enums instead of constant macros.

# Excurse: The Array Decay Problem

- An array passed as a function argument (or used in an expression) degenerates to a pointer.
- Array size information is lost, or has to be passed in a separate argument.
- Just another opportunity for bugs to sneak in.
- Buffer overruns are the biggest source of security vulnerabilities in C/C++ applications.

```cpp
#include <iostream>


void f(int arg[])
{
    std::cout << sizeof(arg) << std::endl;
        // arg is basically int*, so output will be sizeof(int*)

    std::cout << arg[10] << std::endl;
}


int main(int argc, char** argv)
{
    int arr[20];

    f(arr);

    return 0;
}
```

```cpp
#include <iostream>


void f(int arg[20])
{
    std::cout << sizeof(arg) << std::endl;
        // arg is basically int*, so output will be sizeof(int*)

    std::cout << arg[10] << std::endl;
}


int main(int argc, char** argv)
{
    int arr[20];

    f(arr);

    return 0;
}
```

```cpp
#include <iostream>


void f(int arg[20])
{
    std::cout << sizeof(arg) << std::endl;
        // arg is basically int*, so output will be sizeof(int*)

    std::cout << arg[10] << std::endl;
}


int main(int argc, char** argv)
{
    int arr[10]; // this won't even produce a compiler warning

    f(arr);

    return 0;
}
```

```cpp
#include <iostream>


void f(int (&arg)[20]) // ugly - passing array by reference
{
    std::cout << sizeof(arg) << std::endl;
        // this will actually give the expected result.

    std::cout << arg[10] << std::endl;
}


int main(int argc, char** argv)
{
    int arr[20];

    f(arr);

    return 0;
}
```

```cpp
#include <iostream>


void f(int (&arg)[20]) // ugly - passing array by reference
{
    std::cout << sizeof(arg) << std::endl;
        // this will actually give the expected result.

    std::cout << arg[10] << std::endl;
}


int main(int argc, char** argv)
{
    int arr[10]; // this won't compile

    f(arr);

    return 0;
}
```

# Safer Alternative to arrays: Array Template

- Avoids array decay problem.
- Enforces type safety.
- Enables automatic range checking.
- Impact on performance is neglectable if implemented wisely.

```
template <typename T, std::size_t N>
class Array
{
public:
    // ...

    std::size_t size() const
    {
        return N;
    }

    T& operator [] (std::size_t i)
    {
        if (i < N)
            return values[i];
        else
            throw RangeException();
    }

    const T& operator [] (std::size_t i) const
    {
        if (i < N)
            return values[i];
        else
            throw RangeException();
    }

private:
    T values[N];
};
```

```cpp
#include <iostream>
#include <Array.h>


void f(const Array<int, 20>& arg)
{
    std::cout << sizeof(arg) << std::endl;
        // we will get the correct size (20*sizeof(int))

    std::cout << arg[10] << std::endl;
}


int main(int argc, char** argv)
{
    Array<int, 20> arr;

    f(arr);

    return 0;
}
```

```
#include <iostream>
#include <Array.h>


void f(const Array<int, 20>& arg)
{
    std::cout << sizeof(arg) << std::endl;
        // we will get the correct size (20*sizeof(int))

    std::cout << arg[10] << std::endl;
}


int main(int argc, char** argv)
{
    Array<int, 10> arr;

    f(arr); // this won't compile

    return 0;
}
```

# JSF C++

- Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program (Rev. C)
- Published in December 2005 by Lockheed Martin Corporation.
- Available for free as PDF from http://www.research.att.com/~bs/JSF-AV-rules.pdf

# MISRA-C++

- MISRA-C++: 2008 – Guidelines for the use of the C++ language in critical systems.
- Published in June 2008 by MIRA Ltd (UK).
- Available as PDF for EUR 17 (UKP 15) from http://www.misra-cpp.org/

# C++ Coding Standards History

MISRA-C: 1998

MISRA-C: 2004

**JSF C++ (2005)**

**MISRA-C++: 2008**

# Coding Standard Size

| | JSF C++ | MISRA-C++ | MISRA C |
|---|---|---|---|
| **Rules** | 221 | 228 | 141 |
| **Pages** | 141 | 220 | 119 |

# Coding Standard Rule Classification

| JSF C++ | MISRA-C++ | Note |
|---|---|---|
| *Should* | *Advisory* | recommended way of doing things |
| *Will* | *Required* | mandatory, no verification required |
| *Shall* | | mandatory, verification required |
| | *Document* | |

# JSF C++ on General Design Criteria

- **Reliability**: consistently and predictably fulfill all requirements
- **Portability**: source code should be portable
- **Maintainability**: source code should be consistent, readable, simple in design and easy to debug
- **Testability**: minimize code size, complexity and static path count
- **Reusability**: encourage component reuse
- **Extensibility**: make the product extensible (requirements evolve)
- **Readability**: source code should be easy to read, understand and comprehend

# JSF C++ on Code Size and Complexity

- **AV Rule 1**: Any one function (or method) will contain no more than 200 logical source lines of code.

