

Langage C++

Coding Standard

Sylvain GROSDÉMOUGE



Définitions

Coding Standard
CamelCase

Nommage

Implémentation

Commentaires

Coding standard - Définitions

Notion de 'Coding standard'

Ensemble de règles à suivre pour :

- uniformiser le code
- faciliter sa maintenance
- diffuser les bonnes pratiques de développement
- éviter les erreurs de "classiques"

Définitions

Coding Standard

CamelCase

Nommage

Implémentation

Commentaires

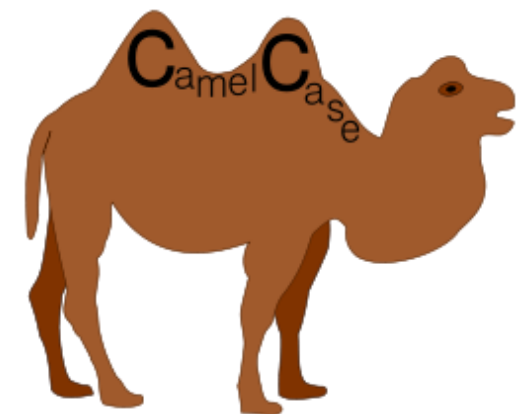
Coding standard - Définitions

Notion de notation 'CamelCase'

Ensemble de mots qui se succèdent et qui commence chacun par une majuscule.

Exemple :

CeciEstUnExempleDeNommageEnCamelCase



Définitions

Nommage

Constantes

Macros

Variables

Enumérations

Méthodes

Structures

Classes

Fichiers

Décorateurs

Et plus généralement...

Implémentation

Commentaires

Nommage des constantes :

Nom : Majuscules + chiffres + caractère '_' uniquement

Exemples : `#define MAX_PATH 256`

`#define SHC_PI 3.141592``#define SHC_EPSILON 10e-6f``#define USE_STL 0`

Définitions

Nommage

Constantes

Macros

Variables

Enumérations

Méthodes

Structures

Classes

Fichiers

Décorateurs

Et plus généralement...

Implémentation

Commentaires

Nommage des macros :

Nom : Majuscules + chiffres + caractère ‘_’ uniquement

Exemples : `#define SH_ISPOWEROF2(x) (!((x) & ((x) - 1)))`

Définitions

Nommage

Constantes

Macros

Variables

Enumérations

Méthodes

Structures

Classes

Fichiers

Décorateurs

Et plus généralement...

Implémentation

Commentaires

Nommage des variables :

Nom : CamelCase + commencent par une minuscule
+ préfixes pour identifier les variables qui ont
une sémantique particulière :

‘p’ devant les pointeurs

‘s’ devant les statiques

‘sz’ devant les strings se terminant par ‘0’

‘m_’ devant les variables internes des classes

Exemples :

```
int          value = 10;

int *        pBuffer = NULL;

static int    sDebugFrameCounter = 0;

const char *  szString = "This is a string";

const char *  m_szString;  ///< in a class.
```

Définitions

Nommage

Constantes

Macros

Variables

Enumérations

Méthodes

Structures

Classes

Fichiers

Décorateurs

Et plus généralement...

Implémentation

Commentaires

Nommage des énumérations :

Type :	CamelCase + commencent par un 'E'
Enum :	Minuscule + chiffres + '_' uniquement + commencent par 'e_'
Exemples :	<pre>enum EPadButton { e_pad_button_a, e_pad_button_b, e_pad_button_x, e_pad_button_y, ... };</pre>

Définitions

Nommage

Constantes

Macros

Variables

Enumérations

Méthodes

Structures

Classes

Fichiers

Décorateurs

Et plus généralement...

Implémentation

Commentaires

Nommage des méthodes :

Nom : CamelCase
(void) dans les declarations
() dans les appels

Exemples :

```
void    MyMethod                (void) ;  
  
void    InitializeSystemConstants (void) ;  
  
MyMethod() ;  
InitializeSystemConstants() ;
```

(void) / () → Permet de facilement différencier les appels des declarations lors d'une recherche...

