Vers une nouvelle approche du calcul scientifique en C++

Gabriel Dos Reis

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Vers une nouvelle approche du calcul scientifique en C++

Gabriel DOS REIS

N■ 3362

Février 1998

THÈME 2

ISSN 0249-6399

Vers une nouvelle approche du calcul scientifique en C++

Gabriel DOS REIS *

Thème 2 — Génie logiciel
et calcul symbolique
Projet SAGA

Rapport de recherche n° 3362 — Février 1998 — 27 pages

Résumé : Des exploitations judicieuses de facilités « génériques » de Standard C++ permettent de surpasser les performances atteintes par les implémentations courantes de bibliothèques FORTRAN. Nous présentons également quelques applications des idées novatrices de la STL dans le cadre de l'optimisation de code. Nous en déduisons une application en BLAS.

Mots-clé : C++, programmation orientée objet, programmation générique, polymorphisme, calcul scientifique, BLAS.

(Abstract: pto)

*INRIA Sophia-Antipolis et CMLA, ENS de Cachan – CNRS URA 1611, France.

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Towards a new Approach of Scientific Computing using C++

Abstract: Judiscious exploitations of template facilities of Standard C++ lead to competitive programms with traditionnal libraries written in FORTRAN. Applications of revolutionnary concepts of STL are derived in code optimization area; in particular a draft implementation of class `valarray` (the Standard C++ BLAS like array) is proposed.

Key-words: C++, Object Oriented Programming, Generic Programming, Scientific Computing, BLAS

1 Introduction

La performance de programmes, en termes de temps d'éxécution, est un problème épineux que rencontre tout programmeur. Ce problème prend une ampleur particlière en calcul scientifique. L'optimisation de code, au niveau du programmeur, est assez délicate car elle fait intervénir des données assez contradictoires : rester « haut niveau » et être le plus éfficace possible. Il est clair qu'obtenir un programme non trivial le plus efficace possible sur une machine *doit* prendre en considération l'architecture particulière, voie royale pour abandonner la portabilité, et les constructions sémantiques du langage utilisé. Cet article s'occupe plutôt de l'aspect sémantique.

Après avoir connu une longue période de disgrâce aux yeux de la communauté scientifique, qui lui préférait FORTRAN, le langage C++ suscite depuis quelques années un regain d'intérêt (cf. [2], [8], [12]). Les opposants à l'implémentation de bibliothèques scientifiques en C++ avançaient comme principal argument la rapidité d'exécution des programmes écrits en FORTRAN par rapport à leurs versions C++. Bjarne STROUSTRUP [9] reconnaît le problème :

> Neither C nor C++ were designed primarily with numeric computation in mind. However numeric computation typically occurs in the context of other work — such as database access, networking, instrument control, graphics, simulation, financial analysis, etc. — so C++ becomes an attractive vehicle for computations that are part of a larger system. Furthermore, numeric methods have come a long way from being simple loops over vectors of floating-point numbers. Where more complex data structures are needed as part of a computation, C++'s strengths become relevant. The net effect is that C++ is increasingly used for scientific and engineering computation involving sophiscated numerics. Consequently, facilities and techniques supporting such computation have emerged.

Dans cet article nous analysons les reproches quotidiens faits au langage C++ et quelques concepts et nouvelles techniques de programmation qui permettent de contourner ces difficultés. Ce travail trouve ses racines dans les problèmes rencontrés dans les implémentations de bibliothèques du projet FRISCO. Ces problèmes se posent, en réalité, partout où l'on utilise (mal) certaines facilités de C++.

Je tiens à remercier les membres des projets SAGA et SAFIR, plus particulièrement Bernard MOURRAIN et Ioannis Z. EMIRIS. Mes remerciements vont également à Christophe LABOURDETTE et Marc TAJCHMAN du Centre de Mathématiques et

de Leurs Applications (ENS Cachan – CNRS URA 1611) pour leurs nombreux et fructueux commentaires sur le manuscrit de cet article.

2 Les problèmes posés par le modèle objet C++

Le langage C++ est un langage de programmation multi-paradigme : il supporte beaucoup de styles (procédural, générique, orienté objet, ...). Son côté le plus attrayant est probablement le support de la programmation orientée objet. Il offre également d'autres « sucres syntaxiques » comme les surcharges de fonctions et d'opérateurs. Cette section montre comment l'utilisation *naïve* de ces facilités devient une pénalité. Cette pénalité est intimement liée à la sémantique et au modèle objet du langage. Pour une discussion détaillée de ce modèle, nous renvoyons le lecteur à l'excellent ouvrage de Stanley LIPPMAN[4].

