ISIMA 2

Introduction au langage C++

1 Introduction

Le langage C++ est le résultat du travail de Bjarne Stroustrup aux laboratoires Bell [STROUSTRUP 86] [STROUSTRUP 92]. A l'origine, ce langage fut conçu pour traiter des problèmes de simulation. Le lecteur averti sait en effet que ce langage s'inspire principalement de SIMULA 67 [DAHL 68] pour les notions de classes d'objets, d'héritage et de polymorphisme. Le langage C++ englobe le langage C à peu d'exceptions près, en effet le langage C a été conservé pour sa polyvalence, sa concision, sa capacité a réaliser efficacement des tâches systèmes de bas niveau (ce pourquoi il avait été conçu par Dennis Ritchie). Un autre atout important reste le fait que le langage C, qui sert de base à la majorité des compilateurs C++, s'exécute sur tout matériel ce qui assure une portabilité très honorable à défaut d'être parfaite. Enfin la masse actuelle de développeurs en langage C est susceptible de transiter facilement de la programmation structurée à la programmation orientée objet en récupérant les millions de lignes de code C existantes sous formes d'applications, d'utilitaires ou de bibliothèques systèmes. Il semblait donc intéressant d'essayer de compenser les défauts du langage C tout en conservant les avantages acquis.

La première ébauche de C++ fut nommée "C avec classes" et ce n'est qu'après de nombreuses évolutions du langage que Rick Mascitti a suggéré le nom subtil de C++. Les différentes interprétations de ce nom sont données dans [STROUSTRUP 86], [KOENIG 88a]. L'objectif de base de ce langage est de permettre d'exprimer dans un langage proche du problème à résoudre les concepts d'une solution.

La présentation du langage C++ que nous allons faire est une introduction qui ne saurait se substituer à la lecture d'ouvrages tels que [ELLIS 90] [STROUSTRUP 92] [LIPPMAN 92] [COPLIEN 92]. Les langages à objets entraînent une modification de la manière dont un concepteur doit raisonner. Il faut accepter de changer un certain nombre d'habitudes acquises en programmation structurée au cours de nombreuses années d'expérience. Stroustrup précise notamment que plus on connaît C, plus il semble long et difficile de changer son style de programmation afin de tirer avantage des potentialités de C++. Cependant, moyennant un effort raisonnable les développeurs C conviennent rapidement que C++ est un outil d'une grande puissance pour aborder la conception de problèmes complexes. Par la suite, nous nous attacherons à présenter le langage C++ autour d'exemples tirés de la conception d'un éditeur graphiques d'objets de production.

2 La modélisation d'un problème à l'aide de classes

2.1 Abstraction et encapsulation

Un des buts du langage C++ est d'exprimer directement ses idées dans le langage du problème auquel on se trouve confronté, et ce d'une manière pratique et sûre sans rien concéder au niveau de l'efficacité. Le programmeur doit donc pouvoir définir des abstractions de la réalité que l'on nomme classes en C++, puis, par la suite, manipuler des objets instances de ces classes.

Supposons que nous cherchions à construire un éditeur d'entités graphiques. Nous allons définir une abstraction des entités que nous voulons manipuler. Une des solutions consiste à trouver les caractéristiques minimales d'une classe d'objet qui servira de classe de base pour dériver toutes les autres classes. La première classe d'objet que nous allons manipuler s'appellera "obj_graphique".

Nous savons que l'un des concepts centraux de la programmation orientée objet repose sur une théorie unificatrice qui consiste à regrouper du code et des données dans une entité appelée objet. En C++ les attributs sont appelés *données membres* de l'objet et les méthodes sont appelés *fonctions membres*.

Le C++ donne la possibilité de fusionner le code et les données en étendant le concept des structures **struct** et des unions **union** du langage C avec le concept des classes **class** de SIMULA.

```
class Obj_graphique        // Déclaration de la classe obj_graphique
{
  // Attributs
  // Méthodes
};

Obj_graphique     cercle ; // Déclare une instance de la
                           // classe obj_graphique
```

Remarque : les commentaires "//" sont monolignes. Si vous généralisez leur utilisation dans vos programmes, ils permettront la mise en commentaire de zones entières de code avec les symboles classiques de commentaire du C "/**/".

Il est possible d'utiliser cette classe d'objet comme un nouveau type et ainsi déclarer des tableaux d'objets graphiques, des pointeurs sur des objets graphiques comme nous l'aurions fait sur de simples entiers. Le langage C++ étant considéré comme hybride, il n'est pas possible de considérer tous les types (ceux du prédéfinis du langage et ceux des utilisateurs) comme des classes d'objets, les types prédéfinis ne sont pas des objets.

Il est possible de contrôler, avec une très grande souplesse, l'accès aux données et aux fonctions membres d'une classe. La visibilité des membres peut en effet être modifiée au moyen des mots clés **private, protected,** et **public.**

Le mot clé **private** ne s'applique qu'aux déclarations de classes et de structures et signifie que tous les membres (données et fonctions) déclarés derrière **private** (et ce jusqu'au prochain mot clé de protection) ne seront accessibles qu'aux fonctions membres de l'objet en cours de déclaration.

Le mot clé **protected** ne s'applique qu'aux déclarations de classes, il est équivalent à **private** pour les membres de la classe déclarée mais il permet également aux fonctions membres des sousclasses dérivées d'accéder aux membres de la classe mère. Ce mécanisme sera repris lorsque nous aborderons la notion d'héritage.

Le mot clé **public** spécifie que tous les membres déclarés entre **public** et un autre mot clé de protection ou une fin de déclaration, sont visibles par tous sans autre restriction que la portée de l'objet.

Les protections par défaut sont les suivantes : pour les classes la protection est **private**, pour les structures elle est **public**, quant aux unions les membres en seront toujours **public** sans que cela puisse être modifié.

2.2 Déclaration de classes

Nous avons déjà commencé à nous poser le problème d'un éditeur d'objets graphiques. Raisonnons dans un premier temps comme nous l'aurions fait avec C. La majorité des éditeurs du commerce repèrent les objets graphiques à l'aide de deux coordonnées qui spécifient une fenêtre dans laquelle se trouve contenu un objet. Cette approche classique pour les objets graphiques est reprise et justifiée par [GORLEN 87]. Rassemblons ces coordonnées dans une structure de façon à pouvoir déclarer un certain nombre d'entités pour ce nouveau type.

```
struct Obj_graphique
{
   char     nom[30] ;       // Nom de l'objet
   int      xig , yig ;     // Coordonnées inférieures gauches
   int      xsd , ysd ;     // et supérieures droites.
};
Obj_graphique    cercle, * ptr_obj , tab_obj[ 10 ] ;
```

Les programmeurs C auront remarqué l'absence du mot clé **struct** au niveau de la déclaration des instances. Nous avons défini un nouveau type sans avoir explicitement utilisé le mot clé **typedef**. En fait on peut considérer les structures et les unions comme des classes d'objet particulières avec lesquelles il n'est pas possible de construire une hiérarchie de classes. Ainsi cercle est une instance de la classe obj_graphique, ptr_obj pointe sur un objet graphique et tab_obj est un tableau de 10 objets graphiques.

2.3 Les méthodes

Intéressons nous maintenant à ce que l'on veut faire de ces objets graphiques. Dans un premier temps, attachons nous seulement à des actions simples. Il serait intéressant de connaître le nom d'un objet ou de savoir si il vient d'être sélectionné avec une souris. Il est possible grâce aux fonctions membres de spécifier ces actions dans le bloc de déclaration d'une classe, d'une structure ou d'une union.

```
struct Obj_graphique
{
   char     nom[30] ;          // Nom de l'objet
   int      xig , yig ;        // Coordonnées inférieures gauche
   int      xsd , ysd ;        // et supérieures droite.

   int      contient( int x , int y );    // Méthodes ou fonctions membres
   char*    sonnom() { return nom; }
};
```



```
int Obj_graphique::contient (int x = 0, int y = 0)
{
   if ( x >= xig && x <= xsd && y >= yig && y <= ysd ) return 1;
   else return 0;
}
```

Cet exemple nous fait découvrir de nouveaux aspects du langage C++. Tout d'abord, il est possible de définir le corps d'une fonction dans une déclaration de méthodes. Ici la fonction membre "sonnom" est dite déclarée de type **inline** implicite lorsque les instructions du corps de la fonction sont explicités dans la déclaration de la classe; il est alors inutile de spécifier le mot clé **inline**. Les fonctions **inline** sont en fait des macros, elles sont utilisés pour des raisons d'efficacité et généralement pour des fonctions très courtes, le code étant recopié. L'exemple donné ci-dessus montre une utilisation idéale du mot clé **inline** pour une méthode d'accès à une donnée membre. Ceci permet à la fois un style de programmation élégant, respectant les principes de la programmation orientée objet, sans pour autant entraîner de surcoût en matière d'efficacité. Toute fonction peut cependant être déclarée **inline** en dehors d'une définition de classe, il suffit de faire précéder la déclaration de la fonction par **inline**.

Une autre nouvelle notion est celle de l'opérateur de résolution de portée "::". Il permet de nommer la classe, la structure ou l'union propriétaire de fonctions membres que l'on appelle fonctions déportées. Ce terme "déportée" est employé car le contenu de ces fonctions n'est pas détaillé dans la déclaration de la classe mais à l'extérieur de cette déclaration. Ici *obj_graphique::contient()* permet d'identifier clairement la fonction *contient()* car d'autres fonctions pourraient avoir le même nom dans d'autres classes ou simplement dans le corps du programme. L'opérateur de résolution de portée peut également s'appliquer à tous les membres d'un objet.

