Coding Standards

1. Content

2. Purpose

This document provides a guidance to good programming style and design to obtain readable and maintainable code.

Readable: code should be written in a manner that is easy to read and understand.

Maintainable: code should be written in a manner that is consistent, readable and simple in design.

It only handles rules regarding programming seen in this module.

Remarks made in labs and lectures still apply.

3. Code guidelines

3.1. Naming

Using naming conventions reduces the effort needed to read and understand code.

3.1.1. Example

```
class MyClass
{
public:
		MyClass();
		void DoSomething();
		int GetProperty() const;
		void SetProperty(int value);
protected:

private:
	int m_Property;
	static const int m_NrGrades{10};
	int m_Grades[m_NrGrades];
	OtherClass *m_pOtherClass;
}
```

3.1.2. Naming identifiers

An identifier is a name which is used to refer to a variable, constant, method or type (e.g. class, enumeration, struct).

a. General

Rules	Examples
The name of an identifier should suggest the usage of it. It should consist of a meaningful descriptive name. Note: Conventional usage of simple identifiers (i, x, y…) in small scopes can lead to cleaner code and will therefore be permitted	**CreateWindow**: it is obvious that this method contains the code that creates a window.
The name can include abbreviations that are generally accepted.	nrLives

When the name consists of more words, they are written as one without underscores and each new word starts with a capital letter.
Exception: an underscore is used in the prefix that indicates a class private member (m_) or a global variable (g_)(see further)
	DoSomething();
	ProcessKeyDownEvent()
	m_Position
	g_SomeBadGlobal

b. Variables

Rules	Examples
Always start with a **lower case** letter.	bool **i**sDown int **b**ulletCntr int **w**idth
The **private member** variables always start with the prefix **m_. The name starts with a upper case letter.**	int **m_**Width int **m_**Height

The **public member** variables **for structs** follow the local variables rules.	e.g. in Point2f struct
	float x;
	float y;

The **global** variables always start with the prefix **g_**	std::string g_WindowWidth
Pointer variables have the lowercase prefix **p**	Hero* m_**p**Hero Enemy* m_**p**Enemy
	Hero* pHero Enemy* pEnemy
For the **boolean** variables, choose a name that corresponds with the true value.	**bool m_IsHuman** says more than **bool m_Form**
Array or container identifiers are always plural form	m_Numbers m_pEnemies

c. Methods/functions

Rules	Examples
Always start with a **uppercase letter**. First word is a verb. The name of the method describes what it does. Exception to this rule is the **main** function.	ProcessKeyDownEvent CreateRenderer

Start with the word **Get** if the method returns the value of a variable, however, in case of a **bool** variable, start with **Is** instead. Or just give the method the same name as the variable, this is an exception to the verb rule	**Get**Width **Get**Height **Is**KeyPressed Length
Start with the word **Set** if the method changes the value of a data member. The method needs an argument.	**Set**Color **Set**Font **Set**KeyPressed(bool state)

d. Types: Classes, structures and enumerated types

Rules	Examples
The name starts with an uppercase letter and has an uppercase letter for each new word, with no underscores The name reflects the logical entity for which it provides the definition.	**H**ero **E**nemy
Is **singular** when the class describes **one** entity (note that we can make more objects of one class).	Class, which contains - among others - a data member to hold the x position (m_X) and a data member to hold the y position (m_Y) of an enemy: **Enemy**
A class is **plural** if the class describes a collection of entities. This type of class often contains a container data member to keep track of those entities.	Class that manages the enemies in a game: **Enemies**

Enumerated types follow the same rules as classes.
Notice that the enumerators themselves start with lower case.
	enum class **LightState**
	{
	 on, off, defect
	};

e. Namespaces

Start with lower case letter, e.g. dae, utils,…

3.1.3. Naming files

Header files have the extension .h, implementation files the extension .cpp.

The files have the same name as the containing entity definition.

3.2. Format and appearance of code

You can customize the code formatting in Visual Studio via Tools/Options/Text Editor/C++

Imposing constraints on the format makes code easier to read due to consistency in form and appearance.

