



CLUB

COMMUNICATIONS ET SYSTÈMES

SYSTÈMES D'INFORMATIONS

DIRECTION ESPACE

Édition : **6** Date : **22/10/2000**

Révision : **9** Date : **04/03/2005**

Réf. : ESPACE/MS/CHOPE/CLUB/MU/001

Manuel d'utilisation de la bibliothèque CLUB

Rédigé par : L. Maisonobe, O. Queyrut G. Prat, F. Auguie, S. Vresk	le : CS SI/Espace/FDS	
Validé par : G. Prat	le : CS SI/Espace/FDS	
Pour application : C. Fernandez-Martin	le : CS SI/Espace/FDS	

Pièces jointes :

DIFFUSION INTERNE

Nom	Sigle	BPi	Observations
CS SI	ESPACE/FDS		2 exemplaires

DIFFUSION EXTERNE

Nom	Sigle	Observations
CNES	DCT/SB/MS	3 exemplaires

BORDEREAU D'INDEXATION

CONFIDENTIALITÉ : NC	MOTS CLEFS : bibliothèque, C++
-------------------------	--------------------------------

TITRE DU DOCUMENT :

Manuel d'utilisation de la bibliothèque CLUB

AUTEUR(S) : L. Maisonobe, O. Queyrut

G. Prat, F. Auguie, S. Vresk

RÉSUMÉ :

Ce document décrit la bibliothèque des CLasses Utilitaires de Base CLUB. Cette bibliothèque fournit des classes C++ implantant des utilitaires de gestion de chaînes de caractères, de gestion de texte, de gestion de données structurées dans des fichiers, de traitement d'erreurs, de gestion d'options et de partage de mémoire.

DOCUMENTS RATTACHÉS : ce document vit seul		LOCALISATION :	
VOLUME : 1	NBRE TOTAL DE PAGES : 147	DOCUMENT COMPOSITE : N	LANGUE : FR
	DONT PAGES LIMINAIRES : 3		
	NBRE DE PAGES SUPPL. :		
GESTION DE CONF. : Oui		RESP. GEST. CONF. : LUC MAISONOBE - CSSI	
CAUSES D'ÉVOLUTION : annonce des modifications apportées en version 10.0			
CONTRAT : Néant			
SYSTÈME HÔTE : Unix – L ^A T _E X 2 _ε			

MODIFICATIONS

Éd.	Rév.	Date	Référence, Auteur(s), Causes d'évolution
3	0	25/06/99	SCS/CLUB/MU/99-001, passage au format CS
3	1	12/08/99	SCS/CLUB/MU/99-001, documentation du passage du code d'erreur dans le TamponPartage de la classe BaseErreurs
3	2	15/10/99	SCS/CLUB/MU/99-001, changement de numéro de révision
3	3	30/03/00	SCS/CLUB/MU/99-001, utilisation de notation UML
4	0	04/08/00	SCS/CLUB/MU/99-001, ajout de la documentation de l'utilitaire difference
5	0	04/09/00	SCS/CLUB/MU/2000-001, passage en feuille de style notechope et utilisation de la STL dans CLUB
5	1	23/09/00	SCS/CLUB/MU/2000-001, utilisation de libtool pour générer club sous forme partagée
6	0	22/10/00	SCS/CLUB/MU/2000-001, description des classes UniqDataFile, MadonaFile, XMLFile, StructureFile et DataFile
6	1	21/11/00	SCS/CLUB/MU/2000-001, description des modifications de configuration entre les versions 8.0 et 8.1
6	2	04/12/00	SCS/CLUB/MU/2000-001, description des modifications de configuration entre les versions 8.1 et 8.2, ajout de la description de club-config
6	3	04/04/01	SCS/CLUB/MU/2000-001, description des modifications entre les versions 8.2 et 9.0, élimination de la classe Adressage, reconnaissance des réels fortran, corrections
6	4	22/06/01	ESPACE/MS/CHOPE/CLUB/MU/001, description des modifications entre les versions 9.0 et 9.1, ajout de la classe CallTrace, corrections
6	5	28/03/03	ESPACE/MS/CHOPE/CLUB/MU/001, description des modifications entre les versions 9.1 et 9.2, ajout de constructeurs, destructeurs
6	6	28/07/03	ESPACE/MS/CHOPE/CLUB/MU/001, description des modifications entre les versions 9.2 et 9.3
6	7	05/12/03	ESPACE/MS/CHOPE/CLUB/MU/001, description des modifications entre les versions 9.3 et 9.4
6	8	11/06/04	ESPACE/MS/CHOPE/CLUB/MU/001, description des modifications entre les versions 9.4 et 9.5
6	9	04/03/05	ESPACE/MS/CHOPE/CLUB/MU/001, description des modifications entre les versions 9.5 et 10.0

SOMMAIRE

GLOSSAIRE	5
DOCUMENTS DE RÉFÉRENCE	6
DOCUMENTS APPLICABLES	6
1 présentation	7
2 description du contexte	7
3 conventions	7
3.1 conventions de nommage	7
3.2 gestion des erreurs	8
3.3 utilisation de la STL	11
4 catalogue	15
4.1 classes disponibles	15
4.2 utilitaires disponibles	16
4.3 routines d'interfaçage	17
5 environnement	17
6 installation	18
7 messages d'avertissements et d'erreurs	20
8 tests	28

9	maintenance	28
9.1	portabilité	28
9.2	environnement de maintenance	29
9.3	installation de l'environnement de maintenance	29
9.4	compilation	29
9.5	procédures de maintenance	30
9.6	archivage	31
10	changements depuis les versions précédentes	31
10.1	évolutions entre la version 9.5 et la version 10.0	31
10.2	évolutions entre la version 9.4 et la version 9.5	32
10.3	évolutions entre la version 9.3 et la version 9.4	32
10.4	évolutions entre la version 9.2 et la version 9.3	32
10.5	évolutions entre la version 9.1 et la version 9.2	32
10.6	évolutions entre la version 9.0 et la version 9.1	33
10.7	évolutions entre la version 8.2 et la version 9.0	33
10.8	évolutions entre la version 8.1 et la version 8.2	34
10.9	évolutions entre la version 8.0 et la version 8.1	34
10.10	évolutions entre la version 7.0 et la version 8.0	34
10.11	évolutions entre la version 6.3 et la version 7.0	34
11	évolutions possibles	35
12	description des classes	36
12.1	classe AnalyseurLexical	36
12.2	classe BaseErreurs	40
12.3	classe ChaineSimple	45
12.4	classe CallTrace	49
12.5	classe ClubAllocHandler	51
12.6	classe ClubErreurs	52
12.7	classe DataFile	56
12.8	classe FichierStructure	59

12.9	classe FormatC	63
12.10	classe FormatFortran	64
12.11	classe IterateurCaractere	67
12.12	classe IterateurChamp	70
12.13	classe IterateurLigne	72
12.14	classe IterateurChampLigne	75
12.15	classe MadonaFile	78
12.16	classe OptionBase	79
12.17	classes dérivées de OptionBase	82
12.18	classe OptionsAppli	86
12.19	classe StructureFile	90
12.20	classe TamponAscii	92
12.21	classe TamponPartage	95
12.22	classe TamponTexte	97
12.23	classe Traducteur	103
12.24	classe UniqDataFile	106
12.25	classe Unit	116
12.26	classe XMLData	118
12.27	classe XMLFile	121
12.28	classe XMLUnits	123
13	description des utilitaires	125
13.1	difference	125
13.2	club-config	130
14	description des routines	132
14.1	guide d'utilisation	132
14.2	fonctions C	133
14.3	sous-routines FORTRAN	134

15 description des formats de fichiers	136
15.1 fichiers structurés	136
15.2 fichiers Madona	140
15.3 fichiers XML	140
15.4 fichiers d'unités XML	142

GLOSSAIRE

CLUB

CLasses Utilitaires de Base

CANTOR

Composants et Algorithmes Numériques Traduits sous forme d'Objets Réutilisables

DTD

Document Type Definition

MADONA

Moyens d'Accès à des DONnées Ascii

MARMOTTES

Modélisation d'Attitude par Récupération des Mesures d'Orientation pour Tout Type d'Engin Spatial

STL

Standard Template Library

XML

eXtensible Markup Language

DOCUMENTS DE RÉFÉRENCE

- [DR1] *MARMOTTES – documentation mathématique*
ESPACE/MS/CHOPE/MARMOTTES/DM/001 , édition 4.0, 01/02/2002
- [DR2] *MARMOTTES – manuel d'utilisation*
ESPACE/MS/CHOPE/MARMOTTES/MU/001 , édition 5.7, 04/03/2005
- [DR3] *Manuel d'utilisation de la bibliothèque CANTOR*
ESPACE/MS/CHOPE/CANTOR/MU/001 , édition 5.11, 04/03/2005
- [DR4] *PIMS – manuel utilisateur MADONA*
PIMS-MU-11-2010-CIS , édition 3.3, 19/05/2000
- [DR5] *Compilateurs – principes, techniques et outils*
A. AHO, R. SETHI, et J. ULLMAN, InterEditions, 1989
- [DR6] *GNU Coding Standards*
R. STALLMAN, 09/09/1996
- [DR7] *Libtool*
G. MATZIGKEIT, A. OLIVA, T. TANNER et G. VAUGHAN, édition 1.4.1, 06/04/2001
- [DR8] *Autoconf*
D. MACKENZIE et B. ELLISTON, édition 2.52, 17/07/2001
- [DR9] *Automake*
D. MACKENZIE et T. TROMEY, édition 1.5, 22/08/2001
- [DR10] *Version Management with CVS*
documentation de CVS 1.10
- [DR11] *GNU g++*
R. STALLMAN
- [DR12] *Programmer en langage C++*
DELANNOY Eyrolles 1999
- [DR13] *Le langage C++*
B. STROUSTRUP Addison-Wesley 1992
- [DR14] *Étude de la bibliothèque STL*
M. JULIEN CSSI 1999
- [DR15] <http://www.w3.org/XML>
- [DR16] <http://xml.apache.org/dist/xerces-c>

DOCUMENTS APPLICABLES

Néant

1 présentation

La bibliothèque CLUB regroupe des classes implantant des utilitaires C++ de bas niveau. Elle offre principalement des services liés aux entrées-sorties (fichiers et arguments de la ligne de commande, erreurs) et au texte (chaînes de caractères, traduction de messages et mots-clefs, analyse lexicale).

Ce document décrit la version 10.0 de la bibliothèque.

2 description du contexte

La bibliothèque CLUB contient des routines de bas niveau (son nom signifie CLasses Utilitaires de Base). Elle s'appuie sur les éléments des bibliothèques standards C++ et C et si elles sont disponibles elle peut également s'appuyer sur les bibliothèques XERCES et MADONA. La liste ordonnée des bibliothèques à spécifier à l'éditeur de liens varie selon le compilateur utilisé pour générer la bibliothèque et selon la disponibilité des bibliothèques optionnelles.

Les versions de CLUB ultérieures à la version 6.0 utilisent le mécanisme des exceptions, qui n'est mis en place correctement que si l'édition de liens est réalisée par le compilateur C++ qui a servi à compiler la bibliothèque. Cette condition impérative a un impact à l'exécution, pas lors de l'édition de liens elle-même, ce qui la rend insidieuse. En effet, en cas d'absence du mécanisme de récupération, aucune exception lancée ne peut pas être interceptée, le programme s'arrête alors sur une erreur fatale dès la première exception. Le résultat typique est une violation mémoire avec génération d'un fichier `core`). Il faut noter également que le mélange des compilateurs (SUN et GNU par exemple) ne fonctionne pas.

Les bibliothèques MARMOTTES [DR1], CANTOR [DR3] ainsi que des d'autres bibliothèques internes du CNES s'appuient toutes sur CLUB.

3 conventions

Un certain nombre de principes ont été respectés lors de la conception et du développement de cette bibliothèque. Ces principes sont placés en exergue ici et ne seront pas répétés lors de la description des classes et des méthodes.

3.1 conventions de nommage

- chaque mot constitutif d'un nom de classe est capitalisé (exemple : `TamponTexte`) ;
- le premier mot constitutif des méthodes et des attributs est en minuscules, mais les mots suivants sont capitalisés (exemple : `TamponTexte::ajouteSeparateurs()`) ;
- les noms des attributs se terminent par le caractère '_' (exemple : `TamponTexte::sortie_`) ;
- les noms des fichiers déclarant ou implantant les classes sont les noms des classes, auxquels on ajoute le suffixe `.cc` ou `.h` (exemple : `TamponTexte.cc`) ;
- les noms des fichiers sources n'implantant pas de classes suivent la même convention de nommage, par homogénéité (exemple : `TraducteurCC.cc`) .

3.2 gestion des erreurs

La gestion des erreurs dans la bibliothèque CLUB était, pour toutes les versions inférieures à 6.0, basée sur le mécanisme de retour d'erreur dans une instance de classe spécialisée. Les versions 6.0 et supérieures de CLUB utilisent le mécanisme des exceptions. Afin d'assurer temporairement la compatibilité de CLUB avec les versions antérieures, deux hiérarchies d'objets cohabitaient dans la version 6.0 :

- la première utilisait le mécanisme de retour d'erreur – le nom des classes de cette hiérarchie était suffixé par `_OBSOLETE`,
- la deuxième utilisait le mécanisme des exceptions.

Désormais seule la deuxième hiérarchie subsiste dans la version 7.0. Un compilateur intégrant le principe des exceptions est donc requis pour l'installation.

déclenchement d'une exception

Les constructeurs et méthodes des classes susceptibles de rencontrer un cas d'erreur peuvent lever une exception. Les exceptions utilisées dans CLUB sont des instances de la classe `ClubErreurs`.

Dans la description des classes qui est faite dans ce document, le lecteur trouvera pour chaque méthode décrite, la liste des codes d'erreur pouvant être contenus dans l'instance d'exception. Ainsi, pour une méthode susceptible de lever une exception avec un code `iterateur_invalide` ou `tampon_vide`, on pourra discriminer l'exception reçue dans le bloc `catch` comme dans l'exemple suivant :

```
...
catch (ClubErreurs ce)
{
    if (ce.code() == ClubErreurs::iterateur_invalide)
        // traitement d'erreur sur iterateur invalide
    else
        if (ce.code() == ClubErreurs::tamponvide)
            // traitement d'erreur sur tampon vide
        else
            throw;
}
```

Dans le cas d'un développement rapide de code *jetable* on peut dans un premier temps ne pas tenir compte des exceptions. La levée de l'exception provoquera alors une fin anormale du programme (`abort`) avec production d'un fichier core. On peut aussi mettre tout le code dans un unique bloc `try` suivi d'un bloc `catch` similaire à celui-ci :

```
catch(ClubErreurs ce)
{
    return 1;
}
```

Ce qui provoquera l'affichage du message d'erreur lors de la destruction de l'instance de `ClubErreurs` avant l'arrêt du programme.

Dans le cadre d'un développement ayant des objectifs de qualité élevés, il est nécessaire de faire une utilisation plus fine des blocs `try` et `catch`.

Il est intéressant de remarquer que dans tous les cas (même pour un développement rapide) la génération du message est faite à très bas niveau, là où l'on dispose de l'information la plus complète, et que le formatage de la chaîne de caractères peut être réalisé dans la langue de l'utilisateur si l'environnement comporte les fichiers de traduction des formats internes (si l'environnement n'est pas en place, un message est tout de même affiché, mais bien sûr sans traduction).

Un exemple de traitement d'erreur est donné par les lignes suivantes. En cas de levée d'exception, le `return 1` dans le bloc `catch` provoquera la destruction de l'instance d'erreur et l'affichage du message d'erreur avant la terminaison du programme.

```
int main (int argc, char **argv)
{
    try
    {
        // lecture du fichier
        TamponTexte texte (argv [1]); //peut lever une exception

        texte.elimineCommentaires (); //peut lever une exception
        texte.elimineBlancs ();        //peut lever une exception

        // affichage des champs un par un (les numéros commencent à 1)
        for (int i = 1; i <= texte.nombreChamps (); i++)
            cout << i << " : \" " << texte.champ (i) << "\"\"<<endl;

        // fin normale du programme
        return 0;
    }
    catch(ClubErreurs ce)
    {
        // on rattrape l'exception et on quitte proprement.
        return 1;
    }
}
```

retour d'erreur par une instance de classe spécialisée

Traditionnellement, les fonctions C générant une erreur retournent un entier non nul représentant le code d'erreur ou retournent 0 en cas de succès. Cette convention est respectée dans les versions de CLUB inférieures strictement à la version 6.0. Cependant il s'avère que dans une bibliothèque un entier n'est pas suffisant pour que l'appelant génère un message d'erreur pertinent ; de plus en C++ les constructeurs ne retournent aucune valeur qui puisse être testée. La convention adoptée était donc que le code de l'erreur (destiné à être testé par l'appelant) et le message d'erreur (destiné à être lu par l'opérateur) étaient également retournés dans une instance de classe spécialisée (dérivée de BaseErreurs) fournie par l'appelant.

- Si la fonction ne retourne pas de valeur testable simplement (par exemple un constructeur), l'instance d'erreur peut elle-même être testée ;
- L'appelant peut s'abstenir de fournir l'instance pour le retour des erreurs¹, auquel cas l'erreur est générée dans un objet local qui est soit retourné à l'appelant par le mécanisme des exceptions soit construit puis détruit localement ;

¹l'instance est généralement fournie sous la forme d'un pointeur optionnel en dernier argument

- Lorsqu'une instance d'erreur contenant un compte rendu d'échec est détruite, son destructeur a pour effet de bord de réaliser l'affichage du message à l'aide d'une fonction qui peut être personnalisée par l'utilisateur.

transition vers la gestion des erreurs par les exceptions

Dans la grande majorité des cas, le code existant qui utilise la gestion d'erreurs par pointeur sur ClubErreurs devra être adapté pour utiliser la nouvelle version de CLUB.

Cette adaptation implique des changements importants sur la structure du programme utilisateur de la bibliothèque. Le code fonctionnel et le code de gestion d'erreurs sont maintenant séparés dans des blocs différents. Les variables déclarées dans un bloc `try` ne sont visibles qu'à l'intérieur de celui-ci.

L'exemple suivant illustre cette transformation :

Code original s'appuyant sur une version de CLUB inférieure à 6.0 :

```
#include "club/TamponTexte.h"
#include "club/ClubErreurs.h"

int main (int argc, char **argv)
{
    ClubErreurs ce;

    // lecture du fichier
    TamponTexte texte (argv [1], &ce);
    if (ce.existe ())
        return 1; // traitement d'erreur juste apres la creation

    if (texte.elimineCommentaires (&ce) || (texte.elimineBlancs (&ce)))
        return 2;

    // affichage des champs un par un (les numéros commencent à 1)
    for (int i = 1; i <= texte.nombreChamps (); i++)
        cout << i << " : \" " << texte.champ (i) << "\"\"<<endl;

    // fin normale du programme
    return 0;
}
```

Code modifié pouvant utilisé les versions 6.0 et supérieures :

```
#include "club/TamponTexte.h"
#include "club/ClubErreurs.h"

int main (int argc, char **argv)
{
    try
    {
        // lecture du fichier
        TamponTexte texte (argv [1]);
```

```

try
{ // try imbrique car on utilise l'objet texte
  // et on différencie la cause de l'erreur
  texte.elimineCommentaires ();
  texte.elimineBlancs ();
}
catch(ClubErreurs ce)
{
  // on peut utiliser l'objet texte dans ce bloc catch
  cout << texte.nom() << endl ;
  return 2;
}

// affichage des champs un par un (les numéros commencent à 1)
for (int i = 1; i <= texte.nombreChamps (); i++)
  cout << i << " : \" " << texte.champ (i) << "\"\"<<endl;

// fin normale du programme
return 0;
}
catch(ClubErreurs ce)
{
  // le traitement d'erreur de creation se retrouve en fin de code.
  return 1;
}
}

```

Note Importante : Si un constructeur lève une exception, l'objet n'est pas créé, contrairement à ce qui se passait avec les pointeurs sur ClubErreurs.

3.3 utilisation de la STL

Comme celle du C, la norme C++ comprend la définition d'une bibliothèque standard dont la majorité des fonctions et des patrons de classes provient de la Standard Template Library (STL).

Cette bibliothèque fournit un ensemble de composants génériques structurés en C++. Ces composants ont été créés dans le but de pouvoir être utilisés à la fois sur les structures fournies par la bibliothèque et sur les structures du langage C++.

La bibliothèque CLUB implémente des classes et fonctions dont certaines s'apparentent à celles présentées par la STL. Comme il est fortement conseillé de suivre les standards de codage, la version 7.0 s'appuie désormais sur cette bibliothèque. L'implémentation et l'interface publique de nombreuses classes s'en trouvent ainsi modifiées. Il en résulte que la version 6 et la version 7 de CLUB contiennent assurément des incompatibilités qu'il faut surmonter.

Notamment, les classes dérivées de BaseErreurs (classes permettant de gérer ses propres exceptions) doivent impérativement changer la signature de la méthode protégée **formate** pour pouvoir utiliser la nouvelle version de club. La section 12.2 page 44 présente en détail la marche à suivre. Il est recommandé à cette occasion de faire le passage à **string** de la totalité des classes utilisatrices de CLUB.

Cette section présente les méthodes de substitution des classes de CLUB par celles équivalentes de la STL en s'inspirant des concepts décrits dans le document [DR14].

ChaineSimple et string

Les classes ChaineSimple de CLUB et string de la STL fournissent des fonctionnalités communes pour la manipulation des chaînes de caractères.

La version 7.0 de CLUB s'appuyant désormais sur les strings, il est fortement conseillé aux utilisateurs de ChaineSimple de privilégier désormais la classe string. Pour cela, deux stratégies peuvent être utilisées :

- substitution totale des ChaineSimples par des strings ou
- évolution progressive des ChaineSimples vers les strings.

La première alternative permet d'utiliser « proprement » l'interface publique de CLUB 7.0 mais elle nécessite de faire évoluer de façon importante les codes utilisateur. En effet, l'utilisation de string et ChaineSimple bien que similaire diffère en certains points. La table 1 fournit les équivalents STL des méthodes de la classe ChaineSimple.

TAB. 1: Méthodes publiques de ChaineSimple et string

ChaineSimple	string
ChaineSimple (int <i>taille</i> = -1)	string () void resize (size_type n, charT c = charT())
ChaineSimple (const char * <i>chaine</i>) ChaineSimple (const char * <i>chaine</i> , int <i>longueur</i>) ChaineSimple (char <i>caractere</i> , int <i>repetition</i>) ChaineSimple (const ChaineSimple& <i>c</i>)	string (const charT * <i>chaine</i>) string (const charT * <i>chaine</i> , size_type <i>longueur</i>) string (size_type <i>repetition</i> , charT <i>caractere</i>) string (const string& <i>s</i> , size_type <i>pos</i> =0, size_type <i>n</i> =npos)
ChaineSimple& operator = (const ChaineSimple& <i>c</i>) ChaineSimple& operator = (const char * <i>chaine</i>)	string& operator = (const string& <i>c</i>) string& operator = (const charT * <i>chaine</i>)
~ ChaineSimple ()	~ string ()
operator const char * ()	
void reinitialise (const ChaineSimple& <i>c</i>)	string& replace (size_type <i>pos</i> , size_type <i>n</i> , const string& <i>s</i>)
void reinitialise (const char * <i>chaine</i>) void reinitialise (const char * <i>chaine</i> , int <i>longueur</i>) void reinitialise (char <i>caractere</i> , int <i>repetition</i>)	string& replace (size_type <i>pos</i> , size_type <i>n</i> , const charT* <i>s</i>)
void formate (const char * <i>format</i> ...) void vFormate (const char * <i>format</i> , va_list <i>ap</i>)	
void modifieCaractere (int <i>i</i> , char <i>c</i>)	string& operator [(size_type <i>n</i>)] ²
à suivre ...	

²ch[i] = 'c'

TAB. 1: Méthodes publiques de ChaineSimple et string (suite)

ChaineSimple	string
void elimineBlancsInitiaux ()	size_type find_first_not_of (charT c, size_type pos = 0) const string& erase (size_type pos = 0, size_type n = npos)
void elimineBlancsFinaux ()	size_type find_last_not_of (charT c, size_type pos = 0) const string& erase (size_type pos = 0, size_type n = npos)
ChaineSimple& operator += (const ChaineSimple& c) ChaineSimple& operator += (const char *chaine) ChaineSimple& operator += (char c)	string& operator += (const string& s) string& operator += (const charT* s) string& operator += (charT c)
inline int taille () inline int longueur () inline const char * premier ()	size_type capacity () const size_type length () const const charT* c_str () const
inline const char * dernier ()	
ChaineSimple operator () (const char *debut, const char *fin) const int operator == (const char *chaine) int operator == (const ChaineSimple& chaine)	string substr (size_type pos = 0, size_type n = npos) const bool operator == (const string& s1, const charT* s2) bool operator == (const string& s1, const string& s2)
int operator != (const char *chaine) int operator != (const ChaineSimple& chaine)	bool operator != (const string& s1, const charT* s2) bool operator != (const string& s1, const string& s2)
ChaineSimple operator + (const ChaineSimple& c1, const ChaineSimple& c2) ChaineSimple operator + (const ChaineSimple& c1, const char *c2) ChaineSimple operator + (const ChaineSimple& c1, char c2) ChaineSimple operator + (const char *c1, const ChaineSimple& c2) ChaineSimple operator + (char c1, const ChaineSimple& c2)	string operator + (const string& s1, const string& s2) string operator + (const string& s1, const charT* s2) string operator + (const string& s1, charT c) string operator + (const charT* s1, const string& s2) string operator + (charT c, const string& s2)

L'évolution progressive des ChaineSimple vers les strings est rendue possible par l'ajout dans la classe ChaineSimple :

- d'un constructeur de ChaineSimple à partir d'un string,
- d'un opérateur de conversion implicite de ChaineSimple en string.

Néanmoins, cette évolution progressive pose des difficultés supplémentaires qui proviennent principalement des conversions implicites entre ces deux classes. Ainsi, le code suivant générera une erreur de compilation :

```
ChaineSimple ch ("Hello World");
AnalyseurLexical al (ch);
```

La classe `ChaineSimple` fournissant deux opérateurs de conversion implicite (conversion en `const char*` ou en `string`), le compilateur déclare que l'expression `AnalyseurLexical al (ch) ;` est ambiguë. En effet, pour réaliser cette opération, les deux voies suivantes sont possibles :

- convertir `ch` en `string` et appliquer le constructeur `AnalyseurLexical (const string&)`
- convertir `ch` en `const char*` et appliquer le constructeur `AnalyseurLexical (const char*)`

Le premier moyen d'éviter cette ambiguïté est de remplacer explicitement la `ChaineSimple` en `const char*` par la méthode publique `premier`.

```
ChaineSimple ch ("Hello World");
AnalyseurLexical al (ch.premier ());
```

Le second moyen, recommandé, est d'abandonner `ChaineSimple` au profit de `string`.

Dans d'autres cas, cette difficulté ne se posera pas :

```
ChaineSimple ch ("Hello World");
ChaineSimple premLexeme;
ChaineSimple secondLexeme;

AnalyseurLexical al (ch.premier ());
al.suivant ();
premLexeme = al.lexeme ();
al.suivant ();
secondLexeme = al.lexeme ();
```

Dans l'exemple ci-dessus, bien que la méthode `lexeme` retourne un `string`, l'affectation d'une `ChaineSimple` à partir d'un `string` pourra être réalisée en convertissant le `string` en `ChaineSimple` (grâce au constructeur de `ChaineSimple` à partir d'un `string`) puis en appliquant l'opérateur `=` entre deux `ChaineSimple`s.

Adressage et map

La classe patron `Adressage<T>` implémente une notion limitée de table d'adressage dispersé, avec une clef de type chaîne de caractères, elle est substituable par la classe de la STL `map<TypeClé, TypeDonnée, FonctionComparaison>`. Une classe réalisant la comparaison par simple appel à l'opérateur `<` est utilisée si aucune fonction de comparaison n'est fournie.

Le type de la clef le plus approprié est `string` afin d'offrir les mêmes services de manipulation de chaînes de caractères que dans `ChaineSimple`. Cette classe disposant d'un opérateur `<`, il suffit souvent de spécifier `map<string, TypeDonnees>`.

La remarque importante suivante concerne la fonctionnalité (fournie par `Adressage`) d'accès direct par index numérique qui n'est pas assurée par `map`. En revanche, il est possible de parcourir l'ensemble des valeurs contenues dans la table de hachage à l'aide d'un itérateur STL. Ce parcours par itérateur n'est cependant pas équivalent au parcours par index numérique dans `Adressage`, car l'ordre utilisé par `map` s'appuie sur la relation d'ordre sur les clefs et non sur l'ordre de remplissage de la table (comme c'est le cas dans `Adressage`).

La substitution de `Adressage` nécessite donc de revoir les mécanismes d'accès aux éléments lorsque l'accès est réalisé à l'aide d'un index. L'exemple suivant illustre cette transformation :

Code original :

```
#include "club/Adressage.cc"
template class Adressage<OptionBase *>;

Adressage<OptionBase *> table;

for (int i = 0; i < table.nbElements (); ++i)
{
    const ChaineSimple& clef    = src.clef    (i);
    const OptionBase    *valeur = src.valeur (i);

    cout << *valeur.code () << endl;
}

Code modifié :
#include <map>
#include <string>

...

map<string, OptionBase *> table;
map<string, OptionBase *>::const_iterator i;

for (i = table.begin (); i != table.end (); ++i)
{
    const string&    clef    = (*i).first;
    const OptionBase *valeur = (*i).second;

    cout << *valeur.code () << endl;
}
```

4 catalogue

4.1 classes disponibles

AnalyseurLexical reconnaît les lexèmes successifs d'un texte (noms, chaînes, séparateurs, entiers, réels, opérateurs, parenthèses);

BaseErreurs est une classe virtuelle permettant la création de classes de gestions d'erreurs spécialisées pour d'autres bibliothèques ou applications³;

CallTrace permet d'enregistrer dans un fichier tous les appels à des fonctions qui ont été instrumentées dans ce but, ce qui permet aux développeurs de reproduire des cas engendrant des erreurs chez les utilisateurs et qui permet également de faire des tests de non-régression sans pour autant disposer du code appelant réel;

ChaineSimple réalise les opérations de base sur les chaînes de caractères (allocation, extension, copie, comparaison, formatage);

ClubAllocHandler permet de gérer les erreurs de dépassement de capacité mémoire (opérateur **new**) à l'aide de la classe **ClubErreurs**;

³la classe **ClubErreurs** gérant les erreurs de la bibliothèque elle-même dérive de **BaseErreurs**

ClubErreurs gère les erreurs de la bibliothèque, et offre une interface de formatage des erreurs système ;

DataFile gère un tampon partagé sur une instance de UniqDataFile. Chaque service de DataFile appelle le service équivalent de la classe UniqDataFile ;

FichierStructure gère de façon structurée (éventuellement récursive) des blocs d'informations imbriqués contenus dans un ou plusieurs fichiers de ressources ;

FormatC permet d'analyser, de comparer et d'utiliser des formats d'écriture du C (du genre `%-12.6`) ;

FormatFortran permet d'analyser, de comparer et d'utiliser des formats d'écriture du FORTRAN (du genre `'(3(X, F14.3), /, ''mode : '', I2)'`) ;

IterateurCaractere permet de se déplacer caractère par caractère dans un TamponAscii ;

IterateurChamp permet de se déplacer champ par champ dans un TamponAscii ;

IterateurLigne permet de se déplacer ligne par ligne dans un TamponAscii ;

IterateurChampLigne permet de se déplacer par lignes et par champs dans un TamponAscii ;

MadonaFile est une classe de gestion des fichiers au format Madona. Elle dérive de UniqDataFile et s'appuie sur les services de la bibliothèque Madona ;

OptionBase et ses classes dérivées permettent de définir, une par une, les options passées à une application selon les types d'arguments qu'elles attendent et leur domaine de validité :
 – pas d'argument ;
 – chaîne de caractères (ou tableau) pouvant être limités en taille ou par une liste de valeurs ;
 – entier (ou tableau) pouvant être limité à un domaine ;
 – réel (ou tableau) pouvant être limité à un domaine ;
 les nombres d'occurrences minima et maxima et les valeurs par défaut peuvent être spécifiés option par option ;

OptionsAppli regroupe les définitions d'options pour une même application dans un analyseur d'arguments ;

StructureFile est une classe de gestion des fichiers au format FichierStructure. Elle dérive de UniqDataFile ;

TamponAscii classe de service pour manipuler du texte, utilisée par TamponTexte ;

TamponPartage offre des services de partage de zones de mémoire allouées entre instances de classes en évitant les copies ;

TamponTexte offre des services de gestion en mémoire d'un texte permettant de manipuler simultanément les structures de lignes, de champs (globalement ou à l'intérieur des lignes) ou l'absence de structure, d'ignorer les commentaires et les lignes vides à volonté, et de prendre en compte des caractères spéciaux (par exemple pour mettre un passage à la ligne (`\n`) à l'intérieur d'un champ) ;

Traducteur offre un service de traduction dans la langue de l'utilisateur de chaînes de caractères, de formats FORTRAN et de formats C avec vérification de cohérence entre le format de base et le format traduit (les messages d'erreurs de la bibliothèque utilisent ce mécanisme) ;

UniqDataFile est une classe abstraite fournissant une abstraction commune pour la gestion de fichiers sous divers formats structurés (Madona, XML, FichierStructure) ;

XMLFile est une classe de gestion des fichiers au format XML. Elle dérive de UniqDataFile et s'appuie sur les services de la bibliothèque Xerces.

4.2 utilitaires disponibles

difference cet utilitaire permet de comparer deux fichiers tout en acceptant des différences numériques dès lors qu'elles sont inférieures à un seuil paramétrable dans la ligne de commande.

4.3 routines d'interfaçage

Seul le système de traduction dispose d'interfaces avec les langages C et FORTRAN.

4.3.1 interface C

AjouterDomaine permet d'ajouter un domaine de traduction spécifié par son nom ;

TraduitVersExterne traduit une chaîne de caractères de la langue interne vers la langue de l'utilisateur ;

TraduitVersInterne traduit une chaîne de caractères de la langue de l'utilisateur vers la langue interne ;

TraduitFormatCVersExterne traduit un format du langage C vers la langue de l'utilisateur, en vérifiant la cohérence entre format initial et format traduit, afin d'éviter les violations mémoires à l'utilisation du format.

4.3.2 interface FORTRAN

domtrad permet d'ajouter un domaine de traduction spécifié par son nom ;

tradext traduit une chaîne de caractères de la langue interne vers la langue de l'utilisateur ;

tradint traduit une chaîne de caractères de la langue de l'utilisateur vers la langue interne ;

tradform traduit un format du langage FORTRAN vers la langue de l'utilisateur, en vérifiant la cohérence entre format initial et format traduit, afin d'éviter les violations mémoires à l'utilisation du format ;

tradecrCH traduit un format et l'utilise pour écrire dans une chaîne de caractères (c'est une fonction à arguments variables) ;

tradecrFD traduit un format et l'utilise pour écrire dans un fichier (c'est une fonction à arguments variables).

5 environnement

La bibliothèque CLUB a besoin, au minimum, des bibliothèques standards du langage C++. Pour pouvoir accéder à des fichiers au format Madona ou XML, la bibliothèque CLUB a besoin respectivement de la bibliothèque MADONA ou de la bibliothèque XERCES.

Le système de traduction a besoin à l'exécution de deux variables d'environnement, une pour spécifier la langue, l'autre pour spécifier une liste de répertoires pouvant contenir les dictionnaires. Ces deux variables sont configurables à l'installation de la bibliothèque (voir la section 6). Leur valeur par défaut est **CLUB_LANG** pour la langue utilisateur et **CLUB_TRADPATH** pour la liste de répertoires. L'installation standard de la bibliothèque pour le service SB/MS spécifie cependant des variables différentes, à la fois pour des raisons historiques et par soucis d'homogénéité avec l'environnement MERCATOR ; les variables sont alors **MRC_USER_LANG** pour la langue utilisateur et **MRC_USER_TRADPATH** pour la liste des répertoires.

La conception du système de traduction permet à celui-ci de tourner en l'absence de tout fichier de traduction. Dans ce cas la langue interne du logiciel est utilisée.

Le support optionnel des fichiers XML a besoin à l'exécution d'une variable d'environnement pour spécifier une liste de répertoires pouvant contenir les DTD (*Document Type definition*, fichiers décrivant les syntaxes

XML adoptées dans CLUB) et le fichier des unités par défaut. Cette variable est configurable à l'installation de la bibliothèque (voir la section 6). Sa valeur par défaut est CLUB_XMLPATH. L'installation standard de la bibliothèque pour le service SB/MS spécifie cependant une variable différente, par souci d'homogénéité avec les deux variables précédentes ; la variable est alors MRC_USER_XMLPATH.

6 installation

La bibliothèque CLUB est livrée sous forme d'une archive compressée dont le nom est de la forme club-vv.rr.tar.gz, où vv est le numéro de version et rr est le numéro de révision.

L'installation de la bibliothèque est similaire à l'installation des produits GNU. En premier lieu, il importe de décompresser⁴ l'archive et d'en extraire les fichiers, par une commande du type :

```
gunzip -c club-vv.rr.tar.gz | tar xvf -
```

Cette commande crée un répertoire club-vv.rr à l'endroit d'où elle a été lancée. Il faut ensuite se placer dans ce répertoire, et configurer les makefiles par une commande du type :

```
./configure
```

On peut modifier les choix du script de configuration par plusieurs moyens. Le cas de loin le plus courant est de vouloir installer la bibliothèque à un endroit spécifique (l'espace par défaut est /usr/local), on doit pour cela utiliser l'option --prefix comme dans l'exemple suivant :

```
./configure --prefix=/racine/espace/installation
```

Par défaut, le script de configuration recherche les bibliothèques XERCES et MADONA dans l'environnement par défaut de l'utilisateur et ne compile les fonctions optionnelles de CLUB qui en dépendent que s'il trouve ces bibliothèques. Les options --with-xerces-c[=PATH] et --with-madona[=PATH] permettent d'aider le script à trouver les bibliothèques lorsqu'elles sont installées dans des répertoires non standards. Lorsque l'on spécifie un chemin du style /racine/specifique, les fichiers d'en-tête sont recherchés sous /racine/specifique/include et /racine/specifique/include/xerces (ou madona selon le cas) et les bibliothèques sous /racine/specifique/lib. On peut également empêcher l'utilisation de ces bibliothèques en spécifiant --with-xerces-c=no et --with-madona=no (ou de façon équivalente en spécifiant --without-xerces et --without-madona).

On peut choisir les noms des variables d'environnement à utiliser pour le système de traduction en les spécifiant dans deux autres variables d'environnement avant de lancer le script de configuration. CL_VAR_LANG permet de nommer la variable décrivant la langue (avec une valeur par défaut de CLUB_LANG si la variable n'existe pas) et CL_VAR_TRADPATH permet de nommer la variable décrivant la liste des chemins de traduction (avec une valeur par défaut de CLUB_TRADPATH si la variable n'existe pas).

On peut choisir le nom de la variable d'environnement à utiliser pour le support de fichiers XML en spécifiant une autre variable d'environnement avant de lancer le script de configuration. CL_VAR_XMLPATH permet de nommer la variable décrivant la liste des chemins d'accès aux DTD et au fichier des unités par défaut (avec une valeur par défaut de CLUB_XMLPATH si la variable n'existe pas). Cette variable n'a pas besoin d'être configurée si le support XML n'est pas activé, que ce soit parce que l'utilisateur a spécifié l'une des options de configuration --with-xerces-c=no ou --without-xerces ou parce que la bibliothèque XERCES n'a pas été trouvée.

Il arrive beaucoup plus rarement que l'on désire modifier les options de compilation, il faut là encore passer par des variables d'environnement (CXXCPP, CPPFLAGS, CXX, CXXFLAGS, LDFLAGS ou LIBS) avant de lancer le script.

⁴l'utilitaire de décompression gunzip est disponible librement sur tous les sites ftp miroirs du site GNU

Si l'on désire partager les options ou les variables d'environnement entre plusieurs scripts `configure` (par exemple ceux des bibliothèques `CLUB`, `INTERFACE`, `CANTOR` et `MARMOTTES`), il est possible d'initialiser les variables⁵ dans un ou plusieurs scripts Bourne-shell. La variable `CONFIG_SITE` si elle existe donne une suite de noms de tels scripts séparés par des blancs, ceux qui existent sont chargés dans l'ordre. Si la variable n'existe pas on utilise la liste par défaut des fichiers `$(prefix)/share/config.site` puis `$(prefix)/etc/config.site` ; cette liste par défaut permet de gérer plusieurs configurations en spécifiant manuellement l'option `--prefix` sans le risque de confusion inhérent aux variables d'environnement peu visibles.

La compilation est ensuite réalisée par la commandes `make` et l'installation par la commande `make install`. La première commande compile localement tous les éléments de la bibliothèque et les programmes de tests, seule la seconde commande installe des fichiers hors de l'arborescence de compilation. Les fichiers installés sont l'archive `libclub.a`, la bibliothèque partagée `libclub.so`, les fichiers de configuration de `libtool` et les liens symboliques des bibliothèques, le répertoire de fichiers d'inclusion `club`, l'utilitaire différence, la documentation, les DTD et le fichier d'unités par défaut pour XML, et les fichiers de traduction `en/club` et `fr/club` dans les sous-répertoires de la racine spécifiée par l'option `--prefix` de `configure`, valant `/usr/local` par défaut.

Si l'utilisateur le désire, il peut faire un `make check` après le `make` pour lancer localement les tests internes de la distribution.

Il est possible de désinstaller complètement la bibliothèque par la commande `make uninstall`.

Pour régénérer l'arborescence telle qu'issue de la distribution (en particulier pour éliminer les fichiers de cache de `configure`), il faut faire un `make distclean`.

La bibliothèque `CLUB` contient deux classes de gestion des formats de fichier XML et Madona nommées respectivement `XMLFile` et `MadonaFile`. Ces classes s'appuient respectivement sur les bibliothèques `XERCES`⁶ et `MADONA`⁷ qui doivent donc impérativement être installées pour pouvoir utiliser leurs services. L'absence de ces bibliothèques n'empêche pas la génération et l'installation de `CLUB`. En revanche, si `XERCES` (respectivement `MADONA`) n'est pas installée, `XMLFile` (respectivement `MadonaFile`) n'est ni compilée ni installée.

Une fois l'installation de `CLUB` réalisée, le répertoire `club-vv.rr` ne sert plus (sauf si l'on a compilé avec une option de déboguage) et peut être supprimé.

