PROGRAMMATION C++

Chargé du cours :

M. LATE Lawson

Analyste programmeur

Tel : 97 07 72 84

Assisté par :

M. YOTTO Alfred

Analyste programmeur

Economiste

PLAN DU COURS

Chapitre 0 : Introduction et historique du C++

Chapitre 1 : Notions de bases

A. Généralité
B. Les variables en C++
C. Les structures de contrôles
D. Les tableaux
E. Les fonctions et les procédures
F. Les pointeurs et les références
G. Exercices

Chapitre 2 : La programmation Orientée Objet (P.O.O) en C++

A. Les classes
B. L'Héritage
C. Le polymorphisme
D. Créer des objets avec du C++
E. Exercices

Chapitre 3 : Programmation graphique en C++ avec Qt Creator

A. Travaux pratique

CHAPITRE 0 : INTRODUCTION ET HISTORIQUE DU C++

A) INTRODUCTION

Le C++ est l'un des langages de programmation les plus utilisés actuellement. Il est à la fois facile à utiliser et très efficace. Il souffre cependant de la réputation d'être compliqué et illisible. Cette réputation est en partie justifiée. La complexité du langage est inévitable lorsqu'on cherche à avoir beaucoup de fonctionnalités. En revanche, en ce qui concerne la lisibilité des programmes, tout dépend de la bonne volonté du programmeur.

Les caractéristiques du C++ en font un langage idéal pour certains types de projets. Il est incontournable dans la réalisation des grands programmes. Les optimisations des compilateurs actuels en font également un langage de prédilection pour ceux qui recherchent les performances. Enfin, ce langage est, avec le C, idéal pour ceux qui doivent assurer la portabilité de leurs programmes au niveau des fichiers sources (pas des exécutables).

Les principaux avantages du C++ sont les suivants :

- ✓ grand nombre de fonctionnalités ;
- ✓ performances du C ;
- ✓ facilité d'utilisation des langages objets ;
- ✓ portabilité des fichiers sources ;
- ✓ facilité de conversion des programmes C en C++, et, en particulier, possibilité d'utiliser toutes les fonctionnalités du langage C ;
- ✓ contrôle d'erreurs accru.

On dispose donc de quasiment tout : puissance, fonctionnalité, portabilité et sûreté.

B) HISTORIQUE DU C++

La programmation orientée objet (en abrégé P.O.O.) est dorénavant universellement reconnue pour les avantages qu'elle procure. Notamment, elle améliore largement la productivité des développeurs, la robustesse, la portabilité et l'extensibilité de leurs programmes. Enfin, et surtout, elle permet de développer des composants logiciels entièrement réutilisables.

Un certain nombre de langages dits "langages orientés objet" (L.O.O.) ont été définis de toutes pièces pour appliquer les concepts de P.O.O. C'est ainsi que sont apparus dans un premier temps des langages comme Smalltalk, Simula ou Eiffel puis, plus récemment, Java. Le langage C++, quant à lui, a été conçu suivant une démarche quelque peu différente par B. Stroustrup (AT&T) ; son objectif a été, en effet, d'adjoindre au langage C un certain nombre de spécificités lui permettant d'appliquer les concepts de P.O.O. Ainsi, C++ présente-t-il sur un vrai L.O.O. l'originalité d'être fondé sur un langage répandu. Ceci laisse au programmeur toute liberté d'adopter un style plus ou moins orienté objet, en se situant entre les deux extrêmes que constituent la poursuite d'une programmation classique d'une part, une pure P.O.O. d'autre part. Si une telle liberté présente le risque de céder, dans un premier temps, à la facilité en mélangeant les genres (la P.O.O. ne renie pas la programmation classique - elle l'enrichit), elle permet également une transition en douceur vers la P.O.O pure, avec tout le bénéfice qu'on peut en escompter à terme.

De sa conception jusqu'à sa normalisation, le langage C++ a quelque peu évolué. Initialement, un certain nombre de publications de AT&T ont servi de référence du langage.

CHAPITRE 1 : NOTIONS DE BASES

A. Généralité

- ✓ **Les langages de programmation**

Le C++ est un **langage de programmation** : il sert donc à écrire des applications informatiques. Il s'agit d'ailleurs d'un des langages de programmation les plus utilisés aujourd'hui. Chaque programme en C++ doit être écrit en respectant des règles d'écriture très strictes que nous étudierons petit à petit.

- ✓ **Un langage compilé**

Le C++ est un langage compilé : pour écrire un tel programme, il faut commencer par écrire un ou plusieurs fichiers source. Ensuite, il faut compiler ces fichiers source grâce à un programme appelé compilateur afin d'obtenir un programme exécutable**. Cette phase s'appelle la compilation.** Les fichiers source sont des fichiers texte lisibles dont le nom se termine en général par .c, .cpp ou .h. Les fichiers exécutables portent en général l'extension .exe sous windows et ne portent pas d'extension sous Linux.

- ✓ **Les compilateurs**

Il existe de très nombreux compilateurs : on peut citer par exemple Visual C++ (de microsoft), C++ Builder (de Borland), ou encore gcc qui est un excellent compilateur libre.