1. Code lines should not be too long, because long lines can be difficult to read and understand.
2. A pair of curly braces should be in the same column.
3. Each expression statement should be on a separate line.
4. All indentations should be consistent.
5. The statements forming the body of an if, else if, else, while, do… while or for statement shall always be enclosed in braces {}, even if they contain only one statement.
6. The public, protected and private sections of a class will be declared in that order (public, protected, private), because this is the order from most to least general interest.
7. All operators should be enclosed by spaces.

   `a+=b;` $\longrightarrow$ `a += b;`

8. The operands of a logical && or || shall be parenthesized if the operands contain binary operators. Binary operators are operators with 2 operands, for example the less than operator: a < b

   `a < b && c < d` $\longrightarrow$ `( a < b ) && ( c < d )`

3.3. Comments

Don't exaggerate, code already tells us how something is done, so don't repeat this in comments. Your code should be self-documenting. If you feel your code is too complex to understand without comments, your code is probably just bad. In this case rewrite it until it doesn't need comments anymore. Herewith consider using better identifier names or splitting up methods that do several things in smaller ones each performing a well-defined small task. If, at the end of that rewriting effort, you still feel that comments are necessary, then you can add comments.

4. Programming guidelines

4.1. Classes

The class design must be simple. Do not create complex class hierarchies.

The class relations (composition, association, aggregation, inheritance) are carefully chosen, e.g. inheritance has to applied only in case of "is a" relationship.

Classes should have clearly defined responsibilities, for example don't draw the level image in the avatar's Draw method.

4.2. Constructors – destructors

A constructor fully initializes the object and makes it ready to use.

Initialization of non-static member variables will be done in the **initializer list** rather than through assignment in the body of the constructor, unless it cannot be initialized by a simple expression. Don't initialize the member variables in the header file.

Constructors are explicit whenever appropriate.

Every object created on the heap should be deleted to avoid having memory leaks. There should be a clear notion of ownership of objects.

4.3. Inheritance

Public inheritance should only be applied for "**is a**" relationships and when run-time selection of implementation (polymorphism) is required.

An inherited non-virtual method should not be redefined in a derived class.

Pure virtual functions have the **pure-specifier** (=0).

All functions that support dynamic binding have the **virtual** specifier.

All functions that override an inherited virtual function specify this by using the **override** keyword.

Overridden functions should also be marked as virtual.

Classes that are not designed to be base classes should have the final specifier.

Base class destructors must be virtual, unless the class is marked as final.

4.4. Access control

Only allow public access to methods when it is absolutely necessary. Anything that doesn't fit this criterion should be private. For example, helper methods - used for splitting up a complex method in smaller ones - should be private.

Make all data members private and provide access to them through accessor methods as needed. Exceptions:

- Data members of an aggregate (struct) may be public (e.g. the x and y data members of a Point2f struct)
- Static const data members when information is needed in other classes.
- Protected access specifier for data members in base classes, when derived classes need this information.

4.5. Methods or functions

Method definitions should be put in the implementation file of the class (cpp) and not in the header file unless it is technically impossible (e.g. templates).

Avoid huge methods by splitting up the functionality in smaller helper methods.

Methods should not have too many parameters because this is difficult to read.

Overloaded methods should have the same purpose and only differ by the parameters.

Small parameters should be passed by value if changes made to them should not reflect in the calling code.

There should be no dead code in the methods. Dead code is code that is never executed. Unused methods should be removed.

Large parameters should be passed by reference. Indicate with **const** that a reference parameter is not changed by the method.

4.6. Const correctness

Const allows you to make it clear that something should not be changed. It gives you the ability to document your program more clearly and actually enforces that documentation.

Use const to indicate that:

- Member functions don’t change the state of an object.
- Parameters passed by reference are not changed by the method.
- Variables don’t change.

4.7. Variables and constants

There should be no unused variables in the code.

