C++ coding guide

Guide de développement logiciel C++

Timothée Royer

Version : 1997/02/25.

Résumé

La forme du code

La structuration d'un projet

L'algorithmique

Abstract

Sommaire

Chapitre 1 - Présentation du guide - Preface

It said : 'The history of every major Galactic civilization tends to pass through three distinct and recognizable phases, those of Survival, Inquiry and Sophistication, otherwise known as the How, Why and Where phases.'
The Hitch Hiker's Guide to the Galaxy
Douglas Adams

Introduction

La mise au point d'un programme informatique est un domaine mal maîtrisé où les dépassements de temps de mise au point et de budget sont presque systématiques. C'est à ce problème que nous souhaitons nous attaquer au travers de ce document. Le développement logiciel au sens strict se découpe en trois étapes : analyse, conception et développement. Les méthodes de mise en oeuvre des deux premières étapes sont bien établies. Ce n'est pas le cas du codage : les recommandations de développement logiciel sont rares et peu utilisées dans l'industrie. Nous souhaitons combler cette lacune. Elle peut être due :

- À l'anxiété des informaticiens de voir leur créativité inhibée ;
- À la difficulté de consigner méthodiquement des erreurs et des propositions de solution générales (ce document est destiné à être amélioré par ses lecteurs, merci de vos retours d'information !) ;
- Au fait qu'être reconnu pour son savoir faire technique peut être incompatible avec une évolution de carrière optimale. Dans cette optique, il préférable de faire valoir ses capacités de synthèse en écrivant une méthode d'analyse.

C'est en tenant compte de ces raisons que nous avons établi ce document. Il s'adresse aux développeurs :

- Qui ont déjà pratiquer la programmation, mais souhaitent améliorer la qualité de leur travail du point de vue de l'implémentation (maintenance, réutilisabilité, portabilité...).
- Qui doivent travailler à plusieurs sur un même code. Dans cette situation, la formalisation du code devient déterminante : les coûts de maintenance dépassent les coûts de développements.

Nous avons consigné des règles :

A posteriori, après avoir découvert un « bug » et trouvé une méthode générale qui aurait permis d'éviter la difficulté. Il faut pas simplement se promettre d'être plus intelligent la prochaine fois.

En constatant que les développeurs experts se sont constitués un ensemble de règles. Elles se ressemblent en partie d'un développeur à l'autre. Ces règles ne peuvent se déduire de la lecture d'un manuel de référence d'un langage.

Unanimement, le langage C est considéré comme puissant mais difficile à mettre en oeuvre. En tant que surensemble du C, le langage C++ est à la fois plus puissant et plus difficile à employer, par les fonctionnalités supplémentaires qu'il apporte.

En tenant compte de l'état de l'art du langage. Celui-ci a beaucoup évolué et le développement industriel n'en est que plus difficile.

Ce document propose un ensemble de recommandations de codage souples et générales pour augmenter la qualité du code source. Il est en particulier destiné aux programmes écrits en C++. Une annexe propose une application de ce guide à la programmation en C. Cependant, une partie des argumentations doit être suffisamment générale pour s'appliquer à d'autres langages : une implémentation objet souple, maintenable et réutilisable correspond à un style et non à un langage.

Selon Dijkstra [Dijkstra 1972] : "As a slow witted human being I have a very small head and I had better learn to leave with it and to respect my limitations and give them full credit, rather than to try to ignore them, for the latter vain effort will be punished by failure". Le développement logiciel est un exercice intellectuel difficile. La taiile et la complexité des projets peut croître plus rapidement que les capacités intellectuelles des développeurs. Il faut rationnaliser l'implémentation. Nous vous proposons un outil pour en débroussailler la complexité.

FIXME Le cerveau humain peut travailler avec un nombre limité d'idées simultanées. Ce nombre est fixé à sept « plus ou moins deux » [Meyer ? ?]. Pourtant, c'est avec ces capacités intellectuelles limitées que des systèmes logiques énormes doivent être conçus, implémentés et maintenus. En particulier, les fonctionalités du C++ remplissent largement les besoins techniques nécessaires à l'élaboration d'un logiciel. Les difficultés de mise en oeuvre arrivent avec l'implémentation, lorsque l'ensemble des capacités du langage ne se représente pas par un modèle gérable par l'humain.

Application

Seule un guide appliqué est intéressant. Il faut plutôt considérer ce papier comme une base de travail qu'il faudra modifier, simplifier ou enrichir en fonction des idées qui viendront à l'usage.

La réalité de la programmation ne tient pas dans un cadre complet et rigide. Cependant, il y a beaucoup d'améliorations applicables systématiquement à un code source et elles ne sont pas toutes élémentaires. Il ne faudrait pas se priver de les inclure dans le guide à cause de quelques exceptions. Chacune des propositions est donc associée à l'un de ces trois niveaux d'exigence :

*** IMPÉRATIF ***
Ces règles sont indiscutablement communes à toutes les implémentations rigoureuses et efficaces.

*** RECOMMANDATION ***
Recommandée. La règle décrit comment résoudre une difficulté de manière systématique. Si la difficulté est bien maîtrisée par les programmeurs ils peuvent logiquement continuer d'appliquer leur ancienne méthode. Il faut simplement s'assurer que cette autre technique est bien applicable de manière systématique. Il suffit de préciser en une phrase la raison du choix. Une modification du guide peut être envisagée.

*** AMÉLIORATION ***
Proposée. Les règles qui semblent difficiles à décrire ou quantifier précisément, quelle que soit la raison, ne sont que proposées comme modèle de codage. De même, certaines améliorations

sophistiquées des recommandations ne sont proposées dans cette catégorie qu'à titre optionnel.

La seule prétention de ce document est d'aider à l'écriture du code, une fois l'analyse et la conception achevées. Un logiciel mal conçu tiendra difficilement dans un guide de programmation utile. Lorsque la conception ou l'implémentation se passent mal, il faut reprendre l'analyse. Le coût engendré sera toujours inférieur à celui de la maintenance d'un module mal écrit.

Conventions de présentation du guide

Les chapitres sont présentés dans un ordre d'abstraction croissant. Ils sont découpés logiquement en paragraphes regroupant quelques règles concernant un même domaine. Chacun regroupe quelques règles traitant d'un sujet précis.

D'une manière générale, la précision des exigences va dans un ordre croissant entre les chapitres et au sein des paragraphes.

Une recommandation peut se décomposer en sept parties. Seules les deux premières sont toujours présentes.

- Le degré de nécessité (impératif, recommandation, amélioration) et le numéro de la règle ;
- L'énoncé de la règle;
- (Pourquoi ?) La justification de la règle;
- (Rappel) Un rappel sur un point technique précis, utile pour comprendre la recommandation;
- (Comment ?) Explique comment faire pour respecter la règle, lorsque ce n'est pas évident;
- (Exception) Les exceptions à la règle;
- (Exemple) Un exemple illustrant la règle.

Attention. Les exemples illustrant les recommandations de ce document sont volontairement concis. Ils peuvent ne pas respecter complètement certaines recommandations de mise en forme du source, en particulier celles conçues pour des projets importants (commentaires standards, corps des méthodes en dehors des déclarations...).

Certaines notions de programmation à maîtriser sont décrites dans le lexique mis en annexe de ce document. Il est destiné à être enrichi en fonction des besoins des utilisateurs de ce guide.

Chapitre 2 - Mise en forme du code source

A program should be light and agile, its subroutines connected like a string of pearls. The spirit and intent of the program should be retained throughout. Their should be neither to o little nor too much, neither needless loops nor useless variables, neither lack of structure nor overwhelming rigidity.
A program should follow the "Law of Least Astonishment". What is that law ? It is simply that the program should always respond to the user in a way that astonishes him least.
A program, no matter how complex, should act as a single unit. The program should be directed by the logic within rather than by outward appearances.
If the program fails in these requirements, it will be in a state of disorder and confusion. The only way to correct it is to rewrite the program.
The Tao of programming.

Dans ce chapitre, nous allons présenter l'aspect "bas niveau" de ce guide : la disposition du texte du code source dans un fichier. Aucune de ces recommandations n'a de conséquence technique directe, pour le compilateur, par exemple.

Mise en page du code source

Dans les paragraphes qui vont suivre, nous allons présenter des recommandations de disposition des lignes de code dans un fichier.

Indentation

*** IMPÉRATIF ***

Le code doit être entièrement indenté de manière cohérente.

(Comment?) Le décalage vers la droite d'une ligne de code source doit être proportionnel à son niveau d'imbrication logique.

(Exemple) Voici une première possibilité :

FIXME // info gcc [Code qual 1994]

En voici une autre :

En revanche, ce code source est mal indenté :

*** RECOMMANDATION ***

La largeur d'une indentation peut être de 4 ou de 8 espaces. Utiliser si possible un caractère tabulation pour indenter.

(Exemple) Une méthode pratique consiste à redéfinir la largeur d'une tabulation à 4 espaces. Tous les éditeurs de texte contemporains le permettent. Exécuter «(setq tab-width 4)» sous emacs ou « :set shiftwidth=4» sous vi.

(Comment ?) Penser à remplacer chaque tabulation par le nombre d'espaces voulus avant de changer d'éditeur pour conserver la présentation.

*** RECOMMANDATION ***

Utiliser un indenteur automatique de code source dont les paramètres sont fixés pour le projet.

(Comment ?) Par exemple, emacs permet en standard d'indenter un code source C++, de manière paramètrable.

Présentation des accolades

Une paire d'accolades délimite un élément de structure.

*** IMPÉRATIF ***

Une accolade fermante se trouve toujours à la verticale de l'accolade ouvrante correspondante.

(Exemple) A ne pas faire :

```
int main(void)
{
      cout << "Les codes ASCII :" << endl;
      int col;
      for(int ligne = 0; ligne < 32; ligne++) {
            for(col = 0; col < 7; col++) {
                  cout << (col*7+ligne) << ' '
                        << char(col*7+ligne) << '\t';
            }
            cout << endl;
      }
}
```

Cette disposition permet de gagner deux lignes de code source. Elle fait cependant moins ressortir la structure logique du code. Elle n'est donc pas souhaitable. Le code source doit plutôt être présenté comme ceci :

```
int main(void)
{
      cout << "Les codes ASCII :" << endl;
      int col;
      for(int ligne = 0; ligne < 32; ligne++)
      {
            for(col = 0; col < 7; col++)
            {
                  cout << (col*7+ligne) << ' '
                        << char(col*7+ligne) << '\t';
            }
            cout << endl;
      }
}
```

*** IMPÉRATIF ***

La colonne dans laquelle se trouve une accolade est proportionnelle à son niveau d'imbrication.

*** IMPÉRATIF ***

Une structure de contrôle utilise toujours une paire d'accolades, même si elle est vide.

(Pourquoi?) Lorsque le corps d'une structure de contrôle ne contient qu'une expression, les accolades qui l'entourent peuvent être ommises. Ce n'est pourtant pas souhaitable. La présence systématique d'accolades facilite la lecture directe et la maintenance du code.

(Exmple) Voici ce qu'il ne faut pas faire :

```
int main(void)
{
	cout << "Les codes ASCII :" << endl;
	int col;
	for(int ligne = 0; ligne < 32; ligne++)
	{
		for(col = 0; col < 7; col++)
			cout << (col*7+ligne) << ' '
				<< char(col*7+ligne) << '\t';
		cout << endl;
	}
}
```

Se référer à l'exemple précédent pour une bonne méthode à appliquer.

*** IMPÉRATIF ***

Pas de caractère point virgule (;) après une accolade fermante délimitant une structure de contrôle.

(Rappel) Le point virgule n'est requis après une accolade fermante que lors d'une déclaration de struct, classe ou template.

(Pourquoi?) Des virgules et des points virgules peuvent être ajoutés un peu partout dans un code source sans modifier son exécution. Sa lisibilité est cependant altérée. Il faut prendre l'habitude de restreindre le syntaxiquement superflu.

(Exemple) Cette ligne de code est valide en c et en c++. Elle peut être ajoutée presque n'importe où. Elle gène seulement la lecture.

```
;;;;;;,,,,,,,;;;;;;;;
```

*** RECOMMANDATION ***

Une accolade et son contenu commencent sur la même colonne.

(Pourquoi ?) Lors de la production de code pour un gros projet, les règles de présentation doivent être unifiées, systématiques et simples. (FIXME)

(Exemple) Voici un programme compilable. Sa présentation respecte les conventions proposées ici, mais il faudrait ajouter des commentaires au début du fichier pour qu'il soit complet.

```
#include <iostream.h>
const char TAB = '\t';
int main(int argc, char** argv, char** env)
{
cout << "Les arguments de la ligne de commande :" << endl;
for(int argumentCounter = 0; argumentCounter < argc;
argumentCounter++)
	{
	cout << TAB << argv[argumentCounter] << endl;
	}
cout << endl;
cout << "Les variables d'environement :" << endl;
```

```
char** environementVariableScanner = env;
while(environementVariableScanner)
      {
      cout << TAB << *environementVariableScanner << endl;
      environementVariableScanner++;
      }
return 0;
}
```

Disposition de chaque ligne

*** IMPÉRATIF ***

Déclarer chaque variable séparément et non pas les unes à la suite des autres, séparées par des virgules.

(Pourquoi ?) Déclarer plusieurs variables simultanément est moins lisible et plus difficilement maintenable. Sans le souci de compatibilité avec le C, Bjarne Stroustrup n'aurait pas implémenté cette possibilité en C++ [Stroustrup 1995].

(Exemple) Exemple de ce qu'il ne faut pas faire et de ce qu'il faut faire :

```
// <:0
char* papaOurs,mamanOurs,boucleDOr = (char*)0;
// d8)
char* papaOurs;
char* mamanOurs;
char* boucleDOr = (char*)0;
```

Il y avait un piège dans cet exemple : ici, papaOurs est de type char* alors que mamanOurs et boucleDOr sont de type char et non pas de type char*. Bien sûr, ce genre de problème a une forte chance d'être détecté par le compilateur lors de l'utilisation de la variable. Cependant, cet exemple reste représentatif du manque de lisibilité de ce type de déclarations.

*** RECOMMANDATION ***

Le source ne doit pas contenir deux espaces consécutifs (hormis l'indentation), ni d'espace ou de tabulation à la fin d'une ligne.

(Pourquoi ?) Ceci est très pratique en particulier pour effectuer des remplacements de texte automatiques. Ceux-ci sont très utiles pour une séquence de mots, et la présentation doit alors être rigoureuse. Les remplacements, parfois nécessaires, peuvent devenir dramatiques si le code n'est pas bien présenté : les erreurs générées automatiquement ne sont pas toujours faciles à détecter et souvent laborieuses à corriger.

*** RECOMMANDATION ***

La largeur d'une ligne ne doit pas dépasser 80 caractères.

(Pourquoi ?) Ceci permet d'éditer et d'imprimer de manière cohérente un code source sur différents supports.

(Comment ?) Lorsqu'une ligne est trop longue, il faut la couper juste avant un opérateur qui sera dans le niveau de parenthèsage le plus externe possible. La suite de l'instruction sera mis sur une ou plusieurs lignes, toutes indentées une seule fois par rapport au début de l'instruction.

Structure des fichiers sources

Dans cette partie de chapitre, nous allons voir comment le code source se répartit entre les différents fichiers d'un programme.

Répartition du code entre les fichiers

*** RECOMMANDATION ***

Les fichiers sources contenant les déclarations ont pour extension «.hh» et les fichiers sources contenant les définitions ont pour extension «.cc».

(Exception) Certains compilateurs ne connaissent pas ces extensions standard. Il est alors possible d'utiliser «.h» pour le fichier de déclarations et «.C», «.cxx» ou «.cpp» comme extension pour le fichier de définition. En revanche, il ne faut pas utiliser ni l'extension «.c», ni aucune autre extension déjà réservée pour maintenir un programme en C++.

*** RECOMMANDATION ***

Chaque classe est définie par deux fichiers sources dont les noms sont le nom de la classe suivi d'un « .hh » ou d'un « .cc ». Le premier contient les déclarations et le deuxième, les définitions.

(Exemple) Une classe "Spool" sera définie par un fichier « Spool.hh » et un fichier « Spool.cc ».

(Exception) Certaines classes simples peuvent être maintenues avec la classe qui l'utilise. En particulier, le code source décrivant les classes dont les instances ne sont accédées que par un pointeur ou une référence peut être inclus dans le fichier de la classe qui l'utilise.

(Rappel) Ce découpage de classes en fichiers est obligatoire pour Java, qui est une sorte de C++ interprété plus récent et plus propre.

*** RECOMMANDATION ***

Les différents fichiers décrivant une classe se trouvent dans un répertoire qui porte le nom de la classe.

