Normes de programmation
En C++

Document conçu dans le cadre du cours
IFT-19946 Programmation orientée objet

Automne 2002

Buts des règles de programmation

Définir des règles permettant d'écrire des programmes qui sont:

- corrects
- avec un style consistent
- faciles à lire et à comprendre
- exempts des types d'erreurs communes
- faciles à maintenir par différents programmeurs.

Le langage C++ est un langage difficile à comprendre et à maîtriser. Le programmeur a une très grande responsabilité et il doit se discipliner. Comme avec le C, le C++ permet d'écrire du code compact mais illisible. Le programmeur doit donc suivre les règles décrites dans ce document dans le but d'éviter ces embûches.

[**R**] Chaque fois qu'une règle est violée, la raison doit être clairement documentée.

Gestion des fichiers

[**R**] Les classes devront être écrites dans des fichiers portant le nom de la classe.
`<NomClasse>.<extension>`

[**R**] Les classes de test devront être écrites dans des fichiers portant le nom de la classe + «Testeur».
`<NomClasse+Testeur>.<extension>`

Les fichiers source dans lesquels le code sera écrit doivent être disposés de la façon suivante :

Fichier d'interface	<NomClasse>.h
Fichier d'implantation	<NomClasse>.cpp
Fichier pour les méthodes paramétrisées (template)	<NomClasse>.hpp

Exemple pour une classe Usager :

```
Usager.h           // Interface de la classe
Usager.cpp         // Implantation de la classe
UsagerTesteur.cpp  // Fichier de test unitaire pour Usager.
```

[**R**] Une seule classe par fichier (comprenant les fichiers .h, .cpp, .hpp).

[**R**] Le fichier d'interface (.h) doit être protégé contre la multiple inclusion
Exemple :

```
#ifndef NOMCLASSE_H_DEJA_INCLU
#define NOMCLASSE_H_DEJA_INCLU
...
// interface de la classe
#endif // #define NOMCLASSE_H_DEJA_INCLU
```

[**R**] Ne **jamais** spécifier de chemin d'accès au complet dans les directives `#include`.
Exemple :

```
// NON!
#include "c:/dev/include/NomClasse.h"
// OK!
#include "NomClasse.h"
```

C'est dans l'environnement de compilation que vous devez spécifier où aller chercher les fichiers d'inclusion.

[**R**] Utiliser la directive `#include "NomClasse.h"` pour les classes préparées par le développeur.

[**R**] Utiliser la directive `#include <NomClasse>` pour les classes provenant de librairie standard, pas de .h dans le nom.

[**R**] Lorsque les définitions suivantes sont utilisées dans l'interface de la classe (.h), elles doivent être **incluses** comme un fichier d'interface séparé :

- La classe de base (s'il y a lieu)
- Les classes utilisées comme attributs
- Les classes qui apparaissent comme types de retour ou paramètres des méthodes.

Mise en forme du code source

Les commentaires

Il y a deux façon d'insérer des commentaires en C++.

1. Les caractères «//» indiquent un début de commentaire et ce, jusqu'à la fin de la ligne où ils se trouvent.

   Exemple:

   ```
   main()
   {
      // --- ceci est un commentaire
      char a; // --- ceci est aussi un commentaire
   }

   //*******************************************
   //              ceci est un bloc
   //           servant de commentaire
   //*******************************************
   ```

2. Pour mettre en commentaire plusieurs lignes, on se sert de:
   `/*` pour commencer le commentaire et `*/` pour le terminer.

   Exemple :

   ```
   /* char a; // --- C'est du code invisible pour le
   compilateur
      on est toujours dans le commentaire
      il se termine ici. */

   ATTENTION:  Pas de commentaire imbriqué.
               Erreur de compilation.

   /* commentaire
   ```

```
/* commentaire */   // --- Erreur de compilation
   commentaire */
```

[**R**] Favoriser l'utilisation de «//» dans les commentaires.