Définitions

Nommage

Constantes

Macros

Variables

Enumérations

Méthodes

Structures

Classes

Fichiers

Décorateurs

Et plus généralement...

Implémentation

Commentaires

Nommage des structures :

Nom : CamelCase
+ commencent par un 'S'

Exemples :

```
struct SSimpleStructure
{
    ...
};
```

[Définitions](#)[Nommage](#)[Constantes](#)[Macros](#)[Variables](#)[Enumérations](#)[Méthodes](#)[Structures](#)[Classes](#)[Fichiers](#)[Décorateurs](#)[Et plus généralement...](#)[Implémentation](#)[Commentaires](#)

Nommage des classes :

Nom : CamelCase
+ commencent par un 'C'

Héritage : Nommage par ordre décroissant de parenté

Exemples :

```
class CVehicle
{
    ...
};

Class CVehicleCar : public CVehicle
{
    ...
}
```

[Définitions](#)[Nommage](#)[Constantes](#)[Macros](#)[Variables](#)[Enumérations](#)[Méthodes](#)[Structures](#)[Classes](#)[Fichiers](#)[Décorateurs](#)[Et plus généralement...](#)[Implémentation](#)[Commentaires](#)

Nommage des fichiers :

Règle : Une classe = 1 fichier .cpp et 1 fichier .h associé

Les fichiers portent le nom de la classe qu'ils contiennent

Exemple :

- `CVehicle.cpp`
- `CVehicle.h`
- `CVehicleCar.cpp`
- `CVehicleCar.h`
- `CVehicleTruck.cpp`
- `CVehicleTruck.h`

Définitions

Nommage

Constantes

Macros

Variables

Enumérations

Méthodes

Structures

Classes

Fichiers

Décorateurs

Et plus généralement...

Implémentation

Commentaires

Nommage des décorateurs:

Exemple : "CClass.h" :

```
#ifndef __CCLASS_H
#define __CCLASS_H

Class CClass
{
    // ...
};

#endif // __CCLASS_H
```

→ Evite au pré-processeur de ré-inclure des declarations.

[Définitions](#)[Nommage](#)[Constantes](#)[Macros](#)[Variables](#)[Enumérations](#)[Méthodes](#)[Structures](#)[Classes](#)[Fichiers](#)[Décorateurs](#)[Et plus généralement...](#)[Implémentation](#)[Commentaires](#)

Et plus généralement :

Donnez des noms de variables explicites !

Exemples :

```
for (int i = 0 ; i < nb_fa ; ++i)  
{  
    fa[i]-> ...  
}
```

```
for (int nFacade = 0 ; nFacade < nbFacades ; ++nFacade)  
{  
    pFacades[nFacade]-> ...  
}
```

Déclaration des variables :

Toujours dans le scope le plus bas.

```
char c = '-';  
for (int i = 0 ; i < nb ; ++i)  
{  
    while (c != 'o' && c != 'n')  
    {  
        scanf("%c", &c);  
    }  
}
```

```
for (int i = 0 ; i < nb ; ++i)  
{  
    char c = '-';  
    while (c != 'o' && c != 'n')  
    {  
        scanf("%c", &c);  
    }  
}
```

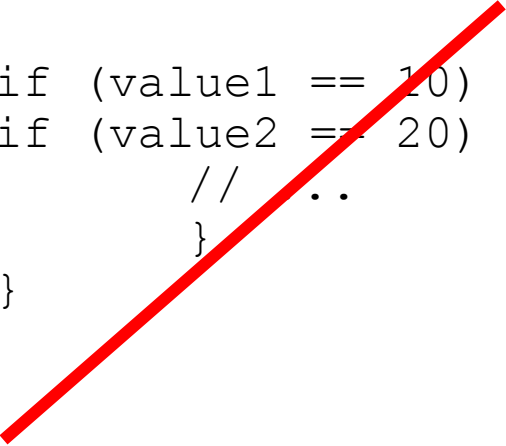
Dans l'exemple de gauche, que se passe-t-il sur la seconde iteration de la boucle ?

