Auteur : Bruno. Wieckowski (Exood4 Studios)
Dernière mise à jour : 27-04-2006

Le LANGAGE C++

Présentation

La programmation par Objet (appelée aussi Programmation Orientée-Objet) est un réponse possible à la demande des programmeurs qui ont à développer des logiciels de plus en plus complexe et volumineux. Son émergence est à rapprocher de celle de la programmation structurée, dans les années 60-70, qui elle aussi répondait à un besoin urgent de standardisation des modes de programmation.

A la différence de la programmation structurée, dans laquelle un programme est organisé sous la forme d'un certain nombre de procédures qui manipulent de entités partagées, la programmation par objet tend à organiser le programme autour de quelques objets dotés de propriétés qui leur permettent d'interagir les uns sur les autres.

C++ doit beaucoup au langage C. Des versions antérieures du langage connues sous le nom "C avec Classes" sont utilisées depuis 1980. Elles étaient utilisées pour des simulations (temps et espace mémoire réduits). C++ a été utilisé la première fois en juillet 1983. Son nom exprime la nature évolutive des changements opérés au langage C. Plus tard, la croissance explosive de l'utilisation du C++ a provoqué des changements.

A l'heure actuelle, on tend vers une formalisation formelle de C++. La langage C a lui aussi évolué, en partie sous l'influence du C++. Le standard ANSI C contient la syntaxe de la déclaration de fonction empruntée au "C avec Classes". C++ est maintenant plus compatible avec C qu'il ne l'était auparavant.

Le passage de C à C++

C++ est un sur-ensemble du C tel qu'il est défini dans la norme ANSI. Ce qui signifie que C++ est compatible avec C : il utilise les types fondamentaux **char**, **int**, **float**, **double** ... Mais le C++ permet d'aller plus loin pour construire des nouveaux types de données et faire de la Programmation Objet en utilisant des types abstraits. Il a été conçu pour rendre la programmation plus agréable.

Incompatibilités entre C et C++

1) Déclarations des fonctions

En C++ toute fonction utilisée doit obligatoirement avoir fait l'objet :

- soit d'une déclaration sous forme de prototype :

```
float fct (int,double,char);
```

-
- soit d'une définition préalable au sein du fichier source, avec chaque argument précédé de son type :

```
float fct (int x, double y, char z)
{
...
}
```

- Une fonction sans valeur de retour doit obligatoirement être de type void.

- contrairement au C, une fonction C++ qui n'a pas d'argument n'a pas besoin d'avoir **void** comme paramètre :

```
int fct (void);// fonction sans argument en C
int fct ();// fonction sans argument en C++
```

2) Les constantes

La norme C Ansi a introduit le qualificatif **const** pour déclarer des constantes. On peut également les simuler en utilisant les macros : **#define**. Les constates sont des entités à part entière. Une constante se déclare de la même façon qu'une variable initialisée mais elle est introduite par le mot réservé **const**. Il est impossible de modifier le contenu d'une constante.

```
const float pi = 3.14159;
const char espace = ' ';
```

C++ admet également l'utilisation des constantes. Lorsque **const** s'applique à des variables locales automatiques, aucune différence n'existe entre C et C++; la portée étant limitée au bloc ou à la fonction concernée par la déclaration. Par contre, lorsque **const** s'applique à une variable globale, C++ limite la portée du symbole au fichier source concernant la déclaration (le langage C ne fait aucune limitation).

Remarque :

Les constantes sont souvent utilisées en C++ dans le cas où les arguments de fonction sont passés par référence alors qu'on ne doit en aucun cas les modifier dans le corps de la fonction.

```
float moyenne (const int notes[], int nbnotes)
{
float somme = 0;
for (int i = 0; i < nbnotes; i++) somme = somme + notes[i];
return (somme / nbnotes);
}
```

3) Définition d'une structure

En C, lorsqu'on définit une structure, il est nécessaire de rappeler **struct** pour déclarer des variables de ce type :

```
struct point
{
float abscisse;
float ordonnee;
};

struct point a, b;
```

Pour éviter le rappel dans une déclaration de variable, on peut introduire une structure par l'intermédiaire d'une définition de type en utilisant le qualificatif **typedef** :

```
typedef struct point
{
float abscisse;
float ordonnee;
};

point a, b;
```

En C++ une structure n'a plus besoin d'être introduite par le qualificatif **typedef** :

```
struct point
{
float abscisse;
float ordonnee;
};

point a, b;
```

Spécificités de C++

C++ dispose, par rapport au C, d'un certain nombre de spécificités qui ne sont pas véritablement axées sur la Programmation Objet.

1) Les entrées/sorties

cin et **cout** remplacent avantageusement les fonctions **printf()** et **scanf()** de la librairie standard d'entrées-sorties du C dont les spécifications se trouvent dans le fichier d'en-tête <stdio.h> :

- **cout** affiche sur la sortie standard **stdout** les valeurs des différentes expressions, suivant une présentation adaptée à leur type.

  cout <<expression1<<expression2<< ... <<expressionN;

- **cin** lit sur l'entrée standard **stdin** les différentes valeurs et les affecte aux identificateurs de variables précisés dans l'appel.

cin >>idenfiticateur1>>identificateur2>> ... >>identificateurN;

Pour pouvoir utiliser **cin** et **cout**, il faut inclure le fichier d'en-tête <iostream.h> dans le programme source :

```
#include <iostream.h>
void main()
{
int a,b;
cout << "Entrer deux entiers";
cin >> a >> b;
cout << "le produit de " << a << " par " << b << "\nest : " << a*b;
}
```

Remarques :

- *Le manipulateur **flush** permet de forcer le vidage de la mémoire tampon associée au fichier de sortie*

- *Le manipulateur **endl** insère d'abord le caractère '\n' avant de faire **flush***

```
#include <iostream.h>
void main()
{
int a,b;
cout << "Entrer deux entiers" << flush;
cin >> a >> b;
cout << "le produit de " << a << " par " << b << "\nest : " << a*b << endl;
}
```

2) Les commentaires

C++ admet une autre forme de commentaires en utilisant //. La fin de ligne est considérée comme fin de commentaire.

```
void main()
{
int a=5, b=8;
// déclarations de deux variables
a = a + b;
// somme
// de deux variables
}
```

3) La déclaration des variables

En C, il est nécessaire de déclarer toutes les variables en début de bloc { .. } ou en début de programme. En C++, la déclaration peut se faire n'importe où. La portée reste limitée au bloc ou à la fonction suivant l'endroit où elle a été déclarée.

```
void main()
{
int nb;
cout << "taille du tableau :";
cin >> nb;
int t[nb];
for (int i = 0; i < nb; i++)
{
cout << "entrer une valeur :";
cin >> t[i];
}
...
}
```

4) Les références

Une donnée est accessible soit par le nom d'une variable, soit par l'adresse de la case mémoire où elle est stockée. Il est possible de faire référence à la même zone qui contient la donnée en utilisant une seconde référence.

```
int i = 1;
int &j = i;
i++;// i et j valent 2
j++;// i et j valent 3
```

Remarque :

Il n'y a aucune opération sur la référence mais uniquement sur l'objet référencé.

