

Standards de développement

INGEDATA

| | |
|------------------|--|
| Document | Documentation sur les standards de développement |
| Référence | Standard développement codes - INGEDATA.doc |
| Version | Version 2.1 |
| Rédacteur | ENGINEERING INGEDATA |
| Date | 11/08/2008 |

Documentation sur les standards de développement INGEDATA

| | | |
|-----|--|----|
| 1 | Introduction..... | 3 |
| 2 | Les variables | 4 |
| 2.1 | VARIABLES GLOBALES | 4 |
| 2.2 | VARIABLES LOCALES..... | 4 |
| 2.3 | PARAMETRES | 4 |
| 2.4 | LES MEMBRES DES CLASSES | 4 |
| 3 | Les types | 5 |
| 4 | Constantes | 6 |
| 5 | Fonctions..... | 7 |
| 5.1 | LES FONCTIONS MEMBRES D'UNE CLASSE : | 7 |
| 5.2 | LES FONCTIONS GLOBALES OU STATIQUES | 7 |
| 5.3 | LES NOMS DES FONCTIONS | 7 |
| 6 | Modules | 8 |
| 7 | L'indentation | 9 |
| 8 | Commentaires | 10 |
| 8.1 | ENTETE DE FICHIER | 10 |
| 8.2 | VARIABLES..... | 10 |
| 8.3 | FONCTION..... | 10 |
| 8.4 | CLASSE | 11 |
| 8.5 | CONSTRUCTEUR..... | 11 |

1 Introduction

Il est important de suivre des standards de programmation afin que les codes soient plus faciles à lire et ainsi plus faciles à maintenir.

Avoir des codes standards consistent à :

- Bien nommer les modules, les variables, les types, les fonctions, etc... de telle façon à ce qu'on puisse reconnaître de ces noms leurs significations, il faut pour cela utiliser des noms pertinents tout en évitant d'utiliser des noms excessivement longs
- Bien documenter les codes afin de comprendre ce qu'ils font avant de lire les lignes de codes
- Bien aérer les codes en utilisant les indentations

Les standards décrits dans cette documentation sont des standards de base, à utiliser pour n'importe quel langage de programmation.

2 Les variables

La lettre initiale est une minuscule et les noms composés se voient séparés par des soulignés. Ainsi, *variable* et *ma_variable_quelconque* sont des noms de variables acceptables.

2.1 Variables globales

Les variables globales : précédées d'un **g_**

Ex : `string g_strConnection` ;

Les variables globales comme des constantes (elles sont assignées au moment du lancement du programme et ne sont jamais modifiées) : précédées d'un **G_**

Ex : `int G_iMax = 100` ;

2.2 Variables locales

Les variables locales ne devraient pas être nommées inutilement. Il serait plutôt superflu de changer le nom de la variable du genre i ou j (int i ou int j). Évitez les noms inutilement longs dans ces situations.

Ex : `int i` ;

2.3 Paramètres

Le nom des paramètres définissent en fait des variables locales à la fonction. Ils devraient obéir aux mêmes règles.

2.4 Les membres des classes

Les variables membres : précédées d'un **m_**

Bien qu'en principe, aucun data member devrait être d'accès public (il y a des accesseurs qui seront prévus à cet effet, du genre `getX()` et `setX(...)`).

Ex : `private string m_strNom`;

3 Les types

On aborde seulement ici les types de bases :

Entier : les variables de type entier sont précédées de **i**

Ex: `int g_iMax ;`

Ce qui veut dire que `g_iMax` est une variable globale de type entier.

Long : les variables de type long sont précédées de **l**

Ex: `long lLargeur ;`

Ce qui veut dire que `lLargeur` est une variable locale de type long.

Double : les variables de type double sont précédées de **d**

Ex: `double G_dPi = 3.14 ;`

Ce qui veut dire que `dRatio` est une variable globale constante de type double.

Float : les variables de type float sont précédées de **f**

Ex: `float gRatio ;`

Ce qui veut dire que `fRatio` est une variable locale de type float.

String : les variables de type string sont précédées de **str**

Ex: `string m_strNom ;`

Ce qui veut dire que `m_strNom` est une variable membre de type string.

Booléen : les variables de type booléen sont précédées de **b**

Ex: `bool bOK ;`

Ce qui veut dire que `bOK` est une variable booléenne.

Structure des noms et des noms composés :

Les noms devraient commencer par une majuscule (après le symbole modifiant) et s'il s'agit d'un nom composé, avoir une majuscule à chaque frontière de mot.

Ex: `oList`, `tMachinTruc`.

Objets :

Les noms d'objet devraient être précédés d'un **o** minuscule.

Ex: `MaClasse oMaClasse ;`

Structs :

Les noms de struct devraient être précédés d'un **s** minuscule.

Ex: `struct ListItem`, `sListItem`.

Enums :

Les types enums devraient avoir leurs noms précédés d'un **e**.

Ex: `typedef enum { E_MODE_READ, E_MODE_WRITE } eMode;`

Pointeurs & Références :

Les types pointeurs sont prefixés d'un **p**

Exemple : `pList`

4 Constantes

Une constante devrait avoir son nom tout en majuscules, et, s'il s'agit d'un nom composé, des soulignés devraient venir séparer les mots.

