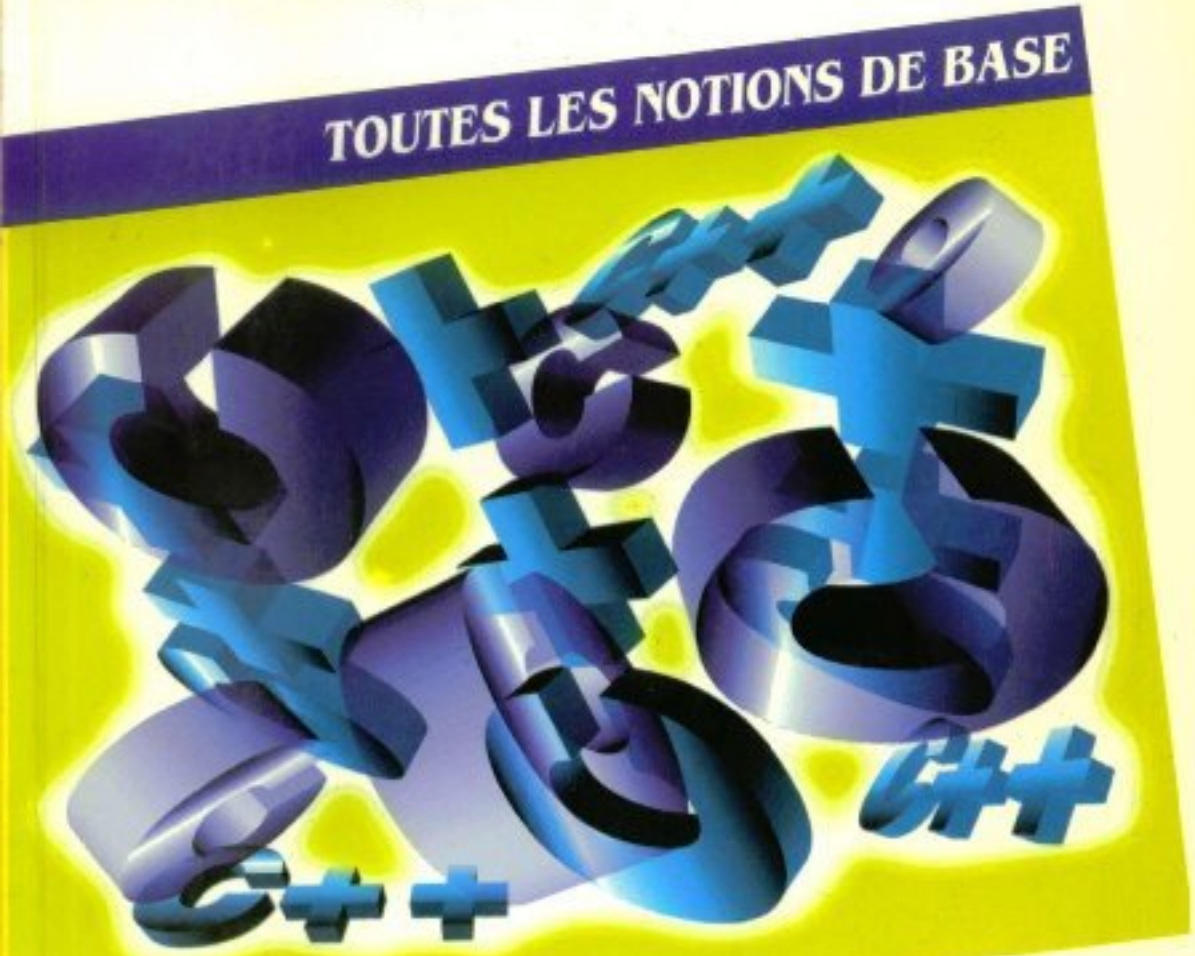
Pont entre C et C++

Pierre-Nicolas Lapointe

TOUTES LES NOTIONS DE BASE



Avec un aide-mémoire C++

Addison-Wesley

Table des Matières

données d'un objet, *données-membres*. Par opposition, les autres données et fonctions sont qualifiées de *hors-classe*.
Reprenons notre exemple. Voici comment nous aurions pu définir la classe Livre :

```
class Livre
{
   private :
        char    titre[20];
        char    auteur[20];
        char    éditeur[20];

   public :
        void    Saisir();
        void    Afficher();
        void    Imprimer();
};
```

Notez la présence du point-virgule après l'accolade fermante.

Comment définir une fonction-membre

Comme vous pouvez le constater, nous n'avons fait que *déclarer* les fonctions Saisir, Afficher et Imprimer. Il faut maintenant les *définir* complètement. La syntaxe est identique au langage C, à ceci près : il faut indiquer au compilateur à quelle classe est rattachée une fonction. Le format de définition d'une fonction-membre (c'est-à-dire inclue dans une classe) est le suivant :

```
type_retourné   NomDeVotreClasse::fonction(paramètres)
{
    // corps de la fonction
}
```

Le C++ n'aime pas forcer la main des programmeurs. Il est possible de définir des données public, c'est-à-dire accessibles directement de l'extérieur de l'objet C'est une pratique à déconseiller, car elle viole l'idée même d'encapsulation.

Ce qui donne dans notre exemple :

```
void        Livre::Saisir()
{
   puts("Entrez le titre du livre : " ) ;
   gets(titre);
   puts("Entrez le nom de l'auteur : " ) ;
   gets(auteur);
   puts("Entrez  l'éditeur  : " ) ;
   gets(éditeur);
}

void        Livre::Afficher()
{
   printf("Titre : %s par %s (%s)",
        titre, auteur, éditeur);
```

```
)

void        Livre::Imprimer()
{
    fprintf(stdprn, "Titre : %s par %s (%s)",
         titre, auteur, éditeur);
}
```

Les fonctions d'une classe peuvent librement accéder à toutes ses données-membres (ici : `titre`, `auteur` et `éditeur`). N'importe quelle fonction d'une classe peut également appeler une autre fonction de la même classe. Voilà ! Notre première classe est écrite ! Voyons maintenant de quelle manière nous pouvons l'utiliser.

Créer un objet

Pour utiliser une classe, il faut d'abord créer un objet qui aura comme type cette classe. La syntaxe est la même que pour déclarer une variable en C :

```
Nom_Classe   nom_objet;
```

Dans l'exemple du livre, cela donne :

```
Livre        mon_livre;
```

Bien entendu, toutes les formes de déclaration du C sont acceptées : tableaux, pointeurs, tableaux de pointeurs, etc. Par exemple :

```
Livre   ma_bibliotheque[20];
             // tableau de 20 objets-livres
Livre    *sur_ma_table_de_chevet;
             // pointeur sur un objet-livre
```

Accéder à la partie publique d'un objet

Maintenant que nous savons comment déclarer un objet, il reste à découvrir comment l'utiliser. En clair : comment saisir, afficher ou imprimer un objet « livre ». Vous ne serez pas dépaysé : l'accès aux fonctions-membres publiques d'un objet se fait comme si vous accédiez à une donnée d'une structure classique du langage C :

```
objet.fonction_publigue(paramètres);
pointeur_sur_objet -> fonction_publique(paramètres);
```

Il est temps d'écrire un petit programme qui va saisir puis afficher dix livres :

```
void         main()
{
    Livre    bouquin[10];
    int       i;

    for (i = 0; i < 10; i++)
         bouquin[i].Saisir();

    for (i = 0; i < 10; i++)
         bouquin[i].Afficher();
}
```

C'est aussi simple que cela. Vous savez maintenant comment écrire des classes simples, et comment utiliser des objets.

Précision importante sur l'accès aux données private

Vous vous rappelez que les données ou fonctions `private` d'une classe ne sont pas visibles à partir des fonctions d'un autre objet. Mais une petite précision s'impose : une fonction membre d'une classe A peut accéder *directement* à toutes les données (y compris `private`) d'autres objets *de classe A.* Ceci n'est pas forcément évident puisqu'elle manipule dans ce cas les données d'un autre objet. Elle peut pourtant accéder aux données et fonctions `private` de cet autre objet car il est de la même classe. Exemple :

```
class A
{
   private :
        int a;
   public :
        // ...
        void fontion_a();
};

class B
{
   private :
        int b;
   public :
        // ...
};
```

```
void        A::fonction_a()
{
   A    autre_a;
   B    objet_b;

   a = 1 ;            // OK : on reste dans cet objet
   autre_a.a = 1 ;    // OK : autre_a est de classe A
   objet_b.b = 1;     // NON : b est private
                      // dans une autre classe
}
```

L'intérêt par rapport au C

Vous l'avez vu, un objet regroupe des données et des fonctions qui opèrent sur ces données. Quel est l'intérêt de raisonner en objets plutôt qu'en fonctions ou en structures, comme c'est le cas en C ?

- Le code gagne en sécurité. Vous savez que les données d'un objet ne sont manipulées que par ses propres fonctions, les fonctions-membres. Vous pouvez donc contrôler l'intégrité des données et cibler vos recherches en cas de bogue.
- Les programmes gagnent en clarté. Si l'architecture des objets est bien conçue*, vous comprenez rapidement le rôle de tel ou tel objet. Une fonction ne se promène pas dans le vide, elle est rattachée à un objet, donc à l'ensemble des données qu'elle manipule.
- La clarté des programmes et leur cloisonnement en objets permettent une maintenance et une évolutivité plus facile.
- Grâce aux trois avantages précédents, des équipes de développeurs peuvent plus facilement envisager l'écriture de très gros programmes.

Certains pourraient prétendre que l'on peut simuler le principe de classe en C. Il suffirait, après tout, de créer une structure contenant des pointeurs de fonction, pour stocker les fonctions membres. Certes, mais s'agissant d'une gymnastique périlleuse, vous perdez le bénéfice de la clarté des programmes*. De plus, rien ne vous interdit d'accéder directement aux données de votre structure, ce qui tord le cou à l'une des règles fondamentales de l'encapsulation : cacher les données. Par ailleurs, vous serez dans l'incapacité de simuler les autres caractéristiques majeures du C++, que nous découvrirons dans les chapitres suivants.

* Les pointeurs de fonction ne rendent jamais un programme plus clair...

Résumé

Nous venons de présenter la notion la plus importante du C++ et des langages orientés-objets : *l'encapsulation.*

Le C++ permet de créer et manipuler des objets. Un objet se compose de *données* et de *fonctions* de traitement qui lui sont propres. Certaines de ces données et de ces fonctions sont cachées, c'est-à-dire qu'elles ne sont accessibles que de l'intérieur de l'objet, à partir de ses fonctions propres.

Pour créer un objet en C++, il faut d'abord définir un moule, appelé *classe* en C++. Les objets sont ensuite créés comme des variables classiques, à partir de leur classe.

Les données et fonctions définies dans une classe sont appelées respectivement *données membres* et *fonctions membres,* par opposition aux données et fonctions *hors-classe.* Par exemple, n'importe quelle fonction de la librairie standard du C (comme `printf`) est une fonction hors-classe.

Pour bien démarrer en C++

En tant que programmeur en langage C, la philosophie *objets* vous déroutera peut-être au début. Vous ne devez plus penser en terme de *fonctions,* mais en terme *d'objets,* regroupant leurs propres données et leurs propres fonctions. Par exemple, si vous voulez afficher le plateau d'un jeu, ne pensez pas à quelque chose comme `afficher_plateau (plateau)` mais plutôt à `plateau.afficher()`, où `plateau` est un objet contenant, par exemple, la position des pions du jeu. Bref, vous partez d'un objet et lui envoyez un message (sous forme de fonction). Rassurez-vous, c'est un réflexe qui viendra rapidement et que vous trouverez de plus en plus agréable. Bien que la philosophie objet nécessite une conception plus soignée qu'auparavant, elle procure un réel plaisir et beaucoup d'avantages.

Compléments

Rappelons que vous pouvez sauter cette section dans un premier temps, et passer directement au chapitre suivant. Si, toutefois, vous désirez en connaître un peu plus sur les classes, continuez votre lecture.

Fonctions inline

Les fonctions *inline* sont une facilité du C++ permettant d'optimiser la vitesse d'exécution des programmes. L'exécution d'une fonction *inline* est en effet plus rapide que celle d'une fonction définie normalement : le compilateur remplace les appels à de telles fonctions par le code de ces fonctions.
Cette remarque en engendre une autre : il vaut mieux que les fonctions *inline* soient très courtes (une ligne ou deux) pour éviter de cloner inutilement de nombreuses lignes de code.

Mise en œuvre C++
Examinons l'exemple suivant :

```
class UnNombre
{
   private :
        int      nbr;
   public :
        int      get_nbr ();
        void     set_nbr (int);
        void     Afficher();
};
```

Comme vous pouvez le voir, cette classe déclare trois fonctions publiques, qui ne sont encore définies nulle part. En théorie, il faudrait définir chacune de ces fonctions en dehors de la classe, de cette manière :

```
int UnNombre: :get_nbr()
{
   return nbr;
}
```

Le C++ met à notre disposition les fonctions *inline,* pour définir complètement une fonction à l'intérieur de la définition de sa classe. Ainsi, au lieu de définir à part des fonctions très

courtes, vous pouvez les inclure directement dans la classe. C'est ce que nous allons faire pour les fonctions `set_nbr` et `get_nbr`:

```
class UnNombre
{
   private :
        int      nbr;

   public :
        int      get_nbr()          { return nbr; }
        void     set_nbr(int n2)    { nbr = n2; }
        void     Afficher();
};
```

Comme vous pouvez le constater, nous avons défini complètement les deux premières fonctions publiques de cette classe (en gras dans l'exemple). Ces fonctions sont appelées *inline*. La troisième, `Afficher`, n'est que déclarée —notez le point-virgule après la déclaration—. Il faudra la définir en dehors de la classe, comme nous avions l'habitude de le faire.
Quand on définit une fonction *inline,* il est inutile d'accoler le nom de la classe et les caractères « : : » avant le nom de la fonction, puisqu'elle est définie dans le corps de sa classe.

Le mot-clé inline

Il existe un autre moyen de bénéficier du mécanisme inline, sans définir les fonctions directement dans la classe : il suffit d'utiliser le mot-clé `inline` en tête de la définition de la fonction. Quand vous spécifiez qu'une fonction est *inline,* vous donnez simplement une indication au compilateur, qui est libre de la suivre ou pas. D'ailleurs certains compilateurs refusent de rendre *inline* des fonctions qui contiennent des mots-clés comme `for` ou `while`.

Ce système est efficace si vous utilisez la compilation séparée. Nous détaillerons cela au chapitre 14.

Si, par exemple, nous avions voulu que `Afficher` soit une fonction inline, sans pour autant inclure son code dans la déclaration de la classe `UnNombre`, il aurait fallu écrire :

```
inline  void     UnNombre::Afficher()
{
   printf("Le nombre est : %d", nbr);
}
```

La déclaration de classe restant identique, mais il faut cette fois-ci indiquer au compilateur le nom de la classe à laquelle appartient la fonction inline (c'est le rôle de « UnNombre : : »), puisque cette fonction n'est pas déclarée directement dans la classe.

Résumé

Remarque : des fonctions hors-classes peuvent elles-aussi êtres définies inline.

Le C++ permet au programmeur de définir des fonctions-membre *inline*. Ces fonctions se comportent comme toutes les autres fonctions d'une classe, à la seule exception que lors de la compilation, chaque appel à une fonction inlinAest remplacé par le corps de cette fonction. Il en résulte un gain de temps à l'exécution. On réserve cependant cet avantage aux fonctions très courtes, afin que la taille finale de l'exécutable ne soit pas trop énorme.

Le mot-clé this

Chaque classe possède automatiquement une donnée cachée un peu spéciale : le pointeur this. Utilisé dans une fonction de classe, this est un pointeur sur l'objet courant à partir duquel on a appelé la fonction membre. Exemple :

```
class A
{
   private:
        int i ;
   public:
        voidf();
);

voidA::f()
{
   this -> i = 1;    // équivaut à i = 1;
}
```

this est de type « pointeur sur l'objet courant qui a appelé la fonction ». Dans l'exemple, this est de type pointeur de A. Vous en déduirez que (*this) représente l'objet courant, et vous aurez raison.

L'utilité de this est patente pour certains opérateurs, ou pour tester l'égalité de deux objets (voir page 61).

Éléments static dans une classe

Vous pouvez déclarer des objets, variables ou fonctions static dans une classe. Ces éléments seront alors communs à tous les objets de cette classe. Ainsi, une variable static aura la même valeur pour *tous les objets* de cette classe. Si l'un des objets change cette variable static, elle sera modifiée pour tous les autres objets.

On distingue ainsi les variables et fonctions *d'instance* (celles qui sont spécifiques à un objet), et les variables et fonctions de *classe* (communes à tous les objets de cette classe). Ces dernières doivent être déclarées static.

Il faut initialiser chaque variable static en dehors de la déclaration de classe, sans répéter le mot-clé static (voir l'exemple ci-dessous).

On ne peut pas utiliser le mot-clé this dans une fonction static, puisque celle-ci ne dépend pas d'un objet précis.

Les fonctions-membres déclarées static doivent être appelées à partir du nom de la classe, comme ceci : NomClasse : : f ( ), sans faire référence à un objet précis. Elles peuvent donc être appelées sans qu'aucun objet de la classe n'existe.

Pour comprendre cet exemple, vous devez d'abord lire le chapitre 2 sur les constructeurs.

```
#include <iostream.h>

class   Flam
{
   protected:
        static   int  nb_objets;
        int      donnée;
   public:
        Flam() : donnée(0) { nb_objets++; }
        static   affiche_nb_objets()
             { cout << nb_objets << endl; }
};

int     Flam::nb_objets = 0;   // initialisation

void    main()
{
   // appel de la fonction static à partir de la classe
   Flam::affiche_nb_objets();

   Flam     a, b, c; // déclarons 3 objets

   // appel de la fonction static à partir d'un objet
   a.affiche_nb_objets();
}

// affichage
// 0
// 3
```

Dans cet exemple, le constructeur de `Flam` ajoute 1 à la variable de classe `nb_objets` (déclarée `static` au sens C++). En créant trois objets `a`, `b` et `c`, nous appelons trois fois ce constructeur. On peut appeler une fonction `static` en la faisant précéder du nom de la classe suivi de deux double-points ou à partir d'un objet. Il n'y a pas de différence entre les deux.

Différence entre le mot-clé static **du C** et **du C++**
Attention : dans ce que nous venons d'expliquer, le mot-clé `static` sert à déclarer des variables ou fonctions de *classe*. En C standard, rappelons que `static` signifie que l'élément n'est accessible que dans son fichier source.

Déclaration de classe anticipée

Tout comme vous pouvez déclarer une structure à un endroit et la définir complètement ailleurs, vous avez la possibilité de le faire pour des classes. La syntaxe est simple : `class nom_de_classe;`. Imaginons qu'une classe `A` contienne un objet de classe `B` et réciproquement. Il faut recourir à une déclaration pour que `A` ait connaissance de `B` avant que `B` ne soit définie :

```
class B;     // déclaration anticipée

class A
{
   protected:
        B    objet_B;
   public :      // ...
};

class B
{
   protected:
        A    objet_A;
   public :  // ...
};
```

2

Les constructeurs
et les destructeurs

La vie et la mort d'un objet C++ sont régies par les constructeurs et les destructeurs. Les fonctions constructeurs sont appelées à la création d'un objet, les fonctions destructeurs sont invoquées dès qu'il sort de sa portée de visibilité, c'est-à-dire dès qu'il meurt.

Les notions de base

L'idée

En C ou en C++, quand vous déclarez une variable sans l'initialiser, son contenu est indéterminé. En C++, quand vous déclarez un objet d'une certaine classe, son constructeur sera appelé *automatiquement.* Un constructeur est une fonction-membre qui porte *toujours* le nom de la classe dans laquelle elle est définie, et qui ne renvoit rien (pas même un void). Si vous ne définissez pas explicitement un constructeur pour une classe, le C++ en utilise un d'office, qui va se contenter de réserver de la mémoire pour toutes les variables de votre classe.

Attention : quand vous définissez un pointeur sur un objet, aucun constructeur n'est appelé. Si vous désirez ensuite allouer de la mémoire et appeler un constructeur, utilisez new (voir chapitre 6, page 83).

Parallèlement aux constructeurs, le C++ nous propose les *destructeurs.* Quand l'objet sort de la portée du bloc dans lequel il a été déclaré, notamment quand l'exécution du programme atteint la fin d'un bloc où l'objet est défini, son destructeur est appelé *automatiquement.*

Mise en œuvre C++ *constructeurs*

Comment définir une fonction constructeur ? Nous savons qu'elle doit porter le même nom que sa classe. Prenons l'exemple d'une classe `Equipage` :

```
class Equipage
{
   private :
        int      personnel;
   public :
        Equipage(int personnel_initial);
};

Equipage::Equipage(int personnel_initial)
{
   personnel = personnel_initial;
}
```

Notre constructeur de la classe `Equipage` ne retourne aucune valeur, pas même un `void`, puisque c'est un constructeur. Il accepte un paramètre, le nombre de personnes dans l'équipage. Cela signifie que lorsque vous voudrez déclarer un objet de classe `Equipage`, il faudra faire suivre l'identifiant de l'objet par les paramètres effectifs du constructeur, entre parenthèses.
À l'exécution du programme suivant, le constructeur `Equipage : : Equipage` est appelé automatiquement, avec comme paramètre l'entier `311` :

Ici, le constructeur est appelé, ce qui initialise la donnée-membre personnel à 311

```
void        main()
{
   Equipage      base_alpha(311);
   // déclaration de l'objet et appel du constructeur
}
```

Constructeur sans paramètre

Si vous définissez un constructeur sans paramètre, par exemple comme ceci :

```
Equipage::Equipage()
{
   nombre = 0;
}
```

il faut faire attention à la manière de l'appeler. L'exemple suivant vous montre ce qu'il faut et ne faut pas faire :

```
void        main()
{
   Equipage     mon_equipage();       // (1) incorrect
   Equipage     mon_autre_equipage;   // (2) correct
}
```

Si vous ajoutez des parenthèses, comme dans le cas 1, le compilateur comprend que vous déclarez une fonction ! Donc, pour appeler correctement un constructeur sans paramètre, il faut omettre les parenthèses, comme dans le cas 2. C'est seulement dans ce cas que le constructeur sans paramètre est appelé.

Constructeurs surchargés

Il est tout à fait possible de définir plusieurs constructeurs homonymes, qui se distingueront alors par le type et/ou la quantité de paramètres. Reprenons la classe Equipage et définissons trois constructeurs :

```
class Equipage
{
   private :
        int      personnel;

   public :
        Equipage(int personnel_initial);    // 1
        Equipage();                         // 2
        Equipage(int nbr_femmes,            // 3
                 int nbr_hommes);
};

// constructeur 1
Equipage::Equipage(int personnel_initial)
{
   personnel = personnel_initial;
}

// constructeur 2
Equipage::Equipage()
{
   personnel = 0;
```

```
}

// constructeur 3
Equipage::Equipage(int nbr_femmes, int nbr_hommes)
{
   personnel = nbr_femmes + nbr_hommes;
}
```

Vous disposez maintenant de trois manières d'initialiser un objet de classe Equipage. L'exemple qui suit illustre ce concept :

```
void        main()
{
   Equipage base_alpha(311);  // appel au constructeur 1
   Equipage mixte(20, 1);     // appel au constructeur 3
   Equipage inconnu;          // appel au constructeur 2
}
```

Mise en œuvre C++ *destructeurs*

Rappelons que le destructeur est appelé *automatiquement* quand un objet sort de la portée du bloc dans lequel il est déclaré — c'est-à-dire quand il n'est plus accessible au programmeur. Une fonction-membre destructeur ne retourne pas de type (pas même un void), et n'accepte *aucun paramètre*. Comme pour les constructeurs, le compilateur génère un destructeur par défaut pour toutes les classes qui n'en sont pas pourvues. Ce destructeur par défaut se borne à libérer la mémoire des données de la classe. Mais si vous désirez faire une opération particulière pour détruire un objet, et notamment si vous avez des pointeurs dont il faut libérer l'espace mémoire, vous pouvez définir votre propre destructeur.

Attention ! Contrairement aux constructeurs, il ne peut exister *qu'un seul* destructeur par classe !

