POLYCOPIE DE C++

1. Introduction

Ce document est actuellement la compilation de notes de cours C++ donné en MIAIF, DESS, MST2.

Plan du cours

2. présentation générale
3. minimum sur les entrées sorties
4. rappel sur les références et les pointeurs
5. spécificités de C++ par rapport à C
6. classes et objets
7. propriétés des fonctions membres
8. construction destruction initialisation d'objets
9. généralisation, héritage
10. fonctions virtuelles
11. templates
12. La STL, par l'exemple de la classe `vector`
13. surdéfinition d'opérateurs =, *, <<, >>, ()
14. fonctions amies
15. entrées sorties «élaborées»

Références

Delannoy, « Le langage C++ », Eyrolles
Très bon livre pour commencer le C++, livre très pédagogique. A servi de base pour ces notes de cours.

Stroustrup, « Le langage C++ », Addison Wesley, Pearson Education
Le livre de référence par l'auteur du C++. Complet mais beaucoup trop riche pour débuter.

Scott Meyers, « Le C++ efficace », Addison Wesley
Des règles pour programmer proprement et efficacement en C++.

Schaum, C++ cours et exercices corrigés
C++ détaillé avec exemples, exercices corrigés. A servi de base pour une partie des notes de ce cours. Par contre, les classes, l'héritage et les fonctions virtuelles ne sont pas assez développés.

Johannes Weidl, « The Standard Template Library Tutorial ».
Un tutorial sur la STL, accessible sur le Web. Pour programmeur confirmé.

2. Présentation générale

Le C++ a été conçu par Bjarne Stroustrup en 1982 aux ATT Bell Laboratories. L'idée était d'ajouter au C des possibilités sur l'orienté objet et de palier aux inconvénients du C.

Le but de ces notes de cours est de donner un premier aperçu sur C++ en ne présentant que ce qu'il est nécessaire de connaître pour bien programmer objet avec C++. Ce document suppose connu le langage C et l'orienté objet sans que cela soit bloquant pour le lire.

3. Le minimum vital sur les entrées sorties

Bien sûr, C++ dispose des routines offertes par la bibliothèque standard du C ANSI <stdio.h>. Mais il comporte aussi des possibilités d'entrées sorties propres. Le but de ce paragraphe n'est pas de les présenter complètement (cf paragraphe plus loin) mais *de présenter le minimum sur les entrées sorties utilisées pour commencer* à programmer en C++.

Ecrire sur la sortie standard

Le premier programme C++ à écrire est le suivant.

```
#include <iostream.h>
main () { cout << " bonjour le monde ! " << endl ; }
```

<iostream.h> est la bibliothèque du C++ permettant de faire des entrées sorties.

Pour écrire, sur la sortie standard on utilise le " stream " de la sortie standard **cout**.

L'opérateur << prend un stream en premier opérande et une chaîne de caractères en deuxième opérande et retourne le premier opérande après avoir écrit la chaîne de caractères sur le stream.

On peut chaîner l'utilisation de << pour écrire plusieurs chaînes de caractères à la suite les unes des autres.

endl signifie le saut de ligne.

Ainsi, la sortie de ce premier programme C++ est :

```
bonjour le monde !
```

On peut afficher la valeur d'une variable.

```
#include <iostream.h>
main () {
    int n = 25 ;
    cout << " la valeur de n est   " << n << endl ;
}
```

Plus généralement on peut écrire :

```
cout << … << … << … ;
```

Le type des variables que l'on peut envoyer sur << est presque quelconque.

Lire sur l'entrée standard

On peut également lire des informations sur l 'entrée standard du programme.

Le programme suivant demande d'entrer une valeur sur l'entrée standard, puis il la lit avec l'opérateur >> et le stream **cin** de l'entrée standard ; enfin il affiche la valeur entrée par l'utilisateur.

```
#include <iostream.h>
main () {
    int n;
    cout << " entrer une valeur :   " ;
    cin >> n ;
    cout << " la valeur entrée est   " << n << endl ;
}
```

De même que pour <<, l'opérateur >> accepte presque tous les types et il est associatif.

On peut donc écrire un programme qui demande des valeurs de types différents de la manière suivante :

```
#include <iostream.h>
main () {
    int n; float x ; char t[64] ;
    cout << " entrer un entier, un flottant et une chaîne de caractères:
  " ;
    cin >> n >> x >> t ;
    cout << " l'entier vaut   " << n ;
    cout << " le flottant vaut   " << x ;
    cout << " la chaîne vaut   " << t << endl ;
}
```

séparateurs

Les valeurs entrées doivent être séparées par un caractère espace ou n'importe quel autre caractère spécial (tabulation, saut de ligne, fin de fichier, etc.).

4. Rappel sur les pointeurs et références

Ce paragraphe rappelle le strict minimum sur ce que sont un pointeur ou une référence. Pour des détails, se reporter par exemple à [Schaum].

Quand on déclare une variable avec un nom et un type, un emplacement mémoire du type de la variable est créé à une certaine adresse avec son nom pour y accéder. L'emplacement mémoire recevra la valeur de la variable lors d'une affectation.

```
int x ;           // une déclaration
x = 3 ;           // une affectation
```

Le C permet de manipuler dans le programme, les adresses des emplacements mémoire des variables. `&x` désigne l'adresse de la variable `x`. On peut déclarer des *pointeurs* d'un type qui sont des variables contenant des adresses de variables de ce type avec le symbole `*`.

```
int * p ;         // un pointeur sur un entier
p = &x ;          // p vaut l'adresse de x
```

L'opérateur `*` s'appelle l'opérateur de *déréférencement* et `*p` désigne la valeur contenue dans l'emplacement mémoire dont l'adresse et `p`.

```
int y = *p ;      // y vaut 3
```

On peut aussi déclarer une *référence* à une variable existante. Une référence se déclare avec l'opérateur `&`. Une référence est un synonyme - un alias (un nom supplémentaire) - pour désigner l'emplacement mémoire d'une variable.

```
int & z = x ;     // z est une référence a x, z vaut 3
```

Si on change la valeur de `x`, la valeur de `z` est aussi changée et inversement. Idem avec `*p`.

```
x = 4;      // x et z valent 4, *p aussi
z = 5;      // x et z valent 5, *p aussi
*p = 6;     // *p, x et z valent 6
```

Noter que les opérateurs `&` et `*` sont inverses l'un de l'autre. On a toujours :

`*(&x) = x` et `&(*p) = p`

Attention, on ne peut pas déréférencer un pointeur qui ne contient pas une adresse valide.

```
int * q ;
*q = 7 ;    // plantage
```

Il faut initialiser le pointeur avant :

```
int * q = &x;
*q = 7 ;    // ok
```

NB : Il est très important de très bien connaître le fonctionnement des pointeurs et des références car l'intérêt de la programmation objet en C++ repose sur leur utilisation intensive.

5. Spécificités de C++ surlangage de C

C++ dispose d'un certain nombre de spécificités par rapport a C qui ne sont pas axées sur l'orienté objet :

- le commentaire
- l'emplacement libre des déclarations
- les arguments par défaut
- la surdéfinition de fonction
- les opérateurs new et delete
- les fonctions en ligne

Commentaire

Pour écrire des commentaires dans un programme, le programmeur dispose toujours des commentaires C avec /* et */ et aussi de commentaires en fin de ligne avec //

```
cout << « bonjour » ;       // ceci est une formule de politesse
```

On peut mêler les deux techniques :

```
/* ceci est un commentaire en C // donc ce double slash ne sert a rien */
```

ou encore

```
// ceci est un commentaire en C++ /* donc ce slash etoile ne sert à rien */
```

mais pas :

```
 // ceci est un commentaire en C++ /* donc ce slash etoile ne sert à rien
et cela donne une erreur de compilation si le etoile slash n'est pas sur la
même ligne */
```

Emplacement des déclarations de variables

L'emplacement des déclarations est *libre* en C++ ; le programmeur n'est pas obligé de les mettre au début d'une fonction. Par contre, on ne peut utiliser la variable déclarée que dans les instructions du bloc où est effectuée la déclaration et postérieures à la déclaration. L'avantage de cette spécificité du C++ est de pouvoir déclarer une variable juste au moment où l'on en a besoin et cela clarifie le programme. Mais cet avantage est bien maigre.

```
void f() {
    ...
    i = 4 ;       // incorrect
    ...
    int i ;       // i est déclaré ici
    ...
    i = 4 ;       // correct
    ...
    {
      ...
      float f ;   // f est declare ici
      ...
      f = 4.0 ;   // correct
      ...
    }             // fin de la portee de f
    ...
    f = 4.0 ;     // incorrect
    ...
    i = 5 ;       // correct
    ...
}                 // fin de la portee de i
```

Passage de paramètres par référence

En C, les arguments sont passés par valeur. Ce qui signifie que les paramètres d'une fonction C sont toujours en entrée de la fonction et pas en sortie de la fonction ; autrement dit les paramètres d'une fonction ne peuvent pas être modifiés par la fonction.

Ci-dessous la fonction `echange` est censée échanger les valeurs des deux paramètres `n` et `p`.

```
void echange(int a, int b) {
      int c = a ;
      a = b ;
      b = c ;
}
main() {
      int n = 10 ; int p = 20 ;
      cout << « avant appel : «  << n << «  «  << p << endl ;
      echange(n, p) ;
      cout << « après appel : «  << n << «  «  << p << endl ;
}
```

Malheureusement, pour la raison invoquée plus haut, la sortie de ce programme est :

```
      avant appel : 10 20
      apres appel : 10 20
```

Les programmeurs C ont l'habitude de palier à cet inconvénient du C en passant l'adresse des paramètres en sortie d'une fonction au lieu de passer la valeur. Ainsi, notre exemple sera écrit :

```
void echange(int * a, int * b) {
      int c = *a ;
      *a = *b ;
      *b = c ;
}
main() {
      int n = 10 ; int p = 20 ;
      cout << « avant appel : «  << n << «  «  << p << endl ;
      echange(&n, &p) ;
      cout << « apres appel : «  << n << «  «  << p << endl ;
```

```
}
```

La sortie de ce programme correspond à ce que notre intuition attend :

```
avant appel : 10 20
apres appel : 20 10
```

Cette manipulation de pointeurs est lourde à gérer pour le programmeur. C++ palie à cet inconvénient en permettant le *passage de paramètres par référence* ; la programmation devient plus légère et le résultat correct. Il suffit de changer la déclaration de la fonction `echange` :

```
void echange(int & a, int & b) { ... }
```

L'appel de la fonction est le même que pour le passage par valeur :

```
echange(n, p) ;
```

On notera qu'il n'y a rien d'extraordinaire. Pascal possède cette propriété de passage de paramètres par référence et C++ ne fait que combler un manque important du C.

