# Introduction à la programmation orientée objet, illustrée par le langage C++

Patrick Cégielski
cegielski@u-pec.fr

Mars 2002

*Pour Irène et Marie*

# Legal Notice


Université Paris Est Créteil - IUT
Route forestière Hurtault
F-77300 Fontainebleau
cegielski@u-pec.fr

# Préface

Vous avez reçu une initiation à la programmation, illustrée par le langage C. Dans *programmation structurée*, nous avons vu ce qui est indispensable pour programmer ; dans *structures de données*, nous avons vu des outils, non théoriquement indispensables, facilitant la tâche du programmeur, en particulier la notion de *sous-programme*. Pour de petits programmes la gestion de ces sous-programmes est facile, mais ceci n'est plus le cas lorsque leur nombre croît ; la **programmation orientée objet** (**poo** en abrégé) est une façon de mieux gérer les sous-programmes.

Un langage de programmation permettant de la mettre en place s'appelle un **langage orienté objet** (**loo** en abrégé), un langage tel que PASCAL ou C étant alors dit **langage impératif** (c'est une suite d'instructions intimant l'ordre de ce ce qu'il faut faire). Un langage orienté objet peut être une extension d'un langage impératif, ce dernier constituant un **noyau** ; c'est le cas du **langage C++** (le nom indiquant une incrémentation du langage C). On parle alors de **langage hybride**. Une telle façon de faire n'empêche pas de programmer sans respecter la philosophie de la programmation orientée objet, ce qui peut être gênant dans les mains d'un programmeur moyen ; il existe donc des langages de programmation objet **purs** qui forcent le programmeur à respecter cette philosophie, tel que **SMALLTALK**.

## 0.1 Philosophie de la programmation orientée objet

## 0.2 Définition du langage C++

Le langage C++ a été défini par Bjarne STROUSTRUP (des *Bell Laboratories*) dans un livre paru en octobre 1985. C'est un descendant des langages SIMULA 67 et C, une première version étant le langage C **avec classes**.

STOUSTRUP, Bjarne, **The C++ Programming Language**, Addison-Wesley, 1986 ; tr. fr.

ELLIS, Margaret A. & STOUSTRUP, Bjarne, **The Annotated C++ Reference Manual**, Addison-Wesley, 1990.

STROUSTRUP, Bjarne, **The C++ Programming Language (second edition)**, Addison-Wesley, 1991 ; tr. fr.

## 0.3 Historique

STROUSTRUP, Bjarne, *A History of C++ : 1979-1991*, **ACM SIGPLAN Notices**, vol.28, 1993, pp. 271-297.

# Table des matières

# Chapitre 1

# Mise en route du langage C++

Les compilateurs de référence pour le langage C++ sont le compilateur **gnu** sous le système d'exploitation UNIX et, pour compatibles PC avec MS-DOS ou Microsoft Windows, le **Turbo-C++ de Borland** (devenu **Borland-C++**) et le **Visual-C++** de Microsoft, ces derniers étant surtout intéressant pour la programmation système Windows.

## 1.1 Le compilateur gnu

## 1.2 Turbo-C++

Devant le succès de son compilateur *Turbo-Pascal* du langage PASCAL, la société BORLAND a conçu un compilateur pour le langage C destiné aux compatibles PC munis du système d'exploitation MS-DOS, tout naturellement appelé **Turbo C**. À l'arrivée de l'engouement pour les langages orientés objet, elle a conçu un compilateur pour le langage C++, appelé **Turbo-C++**, permettant également la compilation des programmes en langage C ANSI. Devant le développement de l'interface graphique WINDOWS, une version pour cette interface a été conçue, à la fois pour fonctionner à partir de cette interface mais aussi pour la programmation système de celle-ci. La version pour MS-DOS ne fut plus ensuite commercialisée, sinon sous la forme **Borland-C++** qui permet de choisir si l'exécutable sera lancé depuis DOS ou depuis Windows.

Nous utiliserons ici la version **Borland-C++** qui permet de programmer sous DOS, Windows 3.1 et Windows 95/98/NT.

### 1.2.1 Installation

L'installation à partir du CD-ROM se fait de façon automatique en exécutant le programme **d :install**.