Ex : PI et RACINE_DE_DEUX

5 Fonctions

5.1 Les fonctions membres d'une classe :

Les fonctions-membre commenceront par une minuscule initiale, puis auront des majuscules aux frontières de mots, comme `.machin` et `.machinTruc`.

Ex :

```
fonction maFonction()  
fonction getNom()
```

5.2 Les fonctions globales ou statiques

Les noms de fonctions globales commenceront par une majuscule; et auront une majuscule aux frontières de mots.

Ex :

```
fonction MachinTruc()  
fonction GetPrenom()
```

5.3 Les noms des fonctions

Le choix du nom de la fonction est crucial.

Les fonctions qui testent pour une propriété devraient commencer par « Is » :

Ex : `IsConnected()`

Les fonctions qui vont chercher une donnée ou le prochain item à traiter commenceront par « Get » ou « get » selon le cas ou c'est une fonction membre ou fonction globale.

Ex :

```
Fonctions globales : getNext(), getTime( )  
Fonctions member d'une classe : getPays()
```

Les fonctions qui chargent et sauvegardent peuvent avoir des noms qui commencent par « Load » et « Save ».

Ex : `LoadGraph()`, `SaveConfig()`

6 Modules

Les modules forment des unités naturelles dans un programme. Il faut bien séparer les fonctions du même type d'actions dans un module à part.

Il ne faut pas hésiter à donner des noms significatifs, voire même longs aux fichiers.
Il faut aussi organiser les fichiers en les classant dans des répertoires et/ou sous-répertoires afin de s'y retrouver facilement. N'hésitez pas non plus à créer des arbres de répertoires pour mieux structurer vos modules.

7 L'indentation

L'indentation aussi a pour but de rendre le code plus aisément lisible.

L'indentation devra permettre déterminer rapidement:

Le scope des while, do, for, if.

Le scope de ces boucles ou if devrait être facilement identifiable.

Ex :

```
while (cond)
{
    ...
} // while (cond)
```

On fait mieux de commenter la fin } fermant le while (ou du for, ou du do ...while) afin de rappeler la condition du while (ou du for, ou du...). C'est d'autant plus utile si la fonction est un peu longue et que tout le code de la boucle n'est pas visible simultanément.

Ex :

```
while (cond)
{
    if (cond)
    {
        // then-code
        .....
    } else {
        // else-code
        .....
    } // if (cond)
} // while (cond)
```

La longueur des espaces d'une indentation est de 4 caractères ou un tab de 4 espaces.

Les définitions de classes ou de structures.

8 Commentaires

Les commentaires sont très importants pour la lisibilité des codes autant pour celui qui produit les codes que ceux qui les modifieront à l'avenir.

Il est donc primordial d'être pertinents et explicites dans ses commentaires, tout en évitant bien sûr d'écrire un romain et de faire des commentaires inutiles.

Voici les types de commentaires qu'il faut nécessairement mettre dans ses codes :

8.1 Entête de fichier

L'entete de chaque fichier contiendra des commentaires succincts sur le fichier. Ils donneront les informations suivantes :

- nom du fichier
- description de son contenu
- le ou les auteur(s)
- la date de création du fichier

Ex :

```

////////////////////////////////////
// Ftp.c
//
// Traitement des échanges FTP
//
// Auteur(s) : A. Blabla
// Date de création : 25/07/2006
////////////////////////////////////

```

8.2 Variables

Il ne faut donner de brève description qu'aux variables non évidentes.

Ex :

string strPrenom ; (plus besoin de description, vu que la variable parle d'elle-même)

8.3 Fonction

La fonction doit être commentée avec les informations suivantes :

- nom de la fonction
- sa description
- le retour de la fonction
- ses arguments détaillés
- ses auteurs : les personnes ayant modifié la fonction

Ex :

```

////////////////////////////////////
// Nom de la fonction   : FTPProcessRecv
// Description          : Fonction qui traite les fichiers reçus par FTP
// Return type          : int : pour savoir si le traitement s'est bien passé
// Argument             : CODBCDynamic* podbcDynamic : pointeur de la base de données
// Auteurs              :
//   - A. Blabla (29/10/2007)
//   - B Bloblo (03/12/2007)
//   Corrections du bug machin
////////////////////////////////////
int FTPProcessRecv(CODBCDynamic* podbcDynamic)
{
    TCHAR          szUAFtpOut[_MAX_PATH + 1];
    TCHAR          szBuffer[512];
    .....
    return 0;
}

```

8.4 Classe

Il faut indiquer le nom de la classe et sa description

Ex :

```
////////////////////////////////////  
// Nom de la classe      : Clients  
// Description          : Classe pour manipuler les clients  
////////////////////////////////////  
public class Clients : Object  
{  
    .....  
}
```

8.5 Constructeur

Il faut indiquer la description et les arguments d'un constructeur

Ex :

```
////////////////////////////////////  
// Description          : Constructeur de la classe Clients  
// Argument            :  
////////////////////////////////////  
public Clients()  
{  
    .....  
}
```