5) La transmission par référence

En C, la transmission des paramètres par référence n'existe pas. On est obligé de la gérer en manipulant explicitement des pointeurs. En C++, la transmission par référence est possible en faisant précéder le nom de l'argument dans l'en-tête d'une fonction par le symbole **&**.

Langage C	Langage C++

`#include <stdio.h>`
 	
`void somme (int x, int y, int *z)`
`{`
`*z = x+y;`
`}`
 	
`void main(void)`
`{`
`int a, b, c;`
 	
`printf ('entrer a ';);`
`scanf ("%d",&a);`
`printf ("entrer b");`
`scanf ("%d",&b);`
`somme (a, b, &c);`
`printf ("somme : %d\n",c);`
`}`
	`#include <iostream.h>`
	
	`void somme (int x, int y, int &z)`
	`{`
	`z = x + y;`
	`}`
	
	`void main()`
	`{`
	`int a, b, c;`
	
	`cout << "entrer a ";`
	`cin >> a;`
	`cout << "entrer b ";`
	`cin >> b;`
	`somme (a, b, c);`
	`cout << "somme : " << c << "\n";`
	`}`

6) Les arguments par défaut

Dans la déclaration d'une fonction il est possible de prévoir pour un ou plusieurs arguments une valeur par défaut. Elle est indiquée par le symbole = à la suite de l'argument. Les arguments par défaut sont obligatoirement les derniers de la liste.

```
#include <iostream.h>

int somme (int x, int y=1, int z=1)
{
return (x+y+z);
}

void main()
{
int a=2, b=3, c=4;

cout << "somme : " << somme (a, b, c) << "\n";// somme : 9
cout << "somme : " << somme (a, b) << "\n";// somme : 6
cout << "somme : " << somme (a) << "\n";// somme : 4
}
```

7) La surcharge des fonctions

Au sein d'un même programme, il est possible que plusieurs fonctions possèdent le même nom. Le choix de la fonction utilisée dépendra du type des arguments à l'appel.

```
float somme (float x, float y) { return (x+y); }

int somme (int x, int y) { return (x+y); }

void main ()
{
int a=3, b=4;
float c=2.5, d=1.0;

cout << somme (a, b) << "\n";// 7
cout << somme (c, d) << "\n";// 3.5
}
```

8) Allocation dynamique

En C++, les fonctions **malloc()** et **free()** sont remplacées par les opérateurs **new** et **delete**.

Exemple : *allocation dynamique d'un emplacement mémoire pour y mettre un réel*

Langage C	Langage C++

#include <stdlib.h>
...
float *x;
x = (float *) malloc (sizeof(float));
*x = 126.5;
...
free(x);
...
	
	...
	float *x;
	x = new float;
	*x = 126.5;
	...
	delete x;
	...

En cas de succès, l'opérateur **new** fournit un pointeur sur la zone mémoire allouée. En cas d'échec, **new** retourne un pointeur **NULL**. L'opérateur **delete**, quant à lui permet de libérer la place mémoire allouée par **new**.

Exemple : *on veut allouer et libérer dynamiquement un emplacement mémoire de 20 caractères consécutifs*

```
char *ptr_tab;

ptr_tab = new char[20];
delete ptr_tab;// équivalent à : delete[20] ptr_tab</FONT<
```

9) L'opérateur de portée

En C, tout variable globale est masquée dans un bloc si une variable locale de même nom y est déclarée; la variable globale est donc inaccessible dans ce bloc.

```
int var = 1;
...
void fct()
{
int var = 134;

printf ("%d\n", var);// affiche 134
}
```

En C++, il existe un opérateur (noté '::') appelé opérateur de résolution de portée, qui permet l'accès explicite à une variable globale.

```
int var = 1;
...
void fct()
{
int var = 134;

cout << var << ::var;// affiche 1 et 134
}
```

10) Les conversions explicites

En C, pour effectuer une conversion explicite, on utilise un forçage de type (cast) qui permet de transformer une expression dans le type voulu : **(type) expression**

Exemple : *double moyenne = (double) somme/nb;*

C++ autorise une autre écriture pour réaliser la même chose : **type (expression)**

Exemple : *double moyenne = double(somme)/nb*

Remarque :

> *En C++, le type void *, contrairement au C, ne peut être converti implicitement en un pointeur d'un autre type. Il faut obligatoirement effectuer un forçage de type. Par contre tout pointeur peut être converti implicitement en un pointeur de type void *.*

11) Les fonctions "en ligne"

En C, lorsque le corps d'une fonction est courte et que le temps d'exécution est à privilégier par rapport à l'espace mémoire, on utilise une macro. En C++, l'écriture des fonctions peut se faire "en ligne" par l'intermédiaire du qualificatif **inline**. Dans la macro, la substitution se fait par le préprocesseur, dans la fonction **inline**, le compilateur sait que chaque occurrence de l'appel de la fonction devra être remplacée par le code machine correspondant.

Ce type de fonction permet également d'avoir un contrôle plus strict au niveau des paramètres transmis.

```
// macro de calcul d'un maximum
#define max(a,b)a>b?a:b// aucun contrôle sur le type de a et b

// fonction en ligne du calcul du maximum
inline int max (int a, int b) { return (a>b?a:b); }// ici a et b devront être des entiers
```

Limite des fonctions inline :

> Le compilateur peut très bien ignorer une fonction **inline** si cette dernière est trop "complexe". Par exemple, dans le cas d'une fonction trop longue ou contenant des boucles, le compilateur la considèrera comme une fonction simple. Il est donc nécessaire d'utiliser les fonctions **inline** pour des opérations rapides (séquences d'instructions, affectations etc...)