L'indentation et l'imbrication

[**R**] Le code doit être entièrement indenté de manière cohérente pour augmenter la visibilité et la compréhensibilité.

L'***indentation*** permet de visualiser la structure d'un programme. Une indentation trop faible nuit à la lisibilité, tandis qu'une indentation trop grande décale de façon inutile tout le code vers la droite.

[**R**] La norme sera donc une indentation de **trois espaces**. Ne pas utiliser de tabulateur. Remplacer par 3 trois espaces pour garantir l'uniformité de la présentation peut importe l'éditeur de texte.

L'***imbrication*** poursuit le même but que l'indentation, soit d'augmenter la lisibilité du code.
L'imbrication de début de bloc doit être placée à la ligne suivant la condition, et à la même colonne que la condition.

```
if (a > 0)
{
   // --- les crochets au même niveau
   // --- indentation de 3 espaces
   b = 0;
}
```

La nomenclature

Nom de classe et de structure

[**R**] Le nom d'une classe ou d'une structure commence toujours par une lettre majuscule. Si le nom est un nom composé, les autres mots commenceront aussi par une lettre majuscule.

```
Message
SystemeMessagerie
```

Nom de méthode

[**R**] Les noms des méthodes doivent commencer par une lettre minuscule, sauf pour les constructeurs et le destructeur dont le nom est identique au nom de la classe.
Lorsqu'un nom de méthode est formée par un mot composé, la première lettre des autres mots sont en majuscule.
Autant que possible, un nom de méthode ou de fonction doit être de la forme <verbe+ complément> et exprimer clairement ce que fait la méthode ou la fonction.

```
envoisMessage
```

```
recoisMessage
asgDateHeure
```

Quelques règles pour les noms de méthode

[**R**] Il faut standardiser le nom des méthodes et de fonctions usuels fréquemment rencontrées. Utiliser la version **française**.

Voici un tableau qui présente les versions anglaises et françaises avec exemples.

Préfixe anglais	**Préfixe français**	**Exemple**
create	cree	creeMessage()
destroy	detruit	detruitMessage()
init	init	initSysteme()
terminate	termine	terminateProcessus()
on	on	on()
off	off	off()
alloc	alloue	alloueEspace()
free	libere	libereEspace()
send	envoie	envoieMessage()
receive	recois	recoisMessage()
read	lit	litDonnees()
write	ecrit	ecritDonnees()
open	ouvre	ouvrePortes()
close	ferme	fermePortes()
status	etat	etatSysteme()
control	controle	controleZoom()
next	suivant	suivantMessage()
previous	precedent	precedentMessage()
up	haut	haut()
down	bas	bas()
stop	arrete	arreteExecution()
go	demarre	demarreExecution()
get	req	reqMessage()
is	est	estValide()
set	asg	asgMessage()
put	met	metCharactere()
do	fait	faitRechercheUsager()
find	cherche	chercheMessage()
check	verifie	verifieConnexion()
add	ajoute	ajouteMessage()
delete	detruit	detruitMessage()
empty	vide	videConteneur()
edit	edite	editeMessage()
clear	efface	effaceParametres()

select	selectionne	selectionneItem()
import	importe	importeMessagerie()
export	exporte	exporte Messagerie()

```
void asgCeci   ();
Cela reqCela   ();
bool estValide ();
```

Nom d'attribut

[**R**] Les noms d'attributs doivent commencer par une lettre minuscule et lorsqu'un attribut est composé de plus d'un mot, chaque première lettre des autres mots sont en lettres majuscules.

