

Autres aspects de la POO

1. Conversion dynamique

a. Conversions depuis un autre type

Les constructeurs permettent de convertir des objets à partir d'instances (de valeurs) exprimées dans un autre type.

Prenons le cas de notre classe `Chaine`. Il serait intéressant de "convertir" un `char*` ou un `char` en chaîne :

```
#include <string.h>

class Chaine
{
private:
    char*buffer;
    int t_buf;
    int longueur;
public:
    // un constructeur par défaut
    Chaine()
    {
        t_buf=100;
        buffer=new char[t_buf];
        longueur=0;
    }
    Chaine (int t_buf)
    {
        this->t_buf=t_buf;
        buffer=new char[t_buf];
        longueur=0;
    }

    Chaine(char c)
    {
```

```

    t_buf=1;
    longueur=1;
    buffer=new char[t_buf];
    buffer[0]=c;
}

Chaine(char*s)
{
    t_buf=strlen(s)+1;
    buffer=new char[t_buf];
    longueur=strlen(s);
    strcpy(buffer,s);
}

void afficher()
{
    for(int i=0; i<longueur; i++)
        printf("%c",buffer[i]);
    printf("\n");
}
};

int main(int argc, char* argv[])
{
    // Ecriture 1
    // Conversion par emploi explicite de constructeur
    Chaine x;
    x=Chaine("bonjour"); // conversion
    x.afficher();

    // Ecriture 2
    // Transtypage (cast) par coercion
    Chaine y=(char*) "bonjour"; // conversion (cast)
    y.afficher();

    // Ecriture 3
    // Transtypage (cast), approche C++
    Chaine z=char('A'); // conversion (cast)
    z.afficher();
    return 0;
}

```

```
}
```

La fonction `main()` illustre les trois façons de convertir vers un type objet :

- par l'emploi d'un constructeur explicite (1) ;
- par l'emploi d'un transtypage, comme en C (2) ;
- en utilisant la syntaxe propre à C++ (3).

Nous verrons que la surcharge de l'opérateur de conversion permet d'aller plus loin, notamment pour convertir un objet vers un type primitif. En conjonction avec la surcharge de l'opérateur d'affectation, il devient possible de réaliser pratiquement n'importe quel type de conversion, sans avoir à recourir à des méthodes spécialisées (`ToInt`, `ToString`, `ToChar`, `FromInt`, `FromString` ...).

b. Opérateurs de conversion

Le langage C++ dispose de deux opérateurs de conversion spécifiques :

```
const_cast<type> expressionconst_cast
```

Convertit une expression comme le ferait `(type) expression` si le type de `expression` diffère uniquement par les modificateurs `const` ou `volatile`.

```
reinterpret_cast<type> expressionreinterpret_cast
```

Convertit un pointeur en un autre pointeur sans trop se soucier de la cohérence de l'opération.

Le premier opérateur, `const_cast`, sert à oublier provisoirement l'utilisation du modificateur `const`. Les fonctions constantes, marquées `const` après leur prototype, engagent à ne pas modifier les champs de la classe. Toutefois, cette situation peut être parfois gênante. `const_cast` autorise la conversion d'un pointeur de classe vers une classe identique, mais débarrassée des marqueurs `const`.

```

class CCTest
{
public:
    void setNombre( int );
    void afficheNombre() const;
private:
    int nombre;
};

void CCTest::setNombre(int num)
{
    nombre = num;
}

void CCTest::afficheNombre() const
{
    printf("Avant: %d",nombre);
    const_cast< CCTest * >( this )->nombre--;
    printf("Après %d",nombre);
}

int main()
{
    CCTest X;
    X.setNombre( 8 );
    X.afficheNombre();
}

```

C'est dans la méthode `afficheNombre` que l'interdiction a été provisoirement levée. Signalons également que cet opérateur ne permet pas directement de lever l'interdiction de modification sur des champs `const`.

Quant à l'opérateur `reinterpret_cast`, il se révèle potentiellement assez dangereux et ne devrait être utilisé que pour des fonctions de bas niveau, comme dans l'exemple du calcul d'une valeur de hachage basée sur la valeur entière d'un pointeur :

```

unsigned short Hash( void *p )
{

```

```

    unsigned int val = reinterpret_cast<unsigned int>(p);
    return ( unsigned short )( val ^ (val >> 16));
}

```

c. Conversions entre classes dérivées

Deux opérateurs supplémentaires contrôlent les conversions de type, tout particulièrement lorsqu'il s'agit de classes dérivées. Lors de l'étude des méthodes virtuelles, nous avons envisagé deux classes, `Base` et `Derive`, la seconde dérivant de la première.

L'opérateur `static_cast` s'assure qu'il existe une conversion implicite entre les deux types considérés. De ce fait, il est dangereux de l'utiliser pour remonter un pointeur dans la hiérarchie des classes, c'est-à-dire d'utiliser un pointeur `Derive*` pour désigner un `Base*`. Finalement, l'opérateur `static_cast`, qui donne la pleine mesure de son action lors de la compilation, est beaucoup plus utile pour réaliser des conversions propres entre des énumérations et des entiers.

```

enum Couleurs { rouge, vert, bleu };

int conv_couleur(Couleurs coul)
{
    return static_cast<int>( coul );
}

```

Pour utiliser l'opérateur `dynamic_cast`, il faut parfois activer un commutateur du compilateur, comme `ENABLE_RTTI` (*Run Time Type Information*) pour le compilateur Microsoft. Il devient alors trivial d'envisager des conversions pour lesquelles un doute subsiste quant au type réellement manipulé : `dynamic_cast`.

```

void conie_sans_risque(Base*b)
{
    Derive*d1 = dynamic_cast<Derive*>(b);
}

```

Si l'argument désigne effectivement un objet de type `Base`, sa promotion directe en

`Derive` (par coercion) est assez risquée. L'opérateur `dynamic_cast` renverra `NULL`, voire lèvera une exception. Toutefois, la détection devra attendre l'exécution pour déterminer le type exact représenté par `b`.

2. Champs et méthodes statiques

a. Champs statiques

Un champ statique est une variable placée dans une classe mais dont la valeur est indépendante des instances de cette classe : elle n'existe qu'en un seul et unique exemplaire. Quelle est l'opportunité d'une telle variable ? Tout d'abord, il existe des situations pour lesquelles des variables sont placées dans une classe par souci de cohérence.

Envisageons la classe `Mathematiques` et la variable `PI`. Il est logique de rattacher `PI` dans le vocabulaire mathématique, autrement dit de classer `PI` dans `Mathematiques`. Mais la valeur de `PI` est constante, et elle ne saurait de toute façon varier pour chaque instance de cette classe. Pour évoquer ce comportement, il a été choisi de marquer le champ `PI` comme étant **statique**.

