Chapter 1

Spécificités de C++ par rapport à C

1.1 Le mot clef `const`

Ce mot clef n'est pas spécifique au Langage C++, il existe dans la version ANSI du Langage C. Une première utilisation "naïve" du mot clef `const` est de définir des constantes. Dans ce cas, on indique explicitement au compilateur lors de la **définition** d'une variable qu'elle ne peut être modifiée. Le compilateur a en charge de vérifier que cette règle n'est pas enfreinte par les programmes. La première utilisation qui peut être faîtes de ce mot clef est la définition de constante symbolique.

```
const double VITESSE_DE_LA_LUMIERE= 3.0e+8;

main()
{
  VITESSE_DE_LA_LUMIERE = 0.0;
}
```

```
const1.cc: In function 'int main()':
const1.cc:6: assignment of read-only variable 'double const VITESSE_DE_LA_LUMIERE'
```

Cette exemple illustre la vérification effectuée par le compilateur, lors de la tentative d'affectation d'une constante. L'avantage de l'utilisation de `const`, pour les constantes symboliques, réside dans la définition du type en plus de la définition de la valeur. Sur l'exemple précédent, la définition de la constante `VITESSE_DE_LA_LUMIERE` spécifie, outre la valeur de cette constante, son type. Cette précision supplémentaire est importante pour le langage C++ du fait de l'existence d'un mécanisme de *surcharge* (cf **??**). En effet ce mécanisme repose sur le nombre et le type des paramètres définis par la signature de la fonction surchargée.

Remarque 1 *Pour des raisons de convention d'écriture, nous conserverons l'usage des majuscules pour les constantes symboliques.*

L'utilisation des constantes en Langage C++ ne souffre pas de certaines des restrictions inhérentes au Langage C. Considérons le programme suivant:

```
const int SIZE = 23;

int tab[SIZE];

int main()
{
  return 0;
}
```

La compilation de ce programme par `gcc -ansi -Wall...` produit le message d'erreur suivant:

```
const2.c:3: variable-size type declared outside of any function
```

Alors que la compilation avec `g++ -ansi -Wall` ne produit aucune erreur. Cette possibilité est due à des stratégies différentes d'initialisation des variables statiques.

Nous allons maintenant illustrer les diverses utilisations possibles du mot `const`.

```
char *pt = "hello";
// Le pointeur peut etre deplace pt = malloc(...);
// Le contenu pointe peut etre modifie pt[0] = 'c';

const char *pt="hello";
// Le pointeur peut etre deplace pt = pt + 1;
// Le contenu pointe est constant pt[0] = 'c';
// XX.cc:?: assignment of read-only location

char * const pt="hello";
// Le pointeur ne peut etre deplace pt++;
// XX.cc:?:increment of read-only
//          variable 'char * const pt'
// Le contenu peut etre modifie pt[0] = 'c';

const char * const pt="hello";
// Le pointeur ne peut etre deplace;
// Le contenu ne peut etre modifie;
```

L'intérêt du mot clef `const` réside plus dans sa signification conceptuelle que dans le contrôle syntaxique. En effet, il est toujours possible de contourner la vérification syntaxique en utilisant une conversion ou un pointeur. Par exemple, le programme suivant permet de changer la valeur d'une chaîne constante.

```
const char * pt = "hello";

int main()
{
    ((char *) pt)[0] = 'c';
    printf(" %s", pt);
    // Le result produit l'affichage de  cello
    return 0;
}
```

1.1.1 Le mot clef `inline`

.

Une possibilité proposée par le langage C++ est de remplacer l'appel de fonction par le code de la fonction elle-même, on parle parfois de développement en-ligne. Du point de vue syntaxique, une fonction candidate au développement en ligne doit être précédée du mot clef `inline` lors de sa définition et non lors de sa déclaration.

```
#include <stdlib.h>
#include <stdio.h>

inline int max(int p1, int p2)
{
    return p1 > p2? p1:p2;
}

main()
{
    int i = 2;
    int j = 3;
    printf("Le max de %d et de %d est %d \n", i, j,max(i, j));
}
```

Le fait de définir une fonction `inline` autorise le compilateur à substituer le code de la fonction à l'appel de fonction. Cette substitution n'est pas certaine car elle dépend de la complexité du code ainsi que du

compilateur. Sur l'exemple précédent, le compilateur essayera de remplacer l'appel à la fonction `max` par son code.

La substitution de code s'effectue durant la phase de compilation et non pas pendant l'édition de lien. De ce fait, une fonction définie dans le fichier ".c" ne peut être insérée que dans ce fichier et que dans la portion de code suivant sa déclaration. Les fonctions récursives ont une probabilité quasi nulle d'être substituées.

Deux problèmes se posent lors de l'utilisation des fonctions `inline`:

1) Un problème d'efficacité, en effet la substitution de code fait croître le code exécutable, et donc des défauts de pages peuvent apparaître. Ces phénomènes risquent de pénaliser les performances d'exécution.

2) Un problème de conception. Si une fonction doit être "inliner" dans la totalité d'une application, elle doit être définie dans le ".h" du module. Dans ce cas, le principe de séparation entre *interface* et *implémentation* est enfreint, du moins au niveau du code objet[1].

L'utilisation des fonctions "inlines" est donc délicates. La décision de mettre des fonctions en lignes ne doît être prises qu'en phase finale de développement,

1) lorsque l'implémentation du module est figée.

2) après une étude détaillée du code et seulement pour les fonctions intensivement utilisées.

Une exception à ces règles est autorisée pour les fonctions ayant un statut de pseudo-fonctions. L'utilisation de fonctions en ligne est plus précise que les pseudo-fonctions car le type des paramètres est défini. De ce fait, le compilateur exerce d'avantage de contrôle. De plus certains problèmes liés à l'itération des effet de bord lors de l'appel sont supprimés.

```
#include <stdlib.h>
#include <stdio.h>

#define CARRE(_x) (_x) * (_x)

inline int carre(int i)
{
    return i * i;
}

main()
{
    int i = 2;
    printf(" Par fonction %d \n", carre(++i));
    // Le resultat affiche --> Par fonction 9
    i = 2;
    printf(" Par pseudo-fonction %d \n", CARRE(++i));
    // Le resultat affiche --> Par pseudo-fonction 12
}
```

On peut également être plus laxiste dans l'application de ces règles dans les cas où la recompilation des clients ne pose pas de problème. Mais même dans une telle situation l'argument de l'efficacité reste valable.

1.1.2 La référence

Il existe en Langage C++, un mécanisme permettant de référencer une *variable* (au sens large du terme). Une **variable référence** n'est ni un pointeur ni une variable à proprement parler. Elle doît être considérée comme une manière de nommer ou identifier différemment une entité existante. La déclaration d'une variable référence s'obtient en utilisant le symbole `&`. Par exemple, la déclaration

```
int &ref = reelle;
```

[1] Ce ne sera pas la dernière fois avec le langage C++.

implique la déclaration préalable de la variable `reelle`. Maintenant, la variable référence `ref` est un autre moyen de manipuler la variable `reelle`. Le terme "*un autre moyen*" signifie qu'il est possible d'identifier l'entité `reelle` en utilisant le nom `ref` (`ref` est un synonyme de `reelle`). Durant toute sa durée de vie la variable `ref` ne pourra être utilisée que pour identifier `reelle`.

La définition d'une variable référence implique nécessairement son initialisation, puisqu'elle nomme autrement une entité existante.

```
int &ref;
//X.cc:?: 'ref' declared as reference but not initialized
```

Par contre, ce qui est incorrect pour les définitions est autorisé pour les déclarations. L'exemple suivant est tout à fait licite,

```
extern int &ref;
```

il implique simplement que la variable référence `ref` devra être définie ultérieurement.

Remarque 2 *La référence est implémentée en utilisant les pointeurs. Le programme suivant permet de vérifier que les variables* `ref` *et* `reelle` *identifient une même entité en pointant sur le même espace mémoire.*

```
struct point
{
  int x;
  int y;
};

int main()
{
  struct point p1 = { 0, 1};
  struct point &p2;

  if(&p1 == &p2)
    printf(" Vrai \n");
  else
     printf(" Faux \n");

  return 0;
}
```

L'utilisation de la référence pose le problème de l'interprétation de l'affectation. Considérons le programme suivant:

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    int reelle = 0;
    int tempo = 3;
    int &ref = reelle;
    ref = tempo;

    printf("la valeur de reelle est %d \n", reelle);
    // la valeur de reelle est 3
    printf("la valeur de ref est %d \n", ref);
    // la valeur de ref     est 3
    return 0;

}
```

L'instruction `ref = tempo` ne demande pas à changer la référence de `ref` en lui demandant d'identifier dorénavant la variable `tempo`. Elle va déclencher l'appel à l'opérateur d'affectation dont le prototype est `int`

`& operator=(const int &)`. Le résultat de cette opération affecte la valeur 3 à la variable `ref`. Comme la variable `ref` identifie le même espace physique que la variable `reelle`, l'espace physique contient maintenant la valeur 3 et donc la valeur de `reelle` est 3. **Lorsqu'une référence est définie, une liaison insécable est définitivement établie entre une variable référence et l'entité qu'elle référence**.

Le mécanisme de référence trouve une réelle justification lors de son utilisation pour le passage de paramètres lors de l'appel de fonction. Le passage de paramètres utilisant les variables références, pour ce qui esr des modifications du contexte de l'appelant par l'appelé, offre les mêmes possibilités que les passage de paramètres utilisant les pointeurs .

Le reproche qui peut être fait à l'utilisation de la référence est une perte de lisibilité. En effet, lors de l'appel de fonction il est impossible, du point de vue syntaxique, de différencier un passage par valeur d'un passage par référence. Par exemple, si `i` est un entier, après l'appel `f(i)` on ne peut savoir si la valeur de `i` est susceptible d'être modifiée par l'appel de fonction. En effet, le résultat dépend en grande partie du *prototype* de la fonction `f`

1) `void f(int)`

2) `void f(int &)`

3) `void f(const int &)`

La sémantique des deux déclarations suivantes est identiques.

```
<type> &nom_var = variable;
<type> * const nom_var = variable;
```

Cette équivalence est établie car la référence est implémentée à l'aide de pointeur, et parce que le compilateur n'interprète pas l'affectation d'une variable référence comme un changement de référence. En utilisant les références et les constantes il existe 7 façons de définir le passage de paramètre.

1) `f(type )`

2) `f(const type * const)`

3) `f(const type &)`

4) `f(const type *)`

5) `f(type *)`

6) `f(type &)`

7) `f(type * const)`

Nous allons préciser certaines conventions de prototypage des fonctions afin d'augmenter la lisibilité des programmes. Les prototypes de 1 à 4 ont une sémantique de passage de paramètres par valeur. En effet quelque soit le code de la fonction `f` les modifications effectuées sur le paramètre n'affecteront pas le programme appelant. Afin de conserver, la même écriture `f(i)` pour l'appel de fonction nous n'utiliseront que les notations 1 et 3 pour le passage de paramètres par valeur. L'écriture numéro 3 est utilisée pour des raisons d'efficacité, en effet on ne recopie pas dans la pile la valeur du paramètre, on utilise directement son adresse et on contrôle qu'il n'est pas modifié. Pour les objets complexes le gain d'efficacité est très important. Les appels 5 et 6 correspondent à l'appel par référence, nous ne conserverons que l'appel 5 pour des raisons de lisibilités.

Il est toujours possible de faire retourner une référence par une fonction par exemple:

```
#include <stdio.h>

int globale = 3;

int& exemple()
{
  return globale;
}
```

```
int main()
{
  exemple() = 9;
  printf("valeur de la variable globale %d \n", globale);
}
```

Sur cette exemple, la valeur affichée est 9. Que ce soit pour le passage de paramètres ou pour le retour de fonction, l'utilisation de la référence bloque le passage par valeur. C'est à dire que seule l'adresse est utilisée dans la pile, il n'y a pas de recopie explicite.

Une dernière précision concerne la référence des

1) Valeur littérales;

2) Valeur résultant d'une conversion;

3) Valeur résultant d'un appel de fonction;

Pour référencer ces entités il est nécessaire d'utiliser des références sur des variables constantes.

```
#include <stdio.h>
#include <math.h>


int main()
{
  int tmp = 4;

  const int &i = 3;
  const double &rac_2 = sqrt(2.0);
  const double &d = tmp;
}
```

1.2 Le mécanisme de surcharge des fonctions dans le langage C++

La surcharge de fonctions en langage C++ est la possibilité, dans un contexte donné, d'employer un même nom pour définir un ensemble de fonctions. La surcharge est licite si la signature des fonctions diffère soit par le nombre d'arguments, soit par le type des arguments. **Attention,** le type de retour de la fonction n'est pas un élément distinctif de la surcharge de fonction.

```
#include <stdio.h>

void affiche(int v)
{
    printf("un entier %d \n", v);
}

void affiche(double v)
{
    printf("un reel %lf \n", v);
}

void affiche(char *v)
{
    printf("une chaine %s \n", v);
}

int main(int argc, char **argv)
{
    int entier = 3;
    double reel = 3.0;
```

```
    affiche("Une chaine");
    affiche(3);
    affiche(3.0);
    return 1;
}
```

Le langage C utilise ce concept dans ces mécanisme internes, mais il n'autorise pas le programmeur à créer ses propres surcharges de fonctions. Par exemple, les fonction évoquées par les expressions suivantes

```
float x = 1.0 / 2.3;
int  x = 1 / 2;
```

sont différentes. La première expression évoque le code de l'opérateur de division dont le prototype pourrait être

```
float operator/(const float &, const float &);
```

alors que la seconde évoque le code d'un autre opérateur dont le prototype serait[2]

```
int operator/(const int &, const int &);
```

Ces deux opérateurs ont la même sémantique mais s'appliquent sur des types différents. Il semble naturel que la sélection de la fonction adéquate incombe au compilateur et non au programmeur.

La surcharge de fonction accroît la lisibilité d'un programme en réduisant la complexité lexicale. Considérons par exemple, un traitement consistant à afficher sur un terminal graphique diverses formes géométriques (carré, cercle, ellipse, polygone). Une façon de faire en langage C, consisterait à définir les fonctions suivantes:

1) `void affiche_carre(carre)`

2) `void affiche_cercle(cercle)`

3) `void affiche_ellipse(ellipse)`

Cette longue énumération de fonctions sera à la longue légèrement fastidieuse, voire rébarbative.

En langage C++, il est possible de donner le même nom à des fonctions tant que leur signature est différente, soit par le type des arguments soit par le nombre d'arguments. En aucun cas, la distinction s'effectue sur le type de retour de la fonction. Sur l'exemple précédent, la surcharge est illustrée en langage C++ par la déclaration d'un certains nombre de fonctions portant le même nom, mais dont la signature diffère.

1) `void affiche(carre)`

2) `void affiche(cercle)`

3) `void affiche(ellipse)`

Deux phases sont à distinguer dans l'emploi des fonctions surchargées:

1) La première phase concerne la possibilité de déclarer une fonction surchargée;

2) La seconde phase concerne la sélection d'une fonction surchargée au moment de son évocation.

Si un même nom de fonction apparaît plus d'une fois dans un programme, les déclarations suivantes seront interprétées:

1) Comme éronnées si les fonctions ne différent que par le type de retour des paramètres.

2) Comme deux instances de fonctions surchargées si elles différent par le type des paramètres dans l'ordre de leur déclaration ou par leur nombre.

La définition de fonction surchargée se base sur la distinction de types. Certaines définitions de type bien que syntaxiquement différentes sont équivalentes pour le compilateur. Nous allons maintenant énumérer certaines configurations autorisant ou non la surcharge (Cf. ARM pp 309-312).

1) Il est impossible de distinguer les types `T`, et `T &`. Dans ce cas une erreur de compilation est produite (pas répercutée par Mr. Gnu dans la version 2.7.2).

[2] Attention, ces définitions ne sont pas légales. Elles n'impliquent en effet que des types de base.

2) Il en est de même les types `const T`, `T`.

3) Par contre, le compilateur distingue les types `T &`, `const T&` ainsi que les types `const T*` et `T*`.

4) Comme le mot clef `typedef` ne crée pas un type distinct mais un synonyme, la surcharge utilisant des `typedef` n'est pas licite.

5) Par contre les types énumérés autorisent la surcharge.

```
enum F { X=1, Y=2, Z=3}
void f(int)
void f(X)
```

6) Les notations tableau `[]` et pointeurs `*` sont équivalentes. Il est à remarquer que dans les tableaux multi-dimensionnels une distinction s'opère à partir de la taille de la deuxième dimension. Le type `type ** var` est différent de `type *var[]`.

7) La distinction entre les types `signed` and `unsigned` permet la surcharge.

Remarque 3 *Il ne faut pas confondre* surcharge *et* masquage de la visibilité.

La deuxième étape concerne la résolution de la sélection d'une fonction surchargée lors de son appel. Trois cas de figure peuvent se produire:

1) Une seule fonction surchargée peut être sélectionnée. Dans ce cas, il existe une unique *adéquation* entre l'appel et la définition d'une fonction.

2) Plusieurs fonctions surchargées peuvent être sélectionnées à la même étape de l'algorithme de résolution. Le compilateur n'a aucune raison objective de sélectionner une fonction candidate particulière. On parle dans ce cas d' *adéquation ambigüe*.

3) Aucune fonction ne peut être déterminée par l'algorithme de résolution. Dans ce cas, on parle d'*inadéquation*

La détermination de l'adéquation se base sur quatre étapes traitées suivant l'ordre de priorité:

1) Une adéquation exacte entre le type de l'argument courant et celui du prototype.

2) Une adéquation par promotion:

3) Une adéquation par conversion standard

4) une adéquation par application d'une conversion définie par utilisateur.

Nous allons maintenant illustrer les différents résultats de la résolution à travers des exemples:

```
void f(unsigned int);
void f(char *);
void f(int );
void f(long);
```

1) **Illustration de l'adéquation**

```
f(1);  // f(int)  r\'esolution \'etape 1;
f(1u); // f(unsigned int) r\'esolution \'etape 1;
char c = '1';
f(c);  // f(int) promotion de char -->int;
```

f("toto"); // f(char *);

2) **Illustration de l'inadéquation**

```
int *ip;
f(ip); // pas d'ad\'equation
\\ b.cc: In function `int main()':
\\ b.cc:9: bad argument 1 for function
\\      `void print(unsigned int)'
\\      (type was int *)
```

3) **Adéquation ambigüe**. Dés qu'une cohercicion est impliquée par le transtypage, la résolution sera ambigüe.

```
unsigned long ul;
print(ul);

// "b.cc", line 14: more than one instance of
                    overloaded function "f"
                    matches the argument list:
            function "f(unsigned int)"
            function "f(int)"
            function "f(long)"
      f(c);
```

Remarque 4 *Dans le cas des fonction à plusieurs arguments, la résolution s'effectue suivant la stratégie suivante.*

Pour chacun des arguments de l'appel on sélectionne les fonctions qui sont les meilleures candidates en fonctions des ordres de priorités déjà évoqués. Dans un second temps, on calcule l'intersection des fonctions sélectionnées pour chaque argument, si cette ensemble est un singleton dans ce cas il y a adéquation, si l'intersection est vide il y a inadéquation, dans les autres cas il y adéquation ambigüe.

Chapter 2

Les classes en C++

2.1 Un exemple de classe en C++

Comme en C, une classe C++ pourra être constituée de deux fichiers,

1) Un fichier représentant la déclaration de la classe (suffixé `.H` ou `.h`).

2) Un fichier représentant la définition de la classe (suffixé `.cc` ou `.CC`).

La principale différence entre une classe C et une classe C++, se situe au niveau des informations présentes dans la déclaration de la classe. En effet, l'**encapsulation** en C++ ne s'obtient pas par masquage d'information, mais par vérification de l'accès à l'information lors de la compilation. Dans ce cas, la déclaration d'une classe contiendra des descriptions concernant seulement l'implémentation de la classe. Ces informations constitueront la partie **privée** de l'interface d'une classe. De ce fait, il n'existe pas en C++ une réelle séparation entre interface et implémentation au niveau du code objet. Ce choix se justifie pour des raisons d'efficacités. En effet, le compilateur à besoin de connaître la taille des instances d'une classe pour les instancier sur la pile. De plus, il est possible de définir dans la déclaration de la classe le code de certaines méthodes de la classe. Ces méthodes pouvant être par la suite "inliner" dans le code.

Le fichier `"point.h"` représente la déclaration de la classe `Point`.

```
class Point
{
 private:
   int _coord_x;
   int _coord_y;
 public:
   Point(int x, int y);
   float distance();
};
```

Les objets de cette classe possèdent deux variables d'instance, `_coord_x, _coord_y`. Ces deux variables appartiennent à la partie privée de la classe, seules les méthodes déclarées dans la classe `Point` pourront accéder à ces variables. Par contre, si un changement s'opère dans la partie privé, tous les clients de cette classe devront être recompilés.

Une seule méthode, `distance`, est déclarée pour cette classe. La fonction `Point(int x, int y)` est appelée un **constructeur** sont rôle est de pouvoir définir la valeur d'un objet point lors de sa création (cf. **??**).

Le fichier `"point.cc"` représente la définition de la classe `Point`. Il doit contenir la définition du constructeur `Point` et de la méthode `distance`.

```
#include <math.h>
#include <point.h>

static inline int carre(int x)
{
  return x*x;
}
```

```
Point::Point(int x, int y)
{
  _coor_x = x;
  _coor_y = y;
}

float Point::distance()
{
  return sqrt(carre(_coor_x) + carre(_coor_y));
}
```

La définition d'un élément `Elem` déclaré dans une classe`X` se fera en utilisant la syntaxe suivante `X::Elem`. Par exemple,la méthode `distance` de la classe `Point` est nommée float Point::distance() pour la définition.

Toute méthode d'une classe. s'exécute dans le contexte de l'objet récepteur du message. En C++, cet objet est nommé `this`. Le code de la méthode `distance` pourrait aussi s'écrire.

```
float Point::distance()
{
  return sqrt(carre(this→_coor_x) + carre(this→_coor_y));
}
```

Dans ce cas, le type de `this` est `Point const *this`. Ce prototype interdit de changer l'adresse de l'objet récepteur.

2.2 Structure générale d'une classe C++

La déclaration d'une classe pourra comporter les éléments suivants:

```
class <nom_classe>
{
    Definition des relations d'amities

    Definition de nouveau type
    y compris des classe locales;

    Variables de Classes;
    Fonction  de Classes;

    Constructeur;
    Destructeur;

    Methodes;

    Variables d'instances;
};
```

Les **relations d'amitiés** permettent à des classes, fonctions, méthodes déclarés à l'extérieur d'une classe à accéder aux données privées d'une classe (cf **??**).

Les **variables de classes**, sont des variables partagées par toutes les instances d'une classe. Il n'existe qu'un seul exemplaire de ces variables par classes. Elles permettent de définir un contexte de classe. Par exemple, si on voulait compter le nombre d'objets `Point` d'une classe qui ont été instancié lors d'une exécution.

```
class Point
{
 private:
```

```
  int _nb_Point;

 private:
  int _coord_x;
  int _coord_y;
 public:
  Point(int x, int y);
  float distance();
};

int Point::_nb_Point = 0;

Point::Point(int x, int y)
{
  _nb_Point++;
  // on peut aussi ecrire
  // Point::_nb_Point++;

  _coor_x = x;
  _coor_y = y;
}
```

Les **fonctions de classes** peuvent être assimilés à des fonctions accessibles uniquement en utilisant le nom de la classes. Leur utilisation en C++ est relativement rare , car dans la plus part des cas, elle peuvent être avantageusement remplacées par des fonctions globales amies. Leur utilisation se justifie lorsqu' il est nécessaire de modifier, ou de consulter le contexte définie par les variables de classes privées. Une deuxième utilisation est de permettre la surcharge de fonction de classes ayant le même ensemble de paramètres. Par contre, en Java, leur utilisation sera fréquente, car il n'existe pas de fonctions globales.

Nous allons maintenant illustrer la syntaxe des différentes informations déclarées dans une classe.

```
class X
{
  public:

    typedef int type_public;
    struct _type_public
        {
            type_public champs;
        };
    static int variable_classe_public;
    static void fonction_classe_public(void){}

    X();
    ~X();

    int methode1(){};

  private:
    typedef int (*type_private)(void);
    struct _type_private
      {
          type_private champs;
      };
    static int variable_classe_private;
    static void fonction_classe_private(void){};
};

int X::variable_classe_public = 4;
```

```
main()
{
    X::type_public t1;
    X::_type_public t2;
    X::fonction_classe_public();
    X::variable_classe_public++;
}
```

Figure 2.1: Structure générale d'une classe

A l'intérieur des définitions des entités d'une classe, il n'est pas nécessaire de faire explicitement référence au type de la classe.

