Richard Kaiser
www.rkaiser.de

The Misra C++:2023 Guidelines

10/25

Contents

1 Introduction

Since it is impossible to cover all the 179 MISRA C++: 2023 rules in a 60 minutes talk, I will present only some basic rules to give you an idea how

If you want a more detailed coverage, you can attend my one day online training

Mo. 24.11.2025 The MISRA C++:2023 Guidelines

And if you are interested in a training to programming embedded systems using modern C++, please engage me for an inhouse training

in German: Embedded C++Grundkurs / Embedded C++ Aufbaukurs
in English: Embedded C++ Basics Training / Embedded C++ Advanced Training

or come to my public trainings in German

Technische Akademie Esslingen: Embedded C++ Grundkurs
Technische Akademie Esslingen: Embedded C++ Aufbaukurs

or book an online training

https://meetingcpp.com/mcpp/training/trainingslisting.php

These trainings cover the full modern C++ together with the MISRA C++: 2023 guidelines. After introducing some language feature, examples are given that satisfy and violate the guidelines.

Many of the MISRA Guidelines are not only guidelines for embedded applications but are just good style that is recommended for applications that are not from the embedded field. Many MISRA C++ 2023 Guidelines are close or even identical to the C++ Core Guidelines

https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

Compared to the Core Guidelines, the MISRA guidelines are more streamlined and more consistent. This allows code checkers that check your code for compliance. For the Core Guidelines, there exist no tool that is close to the completeness of these MISRA checkers.

1.1 The Intention of the MISRA C++ Guidelines

The Misra Guidelines define a subset of C++ in which the opportunity to make mistakes is removed or reduced.

Originally created to support safety-critical systems in domains such as automotive, aerospace, and medical devices, MISRA has long been a cornerstone of dependable embedded software development.

However, the first MISRA C++ Guidelines (published in 2008 and based on C++03) struggled to keep pace with the evolution of the language. As C++11 and later standards introduced powerful new features, developers often encountered false “non-compliance” warnings, and the rules themselves proved overly restrictive—prohibiting dynamic memory and most use of the standard library. In addition, the MISRA C++ 2008 guidelines had very much C in mind, and not much C++. I completely agree with Peter Sommerlad when he says in his talk

https://www.youtube.com/watch?v=v8JmiIdi1wg

at Minute 3.05 that he finds the 2008 guidelines absolutely miserable. Every programmer that had to follow this rules had cannonballs bound to his feet making very hard to get forward.

That's why the community needed an update—and it has arrived. MISRA C++:2023, published in October 2023, brings the guidelines up to date with C++17, modernizing the rules while maintaining MISRA's focus on safety and reliability. The new version embraces modern C++ features, allows standard library containers, and significantly reduces unnecessary restrictions.

The result is a modern, pragmatic, and widely applicable style guide—not just for safety-critical embedded systems, but for any organization seeking clean, maintainable, and verifiably correct C++ code. With the support of today's advanced static analysis tools, achieving MISRA compliance has never been easier. While it was a hard punishment if you have to be compliant with the 2008 guidelines, the 2023 guidelines are a big assistance to bring you to the right way.

This talk introduces some key ideas behind MISRA C++:2023 with many examples. The MISRA C++: 2023 guidelines are sometimes compared to the MISRA C++2008, MISRA C and the C++ Core Guidelines

1.2 Static Guideline Checkers

There are a few static analysis tools that you can use to check your code for compliance with the Guidelines. For this talk, I used the ones listed below to give you examples.

By introducing these products, I don't want to express that I consider any of them to be better or worse than any other one. I only want to show how they can help you.

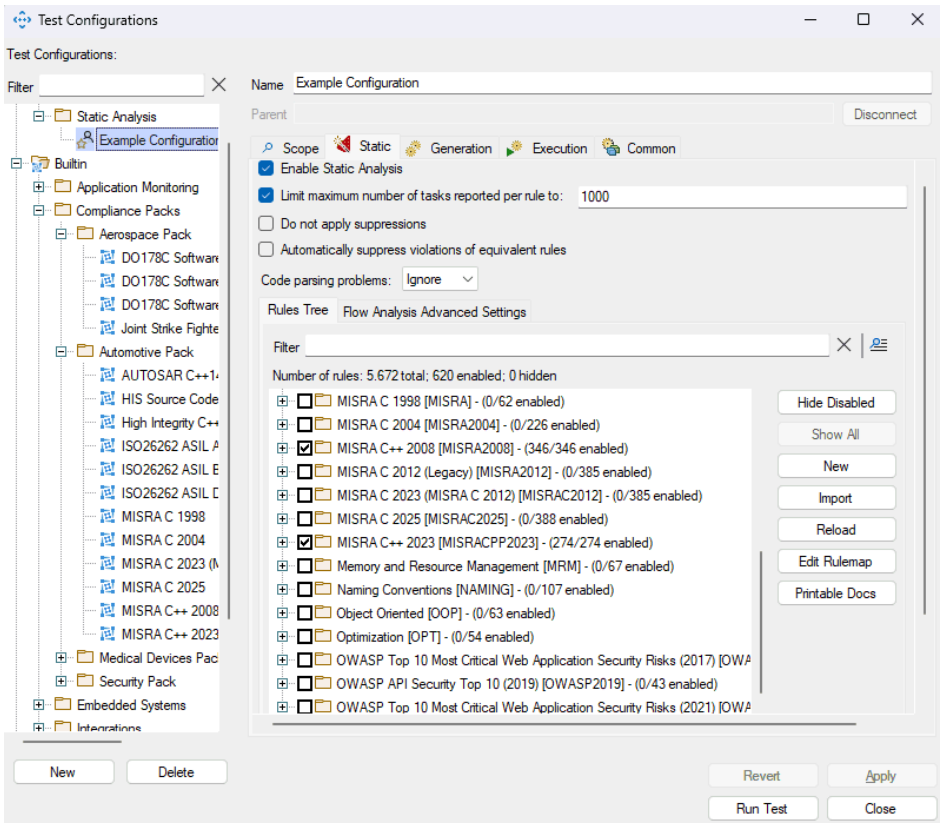
Except of the Visual Studio Core Guideline checker, these static analysis tools are not free. Concerning the price for these checkers, please take into account the time they save for your code reviews.