Deuxièmement, les méthodes statiques (cf. partie suivante) n'ont pas accès aux variables d'instance, c'est-à-dire aux champs. Si une méthode statique a besoin de partager une valeur avec une autre méthode statique, il ne lui reste plus qu'à adresser une variable statique. Attention, il s'agit bien d'un champ de la classe et non d'une variable locale statique, disposition qui a heureusement disparu dans les langages de programmation actuels.

```
class Mathematiques
{
public :
    static double PI ;
};
double Mathematiques ::PI=3.14159265358 ;
```

Vous aurez noté que la déclaration d'un champ statique n'équivaut pas à son allocation. Il

est nécessaire de définir la variable à l'extérieur de la classe, en utilisant l'opérateur de résolution de portée `::` pour réellement allouer la variable.

Comme autre exemple de champ statique, reportez-vous à l'exemple précédent de la classe `Compte`. Le prochain numéro de compte, un entier symbolisant un compteur, est incrémenté par une méthode elle-même statique.

Soulignons qu'il y a une forte similitude entre un champ statique, appartenant à une classe, et une variable déclarée dans un espace de noms, que d'aucuns appellent module.

b. Méthodes statiques

Les méthodes statiques reprennent le même principe que les champs statiques. Elles sont déclarées dans une classe, car elles se rapportent à la sémantique de cette classe, mais sont totalement autonomes vis-à-vis des champs de cette classe. Autrement dit, il n'est pas nécessaire d'instancier la classe à laquelle elles appartiennent pour les appliquer, ce qui signifie qu'on ne peut les appliquer à des objets.

Restons pour l'instant dans le domaine mathématique. La fonction `cosinus(angle)` peut être approchée à partir d'un développement limité, fonction mathématique très simple :

$$\text{cosinus}(\text{angle}) = \sum (\text{angle}_i / i!)$$

pour $i = 2 * k$, k variant de 0 à n .

Nous en déduisons pour C++ la fonction `cosinus` :

```
double cosinus(double a)
{
    return 1-a*a/2+a*a*a*a/24+a*a*a*a*a*a/720 ;
}
```

Nous remarquons que cette fonction est à la fois totalement autonome, puisqu'elle ne consomme qu'un argument `a`, et qu'elle se rapporte au domaine mathématique. Nous décidons d'en faire une méthode statique :

```

class Mathematiques
{
public :
double cosinus(double a)
{
    return 1-a*a/2+a*a*a*a/24+a*a*a*a*a*a/720 ;
}

};

```

Maintenant, pour appeler cette méthode, nous n'avons qu'à l'appliquer directement à la classe `Mathematiques` sans avoir besoin de créer une instance :

```

double angle=Mathematique ::PI/4 ;
printf("cos(pi/4)=%g",Mathematiques::cosinus(a));

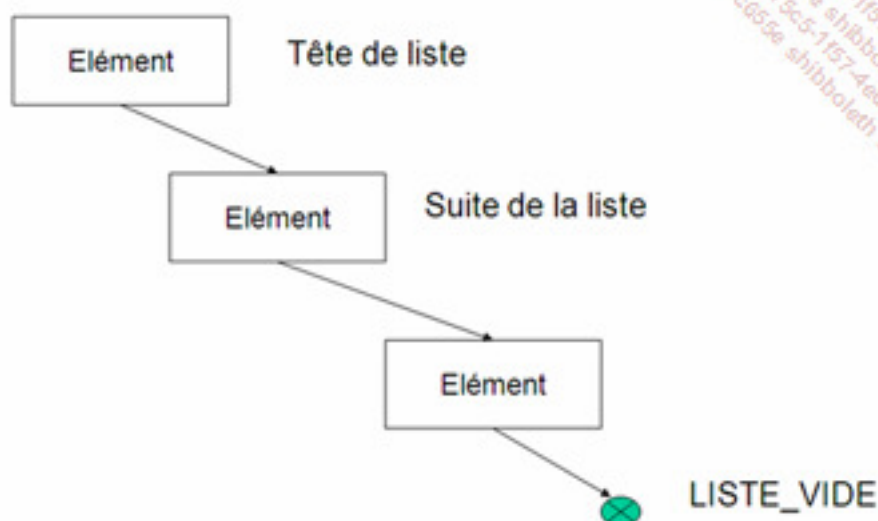
```



Le résultat approche $\pi/2$, soit 0,707. Le microprocesseur ne s'y prend pas autrement pour fournir la valeur d'un cosinus à la fonction `cos()` disponible dans `<math.h>`. Certains processeurs numériques factorisent la variable `a` et utilisent une table de poids pour augmenter la rapidité du calcul.

Nous donnons maintenant un exemple de structure dynamique, la liste, qui a la faveur des enseignements en informatique. Cette structure stocke des éléments de natures diverses - entiers, chaînes... - et sa taille augmente au fur et à mesure des besoins.

Une liste



On utilise en général les fonctions suivantes pour manipuler les listes : `cons()` qui sera notre constructeur, `suite()` et `est_vide()` qui teste la nullité d'une liste. Les applications des listes sont innombrables, et leur apprentissage est un plus pour tout programmeur.

Traditionnellement traitée en langage C, la liste gagne à être implémentée en C++, notamment par l'emploi de méthodes statiques. Voici donc l'implémentation proposée. Nous commençons par le fichier `Liste.h`, qui possède plusieurs membres statiques.

```

#pragma once

#include <stdio.h>

class Liste
{
public:
    char*element;
    Liste* _suite;
    static const Liste* LISTE_VIDE;

    Liste()
    {
        _suite=const_cast<Liste*>(LISTE_VIDE);
    }
}
  
```

```

    }

    Liste(char*e,Liste*suite)
    {
        _suite=suite;
        element=e;
    }

    Liste* suite()
    {
        return _suite;
    }

    static bool est_vide(Liste* l)
    {
        return l==LISTE_VIDE;
    }

    void afficher()
    {
        Liste::afficher(this);
    }

    static void afficher(Liste*l)
    {
        if(Liste::est_vide(l))
            return;
        printf("%s",l->element);
        Liste::afficher(l->suite());
    }
};

```

La fonction `est_vide` étant totalement autonome, il est logique de la définir statique. L'affichage d'une liste est grandement facilité par l'écriture d'une fonction récursive prenant une liste comme paramètre. De ce fait, la fonction ne s'applique à aucune instance en particulier et la méthode correspondante devient elle-même statique. L'exposition de cette méthode récursive avec le niveau public ou privé est un choix qui appartient au programmeur.