```
X::X()
{
    _type_private t1;

    variable_classe_public = 3;
    fonction_classe_public();
}
```

2.3 Encapsulation

Lors de la déclaration d'une classe, il est possible de définir des informations appartenant soit à l'interface publique soit à l'implémentation (interface privée).

A l'exception des relations d'amitiés tous les autres éléments intervenants dans la définition d'une classe pourront voir leur visibilité.

- étendue à l'ensemble de programme, dans ce cas il seront précédé du mot clef `public`.
- restreint aux entités déclarées dans la classe, dans ce cas il seront précédé du mot clef `private`

Lorsque le mot `public` toutes les déclarations suivantes sont visibles de l'extérieur, jusqu'à l'apparition du mot clef `private` qui fera basculer la visibilité en la restreignant à la classe. Par défaut, les éléments d'une classe sont privés.

Comme `public` et `protected`, le mot clef `protected` sert lui aussi à modifier la visibilité des éléments déclarés à sa suite. Ce mot clef établit des relation de visibilité privilégiées entre une classe et ses classes héritières.

Une fois de plus, il est nécessaire de rappeler que l'encapsulation s'obtient pas contrôle de l'accès et non par masquage de l'information. Dans ce cas, il est fortement préférable de ne déclarer dans l'interface d'une classe, que les informations nécessaires à son existence. La déclaration de type local, ou de fonction de classe ou de variables de classes concernant l'implémentation devront être dans un premier temps définis dans le fichier ".c" de la classe. Leur réapparition dans l'interface ne devront être faites que lorsque des fonctions devront être "inliner".

Remarque 5 *Les variables d'instances privées sont nécessaires pour permettre l'instanciation dans la pile. Toutes les autres informations privées servent seulement pour pouvoir insérer le code à l'intérieur d'application. Le mis en ligne d'une classe ne doit être effectuée que lorsqu'on est assurée que son implémentation ne changera plus. Le mise en ligne peut freiner considérablement sa réutilisabilité et son extensibilité.*

2.4 Forme Canonique de Classe

Nous allons créer maintenant notre première classe:

```
classe minimale
{};

minimale f(minimale x)
```

```
{
  return x;
}
void main()
{
  minimale x;
  minimale y;

  x = y;
  y = f(x);
}
```

Aucun code n'a été définie pour la classe `minimale` pourtant, il a été possible

1) de créer des instances de la classe minimale;

2) d'affecter une instance de la classe minimale;

3) de passer en paramètre à une fonction une instance de `minimale`.

4) de retourner par une fonction, une instance de la classe `minimale`.

5) enfin une opération qui n'est pas réellement décelable qui est la destruction d'un objet.

Lors de la définition d'une classe, le compilateur offre par défaut ce comportement minimal. Ce comportement constitue ce que l'on appelle *la forme canonique de classe*. La définition par défaut de la classe minimale produite par le compilateur est:

```
class X
{
    public:
    X();                                    // Constructeur par defaut
    X(const X&);                            // Constructeur de copie
    ~X();                                   // Destructeur

   X & operator = (const X &);              // Operateur d'affectation

};
```

Figure 2.2: Forme Canonique de classe

2.5 Constructeur

Définition 1 *Un* **constructeur** *est une fonction membre d'une classe qui porte le même nom que la classe. Aucune valeur n'est retournée par un constructeur.*

Le rôle d'un constructeur est de définir l'état d'un objet dans l'instant qui suit son instanciation. Il doit donc définir la valeur des variables d'instance de l'objet. L'allocation mémoire de l'instance et des entités qui la composent est effectuée soit lors de la compilation (allocation sur la pile) soit par appel à la fonction `new` (allocation dynamique sur le tas). A la suite de cette étape, le code du constructeur est exécuté.

2.5.1 Constructeur sans argument

Définition 2 *On appelle* constructeur sans argument *un constructeur qui peut être invoqué sans spécifier d'arguments.*

- `X::X()` *réellement sans arguments.*
- `X::(int = 0)` *arguments initialisé par défaut.*

Remarque 6 *Lorsque ces deux constructeurs sont déclarés, il y a ambiguïté au niveau de la déclaration.*

```
class X{
public:
  X(){};
  X(int v=0){};
};
main()
{
  X x;
}
// b.cc: In function 'int main()':
// b.cc:8: call of overloaded constructor 'X()' is ambiguous
// b.cc:4: candidates are: X::X(int)
// b.cc:3:                 X::X()
// b.cc:8: in base initialization for class 'X'
```

Figure 2.3: Ambiguïté de la surcharge sur les constructeur sans argument.

```
class X{
public:
  X(){printf("constructeur sans arguments\n");};
};
main()
{
  X x;
}
//———-$:main
//—————constructeur sans arguments

main()
{
   x x[3];
}
//———-$:main
//—————constructeur sans arguments
//—————constructeur sans arguments
//—————constructeur sans arguments
```

Figure 2.4: Illustration de l'utilisation des constructeurs.

C'est le constructeur sans argument qui est appelé lors de l'initialisation d'un vecteur. Lorsque le constructeur sans arguments n'existe pas, le compilateur génère un constructeur par défaut, mais il n'y a aucune garantie sur la façon dont les objets de cette classe seront initialisés. Le code du constructeur par défaut généré par le compilateur est vide. Si les données membres ont un constructeur par défaut, elle seront correctement initialisées par l'appel à ce constructeur. Sinon le processus est appliqué récursivement. Si la donnée membre est d'un type de base, sa valeur est indéfinie à l'exception des initialisation des variables globales.

L'exemple suivant illustre le mécanisme de construction de l'objet `x` de la classe `X`.

```
class Z{
    public:
        Z(){
          printf("constructeur Z \n");
        }
};
class Y{ Z z; };
class X{ Y x; };
main()
```

```
{
  X x;
}
$:main
constructeur Z
```

La classe `X` ne définit pas de constructeur sans argument, elle utilise le constructeur du compilateur pour instancier `x`. L'objet `x` contient une instance `y` qui doit être elle aussi générée. Comme la classe `Y` ne définit pas de constructeur sans argument, on utilise le même mécanisme pour construire `y`. Cet objet contient une instance `z`, comme la classe `Z` définit un constructeur sans argument, c'est ce code qui est utilisé pour construire `z`.

Masquage du constructeur sans argument

Lorsque un constructeur est défini dans une classe, le constructeur par défaut est masqué s'il n'est pas défini. En d'autre termes, si vous définissez un constructeur, le compilateur n'en génèrera aucun . Ce fonctionnement est illustré par le code suivant:

```
class X{
 public:
  X(int i){};
};
main()
{
  X x;
}
// g++ b.cc
// b.cc: In function 'int main()':
// b.cc:8: no matching function for call to 'X::X ()'
// b.cc:5: candidates are: X::X(const X &)
// b.cc:4:                 X::X(int)
// b.cc:8: in base initialization for class 'X'
```

Il est normal que la définition d'un constructeur dans une classe masque le constructeur par défaut. En effet, si on ne peut créer des objets qu'en utilisant une information, il est inconcevable qu'un mécanisme du langage permette d'en créer à partir de rien.

2.5.2 Constructeur de copie

Définition 3 *Le* **constructeur de copie** *d'une classe X est le constructeur qui est appelé pour copier un objet de la classe X. Son prototype est* `X::X(const X & )` *ou* `X::X(X &,int = 0), ....`

Le constructeur de copie est notamment utilisé pour le passage de paramètres et pour le retour de fonction.

Sur cette exemple, les deux appels au constructeurs sans arguments sont produit respectivement par la construction de la variable globale `X y` et la variable local `X x`. Le premier appel au constructeur de copie est du au retour de fonction `f();`. Cette fonction ne retourne pas une référence de `y` mais un nouvel objet dont l'état est le même que celui de `y`. Le deuxième appel est du au passage de paramètre lors de l'appel à `g(x)`.

```
#include <stdio.h>

class X{
  public:
    X(){printf("constructeur par defaut \n");};
    X(const X &x){printf("constructeur de copy \n");};
};

X y;
X f(){
    return y;
}
```

```
void g(X x)
{
}

int main()
{
    X x;
    f();
    g(x);

}
//    $:main
//       constructeur par defaut
//       constructeur par defaut
//       constructeur de copy
//       constructeur de copy
```

Récursion infinie

Le compilateur prévient lors de récursion infinie sur l'opérateur de copie.

```
#include <stdio.h>
class X{
  public:
    X(X x){printf("constructeur de copy \n");};
};

int main()
{
    X x;
}
//   b.cc:4: invalid constructor; you probably meant 'X (X&)'
//   b.cc:4: syntax error before '{'
//   b.cc:5: parse error before '}'
//   b.cc: In function 'int main()':
//   b.cc:9: too few arguments for constructor 'X::X(const class X &)'
//   b.cc:9: in base initialization for class 'X'
```

Ce cas pose problème car la définition du constructeur de copie utilise elle même sa propre définition. L'utilisation de la référence pour le passage de paramètre ou pour le retour de fonction bloque l'appel au constructeur de copie. Car celui est réservé au passage par valeur et non au passage par référence.

Constructeur de copie par défaut

Lorsque le constructeur de copie n'est pas défini le compilateur génère lui même un constructeur de copie. Le constructeur généré par le compilateur appelle les constructeur de copie sur les données membre s'ils existent, sinon il utilise la copie bit á bit. Le problème lié à la copie bit à bit apparaît lorsqu'un pointeur ou une référence est membre de la classe. En effet, l'objet et sa copie adressent le même espace mémoire.

```
#include <stdlib.h>
#include <stdio.h>
class test2
{
  public:
    void init(char *foo)
        {
            s = (char *) malloc(strlen(foo) + 1);
            strcpy(s,foo);
        }
    void modifie(int pos, char c)
```

```
        {
            s[pos] = c;
        }
    void print_value()
        {
            printf(" valeur de S %s \n", s);
        }
  private:
    char *s;
};
void bidon(test2 t)
{
    t.modifie(0,'x');
}
main()
{
    test2 t1;
    t1.init("aaaaaa");
    bidon(t1);
    t1.print_value();
}
//     $:main
//     valeur de S xaaaaa
```

Figure 2.5: Problème de la copie bit à bit

Si on ne veut pas que la copie et l'objet référence le même espace mémoire, il est nécessaire de définir le constructeur de copie.

```
#include <stdlib.h>
#include <stdio.h>
class test2
{
  public:
    test2(char *foo)
        {
            printf(" Je suis le constructeur avec char * \n");
            s = (char *) malloc(strlen(foo) + 1);
            strcpy(s,foo);

        }
    test2(const test2 &t)
        {
            printf(" Je suis le constructeur de copie \n");
            s = (char *) malloc(strlen(t.s) + 1);
            strcpy(s,t.s);
        }
    void modifie(int pos, char c)
        {
            s[pos] = c;
        }
    void print_value()
        {
            printf(" valeur de S %s \n", s);
        }
  private:
    char *s;
};
void bidon(test2 t)
```

```
{
    t.modifie(0,'x');
}
main()
{
    test2 t1("aaaaaa");
    bidon(t1);
    t1.print_value();
}
//   Je suis le constructeur avec char *
//   Je suis le constructeur de copie
//   valeur de S aaaaaa
```

Figure 2.6: Création d'un constructeur de copie

Existence du constructeur de copie

Comme le constructeur de copie est utilisé pour l'appel de fonction, il ne peut être masqué par la déclaration d'un constructeur.

```
#include <stdio.h>

class X{
  int data;
  public:
  X(int i) { data = i;}
};


void g(X x)
{
}

int main()
{
    X x(3);
    g(x);

}
```

.

2.5.3 Constructeur et liste d'initialisation

Les données membres ont déjà été initialisées lorsqu'on exécute le corps du constructeur.

```
#include <stdio.h>
class Z{
  int i;
public:
  Z(int v = 0){ i = v; printf("constructeur Z valeur de i %d \n", i);};
};

class Y
{
  int j;
public:
  Y(int v = 0){ j = v; printf("constructeur Y valeur de j %d \n", j);};
```

```
};

class X{
  Y y;
  Z z;
public:
  X(int p1 = 0, int p2 = 0){ y = Y(p1); z = Z(p2);};

};
main()
{
  X x(1,2);
}
//   $:main
//     constructeur Y valeur de j 0
//     constructeur Z valeur de i 0
//     constructeur Y valeur de j 1
//     constructeur Z valeur de i 2
```

Sur l'exemple précédent, avant d'exécuter le code du constructeur de `X`, les instances `y,z` sont déjà construites et initialisées. Ensuite, le code du constructeur de `X` est exécuté.

Pour éviter de construire plusieurs fois les mêmes objets, il est possible de définir explicitement la manière d'initialiser les variables d'instances d'un objet en utilisant une *liste d'initialisation*.

```
#include <stdio.h>
class Z{
  int i;
public:
  Z(int v = 0):i(v){ printf("constructeur Z valeur de i %d \n", i);};
};

class Y
{
  int j;
public:
  Y(int v = 0):j(v){printf("constructeur Y valeur de j %d \n", j);};
};

class X{
  Y y;
  Z z;
public:
  X(int p1 = 0, int p2 = 0):y(p1), z(p2){};

};
main()
{
  X x(1,2);
}
//       $:main
//         constructeur Y  valeur de j 1
//         constructeur Z valeur de i 2
```

Lors d'un appel à un constructeur (pas nécessairement au constructeur sans arguments) les données sont initialisées suivant l'ordre de leur déclaration dans la classe.

```
class Y
{
  public:
    Y(){printf("Constructeur Y \n");};
```

```
};
class Z
{
  public:
    Z(){printf("Constructeur Z \n");};
};
class X{
      public:
      X(int i):y(), z() {printf("Constructeur X \n");};

    private:
      Z z;
      Y y;
};
main()
{
  X x(1);
}
//      $:main
//        Constructeur Z
//        Constructeur Y
//        Constructeur X
```

Figure 2.7: Ordre d'initialisation

Remarque 7 *Pour des raisons de lisibilité, et pour éviter des initialisations incorrectes prenez l'habitude, de définir l'initialisation des données membres dans le même ordre que leur déclaration dans la classe.*

2.6 Destructeur

Définition 4 *Une fonction membre d'une classe X notée~X() est appelé un destructeur de la classe X.*

Le rôle d'un destructeur est de restitué au système la mémoire utilisé par un objet, lorsque celui-ci n'est plus visible. Le destructeur, comme le constructeur, n'a pas de type de retour. Il n'est pas non plus possible d'évoquer l'adresse d'un constructeur ou d'un destructeur. Le code du destructeur est exécuté avant que les données membres de la classe aient été détruites. Les données membres sont détruites dans l'ordre inverse de leur déclarations.

```
class Y
{
  public:
   Y(){printf("Constructeur Y \n");};
  ~Y(){printf("Destructeur Y \n");};
};
class Z
{
  public:
    Z(){printf("Constructeur Z \n");};
   ~Z(){printf("Destructeur Z \n");};
};
class X{
  public:
    X():y(), z() {printf("Constructeur X \n");};
   ~X(){printf("Destructeur X \n");};
  private:
    Z z;
    Y y;
};
```

```
main()
{
  X x;
}
//   Constructeur Z
//   Constructeur Y
//   Constructeur X
//   Destructeur  X
//   Destructeur  Y
//   Destructeur  Z
```

Figure 2.8: Ordre de création et de destruction

2.6.1 Effet de bord et destructeur

Lorsqu'un objet n'est plus vivant son destructeur est invoqué.

```
class Y
{
  public:
   Y(char *s){
                name = (char *) malloc(strlen(s)+1);
                strcpy(name,s);
                printf("Constructeur Y de %s \n",name );
              };
  ~Y(){
       printf("Destructeur Y de %s \n",name );
       free(name);
     };
  private:
   char *name;
};

void f(Y y)
{
}
main()
{
   Y y1("une chaine de caracteres");

   f(y1);
// est ce que le nom de y1 est encore valide;
}
```

Figure 2.9: Un destructeur dangereux

Sur cette exemple comme le constructeur de copie n'a pas été défini le constructeur de copie par défaut est utilisé. Lorsque la fonction `f` se termine la chaîne `name` est libérée par l'appel au destructeur. Dans le programme principal, la variable `y1` est maintenant dans un état incohérent. Ce cas ne fait que renforcer la nécessité de définir un constructeur de copie.

2.7 Opérateur d'affectation

Définition 5 *L'opérateur d'affection d'une classe X peut être redéfini. Le prototype de la fonction est alors* `X& X::operator = (const X &)`.

```
main()
{
```

```
    X x1;
    X x2;

    x1 = x2;                                    // x1.operator=(x2);
}
```

Si l'opérateur d'affectation n'est pas redéfini par l'utilisateur, le compilateur génère là encore un opérateur d'affectation par défaut. Le principe est le même que pour le constructeur de copie. C'est à dire que si l'opérateur d'affectation n'est pas redéfinie pour les variables d'instances, on utilise la copie bit à bit. Le problème du partage d'espace mémoire et du destructeur sont alors les mêmes que pour les exemple précédent.

2.7.1 Traiter l'auto-affectation

Lorsqu'on définit l'opérateur d'affectation, il est nécessaire de se prémunir de l'auto-affectation (`x =x`). Ce problème survient rarement sous cette forme, mais il peut être plus difficilement détectable lors de l'utilisation des références.

Si on utilise la première version de ce code pour définir l'affectation, dans le cadre de l'auto-affectation la fonction `strcpy` utilise une chaîne qui est djà détruite. La deuxième version de l'opérateur d'affectation remédie à ce problème.

2.7.2 L'affectation et le constructeur de copie

Attention seul le constructeur de copie est utilisé lors de la définition de variable, en aucun cas l'opérateur d'affectation.

2.7.3 Utilisation de `const` pour l'affectation

L'utilisation du mot clef `const` permet de différencier l'utilisation en membre droit ou en membre gauche.

```
class test2
{
  public:
    test2()
        {
            s = (char *) malloc(1);
            s[0] = '\0';
        }
    test2(char *foo)
        {
            printf(" Je suis le constructeur avec char * \n");
            s = (char *) malloc(strlen(foo) + 1);
            strcpy(s,foo);

        }
    test2(const test2 &t)
        {
            printf(" Je suis le constructeur de copie \n");
            s = (char *) malloc(strlen(t.s) + 1);
            strcpy(s,t.s);
        }
    ~test2()
        {
            printf(" Je suis le destructeur \n");
            free(s);
        }

    void modifie(int pos, char c)
        {
            s[pos] = c;
        }

    void print_value()
        {
            printf(" valeur de S %s \n", s);
        }
  private:
    char *s;
};

main()
{
    test2 t1("aaaaaa");
    test2 t2;
    t2 = t1;
    t2.modifie(0,'D');
    t1.print_value();
    t2.print_value();
}
// Je suis le constructeur avec char *
// valeur de S Daaaaa
// valeur de S Daaaaa
// Je suis le destructeur
// Je suis le destructeur
```

Figure 2.10: Constructeur de copie par défaut et affectation

```
test2 & operator =(const test2 &t)
{
        free(s);
        s = (char *) malloc(strlen(t.s) + 1);
        strcpy(s,t.s);
        return *this;
}
main()
{
    test2 t1("abcd");

    t1 = t1;
}

test2 & operator =(const test2 &t)
{
       if(this ≠ &t)
         {
            free(s);
            s = (char *) malloc(strlen(t.s) + 1);
            strcpy(s,t.s);
         }
        return *this;
}
```

Figure 2.11: Les dangers de l'auto-affectation

```
int main()
{
  test2 t= "abcd";                                          // t("abcd");
  test2 s = t ;                                                // s(t);
}
//    $:main
//    Je suis le constructeur avec char *
//    Je suis le constructeur de copie
//    Je suis le destructeur
//    Je suis le destructeur
```

Figure 2.12: L'affectation et le constructeur de copie

```
class X
{
  public:

    X(){}
    ~X(){}

    X(const X &x){};

    X& operator =(const X&){ return *this;}
};

main()
{
    X x1,x2,x3;


    (x1 = x2) = x3;
}
```

Figure 2.13: Membre gauche autorisé

```
class X
{
  public:

    X(){}
    ~X(){}

    X(const X &x){};

    const X& operator =(const X&){ return *this;}
};
main()
{
    X x1,x2,x3;


    (x1 = x2) = x3;
}
//    $:main
//    b1.cc: In function `int main()':
//    b1.cc:10: non-const member function `X::operator =(const X &)'
//    b1.cc:17: called for const object at this point in file
```

Figure 2.14: Membre gauche non autorisé

Chapter 3

La classe string

Dans ce chapitre, nous allons aborder la conception de la classe `string`. Nous allons faire évoluer la classe string pour la rapprocher d'un type prédéfini. Nous sommes dans un contexte très particulier, car la classe que nous définissons est très proche d'un type de base.

La point de départ de la construction de la classe `string` est la forme canonique de la classe `String`

```
#ifndef _STRING_H
#define _STRING_H

class string{

 public:
  string(const char * ="");
  string(const string &);
  ~string();

  string & operator=(const string &);
 private:
  int length;
  char *data;
};
#endif
```

Dans cette interface, nous utilisons la valeur par défaut des paramètres pour définir le constructeur sans arguments. Les variables privées de cette classe correspondent respectivement à la chaîne de caractères C et à la longueur de la chaîne.

```
#include <stream.h>
#include "string1_inter.cc"

string::string(const char *s):length(strlen(s)),data(new char[length+1])
{
  ::strcpy(data,s);
}

string::string(const string &st):length(st.length),data(new char[length + 1])
{
  ::strcpy(data,st.data);
}

string::~string()
{
  delete [] data;
}
```

```
string & string::operator=(const string &st)
{
  if(this ≠ &st)
    {
      delete [] data;
      length = st.length ;
      data = new char [length + 1];
      ::strcpy(data,st.data);
    }
  return *this;
}
```

L'implémentation de cette classe se fait par délégation à la bibliothèque C de toutes les opérations concernant la manipulation d'une chaîne représentée par un `char *`. On utilise la notation `::strcpy, ::strlen` pour accentuer l'utilisation de fonction globale. Le code des constructeurs sans arguments ou de copie, utilise l'ordre de déclaration des variables d'instances. Cette écriture n'est pas des plus facile à lire, elle sert simplement à illustrer l'utilisation de la liste d'initialisation et l'ordre d'initialisation des variables. L'opérateur d'affectation traite correctement le problème de l'auto-affectation.

Le constructeur `string(const char * ="");` permet de convertir implicitement tout `char *` en une instance de `string`. Comme l'illustre l'exemple suivant:

```
main()
{
  string s1;
  s1 = "abcd";
}
```

Le compilateur essaye d'évoquer l'opérateur d'affectation de la classe `string`. Le seul opérateur défini est `string & operator(const string &)`, pour satisfaire cet appel il essaye de convertir un `char *`, cette possibilité lui est offerte par le constructeur `string(const char * ="");`.

3.1 Interface complète et minimale

Nous allons maintenant étendre la forme canonique de la classe `string`. Les but principaux pour définir l'interface d'une classe sont:

- **Interface Complète**: Tout ce qui est raisonnable de faire avec le type défini par la classe peut s'obtenir à partir des fonctionnalités de l'interface. La satisfaction de cet objectif tend à accroître l'interface.
- **Interface Minimale**: Les fonctionnalités présentent dans l'interface ne sont pas redondantes. La satisfaction de cet objectifs tend à restreindre l'interface.

Une interface complète et minimale permet une réelle utilisation de la classe.
Augmenter une classe peut:

- Diminuer la lisibilité:
  - Oblige l'utilisateur à d'avantage d'efforts;
  - Peut impliquer de la duplication de code;
  - Risque d'engendrer de l'ambiguïté. Deux méthodes réalisent le même traitement mais avec des noms différents.
- Freiner la maintenance. Une classe réduite est plus facile à maintenir qu'une classe trop complexe.
- Diminuer la cohérence de la classe. A force d'accroître on risque de migrer vers un autre type.

Avant détendre une classe, il faut mesurer le coût en terme de complexité et de maintabilité. L'ajout de méthodes qui pouvaient s'obtenir à partir de méthodes existantes peut se justifier par un gain en efficacité, ou par une manipulation plus explicite de la classe.

Remarque 8 *Une classe est le plus souvent conçue lors d'une application particulière. L'ajout d'une méthode peut se justifier pour les besoins de cette application précise. Avant de rajouter cette méthode, il faut vérifier quelle appartient réellement aux fonctionnalités de la classe. Sinon il existe le risque de créer une trop forte dépendance entre la classe et cette application, freinant ainsi la réutilisabilité.*

3.2 Définition de la méthode length

La fonction membre `length` est la première fonctionnalité ajoutée à la forme canonique de classe. Cette fonctionnalité ne peut s'obtenir à partir de la forme canonique de classe et son utilité n'est plus à démontrer. Le prototype de cette méthode est alors `int length()` et le code est:

```
int string::length()
{
  return ::strlen(data);
}
```

La création de cette nouvelle méthode doit être pris en compte par l'implémentation déjà existante. En effet, en dehors de considérations d'efficacités, la variable privée n'a plus raison d'exister.

3.2.1 Les méthodes `const`

L'utilisation de la méthode `const` sur un objet `string` ne modifie pas l'état de cet objet. Le langage C++, permet de préciser dans le prototype des méthodes qu'une méthode ne modifie pas l'état d'un objet en utilisant le mot clef `const`. Le nouveau prototype de la méthode `length` est alors `int length() const`. Le code d'une méthode constante ne peut pas modifier la valeur des variables d'instances de l'objet. Cette règle est vérifié par le compilateur autant qu'il peut le faire. Pour cela, il interdit l'utilisation des variables d'instance en membre gauche, on ne peut appeler que des méthodes constantes sur l'objet `this` ou ces variables d'instances. Ces entités ne peuvent être passées en paramètres que si l'argument leur correspondant est lui-même `const`.

Remarque 9 *Comme pour les utilisations précédentes de* `const`*, il est toujours facile de leurrer le compilateur au travers d'une conversion. Il est alors nécessaire de s'assurer de la constance conceptuelle de l'objet plutôt que de sa constance bit à bit. Une méthode est constante si quelque soit l'objet considéré son état est inchangé par l'appel de la méthode. En supposant que* `bool operator==(const X &, const X &)` *compare l'état de deux objets et non la valeur de leurs bits. Une méthode* `m(....)` *selon*

```
X a;
X b(a);
a.m(....);

if( a == b)
      printf("methode constante");
else
      printf("methode non constante");
```

3.3 Fonctions Amies et fonctions globales

Lors de la définition d'une classe, on crée un nouveau type. Ce type peut avoir des interactions avec les types déjà existant. Il est souvent trop coûteux ou impossible de modifier les classes existantes pour qu'elles interagissent avec le nouveau type. Il est alors nécessaire de créer de nouvelles fonctions pour prendre en compte ces interactions. Considérons l'exemple d'école qui consiste à faire afficher les instances de la classes `string`. Une première solution irréalisable, serait de reprendre la classe `ostream` et de surcharger l'opérateur

```
ostream &operator <<(const string &)
```

et de recompiler cette classe. Les recompilations impliquées seraient alors catastrophiques. La solution consiste alors à définir la fonction globale.