Arguments par défaut

En C il est indispensable que l'appel de la fonction contienne exactement le même nombre et type d'arguments que dans la déclaration de la fonction. C++ permet de s'affranchir de cette contrainte en permettant l'usage d'arguments par défaut.

```
void f(int, int = 12) ;
main () {
      int n = 10 ; int p = 20 ;
      f(n, p) ;
      f(n) ;
}
void f(int a, int b) {
      cout << « premier argument : «  << a ;
      cout << «  second argument : «  << b << endl ;
}
```

La sortie de ce programme est :

```
premier argument : 10 second argument : 20
premier argument : 10 second argument : 12
```

Lors d'une déclaration avec des arguments par défaut, ceux-ci doivent être les derniers de la liste des arguments.

Le programmeur doit fixer les valeurs par défaut dans la déclaration de la fonction ou dans la définition.

Surdéfinition de fonction

On parle de surdéfinition lorsqu'un symbole prend plusieurs significations différentes. Ci-dessous le symbole `sosie` possède deux significations.

```
void sosie(int) ;
void sosie(float) ;
main () {
      int n = 10 ; float x = 4.0 ;
      sosie(n) ;
      sosie(x) ;
}
void sosie(int a) {
      cout << « sosie avec INT : «  << a << endl ;
}
void sosie(float b) {
      cout << « sosie avec FLOAT : «  << b << endl ;
}
```

La sortie de ce programme est :

```
sosie avec INT : 10
sosie avec FLOAT : 4.0
```

Opérateurs new et delete

En C, la gestion dynamique de la mémoire fait appel aux fonctions `malloc` et `free`. En C++, on utilise les fonctions `new` et `delete`.

Avec la déclaration :
```
int * ad ;
```
en C++ on alloue dynamiquement comme cela :
```
ad = new int ;
```
alors qu'en C il fallait faire comme ceci :
```
ad = (int *) malloc (sizeof(int)) ;
```

Plus généralement, si on a un type donné `type`, on alloue une variable avec :
```
new type ;
```
ou un tableau de n variables avec :
```
new type [n] ;
```

On désalloue dynamiquement de la mémoire (allouée avec `new`) comme cela :
```
delete ad;
```

Plus généralement, on désalloue une variable `x` (allouée avec `new`) avec :
```
delete x;
```
ou un tableau de variables (allouée avec `new []`) avec :
```
delete [] x ;
```

En C++, bien que l'utilisation de `malloc` et `free` soit toujours permise, il est très conseillé de n'utiliser que `new` et `delete`.

Spécification « inline »

Rappel sur les macros en C

En C, on peut définir une macro avec le mot-clé `define` :

```
#define carre(A)      A*A
```

Cela permet à priori d'utiliser carre comme une fonction normale :

```
int a = 2 ;
int b = carre(a) ;
cout << « a = «  << a << « b = «  << b << endl ;
```

le résultat sera correct dans ce cas simple :

```
a = 2       b = 4
```

Quand il voit une macro, le C remplace partout dans le code les expression `carre(x)` par `x*x`.

L'avantage d'une macro par rapport à une fonction est la rapidité d'exécution. En effet, le temps a recopier les valeurs des paramètres disparaît. La contrepartie est un espace mémoire du programme plus grand.

Plus embêtant sont les effets de bord des macros. Si l'on programme :

```
b = carre(a++) ;
```

On attendrait le résultat suivant :

```
a = 3 b = 4
```

En réalité, ce n'est pas le cas car, à la compilation, le C remplace malheureusement `carre(a++)` par `a++*a++`. Et à l'exécution, le programme donne le résultat suivant :

```
a = 4 b = 6
```

Les fonctions inline

Le C++ palie à cet inconvénient en permettant de définir des fonctions, dites en ligne, avec le mot clé '**inline**'.

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

Une fonction définie avec le mot-clé `inline` aura le même avantage qu'une macro en C, à savoir un gain de temps d'exécution et le même inconvénient, à savoir une plus grande place mémoire occupée par le programme.

L'avantage des fonctions en ligne est la disparition des effets de bord.

6. Classes et objets

Ce paragraphe aborde les caractéristiques de C++ vis-à-vis de la programmation orientée objet. Il rappelle d'abord le vocabulaire objet, notamment ce que signifie *encapsulation*, puis il montre comment déclarer une classe C++, comment définir le corps des fonctions membres, la distinction entre membres privés et publiques, l'affectation d'objets, les constructeurs et les destructeurs, l'exploitation d'une classe avec des fichiers .h et .cpp et enfin les membres statiques d'une classe.

Vocabulaire objet

Ce paragraphe rappelle le vocabulaire employé en orienté objet. Et il donne la correspondance entre le vocabulaire objet et le vocabulaire spécifique de C++.

Objet

Un *objet* est une entité, qui possède un état et un comportement. Pour définir ce qu'est un objet, les livres donnent souvent une équation du style de :

objet = état + comportement

Classe

Vue de la programmation objet, une *classe* est un type structuré de données. Nous verrons qu'une classe C++ est le prolongement des structures C (mot-clé `struct`).

Vue de la modélisation objet, une classe correspond à un concept du domaine modélisé. Une classe regroupe des objets qui ont des propriétés et des comportements communs. En C++, pour une classe, on dit aussi une classe.

Instance

Pour désigner un objet de la classe, on dit aussi une *instance*. « instance » est un anglicisme qui possède une signification proche de celle de « exemple » en français. On dit souvent qu'une instance « instancie » une classe. Cela signifie que l'instance est un exemple de la classe. En C++, pour désigner une instance on dit plutôt un objet.

Attribut ou membre

L'état d'un objet est l 'ensemble des valeurs de ses attributs. Un *attribut* est une propriété de l'objet. En C++, on dit *membre*.

Méthode ou fonction membre

Vue de la modélisation objet, une *méthode* est une opération que l'on peut effectuer sur un objet. Vue de la programmation objet, une méthode est une fonction qui s'applique sur une instance de la classe. En C++, on dit *fonction membre*.

Message

A l'origine de la programmation orientée objet, un objectif était d'avoir des comportements d'objets concurrents. On voulait qu'ils « communiquent » en « envoyant » et « recevant » des « messages ». Actuellement, tous les langages objet implémentent les objets de manière fonctionnelle. C'est-à-dire *qu'au lieu « d'envoyer un message à un objet », on appelle une méthode*, c'est-à-dire que l'on appelle une fonction qui manipulent des données. Donc, il faut être conscient de la correspondance suivante :

envoyer un message à un objet	=	appeler une méthode associée à l'objet
recevoir un message	=	entrer dans le corps de la méthode appelée

Encapsulation des données

En POO pure, on suit le *principe d'encapsulation* :

Tout attribut d'une classe est caché dans l'implémentation de la classe et il est accédé par une méthode de lecture et une méthode d'écriture située dans l'interface de la classe.

Pour répondre à ce principe, C++ prévoit les mot-clés '**private**' et '**public**' pour dire si un attribut est visible d'un programme utilisateur de la classe. En ce sens, il permet de ne pas encapsuler tous les attributs d'une classe et permet de ne pas suivre le principe d'encapsulation a la lettre. Nous pensons que c'est une bonne chose car de nombreuses applications ne nécessitent pas d'encapsuler toutes les données, d'autres oui. D'autres langages objet, au contraire de C++, obligent le programmeur à suivre ce principe (Smalltalk par exemple).

Déclaration de classe

Pour déclarer une classe Bidule en C++, on utilise le mot-clé '**class**' :

```
class Bidule { ...} ;
```

Le mot-clé `class` de C++ est une extension du mot-clé '**struct**' de C et il suit une syntaxe proche.

Exemple :

```
class Point {
      int x ;                        // un membre
      int y ;                        // un autre membre
public :
      void initialise(int, int) ;    // une fonction membre
      void deplace(int, int);              // encore une
      void affiche() ;               // encore une fonction membre
};
```

`x` et `y` sont des membres de la classe `Point`.

`Initialise`, `deplace` et `affiche` sont des fonctions membres de la classe `Point`.

Noter que cet exemple suit le principe d'encapsulation car, par défaut, les premiers membres de la classe sont privés (`x` et `y`). Les fonctions membres dont la déclaration est postérieure au mot-clé `public` sont publiques.

Définition des fonctions membres de la classe

La déclaration d'une classe doit être suivie de la définition des ses fonctions membres. Chaque définition d'une fonction membre suit la syntaxe de la définition d'une fonction C avec un nom préfixé par le nom de la classe et quatre points '**::**'.

Le symbole '**::**' s 'appelle *l'opérateur de résolution de portée*.

Ci-dessous figure un exemple de définition des fonctions membres de la classe `Point`.

```
void Point::initialise(int a, int b) {
    x = a;
    y = b;
}
void Point::deplace(int dx, int dy) {
    x += dx;
    y += dy;
}
void Point::affiche() {
    cout << « x = «  << x << endl;
    cout << « y = «  << y << endl;
}
```

Noter que les membres (x et y) sont visibles des fonctions membres de la classe Point. Ils sont utilisés simplement avec leur nom de variable déclaré dans la classe.

Voici un exemple d'utilisation de la classe Point par un programme :

```
main() {
      Point a, b ;
      a.initialise(5, 2) ;
      a.affiche() ;
      a.deplace(8, 4) ;
      a.affiche() ;
      b.initialise(-1, 1) ;
      b.affiche() ;
}
```

Pour appeler la fonction membre initialise sur l'objet a, noter qu'il faut écrire le nom de l'objet (a), un point (.) et le nom de la fonction membre (initialise).

