The C++ Core Guidelines for Safer Code

Rainer Grimm

Training, Coaching and, Technology Consulting

www.ModernesCpp.de

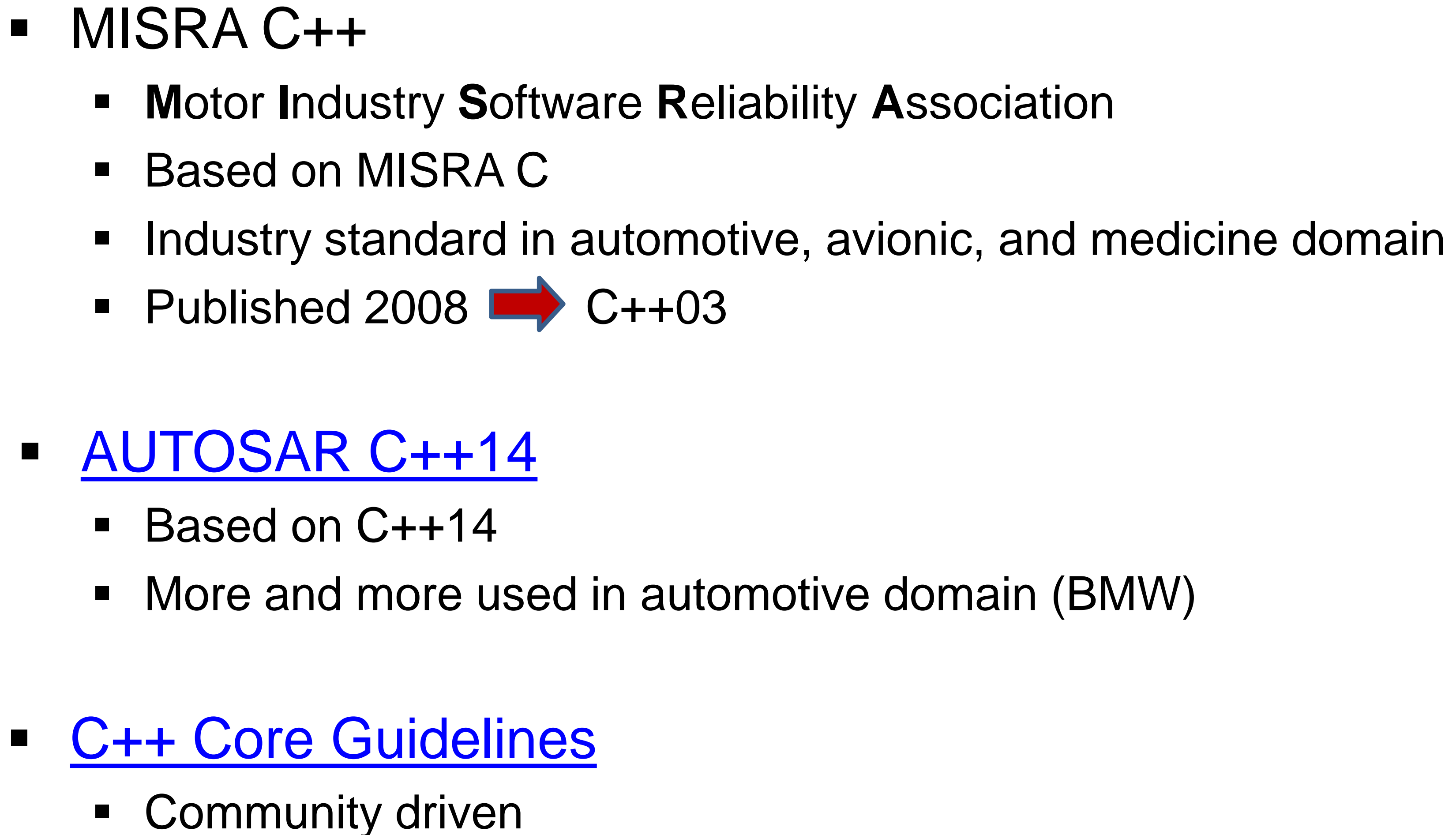
Guidelines

Best Practices for the Usage of C++

- Why do we need guidelines?
  - C++ is a complex language in a complex domain.
  - A new C++ standard is published all three years.
  - C++ is used in safety-critical systems.