```
ostream & operator<<(ostream &, const string &)
```

Seule la classe `string` accède à ses données, et il n'existe pas d'exportation de la donnée membre `data`. Il est donc impossible de faire afficher le contenu d'une instance de `string` sans violer l'encapsulation. Le langage C++ offre le mot clef `friend` afin de permettre se type de fonctionnement.

3.3.1 Fonctions, Méthodes et Classes Amies

Les fonctions globales qui ont besoin d'accéder aux données membres privées d'une classe doivent être définies comme amies de cette classe. Certaines fonctions globales peuvent accéder ainsi aux données membres d'une classe.

```
class Y
{
public:
  Y(int i = 0):data(i){}
  friend f(Y &);
  private:
  int data;
};

f(Y &y){ y.data = 3;}
```

Remarque 10 *Une fonction amies n'a pas besoin d'être définie dans la classe qu'elle consulte. Elle a seulement besoin d'être déclarée.*

Une méthode d'une classe peut elle aussi enfreindre l'encapsulation d'une classe précise.

```
class Y;
class X
{
    public:
      void f(Y &y);
      void g(Y &y);
};
class Y
{
    public:
      Y(int i = 0):data(i){}
  private:
      friend void X::f(Y&);
      int data;
};

void X::f(Y &y){y.data = 3;};
void X::g(Y &y){y.data = 2;};
friend.cc: In method `void X::g(class Y &)':
friend.cc:19: member `data' is a private member of class `Y'
```

Enfin, toutes les fonctions d'une classe peuvent passer outre l'encapsulation.

```
class Y
{
    public:
      Y(int i = 0):data(i){}
    private:
      friend class X;
      int data;
};
class X
{
    public:
      f(Y &y){ y.data = 2;};
```

```
};
main()
{
  X x; Y y;
  x.f(y);
}
```

3.3.2 `operator ≫ et operator≪`

La première extension de la forme canonique de la `class string` consiste à rajouter les deux fonctions globales représentant les opérateurs de lecture et écriture. Ces deux fonctions ont besoin d'accéder à la donnée membre data, elles doivent donc être déclarées amies de la classe `string`.

```
static inline int is_separator(int c)
{
  return (c == EOF) || c == '\n';
}


static char * temporary_func(istream &input, int size)
{
  int c;
  char *tmp;

  c = input.get();

  if(is_separator(c))
    {
      tmp = new char[size+1];
      tmp[size] = '\0';
      return tmp;
    }
  else
    {
      tmp = temporary_func(input, size + 1);
      tmp[size] = c;
    }
  return tmp;
}


ostream & operator≪(ostream &output, const string &st)
{
  output ≪ st.data;
  return output;
}

istream & operator≫(istream &input, string &st)
{
  int c;
  int i = 0;

  delete [] st.data;
  st.data = temporary_func(input, i);
  return input;
}
```

3.4 operator+

Il existe deux possibilités pour définir cet opérateur,

- soit comme une fonction membre
- soit comme une fonction globale

3.4.1 Operator + comme fonction membre

.

```
string string::operator+(const string &st) const
{
  string tmp;
  delete [] tmp.data;
  tmp.data = new char [this→length() + st.length() + 1];
  ::strcpy(tmp.data,this→data);
  ::strcat(tmp.data,st.data);

  return tmp;
}

main()
{
  string s1 = "asdf";
  string s2 = "efgh";

  cout ≪ " CONCATENATION string + string" ≪ endl;
  s1 = s1 + s2;

  cout ≪ " CONCATENATION string + char *" ≪ endl;
  s2 = s1 + "efgh";

}

//     CONCATENATION string + string
//     constructeur defaut dans operator+
//     constructeur copie asdfgefgh du a return tmp

//     CONCATENATION string + char *
//     constructeur defaut abcd pour passer de char* a string
//     constructeur defaut dans operator+
//     constructeur copie asdfgefgh du a return tmp
```

L'opérateur `string string::operator+(const char *)` n'est pas obligatoire, car le constructeur `string::string(cons char *)` assure la conversion.

Il demeure un problème lorsque le premier argument de l'opérateur de concaténation est de type `char *`. En effet une fonction membre ne peut être évoquée qui si l'opérande de gauche est explicitement un objet de la classe.

```
s2 = "asdfg" + s1;

//main.cc: In function 'int main()':
//main.cc:16: no match for 'operator +(char[5], class string)'
```

3.4.2 Operator + comme fonction globale

.

Il est alors nécessaire de remplacer la fonction membre par la fonction globale.

string operator+(const string &, const string &);

Ce changement est tout à fait licite, car concaténer deux chaînes ne se traduit pas par demander à une chaîne d'utiliser ces informations internes pour en produire une nouvelle.

Comme cette fonction globale accède aux données membres de la classe, il est nécessaire de la déclarer comme une fonction amie.

L'opération qui consiste à concaténer deux chaînes de caractères pour produire une instance de `string` n'est pas possible, car l'opération que l'on essaye d'effectuer est l'addition de deux pointeurs de `char *` convertis en `long`.

```
s2 = "asdfg" + "jklmn";
main.cc: In function 'int main()':
main.cc:9: invalid operands to binary +
```

Il est "*moral*" que cet exemple ne marche pas, car à aucun moment le type `string` n'est évoqué. Et donc l'environnement de la classe *string* ne doit pas modifier un contexte où elle n'est pas évoquée.

3.5 L'opérateur +=

Pour définir l'opérateur operator += il ne suffit pas de définir l'opérateur d'affectation et l'opérateur de concaténation. Il faut explicitement le définir et lui une donner une sémantique équivalente à la composition de la concaténation et de l'affectation.

```
string& string::operator+=(const string &st)
{
  string tmp = *this + st;

  *this = tmp;

  return *this;
}

// cet implementation doit pouvoir etre optimisee
```

3.6 L'opérateur[ ]

Les clients du type `string` ont besoin de pouvoir accéder aux éléments de la chaîne en fonction d'un index. Cette fonctionnalité doît être présente de façon à savoir si un objet `string` contient une chaîne ou un autre objet de type `string`.

Dans un premier on s'intéresse seulement à l'accés en lecture des caractères. L'opérateur d'indexation défini par C++ est `operator[]`.

Comme cette fonction membre ne modifie pas la valeur des données membres de `string`, il s'agit d'une fonction membre qui doit être déclarée comme constante. Pour permettre au compilateur de vérifier que l'indexation est bien utilisée comme une *r-value* le prototype de la cette fonction membre est:

```
const char &string::operator[](int indice) const
{
  return this→data[indice];
}

main()
{
  string s1 = "asdf";

  for(int i = 0; i < s1.length(); i++)
    cout ≪ s1[i];
}
//      $main:
//      asdf
```

```
main()
{
  string s1 = "asdf";

  s1[0] = 'a';
}
//       main.cc: In function `int main()':
//       main.cc:11: assignment of read-only location
```

3.7 L'opérateur de conversion

L'opérateur de conversion permet de convertir un objet de type `string` vers un `char *`.

```
string::operator char *()
{
  return data;
}
main()
{
  const string s1 = "asdf";

  char *tmp = s1;
  tmp[0] = 'X';

  cout << tmp << endl;
}
```

Pour éviter des modifications sur les objets `string` qui sont déclarés comme `const`, il faut que la fonction de conversion ne puisse exporter qu'un pointeur sur une chaîne constante. `string::operator const char *() const;`

3.8 Le code final de classe String

```
#ifndef _STRING_H
#define _STRING_H

#include <stream.h>

class string{

 public:

   string(const char * ="");
   string(const string &);
   ~string();


   string & operator=(const string &);
   int length() const;

   string& operator+=(const string &st);
   const char& operator[](int) const;
   char& operator[](int);

   operator const char *() ;

   friend ostream & operator<<(ostream &output, const string &st);
   friend istream & operator>>(istream &input, string &st);

   friend string operator+(const string &, const string &);

 private:
   char *data;
};
#endif
```

```
#include <stream.h>
#include "string2.h"

static inline int is_separator(int c)
{
  return (c == EOF) || c == '\n';
}

string::string(const char *s):data(new char[strlen(s) + 1])
{
  ::strcpy(data,s);
  cout << "constructeur defaut " << s << endl;
}

string::string(const string &st):data(new char[st.length() + 1])
{
  ::strcpy(data,st.data);
  cout << "constructeur copie "<< st.data << endl;
}

string::~string()
{
  delete [] data;
  cout << "destructeur" << endl;
}


string & string::operator=(const string &st)
{
  if(this ≠ &st)
    {
      delete [] data;
      data = new char [st.length() + 1];
      ::strcpy(data,st.data);
    }
  cout << " affectation " << endl;
  return *this;
}

int string::length() const
{
  cout << " longueur " << endl;
  return ::strlen(data);
}



string& string::operator+=(const string &st)
{
  string tmp = *this + st;

  *this = tmp;

  return *this;
}

const char &string::operator[](int indice) const
{
  cout << " operator [] sur un objet constant" << endl;
```

```
  return this→data[indice];
}

char &string::operator[](int indice)
{
  cout ≪ " operator [] sur un objet non constant" ≪ endl;
  return this→data[indice];
}

string::operator const char *()
{
    cout ≪ "const " ≪ endl;
  return data;
}

string operator+(const string &begin, const string &end)
{

  string tmp;

  delete [] tmp.data;
  tmp.data = new char [begin.length() + end.length() + 1];

  ::strcpy(tmp.data,begin.data);
  ::strcat(tmp.data,end.data);

  cout ≪ "Fonction amie " ≪ endl;
  return tmp;
}



static char * temporary_func(istream &input, int size)
{
  int c;
  char *tmp;

  c = input.get();

  if(is_separator(c))
    {
      tmp = new char[size+1];
      tmp[size] = '\0';
      return tmp;
    }
  else
    {
      tmp = temporary_func(input, size + 1);
      tmp[size] = c;
    }
  return tmp;
}



ostream & operator≪(ostream &output, const string &st)
{
  output ≪ st.data;
  return output;
```

```
}

istream & operator≫(istream &input, string &st)
{
  int c;
  int i = 0;

  delete [] st.data;
  st.data = temporary_func(input, i);
  return input;
}
```

3.9 Les Bonnes Questions

Les questions à ce poser au moment de la définition (cf. le C++ efficace):

1) Comment les objets doivent être créés et détruits? Cela pose le problème de la gestion mémoire mais quel sont les protypes des constructeurs vus par l'extérieur.

2) En quoi l'initialisation est différente de l'affectation.

3) Que signifie passer des instances par valeurs?

4) Quels sont les contraintes sur les valeurs des données membres?

5) Quelles sortes de conversions de types sont autorisées.

6) Quels sont les opérateurs et les fonctions qui sont significatifs pour le nouveau type.

7) Quels opérateurs et méthodes standards doivent être interdites.

8) A qui permettre l'accés aux données membres et aux méthodes du nouveau type.

3.10 Les opérateurs de C++:

- Les opérateurs non surchargeables

  :: .* . ?:

- Les opérateurs surchargeables

+	-	*	/	%	^	&	|
~	!	,	=	≺	≻	≺=	≻=
++	–	≫	≪	==	!=	~~	——
+=	-=	/=	%=	^ =	~=	—=	*=
≪=	≫ =	[]	()	→	→*	new	delete

- Les opérateurs =, [], () et → ne peuvent être que des fonctions membres.

Remarque 11 *Attention, si vous voulez conserver à vos classes une certaine lisibilité, ne surcharger pas les opérateurs pour le plaisir de les surcharger. Lorsqu'on opérateur est définit par une classe il faut que la signification initiale de l'opérateur soit conservée par la nouvelle définition et non pas adaptée.*

Chapter 4

Gestion dynamique de la mémoire

La gestion dynamique de la mémoire en C++, se fait en utilisant les fonctions `new` et `delete` qui servent respectivement à allouer et à libérer la mémoire. Il n'existe pas d'équivalent à la fonction `realloc`.

4.1 L'opérateur new

L'opérateur `new` de prototype `void *operator new (size_t);` de C++, permet l'allocation dynamique. Il se différencie de la fonction d'allocation `malloc` par l'appel au constructeur afin d'initialiser la mémoire allouée. Dans le cas de l'allocation dynamique, les mécanismes de construction sont exactement les mêmes que ceux de l'allocation statique. La mémoire allouée dynamiquement est initialisé par l'appel au constructeur.

```
class X
{
public:
  X():i(1){ cout << "constructeur par defaut"<< endl;}
  X(int iv):i(iv){ cout << "constructeur avec int" << endl;}

private:
  int i;
};

main()
{
  X *x1 = new X;
  X *x2 = new X(3);
}
//constructeur par defaut
//constructeur avec int
```

4.1.1 Allocation d'un vecteur

Lors de la création d'un vecteur, chacun des éléments alloués est initialisé par l'appel au constructeur par défaut. Seul le constructeur par défaut peut être évoqué avec les tableaux. Si on veut utiliser un constructeur particulier pour initialiser les éléments d'un vecteur il est nécessaire de passer par une indirection supplémentaire.

```
main()
{
      X x[3];
      X **pt_x;

      pt_x = (X*)[3];

      for(int i=0; i < 3; i++)
        pt_x[i] = new X(i);
```

```
}

//initialisation vecteur
//constructeur par defaut
//constructeur par defaut
//constructeur par defaut

// initialisation vecteur de pointeurs
//constructeur avec int
//constructeur avec int
//constructeur avec int
```

4.2 L'opérateur delete

L'opérateur `delete` restitue la mémoire allouée par l'opérateur new. Avant de libérer réellement cet espace mémoire, il appelle le destructeur associé à la classe de l'objet pointé.

Lorsque il s'agit de détruire un tableau, l'appel au destructeur est alors le suivant:

```
delete [] tab
```

Dans ce cas, il appelle le destructeur sur chacun des éléments du tableau.

Il est possible d'appeler l'opérateur `delete` sur un pointeur dont la valeur est nulle.

4.3 Surcharge de l'opérateur new et delete

Il est possible de surcharger au niveau d'une classe les opérateurs `new` et `delete`. Cette surcharge se justifie lorsqu'on veut prendre en compte la gestion de la mémoire pour une classe. La surcharge globale de l'opérateur `new` ne peut être justifiée que lorsqu'on dispose d'une mémoire dynamique de taille fixe.

Lorsque l'opérateur `new` est surchargée dans une classe, il n'est pas utilisé pour l'allocation d'un tableau d'objets de cette classe.

```
class X
{
public:
  X():i(1){ cout << "constructeur par defaut"<< endl;}
  X(int iv):i(iv){ cout << "constructeur avec int" << endl;}

  void * operator new (size_t s)
    { cout << "appel a operateur new X" << endl;
      return new char [s];
    }
private:
  int i;
};


main()
{
  X x[3];
  X *x1 = new X;
  X *x2 = new X(3);
}

//constructeur par defaut
//constructeur par defaut
//constructeur par defaut
// Pas d'appel a new surcharge dans la classe X

//appel a operateur new X --> X *x1  = new X;
```

```
//constructeur par defaut

//appel a operateur new X  —> X *x1 = new X(3);
//constructeur avec int
```

Une application de ce mécanisme peut être la délégation au niveau d'une classe de la gestion mémoire de ces instances.

```
#include <stddef.h>

struct linked_list
{
 linked_list *next;
};

class X
{
  static linked_list *l;
  void *data;

 public:
  void *operator new(size_t s);
  void operator delete(void *);
};

linked_list * X::l = NULL;

void * X::operator new(size_t s)
{
  if( l == NULL)
    return (void *) ::new char[sizeof (X)];
  else
    {
      linked_list *tmp = l;
      l = l→next;
      return (X *) tmp;
    }
}

void X::operator delete(void *add)
{
  linked_list *tmp = (linked_list *) add;
  tmp→next = l;
  l = tmp;
}
```

4.4 Association d'un gestionnaire à l'opérateur new

Il est possible d'associer à l'opérateur `new` une fonction qui sera appelée si l'opérateur `new` renvoie une valeur nulle. L'association se fait en utilisant la fonction `void void (*set_new_handler(void (*)()))()` qui retourne l'ancien gestionnaire et positionne le gestionnaire courant au paramètre passé.

```
#include <new.h>
#include <stream.h>
#include <stdlib.h>

void Erreur_Allocation_Memoire()
{
    cerr ≪ " Allocation memoire impossible " ≪ endl;
```

```
    abort();
}

main()
{
    set_new_handler(Erreur_Allocation_Memoire);

    double *X= new double[100000000];
}
//Allocation memoire impossible
//Abort (core dumped)
```

L'utilisation de ce mécanisme permet de traiter les défauts mémoire au niveau d'une classe. Ce mécanisme peut être fort utile si l'on veut créer un mécanisme de ramasse miettes en C++.

4.5 Algorithme de l'allocateur mémoire

L'algorithme général de l'opérateur `new` au niveau global est le suivant. Pour éviter une boucle infinie, la fonction positionnée par `set_new_handler` doit obligatoirement retournée de la mémoire, arrêter le programme, lever une exception, ou repositionner la fonction de traitement à NULL.

```
void *operator new(size_t size)
{
    while(1)
        {
            if(allocation est un succes)
              return un pointeur sur la memoire

            PEHF Current_Handler = set_new_handler(0);
            set_new_handler(Current_Handler);
            if(Current_Handler)
                (*Current_Handler)();
            else
                return 0;
        }
}
```

Les appels consécutifs à `set_new_handler` permettent de récupérer le handler courant. Cet usage se justifie car on ne connaît la variable globale contenant la fonction de gestion.

4.6 Un exemple de la surcharge de new

Dans cet exemple, la classe X surcharge l'opérateur new mais veut tout de même utiliser l'operateur new global. De plus il veut que la fonction du gestionnaire reste inchangée après un appel à ce new spécifique.

L'operateur surchargé commence à sauvegarder la valeur du gestionnaire courant, tout en positionnant son propre gestionnaire. Il appelle ensuite l'allocateur global et termine en repositionnant le gestionnaire à sa valeur initiale.

```
#include <new.h>
#include <stream.h>
#include <stdlib.h>
typedef void (*PEHF) ();
void handler_de_X()
{
    cerr << "allocation memoire problematique dans la classe X" << endl;
}
class X
{
  public:
```

```
    void * operator new(size_t size)
        {
            PEHF gestionnaire_courant = ::set_new_handler(handler_de_X);
            void *memoire = ::new char[size];
            ::set_new_handler(gestionnaire_courant);
            return memoire;
        }
};
void Erreur_Allocation_Memoire()
{
    cerr << " Allocation memoire impossible " << endl;
    abort();
}
main()
{
    set_new_handler(Erreur_Allocation_Memoire);

    X *x = new X;
    double *ess= new double[100000000];
}
```

4.7 Le new avec paramètres

Il est possible de surcharger l'opérateur new en lui passant des paramètres. Cette surcharge offre deux possibilités

1) Passer la fonction du gestionnaire en paramètres

2) Permettre une allocation placée

4.7.1 Le new avec gestionnaire

```
#include <new.h>
#include <stream.h>
#include <stdlib.h>
typedef void (*PEHF) ();

void handler_de_X()
{
    cerr << "allocation memoire problematique dans la classe X" << endl;
}

class X
{
  public:
    void * operator new(size_t size, void (*f_gestionnaire)())
        {
            cout << "appel a new surcharge " << endl;
            PEHF gestionnaire_courant = ::set_new_handler(f_gestionnaire);
            void *memoire = ::new char[size];
            ::set_new_handler(gestionnaire_courant);
            return memoire;
        }
};

main()
{
    X *x = new (handler_de_X) X;
    X *y = new X;                          // erreur du au masquage d'operateur new dans la
```

```
                    // classe X
}
```

4.7.2 Le new avec placement

```
char tab[1024];
int position = 0;
class X
{
 public:
     X(){};
     void * operator new(size_t size, void * place)
          {
               position += size;
               return (void *) place;
       }
     void operator delete (void *add, size_t size)
          {
               position -= size;
          };
     private:
     char t[3];
};
main()
{
     X *x1,*x2,*x3;
     cout << "position --->" << position << endl;
     x1 = new ((void *) (tab + position)) X;
     cout << "position --->" << position << endl;
     x2 = new ((void *)(tab + position)) X;
     cout << "position --->" << position << endl;
     delete x2;
     cout << "position --->" << position << endl;
     x3 = new ((void *) (tab + position)) X;
     cout << "position --->" << position << endl;
}
//position —>0
//position —>3
//position —>6
//position —>3
//position —>6
```

4.7.3 Un placement sans initialisation

```
class X;
class Y
{
 public:
    Y( X &v):x(v){};
  private:
     X &x;
};
class X
{
 public:
     X(Y &v):y(v){};
     void * operator new(size_t size, void * place)
          { return (void *) place;}
     void *operator new(size_t size)
```

```
        { return ::new char[size];}
  private:
    Y &y;
};
// X x(y); y was not declared in this scope
// Y y(x);

char *x_mem = new char[sizeof(X)];
Y y((X &) x_mem);
X &x = *(new (x_mem) X(y));
main(){}
```

Chapter 5

Héritage

La relation d'héritage public entre deux classes traduit une relation **est un**. Par exemple, une 2CV est une voiture, dans ce cas la classe 2CV hérite de la classe voiture.

Attention à ne pas confondre la relation **est une instance de** avec la relation d'héritage. La première relation relie un objet à sa classe, la deuxième relation est une relation entre classes. Par exemple, la 2CV immatriculé `234 zz 33` est une instance de la classe 2CV, mais elle n'hérite pas de la classe `voiture`.

La relation d'héritage établit une hiérarchie entre deux classes. On parle aussi de **surclasse** et de **sous-classe**. Lorsqu'on utilise l'héritage simple, on dit parfois que la surclasse est la **classe mère** et la sous-classe **la classe fille**.

Comme la relation d'héritage traduit une relation **est un** il existe en C++ une conversion implicite d'une instances de la sous-classe vers une instance de la surclasse. Attention, cette relation est orientée, toutes les voitures ne peuvent être converties en 2CV.

La "*phrase choc*" de l'héritage est alors la suivante:
Tout ce qui caractérise la surclasse (comportement et attributs) caractérise la sous-classe. La sous-classe à la possibilité de se distinguer de la surclasse:

- Soit en modifiant le comportement d'une fonction membre en la spécialisant;
- Soit en enrichissant le comportement par le rajout de nouvelles fonctions membres.

Le comportement de la surclasse est toujours inclut dans le comportement de la sous-classe, en aucun cas, la sous classe ne pourra avoir un comportement inférieur à celui de sa surclasse. Attention, du fait de la possibilité de surcharger certaines fonctions mais pas toutes il peut exister des pertes de potentialités dues à la surcharge.

La sous-classe peut se définir comme une spécialisation de la surclasse, l'héritage va de la définition la plus petite vers la plus complète.

5.1 L'héritage public en C++

Pour dire qu'une classe hérite publiquement d'une autre classe, il suffit d'utiliser la syntaxe suivante:

```
class fille:public mere
{
  bloc de definition.
}
```

Toute les données ou fonctions publiques de la classe mère peuvent être évoquées par une fille. *Tout ce que fait ma mère, je peux le faire même si c'est de manière différente et même si je peux en faire plus.*

```
class mere
{
public:
  void methode1();

  int data;
};
```

```
void mere::methode1()
{
  cout ≪ "La methode 1" ≪ endl;
}

class fille: public mere{};

main()
{
  char *s = new char[strlen("fille") + 1];
  fille f;
  // Illustration de la conversion fille -> mere
  mere &m = f;

  // Illustration de l'existence de la methode1 sur les fille
  f.methode1();
  // Illustration de l'existence de la variable data sur les filles
  f.name = s;
}
```

Cet exemple illustre, la conversion implicite fille en mère et le fait que l'interface publique de la mère existe dans la fille, sans que cette dernière est besoin de le redéfinir.

5.2 Protection de données privilégiée

La relation d'héritage publique entre deux classes crée une relation particulière entre deux classes.

Les données ou fonctions publiques de la surclasse sont accessibles au niveau de la sous-classe comme pour tout autres classes d'ailleurs.

Les données ou fonctions privées de la surclasse ne sont pas accessibles au niveau de la sous-classe comme pour tout autres classes.

Il existe la possibilité de définir des données ou des fonctions qui sont publiques pour les classe héritées et privées pour les autres classes. Ce mécanisme est implémenté en utilisant le mot clef `protected`.

