IUT Blagnac **Département Informatique 2ème année**

C++

Rédigé par Alain Caplain

caplain@univ-tlse2.fr

Version 0.95 du 27 janvier 2013

AVERTISSEMENT

Ce document n'a pas l'ambition d'être un cours de C++. C'est un document destiné aux étudiants en informatique de l'IUT de Blagnac qui résume les idées de bases qui ont été développées pendant les cours , TD et TP dispensés dans cet établissement.

Ce support est appelé à évoluer grâce à l'aide des enseignants, des étudiants ou de tout autre participant volontaire.

Table des matières

DU LANGAGE C AU LANGAGE C++

Le langage C++ hérite du langage C. Il en possède la syntaxe de base et les structures de contrôle. Il est cependant moins permissif que le langage C.

Les commentaires.

// commentaire jusqu'à la fin de la ligne.
/ bloc de commentaires*
*sur plusieurs lignes */*

Les entrées sorties.

cin >> *id_var* [>> *id_var*]*
cout << *expr* [<< *expr*]*

Ces fonctions sont déclarées dans le fichier iostream.h (ou iostream)
<< et >> sont des opérateurs surchargés. Ils s'adaptent automatiquement aux types des arguments.

Exemple : `cout<< "valeur de i : "<< i ;`

Les variables.

Déclaration.

Une déclaration de variable peut intervenir n'importe où dans le programme. Un objet est considéré comme une variable ayant sa classe pour type.

- Existence de la variable : de la déclaration à la fin du bloc où se trouve cette déclaration.

Utilité : déclaration d'entité nécessitant une initialisation provenant d'un calcul ou d'une saisie préalable. Initialisation de constantes. Instanciation d'objets par recopie.

!! Attention pour le for (int i= ...), la variable i n'existe que dans le bloc correspondant au « for ». => déclaration pour chaque boucle « for »..

La clause const. (Ne fait pas partie des spécificités du C++, sauf pour les fonctions)

Permet de déclarer des entités non modifiables.

Appliquée aux « variables »

Sur constante globale, limite la portée au fichier (? dans un fichier include). Remplace avantageusement les **#define** du langage C.

Syntaxe : **const** *type nom_var* = *valeur* **;**

- L'initialisation est obligatoire.

La déclaration d'une constante est prise en compte par le compilateur et non par le précompilateur. => emplacement mémoire, possibilité de conversions implicites.

Exemple : `const int MAX=30 ;`

Appliquée aux paramètres formels.

Permet de s'assurer que les paramètres formels ne sont pas modifiés dans la fonction (voir paramètres références).

Appliquée à des pointeurs :

```
const char * ptr  // La chaîne est constante, ptr est modifiable.
char * const ptr  // La chaîne est modifiable, ptr est constant.
const char * const ptr  // Les deux sont constants.
```

Les types.

Le type bool.

Type prédéfini composé des valeurs **true** et **false**.
Conversions implicites de et vers les entiers.

Transtypages.

Syntaxe : *type* **(***expression***)**
=> même syntaxe qu'une fonction. Ancienne syntaxe toujours valide.

Exemple :
```
int i ; float f ;
i=int(f) ;
f=float(i) ;
```

Dans le cas d'un type composé, l'ancienne syntaxe est obligatoire.

Cas particulier : les pointeurs.

Les conversions du type **void*** vers un pointeur typé ne sont pas implicites ; il faut obligatoirement un transtypage. Dans ce cas, l'ancienne syntaxe est obligatoire. Par contre, la conversion d'un pointeur typé vers un pointeur générique est implicite.

Exemple :
```
int * ptrInt ;
void * ptrGen ;
ptrInt = ( int * ) ptrGen ;
ptrGen = ptrInt ;   // OK implicite
```

Les fonctions de transtypage :

Bien que les conversions implicites fonctionnent pour beaucoup de types simples, il est conseillé d'utiliser les fonctions de transtypage.

static_cast : Permet des conversions plus ou moins implicites (void * → *type**, int→ float, float → int, ...). Cette fonction ne peut modifier l'état (constant ou volatile).

reinterpret_cast : Conversions de variables ou pointeurs sur données complètement différentes (transformations d'octets en structures).

const_cast : Modification de l'état const volatile.

dynamic_cast : Cette fonction permet, lorsque l'on utilise les possibilités du polymorphisme, de réaliser des liaisons dynamiques.

- Syntaxe : i*d_var* = **static_cast<** *nouveau_type* **>** (*expression*)

Dans la pratique, nous n'utiliserons que les fonctions "**static_cast**" et **dynamic_cast**".

Les références.

Appliquées à des variables.

Permet de donner un autre nom à une variable (synonyme) ou de nommer un emplacement mémoire qui n'a pas de nom propre (par ex. un élément d'un tableau).

- Une variable référence doit être impérativement initialisée..
- Syntaxe : *type* **&** *idReference* = *identificateur*;

La modification de l'un entraîne la modification de l'autre.

Exemples :

```
int i, tab[10] ;
int & r = i ;
int & q = tab[3] ;
```

Appliquées à des paramètres formels.

Utilisation en temps que paramètre formel d'une fonction (le vrai passage par référence).

- Syntaxe : *type* **&** *idParametre*

Il n'y a pas de recopie de valeur dans la pile d'exécution. Il y a donc modification de l'environnement de la fonction appelante sans utiliser de pointeurs. => les modifications effectuées sur le paramètre dans la fonction le sont dans la zone de données du paramètre réel de la fonction appelante. Evite une recopie des paramètres effectifs. Intéressant lorsque les données sont importantes en taille.

Exemple :

Déclaration : `void swap (int &, int &) ;`
Appel : `swap (a, b)`

✗ Passage de pointeurs par référence :

type *& p
p est une référence sur un pointeur remplace type **p en C

Possibilité d'empêcher la modification des paramètres :

- Syntaxe : **const** *type* **&** *idParamètre*

Fonction retournant une référence :

- Syntaxe : *type* **&** *nom_fonction* **(** **[** *parametres* **]** **) ;**

La fonction retourne une variable, plus précisément l'emplacement mémoire de la variable.

Une fonction retournant une référence permet un appel de la forme :

nom_fonction **(** … **)** = *expression* **;**

Les types structurés.

En C, définir un type structuré (struct, enum, union), n'est pas créer un nouvel identificateur de type, d'où l'obligation d'utiliser typedef ou de répéter à chaque utilisation le mot struct.
En C++, définir un type structuré (struct, enum, union), équivaut à créer un nouvel identificateur de type.
Possibilité de définition différée d'un type structuré.

- Syntaxe : **struct** *type* **;** //Déclaration d'un type défini ultérieurement (référence en avant)

Ce genre de déclaration permet d'utiliser des structures ou des classes avec des références croisées.

Appel à des fonctions externes sans avoir à inclure le fichier.

Exemple :

```
struct Date;
extern Date to_day();
```

Gestion dynamique de la mémoire.

Rappel : en C, les fonctions malloc, calloc, realloc ne sont pas typées, elles nécessitent un transtypage et une spécification de la taille de la mémoire allouée.

Exemple :

```
Date * Nouveau;
Nouveau = ( Date * )malloc( sizeof( Date ) );
```

Allocation.

En C++, la fonction **new** est une fonction typée (par surcharge)

- Syntaxe : *id_ptr* **= new** *type* **;** // pointeur simple
  id_ptr **= new** *type* **[** *nb* **] ;** // tableau dynamique

Exemples :

```
Date * Nouveau, * Tableau;
Nouveau = new Date;
Tableau = new Date [10];
```

Libération.