Vous aurez sans doute également remarqué que des valeurs par défaut (implicites) peuvent être données aux arguments de la fonction contient. En effet, comme avec le langage C, il est possible d'appeler une fonction sans lui donner tout ses paramètres, voire sans lui en donner du tout ! (Moins on empile de paramètres plus on est efficace). Les valeurs implicites seront affectés aux paramètres qui ne seraient pas spécifiés lors d'un appel de fonction.

Etudions maintenant l'accès aux membres d'une classe. Comme pour les membres d'une structure C, l'opérateur de résolution "." permet à une instance de classe (ou objet) d'accéder aux attributs comme aux méthodes. La notation pointée "->" du langage C a toujours cours, un pointeur sur une classe d'objet accédera aux membres par cette notation pointée.

```
cercle.contient(76,235) ;        // Teste si le cercle contient cette
                                 // coordonnée
Obj_graphique *  ptr_obj ;       // Pointeur sur un objet
                                 // graphique
ptr_obj = new Obj_graphique;     // Allocation de mémoire
ptr_obj->contient(100,200) ;     // Appel par notation pointée
                                 // de la fonction membre contient
```

2.4 La construction et la destruction d'objets

Lors de la déclaration d'une instance de classe, l'espace nécessaire aux données membres et aux pointeurs sur les fonctions membres est alloué. Comme dans toute approche structurée, il convient de décider de l'initialisation des membres de l'objet. On peut comme en C écrire le code nécessaire à l'initialisation après chaque création d'instance. Cette démarche est très lourde, une fonction qui réaliserait ce travail serait la bienvenue. Ceci a été prévu dans le langage C++ qui propose d'utiliser une fonction membre spéciale appelée "constructeur"

d'une classe. Le nom de ce constructeur est celui de la classe à construire, si le programmeur oublie de spécifier un constructeur, il est généré automatiquement par défaut sans arguments.

L'extrait de source C++ qui suit donne le code d'un constructeur pour la classe "obj-graphique". Ce constructeur peut être appelé de deux manières : soit lors d'une déclaration statique d'objet, soit avec une déclaration dynamique d'objet. L'opérateur new remplace avantageusement la fonction *malloc()* du langage C (pour plusieurs raisons qui vous seront exposées ultérieurement).

```
class Obj_graphique
{
  protected :
     char      nom[30] ;            //    Nom de l'objet
     int       xig , yig ;          //    Coordonnées inférieures gauches
     int       xsd , ysd ;          //    et supérieures droites.
  public :
     int       contient( int x , int y ) ;
     char*     sonnom() { return nom ; }
     Obj_graphique ( int x1, int y1, int x2, int y2, char *name) ;
     ~Obj_graphique () { }
};


Obj_graphique::Obj_graphique (   int x1, int y1,
                                 int x2, int y2, char *name )
{
   xig   =  x1 ;
   yig   =  y1 ;
   xsd   =  x2 ;
   ysd   =  y2 ;
   strcpy(nom,name) ;
}
```

Il est également possible à un constructeur d'affecter des valeurs aux données membres d'une classe en utilisant une liste de valeurs d'initialisation avant le corps de la fonction. La même fonction constructeur peut donc s'écrire comme ci-dessous (de plus, il est possible de combiner les méthodes d'initialisation).

```
Obj_graphique::Obj_graphique (int x1, int y1, int x2, int y2, char *name) :
                              xig(x1), yig(y1), xsd(x2), ysd(y2)
{
     strcpy(nom,name) ;
}
```

Le destructeur, quant à lui, est aussi une fonction membre spéciale: son rôle est d'effacer les données membres et de rendre la mémoire allouée à l'objet. Le nom d'un destructeur est celui de la classe de l'objet mais il est précédé d'un tilda (~) pour le différencier du constructeur. Si on omet de déclarer un destructeur, le C++ en crée un (comme pour les constructeurs). Les destructeurs sont appelés automatiquement dès qu'un objet sort de sa portée ou plus tôt si l'on utilise l'opérateur **delete** (le pendant de **new**).

Dans cet exemple, nous avons utilisé une définition de classe à la place d'une définition de structure. Pour une structure les membres sont **public**. Les données membres ont été placées derrière une spécification **protected** de manière à ce que les fonctions membres d'une classe dérivée puissent y accéder. En effet, si rien n'est spécifié, les données membres d'une classe sont **private** et les classes dérivées n'y auraient pas accès. Par contre, les fonctions membres ont été spécifiées **public**, de manière à être accessibles de n'importe quel endroit d'un futur programme.

Il est possible de spécifier autant de contrôle d'accès que l'on souhaite, quoiqu'il semble plus cohérent de regrouper au même endroit tous les membres bénéficiant du même type de protection.

```
class Objet_exemple
{
  private :
     // Membres (données et fonctions) visibles uniquement
     // des instances  de la classe objet_exemple
  protected :
     // Membres (données et fonctions) visibles  des instances
          // de la classe objet_exemple et des instances
     // d'objets dérivées en public de la classe objet_exemple
  public :
     // Membres (données et fonctions) visibles de tous

}
```

Si par défaut les membres d'une classe sont considérés comme **private**, par contre, les membres d'une structure **struct** sont considérés à accès **public** et les membres d'une **union** sont obligatoirement à accès **public** non modifiable.

2.5 Les déclarations de classes et les modules

Dans un programme C++, une convention veut que les déclarations de classes soient regroupées dans des fichiers "entête" (ou header) qu'il faut inclure dans les fichier corps de programme. Le corps de la déclaration d'une classe
Le premier type de fichiers (les entêtes) utilise les extensions ".h",".hpp" ou "hxx" et ne laisse entrevoir que le côté conception d'une classe, le second type utilise les extensions ".cpp", ".CC", ".C" ou ".cxx" et développe l'implémentation de la classe. Bien sûr il est conseillé de travailler en compilation séparée avec si possible un fichier entête et un fichier corps par classe, la modularité reste encore un des éléments clés de la programmation orientée-objet. S'il est difficile pour un développeur moyen de comprendre un programme d'un million de lignes de code écrit d'un seul tenant, il lui est plus facile de comprendre mille fois un module de mille lignes.

3 La notion d'héritage et de hiérarchie

3.1 Généralités

La modélisation d'un système complexe est facilitée par les possibilités d'abstraction et d'encapsulation qui permettent de cacher la représentation interne des programmes. De plus, la modularité d'un programme nous permet de décomposer de manière logique un problème complexe jusqu'à obtenir des problèmes élémentaires. Malgré cela il reste difficile d'appréhender un ensemble d'abstractions sans une approche hiérarchique rigoureuse. Il convient donc d'essayer de classifier ces abstractions, ce processus de classification s'appelle la taxonomie. Les biologistes proposent de classer des insectes suivant leurs familles, Andrew Koenig se propose de classer des véhicules [KOENIG 88a], tous cherchent à montrer que la construction d'arbres généalogiques ou de graphes, appliquée à la programmation orientée objet apporte une méthode puissante de représentation de la connaissance. Le principe de l'héritage peut s'exprimer en terme simple de la manière suivante : un objet 'o' possède un ensemble de caractéristiques de base et on cherche à étendre ces caractéristiques en lui rajoutant quelques attributs et quelques méthodes. Au lieu de modifier la classe de l'objet initial, il suffit de créer un nouveau type d'objet qui possède toutes les caractéristiques de l'objet initial plus les nouvelles que l'on cherchait à lui

adjoindre. En C++ on parle de classe de base pour une superclasse et de classe dérivée pour toutes les sousclasses qui héritent de la classe de base.

Les arbres ne sont pas tous strictement hiérarchiques : une hiérarchie simple autorise des enfants à hériter de leur mère (d'où la notion d'héritage simple) mais il existe des problèmes plus complexes mettant en oeuvre l'héritage multiple (héritage du côté du père et de la mère) qui engendrent (pour l'informatique) un certain nombre de problèmes que nous évoquerons par la suite.

3.2 Extension d'une classe et héritage simple

La classe que nous avons précédemment définie n'est pas en elle même d'une grande utilité. En fait, il s'agit d'une classe de base à partir de laquelle il devient possible de construire une myriade de classes dérivées. Nous allons essayer de construire une classe générale pour des objets de production. Cette classe sera susceptible d'acceuillir des machines, des stocks, des pousse-pousses et tout autres types d'objets que l'on trouverait dans un atelier de production. Il s'agit d'enrichir la classe de base *obj_graphique* avec des nouveaux attributs et des nouvelles méthodes. Voici comment nous pouvons écrire cette nouvelle classe à partir de notre connaissance actuelle de *obj_graphique*.

```
class Obj_graphique
{
  protected :
    char        nom[30] ;      // Nom de l'objet
    int         xig , yig ;       //    Coordonnées inférieures gauches
    int         xsd , ysd ;       //    et supérieures droites.
  public :
    int         contient( int x , int y ) ;
    char*       sonnom() { return nom; }
    Obj_graphique ( int x1, int y1, int x2, int y2, char *name) ;
    ~Obj_graphique () { }
};

class Obj_prod : public Obj_graphique
{
  protected :
    int      couleur ;
    int      taille ;
    int      capacité ;
    int      etat ;
  public :
    Obj_prod (int   x1,int y1,int x2,int y2,
              char  *n,int c,int t,int s,int e) ;
    ~Obj_prod () { }
    int      sacouleur() {  return couleur; }
    int      sataille()  {  return taille; }
    ...
    void     enpanne()   {  etat = 0 ; }
};
```



```
Obj_prod::Obj_prod (int x1,int y1,int x2,int y2,
                    char* n,int c,int t,int s,int e)
  : Obj_graphique (x1,y1, x2,y2,n)
{
  couleur  = c ;
  taille   = t ;
  capacité = s ;
  etat     = 1 ;            // L'objet de production fonctionne
}
```

Les déclarations de dérivations s'effectuent en spécifiant la classe dérivée, puis la classe de base précédée de ":" et du modificateur d'accès à la classe de base.