Le nom de la fonction destructeur est de la forme « `-NomDeClasse` ». Le premier signe est un « tilde ».

Exemple

Imaginons une classe qui contienne une donnée-membre pointant vers une chaîne de caractères :

```
#include <stdio.h>
#include <string.h>

class Capitaine
```

```
{
   private :
        char *nom;

   public :
        // constructeur
        Capitaine) char *nom_initial );
        // destructeur
        -Capitaine();
};

// constructeur
Capitaine::Capitaine( char *nom_initial )
{
   nom = (char*) malloc(strlen(nom_initial) + 1);
   strcpy(nom, nom_initial);
}

// destructeur
Capitaine::-Capitaine()
{
   free( nom );
}
```

Compléments

Constructeur copie

Lorsqu'il faut initialiser un objet *avec un autre objet* de même classe, le *constructeur copie* est appelé automatiquement. Voici un petit exemple de code appelant le constructeur copie de la classe Beta :

```
class   Beta

   // le contenu ne nous intéresse pas ici


void    fonction_interessante(Beta beta)

   // traitement sur beta


void        main()
```

```
    Beta     premier,
             deuxième (premier),        // (1)
             troisième = premier;       // (2)

    fonction_interessante(premier);     // (3)
}
```

* voir le paragraphe sur les listes d'initialisations, page 47,

* Dans ce cas, l'opérateur = n'est pas appelé puisque ce n'est pas une affectation.

- La ligne (1) signifie que l'objet deuxième doit être initialisé avec l'objet premier*.
- La ligne (2) initialise elle aussi l'objet troisième avec l'objet premier. Ce n'est pas une affectation, c'est une initialisation !*
- Enfin, la ligne (3) fait appel à une fonction qui va recopier le paramètre effectif, premier, dans son paramètre formel, beta.

Bref, ces trois lignes font *implicitement* appel au constructeur copie. Comme pour les constructeurs ordinaires ou les destructeurs, le C++ en génère un par défaut si vous ne l'avez pas fait vous-même. Ce constructeur copie par défaut effectue une copie « bête », membre à membre, des données de la classe. Dans les cas où vos classes contiennent des pointeurs, cela pose un problème : en ne copiant que le pointeur, l'objet copié et l'objet copiant pointeraient sur la même zone mémoire. Et ce serait fâcheux ! Cela signifierait qu'en modifiant l'un des deux objets, l'autre serait également modifié de manière « invisible » !
Le schéma suivant illustre ce problème.

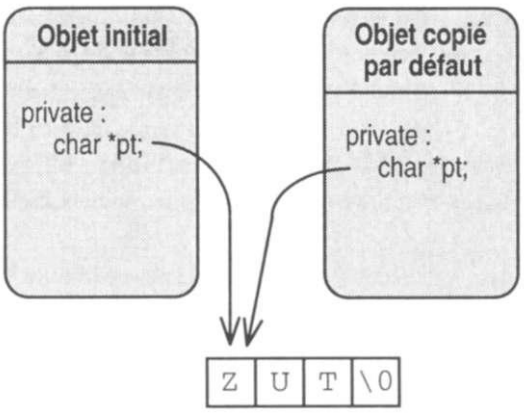


Schéma illustrant le problème d'une copie membre à membre

Pour résoudre ce problème, vous devez donc écrire votre propre constructeur copie, qui effectuera une copie propre — avec, par exemple, une nouvelle allocation pour chaque pointeur de l'objet copié. Nous obtenons le schéma suivant, après une copie « propre ».

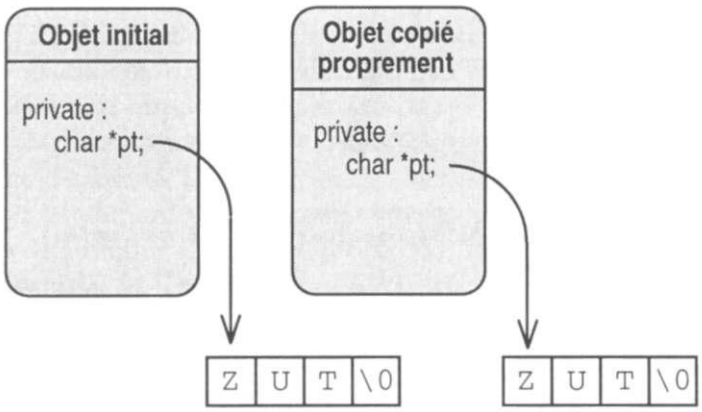


Schéma illustrant le résultat d'une copie propre.

Comment définir son propre constructeur copie ?

La forme d'un constructeur copie est toujours la même. Si votre classe s'appelle `Gogol`, son constructeur copie s'appellera `Gogol::Gogol(const Gogol&)`. Le paramètre est une *référence** à un objet de classe `Gogol`. Ici `const` n'est pas nécessaire mais fortement recommandé, puisque vous ne modifierez pas l'objet à copier.

* les références sont traitées au chapitre 9, page 113.

Écrivons la classe `Gogol`, possédant un pointeur de `char`, et efforçons-nous d'écrire son constructeur de copie :

```
#include <string.h>
#include <iostream.h>      // pour cout*

class Gogol
{
    private:
            char    *pt;

    public:
           // constructeur normal
           Gogol(char *c);

           // constructeur de copie
           Gogol(const Gogol &a_copier);
```

*cout est un mot clé C++ utilisé pour l'affichage de caractères. Voir chaptre 7, page 89.

```
            // accès aux données membre
            void set_pt(char *c);
            char *get_pt();

            // autres fonctions
            void Afficher();:
>;

// constructeur normal
Gogol::Gogol(char *c)
{
    pt = new char [strlen(c) + 1];
    strcpy(pt, c);
}

// constructeur de copie
Gogol::Gogol(const Gogol &a_copier)
{
    if (this != &a_copier) // voir dans la marge
    {
        // effaçons l'ancien pt
        delete pt;

        // allouons de la place pour le nouveau pt
        pt = new char [strlen(a_copier.pt) + 1];

        // donnons une valeur au nouveau pt
        strcpy(pt, a_copier.pt);
    }
}

void Gogol::set_pt(char *c)
{
    delete pt;
    pt = new char [strlen(c) + 1];
    strcpy(pt, c);
}

char *Gogol: :get_pt()
{
    return pt;
}

void Gogol::Afficher()
{
    cout << pt << endl;
}

void    maint)
{
    Gogol   gog("Zut"),
            bis = gog;  // appel du constructeur copie

    gog.Afficher();  // Zut
    bis.Afficher();  // Zut
```

Il faut vérifier que l'objet source et l'objet destination sont différents, car s'ils étaient identiques, delete pt effacerait à la fois les pointeurs source et destination !

```
    // on modifie la chaine de gog
    gog.set_pt("Mince  alors");

    gog.Afficher();  // Mince alors
    bis.Afficher();  // Zut
}
```

Conclusion : les objets `gog` et `bis` possèdent bien deux pointeurs désignant deux zones mémoires différentes. C'est parfait. Mais si nous n'avions pas écrit notre propre constructeur copie, celui généré par défaut aurait copié les données membre à membre. Ainsi, dans `gog` et `bis, pt` aurait été le même. Donc, le fait de détruire le pointeur de `gog` aurait également détruit celui de `bis` par effet de bord. Inutile de préciser les conséquences désastreuses d'un tel événement.

Constructeurs de conversion

Un constructeur n'ayant qu'un seul argument de type `T` spécifie une conversion d'une variable de type `T` vers un objet de la classe du constructeur. Ce constructeur sera appelé automatiquement chaque fois qu'un objet de type T sera rencontré là où il faut un objet de la classe de constructeur.
Par exemple, imaginons qu'on veuille une conversion automatique d'un entier vers un objet de classe `Prisonnier,` où l'entier représente son numéro matricule. Ainsi, quand le compilateur attend un objet `Prisonnier` et qu'il n'a qu'un entier à la place, il effectue la conversion automatiquement en créant l'objet `Prisonnier` à l'aide du constructeur de conversion. Exemple :

Si vous avez besoin de l'opération réciproque (convertir un objet de votre classe vers une variable de type T), allez voir page 61 où l'on vous dira que c'est possible.

```
#include <iostream.h>

class Prisonnier
{
   protected:
        int  numéro;
        char nom[50];
   public:
        //  ...
        Prisonnier(int n)  : numéro(n)
             { cout << "Conversion de " << n << endl;
               nom[0] = 0 ; }
};

void    fonction(Prisonnier p)
{
        // ...
```

```
}

void    main()
{

        Prisonnier    p = 2 ;    // donne p = Prisonnier(2)

        fonction(6); // donne fonction(Prisonnier(6))
}
// affiche :
// Conversion de 2
// Conversion de 6
```

L'héritage simple

L'héritage vous permet de modéliser le monde réel avec souplesse, d'organiser vos classes et de réutiliser celles qui existent déjà. C'est un concept fondamental en programmation orientée-objets. Nous parlons ici d'héritage simple, par opposition à l'héritage multiple qui sera développé dans la deuxième partie du livre.

Notions de base

L'idée

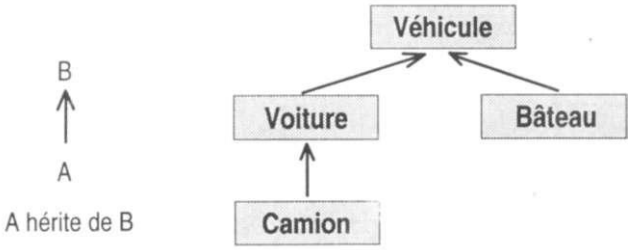
Les informaticiens se feraient beaucoup moins de soucis s'ils pouvaient régulièrement réutiliser des portions de code qu'ils ont déjà écrit pour des projets antérieurs. Grâce à l'héritage, une part du rêve devient réalité : vous pouvez profiter des classes déjà existantes pour en créer d'autres. Mais là où l'héritage est intéressant, c'est que vous pouvez ne garder qu'une partie de la classe initiale, modifier certaines de ses fonctions ou en ajouter d'autres.

L'héritage s'avère très adapté aux cas où vous devez manipuler des objets qui ont des points communs, mais qui diffèrent légèrement.

Le principe

Le principe initial de l'héritage est proche de celui de la classification en catégories : on part du concept le plus général pour se spécialiser dans les cas particulier. Admettons que nous voulions parler de véhicules*, et essayons de tisser des liens d'héritage entre quelques véhicules bien connus :

* Nous aurions pu parler de la classification des cornichons, des jeunes filles ou des flippers, mais un exemple classique est sans doute mieux adapté à un large public.



Le sens des flèches d'héritage pourrait être : « possède les caractéristiques de » ou « est un », au sens large du terme. Ainsi, *voiture* est un *véhicule.* De même pour le *bateau.* Le *camion,* lui, possède les caractéritiques de *voiture* et, indirectement, de *véhicule.*

On dit que voiture *hérite* de véhicule. De même, bateau hérite de véhicule, et camion de voiture.

D'un cas général, représenté par *véhicule,* nous avons tiré deux cas particuliers : *voiture* et *bateau. Camion* est lui-même un « cas particulier » de voiture, car il en possède *grosso modo* toutes les caractéristiques, et en ajoute d'autres qui lui sont propres.

Mise en œuvre C++

Dans l'exemple ci-dessus, chaque élément du schéma est représenté par une classe. Une remarque de vocabulaire : quand une classe A hérite d'une classe B, on dit que A est la classe *de base* et B la classe *dérivée.* Pour y voir plus clair, écrivons les classes véhicule et voiture, et examinons ensuite ce que cela signifie au niveau du C++ :

Le mot clé protected est expliqué dans les pages qui suivent.

```
class Véhicule
{
   protected :
        int      nombre_de_places;
        // ...
```

```
    public :
         // constructeurs
         Véhicule();
         // ...
         // fonctions de traitement
         void     Afficher));
         // . . .
};
```

Remarque : nous n'avons pas détaillé ces classes afin de nous concentrer sur le mécanisme d'héritage. Vous retrouverez les classes complètes en fin de chapitre.

```
class Voiture : public Véhicule
{
    protected :
         int  chevaux_fiscaux;
         // . . .
    public :
         // constructeurs
         Voiture();
         // fonctions de traitement
         void     CalculVignette();
};
```

Rien ne distingue ces classes d'autres classes ordinaires, si ce n'est la chaîne « `: public Véhicule` » à la suite du nom de la classe `Voiture`. C'est bel et bien cette chaîne qui indique au compilateur que la classe `Voiture` *hérite* de la classe `Véhicule`. Le mot-clé `public` qualifie la *spécification d'accès* de l'héritage. Nous verrons ci-dessous que le type d'héritage détermine quelles données de la classe de base sont accessibles dans la classe dérivée.

Voilà, nous avons déclaré que la classe `Voiture` héritait de la classe `Véhicule`. Que cela signifie-t-il concrètement ? Que dans des objets de classe dérivée (`Voiture`), vous pouvez accéder à des données ou fonctions membres d'objets de la classe de base (`Véhicule`).

Accès aux membres de la classe de base

* Rappelons que la spécification d'accès est déterminée par le mot-clé qui suit les deux-points («:») après la ligne class *NomClasseDérivée*.

Pour savoir précisément quelles données sont accessibles dans la classe dérivée, il faut considérer la spécification d'accès* ainsi que le type des données (`public`, `protected` ou `private`) de la classe de base. Le tableau suivant donne les combinaisons possibles :

type d'une donnée dans la classe de base ▼	spécification d'accès : public	protected	private
public	public	protected	private
protected	protected	protected	private
private	inaccessible	inaccessible	inaccessible

Le contenu du tableau indique le type du membre (public, protected, private ou inaccessible) dans la classe dérivée.

Autrement dit, le mot-clé protected est identique à private si vous n'utilisez pas l'héritage.

Vous découvrez dans ce tableau un nouveau spécificateur d'accès : protected. Ce spécificateur est identique à private à l'intérieur de la classe. Il diffère uniquement en cas d'héritage, pour déterminer si les éléments protected sont accessibles ou non dans les classes dérivées.

Pour mieux comprendre ce tableau, voici trois petits exemples de dérivation :

```
class Base
{
   private :
        int      detective_prive;
   protected :
        int      acces_protege;
   public :
        int      domaine_publicp±e;
};

class Deriveel : public Base
{
    // detective_prive est inaccessible ici
    // acces_protege est considéré comme « protected »
    // domaine_public est considéré comme « public »
};

class Derivee2 : protected Base
{
    // detective_prive est inaccessible ici
    // acces_protege est considéré comme « protected »
    // domaine_public est considéré comme « protected »
>;

class Derivee2 : private Base
{
    // detective_prive est inaccessible ici
    // acces_protege est considéré comme « private »
    // domaine_public est considéré comme « private »
};
```

Ces exemples amènent plusieurs remarques :

- Une donnée ou fonction membre `private` est *toujours* inaccessible dans ses classes dérivées
- La spécification d'accès d'une donnée (`public`, `protected` ou `private`) dans une classe dérivée est en fait la spécification d'accès la plus forte. Par exemple, si une fonction-membre est `protected` dans la classe de base, et que l'héritage est `public`, elle devient `protected` dans la classe dérivée.
- En règle générale, on utilise l'héritage `public` pour modéliser des relations du monde « réel ». Voir le chapitre 15 sur les conseils.

Redéfinition d'une fonction-membre

Vous pouvez définir des fonctions aux entêtes identiques dans différentes classes apparentées par une relation d'héritage. La fonction appelée dépendra de la classe de l'objet qui l'appelle, ce qui est déterminé statiquement à la compilation. Reprenons : une fonction-membre d'une classe dérivée peut donc avoir le même nom (et les *mêmes* paramètres) que celle d'une classe de base.
Prenons l'exemple d'une gestion de textes qui distingue les textes bruts des textes mis en forme, et définissons deux fonctions aux entêtes identiques (mais au corps différent) :

```
#include <stdio.h>

class TexteBrut

   public:
        void     Imprimer(int nb);

void    TexteBrut::Imprimer(int nb)

   printf("Imprimer de la classe TexteBrutXn");

class TexteMisEnForme : public TexteBrut    // héritage

   public:
        void     Imprimer(int nb);

void    TexteMisEnForme::Imprimer(int nb)

   printf("Imprimer de la classe TexteMisEnFormeXn");
```

```
void    main()
{
   TexteBrut         txt;
   TexteMisEnForme   joli_txt;

   txt.Imprimer(1);
   joli_txt.Imprimer(1);
}

// affichage :
Imprimer de la classe TexteBrut
Imprimer de la classe TexteMisEnForme
```

Conversion de Dérivée* vers Base*

Dans un cas où une classe `Dérivée` hérite publiquement d'une classe `Base`, les conversions de pointeurs de `Dérivée` vers des pointeurs de `Base` est faite automatiquement aux endroits où le compilateur attend un pointeur sur `Base` :

```
class Base
{ >;

class Dérivée : public Base
{ };

void    main()
{
   Base     *base;
   Dérivée  *derivee;

   base = dérivée;   // OK
}
```

Attention : cette conversion n'est pas autorisée si l'héritage est `protected` ou `private` ! Si vous remplacez `public` par `protected` dans le listing ci-dessus, la dernière ligne du `main` provoquera une erreur de compilation. Pourquoi ? Parce que pour qu'un pointeur sur une classe dérivée puisse être converti sur une classe de base, il faut que cette dernière soit accessible, c'est-à-dire qu'elle possède des membres `public` accessibles. Or ce n'est pas le cas d'un héritage `protected` ou `private`.

Résumé

En **C++,** une classe peut hériter d'une autre. On dit alors que la classe qui hérite est une classe *dérivée,* l'autre classe étant appelée classe *de base.* Quand une classe A hérite d'une classe B, elle peut accéder à certaines données ou fonctions

membres de la classe B. C'est un petit peu comme si une partie de la classe B avait été recopiée dans le corps de la classe A.
Pour savoir ce qui est accessible dans A, il faut considérer la spécification d'accès (public, protected ou private), le type d'accès des données de B (public, protected ou private) et consulter le tableau de la page 40.

L'intérêt par rapport au C

Là, le C est complètement battu. Grâce à l'héritage, vous allez enfin pouvoir réutiliser facilement le code que vous avez déjà écrit. Vous pourrez enfin « factoriser » les points-communs d'une structure d'objets. Il n'existe rien, dans le langage C, qui permette de mettre en œuvre facilement un mécanisme similaire.
Prenons un exemple : vous devez gérer les employés d'une centrale nucléaire. À partir d'une classe de base Employé, vous faites hériter d'autres classes correspondant aux catégories de personnel : Ouvrier, Cadre, AgentSécurité, etc. Si à l'avenir d'autres postes sont créés (par exemple PorteParole ou Avocat), vous pourrez créer la classe correspondante et la faire hériter de Employé ou de classes spécifiques. Dans notre exemple, un PorteParole pourrait hériter de la classe Cadre. L'avantage : vous n'avez pas à ré-écrire tout ce que les portes-paroles et les cadres ont en communs.

Compléments

Héritage et constructeurs

Les constructeurs ne sont jamais hérités. Quand vous déclarez un objet d'une classe dérivée, les constructeurs par défaut des classes de bases vont être appelés automatiquement avant que le constructeur spécifique de la classe dérivée ne le soit. Voici un exemple :

```
class Base
{
   public :
        Base();
```

```
};

class Deriveel : public Base
{
   public :
        Deriveel();
};

class Derivee2 : public Deriveel
{
   public :
        Derivee2();
};

void        main()
{
   Base     b;   // appel à Base()
   Deriveel dl;  // appels à Base() puis à Deriveel()
   Derivee2 d2;  // appels à Base() , Deriveel()...
                 // puis Derivee2()
   // corps de la fonction
}
```

Pour ne pas appeler les constructeurs par défauts mais des constructeurs avec des paramètres, vous devez recourir aux listes d'initialisations, expliquées plus loin dans ce chapitre.

Héritage et destructeurs

Alors que les constructeurs sont appelés dans l'ordre descendant (de la classe de base à la classe dérivée), c'est l'inverse pour les destructeurs : celui de la classe dérivée est appelé en premier, puis celui de la classe de base immédiatement supérieure, et ainsi de suite jusqu'à la classe de base la plus haute. Complétons l'exemple précédent :

```
void        main()
{
   Base     b;   // appel à Base()
   Deriveel dl;  // appels à Base{) puis à DeriveelO
   Derivee2 d2;  // appels à Base(), Deriveel()...
                 //  ...puis Derivee2()

   // corps de la fonction

}
// mort de b : appel à -Base()
// trépas de dl : appels à -Deriveel() puis -Base()
// éradication de d2 : appels à ~Derivee2(),
//                     -Deriveel() puis -Base()
```

Exemple complet

Reprenons notre problème de véhicules. Voici le code complet des classes `Véhicule` et `Voiture`, et des exemples d'accès aux données de la classe de base, à savoir `Véhiculé`.

```
class Véhicule
{
   private :
        char     buffer_interne[128];
             // uniquement utilisé par les fonctions de
             // la classe Véhicule. Inaccessible aux
             // classes dérivées.

   protected :
        char     nom_vehicule[20];
        int      nombre_de_places;
             // données protégées, transmises aux classes
             // dérivées mais toujours inaccessibles
             //à l'extérieur de l'objet

   public :
             // fonctions qui resteront publiques
             // dans les classes dérivées (à part les
             // constructeurs)

        // constructeur
        Véhicule(char *nom, int places);

        // fonctions d'accès aux données membre
        char     *get_nom_vehicule();
        int      get_nombre_de_places();
        void     set_nom_vehicule(char *nom);
        int      set_nombre_de_place(int p);

        // fonctions de traitement
        void     Afficher();;
};

// définition des fonctions

Véhicule::Véhicule(char *nom, int places)
{
   sprintf(nom_vehicule, "%20s", nom);
   // on écrit dans la chaine nom_vehicule les 20
   // premiers caractères de la chaine nom.
   nombre_de_places = places;
}

char         *Vehicule::get_nom_vehicule()
{
   return nom_vehicule;
}

int     Véhicule::qet nombre de places!)
{
   return nombre_de_places;
```

```
}

void        Véhicule::set nom„vehicule(char *nom)
{
    sprintf(nom_vehicule, "%20s", nom);
    // on écrit dans la chaine nom_vehicule les 20
    // premiers caractères de la chaine nom.
}

int     Véhicule::set_nombre_de_place(int p)
{
    nombre_de_places = p;
}

// ________ Voici maintenant la classe dérivée ________

class Voiture : public Véhicule
{
   private :
        float    SavantCalcul ();
   protected :
        int  chevaux_fiscaux;
   public :
        // constructeurs
        Voiture(char *nom, int places, int chevaux);

        // fonctions d'accès
        int get_chevaux_fiscaux();
        void set_chevaux_fiscaux(int ch);

        // fonctions de traitement
        float    CalculVignette();
};

Voiture::Voiture(char *nom, int places, int chevaux)
{
    sprintf(nom_vehicule, "%20s", nom);
    nombre_de_places = places;
    chevaux_fiscaux = chevaux;
}

int Voiture : :get_chevaux_fiscaux()
{
    return chevaux_fiscaux;
}

void    Voiture::set_chevaux_fiscaux(int ch)
{
    chevaux_fiscaux = ch;
}

float Voiture::SavantCalcul(int code)
{
    float    savant_calcul;

    // on fait un savant calcul avec "code"
    return savant_calcul;
```

```
}

float   Voiture::CalculVignette()
{
    int code;

    if (chevaux_fiscaux < 2)
         code = 0;
    else
         code = chevaux_fiscaux - 1;
    return (SavantCalcul(code));
}
```

La fonction CalculVignette fait n'importe quoi, soyez certain que l'auteur en a parfaitement conscience.