1.2.1 Parasoft C/C++test for checking MISRA compliance

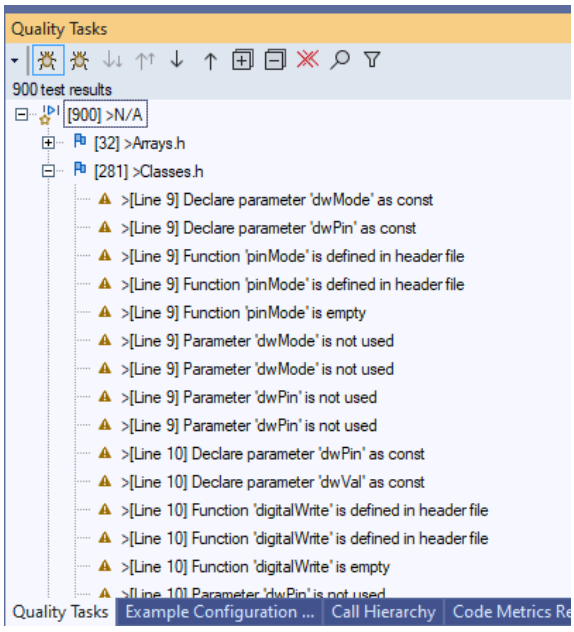
Parasoft kindly provided me an evaluation license for **Parasoft C/C++test**, which was used to create numerous examples illustrating the MISRA rules.

For the students of my trainings, Parasoft offers a 30-day evaluation licence. To take advantage of this offer, please contact info@parasoft.com with the subject "MISRA Trial".

It allows a flexible configuration of the checks



and shows the violations with additional informations:

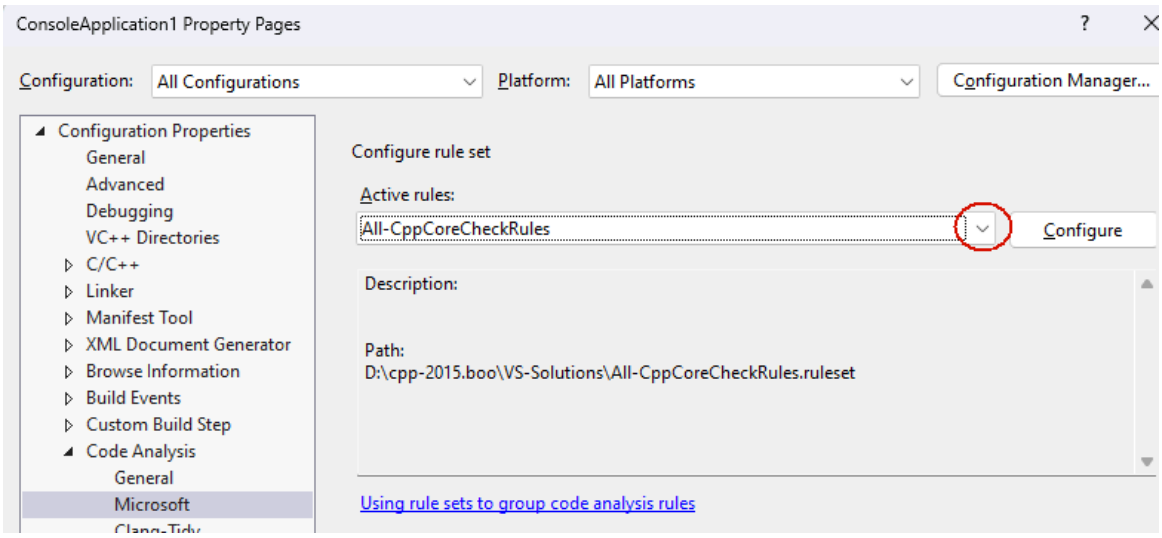


1.2.2 CppCheck

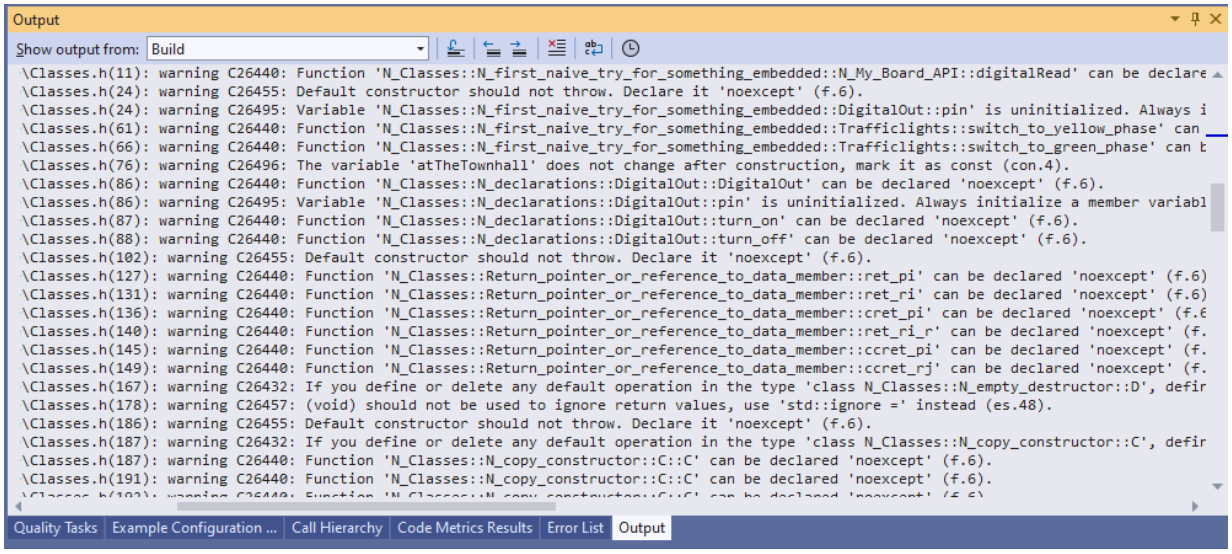
CppCheck kindly provided me a license. However, it is not yet covered in this version of this talk because I got it too late.

