C++ Code Guideline

CONFIDENTIAL
(Edition 6)

Submitted By:	*Max Lyadvinsky*
Submitted To:	Developers

1. Revision History

Date	Updated by	Comments
4/20/2005	Max Lyadvinsky	Document created.
4/21/2005	Max Lyadvinsky	Updated after discussion with the project managers and the core team.
8/22/2005	Max Lyadvinsky	Added language requirements for comments.
7/16/2007	Vladimir Solontsov	Added rule about Similar Identifiers. Added section Library Usage.
12/20/2007	Alexander Vorobiev	4.7.2 was changed
02/05/2008	Andrew Maksimov	5.1.3 was changed
7/31/2008	Eugene Fedorov	4.4.1, 4.4.2 were changed 4.4.3.1-4.1.3.6 were added
7/31/2008	Alexey Andreev	4.5.7, 5.1.1, 5.1.3 were changed Added section Casts Added section Initialization list formatting
8/19/2008	Andrew Maksimov	4.3.3, 4.4 were changed
8/20/2008	Eugene Fedorov	4.4.6-4.4.9 were changed 4.4.10 was added
8/20/2008	Alexey Andreev	4.4.9, 4.6.6, 4.6.9, 4.6.16 were changed Added rule Exceptions Added rule Headers
8/21/2008	Andrew Maksimov	4.7 was added
12/16/2008	Alexander Vorobiev	4.4.6 - 4.4.7 were fixed
2/2/2011	Anatoly Nikolaev	7 - 7.2 were added
1/15/2013	Manukyan Emil	Fixed typos and conversion artifacts
1/25/2013	Manukyan Emil	Updated 3. with status of guidelines and refined single statement formatting rules in 4.3.3
2/11/2013	Manukyan Emil	Updated 4.7.1 and added 5.5
4/26/2013	Manukyan Emil	Added 4.7.3 and 5.4.1
9/14/2019	Vitaly Murashev	Added 6.3
1/28/2021	Vitaly Murashev	Updated 5.3. catch(...) is bad practice
4/13/2021	Andrey Zinovyev	Updated 3 and 4.4.4

2. Table of Contents

3. Overview

This document is based on the .NET Framework Design Guidelines and Brad Abrams' Internal Coding Guidelines (01/26/2005). The guidelines outlined below are mandatory for all software developers working at Acronis. Adherence to guidelines is mandatory. If on review newly written code does not conform to any section formulated by "must" term, reviewer must mark it with Defect of Style type. It is prohibited to create defects of Style type if no formal rule is requiring statement of the defect. For legacy code reviewer can ask via Comment but cannot require via Defect to change unmodified code not conforming to coding guidelines unless newly added code does not break guidelines in legacy code.

4. Style Guidelines

4.1. Existing Acronis Libraries

Do use the style of the existing library when adding your files to this library. Do use the following rules when creating your own library or the code outside the existing libraries.

4.2. Tabs and Indenting

Tab characters (0x09) must not be used in the source code. All indentation must be done with 2 space characters.

4.3. Bracing

4.3.1. General Issues

Open braces must always be at the beginning of the line after the statement that begins the block. Contents of the brace must be indented by 2 spaces.
For example:

Right:

```
if (someExpression)
{
  DoSomething();
}
else
{
  DoSomethingElse();
}
```

4.3.2. Case Statements

The "case" statements must be indented from the "switch" statement like this (the Visual Studio smart indenting works the same way):

Right:

```
switch (someExpression)
{
case 0:
  DoSomething();
  break;

case 1:
  DoSomethingElse();
  break;

case 2:
  {
    int n = 1;
    DoAnotherThing();
  }
  break;
}
```

4.3.3. Single Statements

Braces can be considered optional if the inner scope is represented by single line (not single statement). Though you can omit braces when bodies of "if", "while", "for", etc. consist of single line you must follow the following rules:

1. Bracing style within single function should be consistent
2. When omitting braces you must follow the statement by an empty line

Wrong (for's body consists of two lines)

```
for (int i = 0; i < 100; i++)
  if (Check())
    DoSomething();
```

Wrong (inconsistent bracing style within single function)

```
for (int i = 0; i < 100; i++)
{
  if (Check())
    DoSomething();
}
```

Wrong (absence of empty line after omitted braces)

```
if (Check())
  DoSomething();
DoSomethingElse();
```

Right:

```
for (int i = 0; i < 100; i++)
{
  if (Check())
  {
    DoSomething();
  }
}
```

Right:

```
if (Check())
  DoSomething();

DoSomethingElse();
```

Right:

```
if (someCondition)
  DoSomething();
else if (someOtherCondition)
  DoSomethingElse();
else
  HandleDefaultCase();

DoSomethingDifferent();
```

Please do not write several statements at a single line. It is hard to debug such statements later.