Nous trouvons ensuite dans le fichier `Liste.cpp` la définition du champ

statique `LISTE_VIDE`. Notez au passage la combinaison des modificateurs `static` et `const`, combinaison fréquente s'il en est.

```
#include "..\liste.h"

const Liste* Liste::LISTE_VIDE=NULL;
```

La définition déportée de cette variable ne pouvait en aucune manière figurer dans le fichier `Liste.h`, car son inclusion par plusieurs modules `cpp` aurait entraîné sa duplication entre chaque module, rendant ainsi impossible l'édition des liens.

La fonction `main()` crée un exemple de liste puis appelle la méthode `afficher()` :

```
#include "Liste.h"

int main(int argc, char* argv[])
{
    Liste* liste=new Liste("bonjour",
        new Liste("les",
            new Liste("amis",
                const_cast <Liste*> (Liste::LISTE_VIDE) )));
    liste->afficher();
    return 0;
}
```

Au passage, vous aurez noté l'emploi de l'opérateur `const_cast` pour accorder précisément les types manipulés.



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The output of the program is displayed as follows:

```
bonjour,les,amis,
Appuyez sur une touche pour continuer... █
```

3. Surcharge d'opérateurs

Le concepteur de C++ a voulu que son langage soit le plus expressif possible ; définir des classes et créer de nouveaux types est une chose, conserver une programmation simple et claire en est une autre. La surcharge des opérateurs, c'est-à-dire leur adaptation à des classes définies par le programmeur, va justement dans cette direction.

Un opérateur est en fait assimilable à une fonction. Cette fonction réunit un certain nombre d'opérandes et évalue un résultat d'un type particulier. Pour qu'un opérateur soit surchargé, il doit figurer dans une classe comme n'importe quelle méthode.

a. Syntaxe

En fait, il est laissé une très grande liberté au programmeur dans le type des arguments applicables aux opérateurs surchargés. Le respect de la commutativité, des priorités, de la sémantique même d'un opérateur peuvent être remis en cause sans que le compilateur ne trouve à y redire.

La syntaxe est la suivante :

```
class Classe
{
    type_retour operator op (type_op operande)
    {
        ...
    }
};
```

Cet opérateur, `op`, est destiné à figurer entre une instance de la classe `Classe` et un opérande de type `type_op`. La fonction est libre de modifier l'instance à laquelle elle est appliquée et peut renvoyer un résultat de n'importe quel type, même `void`.

Pour illustrer cette notation, reprenons l'exemple de la classe `Chaine` vu précédemment. Ajoutons la surcharge de l'opérateur `+` dans le but de concaténer un unique caractère :

```
Chaine& operator+(char c)
{
    buffer[longueur++] = c;
```

```
return *this;
}
```

Il est assez judicieux de renvoyer la **référence de l'objet courant**, ce qui permet d'enchaîner les opérations :

```
Chaine c("bonjour ");
c = c+'V';
c + 'o';
Chaine d;
d = c + 'u' + 's'; // enchaînement
d.afficher(); // affiche bonjour Vous
```

Vous aurez noté que notre opérateur + effectue une modification de l'instance à laquelle elle est appliquée, alors qu'en termes mathématiques additionner deux variables ne modifie ni l'une, ni l'autre. Il aurait été possible à la place de construire une nouvelle chaîne et de se contenter d'évaluer la concaténation. Nous gagnerions ainsi en souplesse mais perdriions vraisemblablement en performance.

Pratiquement tous les opérateurs sont disponibles pour la surcharge, à l'exception notable de ceux figurant dans le tableau suivant :

.	Accès aux champs.
::	Résolution de portée.
sizeof	Taille d'un type.
? :	Forme prédicative (voir <code>if</code>).
.*	Adressage relatif.

Ces exceptions faites, le programmeur a donc une grande liberté dans les sémantiques proposées par les opérateurs, à condition toutefois de conserver le nombre d'opérandes (binaire, unaire), la précedence, de respecter l'associativité. Le programmeur ne peut pas non plus déterminer l'adresse d'un opérateur (pointeur de fonction), ni proposer des valeurs par défaut pour les arguments applicables.

Les opérateurs peuvent être implémentés sous forme de méthodes ou de fonctions, ces dernières ayant tout intérêt à être déclarées amies (`friend`). L'intérêt de cette approche est qu'il est possible d'inverser l'ordre des opérandes et d'augmenter ainsi le vocabulaire d'une classe existante.

Cette technique est d'ailleurs mise à profit pour la surcharge de l'opérateur `<<` applicable à la classe `std::ostream`.

b. Surcharge de l'opérateur d'indexation

Nous proposons son implémentation car il convient parfaitement à l'exemple de la classe `Chaine`. Cet opérateur est utile pour accéder à la nième donnée contenue dans un objet. Dans notre cas, il s'agit d'énumérer chaque caractère contenu dans une chaîne :

```
int length()
{
    return longueur;
}

char operator[](int index)
{
    return buffer[index];
}
```

L'affichage caractère par caractère devient alors trivial :

```
for(int i=0; i<d.length(); i++)
    printf("%c ",d[i]);
```

c. Surcharge de l'opérateur d'affectation

L'opérateur d'affectation est presque aussi important que le constructeur de copie. Il est fréquent de définir les deux pour une classe donnée.

```
Chaine& operator=(const Chaine& ch)
{
    if(this != &ch)
    {
        delete buffer;
        buffer = new char[ch.t_buf]; // ch référence
        for(int i=0; i<ch.longueur; i++)
            buffer[i] = ch.buffer[i];
        longueur = ch.longueur;
    }
    return *this;
}
```

d. Surchage de l'opérateur de conversion

L'opérateur de conversion réalise un travail symétrique aux constructeurs prenant des types variés comme arguments pour initialiser le nouvel objet. Dans le cas de la classe `Chaine`, la conversion vers un `char*` est absolument nécessaire pour garantir une bonne traduction avec les chaînes gérées par le langage lui-même. Faites attention à la syntaxe qui est très particulière :

```
operator char*()
{
    char*out;
    out = new char[longueur+1];
    buffer[longueur] = 0;
    strcpy(out,buffer);
    return out;
}
```

L'application est beaucoup plus simple :

```
char*s=(char*) d;
```

```
printf("d=%s",s);
```

Cet opérateur sert à convertir par transtypage (`cast`) un objet de type `Chaine` vers le type `char*`.

4. Fonctions amies

Les fonctions amies constituent un recours intéressant pour des fonctions n'appartenant pas à des classes mais devant accéder à leurs champs privés.