Listes d'initialisations

Les lecteurs les plus alertes ne manqueront pas de constater quelques lourdeurs dans l'exemple précédent. Examinez les constructeurs de la classe `Voiture`. Examinez maintenant les constructeurs de la classe `Véhicule`. Je récapitule pour vous les deux constructeurs ci-dessous. Ne remarquez-vous pas une certaine ressemblance ?

```
Véhicule::Vehicule(char *nom, int places)
{
    sprintf(nom„vehicule, "%20s", nom);
    nombre_de_places = places;
}

Voiture::Voiture(char *nom, int places, int chevaux)
{
    sprintf(nom_vehicule, "%20s", nom);
    nombre de_places = places;
    chevaux_fiscaux = chevaux;
}
```

Oh, mais c'est bien sûr ! Les deux premières lignes sont identiques ! N'est-ce pas là le signe provocateur d'une redondance inutile et dangereuse propre à susciter une colère aussi vive que justifiée ?
Le problème vient du fait que les constructeurs (ainsi que les destructeurs, mais ce n'est pas le propos ici) ne sont pas hérités. Cela signifie que, quelque part dans le constructeur de `Voiture`, vous devez initialiser les variables-membres définies dans `Véhicule`, et dont `Voiture` a hérité. Ici, il s'agit des variables `nom` et `places`. Le problème est en fait simple : comment faire pour appeler un constructeur de la classe de Base, depuis un constructeur d'une classe dérivée ? Deux solutions :

- Faire comme dans l'exemple, c'est-à-dire recopier les lignes de code du constructeur de la classe de base qui vous intéresse. C'est très laid : dupliquer bêtement du code, c'est alourdir le programme, et si jamais ce code doit être modifié, vous devrez retrouver tous les endroits où vous l'avez recopié. Imaginez un peu ce que cela peut donner dans une hiérarchie de plusieurs centaines de classes !
- La meilleure solution : utiliser une ***liste d'initialisation.*** Qu'est-ce donc ? C'est un moyen pratique d'appeler directement des constructeurs pour des données-membre qu'il faut initialiser. Voici un exemple d'utilisation de la liste d'initialisation :

Vous êtes en train de paniquer ? Vous avez l'impression d'être noyé sous les nouveautés syntaxiques du C++ ? Ne tremblez plus, car l'aide-mémoire C++ est arrivé ! Il est joli et il vous attend sur le rabat de la couverture !

```
// constructeur de la classe Dérivée, amélioré
Voiture::Voiture(char *nom, int places, int chevaux)
     : Véhicule(nom, places)
{
     chevaux_fiscaux = chevaux;
}
```

Juste après les paramètres du constructeur, tapez le caractère : (deux points) et appelez ensuite les constructeurs des classes de bases que vous avez choisis, avec leurs paramètres bien entendu. Si vous appelez plusieurs constructeurs, séparez-les par une virgule.
Dans l'exemple ci-dessus, les paramètres nom et places utilisés pour appeler le constructeur de Véhicule sont ceux que vous avez passés au constructeur de Voiture.
Vous pouvez également utiliser une liste d'initialisation pour vos variables simples (entier, caractère, etc.) Exemple :

```
// constructeur de la classe Dérivée, encore amélioré
Voiture::Voiture(char *nom, int places, int chevaux)
  : Véhicule(nom, places), chevaux_fiscaux(chevaux)
{ }
```

Les différents éléments de la liste d'initialisation sont séparés par une virgule. Vous remarquez que maintenant, il n'y a plus d'instruction dans le corps de la fonction !
La liste d'initialisation est le seul moyen d'initialiser une donnée-membre spécifiée constante. Rappelons qu'une

constante ne peut être qu''*initialisée* (et jamais *affectée),* voir page 67. Un autre avantage des listes d'initialisation : elles ne créent pas d'objets temporaires comme une fonction ordinaire. Les objets sont initialisés directement, d'où gain de temps certain.

Résumé

Une liste d'initialisation peut appeler des constructeurs des classes de base ou initialiser des variables simples. Résumé de la syntaxe :

```
Classe::Classe(paramètres) : liste_ini
{
    // corps du constructeur de Classe
}
```

Où liste_ini contient un ou plusieurs éléments suivants, séparés par des virgules :

```
ConstructeurClasseDeBase(paramètres) ou
Variable(valeur_initiale)
```

4

La surcharge

Le C++ offre aux programmeurs la possibilité de définir des fonctions homonymes. Ce principe est appelé « surcharge de fonction ».

Notions de base

L'idée Écrire deux fonctions qui portent le même nom est une chose impensable en langage C. Pas en C++. Imaginez que vous vouliez écrire une fonction qui trie des tableaux *d'entiers*. L'entête de cette fonction ressemblerait à ceci :

```
void        tri(int tab[], int taille_tableau);
```

Si par la suite vous avez besoin d'une fonction qui trie des tableaux de *réels*, vous devez réécrire la fonction, en lui donnant un autre nom :

```
void        tri_reels(float tab[], int taille_tableau);
```

Si vous êtes logiques avec vous-même, vous devrez également changer le nom de la fonction initiale :

```
void        tri_entiers(int tab[], int taille_tableau);
```

Ce qui vous oblige à modifier tous les appels à cette fonction. Grâce au mécanisme de *surcharge de fonction*, vous pouvez donner le nom `tri` aux deux fonctions, le compilateur pouvant les différencier grâce aux types d'arguments.
Notez que le langage C propose déjà un type de surcharge pour les opérateurs arithmétiques : le mécanisme mis en œuvre dépend du type des objets sur lesquels on invoque l'opérateur.

Mise en œuvre C++

Reprenons notre exemple de tri, et appliquons-lui le principe de surcharge. Les deux entêtes deviennent :

```
void         tri(int   tab[], int taille_tableau);
void         trifloat tab[], int taille_tableau);
```

La seule différence étant le type du premier paramètre. Examinons maintenant les appels de fonctions suivants :

```
void main()
{
    float    tab_f[] = { 1.5, 1.2, 7.2, 0.07 } ;
    int      tab_i[] = { 3,   2,   1,   0 };

    tri(tab_f, 4);
    tri(tab_i, 4);
}
```

Le premier appel à `tri` fait automatiquement référence à la deuxième fonction `tri`, qui accepte un tableau de flottants en premier paramètre. Le deuxième appel à `tri` se fait naturellement avec la fonction `tri` qui s'occupe des entiers.

Comment le compilateur fait-il la différence entre plusieurs fonctions homonymes ?
Tout d'abord, seules les fonctions situées dans la portée de l'appel —c'est-à-dire « visibles »— sont prises en compte. Ensuite, les critères suivants sont examinés :

1. Le type ou le nombre de paramètres. Dès que deux fonctions homonymes diffèrent par le type ou le nombre de l'un ou de plusieurs de leurs paramètres, le compilateur reconnaît la bonne fonction selon les types réels passés à l'appel. C'est l'exemple que nous venons de voir.

Attention ! Le type de retour de la fonction n'est jamais pris en compte ! Par exemple, il est illégal de déclarer les deux fonctions suivantes :

```
int       truc(int);
float     truc(int);
```

Le compilateur ne pourrait faire la différence. En revanche, il est toujours possible d'écrire :

```
int       truc(int);
float     truc(float);
```

Puisque, cette fois, le type des paramètres est différent.

2. Si l'étape 1 ne permet pas d'identifier la fonction à appeler, le compilateur essaie de convertir les paramètres réels vers les paramètres des fonctions homonymes. Ainsi, un `char` ou un `short` seront convertis vers un `int`, les `float` seront convertis vers des `double`, les pointeurs de classes dérivées seront convertis en pointeurs de classes de base*, etc.

* Les classes de base et les classes dérivées seront expliquées au chaprtre 3, sur l'héritage.

En cas d'ambiguïté insoluble, le compilateur signale une erreur. De toutes manières, mieux vaut éviter les cas ambigus qui sont source d'affres éternelles.

L'intérêt par rapport au C

On pourrait croire que les fonctions homonymes rendent les programmes moins lisibles et plus confus. Certes, si vous définissez plusieurs fonctions « traite_donnees », qui font des choses totalement différentes, le gain en clarté sera plutôt discutable.

En revanche, dans les cas où vous disposez d'un jeu de fonctions qui fait pratiquement la même chose, mais dont certains paramètres diffèrent, vous avez intérêt à utiliser la surcharge. Bref, chaque fois que le *sens* des fonctions est sensiblement identique, donnez-leur le même nom.

Dans tous les cas, le langage C vous oblige à donner des noms différents, simplement pour que le compilateur reconnaisse ses petits. Remarquons que dans l'exemple du tri,

nous pourrons aussi nous servir du mécanisme de la généricité (mis en œuvre par les *templates),* exposé au chapitre 10.

Compléments

Surcharge d'opérateurs

En C++, les opérateurs courants (+, -, ==, etc.) peuvent être surchargés. Comment est-ce possible ? En fait, quand vous utilisez un opérateur *op* sur un objet, le compilateur génère un appel à la fonction `operator` *op*. Les trois écritures suivantes sont équivalentes :

```
objet_l = objet_2 + objet_3;
objet_l = operatort (objet2, objet3);        // (1)
objet_l = objet_2.(operator+(objet_3));      // (2)
```

« operator *op* », où *op* est un opérateur standard (ici +) est en réalité un nom de fonction comme un autre, qui est défini par défaut pour les types de base du C++. Donc, pour personnaliser le comportement des opérateurs, il vous suffit de surcharger la fonction « operator *op* » correspondante, comme vous savez déjà le faire pour vos propres fonctions.
Attention ! Quand vous surchargez un opérateur, il faut veiller à choisir l'une ou l'autre des deux méthodes de l'exemple ci-dessus :

- Dans le cas (1), il s'agit de la fonction `operator+` globale, définie en dehors de toute classe. Elle prend deux paramètres, correspondant aux deux membres de l'addition.
- Dans le cas (2), il s'agit de la fonction-membre `operator+,` définie dans la classe d'`objet_2`, qui suppose que l'objet sur lequel elle est appelée est le premier paramètre de l'opération d'addition (ici `objet_2`).

Si vous utilisiez à la fois ces deux méthodes de surcharge pour la même fonction, le compilateur ne pourrait pas faire la différence.

Vous pouvez surcharger les opérateurs suivants :

```
+    -    *    /    %    ^    &         ~    !    =    <    >
+=   -=   *=   /=   %=   ^=   &=   |=   <<   >>   >>=  <<=  ==
!=   <=   >=   &&   ||   ++   --   ,    ->*  ->   ()   []
```

Voyons l'exemple d'un programme complet redéfinissant l'opérateur + pour des objets de classe Color :

```
#include <stdio.h>

class Color
{
    private :
            int     rouge;
            int     vert;
            int     bleu;
    public :
             // constructeurs (voir chapitre 2)
           Color(int r, int v, int b);
           Color();
           void    Afficher));
           Color   operator+ (Color Sdroite);
};

Color;:Color()
{
    rouge = vert = bleu = 0;
}

Color::Color(int r, int v, int b)
{
    rouge = r;
    vert = v;
    bleu = b;
)

void    Color::Afficher()
{
    printf("rouge=%d vert=%d bleu=%d\n",
        rouge, vert, bleu);
}

Color   Color::operator+ (Color Sdroite)
    // droite est passé par « référence »*
{
     // fonction redéfinissant l'opérateur + :
     // RESULTAT = A + B
     //A est l'objet Color courant
     // B est le paramètre "droite"
     // RESULTAT est la valeur retournée

     Color    résultat) (rouge + droite.rouge) / 2,
                        (vert + droite.vert) / 2,
                        (bleu + droite.bleu) / 2);
```

* Le passage par référence est une exclusivité C++ ! Vous en saurez plus en lisant le chapitre 9, page 113.

```
    return résultat;
}

void    main()
{
    Color   rouge_pur(10, 0, 0),
            bleu_pur(0, 0, 10),
            mixage;

    mixage = rouge_pur + bleu_pur;
//  cette ligne éqruivaut à :
//  mixage = rouge_pur.operator+ (bleu_pur)

    mixage.Afficher();
    // affiche "rouge=5 vert=0 bleu=5"
}
```

Plusieurs remarques :

- Le premier paramètre d'une fonction opérateur (`Color &droite` dans l'exemple) est passé par *référence* (voir chapitre 9, page 113). En deux mots : quand vous passez un paramètre par référence, cela équivaut à passer l'objet *original.* Ainsi, modifier un paramètre passé par référence revient à modifier le paramètre réel, utilisé à l'appel de la fonction.
- Un opérateur retourne un objet de la même classe que ses paramètres (ici, un objet de classe `Color`). Vous devrez donc créer un objet de cette classe dans la fonction opérateur, l'initialiser en fonction des deux paramètres de l'opérateur, et le retourner.

Pourquoi passer droite par référence dans operator+ ?

Expliquons plutôt pourquoi un passage par *valeur* habituel serait problématique. Admettons que nous ayons défini `operator+` comme suit :

```
Color   Color::operator+ (Color droite)
    // droite est passé par valeur
{
    // ...
}
```

Le paramètre `droite` étant passé par valeur, une copie s'opère entre le paramètre effectif (l'objet `bleu_pur`) et le paramètre formel utilisé dans la fonction (l'objet `droite`).

Dans ce cas, la copie ne pose pas de problème, mais si, par exemple, votre classe comportait un pointeur alloué, la copie d'objet par défaut serait mauvaise.

Avec un passage par *référence*, il n'y a pas de copie, puisque la fonction `operator+` agit directement sur le paramètre effectif, `bleu_pur`. En d'autres termes, manipuler l'objet `droite` équivaut à manipuler l'objet `bleu_pur`.

Récapitulons :

- Soit vous utilisez un passage par référence et vous n'avez pas besoin de vous soucier de la copie de votre objet,
- Soit vous utilisez un passage par valeur et vous devez vérifier que la copie s'effectue bien. Si nécessaire, définissez un constructeur de copie (voir page 31).

Inutile de préciser — mais je le fais quand même — que la première solution est plus intéressante que la seconde.

Surcharge de l'opérateur =

Pourquoi redéfinir l'opérateur = ?

L'affectation d'un objet, via l'opérateur =, est une opération sensible qui mérite qu'on s'y attarde. Si vous ne définissez pas d'opérateur = pour une classe, le compilateur en génère un par défaut qui effectue une affectation « bête », données membre à données membre. Bien que ce genre de comportement soit satisfaisant dans le cas de classes simples, il devient parfois dangereux si vous utilisez des pointeurs. Le problème est similaire à celui du constructeur-copie (voir page 31).

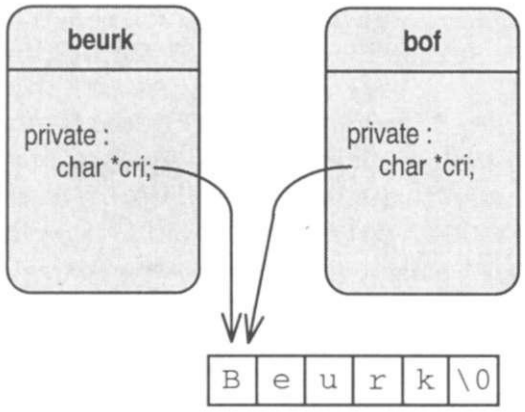
Considérons l'exemple suivant :

```
Sinclude <stdio.h>
tinclude <string.h>

class Exclamation
{
   private:
        char *cri;
   public:
        Exclamation(char *c)
         { cri = new char[strlen(c)+1]; strcpy(cri, c); }

        void set_cri(char *nc);

        void Affiche()
         { printf("%s\n", cri); )
};
```

Le constructeur Exclamation ainsi que la fonction Affiche sont définis « inline » (voir page 21). new et delete remplacent malloc et free (voir chapitre 6 page 83).

```
void    Exclamation::set_cri(char *nc)
{
   // vérifions si nous n'avons pas affaire aux
   // même pointeurs
   if (ne == cri)
        return;

   // libérons la mémoire de l'ancienne valeur de cri
   delete cri;

   // allouons de la mémoire pour la nouvelle valeur
   cri = new char[strlen(nc)+1];

   // copions le paramètre dans la donnée cri.
   strcpy(cri, ne);
}

void    main()
{
   Exclamation  beurk("Beurk"),
                bof("Bof");

   beurk.Affiche();  // affiche "Beurk"
   bof.Affiche();         // affiche "Bof"

   bof = beurk;       // operateur= défini par défaut

   beurk.Affiche();       // affiche "Beurk"
   bof.Affiche();         // affiche "Beurk"

   bof.set_cri("Ecoeurant");   //on modifie bof et beurk

   beurk.Affiche();       // affiche n'importe quoi
   bof.Affiche();         // affiche "Ecoeurant"
}
```

La copie par défaut a ici des effets désastreux : en copiant uniquement le pointeur `cri` et non *ce qui* est pointé, nous nous retrouvons avec deux objets dont la donnée `cri` pointe sur la même zone mémoire :



Le résultat de cette décadence ne se fait pas attendre : il suffit de modifier l'un des deux objets pour que son jumeau soit affecté. C'est ce que nous faisons dans la ligne `bof.set_cri("Ecoeurant");` Cette instruction libère l'ancienne valeur de `cri` en appelant `delete`. Or, cette ancienne valeur est également partagée par l'objet `beurk`. Sans le savoir, nous avons donc effacé deux cris au lieu d'un.

Comment redéfinir l'opérateur = ?

Ces constatations affreuses nous poussent à écrire *notre propre opérateur d'affectation,* qui va se charger de copier le contenu des pointeurs cris. Rappelons que l'usage de = appelle la fonction `opérâtor=`.

Voici l'exemple complet :

```
tinclude<stdio.h>
#include<string.h>

class Exclamation
{
   private:
        char *cri;
   public:
        Exclamation(char *c)
         { cri = new char[strlen(c)+1]; strcpy(cri, c); }

        void set_cri(char *nc);

        void Affiche()
         { printf("%s\n", cri); )

        Exclamation   &operator=(Exclamation &source);
};
```

```
void    Exclamation::set_cri(char *nc)
{
   if (ne == cri)
        return;
   delete cri;
   cri = new char(strlen(ne)+1];
   strcpy(cri, ne);
}
```

Cette fonction, qui retourne une référence, est expliquée à la suite du listing.

```
Exclamation &Exclamation::operator=(Exclamation Ssource)
{
   // évitons d'affecter deux objets identiques
   if (this == &source)
        return *this; // retournons l'objet courant

   delete cri;
   cri = new char[ strlen(source.cri)+1 ];
   strcpy(cri, source.cri);
   return *this;     // retournons l'objet courant
}

void    main()
{
   Exclamation  beurk("Beurk"), bof("Bof");

   beurk.Affiche();  // affiche "Beurk"
   bof.Affiche();    // affiche "Bof"

   bof = beurk;      // operateur= défini par nos soins

   beurk.Affiche();  // affiche "Beurk"
   bof.Affiche();    // affiche "Beurk"

   bof.set_cri("Ecoeurant");  // on ne modifie que bof

   beurk.Affiche();  // affiche "Beurk" : parfait !
   bof.Affiche();    // affiche "Ecoeurant"
}
```

Pourquoi 'return *this'?
L'opérateur = doit retourner un objet de type `Exclamation`. Plus exactement, il doit retourner l'objet sur lequel `operator+` a été appelé, après que l'affectation a été faite. En retournant `*this`, nous retournons bien l'objet courant que nous venons de modifier.
Mais pourquoi retourner une *référence* plutôt qu'un objet `Exclamation` tout simple? Pour répondre correctement au cas de figure suivant :

```
(bof = beurk).Affiche();
```

Si nous retournions une simple copie du résultat, l'opération Affiche ne s'effectuerait pas sur bof mais sur sa copie.

Pourquoi faire le test 'if (this == &source)' ?
Ce test a pour but de vérifier si nous ne voulons pas affecter l'objet lui-même (ce serait le cas de l'instruction a = a;). Si ce test n'était pas réalisé, nous nous retrouverions dans une situation très désagréable. Regardez un instant la première ligne après le test. Oui, c'est bien delete cri. Imaginez maintenant qu'un coquin ait écrit beurk = beurk. En faisant delete cri, nous effacerions le cri de l'objet source *mais aussi*, sans le vouloir, celui de l'objet destination. Dès lors, rien d'étonnant à ce que nous nous exposions à l'imprévu en accédant au cri de l'objet source.

Opérateurs de conversion de types

Pour autoriser et contrôler la conversion d'une classe A vers un type T quelconque, vous pouvez définir un opérateur de conversion de nom `operator T()`. Chaque fois qu'un objet de classe A doit être considéré comme une variable de type T, l'appel à cet opérateur se fera automatiquement. *Attention !* Ce genre d'opérateur *ne doit pas* avoir de type de retour, mais doit tout de même retourner une variable de type T (on évite ainsi la redondance du nom du type et des risques d'incohérence liés à cette redondance superflue).
Admettons que vous vouliez modéliser des articles dans un magasin. La classe `Article`, qui contient diverses informations, doit pouvoir être utilisée dans des expressions de calcul de prix. Il faut donc définir `operator float();`

L'opération réciproque, à savoir la conversion d'un type quelconque en un objet de la classe, est tout à fait possible ! Il suffit d'utiliser un constructeur spécial (voir page 35),

On ne devrait pas spécifier la taille du tableau directement dans la classe, mais plutôt utiliser une constante. Toutes mes excuses pour cette hérésie éhontée.

```
#include <string.h>
#include <stdio.h>

class Article
{
   protected:
        char     nom[50];
        float    prix;
        int      nb_disponible;
   public:
        Article(char *n, float p, int nb)
             : prix(p), nb_disponible(nb)
             { strcpy(nom, n); }
        operator float();
};
```

```
Article::operator float()
{
   return prix;
}
void    main()
{
   float    total;
   Article  b("Bonne bière", 12., 30);

   total = 5 * b;    // conversion de b en float
   printf("5 bonnes bières valent %.2f F",  total);
}
// affichage : 5 bonnes bières valent 60.00 F
```

Les petits + du C++

Pour conclure cette première partie, attardons-nous sur les petits changements qui rendent la vie du programmeur plus agréable. Contrairement aux autres chapitres, celui-ci ne se décompose pas en deux parties (notions de base et compléments). Chaque point est expliqué en une seule fois, sans distinction de « niveau de difficulté ».

Les commentaires

Vous êtes un programmeur consciencieux, vous commentez donc abondamment vos programmes. Le C++ a pensé à vous et propose une nouvelle manière d'introduire des commentaires dans les codes-sources :

```
void         main()
{
    // commentaire pertinent jusqu'à la fin de la ligne
}
```

Les deux caractères « // » indiquent au pré-processeur le début d'un commentaire, qui s'arrête automatiquement à la fin de la ligne. Vous n'avez donc pas besoin de répéter // pour mettre fin au commentaire.

Bien entendu, vous pouvez toujours utiliser l'ancien style de commentaire (entre /* et */), très pratique pour les programmeurs qui ont beaucoup de choses à raconter — sur plusieurs lignes. Les deux styles de commentaires peuvent être imbriqués : une section de code entre /* et */ peut contenir des commentaires //, et réciproquement.

Ruse de sioux

Il existe un autre moyen de mettre en commentaire de nombreuses lignes, très utile si ces lignes contiennent déjà des commentaires avec /* et */. Pensez au pré-processeur et utilisez #ifdef et #endif. Voici le listing initial :

```
void         fonction_boguee()
{
    char     *pointeur;    // pointeur sur un truc

    /* début de la fonction */
    strcpy(pointeur, "Plantage");
}
```

Vous voulez mettre en commentaire tout le contenu de la fonction. A moins de supprimer le commentaire intérieur, vous ne pouvez pas utiliser /* et */, puisque le premier /* serait fermé par le */ du commentaire intérieur.
La solution :

```
void         fonction_boguee()
{
#if 0
    char     *pointeur;    // pointeur sur un truc

    /* début de la fonction */
    strcpy(pointeur, "Plantage");
#endif
}
```

Rappelons que #if *expression* est évalué par le pré-processeur. Si *expression* est vrai, le code qui suit (jusqu'à #endif) est compilé. Il suffit donc de donner une expression fausse pour mettre en « commentaires » la zone de code voulue. C'est pourquoi nous utilisons #if 0 qui n'est jamais vrai.

