Reglas de codificación para lenguaje C++

J. Daniel Garcia (coordinador)
Arquitectura de Computadores
Departamento de Informática
Universidad Carlos III de Madrid

2024

1. Formato general del código

Como norma general, el código debe estar bien estructurado y organizado, así como apropiadamente documentado.

Para garantizar la legibilidad del código y un formato homogéneo, se formateará todo el código mediante **clang-format**. Se utilizará una configuración común como la que se describe a continuación:

```
AccessModifierOffset: −2
AlignAfterOpenBracket: Align
AlignArrayOfStructures: Right
AlignConsecutiveAssignments:
  Enabled: true
  AcrossEmptyLines: false
  AcrossComments: false
  AlignCompound: true
  PadOperators: true
AlignConsecutiveBitFields:
  Enabled: true
  AcrossEmptyLines: false
  AcrossComments: false
  AlignCompound: true
  PadOperators: true
AlignConsecutiveDeclarations: None
AlignConsecutiveMacros: None
AlignEscapedNewlines: Left
AlignOperands: Align
AlignTrailingComments:
  Kind: Always
AllowAllArgumentsOnNextLine: true
AllowAllConstructorInitializersOnNextLine: true
AllowAllParametersOfDeclarationOnNextLine: false
AllowShortBlocksOnASingleLine: Always
AllowShortCaseLabelsOnASingleLine: false
AllowShortEnumsOnASingleLine: true
AllowShortFunctionsOnASingleLine: Inline
AllowShortIfStatementsOnASingleLine: AllIfsAndElse
AllowShortLambdasOnASingleLine: None
AllowShortLoopsOnASingleLine: true
AlwaysBreakAfterDefinitionReturnType: None
AlwaysBreakAfterReturnType: None
AlwaysBreakBeforeMultilineStrings: false
AlwaysBreakTemplateDeclarations: Yes
BinPackArguments: true
BinPackParameters: true
BitFieldColonSpacing: Both
BraceWrapping:
  AfterCaseLabel: false
  AfterClass: false
```

```
    AfterControlStatement: MultiLine
    AfterEnum: false
    AfterFunction: false
    AfterNamespace: false
    AfterStruct: false
    AfterUnion: false
    AfterExternBlock: false
    BeforeCatch: true
    BeforeElse: true
    BeforeLambdaBody: false
    BeforeWhile: false
    IndentBraces: false
    SplitEmptyFunction: false
    SplitEmptyRecord: false
    SplitEmptyNamespace: false
BracedInitializerIndentWidth: 2
BreakAfterAttributes: Never
BreakBeforeBinaryOperators: None
BreakBeforeBraces: Attach
BreakBeforeConceptDeclarations: Always
BreakBeforeInlineASMColon: Always
BreakBeforeTernaryOperators: true
BreakConstructorInitializers: BeforeColon
BreakInheritanceList: AfterComma
BreakStringLiterals: true
ColumnLimit: 100
CompactNamespaces: false
ConstructorInitializerIndentWidth: 2
ContinuationIndentWidth: 4
Cpp11BracedListStyle: true
EmptyLineAfterAccessModifier: Never
EmptyLineBeforeAccessModifier: Always
FixNamespaceComments: true
IncludeBlocks: Regroup
IndentAccessModifiers: true
IndentCaseBlocks: true
IndentCaseLabels: true
IndentExternBlock: Indent
IndentGotoLabels: true
IndentPPDirectives: BeforeHash
IndentRequiresClause: true
IndentWidth: 2
IndentWrappedFunctionNames: true
InsertBraces: true
InsertNewlineAtEOF: false
KeepEmptyLinesAtTheStartOfBlocks: false
LambdaBodyIndentation: Signature
Language: Cpp
MaxEmptyLinesToKeep: 1
NamespaceIndentation: All
PPIndentWidth: 2
PackConstructorInitializers: BinPack
PointerAlignment: Middle
QualifierAlignment: Right
ReferenceAlignment: Middle
ReflowComments: true
RequiresClausePosition: OwnLine
RequiresExpressionIndentation: OuterScope
SeparateDefinitionBlocks: Always
ShortNamespaceLines: 0
SortIncludes: CaseInsensitive
SortUsingDeclarations: Never
SpaceAfterCStyleCast: true
SpaceAfterLogicalNot: false
SpaceAfterTemplateKeyword: true
SpaceAroundPointerQualifiers: Both
SpaceBeforeAssignmentOperators: true
SpaceBeforeCaseColon: false
SpaceBeforeCpp11BracedList: false
```

```
SpaceBeforeCtorInitializerColon: true
SpaceBeforeInheritanceColon: true
SpaceBeforeParens: ControlStatementsExceptControlMacros
SpaceBeforeParensOptions:
  AfterControlStatements: true
  AfterFunctionDeclarationName: false
  AfterFunctionDefinitionName: false
  AfterForeachMacros: false
  AfterIfMacros: false
  AfterOverloadedOperator: false
  BeforeNonEmptyParentheses: false
SpaceBeforeRangeBasedForLoopColon: true
SpaceBeforeSquareBrackets: false
SpaceInEmptyBlock: true
SpacesBeforeTrailingComments: 2
SpacesInAngles: Never
SpacesInLineCommentPrefix:
  Minimum: 1
  Maximum: 1
SpacesInParentheses: false
SpacesInSquareBrackets: false
Standard: c++20
UseTab: Never
```