  *Rationale*: Long functions tend to be complex and therefore difficult to comprehend and test.

- **AV Rule 2**: There shall not be any self-modifying code.

  *Rationale*: Self-modifying code is error-prone as well as difficult to read, test, and maintain.

- **AV Rule 3**: All functions shall have a cyclomatic complexity number of 20 or less.

  *Rationale*: Limit function complexity.

# JSF C++ and MISRA-C++ on 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’.
- Compare **MISRA-C++ Rule 2-10-1**: Different identifiers shall be typographically unambiguous.

# JSF C++ and MISRA-C++ on Names

- Readability.
- Avoid stupid names.
- Programmers make mistakes when they are tired or have poor paper or screens.
- Example:

```
int I0, I1, l0, l1;
I0 = 10;
l0 = 11;
I1 = l0;
l1 = 11 + 10*I0 + l0;
```

# The Preprocessor

- Use of the preprocessor in both JSF C++ (AV Rules 26 – 32) and MISRA-C++ (Rules 16-0-1 – 16-3-2) is basically restricted to header file inclusion.
- **AV Rule 29**: The #define pre-processor directive **shall not** be used to create inline macros. Inline functions shall be used instead. (Compare MISRA-C++ Rule 16-0-4)
- **AV Rule 30**: The #define pre-processor directive **shall not** be used to define constant values. Instead, the const qualifier shall be applied to variable declarations to specify constant values. (Compare MISRA-C++ Rule 16-2-1)
- **AV Rule 31**: The #define pre-processor directive **will** only be used as part of the technique to prevent multiple inclusions of the same header file. (Compare MISRA-C++ Rule 16-2-2)

```
#define max(a, b) ((a > b) ? a : b)          // very bad

#define max(a, b) (((a) > (b)) ? (a) : (b)) // better, but still forbidden

template <typename T>                        // correct
T max(T a, T b)
{
    return (a > b) ? a : b;
}
```

```
#define SAMPLE_RATE 44100                    // forbidden

const int SAMPLE_RATE = 44100;               // correct
```

# Implementation-defined, Unspecified and Undefined Behavior

# Implementation-Defined Behavior

- Freedom to implement a certain language construct in any way seen as appropriate by the compiler writer, as long as
  - the exact behavior is consistent,
  - the exact behavior is documented, and
  - compilation succeeds.

# Implementation-Defined Behavior Examples

- The number of bits in a byte.
- The size of int or bool.
- Whether char is signed or unsigned.
- The sign of the result of an integer division or modulo operation when the operands have different signs.
- The representation of floating-point values.
- The linkage of main().

# Unspecified Behavior

- Freedom to implement a certain language construct in any way seen as appropriate by the compiler writer.
- Consistency and documentation are not required.

# Unspecified Behavior Example

```
// Q: What's the output of this program?
// A: It depends...

void f(int a, int b, int c)
{
    std::cout
        << “a = “ << a
        << “, b = “ << b
        << “, c = “ << c
        << std::endl;
}

int main(int argc, char** argv)
{
    int i = 0;
    f(i++, i++, i++);
        // Function arguments may be evaluated in any order.
        // The increment operation may be delayed until all
        // arguments have been evaluated.

    return 0;
}
```

# Undefined Behavior

- No requirements whatsoever.
- The compiler may fail.
- The resulting program may silently produce incorrect results.
- The program may even make your computer explode.
- Exactly what you want in a safety-critical system.

# Undefined Behavior Examples

- The effect of an attempt to modify a string literal.
- The effect of using an invalid pointer value.
- The effect of dereferencing a pointer returned by a new for a zero-size object.
- The result of a divide by zero.

> Various rules in JSF C++ and MISRA-C++ forbid code constructs that lead to unspecified or undefined behavior, or that rely on implementation-defined or otherwise poorly-defined behavior.

