Langage C++ et programmation orientée objet

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence RESIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus - bureau 241
E-mail : vincent.vidal@univ-lyon1.fr
Supports de cours et TPs : `https://claroline connect.univ-lyon1.fr/` espace d'activité "M41L02C - Langage C++"
26H prévues ≈ 24H de cours+TDs/TPs, et 2H - examen final
Évaluation : Contrôle continu (TPs) + examen final

Plan

Plan

Introduction

Nous souhaitons avoir une fonction pour calculer le minimum entre 2 éléments de type **double** et une autre telle fonction pour calculer le minimum entre 2 éléments de type **int**.

Cela est possible grace à la surdéfinition de fonction.

Introduction

Nous souhaitons avoir une fonction pour calculer le minimum entre 2 éléments de type **double** et une autre telle fonction pour calculer le minimum entre 2 éléments de type **int**.

Cela est possible grace à la surdéfinition de fonction.

Introduction

```
double minimum(double d1, double d2) { return ((d1<d2)?d1:d2) ; }

int minimum(int i1, int i2) { return ((i1<i2)?i1:i2) ; }
```

Le même calcul est réalisé dans les 2 fonctions minimum.

Seuls les types des arguments et le type de retour sont différents.

Serait-il possible de passer en argument le type, pour n'écrire qu'une seule fonction et spécifier le type à l'appel de cette fonction ?

Oui, avec la notion de **patron de fonctions**.

Introduction

```
double minimum(double d1, double d2) { return ((d1<d2)?d1:d2) ; }

int minimum(int i1, int i2) { return ((i1<i2)?i1:i2) ; }
```

Le même calcul est réalisé dans les 2 fonctions minimum.
Seuls les types des arguments et le type de retour sont différents.

Serait-il possible de passer en argument le type, pour n'écrire qu'une seule fonction et spécifier le type à l'appel de cette fonction ?

Oui, avec la notion de **patron de fonctions**.

Introduction

```
double minimum(double d1, double d2) { return ((d1<d2)?d1:d2) ; }

int minimum(int i1, int i2) { return ((i1<i2)?i1:i2) ; }
```

Le même calcul est réalisé dans les 2 fonctions minimum.
Seuls les types des arguments et le type de retour sont différents.

Serait-il possible de passer en argument le type, pour n'écrire qu'une seule fonction et spécifier le type à l'appel de cette fonction ?

Oui, avec la notion de **patron de fonctions**.

Introduction

```
double minimum(double d1, double d2) { return ((d1<d2)?d1:d2) ; }

int minimum(int i1, int i2) { return ((i1<i2)?i1:i2) ; }
```

Le même calcul est réalisé dans les 2 fonctions minimum.
Seuls les types des arguments et le type de retour sont différents.

Serait-il possible de passer en argument le type, pour n'écrire qu'une seule fonction et spécifier le type à l'appel de cette fonction ?
Oui, avec la notion de **patron de fonctions**.

Création d'un patron de fonctions minimum

```
// Patron ou modèle des fonctions minimum
template< typename T >
T minimum(T val1, T val2) { return ((val1<val2)?val1:val2) ; }
```

Remarque : on peut utiliser le mot-clé **class** à la place de **typename**, mais cela signifiera la même chose, à savoir que le paramètre formel qui suit est un type.

Création d'un patron de fonctions minimum

```
// Patron ou modèle des fonctions minimum
template< typename T >
T minimum(T val1, T val2) { return ((val1<val2)?val1:val2) ; }
```

Remarque : on peut utiliser le mot-clé **class** à la place de **typename**, mais cela signifiera la même chose, à savoir que le paramètre formel qui suit est un type.

