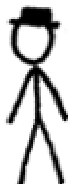
C++ Avancé - STL - Bibliothèques scientifiques

LJK

27 janvier 2012

MaiMoSiNE - Réseau calcul


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

Plan

Introduction

- C n'est pas un sous-ensemble de C++ : certains programmes C ne sont pas valides en C++.
- Certains programmes ont un sens différent en C et C++.
- Toutes les techniques de programmation valable en C sont également valable en C++.
- Les programmes biens écrit en C sont valables en C++.
- Un programme C convertit en C++ s'exécutera aussi rapidement dans les 2 cas, et auront la même empreinte mémoire.
- C++ supporte des paradigmes multiples : programmation procédurale, programmation objet et programmation générique.
- Il est possible de n'utiliser qu'un seul paradigme pour écrire un code C++ : ce choix se fait souvent au détriment de la maintenabilité et de l'élégance.

Frank and Ernest
OH WOW! PARADIGM SHIFT!

L'objet

- Le `C++` supporte tous les éléments de la programmation objets :
  - Encapsulation des données : les données peuvent avoir des données de visibilités différentes (publique, privée ou protégée). Avec le `C++`, l'encapsulation est une possibilité, elle n'est jamais forcée.
  - Abstraction : mécanisme permettant de réduire le niveau de détail d'un code. On regroupe les classes selon des caractéristiques communes. Une abstraction bien conçue est généralement simple et s'utilise facilement.
  - Héritage : les propriétés et les fonctionnalités d'une classe existante peuvent être transmise par définition à une autre classe. Les principaux concepts sont l'extensibilité et la reutilisabilité.
  - Polymorphisme : capacité de manipuler à l'exécution des objets en fonction de leur type et de leur utilisation. Le polymorphisme peut-être obtenu soit à la compilation en utilisant les surcharges d'opérateurs et de fonctions, soit à l'exécution en utilisant des fonctions virtuelles.

Les concepts avancés du C++

Afin de mettre en oeuvre les techniques de la programmation orientée objet, le C++ utilise les techniques suivantes :

- Les classes et la surcharge d'opérateur et de fonctions pour le polymorphisme
- Les espaces de noms.
- Les exceptions qui permettent de transférer le contrôle du programme à des fonctions spécifiques.
- La conversion de types complexes.
- Les signaux.
- Flux.
- La gestion de la mémoire dynamique.
- Patrons.

Plan

Pour quoi faire ?

les patrons : réfléchir plus pour travailler moins :-)

- pour les types de bases (`double`, `int`,...), et les types plus complexes (`point`, `vecteur`,...), on est souvent amené à implémenter les mêmes algorithmes avancés : `sort`, `quicksort`, `bubblesort`,...
- ces algorithmes requièrent l'implémentation, pour le type considéré, des opérateurs de comparaisons. l'algorithme est en tout point identique, quelque soit le type utilisé.
- il est possible d'effectuer l'implémentation en utilisant 3 techniques différentes :
  - on répète l'implémentation pour chaque type : il est possible, mais fort peu probable d'obtenir du code plus performant, on augmente le risque d'erreur, c'est fastidieux.
  - on écrit un code générique pour une classe de base, de type `void` par exemple : on ne peut plus vérifier le type des objets, on obtient un code difficile à maintenir.
  - on utilise les commandes de preprocessor : on perd la structure du code et on perd la notion de localisation en ignorant les portées et les types
- les patrons (`template`) permettent de remplacer ces techniques.

Mise en place

Un patron se compose de plusieurs éléments

1. le mot clé **template** : indique que nous allons définir un patron pour une fonction, une structure ou une classe. Le mot clé est suivi des chevrons $<$ et $>$.
2. les chevrons $<$ et $>$: indiquent la liste de type utilisé dans le patron.

   Attention à séparer les chevrons imbriqués par des espaces afin d'éviter la confusion avec les opérateurs $<<$ ou $>>$.
3. un type :
   - en utilisant le mot clé **typename** ou **class** : indique que le type utilisé est abstrait. Le compilateur aura la charge de déterminer le type correct de la variable à la compilation.
   - en utilisant un des types existant suivant :**int**, un pointeur, une référence.
   - un patron
4. un symbole alpha-numérique : désigne le type .