Membres privés ou publiques

Pour l'exemple précédent, une instruction du style de a.x = 3 ; dans le main() serait rejetée à la compilation car x est un membre privé de la classe Point.

NB : Si x était un membre publique de la classe Point on pourrait le faire. Il faut noter que la syntaxe est la même que pour les fonctions membres : on écrit le nom de l'objet (x), un point (.) et le nom du membre (x).

La déclaration d'une classe est toujours du type :

```
class Toto {
      ...         // des membres prives
public :
      ...         // des membres publiques
private :
      ...         // encore des membres prives
public :
      ...         // encore des membres publiques

      ...         // etc.
};
```

On peut mettre un mot-clé public ou private pour chaque membre si on le souhaite.

Si on omet le mot-clé public dans une classe, aucun de ses membres n'est accessible de l'extérieur de la classe...

Affectation d'objets

On peut affecter un objet dans un autre :

```
a = b ;
```

Dans ce cas, les membres de `b` sont recopiés dans ceux de `a`.

Constructeur et destructeur d'objets

En fait, une caractéristique de la POO est de maîtriser la création d'objet, leur initialisation et leur destruction. C++ répond à ce besoin avec la notion de constructeur d'objet pour la création et l'initialisation et avec la notion de destructeur.

Un constructeur d'objet est une fonction membre particulière d'une classe qui est appelée lors de la création d'un objet, quel qu'il soit. Son but est :

- d'allouer un emplacement mémoire pour l'objet,
- d'initialiser les membres de l'objet avec de bonnes valeurs de départ et
- de retourner l'adresse de l'emplacement mémoire choisi.

Dans notre exemple, le constructeur fait l'équivalent de la fonction `initialise` et fait l'allocation de l'emplacement mémoire.

Un destructeur d'objet :

- remet l'objet dans un état terminal et
- libère l'emplacement mémoire associé à l'objet.

En fait, notre exemple précédent était approximatif et en bon C++, il faut écrire :

```
class Point {
      ...
public :
      Point(int, int) ; // le constructeur de la classe
      ~Point() ;        // le destructeur de la classe
      ...
};

Point : : Point(int a, int b) {
      x = a ;
      y = b ;
}
Point : : ~Point() {}   // le destructeur ne sert pas dans cet exemple ☹

main() {
      Point a(5, 2) ;
```

```
        ...
        Point b(-1, 1) ;
        ...
}
```

A partir du moment où un constructeur existe, il n'est pas possible de créer un objet sans fournir les arguments requis par le constructeur.

Noter qu'un destructeur ne possède jamais d'argument.

L'exemple suivant permet de comprendre les moments où sont appelés les constructeurs et destructeurs d'objets.

```
class Test {
      int num ;
      Test(int) ;
      ~Test() ;
} ;
Test : :Test(int n) {
      num = n ;
      cout << «++ « << num << endl ;
}
Test : :~Test() {
      cout << «--«   << num << endl ;
}
void fct(int p) {
      Test x(2*p) ;
}
main() {
      Test a(1) ;
      fct(1) ;
      fct(2) ;
}
```

La sortie du programme est :

```
++ 1
++ 2
-- 2
++ 4
-- 4
-- 1
```

a est détruit à la sortie du programme `main` et les objets `x` sont détruits à la sortie de `fct`.

Régles d'utilisation des fichiers .cpp et .h

Pour pouvoir effectivement réutiliser les classes programmées et les compiler séparément, il est indispensable de les exploiter proprement et les ranger dans des fichiers qui portent des noms corrects.

La déclaration d'une classe `Classe` doit être mise dans un fichier `classe.h`
La définition d'une classe `Classe` doit être mise dans un fichier `classe.cpp`

Si l'on fait une analogie avec la programmation modulaire,
(rappel 1 module = 1 interface visible + 1 corps caché)
on peut dire que :

Le fichier `.cpp` est le corps de la classe.
Le fichier `.h` est l'interface de la classe.

Pour l'instant, retenir qu'un fichier `classe.h` possède la structure suivante :

```
// fichier classe.h
#ifndef CLASSE_H
#define CLASSE_H

#include ... // includes de classes eventuelles
...

class Classe {
      ...
} ;
#endif
```

Et qu'un fichier `classe.cpp` possède la structure suivante :

```
// fichier classe.cpp
#include « classe.h »   // include de sa propre classe
#include ...            // autres includes optionnels
...
Classe : :Classe(...) { // definition du constructeur
      ...
}
Classe : :~Classe() {   // definition du destructeur
      ...
}
...                     // autres definitions de fonctions membres
```

Ces structures de fichiers sont correctes mais incomplètes. Elles seront affinées lors des paragraphes futurs.

Elles permettent la réutilisation et la compilation séparée.

`#ifndef` `#define` et `#endif` dans le `.h` servent de garde-fou pour que le fichier ne soit effectivement inclus qu'une seule fois lors d'une compilation.

Noter l'include obligatoire de `classe.h` dans le fichier `classe.cpp` ; les autres sont optionnels.

Membres statiques

Avec le qualificatif '**static**' avant un membre d'une classe on peut spécifier qu'un membre est commun à tous les objets de la classe.

En modélisation objet, cela correspond aux attributs « de classe ». Un membre sans le mot-clé `static` correspond à un attribut « d'instance ».

```
Class Exemple {
      static int a ;
      float b ;
public :
      Exemple(float) ;
      ...
} ;
...
Exemple : :a = 0 ;// definition d'un membre statique necessaire
...
Exemple e(1), f(2) ;
...
```

Dans la classe `Exemple`, `a` est un membre statique, `b` un membre non statique.

Un membre statique doit être défini dans le `.cpp` tout comme une fonction membre sinon une erreur sera produite à l'édition de liens. La définition du membre statique permet son initialisation. Ici `a = 0 ;`

`e` et `f` sont des objets avec `e.b = 1` et `f.b = 2`

`e` et `f` ont le membre `a` en commun `e.a = f.a = 0`.

7. Propriétés des fonctions membres

De manière générale, toutes les améliorations concernant les fonctions restent valables aussi les fonctions membres :

- surdéfinition de fonctions membres,
- arguments par défaut,
- objets transmis en argument d'une fonction membre,
  par valeur, adresse ou référence.

Nous ne les rappellons pas. Il suffit de regarder le paragraphe 5 sur les spécifictés de C++.

Ce paragraphe comprend les améliorations propres aux fonctions membres :

- fonctions membres en ligne,
- fonctions membres statiques,
- appel des fonctions membres.

Fonctions membres en ligne

On peut mettre une fonction membre en ligne mais la syntaxe est différente de celle des fonctions tout court.

Au lieu de mettre le mot-clef " **inline**" devant la définition de la fonction, on écrit le corps de la fonction au même endroit que sa déclaration dans la classe.

Ci-dessous la fonction membre `deplace` de la classe `Point` est normale :

```
class Point { …
    void deplace(int, int);
};
void Point::deplace(int dx, int dy) { x += dx; y += dy; };
```

Ci-dessous elle est inline bien que le mot-clef ne soit pas présent.

```
class Point { …
    void deplace(int dx, int dy) { x += dx; y += dy; };
};
```

Fonctions membres statiques

L'orienté objet définit deux types de méthodes dans une classe : les méthodes associées à une instance de la classe (les méthodes dites " d'instance ") et les méthodes non associables à une instance de la classe mais associées à la classe toute entière (les méthodes dites " de classe ").

Le C++ permet ce genre de distinction avec le mot-clef "**static**".

Une fonction membre statique C++ correspond à une méthode de classe dans le vocabulaire objet. Une fonction membre non statique C++ correspond à une méthode d'instance dans le vocabulaire objet.

Ci-dessous, la fonction membre `affiche_tout` de la classe `Point` est statique et la fonction membre `affiche` est normale.

```
class Point { …
    void affiche();
    static void affiche_tout();
};
```

Autoréférence: le mot-clé *this*

Quand le programme est dans une fonction membre d'instance, le programmeur peut manipuler l'adresse de l'instance courante avec le mot-clef '**this**'. En effet, `this` désigne l'adresse de l'instance courante.

```
class Point { …
    void affiche() { cout << " mon adresse est " << this << endl; }
};
```

`this` est utile, par exemple, dans un constructeur (ou un destructeur) pour stocker l'adresse de l'objet construit (ou effacer l'adresse de l'objet détruit) d'un tableau ou liste d'instances ou d'un attribut d'un autre objet.

`this` est un paramètre implicite des fonctions membres d'instance. Il faut savoir que le préprocesseur d'un compilateur C++ transforme toutes les fonctions membres d'instance C++ par des fonctions C dont le premier paramètre est l'adresse de l'instance.

`this` n'a évidemment pas de sens dans une fonction membre de classe.

Appel des fonctions membres

L'appel d'une fonction membre peut se faire de plusieurs manières selon que la fonction est une fonction membre d'instance ou bien de classe.

Les règles sont les suivantes.

Pour une fonction membre de classe

Dans ce cas, le critère important est le *lieu de l'appel*:

Appel dans la classe ou dans une classe descendante :

Une fonction membre statique s'appelle avec son nom au sens strict (cas A).

Appel hors de la classe et de ses descendantes:

Une fonction membre statique s'appelle avec son nom préfixé du nom de la classe et de **::** (cas B).

Pour une fonction membre d'instance

Dans ce cas, le critère important est l'instance sur laquelle on appelle la fonction:

Avec l'instance courante

une fonction membre d'instance s'appelle avec son nom au sens strict (cas C) [ou bien, mais c'est redondant, préfixé de `this->` (cas D)].

Avec une autre instance que l'instance courante

une fonction membre d'instance s'appelle avec son nom au sens strict préfixé du nom de l'instance et d'un point (.) (cas E) ou bien avec son nom au sens strict préfixé du nom d'un pointeur sur l'instance considérée et d'une flèche (->) (cas F).