```
class mere
{
  public:
    mere():public_pour_fille(0), reellement_privee(1){}
    void methode1();
  protected:
    int public_pour_fille;
    void methode2();

  private:
    int reellement_privee;
};
class fille:public mere
{
  public:
    void methode();
};

void fille::methode()
{
    methode1();
    public_pour_fille = 1;

    methode2();
    reellement_privee = 2;
```

```
    //  member 'reellement_privee' is private
}
main()
{
    mere m;
    m.methode2();
    // method 'void  mere::methode2 ()' is protected
    m.public_pour_fille = 3;
    // member 'public_pour_fille' is
    // a protected member of class 'mere'
}
```

Détail amusant, une classe `F` ayant `M` pour super classe ne peut accéder au membres protégés de de `M` *qu'a travers* un objet dont le type est connu pour être au moins `F`.

```
class M {
protected:
  int i;
};

class F1 : public M {
  void f(M& m) { int i = m.i;}  // member 'i' is a protected member of class 'M'
  void f(F1& f1) { int i = f1.i;}  // ok
};

class F2 : public M {
  void f(F1& f1) { int i = f1.i;}  // member 'i' is protected
};
```

5.3 Constructeur et destructeur d'une classe dérivée

Le constructeur d'une sous-classe commence par appelé le constructeur de la surclasse. Lors de la destruction les opérations s'effectuent dans l'ordre inverse de la construction.

```
class X{
public:
  X(){ cout ≪ " Constructeur Classe X" ≪ endl;}
  ~X(){cout ≪ " Destructeur Classe X" ≪ endl;}
};
class mere
{
public:
  mere() { cout ≪ " Constructeur Classe Mere" ≪ endl;}
  ~mere(){ cout ≪ " Destructeur Classe Mere" ≪ endl;}
};
class fille: public mere
{
public:
  fille():x(){cout ≪ "Constructeur Classe Fille" ≪ endl;}
  ~fille(){cout ≪ "Destructeur Classe Fille" ≪ endl;}
private:
  X x;
};
main(){fille f; }
//Constructeur Classe Mere
//Constructeur Classe X
//Constructeur Classe Fille
```

```
//Destructeur Classe Fille
//Destructeur Classe X
//Destructeur Classe Mere
```

5.3.1 Les constructeurs et les destructeurs ne sont pas hérités

.

Les constructeurs et les destructeurs ne sont pas hérités dans les sous-classes.

```
class mere
{
 public:
   X(int i){}
   X(char *chaine);
};


class fille{};

main()
{
   fille f1("fille");
   fille f2(3);
}
// unherit.cc: In function 'int main()':
// unherit.cc:13: no matching function for call to 'fille::fille (char[6])'
// unherit.cc:9: candidates are: fille::fille(const fille &)
// unherit.cc:9:                 fille::fille()
// unherit.cc:13: in base initialization for class 'fille'
// unherit.cc:14: no matching function for call to 'fille::fille (int)'
// unherit.cc:9: candidates are: fille::fille(const fille &)
// unherit.cc:9:                 fille::fille()
// unherit.cc:14: in base initialization for class 'fille'
```

5.3.2 Liste d'initialisation d'une classe dérivée

Pour les mêmes raisons d'efficacité que lors de l'initialisation des données membres, il vaut mieux appelé le constructeur de la surclasse dans la liste d'initialisation de la sous-classe.

```
fille::fille():mere(),x()
{
}
main(){
    fille f;
}
//Constructeur Classe Mere
//Constructeur Classe X
//Constructeur Classe Fille
//Destructeur Classe Fille
//Destructeur Classe X
//Destructeur Classe Mere
```

Pour des raisons de lisibilité, respectez l'ordre utilisé par le compilateur lors de l'initialisation. D'abord les surclasses, puis les données membres dans l'ordre de leur déclaration.

5.4 Liaison statique

La résolution de l'appel des fonctions membres en C++, s'effectue de manière statique par défaut. Au moment de la compilation, il utilise le type de l'objet pour déterminer la fonction membre à évoquer.

Il est toujours possible de spécialiser au niveau de la sous-classe une fonction de la surclasse. **Vous n'aurez le droit de spécialiser une fonction que si vous utiliser les liaisons dynamique. Imaginer les réactions d'un client qui obtient les résultats suivant**

```
class mere{
public:
  void methode1(){ cout << "methode1 de mere" << endl;}
};
class fille:public mere{
public:
  void methode1(){ cout << "methode1 de fille" << endl;}
};
main()
{
  fille f;
  mere &m = f;
  f.methode1();
  m.methode1();
}
//methode1 de fille
//methode1 de mere
// Le meme objet change de comportement
// en fonction du type
```

Les fonctions d'une surclasse dont la liaison est statique ne doivent jamais être redéfinies dans les sous-classes, elles ne servent qu'à éviter la duplication de code, et elles n'utilisent que les attributs de la surclasse. Par contre, le code de méthode liée statiquement peut utiliser des méthodes définis dans la fille en utilisant les liaisons dynamique.

Dans la programmation objet, l'élément essentiel est l'objet. Ce qui est fondamental pour un objet c'est son comportement. Il est inconcevable que le comportement d'un objet soit dépendant du type qu'on l'on utilise pour l'évoquer.

5.5 Liaison dynamique

Il existe en C++ la possibilité de lier dynamiquement les fonctions membres. Cela consiste à préfixer la déclaration de la fonction du mot clef `virtual`. Si la fonction ainsi déclarée est **redéfinie** dans la sous-classe, se sera elle qui sera toujours appelé pour les instances de la sous-classe.

```
class mere{
public:
  virtual void methode1(){ cout << "methode1 de mere" << endl;}
};
class fille:public mere{
public:
  virtual void methode1(){ cout << "methode1 de fille" << endl;}
};

main()
{
  fille f;
  mere &m = f;
  f.methode1();
  m.methode1();
}
//methode1 de  fille
//methode1 de  fille
// Un meme objet conserve son comportement
// quelque soit le type que l'on utilise
// pour l'\'evoquer.
```

Attention sur une branche d'héritage il suffit qu'une fonction soit déclaré avec le mot clef `virtual` pour que toutes les fonctions définies en dessous, soient elle aussi liées dynamiquement. Pour des raisons de lisibilités, si vous redéfinissez une liaisons dynamique dans vos sous-classes, répéter le mot clef `virtual` dans votre interface.

5.5.1 Liaison dynamique dans les constructeurs et les destructeurs

Les liaisons dynamiques marche de manière normale dans les constructeurs et destructeurs avec cependant une exception illustrés par ce cas de figure.

```
#include <stream.h>

class mere
{
  public:
    mere(){ f(); }
    ~mere() { f();}
    virtual void f(){ cout<< "Mere" << endl;}
};

class fille: public mere
{
  public:
    fille(){ f(); }
    void f(){ cout<< "Fille" << endl;}
};


void main()
{
        fille f;
// Mere
// Fille
// Mere
}
```

Dans cette exemple, lorsqu'on est entrain de créer une mère, à partir d'une fille, les liaisons dynamiques sur f() ne sont pas utilisées. Il en est de même pour le destructeur, dans le contexte de mère on utilise que le code défini dans mère, on utilise pas la liaison dynamique en utilisant la mère.

5.6 Choix de la nature du comportement

Il est impossible de lier dynamiquement des fonctions de classes, c'est pour cela que certaines fonctions qui ont une sémantique de fonction de classe ou de fonction globale sont transformées en méthodes afin de pouvoir être spécialiser dans les sous-classes.

Comment choisir la définition d'une fonction `f` rattachée à une classe `C` (*"Le C++ efficace"*).

Définir correctement une fonction

```
// les fonctions virtuelles doivent etre membres
if( f a besoin d'etre virtuel)
   faire de f une methode
else
{
   // operator<< et operator>> ne sont jamais membres
   if( f  est operator>> ou operator<<)
     {
       faire de f une fonction globale
       if( f a besoin d'acceder aux donnees membres privees)
         faire de f une fonction amie
     }
   else
     {
       if(f a besoin de conversion de type sur son argment a gauche)
          {
            faire de f une fonction globale
            if( f a besoin d'acceder aux donnees membres privees)
              faire de f une fonction amie
          }
       else
 faire de f une methode
     }
}
```

5.7 Méthode virtuelle et constructeur

Lorsque le constructeur ou le destructeur de la classe mère est évoqué depuis une classe fille, les fonctions virtuelles évoquées durant cette étape sont celles de la classe mère.

```
class mere
{
  public:
    mere(){ methode1();}
    virtual void methode1();
};
void mere::methode1()
{
    cout << "Appel methode1 de la mere" << endl;
}

class fille:public mere
{
  public:
    virtual void methode1();
};
void fille::methode1()
{
    cout << "Appel methode1 de la fille" << endl;
```

```
}
main()
{
 fille f;
}
// Appel methode1 de la mere
```

5.8 Méthode virtuelle et destructeur

Le destructeur comme le constructeur n'utilisent pas les liaisons dynamiques à l'intérieur de son code.

```
#include <iostream.h>

class mere
{
  public:
    virtual void m(){ cout << "mere" << endl;}
    virtual ~mere(){ m();}
};

class fille:public mere
{
  public:
    virtual void m(){ cout << "fille" << endl;}
};


main()
{
    fille f;
}
```

Le destructeur d'une classe doît toujours être déclaré comme virtuel, sinon des fuites mémoires peuvent avoir lieu.

5.9 Partage entre la mère et la fille

Ce qui n'est pas hérité de la surclasse:

1) Les constructeurs y compris le constructeur de copie;

2) Les destructeurs;

3) L'opérateur d'affectation

4) Les fonctions membres cachées. Lorsqu'une fonction membre n'est pas redéfinie dans la sous-classe mais qu'elle est surchargée , elle ne fait plus partie du comportement de la sous-classe.

```
class mere
{
  public:
    virtual void methode1(int);
};
void mere::methode1(int i)
{
    cout << "methode1 avec int" << i << endl;
}

class fille:public mere
{
```

```
  public:
    virtual void methode1();
};
void fille::methode1()
{
    cout ≪ "methode1 sans parametre" ≪ endl;
}
main()
{
 fille f;
 mere &m = f;

 f.methode1();
 m.methode1(3);
 f.methode1(3);
 // too many arguments for method 'void fille::methode1()'
}
```

Sur cette exemple, la classe `fille` définit une nouvelle méthode dont le prototype est `virtual void methode1()`. Cette nouvelle fonction est une surcharge de la méthode`mere::void methode1(int)` qui n'est pas redéfini dans la fille. La stratégie utilisée par C++ pour résoudre la sélection de l'appel `f.methode1(3)` consiste à rechercher dans le contexte de la classe fille une méthode dont le nom est `methode1()`. Ensuite comme la surcharge existe, il utilise le type des paramètres pour trouver une adéquation **dans le même contexte**. Comme cette résolution échoue il n'explore pas le contexte de la mère est génère donc une erreur. Dans ce cas, le comportement d'un objet serait dépendant du type par lequel on l'accède, ce qui est contraire au principe des liaisons dynamiques et de l'héritage.

Pour éviter ce genre de désagrément, lorsqu'une sous classe doit étendre le comportement de sa mère en utilisant la surcharge il est nécessaire de redéfinir au niveau de la sous-classe la totalité des fonctions ayant le même nom.

```
class fille:public mere
{
  public:
    virtual void methode1() { cout ≪ "methode1 sans parametre" ≪ endl; };
    virtual void methode1(int i) {mere::methode1(i);}
};
```

5.9.1 Fonction virtuelle pure

Une fonction virtuelle qui est déclarée avec " = 0" après la liste des arguments

```
class X
{
    void methode() = 0;
};
```

est une **fonction virtuelle pure**. Il n'est pas nécessaire de fournir une définition de cette fonction. Toute classe qui déclare une fonction virtuelle pure est une classe **abstraite ou virtuelle**. La propriété d'une classe virtuelle est de ne pouvoir être instanciée.

Une classe abstraite est utilisée pour fournir une interface sans en donner la réalisation. Cette déclaration sert à unifier les sous-classes dérivées en donnant leur structure commune.

Une classe dérivée qui ne redéfinit pas une fonction virtuelle pure est elle même une classe abstraite.

```
class mere{
  public:
    virtual void methode1(char *s) = 0;
    virtual void methode2() = 0;
};
void mere::methode1(char *s)
```

```
{
    cout ≪ s ≪ endl;
}
class fille:public mere
{
  public:
    virtual void methode1(char *s);
    virtual void methode2();
};
void fille::methode1(char *s)
{
    mere::methode1(s);
}
void fille::methode2()
{
    cout ≪ "la methode2 de la classe fille" ≪ endl;
}
main()
{
//     mere m;
//ex8.cc: In function 'int main()':
//ex8.cc:33: cannot declare variable 'm' to be of type 'mere'
//ex8.cc:33:    since the following virtual functions are abstract:
//ex8.cc:33:        void mere::methode2()
//ex8.cc:33:        void mere::methode1(char *)

    fille f;

    f.methode1("je suis une fille");
    f.methode2();
}
//je suis une fille
//la methode2 de la classe fille
```

La déclaration d'une méthode virtuelle pure, n'exclut pas la définition de son code. En effet, le code de la méthode virtuelle pure peut être utilisé dans les classes filles.

```
#include <stream.h>

classe mere
{
  public:
    virtual void methode() = 0;
};

void mere::methode()
{
    cout ≪ "Evocation d une methode virtuelle pure "≪ endl
}

class fille
{
  public:
    virtual void methode(){ mere::methode()};
}
```

Il est aussi possible de définir des classes abstraites en déclarant protected les constructeurs. Dans ce cas l'extérieur n'a plus la possibilité d'instancier cette classe. Elle n'aura d'existence qu'á travers les classes

dérivées.

```
class mere{
  public:
    virtual void methode1(char *s);
  protected:
    mere(){};
};
void mere::methode1(char *s)
{
    cout << s << endl;
}
class fille:public mere
{
  public:
    virtual void methode2();
};
void fille::methode2()
{
    cout << "la methode2 de la classe fille" << endl;
}
main()
{
//     mere m;

// ex9.cc: In function 'int  main ()':
// ex9.cc:7: constructor 'mere::mere ()' is protected
// ex9.cc:28: within this context
// ex9.cc:28: in base initialization for class 'mere'

    fille f;
    f.methode1("je suis une fille");
    f.methode2();
}
//je suis une fille
//la methode2 de la classe fille
```

5.10 Utilisation de l'héritage

Les avantages de l'héritage public sont:

1) Factorisation du code

2) La possibilité de définir correctement des abstractions.

3) Polymorphisme *longrightarrow* extensibilité et maintenabilité.

Quand doit on définir une relation d'héritage:

1) **Spécialisation**: Lorsqu'il est nécessaire de spécialisé une classe. En effet, dans certains cas, une classe est amenée à évoluer lors de la conception, plutôt que de remettre une simulation du typage dynamique il vaut mieux créer une nouvelle classe est établir un lien d'héritage avec la classe existante. Par exemple, on commence à définir une classe avion dans laquelle, on considère que l'avion ne possède qu'un seul moteur. Par la suite, l'évolution du problème amène à considérer des avions bimoteur. L'erreur de conception est de programmer la classe avion de cette manière:

```
class avion
{
public:
  enum type_avion {MONOMOTEUR , BIMOTEUR};
```

```
    avion(type_avion t);
    void vole();

  private:
    type_avion type;
};

void avion::vole()
{
  switch(type)
    {
    case MONOMOTEUR:
      monomoteur_vole();
      break;
    case BIMOTEUR:
      bimoteur_vole();
      break;
    }
}
```

Même si l'on améliore le programme en utilisant les concepts de la programmation dirigée par les données, on aboutit à une aberration. La bonne manière de concevoir la solution est de créer une classe avion virtuelle, et de faire hériter de cette classe les classes `avion_monomoteur` et `avion_bimoteur`.

2) **Généralisation**: La généralisation est la démarche complémentaire à la précédente. Reprenons le même problème à résoudre mais vu par un concepteur différent. Ce concepteur a correctement identifié les classes `avion_monomoteur` et `avion_bimoteur`, mais il n'a pas créer de classe `avion`. Dans ce cas, comment peut il faire voler ces instances dans le programme principal? De nouveau, il va devoir prendre a son propre compte la résolution du type. Pour obtenir la bonne conception il est alors nécessaire de généraliser les classes précédentes en créant la classe `avion`. Le code qui fonctionnera avec cette classe, pourra alors sans modification majeure s'étendre sans problème aux avions à réaction.

Remarque 12 *Tout cet exposé ne peut fonctionner que si la liaison dynamique existe en C++. En effet, le mécanisme de liaison dynamique permet de sélectionner le comportement d'un objet en fonction de sa véritable nature et non pas de son type.*

5.11 Polymorphisme

Le concept fondamental que permet l'héritage public, couplé avec la liaison dynamique est le **polymorphisme**. La première étape du polymorphisme consiste à unifier le type des sous-classes en les faisant héritées d'une même surclasse. La deuxième étape consiste à sélectionner le comportement d'une instance non pas en fonction de son type à un moment donné, mais en fonction de sa véritable nature. Le polymorphisme permet une extensibilité réelle des programmes.

```
class avion
{
  public:
    virtual void vole() = 0;
};

class avion_bimoteur:public avion
{
  public:
    virtual void vole();
};

class avion_monomoteur:public avion
{
  public:
```

```
    virtual void vole();
};

void avion_monomoteur::vole()
{
    cout ≪ "je vole comme un mono_moteur" ≪ endl;
}

void avion_bimoteur::vole()
{
    cout ≪ "je vole comme un bimoteur" ≪ endl;
}

main()
{
    avion *tab[2];

    tab[0] = new avion_monomoteur;
    tab[1] = new avion_bimoteur;

    tab[0]→vole();
    tab[1]→vole();
    delete [] tab;
}

//je vole comme un mono_moteur
//je vole comme un bimoteur
```

Chapter 6

Les exceptions en C++

6.1 Mécanisme des exceptions

La gestion des exceptions en C++ se fait par l'utilisation de trois mots clefs `try, catch, throw`. Le mot clef `try` sert à définir un bloc dans lequel les exceptions sont susceptibles d'être capturées. Après la définition de ce bloc se trouve la liste des gestionnaires d'exception. Un gestionnaire d'exception est défini en utilisant le mot réservé `catch`. La liste des gestionnaires d'exception ressemble à un ensemble de fonctions surchargées. Une exception est levée en utilisant le mot clef `throw`. L'identification de l'exception, se fait en utilisant le type. Par exemple, la fonction `f` peut lever deux types d'exception, une identifier par un entier, l'autre par une chaîne de caractères constantes.

```
#include <stream.h>

void f(int i)
{
  if (i == 0)
    throw i;

  if (i ≠ 0)
    throw "une erreur";
}
void g(int i)
{
  try
    {
      f(i);
    }
  catch(const char *s){ cout ≪ "exception" ≪ endl;}
  catch(int j){ cout ≪ "exception" ≪ j ≪ endl;}
}
main()
{
  g(0);
  g(1);
}
```

6.2 Recherche du gestionnaire d'exception

On remonte récursivement dans les blocs `try` en cherchant un gestionnaire `catch` dont les arguments correspondent. Une clause `throw` pour un type `T` correspond à un handler de type `E, const E, E &, const E &` si

1) `T` et `E` sont du même type.

2) `T` est une classe hérité de `E`

3) `T` et `E` sont des pointeurs qui peuvent être mis en correspondance par une conversion standard.

La première correspondance trouvée stoppe la recherche de l'exception. Il est alors nécessaire de déclarer les exceptions de la plus précise vers la moins précise afin de ne pas masquer des traitements.

Dans le cas, ou aucune correspondance n'est trouvée la fonction `unexpected()` est évoquée, par défaut cette fonction appelle la fonction `terminate()` qui elle même appelle la fonction `abort()`. Dans le cas, ou aucune correspondance n'est réellement trouvée il est possible de définir sa propre fonction `unexpected()` en utilisant le fonction `void (*)() set_unexpected(void (*)())`. Le principe de fonctionnement est exactement le même que celui permettant d'associer un gestionnaire à la fonction `new` en utilisant `set_new_handler()`.

6.3 Extension du `throw` et du `catch`

On peut continuer la transmission de l'exception à un contexte supérieur en utilisant la clause `throw` sans type. Dans ce cas, on utilise le même objet que celui qui nous a amené dans cette portion du code. Il est aussi possible de définir un gestionnaire `catch(...)` qui est appelé quelque soit l'exception. Au vue de la stratégie utilisée par C++, cette clause `catch` doit être la dernière de la liste.

```
#include <stream.h>

void h()
{
  throw 1;
}

void f(int i)
{
  try
    {
      h();
    }
  catch(int i)
    {
      cout << "exception de la fonction f " << endl;
      throw ;
    }
}

void g(int i)
{
  try
    {
      f(i);
    }
  catch(...){ cout << "exception de la fonction g " << endl;}
}

main()
{
  g(1);
}
// exception de la fonction f
// exception de la fonction g
```

6.4 Les exceptions et le changement de contexte

Le déclenchement d'une exception implique un changement de contexte. Le problème lié au changement de contexte est celui de la validité des objets créé entre le contexte où à lieu l'exception et celui où est traitée l'exception. Le langage C++ se charge de libérer les objets qu'il a construit et il reste, à l'utilisateur, à gérer les objets qu'il a lui même alloués dynamiquement.

```
#include <stream.h>
#include <strings.h>
class X
{
public:
  X(){}
  ~X(){ cout << "destruction de l'objet" << endl;}
private:
};
void h()
{
  X x;
  X *y = new X;                                   // une place memoire perdue
  throw 1;
}
void f(int i)
{
  X y;

  try
    {
      h();
    }
  catch(int i)
    {
      cout << "exception de la fonction f " << endl;
      throw i;
    }
}
void g(int i)
{
  try
    {
      f(i);
    }
  catch(int i){ cout << "exception de la fonction g " << endl;}
}
int main()
{
  g(1);
}
//destruction de l'objet
//exception de la fonction f
//destruction de l'objet
//exception de la fonction g
```

Cette stratégie est raisonnable, car il se peut que des objets créer entre les deux contextes, doivent continuer à vivre car il ne sont pas concernés par l'exception qui a été levé. Lors de l'utilisation des exceptions, il est utile d'avoir une pile ou une liste d'objets à libérer en fonction du type de l'exception levée.

6.5 Spécification des exceptions

La déclaration des exceptions doit faire partie des spécifications d'une fonction ou des fonctions membres. Il est possible de spécifier les exceptions d'une fonction au moment de sa déclaration. Cette spécification ne peut pas remplacer une description complète de l'exception.

Spécification d'exception

```
void f() throw(int, const char*);
```

Sur cet exemple la fonction `f` est susceptible de lever les exceptions correspondante à un entier et à une chaîne de caractères. Si cette fonction essaye de lever une exception différentes aucune erreur de compilation apparaîtra.