- Exemple :

```
template <typename T1, typename T2,...>
```
1

Plan

Macro

- On peut dans un premier temps faire de la surcharge : par exemple, pour la fonction `min`

```
inline int min(int a, int b){return (a<b?a:b);}
inline double min(double a, double b){return (a<b?a:b);}
```

- Il faut écrire autant de fonctions que l'on a de types, mais on a
  - la vérification de type sur le code généré
  - une optimisation possible des performances avec l'utilisation de **`inline`**
  - la possibilité de créer des versions alternatives pour des types plus complexes
- On peut donc utiliser l'implémentation suivante

```
#define min(a,b) ( (a<b ? a : b))
```

- Le code n'est pas dupliquer, il est performant, mais on pourrait éventuellement
  - comparer un `string` et un **`double`**.
  - avoir des effets de bords.
- Les patrons permettent d'éviter ce genre de problème.

Patron pour la fonction `min`

- Le patron de fonction correspondant est

```
template <typename T>
inline const T & min(const T & a, const T &b)
{ return (a<b ? a :b); }
```

Patron de fonctions

- Syntaxe générale d'une fonction patron

```
template <typename T1,typename T2,...>
type_retour nom_fonction(type_arg1, type_arg2,...)
```

- Les types abstraits `T1`, `T2`, ... doivent tous intervenir dans la liste des arguments d'entrées et peuvent intervenir dans l'argument de sortie :
  le compilateur n'analyse que les arguments de la signature !

```
template <typename T>
 T fonc(int &i) {return T(i);}
int main(int argc, char *argv[])
{
 double x=fonc<int>(2);
}
```

Patrons de fonctions : passage de type

- On peut généraliser cette fonction de conversion en ajoutant un type abstrait supplémentaire :

```
template <typename T1, typename T2>
T1 fonc(T2 & x)
```

- Syntaxe équivalente

```
template <typename T1><typename T2>
T1 fonc(T2 & x)
```

- On peut remplacer `typename` par `class`

```
template <class T1><class T2>
T1 fonc(T2 & x)
```

Patrons : fonction minimum

- Un patron peut-être surchargé

```
template<typename T>
T min(const T& a,const T& b){return (a<b?(a):(b);}
template<typename T>
T min(const T& a,const T& b,const T& c)
  {return min(min(a,b),c);}
```

- Ces patrons sont compatibles avec tout les types et les classes supportant l'opérateur $<$
- La fonction `min` peut également être définie entre types différents

```
template<typename T1, typename T2>
T1 min(const T1 &a,const T2& b){return ((a<b)?(a):T1(b));}
```

- Cette fonction suppose un transtypage consistant du type `T2` vers le type `T1` détection à la compilation.

  Il est fortement déconseillé d'utiliser ce type de fonctionnalité

```
min(2.5,3) = 2.5 et min(3,2.5) = 2
```

Limitations

- On ne peut pas avoir les deux versions suivantes de `min`

```
template <typename T1, typename T2>       1
  T1 min(T1 & a,T2 & b)                    2
template <typename T>                      3
  T min(T & a,T & b)!                      4
```

- Le compilateur n'effectue pas la conversion automatique des variables.

```
template<typename T>                                       1
  T min(const T& a, const T& b){return ((a<b)?(a):(b));}   2
int main(int argc,char *argv[])                            3
{ int i = 2; double b = 1.9;                               4
  min(i,b);                                                5
  return 0; }                                              6
```

- Pour les chaînes de caractères `char *`, on doit redéfinir la fonction `min`

```
const char *min(const char *l, const char *g)   1
{ if (strcmp(l, g) > 0) return g;               2
  else return l; }                              3
```

- La version dédiée est toujours choisie

Plan

Classe patron

- Généralisation du concept de patron de fonctions aux classes

```
template<typename T1, typename T2,...>          1
class nom_classe{...};                          2
```

