

C/C++ Language Syntax, Formatting, Compartmentalization, Style Advice and General Guidelines

C/C++ language usage and syntax:

Use platform-agnostic, ISO-endorsed, C++ syntax and STL libraries to ensure multi-platform compatibility. New features from C++11/14/17 which should be used for easier code readability and maintenance that we could use:

01. Usage of **'final'** keyword preventing inheritance
02. Usage of standard function creation/deletion:
 - = **delete**
 - = **default**
03. Usage of **'override'** keyword
04. Usage of **'nullptr'** keyword instead of C NULL macro
05. Usage of **move constructors** and **move assignments**
06. Usage of **range-based (foreach)** loops, ie **for (auto& value : container)**
07. Usage of the **'auto'** and **'auto&'** keywords
08. Usage of emplacement instead of insertion (emplace/emplace_back vs direct assignment/push_back)
09. Usage of class variable initialization in header files
10. Usage of **vector.data()** instead of &vector[0]
11. Usage of **array<T, N>** instead of C-style array T[N]
12. usage of **make_unique<[]>** instead of C-style array T* = new[N] on the heap
13. Usage of **unordered_map** associative containers
14. Usage of the **noexcept** keyword
15. Usage of **tuple & tie()** (instead of get<n>) when appropriate
16. **Threads & atomics API** usage through a new CPU thread N-CP multicore **'CUPParallelism'** library with a concurrent blocking queue and thread pool implementation underneath, with a CSharp/Java-style efficient **parallelFor()** using lambdas
17. **<filesystem>** library usage from C++17 specs for file I/O
18. Usage of the utils C++11/14/17 based library namespace with these internal namespaces:
 - using namespace utils::AccurateTimers;**
 - using namespace utils::CUPParallelism;**
 - using namespace utils::Randomizers;**
 - using namespace utils::SIMDVectorizations;**
 - using namespace utils::UtilityFunctions;**

C/C++ Formatting and Code Compartmentalization:

We will follow the generic Linux way of writing code to be multi-platform friendly. Please make sure to set up your favourite IDE (Visual Studio, Eclipse, Notepad++, etc) with these guidelines:

01. Usage of whitespace instead of tabs for multi-platform friendliness and compatibility with various IDEs.

02. Usage of two (2) whitespace characters for tab indentation for control blocks, functions, structs, classes and namespaces. Examples:

```
void Function()  
{  
    while (cin >> word)  
    {  
        if (word.length() > 2)  
        {  
        }  
    }  
}
```

```
class Class1  
{  
    class Class2  
    {  
    };  
};
```

```
namespace Namespace1  
{  
    namespace Namespace2  
    {  
    }  
}
```

03. Place 'else' on a new line:

```
if (a < b)  
{  
}  
else  
{  
}
```

04. Separate C++ statements (if, for, while, switch, etc) from their enclosing parentheses with one (1) whitespace:

```
if (a < b)  
{  
}  
else if (a == b)  
{  
}  
else // if (a > b)  
{
```

```
}
```

05. Use C++ namespaces extensively to compartmentalize the C++ codebase. Use the C#/Java *package* standard of having a *namespace* as a directory/folder structure with the same name and putting all namespace relevant code inside that:

Directory/folder structure:

utils -> AccurateTimers.h

utils -> CPUParallelism -> CPUParallelismNCP.h

utils -> Randomizers.h

utils -> SIMDVectorizations.h

utils -> UtilityFunctions.h

Namespace usage in C++ code:

```
using namespace utils::AccurateTimers;
```

```
using namespace utils::CPUParallelism;
```

```
using namespace utils::Randomizers;
```

```
using namespace utils::SIMDVectorizations;
```

```
using namespace utils::UtilityFunctions;
```

C/C++ Style and general advice:

01. Use the camel notation for C++ code, with extra whitespace alignment for easy eye pattern recognition of code:

```
namespace utils
```

```
{
```

```
namespace AccurateTimers
```

```
{
```

```
struct AccurateTimerInterface
```

```
{
```

```
virtual void startTimer() = 0;
```

```
virtual void stopTimer() = 0;
```

```
virtual double getElapsedTimeInNanoSecs() = 0;
```

```
virtual double getElapsedTimeInMicroSecs() = 0;
```

```
virtual double getElapsedTimeInMilliSecs() = 0;
```

```
virtual double getElapsedTimeInSecs() = 0;
```

```
virtual double getMeanTimeInNanoSecs() = 0;
```

```
virtual double getMeanTimeInMicroSecs() = 0;
```

```
virtual double getMeanTimeInMilliSecs() = 0;
```

```
virtual double getMeanTimeInSecs() = 0;
```

```
};
```

```
}
```

02. Try to minimize usage of **#include** statements in header files. Double check if these **#include** statements are really necessary (maybe already included somewhere else?). Use forward declarations for pointer and reference type if possible. Keep in mind that an **#include** statement physically opens a file from disk for reading, ie involves I/O, thus it can become time-consuming for a compiler to load, especially for very big projects.

03. Never use a **'using namespace XYZ'** statement within a header file.

04. Use an **#include guard** or a **#pragma once** for header files. In fact, to ensure maximum compatibility between multiple platforms and compilers, use both:

#pragma once

```
#ifndef __CPUParallelismUnitTests_h
#define __CPUParallelismUnitTests_h
```

```
namespace utils
{
    namespace CPUParallelism
    {
        class CPUParallelismUnitTests final
        {
        public:
            ....
        private:
            ....

        };
    }
}
```

```
#endif // __CPUParallelismUnitTests_h
```

05. Use the **anonymous namespace syntax** (as per Stroustrup's advice) for cpp compilation unit locality, instead of C's **static** keyword. As a consequence, the **static** keyword should only be used in the strict C#/Java OOP sense, ie inside classes' variables and functions definitions:

```
// in C++ cpp (compilation unit) file
namespace // anonymous namespace instead of deprecated 'static' keyword used for cpp variable locality
{
    const int someVariable = 0;
    ....
    void someFunction(...)
    {
        ....
    }
}
```

06. Inside a `cpp` (compilation unit) file use directly `std` (and any other heavily used namespaces) with a **'using namespace std'**. Avoid code pollution of `std::copy()`, `std::vector<T>` etc, syntax, which is more difficult to easily read and maintain:

```
// inside AccurateTimers.cpp
```

```
using namespace std;  
using namespace std::chrono;  
using namespace utils::AccurateTimers;  
.....  
....  
....  
long long AccurateCPUtimer::getMillisecondsTimeSinceEpoch()  
{  
    return duration_cast<milliseconds>(high_resolution_clock::now().time_since_epoch()).count();  
}
```