MTRN2500 CPP Style Guide

- MTRN2500 CPP Style Guide
  - 0. Purpose
  - 1. Features to avoid
  - 2. Naming and Comments
    - Naming
    - Functions
    - Comments
      - #CS201 Make code tell the story
      - #CS202 Prefer readable code over comment
      - #CS203 Explain non trivial function
      - #CS204 File header block
  - 3. Coding Style
    - #CS300 Tools
    - Naming Conventions
      - #CS301 Pascal case for type names
      - #CS302 Camel case for variable names
      - #CS303 Non-static private class variables should have the "m" prefix.
      - #CS304: Static private class variables should have the "c" prefix.
      - #CS305: Static variables (non-class members) should have the "g" prefix.
      - #CS305M Global constant names (including enum values) must be all uppercase using underscore to separate words.
      - #CS306 Names representing methods or functions must be written in mixed case starting with lower case.
      - #CS307 Names representing template types should be a single uppercase letter.
      - #CS308 Generic variables should have the same name as their type.
      - #CS308 Variables with a large scope should have long names, variables with a small scope can have shorter names.
      - #CS309 The name of the object is implicit, and should be avoided in a method name.
    - Specific Naming Conventions
      - #CS311 The term set must be used where an attribute is changed directly.
      - #CS312M Methods that return a value should be named after the value they return, with get prefix.
      - #CS313 The `compute` prefix should be used in methods where something (complex) is computed.
      - #CS314 The `find` prefix should be used in methods where something is looked up.
      - #CS315 The `init` prefix should be used where an object or a concept is established and match with a corresponding `cleanup` prefix.

        - #CS316 Variables representing GUI components should be suffixed by the component type name.
        - #CS317 Plural form should be used on names representing a collection of objects.
        - #CS318 The `n` prefix should be used for variables representing a number of objects.
        - #CS319M Use normal variable name for range-based for loop, Iterator variables should be called i, j, k etc.
        - #CS320 The `is` prefix should be used for boolean variables and methods.
        - #CS321 Complement names must be used for complement operations.
        - #CS322 Negated boolean names must be avoided.
    - Files
        - #CS324 Use pascal case for source files
        - #CS325 A class should be declared in a header file and defined in a source file where the name of the files match the name of the class.
        - #CS326 Inline simple methods
        - #CS327 Inline performance critical methods
    - Include Files and Include Statements
        - #CS329 Header files must contain an include guard
        - #CS330 Sort `#include` statements in header blocks
        - #CS330M Only `#include` header file needed to declare a class in the class header file
        - #CS331 Don't use an #include when a forward declaration would suffice
    - Statements
        - #CS333 Types that are local to one file only can be declared inside that file.
        - #CS334 Use a struct only for passive objects that carry data; everything else is a class.
        - #CS335 The parts of a class must be sorted public, protected and private. All sections must be identified explicitly.
    - Inheritance
        - #CS337 Do not change the public/protected/private status of a method in derived classes.
        - #CS338 Use virtual in front of a derived function declaration if the base class function was declared virtual.
        - #CS339 Always declare destructors as virtual except in classes that are not meant to be derived.
        - #CS340 Declare a function virtual only if it is going to be used polymorphically.
        - #CS341 Use public inheritance exclusively.
    - Variables
        - #CS343B Variables should be initialized where they are declared.
        - #CS343M Variable should be declared where they are used
        - #CS344 Member variables should be initialized in the constructor initialization list provided it doesn't yield code duplication in other constructors.
        - #CS345 Place a function's variables in the narrowest scope possible (block, function, class), and initialize variables in the declaration.
        - #CS346 Do not use global variables, use singletons or static methods instead.
        - #CS347 Use the copy constructor for an object definition