La fonction **delete** libère la mémoire mais ne remet pas le pointeur à NULL.

Exemples :

```
delete Nouveau;
delete [ ]Tableau;
```

Les fonctions (non membres).

Déclaration et définition.

En C++, la déclaration des fonctions est obligatoire avec le prototype.

- Syntaxe de déclaration :
  type idFonction **(** **[** *liste des types des paramètres* **]** **);**
  // si pas d'argument, parenthèses vides ou void
- Syntaxe de définition :
  type idFonction **(** **[** *liste des déclarations des paramètres* **]** **)**
  {
  corps de la fonction
  }; // Sauf fonctions membres d'une classe

Une fonction qui ne retourne aucune valeur doit obligatoirement être déclarée de type void

Valeurs par défaut des paramètres.

On peut affecter une valeur par défaut à un ou plusieurs paramètres, lors de la déclaration ou (exclusif) dans l'entête de la définition d'une fonction.
Les paramètres ayant une valeur par défaut doivent être les derniers de la liste.
Lors de l'appel, on ne peut omettre que les derniers paramètres (ne pas omettre le pf de rang i et spécifier le pf de rang i+1).

Exemples :

Déclaration :

```
void fct (int, char = 'X', float = 1.5) ;
```

Appels :

```
fct ( 2, 'A', 2.5 ) ;    // correct
fct ( 2, 'A' ) ;    // correct
fct ( 2 ) ;         // correct
fct ( ) ;      // illégal
fct ( 2, 2.5 ) ;         // illégal
```

Surcharge des noms de fonctions.

Le langage C++ permet de définir des fonctions différentes ayant le même identificateur.
Dans la pratique, s'applique à des fonctions exécutant des traitements similaires appliqués à des données de types différents.

- Les fonctions surchargées doivent être déclarées dans la même zone de déclaration.
- Les paramètres des fonctions de même nom doivent être différents en nombre ou en type.

Les fonctions peuvent éventuellement renvoyer des valeurs de types différents, mais le type de la valeur de retour n'est pas un critère de différenciation.
Le choix de la fonction par le système dépend des types des paramètres réels avec éventuellement conversion implicite de type.

Exemples :

Déclarations :

```
int Puissance (int, int) ;
float Puissance (float, int) ;
float Puissance (float, float) ;
```

Appels :

```
a = Puissance (5, 2)          //   appel   de   la   1ère
fonction
x = Puissance (1.5, 2)        //   appel   de   la   2ème
fonction
x = Puissance (1.5, 2.1)      //   appel   de   la   3ème
fonction
x = Puissance (1, 1.2)        // Erreur à la compilation
```

Exemple de surcharge non valide :

```
int fct (float, int) ;
float fct (float, int) ;
```

Surcharge et transtypages implicites.

Les transtypages implicites entre paramètres réels et paramètres formels sont limités :
bool → int, char → int, float → double, tableau → pointeur

Voir également les possibilités de transtypage avec l'utilisation des classes et de l'héritage.

Surcharge et paramètres formels avec valeurs par défaut.

Une fonction ayant un paramètre formel avec une valeur par défaut peut être appelée sans spécification du paramètre réel correspondant. Cette fonction est alors assimilable pour le système à une fonction ayant un paramètre en moins.

Par exemple, les fonctions « float fct (float, int = 0) ; » et « float fct (float) ; » ne peuvent être distinguées par le système. ces déclarations entraînent une erreur de compilation.

Attention, la surcharge de fonctions n'est pas le polymorphisme. Elle ne nécessite pas de relation d'héritage.

Les fonctions inline (expansion en ligne) .

Les fonctions inline sont conçues pour optimiser le temps d'exécution.

Rappel : Appel d'un sous programme = empilement des paramètres d'appel, rupture de séquence pour exécution du sous programme, rupture de séquence pour retour au programme appelant, dépilement des paramètres d'appel.

Solutions :

- Macro : Substitution de code source et des paramètres. Exécutée par le précompilateur. La taille du code source (et de l'exécutable) augmente. Pas de contrôle sémantique des paramètres ni de conversion implicite.

  Exemple :

```
#define Division_reelle (x, y) ( (x) / (y))
float f = Division_reelle (13, 5);          //      f vaut 2
```

- Expansion en ligne : Substitution de chaque appel par le corps de la fonction avec contrôle sémantique et conversions de types.

  Exemple :

```
inline float Division_reelle (float x, float y)
{ return x/y ;}
float f = Division_reelle (13, 5);          //      f vaut 2.6
```

Attention à la taille du code. Le compilateur peut ne pas exécuter la substitution si trop coûteux en taille.
Les fonctions inline ne peuvent pas être exportées (La fonction n'a pas d'adresse)

LES CLASSES

Les classes sont des structures composées de membres : Données (attributs) et méthodes (routines ou fonctions membres).

Déclaration d'une classe :

La déclaration d'une classe est placée dans un fichier d'extension « h », alors que les définitions de fonctions sont placées dans un fichier d'extension « cpp ».

- Syntaxe :

class *nom_classe*
{
déclaration des attributs et des méthodes
};

Les attributs sont déclarés comme les variables.
Certains attributs peuvent être des objets d'autres classes => relation de composition.
Les méthodes doivent être déclarées par leur prototype.

Accessibilité des membres.

Par défaut, les membres sont privés, c'est-à-dire, non accessibles par les classes clientes ou dérivées, ou par les fonctions clientes.

Pour rendre des membres accessibles, on utilise le mot réservé public :

class *idClasse*
{ [**private :**]
déclaration des attributs (privés)
déclaration des méthodes (privées)
public :
déclaration des attributs (publics)
déclaration des méthodes (publiques)
};

On peut utiliser le mot réservé **private** pour déclarer des membres privés ailleurs qu'en début de classe.
Les mots **private** et **public** peuvent être utilisés plusieurs fois dans la déclaration de la classe.
Il existe également un mode **protected** qui sera étudié lorsque sera abordé l'héritage.

- **Principe d'encapsulation :**

Certaines méthodes publiques permettent à l'utilisateur (classe ou fonction cliente, classe descendante) de consulter, éventuellement de modifier les attributs qui doivent être privés si l'on veut respecter les principes de la P.O.O.

Définition des méthodes.

Les méthodes sont généralement définies en dehors de la déclaration de la classe (sauf pour des fonctions inline), de préférence dans un autre fichier.

- Syntaxe :
  typeMéthode idClasse **::** *idMéthode* (**[** *paramètres* **]**) **[const]**
  {
  corps de la méthode
  }

« **::** » est l'opérateur de résolution de portée. Il indique que la fonction définie « est membre » de la classe nommée.

- **Les méthodes const :**

  Ce sont des méthodes membres d'une classe qui ne modifient pas les objets auxquels elles s'appliquent. Vérification par le compilateur.
  Les méthodes const sont les seules à pouvoir être appliquées à des objets const.

  Syntaxe :

  Déclaration : *enTete* **const** ;
  Définition : *enTete* **const {***corps***}**

  Une méthode « const » est généralement une implantation d'un observateur du TAA représenté par la classe.

Exemples de déclarations de classes :

```
class Point
{ private :
    float x, y ;
  public :
    ...
    float Abscisse() const;
    float Ordonnee() const;
} ;

class PileDeReels
{ private :
    float * Contenu ;
    int IndiceSommet ;
    int MaxElts ;
  public :
    .....
    void Empiler(float) ;
    void Depiler() ;
    bool PileVide() const ;
    float Sommet() const ;
} ;
```

Exemples de définitions de méthodes :

```
float Point::Abscisse() const
{ return x ; }
float Point::Ordonnee() const
{ return y ; }

void PileDeReels::Depiler()
{IndiceSommet-- ;}
bool PileDeReels::PileVide() const
{ return (Indice_Sommet==-1 ); }
float PileDeReels::Sommet() const
{ return Contenu[IndiceSommet] ;}
```

Fonctions de classes et fonctions « globales ».