# Language Constructs

- **MISRA C++ Rule 6-3-1:** The statement forming the body of a *switch*, *while*, *do ... while* or *for* statement shall be a compound statement.
- **MISRA C++ Rule 6-4-1:** An *if (condition)* construct shall be followed by a compound statement. The *else* keyword shall be followed by either a compound statement, or another *if* statement.
- **MISRA C++ Rule 6-4-3:** A *switch* statement shall be a *well-formed switch* statement.
- **MISRA C++ Rule 6-4-5:** An unconditional *throw* or *break* statement shall terminate every non-empty *switch-clause*.

```
// This is valid C++!!!

void copy(char* to, const char* from, int count)
{
    int n = (count + 7)/8;
    switch (count % 8)
    {
    case 0: do { *to++ = *from++;
    case 7:      *to++ = *from++;
    case 6:      *to++ = *from++;
    case 5:      *to++ = *from++;
    case 4:      *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
            } while (--n > 0);
    }
}
```

"Many people have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against."

Tom Duff (Inventor of "Duff's Device")

# The Type System

- The C++ type system has many loopholes that make it easy to circumvent it.
- Implicit Conversions
  - Integral Promotion Conversions
  - Assigning Conversions
  - Balancing Conversions
- Explicit Conversions (casts)

# Integral Promotion

- Arithmetic operations are always conducted on integer operands of type *int* or *long* (*signed* or *unsigned*).
- Operands of type *char*, *short* and *bool* are always converted to type *int* or *unsigned int* whilst those of type *wchar_t* and *enum* ("small integer types") may be converted to *int*, *unsigned int*, *long* or *unsigned long*.
- The result of adding (or multiplying, etc.) two objects of type *unsigned short* is always a value of type *signed int* or *unsigned int*.

# Integral Promotion

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

- If int is 32 bits (and short is 16 bits), the result will be 80000.
- If int and short are both 16 bits, the result will be undefined.
- The type of the assigned object does not influence the type of the assignment expression.

# Assigning Conversions

- type of assignment expression to type of assigned object
- type of initializer expression to type of initialized object
- type of function call argument to type of formal parameter
- type of return statement expression to return type of function
- type of case label expression to type of controlling expression

# Balancing Conversions

- two operands of a binary operator are balanced to a common type
- this always happens after integral promotion

# Dangers of Type Conversion

- Loss of value
- Loss of sign
- Loss of precision

- **MISRA-C++ Rule 5-0-1:** The value of an expression shall be the same under any order of evaluation that the standard permits.
- **MISRA-C++ Rule 5-0-2:** Limited dependence should be placed on C++ operator precedence rules in expressions.
- **MISRA-C++ Rule 5-0-3:** A cvalue expression shall not be implicitly converted to a different underlying type.
- **MISRA-C++ Rule 5-0-4:** An implicit integral conversion shall not change the signedness of the underlying type.
- **MISRA-C++ Rule 5-0-8:** An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
- **AV Rule 180:** Implicit conversions that may result in a loss of information shall not be used.

# Classes

# Classes

- Encapsulation is your friend - use private for all member variables.
- Prevent undesired access to private member variables.
- Avoid Multiple Inheritance
  (except for multiple interface classes).
- Be careful with compiler-generated default constructors and assignment operators.

# Dynamic Memory

- Use of new/delete is strongly restricted.
- **AV Rule 206:** Allocation/deallocation from/to the free store (heap) shall not occur after initialization.
- **MISRA-C++ Rule 18-4-1:** Dynamic heap memory allocation shall not be used.

# Safer C++

# Safer C++

- RAII (Resource Acquisition is Initialization)
- Exceptions
- Templates

```
...

static Mutex mutex;
mutex.lock();

...

mutex.unlock();

...
```

return

throw

break

continue

```
class ScopedLock
{
public:
    ScopedLock(Mutex& mutex):
        _mutex(mutex)
    {
        _mutex.lock();
    }

    ~ScopedLock()
    {
        _mutex.unlock();
    }

private:
    ScopedLock();
    ScopedLock(const ScopedLock&);
    ScopedLock& operator = (const ScopedLock&);

    Mutex& _mutex;
};
```