La démarche de compilation normale est de désarchiver, de configurer, de compiler, d'installer, puis de supprimer le répertoire des sources, pour ne conserver que l'archive compressée. Le répertoire des sources n'est pas un espace de stockage permanent, les directives du `Makefile` ne supportent en particulier pas les compilations simultanées sur un espace unique, et le `Makefile` lui-même dépend de la configuration (il n'est d'ailleurs pas livré pour cette raison). Si un utilisateur désire installer la bibliothèque sur plusieurs machines ayant chacune un espace privé pour les bibliothèques (typiquement `/usr/local`) mais se partageant le répertoire d'archivage par un montage de disque distant, il ne faut pas décompresser l'archive dans l'espace commun. On préférera dans ce cas une série de commandes du type :

⁵l'option `--prefix` s'initialise à l'aide d'une variable shell `prefix`

⁶une distribution de `XERCES` est disponible sur le site [DR16]

⁷l'installation de `MADONA` est expliquée dans le document [DR4]

```
cd /tmp
gunzip -c /chemin/vers/l/espace/partage/club-vv.rr.tar.gz | tar xvf -
cd club-vv.rr
env CL_VAR_LANG=MRC_USER_LANG \
    CL_VAR_TRADPATH=MRC_USER_TRADPATH \
    CL_VAR_XMLPATH=MRC_USER_XMLPATH \
    ./configure --prefix=/chemin/vers/l/espace/prive \
    --with-xerces-c=/opt --without-madona
make
make install
cd ..
rm -fr club-vv.rr
```

7 messages d'avertissements et d'erreurs

La bibliothèque CLUB peut générer des messages d'erreurs en fonction de la langue de l'utilisateur. La liste suivante est triée par ordre alphabétique du format d'écriture dans le fichier de traduction français d'origine⁸. Le format de la traduction anglaise d'origine est indiqué entre parenthèses. Si un utilisateur modifie les fichiers de traduction d'origine (ce qui est tout à fait raisonnable), la liste suivante devra être interprétée avec une certaine tolérance.

`">' manquant après \"%s\" (fichier \"%s\")"`
`("missing '>' after \"%s\" (file \"%s\")"`) ce message indique une erreur de syntaxe à la lecture d'une directive d'inclusion `<nom_de_fichier>` dans un FichierStructure.

`"{' manquant (bloc \"%s\" du fichier \"%s\")"`
`("missing '{' in bloc \"%s\" (file \"%s\")"`) ce message indique une erreur de syntaxe après un nom de bloc dans un FichierStructure.

`"accolades non équilibrées dans le fichier \"%s\""`
`("non matching braces in file \"%s\""`) ce message indique une erreur de syntaxe dans un FichierStructure.

`"analyseur d'options de %s déjà initialisé"`
`("options analyser %s already initialized"`) ce message est généré lors de la mise en place d'un analyseur d'option si la méthode `OptionsAppli::initialiser` est appelée plus d'une fois ; il s'agit probablement d'une erreur de codage.

`"analyseur d'options de %s non initialisé"`
`("options analyser %s not initialized"`) ce message est généré lors de la mise en place d'un analyseur d'option si la méthode `OptionsAppli::initialiser` n'est pas appelée avant de commencer l'extraction des arguments ; il s'agit probablement d'une erreur de codage.

`"argument de l'option -%s trop long : \"%s\" (%d caractères max)"`
`("too long argument for -%s option : \"%s\" (%d charactersmax)"`) ce message indique à l'utilisateur qu'il viole une limite de longueur autorisée pour l'argument d'une option de type chaîne de caractères (ou tableau de chaînes) lors de l'exécution.

⁸la langue interne est le français sans les accents et c'est le fichier de traduction français qui introduit les accents, le fichier de traduction anglais reste ainsi lisible par un utilisateur anglophone qui n'aurait pas configuré son éditeur de texte de sorte qu'il affiche correctement les caractères utilisant le codage iso-8859-1 (IsoLatin 1)

"argument hors domaine pour l'option -%s : \"%s\", arguments autorisés :"
 ("unauthorized argument for -%s option : \"%s\", authorized arguments : ") ce message indique à l'utilisateur que l'argument d'une option de type chaîne de caractères (ou tableau de chaînes) qu'il a donné n'appartient pas à la liste autorisée (la liste est affichée juste après, par un autre format).

"argument manquant pour l'option -%s"
 ("missing argument for -%s option") ce message indique à l'utilisateur qu'il a passé une option attendant un argument et qu'il n'a pas donné cet argument (ce message est affiché lorsque l'analyse arrive à la fin de la liste des arguments).

"arguments de l'option -%s accessibles uniquement par tableau de taille %d"
 ("access to arguments of -%s option allowed only through array of size %d") l'apparition de ce message indique qu'une méthode OptionsAppli::lireValeur a été appelée avec un tableau dont la taille ne correspondait pas à ce qui avait été paramétré lors de la création de l'option correspondante ; il s'agit probablement d'une erreur de codage.

"arguments non reconnus par l'analyseur d'options de %s :"
 ("options analyser %s unable to recognize : ") ce message est généré par l'appel de la méthode OptionsAppli::garantirComplet s'il reste des arguments non reconnus dans la ligne de commande.

"attribut %s manquant dans l'élément %s"
 ("missing %s attribute in element %s") ce message indique que l'attribut spécifié n'est pas renseigné dans un élément d'un fichier XML ; il s'agit probablement d'une erreur dans le fichier (l'erreur n'étant générée que si l'attribut est indispensable).

"bloc \"%s\" introuvable dans le fichier \"%s\""
 ("no bloc \"%s\" in file \"%s\"") ce message est généré en cas d'échec d'une tentative d'extraction d'un bloc de haut niveau d'un FichierStructure, il s'agit le plus souvent d'une erreur de nommage de ce bloc (soit dans l'appel soit dans le fichier).

"bloc \"%s\" non terminal dans le fichier \"%s\""
 ("non terminal bloc \"%s\" in file \"%s\"") ce message est généré en cas de tentative d'extraction directe de données d'un bloc qui contient encore un niveau de structuration dans un FichierStructure ; il s'agit probablement d'une erreur d'analyse de la structure du fichier si celle-ci est variable ou d'une erreur de syntaxe dans le fichier si la structure est fixe.

"caractère %d hors limites [%d ; %d], tampon \"%s\""
 ("character number %d out of range [%d ; %d], buffer \"%s\"") ce message indique une erreur dans le numéro d'index de caractère lors d'un appel à l'une des méthodes d'accès par caractère de TamponAscii ; il s'agit probablement d'une erreur de codage, des méthodes étant disponibles pour vérifier le nombre de caractères dans un tampon avant de tenter une opération.

"caractère \" ou caractère \' manquant dans le tampon \"%s\""
 ("missing character \" or \' in buffer \"%s\"") un tel message est généré lorsque le guillemet (ou l'apostrophe) fermant une chaîne du langage C (respectivement FORTRAN) ne peut être trouvé(e) lors de l'analyse d'un texte à travers un TamponTexte avec interprétation des caractères spéciaux ; il s'agit probablement d'une erreur de syntaxe dans le fichier.

"champ %d hors limites [%d ; %d], ligne %d du tampon \"%s\""
 ("field %d out of range [%d ; %d], line %d, buffer \"%s\"") ce message indique une erreur dans le numéro de champ lors d'un appel à l'une des méthodes d'accès par champs et lignes de TamponTexte ; il s'agit probablement d'une erreur de codage, des méthodes étant disponibles pour vérifier le nombre de champs dans une ligne avant de tenter l'extraction.

"champ %d hors limites [%d ; %d], tampon \"%s\""
 ("field %d out of range [%d ; %d], buffer \"%s\"") ce message indique une erreur dans le numéro de champ lors d'un appel à l'une des méthodes d'accès par champs simple de TamponTexte ; il s'agit probablement d'une erreur de codage, des méthodes étant disponibles pour vérifier le nombre de champs avant de tenter l'extraction.

"code d'option \"%s\" non reconnu"
 ("unknown option code \"%s\"") ce message est généré lors de l'analyse des arguments de la ligne de commande par la classe OptionsAppli, si l'on cherche à récupérer une occurrence qui n'avait pas été décrite à l'analyseur au préalable ; il s'agit probablement d'une erreur de codage.

"création de l'index %s interdite"
 ("creation of index %s is forbidden") ce message est émis lorsque l'utilisateur tente de créer une donnée en indiquant un numéro d'index (par exemple : `createStringData ("[12]", "valeur")`).

"désignation \"%s\" non valide"
 ("invalid data name \"%s\"") ce message indique que la désignation d'une donnée n'est pas valide.

"donnée courante \"%s\" est un tableau"
 ("current data \"%s\" is a table") ce message indique que la donnée courante est un tableau ce qui rend la fonction appelée non réalisable (par exemple : accès à une donnée par son nom alors que la donnée courante est un tableau).

"donnée courante \"%s\" n'est pas un tableau"
 ("current data \"%s\" is not a table") ce message indique que la donnée courante n'est pas un tableau ce qui rend la fonction appelée non réalisable (par exemple : accès à une donnée par son index alors que la donnée courante n'est pas un tableau).

"donnée \"%s\" déjà définie"
 ("data \"%s\" is already defined") ce message est émis lorsqu'on essaie de créer une donnée qui existe déjà.

"le format du fichier est invalide"
 ("type of file is not valid") ce message est émis lorsqu'on essaie de créer une instance de DataFile en fournissant un type de fichier inconnu, ou en désignant un fichier dont le format ne respecte pas celui des fichiers structurés, des fichiers XML (et la DTD) et Madona.

"donnée \"%s\" non définie"
 ("no data named \"%s\"") ce message est émis lorsqu'on essaie d'accéder à une donnée non définie.

"entité générale %s inconnue dans la chaîne \"%s\""
 ("unknown general entity %s in string \"%s\"") ce message indique qu'une entité générale du type `µ` n'a pas été reconnue (voir la section 12.27.1 pour la liste des entités reconnues). Il s'agit généralement d'une erreur de programmation dans une constante chaîne de caractères représentant une unité (les erreurs dans les fichiers XML eux-mêmes sont générées par la bibliothèque Xerces elle-même).

"erreur à l'ouverture du fichier %s"
 ("error while opening file %s") ce message est généré lorsque la bibliothèque tente d'ouvrir un fichier dans le mode désiré (écriture pour le fichier des traces d'exécution). Il faut généralement vérifier le nom et le chemin d'accès au fichier, les droits de l'utilisateur sur le fichier (ou le répertoire qui le contient) et la place disque disponible.

"erreur de syntaxe dans l'unité %s"
 ("syntax error in unit %s") ce message indique qu'une erreur de syntaxe est présente dans la définition

d'une unité composée. Il s'agit soit d'une erreur de programmation dans une constante chaîne de caractères représentant une unité, soit d'une erreur dans un fichier d'unités.

"erreur de syntaxe d'appel à une entité dans la chaîne \"%s\""

("entity reference syntax error in string \"%s\"") ce message est généré lorsqu'un appel d'entité paramètre du type `µ`, ou un appel de caractère du type `µ` est mal fait (typiquement entité vide, caractères interdits ou oubli du point-virgule de fin). Il s'agit généralement d'une erreur de programmation dans une constante chaîne de caractères représentant une unité (les erreurs dans les fichiers XML eux-mêmes sont générées par la bibliothèque Xerces elle-même).

"erreur d'analyse du format fortran \"%s\""

("analysis error for fortran format \"%s\"") ce message est généré par le méthode analyse de la classe FormatFortran si la chaîne fournie en argument n'est pas reconnue. Il faut noter que le système de traduction qui utilise cette méthode récupère cette erreur pour sécuriser l'utilisation ultérieure du format ; elle ne remonte pas jusqu'à l'utilisateur dans ce cas.

"erreur du domaine %s numéro %d non reconnue"

("unknown domain %s error (number %d)") ce message ne devrait jamais apparaître, il indique un oubli dans le code de sélection des messages d'une classe dérivée de BaseErreurs. Toute apparition d'un message de ce type doit être signalée au responsable de la maintenance de la bibliothèque concernée (que l'on reconnait par le nom du domaine).

"erreur fichier \"%s\": \"%s\""

("error in file \"%s\": \"%s\"") ce message indique qu'une erreur est survenue lors de la lecture ou de l'écriture d'un fichier de données.

"erreur interne de CLUB, veuillez contacter la maintenance (ligne %d, fichier %s)"

("CLUB internal error, please contact the support (line %d, file %s)") Ce message ne devrait normalement pas apparaître ... Il indique qu'un cas théoriquement impossible a été rencontré, ce qui témoigne d'une erreur de la bibliothèque. Dans une telle éventualité, il faut prévenir la maintenance.

"erreur Madona : \"%s\""

("Madona error : \"%s\"") ce message est émis lorsqu'une erreur s'est produite lors de l'appel à une méthode de la bibliothèque Madona. Le message d'erreur est fourni dans le texte.

"erreur XML : \"%s\""

("XML error : \"%s\"") ce message est émis lorsqu'une erreur s'est produite lors de l'appel à une méthode de la bibliothèque Xerces. Le message d'erreur est fourni dans le texte.

"fonction \"%s\" non accessible car la bibliothèque Madona n'est pas installée"

("as Madona is not installed, function named \"%s\" is not accessible") ce message indique qu'il n'est pas possible de réaliser la fonction appelée car elle nécessite l'appel à des méthodes de Madona qui n'est pas installée.

"fonction \"%s\" non accessible car la bibliothèque Xerces n'est pas installée"

("as Xerces is not installed, function named \"%s\" is not accessible") ce message indique qu'il n'est pas possible de réaliser la fonction appelée car elle nécessite l'appel à des méthodes de Xerces qui n'est pas installée.

"fonction \"%s\" non implémentée"

("function \"%s\" not implemented") ce message est émis lorsqu'une fonction de l'interface est appelée mais qu'aucune implémentation de cette fonction n'est réalisée.

"impossible d'effectuer une conversion entre les unités incompatibles %s et %s"

("forbidden conversion between incompatible units %s and %s") ce message indique qu'une tentative

de conversion entre deux unités incompatibles a été tentée. Il s'agit soit d'une erreur de programmation (passage de la mauvaise unité pour une donnée), soit d'une erreur dans un fichier de données (unité erronée), soit d'une erreur dans le fichier de définition des unités (dimensions physiques de l'unité).

"impossible de convertir \"%s\" en entier"

("converting \"%s\" into an integer is impossible") ce message peut être émis dans deux cas : premièrement, lorsqu'on veut accéder à une donnée en indiquant une valeur d'index erronée (exemple : «[12a]»); deuxièmement lorsqu'une donnée entière tente d'être lue alors que sa valeur n'est pas un entier.

"impossible de convertir \"%s\" en réel"

("converting \"%s\" into a real is impossible") ce message est émis lorsqu'une donnée réelle tente d'être lue alors que sa valeur n'est pas un réel.

"impossible de remonter d'un niveau depuis la racine du document"

("moving up from the root of document is impossible") ce message précise qu'il est impossible de remonter dans la structure de données lorsque la donnée courante est l'élément racine.

"incohérence entre un itérateur et le tampon \"%s\""

("iterator out of date towards buffer \"%s\"") ce message est généré lorsque des opérations de modification d'un tampon ont été effectuées après la déclaration d'un itérateur, et que cet itérateur n'a pas été ensuite synchronisé sur le nouvel état du tampon.

"index %d de la table \"%s\" non défini"

("no index number %d in table \"%s\"") ce message est émis lorsqu'on essaie d'accéder à une donnée par un index indéfini dans une table.

"l'option sans nom doit avoir un argument et un seul"

("the unnamed option should have one and only one argument") ce message est généré lors de la mise en place d'un analyseur d'options, il indique que l'on a tenté de créer une option sans nom sans respecter son nombre d'argument imposé (à 1); il s'agit probablement d'une erreur de codage.

"le tag %s manque"

("missing tag %s") ce message indique l'absence d'une balise dans un document XML.

"ligne %d hors limites [%d ; %d], tampon \"%s\""

("line %d out of range [%d ; %d], buffer \"%s\"") ce message indique une erreur dans le numéro de ligne lors d'un appel à l'une des méthodes d'accès par lignes de TamponTexte; il s'agit probablement d'une erreur de codage, des méthodes étant disponibles pour vérifier le nombre de lignes avant de tenter l'extraction.

"noeud %s de type inconnu"

("unknown type for node %s") ce message indique que la lecture du noeud n'est pas possible car son type n'est pas reconnu.

"nom d'option illégal (ne commence pas par une lettre) : -%s"

("illegal option name (should begin with a letter) : -%s") ce message est généré lors de la mise en place d'un analyseur d'options, il indique que l'on a tenté de créer une option sans respecter les règles de nommage; il s'agit probablement d'une erreur de codage.

"nom de fichier à inclure manquant (fichier \"%s\")"

("missing name of file to include (file \"%s\")") ce message indique une erreur de syntaxe dans un fichier lu à travers la classe FichierStructure, il s'agit d'une direction d'inclusion <nom> dans laquelle aucun nom de fichier n'a été spécifié.

"nombre d'arguments de l'option -%s incompatibles : %d trouvés, %d attendus"

("wrong arguments number for -%s option : %d found, %d expected") ce message est généré lors de

l'analyse des arguments de la ligne de commande, il indique que la ligne ne correspond pas à ce que l'analyseur attendait⁹.

"nombre d'arguments de l'option -%s négatif : %d"

("negative arguments number for -%s option : %d") ce message est généré lors de la mise en place d'un analyseur d'options, il indique que l'on a tenté de créer une option attendant un nombre négatif d'arguments ; il s'agit probablement d'une erreur de codage.

"nombre d'occurrences de l'option -%s hors limites [%d ; %d] : %d"

("instance number of -%s option out of range [%d ; %d] : %d") ce message est généré lors de l'analyse des arguments de la ligne de commande, il indique à l'utilisateur qu'il n'a pas respecté les limites que l'analyseur imposait.

"occurrence %d de l'option -%s inexistante (%d occurrences passées)"

("instance %d of -%s option not available (%d instances available)") ce message est généré lors de l'analyse des arguments de la ligne de commande, il indique que l'on cherche à récupérer une occurrence qui n'a pas été fournie par l'utilisateur ; il s'agit à la fois d'une erreur de l'utilisateur (qui n'a pas fourni ce qu'il fallait) et d'une erreur de codage, puisque la validité de la ligne de commande n'a pas été vérifiée avant son l'exploitation.

"option -%s absente"

("missing -%s option") ce message est généré lors de l'analyse des arguments de la ligne de commande, il indique à l'utilisateur qu'il n'a pas spécifié une option obligatoire (ce message est similaire à celui portant sur le nombre d'occurrences hors limites, il est simplement adapté au cas où l'on attend une occurrence et une seule).

"option -%s définie deux fois"

("-%s option defined twice") ce message est généré lorsque l'on tente de stocker une option dans un analyseur (class OptionsAppli) alors qu'une option de même nom y est déjà présente ; il s'agit probablement d'une erreur de codage.

"option -%s hors limites [%d ; %d] : %d"

("-%s option out of range [%d ; %d] : %d") ce message est généré lors de la vérification de la validité des arguments de la ligne de commande à travers la classe OptionAppli, il indique que l'utilisateur n'a pas fourni ce qu'il fallait (ou qu'à la construction la valeur par défaut était déjà hors limites, ce qui résulte probablement d'une erreur de codage).

"option -%s hors limites [%d ; +infini[: %d"

("-%s option out of range [%d ; +infinity[: %d") ce message est généré lors de la vérification de la validité des arguments de la ligne de commande à travers la classe OptionAppli, il indique que l'utilisateur n'a pas fourni ce qu'il fallait (ou qu'à la construction la valeur par défaut était déjà hors limites, ce qui résulte probablement d'une erreur de codage).

"option -%s hors limites [%f ; %f] : %f"

("-%s option out of range [%f ; %f] : %f") ce message est généré lors de la vérification de la validité des arguments de la ligne de commande à travers la classe OptionAppli, il indique que l'utilisateur n'a pas fourni ce qu'il fallait (ou qu'à la construction la valeur par défaut était déjà hors limites, ce qui résulte probablement d'une erreur de codage).

"option -%s hors limites [%f ; +infini[: %f"

("-%s option out of range [%f ; +infinity[: %f") ce message est généré lors de la vérification de la

⁹il est probable qu'en fait cette erreur ne puisse jamais être générée, les nombres d'arguments étant vérifiés avant d'appeler la méthode concernée ...

validité des arguments de la ligne de commande à travers la classe OptionAppli, il indique que l'utilisateur n'a pas fourni ce qu'il fallait (ou qu'à la construction la valeur par défaut était déjà hors limites, ce qui résulte probablement d'une erreur de codage).

"option -%s hors limites]-infini ; %d] : %d"

("-%s option out of range]-infinity ; %d] : %d") ce message est généré lors de la vérification de la validité des arguments de la ligne de commande à travers la classe OptionAppli, il indique que l'utilisateur n'a pas fourni ce qu'il fallait (ou qu'à la construction la valeur par défaut était déjà hors limites, ce qui résulte probablement d'une erreur de codage).

"option -%s hors limites]-infini ; %f] : %f"

("-%s option out of range]-infinity ; %f] : %f") ce message est généré lors de la vérification de la validité des arguments de la ligne de commande à travers la classe OptionAppli, il indique que l'utilisateur n'a pas fourni ce qu'il fallait (ou qu'à la construction la valeur par défaut était déjà hors limites, ce qui résulte probablement d'une erreur de codage).

"paramètre \"%s\" non défini"

("no parameter named \"%s\"") ce message est émis lorsqu'on essaie d'accéder à un paramètre non définie.

"pas d'argument pour l'option -%s"

("no argument for -%s option") ce message est généré lors de la mise en place d'un analyseur d'options, il indique qu'une option typée (chaîne de caractères, entier, réel) est créée comme n'attendant aucun argument ; il s'agit probablement d'une erreur de codage.

"pas de champ %d dans le bloc \"%s\" du fichier \"%s\""

("no %d field within bloc \"%s\" (file \"%s\")") ce message est généré lors de la recherche d'un champ dans un bloc terminal d'un fichier lu par FichierStructure, il s'agit le plus souvent d'une erreur de syntaxe dans le fichier de données.

"passage par la ligne %d du fichier \"%s\""

("passing through line %d (file \"%s\")") ce message est généré lors de l'exécution d'une ligne de code où une trace a été laissée, il s'agit normalement d'une aide au développement qui devrait disparaître dans la version définitive de tout logiciel.

"problème d'allocation mémoire"

("memory allocation problem") ce message peut être généré au cours de toute allocation mémoire par les opérateurs `new` et `new []` au cours de l'exécution d'une portion de code protégée par une instance de ClubAllocHandler (c'est à dire pendant la durée de vie de cette instance, même si elle n'est pas dans la fonction courante). Elle indique un dépassement des capacités du système hôte.

"problème d'obtention de l'état du fichier \"%s\""

("unable to get \"%s\" file state") ce message est généré lors d'une tentative de lecture de l'état d'un fichier, il est normalement suivi d'une copie du message système expliquant la raison de l'échec (droits d'accès, nombre de fichiers ouverts, ...).

"problème d'ouverture du fichier \"%s\""

("unable to open file \"%s\"") ce message est généré lors d'une tentative d'ouverture de fichier, il est normalement suivi d'une copie du message système expliquant la raison de l'échec (droits d'accès, nombre de fichiers ouverts, ...).

"redéfinition de l'unité %s"

("unit %s redefined") ce message indique qu'un même symbole a été utilisé deux fois dans un fichier XML d'unités, il s'agit d'une erreur dans le fichier, les symboles devant être uniques.

"référence xpointer \"%s\" incorrecte"

("invalid xpointer reference \"%s\"") ce message indique qu'une référence XML n'est pas à la norme xpointer.

"sous-bloc \"%s\" introuvable dans le bloc \"%s\" du fichier \"%s\""

("no bloc \"%s\" inside bloc \"%s\" (file \"%s\")") ce message est généré lors de la recherche d'un sous-bloc dans un bloc non terminal d'un fichier lu par FichierStructure, il s'agit le plus souvent d'une erreur de syntaxe dans le fichier de données.

"structure de données non valide : %s"

("invalid data structure : %s") ce message indique soit que la structure de donnée ne présente pas les unités de la manière définie dans la DTD, soit que la donnée mère de la donnée courante n'est pas un élément.

"structure ou table \"%s\" non définie"

("no structure or table named \"%s\"") ce message est émis lorsqu'on essaie de descendre dans une table ou une structure non définie.

"table \"%s\" de type \"%s\" ne peut contenir des éléments de type \"%s\""

("table \"%s\" (type \"%s\") can not contain element of type \"%s\"") ce message indique qu'une fonction d'accès, de création ou de mise à jour d'un élément a été réalisée alors que la table n'est pas du bon type (par exemple : appel à la méthode getStringData (int index) alors que la donnée courante est un tableau de réels).

"tampon \"%s\" vide"

("\"%s\" buffer empty") ce message peut être généré par quasiment toutes les méthodes d'accès de la classe TamponTexte si le tampon courant est vide.

"trace des exécutions désactivée"

("call trace not activated") Ce message est activé lorsque l'on tente d'utiliser le mécanisme de trace des exécutions sans l'avoir activé. Il indique une erreur de codage. si l'utilisateur n'a pas tenté d'utiliser les routines internes de ce mécanisme sans précaution, alors il s'agit d'une erreur interne à la bibliothèque et il faut prévenir la maintenance.

"types incompatibles à la lecture de l'option -%s : %s/%s"

("non matching types while reading -%s option : %s/%s") ce message est généré par les diverses méthodes lireValeur des classes dérivées de OptionBase, il indique par exemple que l'on tente d'extraire un entier d'une option de type chaîne de caractère; il s'agit probablement d'une erreur de codage.

"unité \"%s\" non reconnue"

("unknown unit \"%s\"") ce message indique que l'unité spécifiée (pour une conversion par exemple) est inconnue.

"usage : "

("usage : ") ce format est le début d'un message d'usage généré soit par l'appel direct à la méthode usage de la classe OptionsAppli, soit par l'appel à la méthode garantirComplet.

"valeur par défaut manquante pour l'option -%s"

("missing default value for -%s option") ce message est généré lors de la mise en place d'un analyseur d'options lorsque l'on tente de construire une option qui doit avoir des valeurs par défaut et que l'on omet ces valeurs; il s'agit probablement d'une erreur de codage.

"variable d'environnement \"%s\" manquante"

("missing environment variable \"%s\"") ce message indique qu'une variable d'environnement nécessaire au fonctionnement de la bibliothèque n'est pas initialisée; l'utilisateur doit mettre son environnement à jour pour pouvoir utiliser tout programme basé sur CLUB

8 tests

La bibliothèque CLUB dispose d'un certain nombre de tests de non régression automatiques que l'on peut exécuter par la commande **make check** disponible dans la distribution standard. Cette commande lance les tests existants et compare automatiquement (à l'aide de l'utilitaire **difference**) le résultat avec le résultat de référence archivé.

Les classes disposant de tests spécifiques sont :

- AnalyseurLexical
- FichierStructure
- FormatC
- FormatFortran
- ItérateurLigne
- ItérateurChamp
- ItérateurChampLigne
- MadonaFile
- OptionsAppli
- StructureFile
- TamponCaractere
- TamponPartage
- TamponTexte
- Traducteur
- XMLData
- XMLUnits

Les diverses classes dérivées de OptionBase sont testées par le test de OptionsAppli.

La classe TamponAscii est testée par l'intermédiaire des tests sur les itérateurs et TamponTexte.

Les classes ChaineSimple, BaseErreurs, ClubErreurs, ClubAllocHandler n'ont pas de test. Elles ont cependant été utilisées de façon très intensive (dans cette bibliothèque et dans d'autres) et peuvent être considérées comme étant les classes les plus validées de la bibliothèque.

Les tests automatiques de CLUB ont été analysés à l'aide de l'outil de vérification de mémoire **purify** à l'occasion de la distribution de la version 8.3 de la bibliothèque. Les erreurs détectées correspondant à la bibliothèque elle-même ont été corrigées. Il subsiste des fuites qui semblent liées à l'implémentation de la STL dans le compilateur SUN et des *fuites potentielles*¹⁰ qui ne semblent pas réelles d'après une analyse manuelle du code correspondant.

9 maintenance

La maintenance de la bibliothèque CLUB s'effectue selon quelques principes qui concernent la portabilité, les outils utilisés, les procédures de maintenance, les fichiers descriptifs et l'archivage sous **cvs**.

9.1 portabilité

La bibliothèque a dépassé le stade de l'outil spécifique d'un département et est utilisée sur plusieurs sites dans des environnements différents. La portabilité est donc un point fondamental à garder en permanence à

¹⁰selon les comptes-rendus de l'outil

l'esprit.

Le modèle suivi a donc naturellement été celui de la lignée de produits GNU. Le document "GNU coding standards" est très utile pour comprendre cette organisation (cf [DR6]). Dans cet esprit, l'environnement de maintenance nécessite beaucoup d'outils mais les utilisateurs finaux n'ont guère besoin que de **gunzip** pour décompresser la distribution, et d'un compilateur C++.

9.2 environnement de maintenance

Les produits suivants sont indispensables :

- **libtool** [DR7] (version 1.5.14 au moins)
- **autoconf** [DR8] (version 2.59 au moins)
- **automake** [DR9] (version 1.9.5 au moins)
- **cvs** [DR10]
- **g++** [DR11] (version 3.3 au moins)
- **gzip**
- GNU **m4**
- GNU **make**
- **perl**
- T_EX/L_AT_EX/**dvips** (de plus **xdvi** est recommandé)
- le paquetage L_AT_EX «**notechope**»
- les paquetages L_AT_EX «**babel**» et «**longtable**»

9.3 installation de l'environnement de maintenance

Le développeur récupère tout d'abord le module club par une commande **cvs checkout club**, il lui faut ensuite générer certains fichiers. Il suffit de passer cinq commandes pour obtenir un environnement complet :

```
aclocal
autoheader
autoconf
automake
```

aclocal génère le fichier **aclocal.m4**

autoheader génère le fichier **src/ClubConfig.h.in**

autoconf génère le script **configure**

automake génère tous les fichiers **Makefile.in**

9.4 compilation

Une fois les fichiers indiqués au paragraphe précédent créés, on se retrouve dans une situation similaire à celle d'un utilisateur qui reçoit la distribution (on a même quelques fichiers en plus, par exemple ceux liés à la gestion **cvs**). Il suffit alors de générer les fichiers **Makefile** par la commande :

```
./configure
```

ou bien

```
./configure --prefix=$HOME
```

si l'on préfère travailler entièrement dans l'environnement de maintenance.

Il faut noter que les **Makefile** générés savent non seulement compiler la bibliothèque, mais qu'ils savent également relancer les commandes initialisant le mécanisme, ceci signifie que d'éventuelles modifications des fichiers **configure.ac** ou **Makefile.am** utilisés par les commandes précédentes seront correctement répercutées partout.

Par défaut, la bibliothèque **CLUB** est générée sous forme statique. Pour générer **CLUB** en version dynamique, il faut utiliser l'option **--enable-shared** du script **configure**. Il faut cependant prendre garde que pour pouvoir générer une bibliothèque sous forme dynamique, toutes les bibliothèques dont elle dépend doivent être disponible sous cette forme. Si on active les supports **MADONA** et **XERCES**, il faut vérifier qu'elles soient disponibles en dynamique (c'est généralement le cas pour **XERCES**, mais c'est généralement faux pour **MADONA**) avant d'activer l'option **--enable-shared**. On peut également désactiver la génération de la bibliothèque statique à l'aide de l'option **--disable-static**.

9.5 procédures de maintenance

L'ensemble des sources (que ce soient les sources **C++** ou les fichiers de configuration des outils de génération de scripts) sont gérés sous **cvs** (cf. [DR10]). Les fichiers pouvant être générés automatiquement *ne sont pas gérés sous cvs*.

Il ne faut bien sûr pas éditer les fichiers générés, mais éditer les fichiers sources correspondant. Ces fichiers sources sont de plus considérablement plus simples à comprendre. La difficulté est de savoir quels fichiers sont générés et à partir de quels fichiers sources. On ne peut pas toujours se fier au nom, ainsi **src/ClubConfig.h.in** et tous les fichiers **Makefile.in** sont générés, leur suffixe **.in** signifie simplement qu'une fois générés (par **autoheader** et par **automake** respectivement) ils servent de sources à **autoconf** (qui génère alors **src/ClubConfig.h** et **Makefile**). Les fichiers éditables sont donc : **configure.ac**, et tous les **Makefile.am**.

D'autre part la bibliothèque est aussi maintenue à l'aide du mécanisme de **ChangeLog** qui présente un avantage majeur : les modifications sont présentées dans l'ordre historique des actions de maintenance, ce qui d'une part est en corrélation avec le processus de maintenance et d'autre part peut aider à déterminer par exemple à quels moments certains bugs ont pu être introduits.

Pour tout changement de fichier, il est recommandé de mettre une entrée dans le fichier **ChangeLog** (il y a un fichier de ce type pour chaque sous-répertoire). Sous **emacs** il suffit d'utiliser la commande **M-x add-change-log-entry** en étant à l'endroit où l'on a fait la modification, **emacs** remplissant seul la date, l'auteur, le nom de fichier, et le contexte (nom de fonction, de classe, ...). Pour savoir comment remplir ce fichier, il est recommandé de lire le document décrivant le standard [DR6]. Ces modifications de niveau source ne doivent pas être mises dans le fichier **NEWS**, qui contient les nouveautés de niveau utilisateur, pas développeur.

Pour savoir ce qui peut poser des problèmes de portabilité et comment résoudre ces problèmes, il est fortement recommandé de lire le manuel **autoconf** [DR8] (à cette occasion, on pourra également se pencher sur le manuel **automake**). On peut également utiliser **autoscan** (qui fait partie de la distribution **autoconf**) pour détecter automatiquement les problèmes communs et proposer des macros les prenant en compte pour **configure.ac**.

Pour faire les tests, il faut utiliser la cible **check** du **Makefile**.

Les fichiers décrivant les spécificités de la bibliothèque **CLUB**. Ces fichiers concernent soit l'utilisation soit la maintenance de la bibliothèque.

9.5.1 fichiers descriptifs destinés à l'utilisateur

`README` donne une définition globale de la bibliothèque et indique les particularités de l'installation pour certains environnements.

`NEWS` décrit l'évolution de la bibliothèque, il indique les changements visibles par l'utilisateur.

9.5.2 fichiers descriptifs destinés au mainteneur

`README.dev` définit les principes de maintenance, décrit l'environnement nécessaire pour maintenir la bibliothèque, rappelle les commandes à exécuter à l'aide des produits GNU pour créer un espace de développement et compiler la bibliothèque.

9.6 archivage

La politique d'archivage dans le serveur `cvs` est qu'il faut archiver à chaque nouvelle fonctionnalité ajoutée, ou à chaque bug corrigé. Ceci permet de cerner plus facilement les portions de code touchées (pour les `cvs diff` et autres `cvs update -j`). Il n'est pas recommandé d'archiver des versions intermédiaires non compilables (on peut cependant y être obligé si plusieurs développeurs doivent s'échanger les fichiers).

A priori, les incrémentations de versions ne se font qu'à l'occasion de distributions hors de l'équipe de développement et lors des créations de branches pour des corrections de bugs ou des évolutions susceptibles de durer. Pour les distributions, les tags doivent être de la forme : `release-4-5`.

À chaque nouvelle distribution, le fichier `NEWS` doit être mis à jour avec toutes les informations pertinentes pour les utilisateurs (l'objectif est donc différent de celui des fichiers `ChangeLog`).

Pour générer une distribution, utiliser la cible `dist` du `Makefile` (il existe également une cible `distcheck` qui permet de vérifier cette distribution). Le numéro de version de la distribution est paramétré par la macro `AC_INIT` dans le fichier `configure.ac`, ce numéro est ensuite propagé sous forme d'un `#define` dans le fichier `src/ClubConfig.h` généré par `configure`, et c'est ce `#define` qui est utilisé par la fonction `clubVersion`.

10 changements depuis les versions précédentes

10.1 évolutions entre la version 9.5 et la version 10.0

La DTD des fichiers XML a été complètement modifiée de façon à séparer la description des unités (généralement unique et partagée par tous) des données elles-mêmes. Le système de gestion des unités a été très largement étendu et amélioré à cette occasion. Cette modification introduit une incompatibilité avec les versions précédentes de la bibliothèque. La version de la bibliothèque XERCES utilisée est la 2.6.0 avec un patch corrigeant le problème XERCESC-1356 (DM-ID 242).

Les méthodes `getData` de la classe `DataFile` (et de sa classe dérivée `UniqDataFile`) retournant des instances aux classes `DataProxyName` et `DataProxyIndex` ont été éliminées (DM-ID 241).

Les outils libres utilisés pour le développement ont été mis à jour (voir section 9 pour les numéros de version courants), ces modifications n'ont aucun impact sur les utilisateurs.

10.2 évolutions entre la version 9.4 et la version 9.5

Corrections mineures de formats d'écriture de réels (FA-ID 33). Un indicateur de format long générait des avertissements sur le compilateur GNU sous Solaris.

Amélioration de la documentation de l'utilitaire difference (DM-ID 34). Explication des calculs d'erreurs et des seuils associés.

10.3 évolutions entre la version 9.3 et la version 9.4

Creation du service `TamponTexte::getTampon` pour éviter que la fonction **inline operator <<** ne retourne directement un membre protégé de la classe (DM-ID 29).

Amélioration de la documentation (DM-ID 30).

Suppression de la contrainte de fourniture du type de fichier pour l'instanciation de la classe `DataFile` (DM-ID 31).

La gestion des fichiers XML nécessite désormais la version 2.4.0 de la bibliothèque **Xerces**. L'option de configuration a également changé de nom pour refléter le nom standard sous lequel la bibliothèque est installée, il faut désormais utiliser `--with-xerces-c=/repertoire/installation/xerces` pour spécifier un répertoire d'installation non standard ou `--with-xerces-c=no` pour inhiber manuellement le support de **Xerces** (DM-ID 32).

10.4 évolutions entre la version 9.2 et la version 9.3

Une erreur de syntaxe a été corrigée dans `MadonaFile.cpp`. Cette erreur empêchait la compilation du support optionnel de la bibliothèque MADONA.

Un problème introduit après la version 3.6 de la bibliothèque MADONA a été contourné. Ce problème est lié à la gestion des références entre données, qui ne semble plus transparente.

Des erreurs d'initialisation introduites lors des interventions qualité du projet ATV ont été corrigées.

Les scripts de configuration ont été mis à niveau par rapport aux versions courantes des outils de développement GNU (`autoconf` version 2.57, `automake` version 1.7.5 et `libtool` version 1.5). Cette modification n'a pas d'impact pour les utilisateurs.

10.5 évolutions entre la version 9.1 et la version 9.2

La bibliothèque a été mise en conformité avec **Xerces** 1.7. Les dernières versions de la bibliothèque d'analyse de fichiers XML **Xerces** placent les fichiers d'en-tête dans un sous répertoire. Les directives d'inclusion de ces fichiers ont été modifiées en conséquence. La compatibilité avec des versions très anciennes de **Xerces** n'est pas assurée : les utilisateurs doivent mettre à jour leur version de **Xerces**.

Les scripts de configuration ont été corrigés. Quelques erreurs rares de configuration dans des cas inhabituels ont été corrigées. Il s'agit de corrections mineures n'affectant pas les utilisateurs habituels.

La méthode `FichierStructure::listeSousBlocs` a été ajoutée. Cette méthode permet d'explorer une structure pour déterminer les éléments qu'elle contient.

Les versions des outils de la suite de développement GNU (`libtool`, `autoconf` et `automake`) ont été prises en compte. Ces versions facilitent le développement en réduisant les dépendances par rapport aux autres produits GNU et en simplifiant les macros de test.

Pour répondre aux exigences qualité du projet ATV, les constructeurs, destructeurs et opérateurs d'asignation ont été explicitement définis dans chaque classe de la bibliothèque. Si ces méthodes ne devaient pas être disponibles dans l'interface publique, elles ont été créées en restreignant leur accès (`protected` ou `private`). L'ordre des sections `public`, `protected` et `private` a par ailleurs été uniformisé afin de faciliter les activités de maintenance. Ces modifications n'ont aucun impact sur le code appelant des bibliothèques.

Enfin les extensions des fichiers sources qui étaient de la forme `.cc` ont été modifiés en `.cpp` afin de faciliter un portage ultérieur de la bibliothèque sous Windows.

10.6 évolutions entre la version 9.0 et la version 9.1

La classe `CallTrace` a été transférée depuis la bibliothèque `MARMOTTES`.

Une violation mémoire dans l'utilitaire `difference` a été corrigée. Cette erreur se produisait lors de l'affichage des différences entre lignes très longues (merci à Ole Kristian KVERNELAND).

La version 3.0 du compilateur `gcc` a détecté quelques erreurs nouvelles qui ont été corrigées. La bibliothèque reste compilable avec les versions 2.95.x du compilateur `gcc`.

10.7 évolutions entre la version 8.2 et la version 9.0

La classe `Adressage` a été supprimée de la bibliothèque. Cette classe n'était conservée qu'à titre de compatibilité. Les utilisateurs sont invités à s'appuyer sur les conteneurs disponibles dans la `STL`, par exemple la classe `map`. Le fichier d'en-tête `ClubHashFun.h` a également été supprimé. La classe `ChaineSimple` devrait suivre le même chemin et être supprimée rapidement.

Les tables de hachage dans les classes `OptionsAppli` et `Traducteur` implémentées jusqu'à présent avec `hash_map`, sont désormais représentées par la classe patron `map` de la `STL`, la classe `hash_map` étant une extension qui n'était pas disponible sur toutes les implémentations de la `STL`. Cette évolution n'a aucune incidence sur l'interface publique. Les implications de cette suppression sont que les performances d'accès aux clefs du système de traduction doivent passer de $O(1)$ à $O(\ln(n))$, où n est le nombre d'éléments du dictionnaire, et que la méthode `OptionsAppli::usage` affiche désormais les options dans l'ordre lexicographique au lieu de l'ordre dans lequel elles ont été créées.

La classe `AnalyseurLexical` peut désormais reconnaître les réels du langage `FORTTRAN` (c'est à dire les réels écrits en utilisant les lettres `d` ou `D` comme indicateur d'exposant). Cette reconnaissance est paramétrable.

L'utilitaire `difference` peut désormais traiter des fichiers créés par des programmes `fortran` utilisant les lettres `d` ou `D` comme indicateurs d'exposant.

Plusieurs erreurs détectées par l'outil `purify` ont été corrigées. Il s'agissait essentiellement de fuites de mémoire (dans la bibliothèque et dans de simples programmes de test), d'une désallocation avec le mauvais opérateur, et d'une réutilisation d'un pointeur en pile après retour d'une fonction. L'outil signale encore des fuites avérées ou potentielles qui ont été analysées individuellement, aucune erreur n'a été identifiée.