Paramètres par défaut

Là aussi, une idée simple et efficace : vous pouvez donner une valeur par défaut pour un ou plusieurs paramètres de fonction, dans la *déclaration* de cette fonction. Ainsi, à l'appel, vous pouvez omettre les paramètres qui possèdent une valeur par défaut. Exemple :

Ici, on donne une valeur par défaut (18.6) au deuxième paramètre de la fonction calcul_tva.

```
/* fichier fonction.h */
float   calcul_tva(float montant,  float tva = 18.6);


/* fichier fonction.cpp */
float   calcul_tva(float montant,  float tva)
{
   if (tva > 0. && tva < 100.)
        return montant *  (tva / 100.);
   else
   {    /* traitement d'erreur...  */ }
}


/* fichier main.cpp */
#include "fonction.h"

void         main()
{
   float    res;

   res = calcul_tva(100.);    // 2ème argument vaut 18.6
   res = calcul_tva(100., 5.5);
   res  =  calcul_tva(100.,  18.6);
}
```

Il existe deux possibilités d'appeler la fonction calcul_tva :

- Avec deux paramètes : c'est la solution classique. Dans ce cas, le paramètre par défaut est « écrasé » par la valeur d'appel du deuxième paramètre.
- Avec un seul paramètre : dans ce cas, le compilateur suppose que le deuxième paramètre prend la valeur par défaut (dans l'exemple, 18.6).

Attention : tous les paramètres qui disposent d'une valeur par défaut doivent être déclarés *après* les paramètres normaux, c'est-à-dire à la fin de la liste des paramètres.

Plusieurs paramètres par défaut

Il est possible de définir une valeur par défaut pour plusieurs paramètres d'une fonction. Veillez cependant à les déclarer *après* les paramètres normaux. L'exemple suivant est impossible car ambigu :

```
void        phonquession(int a = 1, int b, int c = 3);
// erreur !
```

Cette déclaration est impossible : le premier paramètre a une valeur par défaut, mais pas le deuxième. Pourquoi n'est-ce pas autorisé ? C'est assez logique. Imaginez que vous appeliez la fonction comme ceci :

```
phonquession(12,13);
```

Mettez-vous un instant dans la peau du pauvre compilateur : que veut dire cet appel ? Que le premier paramètre vaut 12, et le deuxième 13 ? Ou bien le deuxième 12 et le troisième 13 ? Allons, soyons raisonnables, et contentons-nous de déclarer les valeurs par défaut à la file, en fin d'entête :

```
void   phonquession(int  b, int a = 1, int c = 3);
// mieux.
```

Ainsi, nous levons toute ambiguïté. Voici comment seraient interprêtés différents appels à phonquession :

```
phonquession(10);          // b:10  a:l   c:3
phonquession(10, 11);      // b:10  a:ll  c:3
phonquession(10, 11, 12); // b:10  a:ll  c:12
```

Les constantes

Habitué(e) du langage C, vous devez certainement utiliser `#define` à tour de bras pour définir vos constantes. Les plus audacieux d'entre vous auront peut-être adopté le mot-clé `const` (disponible depuis la norme `ANSI` du langage C). Le C++ va plus loin en étendant les possibilités de `const`. Nous allons voir tout cela au travers des différents moyens d'utiliser des constantes en C++.

Une constante simple, visible dans un bloc ou un seul fichier source

C'est la manière la plus simple de déclarer une constante :

```
const   float    TVA = 0.186;
```

Il est indispensable d'initialiser la constante au moment de sa déclaration. On ne peut pas simplement déclarer `const float TVA`, puis, quelques lignes en dessous, affecter une valeur à `TVA`. Tout compilateur C++ vous insultera si vous agissez de la sorte.

Dans le cas d'un tableau constant, ce sont tous ses éléments qui sont constants :

```
const   float    tab[3] = { 2.2, 3.3, 4.4 } ;
tab[2] = 1.1;    // impossible, refusé par le compilateur
```

Comme toutes les autres variables, la portée d'une constante dépend de l'endroit où elle a été déclarée.

Constantes et pointeurs

Il faut bien distinguer *ce qui* est pointé du *pointeur* lui-même. Grâce à `const`, nous sommes en mesure de spécifier au compilateur si nous voulons que le *pointeur* soit constant, ou bien que les *données pointées* soient constantes, ou encore que les deux soient figés. Détaillons ces trois cas :

```
char    chainel[] = "baratin";
char   chaine2[ ] = "billevei/sées" ;
char    chaine3[] = "balivernes";

// ce qui est en gras est constant

const char *       p = chainel;    // (1)
      char * const p = chaine2;    // (2)
const char * const p = chaine3;    // (3)
```

- Dans (1), la chaîne "baratin" est constante, mais pas le pointeur p. Vous pouvez donc écrire p=0 (après vous être assuré qu'un autre pointeur désigne "baratin", sans quoi vous ne pouvez plus y accéder).
- Dans (2), c'est le pointeur p qui est constant. Si vous utilisez un autre pointeur pour accéder à "billeveusées", vous pourrez la modifier librement.
- Dans (3), tout est verrouillé. Ni p ni la chaîne "balivernes" ne peuvent être modifiés.

Illustrons une dernière fois ce qui est possible et ce qui ne l'est point :

```
void        main()
{
    char chainel[] = "baratin";
    char chaine2[] = "billeveusées";
    char chaine3[] = "balivernes";

    const char *       p = chainel; // (1)
          char * const p = chaine2; // (2)
    const char * const p = chaine3; // (3)

      pl[2] = 'X';             // impossible
      pl = new char[10];       // OK

      p2[2] = 'X•;             // / / OK
      p2 = new char[10];       // impossible

      p3[2] = 'X';             // impossible
      p3 = new char[10];       // impossible
}
```

Pourquoi ne pas avoir déclaré les chaînes comme ceci :

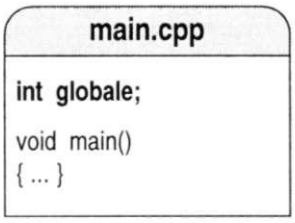
```
const char * p = "baratin";
```

En codant la chaîne "baratin" en « dur » comme ci-dessus, nous n'aurions pas pu la modifier. En effet, une chaîne dé-

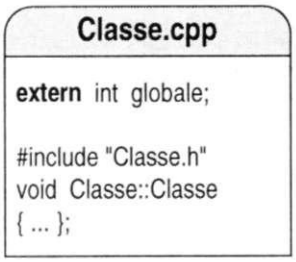
clarée ainsi est stockée dans une zone de mémoire protégée. Tout accès à cette zone engendre un plantage plus ou moins immédiat. Nous avons donc été contraints d'utiliser des variables intermédiaires, qui stockent les données dans une zone de mémoire modifiable.

Une constante simple, visible dans plusieurs fichiers sources

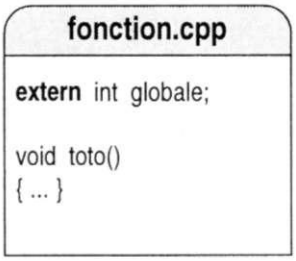
Un petit rappel de C standard s'impose. Si vous voulez déclarer une variable globale visible dans plusieurs fichiers sources, vous procédez ainsi :



définition d'une variable globale



déclaration de la variable globale, définie « ailleurs ».



déclaration de la variable globale, définie « ailleurs ».

Comme vous le voyez, la variable `globale` n'est définie qu'une seule fois, dans le fichier `main.cpp`. Chaque fois qu'un autre fichier veut pouvoir l'utiliser, il doit indiquer que cette variable existe et est définie « ailleurs », dans un autre fichier (c'est le rôle du mot-clé `extern`).

Déclarer qu'une fonction retourne une constante

Ne lisez pas ce paragraphe avant de connaître les références, exposées au chapitre 9, page 113. Pour qu'une fonction retourne une référence constante, il suffit d'indiquer `const` avant le type de retour de la fonction. L'intérêt de cette technique est d'éviter la copie d'un objet entre la fonction appelée et la fonction appelante, tout en préservant l'intégrité de l'objet. Exemple :

```
const int &f()
{
   static big_objet big;
   // manipulations sur big
   return big;
}

void    main()
{
   f() = 0; // déclenche une erreur de compilation
            // à cause du const.
}
```

Déclarer une constante interne à une classe

C'est un problème intéressant : comment déclarer une constante qui ne sera visible et utilisable que dans sa classe ? Vous seriez peut-êter tenté de procéder ainsi :

```
class MaClasse
{
   private:
        const int MAX = 100;   // incorrect, hélas
   public :
        // . . .
};
```

C'est oublier que l'on ne peut pas affecter de valeur directement dans la déclaration d'une classe. Vous ne pouvez que déclarer des variables, pas les initialiser. Malin, vous vous dites que vous aller initialiser la constante en dehors de la classe :

```
class MaClasse
{
   private:
        const int MAX;    // toujours incorrect
   public :
        // ...
```

```
};
MaClasse::MAX = 100;      // toujours incorrect
```

C'est encore impossible, car vous ne pouvez affecter aucune valeur à une constante, en dehors de son initialisation. Et ce n'est pas une initialisation ici, car MAX est une variable d'instance (liée à un objet précis) et non de classe. Elle ne peut donc pas être initialisée sans être static. Que faire, me direz-vous ? Il existe deux solutions. Une avec static const, l'autre avec enum.

La static const solution.

Une constante propre à une classe doit être logiquement déclarée static const. Elle doit être static car c'est une variable de classe, dont la valeur est commune à tous les objets de cette classe ; et elle doit être const car on veut interdire les modifications de sa valeur.

La seule manière d'initialiser une variable static const déclarée dans une classe, c'est de l'initialiser en dehors de la classe. En voici la démonstration :

```
class MaClasse
{
   private:
        static const int MAX;  // constante de classe
   public :
        MaClasse();
};
const int MaClasse::MAX = 100;
```

L'enum solution

Si vos constantes sont de type entier, vous pouvez utiliser enum pour les déclarer et les initialiser à l'intérieur d'une classe. Bien que cela ne marche que sur les entiers, cette solution présente l'avantage d'initialiser la constante *immédiatement.* C'est la seule méthode que je connaisse pour initaliser une constante de classe dans le corps de la classe, et donc qui permette de l'utiliser pour dimensionner un tableau dans la classe. Voici un exemple :

```
class MaClasse
{
   private:
        enum { MAX = 100 };    // MAX est de type int
        int tab[MAX];          // OK
   public:
        // ...
>;
```

Déclarations

En C++ vous pouvez déclarer vos variables ou objets n'importe où dans le code, y compris après une série d'instructions. La portée de telles variables va de l'endroit de leur déclaration à la fin du bloc courant (l'accolade fermante en général).

```
void        main()
{
    int i;

    for (i=0; i<100; i++)
    {
        int j ;
        // traitement
    }
    // ici j est inconnu
}
```

Vous pouvez déclarer dans un sous-bloc une variable portant le même nom qu'une autre variable déclarée dans un bloc supérieur. Dans ce cas, la variable la plus proche de l'endroit de l'instruction est utilisée. Exemple :

```
void        main()
{
    int i;

    for (i=0; i<100; i++)
    {
        int i ;
        i = 1;
```

```
    }
    // ici i vaut 100 et pas 1
}
```

Variable de boucle

Voici une construction intéressante : vous pouvez déclarer une variable de boucle directement *dans* l'instruction `for ( , , )`, ce qui vous garantit que personne n'utilisera cette variable en dehors de la boucle :

```
void    main()
{
    for (int i = 0; i < 100; i++)
    {
        // traitement
    }
)
```

INTERLUDE

Résumé

La découverte du C++ est un peu déroutante, tant les termes et concepts inédits sont nombreux. C'est pourquoi nous vous proposons ici un résumé rapide de la première partie du livre. Il vous permettra d'avoir les idées claires avant d'aborder la suite, riche en rebondissements...

Encapsulation

Le principe fondateur de la programmation orientée-objet est ***l'encapsulation.*** Au lieu de résoudre un problème par des structures et des fonctions sans lien explicite (comme en C), l'encapsulation propose d'utiliser des ***objets,*** entités informatiques regroupant données et fonctions intimement liées. Les données d'un objet ne peuvent être manipulées que par les fonctions créées spécialement pour cela. En C++, un objet est une instance de ***classe,*** que l'utilisateur peut définir comme il le veut.
Les fonctions qui manipulent les données d'un objet sont appelées fonctions-membres.
De multiples objets peuvent être créés à partir de la même classe, chacun contenant des valeurs différentes. Les objets sont issus d'une classe.

```
class NomClasse
{
   private:
        // variables et objets
   public:
        // fonctions de manipulation de ces variables
        //et objets
};

void        main()
{
   NomClasse     objetl;
   NomClasse     objet2;

   objetl.nom_de_fonction_public();
}
```

Constructeurs & destructeurs

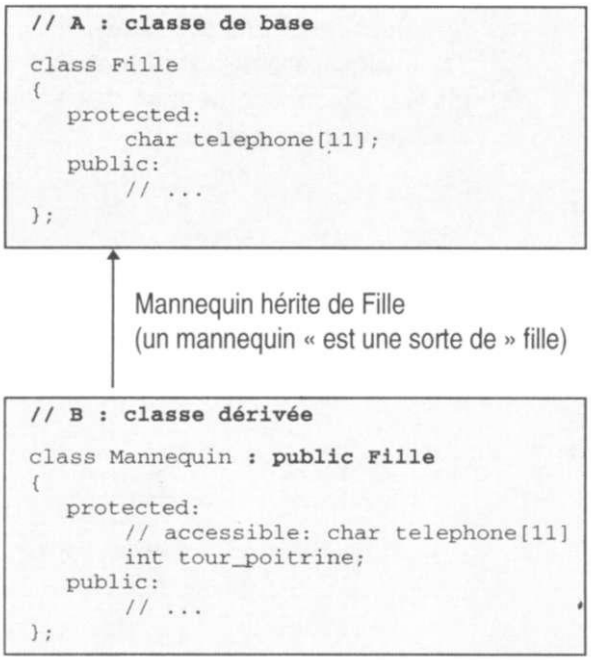
Chaque classe doit posséder au moins une fonction ***constructeur*** et une fonction ***destructeur.*** Si le programmeur n'en définit pas, le compilateur en génère par défaut.
Le constructeur est appelé à chaque création d'objet*, et permet au programmeur d'initialiser correctement l'objet. Le destructeur est appelé dès qu'un objet sort de sa propre portée ou est désalloué explicitement par un appel à `delete`.

* qu'il sort simplement déclaré stàtiquement ou alloué dynamiquement par l'opérateur new.

Héritage

L'héritage permet de mettre en relation des classes. Quand une classe B *hérite* d'une classe A, A est appelée classe de base et B classe dérivée.

Au niveau sémantique, B « est une sorte » de A (voir exemple page suivante). Au niveau du C++, la classe B « recopie » (en quelque sorte) les données et fonctions de la classe A. Un objet de classe B intègre donc les données et fonctions accessibles de A.



La classe dérivée B peut redéfinir des fonctions de la classe de base A. On parle alors de ***redéfinition de fonction membre,*** ce qui signifie qu'un même entête de fonction est partagé par toute une hiérarchie de classes.

Surcharge

La ***surcharge*** de fonction permet au programmeur de donner le même nom à des fonctions situées dans la même classe, ou en dehors de toute classe. La seule contrainte est que le compilateur puisse faire la distinction grâce au nombre ou au type des paramètres de ces fonctions « surchargées ».

```
int     carre(int a)
{
   return (a * a);
}
float   carre(float a)
{
   return (a * a);
}
```

PARTIE 2

Encore plus de C++

Vous connaissez maintenant les grands principes du C++. Mais votre voyage au pays des objets n'est pas fini : il vous reste à découvrir d'autres concepts qui enrichissent encore plus le langage. Une bonne connaissance des éléments présentés dans la première partie facilitera votre lecture.

6

La fin du malloc
new et delete

Nous sommes au regret de vous annoncer le décès de malloc, survenu après de nombreuses années de bons et loyaux services... En réalité, bien que vous puissiez toujours utiliser malloc, calloc et free, le C++ met à votre disposition new et delete, leurs talentueux remplaçants.

Notions de base

L'idée

Le C++ propose deux nouveaux mots-clés, new et delete, pour remplacer leurs aînés malloc et free. Pourquoi un tel changement ? Pour s'adapter aux nouvelles caractéristiques du C++ que sont l'héritage, les constructeurs et les destructeurs. En effet, alors que malloc et free sont deux fonctions de la bibliothèque standard du C, new et delete sont complètement intégrés au langage C++. Ce sont à la fois des mots-clés (réservés) et des opérateurs.

Le rôle de new est double : il réserve l'espace mémoire qu'on lui demande, puis l'initialise en appellant les constructeurs des objets créés.

Le rôle de delete est symétrique : il appelle les destructeurs des objets créés par new, puis libère l'espace mémoire réservé.

Mise en œuvre C++

L'opérateur new

Comme malloc, l'opérateur new retourne soit l'adresse-mémoire nouvellement allouée, soit 0 (zéro) si l'opération a échoué.

On peut utiliser new de trois manières différentes :

Dans les cas ci-contre, la variable pt désigne toujours un pointeur de T (déclaré avec T *pt).

- `pt = new T;`
  L'opérateur new recherche un espace mémoire pour un seul élément de type T. Ici, T peut désigner un type standard (char, int, etc.) ou une classe. Dans ce dernier cas, new appellera le constructeur par défaut de la classe T.
- `pt = new T` *{arguments d'un constructeur de classe T)* -,
  Ici, au lieu d'invoquer le constructeur par défaut de la classe T, new appelera le constructeur correspondant aux paramètres indiqués.
- `pt = new T` *[taille du tableau] ;*
  Cette syntaxe demande à new de chercher un espace-mémoire pouvant contenir tai12e_tableau éléments consécutifs de type T (T pouvant être un type standard ou une classe). Le constructeur par défaut de la classe T est invoqué pour chaque élément.

L'opérateur delete

Attention ! Détruire un objet individuel avec l'instruction delete [ ] provoque quelque chose d'indéfini mais de généralement nuisible (loi de Murphy). De même, utiliser delete pour détruire un tableau entraine des conséquences tout aussi aléatoires !

Il est fortement recommandé d'appeler delete pour libérer l'espace-mémoire alloué par new. La syntaxe de delete est simple :

- `delete pt;`
  Appelle le destructeur de pt (si pt est un pointeur vers un objet) puis libère l'espace mémoire préalablement alloué par un new.
- `delete [] pt;`
  Même chose, mais pour libérer un tableau alloué avec new T [*taille tableau].*

Les spécifications officielles du C++ vous garantissent que rien de mauvais n'arrivera si vous appelez delete sur un pointeur nul.

Exemple

Regardons maintenant comment utiliser new et delete pour allouer et libérer des chaînes de caractères. Prenons comme exemple la classe CDRom, définie ci-dessous :

```
class CDRom
{
   protected :
        char     *titre;
   public :
        CDRom();
        CDRom(char  *tit);
        -CDRom();
};

// définition des constructeurs
CDRom: :CDRom()
{
   titre = new char[8];
   if (titre == 0)
        badaboum();
   strcpy(titre, "<Aucun>");
}

CDRom::CDRom(char *tit)
{
   titre = new char[ strlen(tit) + 1 ];
   if (titre == 0)
        badaboum();
   strcpy(titre, tit);
}

// définition du destructeur
CDRom::-CDRom()
{
   delete [] titre;
}
```

Vous constarerez que nous avons bien respecté la symétrie new/delete. La fonction badaboum s'occupe de traiter les erreurs, lorsque new n'est pas parvenu à allouer la mémoire demandée.

Examinons maintenant comment utiliser new et delete avec des objets. Puisque nous l'avons sous la main, gardons la classe CDRom :

```
void        main()
{
   CDRom    *pt_inconnu,                          // (1)
            *pt_cd_X;                             // (2)
   CDRom    rebel_assault("Rebel Assault");       // (3)

   pt_inconnu = new CDRom;                        // (4)
   pt_cd_X = new CDRom("Virtual Escort");         // (5)
   // blabla                                      // (6)
   delete pt_inconnu;                             // (7)
   delete pt_cd_X;                                // (8)
}                                                 // (9)
```

Les instructions (1) et (2) n'appellent aucun constructeur, et ne réservent de la place que pour les pointeurs eux-mêmes. La ligne (3) appelle le deuxième constructeur de la classe `CDRom`, et initialise l'objet `rebel_assault`. Rien de neuf ici. En revanche, l'instruction (4) va trouver de la place en mémoire pour un objet de classe `CDRom`, initialise par le premier constructeur (le constructeur par défaut, puisqu'il ne prend aucun paramètre). Vous pouvez maintenant utiliser `pt_inconnu` pour appeler des fonctions publiques de la classe `CDRom`. L'instruction (5) va créer un objet initialise par le deuxième constructeur, et retourner l'adresse de cet objet. Enfin, les lignes (7) et (8) appellent le destructeur de la classe `CDRom` pour détruire les objets pointés par `pt_inconnu` et `pt_cd_X`. Ces destructeurs vont libérer les chaînes de caractères allouées.

L'intérêt par rapport au C

Comparons les deux lignes suivantes :

```
tab = (char *) malloc (TAILLE * sizeof(char));   // C
tab = new char [TAILLE] ;                        //,/ C++
```

L'intérêt du new devrait vous frapper de plein fouet. C'est vrai : non seulement new est plus simple à utiliser, mais en plus, il fait partie intégrante du C++, donc il est portable et ne nécessite pas l'adjonction d'une librairie ou l'inclusion d'un fichier d'entête. Par ailleurs, l'utilisation du new permet la vérification de type (ce que ne permet pas `malloc`), donc une plus grande sûreté du code. En outre, new appelle le constructeur voulu, c'est un *opérateur,* il peut être redéfini par vos soins* et il peut être hérité. Que voulez-vous de plus ?

* C'est une possibilité réservée aux experts...

Que celui qui n'a pas pesté la première fois qu'il a découvert le profil de malloc se lève.

Certes, vous pouvez toujours utiliser `malloc` en C++. Mais dans ce cas, vous devrez appeler vous-même les constructeurs voulus, et, ce qui est nettement plus fâcheux, vous risquerez de passer au travers de new spécifiques à certaines classes. Je m'explique : si un gentil programmeur réécrit un joli new pour sa classe et que vous ignorez superbement ses efforts en utilisant `malloc`, non seulement votre objet risquera d'être mal initialise, mais, plus grave, vous vous attirerez inévitablement son inimitié.

Un conseil : mettez-vous au new, vous ne le regretterez pas.

La fin du printf
cout, cin et cerr

Nous venons d'enterrer la famille malloc, passons maintenant à la famille printf. Et donnons maintenant sa chance à la nouvelle génération : la famille cout. Notons que cout, cin et cerr sont des objets issus de classes C++, et qu'ils illustrent très bien le concept de réutilisabilité.

Notions de base

Le C++ intègre dans sa librairie standard différents mécanismes d'entrées et de sorties. Il s'agit des flux cout, cin et cerr, couplés aux opérateurs >> et <<. L'opérateur >> permet de lire un flux, alors que « vous autorise à écrire dedans. Voici le rôle des trois flux standards du C++* :

flux	rôle	exemple
cout	sortie standard (écran)	cout << "Bonjour";
cin	entrée standard (clavier)	cin >> entier;

tream, et cin un objet de classe istream. Tout ce beau monde est défini dans le fichier iostream.h. Nous allons voir comment utiliser les flux C++ et pourquoi ils sont plus intéressants que ceux de C.

Mise en œuvre C++

Prenons un exemple simple : nous allons écrire un petit programme qui calcule la moyenne des notes d'un étudiant.

```
#include "iostream.h"

void        maint)
{
   char     nom[50];
   float    notes[3];
   float    cumul = 0;
   int      i;

   cout << "Entrez le nom de l'étudiant :
   cin  >> nom;
   for (i=0; i<3; i++)
   {
        cout << "Entrez la note n°" << i+1 << '  '; // (1)
        cin >> notes[i];
        cumul += notes[i];
   }
   cout << nom << " a pour moyenne " << cumul/3. << endl;
}

//Ce programme affiche ceci :
//  (les caractères gras sont tapés par l'utilisateur)

Entrez le nom de l'étudiant : GaryMad
Entrez la note n°1 0.5
Entrez la note n°2 7
Entrez la note n°3 19.5
GaryMad a pour moyenne 9
```

Notez que, tout comme pour scanf, vous ne pouvez pas entrer d'espace(s) pendant la saisie d'une chaîne. En fait, pour être exact, vous pouvez le faire, mais vous rendrez votre programme complètement fou, et vous avec.

Remarque

Vous pouvez afficher plusieurs variables ou constantes dans la même instruction, du moment que vous les séparez par <<. C'est le cas en (1), dans la première instruction de la boucle, où nous affichons une chaîne, puis l'entier i, puis le caractère espace. Cette remarque est également valable pour les lectures avec cin et >>.