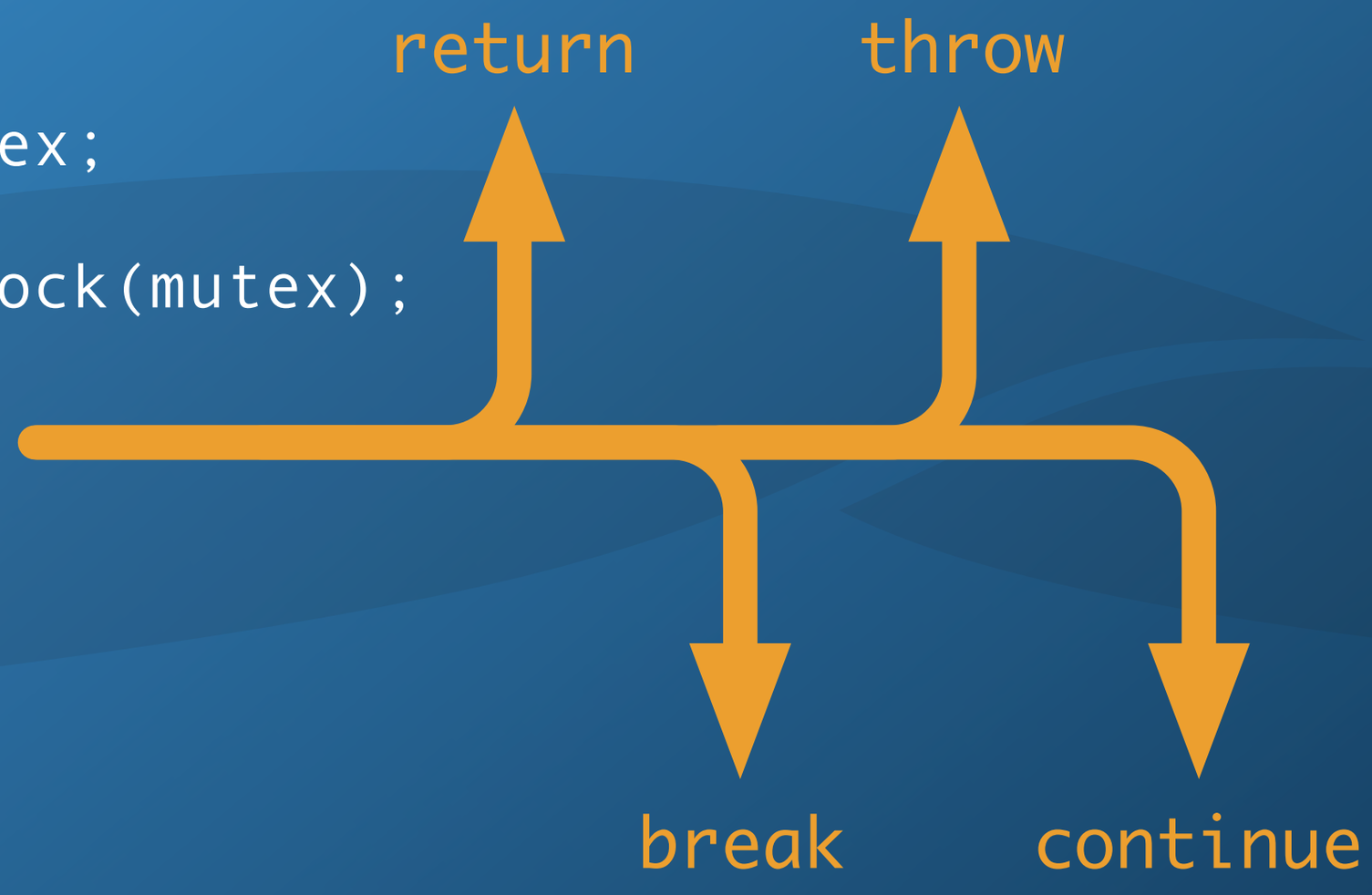
```
...

static Mutex mutex;
{
    ScopedLock lock(mutex);

    ...

}

...
```



# Exceptions

- Error handling in C is a constant pain.
- Choose your poison:
  - global (or thread-local) error code/status variable
  - return value or output parameter indicating error
  - combination of both

```
int rc = f1();
if (rc == OK)
{
    rc = f2();
    if (rc == OK)
    {
        rc = f3();
        if (rc == OK)
        {
            rc = f4();
            // ...
        }
        else
        {
            handleError(rc);
        }
    }
    else
    {
        handleError(rc);
    }
}
else
{
    handleError(rc);
}
```

```
try
{
    f1();
    f2();
    f3();
    f4();
}
catch (Exception& exc)
{
    handleError(exc);
}
```