### 1.2.2 Démarrage

Sous Windows, il suffit de choisir Borland-C++ dans le menu des programmes. On se trouve alors dans un environnement intégré.

Exercice 1.- Manipuler et décrire cet environnement intégré.

# Chapitre 2

# Le noyau C du langage C++

Nous avons déjà dit que le langage C++ est un langage orienté objet qui est une extension du langage C. Nous supposons acquis le cours sur l'initiation au langage C. Le noyau C du langage C++ ajoute quelques spécificités supplémentaires au langage C et, d'autre part, un certain nombre d'incompatibilités existent entre les langages C ANSI et C++. Nous allons voir dans ce chapitre ce qu'il est convenu d'appeler le **noyau** C du langage C++ ou, plus exactement, les différences existant entre le langage C et ce noyau.

## 2.1 Premières variations

### 2.1.1 Les nouvelles entrées-sorties

Introduction.- Les instructions d'entrées-sorties habituelles du C sont disponibles, mais les nouvelles entrées-sorties `cin` et `cout` évitent le recours aux formats.

Fichier en-tête concerné.- Ces instructions exigent un nouveau fichier en-tête, appelé `iostream.h` (pour *flot d'entrée-sortie*).

Syntaxe.-

- $1^o$) La syntaxe d'une sortie est :

```
cout << expr1 << ...<< exprn ;
```

où `expr1`, ..., `exprn` sont des expressions de type caractère, entier, réel, chaîne de caractères ou pointeur (on obtient l'adresse dans ce dernier cas).

- $2^o$) La syntaxe d'une entrée est :

```
cin >> var ;
```

où `var` est une variable de type caractère, entier, réel ou chaîne de caractères.

On peut en fait, comme d'habitude, entrer plusieurs valeurs à la fois mais nous le déconseillons toujours autant.

Exemple.- Écrivons un programme qui commence par dire *Bonjour*, demande un entier et affiche le double de cet entier :

```
/* bonjour.cpp */
#include <iostream.h>
void main(void)
     {
     int n ;
     cout << "Bonjour\nEntrer un entier : " ;
     cin >> n ;
     cout << "Le double de "<< n << " est " << 2*n << '\n' ;
     }
```

### 2.1.2 Les commentaires de fin de ligne

Introduction.- La façon d'introduire les commentaires en langage C, dits maintenant **commentaires multi-lignes**, avec `/*` et `*/`, continue à être permise. Mais on peut également utiliser deux signes de division `//`. Dans ce cas, tout ce qui suit sur la ligne est alors considéré comme un commentaire. On parle de **commentaire de fin de ligne**.

Exemple.- Introduisons quelques commentaires à l'exemple précédent :

```
// comment.cpp
#include <iostream.h> // pour utiliser cin et cout
void main(void)
     {
     int n ; // n est un entier
     cout << "Bonjour\nEntrer un entier : " ;
     cin >> n ; // saisie de n
     cout << "Le double de " << n << " est " << 2*n << '\n' ;
     }
```

Commentaire.- On utilise surtout `//` pour les commentaires n'occupant qu'une ligne et le couple `/*` et `*/` pour les commentaires occupant plusieurs lignes.

### 2.1.3 Emplacement de la déclaration des variables

Introduction.- En langage C, les variables doivent être déclarées en début de bloc. En C++ une variable peut être déclarée à n'importe quel endroit mais avant son utilisation. Sa portée correspond à la partie de bloc dans laquelle elle a été déclarée, depuis la déclaration jusqu'à la fin du bloc.

Exemple.- Écrivons un programme qui demande le côté d'un carré et affiche l'aire du carreau correspondant. Il est inutile de déclarer dès l'abord les variables pour le côté et l'aire, il suffit de les déclarer au moment où on en a vraiment besoin. Cette façon de faire rend les gros programmes plus clairs.

```
// variable.cpp
#include <iostream.h>
void main(void)
     {
     float cote;
     cout << "Entrer la largeur : ";
     cin >> cote;
     float aire;
     aire = cote*cote;
     cout << "L\'aire est " << aire << '\n';
     }
```

Remarque.- La déclaration tardive d'une variable permet de l'initialiser avec une expression dont la valeur n'était pas connue au début du corps de la fonction.

