COMPLEMENT DE COURS

PROGRAMMATION ORIENTEE OBJETS

LA2-T

II- LES SPECIFICITES DE LANGAGE C++

INTRODUCTION

C++ est un langage orienté objets dérivé du langage C. Le langage C a été développé dans les années 1970 par B. Kernighan et D. Ritchie pour en faire le langage de programmation structuré d système UNIX. Mais son utilisation est aujourd'hui beaucoup plus répandue. Il est employé pour l'écriture de logiciels dans des domaines divers (programmation scientifique, informatique, industrielle, gestion ,...) et avec des systèmes d'exploitation différents. Il est également renommé pour son efficacité (code compact et proche du langage machine).
A chaque langage sont associés des modèles de programmation qui présentent l'ensemble des techniques à appliquer lors de la conception et l'implémentation des programmes. Le langage C++ a été développé en 1983 par Bjarne Stroustrup, des laboratoires AT&T Bell. Il permet d'améliorer la qualité des programmes écrits en C, grâce à l'ajout des principes de la POO : la modularité, l'encapsulation des données, l'héritage, la surcharge des fonctions et le polymorphisme. Le passage du C au C++ correspond à une évolution naturelle de la programmation structurée classique vers la programmation orientée objets. C'est pourquoi le langage C++ représente de plus en plus la nouvelle manière de programmer des développeurs en C. Son apprentissage direct se justifie donc pleinement pour les programmeurs déjà familiarisés à la programmation classique.

LES COMMENTAIRES

Le langage C++ offre un nouvelle façon d'ajouter des commentaires.

En plus des symboles /* et */ utilisés en C, le langage C++ offre les symboles // qui permettent d'ignorer tout jusqu'à la fin de la ligne.

Exemple :

```
/* commentaire traditionnel
   sur plusieurs lignes
   valide en C et C++
*/

void main() { // commentaire de fin de ligne valide en C++
#if 0
     // une partie d'un programme en C ou C++ peut toujours
     // être ignorée par les directives au préprocesseur
     // #if  ....   #endif
#endif
}
```

Il est préférable d'utiliser les symboles // pour la plupart des commentaires et de n'utiliser les commentaires C (/* */) que pour isoler des blocs importants d'instructions.

Les ENTREES/SORTIES

Les entrées/sorties en langage C s'effectue par les fonctions *scanf* et *printf* de la librairie stand
ard <stdio.h>du langage C.
Il est possible d'utiliser ces fonctions pour effectuer les entrées/sorties de vos programmes, mais cependant les programmeurs C++ préfèrent les entrées/sorties par flux (ou flot ou stream).
Les deux flots sont prédéfinis lorsque vous avez inclus le fichier d'en-tête *iostream.h* :

- *cout* qui correspond à la sortie standard
- *cin* qui correspond à l'entrée standard

L'opérateur (surchargé) **<<** permet d'envoyer des valeurs dans un flot de sortie, tandis que **>>** permet d'extraire des valeurs d'un flot d'entrée.

Exemple :

```
 #include <iostream.h>

void main() {
   int i=123;
   float f=1234.567;
   char ch[80]="Bonjour\n", rep;

   cout << "i=" << i << "  f=" << f << "  ch=" << ch;
   cout << "i = ? ";
   cin >> i;         // lecture d'un entier
   cout << "f = ? ";
   cin >> f;         // lecture d'un réel
   cout << "rep = ? ";
   cin >> rep;       // lecture d'un caractère
   cout << "ch = ? ";
   cin >> ch;        // lecture du premier mot d'une chaîne
   cout << "ch= " << ch;  // c'est bien le premier mot ...
}
/*-- résultat de l'exécution ----------
i=123  f=1234.57  ch=Bonjour
i = ? 12
f = ? 34.5
rep = ? y
ch = ? c++ is easy
ch= c++
-------------------------------------------------*/
```

- tout comme pour la fonction *scanf*, les espaces sont considérés comme des séparateurs entre les données par le flux *cin*.
- notez l'absence de l'opérateur & dans la syntaxe du *cin*. Ce dernier n'a pas besoin de connaître l'adresse de la variable à lire.

LES CONVERSIONS EXPLICITES

En C++, comme en langage C, il est possible de faire des conversions explicites de type, bien que le langage soit plus fortement typé :

```
double d;
 int i;
 i = (int) d;
```

Le C++ offre aussi une notation fonctionnelle pour faire une conversion explicite de type :

```
double d;
 int i;
 i = int(d);
```

Cette façon de faire ne marche que pour les types simples et les types utilisateurs.

Pour les types pointeurs ou tableaux le problème peut être résolu en définissant un nouveau type :

```
double d;
 int *i;
 typedef int *ptr_int;
 i = ptr_int(&d);
```

La conversion explicite de type est surtout utile lorsqu'on travaille avec des pointeurs du type *void ** .

DEFINITION DE VARIABLES

En C++ vous pouvez déclarer les variables ou fonctions n'importe où dans le code.

La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant.

Ceci permet :

- de définir une variable aussi près que possible de son utilisation afin d'améliorer la lisibilité. C'est particulièrement utile pour des grosses fonctions ayant beaucoup de variables locales.
- d'initialiser un objet avec une valeur obtenue par calcul ou saisie.

Exemple :

```
#include <stdio.h>

void main() {
  int i=0;      // définition d'une variable
  i++;          // instruction
  int j=i;      // définition d'une autre variable
  j++;          // instruction
  int somme(int n1, int n2); // déclaration d'une fonction
  printf("%d+%d=%d\n", i, j, somme(i, j)); // instruction

  cin >> i;
  const int k = i; // définition d'une constante initialisée
```

```
                // avec la valeur saisie
}
```

VARIABLE DE BOUCLE

On peut déclarer une variable de boucle directement dans l'instruction *for*. Ceci permet de n'utiliser cette variable que dans le bloc de la boucle.

Exemple :

```
#include <iostream.h>

void main() {
  for(int i=0; i<10; i++)
    cout << i << ' ';
  // i n'est pas utilisable à l'extérieur du bloc for
}
/*-- résultat de l'exécution --------
0 1 2 3 4 5 6 7 8 9
-------------------------------------*/
```

VISIBILITE DES VARIABLES

L'opérateur de résolution de portée *::* permet d'accéder aux variables globales plutôt qu'aux variables locales.

```
#include <iostream.h>
int i = 11;

void main() {
    int i = 34;
    {  int i = 23;

       ::i = ::i + 1;
       cout << ::i << "  " << i << endl;
    }
    cout << ::i << "  " << i << endl;
}
/*-- résultat de l'exécution ------------------
12  23
12  34
```

L'utilisation abusive de cette technique n'est pas une bonne pratique de programmation (lisibilité). Il est préférable de donner des noms différents plutôt que de réutiliser les mêmes noms.

