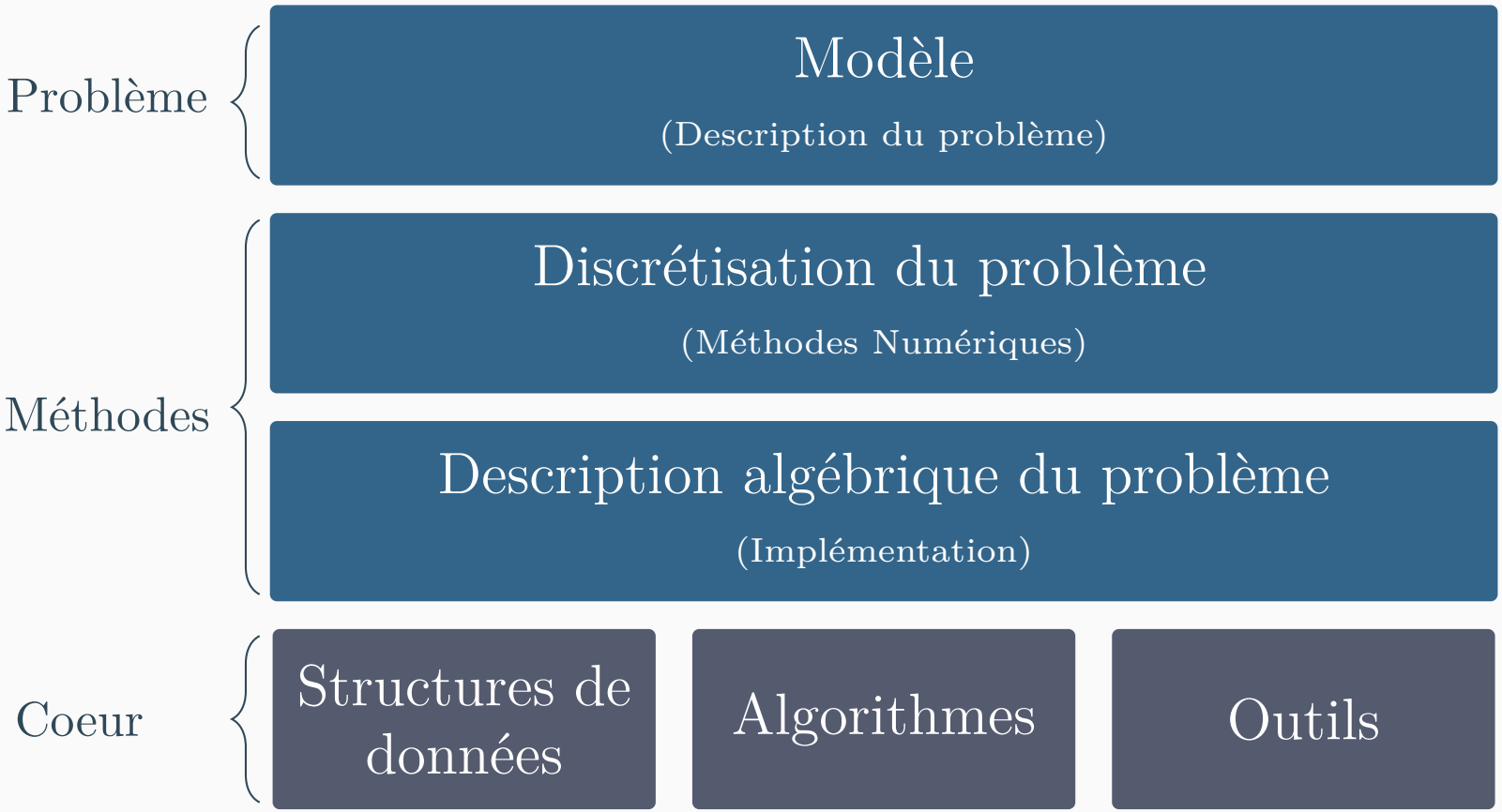
Modélisation et programmation

Cours 4

ENSIMAG 2A – MMIS

Rappels

Rappels


Problème
Modèle
(Description du problème)
Méthodes
Discrétisation du problème
(Méthodes Numériques)
Description algébrique du problème
(Implémentation)
Coeur
Structures de données
Algorithmes
Outils