- On peut alors utiliser dans la classe les types abstraits `T1`, `T2`, ... comme des types standards

```
template <typename T> class vect                                  1
{                                                                 2
 public :                                                         3
   T* val;                                                        4
   int dim;                                                       5
   vect<T>(const int d=0)                                         6
   { dim d;                                                       7
     if(d>0) { val = new T[d]; } }                                8
   T& operator()(const int i)                                     9
   { if(i>0 && i<= dim) { return val[i-1]; //Erreur } }           10
```

- On obtient une classe gérant des vecteurs de tout type.
- Certains types seront incompatibles avec les opérations prévues

Classe patron : instanciation

- L'instanciation d'un objet de la classe `vect` est réalisé en précisant le type abstrait `T`

```
int main()
{
  vect<double> V(2);
  V(1) = 1.;
  V(2) = 2.;
  vect<char *> L(2);
  L(1) = "chaine␣1";
  L(2) = "chaine␣2";
}
```

- A la compilation, le compilateur
  1. recherche le type `vect<double>`
  2. trouve `template<typename T> class vect`
  3. génère le code explicite en remplaçant `T` par `double`
  4. compile le code généré
  5. recherche le constructeur `vect<double>(int)`
- Type composé : `vect< vect<double> > M(2);` pour un vecteur de vecteur réel (matrice réelle)

Classe patron : champ opératoire

- Le patron est une technique d'abstraction permettant de manipuler des objets complexes en se focalisant sur les opérations de structure
- Comment définir les opérations de structure ?
- Exemple avec l'opérateur `+=` pour la classe `vect`

```
template <typename T> class vect
{
 public :
 ...
 vect<T>& operator+=(const T& V)
 {
   for(int i=0;i<dim;i++)
   {
     val[i]+=V.val[i];
     return *this;
   }
 }
```

- La classe `vect` ne pourra plus gérer des vecteurs de `char*` car l'opérateur `+=` n'est pas défini pour le type `char *`
- Choix de conception : quel est le niveau d'abstraction souhaité par le concepteur ?

Classe patron : membre patron

- Fonction membre générique dans une classe
- Exemple : produit par un scalaire quelconque d'un `vect`

```
template<typename T> class vect                                    1
{                                                                  2
 public :                                                          3
 ...                                                               4
 template<typename S> vect<T>& operator*=(const S& s)              5
 {                                                                 6
  for(int i=0;i<dim;i++)                                           7
  {                                                                8
   val[i]*=s;                                                      9
  }                                                               10
   return *this;                                                  11
  }                                                               12
 }                                                                13
```

- Plus général que `vect<T>& operator*=(const T& s);`
- Le type abstrait `S` est spécifique à l'opérateur `*=`

Problèmes avec des membres patrons

- Exemple avec des nombres complexes

```
vect<double> V(2);
bool cas_Complexe = false;
if(cas_Complexe)
{
  V*=complex<double>(0,1);
}
else
{
  V*=2;
}
```

- Erreur de compilation `double*=complex<double>` impossible même si on ne passe pas dans le cas complexe.

Instanciation

Il y a deux modes d'instanciation d'un patron

- implicite : réalisé si possible par le compilateur
- explicite : réalisé par l'utilisateur
  - pour une fonction

    ```
    int i=min<int>(2,3.);
    ```

    force l'instanciation de `template <class T> min(T &,T &)`
  - pour une classe

    ```
    vect<double>
    ```

    force l'instanciation de la classe `vect` en `double`

Spécialisation

- On peut être amené à définir des cas particuliers de patrons afin de prévenir d'une mauvaise utilisation de patron ou réaliser un traitement spécifique.

```
template<typename T> class vect
{
 public :
 ...
 template <typename S> vect<T>& operator*=(const S& s)
 {
  for(int i=0;i<dim;i++) { val[i]*=s; }
  return *this;
 }
};

vect<double>& operator*=(const complex<double>& s)
{ //traitement specifique }
```

- Redéfinition d'une instance particulière de l'opérateur `*=`
- Permet de résoudre le problème de compilation évoqué : il existe une version `double*=complexe` admissible.