Les fonctions de classes sont des méthodes devant être appliquées à des objets de la classe. Les fonctions « globales » sont utilisées comme les fonctions du C.

Attributs de classes et méthodes statiques.

Rappel : Un attribut statique est un attribut commun à tous les objets de la classe.

En C++, tout attribut de classe est déclaré avec le qualificatif "**static**".

Syntaxe :

static *type idAttribut* **;**

Un attribut de classe doit être initialisé en dehors de toute méthode de la classe. Cette initialisation est généralement faite au début du fichier de code (.cpp) correspondant à la classe.

C++ permet également de définir des méthodes statiques qui ne peuvent être appliquées à des objets et sont accessibles en utilisant la syntaxe :

idClasse **::** *idMéthode ...*

LES OBJETS

Déclaration.

Un objet est une instance d'une classe (c'est à dire une variable ayant la classe comme type).
La déclaration d'un objet implique son initialisation par appel à un constructeur.

- Syntaxe de déclaration (simplifiée) :

  *idClasse idObje*t **;**

Exemples :

```
Point Pt ;
PileDeReels PilPoil ;
```

Opérations applicables :

- Affectation (opérateur = synthétisé) et initialisation (→ constructeurs).

  Exemples :

```
Point Pix( 1.2, 2.5 );   // Initialisation ( appel à un constructeur)
Pt = Pix ;     // Affectation
```

- Application d'une fonction membre de la classe de l'objet.

  Syntaxe :

  *idObjet***.***idMembre* **[([** *arguments* **])]**

  La syntaxe est dépendante de la nature du membre : Pas de parenthèses pour un attribut, parenthèses (éventuellement vides) pour une fonction.

  Remarque : Le principe d'encapsulation préconise de déclarer les attributs privés. Si l'on respecte ce principe, l'accès à un attribut d'un objet n'a pas lieu d'être.

  Exemples :

```
cout << Pt.Abscisse( ) ;
PilPoil.Empiler( 5.5 ) ;
…
if( !PilPoil.PileVide( ) )
PilPoil.Dépiler( ) ;
```

La notion d'objet courant : Le pointeur this.

Lors de leurs définitions, les fonctions membres d'une classe font appel à des attributs, voire à des méthodes de cette même classe. Ces membres doivent être appliqués à un objet. Lequel ?

L'objet auquel va être appliquée la fonction lors de l'exécution. Cet objet n'existe que pendant l'exécution. On anticipe son existence pendant le développement : C'est l'**objet courant**.

Dans le corps des fonctions membres, l'objet courant n'est généralement pas nommé. Si l'on veut l'appeler explicitement, il faut utiliser le mot réservé this qui désigne le **pointeur** sur l'objet courant.

Création d'un objet : les CONSTRUCTEURS

La création d'un objet consiste à réserver un emplacement mémoire pour les attributs de l'objet et à initialiser ces attributs.

La création est réalisée par un **constructeur** qui est appliqué à l'objet lors de la déclaration, ou lors de l'allocation dans le cas d'un pointeur.

Constructeurs

- Un constructeur est une méthode de la classe qui a pour identificateur le nom de la classe. Le constructeur n'a pas de type de retour (Cf. procédure) mais peut posséder des arguments.

Il existe rois types de constructeurs :

- **Par défaut** : Constructeur sans paramètres.
- **Paramétré** : Permet d'initialiser les attributs de la classe à l'aide des paramètres.
- **Par recopie** : prend pour paramètre une référence (constante ou non) sur un objet de la classe et recopie champs par champs les données de l'objet paramètre dans l'objet courant.

Constructeurs synthétisés :

Toute classe possède deux constructeurs synthétisés (prédéfinis) :

- un constructeur par défaut sans paramètres (pas d'initialisation pour les attributs de type simple).
- un constructeur par recopie.

Redéfinition des constructeurs synthétisés.

Il est possible de redéfinir les constructeurs synthétisés et de définir d'autres constructeurs (paramétrés). La définition d'un constructeur paramétré rend le constructeur par défaut synthétisé inaccessible (masqué).

=> Il est fortement conseillé de toujours redéfinir le constructeur par défaut.

Exemples : Constructeurs de la classe Point :

Déclarations :

```
Point( ) ;              // Par défaut
Point(float, float) ;   // Paramétré

   // Ces deux constructeurs peuvent être remplacés par :
Point( float = 0.0, float = 0.0 ) ;
```

// Pour la classe Point, il n'est pas nécessaire de redéfinir le constructeur par recopie

Définitions :

```
Point::Point()
{ x = 0.0 ;
  y = 0.0 ;
}
```

```
Point::Point(float a, float b)
{ x = a ;
  y = b ;
}
```

Redéfinition du constructeur par recopie.

Le constructeur par recopie doit permettre de réaliser un clone de l'objet original.
Lorsqu'un des attributs de la classe est un pointeur, le constructeur par recopie synthétisé va recopier une adresse. L'objet cloné va donc pointer sur le même emplacement mémoire que l'objet source. Il y a copie superficielle.

Dans ce cas, il est nécessaire de définir un constructeur par recopie en profondeur.

Exemples : Constructeurs de la classe PileDeReels :

Déclarations :

```
PileDeReels (int = 10) ; //  A  la  fois  constructeur  par
défaut
                  // et constructeur paramètré
PileDeReels (const PileDeReels &);
                  //  Le  constructeur  par  recopie  doit
être redéfini.
```

Définitions :

```
PileDeReels :: PileDeReels  (int nb)
{ Max_Elts = nb ;
  Contenu = new float [nb];
  Indice_Sommet = -1 ;
}
PileDeReels  :: PileDeReels (const PileDeReels  & P)
{ Max_Elts = P.Max_Elts ;
  Contenu = new float [Max_Elts];
  Indice_Sommet = P. Indice_Sommet ;
  For( int i = 0 ; i < Indice_Sommet ; i++ )
    Contenu[i] = P.Contenu[i] ;
}
```

- Il est nécessaire de redéfinir le constructeur par recopie lorsque la classe contient un pointeur.

Exemples d'appels de constructeurs.

```
Point Pinot ;       // Par défaut, x=0, y=0
Point Potin(2.3) ;  // Paramètré
Point Piton(Potin) ;     // Par recopie
Point Topin = Piton;     // Par recopie
Point * Figure ;    // Pas d'appel de constructeur
Figure = new Point ;     // Par défaut, x=0, y=0
Figure = new Point(1.0, 2.8 ) // Paramètré
Figure = new Point( Pinot )   // Par recopie
PileDeReels  PilPoil ;   // Par défaut, nb=10
PileDeReels  Pilone(15) ;     // Paramètré
```

```
PileDeReels  Piloti(Pilone) ; // Par recopie
PileDeReels  PilouFace = Piloti ;  // Par recopie
Pile = PileDeReels  (PilouFace) ;  // Par recopie

// soit la fonction fct déclarée ainsi :

void fct(PileDeReels) ;
fct(PilPoil) ; //  Appel  au  constructeur  par  recopie,
passage par valeur
```

Constructeurs d'objets membres.

- Lors de la création par défaut ou paramétrée d'un objet appartenant à une classe composée d'autres objets, les constructeurs par défaut de ces autres objets sont appelés implicitement, avant le constructeur de la classe cliente, dans l'ordre des déclarations des objets.