Structuration des fichiers avec CMake

Organisation du projet

```
TP_LOGIN1_LOGIN2/
├── CMakeLists.txt
├── README.md
├── doc/
│   ├── CMakeLists.txt
│   ├── analyse.txt
│   └── doxyfilerc
├── src/
│   ├── CMakeLists.txt
│   └── *.cxx *.hxx
├── test/
│   ├── CMakeLists.txt
│   └── *.cxx *.hxx
├── cmake
└── build/
```

CMakeLists secondaire

```
# Créer un nouveau projet
project (libocean)

# Créer une variable contenant l'ensemble des sources cxx à compiler
set(SOURCES
    src/file1.cxx src/file2.cxx
)

# Ajoute l'ensemble des sources dans le projet
add_library(${PROJECT_NAME} ${SOURCES})
```

CMakeLists principal

```
cmake_minimum_required (VERSION 2.8.11)

project(Demo)

set(CMAKE_CXX_FLAGS "-g -Wall")

add_subdirectory(src)

add_executable(exe main.cpp)

target_link_libraries(exe
    PUBLIC
    libocean
)
target_include_directories(exe PUBLIC src)
```

CMakeLists tests

```
# Télécharge googletest au moment de la configureation
configure_file(CMakeLists.txt.in
               googletest-download/CMakeLists.txt)
execute_process(COMMAND ${CMAKE_COMMAND} -G "${CMAKE_GENERATOR}" .
  WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/googletest-download )
execute_process(COMMAND ${CMAKE_COMMAND} --build .
  WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/googletest-download )

# Ajout googletest directement au moment de la compilation
add_subdirectory(${CMAKE_BINARY_DIR}/googletest-src
                 ${CMAKE_BINARY_DIR}/googletest-build)

add_subdirectory(test)
enable_testing ()

# Cette ligne est à ajouter dans le CMakeLists.txt de test
add_executable(test_ocean test_ocean.cxx)
target_link_libraries(test_ocean gtest libocean)

add_test (NAME ocean_test
    COMMAND test_ocean
    )
```

Test de functions

```cpp
#include "gtest/gtest.h"
#include "functions.h"

TEST(IntAddition, Negative) {
  EXPECT_EQ(-5, int_addition(-2, -3))
    << "This will be shown in case it fails";
  EXPECT_EQ(-3, int_addition(5, -8));
}

TEST(IntAddition, Positive) {
  EXPECT_EQ(4, int_addition(1, 3));
  EXPECT_EQ(9, int_addition(4, 5));
}

int main(int argc, char **argv) {
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Test de classe

```
#include "gtest/gtest.h"
#include "Dvector.h"

class DvectorTest: public ::testing::Test {
 protected:
  virtual void SetUp() {
  }

  virtual void TearDown() {
    // Code here will be called immediately after each test
    // (right before the destructor).
  }
};

TEST_F(DvectorTest,constructor){
    Dvector vec(3,1);
    EXPECT_EQ(3,vec.size());
    EXPECT_EQ(1,vec(2));
}
```

Programmation générique

Pour faire quoi?

Les patrons : réfléchir plus pour travailler moins

Mise en place

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

Alternatives

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

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

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

Patron de fonction

- Syntaxe générale d'une fonction patron

```
1   template <typename T1,typename T2,...>
2   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 !

```
1   template <typename T>
2   T fonc(int &i) {return T(i);}
3   int main(int argc, char *argv[])
4   {
5     double x=fonc(2);
6   }
```

- erreur compilateur : pas de fonction `fonc(int)`. Le compilateur
  1. cherche `fonc(int)`
  2. trouve `template<typename T> fonc(int &i)`
  3. ne sait pas par quoi remplacer T
  4. répond qu'il n'a pas trouvé `fonc(int)`

Patrons de fonctions: passage de type

- Dans le cas précédent, on demande au compilateur de déterminer quel est le type de retour de la fonction.
- Comme le type ne fait pas parti des paramètres d'entrée, **`int`** et **`double`** sont des types différents, le compilateur ne sait pas faire.
- Pour palier ce problème, on passe le type en argument de la fonction au moment de l'appel à l'aide des chevrons < et >

```
1   int main(int argc, char *argv[])
2   { double x=fonc<int>(2); }
```

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

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

- Syntaxe équivalente

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

- On peut remplacer **`typename`** par **`class`**

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

Patrons : fonction minimum

- Un patron peut-être surchargé

```
1   template<typename T>
2   T min(const T& a, const T& b){return (( a<b ) ? (a): (b));}
3   template<typename T>
4   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

```
1   template<typename T1, typename T2>
2   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

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

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

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

Comprendre avant d'utiliser!