(Exemple) Soit une classe ScreenDriver, destinée à gérer un écran. Les fichiers ScreenDriver.hh et ScreenDriver.cc qui la décrivent se trouvent dans un répertoire ScreenDriver. Dans ce répertoire pourront aussi se trouver un Makefile (sous unix) et un fichier ScreenDriverTester.cc qui teste la classe ScreenDriver pour sa sécurité et qui en donne des exemples d'utilisation.

*** RECOMMANDATION ***

Le corps des fonctions et des méthodes inline doit se trouver dans les fichiers « .hh ».

(Pourquoi ?) Le corps d'une fonction doit être directement disponible pour le compilateur lors de son insertion dans le corps de la fonction appelante.

*** AMÉLIORATION ***

Le code source d'un module peut être réparti en trois fichiers au lieu de deux : les fonctions inline peuvent être maintenues à part dans un fichier ayant l'extention « XXX.icc ». Le fichier « XXX.cc » inclus alors son « XXX.hh » et son « XXX.icc ». Ceux-ci ne s'incluent pas mutuellement Les autres fichiers peuvent inclure le « XXX.hh » et éventuellement et « XXX.icc ».

(Pourquoi ?) Ceci permet de résoudre clairement les inclusions mutuelles de deux « .o » contenant chacun du code inline provenant de l'autre fichier.

*** AMÉLIORATION ***

De chaque classe peut dépendre un troisième fichier : « XXXTester.cc » qui teste la classe et donne un exemple de son interface.

(Exemple) L'implémentation de la classe Spool est contenue dans trois fichiers : « Spool.hh », « Spool.cc » et « SpoolTester.cc ».

Les inclusions de fichier d'entête (headers)

*** IMPÉRATIF ***

Ne pas inclure un fichier d'entête simplement parce qu'un autre module ou fichier d'entête en a besoin. N'inclure dans un fichier que les entêtes directement nécessaires.

(Pourquoi ?) Un fichier ne doit pas être inclus inutilement : cela gène la compréhension, augmente le temps de compilation, le nombre de fichiers recompilés et pollue l'espace de nommage.

(Comment ?) Cette même technique doit être utilisée pour inclure les headers standards comme les headers utilisateurs.

(Exemple) Voici comment inclure les fichiers d'entête :

```
// String.hh
#include <iostream.h> // <- La méthode «SelfDisplay» reçoit une
                      // instance de ostream. Or cette classe est
                      // définie dans iostream.h qui doit donc être
                      // inclus.
class String
      {
      char* Data;
      long Size;
      // ...
      String(const char* const);
      void SelfDisplay(ostream& _targetStream);
      };
```

```
// String.cc
#include "String.hh"
#include <string.h> // <- Aucune référence à ce package
                    // standard de gestion de chaînes de
                    // caractères n'était faite dans le fichier
                    // String.hh. En revanche, le fichier
                    // String.cc inclus le header pour
                    // pouvoir utiliser le fonction str*.
// ...
String::String(const char* const _clone)
      {
      Size = strlen(_clone) + 1;
      Data = new char[Size];
      strcpy(Data,_clone);
      }
```

*** IMPÉRATIF ***

#inclure dans le fichier d'entête (.hh) seulement les fichiers d'entête nécessaires à ce header. Et dans le fichier de définition (.cc) correspondant, ce header, ainsi que les autres fichiers d'entête .hh utiles au .cc.

(Pourquoi ?) Les autres modules qui incluent le .hh n'incluront que les fichiers d'entêtes nécessaires à ce .hh, mais pas au code associé.

*** RECOMMANDATION ***

Pour éviter toute ambiguïté, inclure les fichiers d'entête non standards en précisant leur répertoire père lorsqu'il s'agit de classes utilisateur.

(Exemple)

```
#include <String/String.hh>
```

*** AMÉLIORATION ***

Pour améliorer la vitesse de compilation, utiliser un mécanisme d'inclusion conditionnel de fichiers d'entêtes.

(Exemple) FIXME Mesure quantitative

```
#if !defined(String_hh)
#include <String/String.hh>
#endif // !defined(String_hh)
```

Les entêtes des fichiers source

*** RECOMMANDATION ***

Tout fichier source commence par un commentaire indiquant les informations suivantes :

```
///////////////////////////////////////////////////////////////
//
// File name : CodeQuality.txt
//
// Creation : 1995/07/07
//
// Version : 1995/07/11
//
// Author : Timothy Royer
//
// email : tim@puff.frmug.fr.net
//
// Purpose : Provide an efficient coding standard.
//
// Distribution : Unlimited (Copyright?)
//
// Use : Read this description. Try it. Improve it.
//
///////////////////////////////////////////////////////////////
```

Le champ «Distribution» indique dans quelles conditions le fichier source peut être distribué. Le champ «Use» indique comment utiliser/compiler le fichier.

De plus, le .cc contient ces informations qui sont destinées à évoluer :

```
//
// Todo :
//
// O Eliminer les fautes d'orthographe
// / Ajouter des exemples
```

```
// X Ajouter un todo
//
// History :
//
// 1994/01/01 : Johnny B. Goud : added History
//
```

*** AMÉLIORATION ***

Les « symboles » O, / et X signifient respectivement que la tâche décrite n’est pas commencée, est entamée et est achevée.

*** IMPÉRATIF ***

Le cas échéant, un copyright doit être indiqué explicitement. Pour distribuer librement un programme, inclure un avertissement et un copyright dégageant l’auteur de toute responsabilité.

(Comment ?) Voir le fichier « Conditions » fourni avec toutes les distributions de la Free Software Foundation. Ce texte est directement disponible sous emacs avec la commande «C-h C-c».

Le fichier d’entête (« .hh »)

Le fichier d’entête contient les déclarations du programme. L’orientation objet fait de lui le fichier central du développement : il décrit l'interface de chaque classe.

*** IMPÉRATIF ***

Chaque fichier d’entête (header) doit contenir un mécanisme évitant que son contenu ne soit utilisé deux fois lors d’une compilation.

(Pourquoi ?) Soient deux classes déclarées dans deux fichiers d’entête différents. Tous deux incluent le même entête XXX.hh. Si un fichier inclut ces deux fichiers d’entête de départ, il inclura indirectement deux fois le contenu de XXX.hh. Ceci poserait de problèmes : redéfinition de constantes de préprocesseur, double déclaration de classe... Un mécanisme gère ces difficultés de manière transparente : une constante de préprocesseur est définie lors de la première lecture du code. Un test sur la définition de cette variable permet de ne pas tenir compte du contenu du fichier d’entête à partir de la deuxième lecture.

(Rappel) Noter que ce mécanisme permet d’inclure plusieurs fois le même fichier lors de la compilation d’un fichier objet (.o). En revanche, il est pas utilisable pour éviter que la même information ne se trouve dans plusieurs fichiers objets d’un même projet. Par exemple, une instanciation de variable globale, comme une données membre statique, ne doit jamais se trouver dans un fichier d’entête.

(Exemple) Ce mécanisme simple est toujours le même. Voici un exemple pour une classe Foo, définie dans les fichiers Foo.hh et Foo.cc :

```
// Fichier Foo.hh
#if !defined(Foo_hh)
#define Foo_hh
// CONTENU DU HEADER Foo.hh
#endif // !defined(Foo_hh)
```

(Exemple) Ce même mécanisme peur aussi être employé avec une syntaxe légèrement différente :

```
// Fichier Foo.hh
```

```
#ifndef Foo_hh
#define Foo_hh
// CONTENU DU HEADER Foo.hh
#endif // ifndef Foo_hh
```

(Exception) Le fichier standard « assert.h » ne comprend pas ce mécanisme. Ceci permet

*** IMPÉRATIF ***

Un fichier d'entête ne doit rien contenir (sauf commentaires) avant « #if !defined XXX_hh » ni après « #endif // XXX.hh ».

*** AMÉLIORATION ***

Utiliser un mécanisme qui détecte l'inclusion d'un fichier d'entête par lui-même, directement ou non.

(Exemple) FIXME

*** AMÉLIORATION ***

Pour un gros projet, il est peut être souhaitable de ne pas inclure de fichier d'entête utilisateur dans un fichier d'entête.

(Pourquoi?) Dans un gros projet, plusieurs classes peuvent s'inclure mutuellement. Dans ce cas, deux problèmes surgissent : d'une part des inclusions mutuelles de fichier d'entête provoquent des problèmes lors de la compilation et d'autre part les temps de recompilation du projet deviennent pénibles. En effet, la plupart des fichiers d'entêtes sont inclus par la plupart des fichiers de définition. Chacun de ceux-ci doivent être recompilés chaque fois que l'un de ceux là est modifié.

A noter que cette difficulté syntaxique correspond pour une fois à une réelle difficulté d'annalyse et de conception : la dépendance mutuelle de modules.

Le fichier de définition (« .cc »)

*** IMPÉRATIF ***

Lorsqu'un fichier doit inclure plusieurs entêtes, il doit impérativement pouvoir les inclure dans n'importe quel ordre sans que cela change quoi que ce soit à la compilation ou à l'exécution.

(Pourquoi ?) Cela libère le développeur d'une responsabilité inutile, dont la difficulté croît exponentiellement avec la taille du projet.

*** RECOMMANDATION ***

Le fichier source, les fichiers objets «.o» et les exécutables doivent répondre à la commande what(1) unix.

(Pourquoi ?) Ceci permet de savoir à quoi servent les fichier sans avoir à les ouvrir.

(Comment ?) Il faut que chaque fichier de définition contienne une chaîne de caractères décrivant l'utilité du fichier. Cette chaîne doit être précédée de la séquence escape «@(#)». La chaîne de caractères est affectée à une variable globale statique.

(Exemple) Voici un exemple de ligne d'information :

```
static const char* const Autodescription =
    "@(#)Projet Pipo-Mollo. Module de gestion de l'écran.
V6.66";
```

Lors de la compilation de ce fichier bimbo.c vers un bimbo.o et de l'édition de liens vers un a.out, l'appel à la commande produira la sortie suivante :

commande :

```
      what bimbo.c bimbo.o a.out
```

affichage :

```
      bimbo.c :
            Projet Pipo-Mollo. Module de gestion de l'écran.
      bimbo.o
            Projet Pipo-Mollo. Module de gestion de l'écran.
      a.out
            Projet Pipo-Mollo. Module de gestion de l'écran.
```

Présentation d'instructions

*** IMPÉRATIF ***

Utiliser syntaxiquement un pointeur sur fonction comme un nom de fonction.

(Exemple) Ces deux notations pour l'appel de fonction avec un pointeur sont correctes. La seconde est plus lisible que la première.

```
int f(int);
int (*pf)(int) = f;
(*pf)(999); // Notation inutilement surchargée
            // Son unique intérêt est de différencier
            // explicitement un pointeur sur fonction.
pf(333);    // Notation claire.
```

*** IMPÉRATIF ***

Ne pas utiliser l'opérateur « , ».

(Pourquoi ?) Le seul intérêt de l'opérateur «,» est de réduire syntaxiquement la taille du code source, au dépend de la lisibilité.

(Exemple) Voici un exemple d'utilisation de l'opérateur «,», dont l'usage n'est jamais souhaitable.

```
for(int leCompteur=0,someDesNPremiersEntiers=0;leCompteur<x;
      leCompteur++,sommeDesNPremiersEntiers+=leCompteur);
```

Par exemple, cet algorithme aurait pu être codé ainsi :

```
long sommeDesNPremiersEntiers = 0;
for(long leCompteur = 0; leCompteur < thatIndice;
leCompteur++))
      {
      sommeDesNPremiersEntiers += leCompteur;
      }
```

*** RECOMMANDATION ***

Laisser un espace avant et après chaque opérateur.

*** RECOMMANDATION ***

Utiliser abondamment les parenthèses pour signifier les priorités au sein d'une expression.

(Pourquoi ?) La priorité et l'associativité des nombreux opérateurs hérités du C est difficile à gérer et à maintenir. De plus, un opérateur peut être redéfini, mais sa priorité ne peut être changée. Cela peut poser des problèmes lorsque le sens logique d'un opérateur change : c'est le cas en particulier des opérateurs qui ont été choisis pour désigner les flux : << et >>. Leur priorité est très forte car ils étaient destinés à l'origine au décalage de bits. Or elle devrait être faible pour permettre aisément l'affichage d'instructions. L'usage de parenthèses lève les ambiguïtés.

Nommage

Dans les paragraphes suivant, nous allons étudier des méthodes pour déterminer le nom des identificateurs. Cet aspect de la programmation est fondamental pour la lisibilité et donc la maintenance de code.

Majuscules et minuscules

*** IMPÉRATIF ***

Les identificateurs de données constantes globales doivent être écrites entièrement en majuscule, les mots séparés par des caractères de soulignement.

(Exemple) EN_MAJUSCULES_SEPAREES_PAR_DES_UNDERSCORES ;

*** RECOMMANDATION ***

Les variables doivent être écrites en majuscules et minuscules selon la règle suivante :

- Variables locales (dont paramètres) : enMinusculesLesMotsSeparesParDesMajuscules ;
- Le reste (Nom de classe, de donnée membre, de méthode et de fonction, instance globale non constante) : ChaqueMotCommenceParUneMajuscule.

*** IMPÉRATIF ***

Aucun identificateur défini par l'utilisateur ne doit contenir deux caractères soulignés successifs (« __ »).

(Pourquoi ?) Ces noms sont réservés aux librairies standards pour éviter les conflits de nommage [Stroustrup 1991].

(Pourquoi ?) Une des difficultés de la compilation du C++ est l'usage d'un éditeur de liens (linker) correspondant à un standard ancien. Pour contourner le problème que pose la surcharge de fonction (overload), le compilateur modifie le nom des identificateurs en leur ajoutant en particulier des caractères soulignés (_). Des conflits de nommage peuvent apparaître si l'utilisateur définit des noms commençant par deux caractères de soulignement. De plus, cette présentation gêne la lisibilité. Il est impensable que deux identificateurs se différencient simplement par un caractère de soulignement à cause d'un risque évident de confusion par le programmeur. Ce caractère peut donc être supprimé.

*** RECOMMANDATION ***

Ne pas différencier deux identificateurs sur un caractère souligné en plus ou en moins. Ne pas différencier deux identificateurs en changeant la casse de certains caractères.

(Exemple) Exemple de noms trop proches l'un de l'autre :

```
int Tuttle ;
char Buttle ;
```

Choix des identificateurs

*** RECOMMANDATION ***

Éviter les abréviations et l'élision des conjonctions dans le choix d'un identificateur.

(Pourquoi ?) Pour ne pas retaper trop souvent des noms longs et employer ainsi des identificateurs explicites, utiliser un éditeur de texte comme emacs (ou même vi) qui permet la « complétion » dynamique des noms de variables en cours d'édition.

(Exemple) Exemples et contre-exemples :

```
int nbAuto; // autorisés ? automatiques ? automates ?
int nbMobilesAutomatiques;
for(int i = 0; i < s; i++) // ...
for(int mobileCounter = 0; mobileCounter < size;
mobileCounter++)
// ...
```

*** RECOMMANDATION ***

Le nom d'un identificateur ne doit pas comprendre de négation [McConnell 1993].

(Exception) Notons cependant que certains cas spécifiques font exception à cette règle, comme une constante de preprocessing utilisée pour une compilation optionnelle, qui ne sera pas définie par défaut.

(Exemple) Exemples et contre-exemples : FIXME meilleur exemple

```
// Demande un effort supplémentaire pour la négation :
bool notFinished = true;
while(notFinished) // ...
// Notation plus immédiate :
bool keyInFinished = false;
while(!keyInFinished) // ...
```

*** RECOMMANDATION ***

Adopter un système régulier de nommage des données membres et de leurs méthodes d'accès.

(Exemple) Voici une première possibilité :

```
Classe Humain
      {
      ...
      public :
      // Data access
      long Id(void) const;
      const String& Name(void) const;
      void Rename(const String& newName);
      ...
      private:
      // Datas
      long IdData;
      String NameData;
      };
```

(Exemple) Voici une seconde possibilité :

```
Classe Humain
      {
      ...
      public :
      // Data access
      long GerId(void) const;
      const String& GetName(void) const;
```

```
        void SetName(const String& newName);
        ...
        private:
        // Datas
        long Id;
        String Name;
        };
```

*** RECOMMANDATION ***

Nommer les instances et les méthodes de manière à ce que ces deux noms accolés lors d'un appel forme une suite de mots ayant un sens.

(Exemple) Voici un exemple :

```
AddressArray.IsSorted();
```

Les commentaires

*** IMPÉRATIF ***

Un commentaire est précédé et suivi d'une ligne vide.