En fait, on utilise beaucoup cet opérateur pour définir hors d'une classe les fonctions membres ou pour accéder à un identificateur dans un espace de noms (cf espace de noms).

LES CONSTANTES

Les habitués du C ont l'habitude d'utiliser la directive du préprocesseur *#define* pour définir

des constantes.

Il est reconnu que l'utilisation du préprocesseur est une source d'erreurs difficiles à détecter.

En C++, l'utilisation du préprocesseur se limite aux cas les plus sûrs :

- inclusion de fichiers
- compilation conditionnelle.

Le mot réservé *const* permet de définir une constante.

L'objet ainsi spécifié ne pourra pas être modifié durant toute sa durée de vie. Il est indispensable d'initialiser la constante au moment de sa définition.

Exemple :

```
const int N = 10; // N est un entier constant.
const int MOIS=12, AN=1995; // 2 constantes entières
int tab[2 * N];  // autorisé en C++ (interdit en C)
```

VARIABLES REFERENCES

En plus des variables normales et des pointeurs, le C++ offre les variables références.
Une variable référence permet de créer une variable qui est un "synonyme" d'une autre. Dès lors, une modification de l'une affectera le contenu de l'autre.

```
int i;
int & ir = i; // ir est une référence à i
int *ptr;

i=1;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de :   i= 1  ir= 1

ir=2;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de :   i= 2  ir= 2

ptr = &ir;
*ptr = 3;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de :   i= 3  ir= 3
```

Une variable référence doit **obligatoirement** être initialisée et le type de l'objet initial doit être le même que l'objet référence.

Intérêt : passage des paramètres par référence

ALLOCATION DYNAMIQUE DE LA MEMOIRE

Le C++ met à la disposition du programmeur deux opérateurs *new* et *delete* pour

remplacer respectivement les fonctions *malloc* et *free* (bien qu'il soit toujours possible de les utiliser).

- **L'opérateur *new***

  L'opérateur **new** réserve l'espace mémoire qu'on lui demande et l'initialise. Il retourne l'adresse de début de la zone mémoire allouée.

```
int *ptr1, *ptr2, *ptr3;

// allocation dynamique d'un entier
ptr1 = new int;

// allocation d'un tableau de 10 entiers
ptr2 = new int [10];

// allocation d'un entier avec initialisation
ptr3 = new int(10);

struct date {int jour, mois, an; };
date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};

// allocation dynamique d'une structure
ptr4 = new date;

// allocation dynamique d'un tableau de structure
ptr5 = new date[10];

// allocation dynamique d'une structure avec initialisation
ptr6 = new date(d);
```

  L'allocation des tableaux à plusieurs dimensions est possible :

```
typedef char TAB[80]; // TAB est un synonyme de : tableau de 80 caractères
TAB *ecran;

ecran = new TAB[25]; // ecran est un tableau de 25 fois 80 caractères
ecran[24][79]='$';
ou
char (*ecran)[80] = new char[25][80];

ecran[24][79]='$';
```

Attention à ne pas confondre :

```
int *ptr = new int[10]; // création d'un tableau de 10 entiers
```

avec

```
int *ptr = new int(10); // création d'un entier initialisé à 10
```

- **L'opérateur *delete***

  L'opérateur **delete** libère l'espace mémoire alloué par *new* à un seul objet, tandis que l'opérateur **delete[]** libère l'espace mémoire alloué à un tableau d'objets.

  ```
  // libération d'un entier
  delete ptr1;

  // libération d'un tableau d'entier
  delete[] ptr2;
  ```

  L'application de l'opérateur *delete* à un pointeur nul est légale et n'entraîne aucune conséquence fâcheuse (l'opération est tout simplement ignorée).

  - A chaque instruction new doit correspondre une instruction delete.
  - Il est important de libérer l'espace mémoire dès que celui ci n'est plus nécessaire.
  - La mémoire allouée en cours de programme sera libérée automatiquement à la fin du programme.
  - Tout ce qui est alloué avec **new []**, doit être libéré avec **delete[]**

DECLARATION DES FONCTIONS

Le langage C++ impose au programmeur de déclarer le nombre et le type des arguments de la fonction.

Ces déclarations sont identiques aux prototypes de fonctions de la norme C-ANSI.

Cette déclaration est par ailleurs obligatoire avant utilisation (contrairement à la norme C-ANSI).

La déclaration suivante *int f1();* où *f1* est déclarée avec une liste d'arguments vide est interprétée en C++ comme la déclaration *int f1(void);*

(La norme C-ANSI considère que *f1* est une fonction qui peut recevoir un nombre quelconque d'arguments, eux mêmes de type quelconques, comme si elle était déclarée *int f1(...);* .)

- Une fonction dont le type de la valeur de retour n'est pas void, doit **obligatoirement** retourner une valeur.

PASSAGE PAR REFERENCE

En plus du passage par valeur, le C++ définit le passage par référence. Lorsque l'on passe à une fonction un paramètre par référence, cette fonction reçoit un "synonyme" du paramètre réel.

Toute modification du paramètre référence est répercutée sur le paramètre réel.

Exemple :

```
void echange(int &n1, int &n2) {
  // n1 est un alias du paramètre réel i
  // n2 est un alias du paramètre réel j
  int temp = n1;
  n1 = n2;   // toute modification de n1 est répercutée sur i
  n2 = temp; // toute modification de n2 est répercutée sur j
}

void main() {
  int i=2, j=3;

  echange(i, j);
  cout << "i= " << i << " j= " << j << endl;
  // affichage de :  i= 3  j= 2
}
```

Comme vous le remarquez, l'appel se fait de manière très simple.

- Utilisez les références quand vous pouvez,
- utiliser les pointeurs quand vous devez.

Cette facilité augmente la puissance du langage mais doit être utilisée avec précaution, car elle ne protège plus la valeur du paramètre réel transmis par référence.

L'utilisation du mot réservé **const** permet d'annuler ces risques pour les arguments de grande taille ne devant pas être modifiés dans la fonction.

```
struct FICHE {
  char nom[30], prenom[20], email[256];
};

void affiche(const FICHE  &f) {
  // passage par référence (plutôt que par valeur) pour des
  // raisons d'efficacité. Une modification de f provoque une
  // erreur de compilation
  cout << f.nom << " " << f.prenom;
  cout <<  " " << f.adresse << endl;
}

void main() {
  FICHE user = {
    "Dancel", "Alain", "alain.dancel@free.fr"
  };
  affiche(user);}
```

Références et pointeurs peuvent se combiner :

```
int ouverture(FILE *&f, const char *nf, const char *mode) {
// passage par référence d'un pointeur.
  f = fopen(nf, mode);
  return (f==null) ? -1 : 0;}
```

```
int main() {
  FILE *fic;

  if (ouverture(fic, "/tmp/toto.fic", "r") == -1)
    exit(1);

  int i;
  fscanf(fic, "%d", &i);
  // etc ...   }
```

VALEUR PAR DEFAUT DES PARAMETRES

Certains arguments d'une fonction peuvent prendre souvent la même valeur.