    - #CS348 Class variables should never be declared public. Use getters and setters instead.
    - #CS349 C++ pointers and references should have their reference symbol next to the name rather than to the type.
    - #CS350 Implicit test for 0 should not be used other than for boolean and pointer variables.
    - #CS351 Do not test boolean expressions against true or false
- Loops
    - #CS353M Loop variables should be initialized immediately before the loop for while loop.
    - #CS354 The use of break and continue in loops should only be used if they give higher readability than their structured counterparts.
    - #CS355 The `while (true)` form should be used for infinite loops.
- Conditionals
    - #CS357 The if-else class of statements should have the following form:
    - #CS358 A for statement should have the following form:
    - #CS359 A while statement should have the following form:
    - #CS360M `if`, `while`, or `for` statement must be written with brackets.
    - #CS361 Assignment in conditionals must be avoided.
- Functions
    - #CS363 Write small and focused functions: the body of a function should not exceed one page (no scrolling).
    - #CS364 High-level and low-level code should not be mixed in a same function.
    - #CS365M When defining a function, parameters should be inputs only, For "out" output values, prefer return values to output parameters
    - #CS366 Prefer pre-incrementation over post-incrementation whenever possible.
- Enums
    - #CS367 Prefer implicit initialization.
    - #CS367B Prefer enum class over enums
        - Reason
- Miscellaneous
    - #CS368 Prefer `int` data type for integer numbers, including unsigned integer numbers.
    - #CS369 Prefer `double` data type for floating point numbers.
    - #CS370 Floating point constants should be written with decimal point and at least one decimal.
    - #CS371 Use 0 for integers, 0.0 for reals, nullptr for pointers, and '\0' for chars.
    - #CS372 Use C++ static_cast<>() instead of C style casts.
    - #CS373 Use C++ dynamic_cast<>() to test the run-time type of an object.
    - #CS374 Do never use the friend keyword (but for operator overloading).
    - #CS375 Avoid defining macros at all costs.
    - #CS375B Use `constexpr` variable or `enum` instead of `#define`
    - #CS376 Avoid operator overloading, except in these two cases:
    - #CS377 Use const whenever it makes sense to do so, e.g.,:
- White Space
    - #CS379 Special characters like Tab and page break must be avoided.

        - #CS380M Basic indentation should be 4 spaces.
        - #CS381 Conventional operators should be surrounded by a space character. Commas and semi-colons (in for loops) should be followed by a white space.
        - #CS382 Logical units within a block should be separated by one blank line.
        - #CS383 Use alignment wherever it enhances readability.
    - Comments
        - #CS385 Use // for all comments, including multi-line comments.
        - #CS386 Use /**/ only for temporary debugging purposes.
        - #CS387 Follow the standard indentation for a `switch` statement.
        - #CS388 Follow the standard indentation for a class declaration
        - #CS389 In a class declaration, order privacy levels, signals and slots, the following way:

- 3. Coding Style

    - #CS300 Tools
    - Naming Conventions
        - #CS301 Pascal case for type names
        - #CS302 Camel case for variable names
        - #CS303 Non-static private class variables should have the "m" prefix.
        - #CS304: Static private class variables should have the "c" prefix.
        - #CS305: Static variables (non-class members) should have the "g" prefix.
        - #CS305M Global constant names (including enum values) must be all uppercase using underscore to separate words.
        - #CS306 Names representing methods or functions must be written in mixed case starting with lower case.
        - #CS307 Names representing template types should be a single uppercase letter.
        - #CS308 Generic variables should have the same name as their type.
        - #CS308 Variables with a large scope should have long names, variables with a small scope can have shorter names.
        - #CS309 The name of the object is implicit, and should be avoided in a method name.
    - Specific Naming Conventions
        - #CS311 The term set must be used where an attribute is changed directly.
        - #CS312 Methods that return a value should be named after the value they return (without get prefix).
        - #CS313 The `compute` prefix should be used in methods where something (complex) is computed.
        - #CS314 The `find` prefix should be used in methods where something is looked up.
        - #CS315 The `init` prefix should be used where an object or a concept is established and match with a corresponding `cleanup` prefix.
        - #CS316 Variables representing GUI components should be suffixed by the component type name.
        - #CS317 Plural form should be used on names representing a collection of objects.
        - #CS318 The `n` prefix should be used for variables representing a number of objects.
        - #CS319M Use normal variable name for range-based for loop, Iterator variables should be called i, j, k etc.