*** IMPÉRATIF ***

Un commentaire structuré précède chaque fonction importante, pour la décrire. Il se présente en trois parties :

- Description de l'effet de la fonction.
- Description des paramètres, le cas échéant ;
- Description de la valeur retournée, au besoin ;

(Exemple) Voici un exemple de commentaire :

```
// Function : QuickSort
//
// Purpose : Sorts an array of element by swaping them.
//
// Parameters :
// thatAreaStart : Element's array address.
// thatNumberOfElements : Number of elements in the array.
// thatElementSize : Size of one element in bits.
// thatComparisonFunction : fonction comparing two elements. It
// must returns 0 if they are equal, 1 if the first one is
// smaller and -1 otherwise.
//
// Returns : Nothing
void QuickSort(
        void* thatAreaStart,
        const int thatNumberOfElements,
        int thatElementSize,
        int (*thatComparisonFunction)
              (const void* const, const void* const))
        {
        // DEEP MAGIC HERE k :)
        }
```

*** IMPÉRATIF 2.5.3 ***

Les commentaires ne doivent pas se trouver sur la même ligne qu'une instruction.

*** RECOMMANDATION ***

Il ne doit pas y avoir de commentaires dans le corps d'une fonction.

(Pourquoi ?) Si le code source est bien écrit, les noms d'identificateurs auto-descriptifs suffisent à comprendre le programme. En revanche, il ne devrait pas être nécessaire de lire un code pour savoir ce qu'il fait. Des commentaires concis (une phrase suffit souvent) doivent :

- Se trouver en début de chaque fichier, pour indiquer son contenu ;
- Avant chaque fonction, pour indiquer son intérêt et son interface.

(Exception) Bien sûr, lorsqu'un algorithme compliqué est utilisé, la théorie peut ne pas transparaître dans l'implémentation. Des commentaires peuvent alors être ajoutés. Cette situation reste très exceptionnelle.

(Exemple) Une autre raison de ne pas mettre de commentaires en bout de ligne. Quel est le problème ? FIXME test + expl

```
#define XTKP // Extraction toolkit procedure
```

*** IMPÉRATIF ***

Pour indiquer un commentaire, utiliser seulement «//». Ne pas utiliser les délimiteurs «/*» et «*/».

(Pourquoi ?) Il y a plusieurs raisons pour cela. Tout d'abord il faut éviter d'imbriquer les commentaires : certains compilateurs l'autorisent alors que la norme l'interdit [Stroustrup 1991]. De plus, les commentaires ne doivent pas être longs. Si une partie du code est mise en commentaire, cela doit être fait explicitement. Sinon, du code mis en commentaire risque d'être modifié lors de la maintenance. Eventuellement, pour masquer très temporairement une partie du code source, utiliser la directive de précompilateur «#if 0».

(Exemple) Exemples de bonnes et de mauvaises présentations de commentaires :

```
//
// Cette fonction calcule l'âge du capitaine.
//
/*
Sommes-nous dans un commentaires ?
*/
/*
** Cette fonction taille un biface.
*/
                         /*
Cette fonction est sûrement buggée.*/
```

Seule la première présentation est bonne. La troisième est adaptée au C.

*** RECOMMANDATION ***

Les commentaires dans une fonction doivent rester exceptionnels.

(Pourquoi ?)Ils ne sont utiles que pour décrire un algorithme compliqué. Mis à part en recherche ou en programmation système, l'algorithmique employée est simple voire élémentaire. Un mauvais codage rend la fonction difficile à lire et motive les commentaires. Ce n'est pas une bonne démarche. Le programmeur doit plutôt produire une implémentation claire et un corps de fonction autodescriptif.

*** RECOMMANDATION ***

Les commentaires sont indentés : ils commencent à la première colonne s'ils ne se trouvent pas dans une fonction.

Chapitre 3 - Structuration logique du programme

Thus spake the master programmer :
"A well-written program is its own heaven; a poorly-written program its own hell."
The Tao of Programming.

Les variables

Nous allons présenter dans les paragraphes suivants les recommandations concernant la déclaration de données en général, pour les types prédéfinis et enfin pour les types utilisateur simples.

Recommandations communes à tous les types

*** IMPÉRATIF ***

Une variable ne doit pas en masquer une autre, ce qui se produit lorsque deux variables ont le même nom et se trouvent dans le même espace (scope).

(Pourquoi ?) Lorsque plusieurs informations sont designées par le même nom et ne sont différentiables que par leur position par rapport aux structures de contrôle, cela réduit la lisibilité du code et favorise l'apparition d'erreurs, en particulier lors de la maintenance du source.

(Exemple) Voici un extrait de source :

```
int i = 0;
      // ...
      {
      int i = 0;
      // ...
      i++;
      // ...
      }
```

Si la deuxième déclaration de i est supprimée, alors il y a un risque pour que l'incrémentation ne le soit pas. Le code resterait valide et le compilateur ne peut détecter aucun problème.

*** AMÉLIORATION ***

Une variable doit être définie dans le plus petit espace de nommage dans lequel elle est utile.

*** AMÉLIORATION ***

Ne rien affecter à un identificateur de tableau.

(Exemple) Les types int a[] et int* a sont complètement différents, même si, une fois déclarés, leur usage est identique. Le programmeur doit être particulièrement vigilant : par exemple, beaucoup d'éditeurs de liens confondraient un int* a et un int a[] définis dans deux modules (.o) différents. Ceci est susceptible de provoquer une erreur fatale.

(Pourquoi ?) Voici une illustration de la différence entre ces deux types [C FAQ] :

```
char a[] = "hello";
char* p = "hello";
```

La variable « a » occupe 6 octets dans l'espace de la mémoire dynamique. Cette zone sera désallouée lorsque la variable sortira de son espace de validité. La variable p occupe 4 octets (taille courrante d'un pointeur). Elle est un pointeur qui référence une région de la mémoire non modifiable. Une nouvelle valeur peut être affectée à «p», mais pas à «a». En fait, un bon compilateur devrait imposer ici le type const char*.

Les types prédéfinis

*** IMPÉRATIF ***

Utiliser le type standard bool, ainsi que les constantes true et false.

(Comment?) Ce type fait désormais partie du langage C++. Son existance peut être simulée pour un compilateur trop ancien.

```
typedef int bool;
const bool true = (0==0);
const bool false = !true;
```

*** RECOMMANDATION ***

Ne pas utiliser de champs de bits.

(Pourquoi ?) Contrairement à une idée reçue ancienne, l'usage de champs de bits augmente la taille du code et son temps d'exécution. Il réduit aussi beaucoup la portabilité et la lisibilité du code. Cet exemple est représentatif des mauvais choix technologiques souvent justifiés par un besoin d'optimisation. L'alignement de champs de bits, comme le signe par défaut de ces champs dépend de l'implémentation.

*** RECOMMANDATION ***

Ne pas utiliser le type void*.

(Pourquoi ?) L'emploi du type (void*) enlève le bénéfice du contrôle de type. Son usage est justifié en C pour la manipulation de zones mémoires avec malloc, par exemple. Ceci n'a pas d'intérêt en C++ où le pointeur retourné par new est typé.

(Rappel) Le type (void*) permet de désigner une zone mémoire sans présomption de son contenu. Cette notion n'est plus ni utile ni souhaitable dans un environnement objet.

*** RECOMMANDATION ***

Ne pas supposer que le type char contient une donnée signée.

(Pourquoi ?) Préciser « signed » ou « unsigned » si les 8 bits du type sont utilisés. Le cas choisi par défaut dépend de l'implémentation.

(Exemple) Définir et utiliser ces deux types si vous avez besoin de savoir si un char est signé ou non :

```
typedef unsigned char uchar;
typedef signed char schar;
```

*** AMÉLIORATION ***

Comme type entier, n'utiliser que long et comme type numérique à virgule, n'utiliser que double.

(Pourquoi ?) À la différence de «int», «long» est le plus souvent codé sur 4 octets, sur les machines 16 bits comme 32 bits. «double» est toujours plus précis que «float». Réduire le nombre de types de base utilisés évite les casts automatiques, difficiles à maîtriser en C. De plus, si une valeur ne doit jamais être négative, il est plus sûr de la coder sur un format de donnée signé et de vérifier qu'elle reste positive. Enfin, la librairie mathématique standard c travaille en «double».

(Exception) Bien sûr, dans certains cas limites bien identifiés, le temps d'exécution est plus important que la sécurisation et la portabilité du code. Il peut alors être nécessaire de faire appel à l'ensemble des types de base que propose le C++.

Les types utilisateurs simples

*** RECOMMANDATION ***

Les seules variables globales à utiliser sont les données constantes, sous forme de type simple (sans constructeur) et const.

*** RECOMMANDATION ***

Lorsqu'une variable doit être déclarée globale (pour une raison sûrement indépendante de la volonté du programmeur : compatibilité ascendante ou maintenance de code par exemple), elle doit être déclarée extern dans le header (fichier *.hh) et instanciée réellement dans le code (fichier *.cc).

(Pourquoi ?) Si la variable est instanciée dans le header et que le header est inclus plusieurs fois, alors il pourra y avoir un problème lors de l'édition de liens. Certains compilateurs tolèrent cette situation, mais elle n'est bien sûr pas souhaitable. Si la variable n'est pas déclarée extern dans le header à inclure au besoin, il faudra la déclarer extern avant chaque usage, dans chaque fichier qui utilise la variable, ce qui est pénible à maintenir. Accessoirement, cette recommandation permet de modifier l'initialisation de la variable en minimisant les recompilations.

*** RECOMMANDATION ***

L'existence d'un très petit nombre de données globales peut être tolérée. Il faut les définir comme membre static.

*** IMPÉRATIF ***

Les pointeurs doivent être déclarés en employant «const» de la manière la plus stricte possible.

(Rappel) Les pointeurs peuvent être constants ou non et pointer sur des données constantes ou non. Ce sont deux paramètres indépendants. Un pointeur constant s'écrit void* const et un pointeur sur une donnée constante s'écrit : const Type*.

(Pourquoi ?) Utiliser ce mécanisme est fondamental :

- pour la sécurité du code : lorsque j'ai apliqué ce mécanisme pour la première fois, il m'a permis de détecter une bombe à retardement qui n'avait pas explosé lors des tests. Il était du type :

```
void foo(int index)
      {
      if(index = 0)
            //...
      }
```

Bien sûr, le paramètre passé par valeur n'était pas destiné à être modifié. En le manipulant comme const en redéfinissant l'entête de la fonction ainsi : void foo(const int index) est apparu le problème lors de la recompilation : j'avais utilisé par inadvertance l'opérateur = à la place de l'opérateur ==;

- pour la maintenance : Savoir si une donnée est destinée à être modifiée après son initialisation est une information précieuse qu'il n'est pas toujours aisé de déterminer lors de la maintenance. Le mécanisme const fourni cette information et garanti même que les données en lecture seule ne seront pas modifiées, quelle que soit la manière dont on y accède (pointeur, passage de paramètre par référence, héritage...).

(Exemple) Voici quelques illustrations :

```
      // Donnée constante :
const int nombreDeChromosomesHumains = 23;
      // Pointeur fixé sur une données constante :
const char* const TITRE = "Oui-oui au pays de la gomme
magique";
      // Pointeur variable sur une donnée constante :
const char* message = "SYNTAX ERROR. OK.";
      // Pointeur fixé sur une donnée variable :
char* const referencedUntilDelete= new char[99];
```

Les structures de contrôle

*** AMÉLIORATION ***

L'espace de validité d'une variable définie entre les parenthèses qui suivent une instruction « for » a changé : désormais, cette donnée n'est plus accessible que dans la boucle, alors qu'elle l'était aussi à l'extérieur de la boucle auparavant.

(Exemple) Le code suivant n'est plus valide :

```
for(int i = 0; i < HeureCourante; i++) cout << "coucou";
for(i = 0; i < MinuteCourante; i++) cout << "bip";
```

Ce code correspond aux nouvelles normes, mais ne sera pas compilé par un compilateur ancien :

```
for(int i = 0; i < HeureCourante; i++) cout << "coucou";
for(int i = 0; i < MinuteCourante; i++) cout << "bip";
```

Cette méthode est valide pour l'ancienne comme pour la nouvelle norme, mais elle n'est pas recommandée : l'espace de validité des compteurs de la boucle est inutilement important :

```
int i;
for(i = 0; i < HeureCourante; i++) cout << "coucou";
for(i = 0; i < MinuteCourante; i++) cout << "bip";
```

Cette manière de déclarer les variables de boucle respecte l'ancienne et la nouvelle norme. Elle est lisible. Celle-ci doit être utilisée.

```
for(int heureC = 0; heureC < HeureCourante; heureC++) cout <<
"coucou";
for(int minuteC = 0; minuteC < MinuteCourante; minuteC++) cout
<< "bip";
```

La classe

*** IMPÉRATIF ***

Toutes les données membres sont privées.

(Pourquoi ?) Les données membres ne doivent être modifiables que par des méthodes d'interfaces (accesseurs).

(Exemple) Exemple d'accès aux données membres :

```
class Capitaine
	{
	int Age;
	public:
	const int& GetAge(void) const { return Age; }
	void SetAge(const int& thatNewAge)
		{ ASSERT(thatNewAge>0); Age = thatNewAge; }
	};
```

Les membres privés peuvent être redéfinis en protégé (protected) pour être accessible aux classes dérivées.

*** IMPÉRATIF ***

Les données membres dont les valeurs sont fixées à l'instanciation de la classe, et ne doivent plus être modifiées ensuite, sont définies comme constantes.

(Exemple) Exemple d'utilisation d'une donnée membre constante durant la vie de l'objet :

```
class Human
	{
	const bool Male;
	public:
	Human(const bool& thatIsMale) : Male(thatIsMale) {}
	// ...
	};
```

Dans cet exemple, le genre du Human instancié est « construit » avant l'entrée dans le constructeur. Sa valeur est fixée par thatIsMale.

*** IMPÉRATIF ***

Cacher les méthodes générées automatiquement par le compilateur, si elles ne sont pas définies. Ces méthodes sont : le constructeur vide, le constructeur par copie, l'opérateur = et le destructeur vide.

(Pourquoi ?) Ces méthodes sont une des failles de l'implémentation des objets en C++. Ces quatre méthodes sont définies pour les struct du C et ont été maintenues pour les classes dans un souci de compatibilité. Si elles ne sont pas définies pour être utilisées, le développeur doit les masquer pour s'assurer qu'elles ne seront pas appelées par inadvertance. Comme pour beaucoup de propositions de ce guide, celle-ci prend bien sûr toute son importance dans un gros projet, où l'utilisateur d'une classe n'est pas toujours celui qui l'a écrite et peut ne pas savoir que le constructeur par copie qui est appelé lors d'un passage de paramètres n'est pas définit et produit un résultat aléatoire.

(Exemple) Voici une technique permettant de masquer ces méthodes lorsqu'elles ne sont pas définies. D'une part, elles doivent être masquées pour l'extérieur de la classe. Pour cela, il suffit de les déclarer private. Ne pas définir leur corps. D'autre part, elles doivent aussi être masquées à l'intérieur de la classe. Pour cela, il faut les déclarer inline et ne pas définir leur corps. En fait il n'est pas nécessaire de les déclarer inline, mais dans ce cas, si l'on tente d'utiliser une méthode masquée, le message d'erreur produit à la compilation est beaucoup plus clair : il est indiqué à l'appel de la méthode et non pas plus tard, lors de l'édition de lien où les messages d'erreur sont beaucoup moins précis.

cf. Annexe C - Exemple de présentation

(Exemple) Voici une implémentation élémentaire d'une classe String. L'usage qui en est fait ici provoque une erreur fatale car les méthodes générées par le compilateur sont utilisées, alors qu'elles n'ont pas été définies.

```
// String.hh :
class String
	{
	char* Data;
	public:
	String(void);
	String(char* const thatCreator);
	~String(void);
	void SelfDisplay(void) const;
	};
// String.cc :
String::String(void)
	{
	Data = new char[1]; Data[0] = '\0';
	}
String::String(char* const thatCreator)
	{
	Data = new char[strlen(thatCreator)+1];
	strcpy(Data,thatCreator);
	}
String::~String(void)
	{
	delete[] Data;
	}
void String::SelfDisplay(void) const
	{
	if(Data)
		{
		cout << Data;
		}
	else
		{
		cout << "(null)";
		}
	}
void DisplayString(const String thatString2Display)
	{
	thatString2Display.SelfDisplay();
	}
int main(void)
	{
	String myName("Foo Bar");
	DisplayString(myName);
	return 0;
	}
```

Dans ce cas, une String myName est instanciée, puis passée en paramètre par valeur. Une String thatString2Display est instanciée en entrant dans DisplayString, par appel au constructeur par copie. Ceci est fait par le compilateur de manière transparente. Or, ce constructeur qui prend une String& en paramètre n'est pas défini. Il est donc défini automatiquement par le compilateur pour effectuer une copie membre à membre. Les données Data de myName et de thatString2Display pointent donc sur la même zone mémoire. Ensuite, à la sortie de DisplayString(), thatString2Display est détruit. La zone pointée par son Data est désallouée lors de l'exécution de son destructeur. myName référence désormais une zone désallouée.