Exemple

```
class Truc { …
    void a();
    void b();
    static void c();
    static void d();
};

void Truc::a() {
    d();                            //  cas A
    b();                            //  cas C
    this->b();                      //  cas D
    Truc t ; t.b();                 // cas E
    Truc * pt = &t ; pt ->b();      // cas F
}

void Truc::b() {…}

void Truc::c() {
    d();                            // cas A
    Truc t ; t.b();                 // cas E
    Truc * pt = &t ; pt ->b();      // cas F
}

void Truc::d() {…}

main() {
    Truc::d();                 // cas B
    Truc t ; t.b();                 // cas E
    Truc * pt = &t ; pt ->b();      // cas F
}
```

Bien sur, on peut être redondant :

```
void Truc::a() {
    Truc::d();                            //  cas A
    Truc::b();                            //  cas C
    Truc t ; t.Truc::b();                 //  cas E
    Truc * pt = &t ; pt ->Truc::b();      //  cas F
}
```

Le C++ acceptera et cela marchera mais il vaut mieux éviter.

8. Construction, destruction et initialisation d'objets

Ce paragraphe montre les différents comportements des objets C++ lors de leur durée de vie. Il précise les qualificatifs des objets suivant leur type de déclaration, comment ils sont initialisésles et surtout le cas du constructeur par recopie.

Les différents qualificatifs des objets suivant leur type de déclaration

les objets automatiques

Les objets automatiques sont les objets déclarés dans une fonction ou dans un bloc.

```
void f() {
    Truc t ;                // t est construit ici
    …
    {
        Bidule b ;    // b est construit ici
        …
    }                 // b est détruit ici
}                     // t est détruit ici
```

`t` et `b` sont des objets automatiques. `t` est visible dans la fonction `f` et `b` est dans le bloc. A la sortie de la fonction `t` est détruit. A la sortie du bloc, `b` est détruit.

les objets statiques

Un objet statique est un objet déclaré avec le mot-clé `static` dans une déclaration de classe ou dans une fonction ou bien à l'extérieur de toute fonction. Un objet statique est créé avant le début de l'exécution du programme et il est détruit à la sortie du programme.

Exemple :

```
static Point a(1, 7);
```

les objets dynamiques

Ce sont eux qui font tout l'intérêt de la programmation orienté objet. Un objet dynamique est un objet créé avec `new` et éventuellement détruit avec `delete`.

```
main () {
    point * adr;
    cout << "&& debut main" << endl;
    adr = new point(3, 7);
    fct(adr);
    cout << "&& fin main" << endl;
}
void fct (point * adp) {
    cout << "&& debut fct" << endl;
    delete adp;
    cout << "&& fin fct" << endl;
}
point::point(int x, int y) { cout << "++ appel constructeur" << endl; }
point::~point() { cout << "-- appel destructeur" << endl; }
```

Initialisation d'un objet lors de sa déclaration

Avec l'utilisation systématique des constructeurs (un constructeur crée un emplacement mémoire pour l'objet et l'initialise) C++ ne permet pas de créer un objet sans l'initialiser en même temps. Pour créer un objet on doit appeler un constructeur déclaré et défini ou bien le constructeur par défaut.

Avec la déclaration de la classe suivante :

```
class Point {
      int x ; int y ;
public :
      Point(int abs) { x = abs ; y = 0 ; }
      ...
} ;
```

on peut écrire :

```
point a(3);
a = 3;
```

Avec la déclaration de la classe suivante :

```
struct paire { int n ; int p ; } ;
class Point { …
    Point(paire q) { x = q.n ; y = q.p; }
} ;
```

on peut écrire :

```
paire s = { 3, 8 } ;
point a(s);
a = s;
```

Constructeur par recopie

Ce paragraphe est important, il traite du cas où le constructeur d'une classe possède un unique paramètre qui est une référence à un objet de la classe. On appelle alors ce constructeur, le *constructeur par recopie*. Ce cas est très utilisé en C++ car bien souvent, dans un programme, on veut dupliquer un objet (afin, par exemple, de travailler sur sa copie et de garder une sauvegarde). Pour dupliquer l'objet a dans un objet b, le programmeur écrira simplement :

```
Point b = a;
```

ou bien :

```
Point b(a);
```

On déclare le constructeur par recopie de la manière suivante :

```
Point (Point &);
```

Il y a alors 2 cas : soit le programmeur a écrit un constructeur par recopie, soit non.

Il n'existe pas un constructeur par recopie

Les valeurs des membres de l'objet sont recopiées, une à une, telles quelles. Cette approche est risquée. Elle ne marche que si les attributs de l'objet sont des valeurs et pas des pointeurs. Elle ne marche pas lorsque l'objet recopié pointe vers d'autres objets ou structures.

Exemple (qui ne marche pas) :

```
class vecteur {
    int nelement;
    double * adr;
    vecteur(int n);
    ~vecteur();
} ;
main() {
    vecteur a(5);
    vecteur b(a);
}
vecteur::vecteur(int n) {
    adr = new double[nelement = n];
    cout << "++ " << this << adr <<endl;
}
vecteur::~vecteur() {
    cout << "-- " << this << adr <<endl;
    delete adr;
}
```

La création de a est correcte : nelement vaut 5 et adr pointe sur un tableau de double de longueur 5. La création de b l'est presque : nelement vaut 5 car recopié depuis a, mais adr pointe aussi sur le tableau de double créé lors de la création de a. Autrement dit, a.adr et b.adr pointent tous les deux vers le même tableau. Lors de la destruction de a, le destructeur de la classe vecteur détruit le tableau pointé par adr et lors de la destruction de b, même chose. Conclusion, le programme se peut se planter lors de la deuxième destruction.

Il existe un constructeur par recopie

La déclaration d'un constructeur par recopie est du type :

```
Point (Point &);
```

Exemple (qui marche) :

```
vecteur::vecteur(vecteur & v) {
      nelement = v.nelement;
      adr = new double[t];
      for (int i=0; i<nelement; i++)
            adr[i] = v.adr[i];
      cout << "&& " << this << adr <<endl;
}
```

Ici, la création de `b` est différente de celle du paragraphe précédent : au lieu de recopier des valeurs de attributs de `a` dans ceux de `b`, le programme exécute le corps du constructeur par recopie ci-dessus. Il met la valeur de `a.nelement` dans `nelement`, c'est-a-dire 5, puis, au lieu de recopier `a.adr` dans `adr`, il crée un nouveau tableau de `double` de taille 5 et recopie un à un les cases du tableau de `double` pointé par `a.adr` dans les cases du nouveau tableau pointé par `adr`. Ce qui fait que `a.adr` et `b.adr` pointent vers des tableaux différents et aucun problème ne pourra survenir lors de la destruction de `a` ou de `b`.

Les exceptions

Les instructions

```
point a = point(1, 5);
```

ou :

```
point a = b;
```

peuvent avoir des comportements différents de ceux vus ci-dessus.

`point a = point(1, 5);` sera traitée comme: `point a(1,5);` on appelle le constructeur usuel.

Si l'opérateur `=` a été redéfini (cf paragraphe redéfinition d'opérateurs) alors `point a = b;` appellera cet opérateur au lieu du constructeur par recopie.

Objets membres

Un membre d'un objet peut éventuellement être un objet.

```
class Pointcol {
      Point p;
      int couleur;
      Pointcol(int, int, int);
};
Pointcol::Pointcol (int abs, int ord, int coul) : Point(abs, ord) { … }
```

Le constructeur de `Point` est appelé avant celui de `Pointcol`.
Pour les destructeurs, c'est l'ordre inverse.

On peut appeler une méthode de la classe `Point` :

```
Pointcol pc (1,2,3) ;
pc.p.affiche() ;
```

9. Héritage

Le concept d'héritage est fondamental en C++ (et en POO en général). Il permet d'écrire un programme concisément et il permet la réutilisation de composants logiciels.

La classe dérivée « hérite » de la classe de base.

1. 1er aperçu de l'héritage en C++

```
class Point { public:
    int x;
    int y;
    Point(int a, int b) { x = a; y = b; }; ~Point();
    void deplace(int dx, int dy) { x += dx; y += dy; };
    void affiche() { cout << "x = " << x << "y = " << y << endl; };
};
class PointCol : public Point { public:
    int couleur;
    PointCol(int, int, int); ~Point();
    void colore(int c) { couleur = c };
};
```

Dans cet exemple, on a défini une classe "dérivée" (`PointCol`) à partir d'une classe "de base" (`Point`).

On dit que la classe `PointCol` « hérite » de la classe `Point`.

Cela signifie que les membres de la classe `Point` sont des membres de la classe `PointCol`.

2. REdéfinition de fonctions membre et de membres

Une fonction membre de la classe dérivée peut avoir la même signature qu'une fonction de la classe de base. On dit alors que la fonction membre de la classe de base est « Redéfinie » dans la classe dérivée.

Quand on est dans la classe dérivée, on appelle la fonction de la classe dérivée de façon habituelle. Si on veut appeler la fonction membre de la classe de base, il faut préciser le nom de la classe de base `::` en préfixe. Exemple:

```
class PointCol : public Point { ...
    void affiche() {
        Point::affiche();
        cout << "couleur " << couleur << endl;
    };
};
main() { PointCol p(10, 20, 5);
    p.affiche();
    p.Point::affiche();
    p.deplace(2,4);
    p.affiche();
```

```
    p.colore(2);
    p.affiche();
}
```

Pour les membres, c'est analogue. Exemple:

```
class A { ... int a; char b; ... };
class B: public A {float A; ... };
```

Si `b` est de type `B`, `b.a` fait référence au membre a de type `float` de la classe `B`.

On accède au membre `a` de type `int` par `b.A::a`

Le membre a de la classe `B` s'ajoute à celui de la classe `A`. Il ne le remplace pas.

NB: on dit REdéfinition et pas SURdéfinition. (Surdéfinition signifie changement de signature de la fonction au sein d'une même classe).

3. Appel des constructeurs et destructeurs

En fait les constructeurs et destructeurs obéissent à des règles précises.