Pour ne pas avoir à spécifier ces valeurs à chaque appel de la fonction, le C++ permet de déclarer des valeurs par défaut dans le prototype de la fonction.

Exemple :

```
void print(long valeur, int base = 10);

void main() {
    print(16);    // affiche 16    (16 en base 10)
    print(16, 2); // affiche 10000 (16 en base 2)
}

void print(long valeur, int base){
    cout << ltostr(valeur, base) << endl;}
```

- Les paramètres par défaut sont obligatoirement les derniers de la liste.
- Ils ne sont déclarés que dans le prototype de la fonction et pas dans sa définition.

FONCTION *inline*

Le mot clé *inline* remplace avantageusement l'utilisation de *#define* du préprocesseur pour définir des pseudo-fonctions.

Afin de rendre l'exécution plus rapide d'une fonction et à condition que celle ci soit de courte taille, on peut définir une fonction avec le mot réservé *inline*.

Le compilateur générera, à chaque appel de la fonction, le code de celle ci. Les fonctions *inline* se comportent comme des fonctions normales et donc, présentent l'avantage de vérifier les types de leurs arguments, ce que ne fait par la directive *#define*.

Exemple :

```
#include <iostream.h>
inline int carre(int n); // déclaration
void main() {
```

```
  cout << carre(10) << endl;}

// inline facultatif à la définition, mais préférable
inline int carre(int n) {
  return n * n;}
```

contrairement à une fonction normale, la portée d'une fonction *inline* est réduite au module dans lequel elle est déclarée.

SURCHARGE DE FONCTIONS

Une fonction se définit par :

- son nom,
- sa liste typée de paramètres formels,
- le type de la valeur qu'elle retourne.

Mais seuls les deux premiers critères sont discriminants. On dit qu'ils constituent la **signature** de la fonction.

On peut utiliser cette propriété pour donner un même nom à des fonctions qui ont des paramètres différents :

```
int somme( int n1, int n2)
      { return n1 + n2; }

int somme( int n1, int n2, int n3)
      { return n1 + n2 + n3; }

double somme( double n1, double n2)
      { return n1 + n2; }

void main() {
  cout << "1 + 2 = " << somme(1, 2) << endl;
  cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
  cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;}
```

Le compilateur sélectionnera la fonction à appeler en fonction du type et du nombre des arguments qui figurent dans l'appel de la fonction.

Ce choix se faisant à la compilation, fait que l'appel d'une fonction surchargée procure des performances identiques à un appel de fonction "classique".

On dit que l'appel de la fonction est résolu de manière statique.

Autres exemples :

```
enum Jour {DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI};

enum Couleur {NOIR, BLEU, VERT, CYAN, ROUGE, MAGENTA, BRUN, GRIS};

void f1(Jour j);
```

```
void f1(Couleur c);
// OK : les types énumérations sont tous différents
```

```
void f2(char *str) { /* ... */ }
void f2(char ligne[80]) { /* ... */ }
// Erreur: redéfinition de la fonction f
// char * et char [80] sont considérés de même type
```

```
int somme1(int n1, int n2) {return n1 + n2;}
int somme1(const int n1, const int n2) {return n1 + n2;}
// Erreur: la liste de paramètres dans les déclarations
//         des deux fonctions n'est pas assez divergente
//         pour les différencier.
```

```
int somme2(int n1, int n2) {return n1 + n2;}
int somme2(int & n1, int & n2) {return n1 + n2;}
// Erreur: la liste de paramètres dans les déclarations
//         des deux fonctions n'est pas assez divergente
//         pour les différencier.
```

```
int    somme3(int n1, int n2) {return n1 + n2;}
double somme3(int n1, int n2) {return (double) n1 + n2;}
// Erreur: seul le type des paramètres permet de faire la distinction
//         entre les fonctions et non pas la valeur retournée.
```

```
int somme4(int n1, int n2) {return n1 + n2;}
```

```
int somme4(int n1, int n2=8) {return n1 + n2;}

// Erreur: la liste de paramètres dans les déclarations
//         des deux fonctions n'est pas assez divergente
//         pour les différencier.
```

```
typedef int entier;      // entier est un alias de int

int somme5(int n1, int n2) {return n1 + n2;}

int somme5(entier e1, entier e2) {return e1 + e2;};

// Erreur : redéfinition de somme5 : des alias de type ne
//  sont pas considérés comme des types distincts
```

RETOUR D'UNE REFERENCE

Fonction retournant une référence

Une fonction peut retourner une valeur par référence et on peut donc agir, à l'extérieur de cette fonction, sur cette valeur de retour.

La syntaxe qui en découle est plutôt inhabituelle et déroutante au début.

```
#include <iostream.h>
int t[20];  // variable globale -> beurk !

int & nIeme(int i) {
  return t[i];
}

void main() {
  nIeme(0) = 123;
  nIeme(1) = 456;
  cout << t[0] << " " << ++nIeme(1);
}
// -- résultat de l'exécution -----------
// 123 457
```

- Tout se passe comme si *nIeme(0)* était remplacé par *t[0]*.
- Une fonction retournant une référence (non constante) peut être une *lvalue*.
- La variable dont la référence est retournée doit avoir une durée de vie permanente.

En C, pour réaliser la même chose, nous aurions du écrire :

```
int t[20];  // variable globale -> beurk !
```

```
int * nIeme(int i) {
  return &t[i];
}

void main() {
  *nIeme(0) = 123;
  *nIeme(1) = 456;
  printf("%d %d\n", t[0] ++(*nIeme(1)) );
}
// -- résultat de l'exécution ----------
// 123 457
```

ce qui est moins lisible et pratique ...

Retour d'une référence constante

Afin d'éviter la création d'une copie, dans la pile, de la valeur retournée lorsque cette valeur est de taille importante, il est possible de retourner une valeur par référence constante.

Le préfixe *const*, devant le type de la valeur retournée, signifie au compilateur que la valeur retournée est constante.

```
const Big & f1() {
  static Big b;    // notez le static ...
  // ...
  return b;
}
void main() {
  f1() = 12;  // erreur
}
```

III- L'ENCAPSULATION

RAPPEL

la classe décrit le modèle structurel d'un objet :

- ensemble des attributs (ou champs ou données membres) décrivant sa structure
- ensemble des opérations (ou méthodes ou fonctions membres) qui lui sont applicables.

Une classe en C++ est une structure qui contient :

- des fonctions membres
- des données membres

Les mots réservés *public* et *private* délimitent les sections visibles par l'application.