(Rappel) Noter d'autre part la différence entre le constructeur par copie et l'opérateur =(). Les notations String titi(3), String toto(« hello »), String tutu(toto) ou String tata(« boo »,10) utilisent le constructeur avec paramètres. Or les trois premières déclarations peuvent aussi s'écrire : String titi = 3, String toto = « hello » et String tutu=toto. Dans ces trois cas, ce sont les constructeurs avec paramètres qui sont utilisés. Pour appeler le constructeur vide, il faut employer la notation suivante : String titi ; titi = 3 ; ou String toto ; toto = « hello » ou encore String tutu ; tutu = toto.

L'appel au constructeur avec le signe « = » n'est possible que pour les constructeurs recevant un unique paramètre. La notation du constructeur avec le signe = est utilisée principalement pour la construction par copie et la notation avec parenthèses est utilisée dans les autres cas.

FIXME + clair

*** RECOMMANDE ***

La déclaration vide de classe n'est pas souhaitable.

(Rappel) Il est possible de signaler l'existence d'une classe sans la déclarer. Il suffit pour cela de faire précéder le nom de la classe du mot-clef «class». Exemple : « class Mollo ; ». Cette simple déclaration ne permet bien sûr pas de présumer quoi que ce soit de la classe, mais permet d'utiliser des pointeurs sur la classe.

(Pourquoi ?) L'apparition de ces indications de classes dans un code source montre un problème de découpage du code source en fichier.

(Comment ?) Il faut inclure le fichier d'entête déclarant la classe à utiliser.

(Exception) Dans deux cas, la déclaration vide d'une classe est nécessaire :

- L'inclusion mutuelle de deux classes;
- Lorsque l'on veut éviter que trop de fichiers d'entête s'incluent mutuellement pour réduire le temps de recompilation d'un projet important.

(Exemple) Voici un exemple illustrant les cas pour lesquels une déclaration vide est utile : les inclusions mutuelles.

```
// Agent.hh :
#if !defined(Agent_hh)
#define Agent_hh
class Word;
class Agent
	{
	World MyWorld;
	public:
	Agent(World& _myNativeWorld);
	void Think(void);
	// ...
	};
#endif // !defined(Agent_hh)
// Agent.cc :
#include "Agent.hh"
#include <World.hh>
Agent::Agent(World& _myNativeWorld) : MyWorld(_myNativeWorld)
	{
```

```
	}
void Agent::Think(void)
	{
	// ...
	currentWeather = MyWorld.AskForTheWeather();
	// ...
	}
// World.hh :
#if !defined(World_hh)
#define World_hh
class Agent;
class World
	{
	Agent** Population;
	long PopulationSize;
	// ...
	inline void OneTurn(void);
	};
#include <Agent.hh>
void World::OneTurn(void)
	{
	for(popCounter = 0; popCounter < PopulationSize;
popCounter++)
		{
		Population[popCounter]->Think();
		}
	}
#endif // !defined(World_hh)
```

*** RECOMMANDATION ***

Il faut éviter de définir plusieurs opérateurs de transtypage pour une classe.

(Exemple) Voici une mauvaise utilisation du mécanisme de surcharge :

```
class String
	{
	char* Data;
	//...
	operator const char*(void) const { return Data; }
	operator int(void) const { return atoi(Data); }
	//...
	};
```

Une classe String définie comme ci-dessus peut paraître ergonomique. En fait plusieurs problèmes apparaissent. Il est interdit de passer directement une instance de cette classe à une fonction surchargée pour recevoir un «const char*» et un «int». Il faut préciser l'opérateur de cast choisi.

*** IMPÉRATIF ***

Indiquer inline, static ou virtual uniquement dans la déclaration de la classe, et pas à la définition de la méthode.

*** RECOMMANDATION ***

Ne pas inclure le corps d'une méthode dans la déclaration d'une classe. Si une fonction doit être déclarée inline, mettre au besoin sa définition dans le header, après la définition de la classe.

*** RECOMMANDATION ***

Éviter d'utiliser le mot clef friend.

(Pourquoi?) Ce mécanisme brise l'encapsulation des données. L'intérêt de la sécurisation des méthodes d'interface diminue pour une classe qui a des friends. Il est cependant préferable de lui ajouter un "friend" pour ne pas avoir à passer des membres de private à public.

*** AMÉLIORATION ***

Éviter d'utiliser le mot-clef protected, en particulier pour qualifier une donnée membre.

(Pourquoi ?) Bjarne Stroustrup, instigateur du C++, considère dans son dernier livre [Stroustrup 1994] que le mécanisme protected n'aurait pas dû être implémenté en C++. Il sera maintenu pour garantir la pérennité des codes existant mais son usage n'est pas recommandé.

(Comment ?) Les méthodes d'interface suffisent dans beaucoup de cas.

*** AMÉLIORATION ***

Indiquer les membres publics, puis les membres privés.

(Exemple) cf. XXX.hh.

*** RECOMMANDATION ***

Disposer les méthodes dans le même ordre dans le fichier d'entête que dans le fichier de définitions.

(Pourquoi?) Pour maintenir la même logique, comme pour faciliter la recherche d'une définition de méthode.

*** RECOMMANDATION ***

Éviter d'utiliser une instance globale dans un constructeur.

(Pourquoi ?) L'instance globale peut ne pas avoir été construite lors de son utilisation par un constructeur, dans le cas où ce constructeur est appelé pour instancier une autre globale.

(Exemple) Metaproblème classique illustré dans un nouveau système formel [Ellemtel-1992] :

```
class Poule;
class OEuf
      {
      public:
      const char* Nom;
      OEuf(const char* const MonNom, const Poule& Mere);
      };

class Poule
      {
      public:
      const char* Nom;
      Poule(const char* const MonNom, const OEuf& Origine)
            : Nom(MonNom)
            {
            cout << "Je m'appelle "<< Nom ;
            cout << " et je proviens de l'oeuf ";
            cout << Origine.Nom << endl;
            }
      };
OEuf::OEuf(const char* const MonNom, const Poule& Mere):
Nom(MonNom)
      {
      cout << "Je m'appelle "<< Nom ;
      cout << " et ma mere est " << Mere.Nom << endl;
      }
extern Poule PremierePoule;
```

```
OEuf PremierOEuf("Cali",PremierePoule);
Poule PremierePoule("Mero",PremierOEuf);
int main(void)
	{
	return 0;
	}
```

Ce programme compile sans aucun avertissement. Lors de l'exécution de ce programme, deux instances globales seront construites avant l'entrée dans la fonction main() : PremierOeuf et PremierePoule. Il n'y a pas de moyen de déterminer lequel sera instancié en premier. Celui qui est construit en premier fait référence au nom de l'autre. Or, ce nom n'est pas encore initialisé car le constructeur qui doit le faire n'a pas encore été exécuté. Un pointeur invalide est donc déréférencé.

(Rappel) De plus le code du constructeur d'une instance globale s'exécute avant l'entrée dans le main(). Le code de son destructeur s'exécute après la fin du main(). Cela complique la compréhension du code source.

NB : Je ne connais pas de compilateur signalant ce problème. Or, il peut n'apparaître que tard au cours d'un développement : lors d'un changement de l'édition de lien ou d'un portage. Il peut être difficile à détecter dans une application importante.

*** RECOMMANDATION ***

Chaque classe comprend une méthode OK() vérifiant un invariant définissant la validité d'une instance. Elle utilise les méthodes OK() de ses instances membres.

(Exemple) Voici un exemple qui permet de tester l'intégrité d'une classe String :

```
class String
	{
	long Size;
	char* Data;
	// ...
	public :
	bool OK(void)
		{
		bool iAmHealthy = false;
		for(long charCount = 0; charCount < Size;
charCount++)
			{
			if(!Data[charCount])
				{
				iAmHealthy = true;
				}
			}
#if !defined NO_DEBUG
		if(!iAmHealthy)
			{
			cerr << "String::OK() failed."<< endl;
			cerr << "Char string is not null terminated.";
			cerr << endl;
			ASSERT(0);
			}
#endif // !defined NO_DEBUG
		return iAmHealthy;
		}
	};
```

Il peut être intéressant de pouvoir lancer interactivement la méthode OK() de chacune des instances pour vérifier au besoin la validité de l'état des données au sein du programme, en particulier lorsqu' un problème survient.

*** RECOMMANDATION ***

Éviter de définir un constructeur recevant un argument, lorsque sa logique ne correspond pas à un clonage. Le compilateur risque d'y faire appel implicitement.

(Exemple) Voici par exemple, deux constructeurs pour une classe String. L'un est souhaitable, l'autre pas.

```
class String
      {
      // ...
      public :
      String(const char* const thatClone);
      String(const long thatSize);
      // ...
      SelfDisplay(ostream& thatStream);
      // ...
      };
void DisplayString(const String& thatStringToDisplay)
      {
      thatStringToDisplay.SelfDisplay(cout);
      }
int main(void)
      {
      const char* const paradigmaticString = "Hello world !\n";
      const long paradigmaticSize = 256;
      DisplayString(paradigmaticString);
      DisplayString(paradigmaticSize);
      return 0;
      }
```

Lors de l'exécution de ce programme, la fonction «DisplayString» est appelée deux fois. Lors du premier appel, une variable temporaire de type String est construite par le constructeur de String recevant un «const char* const». Tout se passe bien : « Hello world ! » apparaît à l'écran. Lors du deuxième appel, la String temporaire est construite à partir d'un entier long. Était-ce vraiment le but recherché ? De toutes façons, l'indication d'une taille de String est un problème dont l'utilisateur de la classe ne doit jamais avoir à se soucier.

Les fonctions et les méthodes

Recommandations communes aux fonctions et aux méthodes

*** IMPÉRATIF ***

Toujours indiquer explicitement le type de la valeur de retour. Utiliser « void » si la fonction ne renvoie rien.

*** IMPÉRATIF ***

Toujours coller la parenthèse gauche d'une fonction au nom de celle-ci.

(Pourquoi ?) Cette règle est représentative de la rigueur avec laquelle un fichier source doit être écrit, surtout pour un projet important. Par exemple, lors de la maintenance, le nom d'une fonction peut être modifiée. Si tous les appels respectent la même présentation, il est aisé de faire une recherche automatique sur tous les appels, pour vérifier que les modifications respectent l'usage qui était fait de la méthode jusque là. Ceci grâce à une commande du type (un*x) :

grep « \.MethodeRecherchee(« `find $PROJECT -name « *.[hc]* »-print`

*** IMPÉRATIF ***

Présenter le prototype des fonctions avec les noms de paramètre de la même manière que pour leur déclaration.

*** IMPÉRATIF ***

Chaque fonction définie par l'utilisateur doit être précédée d'un prototype.

*** RECOMMANDATION ***

Une fonction ne doit pas être longue de plus de 100 lignes.

(Pourquoi ?) Ceci pour des raisons évidentes de lisibilité et de maintenance. Des études statistiques précises [CC FIXME] ont montré que le nombre de bugs par fonction croît exponentiellement avec la taille de celle-ci, à partir d'un certain nombre de lignes.

(Exception) Quelques fonctions peuvent dépasser cette taille, pour des besoins exceptionnels. En particulier, des fonctions concernant des interfaces homme-machine fondées sur des librairies lourdes à mettre en oeuvre.

*** RECOMMANDATION ***

Tous les paramètres sont reçus comme données constantes (const), sauf ceux qui sont passés par référence et qui sont destinés à être modifiés.

*** RECOMMANDATION ***

Lorsqu'une fonction doit toujours recevoir une donnée positive, utiliser un type signé et tester son signe plutôt que utiliser un type non signé.

*** AMÉLIORATION ***

Lorsqu'une méthode ne modifie pas l'instance dans certains cas, il faut la surcharger sur sa caractéristique «const».

(Rappel) Une fonction peut être overloadée sur sa caractéristique const : deux méthodes différentes d'une même classe peuvent avoir exactement le même prototype sauf en ce qui concerne leur caractère « const ». À l'exécution, la méthode const sera appelée dans la mesure du possible, sinon la méthode non-const sera appelée.

(Pourquoi ?) Ceci permet :

- D'identifier les cas où le contenu de la classe a changé ;
- D'appeler une méthode sur une instance const, alors que la version non-const de cette méthode doit modifier l'objet dans d'autres circonstances.

(Exemple) Voici une utilisation possible de la surcharge de fonctions :

```
class String
      {
      //...
      String& operator +=(const char* const
thatCharZero2Append);
      String& operator +=(const char thatChar2Append);
      String& operator +=(const String thatString2Append);
      //...
      };
```

Dans cet exemple, l'opérateur +=() doit permettre dans tous les cas de concaténer des caractères à la fin de la String.

(Exemple) FIXME axc tab const / non => const op = const != op()

*** AMÉLIORATION ***

Éviter d'utiliser plus de un « return » par fonction ou méthode.

(Pourquoi ?) Une fonction qui contient plusieurs returns ne respecte pas les règles élémentaires de la programmation structurée. Elle est difficile à maintenir.

*** RECOMMANDATION ***

Le nom des arguments éventuels d'une fonction doit être le même dans le prototype et dans la déclaration du corps de celle-ci.

(Pourquoi ?) Prévenir le code source des commentaires bas niveau est unanimement recommandée, mais cette méthode requiert une plus grande rigueur. En particulier en ce qui concerne la détermination des noms des identificateurs.

Fonctions

*** RECOMMANDATION ***

Tout le code doit se trouver dans des méthodes. Il n'est pas souhaitable de définir des fonctions sauf dans trois cas précis :

- Surcharge d'opérateurs ;
- newhandler ;
- main()
- Certaines utilisations des STL.

(Pourquoi ?) Malgré les habitudes qu'auront pu prendre les programmeurs en C et en assembleur, il est fortement déconseillé de déclarer des fonctions (par opposition aux méthodes). Dans un langage purement objet personne ne pense seulement déclarer un jour une procédure qui ne soit pas une méthode. Ce n'est vécu ni comme une brimade, ni comme un entrave à la conception d'un projet (cf. Smalltalk ou Java). Pour des raisons historiques, le C++ est un langage qui permet de briser facilement le modèle objet. Ce n'est pourtant pas une pratique recommandable. Cette idée peut sembler nouvelle à certains. Elle va pourtant très loin. Il est même possible d'écrire des logiciels systèmes entièrement en objets (OS, drivers). J'ai ainsi pu travailler sur une phase de boot unix qui avait été entièrement (et brillamment) orientée objet.[Detienne 199 ?]

*** AMÉLIORATION ***

Déclarer les prototypes des fonctions d'interface dans le fichier d'entête (« .hh »). Déclarer les prototypes des fonctions internes dans le fichier (« .cc ») qui les définit.

*** IMPÉRATIF ***

Les dépassements de capacité mémoire doivent être détectés.

(Pourquoi ?) Un dépassement de capacité mémoire indique le plus souvent une « fuite » dans la gestion de la mémoire dynamique (memory leak). Ce problème doit de toute façon être détecté, surtout lorsque le programme s'exécute sous un système d'exploitation qui ne fait pas travailler le processeur en mode protégé (DOS ou Windows) : dans ces cas-là, un problème comme un dépassement de capacité mémoire provoque un comportement imprévisible qui peut obliger à éteindre la machine ou qui peut même endommager le disque dur.

(Exemple) Il existe deux méthodes pour détecter les dépassements de capacité mémoire.

Une méthode consiste à indiquer une fonction utilisateur qui sera appelée automatiquement si un dépassement de capacité se produit. Cela se fait simplement en appelant la fonction standard «set_new_handler()» avec comme paramètre le pointeur sur une fonction utilsateur. Cette fonction doit gérer les dépassements de mémoire :

```
void MyNewHandler(void)
      {
      cerr << "Memory exhausted. Sorry." << endl;
      abort();
      }
int main(void)
      {
      set_new_handler(MyNewHandler);
      // Le programme principal.
      return 0;
      }
```

*** AMÉLIORATION ***

La fonction qui gère le dépassement de mémoire essaie de libérer de la mémoire et permet de reprendre l'exécution, si possible.

*** IMPÉRATIF ***

La fonction main prend zéro, deux ou trois arguments selon les besoins.

(Rappel) Voici le type de ces arguments éventuels :

- int argc : le nombre d'arguments de la ligne de commande, nom de l'exécutable inclus,
- char* argv[] : tableau de pointeurs sur les chaînes/token de la ligne de commande, commençant par le nom de l'exécutable et se terminant par (char*)0 ;
- char* env[] : liste de chaînes représentant les variables d'environnement.