On peut transmettre les paramètres du constructeur dérivé au constructeur de base:

```
class PointCol : public Point { int couleur;
    public: PointCol(int x, int y, int c) : Point(x,y) { couleur = c };
~Point();
    ...
};
```

On peut toujours mettre des arguments par défaut.

Lors de l'appel du constructeur de la classe dérivée, il se produit dans l'ordre:

- appel du constructeur de base,
- exécution du corps du constructeur dérivé.

Lors de l'appel du destructeur de la classe dérivée, il se produit dans l'ordre:

- exécution du corps du destructeur dérivé,
- appel du destructeur de base.

```
class Point { ...
    Point (int abs=0, int ord=0) {
```

```
          x=abs; y = ord; cout << "++ construc Point " << x << " " << y <<
endl; };
    ~Point () {
          cout << "-- destruc Point " << x << " " << y " endl; };
    ...
}
class PointCol : public Point { ...
    PointCol (int abs=0, int ord=0, int c=1) : Point(abs, ord) {
          couleur = c;
          cout << "++ construc PointCol " << couleur << endl; };
    ~PointCol () {
          cout << "-- destruc PointCol " << couleur << endl; };
    ...
}
main () {
    PointCol a(10, 15, 3); PointCol b(2, 3); PointCol c(12); PointCol d;
    pointCol * p; p = new PointCol(12, 25); delete adr;
}
```

la sortie est :

```
    ++ construc Point 10 15
    ++ construc PointCol 3
    ++ construc Point 2 3
    ++ construc PointCol 1
    ++ construc Point 12 0
    ++ construc PointCol 1
    ++ construc Point 0 0
    ++ construc PointCol 1
    ++ construc Point 12 25
    ++ construc PointCol 1
    -- destruc PointCol 1
    -- destruc Point 12 25
    -- destruc PointCol 1
    -- destruc Point 0 0
    -- destruc PointCol 1
    -- destruc Point 12 0
    -- destruc PointCol 1
    -- destruc Point 2 3
    -- destruc PointCol 3
    -- destruc Point 10 15
```

Constructeur par recopie.

Si la classe dérivée n'a pas de constructeur par recopie, alors appel du constructeur par défaut (copie des attributs un à un).

Si la classe dérivée a un constructeur par recopie `B(B&b)`, il n'y a pas forcément d'appel automatique au constructeur par recopie de la classe de base: on peut le spécifier par
B (B & b) : A (b)

conversion implicite de `b` en un objet de la classe `A`.

4. Contrôle des accès

« private » et « public »

Une classe dérivée peut accéder aux attributs publiques de la classe de base et appeler les méthodes publiques de la classe de base. Une classe dérivée n'a pas accès aux membres privés de la classe de base. Un attribut publique (resp. privé) est déclaré avec le mot-clé « **public** » (resp. « **private** »).

Exemple :

```
class Point {
public:
    int x;
private :
    int y;
public :
    Point(int a, int b) { x = a; y = b; }; ~Point();
    void deplace(int dx, int dy) { x += dx; y += dy; };
    void affiche() { cout << "x = " << x << "y = " << y << endl; };
};
class PointCol : public Point { public:
    int couleur;
    PointCol(int, int, int); ~Point();
    void colore(int c) { couleur = c };
};
```

La classe `PointCol` peut accéder à `x` et ne peut pas accéder à `y`.

Les classes utilisatrices `U` de la classe dérivée `D` suivent les mêmes règles d'accès dans la classe de base que la classe dérivée `D`.

« protected »

Avec le mot-clé '**protected**' on faire en sorte que:

La classe dérivée a accès aux membres protégés de la classe de base.

Les classes utilisatrices de la classe dérivée non.

Exemple :

```
class Point { int x; int y;
    public: Point(int a, int b) { x = a; y = b; }; ~Point();
    protected:
    void deplace(int dx, int dy) { x += dx; y += dy; };
    void affiche() { cout << "x = " << x << "y = " << y << endl; };
};
```

Intérêt du statut protégé:
Les membres protégés sont comparables à des membres privés pour un utilisateur de la classe et à des membres publiques pour la classe dérivée.

Mode de dérivation

Le mode de dérivation est le mot-clef qui précède le nom de la classe de base lorsque l'on définit une classe dérivée :

```
class B : private A { ...
```

La classe dérivée a accès aux membres publics de la classe de base.

Les classes utilisatrices de la classe dérivée non.

On parle de *mode de dérivation publique ou privée*.

Ne pas confondre mode de dérivation (`private` ou `public`) placé avant les `{}` et statut de visibilité d'un membre (`private`, `protected` et `public`) entre `{}`.

Rappel: les mots-clés `private` `protected` et `public` peuvent apparaître autant de fois que l'on veut dans une déclaration de classe.

5. L'héritage en général

Ce paragraphe relie le vocabulaire employé en C++ avec celui de l'orienté objet en général.

Vocabulaire pour deux classes A et B en relation d'héritage.

Si B hérite de A on dit aussi :

B est dérivée de A	C++	A est la classe de base de B
B est un A	objet	
B spécialise A		A généralise B
B est une classe fille de A		A est la classe mère de B
B est une sous-classe de A		A est une super-classe de B
B is a A	anglais	

On parle de la relation est-un ou dela relation is-a.

Dans le cas ou plusieurs classes hérite les unes des autres, on parle de *taxonomie* de classes ou d'*arbre d'héritage.*

C est un B D est un B E est un B G est un F B est un A F est un A

On dit que C est une descendante de A

A est une ascendante de C

Si une dérivation multiple contient une seule dérivation privée, alors elle est privée.

En C++ l'héritage multiple est possible mais déconseillé.

6. Exploitation d'une classe dérivée

Rappel du paragraphe « Classes et objets » « Exploitation des .cpp et .h »

Un mode d'utilisation en plus: l'héritage

dans le B.h on met
```
#include "A.h"
```

7. Compatibilité entre objets d'une classe de base et objets d'une classe dérivée

Puisque la relation d'héritage est une relation forte, on peut se demander si l'on peut affecter un objet de la classe de base dans un objet de la classe dérivée ou pas, et inversement.

L'idée est que tout objet de la classe dérivée est un objet de la classe de base et que C++ fait des conversions implicites de type.

On peut mettre un objet de type dérivé vers un type de base mais pas l'inverse.

Idem : *on peut mettre un pointeur sur objet de type dérivé vers un pointeur sur objet de type de base et pas l'inverse.*

Si B est une classe dérivée de la classe A et si on a :
```
A a; B b;
```

alors `a = b ;` est légal alors que `b = a ;` est illégal.

Pour s'en convaincre, imaginons que la classe `C` dérive de la classe `A` et que l'on ait : `C c;` Puisque `c` est dans la classe `C`, il est dans la classe `A` et donc si on pouvait faire `b=a;` on pourrait faire `b=c;` ce qui est absurde. Donc on ne peut pas faire `b=a;`

Si on a :
```
A * pa; B * pb;
```

alors `pa = pb ;` est légal alors que `pb = pa ;` est illégal.

Par contre, très souvent, dans une classe dérivée, le programmeur sait que le pointeur `pa` est de la classe dérivée donc il peut violer la règle avec l'opérateur de cast en écrivant :

```
pb = (B*) pa ;
```

Evidemment, cette opération peut amener des catastrophes si l'objet n'est pas de la classe dérivée `B` mais d'une autre classe dérivée `C`, complètement différente de la classe `B`.

Le typage des objets est statique en C++. Cela signifie que le type d'un objet est connu à la compilation et ne change plus ensuite.

```
Point p(3, 5); PointCol pc(8,6,2); Point * pa = &p; PointCol * pb = &pc;
```

`pa->affiche()` appellera `Point::affiche()`
`pb->affiche()` appellera `PointCol::affiche()`

Mais si on fait : `pa = pb;` malheureusement `pa->affiche()` appelle `Point::affiche()`.

☹

Le choix de la méthode à appeler est fait statiquement (c'est-à-dire par le compilateur).

Par exemple, `pa->colore(4);` est une instruction illégale.

10. Fonctions virtuelles

Rappel:

Un pointeur sur un type d'objet peut recevoir l'adresse de n'importe quel objet descendant mais il existe une lacune de taille: avec ce pointeur on ne peut appeler que des méthodes correspondant au type du pointeur et non pas au type effectif de l'objet pointé. Cette lacune vient du fait du *typage statique* des pointeurs : le type des pointeurs est déterminé à la compilation.

```
class Point { ... void affiche(); ... }
class Pointcol: public point { ... void affiche(); ... }
Point p;
Pointcol pc;
Point * adp = &p;
adp->affiche();
```

appelle la méthode `affiche` de la classe `Point`.

```
adp = &pc;
adp->affiche();
```

appelle la méthode `affiche` de la classe `Point` aussi bien que `adp` pointe vers un objet de type `Pointcol`. On a un *typage statique.*

Pour que l'on puisse appeler une méthode de l'objet descendant, il faut que le typage du pointeur soit déterminé à l'exécution. On parle alors de *typage dynamique*.

Les fonctions virtuelles et le typage dynamique:

```
class Point { ...
      void virtual affiche() {
            cout << « Point »  << endl ;
      } …
}
class Pointcol: public Point { ...
      void affiche() {
            cout << « Pointcol » << endl ;
      } ...
}
Point p;
Pointcol pc;
Point * adp = &p;
adp->affiche();
```

appelle la méthode `affiche` de la classe `Point`.

```
adp = &pc;
adp->affiche();
```

A la compilation, le compilateur ne décide pas quelle méthode appeler. Le compilateur met en place un mécanisme qui lui permettra de décider à l'exécution quelle méthode appeler en fonction du type de l'objet pointé.

A l'exécution, cette instruction appelle la méthode affiche de la classe `Pointcol` car `adp` pointe vers un objet de type `Pointcol`. On a une *liaison dynamique*.