S'agissant de fonctions et non de méthodes, elles n'admettent pas de pointeur `this`, aussi reçoivent-elles généralement une instance de la classe comme paramètre, ou bien instancient-elles la classe en question.

```
class Chaine
{
...
friend void afficher(Chaine&);
};

void afficher(Chaine& ch)
{
    for(int i=0; i<ch.longueur; i++)
        printf("%c",ch.buffer[i]);
}
```

Dans la déclaration de la classe `Chaine`, nous avons volontairement placé la déclaration d'amitié sans préciser s'il s'agissait d'un bloc public, privé ou protégé. Puisque la fonction `afficher()` n'est pas un membre, l'emplacement de sa déclaration n'a aucune importance.

Les fonctions amies présentent un réel intérêt pour surcharger des opérateurs dont le premier argument n'est pas la classe que l'on est en train de décrire. L'exemple habituel consiste à surcharger l'opérateur d'injection `<<` appliqué à la classe `ostream` :


```

#include <iostream>
using namespace std;

class Chaîne
{
private:
    int t_buf;
    char*buffer;
    int longueur;
public:

    friend ostream& operator<<(ostream& out,Chaîne&);
    ...
};

ostream& operator<<(ostream& out, Chaîne& ch)
{
    for(int i=0; i<ch.longueur; i++)
        out << ch[i]; // << surchargé pour ostream et char

    return out;
}

int main(int argc, char* argv[])
{
    Chaîne c("bonjour ");
    cout << c << "\n"; // cout est un objet de type ostream
}

```

Signalons encore que les relations d'amitié ne sont pas héréditaires et qu'il est possible de définir des relations d'amitié entre méthodes. Ces usages sont toutefois à limiter pour augmenter la portabilité des programmes vers des langages ne connaissant pas ce concept.

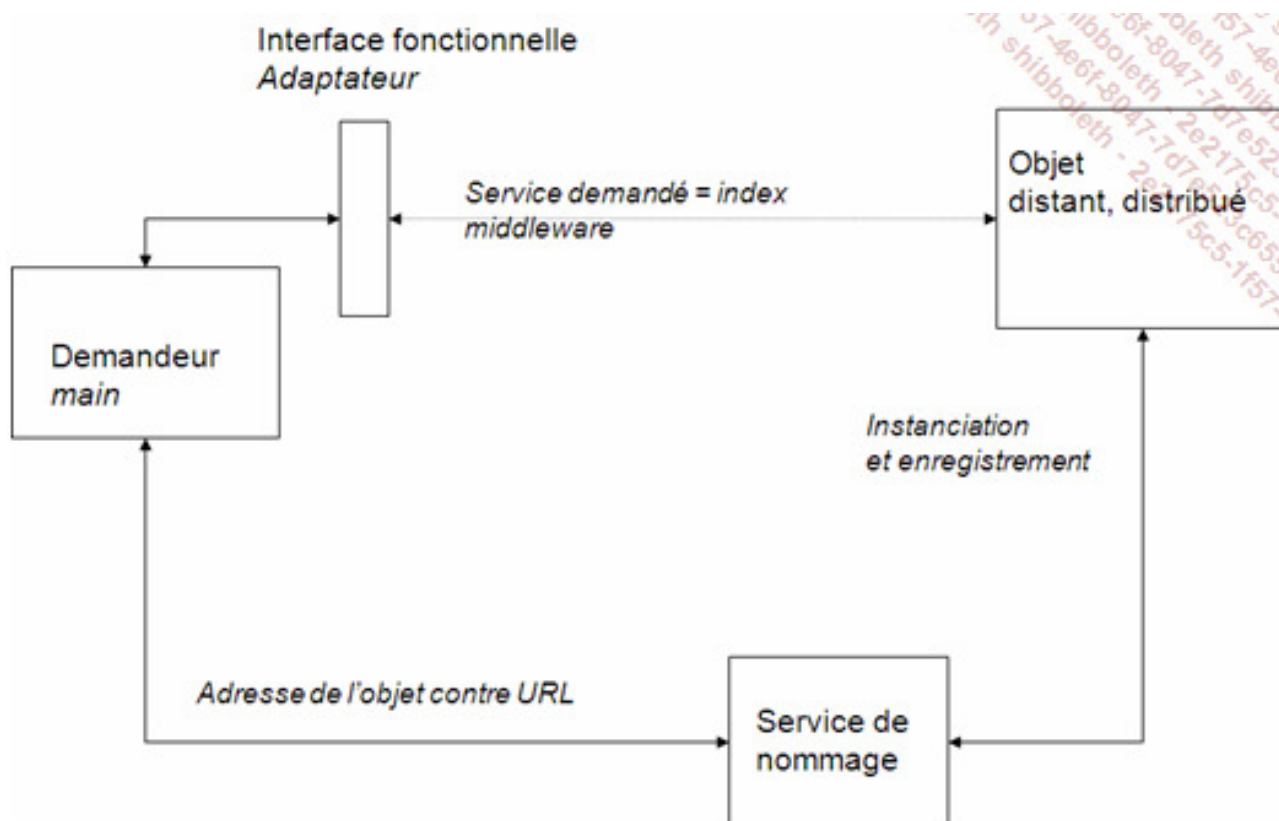
5. Adressage relatif et pointeurs de membres

Le langage C++ a inauguré une nouvelle technique de programmation, la programmation dite distribuée. Cette technique consiste à appeler sur une machine

distante des méthodes appliquées à un objet dont on ne connaît que l'interface. Rappelons-nous que l'interface équivaut à l'ensemble des méthodes publiques d'une classe.

Dans pareille situation, la notion de pointeur au sens où nous l'avons étudié jusqu'à présent est inapplicable. Un pointeur désigne une adresse appartenant à la machine courante, voire au processus courant. Pour invoquer une méthode, ou accéder à un champ à distance, un numéro de membre serait plus approprié, car il est indépendant de l'emplacement mémoire de l'objet. Bien entendu, au dernier moment, ce déplacement sera combiné à une adresse de base, physique celle-là, pour accéder au membre considéré.

Cette adresse de base n'est généralement pas connue du demandeur, il l'obtient à partir d'un service de nommage, souvent au moyen d'une URL appropriée.



De nombreux middlewares orientés objet, tels DCOM, Corba, RMI ou .NET Remoting, travaillent de cette façon. Beaucoup d'entre eux sont d'ailleurs implémentés en C++.

Les pointeurs de membres de C++ permettent justement d'exprimer l'accès à un membre en tant qu'index et non en tant qu'adresse physique.

a. Notations

Trois opérateurs sont dévoués à la prise en charge de l'adressage relatif, terme désignant en fait les pointeurs de membres :

- ˆ l'opérateur `&` (adresse relative d'un membre) ;
- ˆ l'opérateur `.*` (indirection sur une adresse relative) ;
- ˆ l'opérateur `->*` (indirection sur une adresse relative).