Exemple :

```
class Segment
{ private :
    Point origine;
    Point extremite;
  public :
    Segment();
    Segment( const Point &, const Point & );
    ...
};

Segment::Segment()  //  appels  implicites  au  constructeur  par
défaut de la classe Point
{    ...}
Segment::Segment(const Point & pO, const Point & pE)
// appels implicites au constructeur par défaut de la classe Point
{    ...}
```

Appels explicites des constructeurs d'objets membres.

Si l'on veut appeler un constructeur particulier d'un objet membre, on doit le spécifier dans l'en-tête de la définition du constructeur de la classe composée.

Syntaxe de déclaration :

idClasseCliente ([liste des types des paramètres]) ;

Syntaxe de définition :

idClasseCliente::*idClasseCliente*([*paramètres*].) :*idObjetMembre* (*paramètres*)
{....}

Les paramètres d'appel du constructeur du membre sont soit des paramètres formels du constructeur client soit des constantes.

Exemples :

```
Segment::Segment() : origine(1, 2), extremite(3, 4)
{    }
Segment::Segment(const  Point  &  pO,  const  Point  &  pE)  :
origine(pO), extremite(pE)
{    }
```

Autre constructeur éventuel :

```
Segment(float, float, float, float);
Segment::Segment(float pAo, float pOo, float pAe, float pOe) :
origine(pAo, pOo),extremite(pAe, pOe)
{     ...}
```

Constructeur par recopie d'objets membres

Lors de la création par recopie d'un objet appartenant à une classe composées d'autres objets, les constructeurs par recopie de ces autres objets sont appelés implicitement si le constructeur par recopie de la classe composée est le constructeur par recopie synthétisé. Si le constructeur par recopie de la classe composée est redéfini, il appelle implicitement le constructeur par défaut de la classe composante.

Constructeur par recopie de la classe composée	Constructeur composant appelé implicitement	
	Par défaut	Par recopie
Synthétisé		×
Redéfini	×	

Exemple : Supposons l'existence d'une classe TableauDeReels possédant un constructeur prenant deux entiers comme paramètres (indMin et indMax). La classe PileDeReels pourrait être définie ainsi :

```
class TableauDeReels
{ private :
    float * contenu;
    int indMin, inMax;
  public :
    TableauDeReels( int, int );
    TableauDeReels( const TableauDeReels & );
    ...
} ;
class PileDeReels
{ private :
    TableauDeReels Contenu;   // relation de composition
    int IndiceSommet;
...
  public :
    ...
    PileDeReels(int);     // Constructeur
    ...
};
```

Définition du constructeur :

```
PileDeReels::PileDeReels (int nb) : Contenu (1, nb)
{...}
```

Dans ce cas, il n'est pas nécessaire de redéfinir le constructeur par recopie synthétisé de la classe PileDeReels. En effet, ce dernier appelle implicitement le constructeur par recopie de la classe TableauDe Reels pour le membre « Contenu ».

Destruction des objets.

La destruction des objets est réalisée par le destructeur de la classe qui est appelé automatiquement à la fin de la durée de vie de l'objet.

Le destructeur est une méthode qui a pour identificateur le nom de la classe précédé du signe ~ .

Toute classe possède un destructeur synthétisé, qui peut être redéfini en cas de besoin.

Le destructeur doit assurer la libération des ressources affectées à l'objet. Il est appelé implicitement à la fin de la vie de l'objet.

Exemple de destructeur pour la classe TableauDeReels :

```
TableauDeReels::~TableauDeReels ()
{ delete[ ]contenu; }
```

Vie des objets.

- Automatique : Le constructeur est appelé lors de la déclaration et le destructeur à la fin du bloc dans lequel se trouve cette déclaration.
- Dynamique : Constructeur appelé lors de l'allocation dynamique par la fonction **new** , le destructeur lors de la libération mémoire par la fonction delete.
- Objet membre : Les objets membres sont créés avant la création de l'objet contenant (client), dans l'ordre de leurs déclarations. Ils sont détruits après la destruction de l'objet contenant, dans l'ordre inverse de leurs déclarations.

L'HERITAGE

En C++, une classe (dérivée) peut hériter d'une ou de plusieurs autres classe (de base).
La classe dérivée possède tous les membres de la classe de base sauf les constructeurs et le destructeur.
Tout objet de la classe dérivée est un objet de la classe de base.

On peut substituer un objet d'une classe dérivée à un objet de la classe de base, affecter un pointeur sur la classe dérivée à un pointeur sur la classe de base.

Syntaxe.

class *idClasseDerivee* **:** **[** *acces* **]** *idClasseDeBase*
{
déclarations des membres de la classe
} ;

"acces" est la clause de limitation d'accès. elle peut prendre les valeurs **public**, **protected** ou **private** , cette dernière étant la valeur par défaut.

Exemples :

```
class Derivee : public deBase
    //   déclaration héritage simple
class Derivee : public deBase, private A
    //   déclaration héritage multiple
```

Accessibilité aux membres de la classe de base.

- Rappel.

  Les membres d'une classe sont déclarés dans différents paragraphes, chacun de ces paragraphes précisant le mode d'accès aux primitives le composant dans son en tête.

  Il existe trois modes d'accès : private, protected et public.

  private rend les membres inaccessibles en dehors de la classe. Seules les fonctions membres de la classe peuvent accéder à ces membres privés.

  protected rend les membres accessibles aux classes dérivées et inaccessible pour les classes et les fonctions clientes.

  public rend les membres accessibles à tous niveaux.

  Exemple :

```
class DeBase
{ private :
    int mbPriv ;
  protected :
    float mbProt( ) ;
  public :
    void mbPub( ) ;
} ;
```

```
float DeBase :: mbProt ( )
{
  return mbPriv/3.0;          // valide
}
void DeBase :: mbPub( )
{
  cout << mbPriv << mbProt( ) ;    // valide
}
void main()
{ DeBase monObjet ;
  cout << monObjet.mbPriv ;   // illégal
  cout << monObjet.mbProt ( ) ;    // illégal
  monObjet.mbPub( ) ;    // valide
}
```

Effets de la clause de limitation d'accès.

Toute classe dérivée a accès aux membres publics et protégés de sa (ses) classe(s) de base.

Une classe dérivée peut limiter les accès aux membres de sa (ses) classe(s) de base pour ses clients et ses descendants. C'est le rôle de la clause de limitation d'accès.

private (par défaut) : interdit l'accès pour les classes héritières et les classes clientes. Ne modifie que l'accès aux membres protégés ou publics de la classes de base.

protected : autorise l'accès pour les classes héritières, si les membres étaient déjà protégés ou publics au niveau de la classe de base ; interdit l'accès pour les classes clientes.

public : autorise l'accès à ses héritiers (si les membres étaient déjà protégés ou publics au niveau de la classe de base), autorise l'accès à ses clients (si les membres étaient déjà publics au niveau de la classe de base).