➡ Reflect your coding habits.

Most Prominent Guidelines



Overview

- Philosophy
- Interfaces
- Functions
- Classes and class hierarchies
- Enumerations
- Resource management
- Expressions and statements
- Error handling
- Constants and immutability
- Templates and generic programming
- Concurrency
- The standard library
- Guideline support library

Syntactic Form

- About 350 rules and a few hundred pages
- Each rule follows a similar structure
  - The rule itself
  - A rule reference number
  - Reason(s)
  - Example(s)
  - Alternative(s)
  - Exception(s)
  - Enforcement
  - See also(s)
  - Note(s)
  - Discussion

Guidelines Support Library (GSL)

A small library for supporting the guidelines of the C++ core guidelines.

- Implementations are available for
  - Windows, Clang, and GCC
  - GSL-lite works with C++98, and C++03
- Components
  - Views
  - Owner
  - Assertions
  - Utilities
  - Concepts

Interfaces

I.11: Never transfer ownership by a raw pointer (T*)

- `func(value)`
  - `func` has an independent copy of value and the runtime is the owner
- `func(pointer*)`
  - `pointer` is borrowed but can be zero
  - `func` is not the owner and must not delete the pointer
- `func(reference&)`
  - `reference` is borrowed but can't be zero
  - `func` is not the owner and must not delete the reference
- `func(std::unique_ptr)`
  - `std::unique_ptr` is the owner of the pointer
- `func(std::shared_ptr)`
  - `std::shared_ptr` is an additional owner of the pointer
  - `std::shared_ptr` extends the lifetime of the pointer

Interfaces

I.13: Do not pass an array as a single pointer

- What if `n` is wrong?

```
void copy(const int* p, int* q, int n); // copy from [p:p+n] to [q:q+n]
void draw(double* p, int n);             // poor interface; poor code
```

- Use span from the GSL

```
void copy(span<const int> r, span<int> r2);  // copy r to r2
void draw(span<int> p);

int a[100];
int b[100];
...
copy(a, b);

std::vector<int> vec;
...
draw(vec);
```

Functions

F.43: Never (directly or indirectly) return a pointer or a reference to a local object

```
int* f()
{
    int fx = 9;
    // ...
    return &fx;  // BAD
}

int& f()
{
    int x = 7;
    // ...
    return x;    // BAD
}
```

Classes

C.2: Use class if the class has an invariant; use struct if the data members can vary independently

- The data members can vary independently

```
struct Pair {
    string name;
    int volume;
};
```

- The data members has an invariant

```
class Date {
public:
    // validate that {yy, mm, dd} is a valid date and initialize
    Date(int yy, Month mm, char dd);
    // ...
private:
    int y;
    Month m;
    char d;    // day
};
```

Classes

C.20: If you can avoid defining any default operations, do

C.21: If you define or =delete any default operation, define or =delete them all

If you write...

The compiler supplies...

	None	dtor	Copy-ctor	Copy-op=	Move-ctor	Move-op=
dtor	✓	♦	✓	✓	✓	✓
Copy-ctor	✓	✓	♦	✓	✗	✗
Copy-op=	✓	✓	✓	♦	✗	✗
Move-ctor	✓	✗	Overload resolution will result in copying		♦	✗
Move-op=	✓	✗			✗	♦

Copy operations are independent...

Move operations are not.

Sticky Bits - Becoming a Rule of Zero Hero

Enum

Enum.3: Prefer enum classes over “plain” enums

```
enum struct Color2: char{
  red= 126,
  blue, // 127
  green // 128 => ERROR
};
```

```
main.cpp:4:3: error: enumerator value '128' is outside the range of underlying type 'char'
   green // 128 => ERROR
   ^~~~~
```

- Don't implicitly convert to `int`.
- Don't pollute the global namespace.
- The default type is `int,` but you can adjust it.