Wrong:

```
for (int i = 0; i < 100; i++){ DoSomething(i); }
```

Right:

```
for (int i = 0; i < 100; i++)
{
  DoSomething(i);
}
```

Right:

```
for (int i = 0; i < 100; i++)
  DoSomething(i);
```

4.4. Commenting

There are two kinds of commenting: documentation comments and implementation ones. The implementation comments are the text blocks delimited by "/**...**/" or by "//". The documentation comments, based on the Doxygen markup, are always delimited by "/***...**/" or "///".
The implementation comments explain particular implementation details. These comments should provide additional information that is not easily understandable from the code itself.
The purpose of the documentation comments is describing a code specification. These annotations are used mainly by developers who usually do not have the source code at hand. The documentation comments must show the code intention, algorithmic overview, and/or a logical flow. What would be the best is if a new developer could understand the code behavior and functionality only via reading the documentation comments. The comments can be extracted to HTML files by using Doxygen tools.
Comments must be written in English only.
All requirements below are mandatory except when they are explicitly marked optional.

4.4.1. Doxygen Comments

All developers must use the Doxygen markup in their API documentation. Do use the "Javadoc Flavor" Doxygen commenting style while adding any tags, for instance, "@tag" rather than "\tag". Please refer to Doxygen Manual and CommunityDoxygenProject for additional information.

4.4.2. Commenting of Public Library Interfaces

Public library interfaces are headers, which are located in the "project\include\" directory. Public libraries are used by many Acronis projects. Therefore, their usage should be as simple as possible. To help in that endeavor, it must be fully documented to let any developer use it.
The following entities in public library interfaces must have documentation comments.

- Definitions of all classes, templates and enumeration types.
- Declarations of all member functions and data members.

4.4.3. Commenting of Library Implementation

The level of details in library implementation comments is up to the corresponding library maintainer. Even though some very small routines need no commenting at all, it is hoped that most routines and classes will have remarks reflecting the programmer's intent and approach.

4.4.4. Copyright Notice

Each file must start with a copyright notice. The notice must have the following format.

```
/**
@brief    The module short explanation
@details  Copyright (c) 2001-2021 Acronis
@author   The author's name (the author's e-mail)
*/
```

4.4.5. Single-Line Comments

The two-slash style of comment tags should be used in most situations for implementation and documentation remarks. Do use the "Javadoc Flavor" style or a special C++ Doxygen style for adding a single line comment, for instance, "/// @tag comment".
Wherever it is possible, place comments above the code instead of beside it. Below are some examples.
// This is required for WebClient to work through the proxy
GlobalProxySelection.Select = new WebProxy("[http://itgproxy|http://itgproxy]");

```
// Create the object to access Internet resources
WebClient* client = new WebClient();
```

Comments can be placed at the end of a line when space allows. The comments should be aligned for more comfortable reading.

4.4.6. Class Declaration Comments

Please use the following Doxygen tags in class declaration comments, in the order which they are listed in:

```
"@brief"
```

A short description of the class. This description must provide the reason why the class exists, what the class does and where to use it. Examples of the class descriptions are below.

```
"MyWidgetClass is a class to handle lists",
"A widget that shows balls bouncing",
"Monitors directory(s) for changes".
"@author"
```

Please use this tag to indicate the class author(s) if they are have not been mentioned in the file copyright notice. If the class does not have the "@author" tag, "@author" of the file will be responsible for this class.

```
"@details"
```

(optional)
A detailed description of the class.
All members in the class (public, protected and private) must have a corresponding Doxygen comment
Here is a scheme of a documented C++ class using Acronis style.

```
/**
  @brief    The brief description of a class
  @author   Ivan Ivanov (Ivan.Ivanov@acronis.com)
  @details  The detailed description of the class
*/
class SomethingUseful
{
public:
  SomethingUseful();
  /**
   @brief                      The brief description of SomeMethod
   @details                    The detailed description of SomeMethod
   @param newItemHash          the parameter that means something
   @return ItemHash            The method returning some variable
  */
  int SomeMethod(int newItemHash);

private:
  /// @brief Some packet hash
  int ItemHash;
  /// @brief Something was completed
  static bool HasDoneSomething;
};
```