`affiche()` est une fonction membre *virtuelle*.
le mot-clé '**virtual**' le spécifie.

le programme suivant :

```
Point p;
Pointcol pc;
Point * adp = &p;
adp->affiche();
adp = &pc;
adp->affiche();
```

a pour sortie :

```
Point
Pointcol
```

Le mécanisme d'identification des objets:

Comment fait le compilateur pour implémenter les fonctions virtuelles ?

```
class point { ...
      void virtual deplace();
      void virtual identifie(); .... }
class pointcol: public point { ... void identifie(); ... }
```

Pour une classe comportant au moins une fonction virtuelle, il existe une table des adresses de ses fonctions membres:

```
&point::identifie      = @1
&point::deplace        = @2
&pointcol::identifie   = @3
&pointcol::deplace     = @1

&point::deplace = &pointcol::deplace
```

En plus des emplacements mémoires nécessaires à ses membres, tout objet d'une classe comportant au moins une fonction virtuelle possède un emplacement mémoire pour un pointeur contenant l'adresse de sa table des adresses des fonctions virtuelles.
A l'exécution le C++ choisit la bonne table.

Les fonctions virtuelles en général:

1. La redéfinition d'une fonction virtuelle n'est pas obligatoire.

2. Un constructeur ne peut pas être virtuel.

3. Un destructeur peut être virtuel.

11. Templates

Les « templates » (signifie « patron » ou « modèle » en anglais) sont des types paramétrés. On peut définir un template de fonction ou un template de classe.

Template de fonctions

Supposons que l'on veuille généraliser la fonction `echange` qui échange deux entiers `int` à d'autres types de données ou objets.

```
void echange(int & a, int & b) {
      int c = a ;
      a = b ;
      b = c ;
}
```

Il faudrait surcharger la fonction échange avec autant de fonctions `echange` que de types de données ou d'objets sur lesquels appliquer la fonction `echange`.

C++ propose le mécanisme de template de fonctions. Le programmeur écrit :

```
template <class T>
void echange(T & a, T & b) {
      T c = a ;
      a = b ;
      b = c ;
}
```

Ainsi il définit un template de fonctions. A chaque fois que le compilateur C++ rencontrera un appel de la fonction echange pour des données de type `Truc`, il engendrera le code de la fonction `echange` pour le type `Truc`.

```
main() { ...
      Truc x = 1 ;
      Truc y = 2 ;
      Echange(x, y) ;
      ...
}
void echange(Truc & a, Truc & b) { // fonction engendree automatiquement
      Truc c = a ;
      a = b ;
      b = c ;
}
```

Template de classes

On peut aussi paramétrer les classes avec des types et/ou des constantes.

```
template <class T, int N>
class V  { …
      const int taille = N ;
      T * v ;
      V() { v = new T[N]; }
      ~V() { delete [] v; }
} ;
```

Ici les paramètres sont les types `T` et la constante `N`.

Lors de la première déclaration
```
V<Machin, 4> x ;
```

le compilateur engendrera la classe
```
class V<Machin, 4>  { …
      const int taille = 4 ;
      Machin * v ;
      V() { v = new Machin[4]; }
      ~V() { delete [] v; }
} ;
```

Bien sûr, les paramètres des templates de classe doivent être soient des types (au sens large, c'est-à-dire des types prédéfini du langage ou des classes définies par le programmeur) soient des constantes (par exemple n dans l'exemple ci-dessus).

Les déclarations suivantes sont donc correctes :
```
V<Truc, 10> y;
V<Bidule, 100> z;
```

Le programmeur peut imbriquer les templates. Si il veut définir une classe liste de listes de `Truc`, il déclare :
```
Liste<Liste<Truc>> ll ;
```

Généricité et généralisation

Avec des templates, on peut définir des classes Liste<Point> ou Liste<Truc>.

Il suffit d'avoir un patron de classe `Liste` :
```
template <class T>
class Liste {  ...
      void ajouter(T *);
      int enlever(T *);
      T * premier();
      T * prochain();
      ...
};
```

Utiliser des templates s'appelle aussi la *généricité* ; la généricité est concurrente de la *généralisation*.

Inconvénient de la généricité:
Autant de classes `Liste<T>` que de classes différentes. Taille plus importante du code

Avantage :
Ça va plus vite à l'exécution.

La STL est une librairie de templates standards.

12. la classe `vector` de la Standard Template Library

C++ contient une librairie de patrons standards, écrite par Meng Lee et Alex Stepanov, la Standard Template Library (STL). L'idée de la STL est *d'offrir à l'utilisateur des composants abstraits dont l'utilisation est toujours efficace en temps*. La conception de ces composants suit l'idée qu'un *algorithme*, par exemple la recherche dichotomique, est indépendant des *conteneurs*, par exemple un vecteur ou une liste. Pour cela la STL définit des *itérateurs*, des *fonctions objets* (cf surdéfinition d'opérateurs au paragraphe 13) ou des *adaptateurs*. Les itérateurs sont des pointeurs généralisés, c'est-à-dire des objets d'une classe pour laquelle l'opérateur de déréférencement * à été surdéfini. Pour résumer, les patrons standards ou «composants» sont de cinq catégories :

- les *conteneurs*, (vector, list, deque, map, set)
- les *algorithmes*,
- les *itérateurs*,
- les *adaptateurs*
- et les *fonctions objets*.

Nous recommandons vivement au programmeur C++ confirmé souhaitant utiliser la STL de lire le tutorial de Johannes Weidl, disponible sur le Web ou le document de référence des auteurs de la STL. Néanmoins ce paragraphe se fixe l'objectif limité de décrire l'utilisation du template `vector` de la STL sur un exemple très simple. Cela permet de comprendre le minimum sur la STL.

Un `vector<T>` est un vecteur contenants des `T`. Par exemple, `vector<Ballon*>` désigne un vecteur de pointeurs sur des ballons. Pour utiliser un template de vector, il faut mettre #include <vector>. Prenons l'exemple d'un attribut de classe contenant les adresses des instances d'une classe, (ici la classe Ballon).

```
class Ballon { public:
    static vector<Ballon*> * instances; // les instances
    …
};
```

Le type `vector<Ballon*>` déclare un tableau unidimensionnel contenant des `Ballon*`. Sa taille s'adapte dynamiquement au nombre d'éléments du vecteur. `instances` est donc un pointeur vers un vecteur de `Ballon*`. Pour insérer une adresse de ballon, il faut faire :

```
    instances->insert(instances->begin(), this);
```

(nb : l'insertion est faite au début par choix). Pour enlever une adresse de ballon de `instances`, ne connaissant pas sa place dans le vecteur, il faut faire une boucle:

```
    for (Ballon**b=instances->begin(); b !=instances->end() ; b++)
          if (this==*b) { instances->erase(b) ; break ; }
```

Les méthodes `Ballon** begin()`, `Ballon** end()`, `insert(Ballon**, Ballon*)`, `erase(Ballon**)` sont définies implicitement par la classe `vector` de la STL. `Ballon** begin()` retourne le premier *itérateur* (de type `Ballon **`) du vecteur. `Ballon** end()` retourne le dernier. `insert(Ballon**, Ballon*)` permet d'insérer un élément à une position (au début dans l'exemple ci-dessus) et `erase(Ballon**)` permet d'enlever un élément du vecteur connaissant sa position.

Un itérateur est une variable permettant de parcourir le vecteur ou de désigner une position dans le vecteur. *Son type par défaut est un type pointeur sur le type contenu du vecteur*. Donc le type de l'itérateur d'un vecteur de `Ballon*` est `Ballon**`. C'est pourquoi la variable de la boucle `for` ci-dessus parcourant le vecteur est de type `Ballon**`.

Supposons que l'on rajoute une méthode retournant l'adresse de l'instance de la classe `Ballon` ayant un nom connu (de type `string`) , donné en entrée :

```
class Ballon {public:
          static Ballon * getInstance(string);
    …
};
```

Pour écrire son contenu, on utilisera une boucle du style :

```
    for (Ballon**b=instances->begin(); b !=instances->end() ; b++)
          if (((Ballon*)*b)->nom==s) return *b ;
    return NULL;
```

13. Surdéfinition d'opérateurs

Ce paragraphe présente la *surdéfinition* d'opérateurs. Cette partie du C++ n'est pas indispensable pour programmer en C++. Elle permet premièrement *d'écrire des classes dont l'utilisation suit exactement la syntaxe des opérateurs prédéfinis* tels que +, -, * , /, =, >>, <<, ==, etc. De là, elle permet de *programmer dans l'esprit de la STL* (Standard Template Library). Vocabulaire : dans la suite on utilisera indifféremment les mots «surcharge», «surdéfinition» et redéfinition. Tout au long de ce paragraphe, l'exemple suivi est la classe `Rationnel`.

```
class Rationnel {
private :
      int num ;
      int den ;
public :
      Rationnel(int n = 0, int d = 1) { num = n; den = d ; }
      ...
} ;
```

Surcharge de l'opérateur d'affectation =

Pour surcharger un opérateur fondamental du C++, par exemple =, on rajoute la déclaration de `operator=` :

```
      Rationnel & operator=(const Rationnel &);
```

et sa définition :

```
      Rationnel & Rationnel::operator=(const Rationnel & r) {
            num = r.num;
            den = r.den ;
            return *this ;
      }
```

Noter que l'opérateur = est vu comme une fonction membre de la classe `Rationnel` dont le nom est `operator=`. Cette fonction membre passe ses paramètres (en entrée et en retour) par référence. Noter que l'on met `const` pour que le C++ vérifie que l'objet r ne soit pas modifié au cours de la fonction membre. Noter la valeur de retour `*this` de cet opérateur : on retourne le déréferencement de `this`, adresse de l'objet courant; donc cet opérateur retourne l'objet courant.

Sur les exemples suivants, noter la différence entre une initialisation avec un constructeur par recopie (cf paragraphe 8 sur la construction des objets) et une affectation avec la surcharge de l'opérateur =.

```
Rationnel x(22, 7) ;    // constructeur défini par l'utilisateur
Rationnel y(x) ;        // constructeur par recopie
Rationnel w ;           // constructeur par défaut
w = x ;                 // affectation
```

L'opérateur d'affectation = est un « bon » opérateur à surcharger car sa syntaxe, de gauche à droite, suit celle de l'orienté objet . Dans l'expression `x = y ;` x joue le rôle de l'instance sur laquelle on appelle la fonction membre et y joue le rôle du paramètre de la fonction membre. Cette expression est donc analogue à l'expression « objet » `x . operator= (y) ;` .