Pour cette portion de code, c'est obj_graphique qui est la classe de base et obj_prod qui est la classe dérivée. Ici la dérivation s'effectue avec le modificateur d'accès **public**, ce qui signifie que les membres de la classe de base seront vus comme suit dans la classe dérivée :

public s'ils étaient déclarés **public** dans la classe de base,
invisibles s'ils étaient déclarés **private** dans la classe de base,
protected s'ils étaient déclarés **protected** dans la classe de base.

Si le modificateur d'accès est **private** ou s'il n'est pas spécifié (**private** est alors considéré par défaut), les membres de la classe de base seraient vus comme suit dans la classe dérivée :

private s'ils étaient déclarés **public** dans la classe de base,
invisibles s'ils étaient déclarés **private** dans la classe de base,
private s'ils étaient déclarés **protected** dans la classe de base.

Classe de base	Modificateur d'accès	Classe dérivée
public	public	public
private	public	non accessible
protected	public	protected
public	private	private
private	private	non accessible
protected	private	private

Figure N° 1 : Droits d'accès des classes

Si nous considérons la dérivation d'une structure et non d'une classe, le modificateur d'accès par défaut est **public**.

En résumé pour l'exemple que nous avons donné, les fonctions membres de la classe *obj_prod* peuvent ici avoir accès à toutes les données membres et toutes les fonctions membres de la classe *obj_graphique*. Bien sûr, elles disposent également de l'accès à tous les nouveaux membres déclarés (une fonction membre peut en appeler une autre, voire s'appeler elle-même...).

Remarque: le constructeur de la nouvelle classe fait appel au constructeur de la classe de base obj_graphique. Ceci évite de réécrire les lignes d'initialisation des coordonnées de l'objet graphique ainsi que celles d'initialisation du nom de l'objet. Ce genre d'économie de code peut paraître fortuite dans ce cas précis mais, pour des exemples de taille importante, il convient de ne pas la négliger. De plus, l'initialisation de chaque donnée membre n'est faite

qu'à un seul endroit, ce qui a pour effet d'augmenter la fiabilité du programme et de diminuer la taille du code.

Regardons maintenant comment nous pouvons construire des objets **machine** et **stock** à partir de cette nouvelle classe.

```
class Machine : public Obj_prod {
  protected :
     float temps_usinage ;
  public :
     Machine(
        int x1,int y1,int x2,int y2, //  Coordonnées
        char *n,                     //Nom de la machine
        int c, int t, int s, int e,  //  Couleur, taille, capacité, état
        float ts) ;                  //Temps d'usinage
     ~Machine() { }
     float sontemps_usinage() {return temps_usinage; }
  } ;

Machine::Machine
   (int x1,int y1,int x2,int y2,char*n,int c,int t,int s,int e,float u)
   :obj_prod ( x1, y1, x2, y2, n, c,  t,  s,  e)
{
   temps_usinage = u ;
}

class Stock : public Obj_prod {
  protected :
     char  politique ;            //Politique de gestion du stock
  public :
     Stock(
        int x1, int y1, int x2, int y2,
        char *n,
        int c, int t, int s, int e,
        char p ) ;
     ~Stock() { }
     char  sapolitique() { return politique; }
} ;

Stock::Stock( int x1, int y1, int x2, int y2, char*n,
          int c, int t, int s, int e, float u)
     :Obj_prod (x1, y1, x2, y2, n, c, t, s, e)
{
   politique = p ;                  //  Politiques LIFO, FIFO ou VRAC
}
```

Il est important de rappeler que les constructeurs de machine et de stock appellent le constructeur de la classe *obj_prod* qui elle même appelle le constructeur de la classe *obj_graphique*. Pour fixer un peu les idées donnons des exemples de code que l'on peut écrire sur les objets stock et machine.

```
Machine m  (0,0,100,10,"MACHINE 1", BLEUE, 5, 1, MARCHE, 45.876) ;
Stock*  ptr_stock ;

ptr_stock  = new Stock (5,10,200,40,"STOCK A",JAUNE,7,10,MARCHE,FIFO);
m.contient(2,2) ;
ptr_stock->enpanne() ;
float temps    = m.sontemps_usinage() ;
```

Une machine ou un stock peuvent facilement appeler le code de la fonction membre contient(). En effet, ce code est commun à tous les objets dérivant en **public** de la classe obj_graphique. De même, la fonction membre enpanne() est partagée par tous les objets dérivant en **public** de la classe obj_prod, mais seule une machine donne accès à la donnée membre temps d'usinage. En fait, les mécanismes d'héritage fournis par le langage C++ permettent au programmeur de suivre une méthode de conception orientée-objet qui évitera les redondances habituelles d'une programmation structurée classique tout en augmentant très sensiblement la fiabilité du code produit.

3.3 Héritage multiple

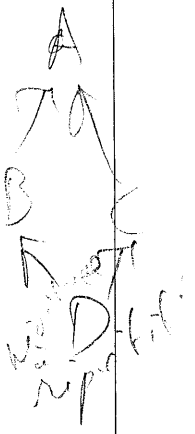
Dans un système de production flexible il est fréquent de rencontrer des robots, or ces robots se comportent souvent à la fois comme des machines et comme des stocks (voire même fréquemment comme des systèmes de transports...). Nous nous trouvons face à un cas d'utilisation de l'héritage multiple; avec le langage C++ il est possible d'exprimer ceci de manière assez simple en donnant une liste des classes dérivées avec leurs modificateurs d'accès :

```
class Classe_dérivée : public Classe_mère, public Deuxième_classe_mère { … };

class Classe_dérivée : public Classe_mère, private Deuxième_classe_mère { … };
```

```
class Robot : public Machine, public Stock {
  protected :
     float tps_extract_stock ;   // Temps d'extraction du stock
  public :
     float sontps_ex_stk() { return tps_extract_stock ; }
     void  modif_param_robot(float t,  char p) ;
     Robot(
        int x1, int y1, int x2, int y2, char *n,
        int c, int t, int s, int e,
        char p, float t,
        float t2
        ) ;
     ~Robot() { }
  } ;

Robot::modif_param_robot(float t1, float t2, char p)
{
  stock::politique = p ;          // Change la politique de la file
  machine::temps_usinage = t1 ; // le temps d'usinage et
  tps_extract_stock = t2 ;        // d'extraction du stock pour
}                                 // gérer un nouveau type de pièces

Robot::Robot( int x1, int y1, int x2, int y2, char *n,
              int c, int t, int s, int e, char p, float t, float t2 )
     : stock( x1, y1, x2, y2,  n, c, t, s, e , p) ,
     machine ( x1, y1, x2, y2, n, c, t, s, e ,t )
{
  tps_extract_stock = t2 ;   // constructeur par défaut de obj_prod appelé
}
```

L'accès aux membres propres de la classe définie s'effectue comme lors d'un héritage simple, par contre lors de l'accès à des membres hérités, il est du ressort du programmeur de spécifier de quelle classe de parents ils proviennent, ce qui est fait dans la fonction modif_param_robot(). L'opérateur de résolution de portée "::" indique à partir quelle classe parente on souhaite récupérer des membres. C'est de cette manière explicite que sont réglés les conflits de l'héritage multiple en C++. Le constructeur de cette classe *robot* appelle les deux constructeurs des classes *stock* et *machine* qui appeleront chacun le constructeur de la classe *obj_prod*. Pour éviter un héritage à répétition de la classe *obj_prod* commune à *stock* et *machine*, il faut spécifier un modificateur d'accès virtuel lors de la définition de ces classes.

```
class Machine : virtual public Obj_prod {…};  // l'emplacement du virtual
                                              // devrait ne pas avoir d'importance
class Stock : virtual public Obj_prod {…};    // le positionnement avant public
                                              // est plus implémentée

class robot : virtual public obj_prod , public machine, public stock
{
    public:
       robot(…) : obj_prod(…), machine(…), stock(…) ; // Chaînage > constructeurs
    …
}
```



Avec ces déclarations, les compilateur C++ n'appelleront qu'une seule fois le constructeur de la classe *obj_prod*, et ne produirons qu'une seule instance de cette classe évitant ainsi l'héritage à répétition. Les constructeurs des classes de base virtuelles sont appelés avant toutes les autres classes de base non virtuelles (lorsqu'un graphe d'héritage contient plusieurs classes de base virtuelles, leurs constructeurs sont appelés dans l'ordre des déclarations). Ensuite, les classes de base non virtuelles sont appelées avant que les constructeurs des classes dérivées ne soient appelés. Un héritage "virtuel" alourdit la gestion des tables des fonctions membres virtuelles et ne permet pas de transtypage descendant (downcast).

Nous conseillons aux lecteurs intéressés de se reporter au manuel de référence ANSI du langage [ELLIS 90] ou aux ouvrages plus récents de Stanley Lipman pour plus de détails

4 Le polymorphisme et les fonctions virtuelles

4.1 Généralités

Le mécanisme de polymorphisme doit être présent dans tous les langages à objets [MASSINI 88] [MEYER 88] [BOOCH 90]. Ce mécanisme est directement issu du langage SIMULA [DAHL 68] [BIRTWISTLE 73] [KIRKERUD 89], il permet de donner le même nom à plusieurs fonctions membres d'une même hiérarchie. Chaque membre fonction exécute un code différent.