1.2.3 Visual Studio Core Guideline Checker

Visual Studio contains a C++ Core Guideline checker that is activated with *Project|Properties*



Violations of the core Guidelines are shown in the Output Window:



1.3 Company wide internal guidelines

Many companies have internal guidelines, often created some time ago. Since it is not easy to develop good guidelines that are up to date, it may be helpful to align them with the MISRA C++ 2023 guidelines.

1.4 Some embedded example

In this talk we will start with a simple embedded example to see where the MISRA C++:2023 Guidelines come in.

Since I do not want to go into the detailed subtleties of some real microcontroller, we will start very naïve, use a static MISRA checker to help us to detect the weaknesses in my code. compliant and improve the code to get it MISRA compliant.

```
namespace N_first_naive_try_for_something_embedded
{
  namespace N_My_Board_API
    { // e.g. the Arduino API
      void pinMode(uint32_t dwPin, uint32_t dwMode) {};
      void digitalWrite(uint32_t dwPin, uint32_t dwVal) {};
      int digitalRead(uint32_t ulPin) { return 0; /* make the compiler happy */ };
      #define INPUT   0x0
```

```
      #define OUTPUT  0x1
      #define LOW     0x0
      #define HIGH    0x1
    }

  using namespace N_My_Board_API;

  class DigitalOut
    {
      int pin;
    public:
      DigitalOut() {} // not good - only to make the compiler happy
      DigitalOut(int pin_)
      {
        pin = pin_;
        pinMode(pin_, OUTPUT);
      }
      void turn_on()
      {
        digitalWrite(pin, HIGH); // turn LED on:
      }
      void turn_off()
      {
        digitalWrite(pin, LOW); // turn LED on:
      }
    };

  class Trafficlights
    {
      DigitalOut red_light, yellow_light, green_light;
    public:
      Trafficlights(int pin_red_light, int pin_yellow_light, int pin_green_light)
      {
      }
      void switch_to_red_phase()
      {
        red_light.turn_on();
        yellow_light.turn_off();
        green_light.turn_off();
      }

      void switch_to_yellow_phase()
      { // similar to void switch_to_red_phase()
      // ...
      }

      void switch_to_green_phase()
      { // similar to void switch_to_red_phase()
      // ...
      }
    };

    Trafficlights atTheTownhall(1,2,3); // MISRACPP2023 - 6_7_2.a: Do not define the
                                                        'atTheTownhall' global variable
} //  namespace N_first_naive_try_for_something_embedded
```

2 MISRA Guidelines for Basic Types and Statements

This first version violates several MISRA Guidelines.

We will start with violations concerning the basic declaration rules. More violations are covered later.

2.1 Integral types

The MISRA C++:2023 Guidelines (Rule 6.9.2) recommend using a name for an integer data type that includes the width. This is the reason for the compliance violation in

```
DigitalOut(int pin_) // MISRACPP2023 - 6_9_2.a: Do not use the 'int' standard integer type
```

This violation is fixed by using the names

int8_t, *int16_t*, *int32_t*, *int64_t*: 8-, 16-, 32- or 64-Bit signed
uint8_t, *uint16_t*, *uint32_t*, *uint64_t*: 8-, 16-, 32- or 64-Bit unsigned

availalable since C++11 after

```
#include <cstdint>
```

These names are not independent data types, but merely other names for *char*, *int*, etc.

To become MISRA compliant, replace the type

```
DigitalOut(uint32_t pin_) // MISRACPP2023 - 6_9_2.a compliant
```

C++14 extensions like quotes as digit separators '

```
uint32_t x = 1'000'000;
```

or binary literals

```
int b = 0b00101; // binary literal, decimal 5
```

are MISRA compliant. Octal literals except 0 should be avoided:

```
int32_t x = 017; // MISRA2008.2_13_2_a and MISRACPP2023 - 5_13_3.a: Octal constant '017' is
                    used
```

2.2 Variables

The C++ Core Guidelines ES.10 and the **MISRA** C++:2023 Guidelines (Rule **10.0.1**) advise against multiple declarations in a declaration statement, as these can lead to misunderstandings in connection with pointers and references (see Section 8.1). Exceptions are structured bindings and parameter lists of a function, for which multiple declarations are permitted in ES.10.

The declaration

```
DigitalOut red_light, yellow_light, green_light; // MISRA2008.8_0_1 and MISRACPP2023 -
  10_0_1.a: Declare variable 'green_light' in a separate declaration statement
```

violates the MISRA-Rule 10.0.1 as well as the C++ Core Guideline

```
C++ Core Guidelines - ES.10: Declare one name (only) per declaration
```

as well as several other rules which will be presented later.