12) Les fonctions génériques

Une fonction générique est une fonction qui ne dépend pas des types des arguments instanciés. Un mécanisme très puissant est prévu en C++ pour pouvoir créer des fonctions génériques tout en conservant le contrôle de type : c'est le concept des **templates**.

Pour créer une fonction générique, l'en-tête de la fonction doit être précédée du mot réservé **template** suivi d'une liste de paramètres entre les symboles < > Chaque paramètre doit être de la forme : **class identificateur**. Le choix de l'identificateur est laissé au programmeur. L'identificateur représente un type qui est instancié au moment de la compilation par un type existant.

Exemple :

> *La permutation de deux valeurs entières et deux valeurs réelles nécessite l'écriture de deux fonctions distinctes dont les instructions sont identiques excepté le type des variables. L'écriture d'une fonction générique permut() permet d'éviter cette redondance de code :*

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

c = a; a = b; b = c;
}

void main ()
{
int x=5, y=8;
float z=2.5, t=8.9;

permut (x, y);
permut (z, t);
}
```

Remarque :

> *Le compilateur détecte une erreur si à l'appel de la fonction les deux paramètres sont de types différents.*

Pour utiliser des types génériques différents dans une fonction générique, il suffit de fournir des noms symboliques différents.

```
template <class T1, class T2>
void fct (T1 a, T2 b)
{
...
}
```

Les classes

Introduction

La notion de classe est très similaire à celle des structures sachant qu'il est possible d'y ajouter des déclarations de fonctions. Les fonctions membres d'une classe sont appelées **méthodes** et les variables déclarées dans une classe sont appelées **attributs**.

En C++, les structures peuvent aussi contenir des fonctions. De là vient le rapprochement entre les structures et les classes :

```
struct point
{
float abscisse, ordonnee;

void initialiser()
{
abscisse = 0;
ordonnee = 0;
}

void afficher();
};

void point::afficher()
{
cout << "(" << abscisse << "," << ordonnee << ")\n";
}

point p;

p.initialiser();
p.afficher();
```

Nous allons voir que la déclaration d'une classe se fait en remplaçant tout simplement le mot-clé **struct** par le mot-clé **class**. Tout programme utilisant une classe A est "client de la classe A".

```
int x;// x est une instance du type int
MaClasse A;// A est une instance de la classe MaClasse
```

A est également appelé **objet**. Tout objet est forcément **instance** d'une classe.

Déclaration, définition et utilisation d'une classe

1) Déclaration

Dans la déclaration d'une classe nous pouvons définir 3 parties distinctes (mais non obligatoires) :

- partie privée (**private**): accessible que par les membres de la classe
- partie protégée (**protected**) : accessible par la classe et ses classes dérivées
- partie publique (**public**): accessible par les membres de la classe et l'extérieur

```
class point
{
private :
float abscisse, ordonnee;

public :
void initialiser (float,float);
void afficher ();
void deplacer (float,float);
};
```

Généralement on effectue les déclarations dans un fichier d'en-tête (header). Dans l'exemple précédent on écrirait la déclaration de la classe point dans le fichier point.h

2) Définition

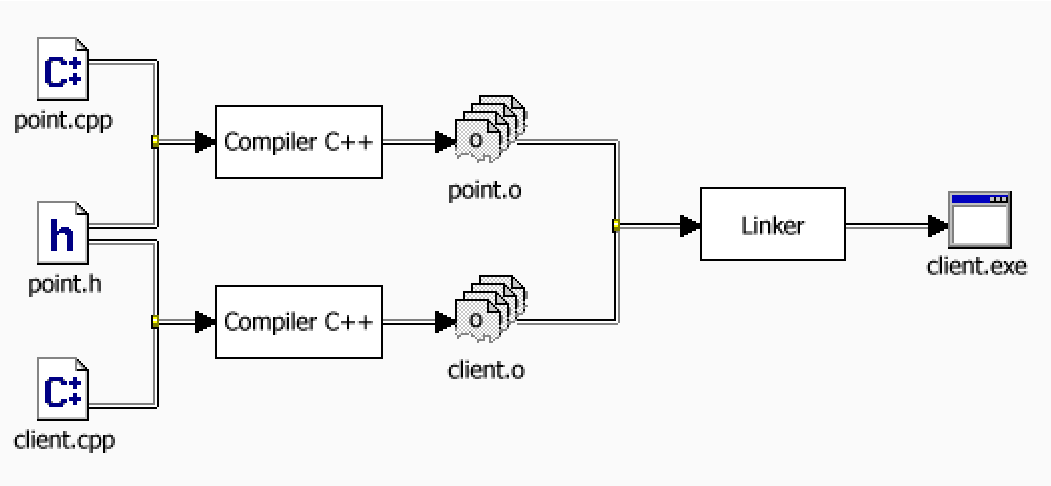
```
#include <iostream.h>
#include "point.h"

void point::initialiser (float x, float y)
{
abscisse = x;
ordonnee = y;
}

void point::afficher ()
{
cout << "("<< abscisse << ",";
cout << ordonnee << ")";
}

void point::deplacer (float x, float y)
{
abscisse = abscisse + x;
ordonnee = ordonnee + y;
}
```

De la même manière que la déclaration, on utilise un fichier différent pour écrire la définition de la classe. Ces fichiers peuvent avoir plusieurs extensions : .C, .cxx, .cpp . Par exemple, on écrirait la définition de la classe point dans le fichier point.cpp

3) Utilisation

```
#include "point.h"

void main ()
{
point p;

p.initialiser (3.5, 6.2);
p.afficher ();
p.deplacer (3.0, 2.0);
p.afficher ();
}
```

Compilation



La variable d'auto-référence this

Dans la définition d'une classe, il est possible de faire référence à l'objet en cours d'utilisation. On utilise pour cela l'identifiant **this** qui est un pointeur de l'objet sur lequel s'applique la méthode.