On doit distinguer les fonctions virtuelles de la surcharge de fonctions. En effet les fonctions virtuelles expriment sous des formes de code différentes (polymorphisme) une action correspondant à une même sémantique mais en autorisant l'accès à la liaison dynamique. De plus, une fonction virtuelle doit avoir rigoureusement la même signature d'arguments dans toutes ses implémentations (même nombre et mêmes types de paramètres). Par contre, les fonctions surchargées se distinguent grâce à leur signature d'argument.

Pour fixer les idées, nous allons considérer l'affichage et l'effacement des objets graphiques sur lesquels nous travaillons. Un objet machine ne devra pas avoir le même formalisme graphique qu'un stock ou qu'un convoyeur. Cependant nous aimerions afficher n'importe quel objet à l'aide d'une seule ligne de code spécifiant que l'on souhaite afficher l'objet sans se soucier de son type. Pour cela il faut disposer d'autant de fonctions membres qu'il y a de types d'objets différents (une fonction pour la classe machine, une pour la classe convoyeur, etc ...). Puisque toutes ces fonctions répondent à la même sémantique, il convient de leur donner le même nom : *affiche()* . Enfin, pour pouvoir lancer l'affichage sans se soucier du type d'objet, il suffit de déclarer la fonction *affiche()* comme étant virtuelle pour la classe de base *obj_graphique*.

Le code qui suit donne un exemple de ce que cela peut donner au niveau des classes que nous avons déjà définies. Il est important de noter qu'aucune adjonction n'est faite au niveau de la classe *obj_production* qui sait déjà qu'elle récupère par héritage **public** une fonction membre virtuelle *affiche()*.


```
class Obj_graphique {
  protected :
    ....                        // Données membres précédemment
                                // déclarées.
  public :
    ....                        // Fonctions membres
                                // précédemment déclarées.
    virtual  void  affiche(int) ;
  } ;

class Machine : public Obj_prod {
  protected :
    float temps_usinage ;
  public :
    ....                        // Fonctions membres
                                // précédemment déclarées.
    void affiche(int mode) ;
    } ;

void Machine::affiche(int mode)
{
  gp_circle(x1, y1, x2, y2, mode); //    Fonction graphique d'affichage
}                                  //    d'un cercle

class Stock : public obj_prod {
  protected :
    char  politique ;
  public :
    ....                        // Fonctions membres
                                // précédemment déclarées.
    void affiche(int mode) ;
} ;


void Stock::affiche(int mode)
{
  gp_rectangle(x1,y1,x2,y2,mode);  // Primitive graphique d'affichage
}                                  // d'un rectangle
```

Ceci correspond à l'utilisation classique des fonctions virtuelles, nous pouvons grâce à ce type de code au sein des classes, utiliser un code principal indépendant du type de l'objet courant. En effet, supposons que tous les objets graphiques d'un modèle quelconque soient stockés dans une table, il suffit alors pour afficher tous les objets (de n'importe quel type) d'appeler la fonction virtuelle *affiche()* pour chaque entrée de la table. Le code résultant pourrait être le suivant.

```
Obj_graphique *   tab[100] ;
Machine           ma(.......);    // Appel des constructeurs
Stock             stk(.......);   // de machine et stock avec les
                                  // paramètres nécessaires
...
tab[i] = &(Obj_graphique) ma ;      //    Stockage des élts de la table
tab[j] = &(Obj_graphique) stk ;  // avec conversion de type

void affiche_tout(int mode)
{
   for(int register i = 0 ; i <= MAXTAB ; tab[i++]->affiche(mode)) ;
}
```

Sur cet exemple vous remarquerez une autre particularité du C++ qui est celle de pouvoir déclarer des variables (ou des objets) à n'importe quel endroit dans le programme source.

4.2 Implémentation au niveau machine

Le code de la fonction *affiche_tout()* est capable d'afficher correctement tout type d'objet graphique. Cependant lors de la génération de code le compilateur ne connaît pas encore la fonction *affiche()* qu'il doit appeler. Doit-il appeler la fonction *affiche()* d'un stock, celle d'une machine, ... Ce n'est qu'à l'exécution que le programme décidera quelle fonction membre de quelle classe d'objet il appellera. On parle d'appel calculé (à l'exécution) et d'édition de lien dynamique (ou ligature dynamique). Comparons le code généré lors de l'appel d'une fonction simple et celui généré par l'appel d'une fonction virtuelle (nous prendrons un code assembleur virtuel).

```
Code C++                Code assembleur virtuel

objet.fonction_simple() ;     Lien statique

            CALL  @fn

objet.fonction_virtuelle() ;  Lien dynamique

            (1)   MOV     RBASE, @base_table_virtuelle
            (2)   MOV     RINDX,selecteur_de_la_methode
            (3)   MUL     RINDX,taille@
            (4)   CALL    [RBASE + RINDX]
```

Ce code peut être explicité comme suit : dans le cas de l'appel classique, le compilateur connaît l'adresse de la fonction à appeler (@fn) et code un simple appel de sous-programme. Par contre dans le cas d'un lien dynamique, (1) le compilateur récupère dans un registre de base l'adresse de base de la table de fonctions virtuelles pour la classe courante (ces tables sont générées automatiquement pour toutes les classes d'objets [ELLIS 90]). C'est le type de l'objet que l'on est en train de manipuler (connu lors de l'exécution du programme) qui permet de sélectionner la bonne table virtuelle. Le sélecteur de la méthode est placé dans un registre d'index (2). Ce sélecteur permet de calculer le déplacement dans la table de fonctions virtuelles (3), il suffit de connaître l'encombrement mémoire en octets d'une adresse de fonction puis de multiplier le code du type par cet encombrement. La multiplication serait bien sûr faite par décalage à gauche et non pas avec un éventuel opérateur MUL coûteux en cycles machine, cependant, il nous est paru plus clair d'exposer le mécanisme avec un mnémonique MUL pour spécifier la multiplication.

Les fans de tableaux de pointeurs de fonctions en langage C auront reconnu dans ce mécanisme des concepts qu'ils manipulent couramment. Notez que l'appel simple est plus performant que l'appel calculé avec double indirection (une pour la table, et une pour l'appel), cette efficacité est toute relative si l'on considère le logiciel dans sa globalité (nous ne tenons pas ici compte de la compétence des développeurs en matière de codage et de choix des structures de données).

Il peut être utile de donner un exemple de code C standard et de le comparer à son équivalent en C++. Le code que nous vous proposons en exemple appelle la fonction *affiche()* pour un certain nombre de classes d'objet. En C le type des objets sera supposé connu et stocké dans un identificateur *type*.

```
switch(objet.type) {       // Chaque type d'objet possède
                           // sa fonction d'affichage
   case  CERCLE :  objet.affichercercle() ;
                   break;
   case  LIGNE :   objet.afficheligne() ;
                   break;
   case  RECTAN:   objet.affichererectanle() ;
                   break;
   ...
   case  TEXTE :   objet.affichetexte();
                   break;
   default :       break;
   }
```

Ce code, écrit en C++, avec les mécanismes de hiérarchies de classes et de fonctions virtuelles, se résume à la ligne suivante.

```
objet.affiche();
```

L'adjonction d'un nouveau type d'objet suppose l'adjonction d'un nouveau 'case' dans l'instruction 'switch' pour la solution C classique. Par contre est inutile de modifier la ligne de C++. Il est bien sûr possible d'obtenir un codage quasi-equivalent en C, en écrivant explicitement l'appel par tableau de pointeurs de fonctions. Les déclarations, les dimensionements, et les initialisations du tableau de pointeurs de fonctions devront être également explicitées, par le programmeur.

```
// Déclaration d'un table de pointeurs de fonctions

int      (* tab_ptr_fn[NB_MAX_OBJET] )() ;

// Exemple d'initialisation

*(tab_ptr_fn + CERCLE) = &AfficheCercle;
...
// Appel de la fonction correspondant au type donné

(* *(tab_ptr_fn + type) ) () ;
```

Vous aurez remarqué que ce code est peu clair et que l'appel de fonctions virtuelles remplace avantageusement ce type de code source au niveau lisibilité, fiabilité et maintenance.

Le mécanisme de fonction virtuelle prend toute son envergure lorsqu'on réalise que le code de la fonction *affiche_tout()* restera la même quel que soit le nombre de classes dérivant de la classe *obj_graphique* que l'on souhaitera rajouter. Il devient possible de rassembler toutes les définitions de classes précédemment décrites dans une bibliothèque (le texte source n'étant plus indispensable). Il n'est plus nécessaire de prévoir quelles classes d'objet dérivées peuvent être rajoutées, les fonctions virtuelles se chargent d'appeler la bonne méthode au bon moment. Il s'agit du mécanisme le plus puissant mis en oeuvre par les langages à objets, le développeur peut enfin concevoir des classes et du code largement réutilisable.

Il est important de signaler que le type d'implémentation qui vient d'être exposé n'est possible que pour des langages à typage statique (comme C++). Pour des langages à typage

dynamiques (Smalltalk, Objective C), le mécanisme est plus complexe et s'il offre moins d'efficacité, il propose par ailleurs beaucoup plus de souplesse et de puissance d'abstraction.

Il nous semble maintenant nécessaire de rappeler quelques précautions dans l'utilisation des fonctions virtuelles. Seules des fonctions membres d'une classe peuvent être virtuelles, toutes les fonctions virtuelles doivent avoir la même signature d'arguments c'est à dire les mêmes paramètres (avec les mêmes types) et le même type pour la valeur de retour. En cas de signatures différentes, les fonctions ne seraient plus virtuelles mais simplement surchargées.