[**R**] L'utilisation du préfixe «m_» pour les noms d'attribut est fortement suggéré.

```
int       m_jour;
DateHeure m_uneDateQuelconque;
```

[**R**] On devra de plus mettre à la fin d'une variable de type pointeur la lettre majuscule **P** pour signaler l'utilisation d'un pointeur.

```
char*     m_chaineP;
```

Nom de constante

[**R**] Les noms de constantes sont toujours définies en lettres majuscules. Les mots composant le nom doivent être séparés par des soulignés.

```
const int VALIDE = 1;
enum Couleur {BLEU,BLANC,ROUGE};
const double NIVEAU_CRITIQUE = 4.0;
```

[**R**] Les constantes doivent toujours être définies avec **const** ou **enum**; ne jamais utiliser **#define**.
Exemples :

```
#define PI (3.1416)       // Mauvais
const double PI = 3.1416; // Meilleur

#define BLEU   (0)
#define BLANC  (1)
#define ROUGE  (2)               // Mauvais
enum Couleur {BLEU,BLANC,ROUGE}; // Meilleur
```

[**R**] Ne jamais utiliser de nombres «magiques» dans le code.
Exemple :

```
// Mauvais
if (niveau >= 3.567)
{
   ...
}
// Meilleur
```

```
const double NIVEAU_CRITIQUE = 3.567;
if (niveau >= NIVEAU_CRITIQUE)
{
   ...
}
```

Les compteurs

[**R**] Les variables utilisées comme compteurs de boucle sont les lettres `i,j,k,l,m`, ... et elles sont définies au moment de leur utilisation uniquement.

```
for (int i=0; i<10; i++)
{
   for (int j=0; j<20; j++)
   {
      ... // --- Code à exécuter
   }
}
```

Le choix des noms

[**R**] Il faut porter une attention particulière au choix des noms de méthodes, d'attributs et de variables pour éviter les ambiguïtés.

```
int           longueurNom;    // à la place de longNm
PrinterStatus etatImprimante; // à la place de etImp

void processTerm ();  // process terminé ou process du
terminal ?
```

[**R**] Éviter les noms avec des caractères numériques pouvant porter à confusion.

```
int IO = 13;
int I0 = IO;
```

[**R**] La longueur d'une ligne de code ne doit pas dépasser la limite considérée comme maximale. On dit parfois que 80 caractères est une limite pour s'assurer d'éditer et d'imprimer sans problème. On peut toutefois aller jusqu'à 100 caractères dans trop de problèmes.

La définition et l'utilisation des variables

[**R**] Définir les variables le plus près possible de leur utilisation.

[**R**] Définir une seule variable par énoncé de déclaration.

[**R**] Une variable doit être initialisée avant d'être utilisée et si possible, préférer l'initialisation à l'assignation.

Exemples :

```
char* strP, str; // Mauvais
char* strP = 0;  // Meilleur
char  str;       // Meilleur

int i;
```

```
... // un paquet de lignes de code
i = 10;     // Mauvais

int i = 10; // Meilleur
```

Les fonctions

La surcharge de fonction

Le langage C++ permet de créer plusieurs fonctions du même nom mais avec des paramètres différents.

Lorsqu'une fonction «**surchargée**» est appelée, le compilateur sélectionne la bonne fonction selon les arguments: nombre, type, ordre.

On se sert de la surcharge de fonctions ayant le même nom lorsqu'elles effectuent des tâches similaires mais sur des données différentes.

Attention: le compilateur utilise seulement la liste des paramètres pour distinguer les fonctions de même nom. Le type de retour n'est pas pris en compte.

```
int    print (int, int);
double print (int, int); // Les deux fonctions sont identiques
                         // pour le compilateur.
```

Exemples corrects:

```
void ajoute (const Usager& usager);
void ajoute (const Message& msg);
void ajoute (const string& texte);
```

Général

[**R**] Éviter les longues et complexes fonctions.

Les longues fonctions ont plusieurs désavantages :

- Lorsqu'une fonction est trop longue, il est très difficile de comprendre ce qu'elle fait. De façon générale, une fonction ne devrait pas dépasser une ou deux pages.
- Si une situation d'erreur se produit à la fin d'une longue fonction, il est extrêmement difficile de faire l'annulation de ce qui a déjà été effectué. En utilisant toujours de courtes fonctions, les erreurs sont faciles à localiser et à traiter.
- Une fonction complexe est difficile à tester ! Si une fonction est constituée d'une quinzaine d'énoncés imbriqués, vous devrez tester (2^{15} → 32768) différentes branches pour une simple fonction !!!