### 2.1.4 Les constantes

Introduction.- En langage C version K-R les constantes sont définies par le préprocesseur grâce à la commande `define`.

Nous avons déjà vu que l'on peut, en langage C version ANSI, utiliser également la déclaration `const`. Ceci est en fait une amélioration due au langage C++ qui a migrée vers la nouvelle version du langage C.

Exemple.- Écrivons un programme qui demande le rayon d'un cercle et affiche l'aire du disque correspondant.

```
// const.cpp
#include <iostream.h>
void main(void)
     {
     const float PI = 3.14159;
     float rayon;
     cout << "Entrer le rayon du cercle : ";
     cin >> rayon;
     float aire;
     aire = PI*rayon*rayon;
     cout << "L\'aire du disque est " << aire << '\n';
     }
```

### 2.1.5 Prototype de fonction

Introduction.- La notion de prototype de fonction est un apport du langage C++ qui, comme la notion de constante, a migrée vers le langage C version ANSI. Elle permet de déclarer une fonction à l'intérieur d'une autre fonction (mais pas de la définir, un programme étant toujours une suite de définitions de fonctions).

Exemple.- Voici un exemple de déclaration locale :

```
// local.cpp
#include <iostream.h>
#include <math.h>
```

```
void main(void)
     {
     double f(double x) ;
     float x, y ;
     cout << "x = " ;
     cin >> x ;
     while (x != 1000)
           {
           y = f(x) ;
           cout << "f(" << x << ") = " << y << "\n" ;
           cout << "x = " ;
           cin >> x ;
           }
     }

double f(double x)
     {
     return((sin(x) + log(x))/(exp(x) + 2)) ;
     }
```

## 2.2 Transmission par référence

Introduction.- Nous avons vu, lors de l'initiation à la programmation, la différence entre passage des paramètres par valeur et par référence. Seul le passage par valeur est implémenté en langage C, le passage par référence étant émulé grâce au passage par adresse avec l'utilisation des pointeurs.

Le passage par référence est par contre implémenté en langage C++.

Syntaxe.- Si on veut qu'une variable `ident` soit transmise par référence, on écrit `& ident` lors de la définition de la fonction, en faisant précéder la variable par une esperluette.

Un exemple.- On veut un programme qui demande deux entiers, les échange puis affiche le résultat.

Un mauvais programme.- Le programme suivant utilise le passage par valeur et donc ne répond pas à notre attente :

```
// swap_1.cpp
#include <iostream.h>
void echange (int a, int b)
     {
     int c ;
     c = a ;
     a = b ;
     b = c ;
     }
void main(void)
     {
     int m, n ;
     cout << "Entrer deux entiers : m = " ;
     cin >> m ;
```

```
     cout << "et n = " ;
     cin >> n ;
     echange(m,n) ;
     cout << "Ces deux entiers sont " << m << " et " << n << '\n' ;
     }
```

Un programme utilisant les pointeurs.- Nous avons vu comment l'utilisation des pointeurs permet de simuler le passage par référence en langage C :

```
// swap_2.cpp
#include <iostream.h>
void echange (int *a, int *b)
     {
     int c ;
     c = *a ;
     *a = *b ;
     *b = c ;
     }
void main(void)
     {
     int m,n ;
     cout << "Entrer deux entiers : m = " ;
     cin >> m ;
     cout << "et n = " ;
     cin >> n ;
     echange(&m,&n) ;
     cout << "Ces deux entiers sont " << m << " et " << n << '\n' ;
     }
```

Utilisation explicite du passage par référence.- Le langage C++ permet le programme suivant utilisant le passage par référence :

```
// swap_3.cpp
#include <iostream.h>
void echange (int & a, int & b)
     {
     int c ;
     c = a ;
     a = b ;
     b = c ;
     }
void main(void)
     {
     int m,n ;
     cout << "Entrer deux entiers : m = " ;
     cin >> m ;
     cout << "et n = " ;
     cin >> n ;
     echange(m,n) ;
```

```
     cout << "Ces deux entiers sont " << m << " et " << n << '\n';
     }
```

## 2.3 Les opérateurs d'allocation dynamique de mémoire

En langage C, l'allocation dynamique de la mémoire s'effectue avec les fonctions `malloc()` et `free()` de la bibliothèque standard. La syntaxe de la première de ces fonctions est inutilement compliquée. Le langage utilise les opérateurs `new` et `delete`, dès son noyau, à la syntaxe plus simple.