*** IMPÉRATIF ***

La fonction main retourne un int : 0 si l'exécution s'est bien passée, une autre valeur sinon.

(Exemple) Tous les programmes retournent une valeur à l'environnement à la fin de leur exécution. Si le type de la valeur de retour de main est forçé à void, la valeur retournée à l'environnement est indéfinie, ce qui peut être gênant pour un script shell, par exemple. Cette valeur est fixée par un « return » dans la fonction « main » ou par un « exit » depuis n'importe quel endroit.

Méthodes

*** IMPÉRATIF ***

Chaque méthode qui ne modifie pas les données de l'instance de la classe à laquelle elle appartient doit être const.

*** IMPÉRATIF ***

Une méthode publique d'une classe ne doit retourner, ni un pointeur de membre non constant, ni une réference non constante, sur une donnée membre.

(Pourquoi ?) Si ces informations sortent de la classe, l'encapsulation des données est rompue et les données privées d'une classe peuvent être modifiées sans contrôle.

*** AMELIORATION ***

Deux classes ne doivent pas avoir une méthode d'interface qui a le même nom.

(Exception) Bien sûr cette recommandation ne s'applique pas aux classes qui ont un rapport d'héritage entre elles.

(Pourquoi?) Lorsqu'une méthode est appelée au sein d'un gros projet, il est difficile de déterminer à quelle classe elle appartient. C'est un danger qu'il faut garder à l'esprit lors de l'implémentation. Mais ce n'est pas toujours réalisable dans de bonnes conditions.

Inclusions mutuelles

Lorsque l'inclusion mutuelle de deux headers n'est pas due à une erreur d'analyse mais correspond bien à une logique incontournable, alors l'application de cette algorithmique particulière doit être traitée avec un soin particulier.

*** RECOMMANDATION ***

Éviter l'inclusion mutuelle de deux fichiers d'entête.

(Pourquoi ?) Programmer deux objets (ou plus !) qui s'utilisent mutuellement pose des problèmes de conception et doit être évité dans la mesure du possible. Cette situation résulte d'une algorithmique complexe impliquant en particulier des récursivité involontaires qui pourraient provoquer des problèmes dont l'origine serait difficile à détecter.

*** RECOMMANDATION ***

Lorsque deux classes s'utilisent réciproquement :

- l'une des classes ne doit pas utiliser un membre de l'autre classe ;
- l'une des classes ne peut contenir que des pointeurs ou des références vers l'autre classe ;
- le constructeur de l'une des classes ne doit pas utiliser une instance globale de l'autre classe ;
- le .hh de chaque classe est structuré normalement (cf. Annexe) à deux exceptions près : l'instruction suivante précède la déclaration de la classe : « class AutreClasse ; » et la directive suivante est inclue juste après la déclaration de la classe : « #include <AutreClasse.hh> ».
- la déclaration de chaque classe est précédée de la ligne :

class AutreClasse ;

*** AMELIORATION ***

Utiliser un mécanisme qui permet la détection de cycles d'inclusions de headers.

(Pourquoi ?) FIXME

(Comment)

```
#if defined(TITI_CURRENT)
#error Header cycle detected
#else // defined(TITI_CURRENT)

// Contenu habituel du fichier d'entête

#undef TITI_CURRENT
#endif // defined(TITI_CURRENT)
```

Le préprocesseur

*** IMPÉRATIF ***

L'usage de #pragma doit être évité au maximum, sauf éventuellement dans une zone de compilation optionnelle dépendant du compilateur.

*** AMÉLIORATION ***

Une macro de préprocessing dans le code source n'a pas besoin d'être suivi d'un point virgule. Il est possible de l'imposer dans un souci de cohérence syntaxique avec un appel de fonction.

(Comment?) Il suffit d'insérer le corps de la macro dans une boucle do {...} while(0).

(Exemple) Voici comment donner l'illusion d'une fonction magique qui implémenterait le mécanisme d'assertion :

```
#define assert(X) do \
      { \
      if(!(X)) \
            { \
            cerr << "Assertion failed : (" << #X << ')'; \
            cerr << endl << "In file : " << __FILE__; \
            cerr << "at line #" << __LINE__ << endl; \
            abort();} \
            } \
      } while(0)
```

(Rappel) La structure do {...} while(...) doit être suivie d'un point virgule. Le "while(0)" est bien sûr éliminé de l'exécutable par tout compilateur digne de ce nom.

*** AMÉLIORATION ***

L'intérêt réel du preprocessing réside principalement dans la compilation optionnelle. Ce mécanisme est fondamental lorsqu'un logiciel est maintenu sur plusieurs plateformes. Les zones de code dont l'inclusion dans le projet est fonction de la compilation doivent être maintenues dans un fichier à part. Cette difficulté doit être masquée au développeur dans toutes les autres parties du projet.

*** IMPÉRATIF ***

Les constantes de préprocessing définies automatiquement selon [Stroustrup 1991] sont :

- __LINE__ : valeur décimale indiquant la ligne courante ;
- __FILE__ : chaîne de caractères indiquant le nom du fichier ;
- __DATE__ : chaîne de caractères indiquant la date de la compilation du module courant selon le format suivant : « Mmm dd yyyy » ;
- __TIME__ : chaîne de caractères indiquant l'heure de la compilation du module courante selon le format suivant : « hh :mm :ss » ;
- __cplusplus : simplement défini pour indiquer que le compilateur attend du C++.

Elle ne sont pas redéfinissables directement (ni par #define, ni par #undef). Cependant la directive #line permet de redéfinir __LINE__ et éventuellement __FILE__. L'usage de la directive « #line » n'est pas recommandé.

*** RECOMMANDATION ***

Les compilations optionnelles se définissent avec la commande de préprocessing #if, suivie éventuellement de « defined(...) » ou de « !defined(...) ».

*** RECOMMANDATION ***

Préférer #if à #ifdef/#ifndef.

(Pourquoi ?) #if est plus général. Il permet au besoin de tester la valeur d'une constante de préprocessing.

(Exemple) Voici une implémentation d'un code pouvant être compilé avec plusieurs niveaux de debug. Noter que la condition de compilation optionnelle est ajoutée après le «#endif» qui termine la zone, pour faciliter la lisibilité et supprimer les ambiguïtés.

```
      #if DEBUG_LEVEL > 0
      // Quelques tests.
      #if DEBUG_LEVEL > 2
      // D'autres tests.
      #endif // DEBUG_LEVEL > 2
      #endif // DEBUG_LEVEL > 0
```

*** RECOMMANDATION ***

Faire suivre la commande de préprocessing #else par un commentaire contenant la condition du #if correspondant. Faire suivre la commande «#endif» de ce même commentaire. Ajouter « else » si le test possède une clause else.

(Exemple) Voici une option de compilation permettant de maintenir un code sur plusieurs versions d'un compilateur :

```
#if COMPILER_VERSION < 2.7
// Oldies
#else // COMPILER_VERSION < 2.7
// News
#endif // else COMPILER_VERSION < 2.7
```

Voici un code utile pour permettre de simuler l'existence du type bool sur des compilateurs ne le proposant pas encore en standard :

```
#if !defined(__GNUG__)
typedef int bool;
const bool false = (0 == 1);
const bool true = !false;
#endif // !defined(__GNUG__)
```

*** RECOMMANDATION ***

Utiliser le mécanisme d'assertion à chaque fois qu'il est significatif.

(Rappel) L'invariant est le fondement de la preuve de programme, science destinée à garantir le résultat d'un traitement de données. Il indique un ensemble de conditions qui doivent rester vraies tout au long de l'exécution. Une assertion permet de vérifier un invariant de manière simple, dans un code source.

(Pourquoi ?) Le mécanisme d'assertion permet de détecter un problème, ainsi que son identification lors de l'exécution, autrement que par la constatation de ses effets. Il est facile d'utilisation : sa syntaxe est concise et il est débrayable facilement lorsque le code est testé. De plus il facilite la maintenance et la relecture du code : il permet d'exprimer l'invariant du code et situe ainsi le contexte de validité des portions de code.

(Comment ?) Le assert est l'un des rares mécanismes sains fondé sur le préprocesseur. Son fonctionnement est simple. La macro ASSERT reçoit une expression booléenne. Si elle est vraie, rien ne se passe. Si elle est fausse, alors le programme est devenu incohérent et ASSERT va indiquer un problème. FIXME assert.h Sous-chapitre ASSERT.

Dans l'idéal, une méthode commence et se termine par un ASSERT() pour :

- S'assurer que la méthode se trouve dans un état valide au moment ou on l'applique ;
- S'assurer que la méthode laisse la situation dans un état valide.

Cette technique est l'application directe de la preuve de programme.

Notons aussi que le mécanisme d'assertion permet d'afficher la date et l'heure de la dernière compilation du code source. Ceci pour identifier un problème dû à une définition incorrecte des dépendances dans le makefile qui aurait empêché une recompilation nécessaire d'avoir lieu.

Enfin, lorsqu'une assertion détecte un problème, elle affiche un message puis produit un core (sous unix) qui permettra *a posteriori* de connaître l'état de la mémoire à ce moment là. Ces informations sont ensuite exploitables avec un debugger.

(Exemple) Voir l'annexe C - Exemple de présentation pour une implémentation de ASSERT.

Voici un exemple de sécurisation de la fonction valeur absolue :

```
#include <limits.h>
long Abs(const long thatValueToAbs)
      {
      ASSERT(thatValueToAbs != LONG_MIN);
      if(thatValueToAbs < 0)
            {
            return -thatValueToAbs;
            }
      else
            {
            return thatValueToAbs;
            }
      }
```

Rappel : les entiers signés sont codés sous le format «complément à deux». Or la valeur absolue de la plus grande valeur négative est supérieure de 1 à la plus grande valeur positive maintenable dans ce format, pour un nombre de bits donné. Ces valeurs limites dépendent de chaque machines et sont maintenues dans le header standard «limits.h».

*** IMPÉRATIF ***

L'usage du préprocesseur doit être limité au strict nécessaire. Lorsqu'une alternative se présente, il faut toujours préférer utiliser une fonctionnalité du langage C ou C++ à une fonctionnalité du préprocesseur [Stroustrup 1995].

(Pourquoi?) Le préprocesseur est un parseur de texte qui ne respecte en aucune façon la signification du code, contrairement au compilateur. De plus les erreurs dûes au préprocesseur peuvent être difficiles à détecter.

*** IMPÉRATIF ***

Définir une variable constante (sic) ou un enum plutôt qu'une constante du préprocesseur.

(Pourquoi ?) Les contrôles sur les types, en C++, offrent une meilleure sécurité sur des variables que sur des constantes du préprocesseur.

(Exemple) Soit une fonction surchargée :

```
float cos(const float& thatAngle) ;
double cos(const double& thatAngle) ;
Laquelle est appelée ?
#define ANGLE1 123.456
#define ANGLE2 12345.6789
const double ANGLE3 = 112233.456 ;
```

(Exemple) Soit une constante :

```
#define MAX 50000
```

Le type de MAX sera différent suivant le compilateur utilisé. Ce n'est bien sûr pas le cas de :

```
const int MAX = 50000;
```

De plus, un bon compilateur avertira le cas échéant du dépassement de capacité de la valeur immédiate pour la constante qu'elle initialise, en fonction de son type.

*** IMPÉRATIF ***

Définir une fonction inline plutôt qu'une macro.

(Exemple) Voici un exemple d'une fonction qui est traditionnellement définie comme une macro. L'implémentation présentée ici est aussi rapide à l'exécution, bénéficie d'un meilleur contrôle sur les types, permet de mieux comprendre un problème éventuel lors de la compilation et permet de comparer des instructions complexes sans risque : elles ne sont évaluées qu'une fois.

Vérifier que, contrairement à une définition de macro, cette fonction résout Max(a,b++) sans effet de bord.

```
inline template<Type> const Type&
Max(const Type& thatFirstArg2Compare, const Type&
thatSecondArg2Compare)
      {
      if(thatFirstArg2Compare > thatSecondArg2Compare)
            {
            return thatFirstArg2Compare;
            }
      else
            {
            return thatSecondArg2Compare;
            }
      }
```

*** IMPÉRATIF ***

Limiter l'usage de la commande #define à des constantes qui seront utilisées par le préprocesseur.

Debug

*** RECOMMANDATION ***

Le code source doit pouvoir compiler pour produire deux versions différentes :

- En version debug, incluant de nombreux tests permettant d'améliorer la qualité du code ;
- En version définitive (release), sans les tests. Cette version devra s'exécuter rapidement, ne devra pas contenir d'information de debug (qui permet entre autres

d’éditer le code source en entier, ce qui n’est pas toujours souhaitable pour une version client).

(Rappel) Il est cependant fondamental que les deux versions exécutent le même code, de la même manière, aux tests près : sinon la version debug n’est plus représentative de version release.

(Comment ?) Les zones de compilations optionnelles permettent d'implémenter ce mécanisme.

(Exemple) Voici une méthode de classe String qui produit l'affichage de la chaîne qu'elle maintient. La classe définit deux données membres : int Size et char* Data. Elle vérifie simplement que la string se termine par le caractère null en version debug.

```
Void String::SelfDisplay(ostream& s) const
      {
      #if !defined(NO_DEBUG)
      int charCount;
      for(charC=0; charC<Size && Data[CharC]; CharC++)
            {
            }
      if(!Data[CharC])
            {
            cerr << "Display invalid string." << endl;
            Data[Size-1] = 0;
            }
      #endif // !defined(NO_DEBUG)
      s << Data;
      }
```

Ce code peut être compilé de deux manières différentes :

Avec vérification pour obtenir la version de travail :

```
cc String.cc -c -g3 +w -DNO_DEBUG
```

Sans vérification pour obtenir la version de livraison :

```
cc String.cc -c -O3 +w
```

La taille de l'exécutable à livrer pourra être réduite grâce à la commande unix strip.

Noter que, en version debug, le programme essaye de corriger le problème en fixant la fin de la chaîne à null. Il pourrait aussi être souhaitble de produire une image de la mémoire pour comprendre le problème au moyen de abort().

*** IMPÉRATIF ***

Les zones du code source destinées seulement à la détection d’erreurs et qui sont débrayables par une option de compilation ne doivent pas modifier les données du programme.

(Pourquoi ?) Si le code destiné au debug modifie des données, le programme s’exécutera différemment en version debug et en version release et celui-là ne sera plus significatif.

*** RECOMMANDATION ***

Par défaut, le programme doit compiler en version debug. Une option de compilation définissant la constante NO_DEBUG permet de compiler en version release.

(Comment ?) Tous les compilateurs C et C++ permettent de définir une constante de préprocessing lors de la compilation en passant sur la ligne de commande une option -D suivie du nom de la constante à définir.

(Exemple) Voici un exemple de ligne de compilation permettant de compiler un fichier String.cc :

Pour obtenir une version de test :

```
CC +w -o String String.cc
```

Pour obtenir une version release :

```
CC +w -o String -DNO_DEBUG String.cc
```

*** RECOMMANDATION ***

Une fonction doit commencer par une section dont la compilation est optionnelle, qui sert au debogage (debugging).

(Pourquoi ?) Lors de la programmation d'une fonction, il faut avoir le réflexe de la sécuriser. Pour cela il faut que ses données de départ soient valides. Tous les cas problèmatiques ne peuvent pas toujours être détectés. Cependant, certains indices doivent être contrôlés, dans la mesure du possible (pointeurs null ou identiques, taille nulle...). Dans certains cas, un invariant de boucle ou de fin de procédure peut être utile.

Bien sûr, la sécurisation d'une méthode d'interface peut rester stricte, alors qu'une méthode interne pourra éventuellement s'exécuter sans test en version release.

(Exemple) Une fonction classique enfin sécurisée :

```
int Abs(const int& ThatValueToAbs)
      {
#if !defined(NO_DEBUG)
      if(ThatValueToAbs == 1 << (sizeof(int) - 1))
            {
            cerr << Abs() exception." << endl;
            }
#endif // !defined(NO_DEBUG)
      if(ThatValueToAbs < 0)
            {
            return -ThatValueToAbs;
            }
      else
            {
            return ThatValueToAbs;
            }
      }
```

(Exemple) Voir le fichier exemple joint en annexe, XXX.cc pour un exemple de macro ASSERT.

*** AMÉLIORATION ***

Pour une plus grande efficacité du debug et pour bien contrôler la mise au point et la livraison du logiciel, plusieurs niveaux de debug peuvent être définis.

(Exemple) Description des niveaux de debug :

0 : aucune sécurité. Utilisé uniquement comme indice de comparaison pour vérifier que le debug ne prend pas trop de temps à l'exécution.

1 : debug minimal, version destinée à être livrée.
2 : Version utilisée lors de la mise au point du programme.
3 : Version contenant les tests coûteux en temps. À utiliser pour identifier un problème particulier.