- ✓ **Les environnements de développement intégrés (EDI)**

On programme très souvent en utilisant un environnement de développement intégré : il s'agit d'un ensemble complet d'outils permettant d'éditer et de modifier des fichiers sources, de les compiler, de lancer l'exécutable, de

"déboguer" le programme, etc... Visual C++ (version express disponible gratuitement), C++ Builder, Dev-cpp (disponible gratuitement et basé sur gcc), Qt Creator et Code::Blocks (lui aussi gratuit mais plus souvent mis à jour que Dev-cpp) sont des environnements de développement intégrés.

Dans le cadre de notre cours nous utiliserons Qt Creator qui est un framework puissant permet d'éditer des programme C++, Java, python, et meme des applications mobile.

- ✓ **Exemple de programme C++**

```
#include <iostream>
 int main()
{
 int iEntier1;
cout << "Saisir un entier : " << endl; // Affiche à l'écran
cin >> iEntier1; // Lit un entier
int iEntier2, iSomme;
cout << "Saisir un autre entier : " << endl;
cin >> iEntier2;
 iSomme = iEntier1 + iEntier2;
cout << "La somme de " << iEntier1 << " et de " << iEntier2 << " vaut : " <<
iSomme << endl; // endl = saut de ligne
 return 0;
}
```

- ✓ **#include <iostream>**

C'est ce qu'on appelle une *directive de préprocesseur*. Son rôle est de "charger" des fonctionnalités du C++ pour que nous puissions effectuer certaines actions.

- #include <nom_fichier> Inclut le fichier nom_fichier en le cherchant d'abord dans les chemins configurés, puis dans le même répertoire que le fichier source,

- #include "nom_fichier" Inclut le fichier nom_fichier en le cherchant d'abord dans le même répertoire que le fichier source, puis dans les chemins configurés.

✓ **using namespace std;**

Cette ligne est un peu plus difficile à comprendre : en effet, on indique par cette ligne l'utilisation de l'espace de nommage std. Un espace de nommage est un ensemble de classes dont cout fait partie. Etant donné que nous voulons utiliser l'objet cout, nous indiquons que l'on utilisera, par défaut, l'espace de nommage std. Pour simplifier, retenons que, dès que l'on veut utiliser cin ou cout, on doit écrire cette directive. Il faut également remarquer que les fichiers d'en-tête standard ne sont désormais plus nommés avec une extension .h (comme iostream.h). Si ces fichiers d'en-tête sont inclus sans être suivi de la commande **using namespace std;**, cela ne fonctionnera pas correctement. Dans certaines versions de g++ , si, lors de la compilation, vous spécifiez un fichier d'en-tête standard avec une extension .h (comme iostream.h), le compilateur utilisera le fichier "backward" compatible et vous signifiera un avertissement.

✓ **La fonction main()**

Notre programme contient une fonction appelée main : c'est à cet endroit que va commencer l'exécution du programme : exécuter un programme en C++, c'est exécuter la fonction main de ce programme. Tout programme en C++ doit donc comporter une fonction main.

Remarque : Entrées/sorties ou cin/cout sont fournies à travers la librairie *iostream*

B. Les variables en C++

De façon claire et simple **une variable** est une partie de la mémoire que l'ordinateur nous prête pour y mettre des valeurs.

Pour déclarer une variable il est indispensable de suivre un certain nombre de principes :

1. Les noms de variables sont constitués de lettres, de chiffres et du tiret-bas _ uniquement.
2. Le premier caractère doit être une lettre (majuscule ou minuscule).
3. On ne peut pas utiliser d'accents.
4. On ne peut pas utiliser d'espaces dans le nom.

Le mieux est encore de vous donner quelques exemples. Les noms ageZero, nom_du_zero ou encore NOMBRE_ZEROS sont tous des noms valides. AgeZéro, _nomzero ne le sont par contre pas.

C. LES STRUCTURES DE CONTROLES

- ✓ **La structure conditionnelle if**

La structure conditionnelle if permet de réaliser un test et d'exécuter une instruction ou non selon le résultat de ce test. Sa syntaxe est la suivante :

```
if (test)
{
opération;
}
```

où test est une expression dont la valeur est booléenne ou entière. Toute valeur non nulle est considérée comme vraie. Si le test est vrai, opération est exécuté. Ce peut être une instruction ou un bloc d'instructions. Une variante permet de spécifier l'action à exécuter en cas de test faux :

```
if (test)
{
opération1;
}else
{
 opération2;
}
```

- ✓ **La boucle for**

La structure de contrôle for est sans doute l'une des plus importantes. Elle permet de réaliser toutes sortes de boucles et, en particulier, les boucles itérant sur les valeurs d'une variable de contrôle. Sa syntaxe est la suivante :

```
for (initialisation ; test ; itération)
{
 opération;
}
```

où initialisation est une instruction (ou un bloc d'instructions) exécutée avant le premier parcours de la boucle du for. test est une expression dont la valeur déterminera la fin de la boucle.
itération est l'opération à effectuer en fin de boucle, et opération constitue le traitement de la boucle. Chacune de ces parties est facultative.