10.8 évolutions entre la version 8.1 et la version 8.2

Les seules modifications introduites dans la version 8.2 de la bibliothèque sont l'ajout du script `club-config` destiné à faciliter la compilation d'applicatifs dépendant de CLUB (cf 13.2, page 130). La version anglaise du fichier de licence a également été ajoutée dans la distribution.

10.9 évolutions entre la version 8.0 et la version 8.1

Les modifications apportées dans la version 8.1 sont essentiellement des modifications de configuration et de tests de non-régression, une petite homogénéisation mineure dans un fichier source a également été réalisée. Les fonctionnalités du code n'ont pas évolué.

La première modification de configuration concerne les tests des bibliothèques optionnelles XERCES et MADONA. Les nouvelles macros couvrent plus de cas, par exemple lorsque les fichiers d'en-tête de madona se trouvent dans un répertoire `include/madona` en plus des cas où ils se trouvent directement dans un répertoire `include`. De plus les dépendances entre bibliothèques sont mieux gérées.

La seconde modification de configuration provient d'un problème rencontré sous solaris lors des tests de MARMOTTES. Les exceptions générées dans CLUB ne sont pas récupérées par CLUB lorsque toutes les bibliothèques sont partagées. Le problème ne se pose pas avec des bibliothèques statiques. Dans l'attente d'une meilleure compréhension du phénomène, la configuration par défaut sous solaris consiste donc à ne construire que des bibliothèques statiques. L'utilisateur aventureux peut toujours construire des bibliothèques partagées en utilisant l'option `--enable-shared` du script de configuration.

Les tests de non-régression de la classe `MadonaFile` ont été améliorés. Ils testent désormais les variables d'environnement `madona` avant de se lancer.

10.10 évolutions entre la version 7.0 et la version 8.0

La bibliothèque supporte de nouveaux formats de fichiers structurés en plus du format historique Fichier-Structure. Ces formats sont Madona (un format propriétaire du CNES) et XML. Ces formats sont accessibles soit directement à travers les classes `MadonaFile`, `StructureFile` et `XMLFile`, soit accessibles à travers une interface commune `DataFile`. Il est dès lors possible d'écrire des applications ignorant dans une large mesure le type exact de fichier réellement utilisé. Ces classes expérimentales sont susceptibles de subir des modifications importantes. La gestion de fichiers au format Madona s'appuie sur la bibliothèque Madona 3.3, la gestion des fichiers XML utilise, quant à elle, la bibliothèque Xerces 1.3.0. Si ces bibliothèques ne sont pas disponibles (ou si l'utilisateur ne souhaite pas les utiliser¹¹) CLUB sera compilée sans le support de ces formats.

La bibliothèque CLUB peut être générée sous forme partagée à l'aide de l'outil GNU `libtool`.

10.11 évolutions entre la version 6.3 et la version 7.0

L'implémentation des classes de CLUB s'appuie désormais sur la STL et notamment sur les strings. La classe `ChaineSimple` n'est donc plus utilisée par CLUB. Elle reste néanmoins disponible dans l'interface publique de la bibliothèque. Il en résulte des problèmes de compatibilité entre les versions 6 et 7 (cf 3.3 page 11).

¹¹voir la description des options `--with-xerces=no` et `--with-madona=no` dans la section 6

La gestion des formats du langage C qui était réalisée en interne de la classe Traducteur, est désormais implémentée dans une classe indépendante nommée FormatC. Elle fournit des services semblables à ceux de la classe FormatFortran.

La classe ChaineSimple a été complétée par un constructeur à partir d'un string de la STL et par un opérateur de conversion implicite de ChaineSimple en string. Cette évolution doit permettre de favoriser l'utilisation de la classe string de la STL plutôt que celle de ChaineSimple (ces deux classes fournissant des services identiques).

Les tables de hachage dans les classes OptionsAppli et Traducteur implémentées jusqu'à présent avec Adressage, sont désormais représentées par la classe patron hash_map de la STL. Pour réaliser cette conversion, a été créé un fichier d'interface nommé ClubHashFun.h déclarant les fonctions de hachage propres à CLUB. Cette évolution n'a aucune incidence sur l'interface publique.

Enfin, les macros m4 de configuration internes au CNES ont été remplacées, lorsque c'était possible, par les macros de l'archive publique autoconf.

11 évolutions possibles

Il serait souhaitable d'augmenter la couverture de tests de la bibliothèque CLUB. Ceci sous-entend à la fois créer des tests pour les classes qui n'en ont pas mais également compléter les tests existants.

La classe AnalyseurLexical reconnaît comme nom toute suite de caractères commençant par une lettre ou le caractère _ et se poursuivant par des lettres, des chiffres, ou des caractères _. Ce choix est calqué sur les règles des langages de programmation classiques, il faudrait peut-être le rendre paramétrable. Cette évolution est à envisager avec précautions, en effet une seule application (la bibliothèque de filtrage de télémétrie décommutée au format MERCATOR) a rencontré cette limite à ce jour, et la difficulté a été contournée sans intervenir sur le moindre logiciel.

L'analyseur d'options devrait reconnaître les abréviations non ambiguës, par exemple si les options entières -maximum et -minimum existaient toutes les deux, alors la ligne de commande `prgm -minmax 10 20` devrait affecter 10 au minimum et 20 au maximum du programme.

La classe FichierStructure n'est utilisée que par la bibliothèque de haut niveau MARMOTTES, il faudrait probablement la changer de bibliothèque.

La classe TamponTexte devrait pouvoir interpréter les retours chariot et les coupures de ligne.

12 description des classes

Les classes et fonctions décrites dans cette section composent l'interface utilisateur de la bibliothèque CLUB.

12.1 classe AnalyseurLexical

description

La classe AnalyseurLexical permet d'extraire les composants élémentaires d'une chaîne de caractères (entiers, réels, noms, chaînes littérales, opérateurs, séparateurs, parenthèses).

Elle peut être utilisée par des analyseurs syntaxiques de plus haut niveau, qui eux gèreront l'interaction des lexèmes extraits, pour constituer par exemple des expressions à partir des noms, des nombres et des opérateurs pour une grammaire d'expressions, ou des dates sous des formats variables à partir d'entiers et de séparateurs.

interface publique

```
#include "club/AnalyseurLexical.h"
```

La classe définit un type énuméré anonyme public pour représenter les différents types de lexèmes reconnus, les méthodes d'analyse retournent des valeurs choisies au sein de cette énumération.

```
enum { codeFin,                codeSeparateur, codeOperateur, codeParenthese,
      codeEntier,             codeReel,          codeNom,          codeChaineC,
      codeChaineFortran, codeInconnu
};
```

TAB. 2: AnalyseurLexical : méthodes publiques

signature	description
AnalyseurLexical (const string& <i>chaîne</i>)	construit un analyseur pour la <i>chaîne</i>
AnalyseurLexical (const char * <i>chaîne</i> = 0)	construit un analyseur pour la <i>chaîne</i>
AnalyseurLexical (const char * <i>chaîne</i> , int <i>longueur</i>)	construit un analyseur pour les <i>longueur</i> premiers caractères de la <i>chaîne</i>
AnalyseurLexical (const AnalyseurLexical& <i>a</i>)	constructeur par copie
AnalyseurLexical& operator = (const AnalyseurLexical& <i>a</i>)	affectation
AnalyseurLexical ()	destructeur de la classe.
void initBlancs (const string& <i>blancs</i>)	indique quels caractères doivent être reconnus comme blancs
void initBlancs (const char * <i>blancs</i>)	indique quels caractères doivent être reconnus comme blancs
void initSéparateurs (const string& <i>separateurs</i>)	indique quels caractères doivent être reconnus comme séparateurs
à suivre ...	

TAB. 2: AnalyseurLexical : méthodes publiques (suite)

signature	description
void initSeparateurs (const char *separateurs) void initOperateurs (const string& operateurs) void initOperateurs (const char *operateurs) void autoriseReels () void interditReels () void autoriseReelsFortran () void interditReelsFortran () void autoriseChainesC () void interditChainesC () void autoriseChainesFortran () void interditChainesFortran ()	indique quels caractères doivent être reconnus comme séparateurs indique quels caractères doivent être reconnus comme opérateurs indique quels caractères doivent être reconnus comme opérateurs autorise la reconnaissance des réels interdit la reconnaissance des réels autorise la reconnaissance des réels du langage FORTRAN interdit la reconnaissance des réels du langage FORTRAN autorise la reconnaissance des chaînes littérales du langage C interdit la reconnaissance des chaînes littérales du langage C autorise la reconnaissance des chaînes littérales du langage FORTRAN interdit la reconnaissance des chaînes littérales du langage FORTRAN
void reinitialise (const string& c) void reinitialise (const char *c) void reinitialise (const char *c, int longueur)	réinitialise l'instance comme si elle venait juste d'être construite avec le même argument réinitialise l'instance comme si elle venait juste d'être construite avec le même argument réinitialise l'instance comme si elle venait juste d'être construite avec les mêmes arguments
void redemarre () int suivant () int precedent () int etatCourant () const int retourneAEtat (int etat)	repositionne l'analyseur au début de la chaîne analyse le lexème suivant et retourne son type reprend l'analyse du lexème précédent et retourne son type (cette fonction peut être coûteuse, car elle reprend toute l'analyse au début) retourne un index représentant l'état courant repositionne l'analyseur dans l'état récupéré par un appel préalable à etatCourant () et retourne le type du lexème correspondant
const string& chaine () const string& lexeme () int debutLexeme () int apresLexeme () int type () int entier () double reel () const string& nom () const string& chaineC () const string& chaineFortran () char separateur () char parenthese ()	retourne la totalité de la chaîne analysée retourne le texte du dernier lexème reconnu donne l'index dans la chaîne du début du dernier lexème reconnu donne l'index dans la chaîne du caractère suivant le dernier lexème reconnu retourne le type du dernier lexème reconnu (sans reprendre l'analyse) retourne la valeur du dernier entier reconnu retourne la valeur du dernier réel reconnu retourne le dernier nom reconnu retourne la dernière chaîne C reconnue retourne la dernière chaîne FORTRAN reconnue retourne le dernier séparateur reconnu retourne la dernière parenthèse reconnue
à suivre ...	

TAB. 2: AnalyseurLexical : méthodes publiques (suite)

signature	description
char operateur ()	retourne le dernier opérateur reconnu

exemple d'utilisation

```
#include "club/AnalyseurLexical.h"
...

// analyse d'une date en calendrier ou en jour julien
AnalyseurLexical al (argv [i]);
double t;

// on commence par tenter un format calendrier
al.initSeparateurs (":/-.");
al.interditReels ();
int n [7];
for (int j = 0; j < 7; j++)
    n [j] = 0;
int nbEntiers = 0;
int ok = 1;

(void) al.suivant ();
while (ok && (al.type () == AnalyseurLexical::codeEntier))
{ // boucle sur les entiers du format
    n [nbEntiers++] = al.entier ();
    ok = (nbEntiers <= 7);

    if (al.suivant () != AnalyseurLexical::codeSeparateur)
    { // les séparateurs sont optionnels entre année et heure ou à la fin
        ok = ok
            && (((nbEntiers == 3) && (al.type () == AnalyseurLexical::codeEntier))
                || (al.type () == AnalyseurLexical::codeFin));
    }
    else
        (void) al.suivant ();
}

ok = ok
    && ((nbEntiers == 3) || (nbEntiers == 6) || (nbEntiers == 7))
    && (al.type () == AnalyseurLexical::codeFin);

if (ok)
{ // correction de l'année si elle n'est que sur deux chiffres
    if (n [2] < 50)
        n [2] += 2000;    // années 2000 à 2049, notées 00 à 49
    else if (n [2] < 100)
```

```

n [2] += 1900;    // années 1950 à 1999, notées 50 à 99

t = date2000 (n [0], n [1], n [2], n [3], n [4], n [5], n [6]);

}
else
{ // le format calendaire ne convenait pas, on essaie un simple réel
  al.initSeparateurs ("");
  al.autoriseReels    ();
  if ((al.suivant () == AnalyseurLexical::codeReel)
      &&
      (al.suivant () == AnalyseurLexical::codeFin))
    t = al.reel ();
  else
    t = 9.9999e99;
}

```

D'autres exemples d'utilisation peuvent être trouvés dans le code des classes FormatFortran et Tampon-Texte, ainsi que dans l'utilitaire `difference` décrit à la section 13.1.

conseils d'utilisation spécifiques

La classe est relativement paramétrable de façon à couvrir plusieurs cas d'analyseurs lexicaux. Il est nécessaire pour l'utiliser correctement de bien prendre garde au comportement par défaut et de positionner les indicateurs selon ses besoins spécifiques avant de commencer l'analyse.

Les comportements par défaut sont :

- la liste des blancs est "`\t\n\v\f\r`" (c'est à dire les caractères reconnus par la macro-définition standard du langage C `isspace`);
- la liste des séparateurs est vide;
- la liste des opérateurs est vide;
- les réels sont reconnus;
- les réels du langage FORTRAN ne sont pas reconnus;
- les chaînes du langage C ne sont pas reconnues;
- les chaînes du langage FORTRAN ne sont pas reconnues.

Lorsqu'un analyseur doit reconnaître des réels, il faut prendre garde que la chaîne `123` produira un lexème de type entier et non réel. Dans la plupart des cas l'utilisateur souhaitera que cette chaîne soit reconnue, il lui faudra alors prendre en compte la distinction entre réel et entier dans les codes de retour de l'analyseur et faire lui-même la coercition de type. Il faut également prendre garde au fait qu'interdire la reconnaissance des réels du fortran seul n'empêche pas la reconnaissance d'un réel du genre `1.23e+7`, elle n'empêche que la reconnaissance des exposants spécifiques au fortran, comme `1.23d+7`.

implantation

Les attributs sont décrits sommairement dans la table 3, les méthodes privées dans la table 4

TAB. 3: attributs de la classe AnalyseurLexical

nom	type	description
chaîne_	string	chaîne à analyser
debut_	const char *	pointeur sur le début du lexème en cours d'analyse
mobile_	const char *	pointeur de parcours des lexèmes
index_	int	index du lexème dans la chaîne
blancs_	string	liste des caractères considérés comme des blancs
separateurs_	string	liste des caractères considérés comme des séparateurs
opérateurs_	string	liste des caractères considérés comme des opérateurs
reconnaitReels_	bool	drapeau indiquant si les réels doivent être reconnus
reconnaitReelsFortran_	bool	drapeau indiquant si les réels de caractères du langage FORTRAN doivent être reconnus
reconnaitChainesC_	bool	drapeau indiquant si les chaînes de caractères du langage C doivent être reconnues
reconnaitChainesFortran_	bool	drapeau indiquant si les chaînes de caractères du langage FORTRAN doivent être reconnues
dernierLexeme_	string	copie du dernier lexème reconnu
dernierType_	int	type du dernier lexème
dernierEntier_	int	valeur du dernier entier reconnu
dernierReel_	double	valeur du dernier réel reconnu
dernierNom_	string	copie du dernier nom reconnu
derniereChaineC_	string	copie de la dernière chaîne C reconnue
derniereChaineFortran_	string	copie de la dernière chaîne FORTRAN reconnue
dernierSeparateur_	char	copie du dernier séparateur reconnu
derniereParenthese_	char	copie de la dernière parenthèse reconnue
dernierOperateur_	char	copie du dernier opérateur reconnu

TAB. 4: AnalyseurLexical : méthodes privées

signature	description
void analyseReelOuEntier ()	fonction d'analyse reconnaissant les réels et les entiers
void analyseEntier ()	fonction d'analyse reconnaissant les entiers
void analyseChaineC ()	fonction d'analyse reconnaissant les chaînes du langage C
void analyseChaineFortran ()	fonction d'analyse reconnaissant les chaînes du langage FORTRAN

12.2 classe BaseErreurs

description

Cette classe est une classe de base permettant de formater et traduire dans la langue de l'utilisateur des messages d'erreur, les classes dérivées devant spécialiser la classe de base pour tel ou tel domaine (ou

bibliothèque). La bibliothèque CLUB par exemple utilise ce mécanisme pour gérer ses propres erreurs, par l'intermédiaire de la classe ClubErreurs, dérivée de BaseErreurs. Cette classe est articulée autour du principe de séparation entre le formatage d'un message d'erreur, qui doit être fait au plus bas niveau (à l'intérieur de la bibliothèque), là où on dispose des éléments pour constituer le message (noms des fichiers que l'on ne peut pas ouvrir, type de l'opération générant un problème, ...), et la diffusion de ce message (sur la sortie d'erreur standard, dans une zone de texte d'une IHM, par requête auprès d'un serveur de journal de bord, ...), qui ne peut être faite qu'à très haut niveau, par le programme principal qui utilise la bibliothèque.

La solution consiste à formater la chaîne à la construction de l'objet, sans rien faire de plus. Si des systèmes de diffusion des erreurs existent dans l'appelant, celui-ci peut récupérer cette chaîne et signaler à l'instance qu'il se charge de tout (en appelant la fonction membre **correction**), il peut également demander son affichage sur un **ostream** particulier. Si, à la destruction de l'instance, l'erreur n'a pas été corrigée, alors le destructeur affiche le message sur la sortie d'erreur standard. Toute copie ou affectation d'une erreur est une copie d'instance dont le message et le code d'erreur sont partagés avec l'erreur d'origine. Ainsi, seule la destruction de la dernière des copies peut donner lieu à un affichage, ce qui évite un affichage multiple des messages d'erreur résultant de la destruction d'objets temporaires créés implicitement par le compilateur, par exemple lors du lancement d'une exception. D'autre part, si l'original ou une des copies est corrigé, toutes les instances partageant le même message et le même code d'erreur, seront également corrigées. Ceci permet d'éviter l'affichage d'un message d'erreur dû à la destruction d'une instance d'erreur temporaire alors que l'erreur originale a été rattrapée.

Ce système présente l'avantage de fonctionner que l'appelant se contente d'une gestion minimaliste des erreurs (s'il appelle **exit**, le destructeur de la classe réalisera seul l'affichage) ou qu'il utilise un système très sophistiqué de journal de bord (il peut demander l'affichage sur un **ostream** encapsulant une **socket** connectée à une machine distante, ou écrire le message dans une zone de texte de l'IHM de contrôle de son programme, ...). La séparation du formatage et de la diffusion de l'erreur autorise un découplage entre le programme appelant et la bibliothèque, le programme pouvant complètement ignorer les types d'erreurs générés par la bibliothèque (il peut ne voir que des pointeurs sur la classe de base s'il le désire). Si au contraire des systèmes de récupération sur erreur existent (c'est le cas dans la bibliothèque CANTOR batie au dessus de CLUB), alors il peut tester les codes d'erreur qui sont disponibles et réagir à bon escient.

interface publique

```
#include "club/BaseErreurs.h"
```

TAB. 5: BaseErreurs : méthodes publiques

signature	description
BaseErreurs ()	construit une instance ne décrivant aucune erreur
BaseErreurs (int <i>code</i> ...)	construit une instance avec le <i>code</i> d'erreur fourni, en utilisant la langue courante, le format lié au <i>code</i> et les arguments variables suivants pour formater le message
BaseErreurs (const char * <i>format</i> ...)	construit une instance avec un code d'erreur inconnu, en utilisant la langue courante, le <i>format</i> du langage C et les arguments variables suivants pour formater le message
BaseErreurs (const BaseErreurs& <i>b</i>)	constructeur par copie, par création d'une instance partagée avec l'erreur d'origine <i>b</i>
BaseErreur& operator = (const BaseErreurs& <i>b</i>)	affectation, avec partage de l'erreur <i>b</i>
à suivre ...	

TAB. 5: BaseErreurs : méthodes publiques (suite)

signature	description
<code>~BaseErreurs ()</code>	destructeur, affiche le message par la méthode affiche s'il n'a pas été corrigé depuis sa création, ne fait rien dans le cas contraire. Cette fonction peut être redéfinie dans les classes dérivées, mais ce n'est pas obligatoire, la fonction de la classe de base pouvant généralement suffire.
<code>void miseAJour (int code ...)</code> <code>void miseAJour (const char *format ...)</code> <code>inline void correction ()</code>	met à jour une instance comme si elle était construite pour la première fois met à jour une instance comme si elle était construite pour la première fois remet à AucuneErreur le code d'erreur de l'instance, ce qui permet d'éviter son affichage lors de la destruction de l'instance
<code>inline ostream& affiche (ostream& s) const</code> <code>inline void affiche () const</code>	affiche le message formaté sur le flot de sortie <i>s</i> affiche le message formaté en passant par le fonction d'affichage de l'utilisateur s'il en a défini une, en écrivant sur cerr dans le cas contraire
<code>const char *why () const</code> <code>int code () const</code> <code>virtual const char *domaine () const</code> <code>int correspond (int code, const char *domaine) const</code> <code>inline int existe () const</code>	retourne le message d'erreur (le nom anglais de cette méthode était destiné à se conformer au standard ... qui a changé) retourne le code de l'erreur retourne le domaine d'erreur (" base " pour la classe de base). Cette fonction est destinée à autoriser une certaine forme de typage dynamique, l'appelant pouvant par cette fonction virtuelle savoir si l'instance est du type de base BaseErreurs ou d'un type dérivé. Il est <i>impératif</i> que cette fonction soit redéfinie dans les classes dérivées. cette méthode permet de savoir si une erreur correspond à un code attendu dans un domaine précisé. Il est primordial d'utiliser cette fonction pour faire les tests plutôt que code () , car les codes du type de base et des types dérivés peuvent se recouper (ce sont des énumérés), tout test sur le code doit donc être associé à un test sur le domaine indique si une erreur existe, c'est à dire si le code est différent de AucuneErreur

TAB. 6: BaseErreurs : méthodes protégées

signature	description
<code>void stockeMessage (const char *message)</code> <code>void stockeMessage (const string& message)</code>	stocke le message donné en argument dans l'instance. Cette méthode est utilisée par le constructeur de la classe et les constructeurs des classes dérivées stocke le message donné en argument dans l'instance. Cette méthode est utilisée par le constructeur de la classe et les constructeurs des classes dérivées mais aussi par toutes les fonctions qui doivent modifier la valeur du code d'erreur
à suivre ...	

TAB. 6: BaseErreurs : méthodes protégées (suite)

signature	description
void stockeCode (int <i>code</i>)	stocke le code donné en argument dans l'instance. Cette méthode est utilisée par le constructeur de la classe et les constructeurs des classes dérivées
virtual const string& formate (int <i>code</i> , va_list <i>ap</i>)	traduit et formate le message en fonction du <i>code</i> . La sélection en fonction du <i>code</i> implique que cette méthode doit <i>impérativement</i> être redéfinie dans les classes dérivées
const string& formate (const char * <i>format</i> , va_list <i>ap</i>)	traduit et formate le message à partir du <i>format</i> du langage C

TAB. 7: BaseErreurs : méthodes de classe

signature	description
void initErreurs (void (*f) (const char*, void*), void* arg, int exceptions = 0, int avertissements = 0)	initialise la gestion <i>globale</i> des erreurs, en enregistrant une fonction d'affichage (dont les arguments sont le message et un pointeur anonyme), en décidant si la méthode avertissements doit ou non être active (pour instrumenter du code), et s'il faut générer des exceptions ou créer et détruire des objets locaux lorsque l'utilisateur n'a pas donné d'instance lors d'un appel aux méthodes erreur
inline int okavt () inline int exceptions () inline void (* fonAffiche ()) (const char*, void*) inline void * argAffiche ()	indique si les appels à la méthode avertissement sont ou non actifs indique si des exceptions doivent être générées lors des appels à erreur retourne un pointeur sur la fonction d'affichage utilisateur initialisée par un appel préalable à initErreurs retourne un pointeur sur les arguments anonymes de la fonction d'affichage utilisateur initialisée par un appel préalable à initErreurs
void avertissement (const char * <i>format</i> ...) void erreur (BaseErreurs * <i>ptr</i> , int <i>code</i> ...) void erreur (BaseErreurs * <i>ptr</i> , const char * <i>format</i> ...)	génère immédiatement un avertissement sur le système courant d'affichage (si les avertissements sont actifs) génère une erreur et la stocke dans l'instance pointée par <i>ptr</i> , s'il est non nul. génère une erreur et la stocke dans l'instance pointée par <i>ptr</i> , s'il est non nul

exemple d'utilisation

```
#include "club/BaseErreurs.h"
...
class NouvelleErreur : public BaseErreurs
{
protected :
    virtual const string& formate (int code, va_list ap) const;

public :
    // liste des codes d'erreurs
    enum { premiere_erreur = 1, deuxieme_erreur, troisieme_erreur };
};
```

```
// constructeurs
NouvelleErreurs      () : BaseErreurs () {}
NouvelleErreurs      (int code ...);
NouvelleErreurs      (const char* format ...);
NouvelleErreurs      (const NouvelleErreurs& e) : BaseErreurs (e) {}
// destructeur
virtual ~NouvelleErreurs () {}

// affectation
NouvelleErreurs& operator =      (const NouvelleErreurs& e);

// fonctions d'accès aux éléments canoniques
virtual const char* domaine      () const { return "nouvelle"; }
// déclenchement général d'une erreur
static int erreur (NouvelleErreurs* ptr, int code ...);
static int erreur (NouvelleErreurs* ptr, const char* format ...);
};
```

conseils d'utilisation spécifiques

Cette classe de base permet de construire simplement des classes d'erreurs assez souples. Les classes dérivées doivent implanter elles-mêmes un certain nombre de méthodes (par exemple pour le formatage, qui ne peut qu'être spécifique). Il se trouve cependant que des portions de code absolument similaires d'une classe à l'autre doivent être recopiées : les méthodes de classe, qui ne peuvent pas être héritées.

La façon la plus simple de créer une classe dérivée de BaseErreurs est donc de recopier directement une classe dérivée existante (par exemple ClubErreurs), puis de faire un remplacement exhaustif des chaînes ClubErreurs par XxxxErreurs, et enfin de changer la liste des codes d'erreurs et la chaîne retournée par **domaine** dans le fichier d'en-tête et le corps de la méthode **formate** dans le fichier .cc.

Le principe de copie d'instance (partage du message d'erreur par un TamponPartage) lors de l'appel au constructeur par copie ou à l'opérateur d'affectation est à conserver.

Les versions supérieures ou égales à 7.0 de BaseErreurs utilisent les **strings** de la STL pour l'implémentation de la classe. L'interface publique n'a pas été modifiée, en revanche, la méthode protégée **formate** a changé de signature. Il en résulte que les classes dérivées de BaseErreurs doivent impérativement évoluer pour être conforme avec la version 7.0 de CLUB.

Cette transition se résume à modifier la signature de la fonction formate et à changer le type de la variable **tampon** et les éventuels appels du type :

```
tampon->formate (trad ("%s\" = %d, hors bornes [%d; %d]"),
                TraduitVersExterne (chaine_1),
                entier_1, entier_2, entier_3);
en
appliqueFormat (trad ("%s\" = %d, hors bornes [%d; %d]"),
                tampon,
                TraduitVersExterne (chaine_1),
                entier_1, entier_2, entier_3);
```


où appliqueFormat est du type :

```
static void appliqueFormat (const string& format, string *ptrDst ...)
{
    va_list ap;
    va_start(ap, ptrDst);
    FormatC f (format);
    ap = f.applique (ap, ptrDst);
    va_end(ap);
}
```

implantation

Les attributs sont décrits sommairement dans la table 8, les attributs statiques dans la table 9

TAB. 8: attributs de la classe BaseErreurs

nom	type	description
donneesPartagees_	TamponPartage	instance de classe contenant le code d'erreur et le message d'erreur traduit et formaté

TAB. 9: attributs statiques de la classe BaseErreurs

nom	type	description
exceptions_	int	indicateur de génération d'exceptions par les méthodes erreur
avertissements_	int	indicateur d'activité de la méthode avertissements
argAffiche_	void *	pointeur anonyme sur les données de la fonction d'affichage utilisateur
fonAffiche_	void (*fonAffiche_) (const char *, void *)	pointeur sur la fonction d'affichage utilisateur

12.3 classe ChaineSimple

description

Cette classe présente à l'utilisateur une interface simple pour les opérations les plus courantes que l'on désire réaliser sur les chaînes de caractères. Les trois opérations principales sont l'initialisation (ou la réinitialisation ou l'affectation), l'extension et la comparaison. Elles peuvent être élaborées à partir d'opérandes de types divers comme des chaînes (ou des sous-chaînes) du langage C (**const char ***), des couples caractère et facteur de répétition, ou encore d'autres instances de **ChaineSimple**.

D'autres opérations de moindre importance sont également proposées par la classe, il s'agit de la modification de caractères individuels dans la chaîne (par exemple pour découper un chemin de la forme **" / : / tmp : / usr / local / bin : . "** au niveau des séparateurs), l'élimination des blancs initiaux, des blancs finaux, l'écriture dans la chaîne à l'aide d'un format C.

Les chaînes respectent le format standard du langage C, c'est à dire qu'elles sont obligatoirement terminées par le caractère `'\0'`. La classe s'en assure seule.

interface publique

```
#include "club/ChaineSimple.h"
```

TAB. 10: ChaineSimple : méthodes publiques

signature	description
ChaineSimple (int <i>taille</i> = -1)	construit une instance vide, ayant au départ au moins <i>taille</i> caractères
ChaineSimple (const char * <i>chaine</i>)	construit une instance en copiant la <i>chaine</i> (jusqu'au premier <code>'\0'</code>)
ChaineSimple (const string& <i>chaine</i>)	construit une instance en copiant la <i>chaine</i> (jusqu'au premier <code>'\0'</code>)
ChaineSimple (const char * <i>chaine</i> , int <i>longueur</i>)	construit une instance en copiant les <i>longueur</i> premiers caractères de la <i>chaine</i>
ChaineSimple (char <i>caractere</i> , int <i>repetition</i>)	construit une instance, constituée de <i>repetition</i> occurrences du <i>caractere</i>
ChaineSimple (const ChaineSimple& <i>c</i>)	constructeur par copie
ChaineSimple& operator = (const ChaineSimple& <i>c</i>)	affectation
ChaineSimple& operator = (const char * <i>chaine</i>)	affectation
~ChaineSimple ()	destructeur, libère la mémoire allouée
operator const char * ()	convertisseur en chaîne du langage C
operator string ()	convertisseur en string de la STL
void reinitialise (const ChaineSimple& <i>c</i>)	réinitialise l'instance à partir de <i>c</i>
void reinitialise (const char * <i>chaine</i>)	réinitialise l'instance à partir de <i>chaine</i>
void reinitialise (const char * <i>chaine</i> , int <i>longueur</i>)	réinitialise l'instance à partir des <i>longueur</i> premiers caractères de <i>chaine</i>
void reinitialise (char <i>caractere</i> , int <i>repetition</i>)	réinitialise l'instance à partir de <i>repetition</i> occurrences du <i>caractere</i>
void formate (const char * <i>format</i> ...)	remplace l'instance par la chaîne résultant de l'écriture des arguments variables selon le <i>format</i> (équivalent d'un sprintf du langage C)
void vFormate (const char * <i>format</i> , va_list ap)	remplace l'instance par la chaîne résultant de l'écriture des arguments variables selon le <i>format</i> (équivalent d'un vsprintf du langage C)
void modifieCaractere (int <i>i</i> , char <i>c</i>)	remplace le caractère d'index <i>i</i> par <i>c</i> (<i>i</i> peut varier de 0 à taille () - 1, <i>c</i> peut être égal à <code>'\0'</code>)
void elimineBlancsFinaux ()	élimine les blancs au sens du isspace du langage C
void elimineBlancsInitiaux ()	élimine les blancs au sens du isspace du langage C
inline void recadre ()	appelle elimineBlancsFinaux () puis elimineBlancsInitiaux ()
à suivre ...	

TAB. 10: ChaîneSimple : méthodes publiques (suite)

signature	description
ChaîneSimple& operator += (const ChaîneSimple& c)	ajoute c à la fin de l'instance
ChaîneSimple& operator += (const char *chaîne)	ajoute chaîne à la fin de l'instance
ChaîneSimple& operator += (char c)	ajoute c à la fin de l'instance
inline int taille ()	retourne la taille de la mémoire allouée
inline int longueur ()	retourne la longueur courante de la chaîne
inline const char * premier ()	retourne un pointeur sur le premier caractère de la chaîne
inline const char * dernier ()	retourne un pointeur sur le dernier caractère de la chaîne (avant le '\0' final)
ChaîneSimple operator () (const char *debut, const char *fin) const	retourne la sous-chaîne débutant au <i>debut</i> et se terminant à la <i>fin</i>
ChaîneSimple operator () (int debut, int fin) const	retourne la sous-chaîne débutant au <i>debut</i> et se terminant à la <i>fin</i> (en comptant les caractères à partir de 0)
int operator == (const char *chaîne)	teste l'égalité du contenu de l'instance et de la chaîne
int operator == (const ChaîneSimple& chaîne)	teste l'égalité du contenu de l'instance et de la chaîne
int operator != (const char *chaîne)	teste l'inégalité du contenu de l'instance et de la chaîne
int operator != (const ChaîneSimple& chaîne)	teste l'inégalité du contenu de l'instance et de la chaîne

TAB. 11: ChaîneSimple : fonctions externes

signature	description
ChaîneSimple operator + (const ChaîneSimple& c1, const ChaîneSimple& c2)	concatène les chaînes c1 et c2
ChaîneSimple operator + (const ChaîneSimple& c1, const char *c2)	concatène les chaînes c1 et c2
ChaîneSimple operator + (const ChaîneSimple& c1, char c2)	concatène la chaîne c1 et le caractère c2
ChaîneSimple operator + (const char *c1, const ChaîneSimple& c2)	concatène les chaînes c1 et c2
ChaîneSimple operator + (char c1, const ChaîneSimple& c2)	concatène le caractère c1 et la chaîne c2

exemple d'utilisation

```
#include "club/ChaîneSimple.h"
...
void nomFichier (const char *repertoire)
{ // recherche du fichier "fic" dans un répertoire

    // constitution du nom complet du fichier
    ChaîneSimple nomFichier (repertoire);
    if (*nomFichier.dernier () != '/')
```

```

    nomFichier += "/";
    nomFichier += "fic";

    if (access (nomFichier.premier (), R_OK) == 0)
        ...
}

```

conseils d'utilisation spécifiques

La classe `ChaineSimple` pouvant être construite à partir d'un seul argument de type `const char*` et pouvant également être convertie en `const char *`, elle peut être utilisée directement à la place d'une chaîne du langage C dans les appels de fonctions les plus divers. Les règles de conversion automatique par le compilateur C++ ne sont cependant pas simples, et il peut être prudent d'utiliser des signatures non ambiguës.

Il faut également prendre garde aux fonctions à argument variable (comme par exemple les `printf` ou les méthodes `erreur` des classes dérivées de `BaseErreurs`), qui ne peuvent pas accepter directement des objets de type `ChaineSimple` (le compilateur le signale). Il faut dans ce cas passer le pointeur vers la chaîne C sous-jacente par la méthode `premier ()`.

Il est recommandé d'utiliser de préférence la classe `string` de la STL qui fournit des services équivalents et qui fait partie de la norme du langage C++. La section 3.3 page 11 donne des précisions sur la manière de substituer `ChaineSimple` par `string`.

implantation

Les attributs sont décrits sommairement dans la table 12, les méthodes privées dans la table 13

TAB. 12: attributs de la classe `ChaineSimple`

nom	type	description
<code>taille_</code>	<code>int</code>	taille de la mémoire allouée
<code>longueur_</code>	<code>int</code>	longueur courante de la chaîne
<code>chaîne_</code>	<code>char *</code>	pointeur sur la mémoire allouée

TAB. 13: `ChaineSimple` : méthodes privées

signature	description
<code>void alloueChaine (int <i>taille</i>)</code>	alloue une chaîne comprenant au moins <i>taille</i> caractères
<code>void etendChaine (int <i>taille</i>)</code>	alloue une nouvelle chaîne comprenant au moins <i>taille</i> caractères et copie dedans les caractères de la chaîne existante

Les chaînes sont constituées de paquets de tailles fixes (appelés cellules dans le code) prélevés dans des tableaux eux-mêmes alloués dynamiquement. Les cellules provenant des chaînes détruites (ou redimensionnées) sont simplement remises à disposition dans les tableaux, lesquels sont désalloués lorsqu'ils sont vides. Chaque tableau garde la même taille tout au long de sa vie.

Ces deux principes de mémoire cache et de cellules de taille fixes ont été choisis spécifiquement pour les chaînes de caractères, qui sont souvent petites et volatiles.

12.4 classe CallTrace

description

Cette classe permet d'instrumenter les méthodes d'autres classes pour enregistrer dans un fichier tous les appels à ces méthodes.

Les méthodes publiques de la classe Marmottes [DR2] ont été instrumentées de la sorte, ce qui permet de rejouer des scénarios sans pour autant disposer des programmes appelants. Ceci est utilisé d'une part pour créer des tests de non-régression représentatifs de l'utilisation réelle de la bibliothèque, et d'autre part pour identifier et corriger les problèmes qu'ils peuvent rencontrer.

Cette classe est une utilisation du modèle de conception¹² du *singleton*. La classe gérant un fichier permettant d'enregistrer les appels, il faut garantir l'unicité de ce fichier.

interface publique

```
#include "club/CallTrace.h"
```

TAB. 14: CallTrace : méthodes publiques

signature	description
static CallTrace * getInstance () throw (ClubErreurs)	retourne l'instance unique (<i>singleton</i>) de la classe
void activate (string <i>fileName</i>) throw (ClubErreurs)	active l'enregistrement des appels aux méthodes instrumentées dans le fichier <i>fileName</i> , le fichier est écrasé s'il existe déjà.
void deactivate ()	désactive l'enregistrement des appels et ferme le fichier.
bool isActive ()	teste si l'enregistrement des appels a été activé ou non.
void putToSleep ()	suspend momentanément l'enregistrement des appels
void wakeUp ()	repréend l'enregistrement des appels
void registerObject (const void * <i>p</i>)	enregistre une référence sur un objet spécifié par le pointeur <i>p</i> , afin de le reconnaître ultérieurement dans un autre appel
void unregisterObject (const void * <i>p</i>)	supprime la référence à un objet enregistré au préalable
bool isRegistered (const void * <i>p</i>) const	teste si un objet était enregistré
void startCall (string <i>functionName</i> , const void * <i>object</i>) throw (ClubErreurs)	démarre l'enregistrement d'un appel à la méthode <i>functionName</i> pour l'objet pointé par <i>object</i>
void endCall () throw (ClubErreurs)	arrête l'enregistrement d'un appel
à suivre ...	

¹² *design pattern*

TAB. 14: CallTrace : méthodes publiques (suite)

signature	description
void startResult () throw (ClubErreurs)	démarre l'enregistrement des résultats d'un appel
void endResult () throw (ClubErreurs)	arrête l'enregistrement des résultats d'un appel
void logBool (bool b) throw (ClubErreurs)	enregistre la valeur d'un booléen, dans un appel ou dans les résultats d'un appel
void logInt (int i) throw (ClubErreurs)	enregistre la valeur d'un entier, dans un appel ou dans les résultats d'un appel
void logDouble (double d) throw (ClubErreurs)	enregistre la valeur d'un double, dans un appel ou dans les résultats d'un appel
void logString (string s) throw (ClubErreurs)	enregistre la valeur d'une chaîne, dans un appel ou dans les résultats d'un appel
void logPointer (void *p) throw (ClubErreurs)	enregistre la valeur d'un pointeur, dans un appel ou dans les résultats d'un appel

exemple d'utilisation

L'exemple suivant montre comment la méthode **reinitialise** de la classe Marmottes a été instrumentée. L'objectif étant de reproduire les appels réalisés par l'utilisateur et uniquement ceux-ci, on supprime temporairement l'enregistrement lors de l'appel à la méthode **senseur** qui est également instrumentée.

```
#include "club/CallTrace.h"

...

if (trace->isActive ())
{
    trace->registerObject ((void *) this);
    trace->startCall (string ("Marmottes::reinitialise"), (void *) this);
    trace->logDouble (date);
    trace->logDouble (position.x ());
    trace->logDouble (position.y ());
    trace->logDouble (position.z ());

    ...

    trace->logString (string (senseur1));
    trace->logString (string (senseur2));
    trace->logString (string (senseur3));
    trace->endCall ();
}

// mise à jour des attributs
trace->putToSleep ();
senseurs (fichier, senseur1, senseur2, senseur3);
trace->wakeUp ();
```

conseils d'utilisation spécifiques

implantation

Les attributs privés sont décrits sommairement dans la table 15, il n'y a pas d'attribut protégé.

TAB. 15: attributs privés de la classe CallTrace

nom	type	description
instance_	static CallTrace *	pointeur vers l'unique instance de la classe (modèle de conception du singleton)
logFile_	ofstream	fichier d'enregistrement
registeredObjects_	map<const void *, const void *>	table des pointeurs d'objets enregistrés
inCall_	bool	indicateur d'appel en cours d'enregistrement
awaken_	bool	indicateur d'enregistrement possible

Les méthodes privées sont décrites dans la table 16. Il s'agit des constructeurs et de l'opérateur d'affectation, qui sont privés afin d'interdire la création de plusieurs instances de la classe (modèle de conception du singleton).

TAB. 16: CallTrace : méthodes privées

signature	description
CallTrace ()	constructeur simple
CallTrace (const CallTrace& c)	constructeur par copie
CallTrace& operator = (const CallTrace& c)	opérateur d'affectation
CallTrace ()	destructeur de la classe.

12.5 classe ClubAllocHandler

description

Cette classe permet d'enregistrer une fonction de réaction aux dépassements de capacité mémoire lors des **new**, fonction générant une exception de type ClubErreurs ayant pour code la valeur : ClubErreurs::allocation_memoi. Cette classe n'est utile que pour les effets de bord de son couple constructeur/destructeur, il suffit donc de définir au début du programme principal (ou du bloc que l'on veut protéger) une variable de ce type pour que les dépassements de capacité génèrent ce message. La protection s'arrête à la fin de la durée de vie de la variable définie.

interface publique

```
#include "club/ClubErreurs.h"
```

Le constructeur ne servant qu'à mémoriser la valeur précédente de la routine de gestion des erreurs d'allocation retournée par `set_new_handler`, il ne prend aucun argument.

L'appel au destructeur est quant à lui inséré directement par le compilateur à tous les endroits où le flot de contrôle quitte la portée de la variable locale déclarée, l'utilisateur n'a pas lieu de s'en préoccuper.

exemple d'utilisation

```
#include "club/ClubErreurs.h"
...

inr main (int argc, char **argv)
{ ClubAllocHandler alloc; // utilisé pour ses seuls constructeurs/destructeur
  ...
}
```

conseils d'utilisation spécifiques

L'utilisation de cette classe se limite à la définition d'une instance, dont la portée règle la zone à protéger. Le plus simple est de protéger la totalité d'un programme en mettant cette définition au début du programme principal, ce qui permet de protéger également les allocations ayant lieu dans les constructeurs des autres variables du programme principal.

implantation

Le seul attribut de la classe est `ancien_`, qui contient l'ancienne routine de gestion des erreurs qu'il faudra remettre en place en quittant la zone protégée, il est de type `void (*) ()`.