4.3 La technique de redéfinition (ou de substitution)

Il est fréquent dans une application, qu'un ensemble de classes d'objets dérivées utilise une fonction membre héritée de la classe de base. Par exemple la fonction *contient(int , int)* de la classe *obj_graphique* teste si une coordonnée saisie à l'aide d'une souris est contenue dans l'objet graphique de manière a sélectionner l'objet en question.

L'application se voit ensuite étendue par l'adjonction d'une classe d'objet cercle et d'une classe d'objet cadre dérivées de la classe de base *obj_graphique*. Pour ces nouvelles classes d'objet la fonction membre *contient()* ne remplit plus exactement son rôle car les cercles et les cadres peuvent contenir d'autres objets graphiques et les masquer lors d'une éventuelle sélection à l'aide d'une souris. Pour pallier cela nous allons faire de la fonction membre *contient()* une fonction virtuelle qui sera redéfine selon la technique dite de substitution. Seules les classes cercle et cadre auront à redéfinir la fonction virtuelle *contient()* testant seulement les contours des objets cercle et cadre, toutes les autres classes utiliseront le code de la classe de base. Grâce a cette technique un ensemble d'objets partagent le même code, et les cas particuliers spécifient leurs codes propres sans que les modules principaux n'aient à en tenir compte. En effet, un module principal appellera simplement la fonction virtuelle *contient()* sans s'occuper des cas particuliers.

```
class Obj_graphique {
  protected :
     ....                  // Données membres précédemment
                           // déclarées.
  public :
     ....                  // Fonctions membres
                           // précédemment déclarées.

     virtual  int    contient( int x = 0, int y = 0 )
     {
        if ( x >= xig && x <= xsd && y >= yig && y <= ysd ) return 1;
        else return 0;
     }
  } ;
```

Les classes cercle et cadre qui suivent seront dérivées en **public** de la classe de base obj_graphique.

```
class Cercle : public Obj_graphique {
  protected :
     float rayon;
  public :
     ...
     contient(int,int);
  }

Cercle :: contient( int x = 0, int y = 0)
{
  ...                          // Algorithme testant que l'on est
  ...                          // sur le contour du cercle
}
```



```
class Cadre : public Obj_graphique {
  public :                     // Pas d'autres attributs
     ...
     contient(int,int);
  }

Cadre :: contient( int x = 0, int y = 0)
{
  ...                          // Algorithme testant que l'on est
  ...                          // sur le contour du cadre
}
```

Il est important d'être conscient qu'une fonction membre ordinaire est plus performante qu'une fonction virtuelle, cependant pour toutes les situations susceptibles d'évoluer au cours de la vie du logiciel, il est préférable d'utiliser des fonctions virtuelles de manière à ce que des extensions futures ne remettent pas en cause les codes généraux à tous les objets. Il n'existe pas encore de règles générales et il convient de suivre

4.4 Les fonctions virtuelles pures et les classes abstraites

Une classe abstraite ne sert que de moule de base pour obtenir des classes dérivées et ne possède pas d'instance. Une telle classe propose par contre une interface commune à toute les classes dérivées par l'intermédiaire de ses fonctions virtuelles. Ces fonctions virtuelles peuvent être déclarées comme étant des fonctions virtuelles pures en affectant la fonction à 0 lors de sa définition. La présence d'une seule fonction virtuelle pure transforme une classe en classe abstraite.

```
classAbstraite {
  ...
  public:
     Abstraite() ;
     virtual void fnvirtuellepure() = 0 ;
     ...
  } ;
```

4.5 Les destructeurs virtuels

Il est possible de déclarer un destructeur virtuel dans une classe de base. Les destructeurs des classes dérivées d'une classe de base munie d'un destructeur virtuel sont eux-mêmes virtuels. Ainsi il est possible de manipuler des références à des objets de la classe de base abstraite (pointant en réalité sur des objets de classes dérivées) et lors de la destruction d'une de ces références d'appeler le destructeur propre à la classe dérivée à laquelle appartient l'objet pointé.

```
classObj_prod {
  ...
  public:
     ...
     virtual ~Obj_prod() ; // Spécification d'un destructeur virtuel

} ;
```

```
class Machine : public Obj_prod {
  …
  public:
    …
    ~Machine() { … };     // Le destructeur de machine est également virtuel
} ;
```

5 Les particularité du langage C++

5.1 Les références

Une référence est une manière de lier temporairement un identificateur à un nom d'objet [KOENIG 89a]. Par exemple, si l'on écrit :

```
int     tab[DIMENSION] ;
int     i  =  10 ;

int &r  =  tab[i] ;
```

Nous spécifions que r est un autre nom pour l'objet tab[i] qui est de la classe "int". Les liens de type référence sont repérés par le symbole "&" à la suite d'un type (ou d'une classe d'objet). Une fois un lien effectué, le nom 'r' réfère à l'objet i tout au long de sa portée. Le code suivant fixera les idées.

```
int     i  =  4 ;
int  &  r  =  tab[i] ;

i++ ;
```

La valeur de i est maintenant de 5, mais r continue de référer à tab[4]. Les habitués du langage C vont se demander qu'elle différence il existe entre une référence et un pointeur sur un objet. En fait, au niveau implémentation, il n'y en a aucune. Voici le codage en assembleur virtuel de quelques séquences de code C++:

```
int     i = 4 ;      MOV [@ i ] , 4     // Initialisation de i
int&    r = i ;      MOV RINDEX , @ i   // Init de la référence
                     MOV [@ r] , @ i    // à l'adresse de i
r++ ;                INC [RINDEX]

int     i = 4 ;      MOV [@ i ] , 4     // Initialisation de i
int *   p = &i ;     MOV RINDEX , @ i   // Init du pointeur p
                     MOV [@ p] , @ i    // à l'adresse de i
(*p)++ ;             INC [RINDEX]
```

Comme vous le voyez, l'implémentation est identique, un registre d'index RINDEX est utilisé pour stocker le pointeur ou la référence. L'écriture et l'emploi de références est beaucoup plus souple que l'utilisation explicite de pointeurs. Il s'agit d'un niveau d'abstraction supplémentaire emprunté à SIMULA.

Une des utilisations fréquente des références peut être celle du passage de paramètres par adresse. Le passage de paramètre par référence correspond au mode de passage par adresse qui évite de traîner les notations pointées du langage C qui deviennent lourdes lorsque l'on dépasse deux niveaux d'indirection. En voici un exemple simple.

Passage par adresse en C ANSI

```
swap(int * x, int * y)
{
  int     tmp = *x ;

  *x  = *y ;
  *y  = tmp ;
}

swap( &a , &b );
```

Puis par référence en C++ ANSI

```
swap( int& x, int& y)
{
   int   tmp = x ;

   x = y ;
   y = tmp ;
}

swap( c , d );
```

En C vous aurez remarqué l'indirection pour accéder à la valeur entière spécifiée par les pointeurs sur x et y . Avec C++ une simple "référence" suffit pour récupérer l'entier en question, il n'y a pas à s'occuper des différents niveaux d'indirection. De plus, le corps de la fonction n'utilise pas la notation étoilée qui, si elle est amusante, n'en reste pas moins lourde a gérer sur plusieurs niveaux. Les performances restent identiques dans les deux cas.

Une autre utilisation fréquente des références consiste à choisir ce mode de passage de paramètres y compris pour des fonctions qui ne modifient pas leurs paramètres d'entrée. En effet, le mode de passage des paramètres par défaut en langage C est le passage par valeur qui consiste a travailler sur des copies des arguments de manière à restituer les valeurs originales des arguments au retour de l'appel de la fonction. Ce mode de passage (de paramètres) est le moins efficace qui soit, c'est pourquoi lorsque certains arguments ne sont pas modifiés, il est intéressant de travailler par adresse (de préférence avec des références). Lorsque l'on choisit le mode de passage par adresse pour des raisons d'efficacité ou tout simplement parce que le type de paramètre passé ne peut être copié, il est conseillé de déclarer les arguments en références constantes. L'exemple suivant illustre ce propos.

```
class Machine : public Obj_prod {
  protected :
     float temps_usinage ;
  public :
     ....                        // Fonctions membres
                                 // précédemment déclarées.

     set_time(const float & tps) { temp_usinage = tps }
  } ;
```

Dans l'exemple précédent une méthode *set_time(...)* de la classe machine peut pour des raisons d'efficacité (en plus de sa déclaration en ligne implicite) récupérer sa nouvelle valeur par adresse. Notez le **const float &** qui d'une part assure que l'argument passé n'est pas modifié et d'autre part rappelle que ce mode de passage avec le sigle **&** est choisi pour des questions de performances.