2.3 Global and local Variables

Global variables shall not be used:

```
namespace N_global_variables
{
  int32_t global_1; // MISRACPP2023 - 6_7_2.a: Do not define the 'global_1' global variable
```

However, global constants are MISRA compliant:

```
  const int32_t global_c = 0; // MISRA compliant
```

Local variables are required to be initialized:

```
  void f()
  {
    int32_t local_1; // MISRACPP2023 - 11_6_1.a: Variable 'local_1' should be explicitly
                                                                          initialized
    int32_t local_2 = 0; // MISRA compliant
  }
```

The MISRA C++:2023 Guidelines (Rule 6.4.1) and the C++ Core Guidelines ES.12 require that **hidden variables should be avoided**: Nested variables shall not hide names from a surrounding scope, This also applies to the names of function parameters.

```
  int32_t x; // MISRACPP2023 - 6_7_2.a: Do not define the 'x' global variable

  void f()
  {
    int32_t x = 1; // MISRA2008.2_10_2_a and MISRACPP2023 - 6_4_1.g: The variable 'x'
                   hides the variable with the same name declared in the outer scope
    {
      int32_t x = 1; // MISRA2008.2_10_2_a and MISRACPP2023 - 6_4_1.g: The variable 'x'
                     hides the variable with the same name declared in the outer scope
    }
  }
}
```

2.4 Type inference with *auto*

auto can be helpful when working with libraries and can save you the tedious task of searching for the return type of a function.

Example: Different versions of a library function sometimes differ in their return type:

```
int analogRead(pin_size_t pinNumber); // Raspberry Pi Pico
uint16_t analogRead(uint8_t pin);     // ESP32
uint32_t analogRead(uint32_t ulPin);  // STM32 Nucleo
```

With

```
auto z = analogRead(17);
```

you get the correct data type for z for each platform.

2.5 Initializerlists and Conversions

Since C++11, restrictive conversions can be prevented with an initializer list. The compiler then generates an error message for such a conversion.

In the simplest case, an initialisation list consists of the value specified between curly brackets. It can be specified when **defining** a variable directly after its name or after its name and an equal sign. The latter notation is also possible for an **assignment**.

Beispiel: Die Initialisierung von *i* und *j* mit einer Initialisierungsliste

```
int i{ 17 };    // i=17
int j = { 18 }; // j=18
```

hat denselben Effekt wie eine Initialisierung mit der klassischen Schreibweise

```
int i(17);      // i=17
int j = 18;     // j=18
```

Eine Initialisierungsliste kann auch in einer Zuweisung verwendet werden:

```
i = { 19 };     // wie i=19
```

Den Vorteil der etwas umständlichen Schreibweise von Initialisierungslisten sieht man in den nächsten Beispielen. Gibt man in der Initialisierungsliste einen Wert an, den der Compiler kennt (z.B. ein Literal), muss dieser Wert im Datentyp der linken Seite darstellbar sein. Ansonsten erzeugt der Compiler eine Fehlermeldung.

Beispiel: Mit einem 32-Bit *int*-Datentyp erzeugen die Initialisierungen

```
int i { 100'000'000'000};   // Fehler: einschränkende
int j = { 100'000'000'000}; //         Konvertierung
```

wie auch die Zuweisung

```
i = { 100'000'000'000 }; // Fehler: einschränkende Konvertierung
```

eine Fehlermeldung. Das ist ein Vorteil gegenüber der traditionellen Schreibweise wie in

```
i = 100'000'000'000; // i = 1'215'752'192
```

bei der keine Prüfung auf den Wert durchgeführt wird. Auch die Zuweisung eines Gleitkommawertes an eine Ganzzahlvariable wird nicht akzeptiert:

```
int k{ 3.0 };    // Fehler: einschränkende Konvertierung
int m = { 6.0 }; // Fehler: einschränkende Konvertierung
```

Gibt man in der Initialisierungsliste einen Wert an, den der Compiler nicht kennt (z.B. eine Variable), muss der Datentyp der linken Seite alle Werte des Datentyps in der Initialisierungsliste darstellen können.

Beispiel: Da *char* nicht alle *int*-Werte darstellen kann, erhält man eine Fehlermeldung

```
int i { 0 };
char c = { i }; // Fehler: einschränkende Konvertierung
```

obwohl *char* den aktuellen Wert von i (also 0) darstellen kann:

```
char d = { 0 }; // kein Fehler
```

Initialisierungslisten können für nahezu alle Initialisierungen verwendet werden: Nicht nur für elementare Datentypen, sondern auch für Klassen (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).

In Regel **11.6.1** der **MISRA** C++:2023 Guidelines und den C++ Core Guidelines ES.23 wird **ausdrücklich empfohlen, Initialisierungslisten** gegenüber den traditionellen Schreibweisen zu bevorzugen. Damit kann man insbesondere bei Programmen, die für verschiedene Plattformen (z.B. 32- und 64-Bit, oder verschiedene Prozessoren) kompiliert werden, Fehler vermeiden, die leicht übersehen werden.

2.6 Float types are MISRA C++ 2023 compliant

Rule 0.3.1 of the MISRA C++:2023 Guidelines recommends particular caution when dealing with floating point values, as such values do not always behave as one would intuitively expect. However, float types are allowed in MISRA C++ 2023

```
float pif = 3.14f; // MISRA2008.3_9_2: The basic numerical type 'float' should not be used
double pid = 3.14; // MISRA2008.3_9_2: The basic numerical type 'double' should not be used
// No Misra C++2023 warnings
```

2.7 *if*-statements

2.7.1 Do not use assignments in *if*-conditions

Rule **8.18.2** of the **MISRA** C++:2023 Guidelines recommends that the result of an assignment operator should not be used. This rule helps to detect errors like here:

```
bool do_not_use_the_result_of_an_assignment(uint32_t x, uint32_t y)
{
  if (x = y) // MISRACPP2023 - 8_18_2.a: Assignment operator '=' is used in improper
             context
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

2.7.2 Unconditional *else* in *if*-statements required

The MISRA C++:2023 Guidelines require in Rule 9.4.1 that every nested *if* statement with more than one branch ends with an unconditional *else*.

