

Coding Standard

Goals

Efficiently develop code and applications with the following qualities :

- **Robust** : runs without crash and protects the data from being lost or corrupted.
- **Secure** : protects the data from being stolen or hacked.
- **Ergonomic** : does exactly what the user needs and can be used in an productive and intuitive manner.
- **Efficient** : minimizes processing times to maximize the user productivity.
- **Maintainable** : is easy to fix and enhance by any programmer in the team.
- **Extensible** : is easy to extend with new features by reusing existing components.
- **Consistent** : looks like it has been designed and implemented by a single developer.

Specificity

This coding standard favors readability over compactness, by :

- Forbidding the use of cryptic acronyms, abbreviations, prefixes and suffixes;
- Using different letter cases for classes, class members and local variables;
- Including the class name in the attribute and variable names.

Rules

- Develop the application and its components with simple, robust and efficient code which will be easy to understand, extend and debug by any programmer in the team.
- Develop any piece of code so that it's :
 - easy to understand just by itself;
 - impossible to guess who has actually worked on it.
- Use **American English** for all the code, including comments.

```
InitializeColor();  
MoveForward();
```

- Use the **meter** as the default distance unit.
- Use the **second** as the default time unit.
- Use **four spaces** instead of tabulations, so that the code indentation does not depend on the editor settings.
- Choose **short meaningful identifiers** for class, attribute, method, constant and variable names.
- Use **standard prefixes** :
 - First, Last, Post
 - Prior, Next
 - Sub, Super, Base

- Initial, Final
- Old, New
- Backward, Forward
- Left, Right
- Back, Front
- Bottom, Top
- Minimum, Maximum
- Lower, Higher, Upper
- Horizontal, Vertical
- Local, Global
- Use **standard suffixes** :
 - Index, Count
 - Array, List, Map, Dictionary
- Use **standard verbs** :
 - Initialize, Update, Finalize
 - Is, Has
 - Reset, Set, Get, Find
 - Clear, Fill
 - Add, Remove
 - AddFirst, AddLast
 - Create, Destroy
 - Start, Stop
 - Begin, End
 - Enter, Exit
 - Open, Close
 - Read, Write
 - Load, Save
 - Pause, Resume
 - Enable, Disable
 - Lock, Unlock
 - Select, Deselect
 - Activate, Deactivate
 - Attach, Detach
 - Increment, Decrement
 - Increase, Decrease
 - Compress, Decompress
 - Connect, Disconnect
 - Send, Receive
 - Grant, Revoke
- Write your **types** (classes, structures, enumerations, etc) in **UPPER_CASE**, without articles.

```
class TANK_SHELL
{
    VECTOR_3
        PositionVector;
    QUATERNION
        RotationQuaternion;

    ...
}
```

- Write your **type members** (methods, attributes, constants, etc) in **PascalCase**, without articles.

```
ShootShell(
    Muzzle.PositionVector,
    Muzzle.RotationQuaternion
);
```

- Write your **local variables** and **method parameters** in **snake_case**, without articles.

```
void ShootShell(
    VECTOR_3 shell_position_vector,
    QUATERNION shell_rotation_quaternion
)
{
    TANK_SHELL
        shot_tank_shell;

    ...

    shot_tank_shell
        = new TANK_SHELL(
            shell_position_vector,
            shell_rotation_quaternion
        );

    ...
}
```

- Don't use acronyms, abbreviations or single-letter variables.

```
TANK FindTank(
    int tank_identifier,
    int first_tank_index,
    int post_tank_index
)
{
    int
        tank_index;

    for ( tank_index = first_tank_index;
```

```

        tank_index < post_tank_index;    // no i, j, n, etc
        ++tank_index )
    {
        if ( TankArray[ tank_index ].Identifier == tank_identifier )
        {
            return TankArray[ tank_index ];
        }
    }

    return null;
}

```

- If you really have to use an acronym, capitalize it in member names.

```

DATABASE_URL
    DatabaseUrl;

```

- If a variable name collides with a predefined identifier, simply add a trailing underscore.

```

CLASS
    class_;

class_ = new CLASS;

```

- Use a noun or noun phrase for classes, constants, attributes and variables.
- Include the meaningful part of the class name in attribute and variable names.

```

Dictionary<PLAYER, string>
    ActivePlayerDictionary;
List<ENEMY>
    CloseEnemyList;
TANK[]
    EnemyTankArray;
VECTOR_3
    InitialShellPositionVector,
    TankVelocityVector;

void ShootShell(
)
{
    SHELL
        last_shot_shell,
        shot_shell;

    ...
}

```

- Use a verb in the imperative mood for methods (Set, Get, Find, ...).
- Use a verb in the indicative mood for boolean inquiries (Is, Has, Can, ...).