Limitation d'accès : private.

```
class deBase
{ private :
    int mbPriv ;
  protected :
    float mbProt() ;
  public :
    void mbPub() ;
} ;
class Derivee : private deBase
{
  void Test() ;
} ;
void Derivee :: Test()
{
  cout << mbPriv ;  // Illégal
  cout << mbProt() ;     // Correct
  mbPub() ;      // Correct
}
class Cliente
{
  Derivee Obj ;
  void Essai() ;
} ;
void Cliente :: Essai()
{
  cout << Obj.mbPriv ;        // Illégal
  cout << Obj.mbProt() ;      // Illégal
  Obj.mbPub() ;     // Illégal
}
class DD : public Derivee
{
  void Sc() ;
} ;
void DD : : Sc()
{
  cout << mbPriv ;  // Illégal
  cout << mbProt() ;     // Illégal
  mbPub() ;       // Illégal
}
```

Limitation d'accès : protected.

```
class deBase
{ private :
    int mbPriv ;
  protected :
    float mbProt() ;
  public :
    void mbPub() ;
} ;
class Derivee : protected deBase
{
  void Test() ;
} ;
void Derivee :: Test()
{
  cout << mbPriv ;  // Illégal
  cout << mbProt() ;     // Correct
  mbPub() ;       // Correct
}
class Cliente
{
  Derivee Obj ;
  void Essai() ;
} ;
void Cliente :: Essai()
{
  cout << Obj.mbPriv ;        // Illégal
  cout << Obj.mbProt() ;      // Illégal
  Obj.mbPub() ;     // Illégal
}
class DD : public Derivee
{
  void Sc() ;
} ;
void DD : : Sc()
{
  cout << mbPriv ;  // Illégal
  cout << mbProt() ;     // Correct
  mbPub() ;       // Correct
}
```

Limitation d'accès : public.

```
class deBase
{ private :
    int mbPriv ;
  protected :
    float mbProt() ;
  public :
    void mbPub() ;
} ;
class Derivee : public deBase
{
  void Test() ;
} ;
void Derivee :: Test()
{
  cout << mbPriv ;  // Illégal
  cout << mbProt() ;     // Correct
  mbPub() ;     // Correct
}
class Cliente
{
  Derivee Obj ;
  void Essai() ;
} ;
void Cliente :: Essai()
{
  cout << Obj.mbPriv ;        // Illégal
  cout << Obj.mbProt() ; // Illégal
  Obj.mbPub() ;     // Correct
}
class DD : public Derivee
{
  void Sc() ;
} ;
void DD : : Sc()
{
  cout << mbPriv ;  // Illégal
  cout << mbProt() ;     // Correct
  mbPub() ;       // Correct
}
```

Le mot clé using peut rétablir un accès supprimé par l'héritage (fortement déconseillé !).
public : using deBase::membre

Création et destruction d'objets.

Les constructeurs et les destructeurs des classes ascendantes ne sont pas transmis par héritage et ne peuvent donc pas être appelés explicitement dans une méthode d'une classe dérivée. Toute classe dérivée possède deux constructeurs et un destructeur synthétisés, pouvant tous être redéfinis.

Lors de la création sans recopie d'objets ayant pour type une classe dérivée, les constructeurs par défaut des classes ascendantes sont appelés implicitement, dans l'ordre de l'arborescence de l'héritage, avant ceux des objets membres.

Si l'on veut appeler un constructeur autre que celui par défaut, il faut spécifier ce constructeur et ses paramètres dans l'en tête du constructeur de la classe héritière.

Exemple : Soit une classe "DeBase" ayant 2 attributs entiers et une classe "Derivee" hérutant de la classe DeBase.

```
Derivee::Derivee(int x) : deBase ( x, 2)
{ ... }
```

Le constructeur par recopie synthétisé d'une classe héritière appelle implicitement le constructeur par recopie de la classe ascendante.

Le constructeur par recopie redéfini d'une classe héritière appelle implicitement le constructeur par défaut de la classe ascendante.

Le destructeur d'une classe dérivée appelle implicitement les destructeurs des classes ascendantes, dans l'ordre inverse de l'arborescence d'héritage.

Exemple : La classe PileDeReels héritant de la classe TableauDeReels.

```
class PileDeReels : TableauDeReels
{ private :
    int Indice_Sommet ;
    int Max_Elts ;
  public :
    PileDeReels ( int = 10 ) ;
    PileDeReels ( const PileDeReels &) ;
    void Empiler(float) ;
    void Depiler() ;
    bool PileVide() ;
    float Sommet() ;
} ;
PileDeReels::PileDeReels ( int N ) :TableauDeReels (1, N )
{
  Max_Elts = N ;
  Indice_Sommet = 0 ;
}
PileDeReels::PileDeReels(const PileDeReels &P):TableauDeReels (P)
{ // ce constructeur ne fait rien de plus que le constructeur
  // par recopie synthéthisé.
  Max_Elts = P.Max_Elts ;
  Indice_Sommet = P.Indice_Sommet ;
}
```

Redéfinition de membres.

Lorsque, dans une classe dérivée, on définit une fonction ayant le même identificateur qu'une fonction d'une classe de base, il y a redéfinition et non surcharge.

Rappel : Pour qu'il y ait surcharge, il faut que les fonctions soient déclarées dans la même zone déclarative (même bloc).

Le membre de la classe héritière masque celui de la classe de base (tous ceux de même nom s'il y a surcharge dans la classe de base).

Il est possible d'accéder à un membre masqué en utilisant l'opérateur de résolution de portée.

Exemple :

```
class ListeDeFiches
{ ...
  public :
    Fiche Consulter( ) const;
    // Renvoie la 1ère fiche de la Liste
    Fiche Consulter( int ) const;
    // Renvoie la fiche dont le rang est passé en paramètre
...
} ;
class ListeOrdDeFiches : ListeDeFiches
{ ...
  public :
     Fiche Consulter (const string & ) const;
    // Renvoie la Fiche dont le numéro est passé en paramètre
    // Cette fonction masque les fonctions Consulter de la
    // classe ListeDeFiches, bien que ces dernières soient
    // accessibles.
    ...
    // Fiche Consulter ( ) ;  a définir
    // Renvoie la 1ère fiche de la Liste Ordonnée
} ;
void main()
{
  ListeOrdDeFiches L ;
  ...
  cout << L.Consulter(« a12 »)
  // appel de la fonction de la classe ListeOrdDeFiches
  cout << L.Consulter()  // illégal
  cout << L.Consulter(3) // illégal
  cout << L.ListeDeFiches ::Consulter(3)     // correct
...
}
```

SURCHARGE D'OPÉRATEURS

La surcharge d'opérateurs permet d'adapter des opérateurs prédéfinis à des objets pour lesquels ils n'ont pas été prévus.

Tous les opérateurs sont surchargeables sauf : « . » « .* » « :: » « ? : » « sizeof » ainsi que les symboles « # » et « ## ».

Il n'est pas possible de créer de nouveaux opérateurs.

La surcharge d'opérateurs conserve les cardinalités, les priorités et l'associativité des opérateurs prédéfinis, mais pas la commutativité et les liens sémantiques avec des opérateurs « composés ». Par exemple, la surcharge de l'opérateur + est indépendante de celle des opérateurs ++ et de +=.

L'affectation (« = ») est un opérateur synthétisé qui peut être redéfini.

Surcharger un opérateur $ consiste à définir une fonction operator $.

Un opérateur peut être surchargé par une fonction membre d'une classe ou par une fonction externe. Les opérateurs « = », « [ ] », « ( ) », « ?? » doivent être surchargés par une fonction membre.

Pour illustrer cette présentation, nous définirons les opérateurs de la classe « Complexe » basée sur la déclaration suivante :

```
class Complexe
{ private :
    float Re, Im ;
  public :
    Complexe() ;
    Complexe (float, float) ;
    float getRe() const ;
    float getIm() const ;
    void setRe(float);
    void setIm(float);
…
} ;
```

Surcharge d'un opérateur par une fonction membre.

La cardinalité de la fonction est égale à la cardinalité de l'opérateur moins 1, le premier opérande étant l'objet courant (pointé par this).

Un appel de la forme x $ y équivaut à l'appel x.operator $ (y).

Il y a possibilité de modification de l'objet courant.