*** AMÉLIORATION ***

La convention de nommage des paramètres diffère de celle des variables locales.

(Pourquoi ?) Lors de la maintenance, modifier le type d'une donnée, n'a pas les mêmes conséquences sur une variable que sur un paramètre. À l'usage, il est très pratique de les différencier instantanément. Bien sûr, une autre méthode pour le faire peut être choisie.

(Exemple) Le nom d'un paramètre commence toujours par «that» ou bien par un caractère de soulignement «_». Les autres mots jamais. (exception : les arguments de : :main(argc,argv) sont admis par tous et doivent être utilisés).

Instruction

Les opérateurs unaires ++ et—doivent précéder la variable qu'ils modifient.

(Pourquoi ?)FIXME.

*** RECOMMANDATION ***

Les opérateurs unaires ++ et—doivent être utilisés seuls.

(Exemple) Voici des cas où le comportement du code est indéterminé. Les problèmes illustrés dans ces exemples ne seraient pas apparus si les opérateurs unaires d'incrémentation ou de décrémentation avaient été utilisés seuls, dans une instruction C++ distincte.

```
int a = 666;
int b;
b = a++;
a += --a;
```

*** RECOMMANDATION ***

Une instruction ne doit pas contenir plusieurs affectations (opérateurs =, +=, -=, *=, /=) ;

(Exemple) Éviter par exemple :

```
      int a = 3;
      int b = 2;
      a -= b /= 3;
```

*** RECOMMANDATION ***

Éviter d'utiliser l'opérateur ternaire ' ? :'. Utiliser plutôt une structure de type "if".

(Pourquoi ?) L'opérateur ternaire « ? :» gêne la lisibilité du code.

*** RECOMMANDATION ***

Une valeur numérique ne doit pas apparaître en clair dans un .cc (sauf 0). Si son usage est nécessaire, elle doit être affectée à une variable constante déclarée extern dans le « .hh » et définie dans le « .cc » correspondant.

*** IMPÉRATIF ***

Lorsque de nombreuses valeurs numériques doivent, être définies, il faut qu'elles le soient dans un fichier texte ASCII lu lors de l'exécution.

*** AMÉLIORATION ***

L'usage d'une valeur numérique immédiate (différente de 0, true ou false) dans le source est la plupart du temps révélatrice d'une difficulté de conception.

*** IMPÉRATIF ***

Utiliser les opérateurs new et delete pour gérer dynamiquement la mémoire. Ne pas utiliser malloc, realloc...

(Exemple) «new» et «delete» respectent mieux l'abstraction des données et offrent les mêmes possibilités que malloc. «new» revoie un pointeur typé, contrairement à alloc. «new» permet l'appel au constructeur.

*** RECOMMANDATION ***

Utiliser « typedef » pour manipuler des types de données non élémentaires.

(Pourquoi ?) Déclarer un tableau de pointeurs sur fonction sans utiliser « typedef ».

*** RECOMMANDATION ***

Une instance ou une méthode ne doit pas être déclarée extern plus d'une fois. Elle est déclarée, au besoin, dans le fichier d'entête (« .hh ») associé au fichier de déclaration (« .cc ») contenant la déclaration réelle.

(Pourquoi?) FIXME

*** RECOMMANDATION ***

Le corps des fonctions inline est défini dans le fichier d'entête.

(Pourquoi ?) Le corps d'une fonction doit être disponible lorsqu'il est inséré.

*** AMÉLIORATION ***

Les templates doivent être définis inline.

(Pourquoi ?) Le corps d'un template doit être disponible lorsqu'il est fixé pour un type particulier. Contrairement à ce que cette architecture suggère, les templates ne sont pas implémentes comme une interprétation , mais elles sont instanciées pour chaque type nécessaire. Ceci permet entre autres d'utiliser facilement l'éditeur de liens standard.

Chapitre 4 - Algorithmique

Dans ce chapitre, nous allons présenter les recommandations les plus abstraites de ce guide. Elles concernent les choix algorithmiques motivant l'implémentation.

FIXME functer class

*** IMPÉRATIF ***

Ne jamais enchaîner plusieurs déréférencement de données membres :

(Exemple) Voilà où conduit un respect approximatif du modèle objet (J'ai déjà trouvé dans un projet comercialisé une série de 6 déréférencements comme ceux-ci...) :

```
titi.toto().->tata.tutu = 255;
```

*** IMPÉRATIF ***

Const doit être utilisé chaque fois que possible.

(Pourquoi ?) L'indication const permet de s'assurer qu'une donnée ne sera pas modifiée, même si elle est passée comme paramètre à une fonction dont le corps n'est pas connu ou si un pointeur sur son adresse est défini.

*** IMPÉRATIF ***

Toujours vérifier la valeur de retour d'un appel système.

(Comment ?) Dans un code bien encapsulé, le nombre de ces appels doit être réduit et ce contrôle ne devrait pas être fastidieux à mettre en oeuvre.

*** IMPÉRATIF ***

Lorsqu'un opérateur = est défini, s'assurer qu'une instance peut être affectée à elle-même.

(Pourquoi ?) Ceci demande un traitement particulier, surtout lorsque de la mémoire doit être libérée ou allouée dynamiquement lors d'une affectation.

(Exemple) Voici un exemple d'opérateur = redéfini qui fonctionne dans tous les cas sauf lorsqu'une instance de cette classe est affectée à elle-même :

```
class String
      {
      char* Data;
      long Size;
      // ...
      public :
      String& operator =(const String& _newValue)
            {
            delete[] Data;
            Size = _newValue.Size;
            Data = new char[Size];
            strcpy(Data,_newValue.Data);
            return *this;
            }
      };
```

Si une instance de la classe String définie ci-dessus est affectée à elle-même, alors sa zone de donnée est d'abord réallouée puis lue ensuite. Un contrôle de ce type permettrait de résoudre ce problème :

```
if(this == &_newValue)
      {
      return *this;
      }
else
      // ...
```

*** IMPÉRATIF ***

Ne jamais imposer une limite arbitraire d'implémentation.

(Exemple) Un des exemples les plus représentatifs est la limite de la commande un*x « tar » qui ne permet pas de manipuler des « path » de plus de 100 caractères. Ne pas écrire de fonctions qui présentent de telles limitations.

*** AMÉLIORATION ***

L'interface des classes doit être minimale.

(Comment ?) Le nombre de méthodes de chaque classe doit être minimal. Le nombre de paramètres que reçoit chaque méthode doit être minimal.

*** RECOMMANDATION ***

Lorsqu'une fonction détecte une erreur, elle doit la traiter et non pas simplement retourner un code d'erreur.

(Pourquoi ?) L'expérience a montré que les programmeurs testaient rarement les valeurs de retour des appels systèmes indiquant si ceux-ci se sont bien passés.

(Comment ?) Il faut adopter une démarche différente en concevant une interface : lorsqu'une méthode détecte un problème qui ne doit pas arriver, elle doit l'indiquer elle-même, au moins en mode debug. Il n'est pas suffisant de renvoyer un code d'erreur.

*** AMÉLIORATION ***

Lorsqu'une classe fonctionne de manière très dynamique, il peut être intéressant de garantir qu'elle a bien été allouée et initialisée à chaque utilisation.

(Comment ?) Un champ (de type long par exemple) maintient l'état de l'instance : allouée, initialisée et desallouée.

(Exemple) Une méthode simple pour utiliser ce système pour une classe est d'hériter de la classe Cookie mise en annexe. L'appel à la classe Cookie : :OK() valide l'existence de l'instance. Cet héritage peut être conditionné par NO_DEBUG.

*** RECOMMANDATION ***

Ne pas passer plus de quatre paramètres.

(Pourquoi ?) L'existence de fonctions contenant un trop grand nombre de paramètres indique un problème de conception. Elle favorise l'apparition d'erreurs lors de son utilisation (ordre des paramètres), augmente le couplage des interfaces, gêne la relecture et la maintenance du code source.

*** RECOMMANDATION ***

Ne jamais traiter directement la mémoire à l'aide d'un pointeur. En particulier, ne jamais utiliser memcpy, ni aucune autre fonction mem*.

(Pourquoi ?) Ces fonctions brisent le modèle objet sans apporter de possibilités supplémentaires.

(Exemple) Si une instance est recopiée avec memcpy et qu'un opérateur = est défini par la suite pour sa classe, celui-ci ne sera pas utilisé. Il l'aurait été si l'opérateur = généré par défaut par le compilateur avait été utilisé.

*** AMÉLIORATION ***

Ne pas définir de fonction dont certains paramètres ont une valeur par défaut.

(Pourquoi ?) En pratique, cela conduit à des appels de fonctions avec un nombre d'arguments variable. Cette souplesse apparente que confère au premier abord une telle interface se traduit plutôt par une confusion lors de l'utilisation de la classe.

(Exemple) Cette définition de construction de Point est -elle souhaitable ?

```
class Point
      {
      Point(int x=0, int y=0, int z=0);
      }

main()
      {
      Point a;
      Point b(1);
      Point c(5,6);
      Point d(6,6,6);
      }
```

Celle-ci, plus stricte semble préferable :

```
class Point
      {
      Point(void);
      Point(int x, int y, int z);
      }

main()
      {
      Point a;
      // Point b(1); // interdit
      // Point c(5,6); // interdit
      Point d(6,6,6);
      }
```

*** AMÉLIORATION ***

Eviter d'inserrer des données enum, #define ou global const int.

(Pourquoi ?) Génère trop d'inclusions des headers. FIXME.

Gestion d'un source maintenu pour plusieurs plateformes.

*** IMPÉRATIF ***

Dater l'année sur 4 chiffres.

(Pourquoi ?) Dans beaucoup de programmes la date est codée sur deux chiffres. Ils ne résisteront pas au passage à l'an 2000. Le lundi 3 janvier 2000 sera le lundi noir de l'informatique. Les visionnaires coderont l'année des dates de leurs programmes sur 5 chiffres ou plus. À propos, noter que l'an 2000 sera la dernière année du vingtième siècle.

*** AMÉLIORATION ***

Utiliser le format japonais pour indiquer une date : aaaa/mm/jj.

(Pourquoi ?) Ce format élimine les ambiguïtés entre le format français et le format anglo-saxon. De plus, le tri alphabétique de dates codées ainsi correspond au tri chronologique.

(Exemple) En gardant la logique actuelle, les dates vont bientôt s'écrire 03/07/02. S'agit-il du 3 juillet ou du 7 mars de l'an 2 ? Préférer : 2002/03/07.

*** RECOMMANDATION ***

Eviter les casts (transtypage/coercition), en particulier s'ils sont implicites.

(Pourquoi?) FIXME

(Exemple) FIXME

*** IMPÉRATIF ***

Ne jamais transtyper une donnée const en non-const.

(Pourquoi ?) Selon l'implémentation, il est possible que les données const soient stockées dans une zone où la lecture est possible, mais pas l'écriture. De plus ce type de transtypage indique une mauvaise maîtrise de la structure de données.

(Comment ?) Si une donnée membre doit pouvoir être modifiée dans une instance const, utiliser le mot clef «mutable» à sa déclaration.

*** RECOMMANDATION ***

Toutes les occurences d'une fonction surchargée doivent remplir le même objectif. Leur intérêt doit se limiter au masquage de difficultés d'implémentation.

(Pourquoi ?) Il est possible de faire coexister deux fonctions qui ont le même nom et qui sont différenciées par le type du paramètre reçu. Il n'est pas souhaitable que ces fonctions remplissent des buts très (ou, pire, légèrement) différents. La surcharge de fonctions est un art difficile à manier avec parcimonie et clairvoyance pour rester bénéfique.

*** RECOMMANDATION ***

Préférer les multiplications et divisions aux décalages de bits.

(Exemple) En particulier, l'effet d'un décalage à droite pour une valeur négative n'est pas défini [Stroustrup 1991].

*** RECOMMANDATION ***

Un pointeur doit toujours pouvoir être désalloué.

(Pourquoi ?) Ceci évite une gestion de flags lourde et génératrice d'erreurs. Pour remplir cet objectif, un pointeur doit, soit désigner une zone allouée, soit être nul. En effet les opérateurs delete et delete[] n'ont pas d'effet s'ils reçoivent un pointeur nul.

(Rappel) L'opérateur «delete» est utilisé pour désallouer une zone réservée pour une instance par l'opérateur «new». L'opérateur «delete[]» est utilisé pour désallouer une zone réservée pour un tableau d'instances par l'opérateur «new[]».

(Exception) La seule exception à cette règle concerne les tableaux codés en dur. (char buff[64] ;) Bien sûr il faut leur usage préférer un template comme ceux des STL.

*** RECOMMANDATION ***

Un pointeur sur char référençant une chaîne de caractères doit toujours désigner une zone allouée et valide.

(Pourquoi ?) FIXME

*** RECOMMANDATION ***

Ne rien supposer de la structure interne des données. D'une manière générale, ne jamais briser l'abstraction des données.

(Pourquoi ?) En particulier, les compilateurs peuvent réarranger l'ordre des champs d'une struct dans un souci d'optimisation. Le code doit toujours être complètement indépendant de ces détails d'implémentation [Stroustrup ? ?].

*** RECOMMANDATION ***

Identifier clairement chaque récursivité.

(Pourquoi ?) Cette algorithmique n'est pas courante en C++. Lors de la maintenance, un lecteur non averti pourra avoir des difficultés à comprendre l'arbre d'appel des fonctions si une récursivité a été utilisée, surtout si elle est indirecte et qu'elle n'est pas explicitement indiquée.

*** RECOMMANDATION ***

Ne pas déclarer de données «static» dans une fonction.

(Pourquoi?) FIXME

*** RECOMMANDATION ***

Il est à la charge de celui qui alloue de la mémoire de la désallouer.

(Exemple) Comment implémenter une fonction qui renvoie un message sous forme de chaîne de caractères ? La solution simple consiste à allouer dynamiquement un tableau de caractères, le remplir avec le message et renvoyer un pointeur sur la chaîne. Cependant, c'est alors celui qui reçoit le message qui doit détruire la zone allouée une fois que la donnée n'est plus nécessaire. Ce type de conception conduit facilement à une fuite d'espace mémoire (memory leak). Voici une meilleure solution à ce problème :

```
class TmpString
      {
      char* Data;
      public:
      TmpString(const char* const thatClone) { ... }
      TmpString(const TmpString& thatClone) { ... }
      operator char*(void) { return Data; }
      };
TmpString ClearErrorMessage(const long& thatErrorcode)
      {
      if(thatErrorCode == 1) return "real bad";
      else return "not too bad";
      }
```

```
int main(void)
      {
      // ...
      char* Buff = new
char[strlen(ClearErrorMessage(CurrentError))];
      strcpy(Buff,ClearErrorMessage(CurrentError));
      cout << "Error: " << ClearErrorMessage(CurrentError) <<
endl;
      return 0;
      }
```

FIXME exemple débile

Pointeurs

*** RECOMMANDATION ***

Il ne doit pas y avoir de pointeur dans le programme, mis à part dans quelques classes de base.

(Pourquoi ?) Les pointeurs invalides sont sans doute la plus grande cause d'erreurs des programmes écrits en C. Ils gênent l'abstraction du code source et obligent le développeur à maintenir à l'esprit une notion simplement technique. L'orientation objet permet d'éviter ces inconvénients.

(Comment ?) L'utilisation des trois classes suivantes permet de programmer sans jamais utiliser de pointeur ni d'allocation dynamique directement : (bien sûr, ceci ne s'applique pas aux cas particuliers comme la programmation système bas niveau)

- String pour gérer les chaînes de caractères ;
- Vector pour gérer les tableaux ;
- List pour gérer les listes chaînées.

Ces classes font partie de la STL (Standard Template Library) qui ont été normalisées fin 1995 avec le C++.

J'ai personnellement écrit plusieurs projets de + de 10.000 lignes qui ne contenaient des pointeurs que dans ces classes de base, dûment testées.

Structures de contrôle

*** RECOMMANDATION ***

Ne pas utiliser plus de quatre niveaux d'imbrication d'accolades.

*** AMÉLIORATION ***

Respecter les règles de la programmation structurée : éviter les mots-clefs :

- break ;
- continue ;
- goto ;
- Éviter aussi l'emploi de plusieurs return dans une fonction.

(Exception) Si un switch doit être utilisé, alors un "break" doit terminer chaque structure "case".

*** AMÉLIORATION ***

L'usage de switch n'est pas recommandé d'une manière générale. Certaines exceptions comme l'identification de touches subsistent cependant.

(Pourquoi ?) La structure logique du switch repose directement sur le « goto » et les labels et ne respecte pas l'idée admise de la programmation structurée. L'implémentation, à cause des « break » en particulier, est une source d'erreurs.