Mise en forme automatique
Vous noterez que vous n'avez pas besoin de préciser le format d'affichage, celui étant défini par défaut pour les types de base de C. Si, néanmoins, vous souhaitez appliquer des contraintes sur l'affichage (nombre de chiffres décimaux d'un réel par exemple), inutile de revenir à `printf`, `cout` sait le faire très bien. Reportez vous aux compléments de ce chapitre pour savoir comment.

Sans adresse
Vous avez noté l'abscence de & dans la syntaxe du `cin`. Ce dernier n'a pas besoin de connaître l'adresse de la variable à lire, comme c'était le cas pour `scanf`.

L'intérêt par rapport au C

L'intérêt principal de `cout`, `cin`, `cerr`, << et >> est la vérification de type. En effet, chaque objet est affiché en fonction de sa classe ; il n'y a pas de risque de confusion ou de quiproquo. Les fonctions `printf` et compagnie sont incapables de vérifier le type des données qu'elles traitent (ce qui accroît les risques d'erreurs). Plus jamais les chaînes de caractères ne seront affichées comme de vulgaires entiers, je vous le promets ...
Par ailleurs, une simple comparaison de lignes suffit pour que la vérité apparaisse de manière éclatante à l'honnête homme : `cout` et ses frères sont remarquablement plus simples à utiliser que leurs aines de la famille `printf`.

```
// C standard (infâme, n'est-ce pas ?)
printf("%s%d%c%f\n", "Voici i:", entier, '+', réel);
scanf("%f", &reel);

// C++ (mieux, tout de même !)
cout << "Voici i:" << entier << '+' << réel << endl;
cin  >> réel;
```

Mais l'intérêt ne se limite pas là : vous pouvez surtout redéfinir les opérateur << et » pour chacune de vos classes ! Ainsi, les opérations d'entrées/sorties garderont toujours la même syntaxe, indépendamment des classes utilisées ! Par exemple, une classe `FilmHorreur` pourra afficher le nom du film et son réalisateur, opération réalisée de la même manière que pour un entier :

Utilisateurs de MS-DOS, prenez garde ! Le nom de fichier utilisé dans l'include fait plus de huit caractères !

```
#include "FilmHorreur.h"  // voir dans la marge

void        main()
{
    FilmHorreur  film("Doume", "Aïdi software");

    cout << film;
}
```

On suppose bien entendu que la classe est définie, et que « l'on a fait ce qu'il faut » pour que l'opérateur « prenne en compte l'affichage de la classe FilmHorreur. Vous allez apprendre comment réaliser ce petit miracle dans les compléments de ce chapitre.

Compléments

Surcharger « et »

Avant de lire ce complément, vous devez connaître les fonctions amies (page 123), ainsi que les principes de la surcharge, développés page 51.

Vous pouvez surcharger les opérateurs << et >> pour afficher vos propres objets. En d'autres termes, il faut que les fonctions `operator<<` et `operator»` sachent quoi faire lorsqu'elles sont appliquées à des objets de votre classe.
Avant de poursuivre les explications, un petit rappel s'impose : quand on utilise un opérateur @, cela revient à appeler la fonction `operator®` correspondante. Voilà ce que cela donne pour + :

```
int résultat, a = 1, b = 1;

résultat = a + b;          // équivaut à
résultat = operatort (a, b);
```

Transposons l'exemple ci-dessus à « :

```
NomClasse    mon_objet;

cout << mon_objet;         // équivaut à
operator<< (cout, mon__objet);
```

Il s'agit ici de l'opérateur << global. Rappelons que `cout` et `cin` sont respectivement des objets de classe `ostream` et

`istream`. Pour surcharger «, il suffit donc d'écrire la fonction hors-classe suivante :

```
ostream operator<< (ostream, NomClasse)
```

La fonction à surcharger est identifiée. Il ne reste plus qu'à l'écrire. Un problème se pose alors : comment accéder aux données de l'objet de classe `NomClasse` ? Deux solutions : passer par les fonctions membres `public` d'accès aux données, ou bien déclarer dans la classe `NomClasse` que la fonction `operator<<` est une *amie*. Dans ce dernier cas, operator« peut accéder librement aux données `private` et `protected` des objets de `NomClasse`. Ce qui donne :

Vous trouverez page 176 un exemple de surcharge de << qui ne déclare pas de fonction amie.

```
#include <iostream.h>

class NomClasse
{
    private:
        int n;

    public:
        NomClasse(int i) : n(i) {}
        int      get_n() const { return n; }

        // déclarons que l'opérateur global << est
        // un ami de notre classe (NomClasse) :
        friend ostreamS operator<<
             (ostream& out, const NomClasse& obj);

};

// surchargeons l'opérateur global << pour qu'il
// sache traiter des objets de classe NomClasse :
ostream&    operator<< (ostreamk out,
                const NomClasse& obj)
{
   // out est l'objet comparable à cout
   // obj est l'objet à afficher
   return out << '[' << obj.n << ' ] ' << endl;
}

void    maint)
{

   NomClasse    objet(1);

   cout << objet;
}
/*
Pour redéfinir >>, utilisez :
   istream& operator>> (istreamk in, NomClasse& obj);
```

Nous utilisons ici des passages par référence (voir chapitre 9, page I 13).

```
*/

// affichage:
// [1]
```

La fonction operator<< retourne une référence sur un objet de classe ostream, pour que vous puissiez utiliser des instructions à la file :

```
cout << objl << obj2;
// équivaut à :
// operator<< (operator<<( cout, objl), obj2);
```

Dans la ligne ci-dessus, operator<<(cout, objl) doit être lui-même l'objet de classe ostream qui vient d'être appelé ; c'est pour cela que nous retournons une référence vers cet objet.

Formater les sorties

Ces formatages ont été vérifiés sur le compilateur Turbo C++ pour PC, mais devraient fonctionner sur tous les autres compilateurs.

* sauf contre indication dans le texte.

Les indications de formatage (présentation) des sorties doivent être passées à cout de la même manière que les données à afficher, à savoir avec l'opérateur <<.
Quand vous utilisez une instruction de formatage, vous devez inclure au début de votre code le fichier iomanip.h en plus du fichier iostream.h.
Les paragraphes qui suivent donnent des exemples pour chaque type de formatage différent. Vous trouverez ensuite un tableau récapitulatif de toutes ces fonctions.
Attention : toutes ces instructions de formatage* restent valables pour tous les cout qui les suivent dans le programme.

Afficher au moins **n** caractères
L'identificateur setw(n) force cout à afficher au moins n caractères, et plus si nécessaire, setw est l'abbréviation de *set zuidth* (« spécifie la largeur »). Par défaut les affichages sont cadrés à droite. Exemple :

```
#include "iostream.h"
#include "iomanip.h"

void        main()
{
    cout << setw(3) << 1 << endl;
    cout << 1.2345 << endl;
}
```

```
// affiche :
  1
1.2345
```

Pour que cout affiche autant de caractères que nécessaire, spécifiez une longueur minimale de 0(setw(0)). C'est d'ailleurs la valeur par défaut.

Aligner les sorties à gauche
Utilisez ici le barbare setiosflags(ios::left). Exemple :

```
#include "iostream.h"
finclude "iomanip.h"

void        main()
{
    cout << setw(5) << "Wax" << endl;                //(l)
    cout << setiosflags(ios::left) << "Wax" << endl;  11(2)
    II setw(5) est toujours valable ici !
}

// affiche :
  Wax
Wax
```

La ligne (1) indique qu'il faut afficher au moins cinq caractères. Par défaut, l'alignement se fait à droite, d'où la première ligne affichée (deux espaces et les trois lettres).
La ligne (2) précise qu'il faut aligner le texte à gauche. Rappelez-vous que la consigne d'afficher au moins cinq caractères est toujours valable. Cette fois-ci, les trois lettres de « Wax » sont affichées, suivies de deux espaces.

Aligner les sorties à droite
C'est le comportement par défaut de cout. Si vous voulez le rétablir, utilisez setiosflags(ios::right) —voir paragraphe précédent.

Afficher n chiffres après la virgule
Utilisez setprecision(n) :

```
#include "iostream.h"
#include "iomanip.h"

void        main()
```

```
{
    cout << setprecision(2) << 1.23456 << endl;
}

// affiche :
1.23
```

Afficher les zéros après la virgule

Utilisez setiosflags(ios::showpoint).

```
#include "iostream.h"
#include "iomanip.h"

void        main()
{
    cout << setw(5) << setiosflags(ios::showpoint)
         << 1.2 << endl;
}

// affiche :
1.200
```

Ne pas afficher les zéros après la virgule

C'est la fonction réciproque de la précédente. Utilisez setiosflags(ios::showbase) :

```
#include "iostream.h"
#include "iomanip.h"

void        main()
{
    cout << setw(5) << setiosflags(ios::showpoint)
         << 1.2 << endl;
    cout << setiosflags(ios::showbase) << 1.2 << endl;
}

// affiche :
1.200
1.2
```

Afficher un '+' pour les nombres positifs

Utilisez setiosflags(ios::showpos) :

```
#include "iostream.h"
#include "iomanip.h"

void        maint)
{
```

```
    cout << setiosflags(ios::showpos) << 1.2 << endl;
}

// affiche :
+1.2
```

Remplacer le caractère blanc par un autre caractère

Nous parlons ici des espaces affichés pour combler les vides. Par exemple, si vous avez spécifié setw(5) et que vous affichez le chiffre 1, quatre espaces seront affichés en plus. Si vous n'aimez pas les espaces, utilisez setfill(car) pour les remplacer par un caractère de votre choix :

```
#include"iostream.h"
#include"iomanip.h"

void        main()
{
    cout << setfill('-') << setw(5) << 1 << endl;
}

// affiche :
1----
```

Afficher les nombres dans une autre base

Sont définies trois bases : l'octal (base 8), le décimal (par défaut) et l'hexadécimal (base 16). il suffit de spécifier l'abréviation (oct, dec, hex) pour que tous les affichages suivants se fassent dans cette base :

```
#include "iostream.h"
#include"iomanip.h"

void        main()
{
    cout << hex << 12 << endl;
}

// affiche :
c
```

Afficher les nombres en notation scientifique

Un rappel pour les non-matheux s'impose : la notation scientifique est de la forme *n Ee*, où *n* est un nombre réel entre 0 et 1 non inclus, et où *e* est également un nombre réel.

Une telle notation représente le nombre *n* multiplié par 10 puissance *e*. Pour formater l'affichage en notation scientifique, utilisez `setiosflags(ios::scientific)`. Exemple :

```
#include "iostream.h"
#include "iomanip.h"

void        main()
{
    cout << setiosflags(ios::scientific) << 12.3 << endl;
}

// affiche :
.123 e2
```

Afficher les nombres en notation fixe
C'est l'affichage par défaut des nombres, il s'obtient en utilisant `setiosflags(ios::fixed)`. Exemple :

```
#include "iostream.h"
#include "iomanip.h"

void        main()
{
    cout << setiosflags(ios::scientific) << 12.3 << endl;
    cout << setiosflags(ios::fixed) << 12.3 << endl;
}

// affiche :
.123 e2
12.3
```

Tableau récapitulatif

Ce tableau reprend les différentes options d'affichage valables pour cout.

Effet voulu	**cout « ...**
au moins n caractères	setw(n)
aligner à gauche	setiosflags(ios::left)
aligner à droite	setiosflags(ios::right)
n caractères après la virgule	setprecision(n)
afficher les 0 après la virgule	setiosf lags(ios: :showpoint)
ne pas afficher les 0 après la virgule	setiosflags(ios::showbase)
'+' devant les nombres positifs	setiosf lags(ios: :showpos)
changer le caractère de remplissage	setfill(c); // c est un char
afficher en décimal, octal ou hexadécimal	dec, oct ou hex
notation scientifique	setiosf lags(ios::scientific)
notation à virgule fixe	setiosflags(ios::fixed)

8

Le polymorphisme
et la virtualité

Vous connaissez déjà les vertus de l'héritage. Découvrez maintenant celles du polymorphisme, mis en œuvre par la virtualité en C++. Si vos notions sur l'héritage sont encore un peu floues, vous gagnerez à relire le chapitre 3 avant d'aborder celui-ci.

Notions de base

L'idée

Le principe de polymorphisme peut s'apparenter à une surcharge et une redéfinition de fonction-membre étendue. Rappelons que la *surcharge* consiste à pouvoir donner le même nom à plusieurs fonctions ; le compilateur faisant la différence grâce aux paramètres. La *redéfinition de fonction-membre* applique ce même principe à travers une arborescence d'héritage, et permet en plus de donner exactement le même entête de fonction.

Le polymorphisme va plus loin : il permet de trouver *dynamiquement* à quel objet s'adresse une fonction, pendant l'exécution du programme. Alors que la redéfinition de fonction-membre déterminait cela à la compilation, c'est-à-dire de manière *statique,* le polymorphisme mène son en-

quête et va jusqu'à retrouver la classe réelle d'un objet pointé.

Limites de **la** redéfinition de fonction-membre

Reprenons l'exemple d'une gestion de textes *bruts* et *mis en forme.* Si l'on utilise un pointeur de TexteBrut qui pointe réellement vers un TexteMisEnForme, la redéfinition de fonction-membre seule montre ses limites :

```
#include <iostream.h>

class TexteBrut
{
   public:
        void     Imprimer(int nb);
};

void    TexteBrut::Imprimer(int nb)
{
   cout << "Imprimer de la classe TexteBrut" << endl;
}

class TexteMisEnForme : public TexteBrut    // héritage
{
   public:
        void     Imprimer(int nb);
};
void    TexteMisEnForme::Imprimer(int nb)
{
   cout << "Imprimer de la classe TexteMisEnForme"
        << endl;
}
void    main()
{
   TexteBrut         *txt;
   TexteMisEnForme   joli_t;xt;

   txt = &joli_txt;
   txt -> Imprimer(1);
}

// affichage :
// Imprimer de la classe TexteBrut
```

Ce n'est pas le résultat espéré : il faudrait que la fonction Imprimer définie dans la classe TexteMisEnForme soit appelée, car l'objet réellement pointé appartient à cette classe. Le polymorphisme permet justement de résoudre ce problème.

Mise en œuvre C++

Le polymorphisme est mis en œuvre par la virtualité en C++. Reprenons l'exemple précédent. Il suffit d'ajouter le mot-clé `virtual` devant l'entête de la fonction `Imprimer,` dans la classe `TexteBrut.` Ce qui donne :

```
#include <iostream.h>

class TexteBrut
{
   public:
        virtual void      Imprimer(int nb);
};

// ... le reste est identique à l'exemple précédent ...

void    main()
{
   TexteBrut          *txt;
   TexteMisEnForme    joli_txt;

   txt = &joli_txt; // txt pointe sur un TexteMisEnForme
   txt -> Imprimer(1);
}

// affichage :
// Imprimer de la classe TexteMisEnForme
```

Examinons le rôle du mot-clé `virtual` dans notre petite architecture de classes.

Le rôle d'une fonction virtuelle

Quand le compilateur rencontre des fonctions-membres virtuelles, il sait qu'il faut attendre l'exécution du programme pour savoir quelle fonction appeler. Il déterminera en temps voulu la *bonne* fonction, c'est-à-dire celle qui est définie dans la classe de l'objet réel auquel la fonction est appliquée.

Mais revenons à notre exemple pour illustrer ces propos : le petit bout de `main ( )` a beau être petit, il n'en est pas moins très riche en enseignements. L'utilisation d'un pointeur sur classe `TexteBrut` mérite quelques éclaircissements.

Pointeurs sur une classe de base

En temps normal, quand vous déclarez un pointeur de type A, vous ne pouvez pas le faire pointer sur une donnée de type B*. En tous cas, cela devrait être contraire à votre éthique. Mais ici, nous sommes dans une situation spéciale : nous déclarons un pointeur sur une classe de base. Dans ce cas précis, et le C++ nous y autorise, vous pouvez faire pointer ce pointeur vers une classe dérivée de cette classe de base. C'est exactement ce que nous faisons dans l'avant-dernière ligne du listing.

* Oui oui, en C c'est possible, car le C est un grand laxiste. Mais en C++, le contrôle des types est plus serré, et vos incartades ne passeront plus : finies les convertions automatiques hasardeuses de truc_machin vers bidule_chose. De la discipline, que diable !

La ligne qui suit (`txt->Imprimer(1)`) devrait donc faire appel à la fonction `Imprimer` définie dans la classe `TexteBrut`, puisque `txt` est un pointeur sur `TexteBrut`. Mais NON, puisque le C++ aura vu que `txt` pointe maintenant sur `joli_txt`, objet de classe `TexteMisEnForme`. Cette ligne appelera donc la fonction `Imprimer` de la classe `TexteMisEnForme`.

Résumé

Dans une hiérarchie de classe, pour déclarer des fonctions qui ont le même nom et les mêmes arguments* et dont les appels sont résolus dynamiquement, il faut qu'elles soient *virtuelles*. Il suffit de déclarer une fonction virtuelle dans la classe de base pour que toutes les fonctions identiques des classes dérivées soient elles aussi virtuelles*.

Pendant l'exécution, la fonction appelée dépend de la classe de l'objet qui l'appelle (et non du type de variable qui pointe sur l'objet). Dans le cas d'un pointeur sur une classe qui appelle une fonction virtuelle, c'est la classe de l'objet pointé *réellement* par ce pointeur qui détermine la fonction appelée.

Ce mécanisme commun à de nombreux langages orientés-objets est appelé *polymorphisme*. La *virtualité* est sa mise en œuvre C++.

* C'est-à-dire même nombre et mêmes types d'arguments.

* Pour la clarté du listing, on pourra répéter le mot-clé virtual devant chaque fonction concernée, tout au long de la hiérarchie.

L'intérêt par rapport au C

Nous avions déjà vu qu'avec l'héritage, le C++ proposait quelque chose de très intéressant, totalement absent du C. Grâce au polymorphisme, vous disposez d'un moyen nouveau, d'une souplesse et d'une puissance étonnante : utiliser des fonctions homonymes, dont les arguments ont le même profil, tout au long d'une hiérarchie de classe, en bénéficiant de la résolution dynamique des appels.

Rions ensemble. Un utilisateur de Macintosh déclarait récemment : « Windows pour PC, c'est mettre du rouge à lèvre à un poulet »

Exemple : vous développez un environnement graphique. Vous le baptisez *Fenêtres pour Groupes de Travail 96**. Grâce à la virtualité, chaque objet graphique de votre système (bouton, menu déroulant, liste, etc.) possédera une fonction-membre `Afficher`, avec les mêmes paramètres. Il vous sera possible d'écrire quelque chose comme :

```
ObjetGraphiqueGenerique    *obj[10];
// tableau de 10 pointeurs

// ... affectation d'objets divers aux pointeurs du ta-
bleau
// (nous ne détaillons pas cette partie)

for (i = 0; i < 10; i++)
{
   obj[i] -> Afficher(paramètres);
}
```

Vous n'avez plus à vous soucier du type exact de l'objet graphique pointé par `obj [i]`, puisque la bonne fonction `Afficher` sera appelée automatiquement.
En C, il aurait fallu passer par des pointeurs sur `void*`, définir plusieurs structures figées contenant des pointeurs sur des fonctions, et autres bidouilles peu catholiques. Une autre solution consisterait à utiliser des `switch`, mais cela rendrait l'extension du code existant très épineuse. Le C n'étant pas conçu pour cela, vous vous exposeriez à de multiples problèmes de portabilité, de fiabilité, de maintenance. Car la véritable force de frappe du C++, c'est l'héritage. Seul l'héritage, et son complice la virtualité, permettent de mettre en œuvre des systèmes complexes et —relativement— facile à modifier.

Rions ensemble. Un utilisateur de Macintosh déclarait récemment : « Windows pour PC, c'est mettre du rouge à lèvre à un poulet »

Exemple : vous développez un environnement graphique. Vous le baptisez *Fenêtres pour Groupes de Travail 96**. Grâce à la virtualité, chaque objet graphique de votre système (bouton, menu déroulant, liste, etc.) possédera une fonction-membre `Afficher`, avec les mêmes paramètres. Il vous sera possible d'écrire quelque chose comme :

```
ObjetGraphiqueGenerique    *obj[10];
// tableau de 10 pointeurs

// ... affectation d'objets divers aux pointeurs du ta-
bleau
// (nous ne détaillons pas cette partie)

for (i = 0; i < 10; i++)
{
   obj[i] -> Afficher(paramètres);
)
```

Vous n'avez plus à vous soucier du type exact de l'objet graphique pointé par `obj[i]`, puisque la bonne fonction `Afficher` sera appelée automatiquement.
En C, il aurait fallu passer par des pointeurs sur `void*`, définir plusieurs structures figées contenant des pointeurs sur des fonctions, et autres bidouilles peu catholiques. Une autre solution consisterait à utiliser des `switch`, mais cela rendrait l'extension du code existant très épineuse. Le C n'étant pas conçu pour cela, vous vous exposeriez à de multiples problèmes de portabilité, de fiabilité, de maintenance. Car la véritable force de frappe du C++, c'est l'héritage. Seul l'héritage, et son complice la virtualité, permettent de mettre en œuvre des systèmes complexes et —relativement— facile à modifier.

Compléments

Destructeurs virtuel

Il est tout à fait possible de définir un destructeur virtuel pour une classe. A quoi sert un destructeur virtuel ? A détruire l'objet réellement pointé, et non un objet imaginaire qui aurait le même type que celui du pointeur. Un petit exemple :

```
#include <iostream.h>

class Véhicule
{
   public:
        virtual -Véhicule!)
             { cout << "-Véhicule" << endl; }
>;

class Camion : public Véhicule
{
   public:
         virtual -Camion()
             { cout << "-Camion" << endl; }
};

void  main()
{
   Véhicule *pt_inconnu;
   Camion   *mon_beau_camion = new Camion;

   pt_inconnu = mon_beau_camion;
   delete pt_inconnu;
}

// affichage :
/• -Camion
// -Véhicule
```

Le `delete` en dernière ligne appelle le destructeur de `Camion` (puis le destructeur de `Véhicule`, car les destructeurs sont appelés en série de la classe dérivée à la classe de base). Si les destructeurs n'étaient pas virtuels, seul le destructeur de `Véhicule` aurait été appelé.
Pour expliquer cela autrement, `pt_inconnu` est déclaré comme un pointeur sur `Véhicule`, mais pointe *en réalité* sur un objet de classe `Camion`. Comme les destructeurs sont ici virtuels, le programme appelle le destructeur qui à la classe de l'objet réellement pointé, c'est-à-dire `Camion`.

Classes abstraites et fonctions virtuelles pures

Le concept de *classe abstraite* (ou *classe abstraite de base)* est commun à de nombreux langages orientés-objets. De quoi s'agit-il ?

Une classe est dite abstraite si elle est conçue dans le but de servir de cadre générique à ses classes dérivées. De plus, une classe abstraite ne peut engendrer *aucun* objet. C'est un point capital, qui permet de souligner que la raison d'être d'une classe abstraite réside dans ses classes dérivées.

Fonctions virtuelles pures

Le C++ autorise la mise en œuvre des classes abstraites, par le biais de fonctions-membres virtuelles pures. En C++, une classe est déclarée abstraite quand elle contient *au moins une* fonction virtuelle pure.

Une fonction virtuelle pure est une fonction-membre dont seul le profil (valeur retournée, nom et paramètres) est défini. On déclare une fonction virtuelle en ajoutant « `= 0;` » après l'entête de la fonction, en plus du mot-clé `virtual`.

Conséquences

Quand vous déclarez une fonction virtuelle pure dans une classe, voilà ce qui se passe :

- la classe est décrétée « classe abstraite » : elle ne peut donc pas être utilisée pour créer un objet.
- la fonction virtuelle pure ne peut être définie : elle n'a qu'un profil, un entête, mais pas de corps
- les classes futures qui hériteront de cette classe abstraite *devront* définir les fonctions-membres virtuelles pures de la classe abstraite.

Intérêt du virtuel pur

L'intérêt des fonctions virtuelles pures et des classes abstraites de base, c'est de s'assurer que les classes dérivées respecteront l'interface de votre classe. Vous imposez un entête pour une fonction générique, et vous forcez les classes dérivées à redéfinir cette fonction. C'est un moyen supplémentaire de garantir une bonne homogénéité de votre architecture de classes.