        - #CS320 The `is` prefix should be used for boolean variables and methods.
        - #CS321 Complement names must be used for complement operations.
        - #CS322 Negated boolean names must be avoided.
    - Files
        - #CS324 Use pascal case for source files
        - #CS325 A class should be declared in a header file and defined in a source file where the name of the files match the name of the class.
        - #CS326 Inline simple methods
        - #CS327 Inline performance critical methods
    - Include Files and Include Statements
        - #CS329 Header files must contain an include guard
        - #CS330 Sort `#include` statements in header blocks
        - #CS330M Only `#include` header file needed to declare a class in the class header file
        - #CS331 Don't use an #include when a forward declaration would suffice
    - Statements
        - #CS333 Types that are local to one file only can be declared inside that file.
        - #CS334 Use a struct only for passive objects that carry data; everything else is a class.
        - #CS335 The parts of a class must be sorted public, protected and private. All sections must be identified explicitly.
    - Inheritance
        - #CS337 Do not change the public/protected/private status of a method in derived classes.
        - #CS338 Use virtual in front of a derived function declaration if the base class function was declared virtual.
        - #CS339 Always declare destructors as virtual except in classes that are not meant to be derived.
        - #CS340 Declare a function virtual only if it is going to be used polymorphically.
        - #CS341 Use public inheritance exclusively.
    - Variables
        - #CS343B Variables should be initialized where they are declared.
        - #CS343M Variable should be declared where they are used
        - #CS344 Member variables should be initialized in the constructor initialization list provided it doesn't yield code duplication in other constructors.
        - #CS345 Place a function's variables in the narrowest scope possible (block, function, class), and initialize variables in the declaration.
        - #CS346 Do not use global variables, use singletons or static methods instead.
        - #CS347 Use the copy constructor for an object definition
        - #CS348 Class variables should never be declared public. Use getters and setters instead.
        - #CS349 C++ pointers and references should have their reference symbol next to the name rather than to the type.
        - #CS350 Implicit test for 0 should not be used other than for boolean and pointer variables.
        - #CS351 Do not test boolean expressions against true or false

- Loops
  - #CS353M Loop variables should be initialized immediately before the loop for while loop.
  - #CS354 The use of break and continue in loops should only be used if they give higher readability than their structured counterparts.
  - #CS355 The `while (true)` form should be used for infinite loops.
- Conditionals
  - #CS357 The if-else class of statements should have the following form:
  - #CS358 A for statement should have the following form:
  - #CS359 A while statement should have the following form:
  - #CS360M `if`, `while`, or `for` statement must be written with brackets.
  - #CS361 Assignment in conditionals must be avoided.
- Functions
  - #CS363 Write small and focused functions: the body of a function should not exceed one page (no scrolling).
  - #CS364 High-level and low-level code should not be mixed in a same function.
  - #CS365M When defining a function, parameters should be inputs only, For "out" output values, prefer return values to output parameters
  - #CS366 Prefer pre-incrementation over post-incrementation whenever possible.
- Enums
  - #CS367 Prefer implicit initialization.
  - #CS367B Prefer enum class over enums
    - Reason
- Miscellaneous
  - #CS368 Prefer `int` data type for integer numbers, including unsigned integer numbers.
  - #CS369 Prefer `double` data type for floating point numbers.
  - #CS370 Floating point constants should be written with decimal point and at least one decimal.
  - #CS371 Use 0 for integers, 0.0 for reals, nullptr for pointers, and '\0' for chars.
  - #CS372 Use C++ static_cast<>() instead of C style casts.
  - #CS373 Use C++ dynamic_cast<>() to test the run-time type of an object.
  - #CS374 Do never use the friend keyword (but for operator overloading).
  - #CS375 Avoid defining macros at all costs.
  - #CS375B Use `constexpr` variable or `enum` instead of `#define`
  - #CS376 Avoid operator overloading, except in these two cases:
  - #CS377 Use const whenever it makes sense to do so, e.g.,:
- White Space
  - #CS379 Special characters like Tab and page break must be avoided.
  - #CS380M Basic indentation should be 4 spaces.
  - #CS381 Conventional operators should be surrounded by a space character. Commas and semi-colons (in for loops) should be followed by a white space.
  - #CS382 Logical units within a block should be separated by one blank line.
  - #CS383 Use alignment wherever it enhances readability.
- Comments
  - #CS385 Use // for all comments, including multi-line comments.

- #CS386 Use /**/ only for temporary debugging purposes.
- #CS387 Follow the standard indentation for a `switch` statement.
- #CS388 Follow the standard indentation for a class declaration
- #CS389 In a class declaration, order privacy levels, signals and slots, the following way:

0. Purpose

The purpose of this style guide is twofold: to ensure uniformity in code style and to prohibit dangerous features.

Adopting a uniform code style for a project increases the readability of the code, making it easier to understand and debug. C and C++ have great flexibility in terms of formatting and variable names. The best formatting style is a personal preference that has no single correct answer. The most important key for a C++ project is to pick one style and stick to it. C++ projects will typically formalise their choice in a style guide (EG: Chromium C++ style guide^chromium, Joint Strike Fighter Air Vehicle C++ coding standards^jsf, ros2 code style^ros, and Webots C++ coding style^webots). New contributors should follow the existing style in a project. Since this course's assessment will be based on Webots platform, we will adopt the Webots C++ coding style with some modification.