Spécialisation : détails

- Redéfinissons la classe `Point` en utilisant 2 paramètres de patron :

```
template <typename T,std::size_T N>
classPoint
{
 public:
   Point();
 private:
   T val[N];
};
```

- Le premier type du patron de la classe `Point` est abstrait. Il permet de dire si l'on travaille avec des entiers, des réels ou des complexes.
- Le second type du patron de la classe `Point` est un entier : donner une valeur à cet entier permet de définir la dimension du point.
- Afin de travailler sur des points de double en dimension 2, on utiliserait une spécialisation de la classe `Point` :

```
template <>
class Point<double,2>
{ //code specifique }
```

Spécialisation : moins, c'est plus

- Parfois, on ne peut vouloir spécialiser qu'une fonction de la classe. Par exemple, la projection d'un point parallèlement à un hyperplan aura des formes différentes en 2D et 3D. Dans ce cas on utilisera la formulation suivante

```
template<> Point<double,2>::projection();
```

- En suivant la même idée, on peut réaliser une spécification partielle de la classe `Point`. Pour déclarer un point en 2D, on utiliserait une spécialisation partielle

```
template<typename T> class Point<T,2>
{
  //code specifique
}
```

- Une fonction ne peut pas être spécialisée partiellement, que ce soit une fonction simple ou une fonction membre.

Implémentation séparée : syntaxe

Implémentation dissociée de la définition (syntaxe)

- pour les fonctions, même syntaxe que la déclaration
- pour les fonctions membres d'une classe patron

```
template<typename T1,typename T2,...>
arg_retour
 nom_classe<T1,T2,...>::nom_fonction(arg_entree1,...)
{...}
```

- Pour les fonctions membres patron d'une classe patron

```
template<typename T1,typename T2,...>
template<typename S1,typename S2,...>
arg_retour
 nom_classe<T1,T2,...>::nom_fonction(arg_entree1,...)
{...}
```

- Les types des patrons de la classe et ceux de la fonction sont dans deux templates distincts.

Implémentation séparée : le fichier de définition

- Comme dans le cas normal, on utilise un fichier séparé pour définir la classe.

```
#ifndef VECT_H
#define VECT_H

template <typename T>
class vect
{
 public :
   T* val;
   int dim;

   vect<T>(const int d=0);

   T& operator()(const int i);

   vect<T>& operator+=(const S& s);
};
#endif
```

Implémentation séparée : le fichier d'implémentation

- On implémente les fonctions dans un fichier séparé, malgré les patrons. On peut le désigner par exemple par l'extension `.txx`
- Dans la norme actuelle, un fichier contenant des patrons ne sera lu par le compilateur uniquement au moment de l'édition finale des liens

```
#include "vect.h"                                                          1
template <typename T>                                                      2
T& vect<T>::operator()(const int i)                                        3
{ if(i>0 && i>=dim) { return val[i-1]; } }                                 4
                                                                           5
template <typename T>                                                      6
vect<T>& vect<T>::operator+=(const T& V)                                   7
{ for(int i=0;i<dim;i++)                                                   8
  { val[i]+=V.val[i]; return *this; } }                                    9
                                                                           10
template <typename T> template <typename S>                                11
vect<T>& vect<T>::operator*=(const S& s)                                   12
{ for(int i=0;i<dim;i++)                                                   13
  { val[i]*=V.val[i]; return *this; } }                                    14
```

Plan

Problèmes de compilation

- Contrairement aux fonctions et aux classes standards, les template doivent impérativement exister lors de la compilation (génération du code de l'instance) : vrai si l'implémentation se trouve dans un header.
  - mélange de la déclaration et de l'implémentation
  - recompilation multiple
  - plusieurs exemplaires du même code dans l'exécutable
- Solutions
  - Les compilateurs proposent tous un mécanisme de précompilation des entêtes pour limiter le problème
  - A l'édition de liens, regroupement des mêmes instances d'un template
  - Mécanisme de gestion des templates par des bases de données
  - Désactivation de l'instanciation automatique.

Problèmes de compilation : pistes

- Pour les codes importants : programmation séparée entête/instanciation et instanciation explicite si le compilateur ne le fait pas.

Options de compilations de g++

- `-fno-implicit-templates` : pas d'instanciation automatique
- `-frepo` : instanciation automatique prise en charge par l'éditeur de liens (génère des fichiers .rpo)
- `-ftemplate-depth-n` : profondeur n dans les templates imbriqués

Et si le compilateur travaillait...