- Declare the method parameters in the same order as in the method name.

```
bool FindPlayerIndexByName(  
    ref int player_index,  
    string player_name  
)  
{  
    ...  
}
```

- Use a positive affirmation for boolean variables and attributes.

```
if ( game_is_paused )  
{  
    ...  
}
```

- When the attribute name starts like its owner type, omit the common prefix.

```
class TANK  
{  
    TANK_SHELL[]  
        ShellArray;  
    bool  
        IsDamaged;  
  
    void ShootShell(  
        TANK_SHELL tank_shell  
    )  
    {  
        ...  
    }  
}
```

- Align matching braces.

```
bool CanShoot(  
    )  
{  
    return ShotShellCount < MaximumShellCount;  
}  
  
// ~~~~  
  
void ShootShell(  
    VECTOR_3 initial_velocity_vector  
    )  
{  
    ...  
}
```

- Use braces even for single statement blocks.

```
if ( remaining_shell_count > 0 )
{
    ShootShell();
}
else
{
    Reload();
}
```

- Declare each attribute, variable and method parameter name on separate line.

```
int
    tank_count,
    tank_index;
```

- Try to declare all local variables at the start of the method, to improve the algorithm readability.
- Group local variables of the same type, and sort the declarations by ascending types and names, so that the declaration of a variable can be located at a glance.

```
int
    shell_count,
    shell_index,
    tank_count,
    tank_index;
string
    player_name,
    target_name;
```

- Try to split statements on several lines if they become wider than 100 characters, so that it's easy to edit two code files side by side on a single monitor.
- When splitting an expression on several lines, start the next lines with an operator and align it with the start of its left operand (or else indent it by 4 spaces).

```
if ( ( tower.GetDistance(
        tower_target,
        weapon_type
    )
    > tower.MaximumShootingDistance )
    || ( tank_distance > maximum distance
        && tank_health > 0.5 ) )
{
}
}
```

- Add exactly one space :
 - after ([,
 - before)]
 - after if while for foreach return ...
 - before and after operators
- Add exactly one empty line :
 - before and after a standard comment;
 - after the local variable declarations;
 - between } and the next statement;
 - between if while for foreach do return and the prior statement.
- Use standard file extensions.
 - C# : cs
 - C : c, h
 - C++ : cpp, hpp
 - Javascript : js
 - HTML : html
 - CSS : css
- Declare one class per source code file.
- Use the class name in lowercase as file name.

```
tank_shell.cpp
tank_shell.hpp
```

- Use the class name in uppercase as Unity file name.

```
TANK_SHELL.cs
```

- Group the class elements by category, declared in the same predefined order :
 - Imports.
 - Types.
 - Constants.
 - Attributes.
 - Constructors.
 - Destructor.
 - Operators.
 - Inquiries : methods which don't change the class attributes.
 - Operations : methods which change the class attributes.
- In a class, declare the called methods before the calling methods, so that the class code can be understood by a single sequential read.
- Use public attributes and methods, unless you really need to declare some of them as private.
- Delimitate the code sections with standard comments.

```
// -- IMPORTS
```

```

...

// -- TYPES

class NAME
{
    // -- CONSTANTS

    ...

    // -- ATTRIBUTES

    ...

    // -- CONSTRUCTORS

    ...

    // -- DESTRUCTOR

    ...

    // -- OPERATORS

    ...

    // -- INQUIRIES

    ...

    // -- OPERATIONS

    ...
}

```

- Don't use standard comments for empty sections.
- Align multiple lines comments with the surrounding statements, and write them as sentences.

```

/*
    A long explanation which is so long that it will have to be
    be split on several lines.
*/

...

```

- Align single line comments with the surrounding statements, and write them as sentences.


```
// A short explanation on a single line.
```

```
...
```

- Put end of line comments exactly four spaces after the statement, and start them in lowercase.

```
DoSomethingWeird();    // a short explanation
```

- Instead of adding comments to explain the code intent, refactor it to :
 - make it easy to understand without comments;
 - improve its reusability.
- Begin C++ header files with `#pragma once`.

```
#pragma once
```

```
// -- IMPORTS
```

```
#include "tank.hpp"
```

```
#include "tank_shell.hpp"
```

```
...
```

- Name the unit test class by simply adding a `_TEST` suffix to the class name.

Advices

- Design before you program, to avoid losing precious time in developing the wrong solution to the wrong problem.
- First find what is really needed, by taking a few minutes to write :
 - a short text explaining how to use the application, to optimize the application interface before implementing it;
 - a short text explaining what the application components will do, to optimize the application architecture before implementing them;
 - a short text or test code explaining to the other programmers how they will use the application components, to optimize their interface before implementing them.

Version

1.0

Author

Eric Pelzer (ecstatic.coder@gmail.com).

License

This document is licensed under the Creative Commons Attribution-NonCommercial 4.0 International.