C++ is a very powerful low-level language that is designed for performance, as a result it contains many features that must be used very carefully. Some C++ features are not recommended due to safety concerns, deprecated, or have better alternatives. Others should only be used after careful assessment. See C++ Core Guidelines^core for a list compiled by domain experts. This course will further prohibit some C features to encourage use of C++ alternatives.

One thing to keep in mind is that a programming style guide is a general guideline, recommendation that should be followed. There may be a use case that justify breaking the guideline, but that should only be done after careful consideration and documented in the code.

1. Features to avoid

For this course, the following language features should not be used because either better alternatives exist in C++ or they are generally considered dangerous.

1. `goto`: makes control-flow difficult to analyse.
2. C style array `array[]` except when interfacing with library code: C style arrays do not contain array size, making it easier to accidentally go out of bounds. Use `std::array` instead.
3. Global variables (except `constexpr`): can cause hard to find bug. Constants are allowed, but use `constexpr` when possible. See CppCoreGuidelines I.2: Avoid non-const global variables.
4. C macro: Macros are simple text replacements that can cause unexpected results. Use `constexpr` variables for magic constants instead.
5. The macro `NULL` or `0` to indicate null pointer: it is too easy to confuse null pointer with 0, in C++ `nullptr` is introduced to mean null pointer exactly.
6. C style Cast: use c++ static_cast<> instead for most use-case, on rare occasion reinterpret_cast<> is required
7. Raw `new`/`delete`: dangerous, use smart pointer `std::unique_ptr` instead, occasionally `std::shared_ptr<>`.

8. `using namespace std` in the global scope: It is also discouraged to do so in general case. The standard library namespace contain vast amount of types that a programmer may not even by aware of and could accidentally lead to name collision. See CppCoreGuidelines SF.6: Use using namespace directives for transition, for foundation libraries...
9. `using namespace` in global scope in a header. Doing so may cause unpredictable name collisions with user's code. cppCoreGuidelines SF.7: Don't write using namespace at global scope in a header file
10. Double underscore __ or leading underscore in names. Most of them are reserved by the standard library.

2. Naming and Comments

Naming

Picking good names will greatly increase the readability of code. Read Chapter 4 of Clean C++ by Stephan Roth (available via UNSW Library website), it contains a very good in-depth discussion on how to chose good names. You should read that chapter.
Here are some useful tips from that chapter:

> 1. Names should be self-explanatory, e.g. `std::vector<Customer>` customers is more descriptive than `std::vector<Customer>` list.
> 2. Names should be long enough to be descriptive but not excessively long as to make the name unutterable.
> 3. Use names from the domain the code is intended for, so that nontechnical stakeholder with domain knowledge can understand it.
> 4. Avoid redundancy when choosing a name, don't repeat the class's name in its attributes or a variable type in its name.
> 5. Avoid cryptic abbreviations, full full words instead.
> 6. Avoid using the same name for different purposes.
> 7. Avoid surprising names. Variable name should be accurate and match the expectation of an ordinary developer would have.
> 8. Variables represent things, so should be named with nouns.
> 9. Functions represent actions, so should be named with verbs.

Functions

Functions can be used to organise code and reduce code repetition. Well written functions will increase readability and make debugging the program easier.

1. Use functions liberally, don't be afraid to refactor code into separate functions.
2. Keep function short and dedicated. A function should have a very precise defined task, it should only do one thing and do it well.
3. Function name should describe the one and only thing the function do.
4. Optimal number of parameters for a function is 1 (including hidden this pointer), up to 3 is acceptable, more is sign of bad design.

Comments

#CS201 Make code tell the story

Code can be understood by both the programmer and the compiler, whereas comments are only good for the programmer. Good code should be easily to read and understand.

A couple things you can do are:

1. Use descriptive name. Make your code read naturally in english. See naming section.
2. Use function to give a section of code a description.
3. Keep code simple. Don't try to be tricky with your code.

#CS202 Prefer readable code over comment

Comments often get out of date as code changes, therefore code is more reliable than comments, when possible, the story should be told by the code. Comment should be used to complement the code with additional details that you cannot state directly in code. Don't say in comment what is obvious in the code.

Example of good use of commands:

1. Stating intent (what is this code supposed to do).
2. Strategy (explain the algorithm implemented).
3. Stating invariants, pre- and post conditions, assumptions made in the code.
4. Consideration and decision you have made about the code.
5. Short explanation of a complex section of code.

