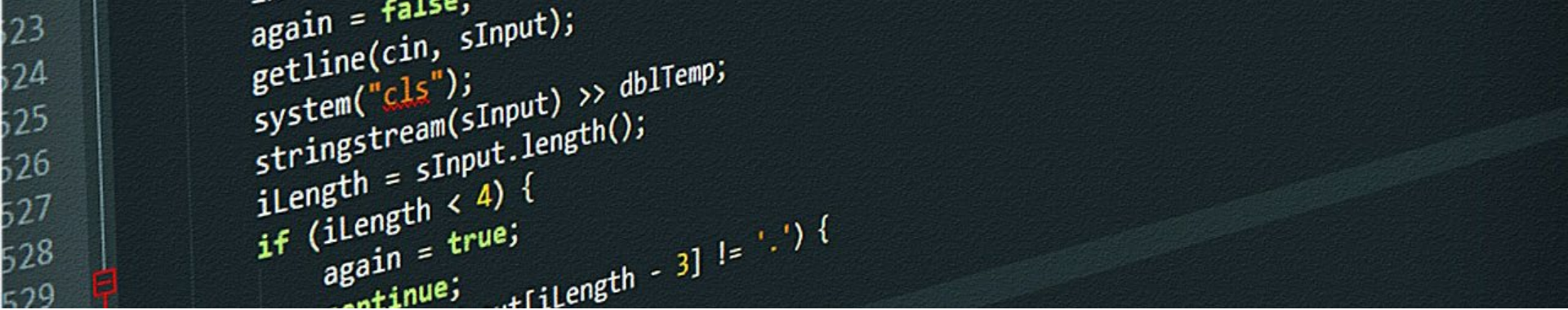


Thomas

C23-12.1 Programming style

Advanced algorithms and programming

v1

Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Coding standards and guidelines

General

- Coding standard provide sets of rules for using a language (here: C++) for a particular purpose in a particular environment
  - No single coding standard is appropriate for all uses and all users
- Many coding standards are a mixture of style guidance and firm rules
- Examples
  - Stroustrup & Sutter: C++ Core Guideline (see later in these slides)
  - Sutter & Alexandrescu: C++ Coding Standards: 101 Rules, Guidelines, and Best Practices ISBN: 978-0321113580
  - MISRA C++ rules (see later in these slides)
  - Google C++ Style Guide
  - Turner: C++ best practices (GitHub)

C++ core guidelines

Introduction

- C++ Core Guidelines: a portable set of guidelines, rules,and proven methods for programming in C++
  - Defined by Bjarne Stroustroup, Herb Sutter and others
  - C++ Core Guidelines (isocpp.github.io)

In	Introduction	A	Architectural ideas
P	Philosophy	NR	Non-Rules and myths
I	Interfaces	RF	References
F	Functions	Pro	Profiles
C	Classes and class hierarchies	GSL	Guidelines support library
Enum	Enumerations	NL	Naming and layout rules
R	Resource management	FAQ	Answers to frequently asked questions
ES	Expressions and statements		
Per	Performance		
CP	Concurrency and parallelism		
E	Error handling		
Con	Constants and immutability		
T	Templates and generic programming		
CPL	C-style programming		
SF	Source files		
SL	Standard Library		

C++ core guidelines

Examples – P: Philosophy

- P.5: Prefer compile-time checking to run-time checking
  - Fosters code clarity and performance (no error handlers required for errors caught at compile time)
  - Example

```
#include <iostream>
using namespace std;

// Int is an alias used for integers
using Int = int;

int main()
{
    // don't: run-time check, avoidable code
    int bits = 0;
    for (Int i = 1; i; i <<= 1)
        ++bits;
    if (bits < 32)
        cerr << "Int too small\n";


    // do: compile-time check
    static_assert(sizeof(Int) >= 4, "Int too small");

}
```

C++ core guidelines

Examples – P: Philosophy

- P.8: Don’t leak any resources
  - Even a slow growth in resources will, over time, exhaust the availability of those resources. This is particularly important for long-running programs and embedded systems programming
  - Example

```
// don't
void f(char* name)
{
    bool something = false;

    FILE* input = fopen(name, "r");
    // ...
    if (something) return;   // bad: if something == true,
                             // then a file handle is leaked
    // ...
    fclose(input);
}
```

```
// do
void f(char* name)
{
    bool something = false;

    ifstream input{ name };
    // ...
    if (something) return;   // OK: no leak
    // ...
}
```