Resource Management

R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)

- RAII-Idiom (Resource Acquisition Is Initialization)
  - The lifetime of a resource is bound to an automatic object.
  - The resource will be initialized in the constructor of the object; released in the destructor of the object.

- Used
  - Containers of the Standard Template Library and `std::string`
  - Smart pointers
  - Locks
  - `std::jthread` (C++20)

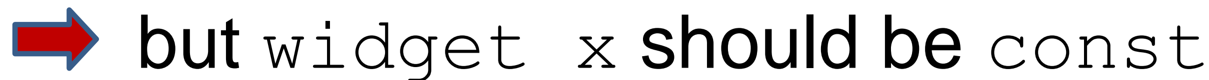
Resource Management

```
class ResourceGuard{
  private:
    const std::string resource;
  public:
    ResourceGuard(const std::string& res):resource(res){
      std::cout << "Acquire the " << resource << "." <<  std::endl;
    }
    ~ResourceGuard(){
      std::cout << "Release the "<< resource << "." << std::endl;
    }
};

int main(){

  ResourceGuard resGuard1{"memoryBlock1"};

  {
    ResourceGuard resGuard2{"memoryBlock2"};
  }

  try{
      ResourceGuard resGuard3{"memoryBlock3"};
      throw std::bad_alloc();
  }
  catch (std::bad_alloc& e){
      std::cout << e.what();
  }

}
```

Expressions and Statements

ES.28: Use lambdas for complex initialization, especially of const variables

```
widget x;   // should be const, but:
for (auto i = 2; i <= N; ++i) {             // this could be some
    x += some_obj.do_something_with(i);  // arbitrarily long code
}                                        // needed to initialize x
// from here, x should be const, but we can't say so in code in this style
```




```
const widget x = [&]{
    widget val;                                // widget has a default constructor
    for (auto i = 2; i <= N; ++i) {            // this could be some
        val += some_obj.do_something_with(i);  // arbitrarily long code
    }                                          // needed to initialize x
    return val;
}();
```

Expressions and Statements

ES.100: Don’t mix signed and unsigned arithmetic

```
#include <iostream>

int main(){

  int x = -3;
  int y = 7;

  std::cout << x - y << std::endl;  // -10
  std::cout << x + y << std::endl;  // 4
  std::cout << x * y << std::endl;  // -21
  std::cout << x / y << std::endl;  // 0

}
```

```
#include <iostream>

int main(){

  int x = -3;
  unsigned int y = 7;

  std::cout << x - y << std::endl;  // 4294967286
  std::cout << x + y << std::endl;  // 4
  std::cout << x * y << std::endl;  // 4294967275
  std::cout << x / y << std::endl;  // 613566756

}
```

Concurrency and Parallelism

CP.8: Don't try to use volatile for synchronization

- `std::atomic`
  - Atomic (thread-safe) access to shared state.
- `volatile`
  - Access to special memory, for which read and write optimisations are not allowed.

Java volatile == C++ atomic != C++ volatile

Concurrency and Parallelism

CP.9: Whenever feasible use tools to validate your concurrent code

Thread Sanitizer detects data races at runtime.

```
g++ threadArguments.cpp -fsanitize=thread -g -o threadArguments
```

Concurrency and Parallelism

CppMem: Interactive C/C++ memory model

(1)

Model

○ standard ◉ preferred ○ release_acquire ○ tot ○ relaxed_only

Program

examples/Paper | sc_atomics.c

◉ C ○ Execution

```
// contrasting with data_race.c, this
// shows a concurrent use of sc_atomic that does
// not have a data race
int main() {
  atomic_int x = 2;
  int y = 0;
  {{{ x.store(3);
  ||| y = ((x.load())==3);
  }}};
  return 0; }
```

(2)

run | reset | help **8 executions; 2 consistent, all race free**

Computed executions

(3)

Display Relations

☑ sb ☐ asw ☐ dd ☐ cd
☑ rf ☑ mo ☑ sc ☑ lo
☐ hb ☐ vse ☐ ithb ☑ sw ☐ rs ☐ hrs ☑ dob ☐ cad
☑ unsequenced_races ☑ data_races