```
void point::afficher ()
{
cout << this->abscisse;// équivalent à : cout << abscisse;
cout << this->ordonnee;// équivalent à : cout << ordonnee;
}
```

Les instances

Le constructeur

Un constructeur de classe :

- est une fonction membre de la classe
- a le même nom que la classe
- peut s'utiliser avec ou sans argument(s)
- ne retourne pas de valeur
- est appelé après l'allocation mémoire de l'objet

Le constructeur sans argument

```
// Déclaration de la classe point
class point
{
private :
float abscisse, ordonnee;

public :
point ();// constructeur sans argument de la classe point

void afficher ();
};

// Définition du constructeur sans argument de la classe point
point::point ()
{
abscisse = 0.0;
ordonnee = 0.0;
}

// Programme principal utilisant la classe point (client de la classe)
void main ()
{
point p;

p.afficher ();// affiche : (0.0, 0.0)
}
```

Le constructeur avec argument

```
// Déclaration de la classe point
class point
{
private :
float abscisse, ordonnee;

public :
point (float, float);// constructeur avec arguments de la classe point

void afficher ();
};

// Définition du constructeur avec arguments de la classe point
point::point (float x, float y)
{
abscisse = x;
ordonnee = y;
}

// Programme principal utilisant la classe point (client de la classe)
void main ()
{
point p (3.5, 5.2);// on ne peut plus écrire : point p;
// car il n'y a plus de constructeur sans argument

p.afficher ();// affiche : (3.5, 5.2)
}
```

La surcharge du constructeur

```
// Déclaration de la classe point
class point
{
private :
float abscisse, ordonnee;

public :
point ();// constructeur sans argument de la classe point
point (float, float);// constructeur avec arguments de la classe point

void afficher ();
};

// Définition du constructeur sans argument de la classe point
point::point ()
{
abscisse = 0.0;
ordonnee = 0.0;
}

// Définition du constructeur avec arguments de la classe point
point::point (float x, float y)
{
abscisse = x;
ordonnee = y;
}

// Programme principal utilisant la classe point (client de la classe)
void main ()
{
point p1 (3.5, 5.2);
point p2;

p1.afficher ();// affiche : (3.5, 5.2)
p2.afficher ();// affiche : (0.0, 0.0)
}
```

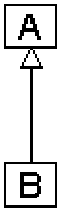
Le constructeur avec argument par défaut

```
// Déclaration de la classe point
class point
{
private :
float abscisse, ordonnee;

public :
point (float=0.0, float=0.0);// constructeur avec arguments par défaut de la classe point

void afficher ();
};

// Définition du constructeur avec arguments par défaut de la classe point
point::point (float x, float y)
{
abscisse = x;
ordonnee = y;
}

// Programme principal utilisant la classe point (client de la classe)
void main ()
{
point p1 (3.5, 5.2);
point p2;

p1.afficher ();// affiche : (3.5, 5.2)
p2.afficher ();// affiche : (0.0, 0.0)
}
```

L' héritage simple

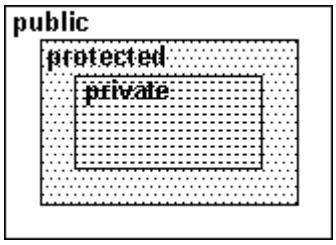
Introduction

L'héritage consiste à dériver une classe à partir d'une autre classe de base.



On dit que la classe B hérite de la classe A, ou B est une classe fille de la classe A, ou encore A est une classe parente de la classe B etc...

Toute classe B, héritant d'une classe A, accède à l'ensemble des déclarations de A sous certaines conditions :



public : accessible par tout le monde (membres de la classe, classes dérivées et clients de la classe)

protected : accessible par les membres de la classe et par les classes dérivées

private : accessible par les membres de la classe uniquement

Classe dérivées publiques

La spécification d'une classe B héritant de A s'écrira :

```
class B : public A
{
...
};
```

Exemple :

```
// Déclaration de la classe point
class point
{
private :
float abscisse, ordonnee;

public :
void point (float, float);
void afficher();
};

// Déclaration de la classe dpoint héritant de la classe point
class dpoint : public point
{
public :
void deplacer (float, float);
}
```

Implémentation :

```
#include "point.h"
#include "dpoint.h"

// Définition de la méthode deplacer de la classe dpoint
void dpoint::deplacer (float x, float y)
{
abscisse = abscisse + x;
ordonnee = ordonnee + y;
}
```

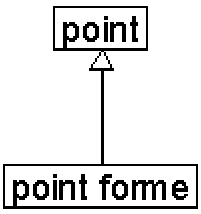
Les attributs **abscisse** et **ordonnee** sont déclarés dans la partie privée de la classe **point**. Pour les rendre accessibles à la classe **dpoint** il faut donc remplacer **private** par **protected** :

```
// Déclaration de la classe point
class point
{
protected :
float abscisse, ordonnee;

public :
void point (float, float);
void afficher ();
};
```

Programme client de dpoint :

```
#include "point.h"
#include "dpoint.h"

void main ()
{
dpoint p;

p.point (5.0, 6.5);
p.deplacer (8.0, 11.5);
p.afficher ();
}
```

Constructeur dans les classes dérivées




```
// Déclaration de la classe pointforme
class pointforme : public point
{
private :
char forme;

public :
pointforme (float, float, char);// constructeur de la classe pointforme

void deformer (char);
void afficher ();
};

// Définition du constructeur de la classe pointforme en appelant celui de la classe point
pointforme::pointforme (float x, float y, char f) : point (x, y)
{
forme = f;
}

void pointforme::afficher ()
{
point::afficher ();
cout << "forme : " << forme << "\n";
}

pointforme p(3.5, 6.2, '*');// utilisation du constructeur de la classe pointforme
```

Fonctions et classes "amies"

La protection des attributs ou des méthodes d'une classe (**private**, **protected**) peut parfois poser problème lors de son utilisation. Il peut être intéressant d'avoir accès à ses membres privés et protégés depuis l'extérieur (pour un certain nombre de fonctions ou de classes) sans pour autant perdre cette protection pour le reste du programme.

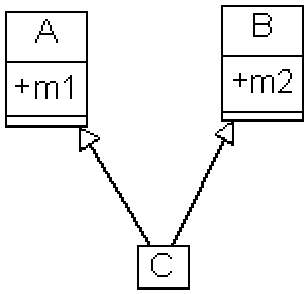
Ceci est tout à fait possible en utilisant la notion de fonction "amie" ou de classe "amie". Le mot clé utilisé est **friend**. C'est dans la classe permettant l'accès à ses informations que l'on déclarera les fonctions ou les classes amies.