```
void unconditional_else(uint32_t x)
{
  uint32_t result = 0;
  if (x == 1)
  {
    result = 1;
  }
  else if (x == 2)
  {
    result = 2;
  }
  // MISRACPP2023 - 9_4_1.a: Provide 'else' after the last 'else-if' construct
}
```

2.8 Functions

2.8.1 Discarding function return values

There are MISRA C++ 2023 guidelines that can be ensured by modern compiler features.

MISRA C++:2023 Rule 0.1.2 requires that the return value of function shall be used. This means that all functions with a return type other than *void* are treated as if they were declared with *[[nodiscard]]*.

```
int32_t plus_1(int32_t n)
{
  return n + 1;
}

[[nodiscard]]int32_t plus_2(int32_t n)
{
  return n + 2;
}

void call_plus()
{
  plus_1(1); // MISRA2008.0_3_2 and MISRACPP2023 - 0_1_2.a: Unused function's "plus_1"
                return value
             // no compiler warning
  int32_t p1=plus_1(1); // MISRA compliant regarding the return value
  plus_2(1); // same MISRA warnings as for plus_1: Unused function's "plus_2" return value
             // compiler warning: discarding return value of function with [[nodiscard]]
                attribute
}
```

2.8.2 Recursive Functions

According to rule **8.2.10** of the **MISRA** C++:2023 Guidelines, recursive functions are only permitted for *constexpr* functions or if justification is provided for how the use of the stack is controlled. There is no such restriction in the C++ Core Guidelines (as of May 2024).

2.9 Namespaces

Namespaces are used to avoid name conflicts and to express explicitly that some variables, functions and types belong together.

As you see in this example, there is only one MISRA C++ 2023 warning when using a global *using* declaration, compared to more warnings MISRA C++ 2008:

```
#include <vector>
namespace N_namespaces
{
  size_t test_vector()
  {
```

You can use a name from a namespace with a full qualification of the namespace:

```
    std::vector<int32_t> v1;
    v1.push_back(1);
```

With a ***using* declaration**, **only the** specified element of the namespace is then available under its name, without specifying the namespace:

```
    using std::vector;
    vector<int32_t> v2;
    v2.push_back(2);
```

A namespace alias is often used to abbreviate long namespace names:

```
    namespace S = std;
    S::vector<int32_t> v3;
    v3.push_back(3);
```

With a ***using* directive**, **all** elements of the namespace are available under their name, without specifying the namespace:

```
    using namespace std; // MISRA2008.7_3_4: 'using' directive was found: namespace 'std'
                         // MISRA2008.7_3_6: The 'using' directive shall not be used in a
                              header file

    vector<int32_t> v4;
    v4.push_back(4);

    return v1.size() + v2.size() + v3.size()+ v4.size();
  }

  using namespace std; // MISRA2008.7_3_4: 'using' directive was found: namespace 'std'
  vector<int32_t> v4;
} // namespace N_namespaces
```

A global ***using* declaration** is not MISRA C++ 2023 compliant:

```
using namespace std; // MISRA2008.7_3_1 and MISRACPP2023 - 6_0_3.a: Do not use the 'using'
                        declaration in the global namespace
                     // MISRA2008.7_3_6: The 'using' directive shall not be used in a
                        header file
vector<int32_t> v5; // MISRA2008.7_3_1 and MISRACPP2023 - 6_0_3.a: Do not declare the 'v5'
                        variable in the global namespace
```

2.10 Using C-Libraries and their C++ equivalents

The **C++ standard libraries** are obtained with a header name without '.h', such as

```
#include <cstdlib>
#include <cmath>
```

Most of the **libraries** in the **C standard** are also part of C++. They are available after a *#include* directive with a header name ending in '.h', as in

```
#include <stdlib.h>
#include <math.h>
```

Most **C standard libraries** have versions **adapted to C++**, whose names begin with the letter 'c' and are followed by the name of the C library without the extension '.h'.

Examples: The C standard libraries in the left-hand column correspond to the C++ libraries in the right-hand column:

```
#include <stdlib.h>      #include <cstdlib>
#include <math.h>        #include <cmath>
```

Although there is often no difference between a C library and its C++ version, there are occasionally minor differences.

In den folgenden Beispielen werden diese Bibliotheken zusammen mit einer *using*-Deklaration wie in der linken Spalte verwendet. Die anderen beiden Varianten sind genauso möglich:

```
#include <cmath>          #include <cmath>      #include <math.h>
using namespace std;      using std::sin;       // kein using notw.
```

The C++ Core Guidelines (SL.2) recommend to prefer the **C++ standard library** over other libraries. In particular, this means that the C standard library (i.e. the **.h headers**) should **not** be **used**. The MISRA C++:2023 Guidelines require that the following functions are not used:

MISRA C++:-2023 Regel	Bibliothek (sowohl in der Form <cname> als auch <name.h>)	Funktionen, die nicht verwendet werden dürfen
21.2.1	`<cstdlib>`	*atof*, *atoi*, *atol*, *atoll*
21.2.2 24.5.2	`<cstring>`, `<cstdlib>`, `<cwchar>`, `<cinttypes>`	über 50 Funktionen
21.2.3	`<cstdlib>`	*system*
21.6.2		*new*, *delete*, *malloc*, *free* usw.
21.10.1	`<cstdarg>`	alle
21.10.2	`<csetjmp>`	alle
21.10.3	`<csignal>`	alle
24.5.1	`<cctype>`, `<cwctype>`	Siehe MISRA Rule 24.5.1
25.5.1	`<clocale>`	*setlocale*, *std::locale:global*
30.0.1	`<cstdio>`, `<cwchar>`	alle

Die nächste Tabelle zeigt, dass die C++-Bibliotheken meist nicht viel mehr Speicher benötigen als die C-Bibliotheken. Eine Ausnahme sind aber die für *std::cout* usw. über <*iostream*> verfügbaren Stream-Klassen.

In Regel **30.0.1** der **MISRA** C++:2023 Guidelines wird verlangt, dass man die I/O-Funktionen der C-Bibliothek (die Funktionen und Makros aus **<cstdio>** sowie die entsprechenden Funktionen aus <cwchar>) **nicht verwenden** soll. Dabei wird ausdrücklich darauf hingewiesen, dass dadurch die Verwendung der Funktionen, Klassen usw. aus <fstream> nicht verboten wird, obwohl diese Funktionen aus <cstdio> verwenden.

2.11 Die MISRA Regeln zu dynamischem Speicher

Nach den MISRA C++:2023 Guidelines (Regel 21.6.2) dürfen *new*, *delete*, *malloc* und *free* (sowie auch *calloc*, *realloc* und *aligned_alloc*) nicht verwendet werden. Nach den C++ Core Guidelines (R.10 und R.11) sollen diesen nicht explizit aufgerufen werden.

Die Formulierung „dürfen nicht“ bei MISRA „sollen nicht“ bei den Core Guidelines schließt auf den ersten Blick Klassen der Standardbibliothek (z.B. *std::vector*) aus, die dynamischen Speicher verwenden. Da MISRA in den Anmerkungen zu dieser Regel aber ausdrücklich darauf hinweist, dass anerkannt wird, dass es eben doch Anwendungen gibt, die dynamischen Speicher benötigen, wird eine Verletzung dieser Regel doch akzeptiert, wenn zusätzlich dokumentiert wird, wie damit verbundenen Probleme vermieden werden.

2.12 Compilezeit-Konstanten mit *constexpr*

Die Klasse *std::array* hat schon seit C++17 einen *constexpr*-Konstruktor:

```
constexpr std::array<int, 100> a = { 1,2 }; // das geht
```

Die Regel **6.7.2** der **MISRA** C++:2023 Guidelines verlangt, dass globale Variablen nur mit *const* oder *constexpr* definiert werden. In Con.4 und Con.5 der C++ Core Guidelines wird das empfohlen.

Bei einer Codeanalyse mit dem Visual Studio C++ Core Guidelines Checker (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**) wird bei einer Konstanten empfohlen, diese mit *constexpr* zu kennzeichnen:

```
const int y = 1; // warning C26814: Die const-Variable "y" kann zur Kompilierzeit berechnet
                    werden. Erwägen Sie die Verwendung von constexpr (con.5).
```

2.13 Explizite Konversionen

Oft kann man eine Konversion eines Werts t des Typs T in einen Wert u des Typs U mit einer expliziten Konversion erzwingen. In C sind solche Konversionen in der **Typecast-Schreibweise** und in der **Funktionsschreibweise** möglich:

```
t = (T)u; // explizite Konversion in der typecast-Schreibweise
t = T(u); // explizite Konversion in der Funktionsschreibweise
```

Diese typecast- und Funktionsschreibweise von C soll man allerdings in C++ vermeiden, da sie auch Konversionen ermöglichen, die zu Fehlern führen können und außerdem leicht übersehen werden. In den MISRA C++:2023 Guidelines (Regel 8.2.2) und den C++ Core Guidelines (ES.48 und ES.49) wird verlangt, dass diese beiden Schreibweisen nicht verwendet werden.

Die MISRA C++:2023 Guidelines verlangen, dass *const_cast* mit Zeigern und Referenzen nicht verwendet wird (Regel 8.2.3), und *reinterpret_cast* generell vermieden wird (Regel 8.2.5). Die C++ Core Guidelines raten in ES.50 von *const_cast* ab.

2.14 Attribute

Mit dem Attribut ***[[nodiscard]]*** bei der Definition einer Funktion oder eines Datentyps gibt der Compiler eine Warnung aus, wenn die Funktion bzw. eine Funktion, die einen Rückgabetyp des Typs hat, nicht verwendet.

Beispiel: Mit den Definitionen

```
struct[[nodiscard]] error_info{};
error_info starteMaschine() { return error_info(); };

[[nodiscard]] int plus1(int n) { return n + 1; }
int plus2(int n) { return n + 1; }
```

erhält man beim Aufruf der Funktionen die als Kommentar angegebenen Warnungen:

```
starteMaschine(); // Warnung C4834 Der Rückgabewert einer
plus1(17);        // Funktion mit dem "nodiscard" - Attribut wird verworfen
plus2(17);        // keine Warnung
```

Mit C++20 wurde ***[[nodiscard]]*** bei etwa 2500 Funktionen der STL aufgenommen, damit eine Warnung ausdrücklich darauf hinweist, dass die Nichtverwendung des Rückgabewertes sehr wahrscheinlich ein Fehler ist.

Mit dem Attribut ***[[maybe_unused]]*** bei einer Variable-Definition kann man dem Compiler zu verstehen geben, dass eine fehlende Verwendung der Variablen beabsichtigt ist und der Compiler deswegen keine Warnung ausgeben soll:

Beispiel:

```
void f5()
{
int x; // Warnung C4101 "x": Unreferenzierte lokale Variable
 [[maybe_unused]] int y; // keine Warnung
}
```

Die Regeln 0.2.1, 0.2.2, 0.2.3 und 0.2.4 der MISRA C++:2023 Guidelines verlangen, dass Variablen, Parameter, Datentypen, Funktionen usw. verwendet werden, wenn sich nicht mit *[[maybe_unused]]* gekennzeichnet sind.

3 MISRA C++:2023 enforces Modern C++

Many MISRA C++:2023 rules require explicitly to use modern C++ features instead of corresponding C features.

3.1 Use *std::array* instead of C-Arrays

The MISRA C++:2023 Guidelines (Rule 11.3.1) and the C++ Core Guidelines (ES.27) recommend not to use C arrays. MISRA recommends *std::array* as an alternative. The C++ Core Guidelines (SL.con.1) recommend *std::vector* in addition to *std::array*.