Le premier opérateur retrouve l'adresse relative d'un membre, en suivant la logique habituelle :

```
Chaine::* index_buffer=&Chaine::buffer;
```

Le type du pointeur est donc `Classe::*`.

Les autres opérateurs combinent l'index à une référence ou à une adresse d'objet, encore une fois en suivant la logique habituelle :

```
Chaine ch;
ch.*index_buffer=new char[100];
```

Pour les pointeurs relatifs de méthode, la signature est aussi conforme aux pointeurs de fonctions usuels :

```
int (Chaine::* pf_length)(void);
pf_length=&Chaine::length;
```

b. Construction d'un middleware orienté objet

Nous proposons d'appliquer les pointeurs de membres à la simulation d'un middleware servant à distribuer nos applications. Dans notre cas, tout fonctionnera à l'intérieur d'un seul processus mais le raisonnement est aisément généralisable à une infrastructure plus complexe.



Un middleware est un ensemble de services système et réseau destiné à faire communiquer des logiciels fonctionnant sur des machines différentes. Il existe des middlewares orientés messages (Mom) dont MQSeries (IBM) et MSMQ sont les plus connus. Il s'agit ici d'étudier un middleware orienté objet (Moo).

Nous allons compléter le schéma présenté en introduction de cette partie à l'aide d'une classe `Adaptateur`. Il s'agit de la couche intermédiaire (middleware) chargée de relayer les invocations du demandeur à travers le "réseau" pour atteindre le véritable objet, désigné sous le nom d'objet distant.

Pour le demandeur, le scénario est le suivant :

1. accès au service de nommage (une seule instanciation).
2. localisation de l'objet distant contre la fourniture d'une URL.
3. accès aux méthodes distantes.

Pour illustrer notre exemple, nous proposons une interface `IObjetDistant` possédant deux méthodes, l'une se contentant de renvoyer la chaîne de caractères "Bonjour tout le monde" et la seconde, plus élaborée, additionnant deux nombres entiers transmis en paramètres.

Interface `IObjetDistant`

Voici pour commencer l'interface `IObjetDistant`, elle ne contient que des méthodes virtuelles pures :

```
#pragma once

// Fichier: IObjetDistant
// Propos : Cette interface contient la liste des méthodes
//          pouvant être appelées à distance.
//          Pour simplifier, toutes les méthodes ont la
//          même signature.
```

```
class IObjetDistant
{
public:
    virtual void* hello_world(int nb_params, void* params)=0;
    virtual void* somme(int nb_params, void* params)=0;
};
```

Implémentation ObjetDistant

Nous poursuivons avec son implémentation, qui ignore totalement qu'elle est amenée à être `ObjetDistant` distribuée, c'est-à-dire instanciée et invoquée à distance :

```
#pragma once

// Fichier: ObjetDistant.h

// Propos : Implémentation de la classe (interface) IObjetDistant
#include "IOjetDistant.h"

class ObjetDistant : public IObjetDistant
{
public:
    void* hello_world(int nb_params, void* params);
    void* somme(int nb_params, void* params);
};
```

Puis :

```
#include "..\objetdistant.h"

// la classe ObjetDistant ignore totalement qu'elle va être distribuée,
// c'est-à-dire appelée à distance.

void* ObjetDistant::hello_world(int nb_params, void* params)
{
    return "Bonjour tout le monde"; // char* en fait
}
```

```

void* ObjetDistant::somme(int nb_params,void*params)
{
    int*local_params=(int*) params;

    // pour renvoyer le résultat, un reinterpret_cast aurait été possible
    // mais peu rigoureux.
    int*resultat=new int[1];
    *resultat=local_params[0]+local_params[1];
    return (void*)resultat ;
}

```

Adaptateur

Nous entrons ensuite dans le vif du sujet avec la classe `Adaptateur`, qui utilise des pointeurs de membres à plusieurs niveaux.

Dans le cas d'une réelle distribution, il suffira de prévoir deux classes (`skeleton` et `stub`) traduisant l'adresse de l'objet `l_objet` entre les deux machines.

```

#pragma once

// Fichier: Adaptateur.h
// Propos : Classe permettant l'invocation sécurisée de méthodes
//          sur l'objet distant.
#include "IObjetDistant.h"

class Adaptateur : public IObjetDistant
{
private:
    IObjetDistant*l_objet;

    void* invoquer_par_numero(
        void* (IObjetDistant::* p_fonction) (int,void*),
        int nb_params,void*params);
public:
    Adaptateur(IObjetDistant*adr);

    void* hello_world(int nb_params,void*params);
    void* somme(int nb_params,void*params);
}

```

```
};
```

L'implémentation suit naturellement. Notez comment les méthodes `hello_world()` et `somme()` se contentent d'appeler les véritables implémentations au moyen de pointeurs de membres :

```
#include ".\adaptateur.h"

Adaptateur::Adaptateur(IObjetDistant*adr)
{
    l_objet=adr; // mémorise l'adresse physique de l'objet distant
}

void* Adaptateur::invoquer_par_numero(
    void* (IObjetDistant::* p_fonction) (int,void*) ,
    int nb_params,void*params)
{
    // Le pointeur de membre est utilisé ici.
    // L'appel combine l'adresse physique, l_objet,
    // avec l'index de la fonction à appeler, p_fonction.
    return (l_objet->*p_fonction) (nb_params,params);
}

void* Adaptateur::hello_world(int nb_params,void*params)
{
    return invoquer_par_numero(
        &IObjetDistant::hello_world, // index de la fonction
        nb_params,params);
}

void* Adaptateur::somme(int nb_params,void*params)
{
    return invoquer_par_numero(
        &IObjetDistant::somme, // index de la fonction
        nb_params,params);
}
```

Service de nommage

Le service de nommage est assez basique. Il assure ses responsabilités d'enregistrement

d'objet (publication) avec celles d'un serveur d'objets (instanciation). Dans une situation réelle de distribution, l'URL désignant l'objet ne se résume pas à un seul segment indiquant le nom. Y figure souvent aussi l'adresse du serveur où l'objet est instancié.

```
#pragma once

// Fichier: Nommage.h
// Propos : Système d'enregistrement et de localisation
// d'objets IObjetDistant
// Pour simplifier, la classe publie automatiquement un objet
#include "IOjetDistant.h"
#include "Adaptateur.h"

class Nommage
{
private:
    IObjetDistant* objet;
    char*url;

public:
    void publier(IOjetDistant*objet,char*url);
    IObjetDistant*trouver(char*url);
    Nommage();
    ~Nommage();
};
```