```
// Déclaration de la classe point
class point
{
private :
float abscisse, ordonnee;

public :
point (float, float);

friend class utilisepoint;// utilisepoint est une classe amie de la classe point

friend void modifierAbscisse (point, float);// modifierAbscisse est une fonction amie
// de la classe point

friend void modifierOrdonnee (point, float);// modifierOrdonnee est une fonction amie
```

```
};

// Déclaration de la classe utilisepoint
class utilisepoint
{
private :
float x, y;

public :
utilisepoint (point);
};

// Définition du constructeur de la classe point
point::point (float x, float y)
{
abscisse = x;
ordonnee = y;
}

// Définition du constructeur de la classe utilisepoint à partir d'un objet de type point
utilisepoint::utilisepoint (point p)
{
x = p.abscisse;// l'accès à abscisse et ordonnee est possible
y = p.ordonnee;// car utilisepoint est une classe amie de point
}

// Définition de la fonction modifierAbscisse
void modifierAbscisse (point p, float x)
{
p.abscisse = x;// l'accès à abscisse est possible car modifierAbscisse
// est une fonction amie de la classe point
}

// Définition de la fonction modifierOrdonnee
void modifierOrdonnee (point p, float y)
{
p.ordonnee = y;// l'accès à ordonnee est possible car modifierOrdonnee
// est une fonction amie de la classe point
}

// Programme principal client des classes point et utilisepoint
void main ()
{
point p(2,3);

modifierAbscisse (p, 5);
modifierOrdonnee (p, 10);

utilisepoint p2(p);// p2 sera initialisé avec les valeurs 5 et 10
}
```

L' héritage multiple

Introduction



m1 : attribut de la classe A
m2 : attribut de la classe B

Par le mécanisme d'héritage, la classe C aura accès aux attributs m1 et m2 des classes A et B.

De façon similaire à l'héritage simple, la déclaration d'une classe C héritant d'une classe A et d'une classe B s'écrira :

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

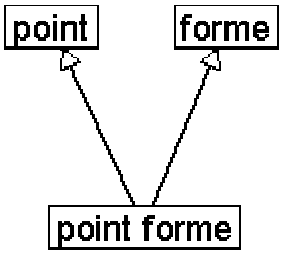
Constructeurs dans les classe dérivées

Soit le fichier d'en-tête **point.h** :

```
// Spécification de la classe point
class point
{
private :
float abscisse, ordonnee;

public :
point (float, float);

void afficher ();
};
```

et le fichier d'en-tête **forme.h** :

```
// Spécification de la classe forme
class forme
{
private :
char forme;

public :
forme (char);
void afficher ();
};
```



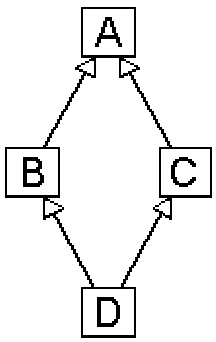
On écrira la spécification de la classe **pointforme** dans le fichier d'en-tête **pointforme.h** comme ceci :

```
// Spécification de classe pointforme héritant des classes point et forme
class pointforme : public point, public forme
{
public :
pointforme (float, float, char);

void afficher ();
};
```

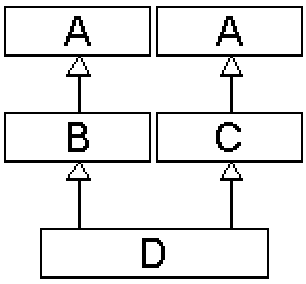
Et enfin son implémentation dans le fichier source **pointforme.cpp** :

```
#include "point.h"
#include "forme.h"
#include "pointforme.h"

// Implémentation du constructeur de la classe pointforme
// en appelant celui des classes point et forme
pointforme::pointforme (float x, float y, char f) : point (x, y), forme (f)
{
}

// Implémentation de la méthode afficher de la classe pointforme
void poinforme::afficher ()
{
point::afficher ();
forme::afficher ();
}
```

Un problème lié à l'héritage multiple



En instanciant la classe D, l'objet va se retrouver, par le mécanisme d'héritage, avec une double représentation de la classe A par le biais des classes B et C. On peut schématiser le résultat obtenu par le graphe d'héritage suivant :

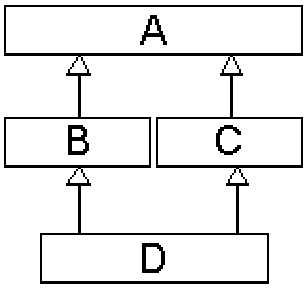


Les classes virtuelles

La solution pour obtenir une seule copie de la classe A en instanciant D est d'utiliser la notion de **classe virtuelle** :

```
class A
{
...
};

class B : public virtual A
{
...
};

class C : public virtual A
{
...
};

class D : public B, public C
{
...
};
```

Ce qui correspond à la représentation suivante :



Constructeurs et destructeurs

Constructeurs

Pour permettre l'affectation entre deux objets de même type, le compilateur utilise le constructeur de recopie, il est alors possible d'écrire :

```
point p1(3.5,5.2);
point p2 = p1;
```

Le constructeur de recopie par défaut recopie champ à champ les attributs d'un objet. Il est néanmoins possible de surcharger le constructeur de recopie dont la spécification est :

```
classe (classe &);
```

Exemple :

```
class point
{
private :
float abscisse, ordonnee;

public :
point (point &);
...
};

point::point(point &p)
{
abscisse = p.abscisse;
ordonnee = p.ordonnee;
}
```

Utilisation du constructeur de recopie

Le constructeur de recopie s'utilise lors de sa déclaration ou lors d'un passage par valeur d'un paramètre à une fonction.

Exemple : dans un programme client de la classe point

```
void fct1 (point p)
{
...
}
void fct2 ()
{
point x;
fct1 (x);// appel du constructeur de recopie
}
```

Remarque :

Ne pas confondre le constructeur de recopie avec l'opérateur d'affectation

Le destructeur

Le destructeur d'une classe est une fonction membre de la classe qui porte le même nom que la classe précédée du symbole ~ .

Le destructeur n'a jamais d'argument et ne retourne rien. Cette fonction est appelée lors de la destruction d'un objet (libération de l'espace alloué pour l'objet).