Les énoncés de contrôle

Les énoncés «if» et «if-else»

L'énoncé `if` teste une condition particulière et lorsque celle-ci est vraie, on exécute une certaine partie de code. Autrement, cette partie est ignorée.

[**R**] Ne pas faire **if (test)** si test est un pointeur. Plutôt, faire **if (test != 0)**.

```
if (condition)
{
   foo();
}

if (condition)
{
   foo();
}
else
{
   boo();
}
```

Les énoncés «switch»

L'énoncé **switch** est surtout utilisé pour remplacer plusieurs niveaux de **if-else-if** imbriqués. Le résultat de l'expression du **switch** est comparé avec une liste de valeurs constantes.

[**R**] Un énoncé switch doit toujours contenir une branche **default** qui tient compte des cas non prévus.

[**R**] Le code qui suit chaque **case** doit toujours se terminer par un énoncé **break**.

```
char c = 'a';
switch(c)
{
case 'a':   // c == 'a'
   ...
   break;
case 'z';   // c == 'z'
   ...
   break;
default:    // c == autre chose
   ...
   break;
}
```

Les boucles «for»

Une boucle «for» est généralement utilisée lorsque la variable d'incrément est augmentée par une quantité constante à chaque itération et lorsque la fin de la boucle est déterminé
par une expression constante.

L'énoncé itératif **for** est conçu pour faciliter la manipulation d'une boucle itérative. La syntaxe est la suivante:

```
for (expr1; expr2; expr3)
   bloc;
```

- La première expression (expr1) est exécutée au début de la boucle seulement. Elle initialise le compteur.
- La seconde expression (expr2) est la condition de boucle. La boucle continue tant et aussi longtemps que cette expression donne un résultat non-nul ou non-faux.
- La troisième expression (expr3) est exécutée à la toute fin de chaque itération de la boucle. Il est garanti que cette expression ne sera pas exécutée si la condition de boucle (expr2) est fausse dès la première itération.

[**R**] Toujours utiliser les limites inférieures inclusivement et les limites supérieures exclusivement.

[**R**] Toujours définir les variables de boucle à l'intérieur de l'énoncé **for**, maintenant la variable définit sera locale à la boucle.

[**R**] Utiliser des variables non signées lorsque la variable ne peut pas avoir de valeurs négatives.

```
int a[10];

for (unsigned int i=0; i<10; i++)  // Meilleur
{
   a[i] = 0;
}

int i;
for (i=0; i<=9; i++)  // Moins bon
{
   a[i] = 0;
}
```

Les boucles «while» et «do-while»

Lorsque la condition de sortie doit être évaluée au début de la boucle, la boucle «while». Lorsque la condition de sortie doit être évaluée à la fin de la boucle, la boucle «do-while».

L'énoncé de contrôle **while** permet d'exécuter une boucle tant que l'expression logique résulte en une valeur non nulle (autre que zéro ou faux). Dès que l'évaluation de l'expression est zéro, la bouche se termine et le traitement continue après le bloc de la boucle.
L'énoncé de boucle **do-while** est utilisé dans le cas où l'on veut qu'une première itération de boucle soit exécutée de façon inconditionnelle.

[**R**] Toujours choisir la boucle appropriée en fonction de leur utilisation.

```
int i=0;
while (i<10)
{
   ...
```

```
   i++;
}
// ici, i == 10;

int i = 10;
do
{
   ...
   i--;
} while( i > 0);
// ici, i == 0;
```

Les classes

Division en packages

La division du code en packages permet de limiter les liens entre les classes et de permettre des designs plus intéressants. Certains langages comme le Java supporte directement la notion de package dans le langage en obligeant le programmeur à s'insérer dans un package.