```
#include <array>
#include <vector>
#include <cstdint>

void arrays()
{
  const int32_t Max = 100;
  int32_t a1[Max] = { 0 }; // MISRACPP2023 - 11_3_1.a: The 'a1' array should not be used
                           // Misra C++2008: no non-compliance warning

  std::array<int32_t, Max> a2{ 0 }; // MISRA compliant
  std::vector<int32_t> v;           // MISRA compliant
```

3.2 Use strongly typed enums

The MISRA C++:2023 Guidelines (Rule 10.2.2) and the C++ Core Guidelines (Enum.3) recommend to use non-type-safe enumeration types (i.e. only *enum class*).

3.3 Use nullptr instead of 0 or NULL

nullptr is recommended in the C++ Core Guidelines (ES.47) and required in the MISRA C++:2023 Guidelines (Rule 7.11.1). *nullptr* can only be implicitly converted to a pointer, but not to any other type (including *int*).

4 Object-oriented programming

4.1 Classes

4.1.1 Datenelemente und Elementfunktionen

Rule 15.1.4 of the MISRA C++:2023 Guidelines and C.48 of the C++ Core Guidelines explicitly recommend default member initializers, as they initialize the data element before it can be used. They are particularly useful when a data element is to be given the same value in all constructors.

```
class DigitalOut
{
  uint32_t pin = 17U; // MISRA C++ 2023, 15.1.4: Default initializers should be preferred
  uint32_t id;
public:
  DigitalOut() :pin( 18U ), id{ 19U }
  {
  }
};
```

4.1.2 ***private*** **and** ***public*** data members

Rule **14.1.1** of the MISRA C++:2023 Guidelines recommends that all non-static data elements should be either **all *private*** or **all *public***. It is expressly stated that this rule is meant to be strict and that ***protected*** data elements are **excluded** by this either/or rule. The C++ Core Guidelines C.134 agree with this and also advise against *protected* data elements in C.133.

```
class C
{ // MISRACPP2023 - 14_1_1.a: Class 'C' should not mix public fields with private fields
  int32_t i = 0;
public:
  int32_t j = 0; // MISRA2008.11_0_1: Member variable 'j' should be declared as private

protected:
  int32_t k = 0; // MISRA2008.11_0_1: Member variable 'k' should be declared as private

};

class D // all data members public is MISRA compliant
{
public:
  int32_t i = 0;
  int32_t j = 0; // MISRA2008.11_0_1: Member variable 'j' should be declared as private
};

class C
{
protected:
  int32_t prot; // MISRACPP2023 - 14_1_1.a: Class 'C' should not contain protected field
'prot'
};
```

4.1.3 Constructors and Destructors

Before C++11, the arguments for the constructor had to be specified in round brackets, as in the examples under 1. Since C++11, they can also be passed in an initialization list:

```
DigitalOut LED_1(3.14); // MISRA2008.5_0_5_a: Implicit conversion from 'double' type to
'int' type in function's argument
DigitalOut LED_2({ 3.14 }); // compiler error: narrowing conversion
DigitalOut LED_3{ 3.14 };   // compiler error: narrowing conversion
```

Rule 11.6.1 of the MISRA C++:2023 Guidelines and ES.23 of the C++ Core Guidelines explicitly recommend using initialisation lists (curly brackets {}) rather than round brackets for initialisation.

Occasionally, constructors are required without specific instructions. Prior to C++11, these were formulated with an empty instruction section. Since C++11, *=default* can also be used. Rule 15.0.1 of the MISRA C++:2023 Guidelines explicitly recommends that a destructor either has a non-empty instruction section or is defined with *=default*.

```
class C {
public:
  ~C() = default; // nicht ~C{}
};

class D
{
public:
  virtual ~D() {}
  // MISRACPP2023 - 15_0_1.b: Do not define the '~D' destructor with an empty body
}; // MISRACPP2023 - 15_0_1.a: Class 'D' shall have a customized copy assignment operator
to be a copy-assignable general manager
           // MISRACPP2023 - 15_0_1.a: Class 'D' shall have a customized copy constructor
        to be a general manager

// VS CGL-Checker: C26432: If you define or delete any default operation in the type
'class N_Classes::N_empty_destructor::D', define or delete them all (c.21
```

4.1.4 *shared_ptr*: Smart Pointer für Klassenelemente und RAII

Da der Destruktor einer Klasse automatisch die Destruktoren aller Datenelemente aufruft (siehe Seite **Fehler! Textmarke nicht definiert.**), kann man Klassen so konstruieren, dass Objekte dieser Klassen am Ende ihrer Lebenszeit automatisch alle Ressourcen freigeben. Das erreicht man dadurch, dass man im Destruktor einer Klasse alle Ressourcen freigibt.

Es wird generell empfohlen, alle Klassen so zu implementieren. Diese Technik wird auch als **RAII** („resource acquisition is initialization“ - „Ressourcenbelegung ist Initialisierung“ bezeichnet und im Kapitel „Exception Handling“ im „Embedded C++ Aufbaukurs“ noch genauer beschrieben.

Für Zeiger auf dynamischen Speicher ist RAII in den smart Pointer Klassen der Standardbibliothek implementiert. Diese geben den Speicher im Destruktor wieder frei und verhindern so, dass die Freigabe des Speichers vergessen wird. Eine ausführliche Beschreibung folgt in Kapitel „Smart Pointer“ im Embedded C++ Aufbaukurs. Hier soll ein kurzer Überblick die wichtigsten Vorteile von smart pointern des Typs *shared_ptr* aufzeigen. Dieser Datentyp steht nach

```
#include <memory> // oder import std.memory; (siehe Abschnitt Fehler! Verweisquelle konnte nicht
gefunden werden.)
using namespace std; // bzw. using std::shared_ptr;
```

zur Verfügung. Dann erhält man mit