Para dar formato al código fuente puede situar el archivo **.clang-format** en el directorio raíz de su proyecto. La mayoría de los entornos de desarrollo reconocerán el archivo y darán formato al código automáticamente.

También puede consultar otras formas de aplicar el formato en la documentación de la herramienta: https://clang.llvm.org/docs/ClangFormat.html.

2. Reglas de codificación

Como regla general, se recomienda tener en cuenta las **C++ Core Guidelines** (disponible en `http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`). No obstante, estas reglas no son de obligado cumplimiento.

La descripción detalladas de las reglas comprobadas por **clang-tidy** se puede encontrar en la página `https://clang.llvm.org/extra/clang-tidy/checks/list.html`.

Para comprobar automáticamente el cumplimiento de las reglas se puede usar una configuración como la siguiente:

```
Checks: '-*,
modernize-*,
bugprone-*,
cert-*,
cppcoreguidelines-*,
misc-*,
modernize-*,
performance-*,
readability-*,
-modernize-use-trailing-return-type'

CheckOptions:
  bugprone-assert-side-effect.AssertMacros: 'assert,Expects,Ensures'
  bugprone-misplaced-widening-cast.CheckImplicitCasts: true
  cppcoreguidelines-pro-type-const-cast.StrictMode: true
  misc-non-private-member-variables-in-classes.IgnoreClassesWithAllMemberVariablesBeingPublic: true
  modernize-deprecated-headers.CheckHeaderFile: true
  readability-function-cognitive-complexity.Threshold: 50
  readability-function-cognitive-complexity.Threshold.DescribeBasicIncrements: true
  readability-function-size.LineThreshold: 40
  readability-function-size.ParameterThreshold: 4

WarningsAsErrors: '*'

HeaderFileExtensions: [ 'h', 'hpp' ]
ImplementationFileExtensions: [ 'cpp']

FormatStyle: file
```

Para comprobar el cumplimiento de las reglas se puede situar el archivo **.clang-tidy** en el directorio raíz de su proyecto. La mayoría de los entornos de desarrollo reconocerán el archivo y realizarán las comprobaciones.

También puede consultar otras formas de aplicar el formato en la documentación de la herramienta: `https://clang.llvm.org/extra/clang-tidy/index.html`.

A continuación se especifican un conjunto de reglas que si deben cumplirse.