En C++, il fut ajouté un nouveau mot-clé et un support du langage pour tenter de gérer cette problématique que l'on pourrait nommer <<la problème de pollution de l'espace de nom global>>. Il s'agit de l'utilisation de **namespace**.

[R] Toujours insérer vos classes, vos fonctions et variables dans un **namespace** défini par votre design. Mots en minuscules séparés par des soulignés. Ne pas faire d'indentation.

[R] Nommer le **namespace** d'un nom significatif selon le design.

```
namespace msg
{
class Messagerie
{
};
}
```

[R] Toutes les classes d'un même package devrait partager le même namespace.

```
namespace msg
{
class Messagerie {…};
}
namespace msg
{
class Usager {…};
}
(…)
```

[R] Ne pas faire de **using** dans l'interface d'une classe (.h). Plutôt directement nommer le **namespace** dans les signatures de méthodes. Faites le **using** dans l'implantation (.cpp)

```
#ifndef USAGER_H_DEJA_INCLU
#define USAGER_H_DEJA_INCLU

#include <string>

namespace msg
{
class Usager
{
public:
   void        asgNom                (const std::string& nom);
   void        asgMotDePasse         (const std::string&
motDePasse);
   std::string reqNom                () const;
   std::string reqMotDePasse         () const;

   static bool valideReglesNom       (const std::string& nom);
   static bool valideReglesMotDePasse (const std::string&
motDePasse);
private:
   std::string m_nom;
   std::string m_motDePasse;
};
}
#endif // USAGER_H_DEJA_INCLUS

// usager.cpp

#include "usager.h"
using namespace std;
namespace msg
{

void Usager::asgNom (const string& nom)
{
   PRECONDITION (Usager::valideReglesNom(nom));
   m_nom = nom;
}
void Usager::asgMotDePasse (const string& motDePasse)
{
   PRECONDITION (Usager::valideReglesMotDePasse(motDePasse));
   m_motDePasse = motDePasse;
}
(…)
}
```

Notez que la librairie standard du C++ a toute été définie dans le namespace **std**.

Considérations à propos des droits d'accès (encapsulation)

[**R**] Ne **jamais** définir des attributs dans la section publique de la classe. Les déclarer dans la section privée et définir des méthodes d'accès.

L'emploi d'attributs publiques est découragés pour les raisons suivantes :

- Un attribut publique viole le principe de base de la programmation orientée objet, i.e. l'encapsulation des données. Lorsqu'on laisse les données publiques, n'importe qui peut modifier celles-ci. Par contre, si on déclare les données privées, seulement les méthodes de la classe pourront modifier ces données.
- Une quelconque fonction dans un programme peut modifier les données publiques ce qui peut mener à des erreurs difficiles à localiser.
- L'encapsulation va permettre la stabilité des interfaces sans nécessairement empêcher la modification des structures de données cachées derrière celles-ci.

Mauvais exemple :

```
Class Date
{
public:
   int m_jour;
   int m_mois;
   int m_annee;
};
```

Meilleur exemple :

```
Class Date
{
public:
   void asgDate (int jour, int mois, int annee);
   int  reqJour () const;
   int  reqMois () const;
   int  reqAnnee() const;
private:
   int m_jour;
   int m_mois;
   int m_annee;
};
```

Exemple d'une interface stable malgré les changements aux données :

```
Class Date
{
public:
   void asgDate (int jour, int mois, int annee);
   int  reqJour () const;
   int  reqMois () const;
   int  reqAnnee() const;
private:
   long m_nbSecondeDepuis1970;
};
```

[**R**] Les sections publiques, protégées et privées doivent être déclarées dans cet ordre:

Exemple :

```
class MaClasse
{
public:
   // --- Méthodes publiques
protected:
   // --- Méthodes protégées
private:
   // --- Attributs et méthodes privées
};
```

[**R**] Il ne devrait y avoir qu'une seule section publique, protégée, et privée.