#CS203 Explain non trivial function

Add a comment about each non trivial function declaration explaining the purpose of the function and how to use it.

```
/// \brief Implement sending marker from one object to display topic.
/// \param[in] parameter_name description. // for any input parameter
/// \param[out] parameter_name description. // for any output parameter
/// \return description of return value
```

#CS204 File header block

For additional explanation of what make a good comment, please read Clean C++ by Stephan Roth Chapter 4 section: comments.

```
// File:        FileName.cpp
// Description: Short explanation
// Author:      Author Name
// zID:         z1234567
// Date:        XX/XX/2020
```

3. Coding Style

This section will detail formatting style chosen for this course based on Webots CPP coding style.

#CS300 Tools

clang format is a widely used tool to automatically format code. It is integrated into many IDE including Visual studio, Atom text editor, and Visual Studio Code. A configuration file ".clang-format" will be provided. Using a common tool helps make sure all code are formatted consistently.

Most of the setting is explained below in the Webot style guide. The only additional setting is to keep maximum column to 128.

Naming Conventions

#CS301 Pascal case for type names

Names representing types must be in `PascalCase` notation: mixed case starting with upper case.

```
Line, SavingsAccount
```

#CS302 Camel case for variable names

C++ Variable names must be in `camelCase`: mixed case starting with lower case.

```
int line;
SavingsAccount savingsAccount;
```

#CS303 Non-static private class variables should have the "m" prefix.

```
private:
    int mLineSize;
    SavingsAccount mSavingsAccount;
```

#CS304: Static private class variables should have the "c" prefix.

```
private:
    static int cLineSize;
    static SavingsAccount cSavingsAccount;
```

#CS305: Static variables (non-class members) should have the "g" prefix.

```
static int gLine;
```

#CS305M Global constant names (including enum values) must be all uppercase using underscore to separate words.

```
static constexpr int MAX_ITERATIONS = 10;
enum { RED, GREEN, BLUE };
```

#CS306 Names representing methods or functions must be written in mixed case starting with lower case.

```
QString name();
double computeTotalWidth();
```

#CS307 Names representing template types should be a single uppercase letter.

```
template<class T> ...
template<class C, class D> ...
```

#CS308 Generic variables should have the same name as their type.

```
void setTopic(Topic *topic)
// NOT: void setTopic(Topic *value)
// NOT: void setTopic(Topic *aTopic)
// NOT: void setTopic(Topic *t)

void connect(Database *database)
// NOT: void connect(Database *db)
// NOT: void connect (Database *oracleDB)
```

#CS308 Variables with a large scope should have long names, variables with a small scope can have shorter names.

#CS309 The name of the object is implicit, and should be avoided in a method name.

```
line.length(); // NOT: line.lineLength();
```

Specific Naming Conventions

#CS311 The term set must be used where an attribute is changed directly.

```
employee.setName(name);
matrix.setElement(2, 4, value);
```

#CS312M Methods that return a value should be named after the value they return, with get prefix.

```
employee.getName();
matrix.getElement(2, 4);
```

#CS313 The `compute` prefix should be used in methods where something (complex) is computed.

```
valueSet->computeAverage();
matrix->computeInverse();
```

#CS314 The `find` prefix should be used in methods where something is looked up.

```
vertex.findNearestVertex();
matrix.findMinElement();
```

#CS315 The `init` prefix should be used where an object or a concept is established and match with a corresponding `cleanup` prefix.

```
printer.initFontSet();
:
printer.cleanupFontSet();
```

#CS316 Variables representing GUI components should be suffixed by the component type name.

```
mainWindow, propertiesDialog, widthScale, loginText, leftScrollbar, mainForm,
fileMenu, minLabel, exitButton, yesToggle etc.
```

#CS317 Plural form should be used on names representing a collection of objects.

```
vector<Point> points;
int values[];
```

#CS318 The `n` prefix should be used for variables representing a number of objects.

```
nPoints, nLines
```

#CS319M Use normal variable name for range-based for loop, Iterator variables should be called i, j, k etc.