- ✓ **Le while**

Le while permet d'exécuter des instructions en boucle tant qu'une condition est vraie. Sa syntaxe est la suivante :

```
while (test)
{
opération;
}
```

où opération est effectuée tant que test est vérifié. Comme pour le if, les parenthèses autour du test sont nécessaires.

✓ Le do

La structure de contrôle do permet, tout comme le while, de réaliser des boucles en attente d'une condition. Cependant, contrairement à celui-ci, le do effectue le test sur la condition après l'exécution des instructions. Cela signifie que les instructions sont toujours exécutées au moins une fois, que le test soit vérifié ou non. Sa syntaxe est la suivante :

```
do
{
opération;
}while (test);
```

opération est effectuée jusqu'à ce que test ne soit plus vérifié.

✓ Le branchement conditionnel

Dans le cas où plusieurs instructions différentes doivent être exécutées selon la valeur d'une variable de type intégral, l'écriture de if successifs peut être relativement lourde. Le C/C++ fournit donc la structure de contrôle switch, qui permet de réaliser un branchement conditionnel. Sa syntaxe est la suivante :

```
switch (valeur)
{
```

case cas1:

[instruction;

[break;]

]

case cas2:

[instruction;

[break;]

]...

case casN:

[instruction;

[break;]

]

[default:

[instruction;

[break;]

]

]

}

valeur est évalué en premier. Son type doit être entier. Selon le résultat de l'évaluation, l'exécution

du programme se poursuit au cas de même valeur. Si aucun des cas ne correspond et si default est présent, l'exécution se poursuit après default. Si en revanche default n'est pas présent, on sort du switch.

Les instructions qui suivent le case approprié ou default sont exécutées. Puis, les instructions du cas suivant sont également exécutées (on ne sort donc pas du switch). Pour forcer la sortie du switch, on doit utiliser le mot clé break.

D. Les tableaux

1. Les tableaux statiques

Un tableau est une collection indicée de variables de même type.

Forme de la d´eclaration :

<*type*> <*nom*> [<*taille*>];

Où : <*type*> est le type des éléments du tableau,

<*nom*> est le nom du tableau,

<*taille*> est une constante entière égale au nombre d'éléments du tableau.

Si t est un tableau et i une expression entière, on note **t[i]** l´élément d'indice i du tableau. Les éléments d'un tableau sont indicés de 0 à taille – 1.

Il est possible de déclarer des tableaux a deux indices (ou plus). Par exemple, en écrivant :

Int M[2][3];

On déclare un tableau d'entiers M a deux indices, le premier indice variant entre 0 et 1, le second entre 0 et 2. On peut voir M comme une matrice d'entiers à 2 lignes et 3 colonnes. Les éléments de M se notent **M[i][j].**

2. Les tableaux dynamiques

Le C++ a introduit la **classe vector**, qui n'a pas ces inconvénients.

Un tableau de 3 éléments de type int peut se déclarer ainsi:

vector<int> tab(3);

On peut accéder aux éléments de **tab** de la même façon qu'on accéderait aux éléments d'un tableau statique:

tab[0] = 4;

Il faut ajouter la ligne #include <vector> au début du programme pour pouvoir utiliser la classe vector.

Attention à ne pas l'oublier. Si on fait:

vector<int> tab;

Le tableau n'a pas d'éléments, et essayer d'accéder à un élément, comme par exemple:

tab[0] = 0;

générera sans doute un *Segmentation fault*.

On peut aussi faire:

vector<int> tab(3, 1);

pour initialiser tous les éléments du tableau à 1 au moment de la déclaration. et:

vector<int> tab2(tab);

pour initialiser les éléments de tab2 aux mêmes valeurs que tab.

Il suffit de changer int pour changer le type des éléments:

vector<double> tab3(5);

vector<bool> tab4(128);

vector<int> tab(3);

On peut maintenant connaître le nombre d'éléments de tab en utilisant la fonction size():

int taille = tab.size();

E. Les fonctions et procédures

1. Les fonctions

En C++, la partie exécutable d'un programme (c'est-`a-dire celle qui comporte des instructions) n'est composée que de fonctions. Chacune de ces fonctions est destinée à effectuer une tache précise et renvoie généralement une valeur, résultat d'un calcul.

Une *fonction* est caractérisée par :

1) son *nom*,
2) le *type* de valeur qu'elle renvoie,
3) l'information qu'elle reçoit pour faire son travail (*paramètres*),
4) l'instruction-bloc qui effectue le travail (*corps* de la fonction).

Les trois premiers éléments sont décrits dans la déclaration de la fonction.
L'élément n◦4 figure dans la définition de la fonction.

Toute fonction doit être définie avant d'être utilisée.

La définition d'une fonction se fait toujours au niveau principal. On ne peut donc pas imbriquer les fonctions les unes dans les autres, comme on le fait en Pascal.

Déclaration d'une fonction

Elle se fait grâce à un ***prototype*** de la forme suivante :

<type> <nom>(<liste de paramètres formels>);

Ou *<type>* est le type du résultat, *<nom>* est le nom de la fonction et *<liste de paramètres formels>* est composé de zéro, une ou plusieurs déclarations de variables, séparées par des virgules.