Alignement des { } :

Retour à la ligne devant chaque '{'

```
if (value1 == 10)
{
    if (value2 == 20)    vs.
    {
        // ...
    }
}
else
{
}
}
```

```
if (value1 == 10) {
if (value2 == 20) {
    // ..
}
}
```



switch / case :

Règle : Un switch/case contient TOUJOURS un 'default'.
Les blocs de code sont TOUJOURS entourés de { }
Attention à ne pas oublier un 'break'...

Exemple :

```
switch (eType)
{
case e_type_int:
    ...
    }
    break;

case e_type_float:
    ...
    }
    break;

default:
    ...
    }
    break;
}
```


Comparaison de nombres flottants

Règle : Ne jamais comparer deux nombres flottants pour une égalité.

Toujours utiliser une notion de precision !

Exemple :

```
if (f == 0.0f)  
{  
    // ...  
}
```

```
if (FEQUAL(f, 0.0f))  
{  
    // ...  
}
```

avec :

```
#define FCMPPRECISION    0.0001f  
#define FEQUAL(f, v)    fabsf(f-v) < FCMPPRECISION
```

Constructeurs

Règle : Déclarer les constructeurs explicites le plus souvent possible.

Exemple :

```
CClass (int i),
```

```
CClass c(1) → OK
```

```
CClass c(1.2f) → OK
```

Dans ce cas, 1.2f est converti implicitement en int, et le constructeur est appelé avec le paramètre '1'

```
explicit      CClass (int i)
```

```
CClass c(1) → OK
```

```
CClass c(1.2f) → Compile error !
```

→ Evite les conversions de type implicites !

Méthodes

Règles : Implémentées dans les .cpp qui correspondent au .h de declaration
Implémentées dans l'ordre de declaration
Rappel des spécificités (explicit, virtual, static, ...)

Exemple :

```
/*explicit*/ CVector2::CVector2(void)
{
    // ...
}

/*virtual*/ CEntity::~~CEntity(void)
{
    // ...
}

/*static*/ int CObject::GetInstanceCount(void)
{
    // ...
}
```

Constructeurs

Règle : Dans l'implémentation, initialiser les variables dans l'ordre de déclaration, avec une variable par ligne, et les ':' et ';' en début de ligne.

Exemple :

```
/*explicit*/ CClassBig::CClassBig(void)
: m_pObject1(NULL)
, m_pObject2(NULL)
//, m_deprecatedValue(2)
{

}
```

Plus lisible qu'avec les ';' à la fin

Plus rapide lorsqu'on veut commenter / supprimer une variable

Destructeurs

Règle : Déclarer les destructeurs virtual (toujours).

Exemple :

```
class CBase
{
public:
    CBase(void) { printf("CBase Constructor\n"); }
    ~CBase(void) { printf("CBase Destructor\n"); }
};

class CDerived : public CBase
{
public:
    CDerived(void) { printf("CDerived constructor\n"); }
    ~CDerived(void) { printf("CDerived destructor\n"); }
};

int main()
{
    Cbase * pBase = new CDerived;
    delete pBase;
}
```

→

```
CBase Constructor
CDerived constructor
CBase Destructor
```

ATTENTION ! Le destructeur de CDerived n'est pas appelé !

Destructeurs

Règle : Déclarer les destructeurs virtual (toujours).