Enfin, contrairement aux pointeurs, au niveau du C++, les adresses des références donnent les adresses des objets référés. L'exemple suivant illustre ce principe :

```
float   f ;
float&  ref=  f ;
float*  ptr=  &f ;

&ref    est égal à        &f (!...)
&f      et différent de &ptr
```

5.2 Une forme simple de polymorphisme : la surcharge des fonctions

Comme pour avec le langage SIMULA [DAHL 68] [BIRTWISTLE 73] [KIRKERUD 89], il est possible de déclarer des fonctions homonymes. Les fonctions virtuelles doivent avoir la même signature (mêmes types d'arguments et même nombre). Mais il peut être utile de définir une fonction qui effectuerait une action de même sémantique pour des types de paramètres différents. Par exemple, nous pourrions chercher à calculer la moyenne d'un tableau d'entier court ou long, ou encore de réels simples ou doubles, la manière de délimiter la taille du tableau pouvant être variable.

```
short   moyenne( short  *table, const int &     taille) ;
long    moyenne( long   *table, int debut, int  fin) ;
float   moyenne( float  *table, unsigned char   taille) ;
double  moyenne( double *table, const int & t   aille) ;
```

De telles définitions "surchargent" la signification de la fonction moyenne, qui devient sensible au contexte spécifié par les paramètres d'appels. Le compilateur est à même de savoir laquelle des quatre fonctions il doit appeller suivant le type et le nombre de paramètres d'appel (ou paramètres effectifs). Cette fonctionnalité apporte une souplesse supplémentaire et une meilleure lisibilité dans la mesure où elle n'est pas utilisée à outrance. La surcharge de fonctions avec un nombre d'arguments différent est particulièrement intéressante pour la définition de plusieurs constructeurs d'une classe.

L'exemple classique est celui des constructeurs d'une classe *string* pour gérer des chaînes de caractères [KOENIG 90b]. Voici les trois constructeurs surchargés proposés par Andrew Koenig, le lecteur intéressé par le code complet se reportera à [KOENIG 88b].

- Un premier constructeur **String::String()** sans paramètre détermine quelle valeur donner à un objet chaîne de caractères qui n'est pas explicitement initialisé.
- Un deuxième constructeur **String::String(char *)** pour pouvoir initialiser une chaîne à partir d'une constante texte passée en paramètre.
- Un troisième constructeur **String::String(const String &)** pour initialiser une chaîne à partir du texte d'une autre chaîne.

5.3 La récupération d'adresse de fonctions surchargées

Pour récupérer l'adresse de fonctions surchargées, l'ambiguïté de nom doit être résolue explicitement en spécifiant les types d'arguments du pointeur de fonction qui récupérera l'adresse de la fonction surchargée. Voici un exemple vous indiquant la manière de coder ce problème :

```
void    fonction(char) ;
int     fonction(double) ;

void    (*ptrfn)(char)     =  &fonction ;
int     (*ptrfn2)(double)  =  &fonction ;
```

5.4 La surcharge des opérateurs

S'il est possible en C++ de surcharger des fonctions utilisateur, il est également possible de surcharger les opérateurs du langage. Le nombre d'opérateurs du langage C++ étant vaste, les exemples intéressants ne manquent pas [ECKEL 89], [KOENIG 89b] et [STROUSTRUP 86]. Les seuls opérateurs qui ne peuvent pas être surchargés sont : (. .* :: ?). Les manuels de référence ANSI 2.0 [ELLIS 90] [STROUSTRUP 92] donnent les particularités de la surcharge d'opérateurs tels que l'indexation [] , l'opérateur d'affectation =, ou encore l'appel de fonction (). Nous allons nous attacher à redéfinir l'opérateur d'indexation et d'addition pour

des tableaux. Contrairement aux tableaux du langage C, cette nouvelle classe indique comment la validité des indices donnés lors d'adressage peuvent être testés à l'exécution.

```
class Tabint {
  protected :
    int      *  tab ;          // l'adresse de départ du tableau
    int         taille ;       // Taille du tableau
  public :
    Tabint (int) ;
    ~Tabint () { delete tab }            // Restitution de l'espace

    int   get_taille() { return taille; }
    int&  operator[] (int) ;             // Surcharge de l'indexation []
    void  operator=(cons Tabint &) ;     // de l'affectation =
    void* operator&() { return tab } ;   // et de l'opérateur adresse

    friend Tabint    operator +  (const Tabint & tab1, const Tabint & tab2);
    friend ostream & operator << (ostream & o, Tabint & table);

  }; // Les fonctions amies (friend) sont expliquées plus loin

Tabint::Tabint(int t) : taille(t)
{
   // taille = t est inutile avec le : taille(t)
   tab       = new int[taille];
   for(int i = 0; i < taile ; tab[i] = i++) ;
}

int & Tabint::operator[] (int i)          // OUTOFBOUNDS supposé défini
{
   if (i < 0 || i >= taille) exit(OUTOFBOUNDS) ;
   return *(tab + i) ;
}
```

L'opérateur d'indexation [] est redéfini pour cette classe, l'indice passé est testé pour s'assurer qu'il est dans les limites définies à la construction de l'objet table.

La redéfinition de l'opérateur d'affectation s'écrit ainsi :

```
Tabint & Tabint::operator = (const Tabint & table)
{
    if (&table != this)
    {
        delete[] tab;
        taille = table.taille;
        tab    = new int[taille];
        for(int i = 0; i < taille ; tab[i] = table.tab[i++]) ;
    }
    return *this;
}
```

Vous venez de remarquer avec la déclaration de l'opérateur = l'utilisation de la variable non déclarée *this* de type pointeur. Il s'agit du pointeur sur l'objet courant (celui précisément dont on exécute la fonction membre operator=). Ce pointeur n'a pas a être déclaré et c'est un mot réservé du langage C++.

L'opérateur d'affectation *operator=()* possède un ensemble de caractéristiques propres qui en font un cas particulier à lui tout seul :

- Il ne peut jamais être hérité.
- Il se comporte comme une destruction de la variable qu'on désire affecter (avec rendu de la mémoire dynamique si besoin) puis construction d'un nouvel objet par copie.
- Il possède une valeur de retour. Les constructeurs retournent this de manière implicite sans que le programmeur ne puisse le contrôler, par contre pour l'opérateur d'affectation, il est conseillé de retourner ***this**. Ceci autorise l'affectation

séquentielle de structures entières, ce qui n'est pas possible avec le langage C. (*struc_a = struc_b = struc_c;*)

Depuis la version 2.0 du C++ ANSI, il est possible de surcharger l'opérateur "," utilisé pour l'évaluation sequentielle de gauche à droite; l'opérateur "->" de manière à implémenter ce que l'on appelle des pointeurs "intelligents", il est également possible de surcharger l'opérateur "->*" pour l'utilisation des pointeurs sur membres.

5.5 La surcharge des opérateurs new et delete

Les opérateurs **new** et **delete** peuvent être surchargés localement à chaque classe et devenir des fonctions membres. Ceci s'effectue sans perdre les opérateurs **new** et **delete** globaux.

L'opérateur **new** doit accepter un argument de type **long** et retourner un pointeur sur **void**. De même, l'opérateur **delete** doit accepter un pointeur sur **void** et ne rien retourner. Notez dans l'exemple suivant l'emploi de l'opérateur de résolution de portée pour accéder aux opérateurs **new** et **delete** globaux.

```
classDemoOpNewDelete {
     static   int Nbinstance ;
  public:
     void *operator new (long taille)
           {
              nbinstance++ ;
              return ::new(taille) ;
           }
     void  operator delete (void * ptr)
           {
              nbinstance-- ;
              ::delete( ptr ) ;
           }
  } ;

int Demo::Nbinstance = 0;  // Initialisation de l'attribut statique
```

Il est également possible de définir l'opérateur **delete** avec un deuxième argument indiquant la taille de l'objet à détruire. Cette taille peut être automatiquement calculée par le compilateur si le programme respecte les règles données à ce sujet dans [ELLIS 90].

L'opérateur **new** global peut servir à créer des tableaux éventuellement multi-dimentionnels alors qu'un opérateur local ne peut le faire. Toutes les dimensions doivent être spécifiées.

```
ptrtabi =  new int[20];
cube    =  ::new float[100][100][100];       // les :: sont optionnels ici
editeur =  ::new char[1000][80];
```

A partir de la version 2.1 du C++ ANSI, il n'est plus nécessaire de spécifier la dimension d'un tableau dynamique détruit par **delete[]** ptr_de_tableau.

```
ptrtab  =  new double[10000];
delete[]ptrtab;              // plus besoin de: delete[10000] ptrtab
```

Par contre avec un new local à une classe ou global, il est possible de donner une ou des valeurs d'initialisation à ce qui est pointé.

```
ptrf =  new float(4.5);       // ptrf pointe sur 4.5
pobj =  new rect(1,1,30,150); // pointeur sur un rectangle initialisé
```


Il est possible en C++ de décrire le corps d'une fonction qui sera appelée en cas d'echec de l'opérateur **new** (lorsqu'il renvoie NULL). Pour indiquer à l'opérateur **new** que l'on souhaite gérer les erreurs éventuelles, il faut effectuer un appel à **set_new_handler** en passant l'adresse de la fonction qui récupère l'erreur. Le prototype de **set_new_handler** est le suivant :

```
void (*set_new_handler (void (*)()) ) ();
```

Cette fonction accepte en paramètre un pointeur sur une fonction qui ne retourne rien et qui n'a pas d'argument. De même **set_new_handler** est déclarée comme retournant un pointeur de fonction (le type de retour de la fonction pointée étant non défini : **void**). En fait **set_new_handler** retourne l'adresse de l'ancienne fonction gestionnaire du new (interne et codée par le compilateur) et fait pointer **new_handler** (pointeur interne) sur la fonction définie.