Exemple :

```
class Avion {
  public :   // fonctions membres publiques
    void init(char [], char *, float);
    void affiche();
  private :  // membres privées
    char  immatriculation[6], *type; // données membres privées
    float poids;
    void  erreur(char *message);     // fonction membre privée
};  // n'oubliez pas ce ; après l'accolade
```

DROITS D'ACCES

L'**encapsulation** consiste à masquer l'accès à certains attributs et méthodes d'une classe.

Elle est réalisée à l'aide des mots clés :

- ***private*** : les membres privés ne sont accessibles que par les fonctions membres de la classe. La partie privée est aussi appelée **réalisation**.
- ***protected*** : les membres protégés sont comme les membres privés. Mais ils sont aussi accessibles par les fonctions membres des classes dérivées (voir l'héritage).
- ***public*** : les membres publics sont accessibles par tous. La partie publique est appelée **interface**.

Les mots réservés *private* , *protected* et *public* peuvent figurer plusieurs fois dans la déclaration de la classe.

Le droit d'accès ne change pas tant qu'un nouveau droit n'est pas spécifié.

RECOMMANDATIONS DE STYLE

Mettre :

- la première lettre du nom de la classe en majuscule
- la liste des membres publics en premier
- les noms des méthodes en minuscules
- le caractère _ comme premier caractère du nom d'une donnée membre

DEFINITION DES FONCTIONS MEMBRES

En général, la déclaration d'une classe contient simplement les prototypes des fonctions membres de la classe.

```
class Avion {
  public :
    void init(char [], char *, float);
    void affiche();
  private :
    char  _immatriculation[6], *_type;
    float _poids;
    void  erreur(char *message);
};
```

Les fonctions membres sont définies dans un module séparé ou plus loin dans le code source.

Syntaxe de la définition hors de la classe d'une méthode :

```
type_valeur_retournée Classe::nom_méthode( paramètres_formels )
{
   // corps de la fonction
}
```

Dans la fonction membre on a un accès direct à tous les membres de la classe.

Exemple de définition de méthode de la classe Avion :

```
void Avion::init(char m[], char *t, float p) {
  if ( strlen(m) != 5 ) {
    erreur("Immatriculation invalide");
    strcpy(_immatriculation, "?????");
  }
  else
    strcpy(_immatriculation, m);
  _type = new char [strlen(t)+1];
  strcpy(_type, t);
  _poids = p;
}

void Avion::affiche() {
  cout << _immatriculation << "  " << _type;
  cout << "  " << _poids << endl;
}
```

La définition de la méthode peut aussi avoir lieu à l'intérieur de la déclaration de la classe.

Dans ce cas, ces fonctions sont automatiquement traitées par le compilateur comme des fonctions *inline*.

Une fonction membre définie à l'extérieur de la classe peut être aussi qualifiée explicitement de fonction *inline*.

Exemple :

```
class Nombre {
  public :
    void setnbre(int n) { nbre = n; } // fonction inline
    int  getnbre() { return nbre; }   // fonction inline
    void affiche();
  private :
    int nbre;
};

inline void Nombre::affiche() {   // fonction inline
  cout << "Nombre = " << nbre << endl;
}
```

Rappel :

la visibilité d'une fonction *inline* est restreinte au module seul dans laquelle elle est définie.

INSTANCIATION D'UNE CLASSE

De façon similaire à une *struct* ou à une *union*, le nom de la classe représente un nouveau type de donnée.

On peut donc définir des variables de ce nouveau type; on dit alors que vous créez des **objets** ou des **instances** de cette classe.

Exemple :

```
Avion av1;     // une instance simple (statique)
Avion *av2;    // un pointeur (non initialisé)
Avion compagnie[10];  // un tableau d'instances
av2 = new Avion; // création (dynamique) d'une instance
```

UTILISATION DES OBJETS

Après avoir créé une instance (de façon statique ou dynamique) on peut accéder aux attributs et méthodes de la classe.

Cet accès se fait comme pour les structures à l'aide de l'opérateur **.** (point) ou -> (tiret supérieur).

Exemple :

```
av1.init("FGBCD", "TB20", 1.47);
av2->init("FGDEF", "ATR 42", 80.0);
compagnie[0].init("FEFGH","A320", 150.0);
av1.affiche();
av2->affiche();
compagnie[0].affiche();
av1.poids = 0; // erreur, poids est un membre privé
```

EXEMPLE COMPLET : PILE D'ENTIERS

CONSTRUCTEURS ET DESTRUCTEURS

Les données membres d'une classe ne peuvent pas être initialisées; il faut donc prévoir une méthode d'initialisation de celles-ci (voir la méthode *init* de l'exemple précédent).

Si l'on oublie d'appeler cette fonction d'initialisation, le reste n'a plus de sens et il se produira très certainement des surprises fâcheuses dans la suite de l'exécution.

De même, après avoir fini d'utiliser l'objet, il est bon de prévoir une méthode permettant de détruire l'objet (libération de la mémoire dynamique ...).

Le **constructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à l'instanciation de l'objet, assurant ainsi une initialisation correcte de l'objet.

Ce constructeur est une fonction qui porte comme nom, le nom de la classe et qui ne retourne pas de valeur (pas même un *void*).

Exemple :

```
class Nombre {
  public :
    Nombre(); // constructeur par défaut
    // ...
  private :
    int _i;
};

Nombre::Nombre() {
  _i = 0;
}
```

On appelle **constructeur par défaut** un constructeur n'ayant pas de paramètre ou ayant des valeurs par défaut pour tous les paramètres.

```
class Nombre {
  public :
    Nombre(int i=0); // constructeur par défaut
    // ...
  private :
    int _i;
```

```
};

Nombre::Nombre(int i) {
  _i = i;
}
```

si le concepteur de la classe ne spécifie pas de constructeur, le compilateur générera un constructeur par défaut.

comme les autres fonctions, les constructeurs peuvent être surchargés.

```
class Nombre {
  public :
    Nombre();      // constructeur par défaut
    Nombre(int i); // constructeur à 1 paramètre
  private :
    int _i;
};
```

Le constructeur est appelé à l'instanciation de l'objet. Il n'est donc pas appelé quand on définit un pointeur sur un objet ...

Exemple :

```
Nombre n1;     // correct, appel du constructeur par défaut
Nombre n2(10); // correct, appel du constructeur à 1 paramètre
Nombre n3 = Nombre(10); // idem que n2

Nombre *ptr1, *ptr2;  // correct, pas d'appel aux constructeurs
ptr1 = new Nombre;  // appel au constructeur par défaut
ptr2 = new Nombre(12); // appel du constructeur à 1 paramètre

Nombre tab1[10]; // chaque objet du tableau est initialisé
                 // par un appel au constructeur par défaut
Nombre tab2[3] = { Nombre(10), Nombre(20), Nombre(30) };
                 // initialisation des 3 objets du tableau
                 // par les nombres 10, 20 et 30
```

Un objet anonyme et temporaire peut être créé par appel au constructeur de la classe :

```
  void analyse(Nombre n);

  void main() {
    analyse( Nombre(123) );
  }
```

évitant ainsi d'écrire :

```
  void main() {
    Nombre n(123);
    analyse( n );
  }
```

De la même façon que pour les constructeurs, le **destructeur** est une fonction membre spécifique de la classe qui est appelée implicitement à la destruction de l'objet.