L'opérateur de cumul : +=

Si l'on considère cet opérateur comme un transformateur de profil Complexe ? Complexe ? Complexe, il doit être implanté sous la forme d'une fonction de type « void » prenant un complexe comme paramètre.

Déclaration :

```
void operator += (const Complexe &) ;
```

Définition :

```
void Complexe :: operator += (const Complexe & nbr)
{ Re = Re + nbr.Re ;    // ou Re += nbr.Re ;
  Im = Im + nbr.Im ;    // ou Im += nbr.Im ;
}
```

En réalité, si l'on veut conserver une syntaxe compatible avec celle du langage C, on doit implanter cet opérateur comme une fonction qui retourne un Complexe :

Déclaration :

Complexe operator += (const Complexe &) ;

Définition :

```
Complexe Complexe :: operator += (const Complexe & nbr)
{ Re = Re + nbr.Re ;    // ou Re += nbr.Re ;
  Im = Im + nbr.Im ;    // ou Im += nbr.Im ;
  return *this ;
}
```

L'opérateur +

Bien que définie comme une fonction membre, cette loi de composition interne ne doit pas modifier pas l'objet courant. De plus, elle doit retourner un complexe pour respecter la syntaxe a = b + c ;

Déclaration :

```
Complexe operator+ (const Complexe &) const ;
```

Définition :

```
Complexe Complexe::operator+(const Complexe & nbr)
{ Complexe temp = *this ;   // clonage de l'objet courant
  temp.Re += nbr.Re ;
  temp.Im += nbr.Im ;
  return temp ;
}
```

L'opérateur ++

L'opérateur ++ est un transformateur qui doit retourner une valeur. Il existe deux versions de cet opérateur d'incrémentation : La pré incrémentation et la post incrémentation.

Rappel : La pré incrémentation modifie l'objet courant avant de retourner sa valeur, la post incrémentation retourne la valeur de l'objet courant avant modification.

Opérateur de pré incrémentation :

Déclaration :

```
Complexe operator++() ;
```

Définition :

```
Complexe Complexe::operator ++()
{    Re++;
     return *this;
}
```

Opérateur de post incrémentation :

Déclaration :

```
Complexe operator++(int) ;
```

Dans ce cas, le paramètre int est un paramètre « virtuel » dont la seule utilité est de permettre au système de distinguer les deux formes de l'opérateur.

Définition :

```
Complexe Complexe::operator ++(int)
{ Complexe temp = *this;
  Re++;
  return temp;
}
```

L'opérateur *

Cet opérateur peut symboliser le produit de deux complexes, mais également le produit d'un complexe par un réel ou le produit d'un réel par un complexe, …

Le produit de deux complexes est très semblable dans sa conception à la somme de deux complexes. Nous laisserons au lecteur le soin de déclarer et de définir cet opérateur.

Nous définirons l'opérateur * produit d'un complexe par un réel comme une fonction ne modifiant pas l'objet courant.

Opérateur de produit d'un complexe par un réel :

Déclaration :

```
Complexe operator *(float) ;
```

Définition :

```
Complexe Complexe::operator *(float nb) const
{     complexe temp = *this;
      temp.Re *= nb;
      temp.Im *= nb;
      return temp;
}
```

D'autres opérateurs sont fréquemment surchargés : Les opérateurs de comparaison.

Les fonctions membres ne permettent de définir que les opérateurs dont le premier opérande est l'objet courant. Tous les autres opérateurs doivent être définis par des fonctions externes.

Surcharge d'un opérateur par une fonction externe (globale).

La cardinalité de la fonction est égale à la cardinalité de l'opérateur.

Un appel de la forme x @ y équivaut à l'appel operator@ (x, y).

Utilité : Lorsque le premier opérande n'est pas un objet de la classe. (conservation de la commutativité avec un opérande d'un type prédéfini)

Problème : Une telle fonction ne peut accéder aux membres non publics de la classe, il peut être utile de la déclarer en tant que fonction amie "friend".

L'opérateur *, produit d'un réel par un complexe.

Déclaration :

```
Complexe operator * (const float&, const Complexe&);
```

Définition :

```
Complexe operator * (const float &f, const Complexe &C)
{ Complexe Temp;
  Temp.setRe(f*C.getRe());
  Temp.setIm(f*C.getIm());
  return Temp;
}
```

Surcharge des opérateurs d'entrée sortie.

Les flots d'entrée sortie.

Le C++ gère des « objets flots » qui doivent être assignés à des fichiers. Parmi ceux-ci, **cout**, objet de la classe "**ostream**" est assigné à la sortie standard et **cin**, objet de la classe "**istream**" est assigné à l'entrée standard.

L'opérateur << envoie des informations d'un type prédéfini vers un objet de la classe ostream, l'opérateur >> envoie des informations d'un objet de la classe istream vers une référence sur un type prédéfini.

Ces deux opérateurs peuvent être surchargés pour gérer des informations de types autres que les types prédéfinis.

L'opérateur de sortie standard <<

Déclaration :

```
ostream& operator << (ostream &, const Complexe&);
```

Définition :

```
ostream & operator << (ostream & Out, const Complexe &Cpl)
{
  if(Cpl.getRe()!=0 ||Cpl.getIm()==0)
    Out<< Cpl.getRe();
  if(Cpl.getIm()>0 && Cpl.getRe()!=0)
    Out<<" + ";
  if(Cpl.getIm()==-1)
    Out<<" - ";
  if(Cpl.getIm()!=1 && Cpl.getIm()!=-1 && Cpl.getIm()!=0)
    Out<< Cpl.getIm();
  if(Cpl.getIm()!=0)
    Out<<" i ";
  return Out;
}
```

L'opérateur d'entrée standard >>

Déclaration :

```
friend istream& operator >> (istream &, Complexe&);
```

Définition :

```
istream & operator >> (istream & In, Complexe &Cpl)
{ char ch,car;
  In >> Cpl.Re;
  In.get(ch);
  if (ch=='i' || ch=='I')
  { Cpl.Im= Cpl.Re;
    Cpl.Re=0;  }
  else if (ch=='+' || ch=='-')
        { In.get(car);
          if (car=='i' || car=='I')
            Cpl.Im=float((ch=='+'?1:-1));
          else
          { In.putback(car);
            In >> Cpl.Im;
            Cpl.Im=(ch=='+'? Cpl.Im:-1* Cpl.Im);
            In.get(ch);
          }
        }
        else
          Cpl.Im=0;
  return In;
}
```

L'opérateur =.

Opérateur d'affectation de recopie.

Pour toute classe, l'opérateur = (qui n'est pas hérité) est synthétisé sous la forme d'un opérateur d'affectation de recopie.

Il recopie champ par champ les valeurs de l'objet passé en paramètre dans l'objet cible (l'objet courant), les deux objets appartenant à la même classe.

Cette recopie étant superficielle, il est nécessaire de redéfinir l'opérateur = lorsque la classe contient des attributs « dynamiques » (id. constructeur par recopie). Dans ce cas, l'opérateur = doit être redéfini par une fonction membre.

L'opérateur = synthétisé appelle implicitement les opérateurs = des classes composantes si nécessaire.

En cas d'héritage, l'opérateur= synthétisé appelle implicitement l'opérateur = de recopie de la classe ascendante.

