Cours 7 - C++ pour les mathématiques appliquées

Programmation générique

Pour faire quoi?

Les patrons : réfléchir plus pour travailler moins

Mise en place

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

Patrons de fonction

Alternatives

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

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

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

Patron de fonction

- Syntaxe générale d'une fonction patron

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

- 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 !

Patron de fonction : exemple

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

- erreur compilateur : pas de fonction . Le compilateur
  i. cherche `fonc(int)`
  ii. trouve `template<typename T> fonc(int &i)`
  iii. ne sait pas par quoi remplacer `T`
  iv. 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 `>`

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

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

```
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 `<` et le constructeur par copie `T(const T&)` , sauf pour les types de bases.

- 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 `T1` type 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
```

Comprendre avant d'utiliser! (1)

Le compilateur n'effectue pas la conversion automatique des variables car il y a ambiguïté (erreur du type conflicting type for parameter T).

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

Comprendre avant d'utiliser! (2)

- 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 *`

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

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

Patron de classe

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
  }
};
```

Classe patron (2)

- 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`
- A la compilation, le compilateur
    i. recherche le type `vect<double>`
    ii. trouve `template<typename T> class vect`
    iii. génère le code explicite en remplaçant `T` par `vect`
    iv. compile le code généré
    v. 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;
  }
};
```

Classe patron : champ opératoire (2)

- 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`

```
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;
      }
};
```

- 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

```
double*=complex<double>
```

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 (1)

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

```
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;
  }
};

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

Spécialisation (2)

- 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 (1)

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

```
template <typename T, std::size_t N>
class Point {
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.

Spécialisation : détails (2)

Afin de travailler sur des points de double en dimension 2, on utiliserait une spécialisation de la classe `Point` :

```
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);
};
#include "vect.txx"
#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`

```
template <typename T>
T &vect<T>::operator()(const int i) {
  if (i > 0 && i >= dim)
    return val[i - 1]; }
template <typename T>
vect<T> &vect<T>::operator+=(const T &V) {
  for (int i = 0; i < dim; i++)
    val[i] += V.val[i];
  return *this; }
template <typename T>
template <typename S>
vect<T> &vect<T>::operator*=(const S &s) {
  for (int i = 0; i < dim; i++)
    val[i] *= V.val[i];
  return *this; }
```

La compilation

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 <>
class Factorial<0> {
public:
  static const long valeur = 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!

Et après . . .

Patron par défaut

- Rappel : **Un patron c'est passer le type comme paramètre d'une classe**
- Reprenons notre classe `Point` dans sa version patron.
- Dans la plupart des cas, nous travaillerons avec des `doubme` 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 types des points P et Q sont rigoureusement identiques.

Manipulation du type (1)

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

  `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; // entier
};
```

- Soit 2 types

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

Manipulation du type(2)

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

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

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

```
template <typename T, std::size_t N>
class Point {
public:
  typename T::sign signe;
}; //Ok pour A, erreur pour B'
```

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

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

Conclusion

A retenir

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

CRTP

- Curiously 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 : matrice

```
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 : Eigen, Boost.

Conclusion (1)

- 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`

Conclusion (2)

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 (3)

- `template` extérieur

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

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