Ce destructeur est une fonction :

- qui porte comme nom, le nom de la classe précédé du caractère (tilda)
- qui ne retourne pas de valeur (pas même un void)
- qui n'accepte aucun paramètre (le destructeur ne peut donc pas être surchargé)

Exemple :

```
class Exemple {
  public :
    // ...
    ~Exemple();
  private :
    // ...
};

Exemple::~Exemple() {
  // ...
}
```

Comme pour le constructeur, le compilateur générera un destructeur par défaut si le concepteur de la classe n'en spécifie pas un.

- Si des constructeurs sont définis avec des paramètres, le compilateur ne générera pas le constructeur par défaut.
- Les constructeurs et destructeurs sont les seules méthodes non constantes qui peuvent être appelées pour des objets constants.

CONSTRUCTEUR PAR COPIE

Présentation du problème :

Reprenons la classe *IntStack* avec un constructeur et un destructeur et écrivons une fonction (*AfficheSommet*) qui affiche la valeur de l'entier au sommet de la pile qui est passée en paramètre.

Exemple d'appel de cette fonction :

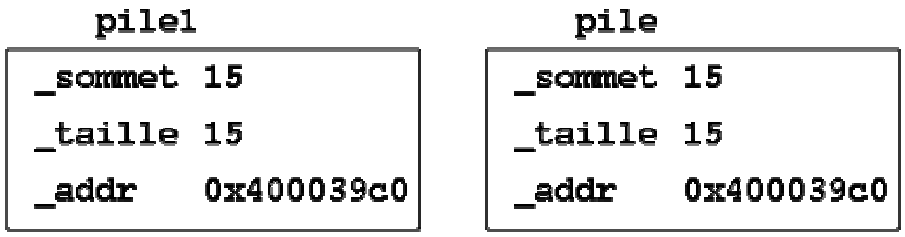
```
void main() {
  IntStack pile1(15);  // création d'une pile de 15 entiers

  // ...

  AfficheSommet( pile1 );
  // ...
}
```

Une version fausse (pour l'instant) de cette fonction peut ressembler au code qui suit :

```
void AfficheSommet( IntStack pile ) {
  cout << "Sommet de la pile : " << pile.pop() << endl;
```

}
Ici, la pile est passée par valeur, donc il y a création de l'objet temporaire nommé *pile* créé en copiant les valeurs du paramètre réel *pile1*.

L'affichage de la valeur au sommet de la pile marche bien, mais la fin de cette fonction fait appel au destructeur de l'objet local *pile* qui libère la mèmoire allouée par l'objet *pile1* parce que les données membres de l'objet *pile* contiennent les mêmes valeurs que celles de l'objet *pile1*.



Pour éviter cela :

il faut définir la fonction *AfficheSommet* comme :

```
void AfficheSommet( IntStack & pile ) {
  cout << "Sommet de la pile : " << pile.pop() << endl;
}
```

Mais une opération comme *pile.pop()* dépile un entier de la pile *pile1* !!!

il faut faire une copie intelligente : création du **constructeur de copie**.

Le constructeur de copie est invoqué à la construction d'un objet à partir d'un objet existant de la même classe.

```
Nombre n1(10); // appel du constructeur à 1 paramètre
Nombre n2(n1); // appel du constructeur de copie
Nombre n3=n1;  // appel du constructeur de copie
```

Le constructeur de copie est appelé aussi pour le passage d'arguments par valeur et le retour de valeur

```
Nombre traitement(Nombre n) {
  static Nombre nbre;
  // ...
  return nbre;      // appel du constructeur de copie
}

void main() {
  Nombre n1, n2;
  n2 = traitement( n1 ); // appel du constructeur de copie
```

```
}
```

Le compilateur C++ génère par défaut un constructeur de copie "bête"

LE POINTEUR *this*

Toute méthode d'une classe X a un paramètre caché : le pointeur *this*.

Celui contient l'adresse de l'objet qui l'a appelé, permettant ainsi à la méthode d'accéder aux membres de l'objet.

Il est implicitement déclaré comme (pour une variable) :

X * const this;

et comme (pour un objet constant) :

const X * const this;

et initialisé avec l'adresse de l'objet sur lequel la méthode est appelée.

Il peut être explicitement utilisé :

```
classe X {
  public:
    int f1() { return this->i; }
    // idem que :  int f1() { return i; }
  private:
    int i;
    // ...
};
```

Une fonction membre qui retourne le pointeur *this* peut être chaînée, étant donné que les opérateurs de sélection de membres . (point) et -> (tiret supérieur) sont associatifs de gauche à droite :

exemple :

```
classe X {
  public:
    X f1() { cout << "X "; return *this; }
    // ...
  private:
    // ...
};

void main() {
  X x;
  x.f1().f1().f1(); // affiche : X X X
}
```

La fonction membre *f1* a tout intérêt de retourner une référence :

```
 X &f1() { cout << "X "; return *this; }
```

- La valeur de this ne peut pas être changée.

- this ne peut pas être explicitement déclaré.

CLASSES ET FONCTIONS AMIES

"Un ami est quelqu'un qui peut toucher vos parties privées."

Dans la définition d'une classe il est possible de désigner des fonctions (ou des classes) à qui on laisse un libre accès à ses membres privés ou protégés.

C'est une infraction aux règles d'encapsulation pour des raisons d'efficacité.

Exemple de fonction amie :

```
class Nombre {
  // ici je désigne les fonctions qui pourront accéder
  // aux membres privés de ma classe
  friend int manipule_nbre(); // fonction amie
  public :
    int getnbre();
    // ...
  private :
    int _nbre;
};

int  manipule_nbre(Nombre n) {
  return n._nbre + 1;  // je peux le faire en tout légalité
                       // parce que je suis une fonction amie
                       // de la classe Nombre.
}
```

Exemple de classe amie :

```
class Window;  // déclaration de la classe Window
class Screen {
  friend class Window;
  public:
    //...
  private :
    //...
};
```

Les fonctions membres de la classe *Window* peuvent accéder aux membres non-publics de la classe *Screen*.

IV- SURCHARGE D'OPERATEURS

INTRODUCTION A LA SURCHARGE D'OPERATEURS

Le concepteur d'une classe doit fournir à l'utilisateur de celle ci toute une série d'opérateurs agissant sur les objets de la classe. Ceci permet une syntaxe intuitive de la classe.