La méthode de classe `handler ()` (de type `void (*) ()`) est la routine de gestion mise en place pendant la durée de vie de la variable, elle génère une exception (ou un objet local) de type `ClubErreurs` et de code `ClubErreurs::allocation_memoire`.

Cette méthode est déclarée `static` de façon à être de type `void()` et non `UtilAllocHandler::void ()`, afin de l'utiliser en argument de la fonction `set_new_handler`.

12.6 classe ClubErreurs

description

Cette classe permet de formater et traduire dans la langue de l'utilisateur des messages d'erreur liés à la bibliothèque CLUB ou les messages simples d'un utilisateur. Elle utilise les mécanismes qui lui sont fournis par sa classe de base.

Certains codes gérés par cette classe peuvent être réutilisés par des fonctions externes à la bibliothèque CLUB. Un code en particulier est pratique à utiliser en mise au point : il s'agit du code `ClubErreurs::trace`. En effet, il suffit d'écrire la ligne :

```
{ ClubErreurs (ClubErreurs::trace, __LINE__, __FILE__); }
```


(sans oublier les accolades) n'importe où dans un fichier source pour qu'un message signale sur la sortie standard chaque exécution de la ligne correspondante (et donc des lignes voisines), en effet cette ligne crée une variable temporaire qui formate le message et est détruite dès que l'accolade de fin est rencontrée, ce qui provoque l'affichage. Afin de faciliter cette utilisation en mise au point, la macro-définition `CLUB_TRACE` a été écrite dans `club/ClubErreurs.h`, elle prend pour valeur la ligne décrite précédemment lorsque `CLUB_CPP_DEBUG` est définie, et prend une valeur vide dans le cas contraire. Il suffit donc de mettre `CLUB_TRACE` dans son code pour avoir une trace conditionnelle.

interface publique

```
#include "club/ClubErreurs.h"
```

Les opérations publiques sont essentiellements celles de la classe de base **BaseErreurs** (voir table 5). Les méthodes qui ne peuvent être héritées (les constructeurs et les méthodes de classe) ont été redéfinies avec des sémantiques équivalentes. Dans ces méthodes redéfinies, les codes d'erreurs (déclarés comme type énuméré publique interne) attendent les arguments suivants dans la liste des arguments variables :

```
allocation_memoire : néant ;
message_simple : char * (pour le message à formater) ;
etat_fichier : char * (pour le nom du fichier) ;
ouverture_fichier : char * (pour le nom du fichier) ;
lecture_types_incompatibles : char * (pour le nom de l'option), char * (pour le type de l'option),
    char * (pour le type désiré) ;
taille_tableau_arguments : char * (pour le nom de l'option), int (pour la taille du tableau) ;
nombre_arguments_incompatibles : char * (pour le nom de l'option), int (pour le nombre d'argu-
    ments trouvés), int (pour le nombre d'arguments attendus) ;
nombre_arguments_negatif : char * (pour le nom de l'option), int (pour le nombre d'arguments) ;
nb_arguments_option_vide : néant ;
nom_option_non_lettre : char * (pour le nom de l'option) ;
nombre_arguments_nul : char * (pour le nom de l'option) ;
valeur_par_defaut_manquante : char * (pour le nom de l'option) ;
option_chaine_trop_longue : char * (pour le nom de l'option), char * (pour l'argument trouvé), int
    (pour la longueur maximale autorisée) ;
option_chaine_hors_domaine : char * (pour le nom de l'option), char * (pour l'argument trouvé),
    string [] (pour la liste des valeurs autorisées) ;
option_entiere_hors_limites : char * (pour le nom de l'option), int (pour la valeur minimale autori-
    sée), int (pour la valeur maximale autorisée), int (pour la valeur trouvée) ;
option_reelle_hors_limites : char * (pour le nom de l'option), double (pour la valeur minimale auto-
    risée), double (pour la valeur maximale autorisée), double (pour la valeur trouvée) ;
code_option_non_reconnu : char * (pour le code de l'option) ;
plus_d_argument : char * (pour le nom de l'option) ;
option_deja_definie : char * (pour le nom de l'option) ;
options_appli_deja_initialise : char * (pour le nom de l'analyseur) ;
options_appli_non_initialise : char * (pour le nom de l'analyseur) ;
```

arguments_non_reconnus : **char** * (pour le nom de l'analyseur);

occurrence_inexistante : **int** (pour le numéro d'occurrence), **char** * (pour le nom de l'option), **int** (pour le nombre d'occurrences passées);

occurrence_hors_limites : **char** * (pour le nom de l'option), **int** (pour le nombre minimal d'occurrences), **int** (pour le nombre maximal d'occurrences), **int** (pour le nombre d'occurrences passées);

tampon_vide : **char** * (pour le nom du tampon);

manque_guillemet : **char** * (pour le nom du fichier);

ligne_hors_domaine : **int** (pour le numéro de ligne), **int** (pour le numéro minimal), **int** (pour le numéro maximal), **char** * (pour le nom du fichier);

champ_ligne_hors_domaine : **int** (pour le numéro de champ), **int** (pour le numéro minimal), **int** (pour le numéro maximal), **int** (pour le numéro de ligne), **char** * (pour le nom du fichier);

champ_hors_domaine : **int** (pour le numéro de champ), **int** (pour le numéro minimal), **int** (pour le numéro maximal), **char** * (pour le nom du fichier);

accolades_non_equilibrees : **char** * (pour le nom du fichier);

bloc_introuvable : **char** * (pour le nom du bloc), **char** * (pour le nom du bloc englobant, qui peut être une chaîne vide), **char** * (pour le nom du fichier);

bloc_champ_inexistant : **int** (pour le numéro de champ), **char** * (pour le nom du bloc), **char** * (pour le nom du fichier);

bloc_nom_terminal : **char** * (pour le nom du bloc), **char** * (pour le nom du fichier);

manque_accolade : **char** * (pour le nom du bloc), **char** * (pour le nom du fichier);

manque_chevron : **char** * (pour le nom du bloc), **char** * (pour le nom du fichier);

nom_inclusion : **char** * (pour le nom du fichier);

msg_errno : **char** * (pour le préfixe au message système);

format_fortran : **char** * (pour le format);

trace : **int** (pour le numéro de ligne), **char** * (pour le nom de fichier).

iterateur_invalide : **char** * (pour le nom du fichier).

caractere_hors_domaine : **int** (pour le caractère), **int** (pour le code minimal), **int** (pour le code maximal), **char** * (pour le nom du fichier).

variable_environnement : **char** * (pour le nom de la variable d'environnement).

calltrace_not_activated : néant

internal_error : **int** (pour le numéro de ligne), **char** * (pour le nom du fichier).

not_implemented : **char** * (pour le nom et la signature de la fonction non implémentée).

name_error : **char** * (pour la désignation incorrecte).

string_to_int_conversion_error : **char** * (pour la chaîne de caractères non convertible en entier).

string_to_double_conversion_error : **char** * (pour la chaîne de caractères non convertible en réel).

file_error : **char** * (pour le nom de fichier), **char** * (pour le message d'erreur associé au fichier).

malformed_data : **char** * (pour le nom de l'élément ou de la désignation mal formé).

missing_tag : **char** * (pour le nom de la balise manquante).

missing_attribute : **char** * et **char** * (pour le nom de l'attribut et le nom de l'élément).

xpointer_syntax_error : **char** * (pour chaîne xpointer invalide).

redefined_unit : **char** * (pour le symbole de l'unité).

incompatible_units : char * et char * (pour les symboles des unités).

entity_syntax_error : char * (pour la chaîne contenant l'appel à l'entité ou au caractère).

unknown_parameter_entity : char * et char * (pour l'entité et pour la chaîne la contenant).

unit_syntax_error : char * (pour la chaîne définissant l'unité composée).

undefined_data : char * (pour le nom absolu de la donnée indéfinie).

undefined_table_or_structure : char * (pour le nom absolu du tableau ou de la structure indéfinie).

undefined_parameter : char * (pour le nom absolu du paramètre indéfini).

undefined_index : int (pour l'index indéfini), char * (pour le nom absolu du tableau).

undefined_unit : char * (pour le nom de l'unité indéfinie).

undefined_node_type : char * (pour le nom du noeud indéfini).

current_data_not_a_table : char * (pour le nom absolu de la donnée courante).

current_data_is_a_table : char * (pour le nom absolu de la donnée courante).

move_up_forbidden : néant.

madona_error : char * (pour le message d'erreur madona).

xml_error : char * (pour le message d'erreur XML).

invalid_element_type : char * (pour le nom de la table), char * (pour le type de la table), char * (pour le type d'élément que l'on souhaite ajouter).

no_madona_lib : néant.

no_xerces_lib : néant.

index_creation_forbidden : int (pour l'index).

data_already_defined : char * (pour le nom absolu de la donnée).

unknown_file_type : néant.

exemples d'utilisation

utilisation sans exception

```
#include <fstream.h>
#include "club/ClubErreurs.h"
...
int lire (const char* nom, ClubErreurs& err)
{ ifstream fichier (nom);
  if (! fichier.good ())
  { err.miseAJour (ClubErreurs::ouverture_fichier, nom);
    return 1;
  }
  ...
  return 0;
}
```

utilisation avec exceptions

```
#include <fstream.h>
#include "club/ClubErreurs.h"
...
void lire (const char* nom)
{ ifstream fichier (nom);
  if (! fichier.good ())
    ClubErreurs::lance (ClubErreurs::ouverture_fichier, nom);
  ...
  return ;
}
```

conseils d'utilisation spécifiques

Cette classe est principalement utilisée pour tester la bonne exécution des fonctions de la bibliothèque CLUB elle-même. Son utilisation se résume donc à tester correctement la présence ou l'absence d'erreurs (méthode `existe ()`), et à décider du comportement à adopter en présence d'une erreur.

Si la même instance d'erreur est utilisée pour tester le retour de plusieurs fonctions, il faut prendre garde de la tester au bon moment ; il est en effet possible qu'une erreur soit générée par le premier appel, qu'elle soit ignorée par l'appelant, qu'une seconde fonction de CLUB se termine ensuite normalement et que l'appelant ne détecte la première erreur qu'à cet instant.

implantation

La classe dérive publiquement de `BaseErreurs`, elle ne possède aucun attribut propre.

12.7 classe DataFile

description

`DataFile` permet de gérer un pointeur partagé sur une instance de `UniqDataFile`. Elle offre alors dans son interface publique l'équivalent des méthodes de `UniqDataFile`. Un membre statique protégé permet de paramétrer le type des fichiers manipulés (fichiers structurés, Madona ou XML via le type énuméré `FileType` déclaré dans le fichier header). Un autre permet de connaître le type de fichier manipulé par l'instance courante.

L'attribut statique est initialisé en testant la présence des bibliothèques `XERCES` puis `MADONA`. Si aucune bibliothèque n'est installée, il est alors initialisé au type de fichier structuré.

L'utilisateur dispose de 3 constructeurs. Celui par défaut utilise le membre statique pour connaître le type de fichier qui va être manipulé. Le second prend en paramètre le type de fichier correspondant à l'instance à créer. Le troisième utilise le nom du fichier qui lui est passé en paramètre et détermine le type du fichier.

Cette classe est sous une forme canonique et offre donc également un constructeur par copie et l'opérateur d'affectation. Ainsi, plusieurs instances de `DataFile` peuvent être construites alors qu'elles gèrent toutes la même instance de `UniqDataFile`.

La classe `DataFile` contient différents types de services :

- Les services d'écriture et de lecture de fichier :
 - readFile** charge en mémoire un fichier de données,
 - writeFile** écrit les données dans un fichier.
- Les services de déplacement relatif et absolu à l'intérieur de la structure de données. Les déplacements peuvent seulement être réalisés sur des données non-terminales (c'est-à-dire seulement sur des structures ou des tableaux). Seul le service **moveTo()** est indépendant de la donnée courante.
 - moveDown** déplace le pointeur de la donnée courante sur une donnée «filie» de cette dernière,
 - moveUp** déplace le pointeur de la donnée courante sur la donnée «mère» de cette dernière,
 - moveTo** déplace le pointeur de la donnée courante sur une autre donnée de la structure, indépendamment du pointeur de la donnée courante.
- Les services de création de données qui travaillent sur la donnée courante :
 - createIntData** crée un paramètre entier,
 - createRealData** crée un paramètre réel,
 - createStringData** crée un paramètre de type chaîne de caractères,
 - createTable** crée un tableau vide,
 - createStructure** crée une structure vide.
 - createReference** crée une référence.
- Les services de mise à jour de paramètre qui travaillent sur la donnée courante :
 - setIntData** met à jour un paramètre entier,
 - setRealData** met à jour un paramètre réel,
 - setStringData** met à jour un paramètre de type chaîne de caractères.
- Les services de récupération et de mise à jour d'information sur une donnée qui travaillent sur la donnée courante :
 - size** renvoie la taille d'un tableau,
 - getDataType** renvoie le type de donnée,
 - getComment** renvoie les commentaires associés à une donnée.
 - setComment** associe un commentaire à une donnée.
- Les services de récupération de paramètre qui travaillent sur la donnée courante :
 - getIntData** renvoie la valeur d'un paramètre entier,
 - getRealData** renvoie la valeur d'un paramètre réel,
 - getStringData** renvoie la valeur d'un paramètre de type chaîne de caractères,
- Les services de suppression de donnée qui travaillent sur la donnée courante :
 - deleteData** supprime une donnée.
- Autres services :
 - writeMemoryStructure** affiche sur la sortie standard la structure de données.

L'utilisation de l'un de ses services fait appel au service de même nom de la classe `UniqDataFile`.

interface publique

```
#include "club/DataFile.h"
```

Il n'y a pas d'attributs publics pour la classe DataFile. La majorité des méthodes publiques de DataFile font simplement appel à leur homologue dans UniqDataFile appliqué sur l'instance partagée. Ces méthodes sont décrites dans la section 12.24.

Le tableau suivant présente les autres méthodes publiques de l'interface.

TAB. 17: DataFile : méthodes publiques

signature	description
DataFile ()	construit un DataFile correspondant au type de fichier désigné par le membre statique protégé <code>currentFileType_</code> Exceptions : <code>no_xerces_lib</code> , <code>no_madona_lib</code> , <code>unknown_file_type</code>
DataFile (FileType <i>type</i>)	construit un DataFile de type <i>type</i> Exceptions : <code>no_xerces_lib</code> , <code>no_madona_lib</code> , <code>unknown_file_type</code>
DataFile (const string& <i>fileName</i>)	construit un DataFile en déterminant le type du fichier désigné par <i>fileName</i> Exceptions : <code>no_xerces_lib</code> , <code>no_madona_lib</code> , <code>unknown_file_type</code> , <code>ouverture_fichier</code>
DataFile (const DataFile& <i>other</i>)	constructeur de copie
DataFile& operator = (const DataFile& <i>other</i>)	opérateur d'affectation
~DataFile ()	détruit l'instance
UniqDataFile* getUniqDataFile ()	retourne l'instance partagée par les DataFiles
FileType getCurrentFileType ()	retourne la valeur du membre statique protégé (<code>currentFileType_</code>) désignant le type de fichier manipulé
void setCurrentFileType (FileType <i>type</i>)	positionne la valeur du membre statique protégé (<code>currentFileType_</code>) désignant le type de fichier manipulé
FileType getFileType ()	retourne le type de fichier manipulé par l'instance

exemple d'utilisation

```
#include "club/DataFile.h"
...
```

implantation

Les attributs protégés sont décrits sommairement dans la table 18, il n'y a pas d'attribut privé.

TAB. 18: attributs protégés de la classe DataFile

nom	type	description
sharedUniqDataFile_	TamponPartage	tampon partagé contenant un pointeur sur une instance de UniqDataFile
currentFileType_	FileType	membre statique désignant le type de fichier manipulé (fichier structuré, Madona, XML)
fileType_	FileType	membre désignant le type de fichier manipulé (fichier structuré, Madona, XML) par l'instance

12.8 classe FichierStructure

description

Cette classe gère les accès aux informations contenues dans un fichier texte (ou plusieurs) structuré sous forme de blocs imbriqués à l'aide de caractères '{' et '}', chaque bloc pouvant être récupéré par son nom (qui peut être une chaîne vide). De plus, si un bloc de nom **xxx** est décrit dans le fichier par :

```
xxx { => { yyy.zzz } }
```

cela signifie que le contenu du bloc **xxx** est égal au contenu du sous-bloc **zzz**, dont le bloc père est **yyy**. Ce système d'indirection évite la duplication de blocs complexes équivalents, il implante une notion d'héritage de données. On peut également lire => comme : *voir aussi*.

Cette classe est principalement utilisée par la bibliothèque MARMOTTES pour lire les fichiers de description de senseurs, on trouvera dans [DR2] des informations complémentaires sur ce que l'on peut faire.

interface publique

```
#include "club/FichierStructure.h"
```

TAB. 19: FichierStructure : méthodes publiques

signature	description
FichierStructure ()	construit une instance vide
FichierStructure (FILE *ptrFic)	construit une instance contenant l'ensemble du fichier <i>ptrFic</i> Exceptions : accolades_non_equilibrees, etat_fichier, manque_guillemet, manque_chevron, iterateur_invalide, caractere_hors_domaine, nom_inclusion.
FichierStructure (const char *nomFichier)	construit une instance contenant l'ensemble du fichier <i>nomFichier</i> Exceptions : accolades_non_equilibrees, etat_fichier, manque_guillemet, manque_chevron, iterateur_invalide, caractere_hors_domaine, nom_inclusion.
FichierStructure (const string& nomFichier)	construit une instance contenant l'ensemble du fichier <i>nomFichier</i> Exceptions : accolades_non_equilibrees, etat_fichier, manque_guillemet, manque_chevron, iterateur_invalide, caractere_hors_domaine, nom_inclusion.
à suivre ...	

TAB. 19: FichierStructure : méthodes publiques (suite)

signature	description
FichierStructure (TamponTexte *fichier)	construit une instance contenant l'ensemble du fichier <i>fichier</i> Exceptions : accolades_non_equilibrees, etat_fichier, manque_guillemet, manque_chevron, itereur_invalide, caractere_hors_domaine, nom_inclusion.
FichierStructure (const char *clef, const FichierStructure *pere)	construit une instance en extrayant le sous bloc <i>clef</i> d'une instance <i>pere</i> Exceptions : bloc_introuvable, manque_accolade.
FichierStructure (const string& clef, const FichierStructure *pere)	construit une instance en extrayant le sous bloc <i>clef</i> d'une instance <i>pere</i> Exceptions : bloc_introuvable, manque_accolade.
FichierStructure (const FichierStructure& f)	constructeur par copie
FichierStructure& operator = (const FichierStructure& f)	affectation
FichierStructure ()	destructeur
void lit (FILE *ptrFic)	réinitialise l'instance en lisant la totalité du fichier <i>ptrFic</i> Exceptions : accolades_non_equilibrees, etat_fichier, manque_guillemet, manque_chevron, itereur_invalide, caractere_hors_domaine, nom_inclusion.
void lit (const char *nomFichier)	réinitialise l'instance en lisant la totalité du fichier <i>nomFichier</i> Exceptions : accolades_non_equilibrees, etat_fichier, manque_guillemet, manque_chevron, itereur_invalide, caractere_hors_domaine, nom_inclusion.
int lit (const string& nomFichier)	réinitialise l'instance en lisant la totalité du fichier <i>nomFichier</i>
void lit (TamponTexte *ptrTampon)	réinitialise l'instance en lisant la totalité du fichier <i>ptrTampon</i> Exceptions : accolades_non_equilibrees, etat_fichier, manque_guillemet, manque_chevron, itereur_invalide, caractere_hors_domaine, nom_inclusion.
bool terminal () const	indique si l'instance est un bloc terminal (c'est à dire s'il n'a pas de sous-bloc)
int nombreChamps () const	retourne le nombre de champs de l'instance courante (0 s'il ne s'agit pas d'un bloc terminal)
void champ (int numero, char *tampon, const int max) const	retourne le champ <i>numero</i> (la numérotation commence à 1) dans le <i>tampon</i> pouvant contenir <i>max-1</i> caractères Exceptions : bloc_champ_inexistant, bloc_non_terminal.
bool contientSousBloc (const char *clef) const	retourne une valeur vraie si l'instance contient le sous-bloc <i>clef</i> Exceptions : manque_accolade.
void listeSousBlocs (vector<string> *ptrTable) const	ajoute dans la table pointée par <i>ptrTable</i> la liste des noms de sous-blocs contenus dans l'instance Exceptions : manque_accolade.
const string& nomBloc () const	retourne le nom du bloc (y compris le nom de ses blocs englobants)
const string& nomFichier () const	retourne le nom du fichier dont est issu l'instance
à suivre ...	

TAB. 19: FichierStructure : méthodes publiques (suite)

signature	description
const string& date () const	retourne la date du plus récent des fichiers lus dans l'instance (format ISO-8601)
unsigned int signature () const	retourne une signature résumant les données

exemple d'utilisation

```
#include "club/FichierStructure.h"
...
// ouverture du fichier complet
FichierStructure fichier ("essai");

// extraction du bloc racine de l'arbre
FichierStructure blocArbre ("arbre", &fichier);

// extraction des sous-blocs recursifs
ArbreBinaire arbre = LitArbre (blocArbre);
...

// fonction de lecture d'un arbre
// depuis un fichier structuré de facon récursive
ArbreBinaire LitArbre (bloc)
{ if (bloc.terminal ())
  { // l'arbre n'a plus de branche
    if (bloc.nombreChamps () != 1)
    { // la feuille n'a pas un champ unique!
      ...
    }

    // extraction de la valeur de la feuille
    const int maxTampon = 80;
    char tampon [maxTampon];
    (void) f.champ (1, tampon, maxTampon);

    // retour de l'arbre compose d'une feuille unique
    return ArbreBinaire (tampon);
  }
  else
  { // le bloc contient des branches

    if (! bloc.contientSousBloc ("gauche"))
    { // le bloc n'a pas de branche gauche!
      ...
    }
    ArbreBinaire gauche = LitArbre (FichierStructure ("gauche", bloc));
    if (! bloc.contientSousBloc ("droit"))
```

```

{ // le bloc n'a pas de branche droite !
    ...
}
ArbreBinaire droit = LitArbre (FichierStructure ("droit", bloc));

// retour de l'arbre formé par les deux branches
return ArbreBinaire (gauche, droit);
}
}

```

conseils d'utilisation spécifiques

La démarche d'utilisation classique consiste à créer un bloc général contenant la totalité d'un fichier (ou de plusieurs si des directives d'inclusion sont présentes), puis à extraire les blocs couche par couche.

Le texte est alloué dynamiquement à la lecture du fichier, et tous les blocs construits à partir d'un fichier unique se partagent cette mémoire. Cette zone est gérée à travers une instance de TamponPartage (voir table 20 et section 12.21), aussi aucune précaution n'est elle nécessaire sur la portée des diverses instances. Il est tout à fait possible de détruire un bloc père avant ses blocs inclus, c'est la dernière instance détruite qui libérera la mémoire allouée.

implantation

Les attributs sont décrits sommairement dans la table 20, les méthodes privées dans la table 21

TAB. 20: attributs de la classe FichierStructure

nom	type	description
nomBloc_	string	nom du bloc
nomFichier_	string	nom du fichier dont provient le bloc
total_	TamponPartage	pointeur sur le texte complet du fichier
debut_	char *	pointeur sur le début du bloc (à l'intérieur de la zone alouée dans la TamponPartage)
fin_	char *	pointeur sur la fin du bloc (à l'intérieur de la zone alouée dans la TamponPartage)
date_	string	date du plus récent des fichiers lus (format iso-8601)
signature_	unsigned int	signature des données lues

TAB. 21: FichierStructure : méthodes privées

signature	description
void rechercheSousBloc (const string& <i>clef</i> , const char ** <i>addrDebut</i> , const char ** <i>addrFin</i> , string * <i>addrNomBloc</i>) const	recherche le sous-bloc <i>clef</i> à l'intérieur des limites <i>addrDebut</i> et <i>addrFin</i> Exceptions : bloc_introuvable, manque_accolade.

12.9 classe FormatC

description

La classe FormatC permet de gérer des formats d'écriture du langage C. Elle est utilisée de façon interne par la classe Traducteur (voir section 12.23), pour tester la compatibilité d'un format traduit par rapport à un format de référence et pour appliquer un format traduit.

interface publique

```
#include "club/FormatC.h"
```

La classe définit un type énuméré anonyme public pour représenter les différents arguments reconnus par le format.

```
enum { ENTIER, REEL, CHAINE, CARACTERE, INCONNU, ERREUR };
```

TAB. 22: FormatC : méthodes publiques

signature	description
FormatC ()	constructeur par défaut
FormatC (const string& <i>format</i>)	construit une instance à partir du <i>format</i> et déclenche son analyse
FormatC (const FormatC& <i>f</i>)	constructeur par copie
FormatC& operator = (const FormatC& <i>f</i>)	affectation
~FormatC ()	destructeur
void analyse (const string& <i>format</i>)	analyse le format pour déterminer le nombre et le type des arguments qu'il attend
va_list applique (va_list <i>ap</i> , string* <i>ptrChaine</i>) const	applique le format déjà analysé à la liste des arguments variables passées par l'intermédiaire de <i>ap</i> .
int compatible (const FormatC& <i>f</i>) const	indique si le format de l'instance est compatible avec <i>f</i> , c'est à dire s'il admet la même liste d'arguments
const string& chaîne () const	retourne la chaîne spécifiant le format
int nbArgs () const	retourne le nombre d'arguments attendus par le format
int typeArg (int <i>i</i>) const	retourne le type de l'argument <i>i</i> (c'est une valeur prise dans l'énuméré public anonyme de la classe)

exemple d'utilisation

```
#include "club/FormatC.h"
...
const char* TraduitFormatCVersExterne (const char* format)
{
    InitTraducteurs ();

    // traduction directe
    FormatC origine (format);
```

```

FormatC traduit (ptrExterne->operator () (format));

// vérification de la compatibilité des formats
if (! origine.compatible (traduit))
{
    string res ("FORMAT TRADUIT INCOMPATIBLE");
    return res.c_str ();
}

// retour du format traduit sous forme compatible
return traduit.chaine ();
}

```

implantation

Les attributs privés sont décrits sommairement dans la table 23, il n'y a pas d'attribut protégé.

TAB. 23: attributs privés de la classe FormatC

nom	type	description
tailleTableParts_	int	taille des tables des parts de formats
nombreParts_	int	nombre de parts de formats dans les tables
type_	int *	table des types de parts
chaine_	string	format c de base

Les méthodes privées sont décrites dans la table 24.

TAB. 24: FormatC : méthodes privées

signature	description
void ajoutePart (int <i>type</i>)	ajoute un élément (fixe ou variable) aux tables d'analyse de parts de format
const char* specificateurSuivant (const char* <i>mobile</i>) const	recherche le premier spécificateur du format c dans les caractères suivant <i>mobile</i>

12.10 classe FormatFortran

description

La classe FormatFortran permet de gérer des formats d'écriture du langage FORTRAN. Elle est utilisée de façon interne par la classe Traducteur (voir section 12.23), pour tester la compatibilité d'un format traduit par rapport à un format de référence et pour appliquer un format traduit.

interface publique

```
#include "club/FormatFortran.h"
```

La classe définit un type énuméré anonyme public pour représenter les différents arguments reconnus par le format.

```
enum { ENTIER, REEL, CHAINE_ARG, BOOLEEN, CHAINE_FIXE, ERREUR };
```

TAB. 25: FormatFortran : méthodes publiques

signature	description
FormatFortran ()	constructeur par défaut
FormatFortran (const string& <i>format</i>)	construit une instance à partir du <i>format</i> et déclenche son analyse, lève une exception <code>format_fortran</code> s'il y a des erreurs d'analyse Exceptions : <code>format_fortran</code> .
FormatFortran (const FormatFortran& <i>f</i>)	constructeur par copie
FormatFortran& operator = (const FormatFortran& <i>f</i>)	affectation
~FormatFortran ()	destructeur, libère la mémoire allouée
void analyse (const string& <i>format</i>)	analyse le format pour contrôler sa validité et déterminer le nombre et le type des arguments qu'il attend Exceptions : <code>format_fortran</code> .
va_list applique (va_list <i>ap</i> , int <i>nbLong</i> , long int <i>tabLong</i> [], string * <i>ptrChaine</i>) const	applique le format déjà analysé à la liste des arguments variables passées par l'intermédiaire de <i>ap</i> . Les <i>nbLong</i> longueurs de chaînes de caractères correspondant aux chaînes précédant <i>ap</i> sont retournées dans le tableau <i>tabLong</i> . Le pointeur d'argument variable <i>ap</i> est retourné après analyse.
int compatible (const FormatFortran& <i>f</i>)	indique si le format de l'instance est compatible avec <i>f</i> , c'est à dire s'il admet la même liste d'arguments
const string& chaîne () const	retourne la chaîne spécifiant le format
int nbArgs () const	retourne le nombre d'arguments attendus par le format
int typeArg (int <i>i</i>) const	retourne le type de l'argument <i>i</i> (c'est une valeur prise dans l'énuméré public anonyme de la classe)

exemple d'utilisation

```
#include "club/FormatFortran.h"
...

void tradecrch_ (char* chaine, int& lUtile, const char* format ...)
{
    // écriture dans une chaîne à partir d'un format fortran
    InitTraducteurs ();

    // traduction
    FormatFortran traduit;
    if (ptrExterne->traductionFormatF (format, &traduit) > 0)
```

```
{
    // il y a un problème d'analyse
    ...
    return;
}

// la fonction a deux arguments chaînes de caractères avant
// la partie variable : chaine et format, leur longueur va donc
// apparaitre entre les derniers arguments variables et les
// premières longueurs

int nbLong = 2;
long int tabLg [2];
string sortie;

// écriture
va_list ap;
va_start(ap, format);
ap = traduit.applique (ap, nbLong, tabLg, &sortie);
va_end(ap);

// copie des résultats dans les variables fortran

lUtile = (sortie.size () < (long unsigned int) tabLg [0])
        ? sortie.size () : tabLg [0];
(void) strncpy (chaine, sortie.c_str (), lUtile);
if (tabLg [0] > lUtile)
    (void) memset (chaine + lUtile, ' ', (unsigned int) (tabLg [0] - lUtile));
}
```

conseils d'utilisation spécifiques

L'utilisation de la classe à partir de programmes en langage FORTRAN passe obligatoirement par des fonctions d'interface. La convention de passage des arguments reconnue est que les types de base (**integer**, **double precision**, ...) sont passés par pointeur et ont un type équivalent en langage C++. Les chaînes de caractères sont passées sous forme d'un pointeur dans la liste normale des arguments, la longueur de la chaîne étant quant à elle rajoutée à la fin de la liste des arguments, sous la forme d'un entier long passé par copie.

Ainsi une fonction appelée depuis le FORTRAN avec en arguments un entier, une chaîne de 10 caractères, un double précision, et une chaîne de 3 caractères sera vue côté C++ comme ayant un pointeur d'entier (**int ***), un pointeur de chaîne de caractères (**char ***), un pointeur de réel (**double ***), un entier long (101), et un autre entier long (31).

Cette convention est assez classique, elle est suivie par les compilateurs fortran de Sun disponibles sur stations Sparc mais également par le compilateur **g77** disponible entre autres sur stations Sparc et sur architectures intel/linux.

L'exemple 12.10 est issu du code de la bibliothèque, il illustre les précautions à prendre pour réaliser correctement le passage de paramètres. La fonction dont il est issu fonctionne sur Sun et sur architecture intel.

implantation

Les attributs sont décrits sommairement dans la table 26, les méthodes privées dans la table 27

TAB. 26: attributs de la classe FormatFortran

nom	type	description
tailleTableParts_	int	taille des tables des parts de formats
nombreParts_	int	nombre de parts de formats dans les tables
type_	int *	table des types de parts
formatC_	string *	tables des formats du langage C permettant d'afficher les parts de formats FORTRAN
chaine_	string	format FORTRAN de base

TAB. 27: FormatFortran : méthodes privées

signature	description
void ajoutePart (int <i>type</i> , const string& <i>format</i>)	ajoute un élément (fixe ou variable) aux tables d'analyse de parts de format
int modificateurs (AnalyseurLexical * <i>al</i> , int * <i>ptrTaille</i> , int * <i>ptrPrecision</i>)	analyse de modificateurs de termes dans un spécificateur de format (par exemple le 6.3 du spécificateur F6.3)
int terme (AnalyseurLexical * <i>al</i>)	analyse un terme de spécificateur de format (par exemple le F6.3 du spécificateur 12F6.3)
int specificateur (AnalyseurLexical * <i>al</i>)	analyse un spécificateur de format (par exemple le 12F6.3 de la liste (12F6.3, 5(3X, I2)))
int liste (AnalyseurLexical * <i>al</i>)	analyse une liste de spécificateurs (le format (12F6.3, 5(3X, I2)) est ainsi une liste de deux éléments, dont le deuxième est également une liste). Un format fortran complet est une liste (éventuellement réduite à un seul spécificateur, mais toujours entre parenthèses)

12.11 classe IterateurCaractere

description

La classe IterateurCaractere permet de parcourir un TamponAscii caractère par caractère. Cet itérateur possède les propriétés suivantes :

- À un instant donné, l'itérateur possède une valeur qui désigne un élément du tampon. On accède à cette valeur par `valeur()` ;
- L'itérateur peut être incrémenté par l'opérateur `++` de manière à pointer sur l'élément suivant ;
- L'itérateur peut également être décrémenté par l'opérateur `--` de manière à pointer sur l'élément précédent (itérateur bidirectionnel) ;
- L'itérateur permet d'accéder directement à un élément du tampon en utilisant la méthode `allerSur(int index)`.

interface publique

```
#include "club/IterateurCaractere.h"
```

TAB. 28: IterateurCaractere : méthodes publiques

signature	description
IterateurCaractere (const TamponAscii& <i>tampon</i>)	construit une instance parcourant <i>tampon</i> Exceptions : -
IterateurCaractere (TamponAscii* <i>tampon</i>)	construit une instance parcourant <i>tampon</i> Exceptions : -
IterateurCaractere ()	destructeur de la classe.
void premier ()	positionne l'itérateur sur le premier caractère du tampon Exceptions : -
void dernier ()	positionne l'itérateur sur le dernier caractère du tampon Exceptions : -
void allerSur (int <i>index</i>)	positionne l'itérateur sur le caractère numéro <i>index</i> du tampon. Les caractères sont numérotés à partir de 1. Exceptions : caractere_hors_domaine.
int termine () const	indique que l'itérateur pointe sur la fin du tampon.
int nombreIterations () const	retourne le nombre de caractères du tampon.
int operator++ (int)	positionne l'itérateur sur le caractère suivant du tampon Exceptions : itérateur_invalide.
int operator-- (int)	positionne l'itérateur sur le caractère précédent du tampon Exceptions : itérateur_invalide
const char* valeur () const	retourne un pointeur sur le caractère courant. Exceptions : itérateur_invalide
int longueurElement (int <i>avecFin</i> = 0) const	retourne la longueur de l'élément pointé par l'itérateur, ici cette longueur est toujours égale à 1 (1 caractère). Exceptions : -
void synchronize (int <i>direction</i> = 0)	Si <i>direction</i> = 1, synchronise la position de l'itérateur sur la position réelle dans le tampon (itérateur altéré lors d'une modification du tampon). Si <i>direction</i> = 0 (par défaut), synchronise l'état du tampon (position courante du curseur) sur la position courante de l'itérateur (mémorisation avant altération du tampon) Exceptions : itérateur_invalide

exemple d'utilisation

```
#include "club/IterateurCaractere.h"
#include "club/TamponAscii.h"
...

try
{
    TamponAscii t(nom_fichier);
    IterateurCaractere itc (t);
```



```
//parcours decremental
itc.dernier();
do
{
    cout << itc.valeur();
}
while (itc--);
}
catch (ClubErreurs ce)
{
    cout << "Erreur : "<<ce.why()<<endl;
    ce.correction();
    exit(1);
}
```

conseils d'utilisation spécifiques

Les opérateurs ++ et -- retournent 0 lorsqu'ils sont arrivés à l'extrémité du tampon. Il est ainsi possible de tester leur code retour dans un while (cf exemple d'utilisation précédent).

Les classes IterateurX développées pour la bibliothèque CLUB sont dédiées à la gestion d'objets TamponAscii. En conséquence, les interfaces de ces classes n'offrent que les méthodes utiles à cette gestion. Si les besoins de parcours d'un tampon sont des besoins classiques, la classe TamponTexte offre les fonctionnalités nécessaires, sans nécessiter la manipulation d'itérateurs.

implantation

Les attributs protégés sont décrits sommairement dans la table 29, il n'y a pas d'attribut privé.

TAB. 29: attributs protégés de la classe IterateurCaractere

nom	type	description
caractereCourant_	char*	pointeur sur le caractère courant du tampon

Les méthodes protégées sont décrites dans la table 30.

TAB. 30: IterateurCaractere : méthodes protégées

signature	description
void actualiseEtat ()	remet l'itérateur en conformité avec l'état courant du tampon.
IterateurCaractere ()	constructeur par défaut.
IterateurCaractere (const IterateurCaractere & <i>other</i>)	constructeur par copie.
IterateurCaractere& operator = (const IterateurCaractere& <i>other</i>)	affectation.

12.12 classe IterateurChamp

description

La classe IterateurChamp permet de parcourir un TamponAscii de champ en champ. Les champs sont délimités par les séparateurs (la liste des séparateurs peut être modifiée par appel de méthode de TamponAscii). Si le tampon est interprété, un commentaire dans sa totalité est alors considéré comme un champ unique. Cet itérateur possède les propriétés suivantes :

- À un instant donné, l'itérateur possède une valeur qui désigne un élément du tampon. On accède à cette valeur par `valeur()` ;
- L'itérateur peut être incrémenté par l'opérateur `++` de manière à pointer sur l'élément suivant ;
- L'itérateur peut également être décrémenté par l'opérateur `--` de manière à pointer sur l'élément précédent (itérateur bidirectionnel) ;
- L'itérateur permet d'accéder directement à un élément du tampon en utilisant la méthode `allerSur(int index)`.

interface publique

```
#include "club/IterateurChamp.h"
```

TAB. 31: IterateurChamp : méthodes publiques

signature	description
IterateurChamp (const TamponAscii& <i>tampon</i> , int <i>compterChamps</i> = 1)	construit une instance parcourant <i>tampon</i> Exceptions : manque_guillemet
IterateurChamp (const TamponAscii* <i>tampon</i> , int <i>compterChamps</i> = 1)	construit une instance parcourant <i>tampon</i> Exceptions : manque_guillemet
IterateurChamp ()	destructeur de la classe.
void premier ()	positionne l'itérateur sur le premier champ du tampon. Exceptions : itérateur_invalide.
void dernier ()	positionne l'itérateur sur le dernier champ du tampon. Exceptions : tampon_vide, manque_guillemet, champ_hors_domaine, itérateur_invalide.
void allerSur (int <i>index</i>)	positionne l'itérateur sur le champ numéro <i>index</i> . Les champs sont numérotés à partir de 1.
int termine () const	indique que l'itérateur est à la fin du tampon.
int nombreIterations () const	retourne le nombre de champs du tampon. Exceptions : itérateur_invalide.
int operator++ (int)	positionne l'itérateur sur le champ suivant du tampon. Exceptions : tampon_vide, manque_guillemet, champ_hors_domaine, itérateur_invalide.
int operator-- (int)	positionne l'itérateur sur le champ précédent du tampon Exceptions : tampon_vide, manque_guillemet, champ_hors_domaine, itérateur_invalide.
à suivre ...	

TAB. 31: IterateurChamp : méthodes publiques (suite)

signature	description
int longueurElement (int avecFin = 0) const	retourne la longueur de l'élément pointé par l'itérateur. Exceptions : manque_guillemet, itérateur_invalide.
const char* valeur () const	retourne un pointeur sur le champ courant. Exceptions : itérateur_invalide.
void synchronize (int argumentdirection = 0)	Si direction=1, synchronise la position de l'itérateur sur la position réelle dans le tampon (itérateur altéré lors d'une modification du tampon). Si direction = 0 (par défaut), synchronise l'état du tampon (position courante du curseur) sur la position courante de l'itérateur (mémorisation avant altération du tampon) Exceptions : itérateur_invalide
void actualise ()	réinitialise l'itérateur sur le début du tampon.

exemple d'utilisation

```
#include "club/IterateurChamp.h"
#include "club/TamponAscii.h"
...
try
{ TamponAscii t(nom_fichier);
  IterateurChamp itc(t);

  // acces direct
  itc.allerSur( 100 );
  cout << " valeur de l'element 100 : " ;
  cout << itc.valeur() << endl;
}
catch (ClubErreurs ce)
{ //exemple de discrimination des exceptions
  if (ce.code()==ClubErreurs::champ_hors_domaine)
  {
    cout << " le champ demande est hors domaine"<<endl;
    ce.correction();
  }
  else
  {
    cout << "Erreur inattendue : "<< ce.why()<<endl;
    ce.correction();
  }
}
```

conseils d'utilisation spécifiques

Les opérateurs ++ et -- retournent 0 lorsqu'ils sont arrivés à l'extrémité du tampon. Il est ainsi possible de tester leur code retour dans un while pour parcourir l'ensemble du tampon.

Les classes `IterateurX` développées pour la bibliothèque `CLUB` sont dédiées à la gestion d'objets `TamponAscii`. En conséquence, les interfaces de ces classes n'offrent que les méthodes utiles à cette gestion. Si les besoins de parcours d'un tampon sont des besoins classiques, la classe `TamponTexte` offre les fonctionnalités nécessaires, sans nécessiter la manipulation d'itérateurs.

implantation

Les attributs protégés sont décrits sommairement dans la table 32, il n'y a pas d'attribut privé.

TAB. 32: attributs protégés de la classe `IterateurChamp`

nom	type	description
<code>debutChamp_</code>	<code>char *</code>	pointeur sur le début du champ
<code>numChamp_</code>	<code>int</code>	numéro du champ courant
<code>nombreChamps_</code>	<code>int</code>	nombre total de champs du tampon

Les méthodes protégées sont décrites dans la table 33.