### 2.3.1 Un exemple

Le problème.- On veut disposer en mémoire centrale d'un répertoire téléphonique. Chaque composant de ce répertoire comportera un nom et un numéro de téléphone.

On utilise un tableau dont les éléments sont des pointeurs sur des structures. On ne fera pointer de façon effective que pour les emplacements nécessaires. Écrivons un programme permettant d'initialiser ce répertoire (en s'arrêtant sur un nom commençant par '#'), puis d'afficher les données initialisées.

Un programme C.- Un programme, écrit à la façon du langage C, est le suivant :

```
// malloc.cpp
#include <iostream.h>
#include <stdlib.h>
void main(void)
     {
     struct donnee
            {
            char nom[30];
            char tel[20];
            };
     struct donnee *repertoire[500], individu;
     int i, j;
// initialisation
     i = -1;
     do
            {
            cout << "\nNom : ";
            cin >> individu.nom;
            if (individu.nom[0] != '#')
                {
                cout << "Telephone : ";
                cin >> individu.tel;
                I++;
                repertoire[i] = (donnee *) malloc(sizeof(individu));
                *(repertoire[i]) = individu;
                }
            }
     while ((individu.nom[0] != '#') && (i < 500));
```

```
// affichage
     for (j = 0; j <= i; j++)
            cout << '\n' << repertoire[j]->nom
                << " : " << repertoire[j]->tel;
     cout << '\n';
// liberation des pointeurs
     for (j = 0; j <= i; j++)
            free(repertoire[j]);
     }
```

Un programme C++.- Un programme utilisant `new` et `delete` est le suivant :

```
// new.cpp
#include <iostream.h>
void main(void)
     {
     struct donnee
            {
            char nom[30];
            char tel[20];
            };
     struct donnee *repertoire[500], individu;
     int i, j;
// initialisation
     i = -1;
     do
            {
            cout << "\nNom : ";
            cin >> individu.nom;
            if (individu.nom[0]!= '#')
                  {
                  cout << "Telephone : ";
                  cin >> individu.tel;
                  i++;
                  repertoire[i] = new donnee;
                  *(repertoire[i]) = individu;
                  }
            }
     while ((individu.nom[0]!= '#') && (i < 500));
// affichage
     for (j = 0; j <= i; j++)
            cout << '\n' << repertoire[j]->nom
                  << " : " << repertoire[j]->tel;
     cout << '\n';
// liberation des pointeurs
     for (j = 0; j <= i; j++)
            delete repertoire[j];
     }
```

### 2.3.2 Mise en place de l'opérateur d'allocation

Syntaxe.- L'opérateur `new` est un opérateur unaire qui peut être utilisé de l'une des deux façons suivantes :

```
type *ptr ;
ptr = new type ;
ptr = new type [entier] ;
```

où `type` est un type, `ptr` un identificateur (et donc un pointeur de ce type) et `entier` une expression entière (de valeur positive).

Sémantique.- Dans le premier cas cet opérateur alloue la place nécessaire pour un élément de type `type` et fournit comme résultat :

- l'adresse de cet emplacement (valeur d'un pointeur de type `type*`) lorsque l'allocation a réussie,
- le pointeur nul (`NULL`) dans le cas contraire.

Dans le deuxième cas l'opérateur alloue la place nécessaire à `entiers` éléments consécutifs de type `type` et fournit comme résultat :

- l'adresse du premier emplacement (valeur d'un pointeur de type `type*`) lorsque l'allocation a réussie,
- le pointeur nul dans le cas contraire.

On utilise évidemment le deuxième cas pour réserver de la place pour un tableau.

### 2.3.3 Mise en place de l'opérateur de libération

Syntaxe.- On a soit :

```
delete ptr ;
```

soit :

```
delete [ ] ptr ;
```

où `ptr` est une variable pointeur.