4.4.7. Member Function Declaration Comments

Please use the following Doxygen tags in function declaration comments, in the order which they are listed in.
"@brief"
A short description of the member function. Please do not repeat the name of the function - write a short sentence describing the function instead.
"@details" (optional)
A detailed description of the member function.
"@param"
A short description of the parameter, what a function expects its callers. This tag is required for every parameter of the member function.
If the description consists of the only one sentence, there must be no space at the end. The first word must not be capitalized, and there must not be a dash symbol between a parameter name and the description. The "-" symbol is added by the document processor, and if a developer adds another one, then there will be the two in the resulting "HTML/PDF/TEX/etc." documentation.
Do not use a period at the end of a description.

Wrong:

```
@param foo The value to convert
@param foo the value to convert.
@Param foo the value to convert
```

Right:

```
@param foo the value to convert
```

```
"@return"
```

A short mandatory description of what the function returns.

4.4.8. Enumerators

An enumeration declaration must start with the Doxygen comment "///", in which a purpose of an enumeration is described. Each enumeration value must also have the Doxygen comment "///" on the previous line with short explanation of its values.

Example:

```
/// The possible statuses that job may have
enum JobStatuses
{
  /// The job is ready to start
  JOB_STATUS_WAITING,
  /// The job is actively running
  JOB_STATUS_RUNNING,
  /// The job has completed its run successfully
  JOB_STATUS_COMPLETED
};
```

4.4.9. Typedefs and Macros

The "typedef" and "macro" declarations must start with Doxygen comments in the following format.

Example:

```
/**
 @brief    Gets the maximum value
 @details  Macro MAX gets the maximum of given values
*/
#define MAX(x,y) ((x)>(y)?(x):(y))

/**
 @brief    The new type conforms to unsigned int
 @details  Used to economise the free space
*/
typedef unsigned int UINT;
```

While commenting "typedef", it is not necessary to specify the detailed description, so the "@details" tag is optional for them, but it can be used in case of need. This tag is strongly required for macros.

4.4.10. Checking the Spelling and Grammar

Documentation represents a project a lot more than people tend to think. Take the time to give it the proper polish that it deserves. The documentation explains what the class does, but for a lot of developers it will be the only view on how the class works.

4.5. Spacing

Spaces improve readability by decreasing code density. Here are some guidelines for the use of space characters within code.

4.5.1. Commas between Function Arguments

Do use a single space after a comma between function arguments.

Wrong:

```
CreateFoo(myChar,0,1);
```

Right:

```
CreateFoo(myChar, 0, 1);
```

4.5.2. Parenthesis and Function Arguments

Do not use a space after the parenthesis and function arguments.

Wrong:

```
CreateFoo( myChar, 0, 1 );
```

Right:

```
CreateFoo(myChar, 0, 1);
```

4.5.3. Function Name and Parenthesis

Do not use spaces between a function name and parenthesis.

Wrong:

```
CreateFoo ();
```

Right:

```
CreateFoo();
```

4.5.4. Brackets

Do use spaces inside brackets as follows.

Wrong:

```
x = dataArray[ index + 1 ];
```

Right:

```
x = dataArray[index + 1];
```

4.5.5. Flow Control Statements

Do use a single space before flow control statements.

Wrong:

```
while(x==y)
```

Right:

```
while (x == y)
```

4.5.6. Comparison Operators

Do use a single space before and after comparison operators.

Wrong:

```
if (x==y)
```

Right:

```
if (x == y)
```

4.5.7. Pointers and References

Do not use spaces between a type name and "*****" or "&". Use a single space after "*" or "&".

Wrong:

```
Window *mainWindow;
```

Right:

```
Window* mainWindow;
```

4.6. Naming

4.6.1. Capitalization Styles

We will use the following three conventions for capitalizing identifiers.

Style	Description	Samples
PascalCasing	The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters.	BackColor
camelCasing	The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized.	backColor
UPPERCASING	All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters.	System.IO System.Web.UI
lowercasing	All letters in the identifier are lower.	iterator

4.6.2. Hungarian Notation

Do not use Hungarian notation.