Fonction patron : instance d'un patron de fonctions

```
#include <iostream>
using namespace std ;
// Patron ou modèle des fonctions minimum
template< typename T >
T minimum(T val1, T val2) { return ((val1<val2)?val1:val2) ; }

int main()
{
    int a = 5, b = -1 ;
    float f = 2.18f, g = -4.75f ;
    // Ici c'est le compilateur qui va choisir le type T en fonction du type des arguments
    // et l'utiliser comme type effectif
                                          // int minimum(int, int)
    cout << " min(" << a << ", " << b << ") = " << minimum(a, b) << endl ; // OK
                                          // float minimum(float, float)
    cout << " min(" << f << ", " << g << ") = " << minimum(f, g) << endl ; // OK
    cout << " min(" << 2. << ", " << 8. << ") = " << minimum(2., 8.) << endl ; // OK (double)
                                          // ? minimum(float, int)
    cout << " min(" << f << ", " << b << ") = " << minimum(f, b) << endl ; // ERREUR
    cout << " min(" << 2 << ", " << 8. << ") = " << minimum(2, 8.) << endl ; // ERREUR
    cout << " min(" << 2 << ", " << 8. << ") = " << minimum<double>(2, 8.) << endl ; // OK, force
                                                                                    // type à
                                                                                    // double
     return EXIT_SUCCESS ;
}
```

Fonction patron : instance d'un patron de fonctions

*Puis-je utiliser un autre type qu'un type de base (***char***,* **short***,* **int***,* **long***,* **float** *et* **double***) pour notre patron de fonctions* minimum *?*

oui si l'opérateur < est défini pour le type.

Je vais donc pouvoir utiliser les types suivants :

- des types scalaires/simples ((**char**, **short**, etc.) ;
- des classes pour lesquelles l'opérateur < est bien défini ;
- des références : **char&**, **short&**, **int&**, **long&**, ... ;
- et des adresses : **char***, **short***, **int***, **long***, ...

Fonction patron : instance d'un patron de fonctions

*Puis-je utiliser un autre type qu'un type de base (***char***,* **short***,* **int***,* **long***,* **float** *et* **double***) pour notre patron de fonctions* minimum *?*
oui si l'opérateur **<** est défini pour le type.

Je vais donc pouvoir utiliser les types suivants :

- des types scalaires/simples ((**char**, **short**, etc.) ;
- des classes pour lesquelles l'opérateur < est bien défini ;
- des références : **char&**, **short&**, **int&**, **long&**, ... ;
- et des adresses : **char***, **short***, **int***, **long***, ...

Fonction patron : instance d'un patron de fonctions

*Puis-je utiliser un autre type qu'un type de base (***char***,* **short***,* **int***,* **long***,* **float** *et* **double***) pour notre patron de fonctions* minimum *?*
oui si l'opérateur **<** est défini pour le type.

Je vais donc pouvoir utiliser les types suivants :

- des types scalaires/simples ((**char**, **short**, etc.) ;
- des classes pour lesquelles l'opérateur < est bien défini ;
- des références : **char&**, **short&**, **int&**, **long&**, ... ;
- et des adresses : **char***, **short***, **int***, **long***, ...

Fonction patron : instance d'un patron de fonctions

*Puis-je utiliser un autre type qu'un type de base (***char***,* **short***,* **int***,* **long***,* **float** *et* **double***) pour notre patron de fonctions* minimum *?*
oui si l'opérateur **<** est défini pour le type.

Je vais donc pouvoir utiliser les types suivants :

- des types scalaires/simples ((**char**, **short**, etc.) ;
- des classes pour lesquelles l'opérateur **<** est bien défini ;
- des références : **char&**, **short&**, **int&**, **long&**, ... ;
- et des adresses : **char***, **short***, **int***, **long***, ...

Fonction patron : instance d'un patron de fonctions

*Puis-je utiliser un autre type qu'un type de base (***char***,* **short***,* **int***,* **long***,* **float** *et* **double***) pour notre patron de fonctions* minimum *?*
oui si l'opérateur **<** est défini pour le type.