2.1 Les objets temporaires

Les objets temporaires sont créés lorsque le langage a besoin d'un objet et que celui n'est pas nommé comme dans le bout de programme suivant :

```
cout << complex<double>(3.5, .1) << '\n';
```

Ainsi des objets temporaires sont créés chaque fois qu'une fonction doit retourner une valeur (au sens C++) car l'objet (la valeur) de retour d'une fonction n'est pas nommé. Voyons comment le problème se pose en pratique. La facilité de surcharge des opérateurs prédéfinis permet de traduire un algorithme en langage C++ en utilisant des notations assez intuitives. Supposons définie une classe `Vector` dont le propos serait de fournir un type de donnée abstrait ainsi que les opérations usuelles dans les espaces vectoriels. Si `u`, `v` et `w` sont des instances de la classe `Vector`, la somme de `u` et `v` affectée à l'objet `w` peut se dire, en utilisant la surcharge de l'opérateur `+`, comme suit :

```
w = u + v;
```

Cette notation a l'avantage de rendre le code clair ; de plus elle repousse la frontière entre les types et fonctions définis par l'utilisateur et ceux prédéfinis du langage. Cependant les opérateurs (ainsi que les fonctions retournant des objets) recèlent des pièges. Par exemple l'exécution de l'expression

```
w = u + v;
```

implique l'appel du constructeur de copie qui recopiera la somme `u + v` dans un objet temporaire `t`, local à l'opérateur

```
Vector operator+(const Vector&, const Vector&)
```

Cet objet temporaire `t` est ensuite retourné par valeur, d'où un deuxième appel au constructeur de copie. Les choses se compliquent naturellement dès que la somme à évaluer comprend plus de deux opérandes car C++ évaluera la somme par paire d'opérandes, créant des objets temporaires à chaque fois : en effet le langage a besoin d'objets pour contenir les résultats intermédiaires. Il est évident que le prix à payer pour le luxe de surcharger les opérateurs devient rapidement inacceptable lorsque la taille des objets en question est grande, ce qui est le cas en pratique dans le cadre du calcul scientifique. Face à cette situation, plusieurs attitudes sont possibles :

- éviter la surcharge d'opérateurs : des langages, comme le FORTRAN 77 survivent sans ce concept. Écrire à la place, par exemple, des versions « procédurales » :

  ```
  void AddVector(const Vector& operand1,
                 const Vector& operand2,
                 Vector& result);
  ```

  dans notre exemple. Mais le fait que FORTRAN 90 fournisse un mécanisme de surcharge d'opérateurs invite à réfléchir sur le bénéfice réel à tirer de l'abandon de la surcharge d'opérateurs ;

- utiliser les opérateurs augmentés (ici, `+=`). Un grand défaut de ces solutions est que le programme devient assez illisible lorsqu'il s'agit de calculer une somme ayant plus de deux opérandes ;

- implémenter un mécanisme d'évaluation paresseuse : l'idée est de bâtir *dynamiquement* l'arbre d'expression correspondant à la somme et de n'exécuter l'évaluation que lorsque nécessaire. Cette technique peut être laborieuse à

mettre en oeuvre pratiquement (voir [5]). De plus elle laisse très peu d'opportunité à l'optimisateur. Ce qui est bien dommage ;

– faire un décompte des références à un objet (l'idiome « reference counting » en anglais) ; cette solution est délicate à implémenter car se pose le problème du choix de la sémantique (valeur ou référence) à donner à l'affectation et aux copies d'objets. Le choix n'est pas toujours facile à faire ;

– demander l'assistance d'un ramasse-miettes afin de pouvoir retourner des références sur des objets résulats. Ici encore les optimisateurs actuels ont du mal à coopérer efficacement avec les ramasses-miettes.

Ces deux dernières solutions ne résolvent pas complètement le problème des objets temporaires. De plus chacune des méhodes évoquées ci-haut requiert au moins n boucles pour calculer et affecter la somme, lorsque celle-ci comporte n opérandes, alors que moralement une *seule* boucle devrait suffir.

2.2 Classes abstraites et coûts des indirections

En calcul numérique, les cas d'objets mathématiques n'admettant pas de modélisation efficace canonique ne sont pas rares : une matrice peut être pleine, creuse, ou plus généralement structurée. On ne peut décemment pas mépriser cette structure ; de même un polynôme creux est plus efficacement représenté, par exemple, sous forme de listes de monômes alors qu'un tableau convient mieux pour un polynôme dense. Or il est impératif de chercher à isoler, cacher ces détails de stockage des objets, afin de fournir une interface quasi uniforme et en même temps implémenter le meilleur algorithme pour chaque structure. En d'autres termes il faut abstraire tous ces détails et les intégrer dans un même cadre de travail [1]. Ici intervient le bénéfice de la programmation orientée objet : le *polymorphisme*. Une solution traditionnelle à ce type de problème est l'utilisation de classes abstraites (au sens C++) et d'héritage impliquant donc les fonctions virtuelles, en d'autres termes l'utilisation du polymorphisme *dynamique* [2]. Par exemple, le code dont le listing est donné par les figures 1-2 peut être une ébauche d'interface du type de donnée polynôme à une indéterminée.

1. Selon Andrew KOENIG [3] « Abstraction is selective ignorance. »

2. C++ n'est pas le seul langage objet implémentant le polymorphisme dynamique : il y a le précurseur SMALLTALK ; certains langages se proclamant orienté objet ne fournissent que le polymorphisme dynamique, c'est le cas de JAVA.

```
#include <utility>
#include <valarray>
#include <map>
#include <iostream>
class Polynomial {
public:
    Polynomial() {}
    virtual ~Polynomial() {}
    ...
    virtual double operator()(double) const =0;
        // I/O
    virtual ostream& PrintOn(ostream&) const =0;
    virtual istream& ReadFrom(istream&) =0;
};
ostream& operator<<(ostream& os, const Polynomial& P)
{ return P.PrintOn(os); }
istream& operator>>(istream& is, Polynomial& P)
{ return P.ReadFrom(is); }
class DensePolynomial : public Polynomial {
public:
    ...
    double operator()(double x) const
    {
        // implement Horner scheme
    }
    ...
        // implement I/O assuming linear array
    ...
private:
    valarray<double> poly;
};
```