C++ core guidelines

Examples – Enum: Enumerations

- Enum.1: Prefer enumerations over macros
  - Macros do not obey scope and type rules. Also, macro names are removed during preprocessing and so usually don’t appear in tools like debuggers.
  - Example

```
// don’t

// webcolors.h (third party header)
#define RED   0xFF0000
#define GREEN 0x00FF00
#define BLUE  0x0000FF

// productinfo.h
// The following define product subtypes based on color
#define RED    0
#define PURPLE 1
#define BLUE   2

int webby = BLUE;   // webby == 2; probably not what was desired
```

```
// do

enum class Web_color
{ red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };

enum class Product_info { red = 0, purple = 1, blue = 2 };

int webby = blue;   // error: be specific
Web_color webby = Web_color::blue;
```

C++ core guidelines

Examples – F: Functions

- F.4: If a function may have to be evaluated at compile time, declare it constexpr
  - constexpr is needed to tell the compiler to allow compile-time evaluation
  - constexpr does not guarantee compile-time evaluation; it just guarantees that the function can be evaluated at compile time for constant expression arguments if the programmer requires it or the compiler decides to do so to optimize
  - Example

```
constexpr int min(int x, int y) { return x < y ? x : y; }

void test(int v)
{
    int m1 = min(-1, 2);            // probably compile-time evaluation
    constexpr int m2 = min(-1, 2);  // compile-time evaluation
    int m3 = min(-1, v);            // run-time evaluation
    constexpr int m4 = min(-1, v);  // error: cannot evaluate at compile time
}
```

C++ core guidelines

Examples – C: Classes

- C.1: Organize related data into structures (struct or class)
  - Ease of comprehension. If data is related (for fundamental reasons), that fact should be reflected in code
  - Example

```
void draw(int x, int y, int x2, int y2);  // don't: unnecessary implicit relationships

void draw(Point from, Point to);          // do
```

C++ core guidelines

Examples – ES: Expressions and statements

- ES.23: Prefer the {}-initializer syntax
  - The rules for {} initialization are simpler, more general, less ambiguous, and safer than for other forms of initialization
  - Use = only when you are sure that there can be no narrowing conversions. For built-in arithmetic types, use = only with auto.
  - Avoid () initialization, which allows parsing ambiguities
  - {}-initializers do not allow narrowing conversions (and that is usually a good thing) and allow explicit constructors (which is fine, we're intentionally initializing a new variable)

- Example

```
int x{ 7.9 };   // error: narrowing
int y = 7.9;   // OK: y becomes 7. Hope for a compiler warning
int z = static_cast<int>(7.9);  // OK: you asked for it
```

C++ core guidelines

Examples – A: Architectural ideas

- A.1: Separate stable code from less stable code
  - Isolating less stable code facilitates its unit testing, interface improvement, refactoring, and eventual deprecation

- A.4: There should be no cycles among libraries
  - A library should not depend on another library that depends on the first one, because
    - Cycles complicate the build process
    - Cycles are hard to understand and may introduce indeterminism (unspecified behavior)

C++ core guidelines

Examples – SL: Standard library

- SL.1: Use libraries wherever possible
  - Safe time, don’t re-invent the wheel
  - Don’t replicate the work of others, and benefit from other people’s work when they make improvements

- SL.2: Prefer the standard library to other libraries
  - More people know the standard library
  - It is more likely to be stable, well-maintained, and widely available than your own code or most other libraries

- SL.3: Do not add non-standard entities to namespace std
  - Adding to std may change the meaning of otherwise standards-conforming code
  - Additions to std may clash with future versions of the standard

C++ core guidelines

Supporting tools

- **Tools: Clang-tidy**
  - Clang-tidy has a set of rules that specifically enforce the C++ Core Guidelines
  - These rules are named in the pattern cppcoreguidelines-*

- **Tools: CppCoreCheck**
  - The Microsoft compiler's C++ code analysis contains different selectable rulesets
  - One of these sets is aimed at enforcement of the C++ Core Guidelines

Coding guidelines for safety-critical applications

Overview

- **MISRA C++**
  - Guidelines for the Use of the C++ Language in Critical Systems
  - ISBN 978-906400-03-3 (paperback), ISBN 978-906400-04-0 (PDF), June 2008
  - MISRA C++:2008 - Guidelines for the use of the C++ language in critical systems (tlemp.com)

- **AUTOSAR C++14**
  - Guidelines for the use of the C++14 language in critical and safety-related systems (autosar.org)

- **Joint Strike Fighter Air Vehicle C++ Coding Standards**
  - JSF AV rules (stroustrup.com)