Exemple : la classe PileDeReels composée d'un pointeur de float.

```
PileDeReels & PileDeReels::operator=(const PileDeReels & P)
{ if (this != &P)
  { Max_Elts = P.Max_Elts ;
    Indice_Sommet = P.Indice_Sommet ;
    if (Tableau != NULL) // dans ce cas inutile
    { delete [] Tableau;
      Tableau = new float [Max_Elts];
      for( int i = 0 ; i < Indice_Sommet ; i++ )
        Tableau[i] = P.Tableau[i] ;
    }
  }
  Return *this;
}
```

L'opérateur = retourne une référence sur l'objet courant de façon à conserver la syntaxe du langage C : Obj1 = Obj2 = Obj3 …

Opérateur d'affectation de conversion

L'opérateur = peut également être surchargé pour définir une conversion implicite personnalisée.

Exemple :

```
Complexe & Complexe::operator=(const float) ;
```

L'opérateur = de conversion ne masque pas l'opérateur = de recopie synthétisé. Il s'agit d'une surcharge car même espace de définition.

L'opérateur [ ].

Opérateur d'indexation. Doit être défini comme une fonction membre.

Uniquement unaire, la fonction operator[ ] peut renvoyer un objet qui est lui même muni de l'opérateur [ ] ce qui donne la possibilité de traiter des tableaux à plusieurs dimensions.

Le type de l'argument (de l'indice) peut être n'importe quel type, y compris une classe.

Les opérateurs de gestion dynamique de la mémoire.

Les opérateurs **new** et **delete** peuvent être surchargés au niveau d'une classe. Ils doivent alors être définis comme des fonctions membres.

La surcharge de ces opérateurs peut être utile pour effectuer des initialisations ou des allocations en chaîne dans le cas de classes contenant des pointeurs.

LES ESPACES DE NOM

Portée et visibilité.

Blocs.

Un bloc est une portion de code comprise entre "{" et "}". Un bloc constitue une zone de déclaration. Un bloc peut contenir d'autres blocs. Tout élément déclaré dans un bloc à une portée qui s'étend de sa déclaration jusqu'à la fin du bloc dans lequel il est déclaré.

L'espace global.

L'espace global d'une application est constitué de l'ensemble des fichiers de cette application. Tout élément déclaré en dehors d'un bloc à une portée globale, c'est à dire qu'il existe partout dans l'application.

Visibilité.

La visibilité d'un élément est égal à sa portée sauf dans les portions de code où il est masqué. Un élément est masqué dans toute portion de code où un élément de même nom existe.

Opérateur de résolution de portée.

Tout élément masqué est accessible à l'aide de l'opérateur de résolution de portée si sa zone de déclaration est nommée (classe, structure ou espace de nom).

Syntaxe :

nom_de_le_zone **::** *élément...*

Espaces de noms (namespace).

Déclaration d'un espace de nom.

Syntaxe :

```
namespace nom
{
    // toute déclaration dans ce bloc n'est visible que dans ce bloc
}
```

Utilisation d'un espace de nom.

Syntaxe :

using namespace *nom* **;**

Permet d'accéder à tous les éléments déclarés dans l'espace nommé.

L'espace de nom « std ».

C'est l'espace où sont déclarées les classes de la « STL ».

LES FONCTIONS ET LES CLASSES AMIES

Fonctions amies.

Une fonction amie d'une classe est une fonction qui peut accéder aux membres privés ou protégés de la classe sans être elle même membre de cette classe.

Une telle fonction doit être déclarée comme étant « friend » dans la classe, indifféremment dans un bloc privé, protégé ou public.

Déclaration :

friend *type identificateur* **(** *liste des types des paramètres formels* **) ;**

Définition de la fonction en dehors de la classe :

type identificateur **(** *liste des paramètres formels* **)**
{
corps de la fonction
}

Cette définition est généralement située dans un fichier contenant des fonctions globales.

A utiliser avec précaution car affaiblit l'architecture des classes basée sur l'encapsulation et les droits d'accès aux membres des différentes classes.

Principale utilisation : Surcharge d'opérateurs et plus particulièrement des opérateurs d'entrée sortie << et >>.

Classes amies.

Une classe amie d'une autre classe peut accéder à tous les membres de la classe dont elle est amie. Tous les membres de cette classe peuvent accéder à tous les membres de la classe d'origine.

Une classe amie doit être déclarée comme telle dans la classe d'origine.

Syntaxe : **friend class** *identificateur* **;**

Exemple d'utilisation : La classe Liste simplement chaînée amie de la classe cellule chaînée.

Avantages et inconvénients.

Allège le code.

Affaiblit l'architecture

LA GÉNÉRICITÉ

Utilité de la généricité.

Définir des fonctions qui traitent d'une façon semblable des données de différents types et qui s'adaptent automatiquement à ces types différents.

Exemple : La fonction Echange(int, int)

Décrire des classes conteneurs génériques comme celle étudiées à l'aide des types abstraits : Piles, Files, Listes, Arbres, …

Le C++ permet de définir des fonctions génériques et des classes génériques également appelées classes modèles ou classes patrons.

Fonctions génériques.

Syntaxe.

template < class *id_pgf* **[, class** *id_pgf* **]* >**

Cette déclaration est suivie de l'en tête et du corps de la fonction utilisant le paramètre générique formel *id_pgf* , en tant que type de paramètre formel, de variable locale ou type de retour.

Portée de la clause template.

La clause template **< class** *id_pgf* **>** identifie le paramètre générique formel *id_pgf* pour tout le bloc de la fonction et uniquement pour ce bloc.

Exemples.

```
template <class T>
T Maximum ( const T a, const T b )
{ if (a > b)
    return a ;
  else
    return b ;
}
template<class T>
void Echange ( T& x, T& y )
{ T z ;
  z=x ;
  x=y ;
  y=z ;
}
```

Instanciation du paramètre générique formel.

L'instanciation est gérée automatiquement à l'appel de la fonction.

Exemples :

```
int N1, N2, N3 ;
float R1, R2, R3 ;
char C1, C2, C3 ;
//   …..
N3 = Maximum ( N1, N2 ) ;
R3 = Maximum ( R1, R2 ) ;
C3 = Maximum ( C1, C2 ) ;
```

Spécialisation.

La spécialisation est parfois nécessaire lorsque certains types ne sont pas compatibles avec les opérateurs de la fonction générique.

On définit alors une fonction spéciale pour chacun de ces types.

Chacune de ces fonctions a le même nom que la fonction générique : on surcharge la fonction générique.

Exemple :

```
char * Maximum (const char * a, const char * b)
{ if (strcmp(a, b)>0)
    return a ;
  else
    return b ;
}
```

Classes génériques.

Appelées également classes modèles ou classes patrons, ce sont des classes paramétrées.
Elles contiennent des données dont le type réel n'est précisé que lors de l'instanciation d'un objet.

Déclaration.

Syntaxe :

template < class *id_pgf* **[, class** *id_pgf* **]* >**
Déclaration de la classe utilisant les paramètres génériques formels

La clause template<class T> identifie le paramètre générique formel pour tout le bloc de déclaration de la classe.

Exemple :

```
template <class T>
class Pile
{
     …
} ;
```

Instanciation : Les paramètres génériques réels.

L'instanciation est explicite lors de la déclaration des objets de la classe.

Syntaxe :

id_classe < *pgr* [, *pgr*]* > *id_objet* ;

Exemple :

```
Pile < float > P ;
```

- Si un paramètre générique réel est une classe, il doit posséder un constructeur par défaut sous peine d'erreur à l'exécution.

Un pgr peut être une instanciation d'une classe générique.

Exemple :

```
Liste < File <float> > L ;
```

Définition des fonctions membres d'une classe générique.

Les fonctions membres d'une classe générique sont des fonctions génériques, soit parce qu'elles utilisent un pgf, soit simplement parce qu'elles sont membres de la classe, c'est à dire appliquées à des objets de la classe.