2.1. Reglas generales

En general se cumplirán los siguientes principios:

- No se podrán utilizar variables globales que no sean constantes.
- No se podrá pasar arrays primitivos (arrays tipo C) a una función como parámetros de tipo puntero.
- Ninguna función ni función miembro podrá tener más de **4** parámetros.
- Ninguna función podrá tener más de **40** líneas estando apropiadamente formateadas.
- Todos los parámetros se pasarán a las funciones por valor, por referencia o por referencia constante.
- No se podrá usar explícitamente las funciones **malloc()** o **free()** ni los operadores **new** o **delete**.

- No se permite el uso de macros, excepto para la definición de guardas de inclusión.
- No se permite ningún cast excepto **static_cast** o **dynamic_cast**.
- Solamente se podrá usar **reinterpret_cast** en el paso de parámetros a funciones de la biblioteca estándar (p.ej **read()** o **write**).

2.2. Reglas derivadas de las C++ Core Guidelines

Todas estas reglas se encuentran bajo la categoría **cppcoreguidelines**.
A continuación se resumen las principales reglas aplicables.

- **cppcoreguidelines-avoid-const-or-ref-data-members**: No se definirán datos miembro que sean **const** o de tipo referencia.
- **cppcoreguidelines-avoid-do-while**: No se utilizarán bucles del tipo **do-while**.
- **cppcoreguidelines-avoid-goto**: No se utilizará **goto** excepto para salir de bucles anidados a sentencias posteriores.
- **cppcoreguidelines-avoid-non-const-global-variables**: No se definirán variables globales a menos que sean constantes.
- **cppcoreguidelines-init-variables**: No se definirán variables locales sin darles un valor inicial.
- **cppcoreguidelines-interfaces-global-init**: No se realizarán iniciaciones globales que requieran acceso a objetos externos.
- **cppcoreguidelines-macro-usage**: No se utilizarán macros.
- **cppcoreguidelines-narrowing-conversions**: No se permitirán conversiones estrechadoras (*narrowing*) que potencialmente puedan causar pérdida de información.
- **cppcoreguidelines-no-malloc**: No se realizarán invocaciones a **malloc()**, **realloc()**, **calloc()** o **free()**.
- **cppcoreguidelines-prefer-member-initializer**: Se preferirá la iniciación en la secuencia de iniciación del constructor antes que la iniciación en el cuerpo del constructor.

2.2.1. Reglas específicas sobre comprobación de límites

- **cppcoreguidelines-pro-bounds-array-to-pointer-decay**: Se evitará la conversión implícita de un array en un puntero. Se puede utilizar **std:span<T>** o **gsl::span<T>** como alternativa.
- **cppcoreguidelines-pro-bounds-constant-array-index**: No se realizará un acceso a una posición de un array primitivo o un **std::array** que no sea una expresión conocida en tiempo de compilación. Como alternativa, se puede utilizar **gsl::at()**.
- **cppcoreguidelines-pro-bounds-pointer-arithmetic**: No se podrá utilizar aritmética de punteros.

2.2.2. Reglas específicas sobre seguridad de tipos

- **cppcoreguidelines-pro-type-const-cast**: No se podrá utilizar **const_cast<>**.
- **cppcoreguidelines-pro-type-cstyle-cast**: No se podrá utilizar ningún *cast* de tipo C.
- **cppcoreguidelines-pro-type-member-init**: Un constructor deberá dar valor inicial a todos los miembros que podrían quedar en un estado indefinido.
- **cppcoreguidelines-pro-type-reinterpret-cast**: No se podrá utiliza **reinterpret_cast<>**.
- **cppcoreguidelines-pro-type-static-cast-downcast**: No se podrá utilizar **static_cast<>** en los lugares donde un **dynamic_cast** sea más apropiado.
- **cppcoreguidelines-pro-type-union-access**: No se realizarán accesos a uniones.
- **cppcoreguidelines-pro-type-vararg**: No se utilizarán funciones con número variable de argumentos mediante **va_arg**.

2.3. Reglas de modernización

Todas estas reglas se encuentran bajo la categoría **modernize**.
EXCEPCIÓN: No se activarán la siguiente regla:

- **modernize-use-trailing-return-type**.