Je vais donc pouvoir utiliser les types suivants :

- des types scalaires/simples ((**char**, **short**, etc.) ;
- des classes pour lesquelles l'opérateur **<** est bien défini ;
- des références : **char&**, **short&**, **int&**, **long&**, ... ;
- et des adresses : **char***, **short***, **int***, **long***, ...

Fonction patron : instance d'un patron de fonctions

*Puis-je utiliser un autre type qu'un type de base (***char***,* **short***,* **int***,* **long***,* **float** *et* **double***) pour notre patron de fonctions* minimum *?*
oui si l'opérateur **<** est défini pour le type.

Je vais donc pouvoir utiliser les types suivants :

- des types scalaires/simples ((**char**, **short**, etc.) ;
- des classes pour lesquelles l'opérateur **<** est bien défini ;
- des références : **char&**, **short&**, **int&**, **long&**, ... ;
- et des adresses : **char***, **short***, **int***, **long***, ...

Mécanisme d'instanciation

Le compilateur va "fabriquer" toutes les instances du patron de fonctions nécessaires.

Ainsi, **les patrons de fonctions NE DOIVENT PAS ETRE DEFINIS dans un .cpp d'un composant logiciel.**

Cela signifie simplement que le patron doit être entièrement connu (déclaration+définition) par le compilateur pendant la compilation (avant l'édition de lien).

Mécanisme d'instanciation

Le compilateur va "fabriquer" toutes les instances du patron de fonctions nécessaires.

Ainsi, **les patrons de fonctions NE DOIVENT PAS ETRE DEFINIS dans un .cpp d'un composant logiciel.**

Cela signifie simplement que le patron doit être entièrement connu (déclaration+définition) par le compilateur pendant la compilation (avant l'édition de lien).

Mécanisme d'instanciation

Le compilateur va "fabriquer" toutes les instances du patron de fonctions nécessaires.

Ainsi, **les patrons de fonctions NE DOIVENT PAS ETRE DEFINIS dans un .cpp d'un composant logiciel.**

Cela signifie simplement que le patron doit être entièrement connu (déclaration+définition) par le compilateur pendant la compilation (avant l'édition de lien).

Comment séparer les déclarations des définitions ?

```
#include <iostream>
using namespace std ;

// Déclaration du patron dans un fichier .hxx (pour dire que c'est un template)
template< typename T >
T minimum(T val1, T val2) ;

// Définition du patron dans un fichier .txx :
// ce fichier doit être inclut dans le .hxx précédent
// à la suite des déclarations
template< typename T >
T minimum(T val1, T val2) { return ((val1<val2)?val1:val2) ; }
```

Utilisation des types du patron dans la définition

Jusquà présent, les types d'un patron de fonctions n'étaient utilisés que dans l'*en-tête des fonctions*.
Ils peuvent aussi être utilisés dans *le corps des fonctions* pour déclarer des variables locales ou pour certains opérateurs (**new**, **sizeof**...).

On peut passer **plusieurs types en paramètres d'un patron** de fonctions. **Chaque type DOIT apparaître au moins 1 fois dans l'en-tête** de la fonction (au niveau des arguments) pour que le compilateur puisse instancier une fonction particulière de façon automatique.

```
template< typename T, typename U, typename V >
T fct(T val1, T val2, U val3, U* val4, V& val5){ U tmp;... T* ptr = new T[10];... delete [] ptr;}
```

Utilisation des types du patron dans la définition

Jusquà présent, les types d'un patron de fonctions n'étaient utilisés que dans l'*en-tête des fonctions*.
Ils peuvent aussi être utilisés dans *le corps des fonctions* pour déclarer des variables locales ou pour certains opérateurs (**new**, **sizeof**...).