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

```
1   template<typename T>
2   T min(const T& a, const T& b){return (( a<b ) ? (a): (b));}
3   int main(int argc,char *argv[])
4   {
5     int    i = 2;
6     double b = 1.9;
7     min(i,b);
8     return 0;
9   }
```

- Pour les chaînes de caractères **char ***
  - Si T = **char***, la fonction **min** fera la comparaison de deux entiers!!!
  - On doit donc redéfinir la fonction **min** pour **char***

```
1   const char *min(const char *l, const char *g)
2   {
3     if (strcmp(l, g) > 0)
4       return g;
5     else
6       return l;
7   }
```

- En cas de conflit, la version dédiée est toujours préférée.

Classe patron

- Généralisation du concept de patron de fonctions aux classes

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

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

```
template <typename T> class vect
{
  public :
    T* val;
    int dim;
    vect<T>(const int d=0)
    {
      dim d;
      if(d>0) { val = new T[d]; }
    }
    T& operator()(const int i)
    {
      if(i>0 && i<= dim) { return val[i-1]; //Erreur }
    }
};
```

- 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`

```
1  int main()
2  {
3    vect<double> V(2);
4    V(1) = 1.;
5    V(2) = 2.;
6    vect<char *> L(2);
7    L(1) = "chaine 1";
8    L(2) = "chaine 2";
9  }
```

- 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 `vect`
  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`

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

- La classe `vect` ne pourra plus gérer des vecteurs de `char*` à cause de l'opérateur `+=`.
- 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`

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

```
1   vect<double> V(2);
2   bool cas_Complexe = false;
3   if(cas_Complexe)
4   {
5     V*=complex<double>(0,1);
6   }
7   else
8   {
9     V*=2;
10  }
```

- 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

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

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

  - pour une classe

```
1   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 : spécialisation

```
1   template<typename T> class vect
2   {
3     public :
4       ...
5         template <typename S> vect<T>& operator*=(const S& s)
6         {
7           for(int i=0;i<dim;i++) { val[i]*=s; }
8           return *this;
9         }
10  };
11
12  vect<double>& operator*=(const complex<double>& s)
13  {
14    //traitement specifique
15  }
```

- Redéfinition d'une instance particulière de l'opérateur `*=`
- Permet de résoudre le problème de compilation évoqué : il existe une version `double*=complexe` admissible.
- Spécialisation totale dans ce cas, mais la spécialisation partielle existe aussi.

Spécialisation : détails

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

```
1  template <typename T,std::size_T N>
2  classPoint
3  {
4    public:
5      Point();
6    private:
7      T val[N];
8  };
```

- 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` :

```
1  template <>
2  class Point<double,2>
3  {
4    //code specifique
5  }
```

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

```
1   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

```
1   template<typename T> class Point<T,2>
2   {
3     //code specifique
4   }
```

- 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.

```
1   #ifndef VECT_H
2   #define VECT_H
3
4   template <typename T>
5   class vect
6   {
7     public :
8       T* val;
9       int dim;
10
11      vect<T>(const int d=0);
12
13      T& operator()(const int i);
14
15      vect<T>& operator+=(const S& s);
16  };
17  #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