Exemples :

double Moyenne(double x, double y);

char LireCaractere();

void AfficherValeurs(int nombre, double valeur);

Remarque : la dernière fonction n'est pas destinée à renvoyer une valeur ; c'est pourquoi le type du résultat est void (une telle fonction est parfois appelée *procédure*).

Définition d'une fonction

Elle est de la forme suivante :

<type> <nom>(<liste de paramètres formels>)

<instruction-bloc>

Donnons par exemple les définitions des trois fonctions déclarées ci-dessus :

```
double Moyenne(double x, double y)
{
return (x + y) / 2.0;
}

char LireCaractere()
{
char c;
cin >> c;
return c;
}
```

```
void AfficherValeurs(int nombre, double valeur)
{
cout << ’\t’ << nombre << ’\t’ << valeur << ’\n’;
}
```

A retenir :

L’instruction return <*expression*>; *interrompt l’exécution de la fonction. La valeur de l’*<*expression*> *est la valeur que renvoie la fonction.*

Utilisation d’une fonction

Elle se fait grâce à l’*appel* de la fonction. Cet appel est une expression de la forme :

***<nom>*(<*liste d’expressions*>).**

Mécanisme de l’appel :

– chaque expression de la <*liste d’expressions*> est évaluée,
– les valeurs ainsi obtenues sont transmises dans l’ordre aux paramètres formels,
– le corps de la fonction est ensuite exécuté,
– la valeur renvoyée par la fonction donne le résultat de l’appel.

Si la fonction ne renvoie pas de valeur, le résultat de l’appel est de type void.

Voici un bout de programme avec appel des trois fonctions précédentes :

```
double u, v;
cout << "\nEntrez les valeurs de u et v :";
cin >> u >> v;
double  m = Moyenne(u, v);
```

```
cout << "\nVoulez-vous afficher la moyenne ? ";
char  reponse = LireCaractere();
if (reponse == 'o')
AfficherValeurs(2, m);
```

Arguments par défaut

On peut, lors de la déclaration d'une fonction, choisir pour les paramètres des valeurs par défaut (sous forme de déclarations-initialisations figurant à la fin de la liste des paramètres formels).

Par exemple :

```
void AfficherValeurs(int nombre, double valeur = 0.0);
```

Les deux appels suivants sont alors corrects :

```
AfficherValeurs(n, x);
AfficherValeurs(n); //équivaut à : AfficherValeurs(n, 0.0);
```

Voici un petit programme complet écrit en C++ :

```
// ------------------- fichier gazole.cpp ---------------------
#include <iostream.h>  // pour les entrées-sorties
const double prix du litre = 0.89;   // déclaration du prix du litre de gazole (en euros) comme constante (hum!)
int reserve = 10000;  // déclaration  de la réserve de la pompe, en litres
double prix(int nb)    // définition d'une fonction appel´ee "prix" :
      // cette fonction renvoie le prix de nb litres de gazole
{
return nb * prix du litre;
}
```

```
int delivre(int nb) //  définition d’une fonction appelée  délivre: cette fonction renvoie 0
// si la réserve est insuffisante, 1 sinon et délivre alors nb litres
{
if (nb > reserve) // instruction if
return 0;
reserve -= nb;
return 1;
}
int main() // définition de la fonction principale
{
int possible;
do // instruction do : voir paragraphe 2.7.2
{
int quantite;
cout << "Bonjour. Combien voulez-vous de litres de gazole ? ";
cin >> quantite;
possible = delivre(quantite);
if (possible)
cout << "Cela fait " << prix(quantite) << " euros.\n";
}
while (possible);
cout << "Plus assez de carburant.\n";
return 0;  // la fonction main renvoie traditionnellement un entier
}
```

Comme le montre le programme précèdent, il est tout-`a-fait possible de n’écrire qu’un seul fichier source .cpp contenant toutes les d´déclarations et instructions. Cependant, pour un gros programme, il est recommandé d’écrire plusieurs

fichiers-source, chacun étant spécialisé dans une catégorie d'actions précise. **Cette technique sera d'ailleurs de rigueur quand nous ferons de la programmation-objet.**

A titre d'exemple, voici le même programme, mais composé de trois fichiers-sources distincts. On remarquera **que le fichier d'en-tête** est inclus dans les deux autres fichiers.

```
// ------------------- fichier pompe.h ---------------------
const double prix du litre = 0.89; // déclaration du prix du litre de gazole (en euros)
double prix(int nb); // déclaration de la fonction prix
int delivre(int nb); // déclaration de la fonction delivre
// ------------------- fichier pompe.cpp --------------------
#include "pompe.h" // pour inclure les déclarations précédentes
int reserve = 10000; // déclaration de la réserve de la pompe, en litres
double prix(int nb) // définition de la fonction prix
{
..... // comme en (2.5.1)
}
int delivre(int nb) // définition de la fonction delivre
{
..... // comme en (2.5.1)
}
// ------------------- fichier clients.cpp ------------------
#include <iostream.h> // pour les entrées-sorties
#include "pompe.h" // pour les déclarations communes
int main() // définition de la fonction principale
{
..... // comme en vue précédemment
```