Par exemple, il est plus intuitif et plus clair d'additionner deux matrices en surchargeant l'opérateur d'addition et en écrivant :

result = m0 + m1; que d'écrire *matrice_add(result, m0, m1);*

Règles d'utilisation :

Il faut veiller à respecter l'esprit de l'opérateur. Il faut faire avec les types utilisateurs des opérations identiques à celles que font les opérateurs avec les types prédéfinis.

La plupart des opérateurs sont surchargeables.

les opérateurs suivants ne sont pas surchargeables : ***:: . .* ?: sizeof***

il n'est pas possible de :

- changer sa priorité
- changer son associativité
- changer sa pluralité (unaire, binaire, ternaire)
- créer de nouveaux opérateurs

Quand l'opérateur + (par exemple) est appelé, le compilateur génère un appel à la fonction ***operator+***.

Ainsi, l'instruction ***a = b + c;*** est équivalente aux instructions :

```
a = operator+(b, c);   // fonction globale
a = b.operator+(c);    // fonction membre
```

Les opérateurs *= () [] -> new delete* ne peuvent être surchargés que comme des fonctions membres.

SURCHARGE PAR UNE FONCTION MEMBRE

Par exemple, la surcharge des opérateurs ***+*** et ***=*** par des fonctions membres de la classe *Matrice* s'écrit :

```
const int dim1= 2, dim2 = 3;  // dimensions de la Matrice
class Matrice {  // matrice dim1 x dim2 d'entiers
  public:

    // ...
    Matrice operator=(const Matrice &n2);
```

```
    Matrice operator+(const Matrice &n2);
    // ...
  private:
    int _matrice[dim1][dim2];
    // ...
};

// ...

void main() {
  Matrice a, b, c;
  b + c ;     // appel à : b.operator+( c );
  a = b + c;  // appel à : a.operator=(  b.operator+( c ) );
}
```

Une fonction membre (non statique) peut toujours utiliser le pointeur caché *this*.

Dans le code ci dessus, *this* fait référence à l'objet *b* pour l'opérateur *+* et à l'objet *a* pour l'opérateur *=*.

Définition de la fonction membre *operator+* :

```
Matrice Matrice::operator+(const Matrice &c) {
  Matrice m;

  for(int i = 0; i < dim1; i++)
    for(int j = 0; j < dim2; j++)
      m._matrice[i][j] = this->_matrice[i][j] + c._matrice[i][j];
  return m;
}
```

Quand on a le choix, l'utilisation d'une fonction membre pour surcharger un opérateur est préférable. Une fonction membre renforce l'encapsulation. Les opérateurs surchargés par des fonctions membres se transmettent aussi par héritage (sauf l'affectation).

La fonction membre *operator+* peut elle même être surchargée, pour dans l'exemple qui suit, additionner à une matrice un vecteur :

```
class Matrice {  // matrice dim1 x dim2 d'entiers
  public:
    // ...
    Matrice operator=(const Matrice &n2);
    Matrice operator+(const Matrice &n2);
    Matrice operator+(const Vecteur &n2);
    // ...
};

// ...

Matrice b, c;
Vecteur v;

b + c; // appel à b.operator+(c);
```

```
b + v; // appel à b.operator+(v); addition entre une matrice
       // et un vecteur
v + b; // appel à v.operator+(b); --> ERREUR si la classe
       // Vecteur n'a pas défini l'addition entre un vecteur et
       // une matrice
```

SURCHARGE PAR UNE FONCTION GLOBALE

Cette façon de procéder est plus adaptée à la surcharge des opérateurs binaires.

En effet, elle permet d'appliquer des conversions implicites au premier membre de l'expression.

Exemple :

```
Class Nombre{
  friend Nombre operator+(const Nombre &, const Nombre &);

  public:
    Nombre(int n = 0) { _nbre = n; }
    //....
  private:
    int _nbre;
};

Nombre operator+(const Nombre &nbr1, const Nombre &nbr2) {
  Nombre n;

  n._nbre = nbr1._nbre + nbr2._nbre;
  return n;
}

void main() {
  Nombre n1(10);

  n1 + 20;  // OK appel à :  operator+( n1, Nombre(20) );
  30 + n1;  // OK appel à :  operator+( Nombre(30) , n1 );
}
```

OPERATEUR D'AFFECTATION

C'est le même problème que pour le constructeur de copie.

Le compilateur C++ construit par défaut un opérateur d'affectation "bête".

L'opérateur d'affectation est obligatoirement une fonction membre et il doit fonctionner correctement dans les deux cas suivants :

```
X x1, x2, x3; // 3 instances de la classe X
x1 = x1;
x1 = x2 = x3;
```

Exemple : Opérateur d'affectation de la classe Matrice :

```
const int dim1= 2, dim2 = 3;  // dimensions de la Matrice

class Matrice {  // matrice dim1 x dim2 d'entiers
  public:
    // ...
    const Matrice &operator=(const Matrice &m);
    // ...
  private:
    int _matrice[dim1][dim2];
    // ...
};

const Matrice &Matrice::operator=(const Matrice &m) {
  if ( &m != this ) {   // traitement du cas :  x1 = x1
    for(int i = 0; i < dim1; i++)   // copie de la matrice
      for(int j = 0; j < dim2; j++)
        this->_matrice[i][j] = m._matrice[i][j];
  }
  return *this;  // traitement du cas :   x1 = x2 = x3
}
```

SURCHARGE DE ++ et --

- **notation préfixée :**
  - fonction membre : **X operator++();**
  - fonction globale : **X operator++(X &);**
- **notation postfixée :**
  - fonction membre : **X operator++(int);**
  - fonction globale : **X operator++(X &, int);**

Exemple :

```
class BigNombre {
  public:
    // ...
    BigNombre operator++();    // préfixée
    BigNombre operator++(int); // postfixée
    // ...
};

// ...

void main() {
  BigNombre n1;
  n1++; // notation postfixéé
  ++n1; // notation préfixée
}
```

V- L'HERITAGE

HERITAGE SIMPLE

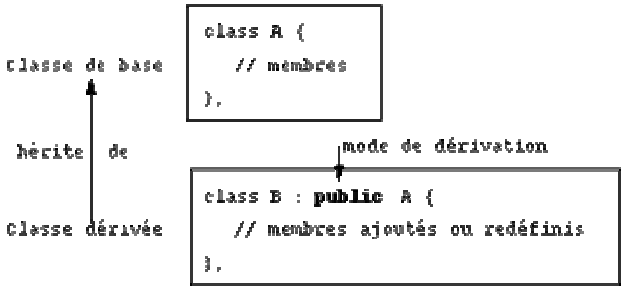
L'héritage, également appelé **dérivation**, permet de créer une nouvelle classe à partir d'une classe déjà existante, **la classe de base** (ou super classe).