(Comment ?) Un algorithme d'exécution conditionnelle fondé sur un tableau de pointeur sur fonctions est plus évolutif, plus propre, moins sujets aux difficultés de maintenance. Accessoirement, son exécution est aussi beaucoup plus rapide qu'un switch.

Pour les cas simples, une structure fondée sur le test «else if» peut permettre de gérer proprement une succession de tests :

```
if(key == ENTER_KEY)
      // ...
else if(key == F1_KEY)
      // ...
else if((key >= 'a') && (key <= 'z'))
      // ...
else
      // Erreur...
```

(Exemple) Par exemple, il est possible qu'un polymorphisme prenne en charge de manière transparente une difficulté résolue par un switch. // FIXME kezako

*** IMPÉRATIF ***

Si une structure « switch » doit être utilisée, un « break » doit terminer chaque clause « case ».

*** RECOMMANDATION ***

Chaque switch doit se terminer par un label « default ».

*** AMÉLIORATION ***

L'usage de do {...} while(...) est inhabituel et n'est pas souhaitable.

*** AMÉLIORATION ***

Restreindre l'usage de la boucle «for» à une itération de 0 à une valeur fixée, avec une incrémentation de 1.

(Pourquoi ?) Ce genre de choix conduit le code à être self-explanatory. Aux dépens parfois de la satisfaction intellectuelle du développeur.

(Exemple)

```
for(int i = 0; i < size; i++)
      {
      cout.width(3);
      cout << i << ' ' << char(i) << endl;
      }
```

*** RECOMMANDATION ***

Lorsqu'un opérateur doit être surchargé et que son premier argument est constant, le définir comme une fonction et non comme une méthode.

(Pourquoi ?) La définition d'un opérateur comme méthode et non comme fonction ne permet pas le cast automatique du premier opérande par appel au constructeur, lors de l'utilisation de l'opérateur. Cependant, ceci présente un intéret lorsque le premier opérande est une rvalue (d'habitude, comme pour «+» ou «==», mais pas comme «=» ou «+=»).

(Exemple) Comparaison des deux techniques pour une classe String :

```
class String
      {
      char* Data;
      public:
      String(void) { Data = new char[1]; Data[0] = '\0'; }
      String(const long& thatLength)
            { Data = new char[thatLength+1]; Data[0] = '\0'; }
      String(char* const thatClone)
            {
            Data=new char[strln(thatClone+1)]
            strcpy(Data,thatClone);
            }
#if defined OPERATOR_IS_METHOD
      Integer operator +(const Integer& thatSecondOperand)
const
            {
            String returnValue(newchar
                  [strln(Value)+strln(thatSecondOperand)+1]);
            strcpy(returnValue.Data,Data);
            strcpy(returnValue.Data,thatSecondOperand);
            return returnValue;
            }
#endif // defined OPERATOR_IS_METHOD
      Integer
      }
```

FIXME exemple trop pourri

?. Optimisation

*** AMÉLIORATION ***

L'optimisation est un art difficile. Une horreur de programmation est souvent « justifiée » par un besoin d'optimisation. L'expérience montre que l'optimisation ne peut se faire efficacement sans profiler. D'une manière générale, très peu de fonctions occupent plus de 1% du temps de calcul. Si le programme doit réellement être optimisé, il faut identifier ces fonctions. D'une manière générale, il faut donc respecter le modèle implémenté, même au prix de quelques cycles de processeur.

*** AMÉLIORATION ***

Une portion de source répétée plus de deux fois, même avec de légères différences, indique une mauvaise conception.

*** AMÉLIORATION ***

Affiner le modèle objet plutôt qu'utiliser des ruses de programmation pour optimiser globalement l'exécution.

(Exemple) Voir L'implémentation du noeud de liste doublement chaînée mise en annexe. La place qu'occupe cette gestion de liste en mémoire est minimale. La taille du code aussi. Enfin, le temps d'exécution est proche de l'optimum : le code ne contient aucun branchement conditionnel (if, ? :) et aucune boucle. De plus, il est «inline». Ceci sans compromis par rapport au modèle objet et sans technique obscure d'optimisation.

?. Développement multi-plateformes

*** IMPÉRATIF ***

Isoler le code source contenant des parties dont la compilation est optionnelle pour des besoins de portages. Ces fichiers doivent contenir tout ce code et uniquement ce code.

Chapitre 5 - Mise en œuvre

"La différence entre la théorie et la pratique est plus importante en pratique qu'en théorie."
Sagesse populaire logicienne.
"C'est pratiquement vrai. En théorie, du moins."
Robert.
"Et réciproquement."
Lulu.

Compilation

*** IMPÉRATIF ***

Il faut compiler en demandant au compilateur d'indiquer tous les avertissements (warnings) et toutes les erreurs. Il faut les éviter tous, sauf lorsque le message est lié à une limitation du compilateur.

(Exemple) Voici par exemple les options de compilation à passer à g++ pour qu'il indique tous les warnings. Les deux derniers ne sont pas utilisables avec iostream.h de la libg++ 2.7.0 :

g++ : -Wall -Wpointer-arith -Wbad-function-cast -Wcast-qual -Wcast-align -Wwrite-strings -Wconversion -Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations -Wnested-externs -Winline -Wsynth -Wredundant-decl -Wshadow

Pour cfront et ses descendants plus directs, l'option «+w» permet généralement d'obtenir tous les avertissements.

*** IMPÉRATIF ***

Une classe ayant une méthode virtuelle doit avoir un destructeur virtuel.

(Pourquoi ?) La définition de méthodes virtuelles sert à implémenter le polymorphisme sur les classes de base. À l'utilisation, un pointeur sur classe de base peut en fait désigner une instance de classe dérivée. Lors de la destruction de cet objet, pour que le bon destructeur soit appelé, il faut que ceux des classes de base soient virtuels.

*** AMÉLIORATION ***

Utiliser plusieurs compilateurs pour compiler le même code.

(Pourquoi ?) Alors que les compilateurs C sont stabilisés et fiables depuis plusieurs années, les compilateurs C++ ne le sont pas. Ils peuvent en particulier laisser passer une instruction invalide. Ils peuvent aussi générer un code invalide (cf. exemple). Compiler un exécutable avec plusieurs compilateurs/sur plusieurs machines est implicitement un moyen de valider la portabilité et la modularité du code.

*** AMÉLIORATION ***

Il est interdit d'affecter un tableau d'intances à un pointeur sur classe abstraite dont hérite la classe de ces instances. A l'exécution, le compilateur ne trouve pas la table d'indirection.

Attention : le compilateur ne prévient pas lorsque ce problème arrive.

(Exemple) En testant 4 compilateurs sur 3 machines différentes (CC sun ancien, gcc 2.7.0, CC silicon 4 et 5 et borland) tous compilaient le code suivant sans warning, mais aucun ne produisait d'exécutable utilisable !

```
#include <iostream.h>
class B
    {virtual ~B() {}};
class D:public B
    {int i; D(){i=1;} virtual ~D(){cout<<i<<endl;}};
int main(void)
      {
      B* pb = new D[2];
      delete[] pb;
      return 0;
      }
```

*** RECOMMANDADTION ***

Utiliser des outils de sécurisation du code.

Le C++ est un outil d'implémentation puissant, mais où une simple faute d'attention peut engendrer une erreur aléatoire très difficile à localiser. Pour contrer cette situation, il faut appliquer des outils de vérification qui se révèlent très performants à l'usage.

(Exemple)

lint++ : vérification du code source plus poussée que celle du compilateur.

Purify : vérification de l'utilisation de la mémoire durant l'exécution.

Bibliographie

[Dijkstra 1972]

Structured programming
O-J Dahl, E. W. Dijkstra, C. A. R. Hoare
220 pages. Anglais.
Academic press. 11ème édition. ISBN 0-12-200550
FIXME

[Ellemtel 1992]

Programming in C++. Rules and Recommandations.
Ellemtel corporation
erik.nyquist@eua.ericsson.se
mats.henric@eua.ericsson.se
88 pages. Anglais.
Document assez complet, dont le but est proche de celui-ci. Il s'adresse à des programmeurs plus spécialisés. Certaines préconisations sont proches de celles de ce document sous de nombreux aspects. Comprend moins de règles et plus d'exemples.

[Stroustrup 1991]

The C++ programming language. 2nd edition. Corrections 1995.
Bjarne Stroustrup
699 pages. En anglais américain.
Addison Wesley Publishing Company
ISBN 0-201-53992-6
699 pages
Seconde édition de la référence du C++. Unique ouvrage à utiliser pour apprendre le C++. Cependant, quelques ajouts au langage manquent (exceptions, type bool).

[Stroustrup 1992]

The Annotated C++ Reference Manual
Margaret A. Ellis - Bjarne Stroustrup
Addison Wesley Publishing Company
ISBN 0-201-51459-1
461 pages

[Stroustrup 1995]

The design and evolution of C++
Bjarne Stroustrup
Addison Wesley Publishing Company
ISBN 0-201-54330-3
461 pages

[Code complete 19??]

Code Complete
Steve McConnel
Microsoft Press
ISBN 1-55615-484-4
859 pages

C++ FAQ
usenet : comp.lang.c++

[Adams 1978]
The Hitch Hiker's guide to the galaxy
Douglas Adams
ISBN 0-330-25864-8
159 pages

Gnu recommandations

Meyer
Characteristics of Sotware Quality, Boehm et al.

Annexe A - Application du guide au code c

Paradoxalement, la programmation en C selon les standards efficaces algorithmiques unanimement reconnus demande une plus grande maîtrise de l'orientation objet et de la syntaxe que la programmation en C++.

Dans certains cas, préférer le C au C++ est compréhensible : besoin de portabilité sur des plateformes possédant un compilateur C, mais pas de compilateur C++ (autre raison ?).

Tous les mécanismes présentés ici peuvent être portés au C au prix d'un effort plus ou moins important. En fait, cet effort est raisonnable dans la plupart des cas. L'exemple fondamental est l'implémentation du mécanisme de classe. Voici un exemple : une struct String. FIXME Data length.

```
struct String
	{
	char* Data;
	};
void StringVoidConstructor(String thatConstructedString)
	{
	thatConstructedString.Data = malloc(sizeof(char));
	thatConstructedString.Data[0] = '\0';
	}
void StringCopyConstructor(String thatClone)
	{
	thatConstructedString.Data =
malloc(strlen(thatClone.Data)+1);
	strcpy(thatConstructedString.Data,thatclone.Data);
	}
void StringCharStarConstructor(char* thatPseudoClone)
	{
	thatConstructedString.Data =
malloc(strlen(thatPseudoClone)+1);
	strcpy(thatConstructedString.Data,thatPseudoClone);
	}
void StringDestructor(String thatStringToDestroy)
	{
	free(ThatStringToDestroy.Data);
	}
String Concat(String thatStart, String thatEnd)
	{
	String ReturnValue;
	ReturnValue.Data =

	malloc(strlen(thatStart.Data)+strlen(thatEnd.Data)+1;
	ReturnValue.Data = strcpy(thatStart.Data);
	ReturnValue.Data = strcat(thatEnd.Data);
	return ReturnValue;
	}
char* CharStarCast(String thatStringToCast)
	{
	return thatStringToCast.Data;
	}
```

Du code source en c et en c++ peuvent créer un exécutable.

Les différents fichiers objets créés (*.o) peuvent être liés entre eux lors du link.

(Exemple) Voici un exemple minimal d'utilisation mutuelle de c et de c++.

Voici une classe dont une méthode appelle une fonction c :

```
#include <iostream.h>

extern "C"
      {
      int IAmACFunction(int i);
      void InstanciateAA(void);
      ThisCFunctionInstanciateAnObject();
      }

class A
      {
      public:
      A(void)
            {
            cout << "A born" << endl;
            }
      };


void InstanciateAA(void)
      {
      new A;
      }

int main(void)
      {
      IAmACFunction(666);
      ThisCFunctionInstanciateAnObject();
      return 0;
      }
```

Voici une fonction c qui provoque la construction d' un objet c++ en appelant une fonction définie dans le code c++ :

```
#include <stdio.h>

extern InstanciateAA();

int IAmACFunction(i)
      int i;
      {
      printf("IAmACFunction that received : %d\n »,i) ;
      }

ThisCFunctionInstanciateAnObject()
      {
InstanciateAA() ;
      }
```

Annexe B - Comment lier des modules C et C++ au sein d'un même exécutable

FIXME explications

```
################################################################################
#
# File : Makefile
#
# Birth : 1996/08/13
#
# Version : 1996/08/13
#
# Purpose : To show how to mix C and C++ files.
#
# Author : Timothee Royer
#
################################################################################

CC=cc
CXX=g++
CFLAGS=+w
CXXFLAGS=-Wall
CXX_STD_INCLUDES=-I   /usr/include/CC   -I   /opt/outils/gnu/lib/g++-include   -
I/usr/include
SRCS=titi.c toto.cc
OBJS=titi.o toto.o
EXEC_NAME=tata

################################################################################

$(EXEC_NAME) : $(OBJS)
	$(CXX) -o $@ $(OBJS)

depend :
	makedepend $(CXX_STD_INCLUDES) -- $(SRCS)

clean :
	rm -f *.o core $(OBJS) $(EXEC_NAME)

# DO NOT DELETE THIS LINE -- make depend depends on it.

titi.o: /usr/include/stdio.h titi.h
toto.o: /opt/outils/gnu/lib/g++-include/iostream.h
toto.o: /opt/outils/gnu/lib/g++-include/streambuf.h
toto.o: /opt/outils/gnu/lib/g++-include/libio.h
toto.o: /opt/outils/gnu/lib/g++-include/_G_config.h toto.hh titi.h
```

```
/******************************************************************************
**
** File : titi.h
**
** Birth : 1996/08/13
**
** Version : 1996/08/13
**
** Purpose : To show how to mix C and C++ files.
**
** Author : Timothee Royer
**
******************************************************************************/

#if defined(titi_RECURSES)
#error Recursive header files inclusion detected in titi.h
#else /* defined(titi_RECURSES) */
#define titi_RECURSES

#if !defined titi_hh
#define titi_hh

/******************************************************************************/

#if defined(__cplusplus)
extern "C"
    {
#endif /* defined(__cplusplus) */

void InstanciateAA(void);
void IAmACFunction(int i);
void ThisCFunctionInstanciateAnObject(void);

#if defined(__cplusplus)
    }
#endif /* defined(__cplusplus) */

/******************************************************************************/

#endif /* !defined titi_hh */

#undef titi_RECURSES
#endif /* esle defined(titi_RECURSES) */
```

```
/*****************************************************************************
**
** File : titi.h
**
** Birth : 1996/08/13
**
** Version : 1996/08/13
**
** Purpose : To show how to mix C and C++ files.
**
** Author : Timothee Royer
**
*****************************************************************************/

#include <stdio.h>

#include "titi.h"

void IAmACFunction(i)
     int i;
    {
    printf("IAmACFunction that received : %d\n",i);
    }

void ThisCFunctionInstanciateAnObject()
    {
    InstanciateAA();
    }
```

```
////////////////////////////////////////////////////////////////////////////////
//
// File : toto.hh
//
// Birth : 1995/08/21
//
// Version : 1996/08/13
//
// Purpose : To show how to mix C and C++ files.
//
// Author : Timothee Royer
//
////////////////////////////////////////////////////////////////////////////////

#if defined(toto_RECURSES)
#error Recursive header files inclusion detected in toto.hh
#else // defined(toto_RECURSES)
#define toto_RECURSES

#if !defined toto_hh
#define toto_hh

////////////////////////////////////////////////////////////////////////////////

extern "C"
    {
    void InstanciateAA(void);
    }

class A
    {
  public:
    A(void);
    };

////////////////////////////////////////////////////////////////////////////////

#endif // !defined toto_hh

#undef toto_RECURSES
#endif // esle defined(toto_RECURSES)
```

```
///////////////////////////////////////////////////////////////////////////////
//
// File : toto.cc
//
// Birth : 1995/08/21
//
// Version : 1996/08/13
//
// Purpose : To show how to mix C and C++ files.
//
// Author : Timothee Royer
//
///////////////////////////////////////////////////////////////////////////////

#include <iostream.h>

#include "toto.hh"
#include "titi.h"

///////////////////////////////////////////////////////////////////////////////

A::A(void)
    {
    cout << "A born." << endl;
    }

void InstanciateAA(void)
    {
    A someA;
    }

///////////////////////////////////////////////////////////////////////////////

int main(void)
    {
    IAmACFunction(666);
    ThisCFunctionInstanciateAnObject();
    return 0;
    }

///////////////////////////////////////////////////////////////////////////////
```

Annexe C - Modèle de fichiers sources

Voir les fichiers XXX.hh et XXX.cc joints.