TAB. 33: `IterateurChamp` : méthodes protégées

signature	description
<code>void actualiseEtat ()</code>	remet l'itérateur en conformité avec l'état courant du tampon.
<code>int analyseChamp (const char* mobile, int sauteFinLigne) const</code>	analyse lexicale du champ courant avec mémorisation de la valeur du champ. Exceptions : manque_guilletmet.
<code>int compteChamps (char* mobile, int sauteFinLigne) const</code>	retourne de nombre de champs à partir d'une position donnée jusqu'à la fin de ligne ou de tampon.
<code>const char* sauteChampAvant (char* mobile, int n, int sauteFinLigne) const</code>	saute <i>n</i> champs à partir d'une position donnée.
<code>void rechercheChamp (int c, int sauteFinLigne)</code>	recherche du champ numéro <i>c</i> Exceptions : tampon_vide, champ_hors_domaine, manque_guilletmet.
<code>void debutDeChamp (const char *mobile, int sauteFinLigne)</code>	positionne l'itérateur sur un début de champ.
<code>IterateurChamp ()</code>	constructeur par défaut.
<code>IterateurChamp (const IterateurChamp & other)</code>	constructeur par copie.
<code>IterateurChamp& operator = (const IterateurChamp& other)</code>	affectation.

12.13 classe `IterateurLigne`

description

La classe `IterateurLigne` permet de parcourir un `TamponAscii` de ligne en ligne. Cet itérateur possède les propriétés suivantes :

- À un instant donné, l'itérateur possède une valeur qui désigne un élément du tampon. On accède à cette valeur par `valeur()` ;
- L'itérateur peut être incrémenté par l'opérateur `++` de manière à pointer sur l'élément suivant ;
- L'itérateur peut également être décrémenté par l'opérateur `--` de manière à pointer sur l'élément précédent (itérateur bidirectionnel) ;
- L'itérateur permet d'accéder directement à un élément du tampon en utilisant la méthode `allerSur(int index)`.

interface publique

```
#include "club/IterateurLigne.h"
```

TAB. 34: IterateurLigne : méthodes publiques

signature	description
IterateurLigne (const TamponAscii& <i>tampon</i> , int <i>compterLignes</i> = 1) IterateurLigne (TamponAscii* <i>tampon</i> , int <i>compterLignes</i> = 1) IterateurLigne ()	construit une instance parcourant <i>tampon</i> Exceptions : <code>iterateur_invalide</code> . construit une instance parcourant <i>tampon</i> Exceptions : <code>iterateur_invalide</code> . destructeur.
void premier () void dernier () void allerSur (int <i>index</i>)	positionne l'itérateur sur la première ligne du tampon. Exceptions : <code>iterateur_invalide</code> . positionne l'itérateur sur la dernière ligne du tampon. Exceptions : <code>ligne_hors_domaine</code> , <code>iterateur_invalide</code> . positionne l'itérateur sur la ligne <i>index</i> du tampon. Exceptions : <code>ligne_hors_domaine</code> , <code>iterateur_invalide</code> .
int termine () const	indique que l'itérateur est à la fin du tampon.
int nombreIterations () const int operator++ (int) int operator-- (int)	retourne le nombre de lignes du tampon. Exceptions : <code>iterateur_invalide</code> . positionne l'itérateur sur la ligne suivante du tampon. Exceptions : <code>ligne_hors_domaine</code> , <code>iterateur_invalide</code> . positionne l'itérateur sur la ligne précédente du tampon. Exceptions : <code>ligne_hors_domaine</code> , <code>iterateur_invalide</code> .
int longueurElement (int <i>avecFin</i> = 0) const const char* valeur () const void synchronize (int <i>direction</i> = 0) void actualise ()	retourne la longueur de la ligne. Exceptions : <code>iterateur_invalide</code> . retourne un pointeur sur la ligne courante. Exceptions : <code>iterateur_invalide</code> . si <i>direction</i> =1, synchronise la position de l'itérateur sur la position réelle dans le tampon (itérateur altéré lors d'une modification du tampon). Si <i>direction</i> = 0 (par défaut), synchronise l'état du tampon (position courante du curseur) sur la position courante de l'itérateur (mémorisation avant altération du tampon) Exceptions : <code>iterateur_invalide</code> . réinitialise l'itérateur sur le début du tampon.

exemple d'utilisation

```
#include "club/IterateurLigne.h"
#include "club/TamponAscii.h"
...

TamponAscii t(nomfichier);
IterateurLigne itl(t);
cout << "nombre de lignes : "<< itl.nombreIterations()<<endl;

// n'afficher que les lignes de plus de 5 caracteres
itl.premier();
do
{
    if(itl.longueurElement() >=5)
        cout << itl.valeur()<<endl;
}
while (itl++);
```

conseils d'utilisation spécifiques

Les opérateurs ++ et -- retournent 0 lorsqu'ils sont arrivés à l'extrémité du tampon. Il est ainsi possible de tester leur code retour dans un while (cf exemple d'utilisation précédent).

Les classes IterateurX développées pour la bibliothèque CLUB sont dédiées à la gestion d'objets TamponAscii. En conséquence, les interfaces de ces classes n'offrent que les méthodes utiles à cette gestion. Si les besoins de parcours d'un tampon sont des besoins classiques, la classe TamponTexte offre les fonctionnalités nécessaires, sans nécessiter la manipulation d'itérateurs.

implantation

Les attributs protégés sont décrits sommairement dans la table 35, il n'y a pas d'attribut privé.

TAB. 35: attributs protégés de la classe IterateurLigne

nom	type	description
debutLigne_	char*	pointeur sur le début de la ligne courante
numLigne_	int	numéro de la ligne courante
nombreLignes_	int	nombre total de lignes du tampon.

Les méthodes protégées sont décrites dans la table 36.

TAB. 36: IterateurLigne : méthodes protégées

signature	description
void actualiseEtat ()	remet l'itérateur en conformité avec l'état courant du tampon.
	à suivre ...

TAB. 36: IterateurLigne : méthodes protégées (suite)

signature	description
void rechercheLigne (int <i>l</i>)	recherche la ligne <i>l</i> Exceptions : ligne_hors_domaine.
int compteLignes (const char* <i>mobile</i>) const	retourne le nombre de lignes de <i>mobile</i> jusqu'à la fin du tampon.
IterateurLigne ()	constructeur par défaut.
IterateurLigne (const IterateurLigne & <i>other</i>)	constructeur par copie.
IterateurLigne & operator = (const IterateurLigne & <i>other</i>)	affectation.

12.14 classe IterateurChampLigne

description

La classe IterateurChampLigne permet de parcourir un TamponAscii par ligne et par champ. Cette classe hérite de IterateurChamp et IterateurLigne. Les champs sont délimités par les séparateurs (la liste des séparateurs peut être modifiée par appel de méthode de TamponAscii). Si le tampon est interprété, un commentaire entier est considéré comme un champ. Cet itérateur possède les propriétés suivantes :

- L'itérateur peut être incrémenté par l'opérateur ++ de manière à pointer sur l'élément suivant ;
- L'itérateur peut également être décrémenté par l'opérateur -- de manière à pointer sur l'élément précédent (itérateur bidirectionnel) ;
- L'itérateur permet d'accéder directement à un élément du tampon en utilisant la méthode **allerSur** (int *index*) ;
- Il est possible d'accéder à une ligne ne contenant pas de champ (ligne vide) en mettant 0 comme numéro de champ à **allerSur**. Dans ce cas l'itérateur ne pointe pas sur un champ et l'appel à **valeur()** déclenchera une exception.

interface publique

```
#include "club/IterateurChampLigne.h"
```

TAB. 37: IterateurChampLigne : méthodes publiques

signature	description
IterateurChampLigne (const TamponAscii & <i>tampon</i> , int <i>compterChampsLignes</i> = 1)	construit une instance parcourant <i>tampon</i> . Exceptions : itérateur_invalide.
IterateurChampLigne (const TamponAscii* <i>tampon</i> , int <i>compterChampsLignes</i> = 1)	construit une instance parcourant <i>tampon</i> . Exceptions : itérateur_invalide.
IterateurChampLigne ()	destructeur de la classe.
void premier ()	positionne l'itérateur sur le premier champ de la première ligne Exceptions : itérateur_invalide.
à suivre ...	

TAB. 37: IterateurChampLigne : méthodes publiques (suite)

signature	description
void dernier ()	positionne l'itérateur sur le dernier champ de la dernière ligne Exceptions : tampon_vide, ligne_hors_domaine, itérateur_invalide, champ_hors_domaine.
void allerSur (int <i>champ</i> , int <i>ligne</i>)	positionne l'itérateur sur le champ numéro <i>champ</i> de la ligne numéro <i>ligne</i> . Exceptions : tampon_vide, ligne_hors_domaine, itérateur_invalide, champ_hors_domaine, champ_ligne_hors_domaine.
void allerSur (int <i>index</i>)	positionne l'itérateur sur le champ numéro <i>index</i> de la ligne courante. Exceptions : tampon_vide, itérateur_invalide, champ_hors_domaine.
int termine () const	indique que l'itérateur est à la fin du tampon
int nombreIterations () const	retourne le nombre de lignes du Tampon Exceptions : itérateur_invalide.
int nombreIterationsChamps (int <i>l</i> = 0)	retourne le nombre de champs dans la ligne <i>l</i> ou dans la ligne courante si <i>l</i> = 0 (défaut) Exceptions : ligne_hors_domaine, itérateur_invalide.
int operator++ (int)	positionne l'itérateur sur le champ suivant dans le tampon. Exceptions : tampon_vide, itérateur_invalide, champ_hors_domaine, ligne_hors_domaine.
int operator-- (int)	positionne l'itérateur sur le champ précédent dans le tampon. Exceptions : tampon_vide, itérateur_invalide, champ_hors_domaine, ligne_hors_domaine.
int longueurElement (int <i>avecFin</i> = 0) const	retourne la longueur de l'élément courant. Exceptions : manque_guillemet, itérateur_invalide.
const char* valeur () const	retourne la valeur du champ courant Exceptions : itérateur_invalide.
void synchronise (int <i>direction</i> = 0)	Si <i>direction</i> = 1, synchronise la position de l'itérateur sur la position réelle dans le tampon (itérateur altéré lors d'une modification du tampon). Si <i>direction</i> = 0 (par défaut), synchronise l'état du tampon (position courante du curseur) sur la position courante de l'itérateur (mémorisation avant altération du tampon) Exceptions : itérateur_invalide.
void actualise ()	réinitialise l'itérateur sur le début du tampon. Exceptions : itérateur_invalide.

exemple d'utilisation

```
#include <iostream.h>
#include <strstream.h>
#include "club/IterateurChampLigne.h"
#include "club/TamponAscii.h"
...
try
{
    TamponAscii t(nom_fichier);
    IterateurChampLigne itcl( t );
    cout << endl << "\t1- Insertion en debut" << endl;
```



```

cout <<endl <<"insertion du numero de l'element au debut de chacun d'eux.";
cout << endl;
char numLig[10]="\0";
int i=0;
    int maxI=0;
    int j=0;
    for( itcl.premier(); !itcl.termine(); itcl++ )
    {
        strstream numToChaine;
        if( i >= maxI ){
            i = 0;
            maxI = itcl.nombreIterationsChamps();
            numToChaine << "[" << ++j << "]";
        }
        i++;

        numToChaine << i;
        numToChaine >> numLig;
        t.insereFin( itcl, numLig );
    }
cout <<endl<< "Contenu du tampon : "<<endl<<t.total()<<endl;
}
catch (ClubErreurs ce)
{
    ...
}

```

conseils d'utilisation spécifiques

Les opérateurs ++ et -- retournent 0 lorsqu'ils sont arrivés à l'extrémité du tampon. Il est ainsi possible de tester leur code retour dans un while pour parcourir l'ensemble du tampon.

Les classes IterateurX développées pour la bibliothèque CLUB sont dédiées à la gestion d'objets TamponAscii. En conséquence, les interfaces de ces classes n'offrent que les méthodes utiles à cette gestion. Si les besoins de parcours d'un tampon sont des besoins classiques, la classe TamponTexte offre les fonctionnalités nécessaires, sans nécessiter la manipulation d'itérateurs.

TAB. 38: IterateurChampLigne : méthodes protégées

signature	description
IterateurChampLigne ()	constructeur par défaut.

implantation

Il n'y a ni attribut privé, ni attribut protégé. Les méthodes privées sont décrites dans la table 39.

TAB. 39: IterateurChampLigne : méthodes privées

signature	description
void actualiseEtat ()	met à jour les différents compteurs ainsi que le pointeur courant
void rechercheChampLigne (int c)	recherche le champ numéro c dans la ligne courante. Exceptions : tampon_vide, champ_hors_domaine.
void debutDeChampLigne (const char* <i>mobile</i>)	repositionne sur le début du champ qui contient mobile pour la ligne courante.
IterateurChampLigne (const IterateurChampLigne & <i>other</i>)	constructeur par copie.
IterateurChampLigne & operator = (const IterateurChampLigne & <i>other</i>)	affectation.

12.15 classe MadonnaFile

description

Cette classe fournit une interface d'accès aux données contenues dans un fichier au format «Madona». Elle encapsule ainsi dans une interface simple et générique les appels aux services de la bibliothèque Madona.

Cette classe dérivant de UniqDataFile, elle hérite de toutes les méthodes publiques et protégées de UniqDataFile et implémente les méthodes virtuelles pures de UniqDataFile.

interface publique

```
#include "club/MadonaFile.h"
```

La majorité des méthodes publiques de la classe MadonnaFile sont décrites dans le tableau 75 page 107.

Le tableau 40 présente les méthodes publiques propres à la classe MadonnaFile.

TAB. 40: MadonnaFile : méthodes publiques

signature	description
MadonaFile ()	construit une instance en ouvrant une zone d'accès Exceptions : madona_error
~MadonaFile ()	détruit l'instance en fermant la zone d'accès.. Exceptions : madona_error

exemple d'utilisation

Un exemple d'utilisation des services généraux de gestion des formats de fichiers (Madona, XML ou FichierStructure) est fourni dans la section 12.24 page 106.

implantation

Les attributs protégés sont décrits sommairement dans la table 41, il n'y a pas d'attribut privé.

TAB. 41: attributs protégés de la classe *MadonaFile*

nom	type	description
<code>access_</code>	ACCES*	buffer partagé sur la zone d'accès <i>Madona</i>
<code>selectCount_</code>	int	compte le nombre de sélections à réaliser pour aller de la donnée racine à la donnée courante

Les méthodes protégées sont décrites dans la table 42.

TAB. 42: *MadonaFile* : méthodes protégées

signature	description
ACCES* getAccess ()	retourne le pointeur sur la zone d'accès <i>Madona</i>
bool canMoveUp ()	teste s'il est possible de remonter d'un niveau dans la structure de données
void reposition ()	repositionne le pointeur <i>madona</i> sur la donnée courante. Le pointeur pouvant être partagé entre plusieurs instances de la classe <i>MadonaFile</i> , le pointeur peut ne pas pointer sur la donnée courante d'une instance. Pour garantir l'accès aux données, chaque opération de lecture, écriture ou de déplacement dans la structure de données fait appel à cette méthode. Exceptions : <i>madona_error</i>

Les méthodes privées sont décrites dans la table 43.

TAB. 43: *MadonaFile* : méthodes privées

signature	description
MadonaFile (const & <i>MadonaFile</i>)	constructeur par copie.
<i>MadonaFile</i> & operator = (const & <i>MadonaFile</i>)	affectation.

12.16 classe *OptionBase*

description

Cette classe est une classe abstraite de description générique d'option pour la ligne de commande d'une application. Elle regroupe tous les attributs et toutes les opérations communes à tous les types d'options. Elle contient également toutes les fonctions virtuelles qui seront implantées dans les classes dérivées.

interface publique

```
#include "club/OptionBase.h"
```

Le fichier de déclaration de la classe définit deux constantes entières destinées au paramétrage des constructeurs des classes dérivées, et un type énuméré anonyme interne destiné à reconnaître les divers types d'options :

```
const int avecValeurDefaut = 1;
const int sansValeurDefaut = 0;

class OptionBase
{ ...
public :
    enum {tableau_d_entiers, tableau_de_reels, tableau_de_chaines,
          sans_valeur};
    ...
}
```

TAB. 44: OptionBase : méthodes publiques

signature	description
OptionBase (const char *code, int occurrencesMin, int occurrencesMax, int valueeParDefaut, int type, int valeursAttendues)	construit la partie générique d'une instance d'option Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_llettre.
OptionBase (const string& code, int occurrencesMin, int occurrencesMax, int valueeParDefaut, int type, int valeursAttendues)	construit la partie générique d'une instance d'option Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_llettre.
OptionBase (const OptionBase& opt)	constructeur par copie
OptionBase& operator = (const OptionBase& opt)	affectation
~OptionBase ()	destructeur, libère la mémoire allouée
const string& code () const	retourne le code de l'option
int occurrencesMin () const	retourne le nombre minimal d'occurrences toléré pour l'option
int occurrencesMax () const	retourne le nombre maximal d'occurrences toléré pour l'option
int occurrencesPassees () const	retourne le nombre d'occurrences passées dans la ligne de commande
int estValueeParDefaut () const	indique si l'option a une valeur par défaut
int type () const	retourne le type de l'option (parmi les valeurs de l'énuméré anonyme de la classe).
int valeursAttendues () const	retourne le nombre d'arguments attendus pour chaque occurrence de l'option
int valeursTrouvees () const	retourne le nombre d'arguments trouvés pour la dernière occurrence de l'option
virtual int verifier (string *ptrMessage) const	vérifie les contraintes de l'option (nombre d'occurrences, domaine de validité) et indique si elles sont violées, en retournant un message explicatif dans la variable pointée par ptrMessage
à suivre ...	

TAB. 44: OptionBase : méthodes publiques (suite)

signature	description
void lireValeur (int *dest, int occurrence = 1) const	lit la valeur de l'occurrence spécifiée de l'option et la met dans dest. Exceptions : lecture_types_incompatibles, taille_tableau_arguments.
void lireValeur (double *dest, int occurrence = 1) const	lit la valeur de l'occurrence spécifiée de l'option et la met dans dest. Exceptions : lecture_types_incompatibles, taille_tableau_arguments.
void lireValeur (char *dest, int occurrence = 1) const	lit la valeur de l'occurrence spécifiée de l'option et la met dans dest. Exceptions : lecture_types_incompatibles, taille_tableau_arguments.
void lireValeur (string *dest, int occurrence = 1) const	lit la valeur de l'occurrence spécifiée de l'option et la met dans dest. Exceptions : lecture_types_incompatibles, taille_tableau_arguments.
virtual void lireValeur (int nombre, int *dest, int occurrence = 1) const	lit les valeurs de l'occurrence spécifiée de l'option et les met dans dest. Exceptions : lecture_types_incompatibles.
virtual void lireValeur (int nombre, double *dest, int occurrence = 1) const	lit les valeurs de l'occurrence spécifiée de l'option et les met dans dest. Exceptions : lecture_types_incompatibles.
virtual void lireValeur (int nombre, char* *dest, int occurrence = 1) const	lit les valeurs de l'occurrence spécifiée de l'option et les met dans dest. Exceptions : lecture_types_incompatibles.
virtual void lireValeur (int nombre, string *dest, int occurrence = 1) const	lit les valeurs de l'occurrence spécifiée de l'option et les met dans dest. Exceptions : lecture_types_incompatibles.
virtual void ajouterOccurrence ()	ajoute une occurrence de l'option Exceptions : nombres_arguments_incompatibles, occurrence_hors_limites.
virtual void ajouterValeur (const char *chaine)	ajoute une valeur au tableau des valeurs de l'occurrence courante de l'option

TAB. 45: OptionBase : méthodes protégées

signature	description
OptionBase ()	constructeur par défaut

exemple d'utilisation

Cette classe étant abstraite, elle ne peut être utilisée directement. Seules ses classes dérivées peuvent être instanciées.

conseils d'utilisation spécifiques

Cette classe étant abstraite, elle ne peut être utilisée directement. Seules ses classes dérivées peuvent être instanciées.

implantation

Les attributs sont décrits sommairement dans la table 46, la seule méthode protégée est la méthode `incrementeValeursTrouvees()` qui incrémente le nombre de valeurs trouvées pour l'occurrence courante de l'option.

TAB. 46: attributs de la classe OptionBase

nom	type	description
<code>code_</code>	string	code de l'option
<code>occurrencesMin_</code>	int	nombre minimal toléré d'occurrences
<code>occurrencesMax_</code>	int	nombre maximal toléré d'occurrences
<code>occurrences_</code>	int	nombre d'occurrences passées dans la ligne de commande
<code>valueeParDefaut_</code>	int	indicateur de valuation par défaut
<code>type_</code>	int	typoe de l'option
<code>valeursAttendues_</code>	int	nombre de valeurs attendues par occurrence
<code>valeursTrouvees_</code>	int	nombre de valeurs trouvées pour la dernière occurrence

12.17 classes dérivées de OptionBase

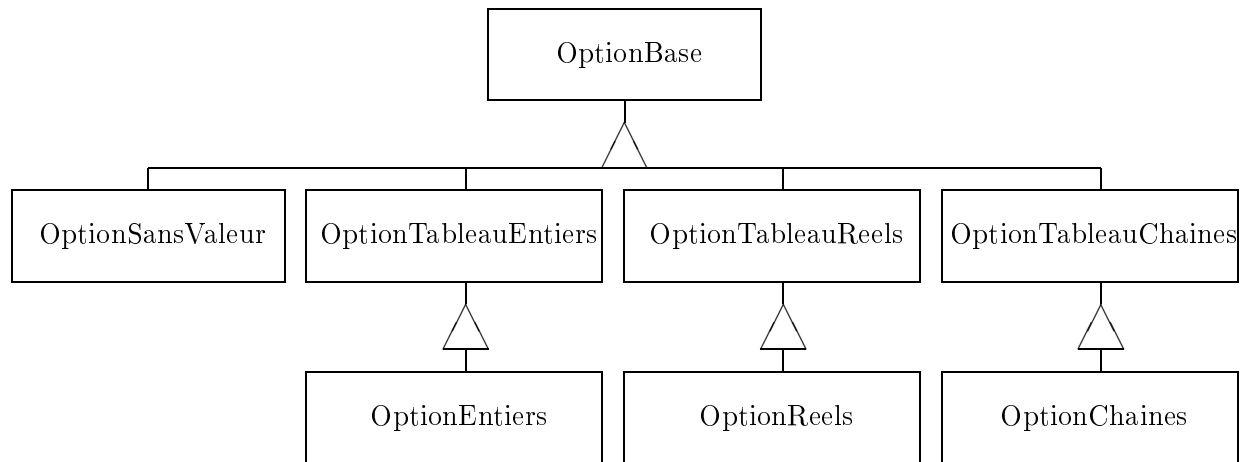
description

Les classes dérivées de la classe OptionBase modélisent les divers types d'options dont on peut avoir besoin dans la ligne de commande d'une application. Ces options sont différenciées par les arguments qu'elles attendent, comme le montre la table 47.

TAB. 47 – classes dérivées de le classe OptionBase

classe	arguments attendus
OptionSansValeur	rien
OptionChaine	une chaîne de caractères
OptionTabChaines	un tableau de chaînes de caractères
OptionEntier	un entier
OptionTabEntiers	un tableau d'entiers
OptionReel	un réel
OptionTabReels	un tableau de réels

FIG. 1 – Hiérarchie d'héritage des classes d'options



interface publique

```
#include "club/OptionsAppli.h"
```

Le fichier de déclaration de la classe OptionsAppli (qui déclare la classe d'analyseur qui regroupera les instances d'options) inclut directement tous les fichiers d'en-tête des diverses options.

Les méthodes publiques de toutes ces classes sont héritées (ou redéfinies) à partir des méthodes de la classe OptionBase, elles sont décrites dans la table 44, page 80.

Les méthodes qui ne sont pas applicables à un type d'option (par exemple lire une valeur réelle pour une option attendant un tableau de chaînes de caractères) génèrent systématiquement un message d'erreur explicitant le conflit de type.

Les seules méthodes nécessitant une description sont les constructeurs, qui sont spécifiques à chaque classe dérivée :

TAB. 48: constructeurs des classes dérivées de OptionBase

signature	description
OptionSansValeur (const char *code, int occurrencesMin, int occurrencesMax)	construit une option n'attendant aucun argument Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_llettre.
OptionSansValeur (const string& code, int occurrencesMin, int occurrencesMax)	construit une option n'attendant aucun argument Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_llettre.
OptionChaine (const char *code, int occurrencesMin, int occurrencesMax, int valueeParDefaut, const char *valeur, int longueurMax = 1000, int nbAutorisees = 0, const char **tabAutorisees = 0)	construit une option attendant une chaîne de caractères dont on peut limiter la taille et dont on peut restreindre la à un ensemble prédéfini Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_llettre, nombre_arguments_nul, option_chaine_trop_longue, option_chaine_hors_domaine, valeur_par_defaut_manquante.
à suivre ...	

TAB. 48: constructeurs des classes dérivées de OptionBase (suite)

signature	description
OptionChaine (const string& code, int occurrencesMin, int occurrencesMax, int valueeParDefault, const string& valeur, int longueurMax = 1000, int nbAutorisees = 0, const string *tabAutorisees = 0)	construit une option attendant une chaîne de caractères dont on peut limiter la taille et dont on peut restreindre la à un ensemble prédéfini Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, option_chaine_trop_longue, option_chaine_hors_domaine, valeur_par_defaut_manquante.
OptionTableauChaines (const char *code, int occurrencesMin, int occurrencesMax, int valueeParDefault, int nombreValeurs, const char **valeurs, int longueurMax = 1000, int nbAutorisees = 0, const char **tabAutorisees = 0)	construit une option attendant un tableau de chaînes de caractères dont on peut limiter la taille et dont on peut restreindre la à un ensemble prédéfini Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, option_chaine_trop_longue, option_chaine_hors_domaine, valeur_par_defaut_manquante.
OptionTableauChaines (const string& code, int occurrencesMin, int occurrencesMax, int valueeParDefault, int nombreValeurs, const string& valeurs, int longueurMax = 1000, int nbAutorisees = 0, const string *tabAutorisees = 0)	construit une option attendant un tableau de chaînes de caractères dont on peut limiter la taille et dont on peut restreindre la à un ensemble prédéfini Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, option_chaine_trop_longue, option_chaine_hors_domaine, valeur_par_defaut_manquante.
OptionEntier (const char *code, int occurrencesMin, int occurrencesMax, int valueeParDefault, int valeur = 0, int limiteInf = INT_MIN, int limiteSup = INT_MAX)	construit une option attendant un entier que l'on peut restreindre à un domaine particulier Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, valeur_par_defaut_manquante, option_entiere_hors_limites.
OptionEntier (const string& code, int occurrencesMin, int occurrencesMax, int valueeParDefault, int valeur = 0, int limiteInf = INT_MIN, int limiteSup = INT_MAX)	construit une option attendant un entier que l'on peut restreindre à un domaine particulier Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, valeur_par_defaut_manquante, option_entiere_hors_limites.
OptionTableauEntiers (const char *code, int occurrencesMin, int occurrencesMax, int valueeParDefault, int nombreValeurs, int valeurs = 0, int limiteInf = INT_MIN, int limiteSup = INT_MAX)	construit une option attendant un tableau d'entiers que l'on peut restreindre à un domaine particulier Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, valeur_par_defaut_manquante, option_entiere_hors_limites.
OptionTableauEntiers (const string& code, int occurrencesMin, int occurrencesMax, int valueeParDefault, int nombreValeurs, int valeurs = 0, int limiteInf = INT_MIN, int limiteSup = INT_MAX)	construit une option attendant un tableau d'entiers que l'on peut restreindre à un domaine particulier Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, valeur_par_defaut_manquante, option_entiere_hors_limites.
à suivre ...	

TAB. 48: constructeurs des classes dérivées de OptionBase (suite)

signature	description
OptionReel (const char *code, int occurrencesMin, int occurrencesMax, int valeurParDefault, double valeur = 0, double limiteInf = -DBL_MAX, double limiteSup = DBL_MAX)	construit une option attendant un réel que l'on peut restreindre à un domaine particulier Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, valeur_par_defaut_manquante, option_entiere_hors_limites.
OptionReel (const string& code, int occurrencesMin, int occurrencesMax, int valeurParDefault, double valeur = 0, double limiteInf = -DBL_MAX, double limiteSup = DBL_MAX)	construit une option attendant un réel que l'on peut restreindre à un domaine particulier Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, valeur_par_defaut_manquante, option_entiere_hors_limites.
OptionTableauReels (const char *code, int occurrencesMin, int occurrencesMax, int valeurParDefault, int nombreValeurs, double valeurs = 0, double limiteInf = -DBL_MAX, double limiteSup = DBL_MAX)	construit une option attendant un tableau de réels que l'on peut restreindre à un domaine particulier Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, valeur_par_defaut_manquante, option_entiere_hors_limites.
OptionTableauReels (const string& code, int occurrencesMin, int occurrencesMax, int valeurParDefault, int nombreValeurs, double valeurs = 0, double limiteInf = -DBL_MAX, double limiteSup = DBL_MAX)	construit une option attendant un tableau de réels que l'on peut restreindre à un domaine particulier Exceptions : nb_arguments_option_vide, nombre_arguments_negatif, nom_option_non_lettre, nombre_arguments_nul, valeur_par_defaut_manquante, option_entiere_hors_limites.

exemple d'utilisation

Un exemple d'utilisation global du système d'analyse des options de la ligne de commande apparaît dans la section 12.18 relative à la classe OptionsAppli.

conseils d'utilisation spécifiques

Voir section 12.18 relative à la classe OptionsAppli.

implantation

La classe OptionSansValeur n'a aucun attribut supplémentaire par rapport à sa classe de base.

Les options sans tableau sont dérivées des classe avec tableau, ce sont des spécialisations pour la cas où la dimension du tableau est 1.

TAB. 49: attributs de la classe OptionTableauChaine

nom	type	description
tabValeurs_	string **	table des valeurs pour chaque occurrence
longueurMax_	int	longueur maximale tolérée pour les chaînes
nbAutorisees_	int	nombre d'alternatives autorisées pour les chaînes
tabAutorisees_	string	table des alternatives autorisées pour les chaînes

TAB. 50: attributs de la classe OptionTableauEntiers

nom	type	description
tabValeurs_	int **	table des valeurs pour chaque occurrence
limiteInf_	int	valeur minimale tolérée
limiteSup_	int	valeur maximale tolérée

TAB. 51: attributs de la classe OptionTableauReels

nom	type	description
tabValeurs_	double **	table des valeurs pour chaque occurrence
limiteInf_	double	valeur minimale tolérée
limiteSup_	double	valeur maximale tolérée

12.18 classe OptionsAppli

description

Cette classe gère un analyseur offrant l'accès à partir de leur code à toutes les informations qui portent sur les options passées à une application par l'intermédiaire de la ligne de commande.

Cet analyseur reconnaît les options attendant des arguments ou non, les options ayant un code en une ou plusieurs lettres introduites par le caractère '-', les options sans nom (par exemple *fichier* dans la commande `ls fichier`), et les options regroupées (comme par exemple `-cvf /dev/fd0` au lieu de `-c -v -f /dev/fd0`). Chaque option peut d'autre part avoir des caractéristiques propres (nombre d'occurrences minimal, maximal, domaine de validité, valeur par défaut).

interface publique

```
#include "club/OptionsAppli.h"
```

TAB. 52: OptionsAppli : méthodes publiques

signature	description
OptionsAppli (const char *nom)	construit un analyseur d'option pour l'application <i>nom</i> . Le <i>nom</i> est généralement la chaîne argv [0], mais peut être n'importe quoi d'autre, elle ne sert qu'à produire les messages d'erreur et d'usage.
OptionsAppli (const string& nom)	construit un analyseur d'option pour l'application <i>nom</i> . Le <i>nom</i> est généralement la chaîne argv [0], mais peut être n'importe quoi d'autre, elle ne sert qu'à produire les messages d'erreur et d'usage.
OptionsAppli (const OptionsAppli& opts)	constructeur par copie
OptionsAppli& operator = (const OptionsAppli& opts)	affectation
~OptionsAppli ()	destructeur, libère la mémoire allouée
void ajouterOption (const OptionEntier& option)	ajoute l' <i>option</i> à l'ensemble déjà géré par l'instance Exceptions : option_deja_definie, options_appli_deja_initialise.
void ajouterOption (const OptionTableauEntiers& option)	ajoute l' <i>option</i> à l'ensemble déjà géré par l'instance Exceptions : option_deja_definie, options_appli_deja_initialise.
void ajouterOption (const OptionReel& option)	ajoute l' <i>option</i> à l'ensemble déjà géré par l'instance Exceptions : option_deja_definie, options_appli_deja_initialise.
void ajouterOption (const OptionTableauReels& option)	ajoute l' <i>option</i> à l'ensemble déjà géré par l'instance Exceptions : option_deja_definie, options_appli_deja_initialise.
void ajouterOption (const OptionChaine& option)	ajoute l' <i>option</i> à l'ensemble déjà géré par l'instance Exceptions : option_deja_definie, options_appli_deja_initialise.
void ajouterOption (const OptionTableauChaines& option)	ajoute l' <i>option</i> à l'ensemble déjà géré par l'instance Exceptions : option_deja_definie, options_appli_deja_initialise.
void ajouterOption (const OptionSansValeur& option)	ajoute l' <i>option</i> à l'ensemble déjà géré par l'instance Exceptions : option_deja_definie, options_appli_deja_initialise.
void initialiser (int argc, const char *const argv [])	initialise l'instance avec les arguments de la ligne de commande et analyse cette ligne Exceptions : plus_d_argument, options_appli_deja_initialise.
const string& nom () const	retourne le nom de l'instance
const char * usage (int largeur) const	retourne une chaîne de description de l'usage de l'application formatée pour un écran de <i>largeur</i> colonnes
int verifier (string *ptrMessage) const	vérifie les contraintes portant sur les arguments de la ligne de commande analysée, retourne une valeur non nulle et initialise la chaîne pointée par <i>ptrMessage</i> en cas de violation d'une de ces contraintes
int argcReste () const	retourne le nombre d'arguments non reconnus après analyse
à suivre ...	

TAB. 52: OptionsAppli : méthodes publiques (suite)

signature	description
int argvReste (char **argv)	rempli le tableau <i>argv</i> avec les arguments non reconnus après analyse (ce tableau doit pouvoir contenir au moins argcReste () éléments). Les chaînes retournées sont allouées dynamiquement, la libération de la mémoire est laissée à la charge de l'appelant. Exceptions : options_appli_non_initialise.
int garantirComplet const	vérifie la validité des options passées et vérifie que l'analyse est complète, c'est à dire qu'il ne reste aucun argument non reconnu. Exceptions : message_simple.
int occurrencesPassees (const char *code) const	retourne le nombre d'occurrences de l'option <i>code</i> passées dans la ligne de commande (si une option n'est pas effectivement passée mais qu'elle a une valeur par défaut, celle est considérée comme une occurrence). Exceptions : options_appli_non_initialise, code_option_non_reconnu.
int estInitialise () const	indique si l'analyseur a été initialisé
void lireValeur (const char *code, int *dest, int occurrence = 1) const	lit la valeur de l' <i>occurrence</i> spécifiée de l'option <i>code</i> et la met dans <i>dest</i> . Exceptions : options_appli_non_initialise, code_option_non_reconnu.
void lireValeur (const char *code, double *dest, int occurrence = 1) const	lit la valeur de l' <i>occurrence</i> spécifiée de l'option <i>code</i> et la met dans <i>dest</i> . Exceptions : options_appli_non_initialise, code_option_non_reconnu.
void lireValeur (const char *code, char *dest, int occurrence = 1) const	lit la valeur de l' <i>occurrence</i> spécifiée de l'option <i>code</i> et la met dans <i>dest</i> . Exceptions : options_appli_non_initialise, code_option_non_reconnu.
void lireValeur (const char *code, string *dest, int occurrence = 1) const	lit la valeur de l' <i>occurrence</i> spécifiée de l'option <i>code</i> et la met dans <i>dest</i> . Exceptions : options_appli_non_initialise, code_option_non_reconnu.
void lireValeur (const char *code, int nombre, int dest [], int occurrence = 1) const	lit la valeur de l' <i>occurrence</i> spécifiée de l'option <i>code</i> et la met dans <i>dest</i> . Exceptions : options_appli_non_initialise, code_option_non_reconnu.
void lireValeur (const char *code, int nombre, double dest [], int occurrence = 1) const	lit la valeur de l' <i>occurrence</i> spécifiée de l'option <i>code</i> et la met dans <i>dest</i> . Exceptions : options_appli_non_initialise, code_option_non_reconnu.
void lireValeur (const char *code, int nombre, char* dest [], int occurrence = 1) const	lit la valeur de l' <i>occurrence</i> spécifiée de l'option <i>code</i> et la met dans <i>dest</i> . Exceptions : options_appli_non_initialise, code_option_non_reconnu.
int lireValeur (const char *code, int nombre, string dest [], int occurrence = 1) const	lit la valeur de l' <i>occurrence</i> spécifiée de l'option <i>code</i> et la met dans <i>dest</i> . Exceptions : options_appli_non_initialise, code_option_non_reconnu.

TAB. 53: OptionsAppli : méthodes protégées

signature	description
OptionsAppli ()	constructeur par défaut.

exemple d'utilisation

```
#include "club/OptionsAppli.h"
...
OptionsAppli analyseur (argv [0]);

// axe
// option axe: une occurrence, pas de valeur par défaut
analyseur.ajouterOption (OptionTableauReels ("axe", 0, 1, sansValeurDefaut, 3));

// angle
// option angle: une occurrence, pas de valeur par défaut
analyseur.ajouterOption (OptionReel ("angle", 0, 1, sansValeurDefaut,
                                0.0, -360.0, 360.0));

// quaternion
// option q: une occurrence, pas de valeur par défaut
analyseur.ajouterOption (OptionTableauReels ("q", 0, 1, sansValeurDefaut, 4));

// vecteur auquel appliquer la rotation
// option u: une occurrence, pas de valeur par défaut
analyseur.ajouterOption (OptionTableauReels ("u", 0, 1, sansValeurDefaut, 3));

// analyse de la ligne
analyseur.initialiser (argc, argv);
analyseur.garantirComplet ();

// recuperation des donnees
if (analyseur.occurrencesPassees ("axe") > 0)
{ // l'utilisateur a passe un axe et un angle
  if (analyseur.occurrencesPassees ("q") > 0)
  { char tampon [1000];
    (void) sprintf (tampon, "options -axe et -q incompatibles");
    ClubErreurs::erreur (0, ClubErreurs::message_simple, tampon);
  }
  double tableau [3];
  analyseur.lireValeur ("axe", 3, tableau);
  VecDBL axe (tableau [0], tableau [1], tableau [2]);

  if (analyseur.occurrencesPassees ("angle") <= 0)
  { char tampon [1000];
    (void) sprintf (tampon, "option -axe sans option -angle");
    ClubErreurs::erreur (0, ClubErreurs::message_simple, tampon);
  }
}
```

```
double angle;
analyseur.lireValeur ("angle", &angle);
angle = radians (angle);

RotDBL r (axe, angle);
AfficheRotation (r);
```

conseils d'utilisation spécifiques

Après création d'une instance vide d'OptionsAppli, on ajoute les options une à une, on initialise l'analyseur à l'aide des arguments de la ligne de commande, puis on récupère les résultats de l'analyse. L'initialisation est une étape après laquelle on ne peut plus ajouter d'options dans l'analyseur, et avant laquelle on ne peut pas récupérer les résultats de l'analyse.

implantation

Les attributs sont décrits sommairement dans la table 54, la seule méthode privée est la méthode **ajouterNonReconnu** (const char **chaine*) qui ajoute un argument à la liste des arguments non reconnus.

TAB. 54: attributs de la classe OptionsAppli

nom	type	description
tableOptions_	hash_map <string, OptionBase *, club_hash<string> >	table contenant l'ensemble des options pouvant être reconnues
nom_	string	nom de l'application
nombreNonReconnus_	int	nombre d'arguments non reconnus après analyse
argumentsNonReconnus_	string *	table des arguments non reconnus après analyse
initialise_	int	indicateur d'analyseur initialisé

12.19 classe StructureFile

description

Cette classe fournit une interface d'accès aux données contenues dans un fichier au format «FichierStructuré». Elle encapsule ainsi dans une interface simple et générique les appels aux services de **FichierStructure** (cf. section 12.8 page 59).

Cette classe dérivant de UniqDataFile, elle hérite de toutes les méthodes publiques et protégées de UniqDataFile et implémente les méthodes virtuelles pures de UniqDataFile. Les méthodes de lecture sont implémentés tandis que l'ensemble des services d'écriture lèvent l'exception nommée **not_implemented**.

interface publique

```
#include "club/StructureFile.h"
```

La majorité des méthodes publiques de la classe StructureFile sont décrites dans le tableau 75 page 107.

Le tableau 55 présente les méthodes publiques propres à la classe StructureFile.

TAB. 55: StructureFile : méthodes publiques

signature	description
StructureFile ()	construit une instance.
~StructureFile ()	détruit l'instance.

exemple d'utilisation

Un exemple d'utilisation des services généraux de gestion des formats de fichiers (Madona, XML ou FichierStructure) est fourni à la page 114.

conseils d'utilisation spécifiques

Seuls les services suivants de l'abstraction commune sont implémentés par StructureFile :

- chargement d'un FichierStructuré en mémoire,
- navigation dans la structure de données,
- lecture des données.

La gestion des commentaires associés à une donnée, l'écriture de la structure dans un fichier, la création, la mise à jour et la suppression des données et la gestion des unités physiques ne sont pas actuellement fournies par cette classe.

L'utilisateur peut néanmoins appeler ces services mais au risque de lever l'exception `not_implemented` qu'il faut rattraper sous peine que le programme se termine avec un `abort`.

implantation

Les attributs protégés sont décrits sommairement dans la table 56, il n'y a pas d'attribut privé.

TAB. 56: attributs protégés de la classe StructureFile

nom	type	description
<code>rootElement_</code>	FichierStructure	pointeur sur la donnée racine
<code>currentElement_</code>	FichierStructure	pointeur sur la donnée courante

Les méthodes protégées sont décrites dans la table 57.

TAB. 57: StructureFile : méthodes protégées

signature	description
FichierStructure& getRootElement ()	retourne le pointeur sur l'élément racine
void setRootElement (const FichierStructure& <i>rootElement</i>)	met à jour le pointeur sur l'élément racine
FichierStructure& getCurrentElement ()	retourne le pointeur sur l'élément courant
void setCurrentElement (const FichierStructure& <i>currentElement</i>)	met à jour le pointeur sur l'élément courant
static UniqDataFile::DataType getFichierStructureType (const FichierStructure& <i>fs</i>)	retourne le type de l'élément <i>fs</i>
bool canMoveUp ()	teste s'il est possible de remonter d'un niveau dans la structure de données

Les méthodes privées sont décrites dans la table 58.

TAB. 58: StructureFile : méthodes privées

signature	description
StructureFile (const StructureFile & <i>other</i>)	constructeur par copie.
StructureFile & operator = (const StructureFile & <i>other</i>)	affectation.