A continuación se resumen las principales reglas aplicables.

- **modernize-deprecated-headers**: Se requieren solamente archivos de cabecera de C++.
- **modernize-make-shared**: No se iniciará un **shared_ptr** a partir de **new**. Se sugiere el uso de **make_shared**.
- **modernize-make-unique**: No se iniciará un **unique_ptr** a partir de **new**. Se sugiere el uso de **make_unique**.
- **modernize-use-nullptr**: No se utilizará la macro **NULL**. En su lugar se usará **nullptr**.

2.4. Reglas sobre legibilidad

Todas estas reglas se encuentran bajo la categoría **readability**.
A continuación se resumen las principales reglas aplicables.

- **readability-avoid-const-params-in-decls**: No se permitirán parámetros de tipo **const**. Si se permitirán parámetros de tipo referencia constante.
- **readability-avoid-nested-conditional-operator**: No se permitirá el uso anidado del operador condicional (**?:**).
- **readability-braces-around-statements**: El cuerpo de sentencias **if**, **while** y **for** siempre estará entre llaves incluso aunque sea una única sentencia.
- **readability-const-return-type**: El tipo de retorno de una función no incluirá el calificador **const**.
- **readability-function-cognitive-complexity**: La complejidad cognitiva máxima será de **50**.

- **readability-function-size**: El número máximo de líneas por función será de **40**.
- **readability-function-size**: El número máximo de parámetros por función será de **4**.
- **readability-identifier-length**: La longitud mínima de los identificadores será de 3 caracteres. Se establece una excepción para los contadores de bucle con nombres **i**, **j** o **k**.
- **readability-magic-numbers**: No se utilizarán *números mágicos* en el código. En vez de esto se definirán constantes apropiadamente.
- **readability-make-member-function-const**: Si una función miembro no modifica el estado del objeto, se marcará como **const**.

2.5. Reglas para evitar errores comunes

Todas estas reglas se encuentran bajo la categoría **cppcoreguidelines**.
A continuación se resumen las principales reglas aplicables.

- **bugprone-assert-side-effect**: Se evitarán efectos laterales en evaluación de aserciones con **assert**. También se evitarán efectos laterales con las aserciones de **GSL Expects** y **Ensures**.
- **bugprone-assignment-in-if-condition**: No se realizarán asignaciones dentro de las condiciones de sentencias **if**.
- **bugprone-bool-pointer-implicit-conversion**: No se realizarán conversiones implícitas de **bool*** a **bool**.
- **bugprone-casting-through-void**: No se realizarán conversiones que involucren **void***.
- **bugprone-easily-swappable-parameters**: No se podrá pasar a una parámetros consecutivos del mismo tipo que pueda dar lugar a confusión.
- **bugprone-implicit-widening-of-multiplication-result**: Se deberá realizar un **static_cast<>** para realizar la conversión a un tipo de mayor tamaño. Si fuese necesario se incluirá el archivo de cabecera **<cstddef>**.
- **bugprone-misplaced-widening-cast**: Se evitarán las conversiones implícitas en las expresiones de cálculo.
- **bugprone-switch-missing-default-case**: Una sentencia **switch** siempre tendrá una alternativa **default**.

2.6. Reglas de CERT C++ Secure Coding

Todas estas reglas se encuentran bajo la categoría **cert**.
A continuación se resumen las principales reglas aplicables.

- **cert-err34-c**: No se utilizarán funciones de conversión de cadena a número que no verifiquen la validez de la conversión como **atoi()** o **sscanf()**.
- **cert-oop57-cpp**: No se utilizarán las funciones de la biblioteca estándar de C **memset()**, **memcpy()**, **memcmp()** y similares sobre tipos no triviales.

2.7. Reglas orientadas al rendimiento

Todas estas reglas se encuentran bajo la categoría **performance**.
A continuación se resumen las principales reglas aplicables.