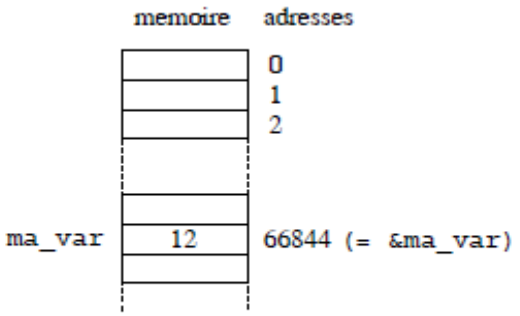
}

F. Pointeurs et Références

1. Adresses

La mémoire d'un ordinateur peut être considérée comme un empilement de cases-mémoire. Chaque case mémoire est repérée par un numéro qu'on appelle son *adresse* (en général un entier long).

Si **ma_var** est une variable d'un type quelconque, l'adresse ou est stockée la valeur de **ma_var** s'obtient grâce à l'*operateur d'adresse* & : cette adresse se note **&ma_var** :



2. Les pointeurs

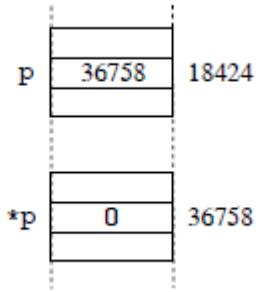
Un pointeur est une variable qui contient l'adresse d'une autre variable.
La déclaration d'un pointeur est de la forme suivante :

<type> *<nom>;

Par exemple :

int *p;

La variable p est alors un pointeur sur un int ; la variable entière dont p contient l'adresse est dite pointée par p et se note *p. Schématiquement :



- ✓ Une variable pointée s'utilise comme une variable ordinaire.

Exemple :

```
int *p, *q, n;
n = -16;
*p = 101;
*q = n + *p; // donne a *q la valeur 85
```

- ✓ Un pointeur est toujours li´e `a un type (sur lequel il pointe).

Exemple :

```
int *ip;
double *dp;
dp = ip; // illegal car ip et dp ne sont pas de même type
*dp = *ip; // correct (conversion implicite de type).
```

- ✓ Il y a dans stdlib.h la valeur pointeur NULL (´egale `a 0) qui est compatible avec tout type de pointeur. Ainsi :

```
dp = NULL; // légal
ip = NULL; // légal
```

3. Références

Une référence est une variable qui coïncide avec une autre variable.

La déclaration d'une référence est de la forme suivante :

<type> &<nom> = <nom var>;

Par exemple :

int n = 10;

int &r = n; // r est une référence sur n

Cela signifie que r et n représentent la même variable (en particulier, elles ont la même adresse). Si on écrit par la suite : r = 20, n prend également la valeur 20.

Attention :

double somme, moyenne;

double &total = somme; // correct : total est une référence sur somme

double &valeur; // illégal : pas de variable à laquelle se référer

double &total = moyenne; // illégal : on ne peut redéfinir une référence.

4. Pointeurs et tableaux

Considérons la d´déclaration suivante :

int T[5]; // T est un tableau de 5 entiers

T est en réalité un pointeur constant sur un int et contient l'adresse du premier élément du tableau.

Donc T et &T[0] sont synonymes. On a le droit d'écrire par exemple :

int *p = T;

p[2]; // = T[2]

A noter :

int U[5];

U = T; // illégal : U ne peut être modifie (pointeur constant) :

Donc :

- il n'y a pas de recopie du contenu du tableau (d'où gain de temps),
- la fonction peut modifier le contenu du tableau.

Par exemple, en définissant :

```
void annule(int V[])   // pour un parametre tableau : inutile d'indiquer la taille
{
for (int i = 0; i < 5; i++)
V[i] = 0;
}
```

On peut mettre à zéro un tableau de cinq entiers T en écrivant annule(T).

CHAPITRE 2 : LA PROGRAMMATION ORIENTEE OBJET

✓ La programmation structurée

La programmation structurée a manifestement fait progresser la qualité de la production des logiciels. Mais, avec le recul, il faut bien reconnaître que ses propres fondements lui imposaient des limitations "naturelles". En effet, la programmation structurée reposait sur ce que l'on nomme souvent "**l'équation de Wirth**", à savoir :

Algorithmes + Structures de données = Programmes

Bien sûr, elle a permis de structurer les programmes, et partant, d'en améliorer **l'exactitude et la robustesse**. On avait espéré qu'elle permettrait également d'en améliorer **l'extensibilité et la réutilisabilité**. Or, en pratique, on s'est aperçu que l'adaptation ou la réutilisation d'un logiciel conduisait souvent à "casser" le module intéressant, et ceci parce qu'il était nécessaire de remettre en cause une structure de données. Précisément, ce type de difficultés émane directement de l'équation de Wirth, qui découple totalement les données des procédures agissant sur ces données.