```
void f() throw(int, const char*)
{
    double d = 3.0;
    throw d;
}

main()
{
    try{
        f();
    }catch(double d){}
}
```

Par contre, une erreur d'exécution aura lieu même si le type de l'exception est prévu dans les clauses `catch`.

Remarque 13 *Les exceptions implémentées par g++ ne correspondent pas à la norme en effet, la résolution des clauses* `catch` *se fait uniquement sur des types exacts. Tous les objets alloués dans la pile ne sont pas systématiquement détruits lors d'un changement de contexte.*

Comme par défaut une fonction ou une méthode est susceptible de lever toutes les exceptions, le compilateur ne veut pas prendre de décision. Nous verrons la convention inverse en Java, ou par défaut une fonction membre ne peut lever aucune exception. En C++ la déclaration d'une fonction qui doit lever aucune exception est:

Spécification d'une liste vide d'exception

```
void f() throw();
```

6.6 Définition d'exception et héritage

Même si le compilateur C++ est laxiste pour la vérification des exceptions, il faut correctement définir la liste d'exception pour les fonctions héritées. La règle de définition doit répondre au critère suivant. Soit une méthode `m() throw`$(T_1, T_2, \ldots, T_n)$ de la classe mère, la méthode `m() throw`$(T'_1, T'_2, \ldots, T'_k)$ redéfinie dans la classe fille doit vérifier la condition suivante $T'_1, T'_2, \ldots, T'_k \subset (T_1, T_2, \ldots, T_n)$ en autorisant les conversions par héritage. Dans ce cas, la liste d'exception de la fille est inférieure ou égale à celle de la mère. Cette condition permet d'utiliser correctement les liaisons dynamiques, en effet une application qui traite correctement les exceptions provenant d'une mère traitera correctement les exceptions provenant d'une fille.

6.7 Utilisation des exceptions

Il existe trois manières différentes d'utiliser les exceptions:

1) La première utilisation consiste à associer un lever une exception lors de la détection d'un dysfonctionnement du programme. La définition de ce type d'exception est délicate car elle peut rentrer en concurrence avec l'utilisation des assertions. Imaginons qu'un utilisateur rentre 50 objets et que malencontreusement il demande à accéder au cinquante et unième. Dans ce cas, une assertion stopperait brutalement le programme et les données saisies seraient perdues. Il est préférable de lever une exception dans ce cas, enfin de gérer correctement l'erreur. Lors de la détection d'une violation du contrat, préférer l'usage des exceptions à celui des assertions. Si le client ne traite pas l'exception, l'effet sera similaire à une assertion. L'usage de l'exception permet de définir une poignée utilisable par le client.

2) La deuxième utilisation consiste à transmettre un événement à un contexte appelant. Par exemple, dans le cadre d'un jeu, il peut être utile de lever une exception pour signaler que le joueur a gagné ou perdu. Cette technique est préférable à la modification d'une variable globale ou à un retour de fonction.

3) La troisième utilisation (plus tirée par les cheveux) est d'utiliser les exceptions comme instruction d'échappement. Par exemple, supposons que la méthode `read()` d'un fichier lève l'exception `EOFException` lorsque la fin de fichier est atteinte. La lecture d'un fichier peut être écrite de la manière suivante:

```
try{
  while()
    fic.read();
}
catch(EOFException e){}

// La suite du programme
```

Chapter 7

Les patrons ou "*template*"

La "*paramétrisation*" (ou encore *polymorphisme paramétrique*) s'obtient en C++ en utilisant des patrons (*template*). Les patrons définissent des familles de types ou de fonctions. Le concept représenté par les patrons est celui de la *généricité*. On écrit une fonction ou une classe générique en utilisant un type paramétré `T`, ensuite ce code peut être utilisé en donnant une valeur spécifique à ce type `T`.

L'utilisation des templates, ne fournit pas une réutilisation du code objet, mais une réutilisation du code source. Dans ce cas, la taille du code objet généré peut croître de manière importante. De plus lorsqu'on utilise les templates, leur déclaration et leur définition doivent être définis dans l'interface, ce qui peut produire de multiples recompilations lors de changements d'implémentation. Il est à remarquer que ces limitations sont partiellement liées au compilateur utilisé.

7.1 Exemple d'une fonction template

Nous voulons écrire une fonction `T max(T,T)` qui compare deux entités de type `T` et qui retourne la plus grande de ces deux entités.

```
// Declaration

template <class T>
T max(const T &, const T &);

// Definition

template <class T>
T max(const T &a, const T &b)
{
    return a > b? a : b;
}
```

Le mot `template` est toujours placé au début de la définition et de la déclaration. Ensuite vient la liste des paramètres formels séparés par des virgules et délimités par des $\prec, \succ$. Chaque paramètre formel est défini par le mot `class` suivi d'un identificateur. L'exemple suivant décrit une fonction template utilisant deux types paramétrés différent.

```
template <class T, class Y>
T f(T t, Y y)
{
  return t + (T) y;
}
```

Si l'on reprend l'exemple avec la fonction `max`, tout les types qui pourront être mis en correspondance avec `T` doivent pourvoir être comparé en utilisant l'opérateur `int operator <(...)`.

```
template <class T>
T max(const T &a, const T &b)
{
    return a > b? a : b;
}

class X
{
 public:
  X(int i=0):data(i){}
  int data;
};

main()
{
  X x1,x2;

  x1 = max(x1,x2);
// In function 'class X max(const class X &, const class X &)':
// no match for 'operator >(class X, class X)'

}
```

Comme cet opérateur n'est pas défini pour la classe `X`, il est impossible d'utiliser la fonction template `max` sur ce type.

Remarque 14 *Les patrons impliquent que tous les types qui seront assimilé au type paramétrique, doivent posséder toutes les fonctionnalités utilisées par la définition des patrons.*

7.2 Résolution de fonctions patrons

Une fonction template peut être surchargée soit par d'autres fonctions de même nom ou par des fonctions template avec la même nom. La résolution s'effectue en trois étapes:

1) Recherche d'une superposition exacte des paramètres;

2) Recherche d'une fonction template qui peut être appelée avec une superposition exacte des paramètres

3) Recherche d'une surcharge ordinaire pour les fonctions (cf. ARM section 13.2).

```
#include <stream.h>
template <class T>
T max(const T &a, const T &b)
{
  cout << "fonction max template " << endl;
  return a > b? a : b;
}

int max (int i, int j)
{
  cout << "fonction max sur les entiers " << endl;
  return i>j?i:j;
}

main()
{
  int x = max(3, 2);                                    // etape 1
  float y = max(3.0,0.2);                               // etape 2
  int z = max(2.0,3);                          // etape 3 coercicion double -> int
}
```

7.3 Les classes patrons

L'utilisation des classes patrons permet de générer des classes génériques. Par exemple, nous voulons écrire une classe représentant une pile générique. Ce conteneur est un conteneur homogène par référence. C'est à dire qu'il ne capture que les références aux objets et non leur valeur.

```
#include <stream.h>

class mere
{
  public:
   mere(){ }
   virtual void f(){ cout≪ "Mere" ≪ endl;}
   static void g(mere l);
};

class fille: public mere
{
  public:
   fille(){ }
   void f(){ cout≪ "Fille" ≪ endl;}
};

void mere::g(mere l)
{
  mere *k = new fille;
  l.f();
  k→f();
}

void main()
{
  fille f;
 mere::g(f);

// Mere
// Fille
}
```

Maintenant que la classe est définie, il reste à préciser les types qui pourront être mis en correspondance avec le type paramétré `T`. Tous les types, pourront utilisée cette classe, car la seule opération dépendant du type est l'affectation des références qui est un mécanisme interne à C++.

7.4 La classe stack de void *

Une amélioration de l'utilisation des classes patrons consiste à factoriser le code en utilisant une classe stack de **void ***. Dans ce cas, la classe template utilise la classe de **void *** en lui déléguant le traitement propre à `stack`. Le rôle de la classe template dans ce cas précis est de convertir le type paramétré en un `void *` et réciproquement.

```
#ifndef _STACK_POLY_H
#define _STACK_POLY_H

class stack_poly{
 public:
  stack_poly();
  ~stack_poly();
// il faudrait une forme canonique de classe.
```

```
  push(const void *);
  const void *pop();
  int empty();

 private:
  struct cell
    {
      struct cell *next;
      const void *data;
    } *top;
};
#endif
```

```
#include <stream.h>
#include <assert.h>
#include "stack_poly.h"
stack_poly::stack_poly():top(NULL)
{
}

stack_poly::~stack_poly()
{
  while(!empty())
    pop();
}

stack_poly::push(const void *e)
{
  cell *tmp = new cell;
  tmp->data = e;
  tmp->next = top;
  top = tmp;
}

const void *stack_poly::pop()
{
  cell *tmp = top;
  const void *e = tmp->data;

  top = top->next;
  delete tmp;

  return e;
}

int stack_poly::empty()
{
  return top == NULL;
}
```

7.4.1 Utilisation pour un conteneur homogène par référence

Le nouveau code de la classe patron `stack` est alors:

```
#ifndef _STACK_H
#define _STACK_H
#include "stack_poly.h"
```

```
template <class T>
class stack{
 public:
   stack();
   ~stack();
   push(const T &);
   T& pop();
   int empty();
 private:
   stack_poly *delegated;
};

template <class T>
stack<T>::stack():delegated(new stack_poly){}

template <class T>
stack<T>::~stack()
{
   delete delegated;
}

template <class T>
stack<T>::push(const T &e)
{
   delegated.push((const void *) &e);
}
template <class T>
T& stack<T>::pop()
{
   return *((T *) delegated.pop());
}
template <class T>
int stack<T>::empty()
{
   return delegated.empty();
}
#endif
```

Cette utilisation des classes templates offre les mêmes avantages que l'utilisation en C des modules génériques. Le définition de la classe template n'est là que pour assurer l'homogénéité du type. C'est à dire que le compilateur vérifiera autant que faire se peut ce qui est insérer dans le module pile.

7.4.2 Utilisation pour un conteneur homogène par valeur

Une nouvelle classe `stack` qui ne laisse pas pénétrer les effets de bords. Ce conteneur stocke les états des objets insérés et non leur valeur. La condition pour pouvoir insérer un objet dans le nouveaux conteneur est que son constructeur par copie soit accessible.

```
template <class T>
class stack{
 public:
   stack();
   ~stack();
   push(const T &);
   T pop();
   int empty() const;
 private:
   stack_poly *delegated;
};
```

```
template <class T>
stack<T>::stack():delegated(new stack_poly)
{
}
template <class T>
stack<T>::stack():delegated(new stack_poly)
{
  while(!empty())
    {
      T *tmp = pop();
      delete tmp;
    }
}

template <class T>
stack<T>::push(const T &e)
{
  delegated.push(new T(e));
}

template <class T>
T stack<T>::pop()
{
  T *tmp = ((T *) delegated.pop());
  T t = *tmp;

  delete tmp;
  return t;
}
```

La seule modification dans l'interface de la classe `stack` et le changement de signature de la méthode `T pop();`. En effet, il n'est plus possible de rentre une référence à un objet car dans ce cas, se serait à l'extérieur à détruire l'objet alloué.

7.5 Héritage privé

Une autre possibilité pour implémenter la délégation est l'utilisation de l'héritage privé.

L'**héritage privé** traduit une relation est *implémentée en terme de* , il traduit exactement la même relation que la délégation. L'héritage privé concerne essentiellement l'implémentation, il ne traduit pas une relation de conception. L'héritage privé n'implique pas une interface commune entre la classe mère privée et la classe fille.

```
class X
{
public:
  void virtual methode1() {};
};

class Y: private X{};

main()
{
  Y y;
  y.methode1();
}
//In function 'int main()':
//method 'void X::methode1()' is from private base class
//within this context
```

Sur cette exemple, la méthode `methode1(...)` ne fait pas partie de l'interface publique de la classe `Y` car elle hérite de manière privée de la classe `X`.

Le classe fille peut être considérée comme une mère uniquement dans son propre contexte.

```
class X
{
public:
  void virtual methode1()
    { cout << "methode 1 mere " << endl;};
protected:
  int data;
};

class Y: private X
{
public:
  void methode2()
    {
      data = 3;
      cout << "methode 2 fille "<< endl;
      methode1(); }
};

main()
{
  Y y;

  y.methode2();
}
```

7.6 Héritage privé et template

L'héritage cumulé avec les templates pour implémenter le container pile. Dans ce cas, tout les appels qui précédemment utilisés la variable `delegated` utilise maintenant des appels au méthodes définies dans la mère.

```
template <class T>
class stack: private stack_poly{
 public:
  stack();
  ~stack();
  push(const T &);
  T pop();
  int empty();
};

template <class T>
stack<T>::stack()
{
}
template <class T>
stack<T>::~stack()
{
  while(!empty())
    {
      T *tmp = pop();
      delete tmp;
    }
}
```

```
template <class T>
stack<T>::push(const T &e)
{
  stack_poly::push(new T(e));
}
template <class T>
T stack<T>::pop()
{
  T *tmp = ((T *) stack_poly::pop());
  T t = *tmp;
  delete tmp;
  return t;
}
template <class T>
int stack<T>::empty()
{
  return stack_poly::empty();
}
```

7.7 Différencier héritage et template

Reprenons les spécifications d'un problème donné dans *Le C++ efficace.*

- on veut représenter une pile d'objets. La pile est un conteneur homogène il n'y a qu'un seul type d'objet dans la pile. Par exemple, on aura une pile de int, de float, ...
- On veut écrire une classe pour représenter des chats. En réalité il faut définir un ensemble de classes pour représenter chaque espèce de chats.

Dans les deux cas, il s'agit de définir un ensemble de classes de types différents. La question à se poser est de savoir si le type intervient réellement dans le comportement de la classe.

Est-il judicieux d'utiliser l'héritage public pour définir l'ensemble des piles, alors que le type ne conditionne pas réellement le comportement de la pile. Par contre un problème apparaîtrait pour connaître le type des paramètres de la fonction `push` dans la surclasse, quel est le type à lui donner. La bonne solution consiste à utiliser un template avec une pile de `void *`.

Par contre pour la classe chat, chaque espèce va devoir spécialiser le comportement de la classe de base. Si l'on veut utiliser une classe template chat, il faut alors spécialiser le comportement en fonction du type. La question à se poser est comment représenter le type à l'intérieur du template.

Chapter 8

Héritage Multiple

L'héritage multiple est délicat à manipuler, surtout s'il existe une ambiguïté pour la sélection des méthodes. La bonne manière de concevoir l'héritage multiple est non pas comme une union de comportements mais comme une intersection de comportements.

```
class D: public B1, public B2{ }
```

Dans ce cas, les ensembles $B1$ et $B2$ ont des éléments communs qui appartiennent à l'ensemble D. Une instance de D est donc à la fois un $B1$ et un $B2$. Si $B1$ et $B2$ ont une intersection vide, il ne faut pas utiliser l'héritage multiple, car l'héritage multiple ne représente pas la relation D *est soit un* $B1$*, soit un* $B2$..

Dans la plus part des cas, une solution préférable à l'héritage multiple est d'utiliser la composition. Cette utilisation sera plus amplement détaillé, lors de la présentation des modèles de conception.

8.1 Ambiguïté lors de la sélection

```
#include <stream.h>

class A
{
  public:
    virtual void m() {cout ≪ "CLASSE A"≪endl;}
};


class B
{
  public:
   virtual void m() {cout ≪ "CLASSE B"≪endl;}
};

class C: public A, public B
{};


void main()
{
  C c;
  c.m();
//mult1.cc: In function 'int main(...)':
//mult1.cc:23: request for method 'm' is ambiguou
}
```

Sur cet exemple, la classe C ne redéfinit pas la méthode `m`. Cette méthode étant à la fois définie dans les classe `A` et `B`, le compilateur ne sait pas laquelle choisir. Pour lui l'évocation de la méthode `m` dans la class `C` est ambigu.

Même si l'on ordonne différemment les relations d'héritage, le problème demeure.

```
class A
{
  public:
    virtual void m() {cout ≪ "CLASSE A"≪endl;}
};


class B: public A
{
  public:
   virtual void m() {cout ≪ "CLASSE B"≪endl;}
};

class C: public A, public B
{};
```

Le compilateur ne donne pas des priorités en fonction de la profondeur dans l'arbre d'héritage.

8.2 Héritage en diamant

Si l'on excepte les ambiguïté du à la sélection des méthodes, le principal problème de l'héritage multiple concerne l'héritage en diamant.

```
#include <stream.h>

class mere {
     int x;
  public:
     mere(int i = 0):x(i){ cout ≪"Mere" ≪ endl;}

};

class A: public mere
{
  public:
     void methode1(){ cout ≪" Classe A" ≪ endl;}

};


class B: public mere
{
  public:
     void methode1(){ cout ≪" Classe B" ≪ endl;}

};

class AB:public A, public B
{
  public:
     void methode1(){ cout ≪" Classe AB" ≪ endl;}
};


void main()
```

```
{
     AB ab;
// Mere
// Mere
// Classe AB

     ab.methode1();
}
```

En exécutant se programme, on s'apperçoit qu'il existe deux appels aux constructeurs de la classe `mere`. Cela est du au fait que pour construire, un élément de la classe `AB`

1) il faut construire un élément de la classe `A` qui déclenche la construction d'une instance de mère.
2) il faut aussi construire un élément de la classe `B` qui déclenche la construction d'une autre instance de mère.

Une conséquence de ce problème est illustré par l'exemple suivant.

```
#include <stream.h>

class mere {
  protected:
    int x;
  public:
    mere(int i = 0):x(i){ cout ≪"Mere" ≪ endl;}

};

class A: public mere
{
  public:
    void inc(){ x++; }

};


class B: public mere
{
  public:
    void dec(){x--;}
    void print(){ cout ≪ x ≪ endl;}

};

class AB: public A, public B
{
  public:

};


void main()
{
     AB ab;

     ab.inc();
     ab.print();
     ab.dec();
     ab.print();
}
```

Sur cet exemple, on s'apperçoit que les appels à `inc()` n'ont aucune répercussion sur le programme. Cela est du au fait qu'il existe pour une instance de la classe `AB` deux occurrences de la variable `x`. Le première occurrence x_a est produite par l'instanciation de la classe `A`, la deuxième occurrence x_b est produite par l'instanciation de la classe `B`. Ces deux occurrences sont indépendantes, les méthodes `dec(), print()` accède à x_b alors que la méthode `inc()` accède à x_a.

Ce problème peut être résolu en utilisant les classes de bases virtuelle.

8.3 Classe de Bases virtuelles

Les classes de bases virtuelles remédient aux problèmes posés par la duplication des instances dans un héritage en diamant. Considérons maintenant le code suivant:

```
#include <stream.h>

class mere {
  protected:
    int x;
  public:
    mere(int i = 0):x(i){ cout <<"Mere" << endl;}

};

class A: public virtual mere
{
  public:
    void inc(){ x++; }

};


class B: public virtual mere
{
  public:
    void dec(){x--;}
    void print(){ cout << x << endl;}

};

class AB: public A, public B
{
  public:

};


void main()
{
     AB ab;

     ab.inc();
     ab.print();
     ab.dec();
     ab.print();
// Mere
// 1
// 0

}
```

Par rapport au code précédent, la déclaration des classes `A` et `B` sont devenues:

```
class A: public virtual mere
class B: public virtual mere
```

Dans ce cas, on dit que `mere` est une classe de base virtuelle des classes `A` et `B`. Le compilateur ne créera qu'une seule instance d'une classe de base virtuelle sur la partie de l'arbre d'héritage concerné. Sur l'exemple précédent on s'appercoit que le constructeur de la mère n'est appelé qu'une seule fois.

Chapter 9

Le Langage Java(tm)

La première caractéristique revendiqué par Java est d'être un langage portable. Pour ce faire, Java est compilé en byte-code, le byte-code ètant un code intermédiaire tournant sur un machine virtuelle. De cette façon, toute machine possédant une implémentation de cette machine virtuelle pourra exécuter du byte-code issu de la compilation d'un programme Java. La phase de compilation de Java consiste donc à transformer la syntaxe du langage en un code pour une machine virtuelle. Cela se fait en utilisant la commande `javac`. Cette commande prend en entrée un fichier suffixé `".java"` et génère un fichier suffixé `".class"`. C'est ce dernier fichier qui représente le pseudo-code interprétable par la machine virtuelle. Le pseudo-code est ensuite interprété sur une machine cible en utilisant la commande `java`.

La deuxième caractéristique revendiquer par Java, est dêtre un langage à objets. Si l'on fait abstraction des types de bases, Java est un véritable langage à objets. En effet, la relation d'héritage est représenté par un arbre avec une unique racine `Object`. Toute classe hérite implicitement de `Object`. De plus, il existe la classe `Class` qui hérite de `Object`. Toutes les classes de Java seront associées à une unique instance de `Class`.

Les principales caractéristiques du langage sont:

- Une véritable hièrarchie de classe;
- Un ramasse miettes;
- Gestion performante des exceptions;
- Le chargement dynamique;
- Evaluation dynamique du type;
- Liaison dynamique;
- Une large bibliothèque de base, comprenant en autre l'AWT, les processus légers, le réseau, etc.

9.1 Première Aplication en Java

La syntaxe de Java est fortement inspirée du langage C. Nous allons commencer par écrire la sempiternelle première application `"hello World"`.

```
class Hello
{
  static public void main(String [] argv)
    {
      System.out.println("Hello World");
    }
}
```

Il n'existe pas de fonction globale en Java. Pour pouvoir créer du comportement il est obligatoire de créer une méthode ou une fonction de classe. Ceci implique nécessairement la création d'une classe. Pour pouvoir faire afficher "Hello World" nous créons le fichier `Hello.java` qui définit une classe `Hello`. Cette classe ne possède qu'une seule fonction de classe dont le prototype est `void main(String [] argv)`. Cette fonction peut être assimilable au programme principal d'une application. La commande `java Hello` demande à l'interpreteur d'éxecuter la fonction de classe `void main(String argv[])`, si elle n'existe pas, le message suivant apparait alors:

```
In class Hello: void main(String argv[]) is not defined
```

Sur cet exemple, on retrouve un élément syntaxique commun avec le langage C++. A savoir que les variables ou fonctions de classes sont préfixées du mot clef `static`. La deuxième chose à remarquer est l'utilisation du mot clef `public` qui a une signification équivalente à celle de C++. La fonction `main()` doit être à la fois `public` et `static`.

Pour l'instant, l'instruction `System.out.printl("Hello World")` consiste à évoquer la méthode `println` de la variable de classe out définit dans la classe `System`.

9.1.1 Visibilité en Java

.

En langage Java, l'unité de compilation est un fichier Java. Il ne peut y avoir qu'une seule classe `public` dans un fichier Java. Cette classe doit porter le même nom que le fichier sans le suffixe Java. Il n'existe pas comme en C++, un fichier pour décrire l'interface et un fichier pour décrire l'implémentation. La déclaration et la définition d'une classe en Java s'effectuent dans le même temps. Par contre, si un fichier suffixé ".java" contient plusieurs classes Java, il existera autant de fichier suffixé ".class" que de classe Java.

La règle pour définir une classe publique nommée `Class_name` est de la définir dans un fichier appelé `Class_name.java`. De la façon suivante:

```
public class Class\_name
{
    Declaration et Definition de Class\_name
}
```

Une classe déclarée publique peut être accessible depuis n'importe qu'elle autre classe Java. Son chemin bien sûr doit être définie dans la variable `CLASSPATH`. C'est à dire que n'importe quelle classe a accés à l'interface publique de la classe `Class_name`.

Par rapport à C++, les mots clefs `static,private, public` ont la même signification. A la différence de C++, les mots clefs `private` et `public` ne définissent pas des basculements de visibilité, ils font partie de la signature de chaque élément de la classe.