Une autre particularité de l'opérateur new est de pouvoir affecter une adresse spécifique à un pointeur. Dans ce cas lorsque l'on souhaite disposer à nouveau de l'espace mémoire alloué, le destructeur doit obligatoirement être appelé explicitement soit par un delete ou par un appel qualifié.

```
char buffer[sizeof(classe)];
...
ptr_obj = new(&buffer) classe;
...
delete ptr_obj;  // ou autre appel possible par : ptr_obj->classe::~classe();
```

5.6 Les fonctions amies

Stroustrup précise que l'exemple que nous vous avons donné pour la surcharge de l'opérateur d'addition n'est pas efficace. Ce n'est pas au niveau de l'appel de la fonction *get_taille()* que se situe la perte de performance car c'est une fonction de type inline implicite. Par contre l'appel de l'opérateur surchargé [] effectue pour chaque appel le test de l'indice passé en argument. En effet les données membres de table sont protégées et il semble difficile de passer outre les protections d'accès à la classe table en se passant de l'interface des fonctions membres. C'est pourquoi le C++ propose des fonctions amies d'une ou plusieurs classes. Ces fonctions amies quoique non membres de ce ou de ces classes, peuvent accéder aux attributs et aux méthodes privées ou protégées de la ou des classes concernées.

En spécifiant le nouvel opérateur + sur les tableaux comme fonction amie de la classe table, le code pourra être plus efficace en adressant directement les éléments du tableau. La somme de deux tableaux ne sera jamais une fonction membre de la classe table car elle n'appartient a aucun objet de cette classe, et c'est le cas des opérateurs diadiques. En effet elle récupère deux tableaux en paramètre et en restitue un troisième différent des deux précédents. On utilise fréquemment des fonctions amies pour ce type d'opérateurs, on peut néanmoins s'en passer si l'on souhaite ne pas passer outre le principe d'encapsulation. De même, pour ce type d'application il serait dommage de passer les paramètres par valeur, c'est donc le passage de paramètre par références constantes qui est choisi.

```
Tabint operator +(const Tabint & tab1, const Tabint & tab2)
{
   Tabint ttmp(tab1.taille);

   if (tab1.taille != tab2.taille)
   {
      exit(-1);
   }

   for(int i = 0;
       i < ttmp.taille ;
       ttmp.tab[i] = tab1.tab[i] + tab2.tab[i++]);

   return ttmp;                    // l'objet temporaire peut être
}                                  // utilise pour le chaînage des opérateurs
```

Après toutes ces déclarations de fonctions l'utilisateur peut enfin écrire le code suivant.

```
Tabint t1,t2 ;

Tabint t3 = t1 + t2 ;      // Addition surchargée et
Tabint t4 = t3 ;           // Affectation surchargée
```

Cet exemple de codage de fonction friend est un bon compromis de performance en terme de temps d'exécution et d'occupation mémoire. Un des articles de Andrew Koenig aborde les fonctions **friend** en détail [KOENIG 89e].

Regardons maintenant comment on peut déclarer un fonction f1 membre d'une classe c1 comme amie de la classe table :

```
class Tabint {
  protected :
    ...
  public :
    friend   void  c1::f1() ;  // f1 membre de c1 amie de table
    ...
  }
```

On peut enfin déclarer une classe entière comme amie d'une autre classe par la déclaration suivante :

```
class Tabint {
  protected :
    ...
  public :
    friendclass     c1;
    ...
  }
```

Après cette déclaration toutes les méthodes de classe c1 peuvent accéder à tous les attributs de la classe table.

5.7 La généricité

Il existe au moins trois façons de la programmer le concept de généricité en C++, l'une d'elle est très souple. La première solution consiste à utiliser l'héritage pour émuler la généricité [MEYER 88].

Etudions maintenant la deuxième solution. Personne n'oserait dire aujourd'hui que le langage C ne possède pas de possibilité d'allocation dynamique, pourtant il faut inclure le fichier d'entête *<alloc.h>* qui donne accès au code normalisé ANSI des fonctions *malloc()* , *free()* , etc... Il en est de même programmer de manière générique en C, il faut inclure un

fichier d'entête très simple *<generic.h>* qui contient les deux macro instructions données ci-après.

```
#ifndef GENERIC_H
#define GENERIC_H

#define name2(n1, n2)   n1 ## n2                  // ## operateur de
#define declare(a,type) a ## declare(type)        // concaténation

#endif  GENERIC_H
```

Pour donner un exemple, lorsque l'on écrit *name2(float,gentab)* la macro instruction développe *floatgentab*. De même *declare(gentab, float)* est développé *gentabdeclare(type)*.

Le but de cette solution est de pouvoir déclarer une classe d'objet manipulant un type générique de la manière suivante :

```
generic_classe(type)    objet ;
```

Pour les fondations de la généricité, la classe d'objet de base doit manipuler des paramètres basés sur le type *void* de manière à pouvoir manipuler toutes les données et les fonctions membres pour n'importe quel type donné. Ceci a été fait de manière intentionnelle sur la classe table. Pour manipuler des tables de n'importe quel type en respectant la forme spécifiée par [STROUSTRUP 86], il faut écrire les lignes de code suivantes :

```
#define generic_table(type) name2(type,generictable)
#define generic_tabledeclare(type)                                    \
  struct  generic_table(type) : table {                               \
    type& operator[] (int i)                                          \
    {return *((type *) table::operator[](i)) }                        \
    type* operator&() { return (type *) table::operator&() }          \
    int      get_taille(){ return taille; }                           \

    void  operator=(table & reftab) { table::operator=(reftab) }
    }     // Pas de point virgule ici il sera rajouté lors de la     \
          // déclaration                                             \
```

La déclaration de ce type de macro est très efficace mais également très lourde. Le caractère \ indique que la définition de la macro n'est pas terminée et que la ligne suivante en fait partie. Cette macro développe une structure dérivée qui garantit que toutes les instances de cette structure partagent le même code.

Stroustrup précise que cette technique ne peut être employée que pour créer des classes d'objets de tailles identiques ou plus petites que celle de la classe de base. Avec les techniques de développement des macros les noms de types composés comme *type ** ne peuvent pas être directement traitées, il est alors conseillé d'utiliser *typedef* [STROUSTRUP 86].

Les opérations pour émuler la généricité sont complexes, c'est pourquoi Bjarne Stroustrup a proposé les types paramétrés sous forme de gabarit ("template"), il s'agit de la troisième solution disponible et maintenant normalisée. Ces gabarits sont des classes d'objet auxquelles on passe les types en paramètres; l'utilisation de ces gabarits est simple. On parle également de classes paramétrées pour évoquer ce concept (Voir Eiffel [MEYER 88]), et il s'agit en fait de métaclasses paramétrées, car ces métaclasses décrivent bien des classes, paramétrées par un type ou une autre classe C++, ce sont ces dernières qui produisent les objets.

```
template <class T> class Vecteur {
   T  *  v;
   int   taille;
public:
   Vecteur(int);
   T&    operator[](int) ;
   T&    elementAt(int i) { return v[i]; }
   }

Vecteur<complex> v(10) ;  // Définit un vecteur de 10 complexes
Vecteur<char>    v(80) ;  // et un de 80 caractères

template <class T> T & Vecteur<T>::operator[](int i)
{
   if (i < 0 || taille <= i) error(OUTOFBOUNDS);
   return v[i];
}

typedef Vecteur<char *> tptchar;

tptchar table(50);

table[3] = table.elementAt(4);
```

A ces notions de gabarits s'ajoutent les notions d'équivalences entre gabarits ainsi que la notion de fonctions gabarits pour une famille de classes. Le lecteur intéressé par la généricité en C++ se reportera à [ELLIS 90].

5.8 Les bibliothèques des flux

Il convient de discuter de la bibliothèque des flux proposée avec le C++. Le langage C est lié à une bibliothèque d'entrée sortie standard *<stdio.h>* cette bibliothèque est conservée par le C++, mais l'utilisation des bibliothèques de flux *<iostream>* ou *<fstream>* est très conseillée. Ces bibliothèques proposent une syntaxe plus simple basée sur la surcharge des opérateurs de décalage >> et << (respectivement à droite et à gauche). Les mêmes opérateurs peuvent être utilisés pour tous les types de données standard ce qui est beaucoup plus souple que l'utilisation des formats de *printf* et *scanf*. Les opérateurs >> et << restent surchargeables pour les types définis par l'utilisateur. Le C++ ANSI propose quatre canaux de flux, dont nous donnons deux rapides exemple d'utilisation :

- l'entrée standard cin,
- la sortie standard cout,
- la sortie des erreurs standard cerr,
- la sortie des erreurs sur buffer clog.

```
// --------- Exemple 1 : Entrées, sorties clavier écran -------- //

#include <iostream>

int main(int, char**)
{
   int      x = 0;

   cout  < <"Entrer x :" ;
   cin   > >x ;
   if (x > 255) cerr < < "x est trop grand" ;
   cout  < <" x * x =" < < x * x << endl; // endl pour passer à la ligne
   return 0;
}
```



```
// --------- Exemple 2 :  -------- //

#include <iostream>                    // .h inutiles avec un gcc ANSI 3.0
#include <fstream>                     // Flux pour la gestion des fichiers
#include <string>                      // classes chaine de la Standard
                                       // Template Library

const int TAILLE_MAX = 10;

int main(int, char**)                  // Lecture d'un fichier d'entier
{                                      // la première valeur trouvée
  ifstream fichier;                    // dans le fichier contient le
  int      taille;                     // nombre d'éléments

  do                                   // Saisie du nom du fichier
  {
     string nom;
     cout << "Nom du fichier de lecture: ";
     cin >> nom;
     fichier.open (nom.c_str(), ios::in);
  }
  while (fichier.fail());

  fichier >> taille;          // lecture du nombre d'entiers
  if (taille > TAILLE_MAX) taille = TAILLE_MAX;

  int *tableau = new int[taille];  // allocation du tableau

  if (!tableau) exit (1);


  // Lecture du tableau
  for (int i=0 ; i < taille ; i++) fichier >> tableau[i];

  fichier.close();            // fermeture du fichier


  cout << "Affichage pour vérifier les valeurs lues " << endl;

  for (int i=0 ; i < taille ; i++)
  {
     cout << "tableau[" << i << "] = " << tableau[i] << endl;
  }

  delete[] tableau;           // Restauration de l'espace alloué

  return 0;
}
```