12.20 classe TamponAscii

description

Cette classe gère un tampon de texte ASCII pouvant comporter des lignes blanches, des commentaires, des chaînes de caractères comportant des caractères blancs ou spéciaux du langage C (`\n`, `\012`, ...). Les caractères —ou le champ complet— doivent alors être inclus entre `""`.

Ce tampon peut être initialisé à partir d'un fichier.

Cette classe de type conteneur déclare `IterateurAscii` comme classe amie, ainsi tous les itérateurs de cette famille peuvent parcourir `TamponAscii`.

interface publique

```
#include "club/TamponAscii.h"
```

TAB. 59: TamponAscii : méthodes publiques

signature	description
TamponAscii (int <i>interprete</i> = 1)	construit une instance vide
	à suivre ...

TAB. 59: TamponAscii : méthodes publiques (suite)

signature	description
TamponAscii (FILE* <i>ptrFic</i> , int <i>interprete</i> = 1)	contruit une instance à partir d'un descripteur de fichier. Exceptions : -
TamponAscii (const char* <i>nom</i> , int <i>interprete</i> = 1)	construit une instance à partir d'un nom de fichier Exceptions : ouverture_fichier
TamponAscii (const TamponAscii& <i>t</i>)	constructeur par copie
TamponAscii (const TamponAscii* <i>t</i>)	constructeur par copie
TamponAscii& operator = (const TamponAscii& <i>t</i>)	opérateur d'affectation.
~TamponAscii ()	destructeur, libère la mémoire allouée
void positionneInterprete (int <i>interprete</i>) int interprete () const void initCarCommentaires (const char* <i>chaine</i>) void ajouteCarCommentaires (const char* <i>chaine</i>) const string& carCommentaires () const void elimineCommentaires ()	modification du drapeau d'interprétation des commentaires. retourne le drapeau d'interprétation des commentaires. utilise les caractères de <i>chaine</i> comme marqueurs de commentaires. Si <i>chaine</i> est vide ou le pointeur nul, cette méthode réinitialise à la valeur par défaut (#) ajoute les caractères de <i>chaine</i> à la liste des caractères marqueurs de commentaires. retourne la liste des caractères marqueurs de commentaires élimine les caractères qui vont d'un caractère marqueur de commentaire à la fin de la ligne (sauf si le marqueur est dans une chaîne entre '"'). Exceptions : manque_guillemet.
void initSeparateurs (const char* <i>chaine</i>) void ajouteSeparateurs (const char* <i>chaine</i>) const string& separateurs () const void elimineBlancs ()	utilise les caractères de <i>chaine</i> comme caractères séparateurs de champs. Si <i>chaine</i> est vide ou le pointeur nul, cette fonction réinitialise à la valeur par défaut qui est "\t\n\v\f\r " ajoute les caractères de <i>chaine</i> à la liste des caractères séparateurs retourne la liste des caractères séparateurs. élimine les caractères blancs de début et de fin de ligne. Pour une ligne entièrement blanche, le '\n' est également éliminé.
void insereDebut (IterateurAscii& <i>pos</i> , const char* <i>valeur</i>) void insereFin (IterateurAscii& <i>pos</i> , const char* <i>valeur</i>)	insère avant l'itérateur <i>pos</i> la chaîne <i>valeur</i> . Exceptions : itérateur_invalide. insère après l'itérateur <i>pos</i> la chaîne <i>valeur</i> . Exceptions : itérateur_invalide.
void supprime (IterateurAscii& <i>pos</i>) void supprime (IterateurAscii& <i>debut</i> , IterateurAscii& <i>fin</i>)	supprime l'élément courant pointé par l'itérateur. Exceptions : itérateur_invalide. supprime la zone comprise entre les 2 itérateurs. Exceptions : itérateur_invalide.
void remplace (IterateurAscii& <i>pos</i> , const char * <i>valeur</i>)	remplace l'élément pointé par l'itérateur par <i>valeur</i> . Exceptions : itérateur_invalide.
à suivre ...	

TAB. 59: TamponAscii : méthodes publiques (suite)

signature	description
void remplace (IterateurAscii& <i>debut</i> , IterateurAscii& <i>fin</i> , const char* <i>valeur</i>)	remplace la zone comprise entre les 2 itérateurs par <i>valeur</i> Exceptions : itérateur_invalide.

exemple d'utilisation

```
#include "club/TamponAscii.h"
...

try
{
    TamponAscii t(nomfichier);
    cout << "contenu du tampon : "<<endl<<t.total();
}
catch(ClubErreurs ce)
{
    cout << "Erreur : "<<ce.why()<<endl;
    ce.correction();
}
```

conseils d'utilisation spécifiques

À moins d'avoir des besoins très spécifiques, il est plus intéressant d'utiliser la classe TamponTexte qui intègre TamponAscii et plusieurs types d'itérateurs, offrant ainsi plus de fonctionnalités.

Après la modification du tampon, les itérateurs courants ne sont plus valides. Il faut les synchroniser par appel à `synchronize`.

implantation

Les attributs protégés sont décrits sommairement dans la table 60, il n'y a pas d'attribut privé.

TAB. 60: attributs protégés de la classe TamponAscii

nom	type	description
carCommentaires_	string	liste des caractères marqueurs de commentaires
separateurs_	string	liste des caractères séparateurs
interprete_	int	drapeau d'interprétation des commentaires

Les méthodes protégées sont décrites dans la table 61.

TAB. 61: TamponAscii : méthodes protégées

signature	description
int estCarGuillemet (char <i>c</i>) const	retourne vrai si le caractère concerné est un guillemet
int estCarCommentaire (char <i>c</i>) const	retourne vrai si le caractère concerné est un commentaire
int estSeparateur (char& <i>c</i>) const	retourne vrai si le caractère concerné est un séparateur
int supprimeZone (char* <i>zdebut</i> , char* <i>zfin</i>)	supprime la zone située entre les 2 pointeurs
int insereZone (char* <i>pos</i> , const char* <i>valeur</i>)	insère <i>valeur</i> à l'endroit pointé par <i>pos</i>

12.21 classe TamponPartage

description

La classe TamponPartagé permet de partager des zones de mémoire allouées dynamiquement entre plusieurs instances de plusieurs classes de telle sorte que la mémoire ne soit désallouée qu'à la destruction de la dernière instance, même si ce n'est pas elle qui avait alloué la zone à l'origine.

La classe gère un compteur du nombre de références portant sur la zone mémoire, ce compteur étant incrémenté à chaque copie et décrémenté à chaque destruction. Les classes utilisatrices peuvent donc se contenter de déclarer un attribut de type TamponPartagé et le copier directement dans leurs opérateurs d'affectation, elles n'ont même pas besoin de gérer cet attribut dans leur destructeur. En fait la seule gestion qu'elles doivent faire concerne la première allocation.

interface publique

```
#include "club/TamponPartage.h"
```

TAB. 62: TamponPartage : méthodes publiques

signature	description
TamponPartage ()	constructeur par défaut
TamponPartage (void * <i>memoire</i> , void (* <i>f</i>) (void *, void *), void * <i>arg</i> = 0)	construit une instance gérant la zone <i>memoire</i> allouée, qui sera à terme libérée par l'appel <i>f</i> (<i>memoire</i> , <i>arg</i>)
TamponPartage (const TamponPartage& <i>t</i>)	constructeur par copie
TamponPartage& operator = (const TamponPartage& <i>t</i>)	affectation
~TamponPartage ()	destructeur, décrémente le compteur de référence, et libère la mémoire lorsque le compteur arrive à zéro, par appel à la fonction enregistrée par l'utilisateur si le pointeur fourni est non nul
int references () const	retourne le nombre de références à la zone allouée
void * memoire () const	retourne un pointeur anonyme sur la mémoire allouée

exemple d'utilisation

```
#include "club/TamponPartage.h"
...
class FichierStructure
{ ...

    TamponPartage total_;
public :
    ...

    // constructeur par copie
    FichierStructure (const FichierStructure& f)
        : nomBloc_ (f.nomBloc_), nomFichier_ (f.nomFichier_),
          total_ (f.total_), debut_ (f.debut_), fin_ (f.fin_) {}

    // pas de destructeur, la mémoire est libérée par TamponPartage
};

static char* AlloueChaine (int longueur)
{ return new char [longueur + 1]; }

static void LibereChaine (void *memoire, void *)
{ delete [] ((char *) memoire); }

static char* SepareDelimiters (const string& nomFichier,
                               const char* brut)
{ ...
    char* memoire = AlloueChaine (s - brut + 2 * nombre);
    ...
    return memoire;
}

void FichierStructure::lit (TamponTexte *ptrFic)
{ ...

    // séparation des '{' et des '}'
    char* tampon = SepareDelimiters (ptrFic->nom (), ptrFic->total ());
    ...

    // mémorisation du fichier
    total_ = TamponPartage (tampon, LibereChaine, 0);
    ...
}
```

conseils d'utilisation spécifiques

La mémoire est libérée par TamponPartage par l'appel à une fonction utilisateur. Il est indispensable que l'allocation soit symétrique de la libération (`new []/delete []`, `new/delete`, `malloc/free`). Il est de plus

recommandé de convertir le pointeur `void *` vu par la classe `TamponPartage` en un pointeur typé dans la fonction de libération, pour éviter des problèmes d'implémentation dépendant du compilateur. La libération par défaut utilisant `delete []` a été supprimée à partir de la version 6.2.1 de `CLUB`.

implantation

Les attributs de la classe sont décrits sommairement dans la table 63, l'attribut `acces_` est un pointeur sur une classe contenant le pointeur alloué, cette structure `Acces` est partagée par toutes les copies de l'instance, ses composantes sont décrites sommairement dans la table 64, il n'y a pas de méthode privée.

TAB. 63: attributs de la classe `TamponPartage`

nom	type	description
<code>acces_</code>	<code>Acces *</code>	pointeur vers l'instance d'accès à la mémoire
<code>fonctionLibération_</code>	<code>void (*fonctionLibération_)</code> <code>(void *memoire, void *arg)</code>	pointeur sur la fonction utilisateur de libération de la mémoire
<code>arg_</code>	<code>void *</code>	pointeur anonyme sur les données de la fonction utilisateur

TAB. 64: attributs de la classe `TamponPartage::Acces`

nom	type	description
<code>compteur_</code>	<code>int</code>	compteur des références à la mémoire
<code>memoire_</code>	<code>void *</code>	pointeur sur la zone mémoire allouée

12.22 classe `TamponTexte`

description

Cette classe gère le modèle logique d'un texte pouvant comporter des lignes de commentaires et des champs séparés par des blancs. Un champ peut contenir des caractères spéciaux des langages `C` et `C++` (`'\n'`, `'\t'`, `'\0'`, `'\324'`, `'\x1a'`, ...), des caractères spéciaux du langage `FORTRAN` (`'\'`), ou des séparateurs si ces caractères sont *protégés* par des `"` ou des `'`. Le texte est accessible sous forme brute comme une chaîne de caractères terminée par un `'\0'`, sous forme de lignes, sous forme de champs dans des lignes, ou sous forme de champs sans tenir compte des lignes. Ces différentes vues peuvent être utilisées conjointement, en lecture et en écriture. Le texte peut également être transféré depuis ou vers un fichier du système.

L'utilisation la plus courante de cette classe est de lire un fichier dans un `TamponTexte`, d'en éliminer les commentaires et les lignes vides, puis d'en extraire les éléments intéressants champ par champ, selon la syntaxe du fichier. L'appelant n'a ainsi à se préoccuper que de la structure logique des données qu'il lit, pas de la structure physique ou des commentaires.

Un commentaire est une zone de texte qui débute par un marqueur de début de commentaire et s'étend jusqu'à la fin de la ligne. Le marqueur peut être situé n'importe où dans la ligne, y compris après du texte utile. Par défaut, seul le caractère `'#'` joue le rôle de marqueur de commentaire. Les méthodes `TamponTexte::initCarCommentaires` et `TamponTexte::ajouteCarCommentaires` (table 65, page 98)

permettent de personnaliser la liste des caractères marqueurs. La méthode `TamponTexte::elimineCommentaires` permet de supprimer les commentaires du texte.

Un champ est une zone de texte constituée de toute suite de caractères ne faisant pas partie de la liste des séparateurs¹³. Par défaut la liste des séparateurs est `"\t\n\v\f\r "` (c'est à dire les caractères reconnus par la macro-définition standard du langage `C isspace`). Les méthodes `TamponTexte::initSeparateurs` et `TamponTexte::ajouteSeparateurs` permettent de personnaliser la liste des séparateurs (on ne peut cependant pas enlever `'\n'`, ni ajouter `'\"'` ou `'\''`). La méthode `TamponTexte::elimineBlancs` permet de supprimer les séparateurs excédants en début et en fin de lignes (ainsi que les lignes entièrement constituées de séparateurs).

Il est possible d'inclure des caractères séparateurs ou des caractères spéciaux utilisant la syntaxe du langage `C` ou du langage `FORTRAN` dans un champ. Les caractères à protéger (ou une partie du champ les contenant, voire le champ complet) doivent être mis entre `"` (pour les séparateurs et les caractères du langage `C`) ou entre `'` (pour les séparateurs et les caractères du langage `FORTRAN`) et le drapeau `interprete_` (cf 13, page 102) doit être actif. Ce drapeau est initialisé par certains constructeurs et par les méthodes de lecture du texte depuis un fichier. Les `"` et les `'` peuvent être présents dans le fichier lu ou dans les chaînes passées en argument des méthodes de modification du texte (`insereXXX`, `ajouteYYY`, `remplaceZZZ`). Lorsque `interprete_` est actif, la vérification de l'équilibre des `"` et des `'` est effectuée à chaque modification du texte. Les champs sont stockés *avec les `'\"'`, les `'\''` et les `'\\'`*, l'interprétation étant faite uniquement par les méthodes qui retournent des copies de champs (`TamponTexte::champ`, `TamponTexte::champLigne`).

interface publique

```
#include "club/TamponTexte.h"
```

TAB. 65: TamponTexte : méthodes générales

signature	description
TamponTexte (int <i>interprete</i> = 1)	construit une instance vide Exceptions : <code>iterateur_invalide</code> .
TamponTexte (FILE * <i>ptrFic</i> , int <i>interprete</i> = 1)	construit une instance en lisant le fichier <i>ptrFic</i> Exceptions : <code>iterateur_invalide</code> .
TamponTexte (const char * <i>nom</i> , int <i>interprete</i> = 1)	construit une instance en ouvrant puis lisant le fichier <i>nom</i> Exceptions : <code>iterateur_invalide</code> .
TamponTexte (const TamponTexte& <i>f</i>)	constructeur par copie Exceptions : <code>iterateur_invalide</code> .
TamponTexte& operator = (const TamponTexte& <i>f</i>)	affectation Exceptions : <code>iterateur_invalide</code> .
~TamponTexte ()	destructeur, libère la mémoire allouée
void lit (FILE * <i>ptrFic</i> , int <i>interprete</i> = 1)	lit le contenu du fichier pointé par <i>ptrFic</i> Exceptions : <code>ouverture_fichier</code> , <code>iterateur_invalide</code> .
void lit (const char * <i>nom</i> , int <i>interprete</i> = 1)	associe le fichier <i>nom</i> à l'instance, l'ouvre et lit son contenu Exceptions : <code>ouverture_fichier</code> , <code>iterateur_invalide</code> .
à suivre ...	

¹³un séparateur peut cependant être inclus dans un champ s'il est entre `"` ou entre `'`

TAB. 65: TamponTexte : méthodes générales (suite)

signature	description
void ecrit (const char * <i>nom</i> = 0)	ouvre le fichier <i>nom</i> (ou le nom associé à l'instance si <i>nom</i> est nul ou vide) et stocke le texte de l'instance dedans Exceptions : ouverture_fichier.
void initCarCommentaires (const char * <i>chaine</i>)	initialise la liste des marqueurs de début de commentaires à <i>chaine</i> , ou à "#" si <i>chaine</i> est nul ou vide
void ajouteCarCommentaires (const char * <i>chaine</i>)	ajoute <i>chaine</i> à la liste des marqueurs de début de commentaires
const string& carCommentaires () const	retourne la liste des marqueurs de début de commentaires
void elimineCommentaires ()	élimine les commentaires du texte Exceptions : manque_guillemet, itérateur_invalide.
void initSeparateurs (const char * <i>chaine</i>)	initialise la liste des séparateurs de champs à <i>chaine</i> , ou à "\t\n\v\f\r " si <i>chaine</i> est nul ou vide
void ajouteSeparateurs (const char * <i>chaine</i>)	ajoute <i>chaine</i> à la liste des séparateurs de champs
const string& separateurs () const	retourne la liste des séparateurs de champs
void elimineBlancs ()	élimine les blancs de début et de fin de lignes du texte ; si une ligne est entièrement blanche, le '\n' de fin de ligne est également éliminé Exceptions : itérateur_invalide
const char * nom () const	retourne le nom du fichier associé à l'instance (ce nom peut être vide si l'instance n'a pas été créée par lecture d'un fichier de nom connu)
int estVide () const	indique si le tampon est vide
const TamponAscii* getTampon () const	retourne l'adresse du tampon

TAB. 66: TamponTexte : vision du texte sans structuration

signature	description
const char * total () const	retourne un pointeur sur le texte stocké
int longueur () const	retourne la longueur du texte (sans compter le '\0')
void insereAAdresse (char * <i>adresse</i> , const char * <i>chaine</i>)	insère <i>chaine</i> à l'adresse spécifiée Exceptions : itérateur_invalide, caractere_hors_domaine.
void insereAuDebut (const char * <i>chaine</i>)	insère <i>chaine</i> au début du texte Exceptions : itérateur_invalide, caractere_hors_domaine.
void ajouteALaFin (const char * <i>chaine</i>)	ajoute <i>chaine</i> à la fin du texte Exceptions : itérateur_invalide, caractere_hors_domaine.
void elimineTout ()	vide le tampon Exceptions : itérateur_invalide, caractere_hors_domaine.
void elimineZone (char * <i>debut</i> , char * <i>fin</i>)	élimine les caractères situés de <i>debut</i> à <i>fin</i> Exceptions : itérateur_invalide, caractere_hors_domaine.
void remplaceTout (const char * <i>chaine</i>)	remplace la totalité du texte par <i>chaine</i> Exceptions : itérateur_invalide.

TAB. 67: TamponTexte : vision du texte par lignes

signature	description
int nombreLignes () const	retourne le nombre de lignes de texte Exceptions : iterateur_invalide.
const char * ligne (int <i>l</i>)	retourne une copie de la ligne <i>l</i> Exceptions : ligne_hors_domaine, iterateur_invalide.
void insereDebutLigne (int <i>l</i> , const char * <i>chaîne</i>)	insère <i>chaîne</i> en début de ligne <i>l</i> Exceptions : ligne_hors_domaine, iterateur_invalide.
void insereFinLigne (int <i>l</i> , const char * <i>chaîne</i>)	insère <i>chaîne</i> en fin de ligne <i>l</i> Exceptions : ligne_hors_domaine, iterateur_invalide.
void elimineLigne (int <i>l</i>)	élimine la ligne <i>l</i> Exceptions : ligne_hors_domaine, iterateur_invalide.
void remplaceLigne (int <i>l</i> , const char * <i>chaîne</i>)	remplace la ligne <i>l</i> par <i>chaîne</i> Exceptions : ligne_hors_domaine, iterateur_invalide.

TAB. 68: TamponTexte : vision du texte par champs et lignes

signature	description
int nombreChampsLigne (int <i>l</i>) const	retourne le nombre de champs de la ligne <i>l</i> Exceptions : ligne_hors_domaine, iterateur_invalide.
const char * champLigne (int <i>c</i> , int <i>l</i>) const	retourne une copie du champ <i>c</i> de la ligne <i>l</i> Exceptions : ligne_hors_domaine, champ_ligne_hors_domaine, tampon_vide, champ_hors_domaine, iterateur_invalide.
void insereAvantChampLigne (int <i>c</i> , int <i>l</i> , const char * <i>chaîne</i>)	insère <i>chaîne</i> au début du champ <i>c</i> de la ligne <i>l</i> Exceptions : ligne_hors_domaine, champ_ligne_hors_domaine, tampon_vide, champ_hors_domaine, iterateur_invalide.
void ajouteApresChampLigne (int <i>c</i> , int <i>l</i> , const char * <i>chaîne</i>)	ajoute <i>chaîne</i> à la fin du champ <i>c</i> de la ligne <i>l</i> Exceptions : ligne_hors_domaine, champ_ligne_hors_domaine, tampon_vide, champ_hors_domaine, iterateur_invalide.
void elimineChampLigne (int <i>c</i> , int <i>l</i>)	élimine le champ <i>c</i> de la ligne <i>l</i> Exceptions : ligne_hors_domaine, champ_ligne_hors_domaine, tampon_vide, champ_hors_domaine, iterateur_invalide.
void remplaceChampLigne (int <i>c</i> , int <i>l</i> , const char * <i>chaîne</i>)	remplace le champ <i>c</i> de la ligne <i>l</i> par <i>chaîne</i> Exceptions : ligne_hors_domaine, champ_ligne_hors_domaine, tampon_vide, champ_hors_domaine, iterateur_invalide.

TAB. 69: TamponTexte : vision du texte par champs simples

signature	description
int nombreChamps () const	retourne le nombre total de champs
const char * champ (int <i>c</i>)	retourne le champ <i>c</i> du texte Exceptions : tampon_vide, manque_guillemet, champ_hors_domaine, iterateur_invalide.
void insereAvantChamp (int <i>c</i> , const char * <i>chaîne</i>)	insère <i>chaîne</i> au début du champ <i>c</i> Exceptions : tampon_vide, manque_guillemet, champ_hors_domaine, iterateur_invalide.
à suivre ...	

TAB. 69: TamponTexte : vision du texte par champs simples (suite)

signature	description
void ajouteApresChamp (int <i>c</i> , const char * <i>chaîne</i>)	ajoute <i>chaîne</i> à la fin du champ <i>c</i> Exceptions : tampon_vide, manque_guillemet, champ_hors_domaine, itérateur_invalide.
void elimineChamp (int <i>c</i> ,)	élimine le champ <i>c</i> Exceptions : tampon_vide, manque_guillemet, champ_hors_domaine, itérateur_invalide.
void remplaceChamp (int <i>c</i> , const char * <i>chaîne</i>)	remplace le champ <i>c</i> par <i>chaîne</i> Exceptions : tampon_vide, manque_guillemet, champ_hors_domaine, itérateur_invalide.

Les méthodes de la classe TamponTexte respectent les principes généraux suivants :

adressage : les adresses brutes (de type **char ***) sont des pointeurs dans le tampon interne de stockage, on peut les calculer en ajoutant un décalage (compris entre 0 et **longueur** ()) au pointeur de début de texte retourné par **total** () ;

numérotation : les numéros de ligne, de champs, et de champs dans les lignes commencent à 1 ;

adressage invalide : si une adresse brute située avant le début du texte est utilisée, elle est ramenée en début de texte avant de réaliser l'opération demandée ; aucune erreur n'est générée (le traitement des adresses au delà de la fin du texte est symétrique) ;

numérotation invalide : si un numéro de ligne ou de champ invalide est utilisé, une erreur est générée et l'opération demandée n'est pas effectuée ;

durée de vie des copies : les méthodes qui retournent des copies d'éléments internes du texte (champs, lignes) utilisent toutes la même zone de mémoire pour retourner ces copies, chaque appel de ce type écrase donc le résultat de l'appel précédent¹⁴ ;

réallocation globale : les fonctions modifiant le texte (à savoir **lit** et les méthodes dont le nom est de la forme **insereXXX**, **ajouteYYY**, **remplaceZZZ**) peuvent libérer le tampon global pour le réallouer selon leur besoin ; un pointeur retourné par **total** () avant une modification n'est donc plus forcément valide après une modification, il vaut mieux réappeler **total** () que conserver son résultat¹⁵.

exemple d'utilisation

```
#include "club/TamponTexte.h"
...
try
{
    TamponTexte fichier ("essai");

    fichier.elimineCommentaires ()
    fichier.elimineBlancs          ()
```

¹⁴il est même possible qu'une fonction libère la zone mémoire allouée par un appel précédent pour en réallouer une plus grande

¹⁵de plus la fonction **total** () est une méthode particulièrement peu coûteuse en temps (fonction *inline* dont la définition est { **return total_;** })

```

if (fichier.estVide ())
{ cerr << "le fichier:" << fichier.nom () << " est vide\n";
  exit (1);
}

// écriture des champs du fichier en sens inverse
for (int l = fichier.nombreLignes (); l > 0; l--)
{
  for (int c = fichier.nombreChampsLigne (l); c > 0; c--)
    cout << fichier.champLigne (c, l) << ' ';
  cout << endl;
}
}
catch (ClubErreurs ce)
{
  cout << "Fin anormale du programme "<<endl;
  exit (1);
}

```

conseils d'utilisation spécifiques

Les méthodes retournant des portions du tampon (des lignes ou des champs) utilisent toutes le même attribut de l'instance pour copier ces portions (il s'agit de l'attribut `sortie_`). Si l'utilisateur doit appeler plusieurs de ces méthodes successivement, chaque appel écrasera la valeur retournée par l'appel précédent. Il est donc très important que l'appelant stocke lui-même ces valeurs dans des variables locales s'il doit en traiter plusieurs simultanément.

implantation

La classe `TamponTexte` est basée sur une hiérarchie d'objets introduite dans la version précédente de la bibliothèque CLUB. Cette hiérarchie gère un tampon de type ascii, avec des mécanismes de déplacement implémentés par des itérateurs et une gestion d'erreurs par lancement d'exceptions. Les attributs de la classe sont des objets de la hiérarchie de gestion de tampon ascii. Ces attributs sont décrits sommairement dans la table 70 (page 102), les méthodes privées étant décrites dans la table 71 (page 103).

TAB. 70: attributs de la classe TamponTexte

nom	type	description
<code>tampon_</code>	<code>TamponAscii</code>	tampon ascii associé au texte
<code>parcoursCaractere_</code>	<code>IterateurCaractere</code>	itérateur de parcours par caractère
<code>parcoursChamp_</code>	<code>IterateurChamp</code>	itérateur de parcours par champ
<code>parcoursLigne_</code>	<code>IterateurLigne</code>	itérateur de parcours par ligne
<code>parcoursChampLigne_</code>	<code>IterateurChampLigne</code>	itérateur de parcours par ligne et par champ

TAB. 71: TamponTexte : méthodes privées

signature	description
void actualiseIterateurs ()	réinitialise les itérateurs sur le tampon ascii (méthode utilisée en cas de modification du tampon) Exceptions : itérateur_invalide.

12.23 classe Traducteur

description

Cette classe permet d'obtenir le contenu d'une chaîne de caractère dans une langue utilisateur, à partir d'une clef indépendante de la langue utilisateur. L'ensemble des couples *clef-chaîne de caractères* est lu dans un ou plusieurs fichiers de ressources (lus à travers une instance de TamponTexte). Les commentaires (qui vont du caractère '#' à la fin de la ligne) sont ignorés. Les champs sont lus un par un et associés par paire, la clef pouvant être sur les champs pairs ou sur les champs impairs (un seul choix pour toutes les paires d'un même fichier). L'exemple suivant montre à quoi peut ressembler un fichier de domaine de traduction :

```
# domaine de traduction exemple

# mots-clefs
"un"
  "one"

"deux"
  "two"

# formats
"format C avec une chaine \"%s\" et deux entiers %d %d\n"
"C format using one string and two integers : %s, %3d, %2.2d\n\n\n"
```

Traducteur permet de gérer en bloc plusieurs domaines de traduction. chaque domaine est associé à un fichier de ressources, et l'utilisateur peut incrémentalement prendre en compte de nouveaux domaines. Ces fichiers de domaines sont regroupés dans des répertoires par langue, les noms des répertoires étant conformes à la norme (*en* pour l'anglais, *fr* pour le français, ...). Les répertoires sont recherchés en suivant une liste spécifiée par variable d'environnement (voir section 5) la recherche dans un répertoire n'est entamée que si la recherche dans les répertoires précédents n'a pas abouti. La langue utilisateur est également spécifiée par variable d'environnement.

Si une traduction est impossible, la clef est renvoyée, ceci permet d'utiliser ce système comme un *plus* capable de réaliser des traductions lorsque les ressources sont disponibles, mais qui n'est pas désarmé si aucune ressource n'est disponible, dans ce cas il ne traduit pas ses résultats, mais il les produit tout de même. On peut donc sans risque utiliser cette classe pour traduire des chaînes de format C ou FORTRAN de messages d'erreurs : les messages seront toujours formatés, qu'il y ait ou qu'il n'y ait pas de base de traduction.

Enfin, des fonctions spécialisées peuvent vérifier dans le cas d'une traduction de format C ou FORTRAN si le format traduit est compatible avec le format initial (en terme de nombre, de type et d'ordre des arguments attendus). Ceci permet d'assurer qu'une fois un logiciel validé, une erreur dans un fichier de traduction modifié après compilation et tests n'engendrera pas de problème d'exécution du type violation de la mémoire.

interface publique

```
#include "club/Traducteur.h"
```

TAB. 72: Traducteur : méthodes publiques

signature	description
Traducteur ()	constructeur par défaut
Traducteur (const char * <i>langueUtilisateur</i>)	construit une instance de traducteur pour la langue utilisateur spécifiée
Traducteur (const Traducteur& <i>t</i>)	constructeur par copie
Traducteur& operator = (const Traducteur& <i>t</i>)	affectation
~Traducteur ()	destructeur, libère la mémoire allouée
void ajouterDomaine (const char * <i>domaine</i> , TamponTexte& <i>fichier</i> , int <i>cleSurChampsImpairs</i>)	ajoute dans l'instance le dictionnaire du <i>domaine</i> spécifié tel qu'il est lu dans le <i>fichier</i> , l'argument <i>cleSurChampsImpairs</i> indique dans quel élément il faut prendre la clef et dans quel élément il faut prendre la traduction
int domaineMembre (const char * <i>domaine</i>) const	indique si <i>domaine</i> est géré par l'instance
const string& langueUtilisateur () const	retourne la langue de l'utilisateur
const string& operator () (const char * <i>clef</i>) const	retourne la traduction de la <i>clef</i> (ou la <i>clef</i> elle-même si la traduction n'est pas disponible)
const string& traductionFormatC (const char * <i>format</i>) const	retourne la traduction du <i>format</i> (ou le <i>format</i> préfixé d'un message si la traduction n'est pas disponible ou est incompatible)
int traductionFormatFortran (const char * <i>format</i> , FormatFortran * <i>ptrTrad</i>) const	initialise l'instance pointée par <i>ptrTrad</i> avec la traduction du <i>format</i> (retourne un code non nul en cas de problème d'analyse du <i>format</i> de base — ce qui n'est pas un problème de traduction)

exemple d'utilisation

```
#include "club/Traducteur.h"
```

```
// définition des variables statiques utiles au système général
// (seront construites dès qu'on en aura besoin)
static Traducteur* ptrExterne = 0;
static Traducteur* ptrInterne = 0;
```

```
static void InitTraducteurs ()
{ // allocation des objets statiques
  if (ptrExterne == 0)
    ptrExterne = new Traducteur;
  if (ptrInterne == 0)
    ptrInterne = new Traducteur;
}
```

```
const char* TraduitVersExterne (const char* interne)
```

```
{ InitTraducteurs ();
    return (*ptrExterne) (interne);
}

const char* TraduitVersInterne (const char* externe)
{ InitTraducteurs ();
    return (*ptrInterne) (externe);
}

const char* TraduitFormatCVersExterne (const char* format)
{ InitTraducteurs ();
    return ptrExterne->traductionFormatC (format);
}
```

conseils d'utilisation spécifiques

Les méthodes de traduction retournent des références constantes soit sur des éléments internes des dictionnaires de traduction, soit sur des variables statiques constituées au moment de l'appel (dans les divers cas d'échecs). Il est donc possible que d'un appel à l'autre les mêmes variables statiques soient réutilisées, il faut donc que l'appelant stocke lui-même la valeur retournée s'il en a besoin un certain temps. D'autre part, il se trouve que le système de gestion des erreurs basé sur la classe BaseErreurs (voir 12.2 et 12.6) utilise Traducteur. Il faut donc faire très attention car la durée de validité de la référence retournée peut être extrêmement courte. Le code suivant est ainsi erroné :

```
Traducteur t (...);
return MaClasseErreurs (MaClasseErreurs::pb_traduction,
                        t (clef));
```

le formatage de `MaClasseErreurs::pb_traduction` utilise en effet très vraisemblablement les méthodes de BaseErreurs, lesquelles écrasent la référence constante résultant de l'analyse du second argument ! Ce cas a été effectivement rencontré, il a été difficile à cerner. Il est important de noter qu'il respecte tout à fait les règles sur la durée de vie des temporaires de la version de travail du comité de normalisation du langage C++.

implantation

Les attributs sont décrits sommairement dans la table 73, il n'y a pas de méthode privée.

TAB. 73: attributs de la classe Traducteur

nom	type	description
tableCorrespondance_	hash_map <string, OptionBase *, club_hash<string> >	dictionnaire
nbDomaines_	int	nombre de domaines gérés par l'instance
domaines_	string *	table des domaines gérés par l'instance
langueUtilisateur_	string	langue utilisateur gérée par l'instance

12.24 classe UniqDataFile

description

La classe UniqDataFile est une classe abstraite offrant des services génériques d'accès à des données situées dans différents types de fichiers. Les différents formats de fichiers supportés sont :

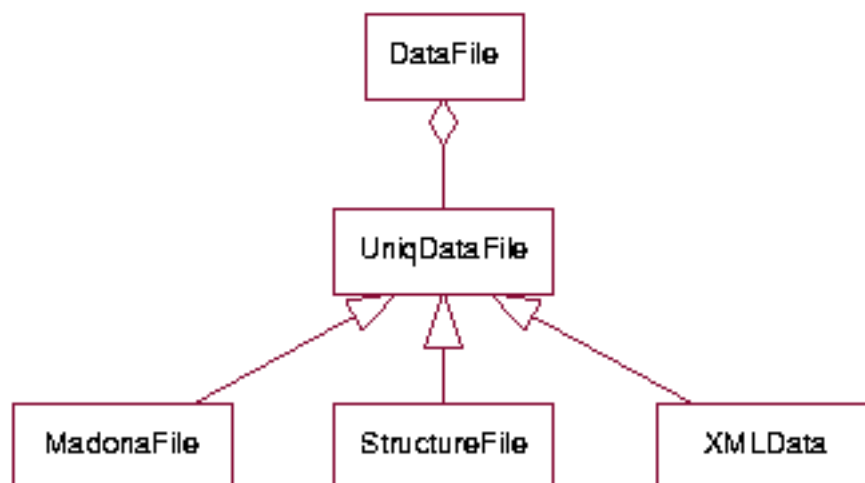
FichierStructure format des fichiers structurés spécifiques à la bibliothèque CLUB. Seuls les services de lecture sont implémentés. Les services d'écriture lèvent l'exception ClubErreurs::not_implemented,

Madona l'ensemble des services de lecture/écriture sont implémentés,

XML l'ensemble des services de lecture/écriture sont implémentés.

Cette classe regroupe tous les attributs et toutes les opérations communes à ces formats. Elle contient principalement des méthodes virtuelles qui sont implémentées dans les classes filles (cf. figure 2).

FIG. 2 – Hiérarchie d'héritage des classes de gestion des fichiers structurés



Les services non abstraits rendus par la classe UniqDataFile sont des services de gestion des désignations de données. Ces services permettent d'identifier de la même manière les données contenues dans les différents types de fichiers. La norme employée pour désigner les données est celle utilisée par le format Madona.

Exemples :

- `my_table[1].my_data` désigne la donnée «my_data» appartenant à l'élément d'index 1 d'un tableau nommé «my_table»,
- `my_structure.my_data` désigne la donnée «my_data» appartenant à la structure «my_structure».

Les catégories de services disponibles sont :

- chargement et écriture de fichiers,
- création de données,
- mise à jour des paramètres,
- lecture de données,
- destruction de données,
- gestion des unités physiques,
- déplacement dans la structure de données.

Toutes les classes filles de UniqDataFile gèrent une structure mémoire hiérarchisée et utilisent un pointeur désignant la donnée courante. La plupart des méthodes définies utilisent ce pointeur.

interface publique

```
#include "club/UniqDataFile.h"
```

TAB. 74: attributs publics de la classe UniqDataFile

nom	type	description
DataType	enum	énumérateur précisant le type de donnée : ParameterType, StructureType, TableType, ReferenceType, UndeterminedType
TableElementType	enum	énumérateur précisant le type des éléments d'une table : IntTable, RealTable, StringTable, StructureTable, TableTable

TAB. 75: UniqDataFile : méthodes publiques

signature	description
UniqDataFile ()	constructeur : initialise la donnée courante Exceptions MadonaFile : madona_error Exceptions XMLFile : xml_error
~UniqDataFile ()	détruit l'instance Exceptions MadonaFile : madona_error
const string getCurrentData () void moveDown (const string& <i>childDataName</i>)	renvoie la désignation courante déplace le pointeur de la donnée courante sur une donnée fille Exceptions MadonaFile : current_data_not_a_table, undefined_index, undefined_table_or_structure, string_to_int_conversion_error, madona_error Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_table_or_structure, string_to_int_conversion_error, xml_error, xpointer_syntax_error Exceptions StructureFile : string_to_int_conversion_error, undefined_table_or_structure
void moveDown (int <i>index</i>)	déplace le pointeur de la donnée courante sur une donnée fille, à condition que la donnée courante soit une table Exceptions MadonaFile : current_data_not_a_table, undefined_index, undefined_table_or_structure, madona_error Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_table_or_structure, xpointer_syntax_error, xml_error Exceptions StructureFile : undefined_table_or_structure
à suivre ...	

TAB. 75: UniqDataFile : méthodes publiques (suite)

signature	description
<p>void moveUp ()</p> <p>void moveTo (const string& <i>absoluteDataName</i>)</p>	<p>remonte d'un niveau dans la structure de données</p> <p>Exceptions MadonnaFile : move_up_forbidden, madona_error</p> <p>Exceptions XMLFile : move_up_forbidden, xml_error</p> <p>Exceptions StructureFile : string_to_int_conversion_error, undefined_table_or_structure, move_up_forbidden</p> <p>déplace le pointeur de la donnée courante sur une autre donnée désignée de manière absolue</p> <p>Exceptions MadonnaFile : string_to_int_conversion_error, undefined_table_or_structure, undefined_index, madona_error, current_data_not_a_table</p> <p>Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_table_or_structure, string_to_int_conversion_error, xml_error, xpointer_syntax_error</p> <p>Exceptions StructureFile : string_to_int_conversion_error, undefined_table_or_structure</p>
<p>void readFile (const string& <i>fileName</i>)</p> <p>void writeFile (const string& <i>fileName</i>)</p>	<p>charge un fichier de données en mémoire et positionne le pointeur de donnée à la racine de la structure mémoire</p> <p>Exceptions MadonnaFile : file_error</p> <p>Exceptions XMLFile : file_error, missing_tag</p> <p>Exceptions StructureFile : accolades_non_equilibrees, nom_inclusion etat_fichier, itereateur_invalide, caractere_hors_domaine, manque_guillemet, manque_chevron,</p> <p>écrit la structure de données en mémoire dans <i>fileName</i></p> <p>Exceptions MadonnaFile : file_error</p> <p>Exceptions XMLFile : ouverture_fichier</p> <p>Exceptions StructureFile : not_implemented</p>
<p>DataType getDataType (const string& <i>childDataName</i>)</p> <p>DataType getDataType (int <i>index</i>)</p>	<p>retourne le type d'une donnée fille</p> <p>Exceptions MadonnaFile : current_data_not_a_table, undefined_index, undefined_data, madona_error, string_to_int_conversion_error</p> <p>Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_data, string_to_int_conversion_error</p> <p>Exceptions StructureFile : string_to_int_conversion_error, undefined_parameter, undefined_data, undefined_index, undefined_table_or_structure</p> <p>retourne le type d'une donnée fille. La donnée courante doit être une table</p> <p>Exceptions MadonnaFile : current_data_not_a_table, madona_error, undefined_index</p> <p>Exceptions XMLFile : current_data_not_a_table, undefined_index</p> <p>Exceptions StructureFile : undefined_table_or_structure, undefined_parameter, undefined_index</p>
à suivre ...	

TAB. 75: UniqDataFile : méthodes publiques (suite)

signature	description
<p>int size ()</p> <p>bool dataExists (const string& <i>childDataName</i>)</p> <p>bool dataExists (int <i>index</i>)</p>	<p>renvoie la taille de la donnée courante qui doit être une table Exceptions MadonnaFile : current_data_not_a_table, madona_error Exceptions XMLFile : current_data_not_a_table Exceptions StructureFile : current_data_not_a_table</p> <p>vérifie l'existence d'une donnée fille Exceptions MadonnaFile : madona_error, string_to_int_conversion_error, current_data_not_a_table Exceptions XMLFile : current_data_not_a_table, string_to_int_conversion_error Exceptions StructureFile : current_data_not_a_table, string_to_int_conversion_error</p> <p>vérifie l'existence d'une donnée fille. La donnée courante doit être une table Exceptions MadonnaFile : current_data_not_a_table Exceptions XMLFile : current_data_not_a_table Exceptions StructureFile : current_data_not_a_table</p>
<p>string getComment (const string& <i>childDataName</i>)</p> <p>string getComment (int <i>index</i>)</p> <p>void setComment (const string& <i>childDataName</i>, const string& <i>comment</i>)</p> <p>void setComment (int <i>index</i>, const string& <i>comment</i>)</p>	<p>retourne le commentaire associée à une donnée fille Exceptions MadonnaFile : current_data_not_a_table, madona_error, undefined_data, string_to_int_conversion_error, not_implemented Exceptions XMLFile : undefined_data, undefined_index, current_data_not_a_table, string_to_int_conversion_error Exceptions StructureFile : not_implemented</p> <p>retourne le commentaire associée à une donnée fille. La donnée courante doit être une table Exceptions MadonnaFile : not_implemented Exceptions XMLFile : current_data_not_a_table, undefined_index Exceptions StructureFile : not_implemented</p> <p>met à jour le commentaire d'une donnée fille Exceptions MadonnaFile : current_data_not_a_table, madona_error, undefined_data, string_to_int_conversion_error Exceptions XMLFile : undefined_data, undefined_index, current_data_not_a_table, string_to_int_conversion_error Exceptions StructureFile : not_implemented</p> <p>met à jour le commentaire d'une donnée fille. La donnée courante doit être une table Exceptions MadonnaFile : not_implemented Exceptions XMLFile : current_data_not_a_table, undefined_index Exceptions StructureFile : not_implemented</p>
à suivre ...	