```
class point
{
private :
...

public :
~point ();
...
};

point::~point ()
{
cout << "destruction\n";
}

void main ()
{
point p1, p2;
}// destruction de p2 puis p1
```

Les destructions se font dans l'ordre inverse des constructions.

Différentes constructions à partir d'une classe

```
class point
{
private :
float abscisse, ordonnee;

public :
point (float=0.0, float=0.0);
~point ();

void afficher ();
};

point::point (float x, float y)
{
cout << "construction\n";
abscisse = x;
ordonnee = y;
}
```

1) Tableau d'instances d'une classe

```
#include "point.h"

void main ()
{
cout << "Debut\n";// Debut

point courbe[4];// construction (4 fois)

courbe[0].afficher ();// (0.0,0.0)
courbe[3].afficher ();// (0.0,0.0)

cout << "Fin\n";// Fin
}// destruction (courbe[3])
// destruction (courbe[2])
// destruction (courbe[1])
// destruction (courbe[0])
```

2) Pointeur d'objet

```
#include "point.h"

void main ()
{
point *p1, *p2;

cout << "Debut\n";// Debut

p1 = new point;// construction
p2 = new point(5.2, 6.3);// construction

p1->afficher ();// (0.0,0.0)

delete p1;// destruction de p1

p2->afficher ();// (5.2,6.3)

cout << "Fin\n";// Fin
}
```

La mémoire allouée pour p2 n'a pas été libérée !

*Tout objet alloué par la fonction **new** doit être libéré par la fonction **delete** avant la fin du programme ...*

3) Pointeur sur un tableau d'instances d'une classe

```
void main ()
{
point *courbe;

cout << "Debut\n";// Debut

courbe = new point[4];// construction * 4

courbe[0].afficher ();// (0.0,0.0)

delete courbe;// destruction * 4
```

```
cout << "Fin\n";// Fin
}
```

4) Tableau de pointeurs d'instances d'une classe

```
void main ()
{
point *courbe[4];

courbe[0] = new point(5.2,6.3);
courbe[3] = new point;

courbe[0]->afficher ();

delete courbe[0];
delete courbe[3];
}
```

Les fonctions virtuelles

Conversion implicite d'un objet d'une classe dérivée

```
point a;
pointforme b;

a = b;// affectation légale
b = a;// incorrect

point *pa;
pointforme *pb;

pa = pb;// correct
pb = pa;// incorrect
```

Typage statique des objets

```
point a, *pa;
pointforme b, *pb;

pa = &a;
pb = &b;

a.afficher ();
pa->afficher ();// méthode afficher de la classe point

b.afficher ();
pb->afficher ();// méthode afficher de la classe pointforme

pa = pb;
pa->afficher ();// méthode afficher de la classe point
```

Fonction virtuelle et typage dynamique

```
class point
{
private :
...

public :
virtual void afficher ();
};

class pointforme : public point
{
private :
...

public :
void afficher ();
}
```

Grâce à la fonction virtuelle **afficher()** de la classe **point**, le dernier appel **pa->afficher()** de l'exemple précédent utilisera la fonction afficher de la classe **pointforme**.

```
point a, *pa;
pointforme b, *pb;

pa = &a;
pb = &b;

a.afficher ();
pa->afficher ();// méthode afficher de la classe point

b.afficher ();
pb->afficher ();// méthode afficher de la classe pointforme

pa = pb;
pa->afficher ();// méthode afficher de la classe pointforme
```

Fonction virtuelle pour éviter des redondances de code

Jusqu'à présent nous déclarons les méthodes **afficher()** de chaque classe comme ceci :

```
void point::afficher ()
{
cout << abscisse << "," << ordonnee << "\n";
}

void pointforme::afficher ()
{
point::afficher ();
cout << forme << "\n";
}

void pointcolore::afficher ()
{
point::afficher ();
```

```
}
```

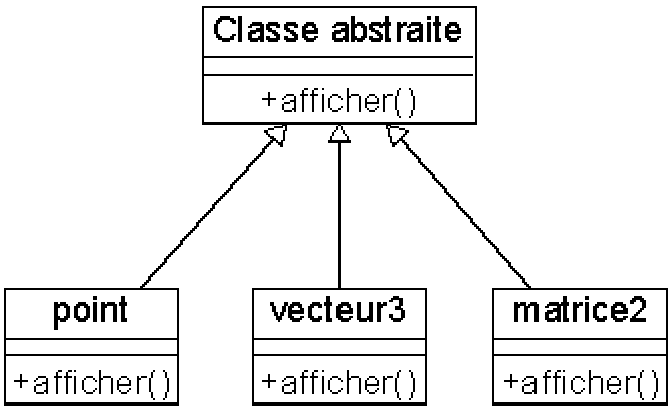
Il est donc nécessaire de répéter à chaque fois l'appel à la méthode **afficher()** de la classe de base **point** en utilisant l'opérateur de résolution de portée ::

Les fonctions virtuelles permettent d'éviter cette répétition de code :

```
//////////////////////////////////////////////////////////////////////
// Déclaration de la classe point
class point
{
private:
...

public :
void afficher ();
virtual void identifier ();
};

// Définition de la nouvelle méthode afficher de la classe point
void point::afficher ()
{
cout << abscisse << "," << ordonnee << "\n";
identifier ();
}

// Définition de la méthode identifier de la classe point
// (méthode qui sera réécrite par les classes dérivées)
void point::identifier ()
{
}

//////////////////////////////////////////////////////////////////////
// Déclaration de la classe pointforme
class pointforme : public point
{
private :
...

public :
void identifier ();// méthode identifier de la classe pointforme
};

// Re-définition dans la classe pointforme de la méthode identifier de la classe point
void pointforme::identifier ()
{
cout << forme << "\n";
}

//////////////////////////////////////////////////////////////////////
// Déclaration de la classe pointcolore
class pointcolore : public point
{
private :
...

public :
void identifier ();// méthode identifier de la classe pointcolore
};

// Re-définition dans la classe pointcolore de la méthode identifier de la classe point
void pointcolore::identifier ()
{
cout << couleur << "\n";
}

point p1;
pointcolore p2;
pointforme p3;
```

```
p1.afficher ();// 0.0,0.0
p2.afficher ();// 0.0,0.0 1
p3.afficher ();// 0.0,0.0 *
```

Fonction virtuelle pure et classe abstraite




```
class classe_abstraite
{
public :
virtual void afficher () = 0;// méthode virtuelle pure
};
```

Si l'on a une méthode virtuelle pure dans une classe, la classe est dite abstraite. Une classe abstraite ne peut jamais être instanciée. Les classes dérivées doivent définir le corps de la méthode virtuelle pure.