L'implémentation est plus intéressante que la déclaration car elle substitue, à la demande du client, l'objet distant par sa version distribuée (Adaptateur) :

```
#include ".\nommage.h"
#include "objetdistant.h"
#include <string.h>

Nommage::Nommage()
{
    // procède à la publication d'un nouvel objet
    printf("Instanciation d'un objet sous le nom objet1\n");
    publier(new ObjetDistant,"objet1");
}
```



```

Nommage::~Nommage()
{
    // détruit l'objet
    delete this->objet;
    this->url=NULL;
}

void Nommage::publier(IObjetDistant*objet,char*url)
{
    printf("Publication d'un objet sous le nom %s\n",url);
    this->objet=objet;
    this->url=url;
}

IObjetDistant*Nommage::trouver(char*url)
{
    printf("Recherche d'un objet sous le nom %s\n",url);
    if(!strcmp(this->url,url))
        return new Adaptateur(this->objet);

    printf("Objet pas trouvé\n");
    return NULL;
}

```

Programme client (demandeur)

La fonction `main()` fait office du demandeur dans notre schéma. Elle est conforme au scénario évoqué ci-dessus :

```

#include "nommage.h"

int main(int argc, char* argv[])
{
    // Utilise le service de nommage ad hoc
    Nommage nommage;

    // recherche d'un objet distant contre une URL

```

```

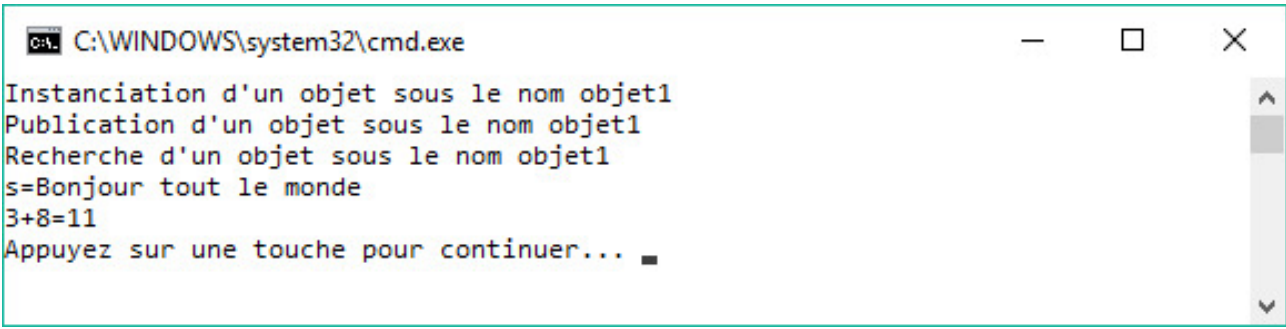
IObjetDistant*objet_distant= nommage.trouver("objet1");

// appel d'une méthode à distance
char*s=(char*) objet_distant->hello_world( 0,NULL);
printf("s=%s\n",s);

// appel d'une autre méthode, à distance toujours
int tab[2]={3,8};
int*r=(int*) objet_distant->somme(2,(void*) tab );
printf("3+8=%d\n",*r);
return 0;
}

```

Après tous ces efforts de programmation, nous proposons les résultats de l'exécution du programme :



```

C:\WINDOWS\system32\cmd.exe
Instanciation d'un objet sous le nom objet1
Publication d'un objet sous le nom objet1
Recherche d'un objet sous le nom objet1
s=Bonjour tout le monde
3+8=11
Appuyez sur une touche pour continuer... █

```

6. Programmation générique

Le langage C est trop proche de la machine pour laisser entrevoir une réelle généricité. Au mieux, l'emploi de pointeurs `void*` (par ailleurs introduits par C++) permet le travail sur différents types. Mais cette imprécision se paye cher, tant du point de vue de la lisibilité des programmes que de la robustesse ou des performances.

Les macros ne sont pas non plus une solution viable pour l'implémentation d'un algorithme indépendamment du typage. Les macros développent une philosophie inverse aux techniques de la compilation. Elles peuvent convenir dans certains cas simples mais leur utilisation relève le plus souvent du bricolage.

Reste que tous les algorithmes ne sont pas destinés à être implémentés pour l'universalité des types de données C++. Pour un certain nombre de types, on se met alors à penser en termes de fonctions polymorphes.

Finalement, les modèles C++ constituent une solution bien plus élégante, très simple et en même temps très sûre d'emploi. Le type de donnée est choisi par le programmeur qui va instancier son modèle à la demande. Le langage C++ propose des modèles de fonctions et des modèles de classes et nous allons traiter tour à tour ces deux aspects.

a. Modèles de fonctions

Tous les algorithmes ne se prêtent pas à la construction d'un modèle. Identifier un algorithme est une précaution simple à prendre car un modèle bâti à mauvais escient peut diminuer les qualités d'une implémentation.

Pour l'algorithmie standard, la bibliothèque STL a choisi d'être implémentée sous forme de modèles. Les conteneurs sont des structures dont le fonctionnement est bien entendu indépendant du type d'objet à mémoriser.

Pour les chaînes, le choix est plus discutable. La STL propose donc un modèle de classes pour des chaînes indépendantes du codage de caractère, ce qui autorise une ouverture vers des formats très variés. Pour le codage le plus courant, l'ASCII, la classe `string` est une implémentation spécifique.

Le domaine numérique est particulièrement demandeur de modèles de fonctions et de classes. Le programmeur peut ainsi choisir entre précision (`long double` ou `double`) et rapidité (`float`) sans remettre en question l'ensemble de son programme. Certains algorithmes peuvent même travailler avec des représentations de nombres plus spécifiques.

Une syntaxe spécifique explicite la construction d'un modèle de fonction. Il s'agit d'un préfixe indiquant la liste des paramètres à fournir à l'instanciation du modèle. Les paramètres sont fréquemment écrits en majuscules pour les distinguer des variables dans le corps de la fonction, mais il ne s'agit aucunement d'une contrainte syntaxique.

```
template<class T> T cosinus(T a)
{
```

```

int i;
T cos = 1;
T pa = a*a;
int pf = 2;
T signe = -1;

for(i=2; i<=14; i+=2)
{
    cos += signe*pa/pf;
    signe = -signe;
    pa = pa*a*a;
    pf = pf*(i+1);
    pf = pf*(i+2);
}
return cos;
}

```

Le modèle doit avant tout se lire comme une fonction dont le type est paramétrable. Imaginons que le type soit `double`, nous obtenons la signature suivante :

```
double cosinus(double a)
```

Dans le corps de la méthode, le type `T` s'écrit à la place du type des variables ou des expressions. Ainsi les variables `cos`, `pa` et `signe` pourraient être des `double` ou des `float`.