Exemple court

Dans le listing qui suit, nous déclarons la fonction-membre afficher virtuelle pure, en ajoutant =0 après son entête. Nous ne pouvons donc pas créer d'objet de cette classe. En revanche, nous pouvons créer des objets de la classe Dérivée qui définit la fonction afficher.

```
class AbstraiteBase
{
   protected : // blabla
   public :
        virtual  void afficher(int x, int y) = 0;
        // c'est ce « = 0 » qui rend la fonction
        // virtuelle pure
};

class Dérivée : public AbstraiteBase
{
   protected : // blabla
   public :
        virtual void afficher(int x, int y);
             // (virtuelle simple)
};

void        Dérivée::afficher(int x, int y)
{
   // traitement
}

void        main()
{
   AbstraiteBase toto;    // impossible !!!
   Dérivée       tata;    // possible.
   tata.afficher(15, 12); // possible.
}
```

Exemple complet

Voici un petit exemple complet mettant en évidence l'intérêt de la virtualité pure. Avant d'en découvrir le listing, quelques éclaircissements :

Dans le cadre du développement d'une interface graphique, nous avons choisi de définir une classe ObjetGraphique, qui contient le minimum indispensable à tous les objets graphiques de notre interface. Par objet graphique, nous entendons tout ce qui est utilisé dans l'interface (bouton, ascenseurs, menus, mais aussi zone pouvant contenir d'autres objets, zone de dessin, etc.) Dans un souci de sim-

plification, nous retiendrons trois caractéristiques : l'objet graphique parent, les coordonnées en x et en y de l'objet dans l'espace de coordonnées associées à l'objet parent. A quoi sert l'objet parent ? A savoir dans quel autre objet est situé l'objet courant. Par exemple, un bouton aura comme objet parent une fenêtre.

Dans le cadre de l'exemple, nous décrirons également la classe BoutonOnOff, correspondant à l'objet «case à cocher ». Pour offrir un minimum de réalisme sans compliquer, nous supposerons que nous travaillons en mode texte, et utiliserons les primitives de Turbo C++ sur PC pour positionner le curseur à l'écran. Rassurez-vous, il n'y a rien de sorcier là dedans. Voici le listing :

```
#include <conio.h>    // propre a Turbo C++,
                      // permet de positionner le curseur

enum Boolean
{   False = (1==0),
    True  = (1==1)
} ;  // type booléen

// _____________________________ classe ObjetGraphique

class ObjetGraphique
{
   protected :
        ObjetGraphique *pere;  // Objet pere
        int             x, y;  // coord. par rapport
                               // au pere

   public :
        // constructeur
        ObjetGraphique(ObjetGraphique *pi = 0,
             int xi = 0,  int yi = 0)
             : pere(pi),  x(xi),  y(yi)
             { }

        // fonctions d'accès
        ObjetGraphique    *get_pere() { return pere; )
        int                get_x() { return x; }
        int                get_y() { return y; }

        // fonctions de modification
        virtual Boolean   set_x(int me) = 0;
        virtual Boolean   set„y(int ny) = 0;

        // fonctions de haut niveau
        virtual void      Redessine() = 0;
};
```

Nous utilisons ici plusieurs techniques propres au C++ et indépendantes des fonctions virtuelles : les paramètres par défaut (voir page 65), la liste d'initialisation (voir page 47), et la définrtion inline des fonctions (voir page 21),

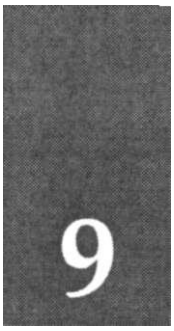
Les constantes sont traitées page 67.

```
// ____________________________________ classe BoutonOnOff

class BoutonOnOff : public ObjetGraphique
{
   private :
        const    char CAR_ON;   //caractère affiché si ON
        const    char CAR_OFF;  //caractère affiché si OFF
        Boolean  état;          //bouton ON (True) ou OFF
   public :
        // constructeur
        BoutonOnOff(ObjetGraphique *pi = 0,
             int xi = 0, int yi = 0,
             Boolean ei = False)
             : ObjetGraphique(pi, xi, yi),
               etat(ei), CAR_ON('x'), CAR_OFF('0')
             { }

        // fonctions virtuelles pures a définir
             // fonctions de modification
             virtual Boolean   set_x(int nx) ;
             virtual Boolean   set_y(int ny);

             // fonctions de haut niveau
             virtual void      Redessine!);

        // fonctions nouvelles de BoutonOnOff
        Boolean  get_etat() { return état; }
        void     set_etat(Boolean ne)
             { état = ne; Redessine(); }
};

// définitions des fonctions de BoutonOnOff
Boolean      BoutonOnOff::set„x(int nx)
{
   i f (nx != x)
   {
        x = nx ;
        Redessine();
   }
   return True;
}

Boolean      BoutonOnOff;:set v(int ny)
{
   if (ny != y)
   {
        y = ny;
        Redessine( );
   }
   return True;
}

void              BoutonOnOff::Redessine()
{
   // positionnons le curseur en x, y
   gotoxy(x, y );
   // affichons un caractère à l'endroit du curseur
```

```
    putch( état ? CAR_ON : CAR_OFF );
}

//_____________________________________programme principal

void        main()
{

    BoutonOnOff  bouton(0, 10, 10);

    bouton.Redessine();
    getch();      // attend qu'une touche soit frappée
    bouton.set_etat(True);
)
```

Ce programme met en évidence l'intérêt de la virtualité pure. Dans la classe de base ObjetGraphique, trois fonctions sont déclarées virtuelles pures : set_x, set_y, et Redessine. Pourquoi ?

- Dans l'idée, pour fournir un cadre général identique à tous les objets graphiques. En forçant les classes dérivées à définir ces trois fonctions, on s'assure que leurs entêtes devront se conformer à un certain profil. Un programmeur sachant utiliser un objet graphique saura automatiquement utiliser tous les autres, y compris ceux qui ne sont pas encore écrits.
- set_x et set_y dépendent bel et bien de chaque objet graphique, car il convient de s'assurer que les nouvelles valeurs sont correctes, notamment, on peut l'imaginer, par rapport à l'objet père qui contient notre objet graphique.
- Redessine dépend aussi de chaque objet graphique. Il ne prend aucun paramètre car un objet doit être capable de se redessiner en fonction de ses données internes (ici ses coordonnées en x et y).



Les références

Seul le passage par valeur est autorisé en C standard. Même lorsque vous passez un pointeur à une fonction, vous passez la valeur de ce pointeur. Le C++ introduit une nouvelle possibilité : le passage par référence (connu depuis longtemps en Pascal). Par ailleurs, ce même mécanisme de référence vous permet de déclarer des synonymes de variables : il s'agit des références libres.

Notions de base

L'idée

Quand vous passez à une fonction un paramètre par *référence,* cette fonction reçoit un synonyme du paramètre réel. En opérant sur le paramètre passé par référence, la fonction peut donc modifier directement le paramètre réel. Vous n'avez pas besoin d'utiliser l'adresse du paramètre pour le modifier, comme vous devriez le faire en C standard.

Mise en œuvre C++

Le caractère & est utilisé dans l'entête de la fonction pour signaler qu'un passage par référence est déclaré :

```
void     plus_l(int   Snombre)
// passage par référence attendu
{
   nombre++;
```

```
)

void    main()
{

   int i = 3 ;

   plus_l(i);
   // i vaut maintenant 4
}
```

Comme vous le constatez, l'appel se fait de manière très simple. La fonction `plus_l` sait que le paramètre entier qu'elle attend est une référence vers le paramètre réel i. En conséquence, toute modification de nombre dans la fonction `plus_l` se répercute sur la variable i du programme principal.

Déclarer une référence libre

Vous pouvez utiliser les références en dehors de toute fonction : en déclarant qu'une variable est « synonyme » d'une autre, c'est-à-dire que l'une est une référence vers l'autre. Dès lors, une modification sur l'une quelconque de ces variables affectera le contenu de l'autre. Par exemple :

```
void main()
{
 int    nombre = 1 ;
 int    &bis = nombre;  // bis est une référence à nombre
 int    *pointeur;

 bis =10;                // bis et nombre valent 10
 nombre = 20;            // bis et nombre valent 20
 pointeur = &bis;
 (*pointeur) =30;        // bis et nombre valent 30
}
```

L'intérêt par rapport au C

Le fait d'utiliser le caractère & pour autre chose que ce qu'il signifie en langage C doit vous paraître étrange. Soyez rassuré : vous pouvez presque totalement vous passer d'utiliser les références, tout en bénéficiant des substanciels avantages du C++. Pourquoi *presque* totalement ? Eh bien, pour être franc, il faut avouer que les constructeurs de copie* ont besoin de types passés par référence.

Vous remarquerez tout de même que, passée la première angoisse, les références se révèlent très pratiques !

* les constructeurs copie sont vus au chapitre 2, page 31.

Compléments

Références et objets provisoires

Pour qu'une référence soit correctement utilisée, il faut que l'objet référence et l'objet initial* soient de même type. Sinon, plusieurs cas peuvent se présenter :

L'objet initial est une constante

```
      int &ref = 1 ;  // (1) NON (voir ci-dessous)
const int &ref = 1 ;  // (2) OUI
```

L'objet initial peut être converti vers l'objet référence

```
int     initial = 5;
        float   &ref = initial;   // (1) NON
const   float   &ref = initial;   // (2) OUI
```

Les deux cas ci-dessus engendrent les mêmes remarques :
(1) La référence n'est pas constante. Votre compilateur doit normalement générer une erreur de type.
(2) La référence est constante. Un objet temporaire est créé ; la référence le désigne.

* Petite précision de vocabulaire : dans la ligne int &ref = ini; ref désigne l'objet référence et ini l'objet initial.

Fonction retournant une référence

Dans son désir infini de repousser toujours plus loin les limites du C, le C++ permet, grâce aux références, d'utiliser les fonctions comme des variables ordinaires auxquelles on peut affecter une valeur. Ainsi, quand une fonction retourne une valeur *par référence*, on peut directement agir sur cette valeur de retour. Mais un petit exemple vaut mieux qu'une explication obscure :

```
// variables globales
int     tab[]={0, 1, 2, 4, 8};
const   int nb_elem = 5 ;  // nbr d'éléments de tab

int &fonction(int j) // retourne une référence
{
    if (j >= 0 && j < nb_elem)
       return tab[j];
    else
       return tab[0];
}

void main()
```

```
{
    fonction(0) = ï0;          // tab[0] vaut -ïb
    fonction(l) = fonction(4); // tab[l] vaut 8
}
```

La première ligne du main est exécutée comme ceci : appel à fonction, qui retourne une référence vers tab[0]. Donc, dans l'expression fonction(0) = 10;, tout se passe comme si fonction(0) était remplacé par tab[0]. On affecte donc bien la valeur 10 à tab[0].
La deuxième ligne fonctionne selon le même principe.

10

Les templates
ou la mise en œuvre de la généricité

Toujours plus, toujours : dans sa débauche d'effets spéciaux, le C++ nous offre les templates, qui vous permettent de paramétrer les fonctions ou les classes par des types. Les templates sont puissants, mais souvent délicats à mettre en œuvre : seuls les compilateurs relativement récents autorisent leur usage de manière simple et fiable.

Notions de base

L'idée

Juste une remarque de vocabulaire : la *généricité* est le concept orienté-objet, alors que les *templates* sont la solution apportée par le C++ pour mettre en œuvre ce concept.

Jusqu'à présent, les fonctions pouvaient avoir comme paramètres des variables ou des objets. Mais jamais vous n'avez eu la possibilité de les paramétrer par des *types*. C'est justement le rôle et l'intérêt de la généricité et des templates.
Des classes peuvent également être déclarées « templates », si vous jugez qu'elles doivent dépendre d'un type. Par exemple, une classe qui gère une liste d'objets quelconques n'a pas à se soucier du type réel de ces éléments : elle ne doit s'occuper que de les classer, d'en ajouter ou d'en supprimer à sa liste, etc. Une telle classe a donc intérêt à être paramétrée

par le type des éléments qu'elle stocke. Il serait judicieux d'en faire une classe « template ».

Mise en œuvre C++

Il est important de distinguer deux types de templates : celles qui concernent les *fonctions* et celles qui concernent les *classes*. Examinons d'abord les templates de fonctions.

Templates de fonctions

Exemple

Prenons l'exemple classique de la fonction qui retourne le minimum de deux chiffres. Il semble légitime de vouloir rendre cette fonction indépendante du type des objets. Voyons la mise en œuvre d'une telle fonction, grâce aux templates :

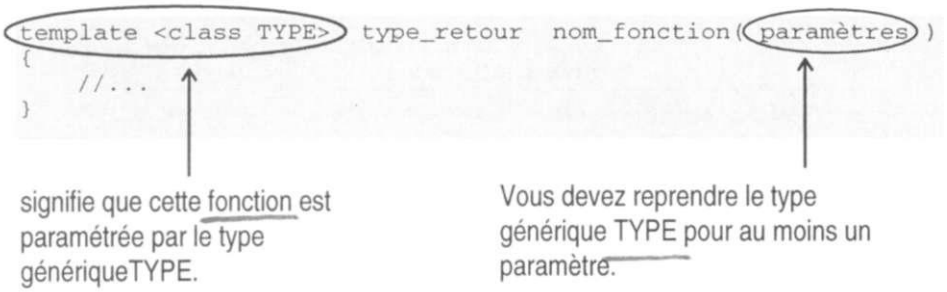
```
tinclude <iostream.h> // pour l'affichage avec cout

template <class TYPE> TYPE min(TYPE a, TYPE b)
{
   return ( a < b ) ? a : b;
}

main()
{
   int      il = 10,  i2 = 20;
   char     cl = 'x', c2 = 'a';

   cout << "min entier : " << min(il, i2) << endl;
   cout << "min char   : " << min(cl, c2) << endl;
}
```

Commentaires

- Observez bien la déclaration de la fonction. Avant le type de retour, vous devez préciser `template <class NomType>` où `NomType` est un identificateur de votre cru, que vous utiliserez ensuite pour désigner le type « paramètre » de votre fonction.
- Vous êtes obligés d'utiliser ce nom de type dans *au moins un* paramètre de votre fonction. Ici, les deux éléments à comparer sont de type `MonType`, et la fonction retourne également un objet de type `MonType`.
- À l'appel, le compilateur reconnaît le type des paramètres et remplace dans la fonction toutes les occurences de

`MonType` par le type voulu (ici, `int` dans le premier appel et `char` dans le deuxième).

Templates de fonctions : cas général



Templates de classes

Exemple

Les templates vous seront également utiles dans des classes. Reprenons l'exemple de la classe qui gère une liste d'objets de type (ou de classe) quelconque. Pour simplifier le propos, nous utiliserons un tableau pour stocker les objets*. Comment la déclarer ?

* Il faudrait bien entendu gérer un nombre variable d'objets, avec allocations et désallocations « à la demande ». Cela compliquerait la classe et nuirait à l'explication des templates. Par ailleurs, la gestion d'erreurs est réduite ici à sa plus simple expression. Nous verrons au chapitre 13, page 137, comment implanter une véritable gestion des erreurs.

```
tinclude <iostream.h> // pour affichage avec cout
#include <stdlib.h>   // pour la fonction exit

template <class TYPE>   class Liste
{
   private:
        enum { MAX = 3 ,} ;     // constante
        TYPE     elem[MAX];

   public:
                 Liste() { }
        TYPE     get_elem(int n);
        void     set_elem(int n, TYPE e);
        void     affiche();
};

template <class TYPE> TYPE Liste<TYPE>::get_elem(int n)
//  retourne le nième élément classé
{
   if (n >= 1 && n <= MAX)
        return elem[ n-1 ];
   else
   {
        cerr <<"Liste::get_elem - Indice hors limite :"
             << n << endl;
```

```
        exit(1);
    }
}

template <class TYPE>
   void     Liste<TYPE>: :set_elem(int n, TYPE e)
// stocke e à la nième place dans la liste
{
    if ( n >= 1 && n <= MAX)
        elem[n-l] = e;
    else
    {
        cerr <<"Liste : :set_elem - Indice hors limite :"
             << n << endl;
        exit(1);
    }

}

template <class TYPE>  void    Liste<TYPE>::affiche()
{
    int     i ;

    for (i = 0; i < MAX ; i + +)
        cout << "Num " << i+1 << ':' << elem[i] << endl;
}

main()
{
    Liste<int>    liste;

    liste.set_elem(1, 10);
    liste.set_elem(3, 30);
    liste.set_elem(2, 20);
    liste.affiche();
    // affichage :
    // Num 1 : 10
    // Num 2 : 20
    // Num 3 : 30
}
```

Quelques explications sur cet exemple

Nous avons choisi de stocker les éléments dans un tableau de taille `MAX`, `MAX` étant une constante spécifique à la classe `Liste`, définie à l'aide d'un `enum` (voir page 71).

La déclaration de la classe indique que c'est une template paramétrée par le type `TYPE` :

```
template <class TYPE>   class Liste
```

Nous utilisons donc le type générique TYPE dans la classe, pour indiquer ce que nous stockons dans le tableau elem :

```
TYPE        elem[MAX];
```

Notez également la définition des fonctions-membres en dehors de la classe :

```
template <class TYPE> TYPE Liste<TYPE>: :get_elem(int n)
```

TYPE est le type de retour de la fonction, dont le nom de classe est Liste<TYPE>. La seule chose qui change vraiment par rapport à une fonction-membre normale, c'est le préfixe « template <class TYPE>».

Utilisation d'une classe template

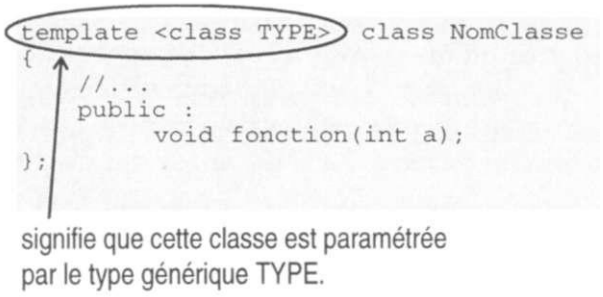
L'utilisation d'une classe template se fait en donnant le nom du type entre chevrons. Dans l'exemple ci-dessus, cela donne :

```
Liste<int>    liste;
```

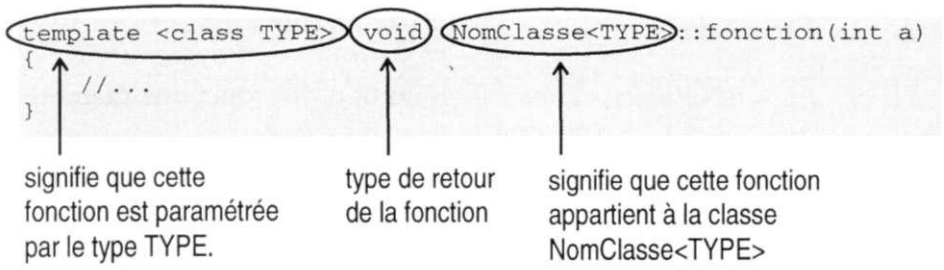
où int aurait pu être remplacé par n'importe quel autre type ou classe.

Templates de classes : cas général

Voici comment l'on déclare une classe template :



Et voici comment l'on définit une fonction-membre d'une classe template :



Compléments

Attention à l'ambiguïté

Il peut vous arriver d'avoir besoin d'imbriquer des templates. Par exemple, vous voulez gérer une liste d'arbres, où la classe Liste et la classe Arbre sont des templates. Il n'y a en théorie pas de problème, mais je tiens à attirer votre attention sur la manière de déclarer un objet Liste :

```
Liste<Arbre<int>>     ma_liste_d_arbres;          // NON
```

Dans cette déclaration, le type de votre liste template n'est autre qu'une classe Arbre, elle-même template, dont le type est int. Cette déclaration provoque instantanément une erreur de compilation, bien que, à première vue, elle semble correcte. La seule chose qui cloche, c'est la présence de ». En C++, les deux caractères supérieurs représentent un opérateur. Il y a donc méprise. Pour y remédier, tapez un espace entre les deux chevrons. La bonne déclaration devient donc :

```
Liste< Arbre<int> >  ma_liste_d_arbres;     // OUI
```

11

Les classes et fonctions amies

L'amitié n'est pas une vertu réservée aux humains. Les classes et les fonctions C++ peuvent, elles aussi, se lier d'amitié. Heureusement, vous seul pouvez les autoriser à s'enticher ainsi les unes des autres.

Notions de base

L'idée

Lorsqu'une *fonction* est déclarée amie (friend) d'une classe, elle peut accéder *directement* à toutes les données private ou protected de cette classe.

Lorsqu'une *classe* est déclarée amie d'une autre classe, ce sont toutes les fonctions de la première classe qui peuvent accéder aux données privées de la deuxième classe.

C'est bien beau, mais si vous avez bien compris le premier chapitre de ce livre, vous devriez bondir de votre siège, et vous écrier « Mais c'est une entorse insupportable à la sacro-sainte règle de l'encapsulation* ! » Et vous auriez raison. Aussi sommes-nous en droit de nous demander *pourquoi* le créateur du C++ a permis que l'on viole — le mot n'est pas

* Rappelons que l'encapsulation consiste à manipuler des objets composés de données et de fonctions de manipulation de ces données. Les données sont cachées au monde extérieur.

trop fort — l'encapsulation ? Essentiellement pour des raisons d'architecture, dues au fait que le C++ n'est pas un langage entièrement orienté objet (il co-existe avec le C), mais aussi pour permettre à un programmeur d'accorder des « droits » entre ses classes internes, et éventuellement pour des raisons d'optimisation (voir aussi les questions-réponses, au chapitre 16). Vous trouverez l'utilité des classes amies dans l'exemple de surcharge de l'opérateur «, page 92. Cela dit, méfiez-vous et n'utilisez les « amies » que si vous y êtes contraint...

Mise en œuvre C++

La déclaration de l'amitié d'une classe A pour une autre entité se fait n'importe où dans la classe A, grâce au mot-clé `friend:`

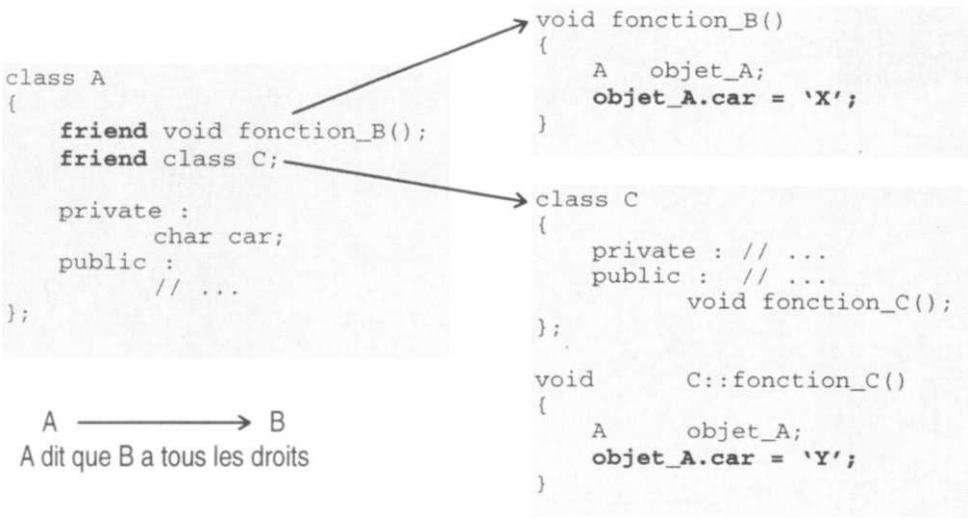
```
class A
{
    friend void fonction_B();   // (1) pour une fonction
    friend class C;             // (2) pour une classe
    // ...
};
```

La ligne (1) signifie que `fonction_B` peut accéder aux données `private` ou `protected` de la classe `A`. Remarquez que si `fonction_B` était une fonction membre d'une autre classe, il faudrait spécifier de quelle classe elle est membre, avec une ligne du style :

```
friend void AutreClasse::fonction_B();
```

La ligne (2) signifie que toutes les fonctions de la classe `C` ont le droit d'accéder directement aux données `private` ou `protected` de la classe `A`.