FIG. 1 – *Ébauche d'une implémentation de la class* `Polynomial` *en utilisant une classe abstraite.*

```
class SparsePolynomial : public Polynomial {
public:
    ...
    double operator()(double x) const
    {
        // implement the most efficient algorithm
    }
    ...
        // implement I/O assuming assoc array
    ...
private:
    map<size_t, double> poly;
};
```

FIG. 2 – *Ébauche d'une implémentation de la class* `Polynomial` *en utilisant une classe abstraite (suite)*

Chacune des deux classes dérivées de la classe `Polynomial` implémentera l'opérateur d'appel de fonction, qui denote ici l'évaluation d'un polynôme en un point, de la manière la plus efficace. Cependant les fonctions virtuelles ne sont pas gratuites : l'appel d'une fonction virtuelle est une décision prise pendant l'éxécution (et consiste en général en des indirections). Un bon compilateur minimisera autant que possible le prix d'une fonction virtuelle mais celui-ci devient non négligeable lorsque la fonction est invoquée intensivement (par exemple à l'intérieur d'une boucle). De plus, l'utilisation effective du polymorphisme « dynamique », à travers les fonctions virtuelles, se fait essentiellement par les pointeurs (ou de manière un peu équivalente, les références) ; or l'appel d'une fonction virtuelle, aussi petite soit elle, ne peut bénéficier du mécanisme « inline » puisque le type exact de l'objet n'est pas connu au moment de la compilation. Scott MEYERS [5] explique de manière approfondie la gestion des fonctions virtuelles et les problèmes connexes en matière de complexité espace-temps.

Pour illustrer ce dernier point, nous examinons un exemple assez simple où les conditions sont assez favorables (figure 3). On trouve dans la bibliothèque Standard de C++ des objets fonctionnels (cf. §3.1.4). Parmi ces objets fonctionnels figurent les

opérateurs binaires. Supposons d'autre part qu'il existe une fonction `eval`[3] qui a pour but d'évaluer un opérateur binaire sur une paire d'opérandes (que l'on supposera de type `int` pour simplifier). Pour que cette fonction s'applique à tous les opérateurs binaires, une solution qui s'inscrit dans l'approche traditionnelle du polymorphisme en C++ consisterait à disposer d'une classe abstraite `BinaryOperation` de laquelle seront dérivés tous les opérateurs binaires concrets. Nous considérons l'addition dans notre cas. Le source assembleur généré par le compi-

```
#include <iostream.h>
struct BinaryOperation {
    BinaryOperation() {}
    virtual ~BinaryOperation() {}
    virtual int Apply(int, int) const =0;
};
struct Addition : public BinaryOperation {
    Addition() {}
    virtual int Apply(int x, int y) const
    { return x + y; }
};
inline int eval(BinaryOperation* pBO, int x, int y)
{ return pBO->Apply(x, y); }
int main() {
    BinaryOperation* pOp = new Addition;
    int s = 0;
    for (int i=0; i<100; ++i)
         s = eval(pOp, i, s);
    cout << s << '\n';
}
```

FIG. 3 – *Implémentation d'opérateurs binaires avec des fonctions virtuelles.*

lateur SunPro-4.1 avec l'option d'optimisation `-O` est reproduit ci-dessous, figure 4 (modulo quelques coupures pour rendre la lecture facile). Comme l'on peut le constater l'adresse de la fonction virtuelle `Apply` est toujours calculée (contenu du registre `l0`) puis un transfert de contrôle est effectué pour faire la petite opération d'addition de deux nombres entiers... Pour anticiper la section suivante, voici une

3. cette fonction n'existe pas en réalité dans la bibliothèque Standard

```
!   15                !int main()
!   16                !{
!   17                !    BinaryOperation* pOp = new Addition;
/* 0x0004         17 */         call    __0OnwUi,1
/* 0x0008            */         or      %g0,4,%o0
/* 0x000c            */         orcc    %g0,%o0,%i1
/* 0x0010            */         be      .L77000004
/* 0x0014            */         or      %g0,0,%o2
                       .L77000002:
/* 0x0018            */         sethi   %hi(.LI21),%o1
/* 0x001c            */         ld      [%o1+%lo(.LI21)],%o1
/* 0x0020            */         st      %o1,[%o0]
/* 0x0024            */         sethi   %hi(.LI22),%o1
/* 0x0028            */         ld      [%o1+%lo(.LI22)],%o1
/* 0x002c            */         st      %o1,[%o0]
                       .L77000004:
!   18                !    int s = 0;
!   19                !    for (int i=0; i<100; ++i)
/* 0x0030         19 */         or      %g0,0,%i0
                       .L77000006:
!   20                !         s = eval(pOp, i, s);
/* 0x0034         20 */         ld      [%i1],%o0
                       .L900000111:
/* 0x0038         20 */         ldsh    [%o0+16],%o0
/* 0x003c            */         ld      [%i1],%o1
/* 0x0040            */         add     %i1,%o0,%o0
/* 0x0044            */         add     %o1,16,%o1
/* 0x0048            */         ld      [%o1+4],%l0
/* 0x004c            */         jmpl    %l0,%o7
/* 0x0050            */         or      %g0,%i0,%o1
/* 0x0054            */         add     %i0,1,%i0
/* 0x0058            */         or      %g0,%o0,%o2
/* 0x005c            */         cmp     %i0,100
/* 0x0060            */         bl,a    .L900000111
/* 0x0064            */         ld      [%i1],%o0
```