On peut passer **plusieurs types en paramètres d'un patron** de fonctions. **Chaque type DOIT apparaître au moins 1 fois dans l'en-tête** de la fonction (au niveau des arguments) pour que le compilateur puisse instancier une fonction particulière de façon automatique.

```
template< typename T, typename U, typename V >
T fct(T val1, T val2, U val3, U* val4, V& val5){ U tmp;... T* ptr = new T[10];... delete [] ptr;}
```

Les contraintes sur l'instanciation

```
// Forcer un argument de type pointeur
template< typename T >
void fct(T* ptr) { ... }

// Forcer un argument de type référence (cela forcera
// le compilateur à vérifier que l'argument est bien une
// lvalue!)
template< typename T >
void fct(T& ref) { ... }

// On peut surcharger les patrons de fonctions : les
// 2 patrons précédents peuvent être définis au sein
// d'un même fichier (2 familles de fonctions) !
```

Les contraintes sur l'instanciation

```
// Surcharge de patron (suite) :
template< typename T >
T minimum(T val1, T val2) {
    return ((val1<val2)?val1:val2) ;
}

template< typename T >
T minimum(T val1, T val2, T val3) {
    return minimum(minimum(val1, val2), val3) ;
}

// On peut avoir des arguments de type "usuel"
template< typename T >
void fct(T* ptr, int cpt, string& s, T val) {...}
```

Ambiguïtés

```
// Ambiguïtés de la surchage :
template< typename T >
void fct(T ptr) { ... }

et

template< typename T >
void fct(T& ptr) { ... }
```

Exemples

```
#include <cstring>
// Spécialisation : changer l'algo pour un type particulier :
template< typename T >
T minimum(T val1, T val2) { return ((val1<val2)?val1:val2) ; }

char* minimum(char* val1, char* val2) { return ((strcmp(val1, val2)<0)?val1:val2) ; }

// Spécialisation partielle : le patron le plus spécialisé sera privilégié
// si on rencontre une adresse.
template< typename T >
void fct(T ptr) { ... }

template< typename T >
void fct(T* ptr) { fct(*ptr); }
```

Plan

Introduction

```
class CPoint2D
{
    float m_x, m_y;
  public :
    CPoint2D(float x=0, float y=0) : m_x(x), m_y(y) {}
    CPoint2D(const CPoint2D& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() const { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }
};
```

Ici, nous imposons que les coordonnées des points 2D soient des **float**. Si nous souhaitons disposer à la fois de points 2D de **float** et de points 2D de **double**...

Il est possible de faire un patron de cette classe en remplaçant les apparitions de **float** par le type voulu.

Introduction

```
class CPoint2D
{
    float m_x, m_y;
  public :
    CPoint2D(float x=0, float y=0) : m_x(x), m_y(y) {}
    CPoint2D(const CPoint2D& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() const { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }
};
```

Ici, nous imposons que les coordonnées des points 2D soient des **float**. Si nous souhaitons disposer à la fois de points 2D de **float** et de points 2D de **double**...

Il est possible de faire un patron de cette classe en remplaçant les apparitions de **float** par le type voulu.

Introduction

```
class CPoint2D
{
    float m_x, m_y;
  public :
    CPoint2D(float x=0, float y=0) : m_x(x), m_y(y) {}
    CPoint2D(const CPoint2D& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() const { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }
};
```

Ici, nous imposons que les coordonnées des points 2D soient des **float**. Si nous souhaitons disposer à la fois de points 2D de **float** et de points 2D de **double**...

Il est possible de faire un patron de cette classe en remplaçant les apparitions de **float** par le type voulu.

Introduction

```
// Déclaration d'un patron de classes
template< typename T >
class CPoint2D
{
    T m_x, m_y;
  public :
    CPoint2D(T x=0, T y=0) : m_x(x), m_y(y) {}
    CPoint2D(const CPoint2D<T>& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() const ;
};

// Définitions en dehors de la classe (autres que en-ligne dans classe) :
template< typename T >
void CPoint2D<T>::affiche() const { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }
```

Le patron de classes doit être entièrement connu à la compilation (mêmes remarques que pour les patrons de fonctions slide 9).