Always initialize variables.

```
char a{ 'A' };
int number{ 42 };
```

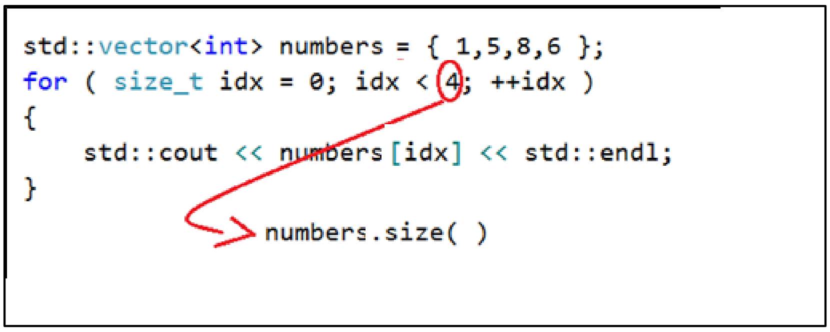
Don’t use the same variable to accomplish different purposes.

If a variable never changes its value, make it const.

Use an array or another container instead of many variables of the same type.

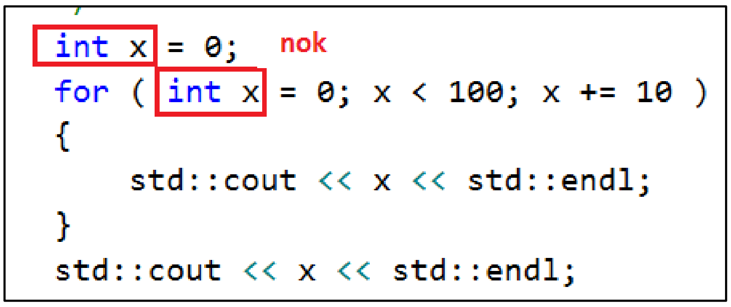
Class **enumeration** types shall be used instead of integer types or constants to select from a limited series of possibilities.

Numeric values – magic numbers - shall not be used in the code; rather enumerator, const values, size()... should be used.



```
std::vector<int> numbers = { 1,5,8,6 };
for ( size_t idx = 0; idx < 4; ++idx )
{
    std::cout << numbers[idx] << std::endl;
}
```

Variables in an inner scope should not use the same name as a variable in an outer scope, and therefore hide that variable because this can be very confusing.



```
int x = 0;
for ( int x = 0; x < 100; x += 10 )
{
    std::cout << x << std::endl;
}
std::cout << x << std::endl;
```

Variable declarations should be at the smallest feasible scope.

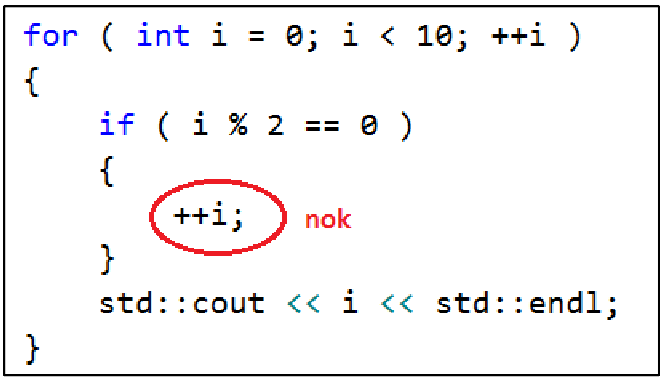
4.8. Flow control

There should be no unreachable code.

There will be no **goto** statements because they make the code both difficult to read and maintain.