FIG. 4 – *Sources assembleur du code de la figure 3*

autre possibilité d'implémentation du problème précédent, dans le cadre que nous préconisons (figure 5). Comparer la sortie assembleur (figure 6) : Remarquer que

```
#include <iostream.h>
struct Addition {
    int Apply(int x, int y) const
    { return x + y; }
};
template<class BinOp>
inline int eval(BinOp op, int x, int y)
{ return op.Apply(x, y); }
int main() {
    int s = 0;
    for (int i=0; i<100; ++i)
        s = eval(Addition(), i, s);
    cout << s << '\n';
}
```

FIG. 5 – *Implémentation « template » des opérateurs binaires*

les fonctions déclarées « inline » sont honorées et que cela correspond bien à une écriture plus directe :

```
int s = 0;
for (int i=0; i<100; ++i)
    s += i;
```

De plus le code résultant est plus petit.

3 Les nouvelles techniques

La standardisation de C++ a provoqué des mutations profondes dans le langage. Le mécanisme de « template » y a trouvé une place importante à tel point qu'il est maintenant impossible de l'imaginer *sans* les templates[4]. L'adoption en Juillet

4. Toute la bibliothèque Standard C++ est conçue uniquement avec des templates.

```
!   10                !int main()
!   11                !{
!   12                !    int s = 0;
/* 0x0004         12 */         or      %g0,0,%o1
!   13                !    for (int i=0; i<100; ++i)
/* 0x0008         13 */         or      %g0,0,%o0
                       .L77000003:
!   14                !        s = eval(Addition(), i, s);
/* 0x000c         14 */         add     %o0,%o1,%o1
                       .L900000109:
/* 0x0010         14 */         add     %o0,1,%o0
/* 0x0014            */         cmp     %o0,100
/* 0x0018            */         bl,a    .L900000109
/* 0x001c            */         add     %o0,%o1,%o1
```

FIG. 6 – *Sortie assembleur du code la figure 5*

1994 d'une bibliothèque (collectivement nommée « Standard Template Library ») de containers et d'algorithmes interfacés par des itérateurs, proposée par Alexander STEPANOV a provoqué une sorte de révolution dans l'approche C++ de la programmation. Toute la bibliothèque Standard C++ a été revisée dans l'esprit template. La STL a inauguré ce que l'on appelle maintenant dans la communauté C++ « le polymophisme statique ». Les difficultés rencontrées, durant l'implémentation de cette bibliothèque, ont fait naître de nouvelles techniques. Celles-ci permettent de contruire des programmes C++ meilleurs en performance que leurs analogues FORTRAN [10]. Ce qui laisse espérer que C++ devrait bientôt constituer le langage par excellence dans la communauté scientifique. Des projets en ce sens voient le jour (cf. [7], [10]).

3.1 Les idées novatrices de la STL

Ce qui est maintenant connu sous le nom de STL est la troisième tentative[5] d'Alexander STEPANOV, depuis les années 70, de la réalisation de l'idée que certains algorithmes sont génériques ou universels en ce sens qu'ils n'ont pas besoin

5. Les deux premières implémentations furent réalisées en ADA et SCHEME ; mais elles ne satisfaisaient pas l'auteur car il manquait à ces langages la notion de « template ».

de connaître les types des données ni les structures de données sur lesquels ils travaillent. Ces algorithmes reposent plutôt sur quelques propriétés sémantiques fondamentales des structures de données : par exemple, aller d'un élément au suivant, parcourir une structure de données du début à la fin. Évidemment un accent particulier est mis sur l'efficacité. Un bénéfice immédiat est le suivant : un programme comprenant T types de données (`int`, `double`,...), S structures de données (`tableaux`, `listes simplement chaînées`,...) et A algorithmes différents (tris, recherche...) requiert $T * S * A$ implémentations différentes : tri de tableaux d'`int`, tri de tableaux de `double`... En utilisant des structures de données paramétrées par les types de données, il ne reste plus que $S * A$ implémentations différentes. L'étape décisive est de paramétrer les algorithmes par les structures de données : un algorithme de recherche doit pouvoir travailler avec un tableau comme une liste. Il en résulte qu'il ne reste plus que $S + A$ implémentations différentes à écrire.

Fondamentalement la STL comprend cinq parties : les containers, les itérateurs, les algorithmes, les objets fonctionnels, les adapteurs.

3.1.1 Les containers

Un container est un objet qui contient une collection d'objets d'un certain type : *liste*, *tableaux*, *tas*, *tableaux associatifs*... La philosophie retenue dans la conception de la bibliothèque Standard est que ces containers devraient être logiquement interchangeables et que c'est l'efficacité qui devrait constituer le critère de choix. Pour que les algorithmes puissent fonctionner de manière quasi-indépendante des structures de données, les containers doivent réduire au minimum le nombre de fonctions d'accès et fournir un nombre assez réduit de propriétés sémantiques foncdamentales. Le pont entre les containers et les algorithmes est réalisé par les itérateurs.