Coding guidelines for safety-critical applications

Examples – MISRA C++

- Rule 0-1-1: The project shall not contain unreachable code
- Rule 0-1-2: The project shall not contain infeasible paths
- Rule 0-1-3: The project shall not contain unused variables

- Rule 2-10-2: Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope

- Rule 3-1-3: When an array is declared, its size shall be either stated explicitly, or defined implicitly by initialization

```
              int32_t array1 [10];                    // compliant
extern        int32_t array2[];                       // non-compliant
              int32_t array3[] = { 0, 10, 15 };       // compliant
extern        int32_t array4 [42];                    // compliant
```

- Rule 5-2-2: A pointer to a virtual base class shall only be cast to a derived class by means of dynamic cast

Code documentation

General

- Document your code
  - For yourself, for your team members, for other teams, for successors
  - To keep an overview of the code, to understand code as quickly as possible

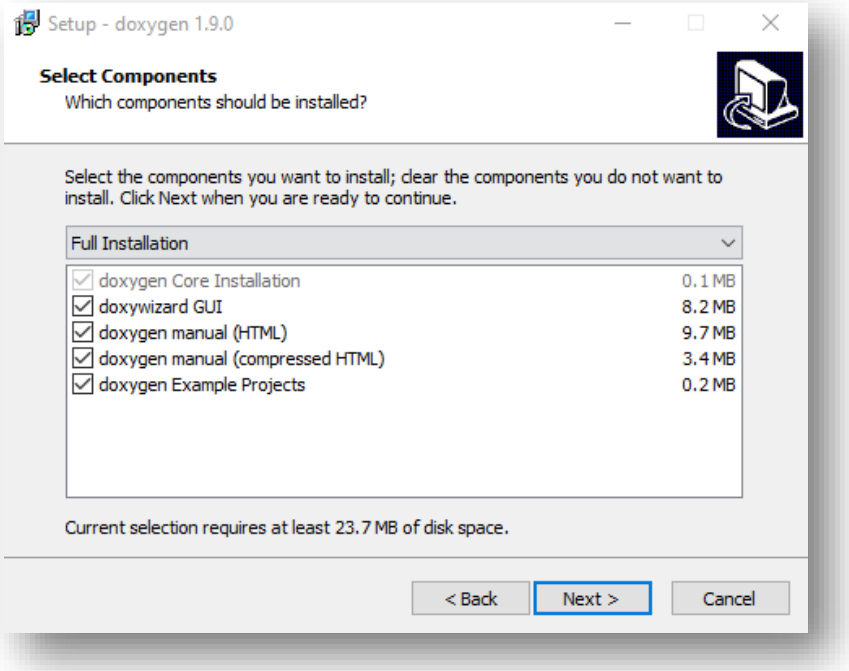
- Document the code inside the code, not in separate documents
  - To keep code and documentation consistent

Doxygen
General

- Documentation system for source code in
  - C++ / C / C#
  - Java
  - Fortran
  - VHDL
  - …
- Keeps documentation consistent with source code
- Useful for quickly finding your way in large source distributions
- Generates HTML, LaTeX, RTF, ManPages, XML…

Doxygen
Installation

- Download from [Doxygen: Downloads]
  - Available for Windows and Linux



Doxygen

Documentation sections

- JavaDoc style

```
/**
 * ... text ...
 */
```

- Qt style

```
/*!
 * ... text ...
 */
```

- C++ style

```
///
/// ... text ...
///
```

```
//!
//!... text ...
//!
```

Doxygen

Sections and keywords

- File section
  - \author, \date, \summary, \version
- Function section
  - \brief, \param, \return
  - \pre, \post, \invariant, \exception
- Member documentation
- General
  - \todo
  - Various formatting and linking keywords

Doxygen
Example

```
/// \brief This is an example file for documentation with Doxygen
/// \author Carsten Thomas
/// \date 03-Jan-2021

/// \brief Class encapsulating the Point functionality
class Point
{
public:
/// \brief Constructor of the Point class
Point()
: m_x{ 0 }, m_y{ 0 }
{}

/// \brief getter function returning the x position of the Point
/// \return x value
int getX() const { return m_x; }

private:
int m_x, m_y; ///< Member variables storing the coordinates of the Point
};

int main()
{

}
```

Doxygen
Alternatives

- Source code documentation with XML comments

  Supported, e.g., by Visual Studio C++

```
/// <summary>
///
/// </summary>
/// <returns></returns>
int main()
{

}
```

- Source code post-processing to generate HTML, e.g., by:
  - DocFX - Static documentation generator
  - Sandcastle Help File Builder (SHFB)

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

www.htw-berlin.de