Modification: Prefer range-based for loop: use regular variable naming as it is more natural.

```
for (auto &row : matrix) {
    for (auto &cell : row) {
        :
    }
}
```

When using iterator:

```
for (int i = 0; i < nTables); ++i) {
    :
}

for (auto i = matrix.begin(); i != matrix.end(); ++i) {
    Row row = *i;
    :
}
```

#CS320 The `is` prefix should be used for boolean variables and methods.

```
bool isSet() const { return mIsSet; }
bool isVisible() const;
bool isFinished() const;

private:
bool mIsFound;
bool mIsOpen;

// There are a few alternatives to the `is` prefix that fit better in some
situations. These are the `has`, `can` and `should` prefixes:
bool hasLicense();
bool canEvaluate();
bool shouldSort();
```

#CS321 Complement names must be used for complement operations.

```
init/cleanup, add/remove, create/destroy, start/stop, insert/delete,
increment/decrement, old/new, begin/end, first/last, up/down, min/max,
next/previous, old/new, open/close, show/hide, suspend/resume, etc.
```

#CS322 Negated boolean names must be avoided.

```
bool hasErrors;   // NOT: hasNoError
bool isFound;     // NOT: isNotFound
bool isRunning()  // NOT: isNotRunning()
```

Files

#CS324 Use pascal case for source files

C++ header files should have the `.hpp` extension. C++ source files should have the `.cpp` extension.

```
MyClass.cpp, MyClass.hpp
```

#CS325 A class should be declared in a header file and defined in a source file where the name of the files match the name of the class.

```
MyClass.hpp:

class MyClass {
   :
```

#CS326 Inline simple methods

Inlining should be used for methods that can be written in one line or so (typically getters and setters), if no additional include is required.

```
class MyClass {
    void setValue(Value value)  { mValue = value; }
    Value value() const         { return mValue; }
```

#CS327 Inline performance critical methods

For optimization reasons, longer methods can also be inlined, in this case the body of the method must be placed in the header file, just below the class declaration. Remember to avoid early optimization.

Include Files and Include Statements

#CS329 Header files must contain an include guard

Include guard format should use file name (note #CS325 file name = class name).

```
#ifndef CLASS_NAME_HPP
#define CLASS_NAME_HPP
    :
#endif
```

#CS330 Sort `#include` statements in header blocks

`#include` statements in a header or a source file should respect the following order, separated by an empty line:

- the most meaningful header comes first, followed by an empty line; usually we include "MyClass.hpp" at the top of MyClass.cpp text
- Webots headers come second in alphabetical order
- Qt headers in alphabetical order
- Ogre headers in alphabetical order
- ODE headers in alphabetical order
- standard headers come last in alphabetical order

```
%From  WbTriangleMesh.cpp

#include "WbTriangleMesh.hpp"

#include "WbBox.hpp"
#include "WbMFInt.hpp"
#include "WbMFVector2.hpp"
#include "WbMFVector3.hpp"
#include "WbRay.hpp"
#include "WbTesselator.hpp"

#include <cassert>
#include <limits>
```

#CS330M Only `#include` header file needed to declare a class in the class header file

Modification of Webots style guide to clarify: `#include` needed to implement the class belong in the cpp file instead.

#CS331 Don't use an #include when a forward declaration would suffice

```
#include "MyOtherClass.hpp"  // NO

class MyOtherClass;  // YES
```

Statements

#CS333 Types that are local to one file only can be declared inside that file.

#CS334 Use a struct only for passive objects that carry data; everything else is a class.

#CS335 The parts of a class must be sorted public, protected and private. All sections must be identified explicitly.

```
class MyClass {
public:
    ...
protected:
    ...
private:
    ...
}
```

Inheritance

#CS337 Do not change the public/protected/private status of a method in derived classes.

#CS338 Use virtual in front of a derived function declaration if the base class function was declared virtual.

#CS339 Always declare destructors as virtual except in classes that are not meant to be derived.

#CS340 Declare a function virtual only if it is going to be used polymorphically.

#CS341 Use public inheritance exclusively.

```
class DerivedClass : public BaseClass {
```

Variables

#CS343B Variables should be initialized where they are declared.

```
QString s {"abc"};

int x, y, z;
```

```
computeCenter(&x, &y, &z);

if (x > z) {
    int w = x * z;
    :
}
```

#CS343M Variable should be declared where they are used

Variable should be declared as they are used to avoid having to scroll up to find the variable declaration and help ensure variable is initialized.

```
int x {0};
use (x);

int y {1};
use (y);
```

#CS344 Member variables should be initialized in the constructor initialization list provided it doesn't yield code duplication in other constructors.

```
MyClass::MyClass() : mMySize(10), mMyObject1("obj1"), mMyObject2("obj2") {
    % non trivial initialization;
}
```

#CS345 Place a function's variables in the narrowest scope possible (block, function, class), and initialize variables in the declaration.

```
int i;
i = f();      // BAD - initialization separate from declaration.

int j = g();  // GOOD - declaration has initialization.
```

#CS346 Do not use global variables, use singletons or static methods instead.