3.1.2 Les itérateurs

Le concept d'itérateur est une abstraction de la notion de pointeur. Ainsi, est considéré comme itérateur, tout objet `i` pour le lequel l'expression `*i` est définie et ayant pour valeur un objet d'une classe, d'une énumération ou d'un type prédéfini[6]. Les itérateurs sont classés en cinq catégories (hiérarchisées) suivant les contraintes qu'ils doivent remplir. Il est à noter qu'aucune de ces catégories d'itérateurs ne correspond à un type (ou objet) spécifique mais plutôt à un concept : une

6. Ou de manière équivalente, on peut définir un itérateur comme étant tout objet se comportant comme un itérateur, en prenant soin de donner un exemple d'itérateur : un pointeur.

classe d'objets sera dite un itérateur de telle catégorie si elle remplit les conditions spécifiques de cette catégorie.

Itérateur d'entrée (*input iterator*) Un itérateur d'entrée doit permettre :

- la lecture (`*p`),
- l'incrémentation (`++p`, `p++`),
- l'accès (`->`),
- et la comparaison (`==`, `!=`).

Itérateur de sortie (*output iterator*) Un itérateur de sortie doit permettre :

- l'écriture (`*p=`),
- et l'incrémentation (`++p`, `p++`).

Itérateur unidirectionnel (*forward iterator*) Un itérateur unidirectionnel satifait les conditions d'itérateur d'entrée et de sortie. Le but avoué est de pourvoir visiter chaque élément d'une suite au plus une fois. Il permet donc :

- la lecture (`*p`),
- l'écriture (`*p=`),
- l'accès (`->`),
- l'incrémentation (`++p`, `p++`),
- et la comparaison (`==`, `!=`).

Itérateur bidirectionnel (*bidirectional iterator*) Un itérateur bidirectionnel remplit les conditions d'itérateur unidirectionnel. De plus il doit permettre la décrémentation (`--`). Sémantiques :

- la lecture (`*p`),
- l'écriture (`*p=`),
- l'accès (`->`),

– l'incrémentation (`++p`, `p++`),

– décrémentation (`--p`, `p++`),

– et la comparaison (`==`, `!=`).

Itérateur à accès direct Un itérateur à accès direct remplit les conditions d'itérateur bidirectionnel, et de plus permet :

– l'addition et l'addition augmentée (`+`, `+=`),

– la soustraction et la soustraction augmentée (`-`, `-=`),

– l'indexation (`[]`),

– et la comparaison d'inégalité (`<`, `>`, `<=`, `>=`).

Un pointeur est un itérateur à accès direct.

3.1.3 Les algorithmes

Un algorithme, afin d'être le plus générique possible, ne travaille pas directement sur les structures de données mais plutôt avec des itérateurs et des objets fonctionnels. Cela est possible car chaque container fournit une certaine catégorie d'itérateurs pour accès. Cette approche permet à tout utilisateur de disposer des algorithmes de la bibliothèque Standard à condition de fournir les itérateurs adéquats. C'est cela la clé du succès de la STL.

3.1.4 Les objets fonctionnels

Un objet d'une classe pour lequel l'opérateur d'application (ou d'appel de fonction) est défini est appelé objet fonctionnel ou foncteur. Ces objets ont été motivés par des soucis d'efficacité. En effet certains algorithmes (de tri, de recherche ou autres) nécessitent des fonctions en paramètres. Or le seul moyen en C/C++ de passer une fonction en paramètre est de passer son adresse. Il est alors très difficile d'acyiver le mécansime « inline » dans ce cas. À l'opposé, il est plus facile dre rendre « inline » les fonctions membres d'une classe. L'idiome consiste à créer une classe pour une opération donnée et à définir « inline » l'opérateur d'application.

3.1.5 Les adapteurs

Les adapteurs sont des objets qui construisent de nouveaux objets (d'une certaine catégorie) à partir d'autres objets de même catégorie. Par exemple un « négateur » est un adapteur qui prend en paramètre un prédicat et rend un objet fonctionnel correspondant à la négation du prédicat original.

3.2 La programmation générique

Comme nous le disions précédemment, la STL a montré le chemin de la programmation générique en C++ : implémentation d'algorithmes indépendants des types de données. Grâce au mécanisme de spécialisation partielle, il est possible de fournir une interface uniforme d'implémentations générale et spécialisée d'un algorithme. Considérons, par exemple, la tâche assez courante de recopie de données, tâche implémentée par la fonction `Copy` :

```
template<class InputIterator, class OutputIterator>
inline void Copy(InputIterator first, InputIterator last,
                 OutputIterator res)
{ while (first != last) *res++ = *first++; }
```

La fonction `Copy`, telle qu'implémentée, est assez générale : elle fonctionnera sur un éventail très large d'itérateurs. Elle peut effectuer des copies entre listes et tableaux. Supposons manitenant que l'on désire faire des copies entre tableaux, il est peut-être plus efficace d'utiliser la fonction bibliothèque `memmove`. Dans ce cas la fonction `Copy` peut être implémentée de la manière suivante :

```
template<class T>
inline void Copy(T* first, T* last, T* res)
{ memmove(res, first, (last-first)*sizeof(T)); }
```

Cette spécialisation permet d'utiliser la version optimisée lorsque `Copy` est appelée avec des pointeurs en arguments et la version générale lorsque la copie se fait entre deux structures de données différentes.