Il va sans dire que la liste des paramètres du modèle, spécifiée entre les symboles `<` et `>`, peut être différente de la liste des paramètres de la fonction, ainsi que le prouve l'exemple suivant :

```

template<class T> void vecteur_add(T*vect,int n,T valeur)

{
    for(int i=0;i<n;i++)
        vect[i]+=valeur;
}

```

Pour utiliser un modèle de fonction, il faut simplement appeler la fonction avec des arguments qui correspondent à ceux qu'elle attend. Cela a pour effet de développer le modèle dans une version adaptée aux types des arguments transmis, arguments qui peuvent naturellement être des valeurs littérales, des expressions ou encore des variables, en respectant les règles habituelles d'appel de fonction.

Nous proposons un premier exemple avec notre modèle de fonction `cosinus`. Dans cet exemple, nous comparons la durée du calcul d'un nombre important de cosinus, ainsi que la précision du calcul. Suivant le type de l'argument passé à la fonction `cosinus` - `float` ou `long double` -, le compilateur produira plusieurs versions à partir du même modèle.

```
int main(int argc, char* argv[])
{
    time_t start, finish;
    double result, elapsed_time;

    int N=1<<28;
    // cosinus en float
    time( &start );
    float a1=3.14159265358F/2;
    float r1;
    for(int i=0; i<N; i++)
        r1=cosinus(a1);
    time( &finish );
    elapsed_time = difftime( finish, start );

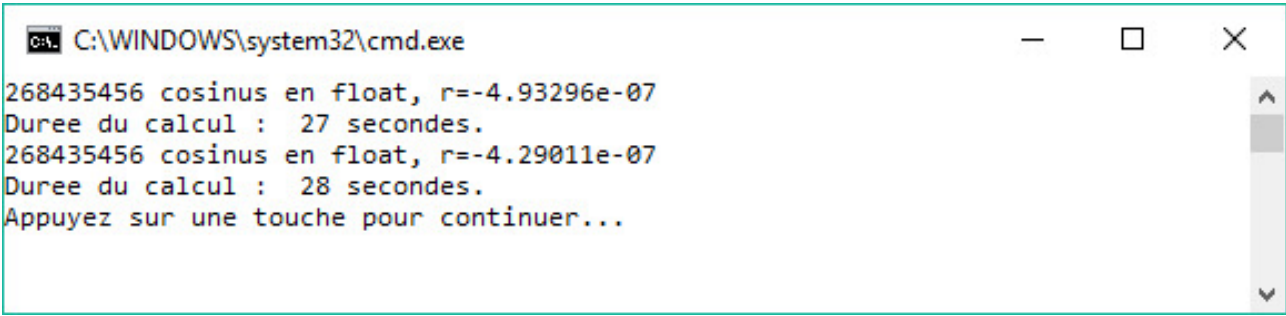
    printf("%d cosinus en float, r=%f\n",N,r1);
    printf("Durée du calcul : %6.0f secondes.\n", elapsed_time );

    // cosinus en long double
    time( &start );
    long double a2=3.14159265358/2;
    long double r2;
    for(i=0; i<N; i++)
        r2=cosinus(a2);
    time( &finish );
    elapsed_time = difftime( finish, start );
```

```
printf("\n%d cosinus en long double, %g\n",N,r2);
printf( "Durée du calcul : %6.0f secondes.\n", elapsed_time );

return 0;
}
```

Nous constatons des différences sensibles dans la durée et la qualité du calcul.



```
C:\WINDOWS\system32\cmd.exe
268435456 cosinus en float, r=-4.93296e-07
Duree du calcul : 27 secondes.
268435456 cosinus en float, r=-4.29011e-07
Duree du calcul : 28 secondes.
Appuyez sur une touche pour continuer...
```

Munis de ces informations, le concepteur et le programmeur peuvent choisir le type de données le plus adapté à leur cahier des charges.

Nous proposons maintenant une application du modèle `vecteur_add()`. Un éditeur de code source performant guide le programmeur dans la saisie des arguments passés à la fonction :

Voilà maintenant le code complété :

```
double valeurs[]={3,-9,8,2,5 };
vecteur_add(valeurs,4,3.0);
```

Le troisième paramètre de la fonction doit impérativement être du type correspondant au tableau de valeurs, dans notre cas pour des raisons d'homogénéité des calculs, mais aussi pour des raisons syntaxiques.

Fournir une valeur littérale ambiguë peut provoquer un avertissement, voire une erreur de compilation :

```
vecteur_add(valeurs,4,3); // 3 littérale d'entier
```

Sachant que cet appel doit être le plus explicite possible, nous en déduisons que le modèle de fonction peut être surchargé, autrement dit, polymorphe.

La surcharge d'un modèle de fonction consiste à décrire une autre fonction portant le même nom mais recevant des paramètres différents.

```
template<class T> void vecteur_add(T& vect,T valeur)
{
    vect+=valeur;
}
```

Les paramètres du modèle peuvent également varier d'une version à l'autre.

À l'appel de la fonction, le compilateur décide quelle est la version la plus appropriée.

Nous obtenons finalement le code suivant pour instancier notre nouveau modèle :

```
vecteur_add(valeurs[0],(double) 3);
```

Lorsqu'une optimisation de l'implémentation est possible pour un type donné, il est d'usage de spécialiser le modèle. Ainsi, nous pourrions écrire une version spécifique de `cosinus()` s'appuyant sur la bibliothèque mathématique lorsque le type est `double` :

```
template<double> double cosinus(double a)
{
    return cos(a);
}
```

En présence du modèle général `template<class T> T cosinus(T a)` et de la version spécialisée, le compilateur prendra la version spécialisée si le type considéré correspond exactement.

La classe `string` de la STL est un exemple de modèle spécialisé de classe `basic_string` appliqué aux `char`. Il va sans dire que les modèles de classes ont aussi la possibilité d'être spécialisés.

b. Modèles de classes

La spécification d'un modèle de classe suit exactement la même logique qu'un modèle de fonction. Nous donnons ici l'exemple d'une classe, `Matrice`, qui peut être implémentée avec différents types, au choix du programmeur.

```
template<class T> class Matrice
{
protected:
    int nb_vecteur,t_vecteur;
    T*valeurs;

public:
    // constructeur
    Matrice(int nb,int t) : nb_vecteur(nb),t_vecteur(t)
    {
        valeurs=new T[nb_vecteur*t_vecteur];
    }

    // référence sur le type T
    T& at(int vecteur,int valeur)
    {
        return valeurs[vecteur*t_vecteur+valeur];
    }

    // transtypage de double vers T
    void random()
    {
        int vect,val;
        for(vect=0; vect<nb_vecteur; vect++)
        {
            for(val=0; val<t_vecteur; val++)
                at(vect,val)=(T) (((double)rand()/RAND_MAX)*100.0);
        }
    }

    // affichage
    void afficher()
    {
        int vect,val;
```



```

cout.precision(2);
cout << fixed;
for(vect=0; vect<nb_vecteur; vect++)
{
    for(val=0; val<t_vecteur; val++)
    {
        cout.width(6);
        cout << at(vect,val) ;
        cout << (val<t_vecteur-1?" ":"");
    }
    cout << endl;
}
}
};

```

Les membres de la classe `Matrice` illustrent différents aspects des modèles de classe.