✓ Les apports de la Programmation Orientée Objet

1. Objet

C'est là qu'intervient la Programmation Orientée Objet (en abrégé P.O.O), fondée justement sur le concept d'objet, à savoir une association des données et des procédures (qu'on appelle alors méthodes) agissant sur ces données. Par analogie avec l'équation de Wirth, on pourrait dire que l'équation de la P.O.O. est :

Méthodes + Données = Objet

2. Encapsulation

Mais cette association est plus qu'une simple juxtaposition. En effet, dans ce que l'on pourrait qualifier de P.O.O. "pure"1, on réalise ce que l'on nomme une encapsulation des données.

Cela signifie qu'il n'est pas **possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par l'intermédiaire de ses méthodes**, qui jouent ainsi le rôle d'interface obligatoire. On traduit parfois cela en disant que l'appel d'une méthode est en fait l'envoi d'un "message" à l'objet.

Le grand mérite de l'encapsulation est que, vu de l'extérieur, un objet se caractérise uniquement par les spécifications de ses méthodes, la manière dont sont réellement implantées les données étant sans importance. On décrit souvent une telle situation en disant qu'elle réalise une "abstraction des données" (ce qui exprime bien que les détails concrets d'implémentation sont cachés). A ce propos, on peut remarquer qu'en programmation structurée, une procédure pouvait également être caractérisée (de l'extérieur) par ses spécifications, mais que, faute d'encapsulation, l'abstraction des données n'était pas réalisée.

L'encapsulation des données présente un intérêt manifeste en matière de qualité de logiciel.

Elle facilite considérablement la maintenance : une modification éventuelle de la structure des données d'un objet n'a d'incidence que sur l'objet lui-même ; les utilisateurs de l'objet ne seront pas concernés par la teneur de cette modification (ce qui n'était bien sûr pas le cas avec la programmation structurée). De la même manière, l'encapsulation des données facilite grandement la réutilisation d'un objet.

Bonus :

• **l'exactitude :** aptitude d'un logiciel à fournir les résultats voulus, dans des conditions normales d'utilisation (par exemple, données correspondant aux spécifications) ;

• **la robustesse :** aptitude à bien réagir lorsque l'on s'écarte des conditions normales d'utilisation ;

• **l'extensibilité :** facilitée avec laquelle un programme pourra être adapté pour satisfaire à une évolution des spécifications ;

• **la réutilisabilité :** possibilité d'utiliser certaines parties (modules) du logiciel pour résoudre un autre problème ;

• **la portabilité :** facilitée avec laquelle on peut exploiter un même logiciel dans différentes implémentations ;

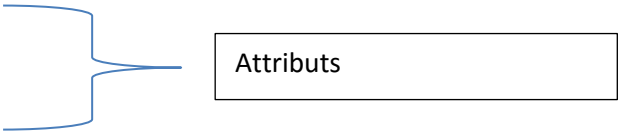
• **l'efficience :** temps d'exécution, taille mémoire...

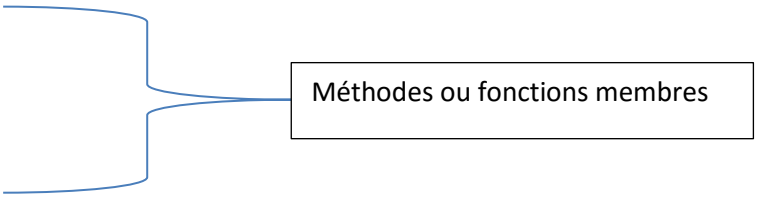
A. Les Classes

En C++ la structure est un cas particulier de la classe. Plus précisément, une classe sera une structure dans laquelle seulement certains membres et/ou fonctions membres seront "publics", c'est-à-dire accessibles "de l'extérieur", les autres membres étant dits "privés".

Exemple de classe :

```
class point
{ /* déclaration des membres privés */
private : /* déclaration des membres privés */
int x ;
int y ;
/* déclaration des membres publics */
public :
void initialise (int, int) ;
void deplace (int, int) ;
void affiche () ;
} ;
```





Déclaration des fonctions membre de la classe point

```
void point::initialise (int abs, int ord)
{
x = abs ; y = ord ;
}

void point::deplace (int dx, int dy)
{
x = x + dx ; y = y + dy ;
}

void point::affiche ()
{
cout << "Je suis en " << x << " " << y << "\n" ;
}
```

Utilisation de la classe point

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

Dans notre exemple :

- a et b sont des instances de la classe point, ou encore que ce sont des objets de type point ;

- tous les membres donnés de point sont privés, ce qui correspond à une encapsulation complète des données. Ainsi, une tentative d'utilisation directe (ici au sein de la fonction main) du membre a :
  a.x = 5
  conduirait à un diagnostic de compilation (bien entendu, cette instruction serait acceptée si nous avions fait de x un membre public).
  En général, on cherchera à respecter le principe d'encapsulation des données, quitte à prévoir des fonctions membres appropriées pour y accéder.
- toutes les fonctions membres étaient publiques. Il est tout à fait possible d'en rendre certaines privées. Dans ce cas, de telles fonctions ne seront plus accessibles de l'"extérieur" de la classe. Elles ne pourront être appelées que par d'autres fonctions membres.
- il existe un troisième mot, protected (protégé), qui s'utilise de la même manière que les deux autres ; il sert à définir un statut intermédiaire entre public et privé, lequel n'intervient que dans le cas de classes dérivées.