Comme vous l'avez constaté, la déclaration `friend...` se fait dans la classe *qui accorde le droit* aux autres de venir tripatouiller ses propres données. Récapitulons :



Attention ! Utilisez avec la plus grande précaution les fonctions `friend` : elles tordent le cou à l'encapsulation, et sont donc dangereuses pour la stabilité de votre application. Elles ne sont intéressantes que pour optimiser des classes, ou pour accorder des droits entre classes internes qui n'intéresseront pas d'autres programmeurs. En résumé, bidouillez vos classes à vous, mais faites du travail propre sur les classes qui seront utilisées/reprises par d'autres.

Gardez bien présent à l'esprit qu'une modification dans les données privées d'une classe qui a déclaré d'autres classes amies doit se répercuter sur ces autres classes ! Aussi, méfiez-vous et indiquez toujours clairement dans la documentation quelles classes sont amies de qui, pour que ceux qui se pencheront sur votre travail dans quelques mois (vous y compris) ne s'arrachent pas les cheveux en se cognant la tête contre les murs.

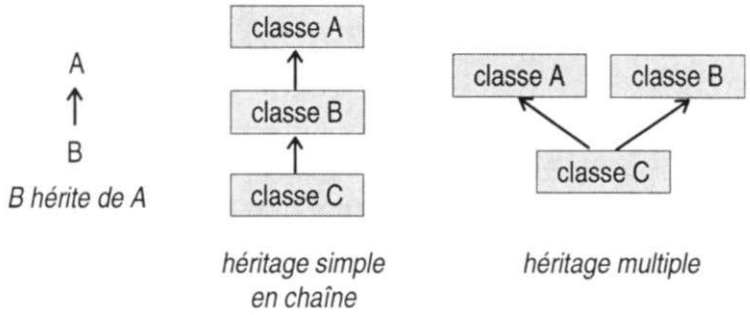
12

L'héritage multiple

L'héritage simple vous permet d'utiliser des classes qui existent déjà pour en créer d'autres. L'héritage multiple est identique, à la différence près que vous pouvez hériter simultanément de plusieurs classes ! Ce qui n'est pas sans engendrer quelques problèmes retors...

Notions de base

L'idée L'héritage multiple vous permet de créer des classes dérivées qui héritent en même temps de plusieurs classes de base. Il ne s'agit pas d'un héritage simple en chaîne mais bel et bien d'un héritage multiple simultané :



Dans le cas présenté à droite du schéma ci-dessus, la classe C hérite du contenu des classes A et B (mais cela dépend toujours des spécificateurs protected, private et public).

Mise en œuvre C++

Un héritage multiple se spécifie comme un héritage simple, en séparant par des virgules les classes de bases désirées. Pour déclarer l'héritage multiple du schéma précédent, il faudrait écrire :

```
class A
{
   // blabla
   public :
        void     fonction_a();
};

class B
{
   // blabla
   public :
        void     fonction_b();
};

class C : public A, public B
{
   // blabla
   public :
        void     fonction_c();
};
```

En admettant que fonction_a, fonction_b et fonction^ ont été définies, vous pourriez écrire quelque chose comme :

```
void        main()
{
   C    objet_c;

   objet_c.fonction_a(); //  appel à A::fonction_a()
   objet_c.fonction_b(); // appel à B: :fonction_b()
   objet_c.fonction_c(); //  appel à C: :fonction_c()
}
```

Il n'y a là rien de bien surprenant. On pourrait même croire que c'est simple. Mais comme de nombreux concepts C++, le dessus de l'iceberg cache un certain nombre de complications difficiles à déceler au premier abord...

Ordre d'appel des constructeurs

Dans le cas d'un héritage multiple, la création d'un objet de la classe dérivée appelle les constructeurs des classes de base dans l'ordre de déclaration de l'héritage. Par exemple :

```
class A
{
};

class B
{
};

class C : public B, public A
// la ligne ci-dessus détermine l'ordre d'appel
// des constructeurs
{
};

void        main()
{
   C   objet_c;
   // appel à B(), puis A(), puis C()
}
```

Les listes d'initialisations sont présentées page 47.

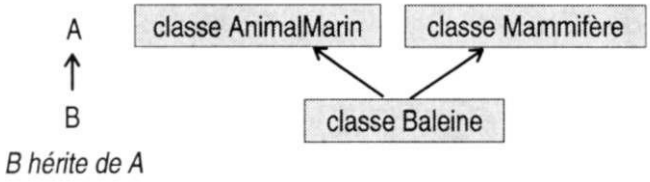
Par ailleurs, si vous précisez une liste d'initialisation*, l'ordre d'appel des constructeurs dépendra toujours de l'ordre de déclaration des classes de bases. Par exemple :

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

C::C() : A(param), B(param) // liste d'initialisation
{
}

void        main()
{
   C   objet_c; // appel à B(param), A(param) puis C()
}
```

L'intérêt par rapport au C

Le langage C ne propose pas d'équivalent de la notion d'héritage simple, il est donc vain de chercher un équivalent d'héritage multiple. Le principal atout de ce dernier est qu'il permet de modéliser de façon plus juste le monde réel.

Par exemple, si vous élaborez une classification des animaux, vous pouvez être amené à considérer la baleine comme un mammifère mais aussi comme un animal vivant sous l'eau. L'héritage multiple vous permettra de garder un schéma cohérent :



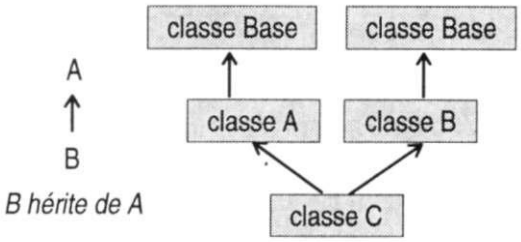
exemple d'héritage multiple

Il faut néanmoins tempérer cette excitation : l'héritage multiple est complexe à manipuler ; nous avons vu à quel point il faut faire attention aux ambiguïtés difficiles à déceler. Essayez autant que possible d'éviter l'héritage multiple. N'oubliez pas que même si vous le maîtrisez parfaitement, ce n'est peut-être pas le cas des futurs utilisateurs de vos classes. Or la compréhension d'une hiérarchie de classes est vitale pour concevoir du code fiable et compréhensible.

Compléments

Duplication de données d'une classe de base

Attardons-nous sur ce qui se passe en mémoire dans certains cas d'héritages multiples. Admettons que les deux classes de base d'un héritage multiple héritent elles-mêmes d'une même classe de base :



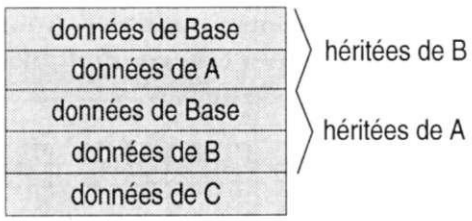
héritage multiple « double »

Traduisons ce schéma en C++ :

```
class Base
{
    int  variable_base;
    // blabla
};

class A : public Base
{
    int  variable_A;
    // blabla
};

class B : public Base
{
    int  variable_B;
    // blabla
};

class C : public A, public B
{
    int variable_C;
    // blabla
};
```

Conséquences : un objet de classe `C` contiendra deux fois les données héritées de la classe `Base`. Pourquoi ? Parce que à la fois l'objet `A` *et* l'objet `B` contiennent les données héritées de la classe `Base`. Or la classe `C` hérite de la classe `A` et de la

classe `B`. `C` bénéficie donc de deux copies *distinctes* des données héritées de la classe `Base`. En mémoire, cela ressemble à ceci :



données d'un objet de classe C

Comment, dès lors, accéder à l'une ou l'autre copie des données de la classe `Base` ? Tout simplement en utilisant l'opérateur de résolution de portée : :. Il suffit de spécifier le nom de la classe à laquelle appartient la donnée voulue. Ainsi, pour accéder à `variable_base` hérité dans la classe `B`, il faut utiliser `B: :variable_base`. De même, pour accéder à la copie de `variable_base` stockée dans la classe `A`, il faut utiliser `A: : variable_base`.
Voici un exemple illustrant la duplicité des données et leur utilisation :

```
class Base

    public :
         int variable_base; // désolé *


class A : public Base

    // blabla


class B : public Base

    // blabla


class C : public B, public A

    // blabla


void main()
```

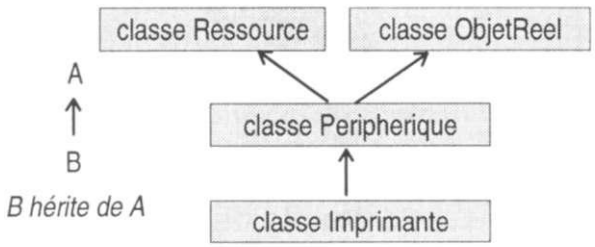
* Pour un exemple simple et court, la variable est déclarée public. C'est un exemple à ne pas suivre (en temps normal). Il faudrait bien entendu la déclarer private ou protected et définir des fonctions public d'accès et de modification.

```
{
    C    objet_c;

    objet_c.variable_base = 1;       // (1) erreur!!!
    objet_c.B::variable_base = 1;    // (2) mieux
    objet_c.A::variable_base = 2;    // (3) mieux
}
```

La ligne (1) ne précise pas à quelle variable_base elle recourt. Le compilateur ne comprend pas et vous demande de lever l'ambiguïté.
En revanche, les lignes (2) et (3) sont sans équivoque. Rappelons que les variables B::variable_base et A::variable_base sont différentes et bien distinctes en mémoire.

Masquage du polymorphisme

Sous ce titre se cache un problème épineux engendré par l'héritage multiple. Le fait d'être obligé de préciser exactement à quelle fonction on fait référence va à l'encontre même du principe de virtualité. Prenons un exemple où cette faiblesse est mise en évidence :



Nous considérons qu'un périphérique est à la fois une ressource et un objet réel, physique. Nous définissons une fonction virtuelle Info pour les classes Ressource, ObjetReel et Périphérique. Voici le listing C++ qui correspond à cette architecture :

```
class Ressource
{
    public:
        virtual void  Info();
};

void   Ressource::Info() { }

class ObjetReel
```

```
{
   public:
        virtual void  Info();

};

void   ObjetReel::Info() { }

class Périphérique : public Ressource, public ObjetReel

{

};

class Imprimante : public Périphérique
{
   public:
        virtual void  Info();
};
void    Imprimante::Info() { }

void         main()
{
    Périphérique       *inconnu;
    Imprimante        hp500;

    inconnu = &hp500;               // (1)
    inconnu -> Info();              // (2) inattendu
    inconnu -> Ressource::Info();   // (3) OK
)
```

Le problème

Voici le problème : en temps normal, grâce au mécanisme de virtualité, la ligne (2) du main devrait appeler la fonction Info définie dans la classe Imprimante*. Mais hélas, à cause de l'héritage multiple, c'est la fonction-membre définie par la classe Périphérique qui est appelée, à notre grand dam !

* En effet inconnu est un pointeur de Périphérique, mais il pointe en réalité sur un objet de classe Imprimante.

Pour plus de clarté, je vais reprendre l'explication d'une autre manière.

Quand une classe, ici Périphérique, utilise l'héritage multiple, cela peut poser un problème d'ambiguité : il suffit que les classes de base héritées (ici Ressource et ObjetReel) possèdent des fonctions homonymes (ici Info) et toc, le compilateur C++ renonce au polymorphisme. Il appelle donc la fonction correspondant au type statique du pointeur (ici Périphérique). Or, la souplesse de la virtualité réside justement dans cette espèce de flou permis au pro-

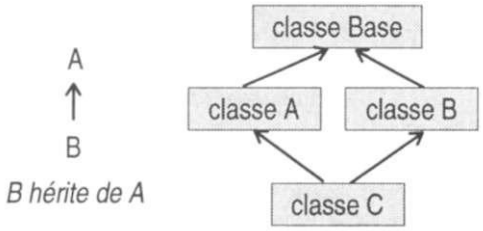
Il existe un moyen d'éviter ce problème : c'est l'héritage virtuel, détaillé ci-après.

grammeur, qui peut utiliser des pointeurs de classes pour appeler des fonctions virtuelles. La bonne fonction étant appelée pendant l'exécution du programme, selon l'objet réellement pointé.

Héritage virtuel

Il vaut mieux lire les paragraphes précédents avant de s'aventurer dans celui-ci.

Comme nous l'avons vu précédemment, les données d'une classe de base peuvent être dupliquées dans certains cas d'héritage multiple. Malheureusement, tel n'est pas forcément le désir du programmeur. Remédiant à ce problème, *l'héritage virtuel* permet de n'avoir *qu'une seule* occurrence des données héritées d'une classe de base.



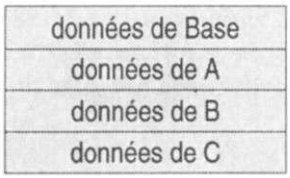
héritage multiple en diamant

Pour que la classe `C` ne possède qu'un seul exemplaire des données de la classe `Base`, il faut que les classes `A` et `B` héritent virtuellement de la classe `Base`. Voici l'exemple C++ qui correspond à cette explication (c'est le même exemple que le paragraphe précédent, à l'exception des mots-clés `virtual` habilement placés) :

```
class Base
{
   public:
        int  variable_base;
        // blabla
};

class A : virtual public Base
{
   int  variable_A;
   // blabla
};

class B : virtual public Base
{
   int  variable_B;
   // blabla
};
```

```
class C : public A, public B
{
    int variable_C;
    // blabla
};
```

Observez maintenant sur ce schéma l'état de la mémoire quand un objet de classe C est créé :



données d'un objet de classe C

Voilà, il n'y a plus qu'une seule occurrence des données de Base. Contrairement au paragraphe précédent où vous étiez obligé de spécifier le « chemin » de vos données, vous n'avez plus à vous en soucier ici. Illustrons cette joviale conclusion par un bout de programme :

```
void main()
{
    C    objet_c;

    ,objet_c.variable_base = 1 ;      // parfait !
>
```

13

Les exceptions

Le traitement des erreurs est depuis toujours une vraie plaie pour les programmeurs. Le C++ nous propose le mécanisme des « exceptions » pour coordonner la lutte contre ces erreurs indésirables. Attention cependant : les exceptions sont encore peu répandues, tant au niveau des compilateurs que des programmeurs C++. C'est donc en quelque sorte un secteur « en chantier »...

Notions de base

L'idée

Un programme, même bien écrit, peut rencontrer des erreurs d'exécutions : disque plein, saturation de la mémoire, etc. Comment traiter ces problèmes ? Le C++ répond simplement par le concept *d'exception.*

Qu'est-ce qu'une « exception » ?
Une exception est une variable, de n'importe quel type, qui symbolise une erreur. Quand une partie de programme rencontre un problème, elle peut *lancer* une exception, qui pourra être *interceptée* par un autre bout de programme.
L'intérêt de ce système est de séparer la détection d'erreur des traitements associés. Par exemple, quand vous écrivez une hiérarchie de classes, il est judicieux de définir les ex-

ceptions que vos classes pourront lancer en cas de problème. De son côté, l'utilisateur de vos classes aura connaissance de la liste de ces exceptions, et pourra envisager différents traitements pour éradiquer le mal.

Mise en œuvre C++

Utiliser des exceptions existantes

Pour ce faire, il faut bien distinguer deux blocs de code : celui qui réalisera les opérations « dangereuses », c'est-à-dire susceptibles de générer des exceptions, et celui qui traitera ces exceptions si elles surviennent. Le premier bloc est précédé du mot-clé try, le second du mot-clé catch. Voyons un petit exemple :

Attention : les exceptions font uniquement référence aux événements internes au programme, et non aux événements extérieurs comme par exemple un clic souris ou l'appui sur une touche du clavier.

```
#include <iostream.h>
#include <except.h>

void main()
{
    try
    {
         int *p = new int[1E99];    // difficile !
         cout << "Fin normale" << endl;
    }
    catch(xalloc)
    {
         cout << "Fin chaotique" << endl;
    }
}

// affichage :
// Fin chaotique
```

L'exemple de xalloc fonctionne avec Turbo C++ sur PC. Consultez la documentation de votre compilateur pour obtenir la liste des exceptions définies.

Allouer $1^{E}99$ entiers est une opération délicate, en tout cas pour mon ordinateur. Il faut savoir également qu'en cas d'échec d'allocation mémoire, une exception de type xalloc est générée. Après le bloc try, nous détaillons un bloc catch qui intercepte justement les exceptions xalloc. L'exécution montre bien que l'exception a été générée et traitée.

Fonctionnement de try et catch

- Quand une exception est détectée dans un bloc try, l'exécution de ce dernier est stoppée, puis déroutée sur le bloc catch correspondant. A la fin du bloc catch, on ne retourne *pas* dans le bloc try fautif. En revanche, le pro-

gramme suit son cours *comme si* le bloc try avait été exécuté.

- Quand une exception est rencontrée, les destructeurs des objets du bloc try sont appelés *avant* d'entrer dans le bloc catch.
- Un bloc try doit obligatoirement être suivi d'au moins un bloc catch.
- Plusieurs blocs catch peuvent être décrits à la file, si chacun intercepte un type d'exception différent.
- Si aucun bloc catch n'intercepte l'exception émise, la fonction terminate est appelée. Par défaut, elle fait appel à la fonction abort qui met fin au programme. Ce point est abordé plus en détail dans les compléments de ce chapitre.

Précisions sur catch

Un bloc catch doit être précédé d'un bloc try. La syntaxe de catch peut être l'une des trois suivantes :

catch (Type)	Intercepte les exceptions de type Type, ainsi que les exceptions de classes dérivées si Type est une classe.
catch (Type obj)	Idem que ci-dessus, mais lèxception est représentée dans le bloc catch par un objet réel : obj.
catch (...)	Intercepte toutes les exceptions non-traitées par les blocs catch précédents.

Intercepter plusieurs exceptions

Il suffit pour cela d'écrire plusieurs blocs catch après le bloc try. Comme nous l'avons vu dans le tableau ci-dessus, vous pouvez intercepter toutes les exceptions, sans distinction, en utilisant catch (...).

```
tinclude <iostreain.h>
#include <except.h>

void main()
{
    try
    {
         // quelque chose qui va déclencher
         // Exceptionl, Exception2 ou une autre exception
    }
    catch(Exceptionl)
    {
```

```
        cout << "Fin chaotique 1" << endl;
    }
    catch(Exception2)
    {
        cout << "Fin chaotique 2" << endl;
    }
    catch (...)
    {
        cout << "Fin très chaotique" << endl;
    }
}
```

Créer et lancer ses propres exceptions

Intercepter les erreurs des autres, c'est bien, mais prévoir les siennes, c'est mieux. Nous avons vu qu'une exception pouvait être de n'importe quel type : une variable ou un objet. La première étape consiste donc à créer les types correspondant à vos exceptions, c'est-à-dire aux erreurs que vous pourrez déclencher. Ensuite, quand vous écrivez une fonction, vous pourrez « lancer » une de vos exceptions grâce au mot-clé `throw`:

throw obj;	Lance l'exception obj.
throw;	Relance la dernière exception lancée.

`throw` sans paramètre peut s'utiliser quand vous n'arrivez pas à résoudre le problème dans un bloc `catch`. Vous relancez donc la dernière exception, en espérant qu'il existe un autre bloc `catch` dans la ou les fonctions appelantes.

Exemple

Dans l'exemple qui suit, la classe `MaClasse` va lancer une exception de classe `MonErreur`, dès que la fonction `FonctionBoguee` est appelée.

```
#include <iostream.h>
#include <except.h>

// définissons une classe exception (en fait une classe
// comme une autre)
class MonErreur
{
   public:
      MonErreur()
         {cout << "Constructeur de MonErreur" << endl;}
};
```

```
// définissons une classe qui va lancer des exceptions
class MaClasse
{
   public:
        void FonctionBoguee!);
};

void    MaClasse::FonctionBoguee()
{
    // admettons que cette fonction soit boguée;
    // elle générera donc une exception :
    throw MonErreur();     // * voir dans la marge
}

void main()
{
    try
    {
        MaClasse obj;

        obj.FonctionBoguee();
    }
    catch (MonErreur)
    {
        cout << "Panique" << endl;
    }
}

// affichage :
// Constructeur de MonErreur
// Panique
```

* Dans la ligne ci-contre, MonErreurO crée un objet temporaire de classe MonErreur. C'est donc bel et bien un objet qui est spécifié après throw.

Cet exemple nous montre que l'exception générée dans FonctionBoguee, de la classe MaClasse, est interceptée dans le main par l'instruction catch (MonErreur).
Nous avons donc mis en évidence le fait que c'est le concepteur de la classe MaClasse qui détecte les erreurs et lance une exception ; alors que c'est l'utilisateur de cette classe, ici la fonction main, qui envisage les solutions selon l'exception rencontrée.

L'intérêt par rapport au C

Le C ne propose aucun système de gestion des erreurs. La pratique la plus courante consiste à renvoyer une valeur signifiant qu'une erreur est intervenue. Cela oblige à tester cette valeur à chaque appel, ce qui alourdit et ralentit considérablement le programme. Par ailleurs, que faire quand une telle erreur est détectée ?

Grâce aux exceptions, les erreurs sont des objets à part entière, déclenchables puis récupérables en cascade, du bloc le

plus proche — en fait le plus compétent pour traiter l'erreur — aux blocs les plus généraux. Mais plus encore, c'est une tentative de formaliser les traitements d'erreur, un cadre général qui permet aux programmeurs de mieux s'y retrouver.

Résumé

Le C++ propose de gérer les erreurs par le mécanisme des exceptions. Une exception n'est rien d'autre qu'une variable qui représente une erreur. Trois mots-clés permettent de les mettre en œuvre : try, catch et throw. Quand une exception est lancée par throw quelque part dans un bloc try, l'exécution de ce bloc est stoppée et déroutée vers le bloc catch correspondant. Si aucun bloc catch n'intercepte l'exception, la fonction terminate est appelée (voir compléments). Par défaut, elle met sobrement fin au programme par un appel à abort ( ).

Exemple complet

Nous allons maintenant détailler un cas concret d'utilisation des exceptions : une classe tableau qui n'accepte pas qu'on écrive en dehors de ses bornes.

```
#include <iostream.h>
#include <except.h>

class ErreurBorne
{
   protected:
        int  borne_fautive;
   public:
        ErreurBorne(int b)
             { borne_fautive = b; }
        int  get_borne_fautive()
             { return borne_fautive; }
};

class Tableau
{
   protected:
        enum { MAX = 10 }; // constante
        int  tab[MAX];
   public:
        int get_MAX() { return MAX; }
        int &operator[](int i);
};
```

* L'opérateur [ ] retourne ici une référence (un synonyme) du tableau pour que l'utilisateur puisse consulter mais aussi modifier sa valeur dans une expression du style t[n] = i. Voir le chapitre sur les références, page 113.

```
int&Tableau: :operator[] (int i)  // * voir dans la marge
{
   if (i<0 || i>=MAX)
        throw ErreurBorne(i);
   else
        return tab[i];
}

void    main()
{
   Tableau  t;
   int     i;

   try
   {
        t[9] = 1;

        t[99] = 2;    // instruction fautive

        i = t[9];     // ligne jamais exécutée
   }
   catch (ErreurBorne err)
   {
        cout << "Erreur de borne : "
             << err.get_borne_fautive() << endl;
   }
}
// affichage :
// Erreur de borne : 99
```

Compléments

Spécifier des exceptions

Vous pouvez indiquer au compilateur et aux futurs utilisateurs de vos classes quelles exceptions vos fonctions sont susceptibles de lancer. Pour ce faire, il faut nommer ces exceptions dans un `throw`, après l'entête normal. Exemple :

Attention : il faut que le throw soit spécifié dans l'entête de déclaration et dans l'entête de définition de la fonction.