Introduction

```
// Déclaration d'un patron de classes
template< typename T >
class CPoint2D
{
    T m_x, m_y;
  public :
    CPoint2D(T x=0, T y=0) : m_x(x), m_y(y) {}
    CPoint2D(const CPoint2D<T>& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() const ;
};

// Définitions en dehors de la classe (autres que en-ligne dans classe) :
template< typename T >
void CPoint2D<T>::affiche() const { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }
```

Le patron de classes doit être entièrement connu à la compilation (mêmes remarques que pour les patrons de fonctions slide 9).

Introduction

```
// Déclaration d'un patron de classes
template< typename T >
class CPoint2D
{
    T m_x, m_y;
  public :
    CPoint2D(T x=0, T y=0) : m_x(x), m_y(y) {}
    CPoint2D(const CPoint2D<T>& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() const ;
};

// Définitions en dehors de la classe (autres que en-ligne dans classe) :
template< typename T >
void CPoint2D<T>::affiche() const { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }

int main()
{   // Le type utilisé apparaît à la déclaration des objets
    CPoint2D<double> p, p1=5, p2(6.2,-7.8); // Instanciation d'une classe où double remplace le T
    CPoint2D<int> p3, p4=5, p5(6,-7);       // Instanciation d'une classe où int remplace le T
    p2.affiche();
    p4.affiche();
    return EXIT_SUCCESS;
}
```

Et un CPoint2D de type quelconque ?

On pourra utiliser le patron de classes précédent avec une classe comme paramètre si :

- La conversion de int (le 0) vers le type T existe ;
- La reconstruction par copie du type T est bien gérée (emplacements dynamiques) ;
- L'affectation du type T est bien gérée (emplacements dynamiques) s'il y a une affectation pour un objet de type T.

Et un CPoint2D de type quelconque ?

On pourra utiliser le patron de classes précédent avec une classe comme paramètre si :

- La conversion de int (le 0) vers le type T existe ;
- La reconstruction par copie du type T est bien gérée (emplacements dynamiques) ;
- L'affectation du type T est bien gérée (emplacements dynamiques) s'il y a une affectation pour un objet de type T.

Et un CPoint2D de type quelconque ?

On pourra utiliser le patron de classes précédent avec une classe comme paramètre si :

- La conversion de int (le 0) vers le type T existe ;
- La reconstruction par copie du type T est bien gérée (emplacements dynamiques) ;
- L'affectation du type T est bien gérée (emplacements dynamiques) s'il y a une affectation pour un objet de type T.

Les contraintes sur les paramètres de type

On pourra :

- avoir plusieurs types en paramètres d'un patron de classes ;
- avoir des types effectifs qui soient eux-mêmes des instances d'un patron de classes, e.g. `CPoint2D< CPoint2D<float> > p;` ;
- avoir des paramètres expressions (paramètres usuels des fonctions), e.g. `int taille`.

On ne pourra pas surdéfinir un patron de classes !

Remarque : les attributs `static` (de classe) d'un patron de classes ne sont communs qu'au sein d'une instance, pas du patron.

Les contraintes sur les paramètres de type

On pourra :

- avoir plusieurs types en paramètres d'un patron de classes ;
- avoir des types effectifs qui soient eux-mêmes des instances d'un patron de classes, e.g. `CPoint2D< CPoint2D<float> > p; ;`
- avoir des paramètres expressions (paramètres usuels des fonctions), e.g. `int taille`.

On ne pourra pas surdéfinir un patron de classes !

Remarque : les attributs static (de classe) d'un patron de classes ne sont communs qu'au sein d'une instance, pas du patron.

Les contraintes sur les paramètres de type

On pourra :

- avoir plusieurs types en paramètres d'un patron de classes ;
- avoir des types effectifs qui soient eux-mêmes des instances d'un patron de classes, e.g. `CPoint2D< CPoint2D<float> > p; ;`
- avoir des paramètres expressions (paramètres usuels des fonctions), e.g. `int taille`.