```
//////////////////////////////////////////////////////////////////
// Déclaration de la classe point
class point : public classe_abstraite
{
private :
float abscisse, ordonnee;

public :
...
void afficher () { cout << abscisse << "," << ordonnee; }
};

//////////////////////////////////////////////////////////////////
// Déclaration de la classe vecteur3
class vecteur3 : public classe_abstraite
{
private :
float x, y, z;

public :
...
void afficher () { cout << x << "," << y << "," << z; }
}

//////////////////////////////////////////////////////////////////
// Déclaration de la classe matrice2
```

```
{
private :
float m[2][2];

public :
...
void afficher () { cout << m[0][0] << "," << m[0][1] << "," << m[1][0] << "," << m[1][1]; }
}
```

Utilisation :

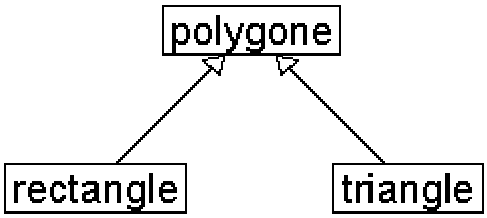
```
void main ()
{
classe_abstraite *t[3];

t[0] = new point(3.5, 5.2);
t[1] = new vecteur3(3.5, 6.2, 7.0);
t[2] = new matrice2(5.0, 5.0, 5.0, 5.0);

for (int i = 0; i < 3; i++)
{
t[i]->afficher ();// La bonne méthode afficher() sera automatiquement reconnue
}
}
```

Polymorphisme

1) Polymorphisme d'objet

On parle de **polymorphisme d'objet** lorsque tout objet instance d'une classe ancêtre peut être remplacé par un objet d'une classe descendante de la classe ancêtre.

Exemple :




```
// Déclaration de la classe polygone
class polygone
{
private :
...

public :
...
};

// Déclaration de la classe rectangle
class rectangle : public polygone
{
private :
...

public :
...
};
```

```
// Déclaration de la classe triangle
class triangle : public polygone
{
private :
...

public :
...
};
```

Soit **tableauPolygones** un tableau d'objet **polygone** :

```
polygone tableauPolygones[10];
```

Dans le tableau **tableauPolygones** on peut mémoriser des polygones, mais aussi des triangles et des rectangles. Nous avons donc plusieurs formes possibles pour **tableauPolygones[i]**. Il s'agit donc ici de polymorphisme d'objets.

2) Polymorphisme de méthode

De façon similaire, le **polymorphisme de méthode** apparaît lorsque l'on est confronté à plusieurs défintions d'une même méthode. Chaque méthode étant spécifique à l'objet (nous avons déjà rencontré ce cas pour la fonction **identifier** de la classe **point**) ...

Exemple :

```
// Déclaration de la classe polygone
class polygone
{
private :
...

public :
virtual float perimetre ();// somme des longueurs des côtés du polygone

...
};
...

// Déclaration de la classe rectangle
class rectangle : public polygone
{
private :
float longueur, largeur;
...

public :
// périmètre d'un rectangle
float perimetre () { return (2*(longueur + largeur)); }

...
};

// Déclaration de la classe triangle
class triangle : public polygone
{
private :
float base, adjacent, oppose;
...

public :
```

```
float perimetre () { return (base + adjacent + oppose); }

...
};
```

En reprenant l'exemple de **tableauPolygones** nous obtiendrons pour **tableauPolygones[i]** le calcul du périmètre du polygone correspondant à l'objet mémorisé. Nous avons plusieurs méthodes utilisées dans le tableau **tableauPolygones**. Il s'agit donc de polymorphisme de méthode.

Surcharge d' opérateurs

Le principe de surcharge

La surcharge des opérateurs consiste à définir le comportement d'un opérateur appliqué à un objet. On ne peut pas changer la pluralité des opérateurs, ni leur priorité (voir priorité des opérateurs en C)

Le mécanisme de surcharge des opérateurs

Notation : **operator op()** (**op** étant le symbole de l'opérateur à surcharger)

Exemple : * produit scalaire de 2 points, + addition de 2 points

```
class point
{
private :
float abscisse, ordonnee;

public :
point();
point(float,float);

void afficher ();

point operator + (point);
float operator * (point);
};

point point::operator +(point p)
{
point resultat;

resultat.abscisse = abscisse + p.abscisse;
resultat.ordonnee = ordonnee + p.ordonnee;

return resultat;
}

float point::operator *(point p)
{
return (abscisse * p.abscisse + ordonnee * p.ordonnee);
}
```

Exemple de programme client :

```
void main ()
{
point a(2.0, 3.0), b(1.0, 2.0), c;

c = a + b;
c.afficher ();
cout << "Produit scalaire : " << a * b << "\n";
}
```

a*b peut également s'écrire : a.operator *(b)

Les opérateurs de conversion

1) Conversion d'un type prédéfini ou existant vers un autre type

Exemple : conversion d'un réel en un point

```
class point
{
...

public :
point (float);

point operator + (point);
};

point::point (float x)
{
abscisse = x;
ordonnee = 0.0;
}

void main ()
{
point c;

c = 3.2;
c.afficher ();// 3.2,0.0
c = c + 2.5;// 2.5 devient (2.5,0.0)
c.afficher ();// 5.7,0.0
}
```

2) Conversion du type classe vers un type existant ou prédéfini

Notation : **operator identificateur_de_type()**

Exemple : conversion d'un point en un réel

```
class point
{
...

public :
...
```

```
};

point::operator float()
{
return abscisse;
}

void main ()
{
point c(3.2, 5.0);
float val;

val = c;
cout << val;// 3.2
val = val + c;// 6.4
}
```

Les classes génériques

Spécification d'une classe générique

En C la notion de généricité peut être simulée par l'utilisation de macros et de remplacements symboliques (voir objets génériques en C). Le C++ possède un outil plus puissant, les templates, que l'on utilise par l'intermédiaire du mot-clé **template**.

Exemple :