```
shared_ptr<int> s = std::make_shared<int>(17);//int* s=new int(17);
shared_ptr<T> t = std::make_shared<T>(); // T* t = new T(); T ein beliebiger Datentyp mit
                einem Standardkonstruktor
```

die smart pointer s und t, die man im Wesentlichen wie die im Kommentar angegebenen Zeiger verwenden kann. Der entscheidende Vorteil eines smart pointers ist aber, dass der zugehörige Speicher nicht explizit mit *delete* freigegeben werden muss, sondern automatische beim Aufruf des Destruktors freigegeben wird.

Da der automatisch erzeugte Destruktor die Destruktoren aller Datenelemente der Klasse aufruft (siehe Abschnitt 4.1.3, 6.), wird auch der Destruktor eines smart pointer Datenelements aufgerufen. Deshalb muss man für eine Klasse, die nur smart pointer und keine gewöhnlichen Zeiger hat, keinen Destruktor definieren.

Beispiel: Wenn eine Klasse einen Zeiger auf einen dynamischen Speicherbereich enthält, muss man in der Klasse einen Destruktor definieren, der diesen Speicher wieder freigibt:

```
class C
{
  int* p;
public:
  C(int x): p(new int(x)) { }
  ~C() // der Destruktor ist notwendig, da die Klasse einen
  {   delete p; }  // Zeiger auf dynamischen Speicher enthält
};
```

Enthält die Klasse anstelle eines Zeigers dagegen einen smart pointer, muss kein Destruktor definiert werden, da ein smart pointer eine Klasse mit einem Destruktor ist, der den Speicherbereich frei gibt:

```
class D
{
  std::shared_ptr<int> u = std::make_shared<int>();
public:
  D(int x): u(std::make_shared<int>(x)) { }
}; // kein Destruktor notwendig
```

Die Freigabe des Speichers könnte man auch mit einem *unique_ptr* erreichen. Da für *unique_ptr* aber keine Zuweisungen möglich sind, könnte man auch keine Variablen des Klassentyps kopieren. Da das aber meist notwendig ist, wählt man für Datenelemente einer Klasse meist *shared_ptr*.

Bei vielen der folgenden Beispiele können Zeiger durch smart pointer ersetzt werden. Um aber auf die mit Zeigern verbundenen Probleme hinzuweisen, werden trotzdem oft gewöhnliche Zeiger verwendet.

4.2 Member initializers

Nach den MISRA C++:2023 Guidelines (Regel 15.1.4) sollen alle nicht statischen Datenelemente einer Klasse initialisiert werden, bevor man auf sie zugreifen kann. Da man auf die Elemente bereits im Konstruktor zugreifen kann, muss eine Initialisierung in einem Elementinitialisierer oder mit einem default member initializer (siehe Abschnitt 4.1.1) erfolgen.

4.3 Hidden member functions in a class hierarchie

The MISRA C++:2023 Guidelines (Rule 6.4.1) and the C++ Core Guidelines ES.12 require that hidden variables be avoided. This also applies to the names of data elements in a derived class. With MISRA Rule 6.4.2, this also applies to non-virtual element functions (see Section 10.4.2) that are defined in a derived class.

```
class Base
{
public:
  void f()
  {
  }
};

class D1 :public Base
{
public:
  void f() // MISRACPP2023 - 6_4_2.a: The function 'f' redefines functions from following
base classes: Base
  {
  }
```

```
};
```

4.4 Virtual Functions

MISRA C++:2023 enforces the C++11 syntax for virtual functions.

```
class Base
{
public:
  virtual void f()
  {
  }
};

class D1:public Base
{
public:
  virtual void f() // MISRACPP2023 - 13_3_1.a: The 'f' function should be declared with
                      the 'override' specifier
  {
  }
};

class D2 :public Base
{
public:
  void f() override // MISRACPP2023 compliant. MISRACPP2008 is confusing
  { // MISRA2008.10_3_2: Use the virtual keyword for 'f' function
  } // MISRA2008.10_3_1: Virtual function 'f' was already defined in class 'Base'
};
```

4.4.1 Calling virtual functions without dynamically created objects

Der direkte Aufruf einer virtuellen Funktion führt nur dann zum Aufruf der letzten überschreibenden Funktion, wenn dieser Aufruf über einen Zeiger oder eine Referenz erfolgt. Beim **Aufruf** einer virtuellen Funktion **in einer Elementfunktion** derselben Klasse wird die zugehörige letzte überschreibende Funktion aber auch dann aufgerufen, wenn die Elementfunktion über ein Objekt aufgerufen wird, das kein Zeiger oder keine Referenz ist. Der Grund dafür ist, dass der Aufruf einer Elementfunktion immer über den *this*-Zeiger erfolgt (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**), und deshalb jeder Aufruf einer Elementfunktion immer ein Aufruf über einen Zeiger ist.

Beispiel: Die Funktion g der Klasse C

```
struct C {
  void g() { f(); };
  virtual void f() {};
} c;

E e; // E: some class derived from C
```

wird vom Compiler folgendermaßen übersetzt:

```
void g(C* this)
{
  this->f();
};
```

Beim Aufruf einer solchen Funktion wird dann der Zeiger auf das aktuelle Objekt als Argument für den *this*-Parameter übergeben:

```
c.g(); // C::f(&c)
e.g(); // E::f(&e)
```

Obwohl f hier jedes Mal über ein Objekt (und nicht über einen Zeiger oder eine Referenz) aufgerufen wird, ruft diese beim ersten Aufruf eine andere Funktion auf als beim zweiten Aufruf.

Auf diese Weise kann man virtuelle Funktionen auch über Variablen aufrufen, die nicht mit *new* angelegt oder als Referenzparameter übergeben wurden.