"Il est plus facile de modifier que de réinventer"

La nouvelle classe (ou **classe dérivée** ou sous classe) hérite de tous les membres, qui ne sont pas privés, de la classe de base et ainsi réutiliser le code déjà écrit pour la classe de base.

On peut aussi lui ajouter de nouveaux membres ou redéfinir des méthodes.

Syntaxe :



La classe *B* hérite de façon publique de la classe *A*.

Tous les membres publics ou protégés de la classe *A* font partis de l'interface de la classe *B*.

MODE DE DERIVATION

Lors de la définition de la classe dérivée il est possible de spécifier le **mode de dérivation** par l'emploi d'un des mots-clé suivants :

public, *protected* ou *private*.

Ce mode de dérivation détermine quels membres de la classe de base sont accessibles dans la classe dérivée.

Au cas où aucun mode de dérivation n'est spécifié, le compilateur C++ prend par défaut le mot-clé *private* pour une classe et *public* pour une structure.

Les membres privés de la classe de base ne sont jamais accessibles par les membres des classes dérivées.

Héritage public :

Il donne aux membres publics et protégés de la classe de base le même statut dans la classe dérivée.

C'est la forme la plus courante d'héritage, car il permet de modéliser les relations "Y est une sorte de X" ou "Y est une spécialisation de la classe de base X".

Exemple :

```
class Vehicule {
  public:
    void pub1();
  protected:
    void prot1();
  private:
    void priv1();
};

class Voiture : public Vehicule {
  public:
    int pub2() {
      pub1();       // OK
      prot1();      // OK
      priv1();      // ERREUR
    }
};

Voiture safrane;
safrane.pub1();  // OK
safrane.pub2();  // OK
```

Héritage privé :

Il donne aux membres publics et protégés de la classe de base le statut de membres privés dans la classe dérivée.

Il permet de modéliser les relations "Y est composé de un ou plusieurs X" .

Plutôt que d'hériter de façon privée de la classe de base X, on peut faire de la classe de base une donnée membre (composition).

Exemple :

```
class String {
  public:
    int length();
  // ...
};

class Telephone_number : private String {
   void f1() {
```

```
    // ...
    l = length();  // OK
};

Telephone_number tn;
cout << tn.length();      // ERREUR
```

Héritage protégé :

Il donne aux membres publics et protégés de la classe de base le statut de membres protégés dans la classe dérivée.

L'héritage fait partie de l'interface mais n'est pas accessible aux utilisateurs.

Exemple :

```
class String {
  protected:
    int n;
};

class Telephone_number : protected String {
  protected:
    void f2() { n++; }  // OK
};

class Local_number : public Telephone_number {
  protected:
    void f3() { n++; } // OK
};
```

Tableau résumé de l'accès aux membres

		Statut dans la classe de base	**Statut dans la classe dérivée**
mode de dérivation	**public**	**public**	**public**
		protected	**protected**
		private	**inaccessible**
	protected	**public**	**protected**
		protected	**protected**
		private	**inaccessible**
	private	**public**	**private**
		protected	**private**
		private	**inaccessible**

REDEFINITION DE METHODES DANS LA CLASSE DERIVEE

On peut redéfinir une fonction dans une classe dérivée si on lui donne le même nom que dans la classe de base.

Il y aura ainsi, comme dans l'exemple ci après, deux fonctions *f2()*, mais il sera possible de les différencier avec l'opérateur ***::*** de résolution de portée.

Exemple :

```
class X {
  public:
    void f1();
    void f2();
  protected:
    int xxx;
};

class Y : public X {
  public:
    void f2();
    void f3();
};

void Y::f3() {
  X::f2();      // f2 de la classe X
  X::xxx = 12; // accès au membre xxx de la classe X
  f1();         // appel de f1 de la classe X
  f2();         // appel de f2 de la classe Y
}
```

AJUSTEMENT D'ACCES

Lors d'un héritage protégé ou privé, nous pouvons spécifier que certains membres de la classe ancêtre conservent leur mode d'accès dans la classe dérivée.

Ce mécanisme, appelé déclaration d'accès, ne permet en aucun cas d'augmenter ou de diminuer la visibilité d'un membre de la classe de base.

Exemple :

```
class X {
  public:
    void f1();
    void f2();
  protected:
    void f3();
    void f4();
};
```

```
class Y : private X {
  public:
    X::f1; // f1() reste public dans Y
    X::f3; // ERREUR: un membre protégé ne peut pas devenir public
  protected:
    X::f4; // f3() reste protégé dans Y
    X::f2; // ERREUR: un membre public ne peut pas devenir protégé
};
```

HERITAGE DES CONSTRUCTEURS / DESTRUCTEURS

Les constructeurs, constructeur de copie, destructeurs et opérateurs d'affectation ne sont jamais hérités.

Les constructeurs par défaut des classes de bases sont automatiquement appelés avant le constructeur de la classe dérivée.

Pour ne pas appeler les constructeurs par défaut, mais des constructeurs avec des paramètres, vous devez employer une **liste d'initialisation**.

L'appel des destructeurs se fera dans l'ordre inverse des constructeurs.

Exemple 1 :

```
class Vehicule {
  public:
    Vehicule() { cout<< "Vehicule" << endl; }
    ~Vehicule() { cout<< "~Vehicule" << endl; }
};

class Voiture : public Vehicule {
  public:
    Voiture() { cout<< "Voiture" << endl; }
    ~Voiture() { cout<< "~Voiture" << endl; }
};

void main() {
  Voiture *R21 = new Voiture;
  // ...
  delete R21;
}
/********** se programme affiche :
Vehicule
Voiture
~Voiture
~Vehicule
**********************************************/
```

Exemple d'appel des constructeurs avec paramètres :

```
class Vehicule {
  public:
```

```
    Vehicule(char *nom, int places);
    //...
};

class Voiture : public Vehicule {
  private:
    int _cv;    // puissance fiscale
  public:
    Voiture(char *n, int p, int cv);
    // ...
};

Voiture::Voiture(char *n, int p, int cv): Vehicule(n, p), _cv(cv)
{ /* ... */ }
```

CONVERSION DE TYPE DANS UNE HIERARCHIE DE CLASSES

Il est possible de convertir implicitement une instance d'une classe dérivée en une instance de la classe de base si l'héritage est public.

L'inverse est interdit car le compilateur ne saurait pas comment initialiser les membres de la classe dérivée.

Exemple :

```
class Vehicule {
  public:
    void f1();
  // ...
};

class Voiture : public Vehicule {
  public:
    int f1();
  // ...
};

void traitement1(Vehicule v) {
  // ...
  v.f1();   // OK
  // ...
}

void main() {
  Voiture R25;
  traitement1( R25 );
}
```

De la même façon on peut utiliser des pointeurs :

Un pointeur (ou une référence) sur un objet d'une classe dérivée peut être implicitement converti en un pointeur (ou une référence) sur un objet de la classe de base.