On ne pourra pas surdéfinir un patron de classes !

Remarque : les attributs static (de classe) d'un patron de classes ne sont communs qu'au sein d'une instance, pas du patron.

Les contraintes sur les paramètres de type

On pourra :

- avoir plusieurs types en paramètres d'un patron de classes ;
- avoir des types effectifs qui soient eux-mêmes des instances d'un patron de classes, e.g. CPoint2D< CPoint2D<**float**> > p; ;
- avoir des paramètres expressions (paramètres usuels des fonctions), e.g. **int** taille.

On ne pourra pas surdéfinir un patron de classes !

Remarque : les attributs static (de classe) d'un patron de classes ne sont communs qu'au sein d'une instance, pas du patron.

Les contraintes sur les paramètres de type

On pourra :

- avoir plusieurs types en paramètres d'un patron de classes ;
- avoir des types effectifs qui soient eux-mêmes des instances d'un patron de classes, e.g. `CPoint2D< CPoint2D<float> > p; ;`
- avoir des paramètres expressions (paramètres usuels des fonctions), e.g. `int taille`.

On ne pourra pas surdéfinir un patron de classes !

Remarque : les attributs `static` (de classe) d'un patron de classes ne sont communs qu'au sein d'une instance, pas du patron.

Exemple de paramètre expression

```
#include <cassert>
template< typename T, int n > // passer n ici conduit à un tableau alloué automatiquement
class CTableau
{
    T tab[n] ;
  public :
    CTableau(T initVal=0) { for(int i=0; i<n; ++i) tab[i] = initVal ; }
    T& operator [](int indice) { assert(0<=indice && indice<n) ; return tab[indice] ; }
    const T& operator [](int indice) const { assert(0<=indice && indice<n); return tab[indice]; }
    void affiche() const ;
};
template< typename T, int n >
void CTableau<T, n>::affiche() const
{
    for(int i=0; i<n; ++i) cout << " " << tab[i] ; cout << endl ;
}
int main()
{
    CTableau<int, 10> tab ; // le paramètre effectif d'un paramètre expression doit être
                            // constant et du type demandé (aucune conversion possible)
    tab[4] = 10;
    tab.affiche();
    return EXIT_SUCCESS ;
}
```

Spécialisation d'une méthode (le plus fréquent)

```
template< typename T >
class CPoint2D
{
    T m_x, m_y;
  public :
    CPoint2D(T x=0, T y=0) : m_x(x), m_y(y) {}
    CPoint2D(const CPoint2D<T>& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() const ;
};
// Définitions en dehors de la classe (autres que en-ligne dans classe) :
template< typename T >
void CPoint2D<T>::affiche() const
{
    std::cout << "(" << m_x << ", " << m_y << ")" << std::endl;
}

template<>
void CPoint2D<char>::affiche() const
{
    std::cout << "(" << (int)m_x << ", " << (int)m_y << ")" << std::endl;
}
```

Spécialisation d'une classe

```
template< typename T >
class CPoint2D
{
    T m_x, m_y;
  public :
    CPoint2D(T x=0, T y=0) : m_x(x), m_y(y) {}
    CPoint2D(const CPoint2D& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() const ;
};
// Définitions en dehors de la classe (autres que en-ligne dans classe) :
template< typename T >
void CPoint2D<T>::affiche() const
{
    std::cout << "(" << m_x << ", " << m_y << ")" << std::endl;
}

class CPoint2D <char>
{
...
};
```