Exemple :

```
class CBase
{
public:
    CBase(void) { printf("CBase Constructor\n"); }
    virtual ~CBase(void) { printf("CBase Destructor\n"); }
};

class CDerived : public CBase
{
public:
    CDerived(void) { printf("CDerived constructor\n"); }
    virtual ~CDerived(void) { printf("Cderived destructor\n"); }
};

int main()
{
    Cbase * pBase = new CDerived;
    delete pBase;
}
```

→

```
Base Constructor Called
Derived constructor called
Derived destructor called
Base Destructor called
```

C'EST BEAUCOUP MIEUX !

Tests d'égalité :

Quel est le mieux ?

```
if (ptr == NULL)           ou           if (NULL == ptr)           ?  
{                               {                                 
    // ...                   // ...  
}
```

If (NULL == ptr) permet de prévenir les erreurs suivantes :

```
if (ptr == NULL)           → OK
```

```
if (ptr = NULL)           → OK
```

(mais dans les faits, ptr prends la valeur NULL et le test est toujours valide !)

Mieux :

```
if (NULL = ptr)           → Erreur de compilation !
```

```
if (NULL == ptr)          → OK
```

De même que :

```
if (NULL != ptr) plutôt que if (ptr != NULL)
```

Notion d'assertion :

Syntaxe :

```
assert(condition);
```

Génère une erreur lors de l'exécution lorsque la condition n'est pas respectée

Exemple :

```
#include <assert.h>
```

```
int i = 5;
```

```
assert(i > 0);
```

 → aucun effet

```
assert(i == 5);
```

 → aucun effet

```
assert(i != 5);
```

 → erreur lors de l'exécution

```
assert(i < 0);
```

 → erreur lors de l'exécution

Utilisation :

```
void myMethod(const char * szString)
{
    assert(NULL != szString);

    // ...
}
```


SAFE_DELETE / SAFE_DELETE_ARRAY :

Définition :

```
#define SAFE_DELETE_ARRAY(ptr)      \  
if (shNULL != ptr)                 \  
{                                  \  
    delete [] ptr;                 \  
    ptr = shNULL;                   \  
}  
  
#define      SAFE_DELETE(ptr)      \  
if (shNULL != ptr)                 \  
{                                  \  
    delete ptr;                     \  
    ptr = shNULL;                   \  
}
```

Utilisation :

```
int * ptr = new int;  
// ...  
SAFE_DELETE(ptr);  
  
int * ptr = new int[256];  
// ...  
SAFE_DELETE_ARRAY(ptr);
```

Optimisation des boucles :

Problème :

```
for (int i = 0 ; i < GetNumberOfItems() ; ++i)
{
    ...
}
```

→ `GetNumberOfItems()` est appelé sur chaque iteration de la boucle (!!!)

Mieux :

```
const int numberOfItems = GetNumberOfItems();
for (int i = 0 ; i < numberOfItems ; ++i)
{
    ...
}
```

Commentaire des constantes / macros / variables globales :

Exemple :

```
/// This macro returns true if x is POT, false if not.  
SH_ISPOWEROF2(x)          (!((x) & ((x) - 1)))  
  
/// Temporary. @todo Remove this !  
int g_debugFrameCounter = 0;
```

Commentaire des énumérations :

Exemple :

```
/// This is an enumeration
enum EEnumeration
{
    e_enumeration_0,      ///< Comment 0
    e_enumeration_1,      ///< Comment 1
    e_enumeration_2,      ///< Comment 2
    e_enumeration_3,      ///< Comment 3
};
```

Commentaire des classes :

Exemple :

CVector2.h

```
//
///< This class represents a vector in 2d space.
class CVector2
{
public:
    explicit          CVector2          (float x, float y);

    ...

protected:

private:
    float    m_x;    ///< Component on the X axis.
    float    m_y;    ///< Component on the Y axis.
};
```

CVector2.cpp

```
///<
///< CVector2 constructor with components initialization.
///< @param x Component on the X axis.
///< @param y Component on the Y axis.
///<
/*explicit*/ CVector2::CVector2(float x, float y)
: m_x(x)
, m_y(y)
{

}

...
```