3.3 Les traits

Les traits sont une invention de Nathan MYERS dans le cadre de la politque générique des flots (`iostream`) et de la classe `string` de la bibliothèque standard C++. Le lecteur intéressé par la genèse de ce concept est renvoyé à [6]. De manière élémentaire, c'est une tentative de mécanisation de l'idée qu'une fonction devrait « savoir », à partir des opérations qu'elle effectue, le type de sa valeur de retour. Prenons le cas d'une matrice dont les cofficients sont de type `double` (le cas de coefficients génériques est immédiat). Il existe au moins deux représentations pour un tel type de donnée : une représentation dense et une creuse. En général l'addition d'une matrice creuse et d'une matrice dense est une matrice dense. Ceci est le cas de la plupart des opérations linéaires entre les deux types de matrices. La portion de code (figure 7) suivant peut être l'ébauche d'une implémentation. Ainsi lors des instantiations le compilateur, ayant été instruit par la classe `Structured-MatrixTraits`, choisira le type approprié pour la valeur de retour de l'opérateur `operator+`.

3.4 Les métaprogrammes

Nous avons vu dans la section précédente une illustration du caractère « interprète » du compilateur C++. Les métaprogrammes (terme inventé par Todd VELDHUISEN) exploitent de manière essentielle cet aspect du complateur C++, i.e. un métaprogramme est un programme (template) C++ qui à la compilation instruit le compilateur de générer du code qui sera compilé. Récemment Dan PIPONI a posté sur le forum de discussion `comp.lang.c++.moderated` un source code C++ qui implémente, sous forme de métaprogrammes, l'ensemble des entiers naturels à partir des axiomes de Peano et en déduit la primalité de l'entier 13 *sans aucun calcul* juste en se basant sur le système de type du compilateur [7]

Supposons que l'on travaille avec des vecteurs dont les tailles sont connues à la compilation. Il est normal d'espérer que le produit scalaire de vecteurs soit partiellement déroulé pendant la compilation afin d'éliminer les délais de branchement qu'imposerait une longue boucle pendant l'exécution (bien sûr il est possible de dérouler la boucle à la main, mais honnêtement, est-ce à l'homme d'être au service de la machine ?). Ce que nous voudrions c'est qu'une suite d'expressions :

7. Des programmes similaires, calculs de nombre premier pendant la compilation, étaient déjà connus depuis 1994 mais l'originalité réside ici dans le codage des axiomes de Peano en utilisant les templates et le système de typage de C++ et également l'absence de calculs.

```
enum Structure { Dense, Sparse };
template<Structure s> class StructuredMatrix;
struct Warning_Return_Type_Not_Defined {};
template<Structured s, Structured u>
struct StructuredMatrixTraits {
    typedef Warning_Return_Type_Not_Defined matrix_type;
};
template<> struct StructuredMatrixTraits<Dense, Dense> {
    typedef Dense matrix_type;
};
template<> struct StructuredMatrixTraits<Dense, Sparse> {
    typedef Dense matrix_type;
};
template<> struct StructuredMatrixTraits<Sparse, Dense> {
    typedef Dense matrix_type;
};
template<> struct StructuredMatrixTraits<Sparse, Sparse> {
    typedef Sparse matrix_type;
};
template<> class StructuredMatrix<Dense> { ... };
template<> class StructuredMatrix<Sparse> { ... };
template<Structure s, Structure u>
inline typename StructuredMatrixTraits<s, u>::matrix_type
operator+(const StructuredMatrix<s>& a,
          const StructuredMatrix<u>& b)
{
    typedef typename
    StructuredMatrixTraits<s, u>::matrix_type matrix_type;
    return matrix_type(a) += b;
}
```

FIG. 7 – *Ébauche d'interface de matrices structurées.*

```
Vector a(N), b(N);
double s = dot(a, b);
```

soit transformée en :

```
double a[N], b[N];
double s = a[0]*b[0] + a[1]*b[1] + ... + a[N-1]*b[N-1];
```

où `N` désigne la longueur des vecteurs, constante connue à l'heure de la compilation. Comment coder cela sous forme de métaprogramme ? Il suffit d'utiliser le vieux truc de conversion récursion terminale – boucle `for` (figure 8). Comme on

```
template<size_t n, size_t i> struct DotProduct {
    static double Expand(double a[], double b[])
    { return a[i]*b[i] + DotProduct<n, i+1>::Expand(a, b); }
};
template<size_t n> struct DotProduct<n, n-1> {
    static double Expand(double a[], double b[])
    { return a[n-1] * b[n-1]; }
};
inline double dot(double a[], double b[])
{ return DotProduct<N, 0>::Expand(a, b); }
inline double dot(double a[], double b[]) {
    double s(0);
    for (register size_t i=0; i<N; ++i) s += a[i] * b[i];
}
double a[16], b[16];
double s = dot(a, b);
```

FIG. 8 – *Produit scalaire par métaprogramme.*

peut le constater sur le code assembleur généré (figure 9) par le compilateur EGCS-1.0.1 [1], le produit scalaire est effectivement déroulé ! Comparer avec la version « naive » compilée avec la même option d'optimisation : Codes assembleurs pour `N` valant 4. Ainsi le produit scalaire métaprogramme s'effectue en 22 cycles. Ci-

```
! Produit scalaire métaprogramme
        sethi %hi(a),%o1
        or %o1,%lo(a),%g3
        sethi %hi(b),%o0
        or %o0,%lo(b),%g2
        ldd [%g3+24],%f6
        ldd [%g2+24],%f2
        fmuld %f6,%f2,%f6
        ldd [%g3+16],%f4
        ldd [%g2+16],%f2
        fmuld %f4,%f2,%f4
        faddd %f4,%f6,%f4
        ldd [%g3+8],%f6
        ldd [%g2+8],%f2
        fmuld %f6,%f2,%f6
        faddd %f6,%f4,%f6
        ldd [%o1+%lo(a)],%f2
        ldd [%o0+%lo(b)],%f4
        fmuld %f2,%f4,%f2
        faddd %f2,%f6,%f2
        sethi %hi(s),%g2
        retl
        std %f2,[%g2+%lo(s)]
```