- Une partie du code en C++ est compilée puis exécutée. L'autre partie est interprétée à la compilation.
- Utilisons les patrons pour tirer partie de cette propriété avec le calcul de la factorielle

```
template <int N> class Factorial
{ public :
   static const long valeur=N*Factorial<N-1>::valeur;};
template <> inline long factorial<0>(void)
template <> class Factorial <0>
{ public :
   static const long valeur=N*Factorial<N-1>::1;};
```

- Le calcul est développé à la compilation si un appel explicite est donné

```
y = Factorial<8>(); // le compilateur donne y = 40320
z = Factorial<n>(); // le compilateur ne fait rien
```

- Rq : également réalisable avec la fonction puissance car les puissances sont souvent statiques !

Plan

Concepts

La STL est une collection de conteneurs, d'algorithmes génériques et d'outils associés.

- Classes conteneurs : permet de contenir des objets de n'importe quel type, et de manipuler cette liste (ajouter, enlever, accéder).
- Les itérateurs : sont une abstraction des pointeurs et permettent de parcourir les conteneurs de manière transparente.
- Les algorithmes génériques : liste d'algorithme que l'on peut appliquer sur les conteneurs. Il y a par exemple le trie, la recherche, la fusion, la séparation.
- Les functions

Les conteneurs séquentiels

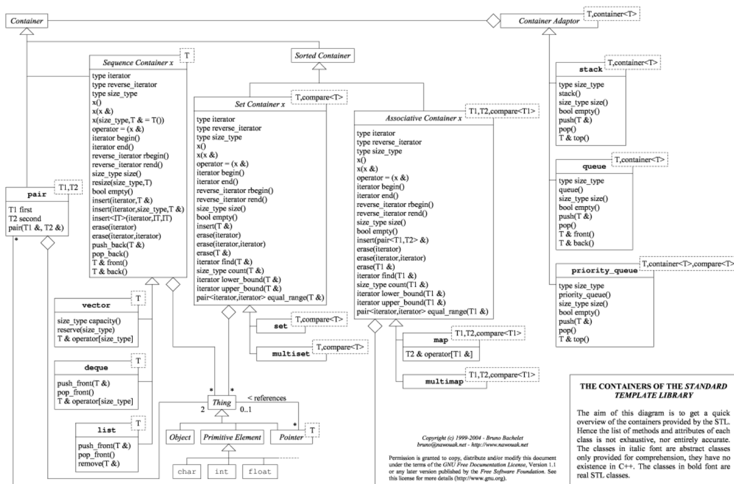
- Contiennent une suite linéaire d'objets. Ils sont de trois types.
- `vector<value_type>` : fourni un accès aléatoires aux objets en temps constant. L'ajout et la suppression d'éléments se réalisent en temps constants à la fin du vecteur et en temps linéaire au milieu du vecteur.
- `deque<value_type>` : accès aléatoire aux objets en temps constant.L'ajout et la suppression d'éléments se réalisent en temps constants à la fin et au début et en temps linéaire au milieu.
- `list<value_type>` : pas d'accès aléatoire. L'ajout et la suppression d'éléments se réalisent en temps constants.
- `stack<value_type>` : adapte le conteneur `deque` pour fournir les fonctionnalités d'une pile.
- `queue<value_type>` : adapte le conteneur `deque` pour fournir les fonctionnalités d'une file.
- `priority_queue<value_type>` : adapte le conteneur `vector` afin de gérer la priorité. L'élément du dessus et le plus grand.

Les conteneurs associatifs

- Permettent de construire des ensembles et des tables de hachage.
- `set<key_type>` : contient une liste de clés unique. Permet de retrouver et de tester rapidement les clés.
- `multi_set<key_type>` : contient une liste de clés éventuellement dupliquées. Permet de retrouver et de tester rapidement les clés.
- `map<key_type,value_type>` : c'est une table de hachage. Contient une suite ordonnées d'un seul type basée sur la valeur des clés.
- `multi_map<key_type,value_type>` : permet d'avoir des clés dupliquées.
- `bitset<size_type>` : contient une collection de bits. Semblables à `vector<bool>` mais est de taille fixe et est optimisé pour les opérations binaires.