- **performance-avoid-endl**: Se evitará el uso de **std::endl**. En su lugar se usará el carácter **'\n'**
- **performance-enum-size**: Siempre se especificará el tamaño de los tipos enumerados.
- **performance-no-int-to-ptr**: Se evitarán las conversiones de tipos integrales a puntero.
- **performance-type-promotion-in-math-fn**: Se evitará la promoción implícita de **float** a **double** en invocaciones a la biblioteca matemática de C (p. ej. **sin()**). En su lugar se invocará a la correspondiente función de la biblioteca matemática de C++ (p. ej. **std::sin()**.

2.8. Reglas varias

Todas estas reglas se encuentran bajo la categoría **misc**.
A continuación se resumen las principales reglas aplicables.

- **misc-confusable-identifiers**: Se evitará usar identificadores que sean muy similares y causen confusión.
- **misc-const-correctness**: Se definirán como **const** aquellas variables que no se modifican.
- **misc-definitions-in-headers**: No se definirán variables en archivos de cabecera.
- **misc-header-include-cycle**: Se evitarán dependencias cíclicas entre archivos de cabecera.
- **misc-include-cleaner**: Se evitarán la inclusión de archivos de cabecera innecesarios.
- **misc-non-private-member-variables-in-classes**: Se requiere que los datos miembros sean privados a menos que se trate de una estructura con todos los miembros públicos.
- **misc-redundant-expression**: Se evitará el uso de expresiones redundantes.
- **misc-static-assert**: Siempre que sea posible se preferirá **static_assert** sobre **assert**.
- **misc-throw-by-value-catch-by-reference**: Las excepciones se lanzarán por valor y se capturarán por referencia.

2.9. Anulación de reglas

En general, no está permitido la anulación de reglas en el desarrollo del proyecto.
No obstante, se establecen las siguientes excepciones:

- Se podrá anular la regla sobre el uso de **reinterpret_cast<>** en la definición de funciones de entrada/salida binaria.
- Se podrán anular las reglas sobre números mágicos en la definición de tests.

2.9.1. Funciones de entrada/salida binaria

A la hora de definir una función de entrada/salida binaria se hace necesario anular la regla que prohíbe el uso de **reinterpret_cast<>**.

```
std::int8_t read_binary(std::istream & input) {
  int8_t value;
  input.read(reinterpret_cast<char*>(&value), sizeof(value)); // !!!
  return value;
}

void write_binary(std::ostream & output, std::int16_t const & value) {
  output.write(reinterpret_cast<char const*>(&value), sizeof(value)); // !!!
}
```

Para anular una regla en una línea de código se puede escribir un comentario en la línea de código anterior con el texto **NOLINTNEXTLINE** seguido de la regla que se desea anular entre paréntesis.

```
std::int8_t read_binary(std::istream & input) {
  int8_t value;
  // NOLINTNEXTLINE(cppcoreguidelines-pro-type-reinterpret-cast)
  input.read(reinterpret_cast<char*>(&value), sizeof(value));
  return value;
}

void write_binary(std::ostream & output, std::int16_t const & value) {
  // NOLINTNEXTLINE(cppcoreguidelines-pro-type-reinterpret-cast)
  output.write(reinterpret_cast<char const*>(&value), sizeof(value));
}
```

2.9.2. Código fuente de pruebas

En el código de pruebas es muy habitual que sea necesario utilizar valores arbitrarios. En estos casos está justificado anular las reglas que prohíben el uso de número mágicos en toda la sección de código.

Para ello se pueden usar entre comentarios los identificadores **NOLINTBEGIN** y **NOLINTEND** seguidos de las reglas que se desea anular.

```
#include ...

// NOLINTBEGIN(cppcoreguidelines-avoid-magic-numbers)
// NOLINTBEGIN(readability-magic-numbers)

...

// NOLINTEND(readability-magic-numbers)
// NOLINTEND(cppcoreguidelines-avoid-magic-numbers)
```