FIG. 9 – *Assembleur généré pour le code de la figure 8.*

dessous le source assembleur généré (figure 10) par EGCS correspondant à la manière traditionnelle. Option d'optimisation : `-O`. cette portion de code (logiquement

```
! Porduit scalaire traditionnel
        sethi %hi(a),%g2
        or %g2,%lo(a),%o1
        sethi %hi(b),%g2
        or %g2,%lo(b),%o0
        sethi %hi(.LLC1),%o2
        ldd [%o2+%lo(.LLC1)],%f6
        mov 0,%g3
.LL12:
        sll %g3,3,%g2
        ldd [%o1+%g2],%f2
        ldd [%o0+%g2],%f4
        fmuld %f2,%f4,%f2
        add %g3,1,%g3
        cmp %g3,3
        bleu .LL12
        faddd %f6,%f2,%f6
        sethi %hi(s),%g2
        retl
        std %f6,[%g2+%lo(s)]
```

FIG. 10 – *Code assembleur du produit scalaire normal*

équivalent au précédent) prend au moins 42 cycles !

3.5 Les templates d'expressions ou expressions formelles

Entre 1994 et 1995 Todd VELDHUIZEN et David VANDEVOORDE ont inventé, indépendemment, une technique permettant de contourner les problèmes des objets temporaires dûs à l'évaluation par paire des opérations en C++. Cette technique est maintenant connue sous le nom de « template expressions » dans la littérature anglo-saxonne. Nous la désignerons par le nom d' « expressions formelles ». L'idée est de bâtir *statiquement*, i.e. pendant la compilation, l'arbre d'une expression. Pour ce faire chaque expression algébrique sera représentée par un type unique

(une classe) et les opérateurs seront surchargés pour retourner des objets de ces types. Afin d'illustrer cette technique d'expressions formelles nous allons considérer le cas d'évaluation d'une expression mathématique en des points différents (problème rencontré en interpolation). En général ce problème s'identifie comme un « callback ». En C++ (comme en C) on ne peut passer ou retourner une fonction comme valeur. Par contre il est possible de « prendre » l'adresse d'une fonction. Donc le problème des callbacks se résolvait en passant l'adresse de la fonction victime. Dans ce schéma il n'y a donc pas de possibilité qu'une fonction bénéficie du mécanisme « inline ». En exploitant l'idée des objets fonctionnels, les expressions formelles offrent la possibilité de passer l'expression à évaluer, telle qu'on l'aurait écrite, en paramètre à l'évaluateur. Cette solution permet d'activer le mécanisme d'expansion pour les fonctions déclarées « inline ». Utilisées dans cet esprit, les expressions formelles appraraissent comme des spécialisations des objets fonctionnels de la bibliothèque Standard. En réalité elles sont plus que cela. Nous illustrons à travers le programme listé dans les figures 11-13 une utilisation possible (assez simpliste il est vrai, mais l'exemple est choisi uniquement pour des propos pédagogiques) des expressions formelles. Puisque les opérateurs utilisés doivent être surchargés et qu'une déclaration brutale

```
template<class T> operator@(T, T);
```

est trop générale pour pouvoir être intéressante, nous avons besoin de restreindre leurs portées. C'est le rôle de la classe générique `Meta<>`. La classe `PlaceHolder<>` aide à simuler les variables symboliques sans ajouter de coûts inutils. La classe `Constant<>` est nécessaire pour traiter des expressions contenant des constantes. Enfin nous avons déjà rencontré les classes `Plus<>`, `Multiplies<>`, `Divides<>` : elles supportent le mécanisme d'objets fonctionnels.

On vérifie (grâce au source assembleur) que la fonction `eval` a subi une expansion en ligne ainsi que l'expression `1/(x*x+1)`. Ce qui est quasi impossible si l'idée d'objets fonctionnels n'est pas utilisée. Grâce à la technique d'expression formelle, les appels de fonctions sont supprimés. On obtient ainsi la performance et la lisibilité.

```
#include <iostream.h>
template<class Expression> class Meta {
public:
    typedef typename Expression::value_type value_type;
    Meta(const Expression& e) : expr(e) {}
    value_type operator()(value_type x) const
    { return expr(x); }
private:
    Expression expr;
};
template<class T> class PlaceHolder {
public:
    typedef T value_type;
    PlaceHolder() {}
    T operator()(T x) const { return x; }
};
template<class T> class Constant {
public:
    Constant(T x) : value(x) {}
    value_type operator()(value_type) const
    { return value; }
private:
    T value;
};
template<class E1, class E2> class Plus {
public:
    typedef typename E1::value_type value_type;
    Plus(const E1& e1, const E2& e2) : expr1(e1), expr2(e2) {}
    template<class T> value_type operator()(T x) const
    { return expr1(x) + expr2(x); }
private:
    const E1& expr1;
    const E2& expr2;
};
```