4.6.3. Namespace

Do use PascalCasing for namespace names.

```
namespace Processor
{
}
```

4.6.4. Classes

Do use PascalCasing for class names. Do not prefix class names with any letter. Use a noun or noun phrase to name a class, and avoid abbreviations.

```
class AppDomain;
```

4.6.5. Typedefs

Do use PascalCasing for names in "typedef". It is recommended to define containers and iterators before using them for declaring class members, etc.

```
typedef list<string> PhoneBook;
```

When you create aliases for smart pointers with "typedef", please use the class name and the suffix "Ptr" for the alias name.

```
typedef Common::shared_ptr<MyClass> MyClassPtr;
```

4.6.6. Exception Classes

Do use PascalCasing for exception class names. Do suffix class names with "Error".

```
class OutOfRangeError;
```

4.6.7. Structures and Unions

Use structures and unions only for external data located on storage devices, in registry (MBR, Diskadm, LDM structure, etc.), or for common interfaces (TCP/UDP packet, etc.). Try to avoid structures otherwise. Do use PascalCasing for structure names.

```
struct Mbr;
```

4.6.8. Enums

Do use PascalCasing for enum names. Do not prefix enums with any letters.

```
enum ErrorLevel;
```

4.6.9. Enums Values

Do use UPPERCASING for enum values. Separate words within the value name by underscores ("_").

```
enum JobStatuses
{
  JOB_STATUS_WAITING,
  JOB_STATUS_RUNNING,
  JOB_STATUS_COMPLETED
};
```

4.6.10. Macros

Use macros only if absolutely required. Do use UPPERCASING for macros. Separate words within the macro name by underscore. Try to avoid macros and use templates/enums instead.

```
#define DECLARE_VXD(DriverClass, MajorVersion, MinorVersion) \
  DriverClass Driver; \
  VxD_Desc_Block VxdDdb = \
  { \
    DDK_VERSION, \
    MajorVersion, \
    MinorVersion, \
    DriverClass::SysControl, \
    'Prev', \
    sizeof(struct VxD_Desc_Block), \
    'Rsv1', \
    'Rsv2', \
    'Rsv3' \
  };
```

4.6.11. Protected and Private Member Variables

Do not use a prefix for member variables ("*'*,*"m*", "s_", etc.). Do use PascalCasing for protected and private member variables.

```
class App
{
private:
  Windows AppWindows;
};
```

4.6.12. Public Member Variables

Do not use public members unless it is absolutely necessary. Do use PascalCasing in case the public member is absolutely required.

```
class App
{
public:
  static App AppInstance; // Never use public members!
};
```

4.6.13. Methods

Do use PascalCasing for class methods. It is better to use verbs or verb phrases to name methods.

```
class App
{
public:
  // App constructor
  App();
  // App method
  DoAppStuff();
};
```

4.6.14. Functions

Do use PascalCasing for function names. It is better to use verbs or verb phrases to name the functions.

```
void DoSuperStuff();
```

4.6.15. Parameters

Do use camelCasing for parameters. Parameter names should be descriptive enough, since the names and types will be used to determine their meaning.

```
void DoStuff(const Stuff& stuffData);
```

4.6.16. Local Variables

Do use camelCasing for local variables.

```
int index, superIndex;
```

You can use well-known short names inside loops ("i","j" and "k"), but the following conditions should be observed.

- A variable must be declared inside the loop statement.
- The loop nesting level should not exceed three.
- The size of the loop body should be reasonable - not more than 20 lines.

4.6.17. Global Variables

Do not use global variables unless it is absolutely necessary. Do use PascalCasing for global variables.

```
Driver DriverInstance;
```

4.6.18. Files

Do use lowercasing for files. Separate words within the file name by underscores.

```
phone_book.h
phone_book.cpp
```

4.6.19. Similar Identifiers

Do not use identifiers with a difference in one symbol, including character case.

Wrong:

```
class App
{
  int Value;
  void foo(int value) // Wrong
  {
    Value = value;    // Wrong
    int value1 = value;    // Wrong
  }
};
```

Right:

```
class App
{
  int Value;
  void SetValue(int newValue)
  {
    Value = newValue;    // Right
  }
};
```

4.7. Formatting

4.7.1. Constructor Member Initialization List Formatting

Use the following formatting style of member initialization list for any number of members and base classes:

```
App::App()
  : BaseClass1(0)
  , BaseClass2(0)
  , Field1(0)
  , Field2(0)
{
}
```

The order of members in the initialization list must match the actual order of initialization which is defined by the declaration order (virtual base classes must be taken into account).

4.7.2. Enumeration Declaration Formatting

The enumeration declaration must be formatted in the following style.

Right:

```
enum ErrorLevel
{
  LITE,
  SMART,
  SUPERSMART
};
```

Never use the following formatting style for enumeration declarations.