# Exception Safety

- There's one potential problem with exceptions...
- An exception in the wrong place may leave an object in an inconsistent state.

```
SimpleString& SimpleString::operator = (const SimpleString& str)
{
    if (&str != this)
    {
        delete [] _pData;
        _length = str._length;
        _pData  = new char[_length];
        memcpy(_pData, str._pData, _length);
    }
    return *this;
}
```

# Exception Safety

- **Basic Guarantee**
  the operation always leaves objects in a valid state and does not leak resources
- **Strong Guarantee**
  in addition to the basic guarantee, the operation either succeeds or leaves all objects unchanged
- **Nothrow Guarantee**
  the operation never throws an exception

# Achieving The Strong Guarantee

```
SimpleString& SimpleString::operator = (const SimpleString& str)
{
    SimpleString tmp(str);
    swap(tmp);
    return *this;
}
```

Nothrow Guarantee

# The Trouble With Exceptions

- Exceptions have a significant overhead
  - memory
  - run-time
    - do not forget the overhead of manual error checking
- Exceptions may have non-deterministic run-time behavior
  - bad news for real-time code
  - still a research topic

# Templates

- Templates are an important tool for writing safe software!
- Implement efficient range-checked arrays.
- Implement efficient fixed-point arithmetic.
- Keep it simple!

```
template <class T0,
            class T1 = NullTypeList,
            class T2 = NullTypeList,
            class T3 = NullTypeList,
            class T4 = NullTypeList,
            class T5 = NullTypeList,
            class T6 = NullTypeList,
            class T7 = NullTypeList,
            class T8 = NullTypeList,
            class T9 = NullTypeList,
            class T10 = NullTypeList,
            class T11 = NullTypeList,
            class T12 = NullTypeList,
            class T13 = NullTypeList,
            class T14 = NullTypeList,
            class T15 = NullTypeList,
            class T16 = NullTypeList,
            class T17 = NullTypeList,
            class T18 = NullTypeList,
            class T19 = NullTypeList>
struct Tuple
{
            typedef typename TypeListType<T0,T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType Type;

            enum TupleLengthType
            {
                        length = Type::length
            };

            Tuple():_data()
            {
            }

            Tuple(typename TypeWrapper<T0>::CONSTTYPE& t0,
                        typename TypeWrapper<T1>::CONSTTYPE& t1 = POCO_TYPEWRAPPER_DEFAULTVALUE(T1),
                        typename TypeWrapper<T2>::CONSTTYPE& t2 = POCO_TYPEWRAPPER_DEFAULTVALUE(T2),
                        typename TypeWrapper<T3>::CONSTTYPE& t3 = POCO_TYPEWRAPPER_DEFAULTVALUE(T3),
                        typename TypeWrapper<T4>::CONSTTYPE& t4 = POCO_TYPEWRAPPER_DEFAULTVALUE(T4),
                        typename TypeWrapper<T5>::CONSTTYPE& t5 = POCO_TYPEWRAPPER_DEFAULTVALUE(T5),
                        typename TypeWrapper<T6>::CONSTTYPE& t6 = POCO_TYPEWRAPPER_DEFAULTVALUE(T6),
                        typename TypeWrapper<T7>::CONSTTYPE& t7 = POCO_TYPEWRAPPER_DEFAULTVALUE(T7),
                        typename TypeWrapper<T8>::CONSTTYPE& t8 = POCO_TYPEWRAPPER_DEFAULTVALUE(T8),
                        typename TypeWrapper<T9>::CONSTTYPE& t9 = POCO_TYPEWRAPPER_DEFAULTVALUE(T9),
                        typename TypeWrapper<T10>::CONSTTYPE& t10 = POCO_TYPEWRAPPER_DEFAULTVALUE(T10),
                        typename TypeWrapper<T11>::CONSTTYPE& t11 = POCO_TYPEWRAPPER_DEFAULTVALUE(T11),
                        typename TypeWrapper<T12>::CONSTTYPE& t12 = POCO_TYPEWRAPPER_DEFAULTVALUE(T12),
                        typename TypeWrapper<T13>::CONSTTYPE& t13 = POCO_TYPEWRAPPER_DEFAULTVALUE(T13),
                        typename TypeWrapper<T14>::CONSTTYPE& t14 = POCO_TYPEWRAPPER_DEFAULTVALUE(T14),
                        typename TypeWrapper<T15>::CONSTTYPE& t15 = POCO_TYPEWRAPPER_DEFAULTVALUE(T15),
                        typename TypeWrapper<T16>::CONSTTYPE& t16 = POCO_TYPEWRAPPER_DEFAULTVALUE(T16),
                        typename TypeWrapper<T17>::CONSTTYPE& t17 = POCO_TYPEWRAPPER_DEFAULTVALUE(T17),
                        typename TypeWrapper<T18>::CONSTTYPE& t18 = POCO_TYPEWRAPPER_DEFAULTVALUE(T18),
                        typename TypeWrapper<T19>::CONSTTYPE& t19 = POCO_TYPEWRAPPER_DEFAULTVALUE(T19)):
                        _data(t0, typename TypeListType<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t1, typename TypeListType<T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t2, typename TypeListType<T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t3, typename TypeListType<T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t4, typename TypeListType<T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t5, typename TypeListType<T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t6, typename TypeListType<T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t7, typename TypeListType<T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t8, typename TypeListType<T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t9, typename TypeListType<T10,T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t10, typename TypeListType<T11,T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t11, typename TypeListType<T12,T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t12, typename TypeListType<T13,T14,T15,T16,T17,T18,T19>::HeadType
                                    (t13, typename TypeListType<T14,T15,T16,T17,T18,T19>::HeadType
                                    (t14, typename TypeListType<T15,T16,T17,T18,T19>::HeadType
                                    (t15, typename TypeListType<T16,T17,T18,T19>::HeadType
                                    (t16, typename TypeListType<T17,T18,T19>::HeadType
                                    (t17, typename TypeListType<T18,T19>::HeadType
                                    (t18, typename TypeListType<T19>::HeadType
                                    (t19, NullTypeList())))))))))))))))))))
            {
            }

            template <int N>
            typename TypeGetter<N, Type>::ConstHeadType& get() const
            {
                        return Getter<N>::template get<typename TypeGetter<N, Type>::HeadType, typename Type::HeadType, typename Type::TailType>(_data);
            }

            template <int N>
            typename TypeGetter<N, Type>::HeadType& get()
            {
                        return Getter<N>::template get<typename TypeGetter<N, Type>::HeadType, typename Type::HeadType, typename Type::TailType>(_data);
            }

            template <int N>
            void set(typename TypeGetter<N, Type>::ConstHeadType& val)
            {
                        Getter<N>::template get<typename TypeGetter<N, Type>::HeadType, typename Type::HeadType, typename Type::TailType>(_data) = val;
            }

            bool operator == (const Tuple& other) const
            {
                        return _data == other._data;
            }

};
```