D'autres opérateurs ne sont pas « bons » en ce sens, par exemple les opérateurs arithmétiques.

Surcharge de l'opérateur produit *

Prenons l 'exemple de l'opérateur produit *. Il y a beaucoup de cas à traiter et de multiples façons de faire.

Premièrement, on peut *ne pas* surcharger l'opérateur * dans la classe `Rationnel` et utiliser une simple fonction membre produit :

```
class Rationnel { ...
      Rationnel produit(Rationnel) ;     // declaration
} ; ...
Rationnel Rationnel::produit(Rationnel y) { //definition
      Rationnel z(num*y .num, den*y.den);
      return z ;
}
...
Rationnel a(3, 4) ;
Rationnel b(9, 5) ;
Rationnel c = a.produit(b) ;       // utilisation banale
```

Si on souhaite avoir une uitlisation uniforme avec les types prédéfinis :

```
Rationnel a(3, 4) ;
Rationnel b(9, 5) ;
Rationnel c = a*b ;                // utilisation habituelle
```

Il suffit de faire comme cela :

```
Rationnel operator*(Rationnel) ;   // declaration
...
Rationnel Rationnel::operator*(Rationnel y) { //definition
      Rationnel z(num*y .num, den*y.den);
      return z ;
}
```

Mais, * étant commutatif (a*b=b*a), on souhaite garder la symétrie entre les deux opérandes de * et on utilise une fonction *non membre* `produit` :

```
// def a l'exterieur la classe Rationnel :
Rationnel operator*(Rationnel x, Rationnel y) {
      Rationnel z(x.num*y .num, x.den*y.den);
      return z ;
}
...
Rationnel a(3, 4) ;
Rationnel b(9, 5) ;          // utilisation symétrique
Rationnal c = a * b ;        // avec operateur non membre
```

Mais cela ne marche pas encore tout à fait car, `operator*` étant un opérateur *non membre*, il n'a pas accès aux membres `num` et `den` (qui sont des membres privés). Pour palier à cet inconvénient sans mettre `num` et `den` en membres publiques, il faut mettre cet opérateur en *fonction am*ie de la classe `Rationnel` avec le mot-clé `friend`:

```
class Rationnel {
private :
      int den ;
      int num ;
public :
      friend Rationnel operator*(const Rationnel&, const Rationnel&) ;
      ...
} ;
...
Rationnel operator*(Rationnel & x, Rationnel & y) {
      Rationnel z(x.num*y .num, x.den*y.den);
```

```
        return z ;
    }               // voilà du C++ pur et dur !
```

Noter que l'on met `const` pour éviter une modification des paramètres dans la fonction et que l'on met des références pour éviter les recopies inutiles.

Conclusion : quand la syntaxe d'un opérateur ne permet pas de le surcharger en fonction membre de la classe (pour des raisons pas toujours défendables de symétrie) on le surcharge en fonction amie de la classe. Le programmeur non soucieux de la symétrie peut faire comme au début de ce paragraphe.

Surcharge d'opérateurs de flux >> et <<

On souhaite surcharger >> et <<. Au lieu de faire banalement comme cela :

```
class Rationnel { ...
      void imprimer() ;
      ...
} ; ...
void Rationnel : :imprimer() {
      cout << num << « / » << den;
}...
Rationnel a(3, 4) ;
a.imprimer() ;          // utilisation banale
```

on veut faire comme cela :

```
class Rationnel {
private :
      int den ;
      int num ;
public :
      friend ostream & operator<<(ostream &, const Rationnel &)
      ...
} ; ...
ostream & operator<<(ostream & ostr, const Rationnel & x) {
      return ostr << x.num << « / » << x.den ;
}
Rationnel a(3, 4) ;
cout << a;
```

Ici le problème de la surcharge des opérateurs de flux est différent de celui des opérateurs arithmétiques. Les paramètre de retour et le premier paramètre sont des streams et pas des `Rationnel`. Il est impossible de les surcharger comme fonctions membres de la classe `Rationnel`. Ici, le programmeur est *obligé* d'utiliser les fonctions amies.

Pour surcharger l'opérateur >> on fait comme cela :

```
      friend istream & operator>>(istream &, Rationnel &)
      ...
istream & operator<<(istream & istr, const Rationnel & x) {
      cout << « numerateur ? » ;  istr >> x.num ;
      cout << « denominateur ? » ;  istr >> x.den ;
      return istr;
}
cin >> a;
cout << a;
```

Surcharge de l'opérateur d'appel (), « fonctions objets »

Ce paragraphe décrit la redéfinition de l'opérateur d'appel `operator()` et les «fonctions objets». Une fonction objet est un objet d'une classe dont l'opérateur d'appel `operator()` est redéfini.

Soit par exemple, une fonction de comparaison d'un entier avec un seuil :

```
static int seuil = 4 ;
boolean f(int v) { return (v<seuil) ; }
```

En C, si l'on veut passer cette fonction en paramètre d'une autre fonction, on passe généralement un pointeur sur cette fonction. En C++, on définit cette fonction comme un objet d'une classe, ici par exemple, la classe `Comparaison` :

```
class Comparaison {
private :
      int seuil ;
public :
      Comparaison(int s) { seuil = s ; }
      boolean operator()(int v) { return (v<seuil) ; }
      ...
} ;
Comparaison inferieurAQuatre(4) ;  // création de l'objet fonction
```

On utilise l'objet fonction avec son opérateur d'appel ().

```
cout << (inferieurAQuatre(3) ? « yes » : « no »)  ;  // appel
cout << (inferieurAQuatre(4) ? « yes » : « no »)  ;  // appel
cout << (inferieurAQuatre(5) ? « yes » : « no »)  ;  // appel
```

La sortie sera : yes no no

L'appel `inferieurAQuatre(3)` est équivalent à `inferieurAQuatre.operator()(3)`. On peut ensuite utiliser l'objet `inferieurAQuatre` pour le passer en paramètre d'une autre fonction. Cette technique est à la base de la STL (Standard Template Library).

14. Les fonctions amies

En POO pure, l'encapsulation des données est obligatoire. Dans ce cadre, une fonction membre d'une classe ou une fonction indépendante ne peut accéder aux données privées d'une autre classe.

Afin de rendre accessibles les données privées d'une classe à une fonction, on déclare cette fonction amie avec le mot-clé **friend**.

Exemple

```
class Point { int x; int y;
    public: Point(int a, int b) { x = a; y = b; };
    ~Point();
    ...
    friend int coincide(Point, Point);
    ...
}; ...
int coincide(Point p, Point q) {
    if ((p.x==q.x)&&(p.y==q.y)) return 1 ;
    else return 0 ;
}
```

Pour qu'une fonction membre machin (char, Point) de la classe Bidule soit amie de la classe Point on écrira :

```
class Point { ...
    friend Bidule::machin(char, Point);
    ...
};
```

Une fonction peut être amie de plusieurs classes.

Un ensemble de fonctions peuvent être amies d'une classe. En particulier toute une classe peut être amie d'une autre. Pour dire que toutes les fonctions membres de la classe Bidule sont amies de la classe Point, on notera :

```
class Point { ...
    friend class Bidule;
    ...
};
```

Les fonctions amies sont utilisées mais rarement. Des exemples sont fournis au paragraphe pour surcharger les opérateurs *, >> ou <<.

15. Entrées sorties « élaborées »

Ce paragraphe traite des entrées sorties de manière plus approfondie. Les entrées sorties minimales présentées au paragraphe 3 étaient des entrées sorties *formattées*. Il existe des primitives de plus bas niveau que >> et << pour faire des entrées sorties élaborées.

Il existe des classes de flux, utilisées pour le traitement de flux de haut niveau, qui héritent de la classe de base **ios** :

istream	est-un	ios
ifstream	est-un	istream
istrstream	est-un	istream
ostream	est-un	ios
ofstream	est-un	ostream
ostrstream	est-un	ostream
iostream	est-un	istream
	est-un	ostream
fstream	est-un	iostream
strstream	est-un	iostream

et il existe la classe **streambuf** pour le traitement de flux de bas niveau.

Les **o** signifient classe de flux de sortie
Les **i** signifient classe de flux d'entrée
Les **f** signifient classe traitement de fichiers
Les **str** signifient classe pour traitement de chaînes de caractères

La classe ios

C'est la classe de base pour les ios. Son but est contrôler le tampon correspondant à un objet flux instancié en extrayant ou en insérant des caractères dedans.

Sa déclaration est :

```
class ios {
public :
    typedef unsigned long fmtflags ;    // chaine de 32 bits
    typedef unsigned char iostate ;     // chaine de 8 bits
    ...
protected :
    streambuf*  _strbuf     // pointe sur le tampon
    ostream*    _tie        // pointe sur ostream lie a istream
    int         _width      // taille du champ de sortie
    int         _prec ;     // precision pour les reels
    char        _fill ;     // bourrage
    fmtflags    _flags ;    // drapeaux
    iostate     _state ;    // etat courant
    ios(streambuf* _strbuf = 0, ostream* _tie = 0) ;
    ~ios() ;
    ...
};
```

Les constructeurs et destructeurs de ios sont *protégés*, ce qui signifie que cette classe n'est pas instanciable hors de la classe.

Les membres aussi, ils ne sont pas accessibles à un utilisateur hors de la classe.

Un objet flux est une instance d'une classe dérivée de ios.

Les attributs de ios :

_strbuf relie l'objet flux à son tampon.
_tie lie un objet d'entrée (comme cin) à un objet de sortie (comme cout).
_width définit la largeur du champ de sortie
_prec définit le nombre de chiffres après la virgule pour les réels
_fill est le caractère de remplissage utilisé entre la fin de la sortie effective et la fin du champ de sortie.
_flags et **_state** sont des chaînes de bits qui contiennent des paramètres booléens.