FIG. 11 – *Illustration d'expressions templates et métaprogrammes.*

```
template<class E>
inline Meta<Plus<Meta<E>, Constant<typename E::value_type> > >
operator+(const Meta<E>& e, typename E::value_type x) {
    typedef typename E::value_type T;
    typedef Plus<Meta<E>, Constant<T> > expr_t;
    return Meta<expr_t>(expr_t(e, Constant<T>(x)));
}
template<class E1, class E2> class Multiplies {
public:
    typedef typename E1::value_type value_type;
    Multiplies(const E1& e1, const E2& e2)
       : expr1(e1), expr2(e2) {}
    template<class T> value_type operator()(T x) const
    { return expr1(x) * expr2(x); }
private:
    const E1& expr1;
    const E2& expr2;
};
template<class E1, class E2> inline
Meta<Multiplies<Meta<E1>, Meta<E2> > >
operator*(const Meta<E1>& e1, const Meta<E2>& e2) {
    typedef Multiplies<Meta<E1>, Meta<E2> > expr_t;
    return Meta<expr_t>(expr_t(e1, e2));
}
template<class E1, class E2> class Divides {
public:
    typedef typename E1::value_type value_type;
    Divides(const E1& e1, const E2& e2)
       : expr1(e1), expr2(e2) {}
    template<class T>
    value_type operator()(T x) const
    { return expr1(x) / expr2(x); }
private:
    const E1& expr1;
    const E2& expr2;
};
```

FIG. 12 – *Illustration d'expressions templates et métaprogrammes (suite).*

```
template<class E> inline
Meta<Divides<Constant<typename E::value_type>, Meta<E> > >
operator/(typename E::value_type x, const Meta<E>& e) {
    typedef typename E::value_type T;
    typedef Divides<Constant<T>, Meta<E> > expr_t;
    return Meta<expr_t>(expr_t(Constant<T>(x), e));
}
template<class Expression>
inline typename Expression::value_type
eval(Expression e,
     typename Expression::value_type start,
     typename Expression::value_type end,
     typename Expression::value_type step)
{
    typedef typename Expression::value_type T;
    for (T i=start; i<end; i += step)
        cout <<  e(i) << '\n';
}
int main(){
    Meta<PlaceHolder<double> > x = PlaceHolder<double>();
    eval(1/(x*x+1), 0., 1., .1);
}
```

FIG. 13 – *Illustration d'expressions templates et métaprogrammes (fin) : passage d'expression à évaluer sans callback.*

4 Applications

Les idées développées ci-haut peuvent être exploitées pour implémenter le BLAS niveau 1 en suchargeant les opérateurs sans perte d'efficacité due aux objets temporaires. Ainsi des lignes de code comme

```
Vector<double> a(N), b(N), c(N), d(N);
d = a + b + c;
```

se traduisent en

```
for (register int i=0; i<N; ++i)
    d[i] = a[i] + b[i] + [ci];
```

sans temporaires, avec possibilité de déroulement de la boucle.

Nous rappelons que la bibliothèque Standard C++ définit une classe `valarray` dédiée au calcul numérique à la BLAS. Aucune implémentation de cette classe, entièrement conforme à la norme, n'est connue de l'auteur à l'heure où nous écrivons cet article. Une implémentation, encore incomplète, a été proposée par l'auteur pour inclusion dans le projet EGCS. La bibliothèque Blitz++ [10] de Todd VELDHUIZEN est entièrement construite sur ces idées. Les performances qu'il obtient sont encourageantes. Actuellement ces techniques sont bien indiquées pour le BLAS niveau 1 mais pas le niveau 2. Nous avons obtenu une adaptation au niveau 2 mais avec des restrictions sur les classes (il ne doit pas y avoir d'alias). Nous espérons pouvoir nous affranchir de ces contraintes. Peut-être faudra-t-il creuser un peu plus la philosophie des expressions templates.

Références

[1] EGCS, 1997, `http://www.cygnus.com/~egcs/`.

[2] John R. CARY and Svetlana G. SHASHARINA, *Comparison of C++ and* FORTRAN *90 for Object-Oriented Scientific Programming*, Computer Physics Communications (November 1996).

[3] Andrew KOENIG and Barbara MOO, *Ruminations on C++: a decade of programming insight and experience*, Addison Wesley, 1997.

[4] Stanley B. LIPPMAN, *Inside The C++ Object Model*, Addison Wesley Publishing Company, 1996.

[5] Scott MEYERS, *More Effective C++*, Addison Wesley, 1996.

[6] Nathan MYERS, *Traits: a new and useful template technique*, C++ Report (June 1995), `http://www.cantrip.org/traits.html`.

[7] Roldan POZO, TNT*: Template Numerical Toolkit*, `http://http://math.nist.gov/tnt/`.

[8] Arch D. ROBINSON, *C++ Gets Faster for Scientific Computing*, Computers in Physics **10** (1996), 458–462, `http://www.kai.com/publications/comp_phys/`.

[9] Bjarne STROUSTRUP, *The C++ Programming Language*, 3rd ed., Addison Wesley, 1997.

[10] Todd VELDHUIZEN, *Blitz++*, `http://monet.uwaterloo.ca/blitz/`.

[11] ———, *Using C++ template metaprogramm*, C++ Report (May 1995), no. 4, 36–43.

[12] ———, *Scientific Computing: C++ vs. Fortran*, Dr. Dobb's Journal (November 1997).

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 46 avenue F´elix Viallet, 38031 GRENOBLE Cedex 1
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399