C++ coding guideline for $cuDFNsys$

Tingchang YIN
Zhejiang University
Westlake University

April 5, 2022

Contents

1 File structures

Let us denote the first level file by *, second level by + and third level by ^. Hierarchy of $cuMechcsysDFN$ should be as follows

```
* include
    + HeaderFile1
        ^ Foo1.cuh
    + HeaderFile2
        ^ Foo2.cuh
* src
    + srcFile1
        ^ Foo1.cu
    + srcFile2
        ^ Foo2.cu
* sandbox1
    + src
        ^ main.cu
    + bin
        ^ main
    + build
        ^ makefile
    + CMakeLists.cmake
    + CompileCode.sh
* sandbox2
    + src
        ^ main.cu
    + bin
        ^ main
    + build
        ^ makefile
    + CMakeLists.cmake
    + CompileCode.sh
```

2 Namespace

Namespace is $cuDFNsys$.

3 Separating source members

Definitions and implementations of classes/structs/functions should be contained in separated files (i.e. header and source files). All words of the names of files should begin with a **"capital"** letter.For example,

- <FractureRadius.cuh>
- <FractureRadius.cu>

```
//<FractureRadius.cuh>
#pragma once
```

```
#include <iostream>
using namespace std;
class FractureRadius
{
    public:
        float Radius;
    public:
        FractureRadius(const uint i);
        float SecondMoments();
};



//<FractureRadius.cu>
#include "FractureRadius.cuh"
FractureRadius::FractureRadius(const uint i)
{
    this->Radius = i * 10;
};

float FractureRadius::SecondMoments()
{
    return pow(this->Radius, 2);
};
```

4 Standard comment headers

In header files, we should include the following information

```
////////////////////////////////////////////////////////////////////
// NAME:                FooBar.h
//
// PURPOSE:             Definition of FooBar class. This class is
//                      responsible for doing FooBar stuff.
//
// FUNCTIONS/OBJECTS:   CFooBar
//
// AUTHOR:              James
////////////////////////////////////////////////////////////////////
```

5 Class/struct/function naming

All words of a class/struct begin with a **capital** letter. For example:

```
class Animal;
struct DriverJames;
```

The naming of a function (including a member function of a class) should be based on the **purpose** or the **method**. Again, all words begin with a **capital** letter. For example,

```
float GetFractureTag(const uint i);
// based on the purpose
void ConjugateGradient();
// based on the method
```

6 Class member organization

For example:

```
class Foo : public FooBase
{
    // methods
    //-----------------------------------
    public :
    protected :
    private :
    // attributes
    //-----------------------------------
    public :
    protected :
    private :
};
```

7 Data variable naming

All words begin with a **capital** letter. For example,

```
class Foo
{
public:
    int             FooTag;
    string          FooName;
    bool            FooState;
};
```

When calling a member data variable in a class member function, "this->" should be always used. For example

```
this->Var1 = 1;
```

For naming global variables related to tolerance, the format is

```
#define _TOL_IdentifyEleNO 1e-7
___device___ const float _TOL_DetectIntersection = 1e-7;
```

8 Member descriptive comments

All member data and functions should be prefaced with single line comments describing the purpose of the member. For example:

```
class Foo
{
public:
    // the tag of the Foo
    uint Tag;

public:
    // delete the tag of the foo
    void DeleteTag();
};
```

9 Descriptive comment of a function

In source (.cpp) files, a function should be supplemented by detailed descriptive comments. For example:

```
// ==========================================================
// NAME         : GetName
//
// DESCRIPTION : Accessor for the name of this object.
// AUTHOR       : DEV1
// DATE         : 10/10/99
// ==========================================================
void Foo::GetName( CString& strName )
{
    strName = this->Name;
} // GetName
```

Also, all functions should include a trailing comment just after the closing brace of the function scope that identifies the function name.

10 About inline functions

Inline function may increase efficiency if it is **small**.

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

- If a function contains a loop. (for, while, do-while)
- If a function contains static variables.
- If a function is recursive.
- If a function return type is other than void, and the return statement doesn't exist in function body.
- If a function contains switch or goto statement.