```
public class Exemple
{
  private static int v_classe;
  public static void f_classe(){};
  private int v_instance;

  public void methode1(){};
  private void methode2(){};
}
```

Sur cette exemple, `v_classe` est une variable de classe accessible uniquement par le code de la classe `Exemple`. La variable `v_instance` est une variable d'instance private. La méthode `methode1()` peut-être évoquée en utilisant une classe de `Exemple` depuis n'importe quelle partie d'une application `Java`.

9.1.2 Paquetage en Java

Un paquetage en Java, est un nom permettant de regrouper un certains nombre de classe ayant une sémantique commune. Physiquement, un paquetage Java est constitué de plusieurs ".java" qui produisent des fichiers ".class".

Lorsqu'une classe appartient a un paquetage, le nom complet de la classe et composé du nom du paquetage suivit du nom de la classe. Par exemple, le nom complet de la classe `Hashtable`, qui appartient au paquetage `java.util`, est `java.util.Hashtable`. La directive `import java.util.*;` nom permet d'utiliser le nom `Hashtable` plutot que `java.util.Hashtable`[1]. Il est important de remarquer que cette directive n'a d'effet que sur le nommage, aucune notion de "chargement" n'y est associée.

L'appartenance des classe d'un fichier ".java" à un paquetage `java.mypack` se fait en utilisant la déclaration `package java.mypack;`

Un paquetage représente plus qu'une organisation du code, il crée certains accés priviliégés entre les classes appartenant à un même paquetage.

Par exemple, le fichier `entity1.java`

[1] Le paquetage `javalang` est inclus automatiquement.

```
package java.mypack;

class entity1
{
}
```

Le fichier suivant `entity2.java`,

```
package java.mypack;

class entity2
{
   void methode1()
      {
        entity1 e;
      }
}
```

Bien que la classe `entity1` ne soit pas définie dans le même fichier que `entity2` et bien que la classe ne soit pas `public`. Il est possible d'évoquer des `entity1` à partir `entity2` car les 2 classes appartiennent au même paquetage.

Il n'existe pas de hiérarchie à l'interieur des paquetages en fonction de leur nom. Par exemple, `java.mypack.value` n'est pas un sous paquetage du paquetage `java.mypack`. Ces deux paquetages différents non aucune relation entre eux.

Dans la section précédente, nous avons vu que si la déclaration d'une classe est précédé du mot clef `public` toutes les autres classes de Java en ont la connaissance. Si on ne met rien devant la déclaration du classe, seules les classes du même paquetage en auront la connaissance. Il en est de même pour les éléments d'une classe. Si aucun modificateurs de visibilité (`private, protected, public`) n'est défini devant une méthode, une variable de classe, ... par défaut cet élément est public pour les classes du paquetage et privé pour les classes extérieures au paquetage.

9.1.3 Le mot clef `protected`

La signification du mot clef `protected` est la même que pour le langage C++. Considerons la déclaration `protected int vi1` à l'interieur de la classe `mere`. Toutes les classe dérivant de `mere` ont acces à la variable `vi1`. De plus toutes les classe appartenant au même paquetage que `mere` ont aussi accés à cette variable.

9.2 La surcharge en Java

Comme en C++, les fonctions de classes et les méthodes d'instances peuvent être surchargées au sein d'une même classe. La distinction se fait comme en C++, en fonction du type et du nombre des paramètres.

```
class entity2
{
   void meth1(){ entity1 e;}
   void meth1(int i){};
   int meth1(char c){ return c;}
}
```

Comme en C++, elle s'appuie sur le nombre et le type des paramètres en aucun cas sur le type de retour. Contrairement à C++, il n'existe pas en Java, la possibilité de redéfinir les opérateurs du langage.

9.3 La référence en Java

La gestion dynamique des objets en Java s'appuie sur un mécanisme de ramasse-miettes. Lors de la création d'un nouvel objet, le ramasse-miettes a en charge de lui donner un certains espace mémoire. Un objet est considéré comme mort lorsque aucune variable ne le référence. Dans ce cas, le ramasse-miettes récupère

l'espace mémoire occupé par l'objet. A l'exception des types de bases. il est impossible en Java d'utiliser la mémoire de la pile pour créer des objets.

L'instanciation d'un objet se fait à l'aide de l'opérateur `new`. Cet opérateur alloue l'espace physique nécessaire à l'objet et appelle le constructeur en utilisant les paramètres. Cet operateur fonctionne syntaxiquement de la même manière que l'opérateur `new` de C++. La seule différence est qui ne peut être ni surchargé ni redéfini.

En Java tout est basé sur les références, à l'exception des types de bases. Le passage de paramètres et les retours de fonctions se font par références.

De se fait le fonctionne des références en Java est très différent de celui de C++. Rappelons qu'en C++, la référence doit se faire au moment de la définition de la variable. Une fois la référence établit il est impossible d'en changer.

Supposont qu'une classe `entity1` existe en Java. L'instruction `entity1 e1;` déclare l'existance d'un variable `e1` qui pourra être utilisée pour référencer des objets convertibles dans le type `entity1`. Mais au moment, de cette déclaration `e1` ne référence rien. Et surtout, cette déclaration n'implique pas la création d'un objet de type `entity1`. En Java la référence est temporaire. A savoir, l'opérateur d'affectation en Java, consiste à changer la référence d'une variable, en aucun la valeur de l'objet de référencé. L'instruction `e1 = e;` implique que `e1` et `e` identifie le même objet physique. L'opérateur de comparaison, `==` compare si deux variables identifie le même objet physique, il ne compare pas les états.

```
import java.lang.*;

class entity1
{
  int x;

  public entity1(int i)
    {
      x = i;
    }

  public void display()
    {
      System.out.print("Je suis l'objet -->");
      System.out.println(this.toString());
      System.out.print("Ma valeur est -->");
      System.out.println(x);
    }
}

public class start
{
  public static void main(String [] argv)
    {
      entity1 tmp = new entity1(4);
      entity1 e;
      entity1 e1;

      e = new entity1(3);
      e1 = e;

      e1.display();
      e.display();

      e1 = tmp;
      e1.display();
    }

}
```

```
// Je suis l'objet ->entity1@404b4878
// Ma valeur est ->3
// Je suis l'objet ->entity1@404b4878
// Ma valeur est ->3
// Je suis l'objet ->entity1@404b4880
// Ma valeur est ->4
```

9.4 Les constructeurs en Java

Les constructeurs en Java fonctionnent comme en C++. Leur rôle est de positionner l'objet dans un certain état lorsqu'il est instancié. Le constructeur intervient aprés que le ramasse-miettes est alloué l'espace mémoire.

Un constructeur en Java est défini comme une fonction qui porte le même nom que sa classe. Comme la surcharge est possible, il existe diverses manières de construire les objets lors de leur création. Si aucun constructeur n'est défini dans une classe Java, le langage fournit par défaut un constructeur sans argument. Ce constructeur est masqué dès qu'un autre constructeur est défini. Comme le langage Java fonctionne exclusivement par référence, il n'existe aucun constructeur de copie.

L'exemple suivant illustre, les différents constructeurs d'un complexe.

```
class Complexe
{
  private double reel;
  private double img;

  Complexe() {reel = img = 0.0;}
  Complexe(double d){ reel = d; img = 0.0;}
  Complexe(double d, double i) { reel = d; img = i;}
}
```

Le langage Java offre certaines particularités pour les constructeurs. Il est possible, de définir une initialisation par défaut pour les variables d'instances (idem pour les variables de classe). Il n'existe pas en Java, de liste d'initialisation mais on peut utiliser un constructeur existant, au debut du code d'un autre constructeur.

```
class Complexe
{
  private double reel = 1.0;
  private double img = 1.0;

  Complexe() {}
  Complexe(double d){ this(d, 0.0);}
  Complexe(double d, double i) { reel = d; img = i;}
}
```

Sur cette exemple, par défaut la valeur d'un complexe est définie par $(1.0, 1.0)$. Comme le code du constructeur sans argument est vide, tous les complexes utilisant ce constructeur, seront dans létat par défaut. Le constructeur de complexe avec un seul paramètre, utilise le code du constructeur avec 2 arguments au moyen de l'appel `this(d, 0.0);`. Cette instruction doit être impérativement la première instruction du code du constructeur.

9.4.1 Faut-il définir une liste d'initialisation

La liste d'initialisation n'existe pas en Java. La première question est de savoir si les problèmes résolus par la liste d'initialisation peuvent se poser en Java. Rappelons que la liste d'initialisation permettait d'indiquer qu'elle type de constructeur devait être utilisé pour initialiser, les variables d'instances ou les instances de la classe mère.

```
class entity1
{
```

```
  entity1(int i){}
}

class entity2
{
  private entity1 e1;

  entity2(){};
}

class j_ex4
{
  public static void main(String []s)
    {
      entity2 e = new entity2();
    }
}
```

Cet exemple pose un problème en C++, car il impossible de créer une `entity1` sans utiliser un argument. L'utilisation de la liste d'initialisation est obligatoire, car le constructeur sans argument est masqué. En Java, le code précédent fonctionne correctement car le code du constructeur `entity1(int i)` n'est pas évoqué par le code `new entity2()`. En effet, pour l'instant un objet de la classe `entity2` contient une référence à un objet de type `entity1`. Mais pour créer un objet `entity2` il n'est pas obligatoire que `e1` référence un objet existant ou crée cet objet.

Un autre problème réglé par l'utilisation de la liste d'initialisation est celui de l'appel au constructeur de la mère lors de la construction d'une fille, dans le cas ou le constructeur par défaut est masqué.

```
class mere
{
  mere(int i){};
}

class fille extends mere
{
  fille(){}
}

public class j_herit
{
  static void main (String [] s)
    {
      fille f = new fille();
//j_herit.java:8: No constructor matching mere() found in class mere.
// fille(){}
    }

}
```

Sur cet exemple, le problème illustré est le suivant: La classe fille hérite de la classe mère, pour créer une fille le compilateur doit instancier une mère. Comme il ne peut utiliser le constructeur sans arguments, le compilateur génère un message d'erreur. Pour permettre, ce genre de construction, le langage java permet au programmeur de choisir le constructeur utilisé pour instancier une sur classe. Ce choix s'effectue en utilisant la syntaxe suivante: `super(..,..,..)`. Le mécanisme est similaire à celui représenter par `this(..,..)` qui permet d'utiliser un autre constructeur.

```
class mere
{
  mere(int i){};
}
```

```
class fille extends mere
{
  fille(){super(0);}
}

public class j_herit
{
  static void main (String [] s)
    {
      fille f = new fille();
    }

}
```

Comme pour l'utilisation de `this(..,..)`, l'instruction `super(..,..,..)` doit être la première instruction du code du constructeur. Elle peut être éventuellement suivit par une instruction d'une instruction `this(..,..,..)`.

9.5 Héritage en Java

L'héritage en Java est un héritage simple d'implémentation et un héritage multiple d'interface.

9.5.1 Héritage simple d'implémentation

L'héritage simple d'implémentation impose qu'une classe peut utiliser seulement le code d'une seule classe mère. L'héritage simple en java se traduit par l'utilisation du mot clef `extends`.

```
class mere
{
  void methode()
    {
      System.out.println("Methode definie dans mere");
    }
}

class fille extends mere
{

}

class j_herit_s1
{
  static public void main(String [] S)
    {
      fille f = new fille();
      f.methode();
      //Methode definie dans mere
    }
}
```

Sur cette exemple, la classe fille hérite de la classe mère, elle ne redéfinit pas la méthode mère. Pourtant on peut évoquer cette méthode en utilisant une instance de la classe `fille`. C'est dans ce cas de figure que l'on parle d'héritage d'implementation.

Liaisons dynamiques

En java toutes les sélections se font de manière dynamique. C'est à dire que la sélection du comportement se fait en fonction du type réel de l'instance, et non en fonction du type par lequel on accède à l'instance. Quelque soit l'endroit du code ou on se situe (constructeur, fonction de classe, méthode) la liaison s'effectue toujours dynamiquement.

```
class mere
{
  mere(){f();}
  void f(){ System.out.println("Mere");}
}

class fille extends mere
{
  fille(){ f(); }
  void f(){ System.out.println("Fille");}
}


class x
{
  static public void main(String [] argv)
    {
      fille f = new fille();
      // Fille
      // Fille
    }
}
```

9.5.2 Accés aux variables d'instances

Contrairement à la sélection du comportement qui se fait dynamiquement, l'accés aux variables d'instances se fait statiquement. La sélection d'une variable d'instance se fait en fonction du type par lequel on accède à l'instance et non en fonction du type réel de l'instance.

```
class mere
{
   public String x = "mere";

}

class fille extends mere
{
   public String x = "fille";
}

class x
{
  static public void main(String [] argv)
    {
      fille f = new fille();
      mere m = f;

      System.out.println(f.x);
      System.out.println(m.x);
    }
}
```

Evocation de la mère dans le code d'une fille

Comme en C++, en l'intérieur d'une méthode, l'objet recepteur du message est identifié par la variable `this`. L'instance de la classe mère est elle identifiée par le mot clef `super`. En java on ne peut évoquer que la sur-classe supérieure, il est impossible de remonter plus haut dans l'arbre d'héritage.

```
class mere
{
```

```
  void methode()
    {
      System.out.println("Methode definie dans mere");
    }
}

class fille extends mere
{
  void methode1()
    {
      super.methode();
// appel a la methode de la mere
    }
}
```

Construction d'une mére lors de la création d'une fille

Comme en C++, lors de la construction d'une fille, une mère est nécessairement instanciée.

```
class mere
{
  mere()
    {
      System.out.println("Construction de la mere");
    }
}

class fille extends mere
{
 fille()
    {
      System.out.println("Construction de la fille");
    }
}

class exe1
{
  static public void main(String [] s)
    {
      fille f = new fille();
      // Construction de la mere
      // Construction de la fille
    }
}
```

De ce fait, il est toujours possible de convertir une instance d'une classe `fille` en une instance d'une classe `mere`.

9.5.3 Classes et méthodes abstraites

En C++, une classe abstraite est définie comme une classe ayant au moins une méthode virtuelle pure. En Java, le principe est le même une méthode peut être déclarée abstraite en utilisant le modificateur `abstract`. Dans ce cas, la classe contenant cette méthode ne peut être instancier. Le compilateur demande que l'on déclare explicitement comme `abstract` une classe contenant une méthode abstraite.

```
abstract class animal
{
   String name;
```

```
    abstract public void deplacement();
    void display()
      {
        System.out.println(this.name);
      }
}
```

Sur cette exemple, la classe `animal` est abstraite car il est impossible de définir le comportement `deplacement` d'un animal. Par contre, cette classe contient une méthode non abstraite, permettant d'afficher le nom de l'animal. Nous avons vu qu'il est possible en C++ de définir du code pour une méthode virtuelle pure. Ce code peut ensuite être utiliser dans le code des classes dérivées.

Une classe héritant d'une classe abstraite, doit encore être définit comme abstraite, si elle ne redéfinit toutes les méthodes abstraites de sa classe mére.

```
abstract class animal
{
    String name;
    abstract public void deplacement();
    void display()
      {
        System.out.println(this.name);
      }
}

public class poisson extends animal
{
  poisson()
    {
      super.name = "poisson";
    }
}
// class poisson must be declared abstract.
// It does not define void deplacement() from class animal.
```

Une classe doit être déclaré abstraite, si une de ces méthodes ne peut être définie, car on est dans l'incapacité de lui associer un code. Par contre, une classe abstraite en Java doit nécessairement contenir un code par défaut. Ce code par défaut, pouvant être la définition de certaines variables d'instances, le code de certaines méthodes ou fonction de classes. En java, il faut considérer qu'une classe abstraite définit une interface publique, mais aussi qu'elle contient du code. Si une classe abstraite décrit seulement une interface publique il est préférable d'en faire une `interface` au sens de java.

Méthode et classe finale

Nous avons vu en C++, les problèmes rencontrés en C++, lors de l'utilisation des liaisons statiques pour les fonctions. Nous avons décidé de manière conventionnelle, que si une fonction ne devait plus être redéfinie nous ne mettons plus le mot clef `virtual` devant la déclaration de son prototype. Ceci ne peut être en aucun cas vérifier par le compilateur. En java, il existe le modificateur `final` permettant d'indiquer qu'une méthode n'a pas le droit dêtre redéfinie dans les sousclasses. Ce modificateur peut aussi faire partie de la déclaration d'une classe, dans ce cas une classe finale ne pourra avoir d'héritières.

```
class mere
{
  final void methode()
    {
      System.out.println("Une methode finale");
    }
}
```

```
class fille extends mere
{
  void methode()
    {
      System.out.println("Il fallait essayer");
    }
//Final methods can't be overriden.
//Method void methode() is final in class mere.
}
```

Une utilisation possible concerne des problèmes de sécurité. Imaginons qu'il existe une classe `user` contenant la méthode `give_passwd()`, permettre la spécialisation de cette méthode peut s'avérer dangereux.

Une autre utilisation du modificateur `final` est de permettre de définit des variables ou d'instances ou de classe constantes.

```
class essai
{
  final int i = 4;
  static final int vclasse = 3;

  void methode()
    {
      i = 5;
      //Can't assign a value to a final variable: i

      vclasse = i;
      // Can't assign a value to a final variable: vclasse
    }
}
```

9.5.4 Héritage multiple d'interfaces

L'héritage multiple en C++, pose de nombreux problèmes comme la sélection d'une méthode définie dans plusieurs classes soeurs, l'unicité des variables d'instances dans le cas des héritages en diamant. Nous ces problèmes sont résolus si les classes impliquées dans la partie supérieure du graphe d'héritage sont virtuelles pures et ne définissent aucun contexte. Dans ce cas, ces classes virtuelle pure ne servent qu'à définir des interfaces publiques. Les sous classes ayant en charge de donner une implémentation de cette interface si elle veulent être instanciable.

Interface en Java

Java a retenu cette solution, pour avoir tous les avantages de l'héritage public sans en avoir les incovénients. Une classe virtuelle pure s'appelle en Java une *interface*. Dans la déclaration d'une classe, le mot clef `interface` remplace le mot clef `class`.

```
interface animal
{
  void deplace();
  void mange(int);
}

interface mamifere extends animal
{
  void allaite();
}

interface oiseau extends animal
{
  void vole();
}
```

```
class chauvesouris implements oiseau, mamiferes
{
  void deplace(){}
  void mange(int){}
  void vole(){}
  void allaite()
}
```

Le type `animal` est représenter par une interface, il est impossible de créer des instances de animal en utilisant l'instruction `new animal()`. Les types `mamifere` et `oiseau` sont des extensions de l'interface `animal`, ils sont aussi représentés par des interfaces. Enfin la classe `chauvesouris` assure la concrétisation des interfaces `mamifere` et `oiseau` en définissant le code. Il s'agit dans ce cas, d'un héritage multiple, car les instances de `chauvesouris` peuvent être implicitement converties en instance de `mamifere` ou de `oiseau` (`mamifere = new chauvesouris` ou `oiseau = new chauvesouris`).

```
interface A {}
interface B {}
interface C extends A,B {}
class D implements A{}
class E extends D, implements B{}
```

Une interface peut elle même héritée de plusieurs interfaces (cf. `interface C`). Il est possible à la fois d'hériter simplement de l'implementation d'une classe et aussi d'hériter de l'interface de plusieurs implementation (cf. `class E`).

9.5.5 Utilisation des interfaces

La grande utilisation des interfaces est de permettre aux applications d'être manipulées depuis l'extérieur par une interface. L'exemple le plus classique et l'écriture d'un environement graphique. Tout le code cet application est écrit en utilisant le type `forme_geometrique` qui est une interface. L'extérieur à la possibilité d'étendre les fonctionnalités en créant diverses concrétisation de `forme_geometrique` comme `carre, cercle, ....`

Une autre application possible peut être l'impression d'un texte.

```
interface outputPeriph
{
  void writeString(String);
  void newline();
  void newpage();
}

class texte
{
    void write(outputPeriph P)
     {
         P.newpage();
         P.writeString();
     }
}
```

Sur cet exemple, si l'on concrétise l'interface `outputPeriph` en créant des imprimantes, des écrans, ... Une instance de la classe `texte` pourra être afficher sur l'ensemble de ces périphériques sans aucune modification de code. Le polymorphisme peut être implémenté en Java, soit en utilisant des classes abstraites soit des classes concrètes.

9.6 Les exceptions en Java

Le principe des exceptions en Java est tout à fait similaire à celui de C++. On utilise d'ailleurs les mêmes mot clefs `throw` pour lever une exception et `catch` pour capturer une exception. Comme en C++, l'association

entre la levée de l'exception et la capture de l'exception se fait uniquement en fonction du type. La remonter récursive dan les contextes appellant afin de trouver un traitement est aussi effectuée. De même les clauses `catch` sont évaluées séquentiellement de la première à la dernière. La première qui convient est la seule qui est traitée.

En langage Java, il existe une classe `Throwable` et une classe `Exception` qui hérite de `Throwable`. Pour qu'un nouveau type soit utilisable dans une clause `throw` il faut qu'il hérite au minimum de `Throwable`, il est préférable par souci des convenances de la faire héritée de `Exception`. La clause `catch(...)` de C++ peut dans ce cas être remplacée par une clause `catch(Exception e)`.

```
class MyException extends Exception
{
  MyException(String name, Object value)
    {
      super("My message" + name);
      ......
    }
}

class essai
{
  void methode1()
    {
      try
        {
          this.methode();
        }
      catch(MyException e){}
      catch(Exception e){}
    }

  void methode() throws MyException
    {
      throw new MyException("is in the class essai", this);
    }

}
```

Sur cette exemple un nouveau type d'exception `MyException` est défini. Autrement le principe est exactement le même que celui présenté en C++.

```
public class except1
{

  void f(){}
  void methode1()
    {
      except1 e;
      try
        {
          e = new except1();
        }
      catch(Exception exp){}
      e.f();
// Variable e may not have been initialized.

    }
}
```

La grande rigueur de java est ici illustrée. Bien que la variable `e` est correctement créé, java déclenche une erreur car il ne peut être sur de l'initialisation. En effet, la variable `e` a été initialisé dans un bloc contenant

un traitement d'exception, comme java ne sait pas si l'instruction `e = new except1();` a pu réelement être effectuée ou si une exception a créer un déroutement avant son exécution, il génère une erreur.

Contrairement ++, une méthode ou une fonction de classe qui ne donne aucune liste d'exception ne peut en lever aucune. De ce fait, le compilateur est capable de vérifier qu'une fonction ou méthode remplira correctement son contrat en ne levant que les exceptions déclarées dans sa signature.

```
class MyException extends Exception
{
  MyException(String name, Object value)
    {
      super("My message" + name);
    }
}

class essai
{
  void m()
    {
      m1();
// Exception MyException must be caught, or it
// must be declared in the throws clause of this method
    }

  void m1() throws MyException
    {
    }
}
```

9.6.1 Héritage et exception

Comme en C++, la liste des exceptions d'une fille doit être inférieure ou égale à celle de la mère. C'est à dire qu'une fille ne peut lever plus d'exception que sa mère. Considérons le cas des interfaces et de leur concrétisation

9.6.2 Tout hérite de `Object`

.