```
template <class T>
class Pile
{
private :
T *vecteur;// pile sous forme de tableau
int taille;// taille max de la pile
int sommet;// indice de sommet

public :
Pile(int);// constructeur avec taille
~Pile();// destructeur

int vide ();
T dernier ();
void empiler (T);
void depiler ();
int getSommet ();
};
```

Implémentation d'une classe générique

```
template <class T>
Pile<T>::Pile (int max)
{
vecteur = new T[max];
```

```
sommet = -1;
}

template <class T>
Pile<T>::~Pile ()
{
delete vecteur;
}

template <class T>
void Pile<T>::empiler (T element)
{
sommet++;
vecteur[sommet] = element;
}

template <class T>
void Pile<T>::depiler ()
{
sommet--;
}

template <class T>
T Pile<T>::dernier ()
{
return (vecteur[sommet]);
}

template <class T>
int Pile<T>::vide ()
{
return (sommet == -1);
}

template <class T>
int Pile<T>::getSommet ()
{
return (sommet);
}
```

Utilisation :

```
void main ()
{
Pile <int> p1(5);
Pile <point> p2(10);

p1.empiler(12);

point a(3.5, 5.2);

p2.empiler(a);

Pile <point *> p3(10);

p3.empiler (new point(3.5, 5.2));

cout << p1.getSommet() << "\n";
}
```

Variables et fonctions de classe

Variables de classe

On appelle variable de classe une donnée membre de la classe qui est partagée par l'ensemble des instances de cette classe. On utilise, pour déclarer ce genre de variable, le mot-clé **static**.

Les variables de classe n'existent qu'en un seul exemplaire, et elles sont automatiquement initialisées à 0.

```
class point
{
private :
float abscisse, ordonnee;
static int nbpoints;

public :
point();
~point ();
};

point::point ()
{
abscisse = 0;
ordonnee = 0;
nbpoints++;
}

point::~point ()
{
nbpoints--;
}
```

Fonction de classe

Une fonction de classe est un service fourni par la classe indépendamment de l'objet. Elle doit également être précédée du mot-clé **static**.

```
class point
{
...

static int compter ();
};

int point::compter ()
{
return nbpoints;
}
```

Utilisation des variables et des fonctions de classe

Pour accéder depuis un programme client à une variable ou une fonction de classe, on n'utilise pas un objet mais la classe directement (suivie de l'opérateur de résolution de portée et de la variable ou fonction que l'on souhaite utiliser).

```
void bidon ()
{
point z;

cout << point::compter () << "\n";
}

void main ()
{
cout << point::compter () << "\n";// 0

point x;

cout << point::compter () << "\n";// 1

point y;

cout << point::compter () << "\n";// 2

bidon ();// -> 3

cout << point::compter () << "\n";// 2
}
```

Les espaces de nommage

Dans un programme, les fonctions globales, les variables globales et les classes sont placés de façon implicite dans un espace de nommage global. Le mot-clé **static** permet de rendre locale une variable ou une fonction. Mais dans un grand projet, le manque de contrôle de l'espace de nommage global peut causer des problèmes.

Comme alternative à cela, les fabriquants (de librairies etc...) créent des noms longs et compliqués pour ne pas risquer d'entrer en conflit de nommage, mais nous sommes alors obligés d'utiliser et d'écrire ces noms longs dans nos programmes. **typedef** est souvent utilisé pour simplifier cela, mais ce n'est pas une solution élégante.

Il est possible de subdiviser l'espace de nommage global en plusieurs sous espaces en utilisant la commande **namespace** du C ++. Le mot-clé **namespace**, semblable à **class**, **struct**, **enum** et **union**, met les noms de ses membres dans un espace distinct. Bien que les autres mots-clés soient complémentaires, la création d'un nouvel espace de nommage est la seule utilisation possible pour **namespace**.

Création d'un espace de nommage

La création d'un espace de nommage est très similaire à la création d'une classe :

```
namespace MonEspace
{
// Déclarations
}
```

Ceci produit un nouvel espace de nommage contenant les déclarations situées entre les accolades. Toute variable, fonction, classe etc... déclarée dans un espace de nommage sera alors accessible en utilisant l'opérateur de résolution de portée ::

Exemple :

```
namespace MonEspace
{
```

```
}

int a;

void main ()
{
a = 1;// la variable a globale reçoit la valeur 1

MonEspace::a = 2;// la variable a de l'espace de nommage MonEspace reçoit 2
}
```

Une autre solution consiste à utiliser le mot-clé **using** :

```
namespace MonEspace
{
int a, b, c;
}

using namespace MonEspace;

void main ()
{
a = 2;// la variable a de l'espace de nommage MonEspace reçoit 2
}
```

Attention aux ambiguïtés lors de la compilation en utilisant using, toute déclaration contenue dans l'espace de nommage est alors susceptible d'entrer en conflit avec les déclarations globales du programme. A n'utiliser que dans un fichier d'implémentation (cpp) !

Il y a des différences importantes par rapport à **class** et **struct** :

- Un espace de nommage peut uniquement être défini en global ou à l'intérieur d'un autre espace de nommage
- Il n'y a pas de point-virgule à la fin de la déclaration d'un espace de nommage (après l'accolade)
- La définition d'un espace de nommage peut se déclarer en plusieurs parties :

```
namespace MonEspace
{
int a, b, c;
}

...
```

```
{
int d;// la déclaration de d se rajoute à l'espace MonEspace défini plus haut
}
```

- Il est possible de renommer un espace de nommage pour ne pas avoir à retaper un éventuel nom d'espace trop long :

```
namespace Ceci_est_un_exemple_d_espace_de_noms// un peu trop long ...
{
...
}

namespace nom_court = Ceci_est_un_exemple_d_espace_de_noms;
```

- Contrraitement aux classes il est impossible de créer une instance d'un espace de nommage.

Les espaces de nommage sans identifiant

Les espaces de nommage peuvent être déclarés sans identifiant, dans ce cas toute déclaration contenue dans cet espace sera considérée comme locale au fichier dans lequel il est défini. L'opérateur de résolution de portée :: ne peut plus être utilisé.

Exemple :

```
namespace// équivalent à déclarer static int a, b, c;
{
int a, b, c;
}

void main ()
{
a = 1;// a, b et c sont directement accessibles dans le module
b = c = 2;// contenant la déclaration de l'espace de nommage
}
```