TAB. 75: UniqDataFile : méthodes publiques (suite)

signature	description
void createStringData (const string& <i>childDataName</i> , const string& <i>value</i>)	crée un paramètre de type chaîne de caractères Exceptions MadonnaFile : current_data_is_a_table, data_already_defined, madona_error, index_creation_forbidden, Exceptions XMLFile : current_data_is_a_table, data_already_defined, xml_error, index_creation_forbidden Exceptions StructureFile : not_implemented
void createIntData (const string& <i>childDataName</i> , int <i>value</i>)	crée un paramètre de type entier Exceptions MadonnaFile : current_data_is_a_table, data_already_defined, madona_error, index_creation_forbidden, Exceptions XMLFile : current_data_is_a_table, data_already_defined, xml_error, index_creation_forbidden Exceptions StructureFile : not_implemented
void createRealData (const string& <i>childDataName</i> , double <i>value</i> , const string& <i>unit</i>)	crée un paramètre de type réel Exceptions MadonnaFile : current_data_is_a_table, data_already_defined, madona_error, index_creation_forbidden, Exceptions XMLFile : current_data_is_a_table, data_already_defined, xml_error, index_creation_forbidden Exceptions StructureFile : not_implemented
void createTable (const string& <i>tableName</i> , UniqDataFile::TableElementType <i>tableType</i> , const string& <i>defaultUnit</i> ="")	crée une table vide Exceptions MadonnaFile : current_data_is_a_table, data_already_defined, madona_error, index_creation_forbidden, Exceptions XMLFile : current_data_is_a_table, data_already_defined, xml_error, index_creation_forbidden Exceptions StructureFile : not_implemented
void createStructure (const string& <i>structureName</i>)	crée une structure vide Exceptions MadonnaFile : current_data_is_a_table, data_already_defined, madona_error, index_creation_forbidden, Exceptions XMLFile : current_data_is_a_table, data_already_defined, xml_error, index_creation_forbidden Exceptions StructureFile : not_implemented
void createReference (const string& <i>childDataName</i> , const string& <i>referencedFileName</i> , const string& <i>referencedDataName</i>)	crée une référence à une autre donnée Exceptions MadonnaFile : current_data_is_a_table, data_already_defined, madona_error, index_creation_forbidden Exceptions XMLFile : current_data_is_a_table, data_already_defined, xml_error, index_creation_forbidden Exceptions StructureFile : not_implemented
à suivre ...	

TAB. 75: UniqDataFile : méthodes publiques (suite)

signature	description
void createStringData (const string& value)	crée dans une table un paramètre de type chaîne de caractères Exceptions MadonnaFile : current_data_not_a_table, madona_error Exceptions XMLFile : current_data_not_a_table, invalid_element_type, xml_error Exceptions StructureFile : not_implemented
void createIntData (int value)	crée dans une table un paramètre de type entier Exceptions MadonnaFile : current_data_not_a_table, madona_error Exceptions XMLFile : current_data_not_a_table, invalid_element_type, xml_error Exceptions StructureFile : not_implemented
void createRealData (double value, const string& unit)	crée dans une table un paramètre de type réel Exceptions MadonnaFile : current_data_not_a_table, madona_error Exceptions XMLFile : current_data_not_a_table, invalid_element_type, xml_error Exceptions StructureFile : not_implemented
void createTable (UniqDataFile::TableElementType tableType, const string& defaultUnit="")	crée dans une table une table vide Exceptions MadonnaFile : current_data_not_a_table, madona_error Exceptions XMLFile : current_data_not_a_table, invalid_element_type, xml_error Exceptions StructureFile : not_implemented
void createStructure ()	crée dans une table une structure vide Exceptions MadonnaFile : current_data_not_a_table, madona_error Exceptions XMLFile : current_data_not_a_table, invalid_element_type, xml_error Exceptions StructureFile : not_implemented
void createReference (const string& referencedFileName, const string& referencedDataName)	crée dans une table une référence à une autre donnée Exceptions MadonnaFile : current_data_not_a_table, madona_error Exceptions XMLFile : current_data_not_a_table, xml_error Exceptions StructureFile : not_implemented
string getStringData (const string& childDataName)	retourne une donnée fille de type chaîne de caractères Exceptions MadonnaFile : current_data_not_a_table, madona_error undefined_index, undefined_parameter, string_to_int_conversion_error Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_parameter, undefined_table_or_structure, xpointer_syntax_error, file_error, string_to_int_conversion_error Exceptions StructureFile : bloc_champ_inexistant, bloc_non_terminal, undefined_parameter, string_to_int_conversion_error
à suivre ...	

TAB. 75: UniqDataFile : méthodes publiques (suite)

signature	description
int getIntData (const string& <i>childDataName</i>)	retourne une donnée fille de type entier Exceptions MadonaFile : current_data_not_a_table, madona_error, undefined_index, undefined_parameter, string_to_int_conversion_error Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_parameter, undefined_table_or_structure, xpointer_syntax_error, file_error, string_to_int_conversion_error Exceptions StructureFile : bloc_champ_inexistant, bloc_non_terminal, undefined_parameter, string_to_int_conversion_error
double getRealData (const string& <i>childDataName</i> , const string& <i>unit</i>)	retourne une donnée fille de type réel Exceptions MadonaFile : current_data_not_a_table, madona_error, undefined_index, undefined_parameter, string_to_int_conversion_error Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_parameter, undefined_unit, undefined_table_or_structure, malformed_data, file_error, string_to_double_conversion_error, string_to_int_conversion_error, xpointer_syntax_error Exceptions StructureFile : bloc_champ_inexistant, bloc_non_terminal, undefined_parameter, string_to_int_conversion_error
string getStringData (int <i>index</i>)	retourne une donnée fille de type chaîne de caractères. La donnée courante doit être une table Exceptions MadonaFile : current_data_not_a_table, madona_error, undefined_index Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_parameter, undefined_table_or_structure, xpointer_syntax_error, file_error Exceptions StructureFile : bloc_champ_inexistant, bloc_non_terminal
int getIntData (int <i>index</i>)	retourne une donnée fille de type entier. La donnée courante doit être une table Exceptions MadonaFile : current_data_not_a_table, madona_error, undefined_index Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_parameter, undefined_table_or_structure, xpointer_syntax_error, file_error, string_to_int_conversion_error Exceptions StructureFile : string_to_int_conversion_error, bloc_non_terminal, bloc_champ_inexistant
à suivre ...	

TAB. 75: UniqDataFile : méthodes publiques (suite)

signature	description
double getRealData (int <i>index</i> , const string& <i>unit</i>)	retourne une donnée fille de type réel. La donnée courante doit être une table Exceptions MadonnaFile : current_data_not_a_table, madona_error, undefined_index Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_parameter, undefined_unit, undefined_table_or_structure, malformed_data, file_error, string_to_double_conversion_error, xpointer_syntax_error Exceptions StructureFile : string_to_double_conversion_error, bloc_champ_inexistant, bloc_non_terminal
void setStringData (const string& <i>childDataName</i> , const string& <i>value</i>)	met à jour un paramètre de type chaîne de caractères Exceptions MadonnaFile : current_data_not_a_table, undefined_index, undefined_parameter, string_to_int_conversion_error, madona_error Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_parameter, string_to_int_conversion_error Exceptions StructureFile : not_implemented
void setIntData (const string& <i>childDataName</i> , int <i>value</i>)	met à jour un paramètre de type entier Exceptions MadonnaFile : current_data_not_a_table, undefined_index, undefined_parameter, string_to_int_conversion_error, madona_error Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_parameter, string_to_int_conversion_error Exceptions StructureFile : not_implemented
void setRealData (const string& <i>childDataName</i> , double <i>value</i> , const string& <i>unit</i>)	met à jour un paramètre de type réel Exceptions MadonnaFile : current_data_not_a_table, undefined_index, undefined_parameter, string_to_int_conversion_error, madona_error Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_parameter, string_to_int_conversion_error Exceptions StructureFile : not_implemented
void setStringData (int <i>index</i> , const string& <i>value</i>)	met à jour un paramètre de type chaîne de caractères. La donnée courante doit être une table Exceptions MadonnaFile : current_data_not_a_table, madona_error, undefined_index Exceptions XMLFile : current_data_not_a_table, undefined_index Exceptions StructureFile : not_implemented
void setIntData (int <i>index</i> , int <i>value</i>)	met à jour un paramètre de type entier. La donnée courante doit être une table Exceptions MadonnaFile : current_data_not_a_table, madona_error, undefined_index Exceptions XMLFile : current_data_not_a_table, undefined_index Exceptions StructureFile : not_implemented
à suivre ...	

TAB. 75: UniqDataFile : méthodes publiques (suite)

signature	description
void setRealData (int <i>index</i> , double <i>value</i> , const string& <i>unit</i>)	met à jour un paramètre de type réel. La donnée courante doit être une table Exceptions MadonaFile : current_data_not_a_table, madona_error, undefined_index Exceptions XMLFile : current_data_not_a_table, undefined_index Exceptions StructureFile : not_implemented
void deleteData (const string& <i>dataName</i>) void deleteData (int <i>index</i>)	supprime une donnée Exceptions MadonaFile : current_data_not_a_table, undefined_index, undefined_data, string_to_int_conversion_error, madona_error, current_data_is_a_table Exceptions XMLFile : current_data_not_a_table, undefined_index, undefined_data, string_to_int_conversion_error Exceptions StructureFile : not_implemented supprime une donnée. La donnée courante doit être une table Exceptions MadonaFile : current_data_not_a_table, madona_error, undefined_index Exceptions XMLFile : current_data_not_a_table, undefined_index Exceptions StructureFile : not_implemented
void writeMemoryStructure ()	affiche sur la sortie standard la structure de donnée en mémoire Exceptions MadonaFile : madona_error Exceptions XMLFile : undefined_node_type Exceptions StructureFile : not_implemented

exemple d'utilisation

```
#include "club/UniqDataFile.h"
#include "club/XMLFile.h"

int main (int argc, char **argv)
{
    try
    {

        UniqDataFile * file = 0;

        // Par défaut on lit un fichier structuré.
        if (argc==2)
        {
            if (argv[1] == "xml")
                file = new XMLFile (false);
            else if (argv[1] == "madona")
                file = new MadonaFile ();
            else
                file = new StructureFile ();
        }
    }
}
```

```

else
    file = new StructureFile ();

file->readFile ("dataIn");

// Déplacement relatifs dans la structure de données
file->moveDown ("table");
file->moveDown (1);
intValue = file->getIntData ("param");

// Déplacement absolu
file->moveTo ("table[2]");
intValue = file->getIntData ("param");

// Création d'une donnée
file->createStringData ("stringParam", "value");

// Ecriture de la structure dans un fichier.
file->writeFile ("dataOut");
}
catch (ClubErreurs ce)
{
    cout << ce.why () << endl;
    ce.correction ();
}
}

```

conseils d'utilisation spécifiques

La classe UniqDataFile étant abstraite, elle ne peut être utilisée directement. Seules ses classes dérivées peuvent être instanciées.

Il est important de souligner que cette classe ne fournit pas de constructeur par recopie et d'opérateur d'affectation. Si ces services sont nécessaires, il faut alors utiliser la classe DataFile qui gère un tampon partagée sur une instance de UniqDataFile.

implantation

Les attributs protégés sont décrits sommairement dans la table 76, il n'y a pas d'attribut privé.

TAB. 76: attributs protégés de la classe UniqDataFile

nom	type	description
currentData_	string	la désignation de la donnée couramment pointée

Les méthodes protégées sont décrites dans la table 77.

TAB. 77: UniqDataFile : méthodes protégées

signature	description
void setCurrentData (const string& <i>currentData</i>)	met à jour la désignation courante
string getAbsoluteDataName (const string& <i>childDataName</i>)	retourne la désignation absolue d'une donnée
string getAbsoluteDataName (int <i>index</i>)	retourne la désignation absolue d'un élément de tableau. La donnée courante doit être une table
void testCurrentDataIsATable ()	lève l'exception <i>current_data_is_a_table</i> si la donnée courante est une table
void testCurrentDataNotATable ()	lève l'exception <i>current_data_not_a_table</i> si la donnée courante n'est pas une table
void testUndefinedStructureOrTable (const string& <i>childDataName</i>)	lève l'exception <i>undefined_table_or_structure</i> si <i>childDataName</i> ne représente pas une donnée de type structure ou table
void testUndefinedParameter (const string& <i>childDataName</i>)	lève l'exception <i>undefined_parameter</i> si <i>childDataName</i> ne représente pas un paramètre
void testUndefinedIndex (int <i>index</i>)	lève l'exception <i>undefined_index</i> si <i>childDataName</i> ne représente pas un index valide
void testUndefinedData (const string& <i>childDataName</i>)	lève l'exception <i>undefined_data</i> si <i>childDataName</i> ne représente aucune donnée
void testMoveUpForbidden ()	lève l'exception <i>move_up_forbidden</i> s'il n'est pas permis de remonter d'un niveau dans la structure de données
void testDataAlreadyDefined (const string& <i>childDataName</i>)	lève l'exception <i>data_already_defined</i> si <i>childDataName</i> représente une donnée déjà définie
bool canMoveUp ()	retourne true s'il est possible de remonter d'un niveau dans la structure de données, false autrement

12.25 classe Unit

12.25.1 description

Cette classe représente une unité particulière, pouvant intervenir dans des conversions de valeurs numériques entre unités.

12.25.2 interface publique

```
#include "club/Unit.h"
```

TAB. 78: Unit : méthodes publiques

signature	description
Unit ()	construit une unité par défaut sans aucune dimension
à suivre ...	

TAB. 78: Unit : méthodes publiques (suite)

signature	description
Unit (const XMLCh* <i>symbol</i> , int <i>dimLength</i> , int <i>dimMass</i> , int <i>dimTime</i> , int <i>dimElectricCurrent</i> , int <i>dimTemperature</i> , int <i>dimAmountOfSubstance</i> , int <i>dimLuminousIntensity</i> , int <i>dimPlanarAngle</i> , int <i>dimSolidAngle</i> , double <i>offset</i> , double <i>factor</i>) Unit (const XMLCh* <i>symbol</i> , const Unit& <i>unit1</i> , int <i>exponent1</i> , const Unit& <i>unit2</i> , int <i>exponent2</i>) Unit (const Unit & <i>u</i>) Unit& operator = (const Unit& <i>u</i>)	construit une unité à partir de son symbole, de ses dimensions, et des caractéristiques de conversion la reliant à l'unité par défaut ayant les mêmes dimensions construit une unité par combinaison multiplicative de deux autres unités (voir la section 12.25.4 pour des précisions sur cette combinaison) constructeur par copie affectation
const XMLCh* getSymbol () const void setSymbol (const XMLCh* <i>symbol</i>) bool isCompatibleWith (const Unit& <i>u</i>) const	retourne le symbole de l'unité modifie le symbole de l'unité indique si l'unité est compatible avec une autre, c'est-à-dire si une valeur ayant cette unité peut être convertie en une valeur ayant l'autre unité
double fromReference (double <i>value</i>) const double toReference (double <i>value</i>) const	convertit la valeur spécifiée considérée comme exprimée dans l'unité par défaut ayant les mêmes dimensions en une valeur considérée comme exprimée dans l'unité de l'instance convertit la valeur spécifiée considérée comme exprimée dans l'unité de l'instance en une valeur considérée comme exprimée dans l'unité par défaut ayant les mêmes dimensions

12.25.3 exemple d'utilisation

```
#include "club/Unit.h"

Unit m (XMLString::transcode("m"), 1, 0, 0, 0, 0, 0, 0, 0, 0, 0.0, 1.0);
Unit km (XMLString::transcode("km"), 1, 0, 0, 0, 0, 0, 0, 0, 0, 0.0, 1000.0);
Unit s (XMLString::transcode("s"), 0, 0, 1, 0, 0, 0, 0, 0, 0, 0.0, 1.0);
Unit h (XMLString::transcode("h"), 0, 0, 1, 0, 0, 0, 0, 0, 0, 0.0, 3600.0);

Unit mPerS (XMLString::transcode("m/s"), m, 1, s, -1);
Unit kmPerH (XMLString::transcode("km/h"), km, 1, h, -1);

// convert a value from km/h into m/s
cout << mPerS.fromReference (kmPerH.toReference (130.0)) << endl;
```

12.25.4 conseils d'utilisation spécifiques

La combinaison multiplicative d'unités réalisée par le constructeur prenant deux unités et les exposants associés élimine les décalages (argument *offset* dans la construction des unités élémentaires). Ce comportement *a priori* surprenant est tout à fait logique. En effet, les combinaisons multiplicatives ont une signification

relative et non absolue. À titre d'exemple, l'unité *degré Celsius par mètre* représente un gradient de température lié à une différence de température pour une différence de position. Ceci implique qu'une valeur de 1 degré Celsius par mètre est convertie en 1,8 degré Fahrenheit par mètre alors qu'une température thermodynamique de 1 degré Celsius est convertie en 33,8 degrés Fahrenheit.

Dans la pratique, même pour des unités élémentaires, il faudrait pouvoir savoir si l'on convertit des données absolues ou relatives (pour reprendre l'exemple précédent, une variation de température thermodynamique de 1 degré Celsius devrait être convertie en une variation de 1,8 degré Fahrenheit). Cette distinction n'est pas implémentée car elle ne peut être levée que par le contexte (en fait, même l'écriture physique usuelle ne permet pas de différencier une valeur relative d'une valeur absolue).

Les seules unités présentant ces problèmes sont les températures thermodynamiques (et elles présentent le problème pour toutes les conversions : Celsius, Fahrenheit et Kelvin). Le compromis adopté et qui consiste à supprimer le décalage pour les combinaisons d'unités semble raisonnable et pragmatique.

12.25.5 implantation

Les attributs privés sont décrits sommairement dans la table 79, il n'y a pas d'attribut protégé.

TAB. 79: attributs privés de la classe Unit

nom	type	description
symbol_	const XMLCh*	symbole de l'unité
dimLength_	int	dimension de l'unité selon l'axe des longueurs
dimMass_	int	dimension de l'unité selon l'axe des masses
dimTime_	int	dimension de l'unité selon l'axe des temps
dimElectricCurrent_	int	dimension de l'unité selon l'axe des courants électriques
dimTemperature_	int	dimension de l'unité selon l'axe des températures thermodynamiques
dimAmountOfSubstance_	int	dimension de l'unité selon l'axe des quantités de matière
dimLuminousIntensity_	int	dimension de l'unité selon l'axe des intensités lumineuses
dimPlanarAngle_	int	dimension de l'unité selon l'axe des angles plans
dimSolidAngle_	int	dimension de l'unité selon l'axe des angles solides
offset_	double	décalage de l'unité par rapport à l'unité de référence ($v_{\text{ref}} = v_u \times f + o$)
factor_	double	facteur multiplicatif de l'unité par rapport à l'unité de référence ($v_{\text{ref}} = v_u \times f + o$)

12.26 classe XMLData

12.26.1 description

Cette classe fournit une interface d'accès aux données contenues dans un fichier au format XML (*eXtended Markup Language*). Elle encapsule ainsi dans une interface simple et générique les appels aux services de la bibliothèque Xerces.

Cette classe dérivant de UniqDataFile, elle hérite de toutes les méthodes publiques et protégées de UniqDataFile et implémente les méthodes virtuelles pures de UniqDataFile.

Les fichiers XML qui peuvent être manipulés par les utilisateurs de la classe doivent suivre certaines conventions qui, pour XML, s'expriment dans des DTD (*Document Type Definition*). La bibliothèque CLUB est livrée avec deux DTD et un fichier d'unités. Ces trois fichiers, nommés respectivement `club-data.dtd`, `club-units.dtd` et `club-units.xml`, sont installés dans le répertoire `/usr/local/share/club` par défaut (à moins qu'un autre préfixe n'ait été choisi à la place de `/usr/local` lors de la configuration de la bibliothèque). Pour retrouver ces fichiers, la bibliothèque utilise une variable d'environnement dont le nom peut être configuré à la compilation de la bibliothèque (voir la section 5). Le nom par défaut de cette variable d'environnement est `CLUB_XMLPATH`, la valeur correspondante est interprétée comme une liste de répertoires.

Chaque instance de `XMLData` gère un DOM (*Document Object Model*) qui est une représentation hiérarchique en mémoire des données.

La gestion des unités est réalisée grâce à un fichier fournissant les informations concernant les conversions entre unités. Ce fichier est au format XML et est livré avec la bibliothèque. Chaque fichier de données peut spécifier un autre fichier d'unités qui lui soit propre en spécifiant dans son élément principal (`<club-data>`) l'attribut `units-file` qui spécifie le nom du fichier des unités, de façon à écraser la valeur spécifiée par défaut dans la DTD. Il est également possible de remplacer le fichier d'unités `club-units.xml` installé par défaut ou bien d'utiliser un autre fichier d'unités reprenant le nom par défaut mais en configurant la liste de répertoires spécifiée par la variable d'environnement décrite plus haut (`CLUB_XMLPATH` dans la configuration par défaut) de façon à ce que ce fichier soit trouvé avant le fichier par défaut ; cette façon de procéder étant cependant propice à erreurs (on a un fichier de données dépendant d'une série d'unités particulières, mais rien ne l'indique dans le fichier), elle n'est pas recommandée.

12.26.2 interface publique

```
#include "club/XMLData.h"
```

La majorité des méthodes publiques de la classe `XMLData` sont décrites dans le tableau 75 page 107.

Le tableau 80 présente les constructeurs et les méthodes publiques propres à la classe `XMLData`.

TAB. 80: `XMLData` : méthodes publiques

signature	description
<code>XMLData () throw (ClubErreurs)</code>	construit une instance vide, dissociée de tout fichier
<code>XMLData(const string& fileName) throw (ClubErreurs)</code>	construit une instance en lisant le fichier <code>fileName</code>
<code>~XMLData ()</code>	détruit une instance et libère les ressources allouées

12.26.3 exemple d'utilisation

Un exemple d'utilisation des services généraux de gestion des formats de fichiers (Madona, XML ou FichierStructure) est fourni à la page 114.

12.26.4 conseils d'utilisation spécifiques

La classe `XMLFile` ne gère pas les cas de bouclage d'inclusions de fichiers (par exemple : un fichier de donnée s'incluant lui-même) et de référence (une donnée se référant elle-même). Il est à la charge du

programmeur d'assurer l'ordonnancement des inclusions et des références.

12.26.5 implantation

Il n'y a pas d'attribut protégé, les attributs privés sont décrits dans la table 81. Les méthodes protégées sont décrites dans la table 82. Les méthodes privées sont décrites dans la table 83.

TAB. 81: attributs privés de la classe XMLData

nom	type	description
units_	const XMLUnits *	pointeur vers le fichier des unités
currentElement_	DOMElement *	élément courant

TAB. 82: XMLData : méthodes protégées

signature	description
bool ignoreNode (const DOMNode* <i>node</i>) const	indique que le nœud spécifié doit être ignoré lors de l'écriture de l'arbre DOM dans un fichier

TAB. 83: XMLData : méthodes privées

signature	description
XMLData (const XMLData & <i>other</i>) XMLData & operator = (const XMLData & <i>other</i>) void includeFile (const string& <i>fileName</i>) throw (ClubErreurs) void deleteIncludedFile (const string& <i>fileName</i>) throw (ClubErreurs) void writeReferencedFiles (DOMElement* <i>element</i>) throw (ClubErreurs) void createNewDOM () throw (ClubErreurs) void getReference (const string& <i>childDataName</i> , string* <i>referencedFileName</i> , string* <i>referencedDataName</i>) throw (ClubErreurs) void getReference (int <i>index</i> , string* <i>referencedFileName</i> , string* <i>referencedDataName</i>) throw (ClubErreurs) UniqDataFile::DataType getElementType (const DOMElement* <i>element</i>) void testCurrentDataNotATable () throw (ClubErreurs)	constructeur par copie, déclaré privé pour éviter tout appel affectation, déclarée privée pour éviter tout appel inclut le fichier <i>fileName</i> dans le DOM, au niveau de l'élément racine supprime du DOM le fichier inclus nommé <i>fileName</i> écrit tous les fichiers référencés dans le sous-arbre dont la racine est l'élément spécifié crée un arbre DOM ne contenant aucune donnée (mais contenant l'élément racine) lit une donnée fille de type référence : retourne le nom du fichier et le nom de la donnée référencée lit une donnée fille de type référence : retourne le nom du fichier et le nom de la donnée référencée. La donnée courante doit être une table retourne le type de l'élément lève l'exception <code>current_data_not_a_table</code> si la donnée courante n'est pas une table
à suivre ...	

TAB. 83: XMLData : méthodes privées (suite)

signature	description
void testCurrentDataIsATable () throw (ClubErreurs)	lève l'exception <code>current_data_is_a_table</code> si la donnée courante est une table
bool canMoveUp ()	retourne <code>true</code> s'il est possible de remonter d'un niveau dans la structure de données, <code>false</code> autrement

12.27 classe XMLFile

12.27.1 description

Cette classe implémente les fonctions élémentaires nécessaires à l'utilisation de fichiers XML, à l'exception des fonctions dépendant de la DTD.

12.27.2 interface publique

Cette classe est destinée à être spécialisée par des classes propres à chaque DTD devant être supportée, elle n'offre donc aucun service public, toutes les méthodes étant protégées ou privées.

12.27.3 exemple d'utilisation

Cette classe est destinée à être spécialisée par des classes propres à chaque DTD devant être supportée, elle n'offre donc aucun service public, il n'y a donc aucun exemple d'utilisation.

12.27.4 conseils d'utilisation spécifiques

Cette classe est destinée à être spécialisée par des classes propres à chaque DTD devant être supportée, elle n'offre donc aucun service public, il n'y a donc aucun conseil d'utilisation spécifique.

12.27.5 implantation

Les attributs protégés sont décrits sommairement dans la table 84, les attributs privés étant quant à eux dans la table 85.

TAB. 84: attributs protégés de la classe XMLFile

nom	type	description
<code>document_</code>	<code>DOMDocument*</code>	pointeur vers le document XML

TAB. 85: attributs privés de la classe XMLFile

nom	type	description
showing_	unsigned long	indicateur général des types d'éléments qui doivent être filtrés lors de l'écriture sur fichier

Les méthodes protégées sont décrites sommairement dans la table 86, les méthodes privées étant quant à elles dans la table 87.

TAB. 86: XMLFile : méthodes protégées

signature	description
XMLFile (const XMLCh* <i>type</i> , const XMLCh* <i>pubID</i> , const XMLCh* <i>sysId</i>) throw (ClubErreurs)	construit une instance avec un arbre DOM limité à l'élément racine
XMLFile (const string& <i>fileName</i>) throw (ClubErreurs)	construit une instance par lecture d'un fichier
~XMLFile ()	détruit l'instance
string toString (const XMLCh* <i>s</i>) const	convertit une chaîne de caractères XML en chaîne de la STL
bool hasChildElements (const DOMElement* <i>element</i>) const	indique si un élément possède des éléments fils
DOMElement* getChildElement (const XMLCh* <i>elementName</i> , const DOMNode* <i>node</i>) const	retourne l'élément fils du nœud spécifié dont le nom correspond à l'argument <i>elementName</i>
DOMElement* getChildElement (const XMLCh* <i>attributeName</i> , const XMLCh* <i>attributeValue</i> , const DOMNode* <i>node</i>) const	retourne l'élément fils du nœud spécifié pour lequel l'attribut <i>attributeName</i> prend la valeur <i>attributeValue</i>
DOMElement* getChildElement (const XMLCh* <i>attributeName</i> , const string& <i>attributeValue</i> , const DOMNode* <i>node</i>) const	retourne l'élément fils du nœud spécifié pour lequel l'attribut <i>attributeName</i> prend la valeur <i>attributeValue</i>
string getString (const DOMElement* <i>element</i> , const XMLCh* <i>attributeName</i>) const throw (ClubErreurs)	retourne la valeur d'un attribut d'élément sous forme d'une chaîne de caractères
int getInt (const DOMElement* <i>element</i> , const XMLCh* <i>attributeName</i>) const throw (ClubErreurs)	retourne la valeur d'un attribut d'élément sous forme d'un entier
double getDouble (const DOMElement* <i>element</i> , const XMLCh* <i>attributeName</i>) const throw (ClubErreurs)	retourne la valeur d'un attribut d'élément sous forme d'un réel
static string findFile (const string& <i>name</i> , const string& <i>baseURI</i>) throw (ClubErreurs)	retourne le nom complet d'un fichier dont on spécifie la partie finale du nom par l'argument <i>name</i> et le chemin de base de recherche pour la résolution des chemins relatifs par l'argument <i>baseURI</i>
void readFile (const string& <i>fileName</i>) throw (ClubErreurs)	initialise l'arbre DOM par lecture d'un fichier
à suivre ...	

TAB. 86: XMLFile : méthodes protégées (suite)

signature	description
void write (XMLFormatTarget *target) throw (ClubErreurs)	écrit l'arbre DOM sur la cible spécifiée
bool ignoreNode (const DOMNode* node) const	indique si le nœud spécifié doit être ignoré lors de l'écriture de l'arbre DOM dans un fichier

TAB. 87: XMLFile : méthodes privées

signature	description
XMLFile (const XMLFile & other)	constructeur par copie, déclaré privé pour éviter tout appel
XMLFile & operator = (const XMLFile & other)	affectation, déclarée privée pour éviter tout appel
void handleError (const DOMException& domError) throw (ClubErreurs)	fonction appelée par l'analyseur XML lorsqu'il rencontre une erreur (fonction spécifiée par la classe de base DOMErrorHandler)
DOMInputSource* resolveEntity (const XMLCh *const publicId, const XMLCh *const systemId, const XMLCh *const baseURI)	retourne un accès à une entité définie par ses identificateurs public et privé, en utilisant la liste de répertoires spécifiée par la variable d'environnement dédiée (dont le nom par défaut est MRC_USER_XMLPATH)
short acceptNode (const DOMNode* node) const	indique si un nœud doit être accepté ou refusé lors du filtrage des nœuds à écrire dans un fichier (fonction spécifiée par la classe de base DOMNodeFilter)
unsigned long getWhatToShow () const	retourne le code des nœuds à filtrer par défaut (fonction spécifiée par la classe de base DOMNodeFilter)
void setWhatToShow (unsigned longshowing)	enregistre le code des nœuds à filtrer par défaut (fonction spécifiée par la classe de base DOMNodeFilter)

12.28 classe XMLUnits

12.28.1 description

Cette classe contient l'ensemble des unités définies dans un fichier XML respectant la DTD club-units.

12.28.2 interface publique

```
#include "club/XMLUnits.h"
```

TAB. 88: XMLUnits : méthodes publiques

signature	description
XMLUnits (const string& fileName) throw (ClubErreurs)	construit une instance par chargement d'un fichier d'unités
~ XMLUnits ()	détruit l'instance
à suivre ...	

TAB. 88: XMLUnits : méthodes publiques (suite)

signature	description
double convert (double <i>value</i> , const XMLCh* <i>fromUnit</i> , const XMLCh* <i>toUnit</i>) const throw (ClubErreurs)	convertit la valeur spécifiée entre depuis l'unité <i>fromUnit</i> vers l'unité <i>toUnit</i>

12.28.3 exemple d'utilisation

```
#include "club/XMLUnits.h"

// this defines the units having "m", "km", "h", and "s" symbols
XMLUnits units ("/usr/local/share/club/club-units.xml");

// perform a conversion, computing the combined units on the fly
cout << units.convert (130.0,
                        XMLString::transcode ("km/h"),
                        XMLString::transcode ("m/s"));
```

12.28.4 conseils d'utilisation spécifiques

L'instance est chargée au préalable avec les unités élémentaires définies dans le fichier, mais elle mémorise également toutes les unités combinées pour lesquelles on lui demande d'effectuer des conversions. La syntaxe simple utilisée pour les combinaisons d'unités est identique à la syntaxe définie par la bibliothèque MADONA. Elle est rappelée à la section 15.4, page 142.

12.28.5 implantation

Les attributs privés sont décrits sommairement dans la table 89, il n'y a pas d'attribut protégé.

TAB. 89: attributs privés de la classe XMLUnits

nom	type	description
units_	mutable UnitsMap	carte des unités

Les méthodes protégées sont décrites sommairement dans la table 90, les méthodes privées étant quant à elles dans la table 91.

TAB. 90: XMLUnits : méthodes protégées

signature	description
string toString (const XMLCh* s) const	convertit une chaîne de caractères XML en chaîne de la STL, en traitant de façon spécifique le caractère Unicode Ω (ohm, ayant pour code 0x2126), de façon à ce qu'il soit représenté par l'appel d'entité prédéfinie Ω

TAB. 91: XMLUnits : méthodes privées

signature	description
XMLUnits (const XMLUnits & other)	constructeur par copie, déclaré privé pour éviter tout appel
XMLUnits & operator = (const XMLUnits & other)	affectation, déclarée privée pour éviter tout appel
XMLCh parseHexaChar (const XMLCh* s, int start, int end) const throw (ClubErreurs)	retourne le caractère Unicode correspondant à l'appel d'entité caractère hexadécimal situé entre les indices spécifiés
XMLCh parseDeciChar (const XMLCh* s, int start, int end) const throw (ClubErreurs)	retourne le caractère Unicode correspondant à l'appel d'entité caractère décimal situé entre les indices spécifiés
XMLCh parsePredefinedEntityChar (const XMLCh* s, int start, int end) const throw (ClubErreurs)	retourne le caractère Unicode correspondant à l'appel d'entité générale prédéfini situé entre les indices spécifiés (les entités prédéfinies sont deg , micro et ohm)
void removeCharReferences (XMLCh* s) const throw (ClubErreurs)	remplace dans la chaîne spécifiée tous les caractères spéciaux représentés par des appels à des entités caractères ou des entités générales prédéfinies par le caractère Unicode correspondant
void cacheSymbol (const XMLCh* symbol) const throw (ClubErreurs)	analyse selon la syntaxe de combinaison d'unité décrite à la section 12.28.4 et mémorise le symbole spécifié

13 description des utilitaires

Cette section décrit les utilitaires fournis par la bibliothèque CLUB. Cette description concerne l'utilisation des utilitaires et les principes généraux auxquels ils répondent.

13.1 difference

description générale

L'utilitaire **difference** permet de comparer deux fichiers (un fichier par rapport à sa référence) et d'afficher leurs différences.

La notion de différence utilisée par cet utilitaire est plus large que celle du diff Unix en ce sens que deux valeurs numériques sont considérées égales ou différentes en fonction d'un critère numérique basé sur un seuil

paramétrable dans la ligne de commande.

Afin de permettre l'utilisation de l'outil sans qu'il connaisse *a priori* les ordres de grandeurs des nombres utilisés, un compromis a été adopté pour la définition du critère numérique. Utiliser des écarts absolus $|x_{\text{res}} - x_{\text{ref}}|$ n'est valable que pour des nombres proches de zéro (ou pire, exactement nuls) et pour lesquels le seuil a une signification physique précise, et dépendante de l'unité des nombres comparés, ce qui n'est pas tolérable pour un outil qui doit être capable de traiter des fichiers comportant des milliers de nombres ayant des significations complètement différentes les uns des autres. Utiliser des écarts relatifs $|x_{\text{res}} - x_{\text{ref}}|/|x_{\text{ref}}|$ n'est valable que lorsque le dernier chiffre significatif de ces nombres commence à grandir sérieusement.

Le compromis utilisé est assez classique et consiste à utiliser une formule hybride :

$$\varepsilon = \frac{|x_{\text{res}} - x_{\text{ref}}|}{1 + |x_{\text{ref}}|}$$

Cette formule se comporte comme une différence absolue lorsque x_{ref} est négligeable devant 1, et comme une différence relative quand 1 est négligeable devant x_{ref} tout en offrant un comportement continu. Cette formule a un comportement qui peut surprendre lorsque ni 1 ni x_{ref} ne peuvent être considérés comme négligeables devant l'autre. Ainsi, si l'on compare un nombre $x_{\text{res}} = 0.008895705529$ à une valeur de référence $x_{\text{ref}} = 0.008895705468$, on obtient une erreur $\varepsilon = 6.046 \times 10^{-11}$ légèrement inférieure à la différence absolue 6.1×10^{-11} et inférieure de deux ordres de grandeur par rapport à la différence relative 6.857×10^{-9} . Ce comportement est connu et considéré comme normal : l'outil a pour vocation d'être une aide à la comparaison et à la détection d'écarts importants. S'il détecte des écarts inattendus, ceux-ci doivent être analysés finement et individuellement. La valeur produite par l'outil doit donc plus être considérée comme un indicateur de type *trop grand* ou *suffisamment petit*.

Difference fournit des statistiques sur les différences numériques rencontrées et offre des options d'utilisation qui se veulent plus proches des besoins exprimés par les utilisateurs.

L'utilitaire **difference** offre des fonctionnalités d'analyse fortement inspirées du **diff unix**. Notamment, l'utilitaire est capable d'identifier les lignes propres à un seul des deux fichiers puis de les sauter pour recalculer l'analyse sur des lignes communes. L'algorithme utilisé pour réaliser cette fonctionnalité est nommé *Longest Common Subsequence* disponible au 24/07/2000 à l'adresse :

<http://www.ics.uci.edu/~eppstein/161/960229.html>

Néanmoins une limitation a été introduite qui concerne le nombre maximal de lignes que l'on peut sauter pour se recalculer. En théorie, ce nombre peut être infini mais en pratique, ceci impose, pour de très grands fichiers, une place mémoire trop importante et un temps d'exécution trop long (voir remarque sur l'option **-shift** dans la section 18 page 129). Ce nombre a donc une borne supérieure finie.

Il est, enfin, important de noter que l'utilitaire est capable de comparer *intelligemment* deux lignes différentes par la présence d'un lexème en plus dans une des deux lignes. C'est à dire que le lexème en plus sera remarqué par l'analyseur et les autres lexèmes seront analysés deux à deux comme il faut. Ceci permet notamment de ne pas altérer les statistiques dans ce cas-là.

Par exemple, la comparaison de « 3 30.2 45.3 4054.122 » et de « 30.2 45.3 4054.122 » entraînera simplement la formation d'un message précisant que « 3 » est en plus dans la première chaîne et aucune différence numérique ne sera relevée.

Néanmoins, l'algorithme qui permet de se recalculer à l'intérieur d'une ligne ne garantit pas que l'analyse trouve la plus longue sous chaîne commune pour des cas complexes¹⁶ même s'il est toujours capable de

¹⁶on appelle *cas complexe*, un cas pour lequel plusieurs repositionnements sont possibles et où le choix ne peut donc être effectué qu'en connaissant auparavant la plus longue sous chaîne commune

trouver une sous chaîne commune. En pratique, cette restriction ne devrait pas être contraignante car pour des cas simples¹⁷, la plus longue sous chaîne commune sera toujours identifiée.

ligne de commande et options

La ligne de commande de l'utilitaire **difference** comporte un certain nombre d'arguments obligatoires et d'autres optionnels. Cette partie s'attache à décrire ces arguments et leurs éventuels effets sur l'analyse.

La ligne de commande a la forme suivante¹⁸ :

```
difference [-shift n] [-rfres n] [-rlres n] [-rfref n] [-rlref n]
          [-rsynth] [-rmess] [-p f.ff] [-e n] [-ee n n]
          -ref xxxx -res xxxx
```

La table 92 décrit plus précisément ces arguments :

TAB. 92: difference : arguments de la ligne de commande

nom	type	valeurs			description
		défaut	min	max	
<i>res</i>	chaîne				nom du premier fichier
<i>ref</i>	chaîne				nom du deuxième fichier utilisé comme référence
<i>shift</i>	entier	1000	50	5000	nombre maximal de lignes pouvant être sautées pour se recaler sur deux lignes identiques
<i>rfres</i> ¹⁹	entier	0			nombre de lignes au début du fichier donné par l'option -res à ignorer dans l'analyse
<i>rlres</i> ²⁰	entier	0			nombre de lignes à la fin du fichier donné par l'option -res à ignorer dans l'analyse
<i>rfref</i> ²¹	entier	0			nombre de lignes au début du fichier donné par l'option -ref à ignorer dans l'analyse
<i>rlref</i> ²²	entier	0			nombre de lignes à la fin du fichier donné par l'option -ref à ignorer dans l'analyse
<i>rsynth</i> ²³					suppression de l'affichage de la synthèse
<i>rmess</i> ²⁴					suppression de l'affichage des messages
<i>p</i> ²⁵	réel	1e-5	0		valeur maximale autorisée pour la différence relative entre deux valeurs (cette différence est calculée par la formule $ x_{\text{res}} - x_{\text{ref}} /(1 + x_{\text{ref}})$)
<i>e</i>	entier				nombre maximal de messages de différence à afficher (les messages de 1 à la valeur fournie sont affichés)
<i>ee</i>	entiers				intervalle de valeurs des numéros de messages affichés

¹⁷on appelle *cas simple*, un cas où une seule possibilité de repositionnement existe

¹⁸les arguments entre crochets sont optionnels

¹⁹Remove First lines of file RES

²⁰Remove Last lines of file RES

²¹Remove First lines of file REF

²²Remove Last lines of file REF

²³Remove SYNTHesis

²⁴Remove MESSages

²⁵Precision

descriptions des sorties

L'utilitaire fournit trois types de sortie : les messages de différence, la synthèse et le code de retour. Les messages de différence sont affichés sur la sortie en erreur alors que la synthèse est affichée sur la sortie standard. L'utilisateur peut supprimer l'affichage de l'un ou l'autre à l'aide des options de la ligne de commande `-rsynth` et `-rmess`.

Ces trois types de sortie ont des utilités différentes et permettent à l'utilisateur de se plonger plus ou moins profondément dans l'analyse des différences des deux fichiers. Le code de retour permet de savoir si oui ou non, les fichiers sont identiques. La synthèse permet de connaître le nombre de différences numériques et lexicales et d'obtenir quelques statistiques sur les différences numériques. Les messages décrivent les erreurs rencontrées tout en indiquant où se trouvent ces erreurs dans les fichiers.

Les valeurs du code de retour de l'utilitaire **difference** suit la convention du **diff unix** et prend donc une des valeurs suivantes :

- 0 : aucune différence entre les deux fichiers,
- 1 : les deux fichiers ont au moins une différence entre eux,
- 2 : un problème est survenu lors de l'analyse des fichiers (par exemple : les fichiers fournis dans la ligne de commande sont inexistantes).

La synthèse permet à l'utilisateur d'obtenir toutes les informations pertinentes sur l'analyse des deux fichiers. Elle lui fournit les informations suivantes :

- le nombre total de messages de différence que l'utilitaire a formaté lors de la comparaison des deux fichiers,
- les numéros des messages qui ont été affichés sur la sortie en erreur,
- le nombre de différences entre les deux fichiers en précisant le nombre d'erreurs d'ordre numérique et le nombre d'erreurs d'ordre lexical,
- le nombre de différences numériques qui sont inférieures au seuil,
- des statistiques sur les différences numériques (minimum, maximum, moyenne).