La classe contient un champ de type `T*`, comme nous avons pu déclarer précédemment une variable locale dans le corps de la fonction `cosinus`.

La méthode `at()` renvoie une référence vers un type `T`, noté `T&` bien entendu.

Le paramètre du modèle `T` peut être utilisé pour effectuer des conversions de type (transtypage) ; la méthode `random()` explicite le fonctionnement de la notation `(T)` pour initialiser de manière aléatoire chaque valeur de la matrice.

Quant à la fonction `afficher()`, elle est moins triviale qu'elle n'y paraît. Si nous devons utiliser `printf()` pour afficher sur la console nos valeurs, nous devrions tester le type réellement utilisé afin de choisir un formateur approprié, `%f` ou `%g` par exemple. Il se trouve que l'opérateur `<<` est surchargé pour les types primitifs de C++, mais pas `printf`.

Sans la classe `ostream`, nous aurions d'autres façons de résoudre ce problème ; remplacer `ostream`, ce qui peut se révéler fastidieux (et inutile). Nous pourrions également créer un modèle de fonction `afficher()`, externe à la classe, puis le spécialiser. Ou bien nous pourrions opérer une spécialisation partielle de la classe `Matrice`. Mais sur ce point, certains compilateurs se montrent plus coopératifs que d'autres.

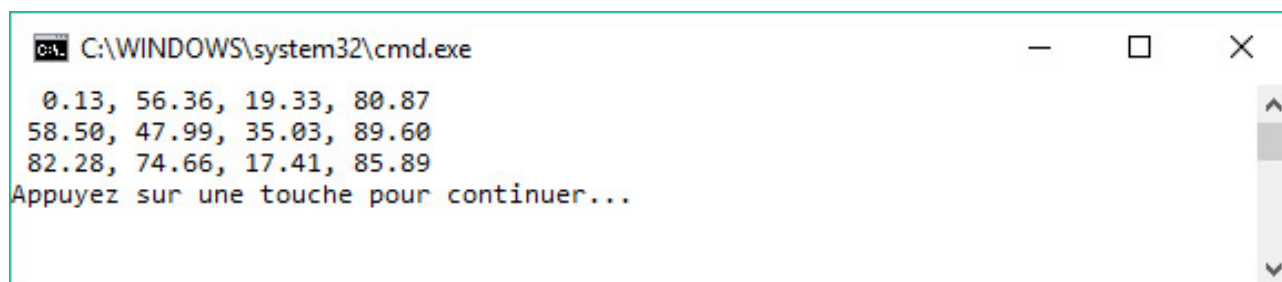
Si nous avons choisi une définition déportée de méthode, elle deviendrait un modèle de fonction. Il conviendrait alors de faire figurer les paramètres du modèle dans le nom de la classe :

```
template<class T> T& Matrice<T>::at(int vecteur,int valeur)
{
    return valeurs[vecteur*t_vecteur+valeur];
}
```

L'instanciation se passe presque comme pour une fonction, si ce n'est qu'il faut fournir explicitement le type associé au modèle.

```
Matrice<double> m(3,4);
m.random();
m.afficher();
```

Notre affichage produit les résultats suivants :



```
C:\WINDOWS\system32\cmd.exe
0.13, 56.36, 19.33, 80.87
58.50, 47.99, 35.03, 89.60
82.28, 74.66, 17.41, 85.89
Appuyez sur une touche pour continuer...
```

La syntaxe C++ admet des valeurs par défaut pour la définition de modèles :

```
template<class T=float> class Matrice
...
```

L'instanciation de la classe (et du modèle) peut alors se faire sans préciser de type, `float` étant la valeur par défaut :

```
Matrice<> mf(5,5); // matrice de float
```

Nous en arrivons au sujet délicat de la spécialisation partielle. Précisons d'abord que les compilateurs C++ supportent plus ou moins bien ce mécanisme. Dans certains cas, la compilation n'aboutit pas ou bien le code généré est erroné, ce qui peut se révéler particulièrement gênant.

Parmi les compilateurs les plus réguliers, citons le GNU C++, le compilateur Microsoft VC++ et la dernière version fournie par la société Intel. D'autres sont évidemment conformes à 100 % à la norme C++ mais les produits cités ont fait la preuve de leur rigueur dans la version indiquée. Attention, car cela n'a pas toujours été le cas, notamment pour le VC++ (avant 2003) et le compilateur d'Intel.

Nous allons commencer par créer un modèle de fonction pour la méthode `afficher()`, c'est-à-dire utiliser la définition déportée :

```
template<class T> void Matrice<T>::afficher()
{
    int vect,val;
    cout.precision(2);
    cout << fixed;
    for(vect=0; vect<nb_vecteur; vect++)
    {
        for(val=0; val<t_vecteur; val++)
        {
            cout.width(6);
            cout << at(vect,val) ;
            cout << (val<t_vecteur-1?" ":".");
        }
        cout << endl;
    }
}
```

Cette version constitue la version de secours, la plus générale. Il se peut qu'elle soit impossible à construire, mais dans notre cas, `cout` se comporte assez bien avec l'ensemble des types usuels.

Nous poursuivons par la fourniture d'une version spécifique à l'affichage des `float`. On note que le modèle de fonction a perdu son paramètre mais que le type de classe, `Matrice<float>`, est indiqué :

```

template<> void Matrice<float>::afficher()
{
    int vect,val;

    cout.precision(2);
    cout << fixed;
    printf("Affichage spécifique float\n");
    for(vect=0; vect<nb_vecteur; vect++)
    {
        for(val=0; val<t_vecteur; val++)
        {
            printf("%f",at(vect,val)) ;
            printf((val<t_vecteur-1?" ":". "));
        }
        printf("\n");
    }
}

```

Comme pour les modèles de fonctions spécialisés, le compilateur utilisera de préférence cette version pour une matrice de `float` et la version la plus générale dans les autres cas.