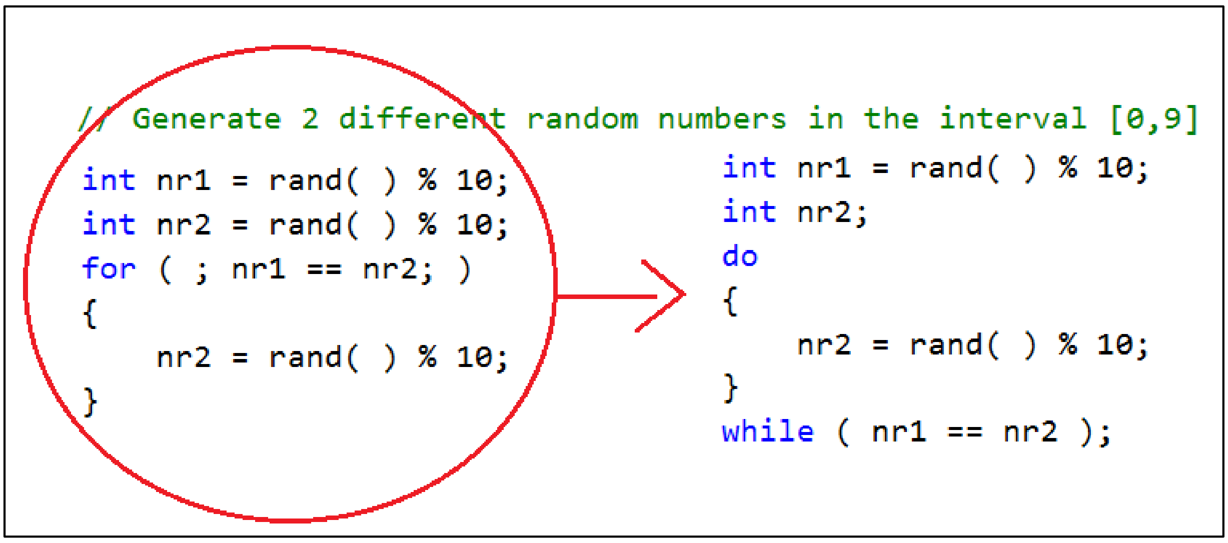
Prefer a **switch** statement to test the content of an **enumerated** type against the possible values.

Variables used within a **for-loop** for iteration counting, should not be modified in the body of the loop.



```
for ( int i = 0; i < 10; ++i )
{
    if ( i % 2 == 0 )
    {
        ++i;
    }
    std::cout << i << std::endl;
}
```

For-loops without initialize and increment expressions will not be used; a **while-loop** should be used instead.



```
// Generate 2 different random numbers in the interval [0,9]
int nr1 = rand( ) % 10;
int nr2 = rand( ) % 10;
for ( ; nr1 == nr2; )
{
    nr2 = rand( ) % 10;
}
```

```
int nr1 = rand( ) % 10;
int nr2;
do
{
    nr2 = rand( ) % 10;
}
while ( nr1 == nr2 );
```

4.9. Use the standard library functionality

Use the functionality of the standard library whenever appropriate. Use the right **containers** and **algorithms**.

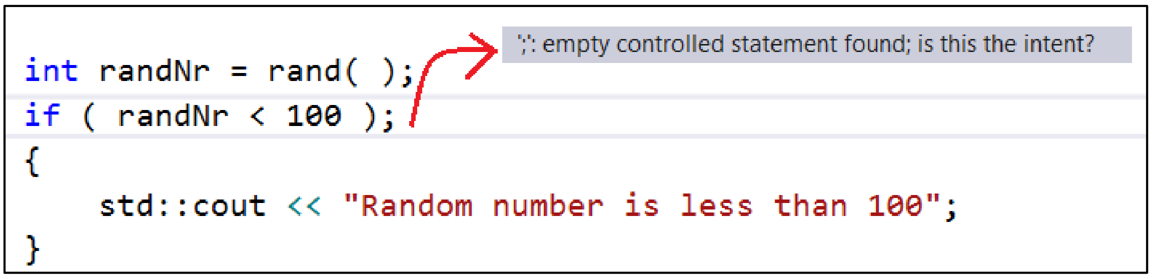
Use **lambdas** instead of functors whenever appropriate, they can make the code easier to read and maintain.

4.10. Warnings

It goes without saying that the code should build without errors.

There should be no level 3 warnings because warnings point out potential problems.

Example of a warning that is a problem:



```
int randNr = rand( );
if ( randNr < 100 );
{
    std::cout << "Random number is less than 100";
}
```

';': empty controlled statement found; is this the intent?