a) Constructeur et destructeur

- ✓ **le constructeur :** Il s'agit d'une fonction membre (définie comme les autres fonctions membres) qui sera appelée automatiquement à chaque création d'un objet. Ceci aura lieu quelle que soit la classe d'allocation de l'objet : statique, automatique ou dynamique.
- ✓ **Un objet pourra aussi posséder un destructeur**, c'est-à-dire une fonction membre appelée automatiquement au moment de la destruction de l'objet. Dans le cas des objets automatiques, la destruction de l'objet a lieu lorsque l'on quitte le bloc ou la fonction où il a été déclaré.

Par convention, le constructeur se reconnaît à ce qu'il porte le même nom que la classe. Quant au destructeur, il porte le même nom que la classe, précédé d'un tilde (~).

B. L'HERITAGE

On sait que le concept d'héritage (on parle également de classes dérivées) constitue l'un des fondements de la P.O.O. En particulier, il est à la base des possibilités de réutilisation de composants logiciels (en l'occurrence, de classes). En effet, il vous autorise à définir une nouvelle classe, dite "dérivée", à partir d'une classe existante dite "de base". La classe dérivée "héritera" des "potentialités" de la classe de base, tout en lui en ajoutant de nouvelles, et cela sans qu'il soit nécessaire de remettre en question la classe de base. Il ne sera pas utile de la recompiler, ni même de disposer du programme source correspondant (exception faite de sa déclaration).

Cette technique permet donc de développer de nouveaux outils en se fondant sur un certain acquis, ce qui justifie le terme d'héritage. Bien entendu, plusieurs classes pourront être dérivées d'une même classe de base. En outre, l'héritage n'est pas limité à un seul niveau : une classe dérivée peut devenir à son tour classe de base pour une autre classe. On voit ainsi apparaître la notion d'héritage comme outil de spécialisation croissante.

Matérialisons l'explication :

- **Déclaration d'une classe de base**

```
class point
{
 /* déclaration des membres privés */
int x ;
int y ;
```

```
/* déclaration des membres publics */
public :
void initialise (int, int) ;
void deplace (int, int) ;
void affiche () ;
} ;
```

Supposons que nous ayons besoin de définir un nouveau type classe nommé pointcol, destiné à manipuler des points colorés d'un plan. Une telle classe peut manifestement disposer des mêmes fonctionnalités que la classe point, auxquelles on pourrait adjoindre, par exemple, une méthode nommée colore, chargée de définir la couleur. Dans ces conditions, nous pouvons être tentés de définir pointcol comme une classe dérivée de point. Si nous prévoyons (pour l'instant) une fonction membre spécifique à pointcol nommée colore, et destinée à attribuer une couleur à un point coloré, voici ce que pourrait être la déclaration de pointcol (la fonction colore est ici en ligne) :

```
class pointcol : public point // pointcol dérive de point
{
short couleur ;
public :
void colore (short cl)
{ couleur = cl ; }
} ;
```

Le mot **public** signifie que les membres publics de la classe de base (point) seront des membres publics de la classe dérivée (pointcol) ; cela correspond à l'idée la plus fréquente que l'on peut avoir de l’héritage, sur le plan général de la P.O.O.

La classe **pointcol** ainsi définie, nous pouvons déclarer des objets de type pointcol de manière usuelle :

pointcol p, q ;

Chaque objet de type **pointcol** peut alors faire appel :

• aux méthodes publiques de pointcol (ici colore),

• aux méthodes publiques de la classe de base point (ici init, deplace et affiche).

Exemple :

```
#include <iostream>
#include "point.h"       // incorporation des déclarations de point
using namespace std ;
/* --- Déclaration et définition de la classe pointcol ----- */
class pointcol : public point      // pointcol dérive de point
{ short couleur ;
public :
void colore (short cl) { couleur = cl ; }
} ;
main()
{ pointcol p ;
p.initialise (10,20) ; p.colore (5) ;
p.affiche () ;
p.deplace (2,4) ;
p.affiche () ;
}
```

C. les fonctions virtuelles et le polymorphisme

Introduction

Le C++ permet le polymorphisme, c'est-à-dire la capacité d'objets de différentes classes reliées par une relation d'héritage de réagir chacun à sa manière à l'appel

d’une même fonction virtuelle. Le polymorphisme est mis en œuvre à travers les fonctions virtuelles

- **Accès à la une fonction membre par pointeur**

  Ex :

```
#include <iostream>
using namespace std;
class Base
{
    public:
        void show()
        {cout <<''Base\n'';}
};
class Derv1: public Base
{
    public:
        void show()
        {cout << ''Derv1\n'';}
};
class Derv2: public Base
{
    public:
```

```
        void show()
        {cout << ''Derv2\n'';}
};

int main()
{
    Derv1 dv1;
    Derv2 dv2;
    Base* ptr;
    ptr = &dv1;
    ptr -> show();
    ptr = &dv2;
    ptr ->show();
    return 0;
}
```

On a mis l'adresse des classes dérivées dans le pointeur de la classe de base. Le problème ne se pose pas car les pointeurs des objets des classes dérivées sont compatibles avec les pointeurs des objets de la classe de base.