Sémantique.- L'emplacement mémoire réservé par la variable pointeur, s'il existe (autrement dit si la variable a été initialisée par `new`, et uniquement dans ce cas), est libéré.

La différence entre les deux cas n'intervient que lorsque le pointeur `ptr` a été initialisé par `ptr = new type [entier]`. L'instruction `delete ptr ;` libère la place mémoire pour un élément de type `type` alors que `delete [] ptr ;` libère la place mémoire pour le tableau dans son entier, c'est-à-dire pour `entier` éléments de type `type`.

## 2.4 Les fonctions en ligne

Les fonctions en lignes correspondent à une forme améliorée de macros qui sera souvent utilisée pour définir les fonctions membres d'une classe.

### 2.4.1 Rappels sur fonctions et macros en C

Introduction.- Il existe deux façons de définir des applications en langage C : grâce à la notion de fonction (la plus utilisée) mais aussi grâce à la notion de macro. Voyons cela à propos d'un exemple.

Un problème.- Écrivons un programme qui demande un réel et affiche son carré.

Pour des raisons de modularité (nous pourrions changer plus tard pour le cube ou autre chose) nous ferons effectuer le calcul à part.

Utilisation d'une macro.- Ceci conduit au programme suivant :

```
// macro.cpp
#include <iostream.h>
#define SQ(X) X*X
void main(void)
     {
     float X, Y;
     cout << "Entrer un reel : ";
     cin >> X;
     Y = SQ(X);
     cout << "Son carre est " << Y << '\n';
     }
```

Utilisation d'une fonction.- Ceci conduit au programme suivant :

```
// fonction.cpp
#include <iostream.h>
float SQ(float X)
      {
      return(X*X);
      };
void main(void)
      {
      float X, Y;
      cout << "Entrer un reel : ";
      cin >> X;
      Y = SQ(X);
      cout << "Son carre est " << Y << '\n';
      }
```

Différences entre les deux méthodes.-

- $1^o$) Les macros ne peuvent être écrites que pour des fonctions très faciles à définir.

- $2^o$) Les instructions correspondant à une macro sont incorporées au programme à chaque appel de celle-ci. En effet le préprocesseur se contente de remplacer une expression par une autre.

- $3^o$) Une fonction est compilée une fois pour toutes, une liaison entre les deux parties compilées est mise en place à chaque appel de fonction. Une fonction permet de gagner de l'espace mémoire en contrepartie d'une perte de temps d'exécution.

- $4^o$) Une macro est moins sûre qu'une fonction. En effet si, dans l'exemple précédent, on utilise :

```
SQ(a + b)
```

on obtiendra :

```
a + b*a + b
```

et non ce qui est voulu, à savoir :

`(a + b)*(a + b)`.

Il suffit dans ce cas de définir la macro de la façon suivante :

```
#define SQ(X) (X)*(X)
```

pour corriger ce phénomène. Mais il existe d'autres types d'**effets de bord**, comme lorsqu'on demande :

```
SQ(a++)
```

qui incrémentera deux fois `a`, ce qui n'est pas ce que l'on veut.

### 2.4.2 Mise en place des fonctions en ligne

Introduction.- Les inconvénients des macros disparaissent avec les fonctions en ligne. On les définit comme des fonctions mais la définition est précédée du mot réservé **inline**. Elles sont compilées comme des macros dans la mesure du possible ; la différence avec une macro est que c'est le compilateur, et non plus le préprocesseur, qui fait le travail.

Un exemple.- Si on reprend le problème précédent, on obtient :

```
// enligne.cpp
#include <iostream.h>
inline float SQ(float X)
       {
       return(X*X);
       };
void main(void)
       {
       float X, Y;
       cout << "Entrer un reel : ";
       cin >> X;
       Y = SQ(X);
       cout << "Son carre est " << Y << '\n';
       }
```

Remarque.- Une fonction en ligne doit être définie dans le même fichier source que celui où on l'utilise. Elle ne peut pas être compilée séparément.

## Exercices

Exercice.- Reprendre l'exemple 2.3.1 en triant le répertoire téléphonique dans l'ordre alphabétique des noms avant de l'afficher.