(4)

Display Layout

○ dot ○ neato_par ◉ neato_par_init ○ neato_downwards

☐ tex

edit display options

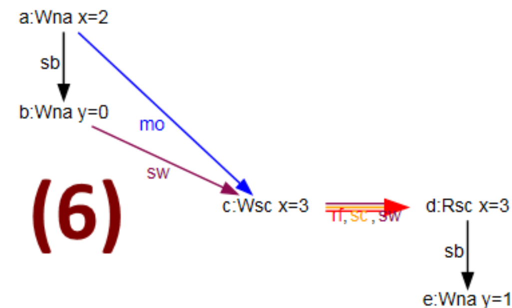
(5)

Execution candidate no. 1 of 8

previous consistent | previous candidate | next candidate | next consistent | 1 | goto

Model Predicates

```
consistent_race_free_execution = true
☑ consistent_execution = true
☑   assumptions                         = true
☑   well_formed_threads                 = true
☑   well_formed_rf                      = true
☑   locks_only_consistent_locks         = true
☑   locks_only_consistent_lo            = true
☑   consistent_mo                       = true
☑   sc_accesses_consistent_sc           = true
☑   sc_fenced_sc_fences_heeded          = true
☑   consistent_hb                       = true
☑   consistent_rf                       = true
☑     det_read                          = true
☑     consistent_non_atomic_rf          = true
☑     consistent_atomic_rf              = true
☑     coherent_memory_use               = true
☑     rmw_atomicity                     = true
☑     sc_accesses_sc_reads_restricted   = true
  unsequenced_races    are absent
  data_races           are absent
  indeterminate_reads  are absent
  locks_only_bad_mutexes are absent
```



Files: out.exc, out.dot, out.dsp, out.tex

Error Handling

E.7: State your preconditions

E.8: State your postconditions

- **Precondition**: should hold upon entry in a function.
- **Postcondition**: should hold upon exit from the function
- **Assertion**: should hold at its point in the computation.

```
int push(queue& q, int val)
  [[ expects: !q.full() ]]
  [[ ensures !q.empty() ]]{
  ...
  [[assert: q.is_ok() ]]
  ...
}
```

Constants and Immutability

Con.2: By default, make member functions const

```
struct Immutable{
    std::mutex m;
    int read() {
        std::lock_guard<std::mutex> lck(m);
        // critical section
        ...
    }
};
```

Constants and Immutability

- **Physical constness**:
  - The object is const and can not be changed.
- **Logical constness**:
  - The object is const but could be changed.

```
struct Immutable{
    mutable std::mutex m;
    int read() const {
        std::lock_guard<std::mutex> lck(m);
        // critical section
        ...
    }
};
```

Templates and Generic Programming

T.10: Specify concepts for all template arguments

- **Concepts** are a compile-time predicate.

- Usage

```
template<Integral T>
T gcd(T a, T b){
  if( b == 0 ){ return a; }
  else{
    return gcd(b, a % b);
  }
}
```

- Definition

```
template<typename T>
concept bool Integral(){
  return std::is_integral<T>::value;
}
```

Templates and Generic Programming

- Core language concepts
  - Same
  - DerivedFrom
  - ConvertibleTo
  - Common
  - Integral
  - Signed Integral
  - Unsigned Integral
  - Assignable
  - Swappable
- Comparison concepts
  - Boolean
  - EqualityComparable
  - StrictTotallyOrdered
- Object concepts
  - Destructible
  - Constructible
  - DefaultConstructible
  - MoveConstructible
  - Copy Constructible
  - Movable
  - Copyable
  - Semiregular
  - Regular
- Callable concepts
  - Callable
  - RegularCallable
  - Predicate
  - Relation
  - StrictWeakOrder

Templates and Generic Programming

```
template <class T>
concept bool Integral() {
    return is_integral<T>::value;
}

template <class T>
concept bool SignedIntegral() {
    return Integral<T>() && is_signed<T>::value;
}

template <class T>
concept bool UnsignedIntegral() {
    return Integral<T>() && !SignedIntegral<T>();
}
```