- On peut également trouver des types associatifs dérivées dans certaines implémentations de la STL : `hash_basic_type` chez Microsoft.

Aperçu



Les itérateurs

- L'un des principaux éléments de la STL sont les itérateurs. Ils permettent de relier les algorithmes génériques aux conteneurs.
- Tous les algorithmes de la STL ont au moins 1 argument qui est un itérateur.
- Un itérateur permet d'accéder à un élément du conteneur sans connaître sa nature.
- Un itérateur peut avoir différents comportements :
  - `input_iterator` : lecture seule (opérations `*,==,=-,++`)
  - `output_iterator` : écriture seule (opérations `*,++`)
  - `forward_iterator` : lecture et écriture avec un déplacement vers l'avant.
  - `bidirectional_iterator` : lecture et écriture avec un déplacement vers l'avant ou vers l'arrière.
  - `random_iterator` : lecture et écriture avec un déplacement vers l'avant ou vers l'arrière. Permet la comparaison arithmétique.

Les algorithmes génériques

- Fournissent un ensemble de fonctions qui permettent de modifier, interpréter ou analyser les données d'un conteneur.
- Certains algorithmes ne fonctionnent que sur une famille précise de conteneurs.
- Les algorithmes sont classées en 4 catégories :
  - Algorithmes *non mutants*

    ```
    std::for_each(v.begin(), v.end());
    ```
  - Algorithmes *mutants*

    ```
    std::fill(v.begin(), v.end(),0.);
    ```
  - Algorithmes de *tris*

    ```
    std::sort(v.begin(), v.end());
    ```
  - Algorithmes *numériques*

    ```
    sum = std::accumulate(v.begin(), v.end(),0);
    ```

Fonceurs

- C'est un objet qui peut-être appelé comme une fonction.
- Un foncteur peut être générateur `f()`, unaire `g(x)` ou binaire `h(x,y)`.
- On peut également classifier les foncteurs en fonction du type de retour.
- Opérations arithmétiques

```
vector<double> V1(100), V2(100), V3(100);
iota(V1.begin(),V1.end(),1);
fill(V2.begin(),V2.end(),10);
transform(V1.begin(),V1.end(),V2.begin(),
 V3.begin(), modulus<int>());
```

- Comparaisons

```
vector<double>::iterator first_nonzero =
find_if(V3.begin(), V3.end(),
 bind2nd(not_equal_to<double>(), 15));
```

Foncteurs (2)

- Opérations logiques

```
vector<bool> V;
transform(V.begin(), V.end(), V.begin(),
  logical_not<bool>());
```

- Opérations d'identité

```
map<int, double> M;
M[1] = 0.3; M[47] = 0.8; M[33] = 0.1;
transform(M.begin(), M.end(),
  ostream_iterator<double>(cout," "),
    select2nd<map<int, double>::value_type>());
```

- Les utilisateurs peuvent créer leurs propres foncteurs en définissant une classe.

Plan

Patron par défaut

- Un patron c'est passé le type comme paramètre d'une classe.
- Pour la classe point précédente, nous souhaitons travailler avec des **double** en 2D.
- Comme le type est un paramètre, nous pouvons lui donner une valeur par défaut

```
template<typename T=double, std::size_t=2>
class Point
{
// code
}

int main()
{
Point P;
Point<double,2> Q;
}
```

- Les points P et Q sont rigoureusement identiques.

Manipulation du type

- On peut récupérer le type utilisé dans le patron : **typedef**

```
template <typename T, std::size_t N>
class Point
{ public : typedef T data_t; };
int main(){ typedef Point<int,2>::data_t data_t; }
```

- Soit 2 types

```
struct A {typedef int sign;};
struct B {int sign;};
```

- La propriété `sign` est utilisé par la classe `Point`

```
template <typename T, std::size_t N>
class Point
{ public : T::sign signe; }; //Erreur
```

- En pratique, il faut signifier au compilateur que `sign` est un type

```
template <typename T, std::size_t N>
class Point
{ public : typename T::sign signe; };
```