Wrong:

```
enum ErrorLevel
{
  LITE
 ,SMART
 ,SUPERSMART
};
```

4.7.3. Inheritance formatting

Single inheritance must be formatted in the following style.

Right:

```
class Heir : public Base
{
};
```

Never use the following formatting styles for single inheritance.

Wrong:

```
class Heir: public Base
{
};
```

Wrong:

```
class Heir
  : public Base
{
};
```

Use the following formatting style for multiple inheritance:

Right:

```
class Heir
  : public Base1
  , public Base2
{
};
```

4.8. Abbreviations

To avoid confusion and guarantee the cross-language interoperation, follow the rules below regarding the usage of abbreviations.

4.8.1. Contractions

Do not use abbreviations or contractions as a part of an identifier name. For example, use "GetWindow" instead of "GetWin".

Right:

```
int firstLetter;
```

Wrong:

```
int fLtr;
```

4.8.2. Acronyms

Do not use acronyms that are not generally accepted in the computing field. For example, do not use UFW (United Farm Workers of America). Although where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use UI for User Interface and LDAP for Lightweight Directory Access Protocol.
Use PascalCasing or camelCasing (see this point for the capitalization rules) for acronyms more than two characters long. For example, use "HtmlButton" or "htmlButton". However, you must capitalize acronyms that consist of only two characters even if such acronyms are part of an identifier.
Use SystemIO instead of SystemIo.
Use GetID() instead of GetId()

5. Coding Techniques

5.1. Casts

Do not use C style casts in C++ code. All casts must be implemented only by using C++ casts.

5.2. Pointers and References

Please prefer to use the references wherever possible.

5.3. Exceptions

Preferably "Common::Error" should be used as a base class for exception classes. Exceptions should be caught within the library they appear in. In case of a strong necessity, libraries can throw exceptions but such cases must be approved by Architects.

Swallowing Exceptions

Swallowing critical exceptions will cause your program to do either of two things – to fail in unexpected ways downstream or prevent the program from fulfilling it's purpose. Sometimes programmers will catch any exception via catch(…) and then swallow them . This is usually done for exceptions that the programmer did not foresee happening. However, this can lead to downstream failure – sometimes with no obvious reason for the failure since the stacktrace disappears with the swallowed exception.
So avoid catch(...) as much as possible

5.4. Headers

Do not use "using namespace" statement in headers, and before all "#include" statements in implementation source files.

5.4.1. Header Guards

Do use #pragma once as header guard in all newly created headers. Fix existing #ifdef/#define guards at your convenience.

5.5. Explicit keyword

Do use explicit keyword with constructors that can be called with single argument unless you use implicit type conversion performed by that constructor in your code.

6. Libraries Usage

6.1. STL and CRT Usage

6.1.1. Restriction of Usage std::string

"std::string" is allowed only as an ASCII data container and only in cases when data source is non-UNICODE. Do use "Common:: String" for wide strings.

6.1.2. Do not Use Streams from STL

Do not use streams in production code except cases agreed with the architects.
In other cases do use classes and functions from "include/file/" to work with files, and "include/processor/utils/format.h" to format data.

6.1.3. Do not Use File Manipulation Routines from CRT

Do not use file manipulation routines in production code except cases agreed with Architects.

6.2. Using Third-Party Libraries

If you want to use a third party library, you have to answer for the next questions.

- Is it open source?
- Is it cross-platform?
- What is about license?
- What value will we have?

Then you have to send the request through your Project Manager and Architects to VP of Engineering. Only libraries approved by VP of Engineering are usable in Acronis.

6.3. Explicit using WINAPI and POSIX API

When WINAPI or POSIX API is used explicitly you are supposed to use type definitions and constants exposed by this API.
Right:

```
HANDLE hStdin = NULL;
HANDLE hStd = GetStdHandle(STD_INPUT_HANDLE);
BOOL ret = DuplicateHandle(GetCurrentProcess(), hStd, GetCurrentProcess(), &hStdin, 0, TRUE,
DUPLICATE_SAME_ACCESS);
```

Wrong:

```
HANDLE hStdin = nullptr;
HANDLE hStd = GetStdHandle(STD_INPUT_HANDLE);
bool ret = DuplicateHandle(GetCurrentProcess(), hStd, GetCurrentProcess(), &hStdin, 0, true,
DUPLICATE_SAME_ACCESS);
```

In plain English - when using platform specific API please never try to substitute platform specific types and definitions with builtin C++ keywords, i.e. for WINAPI it is OK to use BOOL, TRUE, FALSE, NULL and is wrong to substitute them with bool, true, false, nullptr.