Exemple :

```
class MaClasse
{
public:
   // --- Quelques méthodes publiques
private:
   // --- Attributs et méthodes privées
public:
   // --- Quelques autres méthodes NON!
};
```

[**R**] Une méthode publique ne doit jamais retourner une référence ou un pointeur non constant sur un attribut de la classe.

Méthodes constantes

[**R**] Une méthode qui n'affecte pas l'état d'une classe doit être déclarée **const**. Cette règle doit être ***strictement*** respectée.

```
Class Date
{
public:
   void asgDate (int jour, int mois, int annee);
   int  reqJour () const;
   int  reqMois () const;
   int  reqAnnee() const;
private:
   int m_jour;
   int m_mois;
   int m_annee;
};
```

[**R**] Il est possible de définir deux méthodes distinctes dans l'interface de classe : une constante et une autre non-constante. Le mot-clé **const** fait partie de l'interface. Idéal pour des conteneurs pour ne pas cesser la «chaîne de constance».

Exemple :

```
class vecteur
{
```

```
public:
   Item&       obtientItem (int indice);
   const Item& obtientItem (int indice) const;
};
void main()
{
   // Soit un vecteur rempli d'item…

   // Cet appel est résolu par la version non constante.
   // On pourra faire des appels non constants sur itemNC
   Item& itemNC = vecteur.obtientItem(0);

   // Cet appel est résolu par la version constante.
   // On ne pourra faire que des appels constants sur itemC
   const Item& itemC = vecteur.obtientItem(2);
}
```

Méthodes optimisées (inline)

[**R**] Ne **jamais** définir une méthode à l'intérieur d'un fichier d'interface, i.e. pas de code. La raison principale est que vous ne pouvez documenter cette méthode, (voir la section documentation).

Mauvais exemple :

```
class Date
{
public:
   void asgDate (int jour, int mois, int annee);
   int  reqJour () const {return m_jour;}
   int  reqMois () const {return m_mois;}
   int  reqAnnee() const {return m_annee;}
private:
   int m_jour;
   int m_mois;
   int m_annee;
};
```

Meilleur exemple :

```
class Date
{
public:
   void asgDate (int jour, int mois, int annee);
   int  reqJour () const;
   int  reqMois () const;
   int  reqAnnee() const;
private:
   int m_jour;
   int m_mois;
   int m_annee;
};

inline int Date::reqJour () const
{
   return m_jour;
}
inline int Date::reqMois () const
{
   return m_mois;
```

```
}
inline int Date::reqAnnee() const
{
   return m_annee;
}
```

[**R**] Les méthodes d'accès devraient être optimisées.

[**R**] Les méthodes qui redirigent l'appel (forwarding functions) devraient être optimisées.

[**R**] **Ne jamais optimiser** les constructeurs et les destructeurs.

La raison normale pour déclarer une fonction optimisée (inline) est d'augmenter la performance. Cette fonctionnalité a été ajoutée pour répondre aux critiques qui voulaient que ce soit ridicule de mettre les données privées et de faire des méthodes d'accès pour chaque attribut dans les cas simples car un appel de fonction coûte cher. La fonction inline fait en sorte que le compilateur remplace l'appel de la méthode optimisée par le code équivalent directement dans la méthode appelante.

Les méthodes optimisées permettent alors d'avoir le meilleur des deux mondes :

- Données protégées (attributs privés)
- Accès à ces données sans coûts supplémentaires.

[**R**] Si une méthode optimisée (inline) doit inclure un fichier pour réaliser l'implantation, toujours la mettre dans l'implantation normal (non inline) dans le fichier .cpp.

[**R**] La méthode optimisée ne doit pas être complexe, seulement accès, redirection, assignation simple.