Exemple :

```
nombre de messages de difference      : 4
numeros des messages affiches        : 1 a 4
nombre d'erreurs non acceptees       : 4
    2 erreur(s) numerique(s)
    2 erreur(s) lexicales(s)
nombre d'erreurs numeriques acceptees : 3
seuil de tolerance numerique         : 1.000000e-05
erreur numerique min                 : 1.485399e-08
erreur numerique max                 : 4.805970e-03
moyenne des erreurs numeriques       : 1.030984e-03
```

Les messages de différence ont été formatés dans le but de fournir une information synthétique et complète. À cet effet, ont été définies les règles suivantes pour la formation des messages :

- si deux lignes contiennent plusieurs différences, un seul message est formaté et il indiquera le nombre d'erreurs rencontrées,

- s'il y a un décalage de plusieurs lignes entre le fichier résultat et le fichier référence, un seul message sera formaté et il indiquera le groupe de lignes en plus ou qui manquent dans le fichier résultat. Une seule erreur sera comptabilisée pour l'ensemble des lignes en plus.

Exemples :

les lignes suivantes manquent dans res,1:

ref,1 : solution 1: 0.0000

ref,2 : solution 2: 3.4634

ref,3 : solution 3: 5.7322

ref,4 :

la chaine "et 1 bis " manque :

res,1 : solution 1: -5.7322

ref,5 : solution 1 et 1bis: -5.7322

les lignes suivantes contiennent 2 differences :

res,12 : nombre d'appels a la fonction f: 347

ref,16 : nombre d'appels a la fonction f: 342, g: 321

conseils d'utilisation

Avant tout, il est important de préciser que l'utilitaire **difference** a des objectifs certes proches de ceux du **diff unix** mais qui ne peuvent pas être réellement comparés. En effet, le **diff unix** ne se plongeant pas dans l'analyse lexicale des lignes, est plus performant sur le temps d'exécution et sur la place mémoire utilisée et il permet de reconnaître la plus longue séquence de lignes communes quels que soient les fichiers. En revanche, parce qu'il rentre dans l'analyse des lexèmes d'une ligne, **difference** permet d'autoriser des imprécisions numériques et peut gérer des statistiques sur les différences. Mais cette faculté a un coût qui est surtout un temps d'exécution plus important. Ainsi l'analyse de la plus longue séquence de lignes communes qui est coûteuse en temps a dû être simplifiée pour préserver des performances intéressantes.

L'utilitaire **difference** ne doit donc pas être utilisé dans la même optique que le **diff unix** (auquel cas l'utilisateur sera quelque peu déçu par les performances). Son utilisation doit être orientée vers l'analyse plus profonde des différences entre les deux fichiers afin notamment d'identifier et éventuellement d'ignorer les différences numériques négligeables.

Plusieurs contextes d'utilisation sont possibles :

- utilisation de **difference** dans un script shell,
- utilisation de **difference** pour des tests automatiques de bibliothèques pouvant être installées sur plusieurs plates-formes,
- toute autre utilisation.

Suivant l'utilisation de **difference**, certaines sorties de l'utilitaire devront être ignorées. Par exemple, pour une utilisation dans un script shell, on peut n'être intéressé que par le code de retour. Les options **-rmess** et **-rsynth** permettent de supprimer respectivement les messages et la synthèse.

Les options **-rfres**, **-rfref**, **-rlres** et **-rlref** permettent de filtrer les premières et dernières lignes du fichier résultat ou de référence (ces lignes ne seront donc pas analysées et aucun homologue ne leur sera cherché). Ceci est intéressant pour sauter un en-tête ou un pied de fichier.

Pour analyser efficacement les différences entre deux fichiers, il est important, en premier lieu, d'affecter (par l'option **-p**) une valeur significative au seuil maximal accepté pour les différences numériques relatives²⁶. Ensuite, il peut être intéressant de réaliser l'analyse en plusieurs étapes :

1. analyse avec l'option **-rmess** afin de n'obtenir que la synthèse qui fournit notamment le nombre de différences numériques et lexicales, le nombre de messages formatés par l'analyse ainsi que les statistiques sur les différences numériques qui permettront éventuellement de modifier la valeur du seuil par l'option **-p**,
2. s'il y a des erreurs à inspecter plus profondément et si elles sont nombreuses, l'affichage des messages fournit un complément d'information intéressant et l'option **-e** ou **-ee** permet de sélectionner les numéros des messages affichés afin de ne pas avoir un flot immense à l'écran.

Enfin, le dernier conseil d'utilisation concerne l'option **-shift** qui permet de définir le nombre maximal de lignes que l'on s'autorise à sauter pour repositionner l'analyseur sur deux lignes égales. Lorsqu'au cours de l'analyse, la valeur du décalage dépasse le seuil toléré, l'analyse s'interrompt après l'affichage d'un message d'explication.

La recherche des lignes communes se réalise par l'algorithme nommé *Longest Common Subsequence* qui nécessite d'analyser l'ensemble des lignes pour trouver la meilleure solution. La complexité de l'algorithme est en $O(m*n)$ et il nécessite une place mémoire égale à $n*m$ entiers (n = nombre de lignes dans le premier fichier et m = nombre de lignes dans le second). En pratique, dans le cas de fichiers assez longs, cet algorithme n'est pas réalisable pour des raisons évidentes de performances. Un compromis doit donc être fait entre les performances et la pertinence du résultat de la recherche (qui dépend directement du nombre de lignes analysés ensemble).

Ainsi :

- si la valeur fournie par l'option est petite, l'analyse sera rapide et peu coûteuse en place mémoire mais la plus longue sous-séquence commune aura moins de chance d'être trouvée,
- à l'inverse, si elle est grande, l'analyse sera longue et coûteuse en place mémoire mais l'utilisateur aura plus de chance de trouver la plus longue sous-séquence commune.

La valeur par défaut de l'option (1000) est un bon compromis entre ces deux aspects.

13.2 club-config

description générale

L'utilitaire **club-config** permet de déterminer les options nécessaires à la compilation d'applicatifs ou de bibliothèques s'appuyant sur CLUB. Il est principalement destiné à être utilisé dans les règles des fichiers de directives du type **Makefile**.

ligne de commande et options

La ligne de commande a la forme suivante :

```
club-config [--cppflags | --ldflags | --libs | --version]
```

²⁶cette valeur dépendant fortement du contexte d'utilisation, la valeur par défaut (1e-5) ne peut pas convenir à tous les contextes

Si aucune des options `--cppflags`, `--ldflags`, `--libs` ou `--version` n'est utilisée, l'utilitaire indique les options disponibles et s'arrête avec un code d'erreur.

descriptions des sorties

Les sorties de l'utilitaire dépendent des options sélectionnées dans la ligne de commande.

L'option `--cppflags` permet d'obtenir sur la sortie standard les options de compilation, comme dans l'exemple suivant :

```
(lehrin) luc% club-config --cppflags
-I/home/luc/include
(lehrin) luc%
```

L'option `--ldflags` permet d'obtenir sur la sortie standard les options d'édition de liens, comme dans l'exemple suivant :

```
(lehrin) luc% club-config --ldflags
-L/home/luc/lib
(lehrin) luc%
```

L'option `--libs` permet d'obtenir sur la sortie standard les bibliothèques nécessaires à l'édition de liens, comme dans l'exemple suivant :

```
(lehrin) luc% club-config --libs
-lclub -lxcrcs-c
(lehrin) luc%
```

L'option `--version` permet d'obtenir sur la sortie standard le numéro de version de la bibliothèque, comme dans l'exemple suivant :

```
(lehrin) luc% club-config --version
8.2
(lehrin) luc%
```

conseils d'utilisation

L'utilisation classique de `--cppflags` est dans une règle de compilation de fichier **Makefile** du type :

```
client.o : client.cc
    $(CXX) 'club-config --cppflags' $(CPPFLAGS) $(CXXFLAGS) -c client.cc
```

L'utilisation classique de `--ldflags` et `--libs` est dans une règle d'édition de liens de fichier **Makefile** du type :

```
client : client.o
    $(CXX) -o $@ client.o \
        'club-config --ldflags' $(LDFLAGS) \
        'club-config --libs' $(LIBS)
```

Il est possible de combiner les options `--cppflags`, `--ldflags` et `--libs` dans une règle de fichier **Makefile** du type :

```
client : client.cc
    $(CXX) -o $@ 'club-config --cppflags --ldflags --libs' \
        $(CPPFLAGS) $(CXXFLAGS) $(LDFLAGS) $(LIBS) client.cc
```

14 description des routines

Les seules fonctions externes disponibles dans la bibliothèque sont les interfaces C et FORTRAN du système de traduction (voir section 12.23).

Ces routines agissent sur deux instances particulières de la classe Traducteur qui sont allouées dès que l'on a besoin d'elles et restent en place après. L'une de ces instances gère les traductions de la langue interne vers la langue utilisateur définie par variable d'environnement (voir section 5), l'autre gère les traductions en sens inverse.

Ces instances sont dans la pratique des pointeurs statiques dont la portée est limitée au fichier de définition des fonctions d'interface C et FORTRAN ; elles ne font pas partie de l'interface publique de la bibliothèque.

14.1 guide d'utilisation

Si l'on désire utiliser ce système pour un nouveau développement ou pour adapter un logiciel existant, la démarche à suivre est la suivante :

- pour chaque chaîne de caractères utilisée dans les entrées sorties du programme, faire précéder l'utilisation de la chaîne par sa traduction et utiliser la chaîne traduite. Si de plus la chaîne est utilisée dans une boucle (par exemple un format d'impression), alors il est prudent d'extraire la traduction hors de la boucle, pour des questions de performances.

Ainsi, un code qui aurait été écrit :

```
déclarer chaîne
initialiser chaîne
utiliser chaîne
```

deviendra :

```
déclarer chaîne1, chaîne2
initialiser chaîne1
chaîne2 = traduction(chaîne1)
utiliser chaîne2
```

- en début de programme, ajouter une fonction d'initialisation du domaine.

Si l'application est une bibliothèque, la fonction d'ajout de domaine sera placée en début de toutes les fonctions susceptibles d'être appelées en premier par l'extérieur, et protégée de la sorte qu'elle ne s'exécute qu'une fois, par un test du type :

```
déclarer entier statique Déjà=0
si (Déjà=0) faire
    ajouterDomaine("xxx")
    Déjà=1
fin si
```

- en fin de développement (voire longtemps après), on crée le fichier de ressources du domaine regroupant toutes les chaînes et leurs traductions, et ce pour chaque langue que l'on désire implanter.

Cette étape peut également être faite par l'utilisateur qui désire implémenter une nouvelle langue (ou changer les messages pour une traduction qu'il trouve plus explicite). Ceci signifie donc que le système n'est pas seulement un système de développement mais également un environnement d'exécution (avec l'avantage que même s'il est vide, le programme tourne).

14.2 fonctions C

```
#include "club/Traducteur.h"
```

TAB. 93: fonctions C

signature	description
AjouterDomaine (const char * <i>domaine</i>)	ajoute le <i>domaine</i> dans les traducteurs interne et externe statiques
const char * TraduitVersExterne (const char * <i>interne</i>)	traduit la chaîne <i>interne</i> de la langue interne de codage dans la langue de l'utilisateur
const char * TraduitVersInterne (const char * <i>externe</i>)	traduit la chaîne <i>externe</i> de la langue externe de l'utilisateur dans la langue de codage
const char * TraduitFormatCVersExterne (const char * <i>format</i>)	traduit le <i>format</i> du langage C de la langue de codage vers la langue de l'utilisateur, en garantissant la compatibilité des formats au niveau des arguments attendus
const char * VariableLangue ()	retourne le nom de la variable d'environnement utilisée pour la langue
const char * VariableCheminsTraduction ()	retourne le nom de la variable d'environnement utilisée pour les chemins vers les fichiers de traduction

Le mécanisme de traduction est considéré comme un mécanisme de *confort* non indispensable. Dans ces conditions il ne doit pas empêcher l'exécution d'un programme en cas d'indisponibilité des fichiers dictionnaires (sauf s'il s'agit de reconnaître des mots-clefs en entrée). Il est donc *recommandé* d'ignorer le code de retour de la fonction **AjouterDomaine**, sachant que si ce code est non nul de toute façon le programme devrait fonctionner normalement, en affichant ses messages dans la langue de codage interne.

AjouterDomaine permet d'ajouter un domaine de traduction à la table de correspondance déjà gérée, en lisant le fichier de ressources associé au domaine. En cas de problème la fonction renvoie un code retour différent de 0, mais comme expliqué ci-dessus, on peut ignorer les problèmes.

TraduitVersExterne traduit le mot-clef interne dans la langue utilisateur. Le pointeur retourné pointe sur une variable statique interne qui peut être écrasée à chaque nouvel appel. Si aucune traduction n'existe, la chaîne interne est copiée sans modification dans la variable statique avant retour.

TraduitVersInterne traduit le mot-clef externe dans la langue interne de développement. Le pointeur retourné pointe sur une variable statique interne qui peut être écrasée à chaque nouvel appel. Si aucune traduction n'existe, la chaîne externe est copiée sans modification dans la variable statique avant retour.

TraduitFormatCVersExterne traduit le format dans la langue utilisateur, en vérifiant que le nombre, le type et l'ordre des spécificateurs de conversions (%s, %f, %d, %o ...) sont respectés au cours de la traduction. On est donc assuré que la traduction d'un format valide pour une séquence d'appel donné dans une impression formatée sera un format valide dans la même impression. Le pointeur retourné pointe sur une variable statique interne qui peut être écrasée à chaque nouvel appel. Si

aucune traduction n'existe (ou si la traduction n'est pas valide), le format interne est copié sans modification dans la variable statique avant retour.

Remarque : Dans le cas du langage C++, on dispose en plus de la classe Traducteur, qui permet de créer une instance spécifique de traducteur entre la langue interne et une langue quelconque (pas forcément celle de la variable MRC_USER_LANG), avec les domaines que l'on veut.

exemple de programme C utilisant le systeme de traduction

Fichier source :

```
#include <stdio.h>
#include "club/Traducteur.h"

main ()
{
/* initialisation du domaine de traduction */
(void) AjouterDomaine("ExempleC");

/* impression avec traduction du formatage */
(void) printf(TraduitFormatCVersExterne("nom de la couleur : %s,
valeur %d\n"), TraduitVersExterne("rouge"), 255);

return 0;
}
```

Avec le fichier de ressources anglais :

```
"nom de la couleur : %s, valeur %d\n"
  "color name : %s, value %d\n"

"rouge"
  "red"
```

14.3 sous-routines FORTRAN

TAB. 94: sous-routines FORTRAN

signature	description
integer function domtrad (<i>domaine</i>) character*(*) <i>domaine</i>	ajoute le <i>domaine</i> dans les traducteurs internes et externes statiques
subroutine tradext (<i>interne</i> , <i>externe</i> , <i>IUtile</i>) character*(*) <i>interne</i> , <i>externe</i> integer <i>IUtile</i>	traduit la chaîne <i>interne</i> de la langue interne de codage dans la langue de l'utilisateur, <i>IUtile</i> indique le nombre de caractères utiles de la chaîne résultat <i>externe</i>
à suivre ...	

TAB. 94: sous-routines FORTRAN (suite)

signature	description
subroutine tradint (<i>externe, interne, IUtile</i>) character*(*) <i>externe, interne</i> integer <i>IUtile</i>	traduit la chaîne <i>externe</i> de la langue externe de l'utilisateur dans la langue de codage, <i>IUtile</i> indique le nombre de caractères utiles de la chaîne résultat <i>interne</i>
subroutine tradform (<i>interne, externe, IUtile</i>) character*(*) <i>interne, externe</i> integer <i>IUtile</i>	traduit le format <i>interne</i> du langage FORTRAN de la langue de codage vers la langue de l'utilisateur, en garantissant la compatibilité des formats au niveau des arguments attendus, <i>IUtile</i> indique le nombre de caractères utiles de la chaîne résultat <i>externe</i>
subroutine tradecrCH (<i>chaîne, IUtile, format ...</i>) character*(*) <i>chaîne, format</i> integer <i>IUtile</i>	traduit le <i>format</i> du langage FORTRAN et utilise le résultat pour formater les arguments variables qui suivent dans la <i>chaîne</i> , <i>IUtile</i> indique le nombre de caractères utiles de la <i>chaîne</i>
subroutine tradecrFD (<i>fd, format ...</i>) character*(*) <i>format</i> integer <i>fd</i>	traduit le <i>format</i> du langage FORTRAN et utilise le résultat pour formater les arguments variables qui suivent dans le fichier spécifié par le descripteur de fichier <i>fd</i> (il s'agit bien d'un descripteur C, pas d'une unité logique FORTRAN)

Le mécanisme de traduction est considéré comme un mécanisme de *confort* non indispensable. Dans ces conditions il ne doit pas empêcher l'exécution d'un programme en cas d'indisponibilité des fichiers dictionnaires (sauf s'il s'agit de reconnaître des mots-clefs en entrée). Il est donc *recommandé* d'ignorer le code de retour de la fonction **domtrad**, sachant que si ce code est non nul de toute façon le programme devrait fonctionner normalement, en affichant ses messages dans la langue de codage interne.

Le système de traduction est applicable à partir d'un programme FORTRAN. Les routines ci-dessus réalisent l'interface entre le FORTRAN et le C++. Les chaînes fortran n'ayant pas de marqueur de fin, celles-ci sont toujours complétées par des blancs si la chaîne à stocker est plus courte que le tableau. Par conséquent, les routines commencent par éliminer les blancs en fin de la chaîne en entrée avant de chercher la traduction. Il faut donc que les fichiers de ressources n'aient pas de blanc en fin de format, sinon les chaînes ne pourront jamais correspondre. Enfin, il est indispensable que la longueur de déclaration de la chaîne de sortie soit suffisante pour contenir la chaîne d'entrée (sans ses derniers blancs) ; en effet, en cas d'échec de traduction (ou de dépassement de capacité), la chaîne d'entrée sera recopiée dans la chaîne de sortie, cette copie étant effectuée sans vérification de dépassement de capacité.

exemple de programme FORTRAN utilisant le systeme de traduction

Fichier source :

```

c      program couleur

      character*1000 form, nom
      integer          lform, lnom, ier

C
      integer          domtrad
      external         domtrad
  
```

```
C      initialisation du domaine de traduction

      ier = domtrad('exempleF')

C      impression avec traduction du formatage

      form = '("nom de la couleur : ", A, I)'
      tradform(form, form, lform)
      tradext('rouge', nom, lnom)
      write (6, form(1:lform)) nom(1:lnom), 255

C
      stop
      end
```

Avec le fichier de ressources anglais :

```
"('nom de la couleur : ', A, I)"
  "('color name : ', A, I)"

"rouge"
  "red"
```

15 description des formats de fichiers

15.1 fichiers structurés

Le format des fichiers structurés est le format historique supporté par CLUB. Il n'est dans la pratique utilisé que par la bibliothèque MARMOTTES pour les descriptions des senseurs. Il a été créé un peu avant le format Madona, et longtemps avant que CLUB ne s'interface avec la bibliothèque MADONA. Ce format est désormais considéré comme obsolète et les nouveaux développements ne devraient pas s'appuyer sur celui-ci mais plutôt sur les formats madona (voir la section 15.2) et XML (voir la section 15.3). Ce format est conservé pour le support des fichiers senseurs de MARMOTTES.

Le principe des fichiers structurés a été élaboré afin de permettre une représentation textuelle de structures de données imbriquées avec un nombre de niveaux quelconque non déterminé à l'avance, mais découvert au cours de la lecture du fichier. Ceci permet en particulier de décrire des structures récursives (par exemple une rotation peut être définie par combinaison d'autres rotations).

L'analyse est contrôlée par le programme, qui va chercher les blocs dont il a besoin les uns après les autres, inspecte leur contenu, et éventuellement fonde ses choix sur les valeurs lues. Il ne s'agit absolument pas d'un *langage*, pour lequel ce serait plutôt le texte du fichier qui contrôlerait le programme lecteur (traduction dirigée par la syntaxe).

Les lexèmes des fichiers structurés sont le marqueur de commentaire #, les délimiteurs { et }, les séparateurs (espace, tabulation, fin de ligne), et les champs (suite de caractères n'appartenant à aucune des catégories précédentes).

Les commentaires s'étendent du marqueur # à la fin de la ligne, il peut y avoir des données entre le début de ligne et les commentaires.

Hormis leur rôle de séparation des champs, les séparateurs n'ont aucune fonction dans la syntaxe des fichiers structurés, l'utilisateur peut en user à loisir pour améliorer la lisibilité de son fichier en jouant sur les lignes vides et l'indentation.

15.1.1 Blocs

Toute donnée qu'elle soit élémentaire ou composée est définie dans un bloc entre accolades précédé par un nom, la donnée est repérée dans le fichier par le nom de son bloc. On peut ainsi définir un fichier de constantes par :

Expl. 1 – *blocs de données*

```
pi {3.14}
e {2.17}
```

Seul le nom est utilisé pour accéder à la donnée, l'ordre des blocs dans le fichier est indifférent. Les blocs sans nom sont autorisés (on utilise alors une chaîne vide en guise de nom dans les routines d'accès). Le bloc étant référencé par son nom, il ne faut pas que deux blocs différents portent le même nom à l'intérieur d'une zone de recherche unique. La casse utilisée pour les caractères du nom *est* significative.

15.1.2 Données élémentaires

Une donnée élémentaire est une donnée ne contenant pas de sous-bloc (pas d'accolades imbriquées). Une donnée élémentaire peut contenir un ou plusieurs champs, alphabétiques et numériques, séparés par des blancs (espace, tabulation, fins de lignes). On accède à ces champs par leur numéro, l'ordre est donc important à l'intérieur d'un bloc élémentaire, comme le montre l'exemple 2 :

Expl. 2 – *différents types de blocs élémentaires*

```
cible      {Soleil}
precision {0.1}          # unités : degrés
axe        {0.707  0.707  0.0 } # <--  attention à l'ordre !!!
```

15.1.3 Données composées

Une donnée composée est décrite par un bloc contenant des sous-blocs (accolades imbriquées). Chaque sous-bloc a un nom (qui peut être vide) et l'ordre des sous-blocs est indifférent. En fait l'accès à un sous-bloc à partir d'un bloc est similaire à l'accès à un bloc depuis le fichier, exactement comme si un bloc n'était qu'un sous-bloc du fichier²⁷. La recherche d'un sous-bloc étant limitée par le bloc englobant, deux blocs différents peuvent contenir un sous-bloc de même nom sans ambiguïté.

Expl. 3 – *bloc composé*

```
cone { # cône d'axe i et d'angle pi/2 : demi-espace x > 0
      axe {1.0  0.0  0.0}
      angle {90.0}
    }
```

²⁷ce n'est pas un hasard, c'est implanté exactement de cette façon ...

Cette syntaxe permet de décrire une donnée structurée même à définition récursive (comme un arbre) mais pas de donnée engendrant une récursivité infinie (par exemple une liste dont la fin rebouclerait sur le début). Un programme prévu pour lire des données récursives recherche d'abord les blocs principaux, en extrait les sous-blocs, puis les sous-sous-blocs, jusqu'aux blocs élémentaires, à chaque fois par le nom. Selon la façon dont cette structure générale est spécialisée, on peut soit avoir des imbrications figées (un fichier est composé des blocs A, B et C, C étant décomposé en C1 et C2), soit avoir des imbrications variables (un fichier est composé d'un bloc état et d'un bloc liste, un bloc liste étant soit un bloc élémentaire - une tête de liste - soit composé d'un bloc tête élémentaire et d'un bloc queue, la queue étant une liste).

15.1.4 Inclusions

Il est souvent utile de séparer des données ayant trait à des domaines différents (par exemple les vrais senseurs d'un côté, les pseudo-senseurs de l'autre), la syntaxe des fichiers structurés propose donc un mécanisme d'inclusion de fichiers.

Le principe est que seul le nom du fichier *primaire* est fourni aux routines d'accès, mais que si ce fichier contient une chaîne du type : <autre>, l'ensemble de cette chaîne (caractères < et > compris) doit être remplacé par le contenu du fichier *autre*. Si le nom *autre* commence par un caractère / (par exemple </usr/local/senseurs/ires.fr>), il est utilisé tel quel. Si le nom ne commence pas par /, il est ajouté à la fin du nom du répertoire dans lequel le fichier primaire a été ouvert, pour constituer le nom complet du fichier à ouvrir (avec un / entre le nom du répertoire et le nom *autre*).

15.1.5 Héritage

Il est fréquent de prévoir dans un fichier plusieurs configurations pour un même senseur (avec ou sans inhibition, selon plusieurs modes de fonctionnement, en considérant tous les exemplaires montés différemment, ...). Une grande partie des données de chaque configuration est alors similaire, et dupliquer les blocs qui les définissent conduit vite à de très gros fichiers.

Expl. 4 – *duplication d'informations*

```
config_1 { structure_complexe {...}
          valeur {1}
        }
config_2 { structure_complexe {...} # duplication des valeurs de config_1
          valeur {2}
        }

...

config_12 { structure_complexe {...} # duplication des valeurs de config_1
           valeur {12}
         }
```

Si le bloc *structure_complexe* est difficile à décrire, le dupliquer dans le fichier est lourd, peu lisible, et peut conduire à des erreurs en cas de modification (il faut bien penser à mettre à jour toutes les occurrences distinctes dans le fichier simultanément).

Pour pallier à ce genre de problème, la syntaxe générale propose un mécanisme d'héritage : si au moment de la recherche d'un bloc (ou d'un sous-bloc) on ne trouve pas le nom désiré dans la zone de recherche (le bloc

englobant ou le fichier complet), alors on regarde s'il n'y a pas un bloc élémentaire nommé =>²⁸, le contenu de ce bloc est interprété comme le nom d'un bloc dans lequel on peut puiser les sous-blocs manquants.

L'exemple 4 peut être reproduit en évitant la duplication à l'aide de ce mécanisme :

Expl. 5 – *utilisation de l'héritage pour éviter la duplication*

```
config_1 { structure_complexe { ... }
          valeur { 1 }
        }
config_2 { valeur { 2 } => { config_1 } }
...
config_12 { valeur { 12 } => { config_1 } }
```

Le nom du bloc référencé par le pointeur d'héritage => est interprété au niveau fichier, c'est-à-dire qu'il est recherché dans tout le fichier et non à l'intérieur d'un bloc particulier qui l'engloberait. Il est cependant possible d'hériter des données d'un sous-bloc si l'on précise son chemin d'accès complet, avec tous ses blocs ancêtres séparés par des points.

Expl. 6 – *héritage de sous-blocs*

```
primaire { secondaire { tertiaire { i { 1 0 0 }
                                   j { 0 1 0 }
                                   k { 0 0 1 }
                                 }
          }
repere   { => { primaire.secondaire.tertiaire } } # on hérite de i, j, k
```

Il est important de remarquer que ce mécanisme d'héritage n'est utilisé que pour la recherche de blocs *nommés*, on ne peut pas l'utiliser pour hériter un champ d'un bloc élémentaire²⁹.

Les mécanismes d'inclusion et d'héritage peuvent être utilisés conjointement pour mettre en place des senseurs complexes possédant des données ajustables profondément enfouies au sein des structures de données. La démarche recommandée est alors de décrire dans un fichier unique la structure complète, en héritant les parties variables d'un bloc de *paramétrage* externe à ce fichier. L'utilisateur voulant utiliser un tel senseur doit alors inclure le fichier générique dans son fichier de senseurs et définir lui-même les données du bloc de paramétrage. Si de plus il décide d'isoler le bloc de paramétrage dans un fichier inclus, il limite considérablement les risques liés à l'édition manuelle de ce fichier pour ajuster le senseur (par exemple pour suivre les ajustements de paramètres selon les télécommandes).

Les exemples 7, 8 et 9 montrent ainsi que dans le (très) complexe fichier **std15.fr** (situé dans le répertoire **exemples** de la distribution de la bibliothèque MARMOTTES), des pointeurs vers des sous-blocs numérotés de **STD15_PARAMETRES.ZPT1** à **STD15_PARAMETRES.ZPT4** ont été prévus pour définir les angles de début des zones de présence Terre, des pointeurs vers un sous-bloc **STD15_PARAMETRES.LG** permettant de définir la longueur commune de toutes ces zones. L'utilisateur peut ainsi prendre en compte les capacités de programmation du champ de vue du senseur. D'autre part dès lors que le bloc **STD15_PARAMETRES** est défini dans un fichier indépendant inclus par le fichier primaire, la modification des angles de programmation est simple et peu risquée.

²⁸on peut lire => comme : « voir aussi »

²⁹ce serait à la fois difficile à mettre en place, peu lisible, et peu utile

Expl. 7 – *extrait de senseurs.fr*

```
<parametrage.fr>
<std15.fr>
IRES_ROULIS { repere { ... } => { STD15_BASE_ROULIS } }
```

Expl. 8 – *extrait de parametrage.fr*

```
STD15_PARAMETRES { ZPT1 { angle { 80.15 } }
                  ZPT2 { angle { 0.00 } }
                  ZPT3 { angle { 81.57 } }
                  ZPT4 { angle { 1.41 } }
                  LG   { angle { 9.84 } }
                  }
```

Expl. 9 – *extrait de std15.fr*

```
STD15_BASE_ROULIS
{ ...
  zone_1
  { { => { STD15_BASE_ROULIS.trace_1_sans_masque } }
    inter
    { { rotation { axe { 0 1 0 } => { STD15_PARAMETRES.ZPT1 } }
      de      { cone { axe { 0 0 1 } angle { 90 } } }
    }
    inter
    { rotation { rotation { axe { 0 1 0 } => { STD15_PARAMETRES.ZPT1 } }
      de      { axe { 0 1 0 } => { STD15_PARAMETRES.LG } }
    }
    de      { cone { axe { 0 0 -1 } angle { 90 } } }
  }
}
}
...
}
```

15.2 fichiers Madona

Le format Madona est le format supporté par la bibliothèque du même nom. Il est supporté de façon optionnelle par CLUB depuis la version 8.0, sous réserve que la bibliothèque ait été configurée en ce sens (voir la section 6, page 18) en s'appuyant directement sur la bibliothèque MADONA elle-même. Ce format est décrit dans la documentation Madona [DR4].

15.3 fichiers XML

Le format XML est une instantiation XML spécifique à CLUB. Il est supporté de façon optionnelle par CLUB depuis la version 8.0, sous réserve que la bibliothèque ait été configurée en ce sens (voir la section 6, page 18) en s'appuyant sur la bibliothèque XERCES, et a été complètement revu pour la version 10.0.

Le principe adopté lors de la création de ce format a été de réaliser les vérifications de structuration de bas niveau (typage des données, imbrications) comme un service interne fourni par la bibliothèque, de la même façon que pour les deux autres formats (fichiers structurés et Madona), l'application se consacrant

quant à elle à la sémantique du contenu (nom associés aux données). C'est ce choix qui permet à CLUB d'offrir une interface commune permettant à un applicatif de s'affranchir du format de fichier réellement utilisé, par l'intermédiaire de la classe `UniqDataFile` décrite à la section 12.24, page 106.

Ce choix a ainsi conduit à la création d'un format XML utilisant les types en tant qu'éléments XML, les noms et les valeurs des données étant représentées par des attributs de ces éléments, comme le montre l'exemple simple suivant :

Expl. 10 – *éléments XML*

```
<structure name="polynome">
  <int-data name="ordre" value="2"/>
  <real-table name="coefficients">
    <real-data index="0" value="-0.5"/>
    <real-data index="1" value="0.0" />
    <real-data index="2" value="1.5" />
  </real-table>
</structure>
```

La DTD (*Document Type Definition*) de club est définie dans le fichier `club-data.dtd` installé avec la bibliothèque et trouvé automatiquement à l'aide du mécanisme décrit à la section 5, page 17. Cette DTD spécifie que l'élément racine a pour nom `club-data` et possède un attribut obligatoire `version` (dont la valeur doit pour l'instant systématiquement être "1.0") et un attribut implicite `units-file` dont la valeur par défaut est "`club-units.xml`" et qui correspond au fichier d'unités associé à ce fichier de données (la valeur par défaut correspond au fichier qui est installé avec la bibliothèque et dont le but est qu'il réponde à la majorité des besoins). L'élément `club-data` peut contenir une suite d'éléments de données ou d'éléments `include`. Les éléments de données sont `structure`, `reference`, `real-data`, `int-data`, `string-data`, `int-table`, `real-table`, `string-table`, `structure-table` et `table-table`.

Les éléments de type `xxx-table` contiennent des éléments du type prévu auxquels on accède par un numéro (attribut `index`). Les tables de réels peuvent définir une unité par défaut à l'aide de l'attribut `default-unit`. Les éléments de type `xxx-structure` contiennent des éléments quelconques auxquels on accède par leur nom (attribut `name`). Les éléments de type `xxx-data` sont des éléments finaux vides, on y accède soit par leur `index` soit par leur nom selon l'élément qui les contient, et leur valeur est définie par l'attribut `value`. Les données réelles peuvent avoir une unité spécifiée dans l'attribut `unit`, c'est le symbole de l'unité qui doit être indiqué, ou la combinaison de symboles pour une unité composée. La syntaxe utilisée pour la combinaison des unités est identique à la syntaxe définie par la bibliothèque MADONA, elle est rappelée à la section 15.4.

Les éléments `include` permettent d'inclure des fichiers annexes à partir d'un fichier principal, ils sont vides et ont un attribut obligatoire, `included-file` qui contient le nom relatif ou absolu du fichier à inclure.

Les éléments `reference` permettent de faire référence à une donnée qui est physiquement stockée ailleurs, dans le même fichier ou dans un fichier différent, l'accès à l'élément se faisant par l'attribut `reference` à l'aide d'un sous-ensemble de la syntaxe normalisée XPointer sous la forme indiquée dans l'exemple suivant, dans lequel on fait référence à l'élément du fichier `WGSB4.xml` dont l'attribut `name` vaut `ellipsoïde`. Il faut prendre garde au fait que le support Xpointer est limité à cette seule syntaxe. On ne peut pas chercher à un endroit précis de l'arborescence, ni selon un critère autre que la valeur de l'attribut `name`. La raison en est qu'implémenter une syntaxe spécifique aurait imposé l'utilisation de CLUB pour gérer les fichiers, alors que le standard permet de les faire traiter par n'importe quel autre applicatif XML. Ce standard étant assez lourd, cependant, il semblait préférable d'attendre que les bibliothèques XML de bas niveau le supportent de façon native plutôt que de tenter de l'implémenter nous-même. La limitation doit donc être vue comme une limitation temporaire, susceptible d'être levée avec les évolutions de XERCES.

Expl. 11 – *fichier XML complet avec référence externe*

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<!DOCTYPE club-data
    PUBLIC "-//CNES// DTD club-data XML V1.0//EN"
        "http://logiciels.cnes.fr/CLUB/club-data.dtd" >

<club-data version="1.0" >
    <structure name="modèle" >
        <string-data name="name" value="GRIM5 C1" />
        <real-data name="mu" value="398600.4415" unit="km^3/s^2">
    </structure>
    <reference name="ellipsoïde"
        reference="WGS84.xml#xpointer(/*[@name='ellipsoïde'])">
</club-data>
```

L'exemple précédent montre également que l'on peut utiliser n'importe quel type de caractère dans les noms, y compris des caractères accentués dès lors que l'on spécifie l'encodage du fichier (ici iso-8601). L'ensemble des caractères unicode est supporté.

La syntaxe complète et précise des fichiers peut être trouvée dans la DTD installée avec la bibliothèque et qui représente la référence ultime. Il n'est de toute façon pas recommandé de tenter d'écrire de tels fichiers à la main, il vaut mieux s'appuyer sur les services d'écriture fournis par la bibliothèque qui garantissent le respect de la syntaxe.

15.4 fichiers d'unités XML

Les fichiers de données XML supportent les conversions d'unités. Ils s'appuient pour cela sur un fichier externe définissant les unités, ce fichier pouvant être partagé par de nombreux fichiers de données. L'idéal serait même que le fichier par défaut installé par la bibliothèque soit enrichi selon les besoins des utilisateurs (il contient déjà beaucoup d'unités dans sa version initiale) et que ceux-ci n'aient pas besoin d'en spécifier un autre dans l'attribut **units-file** de l'élément racine de leurs fichiers de données.

Le fichier des unités utilise une syntaxe propre pour décrire les unités. Cette syntaxe très simple est spécifiée par la DTD installée par la bibliothèque (fichier **club-units.dtd**). L'élément de base s'appelle **club-units**, il n'a aucun argument. Cet élément contient une suite d'éléments **reference-unit** qui définissent des unités de référence qui doivent être des unités SI ou des combinaisons d'unités SI. Chaque unité doit définir les attributs **description**, **name** et **symbol**, seul le symbole étant important puisque c'est lui qui intervient dans les combinaisons du type **kg*m/s^2**. Les attributs restant (**dim-length**, **dim-mass**, **dim-time**, **dim-electric-current**, **dim-temperature**, **dim-amount-of-substance** et **dim-luminous-intensity** pour les sept axes de référence du système international, **dim-planar-angle** et **dim-solid-angle** pour les deux axes supplémentaires sont des entiers signés qui indiquent la dimension de l'unité. Ces attributs ont 0 comme valeur par défaut, ce qui permet de ne spécifier que les attributs ayant des valeurs non nulles. Ce sont ces dimensions qui permettent au système de convertir correctement par exemple des henry en nanosecondes par siemens et de savoir qu'il est impossible de convertir des webers en teslas par mètre carré alors que rien ne relie aucune de ces six unités de façon explicite dans le fichier. Les éléments **reference-unit** peuvent contenir des éléments **alternate-unit** donnant des unités alternatives ayant la même dimension physique mais des échelles différentes ou des décalages. L'exemple suivant montre ainsi un extrait du fichier des unités installé par la bibliothèque, pour l'axe des temps.

Expl. 12 – *fichier d'unités XML*

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<!DOCTYPE club-units PUBLIC "-//CNES// DTD club-units XML V1.0//EN"
    "http://logiciels.cnes.fr/CLUB/club-units.dtd" >

<club-units>

    <!-- ... quelques unités omises ici ... -->

    <reference-unit description="time" symbol="s" name="second" dim-time="1" >
        <alternate-unit symbol="mn"      name="minute"      factor="60"      />
        <alternate-unit symbol="h"       name="hour"        factor="3600"   />
        <alternate-unit symbol="d"       name="day"         factor="86400"  />
        <alternate-unit symbol="j"       name="jour"        factor="86400"  />
        <alternate-unit symbol="ms"      name="millisecond"  factor="1.0e-3" />
        <alternate-unit symbol="us"      name="microsecond" factor="1.0e-6" />
        <alternate-unit symbol="μs"      name="microsecond" factor="1.0e-6" />
        <alternate-unit symbol="ns"      name="nanosecond"   factor="1.0e-9" />
    </reference-unit>

    <!-- ... quelques unités omises ici ... -->

</club-units>
```

Cet exemple montre que la même unité peut apparaître plusieurs fois avec des symboles différents (*d* et *j* pour les jours). Certaines unités nécessitent d'utiliser des caractères hors de l'ASCII, ainsi le symbole des microsecondes est : μs (mais on utilise souvent *us* en plus).

XML permettant l'utilisation de ces symboles, le fichier des unités a été écrit en les prenant en compte. Cependant, l'encodage de ces symboles varie d'un système à l'autre. Si le fichier est écrit à l'aide de l'encodage iso-latin-1, le caractère μ (qui est présent sur les claviers de PC français) a pour code hexadécimal 0xB5, dans un autre encodage il sera à une position différente. Afin d'éviter aux utilisateurs des problèmes de conversion, les deux DTD *club-data.dtd* et *club-units.dtd* définissent des entités générales pour ces symboles particuliers (voir la table 95). Ces entités générales peuvent être utilisées dans le fichier des unités (l'exemple précédent illustre ce cas), dans les fichiers de données, mais également dans les arguments des fonctions de la bibliothèque (méthode **getRealData** de la classe XMLData et autres méthodes similaires).

TAB. 95 – entités générales prédéfinies

nom	code hexadecimal unicode	exemple	aspect visuel
deg	0x00B0	°C	°C
micro	0x00B5	µs	μs
ohm	0x2126	kΩ	k Ω

Les unités prédéfinies dans le fichier *club-unites.xml* installé par la bibliothèque sont listées dans la table 96 (les unités de référence sont en gras). Les symboles sont tous différents, cette propriété est indispensable. C'est la raison pour laquelle il a été décidé d'utiliser °C et °F pour représenter les degrés celsius et fahrenheit, afin d'éviter les conflits avec les coulombs et les farads.

TAB. 96 – unités du fichier installé

nom	symbole	nom	symbole	nom	symbole
meter	m	kelvin	K	ohm	Ω
kilometer	km	celsius degree	$^{\circ}\text{C}$	kiloohm	$\text{k}\Omega$
decimeter	dm	fahrenheit	$^{\circ}\text{F}$	megaohm	$\text{M}\Omega$
centimeter	cm	mole	mol	watt	W
millimeter	mm	candela	cd	milliwatt	mW
inch	in	radian	rad	kilowatt	kW
foot	ft	degree	deg	megawatt	MW
yard	yd	grade	gra	horsepower	Hp
nautic mile	nmi	steradian	sr	coulomb	C
mile	mi	spat	sp	volt	V
kilogram	kg	hertz	Hz	kilovolt	kV
gram	g	kilohertz	kHz	millivolt	mV
milligram	mg	megahertz	MHz	microvolt	μV
ton	t	gigahertz	GHz	microvolt	μV
pound	lb	newton	N	farad	F
ounce	oz	millinewton	mN	microfarad	μF
second	s	kilogram-force	kgf	microfarad	μF
minute	mn	pascal	Pa	nanofarad	nF
hour	h	hectopascal	hPa	picofarad	pF
day	d	kilopascal	kPa	siemens	S
jour	j	atmosphere	atm	weber	Wb
millisecond	ms	bar	b	henry	H
microsecond	us	millibar	mb	lumen	lm
microsecond	μs	pounds per square inch	psi	lux	lx
nanosecond	ns	joule	J	becquerel	Bq
ampere	A	kilojoule	kJ	gray	Gy
milliampere	mA	megajoule	MJ	sievert	Sv
microampere	μA	tesla	T	katal	kat
microampere	μA	gauss	G		

La combinaison des unités obéit à une syntaxe très simple identique à celle qui est utilisée par la bibliothèque MADONA. Cette syntaxe utilise les opérateurs * pour la multiplication, / pour la division et ^ ou ** pour l'exponentiation et autorise des exposants sous formes d'entiers positifs ou négatifs. L'associativité des opérateurs est de gauche à droite et les exposants sont prioritaires par rapport aux multiplications et divisions. Ceci implique que les définitions suivantes sont toutes équivalentes :

$$\text{m*s}^{-2} \Leftrightarrow \text{m/s/s} \Leftrightarrow \text{m/s*s}^{-1}$$