Obligation d'utiliser la clause template lors de la définition des fonctions

Exemples :

```
template < class T >
bool Pile < T > : : Pile_Vide( )
{
        return  … ;
}
template < class T >
T Pile < T > : : Sommet( )
{
        return  … ;
}
```

La clause template n'est valide que pour le bloc de définition de la fonction, d'où obligation de répéter cette clause pour chaque fonction.

Spécialisation.

Comme pour les fonctions génériques, il est parfois nécessaire de spécialiser des classes génériques pour des problèmes de compatibilité entre types et opérateurs.

Exemples :

```
template < class A, class B >
class C
{    } ;
Cette classe peut être spécialisée en :
template < class A, class B >
class C < A , B* >
{    } ;
```

ou :

```
template < class A >
class C < A , int >
{    } ;
```

C<float, float> Obj ; instancie la classe patron

C<float, char *> Obj ; instancie la 1ère spécialisation

C<float, int> Obj ; instancie la 2ème spécialisation

POLYMORPHISME ET LIAISONS DYNAMIQUES

Rappels : Fonctions, surcharge et masquage.

Zone déclarative.

Une fonction est déclarée et définie dans une zone de déclaration. Exemples : Une classe pour une fonction membre, un fichier pour une fonction globale, un espace de noms défini par le développeur.
Remarque : une fonction peut être accessible hors de sa zone déclarative. Exemple : Fonction (protégée ou publique) appelée dans une classe dérivée.

Surcharge et masquage.

Il y a surcharge (de noms) de fonctions dans la cas ou plusieurs fonctions de même nom sont déclarées et définies dans la même zone déclarative. Exemples : Les différents constructeurs d'une classe.
Dans le cas d'un héritage, une fonction protégée ou publique est accessible par les membres (éventuellement les clients) de la classe héritière. Si, dans cette classe héritière on définit une fonction de même nom que la fonction héritée, il y masquage de la fonction héritée et de toutes ses surcharges par la fonction membre de la classe héritière. Les fonctions masquées sont cependant accessibles via l'opérateur de résolution de portée.

Exemple :

```
class Base
{ public :
    int fct() ;
    int fct(int) ;
} ;
class Herit : public Base
{ public :
    int fct() ;
} ;
.....
{ Herit H ;
  .....
  cout<<H.fct() ;   // C'est la fonction membre de la
  ......  // classe Herit qui est appelée
  cout<<H.fct(5) ;  // Illégal, la fonction fct(int),
  ........// membre de la classe Base est
  ........// masquée.
  cout<<H.Base::fct(5);// Ok. Utilisation de l'opérateur
  .........    // de résolution de portée.
.....
}
```

Héritage : Principe fondamental.

Lorsque l'héritage est public, tout objet de la classe dérivée « est » un objet de la classe de base, c'est à dire qu'il offre les mêmes services (plus éventuellement d'autres).
Un objet de la classe dérivée peut donc prendre la place d'un objet de la classe de base (Mais pas l'inverse ! ! !).

Exemple :

```
class Base
{    public :
     int fct() ;
     int fct(int) ;
} ;
class Herit : public Base
{    public :
     int fct() ;
} ;
.....
{    Base B, *PtrB  ;
     Herit H, *PtrH ;
     ......
     Base &RefB = H ;    // Ok
     Herit &RefH = B;    // Illégal
     PtrB = &H ;    // Ok
     PtrH = &B ;    // Illégal
     B = H ;   // Ok par appel de l'opérateur = de recopie
     H = B ;   // Illégal
}
```

On dit que *PtrB et RefB sont de type statique Base et de type dynamique Herit.

Liaisons statiques et liaisons dynamiques.

On peut alors se demander quel est le résultat d'appels de la forme :

```
RefB.fct() ;
PtrB->fct() ;
```

La fonction appelée est-elle celle du type statique ou celle du type dynamique ?
Dans le cas présent, la liaison est statique, c'est-à-dire que la fonction appelée est celle du type statique.
Il est possible d'établir des liaisons dynamiques. Il faut alors que les fonctions soient déclarées « virtuelles ».

Fonctions virtuelles.

Une fonction membre d'une classe est virtuelle si le mot virtual apparaît en tête de sa déclaration. Alors, dans le cas d'une redéfinition dans une classe dérivée, les liaisons concernant cette fonction seront dynamiques.

Syntaxe de déclaration :

```
virtual type idFonction ( [ paramètres ] ) [ const ];
```

Les fonctions ainsi définies sont appelées fonctions polymorphes.
Les fonctions polymorphes doivent avoir des paramètres de mêmes types, et des types de retour accessibles.
Les règles de masquage s'appliquent aux fonctions virtuelles.
La déclaration d'une fonction virtuelle entraîne la virtualité de toute fonction descendante.
On ne peut déclarer des constructeurs virtuels. Par contre, lorsqu'on utilise le polymorphisme, les destructeurs doivent être virtuels.
Toute classe possédant au moins une fonction virtuelle est appelée classe polymorphe.

Fonctions virtuelles pures, classes abstraites.

Une fonction qui n'est pas définie pour la classe de base est dite virtuelle pure. Elle est alors initialisée à 0 lors de la déclaration, quel que soit son type.

Syntaxe de déclaration :

```
virtual type idFonction ( [ paramètres ] ) [ const ]  = 0  ;
```

Une classe qui contient au moins une fonction virtuelle pure est une classe abstraite. Elle ne peut être instanciée ni être un type de retour de fonction mais peut être un type de pointeur.
Application : Déclarer une classe correspondant à un type abstrait, sans se soucier de l'implantation. Les classes dérivées définissent l'implantation des structures de données correspondantes.

Exemple : Les classes Pile Abstraite, Pile Statique, Pile Dynamique.

Héritage multiple

Pb : Résolution des ambiguïtés de noms.

Quand une classe dérive de plusieurs classes dont certaines contiennent des fonctions de même nom, la classe héritière ne peut appeler ces fonctions qu'en utilisant l'opérateur de résolution de portée.
Si, dans une des classes de base la fonction ambiguë est surchargée, il y aura masquage de toutes les formes de cette fonction dans la classe héritière.
Les constructeurs sont appelés dans l'ordre de la déclaration d'héritage.

Exemple :

```
class BaseUn
{    public :
int fct() ;
int fct(int) ;
} ;
class BaseDeux
{    public :
int fct() ;
} ;
class Herit : public BaseUn, public BaseDeux
{
......
} ;
......
{    Herit H ;
......
     H.fct() ; // Non, ambigu
     H.BaseUn::fct() ;   // Ok
     H.BaseDeux::fct() ; // Ok
     H.fct(5) ;     // Non, ambigu
......
}
```

HÉRITAGE RÉPÉTÉ.

Problème : Ne pas avoir plusieurs occurrences de la classe de Base.

```
class Super
{
     ......
} ;
class BaseUn : public virtual Super
{    public :
int fct() ;
int fct(int) ;
} ;
class BaseDeux : public virtual Super
{    public :
int fct() ;
} ;
class Herit : public BaseUn, public BaseDeux
{
     ......
} ;
......
```

Attention, dans ce cas, le constructeur paramétré de la classe Herit doit appeler explicitement le constructeur paramétré de la classe Super de façon à lui transmettre les paramètres. En effet, les constructeurs sont appelés dans l'ordre Super, BaseUn, BaseDeux. De même, le constructeur par recopie de la classe Herit doit appeler explicitement le constructeur par recopie de la classe Super.

```
Herit :: Herit ( … ): Super(...), BaseUn(...), BaseDeux(...)
{
     ......
}
```