# Conformance

# Conformance

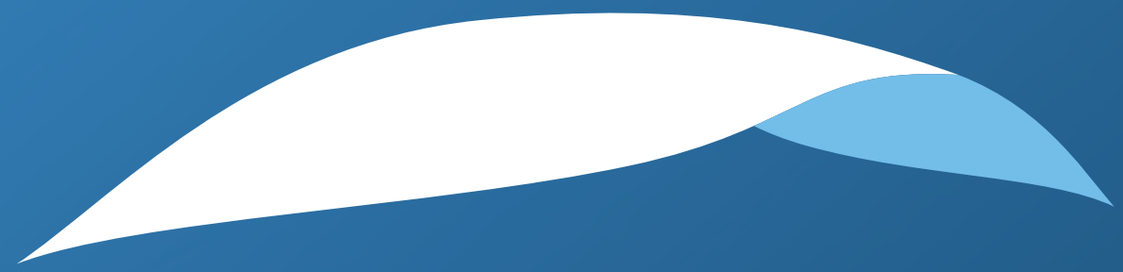
- Manual checking of rule compliance is impossible.
- Tools must be used to verify compliance to MISRA or AV rules.
- MISRA-C++ recommends the creation of a Compliance Matrix, detailing how each rule is checked.
- A deviation procedure must be defined.

# Conclusions

# Conclusions

- C++ can (and has been successfully) used for the development of safety-critical systems.
- Use of C++ requires the adoption of a coding standard.
- Tools must be used to check the compliance of the source code with the chosen coding standard.

# Q&A



# applied informatics

**C++ Libraries, Tools and Services**
**to Simplify Your Life.**

info@appinf.com | www.appinf.com | +43 4253 32596