7. Localization Requirements

7.1. GUI text strings and other localizable resources should always be externalized rather than embedded.

GUI strings are the text lines, descriptions, wizards' headings, messages, etc., that users see in the product interface.

7.2. Do not assume that the dates will always appear in month-day-year format.

Locale-specific representation of MM-DD-YY may be different. For example, Korean time format order is as follows: AM/PM, hours, minutes.
For example:

Wrong (from ATIH 2011):

```
QString str = dt.toString("dd.MM.yyyy - hh:mm:ss.zzz");
```

Hardcode is bad. This format will be changed in some localizations.

Wrong (Taken from early proposed fix):

```
QString str = dt.toString(GET QSTRING(TEXT FULL DATE FORMAT));
```

Taking format values from the text may seem a workaround at first - but the text could be easily broken by translators.

Let's do not invent our own bicycle and not duplicate the code. Let's use IBM, Nokia and Microsoft solution which is freely available and easy to use:

```
QString str = dt.toString(Qt::SystemLocaleLongDate);
```

It's really simple and international.

Using ICU date-time formatting is good when standard facilities (long and short date format) aren't enough. Creating custom formats like (February '99) is possible, but only when using ICU to construct it.

http://site.icu-project.org/

By the way, the ICU version that we use is not the latest one. Some features are not available.

8. .clang-format and .clang-tidy

`clang-format` and `clang-tidy` are modern applications that can enforce many of these rules automatically. While `clang-format` is a simple text formatter, `clang-tidy` needs to know some syntactic analysis, therefore it might be harder to master. For example, on Windows, it was fairly easy to integrate a `clang-format` **VS Code** extension, but I didn't succeed to use one with `clang-tidy` so I finished by call it manually (command line). Anyway, I add my `.clang-format` and `.clang-tidy` files here in hope of future improvement and collaboration.

.clang-format

```
# # We'll use defaults from the LLVM style.
# BasedOnStyle: LLVM

# was BreakBeforeBraces: Linux
BreakBeforeBraces: Custom
BraceWrapping:
  AfterClass: true
  AfterFunction: true
  AfterNamespace: true
  SplitEmptyFunction: false
  SplitEmptyRecord: false

# AllowShortEnumsOnASingleLine: true
BreakBeforeTernaryOperators: false
BreakConstructorInitializers: BeforeComma
BreakInheritanceList: BeforeComma
BreakStringLiterals: false
NamespaceIndentation: All
PointerAlignment: Left
SpaceBeforeInheritanceColon: false

# No: it does it even for one liners
InsertBraces: false
```

.clang-tidy

```
# * modernize-concat-nested-namespaces: it nominally exists since C++17 and
#   it's conflicting with clang-format's NamespaceIndentation: All
# * hicpp-vararg: LogD,  warning: do not call c-style vararg functions [hicpp-vararg]
# * hicpp-exception-baseclass: warning: throwing an exception whose type 'Common::Error'
#     is not derived from 'std::exception' [hicpp-exception-baseclass]
Checks: hicpp-*, readability-identifier-naming,
        -google-readability-braces-around-statements,
        -modernize-concat-nested-namespaces
        -hicpp-vararg, -hicpp-exception-baseclass

CheckOptions:
  - { key: readability-identifier-naming.ClassCase, value: CamelCase}
  - { key: readability-identifier-naming.EnumCase, value: CamelCase}
  - { key: readability-identifier-naming.EnumConstantCase, value: UPPER_CASE}
  - { key: readability-identifier-naming.FunctionCase, value: CamelCase}
  - { key: readability-identifier-naming.MemberCase, value: CamelCase}
  - { key: readability-identifier-naming.MethodCase, value: CamelCase}
  - { key: readability-identifier-naming.NamespaceCase, value: CamelCase}
  - { key: readability-identifier-naming.ParameterCase, value: camelBack}
  - { key: readability-identifier-naming.TypedefCase, value: CamelCase}
  - { key: readability-identifier-naming.VariableCase, value: camelBack}

# readability-braces-around-statements
# also see clang-format:InsertBraces
# aliases: hicpp-braces-around-statements, google-readability-braces-around-statements
# TODO: google-readability-braces-around-statements.ShortStatementLines is always there (--dump-config), 1,
but won't be applied
  - { key: google-readability-braces-around-statements.ShortStatementLines, value: 3}
  - { key: hicpp-braces-around-statements.ShortStatementLines, value: 3}
```