Le problème est que dans tous les deux c'est la méthode associée à la classe de base qui est exécutée : le compilateur ignore le contenu du pointeur ***ptr*** et choisit la fonction membre dont le type correspond au pointeur.

Comment alors accéder aux objets de classes différentes en utilisant la même instruction ?

- **Accès à une fonction membre virtuelle par pointeur**

Soit f une fonction membre d'une classe C. Si les conditions suivantes sont réunies :

• f est redéfini dans des classes dérivées (directement ou indirectement) de C,

• f est souvent appelé à travers des pointeurs ou des références sur des objets de C ou de classes dérivées de C.

Alors f mérite d'être une *fonction virtuelle*. Pour rendre une fonction virtuelle, il suffit de placer le mot-clé ***virtual*** devant sa déclaration.

L'important service obtenu est celui-ci : si f est appelée à travers un pointeur ou une référence sur un objet de C, le choix de la fonction effectivement activée, parmi les diverses redéfinitions de f, se fera d'après le type dynamique de cet objet.

Une classe possédant des fonctions virtuelles est dite classe *polymorphe*. La qualification ***virtual*** devant la redéfinition d'une fonction virtuelle est facultative : les redéfinitions d'une fonction virtuelle sont virtuelles d'office.

Nous allons faire une légère modification dans le programme ci-dessus : nous allons rendre virtuelle la fonction *show()* de la classe de base.

```
class Base
{public:
        virtual  void show()
          {cout <<''Base\n'';}
};
```

Après cette modification, ceux sont les fonctions membres des classes dérivées qui seront exécutées.

- **Classes abstraites et fonctions virtuelles pures**

Pour le compilateur, une *classe abstraite* est une classe qui a des fonctions virtuelles pures. On rend une fonction virtuelle pure en ajoutant l'expression **=0** à sa déclaration.

Tenter de créer des objets d'une classe abstraite est une erreur, que le compilateur signale. D'une part, la fonction virtuelle pure ne peut pas posséder une implémentation. D'autre part, cela crée, pour le programmeur, l'obligation de définir cette implémentation dans une classe dérivée ultérieure. Une fonction virtuelle pure reste virtuelle pure dans les classes dérivées, aussi longtemps qu'elle ne fait pas l'objet d'une redéfinition (autre que « = 0 »).

Une fonction virtuelle pure n'a pa de corps.

Ex : rendre la classe Base précédente abstraite

```
class Base  // classe de base
{
    public:
      virtual     void show()=0;// méthode virtuelle pure
};
```

- **Destructeurs virtuels**

Le destructeur de la classe de base devrait toujours être virtuel. Cela permet de s'assurer que les objets des classes dérivées sont correctement détruits.

Ex :

```
class Base  // classe de base
{
        public:
                ~Base(); // destructeur non virtuel
         //  virtual   Base()// méthode virtuelle pure
                { cout << ''Classe de base détruite\n'' ;}
};

class Derv : public Base
{
        public :
                ~Derv()
                        {cout << ''Dev détruite\n'';}
};
int main()
{
```

```
        Base* pBase = new Derv;
        delete pBase;
        return 0;
    }
```

- **Le pointeur this**

En C++ le mot-clé ***this*** permet à tout moment dans une fonction membre d'accéder à un pointeur sur l'objet manipulé.

```
// Le pointeur this
#include <iostream>
using namespace std;
class Where
{
    private:
        char charray[10];
    public:
        void reveal()
            {
            cout << '\n L'adresse de mon objet est  :' << this;
            }
} ;
```

```
int main()
{
        where w1, w2, w3 ;
          w1.reveal() ;
           w2.reveal();
           w3.reveal();
           w4.reveal();
           cout << endl;
           return 0;
}
```

Accès aux données membres par le pointeur ***this***

Le pointeur this peut être traité comme tout autre pointeur et peut donc être utilisé pour accéder aux données de l'objet qu'il pointe.

Ex :

```
#include <iostream>
using namespace std ;
class What
{
        private:
```

```
        int alpha;
    public:
        void tester()
        {
            this -> alpha =11;
            cout << this ->alpha ; // même chose que cout<<alpha ;
        }
};
int main()
{
    What w ;
    w.tester() ;
    cout << endl ;
    return 0 ;
}
```

CHAPITRE 3 : EXERCICE DE COURS ET CONCEPTION DE QUELQUES D'APPLICATION GRAPHIQUES

BIBLIOGRAPHIE

Christian Casteyde , *Cours de C/C++ ;*

Mathieu Nabra et Matthieu Schaller , *Programmer avec le langage C++ ;*

Claude Delannoy , *C++ pour les programmeurs C ;*

O. Marguin , *C++ : Les bases*, (2003/2004) ;

SITE WEB UTILE

www.openclassroom.com
www.developer.com