```
public final Class getClass()
Returns the Class of this Object. Java has a runtime representation
for classes- a descriptor of type Class- which the method getClass();
returns for any Object.

public boolean equals(Object obj)
Compares two Objects for equality. Returns a boolean that indicates
whether this Object is equivalent to the specified Object. This
method is used when an Object is stored in a hashtable.
       Parameters:
              obj - the Object to compare with
       Returns:
              true if these Objects are equal; false otherwise.

protected Object clone();
Creates a clone of the object.
A new instance is allocated and a bitwise clone of the current object
is place in the new object.
```

```
public String toString()
public final void notify();
public final void notifyAll();
public final void wait(long timeout);
public final void wait(long timeout,int nanos);
public final void wait();

protected void finalize()
Code to perform when this object is garbage collected. The default is
that nothing needs to  be performed. Any exception thrown by a
finalize method causes the finalization to halt. But otherwise, it is
ignored.
```

Chapter 10

Patterns

10.1 Notation

Nous allons décrire les différents diagrammes illustrant les relations et les interactions concernant les classes et les objets. Nous nous baserons pour cela sur *"Design Pattern (E. Gamma, R. Helm, R. Johnson, J. Vlissides) chez Addison-Wesley dans la série Profesional Computing"*, lesquel se basent sur OMT. Ces diagrammes se répartissent en trois catégories.

1) Le **diagramme de Classe**, décrit les classes, leur structure et les relations statiques existantes entre elles.

2) Le **diagramme d'objet** décrit un objet particulier durant l'exécution du programme

3) Le**diagramme d'interaction** montre le flots des messages transitants entre les objets.

Notons que ces diagrammes seront souvent incomplets, on ne montrera généralement que les partie importantes dans un cadre donné.

10.1.1 Le diagramme de classe

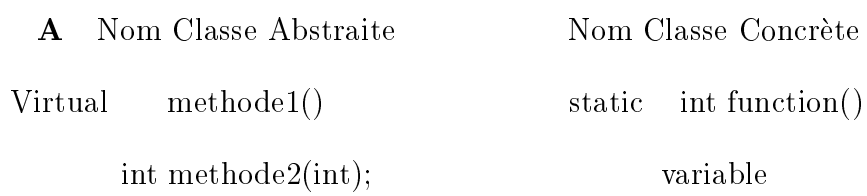
Nous allons tout d'abord voir comment sont représent/'ees les classes à l'aide de cette notation. Une classe s'identifie par un *type*, des *opérations clefs* et des *attributs*, elle sera donc représentée par une boite divisée en trois parties :

1) **L'entête:** Elle contient le nom de la classe, ou son type. On fera précéder ce nom d'un **A** lorsque la classe est abstraite (dans la litérature, on utilise parfois une fonte italique).

2) **Le comportement:** définie les *fonctions membres*, ou *méthodes*, de la classe en donnant la liste des *signatures* de ces fonctions. On adoptera pour ces signatures la syntaxes C++ mais elle pourront être plus ou moins complètes. On pourra également signaler, toujours avec cette syntaxe, si la fonction est *statique* (au sens *de classe*) ou *virtuelle* (i.e. *liées dynamiquement*).

3) **Les attributs:** On y définit les données *membres* ainsi que les variables *de classe*. On distingue ces dernières à l'aide du mot clef `static`. Là encore, il n'est pas obligatoire de définir le type complet de la variable, l'évocation de son nom est suffisante.

La figure suivante illustre les différentes informations nécessaire à la définition d'une classe dans le modèle.



Sur cet exemple, `Nom Classe Abstraite` est une classe sans instance dont le comportement est défini par la fonction membre `methode1()` qui est liée dynamiquement et par `int methode2(int)` qui est liée

statiquement. Cette classe ne possède aucun attribut. La classe `Nom Classe Concrète` possède une fonction de classe `function`, elle ne définie aucun comportement pour ses instances, son seul attribut est `variable`.

Le but final du diagramme de classe, une fois la structure décrite et de préciser les relations existantes entre les classes. Les différentes relations exprimables par un diagramme de classe sont:

- **Héritage :** La relation d'héritage est traduite par une ligne verticale contenant un triangle. On dit alors que la classe rattachée à la base du triangle (`ClasseA`) *hérite* de celle ratachée au sommet (`ClasseB`). Il existe différents termes pour décrire cette relation: class mère/classe fille, sur classe/sous classe, ancêtre/descendant. On dit aussis parfois que (`ClasseA`) est la *super classe* de (`ClasseB`.

  Toutes les caractéristiques de la sur-classe se retrouvent au niveau de la sous-classe, on dit également que l'héritage traduit une relation de type **EST-UN** ou **ISA**[1]. Lorsqu'une méthode `methode()` de la sur-classe est explicitement déclarée au niveau de la sous-classe, il s'agit alors de la spécialisation de `methode()`. Lorsqu'une méthode est spécialisée, il est obligatoire qu'elle ait été déclarée comme liée dynamiquement au niveau de la sur-classe. Notons qu'il s'agit lá d'une règle qu'il est très important d'appliquer de manière systématique si l'on désire éviter les mauvaises surprises, en effet, les règles suivies par le compilateur pour résoudre les appel de méthodes sont, dans ce cas précis, assez fumeuses (ARM ch 10).

  Sur-classe

  Virtual meth1()
  meth2()

  att1

  Sous-classe

  meth1()

  Sur cet exemple, la sous-classe possède comme attribut le champs `att1`, et les méthodes `meth1, meth2`. La fonction membre `meth1` est redéfinie par la sous-classe.

- **L'agrégation :** Cette relation, dans son sens le plus global, indique qu'une classe est propriétaire ou est responsable des instances d'une autre classe, pour prendre un cas simple, une structure C contenant d'autres structures est une sorte d'agrégation. De manière générale, cette relation peut être traduite comme un objet *possède* ou *est composé* d'un autre objet. Cette relation induit que les durées de vie du composant et du composé sont égales. La relation d'agrégation se symbolise par une flêche avec à sa base un losange, la flêche est dirigée de la classe composée vers la classe composante. Il est possible de commenter la relation par le nom de l'attribut agrégé. Il est important de noter que l'on parle en terme de conception et non de syntaxe, une agrégation peut très bien s'implémenter au moyen d'une référence ou d'un pointeur vers le composant pour peu que le composé le gére correctement, la présence ou abscence d'un `&` ou d'un `*` ne donne donc pas, en général, d'indication sur la présence d'une relation d'agrégation.

  Composée Nom Composante

  Sur cet exemple, la classe `Composée` contient dans ces attributs une variable `nom` qui est instance de la classe `composante`.

[1] nous ne traiterons que le cas de l'héritage public

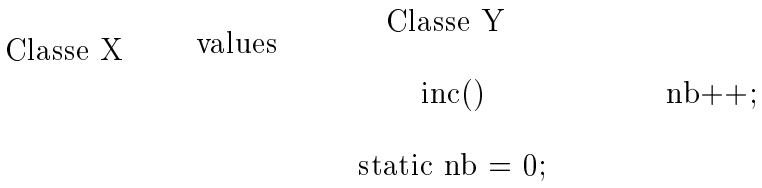
– **L'utilisation:** Cette relation traduit une accointance entre deux objets. Cela signifie qu'un instance d'une classe à besoin d'instance d'une autre classe pour réaliser complètement son comportement, (variable locale, passage de paramètre, retour de fonction, ...), on pourrait aussi dire qu'un objet en *utilise* un autre, mais, accointance, c'est plus joli (et puis ça en jette plus dans certains sallons, cela dit, vous mangez où vous voulez ...). La relation d'utilisation traduit un couplage plus faible entre deux classes que la relation d'agrégation. La relation d'agrégation est une sorte de relation d'utilisation. La relation d'utilisation est représentée par une simple flêche dirigée de la classe utilisatrice vers la classe utilisée.

Sur cet exemple, la classe `utilisatrice` à besoin de manipuler des instances de la classe `utilisée`. La relation d'utilisation ne documente que très pauvrement le type de relation existant entre deux classes.

– **L'instanciation :** La relation d'instanciation signifie dans son acception la plus générale, qu'une classe génère des instances d'une autre classe. La création d'instance peut être du à une relation d'agrégation, ou à une relation d'utilisation. Dans la suite, la signification que nous attacherons à l'instanciation sera de pouvoir exporter vers l'extérieur des instances d'une autre classe. La relation d'instanciation se symbolise par une flêche en pointillée dirigée de la classe instanciatrice vers la classe instanciée.

Il est à remarquer que la manière dont sont implémentées les relation d'utilisation et d'implémentation est indépendante de la sémantique de la relation. Par exemple, une relation de composition peut être implémentée par une référence à un objet de la classe `composante`. De même, une relation d'utilisation peut impliquer la création d'instance de la classe utilisée.

Le diagramme de classe peut être complèmenté par divers éléments comme une vague notion de cardinalité. Par exemple, l'extrémité d'une flêche, peut être complétée par un disque noir, la signification est alors la suivante, la relation addresse plusieurs instances. De même, un pseudo-code peut compléter la déclaration d'une fonction. La manière de fournir un pseudo code, consiste à definir un cercle au niveau de la fonction à commenter, le relier au pseudo-code par une ligne pointillée, le pseudo-code est à l'interieur d'une boîte, dont le coin supérieur droit est grisé.

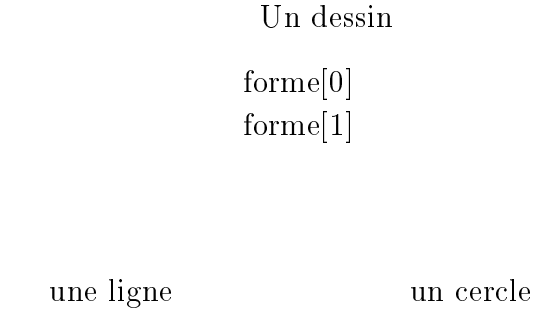


Sur cet exemple, la classe `X` possède un champs `values` qui référence plusieurs instances de la classe `Y`. Le pseudo-code de la fonction membre `inc` consiste à incrémenter la variable de classe `nb`.

10.1.2 Le Diagramme d'objets

Le diagramme d'objet représente une photographie des objets à un moment donné du programme. Pour différencier les objets des classes, ils sont systèmatiquement précédés de l'article indéfini *un*. Le symbole utilisé pour matérialiser un objet est un carré dont les angles ont été chanfrénés. Une flêche indique les objets référencés.

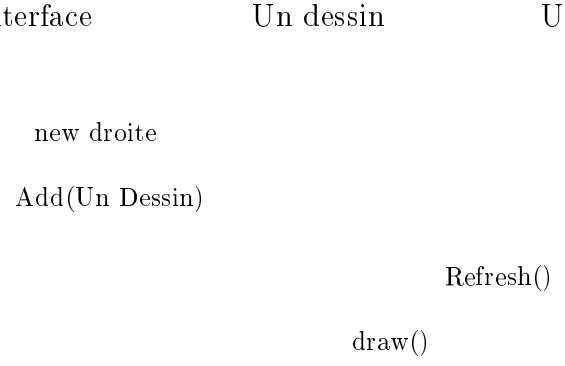
L'exemple suivant montre un objet `Un dessin` qui est composé de deux formes, une des formes est l'objet `un cercle` et l'autre forme est l'objet `une ligne`.



10.1.3 Le Diagramme d'interaction

Le diagramme d'interaction permet de suivre l'exécution d'un scénario au cours du temps. Il s'agit d'un scénario temporel, dans lequel les objets impliqués ainsi que les actions qu'ils recoivent ou déclenchent apparaissent.

Le temps s'écoule du haut du diagramme vers le bas. Une ligne solide indique la durée de vie d'un objet précis. Si l'objet qui intervient dans le scénario n'est pas encore créé, il est matérialisé par une ligne en pointillée. Le fait qu'un objet soit activé pendant une certaine période se matérialise par un rectangle sur sa ligne de vie. L'activation d'une méthode d'un objet receveur par un objet émetteur, se traduit par une flêche dirigée de l'émeteur vers le récepteur identifiée par la méthode utilisée. Si le récepteur et l'émetteur sont le même objet dans ce cas la flêche est une boucle.



Sur cet exemple, l'objet interface commence par instancier une nouvelle ligne, une fois l'instanciation effectuée, il déclenche la méthode `Add` de l'objet `Un dessin`. Cette méthode de l'objet `Un dessin` déclenche la méthode `Refresh()` sur l'objet `Un dessin`, elle déclenche ensuite la méthode `Draw` sur l'objet `Une ligne` qui compose l'objet dessin.

Le diagramme d'objet et le diagramme d'interaction sont deux diagrammes complémentaires.

10.2 Modèle Créationnels

Les modèles créationels abstraient le processus d'instanciation ce qui permet de rendre les système indépendant de la façon dont les objet sont crées représentés et combinés. On distingue le modèle créationnel d'**objets** qui délègue l'instanciation à un autre autre objet et le modèle créationnel de classe qui utilise l'héritage pour déléguer l'instanciation à une sous classe.

Ces modèles deviennent particulièrement important lorsque les systèmes se basent moins sur les relations d'héritages que sur les compositions d'objets, ce qui est souvent le cas lorsque la taille du système augmente. La principale raison étant que la composition d'objet est beaucoup plus souple que l'héritage, l'héritage

permet de cabler un nombre fixé de comportements, la composition permet de définir des comportement complexes à partir d'objets élémentaires. Si l'on garde à l'esprit que les objets ne sont généralement connus qu'à travers une interface abstraite, on comprendra l'intéret de pouvoir créer des objets répondant à certains critères sans avoir à ce préoccuper de la façon dont ces critères sont respectés.

Nous allons évoquer quatre modèles, ces modèles sont parfois concurrants, parfois complémentaires, nous y reviendront à la fin.

- Factory méthod (type classe): Aussis connu sous le nom de constructeur virtuel. L'idée, ici, est de remplacer des appel explicites à des constructeur par des appels à des méthodes virtuelles. On pourra donc changer les classes des objets crées en spécialisant ces méthodes.

- Abstract Factory (type objet): On utilise une usine pour créer des objets, ou produits. On ne connait de l'usine que sont interface, on ne connait donc pas le type précis des objets crées.

- Builder (type objet): Un **builder** reçoit une série d'instructions permettant de définir l'objet désiré, il chosit les "pièces détachées" nécéssaires et prend en charge leur assemblage.

- Prototype (type objet): Un objet s'obtient en dupliquant un objet existant dont la composition est plus ou moins connue.

10.3 Factory Method

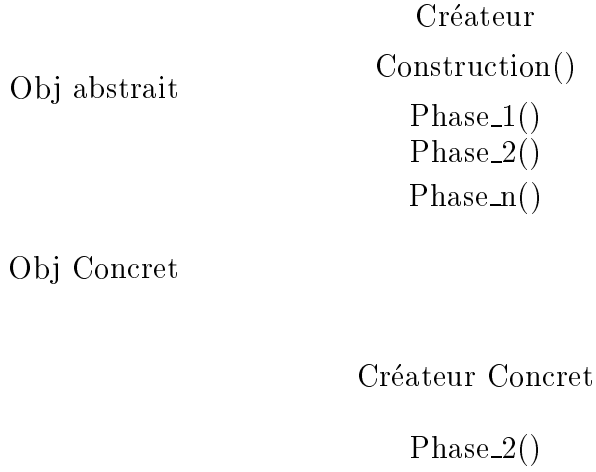
Les Factory Method s'utilisent lorsqu'une classe va devoir instancier et manipuler des objets dont elle connait une interface mais sur laquelle elle ne peut pas ou ne veut pas faire de supposition sur le type concret, soit parce que ce type peut varier d'une invocation à l'autre soit parce que l'on prévoit une évolution de ces types.

Une solution, pour cette classe, consiste donc à encapsuler ces instanciations dans des méthodes virtuelles. On obtiendra alors un code du genre :

```
Classe::methode()
{
   prod1 *p1 = creer_prod1();
   prod2 *p2 = creer_prod2();
   // manipuler p1 et p2 a travers
   // leur interface abstraite.
   ......
}
```

L'on peut obtenir différent types d'objet en spécialisant chacune des méthodes d'instanciation. Celle ci devront obligatoirement retourner un objet de type au moins égal à celui attendu (ce qui est alors vérifié par le compilateur).

Le principe général du modèle des factory méthods est alors le suivant:



Le rôle des créateurs est de produire des instances de la famille `Obj` en utilisant la méthode `Construction`. La classe `Créateur` fournit le code de la méthode de construction. Cette classe peut être soit concrète, soit abstraite. Si elle est concrète, il existe alors un `Obj standart`, qui est une instance de la classe `Obj abstrait`, elle définit alors un code par défaut pour chacune des phases de la construction. Toutes les classe de la famille `Créateur` non pas alors besoin de spécialiser toutes les phases de la création.

Si la classe `Créateur` est abstraite, dans ce cas, une des méthodes `Phase_i()` est virtuelle pure, est par conséquent il n'existe pas de produit standard, et donc la class `Obj abstrait` est une classe virtuelle. Chacune des sous-classes de créateur doit au moins spécialiser la méthode `Phase_i()`, mais elles peuvent en spécialiser d'autres. Une sous-classe de `Créateur` a pour rôle de spécialiser les chaînes de production. Elles ont toutes pour vocation de construire des `Obj Concret`.

Le rôle de la classe `Obj abstrait` est de fournir une interface commune aux `Obj`, cette classe n'est pas obligatoirement abstraite comme nous l'avons vu précédement.

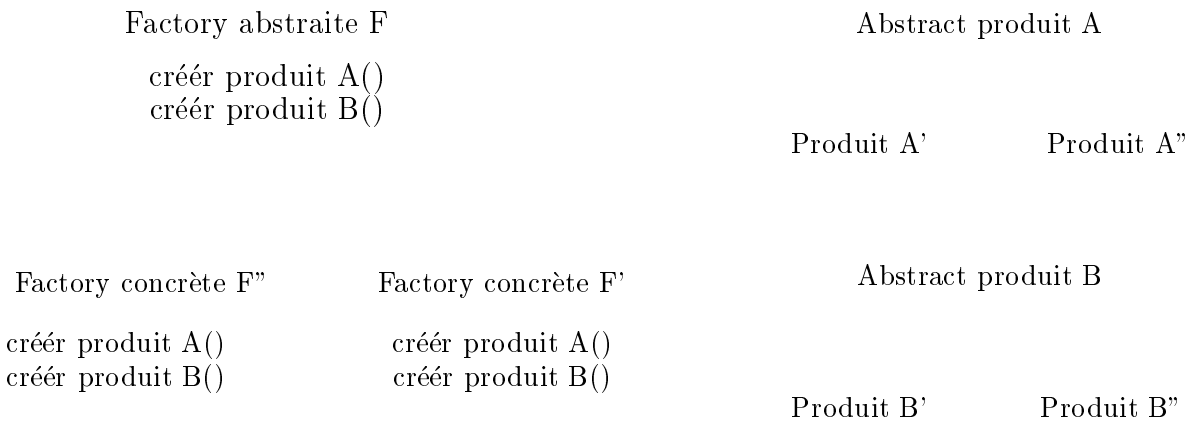
Le rôle de la classe `Obj Concret` est de fournir une implémentation de l'interface de `Obj abstrait`.

Le type de retour de la méthode `Construction()` est nécessairement de type `Obj abstrait`.

L'avantage des factory méthodes est de pouvoir réaliser une partie du code en termes d'`Obj Abstrait`, cette partie est entièrement indépendante de la manière dont sont créé les `Obj Concret`.

10.4 Abstract Factory

Le rôle du modèle créationnel **Abstract Factory** est de founir une interface à la création d'une famille de produit dépendants ou apparentés sans spécifier leur classe concrète. Considérons, une application qui peut sélectionner une famille de produit, lors de son démarrage. C'est à dire qu'elle a le choix entre le famille F', F''. Toutes ces familles ont la même forme à savoir être constituée des produits de type $\{A, B\}$. Les objets d'une même famille doivent collaborer, mais il est impossible de faire collaborer des objets de famille différentes. L'ensemble des familles est alors défini par $F' = \{A', B'\}$ et $F'' = \{A'', B''\}$. L'application est alors implémentée en terme de famille abstraite F qui doit fournir des objets abstraits de type A ou de type B.



La classe `Factory Abstract F` déclare une interface pour la création des produits appartenant à la famille. Chacune des classes `Factory Concrète` spécialise les fonctions de création des produits appartenant à sa famille spécifique. A l'arbre d'héritage des factories concrètes correspond un arbre d'héritage des produits.

Dans ce cas précis, l'abstract factory est implémentée en termes de factory méthodes. De même, que pour le modèle précédent, la classe `Factory Abstract F` n'est pas obligatoirement, une classe abstraite elle peut trés bien fournir un ensemble de produits standards. La contrainte sur les produits standards, est qu'ils peuvent collaborer avec tous les produits des autres familles. D'autres implémentations sont possibles pour les abstracts factory, comme par exemple une implémentation utilisant les prototypes. Il faut retenir qu'une abstract factory, est le représentant d'une famille de produits, le comportement de l'abstract factory est de pouvoir créer des instances de chacun des produits. Une abstract factory peut être utilisée quand:

- Un système doit être indépendant de la manière dont les produits sont créés, composés et représentés.
- Un système doit être configuré avec une famille particulière de produits.

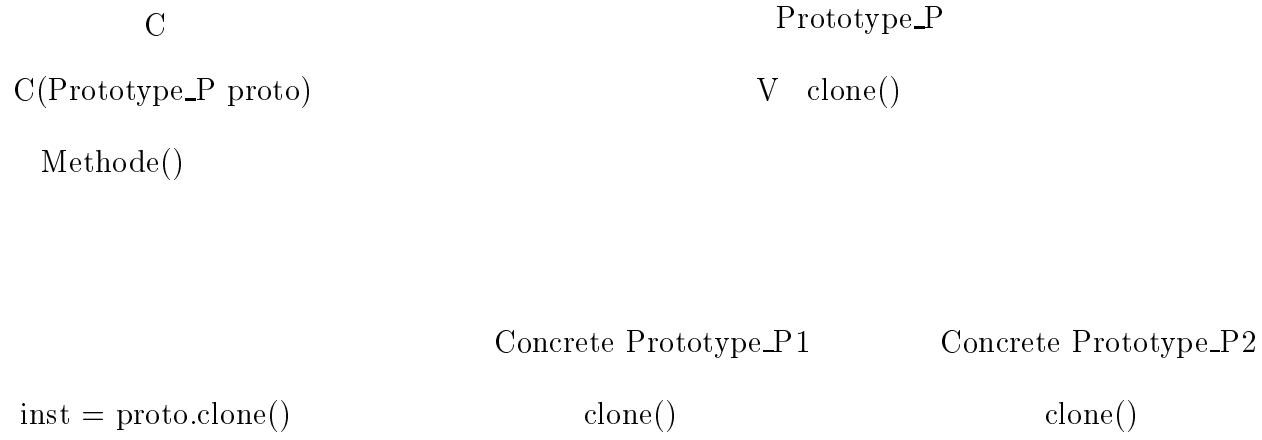
– Seuls les produits d'une même famille doivent travailler entre eux, et cette contrainte a besoin d'être renforcée.

10.5 Prototype

Le modèle des `prototype` permet d'instancier des objets d'une classe connaissant une instance particulière. Le protype ne duplique pas nécessairement une instance particulière, en effet l'objet dupliqué et son `clone` non pas obligatoirement le même état c'est à dire les mêmes valeurs d'attributs.

Une classe C définit un comportement en créant des instances d'un ensemble de catégories $P_1, \ldots, P_n$ de produits, chaque catégorie P_i est la racine d'une hiérarchie de produits $P_{i,1}, \ldots, P_{i,k}$. Si une seule fonction membre a besoin d'utiliser les produits, une simple relation d'utilisation suffit, par contre si les produits sont utilisés par plusieurs fonctions membres, il est plus avantageux d'établir une relation de composition entre C et l'ensemble des produits. Les instances de C peuvent donc utiliser un ensemble de produit appartenant au produit cartésien des catégories $\{P_{1,1}, \ldots, P_{1,k}\} \times \ldots \times \{P_{n,1}, \ldots, P_{n,l}\}$. Une solution serait de définir une abstract factory pour chaque éléments du produit cartésien, mais dans ce cas il y a une explosion du nombre de factory. Si l'application n'utilise que des instances C travaillant avec le même ensemble de produits, cet ensemble étant déterminé à l'exécution, on peut implémenter la factory avec des prototypes.

Dans le cas plus général, chaque catégorie de P_i possède une fonction de clonage `clone` qui est spécialisé par chacune des sous-classes $P_{i,1}, \ldots, P_{i,l}$. Lors de la création d'une instance de C ou leur de l'appel à la fonction spécifique un l'élement du produit cartésien des produits est utilisés. Cet élément sert alors de prototype pour créer les instances nécessaire.



Le schèma précédent n'utilise qu'une seule catégorie de produit P_1. Pour l'étendre il suffit de créer un nouvel arbre d'héritage par catégorie.

Les `prototypes` peuvent être utilisés lorsqu'un système doit être indépendant de la manière dont les produits sont créés, composés et représentés et lorsque :

– les classes à instancier sont spécifiées à l'exécution;

– les instances d'une classe peuvent avoir seulement un nombre restreint d'états possibles. Il est alors plus intéressant d'associer à chaque état un prototype plutôt de surcharger la fonction de création ou de refaire une sélection en fonction d'un identificateur d'état.

– Pour éviter de construire une hiérarchie de *abstract factory* similaire à la hiérarchie des produits.

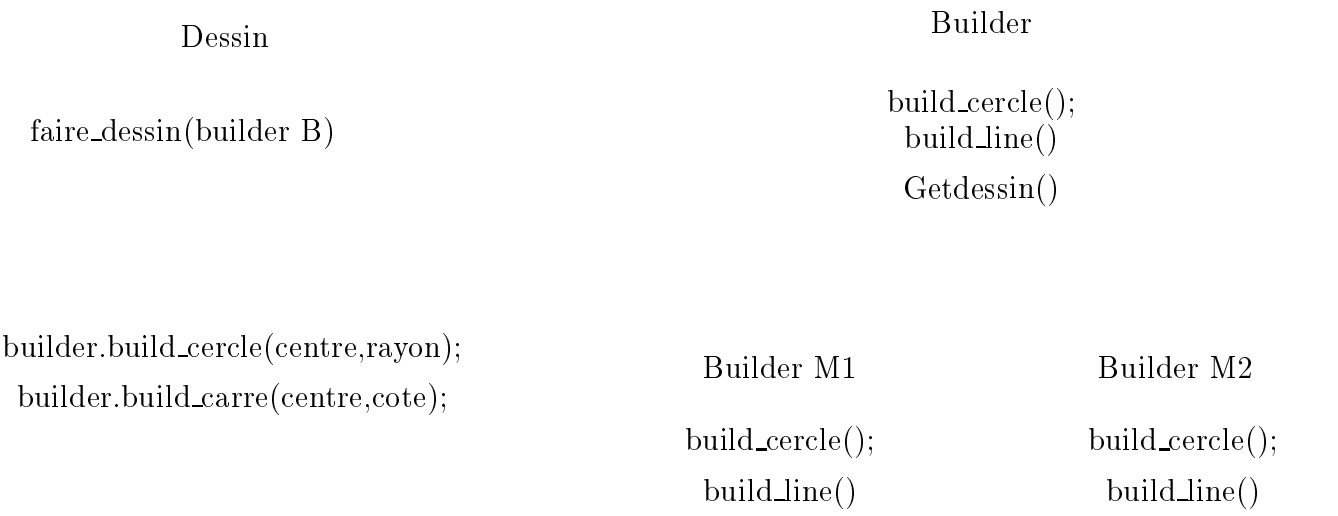
10.6 Builder

Le modèle créationnel **Builder** permet de séparer la construction d'un objet complexe de sa représentation, ainsi un même processus de construction peut créer des objets différents s'ils sont construits par un **builder** different.

Par exemple, je veux réaliser la création d'un dessin composé d'un cercle et d'un carre défini dans un langage comportant deux primitives graphique qui sont `cercle(), carre()`. Ce dessin peut être tracé

– sur une machine ne disposant que d'une seule primitive graphique qui est `point()`.

– sur une machine ayant deux primitives graphiques,`point(), line()`



Sur cet exemple, les objets créer par les builders sont unifiables ce n'est pas toujours le cas. Le rôle d'un builder est de définir les primitives de composition de l'objet, de mèmoriser les différents états de l'objet au cours de sa création, et de donner losque l'objet est totalement construit une référence sur l'instance.

Le diagramme d'interaction est alors le suivant:

Le **Builder** doit être utiliser lorsque

– l'algorithme de création d'un objet complexe doit être indépendant de la manière dont l'objet est construit;

– Un même processus de création peut générer des objets de nature différente pas obligatoirement unifiable par héritage sur un sur type.

10.7 Discussion et relations entre les différent modèles

10.7.1 Prolifération de classes

Nous avons vu quatre modèles permettant d'abstraire le processus de création. Le premier, Factory Method, consiste à encapsuler ces créations dans des méthodes virtuelles qui pourrons être eventuellement redéfinies. L'inconvenient de ce modèle est qu'il faut créer une nouvelle classe à chaque fois que l'on veut introduire un nouvel objet (avec, éventuellement, un effet cascade si le créateur est lui meme crée avec cette technique. Si si, regardez bien). De ce point de vue, Abstract Factory n'offre pas d'avantage flagrant si l'on choisit de créer un classe concrète par famille de produit. En fait, Abstract Factory n'a d'interêt, par rapport à Factory Method, que si elle existait déjà avant l'application ou si on en a besoin ailleurs (passer de Factory Method à Abstract Factory consiste simplement à extraire les méthodes de créations et à les regrouper dans une classe dédiées).

En fait, les seul modèle permettant d'éviter une prolifération de classes est Prototype, puisque une même classe peut accepter n'importe quelle famille de prototypes. Il faut remarquer que ce modèle peut être utilisé pour l'implémentation de tout les autres Modèle.

Le modèle Builder n'est généralement pas responsable d'une prolifération.

10.7.2 Complexité

Il peut sembler difficile de trouver des avantages objectifs à l'utilisation du modèle Factory Method. Il faut cependant remarquer qu'il est très simple à implémenter. Il constitue généralement une première approximation que l'on fait évoluer si besoin est.

Le modèle Builder à ceci d'intéressant qu'il cache beaucoup de complexité puisque qu'il masque non seulement l'instanciation mais aussi le processus de construction. Son interface n'est cependant pas toujours facile à réaliser.

Le modèle Prototype est probablement le plus souple. Si on regarde les autres modèles, on s'aperçois par exemple qu'il est assez difficile d'ajouter un nouvel élément dans la famille de produits ou de charger une nouvelle classe "au vol". Bien que ces problèmes n'aient pas étés évoqués, leur solutions passent généralement par une combinaison Prototype+édition de lien dynamique (en particulier pour des langages statiquement typé comme C++[2])

10.7.3 Combinaisons

L'utilisation de ces modèles n'est pas exclusive :

- Prototype peut être utilisé dans l'implementation de tout les autres modèles.
- Factory Method peut être utilisé pour implémenter Abstract Factory et Builder.
- Abstract Factory peut être utilisé pour implementer Builder.

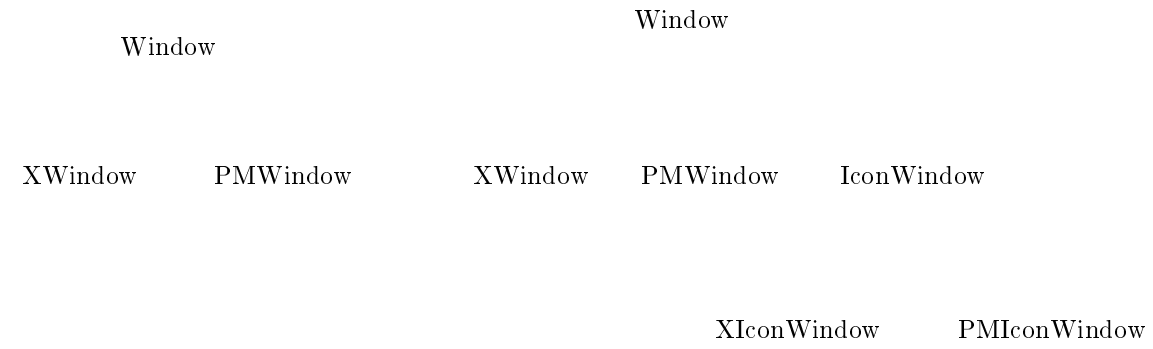
10.8 Modèle structurels

10.8.1 Le Bridge

Le modèle que nous allons aborder maintenant s'appelle un *bridge*. Le rôle du bridge est de séparer une abstraction de son implémentation afin que les deux parties puissent variées indépendement l'une de l'autre.

Lorsqu'une abstraction peut avoir plusieurs implémentations possibles, la manière habituelle de procéder consiste à utiliser l'héritage. Une classe abstraite définit alors l'interface qui est implémentée dans des classes concrètes héritant de la classe abstraite. Cette approche dans certains cas s'avère trop rigide, car l'héritage lié de manière permanente l'abstraction et ses implémentations.

Par exemple, on définit une abstraction représentant une window. Cette abstraction doit permettre son utilisation soit sous XWindow soit sous PMWindow. La manière la plus simple de procéder est d'utiliser l'héritage. On fait hériter Xwindow et PMwindow de window (cf partie a de la figure). Si maintenant on veut spécialiser les windows en utilisant des icones. Les iconwindow sont des windows particulières, dans ce cas il est nécessaire de faire hériter iconwindow de la classe window. Mais comme l'implémentation de la classe window est définie par héritage, l'on dot créér un arbre d'héritage sous iconwindow, qui représente l'implémentation du Xiconwindow et d'une PMwindow (cd partie b de la figure).



Cette profilifération de classe est due au fait que l'héritage traduit le raffinement d'une abstraction, mais qu'il est aussi utiliser pour implémenter l'abstraction.

La solution la meilleure est de séparer l'abstraction de son implémentation en utilisant un bridge. Il existe une hiérarchie de classe pour les abstractions et une hiérarchie de classe pour les implémentations.

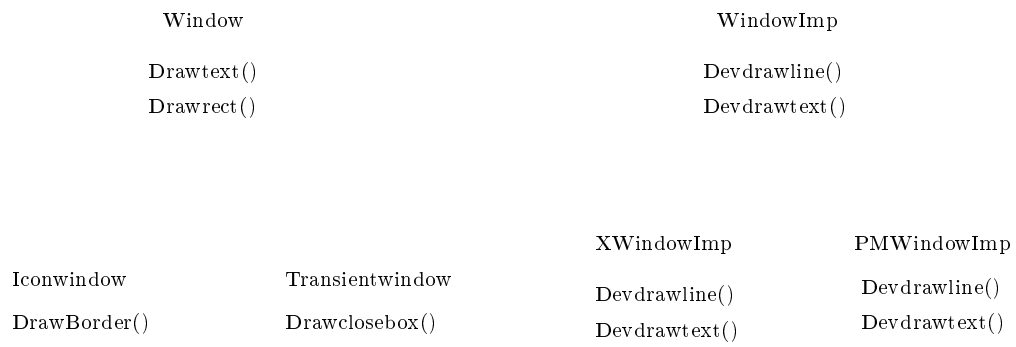
[2] Il faut cependant noter que des Langages comme Java, bien que statiquement typé, offre, dans le langage, une solution à ce problème.

La hiérarchie dímplémentation a comme racine une classe abstraite dont l'interface spécifie les méthodes permettant la réalisation des abstractions. Par exemple, la classe WindowImp définie comme comportement les méthodes `devdrawline(), devdrawtext()`. Les sous classe dérivant de cette sur classe spécialiseront ce comportement en fonction de leur propre spécificitée. Par exemple, la classe XwindowImp, définira le code de `devdrawline()` comme étant `Xdrawline()`.

Toutes les opérations concernant l'abstraction seront définies en utilisant seulement le comportement de WindowImp. Par exemple la méthode drawrect() de Window est:

```
window::drawrect()
{
   imp->devdrawline();
   imp->devdrawline();
   imp->devdrawline();
   imp->devdrawline();
}
```

Comme on le voit sur la figure suivante, l'arbre d'héritage des abstractions peut s'étendre indépendement de celui de l'implémentation.



Il faut utiliser les bridges:

- Pour éviter d'avoir une liaison permamente entre une abstraction et son implémentation.
- Lorsqu'à la fois l'interface et l'implémentation peuvent être raffiner par héritage.
- Eviter une prolifération de classe.

10.8.2 L'adaptater

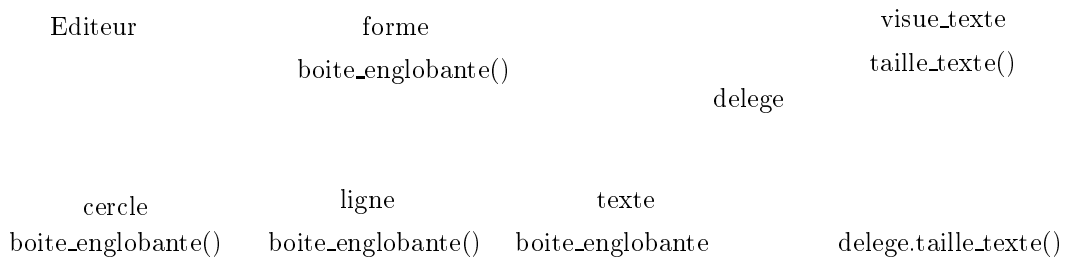
Un **adapter** sert à convertir l'interface d'une classe en celle attendue par le client. Un adapter permet à deux classes de collaborer alors qu'elles ne pourraient le faire sans lui. Un adapter peut être un modèle:

1) Objet si on utilise la délégation
2) Classe si on utilise l'héritage privé.

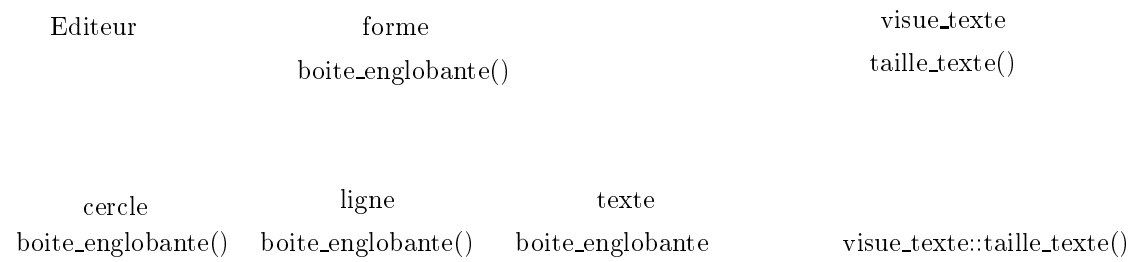
L'on veut développer un éditeur d'objet graphique, pour cela on définit une classe `forme` qui contient une méthode `boite_englobante`, les premiéres classes qui en héritent sont `ligne` et `cercle`. On dispose également un classe utilisable qui s'appelle `visue_texte` et qui contient une méthode `valeur_taille`. On dispose pour cette classe que de l'interface, le code étant indisponible. La méthode `valeur_taille` fournit le même service que `boite_englobante`. A ce stade du développement deux choix sont possibles:

1) Réécrire le code pour la méthode `boite_englobante` de la classe `texte`. Ce n'est pas une économie!!!
2) Essayer d'utiliser la classe `visue_texte`

Les deux possibilitées pour adapter la classe `visue_texte` sont résumées sur le schèma suivant.



IMPLEMENTATION UTILISATION LA DELEGATION



IMPLEMENTATION UTILISANT L'HERITAGE PRIVE

Les possiblités pour implémenter un adapter sont soit l'héritage privée soit la délégation. L'utilisation de l'héritage privé impose que les informations disponibles sont des classes C++. Par exemple, si on veut utiliser un module écrit en langage C, la seule solution pour implémenter un adapter est d'utiliser la délégation.

Ce modèle doit être utiliser:

1) vous voulez une utiliser une classe existante, mais son interface ne correspond pas à vos besoins;

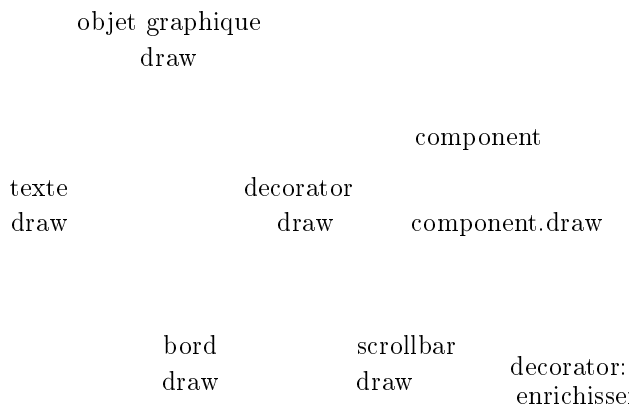
2) vous vouler créer une classe réutilisable qui coopèrent avec d'autres classes sans parentées et sans point commun et qui non pas une interface compatibles

3) (Objet adapatater seulement) vous avez besoin de plusieurs classes existantes, mais il est inacceptable d'adapter leur interface en héritant de chacune. Dans ce cas, un adapter objet peut adapter l'interface de la classe de ses parents.

10.8.3 Le décorateur

Un décorateur permet d'enrichir le comportement d'objet de base de manière dynamique. Il propose une alternative flexible à au raffinement par héritage pour étendre les fonctionnalités d'un objet.

On dispose, par exemple, d'une classe de base `texte` qui permet de visualiser un texte. On veut enrichir cette visualisation en lui adjoignant un `bord` et une `scrollbar`. L'utilisation de l'héritage oblige à créer une classe `scrollbar_texte`, une classe `bord_texte` et une classe `scrollbar_bord_texte`. On se retrouve avec autant de classe que le cardinal du produit cartesien (objet de base) $\times$ (enrichissement). Alors que l'utilisation oblige à créer un nombre de classe qui est égal à la somme des enrichissement et des objets de bases.



Sur cet exemple, on contruit un objet de base `text` on peut enrichir son comportement en utilisant un `decorator bord` qui lui même peut être enrichi en utilisant une `scrollbar`. Le code C++ est alors le suivant.

```
objetgraphique Obj = new scrollbar(new bord (new text)));
```

L'avantage des décorateurs est de pouvoir enrichir le comportement d'un objet durant l'exécution. Par exemple, si on reçoit un objet graphique, il est possible de retourner un objet graphique enrichi mais qui conserve le comportement de l'objet précédent.

Il faut utiliser un décorateur:

- Pour enrichir un objet dynamiquement, sans affecter le comportement des objets de la même classe.
- Pour surcharger certains comportement
- quand la spécialisation est impossible, car elle implique une explosion du nombre de classes

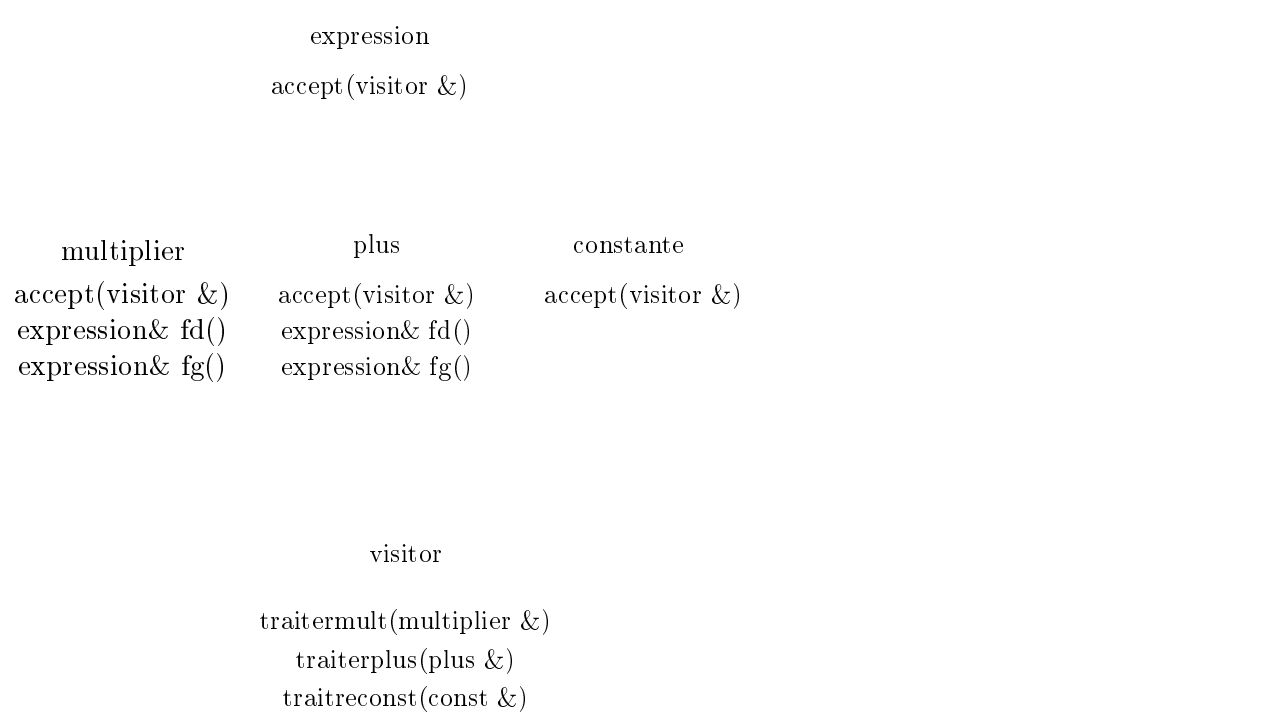
10.9 Les modèles comportementaux

10.9.1 Le modèle des visiteurs

Un visiteur représente une opération qui doit être exécutée sur les élements d'une structure d'objets. Le visiteur permet de définir une nouvelle opération sans changer les classes des éléments sur lequels il opére.

Les visiteurs permettent de retrouver le type instancié lors de la création d'un objet. Mais il permettent aussi de ne pas faire grossir de manière inconsidée une interface. Ils permettent de conserver une interface minimale. Pour que les visiteurs puissent correctement opérer il est nécessaire que les objets de la structure est une interface complète.

Si l'on reprend l'exemple des expressions, la méthode `C_Code` et `Lisp_Code` peuvent être supprimées de l'interface de `expression`, on utilisent alors les visiteurs. Chaque sous classe de `expression` spécialise sont acceptation de visiteur, par exemple la classe `constante` demande au visiteur d'appliquer la méthode qu'il a associé à constante. Chaque visiteur a une méthode spécifique à chaque classe composant la structure d'objets. En utilisant l'héritage on peut créer différents types de visiteurs.



Les visiteurs peuvent être appliqué lorsque:

- Une structure d'objets contient différentes classes d'objets et l'on veut réaliser une opération qui dépend de la classe concrète de l'objet.
- Plusieurs types d'opérations différenes peuvent être appliquées sur une structure d'objets. Et l'on ne veut pas polluer l'interface de ces objets en rajoutant des opérations qui dépendent d'une application particulière.
- Les visiteurs doivent être utiliser lorsque la structure d'objets est stabilisée, c'est à dire que l'ensemble des classes qui la compose ne devra plus évoluer. Sinon il est nécessaire de modifier tous les visiteurs pour qu'ils prennent en compte le nouveau type.

10.9.2 Le modèle de stratégie

C'est un modèle objet. Il permet de définir une famille d'algorithmes, chacun des algorithmes étant interchangeable avec les autres. La stratégie laisse l'algorithme varié indépendament du client qui l'utilise. Une stratégie consiste à encapsuler un algorithme dans une classe.Par exemple, il existe différents algorithmes pour diviser un texte en plusieurs lignes. Intégrer ces différents algorithmes dans la classe qui les utilisent soulève de nombreux problèmes.

1) Cela rend le code de la classe moins lisible, et cela peut le faire grossir de manière conséquente. En effet la selection du type de coupure doit se faire en fonction d'un certains contexte.
2) Si un client utilise qu'un seul type d'algorithme, il est inutile qu'il les possède tous dans son code.
3) Il est difficile de rajouter de nouveaux algorithmes de coupure ou de faire évoluer les algorithmes existants lorsqu'ils font partie intégrante du code du client.

La coupure d'un texte peut se faire en utilisant trois types de coupure

1) `SimpleCoupure` implémente un algorithme simple de coupure qui coupe tout les X caratères.

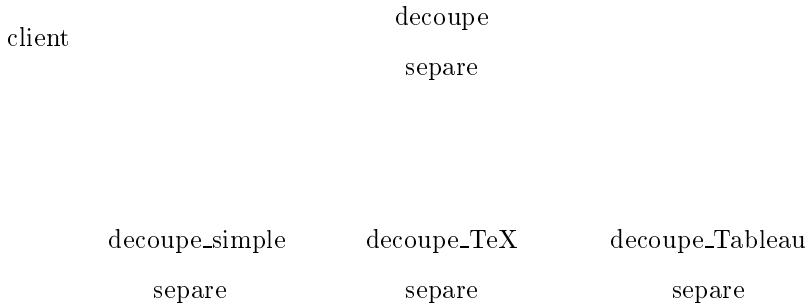
2) `Césure` implémente l'algorithme de TeX pour découper les lignes.

3) `CoupureTableau` implémente un algorithme qui détermine les coupures de telle manière que chaue colonne est un nombre fixé d'éléments.

Pour obtenir un modèle stratégie nous créons une classe abstraite `decoupe`, qui ne contient que la méthode `separe`. Puis nous utilisons l'héritage en définissant les sous-classes :

- `decoupe_simple` dont la méthode `separe` implémente l'algorithme `SimpleCoupure`.
- `decoupe_TeX` dont la méthode `separe` implémente l'algorithme `Césure`
- `decoupe_tableaux` dont la méthode `separe` implémente l'algorithme `CoupureTableau`

Si un client veut maintenant sélectionner un algorithme particulier de découpe, il lui suffit de contenir un objet de la classe qui implémente l'algorithme choisi. Ce modéle est un modèle objet, car on utilise l'objet pour atteindre l'algorithme.



Le modèle des stratégies doit être utilisée lorsque:

- Plusieurs classes apparentées diffèrent seulement dans leur comportement. La stratégie fournit une manière de configurer une classe avec un comportement particulier.
- Vous avez besoin de différentes variantes d'un algorithme. La stratégie peut être utiliser quand ces variantes sont implément'ees comme une hiérarchie de classes d'algorithmes.
- Un algorithme utilisent des données que le client n'a pas à connaître. Le modéle de stratégie évite de montrer les algorithmes spécifiques à certaines structures de données.
- Une classe possède plusieurs comportements qui sont sélectionnés par des conditions multiples lors de ses opérations.