#CS347 Use the copy constructor for an object definition

```
WbVector3 u(0.0, 1.0, 0.0);
WbVector3 v(u); // YES
WbVector3 w = u; // the very same thing as above, but NO
```

#CS348 Class variables should never be declared public. Use getters and setters instead.

```
// BAD!
class MyClass {
public:
    int numEmployee;
}

// GOOD
class MyClass {
public:
    void setNumberOfEmployees();
    int numberOfEmployees() const;

private:
    int mNumberOfEmployee;
}
```

#CS349 C++ pointers and references should have their reference symbol next to the name rather than to the type.

```
float *x, *y, *z;  // NOT: float* x, y, z;
int &y;            // NOT: int& y;
```

#CS350 Implicit test for 0 should not be used other than for boolean and pointer variables.

```
if (isFinished()) {
    :
}
```

Additional example

```
// NO
if (table.nRow()) {
    :
}
```

#CS351 Do not test boolean expressions against true or false

```
// YES
if (isFinished()) ...
if (!isOpen()) ...
```

```
// NO
if (isFinished() == true) ...
if (isOpen() == false) ...
```

Loops

#CS353M Loop variables should be initialized immediately before the loop for while loop.

```
bool isDone = false;
while (!isDone) {
    :
}
```

Modification: For for loop declare loop variable in the for loop initialization step to minimize the scope of the loop variable.

#CS354 The use of break and continue in loops should only be used if they give higher readability than their structured counterparts.

#CS355 The `while (true)` form should be used for infinite loops.

```
while (true) {
    :
}
```

Conditionals

#CS357 The if-else class of statements should have the following form:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
```

```
    statements;
}
```

#CS358 A for statement should have the following form:

```
for (initialization; condition; update) {
    statements;
}
```

#CS359 A while statement should have the following form:

```
while (condition) {
    statements;
}
```

#CS360M `if`, `while`, or `for` statement must be written with brackets.

Modification: While single `if`, `while`, or `for` statement may be written without brackets, it can be difficult to spot and can lead to incorrect code.

```
// BAD:
if (condition)
    for (initialization; condition; update)
        statement;

if (condition) statement;

// GOOD:
if (condition) {
    for (initialization; condition; update) {
        statement;
    }
}
```

Original Webot style guide #CS360: Single `if`, `while` or `for` statement must be written without brackets. The statement part should be put on a separate line. In case of nested block keep brackets.

#CS361 Assignment in conditionals must be avoided.

```
// GOOD
File *fileHandle = open(fileName, "w");
if (!fileHandle) {
```

```
    :
}

// BAD
if (!(fileHandle = open(fileName, "w"))) {
    :
}
```

Functions

#CS363 Write small and focused functions: the body of a function should not exceed one page (no scrolling).

#CS364 High-level and low-level code should not be mixed in a same function.

#CS365M When defining a function, parameters should be inputs only, For "out" output values, prefer return values to output parameters

Modification: This item is modified to follow CPP Core Guideline F.20[^f.20]. [^f.20]: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#f20-for-out-output-values-prefer-return-values-to-output-parameters

Reason: A return value is self-documenting, whereas a & could be either in-out or out-only and is liable to be misused. This includes large objects like standard containers that use implicit move operations for performance and to avoid explicit memory management.

If you have multiple values to return, use a named struct, std::tuple or similar.

Original: #CS365 When defining a function, parameter order is: inputs, then outputs. If a function returns only one value use the return type.

```
void computePoints(const Line &line, Point &p1, Point &p2);
Point computePoint(const Line &line1, const Line &line2);
```

#CS366 Prefer pre-incrementation over post-incrementation whenever possible.

```
vector<MyClass>::iterator i;
for (i = list.begin(); i != list.end(); ++i) { % NOT i++
    Element element = *i;
    :
}
```

Enums

#CS367 Prefer implicit initialization.

```
enum {
    A, // implicitly 0
    B, // implicitly 1
    C = 12,
    D // implicitly 12
}
```

#CS367B Prefer enum class over enums

Modification: This item is added to follow CPP Core Guideline Enum.3 [^enum.3] [^enum.3]: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#enum3-prefer-class-enums-over-plain-enums

Reason

To minimize surprises: traditional enums convert to int too readily. Enum class enable type checking.