La surchage d'un opérateur de flux est également pratique (inclure <iostream>). Nous donnons le code ci-dessous à titre d'exemple :

```
ostream & operator << (ostream & o, Tabint & table)
{

   for(int i = 0; i < table.lg ; i++)  o << table.tab[i] << " ";

   return o;
}
```

5.9 Les déclarations "inline" explicites

Les fonctions **inline** ont été vues lors de leur utilisation implicite dans certaines fonctions membre. Ce nouveau type de fonction vient du fait que l'on s'inquiète du coût des appels des nombreuses petites fonctions d'un programme. Si le besoin de performance se fait sentir, il peut être intéressant de développer dans le corps du programme des fonctions de tailles réduites (1 à 4 lignes). C'est habituellement le rôle des macros *#define* du langage C. Il

est conseillé d'éviter de déclarer en **inline** les fonctions ayant des boucles. Voici un exemple d'utilisation.

```
inline mulpar8(int val) { return (val << 3) ;}
...
Code source                Code réel

mulpar8 (j) ;              j << 3 ;     //   Plus d'appel de fonction
i += j;                    i += j ;
```

5.10 Les membres statiques

Les membres statiques appartiennent aux classes et non aux instances. Qu'il s'agisse de données membres ou de fonctions membres, on rejoint ici le concept Smalltalk préconisant que les classes sont des objets. En C++ les classes ne sont pas des objets, mais pour des raisons évidentes d'analyse et de conception (les fonctions et les données globales polluent l'approche objet) les notions de variables de classe et de méthodes de classe ont également été introduites. Les membres statiques se reconnaissent à l'utilisation du mot clé **static** lors de leur déclaration dans une classe. Les fonctions membres statiques ne peuvent agir que sur les données membres statiques (où sur des variables globales). Il est très important d'initialiser les membres statiques (en dehors des classes, cf. exemple ci-dessous).

```
classe UneClasse {
     static   intNbobj ;
     ...
  public:
     static   void  inc_nbobj() { nbobj++ }
  } ;

int UneClasse::Nbobj = 0;
```

5.11 Génération automatique de l'opérateur d'affectation et du constructeur de copie

Les constructeurs de copie X(X&) (prononcez X de X ref) utilisés pour effectuer des copies lors de passage d'objets par valeur, de même les opérateurs d'affectation peuvent être créés par le compilateur si le programmeur ne les a pas définis. La copie et l'initialisation s'effectuent automatiquement sur tous les membres d'une classe de manière "intelligente". Les constructeurs de copie ainsi que les opérateurs d'affectation sont nécessaires lorsque les objets manipulés ont un constructeur et un destructeur manipulant de la mémoire dynamique. Le code ci-dessous présente le constructeur de recopie de la classe Tabint que nous utilisions plus haut. Certains compilateurs exigent que ces constructeurs ne soient pas déportés (ils sont alors inline implicite).

```
Tabint(const Tabint & table)
{
   taille     = table.taille;
   tab = new int[taille];
   for(int i = 0; i < taille ; tab[i] = table.tab[i++]) ;
}
```

5.12 Les tâches

Dans les dernières caractéristiques du langage C++, il faut inclure la gestion de tâches avec la bibliothèque des tâches AT&T Bell d'entête *<task.h>* décrite dans [HANSEL 90] et qui se base sur la concurrence d'UNIX. Des mécanismes de concurrences différents ont également été proposés aux laboratoires Bell de Murray Hill par [GEHANI 88]. Des exemples significatifs de la construction de classes avec différents degrés de protection sont proposés

dans [GORLEN 87] qui fournit une bibliothèque de classes C++ très complète appelée OOPS pour "Object Oriented Program Support". On y trouve un ensemble de classes simples dont la plupart sont inspirées du Smalltalk-80. En outre on peut citer les classes chaînes de caractères, date, temps, listes chaînées simples, tables d'adressage dispersé etc... Enfin on y trouve les classes "process", "Scheduler", "Semaphore", "File" qui fournissent des possibilités de multiprogrammation avec des coroutines.

5.13 Les exceptions

La gestion des exceptions a été intégrée au mois d'avril 1990 et présenté dans les actes de la conférence USENIX C++ de San Francisco 1990 ansi que dans [ELLIS 90] [KOENIG 91]. Abordons brièvement le mécanisme des exceptions. C'est le mot clé **throw** est qui est utilisé pour signaler une exception, il prend un seul opérande, un objet de type quelconque. Il est préférable que les exceptions dérivent de la classe exception de la STL (Standard Template Library).

```
class C
{
    // détails Omis
  public :
    C (const char *);
    const char * diagnostic ();
};
```

```
void allocate_ressource ( ) throw ()
{
   if (allocation ( ) == FAIL)
   {
      throw C ("Allocation fails!");
   }
}
```

Lorsque **throw** est exécuté à l'intérieur d'un bloc **try**, le contrôle est passé au plus proche gestionnaire d'exception qui peut traiter cette exception (bloc **catch**). L'opérande est passée au gestionnaire de l'exception (dans notre cas un objet de class C). Les gestionnaires d'exception sont spécifiés avec comme suit :

```
void allocate_and_report_results ( )
{
   [ try                                    ]
①     {                                      " bloc try "
         allocate ressource ( );
      }
②  [ catch (C & caught_Xept)
      {
         cout << caught_Xept.what( )<< '\n';
      }

}
```

Le gestionnaire est spécifié dans le bloc catch.

Si aucune exception n'est levée dans le bloc try alors le gérant va être " sauté " et le programme continue.

Pour qu'un gestionnaire puisse être activé, il faut que l'objet levé soit de même type que celui spécifié dans le **catch**, ou d'un type dérivé. Si le **catch** ne correspond pas, on dépile le contexte d'exécution jusqu'à trouver une clause catch adaptée.

```
main( )
{
  try
  {
    // tout le programme
  }
  catch (...)
  {
    // toute exception non
    // atrapée le sera ici
  }
}
```

Un appel a throw sans argument resignale (ou relève) l'exception courante :

```
try
{
  // qqch
}
catch (X& ex)
{
  if (can-handle (ex))
    handle (ex);
  else
    throw; // equiv à throw ex;
}
```

Si il n'existe pas de clause **catch** adaptée, la fonction **terminate** est appelée, par défaut elle arrête le programme. Pour attraper toutes les exceptions non traitées, il faut procéder de la comme ci-dessus avec un **catch(...)**. Il existe aussi une fonction **unexpected** qui peut être appelée si le type d'exception n'est pas attendu ou pas prévu par le prototype du throw qui complète le prototype d'une fonction. Il est possible de préciser ses propres fonctions **terminate** et **unexpected** à l'aide des fonctions : **set_terminate** et **set_unexpected**.

5.14 Les pointeurs sur membres

Un pointeur sur un membre d'une classe doit toujours être associé à un objet. Lors de sa définition, il faut aussi spécifier le type exact sur lequel il pointe sans oublier sa classe.

```
class UneClasse {
   ...
  public:
     int          i;
     ...
     double    fonction(float f) { return f - i; }
     void      affiche();
     void      efface();
  }

int      UneClasse::*ptri ;                  //    Pointeur sur membre entier

double   (UneClasse::*ptrfn)(float) ;        //    Pointeur sur fonction membre

ptri =   & UneClasse::i ;                    //    Init de l'adresse des

ptrfn=   & UneClasse::fonction ;             //    pointeurs sur membres

// Exemple d'utilisation avec une instance puis avec
// un pointeur sur une instance

UneClasse      unObjet;

unObjet.*ptri  = 1 ;                         //    .* déréference le pointeur

double  d1     = (unObjet.*ptrfn)(2.71828) ;

UneClasse *pObj= new(UneClasse) ;
pObj->*ptri    = 2;

double d2      = (pObj->*ptrfn)(3.14159) ;
```

Supposons maintenant que nous disposons d'un ensemble de classe collection d'objet (liste, table, arbre) dérivée d'une classe abstraite *collection* possédant les fonctions virtuelles pures suivantes pour effectuer un parcours de la collection :

fin : Indique la fin du parcours.
courant : Récupère l'élément courant.
suivant : passe au suivant.


```
class Collection {
   ...
  public:
     virtual  int          fin() = 0 ;
     virtual  UneClasse *  courant() = 0 ;
     virtual  void         suivant() = 0 ;
     ...
  } ;
```

Nous allons maintenant montrer comment il est possible de passer un pointeur sur une fonction membre à une fonction de parcours d'une collection (qu'elle que soit la classe de la collection) et ceci afin de paramétrer la fonction à appliquer à chaque parcours.

```
void parcours(Collection & c, void (UneClasse::*ptrfn)())
{
   while (!c.fin()) {
      (c.courant()->*ptrfn)() ;      //    Applique la fonction à l'objet
      c.suivant() ;
      }
}
```

On peut ensuite utiliser le parcours sur différentes structures avec différentes fonctions à exécuter :

```
parcours(arbre, & UneClasse::affiche) ;
parcours(table, & UneClasse::efface) ;
parcours(liste, & UneClasse::affiche) ;
```

Cet exemple de code est hautement réutilisable, de plus, il est possible de passer en paramètre des fonctions virtuelles. Cependant il faut admettre la lisibilité obtenue est critiquable. De plus, la mise en œuvre de ce code demande une bonne maîtrise du langage qui possède un nombre considérable de subtilités qui n'ont pu être exposées ici.