```
1   #include "vect.h"
2
3     template <typename T>
4   T& vect<T>::operator()(const int i)
5   { if(i>0 && i>=dim) { return val[i-1]; } }
6
7     template <typename T>
8   vect<T>& vect<T>::operator+=(const T& V)
9   {
10    for(int i=0;i<dim;i++)
11    { val[i]+=V.val[i]; return *this; }
12  }
13    template <typename T> template <typename S>
14  vect<T>& vect<T>::operator*=(const S& s)
15  {
16    for(int i=0;i<dim;i++)
17    { val[i]*=V.val[i]; return *this; }
18  }
```

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

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

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

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

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

Patron par défaut

- Rappel : Un patron c'est passé le type comme paramètre d'une classe.
- Reprenons notre classe **`Point`** dans sa version patron.
- Dans la plupart des cas, nous travaillerons avec des **`double`** en 2D.
- Comme le type est un paramètre, nous pouvons lui donner une valeur par défaut

```
1   template<typename T=double, std::size_t=2>
2   class Point
3   {
4     // code
5   }
6
7   int main()
8   {
9     Point P;
10    Point<double,2> Q;
11  }
```

- Les points P et Q sont rigoureusement identiques.

Manipulation du type

- On peut récupérer le type utilisé dans le patron. Pour cela on utilise le mot clé **typedef**

```
1  template <typename T, std::size_t N>
2  class Point
3  { public : typedef T data_t; };
4  int main(){ typedef Point<int,2>::data_t data_t; // entier
5  }
```

- Soit 2 types

```
1  struct A
2  {typedef int sign;};
3  struct B
4  {int sign;};
```

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

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

- En pratique, il faut définir, pour le compilateur que **sign** est un type

```
1  template <typename T, std::size_t N>
2  class Point
3  { 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

```
1  template <typpename T> struct DoublePrecision
2  {
3    static const bool value = true;
4  }
5  template <> struct DoublePrecision<float>
6  { static const bool value = false; }
7  template <> struct DoublePrecision<int>
8  { static const bool value = false; }
```

- Utilisation

```
1  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

```
1   template <typename T>
2   void display(const Point<T,N> &P) {
3     for(unsigned i=0;i< P.size();++i)
4       std::cout<< val[i] <<" ";
5     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.

```
1   template <typename T> struct Read {
2     static const T & getValue(const T & v){return v;}
3     static const T * getPointer(const T & v){return &v;} }
4   template <typename T> struct Read<T *> {
5     static const T & getValue(const T * v){return *v;}
6     static const T * getPointer(const T * v){return v;} }
```

- Utilisation

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

Conclusion

Conclusion

- Permet du code générique.
- Il n'y a pas de perte de performance.
- Force à l'abstraction.

Standard Template Library

CRTP

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

```
1   template <class T>
2     struct Base
3   { void interface()
4     { ...
5       static_cast<T*>(this)->implementation();
6       ...  }
7     static void static_func()
8     { ...
9       T::static_sub_func();
10      ...  } };
11
12  struct Derived : Base<Derived>
13  { void implementation();
14    static void static_sub_func(); }
```

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

Exemple partiel

- Matrices

```
1   template<class T_leaftype>
2   class Matrix {
3     public:
4       T_leaftype& asLeaf()
5       { return static_cast<T_leaftype&>(*this); }
6       double operator( int i, int j)
7       { return asLeaf( i,j); } };
8   class SymmetricMatrix : public Matrix<SymmetricMatrix> {
9   };
10  class UpperTriMatrix : public Matrix<UpperTriMatrix> {
11    };
12
13  // Function s'appliquant a n'importe quelle matrice.
14  template<class T_leaftype> double sum(Matrix<T_leaftype>& A)
15
16  // Utilisation
17  SymmetricMatrix A;
18  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.

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

```
1   template <typename First, typename Second, int third>
2   class SomeType;
3
4   // Syntaxe illegale en C++03
5   template <typename Second>
6   typedef SomeType<OtherType, Second, 5> TypedefName;
7
8   // Syntaxe utilsee en C++11
9   template <typename Second>
10  using TypedefName = SomeType<OtherType, Second, 5>;
11
12  // Syntaxe C++03
13  typedef void (*Type)(double);
14  // Syntaxe utilsee en C++11
15  using OtherType = void (*)(double);
```

Conclusion (2)

- **`template`** extérieur

```
1  // Syntaxe C++03
2  template class std::vector<MyClass>;
3  // Syntaxe utilsee en C++11
4  extern template class std::vector<MyClass>;
```

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

- **`template`** variadique