```
enum class Color { red, green = 20, blue };
Color ballColour = Color::blue;
```

Miscellaneous

#CS368 Prefer `int` data type for integer numbers, including unsigned integer numbers.

#CS369 Prefer `double` data type for floating point numbers.

#CS370 Floating point constants should be written with decimal point and at least one decimal.

```
double total = 0.0; // NOT: double total = 0;
double speed = 3.0e8; // NOT: double speed = 3e8;

double sum = (a + b) * 10.0;
```

#CS371 Use 0 for integers, 0.0 for reals, nullptr for pointers, and '\0' for chars.

Modification: This item is modified to follow CPP Core Guideline ES.47 ^es.47 use nullptr rather than 0 or NULL. Reason: Readability. Minimize surprises: `nullptr` cannot be confused with an `int`. `nullptr` also has a well-specified (very restrictive) type, and thus works in more scenarios where type deduction might do the wrong thing on `NULL` or `0`.

```
if (p == nullptr && fabs(a) == 0.0 && size == 0) {
    char c = '\0';
}
```

#CS372 Use C++ static_cast<>() instead of C style casts.

```
A *a = (A*)b;  // NO
A *a = static_cast<A*>(b);  // YES
```

#CS373 Use C++ dynamic_cast<>() to test the run-time type of an object.

```
A *a = dynamic_cast<A*>(b);
if (a) {
    doSomethingWithA(a);
}
```

#CS374 Do never use the friend keyword (but for operator overloading).

#CS375 Avoid defining macros at all costs.

#CS375B Use `constexpr` variable or `enum` instead of `#define`

Modification: `constexpr` is typed variable that compiler will type check for safety whereas `#define` is preprocessor token substitution that may have unintended consequence. This follows CPP Core Guideline [^enum.1]
[^enum.1]:https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#enum1-prefer-enumerations-over-macros Reason: 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.

```
constexpr int MAX_ROW = 10; // YES
#define MAX_ROW 10 // NO

enum class Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF }; // Yes
#define RED 0xFF0000 // NO
```

#CS376 Avoid operator overloading, except in these two cases:

- To define mathematical classes, and only if the operators supports a known math syntax
- To inter-operate with streams

#CS377 Use const whenever it makes sense to do so, e.g.,:

- Declare const any method that does not change a variable of its class.
- Declare const any pointer or reference parameter that will not be altered by a function.
- Declare const any variable which value will not change.

```
double computeTotal(const double v[], int n) const;
const double size = 4.5;
```

White Space

#CS379 Special characters like Tab and page break must be avoided.

#CS380M Basic indentation should be 4 spaces.

#CS381 Conventional operators should be surrounded by a space character. Commas and semi-colons (in for loops) should be followed by a white space.

```
a = (b + c) * d; // NOT: a=(b+c)*d
doSomething(a, b, c, d); // NOT: doSomething(a,b,c,d);
for (i = 0; i < 10; ++i) { // NOT: for(i=0;i<10;i++){ ...
```

#CS382 Logical units within a block should be separated by one blank line.

```
Matrix4x4 matrix = new Matrix4x4();

double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

matrix.setElement(1, 1, cosAngle);
matrix.setElement(1, 2, sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2, cosAngle);

multiply(matrix);
```

#CS383 Use alignment wherever it enhances readability.

Comments

#CS385 Use // for all comments, including multi-line comments.

Notes: It is common to use /// to document the interface (API) for other users, and // for general comment.

#CS386 Use // only for temporary debugging purposes.**

#CS387 Follow the standard indentation for a `switch` statement.

```
switch(choice) {
case 0: printf("Good choice!\n"); break;
```

```
case 1: printf("Not optimal, you can do better\n"); break;
case 2: printf("Please give it a second thought\n"); break;
default: assert(false);
}
```

#CS388 Follow the standard indentation for a class declaration

```
class MyClass: public BaseClass {

public:
    MyClass();
    virtual ~MyClass();
    void setData(Data *data);

private:
    Data *mData;
}
```

#CS389 In a class declaration, order privacy levels, signals and slots, the following way:

```
class MyClass: public QObject {
    Q_OBJECT
public:
    MyClass();
    virtual ~MyClass();
    void setData(Data *data);

signals:
    dataChanged();

protected:
    formatData();

protected slots:
    updateData();

private:
    checkDataValidity();
    Data *mData;
    Picture *mPicture;

private slots:
    showPicture();
}
```