Ils contiennent des « fonds » de classe. Pour les utiliser, il est recommandé de les recopier dans un répertoire où l'on souhaite écrire une nouvelle classe. Remplacer les XXX par le nom de la classe. Remplir les champs de commentaire. Ajouter des méthodes au besoin.

```
/////////////////////////////////////////////////////////////////
//                                                             //
// File name : XXX.hh
//
// Creation : 19??/??/??
//
// Version : 19??/??/??
//
// Author : ??
//
// History :
//    19??/??/?? : Mr ?Name? : ?What?
//
// Rcs Id : "@(#)class XXX declaration."
//                                                             //
/////////////////////////////////////////////////////////////////

#if defined XXX_CYCLE
#error Header cyclic inclusion detected in XXX.hh
#else // defined XXX_CYCLE
#define XXX_CYCLE

#if !defined XXX_hh
#define XXX_hh

/////////////////////////////////////////////////////////////////
//                                                             //

#include <iostream.h>

/////////////////////////////////////////////////////////////////
// class XXX
/////////////////////////////////////////////////////////////////

class XXX
      {

      public :

      // Standard services

      ~XXX(void);

      // Interface

      void SelfDisplay(ostream& thatStream) const;
      bool OK(void) const;

      private :

      // Datas

      // Hidden services

      inline XXX(void);
      inline XXX(XXX&);
      inline operator =(XXX&);

      // Internals

      };

inline ostream& operator <<(ostream& thatStream, const XXX&
thatObjectToDisplay);
```

```
ostream& operator <<(ostream& thatStream, const XXX&
thatObjectToDisplay)
      {
      thatObjectToDisplay.SelfDisplay(thatStream);
      return thatStream;
      }

//                                                              //
//////////////////////////////////////////////////////////////////

#endif // !defined XXX_hh

#undef XXX_CYCLE
#endif // else defined XXX_CYCLE
```

```
/////////////////////////////////////////////////////////////////
//                                                             //
// File name : XXX.cc
//
// Creation : 19??/??/??
//
// Version : 19??/??/??
//
// Author : ??
//
// email : @
//
// Purpose : ??
//
// Distribution :
//
// Use :
//    ??
//
// Todo :
//    O ??
//
// History :
//    19??/??/?? : Mr ?Name? : ?What?
//                                                             //
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
//                                                             //

#include "XXX.hh"

#if defined(NO_DEBUG)
#define ASSERT(x)
#else //defined(NO_DEBUG)
#define ASSERT(x) if(!(x)) \
    { cerr << "Assertion failed : (" << #x << ')' << endl \
    << "In file : " << __FILE__ << "at line #" << __LINE__ <<
endl \
    << "Compiled the " << __DATE__ << " at " << __TIME__ <<
endl; abort();}
#endif // else defined(NO_DEBUG)

const char* const XXX_RCS_ID = "@(#)class XXX definition.";

/////////////////////////////////////////////////////////////////
// class XXX
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// Standard services - public :

XXX::~XXX(void)
      {
      }

/////////////////////////////////////////////////////////////////
// Interface - public :

void XXX::SelfDisplay(ostream& thatStream) const
      {
      thatStream << "< class XXX >" << endl;
      }
```

```
bool XXX::OK(void) const
      {
      return true;
      }

/////////////////////////////////////////////////////////////////
// Internals - private :

//                                                             //
/////////////////////////////////////////////////////////////////
```

Annexe D - Exemple de classe : un noeud de liste doublement chaînée

FIXME add template
FIXME ordre méth
FIXME pas de for, if, else, switch.
FIXME si pas de templates T* -> void*

```
////////////////////////////////////////////////////////////////
//
// File name : DLLNode.hh
//
// Creation date : 1995/06/??
//
// Version : 1995/09/29
//
// Author : Timothe'e Royer
//
// Email : tim@puff.frmug.fr.net
//
// Purpose :
//
// Distribution : Free without warranty.
//
// Todo :
//     X templates
//     O ajouter un tri.
//     O ajouter un scanner.
//
// History :
//     1995/09/28 : suppr. SetNext()/SetPrev()
//     1995/09/29 : templates
//
////////////////////////////////////////////////////////////////

#if !defined(DLLNode_hh)
#define DLLNode_hh

static const char* const DLLNodeId
      = "@(#)class DLLNode declaration and definition";

////////////////////////////////////////////////////////////////
//

template <class T> class DLLNode
      {

      public :

      // Standard

      inline DLLNode(void);
      inline ~DLLNode(void);

      // Services

      inline T& GetNext(void) const;
      inline T& GetPrev(void) const;
      inline void Insert(T& ThatNewNode);
      inline void Remove(void);

      private :

      // Datas

      T* Prev;
      T* Next;

      // Hidden

      inline DLLNode(DLLNode&);
      inline operator =(DLLNode&);
```

```
	};

//////////////////////////////////////////////////////////////
// Standard
//////////////////////////////////////////////////////////////

template <class T> DLLNode<T>::DLLNode(void) : Prev((T*)this),
Next((T*)this)
	{
	}

template <class T> DLLNode<T>::~DLLNode(void)
	{
	Remove();
	}

// Services
//////////////////////////////////////////////////////////////

template <class T> T& DLLNode<T>::GetNext(void) const
	{
	return *Next;
	}

template <class T> T& DLLNode<T>::GetPrev(void) const
	{
	return *Prev;
	}

template <class T> void DLLNode<T>::Insert(T& ThatNewNode)
	{
	ThatNewNode.Prev = (T*)this;
	ThatNewNode.Next = Next;
	Next->Prev = &ThatNewNode;
	Next = &ThatNewNode;
	}

template <class T> void DLLNode<T>::Remove(void)
	{
	Prev->Next = Next;
	Next->Prev = Prev;
	Next = (T*)this;
	Prev = (T*)this;
	}

//
//////////////////////////////////////////////////////////////

#endif // !defined(DLLNode_hh)
```

```
//////////////////////////////////////////////////////////////
//                                                          //
// File name : DLLNodeTester.cc
//
// Creation : 19??/??/??
//
// Version : 19??/??/??
//
// Author : ??
//
// email : @
//
// Purpose : Test class DLLNode. Illustrate its use &
interface.
//
// Distribution :
//
// Use :
//    ??
//
// Todo :
//    O ??
//
// History :
//    19??/??/?? : Mr ?Name? : ?What?
//                                                          //
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
//                                                          //

#include "DLLNodeTester.hh"

#if defined(NO_DEBUG)
#define ASSERT(x)
#else //defined(NO_DEBUG)
#define ASSERT(x) if(!(x)) \
    { cerr << "Assertion failed : (" << #x << ')' << endl \
    << "In file : " << __FILE__ << "at line #" << __LINE__ <<
endl \
    << "Compiled the " << __DATE__ << " at " << __TIME__ <<
endl; abort();}
#endif // else defined(NO_DEBUG)

const char* const DLLNodeTester_RCS_ID = "@(#)class DLLNode
tester.";

//////////////////////////////////////////////////////////////
// class DLLNode tester
//////////////////////////////////////////////////////////////

#include <string.h>

class Man : public DLLNode<Man>
      {
      public :
      Man(const char* const creationName);
      ~Man(void);
      const char* Name(void) const;

      private :
      // Data
      char* DataName;
      // Hidden
```

```
      inline Man(const Man&);
      inline void operator =(const Man&);
      };

Man::Man(const char* const creationName)
      {
      strdup(DataName,creationName);
      }

Man::~Man(void)
      {
      delete DataName;
      }

const char* Man::Name(void) const
      {
      return DataName;
      }

////////////////////////////////////////////////////////////////
//                                                            //

int main(const int, const char** const, const char** const)
      {
      Man titi("titi");
      Man toto("toto"); // FIXME stress test
      return 0;
      }

//                                                            //
////////////////////////////////////////////////////////////////
```

Annexe E - Exemple de sécurisation : La classe Cookie

FIXME explications
FIXME Tester

```
////////////////////////////////////////////////////////////////
//
// Filename : Cookie.hh
//
// Directory : $(HOME)/Classes/Cookie
//
// Creation : 1995/05/19
//
// Version : 1995/05/19
//
// Author : Timothee Royer
//
// Purpose : Base class -> inheritance.
//
// Use : Simply inherit from this class. Use Cookie : :OK() to
know if you are
//    allocated.
//
////////////////////////////////////////////////////////////////

#if !defined(Cookie_hh)
#define Cookie_hh

#include <iostream.h>

// I use #define here cause I got no .cc file

#define COOKIE_ALLOCATED      0xBADF0E
#define COOKIE_CONSTRUCTED    0xF001BABE
#define COOKIE_DELETED        0xDEADBEEF
#if defined(NO_DEBUG)
#define ASSERT(x)
#else //defined(NO_DEBUG)
#define COOKIE_ASSERT(x) if(!(x)) \
    { cerr << "Cookie assertion failed : (" << #x << ')' <<
endl \
    << "In file : " << __FILE__ << "at line #" << __LINE__ <<
endl \
    << "Compiled the " << __DATE__ << " at " << __TIME__ <<
endl; abort();}
#endif // else defined(NO_DEBUG)

class Cookie
      {

      public :

// Standards

      Cookie(void)
            {
            Data = COOKIE_CONSTRUCTED;
            }

      ~Cookie(void)
            {
            COOKIE_ASSERT(Data != COOKIE_CONSTRUCTED);
            Data = COOKIE_DELETED;
            }

// Interface

      void Construct(void)
            {
```

```
            Data = COOKIE_CONSTRUCTED;
            }

      bool OK(void) const
            {
            ASSERT(COOKIE_ALLOCATED)
            return 1;
            }

      private :

// Data
      int Data;

// Hidden
      Cookie(const Cookie&);
      operator =(const Cookie&);

      };

#undef COOKIE_ALLOCATED
#undef COOKIE_CONSTRUCTED
#undef COOKIE_DELETED

#endif // !defined(Cookie_hh)
```

Annexe F - Lexique

- Macro

Une macro est l'équivalent d'une fonction, mais elle est traitée par le préprocesseur. Elle est définie avec la directive « #define ». Exemple classique de définition de macro :

```
#define MAX(x,y) {(x)>(y) ?(x) :(y)}
```

À l'utilisation le code :

```
toto = MAX(titi,10) ;
```

Sera en fait remplacé par celui-ci, avant la compilation :

```
toto = {(titi)>(10) ?(titi) :(10)} ;
```

L'usage des macros n'est pas recommandé.

- Complétion FIXME
  - Instance
  - Assert
  - Transtypage/coercition : cast
  - extern
  - inline

Une fonction ou une méthode peut être déclarée inline. Dans ce cas, lors de la construction du code exécutable, le corps de cette fonction est inséré à chaque appel. Le code généré est plutôt plus gros et plus rapide.

- mutable

Donnée membre modifiable même lorsqu'elle appartient à une instance constante.

- profiler

Un profiler est outil qui permet de déterminer comment est consommé le temps machine lors de l'exécution d'un programme. Son usage est indispensable pour toute optimisation.

- transtyper
- surcharger
- définition
- Déclaration
- Dépendant de l'implémentation :
- invariant
- assertion
- polymorphisme
- lvalue
- rvalue

Annexe G - Historique

1995/08/23
Semblant de mise en chapitres

1995/08/31
Accents ! (merci Mop) à â é è ê ë ï ô ù ç

1995/09/06
Justification !

1995/09/08
wc (nb. de lignes, mots et caractères) : 1586 7551 51981

1995/09/11
(0)(1)(2) -> *** IMPÉRATIF *** [...] [...]
Début de lexique.

1995/09/12
Début de plan !
wc : 1850 8424 60053

1995/09/15
Décimation des FIXME
wc : 2015 9560 67241
Nb 2 règles : imp :50 | rec :62 | amé :37 + (Pq ?) :72 | (Ex) :187 ? | Fix :87

1995/09/19
Nouvelle présentation des règle : nivo ness °/ 1 l. à part.
wc : 2339 11002 76095
Première beta version délivrée.

1995/10/01
#FIXME : 44 -> 14
wc : 2898 13887 97344
Deuxième beta version délivrée

1996/07/04
Version Word
Ennopncés grisés.
P: 96 - l:2668 - parag:1916 - mots:16644 - cars:92053

1996/08/14
Corrections != / add Mix c & c++ / nouvox FIXME
81 pages, 18770 mots, 104608 car, 2273 parag, 3135 lignes

1997/02/21
Le guide intéresse quelques centres de développement de l'armée de terre française.
Le guide est proposé freeware sur internet (www.mygale.org/00/jebdoun).
Le guide est référencé dans plusieurs (+- 15) moteurs de recherche mondiaux et francophones. (Hier)
Annonce de la présence du guide sur fr.comp.soft.objet. (Hier).

Le mot « norme » n’apparaît plus dans le document.
Réécriture du résumé et d’une introduction.
Nouveaux entête et pied de page.

ATTENTION. MERCI DE NE PAS FAIRE CIRCULER CE DOCUMENT QUI EST UNE BETA VERSION. LE TEXTE FINAL SERA MIS EN LIBRE DISTRIBUTION, APRES UNE PERIODE DE TESTS SUFFISANTE. SI VOUS ETES INTERESSES PAR CE DOCUMENT ET EN PARTICULIER SI VOUS SOUHAITEZ ETRE BETA-TESTEUR, CONTACTEZ-MOI DIRECTEMENT A tim@puff.frmug.fr.net.

AVERTISSEMENT. CE GUIDE EST FOURNI TEL QUEL, SANS AUCUNE GARANTIE EXPLICITE OU IMPLICITE. EN PARTICULIER, L'AUTEUR DÉCLINE TOUTE RESPONSABILITÉ QUANT AUX CONSÉQUENCES DE L'USAGE DE CE GUIDE.

MERCI DE M'AIDER À RÉALISER CE DOCUMENT EN ME FAISANT PARVENIR VOS CORRECTIONS, VOS IMPRESSIONS, LES BESOINS DE PRÉCISIONS ET SURTOUT VOS PROPOSITIONS D'AMÉLIORATIONS. EN PARTICULIER, LES VOLONTAIRES POUR M'AIDER À COCHER LES LIGNES DU TODO CI-DESSOUS SONT LES BIENVENUS.

CE GUIDE EST EN DÉVELOPPEMENT ET ÉVOLUE RAPIDEMENT, JE VOUS INVITE DONC À ME RETOURNER VOS IDÉES RAPIDEMENT, POUR ÉVITER DES DOUBLES CORRECTIONS OU UN TRAVAIL INUTILE SUR UN TEXTE DÉPASSÉ.

Author : Timothée Royer
Création : 1995/07/21 (< ?)
Version : 1995/09/29
Todo : (Légende o : à faire | / : commencé | x : fait)
o Script -> instancier fichier exemple
o Filtre yacc/lex.
/ Redéfinir la table des matières.
x Différencier C/C++.
/ Describe Implementation dependant features / multi-plateform dvpt
o numérotation des règles
x Marques de révision
x Warn op = != const/copy
o Reprendre les exemples et les explications
/ Bibliographie : relire, indiquer refs et écrire descr.
x Améliorer la présentation de ce texte : lignes vides, titres...
o include C++ ISO std working paper recomm°
, Beta testeurs
o cf. gnu licence
/ Lexique
o Traduction en anglais
o Index
o Améliorer la présentation
o Compléter le Todo
o Pour chaque règle : req. lvl, #, texte, (Pourquoi ?), (Rappel), (Comment ?), (Exception), (Exemple).

o Annexer 1 formulR de proposition de modificat° de la norme
o Implémenter un «lint» -> la norme
/ Écrire un XXXTester.cc
o Version courte / longue.
o Fournir des classes compatibles STL.

o Phrases + courtes
o Lister les textes de règles à part.
o Chapitre conclusion ?
x document maître + esclaves sous Word.

- string.h != strings.h
- taille fich < 10000 lignes
- num les règles ?
- utiliser long et non int dans les exemples.

o Extern "C"
o Enlever les "on" et les "nous".

- Traduire les citations
- ID : utiliser des gris != pour imperatif, recomm, amelior ? (cadre, police ?)
- ordre d'inclusion des .h => PB grave
- profiling
- liste de Cast sûrs ./ types de base
- expliquer #include "" != <>
- Héritage multiple
- pas de };
- Les params sont : soit &const (const pour type de base), soit & (pour être modifiés).
  (& pb appel -> cstr / copie)
- Nouvelle intro° : Règles emmergentes collectives / Choses à faire pour éviter les problèmes.
- Proposer template <T> MAX(const T&, const T&).... (!= #define MAX(x,y))
- Utiliser const & ref -> données mb.
- Utiliser des Accesseur getZ() et setZ() (!= T& setZ(void))
- Héritages multiples
- if(t==true) != if(t)
- Figures ? ? ?

Marques de révision : [#####] (*****) {%%%%%} <OOOOO>