Paramètres par défaut

```
#include <cassert>
template< typename T=float, int n=10 > // Les paramètres par défaut sont donnés à la déclaration
class CTableau                         // du patron
{
    T tab[n] ;
  public :
    CTableau(T initVal=0) { for(int i=0; i<n; ++i) tab[i] = initVal ; }
    T& operator [](int indice) { assert(0<=indice && indice<n) ; return tab[indice] ; }
    const T& operator [](int indice) const { assert(0<=indice && indice<n); return tab[indice]; }
    void affiche() const ;
};
template< typename T, int n >
void CTableau<T, n>::affiche() const
{
    for(int i=0; i<n; ++i) cout << " " << tab[i] ; cout << endl ;
}
int main()
{
    CTableau<int, 10> tab ; // OK
    CTableau<long> tab1;    // OK
    CTableau<> tab2;        // OK
    CTableau tab3;          // ERREUR
    return EXIT_SUCCESS ;
}
```

Remarques

- Le mécanisme de définition de patrons de fonctions peut s'appliquer à une fonction membre (il faudra alors la précéder de template < ... >).
- CTableau< CTableau<int, 10>, 20 > tab2D est un tableau 2D régulier. Mais tab2D.affiche() ne compilera pas. Pourquoi ?
- 2 instances d'un patron de classes ont le même type pour le compilateur si les types et expressions constantes passées en paramètre sont exactement les mêmes.
  Le type doit être exactement le même pour que l'affectation par défaut fonctionne.

Remarques

- Le mécanisme de définition de patrons de fonctions peut s'appliquer à une fonction membre (il faudra alors la précéder de template < ... >).
- CTableau< CTableau<int, 10>, 20 > tab2D est un tableau 2D régulier. Mais tab2D.affiche() ne compilera pas. Pourquoi ?
- 2 instances d'un patron de classes ont le même type pour le compilateur si les types et expressions constantes passées en paramètre sont exactement les mêmes.
  Le type doit être exactement le même pour que l'affectation par défaut fonctionne.

Remarques

- Le mécanisme de définition de patrons de fonctions peut s'appliquer à une fonction membre (il faudra alors la précéder de template < ... >).
- CTableau< CTableau<int, 10>, 20 > tab2D est un tableau 2D régulier. Mais tab2D.affiche() ne compilera pas. Pourquoi ?
- 2 instances d'un patron de classes ont le même type pour le compilateur si les types et expressions constantes passées en paramètre sont exactement les mêmes.
  Le type doit être exactement le même pour que l'affectation par défaut fonctionne.

Remarques

- Le mécanisme de définition de patrons de fonctions peut s'appliquer à une fonction membre (il faudra alors la précéder de template < ... >).
- CTableau< CTableau<int, 10>, 20 > tab2D est un tableau 2D régulier. Mais tab2D.affiche() ne compilera pas. Pourquoi ?
- 2 instances d'un patron de classes ont le même type pour le compilateur si les types et expressions constantes passées en paramètre sont exactement les mêmes.
  Le type doit être exactement le même pour que l'affectation par défaut fonctionne.

Remarques sur l'amitié

- Classe amie d'une instance particulière du patron de classes CPoint2D : friend class CPoint2D<float>;
- Classe amie d'une instance particulière du patron de fonctions minimum : friend double minimum(double,double);
- On peut dans les déclarations précédentes utiliser un type T du patron de classes pour coupler l'amitié et l'instance de la classe.

Remarques sur l'amitié

- Classe amie d'une instance particulière du patron de classes CPoint2D : `friend class CPoint2D<float>;`
- Classe amie d'une instance particulière du patron de fonctions minimum : `friend double minimum(double,double);`
- On peut dans les déclarations précédentes utiliser un type T du patron de classes pour coupler l'amitié et l'instance de la classe.

Remarques sur l'amitié

- Classe amie d'une instance particulière du patron de classes CPoint2D : `friend class CPoint2D<float>;`
- Classe amie d'une instance particulière du patron de fonctions minimum : `friend double minimum(double,double);`
- On peut dans les déclarations précédentes utiliser un type T du patron de classes pour coupler l'amitié et l'instance de la classe.