Propriétés non intrusives : propriétés

- Notion de traits : donne des informations spécifiques sur la classe. Par exemple savoir si un point est déclaré en double précision ou non

```
template <typpename T> struct DoublePrecision
{
static const bool value = true;
}

template <> struct DoublePrecision<float>
{ static const bool value = false; }
template <> struct DoublePrecision<int>
{ static const bool value = false; }
```

- Utilisation

```
if (DoublePrecision<Point::data_t>::value) code
```

Propriétés non intrusives : politique

- On souhaite différencier l'affichage des coordonnées d'un point dans une fonction externe à la classe

```
template <typename T>
void display(const Point<T,N> &P)
{
  for(unsigned i=0;i< P.size();++i)
    std::cout<< val[i] <<"␣";
  cout<<std::endl;
}
```

- Si `T=double`, on affichera des nombres réels. Si `T=double*`, on affichera des pointeurs.
- On définit une classe qui fera la différenciation.

```
template <typename T> struct Read {
  static const T & getValue(const T &v){return v;}
  static const T * getPointer(const T &v){return &v;} }
template <typename T> struct Read<T* > {
  static const T & getValue(const T* v){return *v;}
  static const T * getPointer(const T* v){return v;} }
```

Propriétés non intrusives : politique (2)

- Utilisation

```
template <typename T>
void display(const Point<T,N> &P)
{
 for(unsigned i=0;i< P.size();++i)
  std::cout<< Read<T>::getValue(val[i]) <<"␣";
 cout<<std::endl;
}
```

CRTP

- Curiousy Recurring Template Pattern ou polymorphisme dynamique optimisé.
- Forme générale

```
template <class T>
struct Base
{ void interface()
  { ...
    static_cast<T*>(this)->implementation();
    ... }
  static void static_func()
  { ...
      T::static_sub_func();
      ... } };

struct Derived : Base<Derived>
{ void implementation();
  static void static_sub_func(); }
```

- Permet d'éviter l'instantiation des objets directement et la reporte à leur utilisation.

Exemple partiel

- Matrices

```
template<class T_leaftype>
class Matrix {
public:
  T_leaftype& asLeaf()
  { return static_cast<T_leaftype&>(*this); }
  double operator( int i, int j)
  { return asLeaf( i,j); } };
class SymmetricMatrix : public Matrix<SymmetricMatrix> {
};
class UpperTriMatrix : public Matrix<UpperTriMatrix> {
$};

// Function s'appliquant a n'importe quelle matrice.
template<class T_leaftype> double sum(Matrix<T_leaftype>& A)
// Utilisation
SymmetricMatrix A;
sum(A);
```

Utilisation du CRTP

- Pour les opérations complexes de type $y = A * x + b$.
- Avec une surcharge d'opérateur classique, on réalise la multiplication $A * x$, puis la somme $A * x + b$, et enfin on recopie le résultat avant de le mettre dans y.

  $\Rightarrow$ de nombreux objets intermédiaires sont manipulés.
- Avec CRTP : définition de la fonction similaire à BLAS `axpb`avec une redirection par pointeur de fonction sur les fonctions équivalentes.

  $\Rightarrow$ pas de recopie de fonctions.
- Peu de bibliothèques utilisent le CRTP : Flens, Eigen, Boost

Plan

Conclusion

- Les **template** sont une construction puissante, mais subtile dans son utilisation.
- Il ne faut surtout pas en abuser.
- La nouvelle norme du C++ apporte quelques évolution concernant les **template**
  - Définition des alias

```
template <typename First, typename Second, int third>
class SomeType;

// Syntaxe illegale en C++03
template <typename Second>
typedef SomeType<OtherType, Second, 5> TypedefName;

// Syntaxe utilsee en C++11
template <typename Second>
using TypedefName = SomeType<OtherType, Second, 5>;

// Syntaxe C++03
typedef void (*Type)(double);
// Syntaxe utilsee en C++11
using OtherType = void (*)(double);
```

Conclusion (2)

- **template** extérieur

```
// Syntaxe C++03
template class std\dotsvector<MyClass>;
// Syntaxe utilsee en C++11
extern template class std\dotsvector<MyClass>;
```

- Le mot clé **extern** permet de dire au compilateur de ne pas instantier la classe.
- **template** variadique