Passage de paramètres

[**R**] Il faut minimiser le nombre d'objets temporaires qui peuvent être créés lorsqu'une fonction retourne une valeur ou reçoit un paramètre. Une construction pour l'objet est appelée lors du passage (ou du retour) et une destruction est appelée lors de la sortie du «scope».

[**R**] Éviter le passage de paramètre par valeur. Utiliser le passage de paramètre par référence ***constante***.

En C++, il n'est plus nécessaire de passer des classes par valeur pour s'assurer que celles-ci ne soient pas modifiées. La passage par référence constante permet d'obtenir exactement le même résultat sans le coût de la copie.

Mauvais exemple :

```
class MaClasse
{
public:
   void nomMethode(MaClasse uneClasse); //Non!
```

```
};
```

Meilleur exemple :

```
class MaClasse
{
public:
   void nomMethode(const MaClasse& uneClasse);
};
```

Méthodes statiques

[R] Si une méthode de classe ne nécessite pas la présence d'un objet pour le traitement, c'est qu'elle doit être une méthode statique. Une méthode statique est comme une fonction mais associée à la classe.

```
class Usager
{
public:
   static bool valideReglesNom (const String& nom);
};

bool Usager::valideReglesNom (const String& nom)
{
   bool bResultat = false;
   // --- La méthode fait des tests qui n'ont rien à voir
   // --- avec un objet Usager.  Par contre , la validation
   // --- du nom concerne cette classe.
   …
   return bResultat;
}
```

[R] Toujours appeler une méthode statique avec le nom de la classe et non pas par un objet de la classe – bien que ce soit permis. Cela rend l'appel plus clair; il s'agit nécessairement d'une méthode statique.

```
void main()
{
   Usager unUsager; // création d'un objet Usager
   bool bRes = false;

   // --- Appel par un objet [A éviter]
   bRes = unUsager.valideReglesNom ("unNomQuelconque");
   // --- Appel par le nom de la classe [A favoriser]
   bRes = Usager::valideReglesNom ("unNomQuelconque");
}
```

Documentation du code source

Voici les exemples de documentation qui peuvent être mis en place. Ces entêtes doivent être toujours bien documentées.

Fichier d'interface de la classe (.h)

```
//*****************************************************************************
// Fichier:     Pile.h
//
// Classe:      Pile
//
// Sommaire:    Classe implantant le concept de pile.
//
// Description: La classe Pile permet d'implanter le concept de pile de réels
//              On y implante le type de données abstrait traditionnel avec
//              les fonctionnalités courantes, soient l'initialisation de la
//              pile, requête pour savoir si la pile est vide ou pleine,
//              l'ajout d'un élément, le retrait d'un élément, et obtenir
//              l'élément sur le dessus de la pile.
//
// Attributs:   deque<double> pile: un vector STL servant à implanter la pile.
//              int capacite : la capacité de la pile.
//
// Note:        La pile n'est conçu que pour les réels.  On rendra la classe
//              paramétrisable sur le type lorsqu'on verra les templates.
//*****************************************************************************
// 10-09-99  Yves Roy  Version initiale
// 06-10-99  Yves Roy  Ajout de la notion de capacité et utilisation de STL
//*****************************************************************************
```

Fichier d'implantation de la classe (.cpp ou .hpp)

```
//*****************************************************************************
// Fichier:     Pile.cpp
// Classe:      Pile
//*****************************************************************************
// 10-09-99  Yves Roy  Version initiale
// 06-10-99  Yves Roy  Ajout de la notion de capacité et utilisation de STL
//*****************************************************************************
```

Fichier des méthodes normales ou optimisées

```
//*****************************************************************************
// Sommaire:     Retourne l'élément sur le dessus de la pile.
//
// Description:  La méthode <code>pop()</code> permet d'obtenir l'élément du
//               dessus de la pile.
//
// Entrée:       <nil>
//
// Sortie:       Retourne un double correspondant au dessus de la pile.
//
// Notes:        Cette méthode ne doit jamais être appelée si la pile est vide.
//*****************************************************************************
```