Cette conversion n'est possible que si l'héritage est public, car la classe de base doit posséder des membres public accessibles (ce n'est pas le cas d'un héritage protected ou private).

C'est le type du pointeur qui détermine laquelle des méthodes f1() est appelée.

```
void traitement1(Vehicule *v) {
  // ...
  v->f1();   // OK
  // ...
}

void main() {
  Voiture R25;
  traitement1( &R25 );
}
```

HERITAGE MULTIPLE

En langage C++, il est possible d'utiliser l'héritage multiple.

Il permet de créer des classes dérivées à partir de plusieurs classes de base.

Pour chaque classe de base, on peut définir le mode d'héritage.

```
class A {
  public:
    void fa() { /* ... */ }
  protected:
    int _x;
};

class B {
  public:
    void fb() { /* ... */ }
  protected:
    int _x;
};

class C: public B, public A {
  public:
    void fc();
};

void C::fc() {
  int i;
  fa();
  i = A::_x + B::_x; // résolution de portée pour lever l'ambiguïté
}
```

Ordre d'appel des constructeurs

Dans l'héritage multiple, les constructeurs sont appelés dans l'ordre de déclaration de l'héritage.

Dans l'exemple suivant, le constructeur par défaut de la classe C appelle le constructeur par défaut de la classe B, puis celui de la classe A et en dernier lieu le constructeur de la classe dérivée, même si une liste d'initialisation existe.

```
class A {
  public:
    A(int n=0) { /* ... */ }
  // ...
};

class B {
  public:
    B(int n=0) { /* ... */ }
  // ...
};

class C: public B, public A {
//       ^^^^^^^^^^^^^^^^^^^
//       ordre d'appel des constructeurs des classes de base
//
  public:
    C(int i, int j) : A(i) , B(j) { /* ... */ }
  // ...
};

void main() {
  C objet_c;
  // appel des constructeurs B(), A() et C()
  // ...
}
```

Les destructeurs sont appelés dans l'ordre inverse de celui des constructeurs.

LE POLYMORPHISME

L'**héritage** nous permet de réutiliser le code écrit pour la classe de base dans les autres classes de la hiérarchie des classes de votre application.

Le **polymorphisme** rendra possible l'utilisation d'une même instruction pour appeler dynamiquement des méthodes différentes dans la hiérarchie des classes.

En C++, le polymorphisme est mis en oeuvre par l'utilisation des **fonctions virtuelles**.

Fonctions virtuelles

```
class ObjGraph {
 public:
   void print() const { cout <<"ObjGraph::print()"; }
};

class Bouton: public ObjGraph {
  public:
    void print() const { cout << "Bouton::print()"; }
};

class Fenetre: public ObjGrap {
  public:
    void print() const { cout << "Fenetre::print()"; }
};

void traitement(const ObjGraph &og) {
  // ...
  og.print();
  // ...
}

void main() {              // Qu'affiche ce programme ???
  Bouton OK;
  Fenetre windows97;
  traitement(OK);          // affichage de .....
  traitement(Window97);    // affichage de .....
}
```

Comme nous l'avons déjà vu, l'instruction *og.print()* de *traitement()* appelera la méthode *print()* de la classe *ObjGraph*.

La réponse est donc :

```
  traitement(OK);          // affichage de ObjGraph::print()
  traitement(Window97);    // affichage de ObjGraph::print()
}
```

Si dans la fonction *traitement()* nous voulons appeler la méthode *print()* selon la classe à laquelle appartient l'instance, nous devons définir, dans la classe de base, la méthode *print()* comme étant **virtuelle** :

```
class ObjGraph {
  public:
    // ...
    virtual void print() const {
       cout<< "ObjetGraphique::print()" << endl;}
};
```

Pour plus de clarté, le mot-clé *virtual* peut être répété devant les méthodes *print()* des classes *Bouton* et *Fenetre* :

```
class Bouton: public ObjGraph {
  public:
```

```
    virtual void print() const {
      cout << "Bouton::print()";
    }
};

class Fenetre: public ObjGrap {
  public:
    virtual void print() const {
      cout << "Fenetre::print()";
    }
};
```

On appelle ce comportement, le **polymorphisme**.

Lorsque le compilateur rencontre une méthode virtuelle, il sait qu'il faut attendre l'exécution pour déterminer la bonne méthode à appeler.

Destructeur virtuel

Il ne faut pas oublier de définir le destructeur comme "virtual" lorsque l'on utilise une méthode virtuelle :

```
class ObjGraph {
  public:
    //...
    virtual ~ObjGraph() { cout << "fin de ObjGraph\n"; }
};
class Fenetre : public ObjGraph {
  public:
    // ...
    ~Fenetre() { cout << "fin de Fenêtre  "; }
};

void main() {
  Fenetre *Windows97 = new Fenetre;
  ObjGraph *og = Windows97;
  // ...
  delete og;  // affichage de :    fin de Fenêtre  fin de ObjGraph
              // si le destructeur n'avait pas été virtuel,
              // l'affichage aurait été :          fin de ObjGraph
}
```

- Un constructeur, par contre, ne peut pas être déclaré comme virtuel.
- Une méthode statique ne peut, non plus, être déclaré comme virtuelle.
- Lors de l'héritage, le statut de l'accessibilité de la méthode virtuelle (public, protégé ou privé) est conservé dans toutes les classes dérivée, même si elle est redéfinie avec un statut différent. Le statut de la classe de base prime.

CLASSES ABSTRAITES

Il arrive souvent que la méthode virtuelle définie dans la classe de base serve de cadre générique pour les méthodes virtuelles des classes dérivées. Ceci permet de garantir une bonne homogénéité de votre architecture de classes.

Une classe est dite abstraite si elle contient au moins une méthode virtuelle pure.

On ne peut pas créer d'instance d'une classe abstraite et une classe abstraite ne peut pas être utilisée comme argument ou type de retour d'une fonction.

Par contre, les pointeurs et les références sur une classe abstraite sont parfaitement légitimes et justifiés.

Méthode virtuelle pure

une telle méthode se déclare en ajoutant un **= 0** à la fin de sa déclaration.

```
class ObjGraph {
  public:
    virtual void print() const = 0;
};

void main() {
  ObjGraph og;  // ERREUR
  // ...
}
```

- On ne peut utiliser une classe abstraite qu'à partir d'un pointeur ou d'une référence.
- Contrairement à une méthode virtuelle "normale", une méthode virtuelle pure n'est pas obligé de fournir une définition pour ObjGraph::print().
- Une classe dérivée qui ne redéfinit pas une méthode virtuelle pure est elle aussi abstraite.