Pour accéder à ces attributs, ios fournit un ensemble de primitives de lecture publiques :

```
class ios {
public :
    streambuf*  rdbuf()     const { return _strbuf ; }
    ostream*    tie()       const { return _tie ; }
    int         width()     const { return _width ; }
    int         precision() const { return _prec ; }
    char        fill()      const { return _fill ; }
    long        flags()     const { return _flags ; }
    int         rdstate()   const { return _state ; }
    ...
};
```

exemple :

```
main() {
    cout << « cout.width()     = «     << cout.width()   << endl ;
    cout << « cout.precision() = «     << cout.precision() << endl ;
    cout << « cout.fill()      = [«    << cout.fill() << « ] »  << endl ;
    cout << « cin.flags()      = «     << oct << cin.flags()   << endl ;
    cout << « cout.flags()     = «     << oct << cout.flags()  << endl ;
}
```

```
    cout.width()     = 0
    cout.precision() = 6
    cout.fill()      =  [ ]
    cin.flags()      = 20000000001
    cout.flags()     = 20000000041
```

La taille du champ de sortie de cout est 0.
La précision sur les réels est 6 chiffres.
Le caractère de remplissage est le blanc.
_skips vaut 1 pour cin ce qui signifie que cin saute les espaces, les tabulations, passages à la ligne, sauts de page.

ios fournit un ensemble de primitives d'écriture publiques :

```
class ios {
public :
    int         width(int w) ;
    int         precision(int p) ;
    char        fill(char c) ;
    long        flags(long f) ;
    ...
};
```

Ces fonctions retournent l'ancienne valeur et la remplace pour la nouvelle.

exemple :

```
main() {
    cout.fill('#');
    cout.width(10);
    cout << «12345 » << endl ;
    cout << «12345 » << endl ;
}
```

#####12345
12345

_fill et _width sont changés pour le premier appel à cout << mais ils reprennent leur valeur par défaut pour le deuxième appel.

Les drapeaux de format de ios

Les drapeaux de format ont l'effet suivant si positionné :

01	skipws	saute les blancs
02	left	sortie justifiée à gauche
04	right	… droite
010	internal	sortie num. justifiée à droite ; signe ou base justifié à gauche
020	dec	base 10
040	oct	base 8
0100	hex	base 16
0200	showbase	préfixe avec la base 0 (octal) 0x (hexa)
0400	showpoint	réels avec point
01000	uppercase	majuscule
02000	showpos	entiers positifs avec +
04000	scientific	notation scientifique pour réels
010000	fixed	notation « fixée » des réels
020000	unitbuf	vidage du flux après chaque insertion
040000	stdio	vidage de stdio et stderr après chaque insertion

La fonction **flags**() positionne les drapeaux à de nouvelles valeurs. Mais elle réinitialise les drapeaux non spécifiés dans l'appel… ☹

exemple :

```
main() {
    int n = 234 ;
    long oldf = cout.flags(ios : :hex | ios : :uppercase) ;
    cout << n << endl ;
    cout.flags(ios : :hex | ios : :showbase) ;
    cout << n << endl ;
    cout.flags(oldf) ;
    cout << n << endl ;
}
    EA
    0xea
    234
```

Pour positionner un drapeau sans effacer les autres on doit utiliser la fonction **setf**() ☺

exemple :

```
main() {
    int n = 234 ;
    long oldf = cout.setf(ios : :hex | ios : :uppercase) ;
    cout << n << endl ;
    cout.setf(ios : :hex | ios : :showbase) ;
    cout << n << endl ;
    cout.flags(oldf) ;
    cout << n << endl ;
}
    EA
    0xEA
    234
```

Pour remettre à zéro des drapeaux, on utilise **unsetf**(). On peut remettre à jour des ensembles de bits :

0160	basefield
016	adustfield
014000	floatfield

exemple :

```
main() {
    char buffer[80] ;
    cin.unsetf(ios : :skipws) ;
    cin >> buffer ; cout << « [«  << buffer << « ] » << endl ;
    cin >> buffer ; cout << « [«  << buffer << « ] » << endl ;
    cin >> buffer ; cout << « [«  << buffer << « ] » << endl ;
    int n = 234 ;
    cout.setf(ios : :hex | ios : :uppercase | ios : :showbase) ;
    cout << n << endl ;
    cout.setf(ios : :basefield) ;
    cout << n << endl ;
}
      hello world
    [ ]
    [ hello]
    [ world]
    0XEA
    234
```

La variable d'état de ios

0 goodbit
01 eofbit
02 failbit
04 badbit

_state est la variable d'état du flux ; elle est modifiée par une entrée sortie.

On peut lire _state avec rdstate()

Exemple

```
main() {
    cout << « cin.rdstate() = » << cin.rdstate() << endl ;
    int n;
    cin >> n ;
    cout << « cin.rdstate() = » << cin.rdstate() << endl ;
}
    cin.rdstate() = 0
    22
    cin.rdstate() = 0
```

exécution normale

```
    cin.rdstate() = 0
    ^D
    cin.rdstate() = 3
```

Les classes istream et ostream

istream définit l'opérateur de flux >> et cin,
ostream définit l'opérateur de flux << et cout.
On appelle les entrées sorties faites avec >> ou <<, les entrées sorties *formattées* car elles reconnaissent les types de base et les caractères d'insertion et d'extraction.

On peut aussi faire des entrées sorties, dites *non formattées*, avec d'autres fonctions membres de la classe istream ou ostream :

Fonctions d'entrée non formattées

Lecture de caractères avec cin.**get**()

```
main() {
    char c ;
    while ((c=cin.get()) != EOF)
          cout << c ;
    cout << endl ;
}
    c'est pas bien de repeter betement
    c'est pas bien de repeter betement
    deux fois la meme chose
    deux fois la meme chose
    ^D
```

Chaque appel à cin.get() lit un caractère et cout << l'insère dans la sortie.
Ces caractères sont stockés dans le tampon. Celui-ci est vidé à chaque fin de ligne.
^D termine la boucle.

Il existe une autre fonction get dans cin :

```
    istream& get (char& c) ;
```

Elle renvoie « faux » lorsqu'elle détecte une fin de fichier.

```
main() {
    char c ;
    while (cin.get(c))
          cout << c ;
    cout << endl ;
}
    c'est encore moins bien de repeter
    c'est encore moins bien de repeter
    quatre fois la meme chose
    quatre fois la meme chose
    ^D
```

Une troisième forme de get est identique à **getline** :

```
    istream& getline(char* buffer, int n, char delim = '\n') ;

main() {
    char buffer[80] ;
    cin.getline(buffer, 8) ;
    cout << « [«  << buffer << « ] » << endl ;
    cin.getline(buffer, sizeof(buffer)) ;
    cout << « [«  << buffer << « ] » << endl ;
}
    1234567890ABCDEFGH
[1234567]
[890ABCDEFGH]
```

La fonction **ignore** avale les caractères du flux d'entrée.

istream& ignore(int n= 1, int delim = EOF) ;

```
main() {
    int jour, an ;
    cout << « entrez la date (jj/mm/aa) : « ;
    cin >> jour ;
    cin.ignore() ;
    cin.ignore(80, '/') ;
    cin >> an ;
    cout << » jour = «  << jour «  << « , an = 19«  << an << endl ;
}
    Entrez la date (jj/mm/aa) : 26/2/98
    Jour = 28, an 1998
```

Il existe encore d'autres fonctions sur cin

peek()
putback()
read()

Fonctions de sortie non formattées

Dans ostream, sont définies des fonctions de sortie non formatées.

La fonction **put** est surdéfinie :

```
    int put(char c) ;
    ostream& put(char c) ;
```

Elles insèrent toutes les deux le caractères c dans le flux de sortie.

Exemple :

```
main() {
    char c ;
    while (cin.get(c))
          cout.put(c) ;
    cout << endl ;
}
```

La fonction **write**() possèdent les fonctions inverses de celle de read.

```
    ostream& write(const char * buffer, int n) ;
    ostream& write(const unsigned char * buffer, int n) ;
```

Traitement des fichiers

Pour écrire dans un fichier :

```
#include <iostream.h>       // definition du flux cout
#include <fstream.h>        // definition de la classe ofstream
#include <stdlib.h>         // definition de exit
main() {
    ofstream fichierSortie(« monFichier.dat », ios : :out) ;
    if ( !fichierSortie) {
          cerr << « erreur : ouverture fichier impossible... » << endl ;
          exit(1) ;
    }
    char id[9], nom[16] ;
    int note ;
    cout << « \t1 : « ;
    int n = 1 ;
    while (cin >> id >> nom >> note) {
          fichierSortie << nom << «  «  << id << «  «  << note << endl ;
          cout << « \t » << ++n << «: « ;
    }
    fichierSortie.close() ;
}
```

Pour lire dans un fichier :

```
#include <iostream.h>       // definition du flux cout
#include <fstream.h>        // definition de la classe ifstream
#include <stdlib.h>         // definition de exit
main() {
    ifstream fichierEntree(« monFichier.dat », ios : :in) ;
    if ( !fichierEntree) {
          cerr << « erreur : ouverture fichier impossible... » << endl ;
          exit(1) ;
    }
    char id[9], nom[16] ;
    int note, somme = 0, compte = 0 ;
    while (fichierEntree >> id >> nom >> note) {
          somme += note ;
          ++count ;
    }
    fichierEntree.close() ;
    cout << « la moyenne vaut «  float(sum)/count << endl ;
}
```

Pour modifier un fichier :

```
#include <iostream.h>       // definition du flux cout
#include <fstream.h>        // definition de la classe fstream
#include <stdlib.h>         // definition de exit
main() {
    fstream fichierModifie(« monFichier.dat », ios : :in | ios : :out) ;
    if ( !fichierModifie) {
          cerr << « erreur : ouverture fichier impossible... » << endl ;
          exit(1) ;
    }
    char c ;
    while ((c=fichierEntree.get()) != EOF) {
          if (islower(c)) {
                fichierModifie.seekp(-1, ios : :cur) ;
                fichierModifie.put(toupper(c)) ;
          }
    }
    fichierEntree.close() ;
}
```