```
void    fl(int a)     throw (MonErreur);
int     f  2  (  )    throw (MonErreur, MaCrainte);
void    f3(char *s)   throw(); // aucune exception
void    f4(float f);          // toutes les exceptions
```

Voilà la signification de ces trois lignes : `f1` ne peut lancer que des exceptions de classe `MonErreur` ou de classes dérivées. f2 peut lancer des exceptions `MonErreur` ou `Ma-`

Crainte ou de classes dérivées, f3 ne peut lancer aucune exception, f4 peut se permettre de lancer toutes les exceptions, car aucune limitation n'est exprimée.

Fonction unexpected
Si, malgré tout, une fonction lance une exception qui ne figure pas dans le throw de son entête, la fonction unexpected (inattendue) est appelée. Par défaut, cette fonction fait appel à la fonction terminâte qui fait elle-même appel à abort. Vous pouvez, si vous le désirez, remplacer la fonction unexpected par défaut par une fonction de votre cru. Il suffit d'utiliser set_unexpected avec comme seul paramètre un nom de fonction respectant l'entête suivant :

```
PFV VotreNomDeFonction(void);
```

où PFV est un type défini par :

```
typedef void(*PFV) ();
```

Ce qui correspond à un pointeur sur une fonction qui ne prend aucun paramètre et qui retourne void.
Exemple complet :

```
#include <iostream.h>
#include <except.h>

// si PFV est inconnu, définissons-le
#ifndef PFV
typedef void( *PFV) ();
#endif

class MonErreur {};
class MonDilemne {};

class MaClasse
{
   public:
        void FonctionBoguee)) throw(MonDilemne);
};

void   MaClasse::FonctionBoguee() throw(MonDilemne)
{
   throw MonErreur(); //n'est pas autorisé par l'entête
}

PFV     MonInconnue()
{
```

```
    //ne doit pas retourner à son appelant
    // doit sortir du programme
    cout << "Je suis dans l'inconnue..." << endl;
    exit(1);
}

void main()
{
    try
    {
        MaClasse obj;

        set_unexpected((PFV)Monlnconnue);
        obj.FonctionBoguee();
    }
    catch (MonErreur)
    {
        cout << "Panique" << endl;
    }
}

// affichage :
//Je suis dans l'inconnue
```

Exception non interceptée

Il peut arriver qu'une exception lancée ne corresponde à aucun bloc `catch` qui suit le bloc `try`. Dans ce cas, la fonction `terminate` est appelée. Par défaut, elle met fin au programme par un appel à `abort()`. Vous pouvez cependant définir votre propre fonction terminate, à l'aide de `set_terminate`. Votre fonction doit correspondre à l'entête suivant (où PFV est défini comme indiqué ci-dessus) :

```
PFV VotreNomDeFonction(void);
```

Voici un exemple de fonction `terminate` personnelle :

```
#include <iostream.h>
#include <except.h>

// si PFV est inconnu, définissons-le
#ifndef PFV
typedef void( *PFV ) ();
#endif

class MonErreur {};
class MonDilemne {};

class MaClasse
{
    public:
        void FonctionBoguee() throw(MonErreur);
```

```
);

void    MaClasse::FonctionBoguee()  throw(MonErreur)
{
    throw MonErreur();
}

PFV     MonTerminator()
{
    // ne doit pas lancer d'exception
    // ne doit pas retourner à son appelant
    cout << "On dirait qu'il y a un problème" << endl;
    exit(1);
}

void main()
{
    try
    {
        MaClasse obj;

        set„terminate((terminate_function)MonTerminator);
        obj.FonctionBoguee();
    }
    catch  (MonDilemne.)
    {
        cout << "Panique" << endl;
    }
}

// affichage :
// On dirait qu'il y a un problème
```

Exceptions en cascade

Que se passe-t-il quand une exception est lancée dans un bloc `catch` ? Réponse en deux temps :

- Le bloc A contenant le bloc `catch` est stoppé.
- Si ce bloc A était lui-même dans un bloc `try`, l'exception serait traitée par un des blocs `catch` correspondant au bloc `try` qui contient A.

La brume de ces explications sera peut-être dissipée par l'exemple suivant : une exception de classe `MonDilemne` est lancée dans un bloc `catch`.

```
#include <iostream.h>
#include <except.h>

class MonErreur {);
class MonDilemne {};

class MaClasse
{
```

```
    public:
         void FonctionBoguee() throw(MonErreur);
};

void    MaClasse::FonctionBoguee() throw(MonErreur)
{
    throw MonErreur(); // autorisé par l'entête
}

void fonction!)
{
    try
    {
         MaClasse obj;
         obj.FonctionBoguee();
    }
    catch (MonErreur)
    {
         cout << "Erreur" << endl;
         throw MonDilemne();
    }
    catch (MonDilemne)
    {
         // ce catch n'est pas appelé
         cout << "Dilemne dans fonction" << endl;
    }
}
void    main()
{
    try
    {
         fonction!);   // déclenchera un MonDilemne
    }
    catch (MonDilemne)
    {
         // ce catch est appelé
         cout << "Dilemne dans le main" << endl;
    }
}

// affichage :
// Erreur
// Dilemne dans le main
```

14

La compilation séparée

Bien que la compilation séparée ne constitue pas une nouveauté du C++, un chapitre lui est consacré car beaucoup de programmeurs C en ignorent les principes. Par ailleurs, le C++ diffère légèrement du C dans ce domaine.

Notions de base

L'idée

La compilation séparée permet de répartir le code-source d'une application dans plusieurs fichiers. Cette technique procure une meilleure clarté dans l'organisation du projet. Vous gagnez aussi du temps : seuls les fichiers modifiés depuis la dernière compilation sont recompilés.
Avant d'aborder la mise en œuvre de la compilation séparée, rappelons brièvement le fonctionnement d'un compilateur.

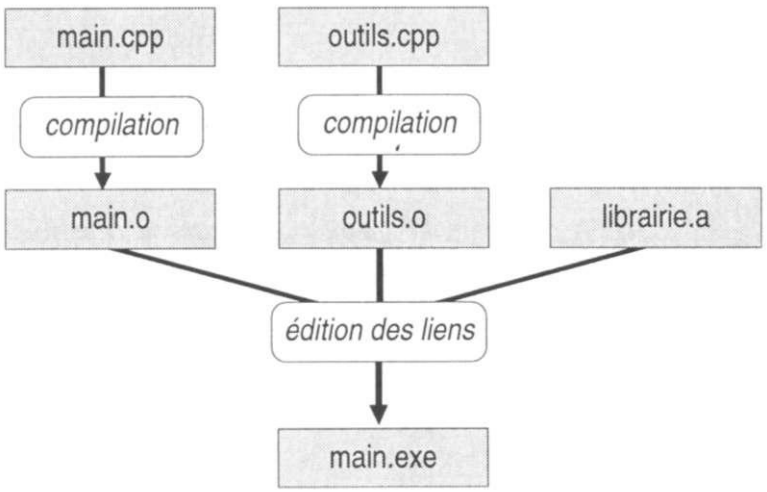
Principe d'un compilateur
Deux grandes étapes sont nécessaires pour transformer votre code-source en exécutable : la compilation et l'édition de liens (parfois appelé « linkage » par certains fous qui refusent d'employer les termes français officiels — *mea culpa...).*
La compilation d'un fichier C ou C++ donne un fichier *objet,*

en général suffixe par `.o`. Ces fichiers `.o` sont ensuite « liés » entre eux, éventuellement avec des librairies*, pour former l'exécutable proprement dit.

Une « librairie » est en quelque sorte un ensemble de fichiers `.o` que vous incluez à l'édition des liens.

Dans l'exemple ci-dessous, deux fichiers sources suffixes par `.cpp` sont compilés séparément puis « linkés » avec une librairie indépendante pour donner l'exécutable `main.exe`.

Si, après une première compilation, vous ne modifiez que le fichier `outils.cpp`, vous n'aurez besoin de recompiler que `outils.cpp` ; vous pourrez réutiliser l'ancien `main.o` pour l'édition des liens.



Compilation séparée

Mise en œuvre

Pour réaliser effectivement une compilation séparée, il faut savoir deux choses :

- quoi mettre, et dans quels fichiers
- comment lancer la compilation

Quoi mettre, et dans quels fichiers ?

Un projet C++ se décompose généralement en beaucoup d'éléments de natures différentes : des classes, des fonctions globales (hors-classes), des constantes, des structures, etc. Comment savoir où placer chaque élément ?

Principe général

Dans la majorité des cas, les fichiers fonctionnent par couple : un fichier suffixe par `.h` pour déclarer une classe ou des entêtes de fonctions, et un fichier suffixe par `.cpp` pour la définition de ces fonctions (membres ou non).

Admettons que vous écriviez dans un fichier A quelconque. Si vous avez besoin d'utiliser une classe ou une fonction définie ailleurs, il faut inclure au début de A le fichier `.h` qui *déclare* ce dont vous avez besoin.

Pour utiliser une autre image, le `.h` est l'ambassadeur de votre travail, il présente comment *utiliser* ce que vous avez fait, alors que le `.cpp` décrit comment c'est *implanté.*

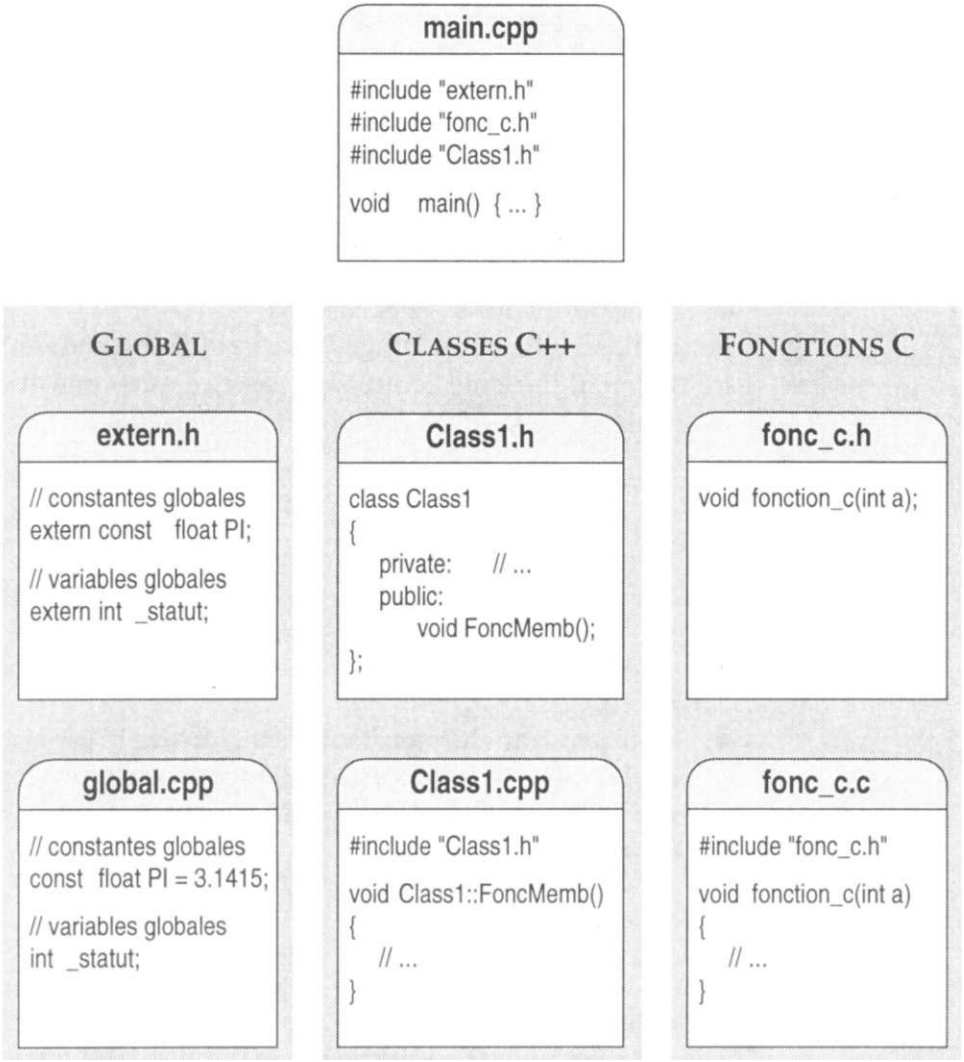
On parlera ici de fichiers .cpp pour désigner les fichiers contenant du code-source C++. Votre compilateur utilise peut-être une autre extension (.cxx ou .C par exemple).

Conseils

- La déclaration d'une classe se fait généralement dans un fichier `.h` qui porte son nom.
- La définition des fonctions-membres d'une classe, ainsi que la définition des constantes et variables de *classe* (`static`) se fait dans un fichier `.cpp` portant le nom de la classe.
- Les objets, variables ou fonctions globales (c'est-à-dire accessibles depuis n'importe quelle fonction du programme) sont définis dans un fichier dont le nom ressemble à `global.cpp`
- La déclaration de ces objets globaux se fait dans un fichier dont le nom ressemble à `extern.h`, où chacun de ces objets est précédé du mot-clé `extern`.
- Les fonctions hors-classe, C standard, figurent dans des fichiers `.c`
- La déclaration de ces fonctions se fait dans des fichiers `.h`

Si vous utilisez de nombreuses classes, vous gagnerez à regrouper plusieurs classes dans un seul fichier `.h`. Il est cependant conseillé de garder un seul fichier `.cpp` par classe — les corps des fonctions prennent en effet plus de place que les déclarations de classes.

Vous trouverez ci-dessous un schéma qui résume ces conseils à travers un exemple. Les listings complets sont situés un peu plus loin dans ce chapitre.



Compilation séparée - exemple

Remarques

Avec la compilation séparée, le programmeur est contraint de respecter certaines règles :

- Chaque fichier .cpp utilisant des éléments d'un autre fichier doit inclure le fichier .h déclarant les éléments en question.

  Ici, main.cpp inclut les fichiers extern.h, fonc_c.h et Classi.h, car il va utiliser des variables et constantes

globales déclarées dans extern.h, des fonctions C déclarées dans fonc_c.h, et des classes déclarées dans Classl.h.

Quand vous modifiez l'entête d'une fonction dans un fichier .cpp, il faut veiller à mettre à jour le fichier .h correspondant.

L'ordre d'inclusion des fichiers .h est important. Si, par exemple, vous utilisez un objet global de classe Classl déclaré dans extern,h, il faut que partout Classl.h soit inclus avant extern.h (sinon le type Classl sera inconnu dans extern.h).

Listing

Voici le listing qui détaille l'architecture présentée dans le schéma de la page précédente :

```
// fichier main.cpp
#include "extern.h"
extern "C"   {    // voir compléments
#include "fonc_c.h"
}
#include "Classl.h"

void    main()
{
   Classl   objetl;
   _statut = 0;
}
```

```
// fichier global.cpp
// constantes globales
const float  PI = 3.1415;

// variables globales
int          _statut;
```

Cette manière d'opérer (avec #ifndef) est expliquée dans les compléments, page I 57.

```
// fichier extern.h
tifndef _EXTERN_H
#define _EXTERN_H

// constantes globales
extern const float        PI;

// variables globales
extern int  _statut;
```

```
#endif


// fichier Classl.h
#ifndef _CLASS1_H
#define _CLASS1_H

class Classl
{
   protected:
        // ...
   public:
        Classl();
        voidFoncMemb();
};

#endif


// fichier Classl.cpp
#include "Classl.h"

// constructeurs
Classl::Classl()
{

}

// autres fonctions-membre de Classl
void    Classl::FoncMemb()
{

}


// fichier fonc_c.h
#ifndef _FONC_C_H
#define _FONC_C_H

#define CONST_C  100

void    fonction_c(int a);

#endif


// fichier fonc c.c
#include "fonc_c.h"

int fonction_c(int a)
{
   return a;
}
```

Comment lancer la compilation ?

Les méthodes de compilation varient beaucoup selon les systèmes. Certains compilateurs s'occupent de presque tout : il vous suffit d'indiquer quels fichiers font partie de votre projet. D'autres en revanche, reposent sur l'utilitaire make.

Make

Cet utilitaire vous permet lancer une compilation séparée d'une seule instruction. Pour cela, il est nécessaire d'écrire un script de compilation, appelé *makefile*. Ce script idendifie les fichiers à compiler, ainsi que les dépendences entre ces fichiers. Si votre makefile est écrit correctement, seuls les fichiers nécessaires seront recompilés.

Le format d'une paire de lignes d'un fichier makefile est le suivant (ici, les crochets indiquent des éléments optionnels) :

```
fichier_cible : fichier_1 [fichier_2 ...]
    instruction à exécuter
```

Si une ligne de votre Makefile est trop longue, vous pouvez la couper en deux à condition que la première ligne se termine par un backslash (\).

Ce qui signifie : pour obtenir `fichier_cible`, j'ai besoin de `fichier_1`, `fichier_2`, etc. et d'exécuter l'instruction de la deuxième ligne. Make en déduit que si l'un des fichiers `fichier_1`, `fichier_2`, etc. est *plus récent* (au niveau des dates de dernière modification) que `fichier_cible`, il faut exécuter l'instruction de la deuxième ligne. Ce format sert principalement à deux genres d'instructions (on supposera que votre compilateur C++ s'appelle `CC`) :

- compiler un `.c` pour obtenir un `.o`

```
fichier.o : fichier.c fichier.h autre_fichier.h
    CC -c fichier.c
```

Ces lignes signifient que si `fichier.c`, `fichier.h` ou `autre_fichier.h` ont été modifiés à une date postérieure de celle de `fichier.o`, il faut recompiler `fichier.c` pour obtenir un nouveau `fichier.o`. L'ordre de compilation dépend bien entendu de votre compilateur.

En fait, on indique après les : le nom du fichier source concerné, ainsi que la liste des fichiers `.h` qu'il utilise.

- lier les . o et les librairies pour former l'exécutable

```
exécutable : fichier.o autre_fichier.o lib.a
    CC -o exécutable fichier.o autre_fichier.o -llib.a
```

Ce qui veut dire: si fichier,o, autre_fichier.o ou lib.a sont plus récents que exécutable, il faut refaire une édition des liens. Nous utilisons ici une librairie à titre d'exemple ; il est tout à fait possible que vous n'en n'ayez pas besoin. Il faudrait dans ce cas écrire :

```
exécutable : fichier.o autre_fichier.o
    CC -o exécutable fichier.o autre_fichier.o
```

Exemple

Voici l'allure du fichier makefile permettant de compiler l'exemple présenté page 152.

```
# fichier makefile
appli  : main.o global.o classl.o fonc_c.o
    CC -o appli main.o global.o classl.o fonc_c.o

main.o  : main.cpp extern.h fonc_c.h classl.h
    CC -c main.cpp

global.o : global.cpp
    CC -c global.cpp

classl.o : classl.cpp classl.h
    CC -c global.cpp

fonc_c.o : fonc_c.c fonc_c.h
    CC -c fonc_c.c
```

Lancer la compilation (enfin!)

Assurez-vous que le fichier makefile est dans le répertoire où sont situés les fichiers-sources de votre projet, et tapez

```
make -f nom_fichier_makefile
```

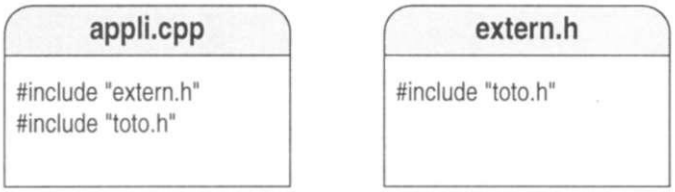
Si vous nommez le fichier makefile Makefile (avec un M majuscule), il vous suffira de taper make pour lancer la compilation séparée de votre projet.

Notez que certains systèmes acceptent indifféremment makefile ou Makefile (avec ou sans majuscule), donnant la préférence au fichier qui commence par une majuscule si les deux sont présents dans le répertoire courant.

Compléments

Comment éviter les redéclarations ?

Le compilateur n'aime pas les déclarations multiples d'un même élément (objet, variable, constante, fonction, etc.) Il faut donc veiller à ne pas inclure deux fois le même fichier .h dans le même fichier. Cela arrive facilement dans un cas de type « poupées russes » :



En traitant appli.cpp, le pré-processeur remplace la ligne #include "extern.h" par le contenu du fichier extern.h. On obtient donc deux lignes #include "toto.h", qui inclueront deux fois le fichier toto.h, ce qui va générer des erreurs de redéclaration.
Il existe un moyen simple d'éviter ces redéclarations, tout en permettant l'inclusion multiple du même .h : utiliser les directives de précompilation #ifndef, #define et #endif. Le truc consiste à ne prendre en compte le contenu du .h que la première fois que le précompilateur le rencontre. Voici un fichier .h qui utilise cette technique :

```
// début du fichier extern.h
#ifndef _EXTERN_H
#define  EXTERN_H

// corps complet du fichier

#endif
// fin du fichier extern.h
```

En fait, on associe à chaque fichier .h un symbole portant son nom (ici `_EXTERN_H`). AU début, ce symbole n'existe pas. La première fois que le précompilateur traite le fichier, il rencontre la ligne `#ifndef _EXTERN_H`. Comme le symbole `_EXTERN_H` n'existe pas, le précompilateur traite le fichier jusqu'à rencontrer un `#endif` (ou un `#elif` d'ailleurs). Le pré-processeur traite donc tout le corps du fichier jusqu'à notre `#endif` final.
Dans cette partie de code prise en compte, la première ligne est `#define _EXTERN_H`, qui définit le symbole `_EXTERN_H`, ce qui servira au prochain passage à rendre la condition `#ifndef _EXTERN_H` fausse, et à éviter d'inclure une nouvelle fois le fichier `extern.h`.

Static, inline et portée

Le gain de temps et de clarté n'est pas le seul intérêt de la compilation séparée. La décomposition en plusieurs fichiers a aussi des conséquences sur la portée de certaines variables et fonctions :

Static global

Toute fonction déclarée `static` ainsi que toute variable déclarée `static` en global*, n'est accessible que dans le fichier où elle est définie. Exemple :

* C'est-à-dire en dehors de toute fonction

PORTÉE DE VAR ET FONCI

fich1.cpp

```
static int  var;
static void fonc1() { ... }

void toto()
{ var = 1; fonc1(); } // OK
```

fich2.cpp

```
extern int  var;
extern void fonc1();

void toto2()
{ var = 1; fonc1(); } // erreur
```

On ne peut accéder ni à `var`, ni à `foncl` depuis `fich2.cpp`, car ils ont été déclarés `static` dans fichl.cpp. Ils ne sont accessibles qu'à l'intérieur de `fichl.cpp`.

Attention ! Ici, static n'a pas le même sens que lorsqu'il qualifie les variables-membres ou fonctions-membres ! Rappelons que dans ces derniers cas, static signifie qu'il s'agit de variables ou fonctions de *classe,* communes à tous les objets issus de la classe.

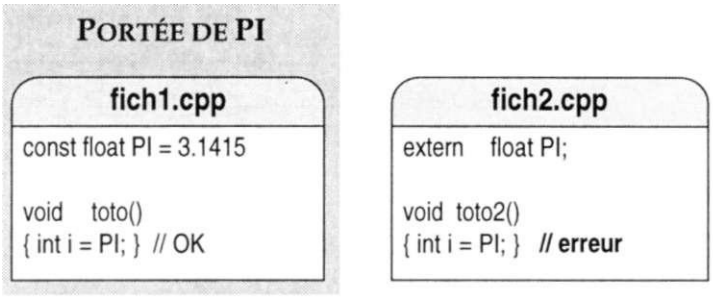
Fonctions inlines

Si vous déclarez une fonction inline en dehors d'une classe, la portée de celle-ci est automatiquement réduite au fichier dans lequel elle est déclarée.

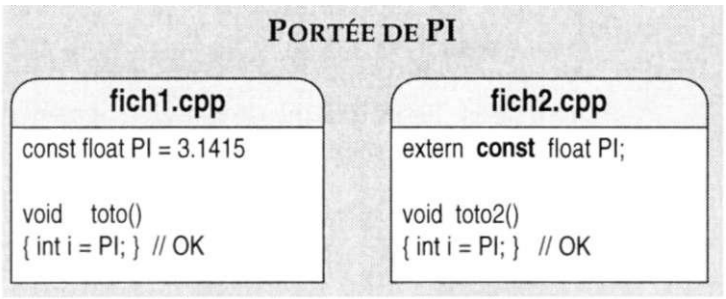
Constantes globales

Une constante déclarée en global n'est visible que dans le fichier où elle est définie, à moins de spécifier explicitement const dans la décla'ration externe.

Exemple 1, où PI est déclaré localement à fichl.cpp :



Exemple 2, où PI est visible à la fois dans fichl.cpp et fich2.cpp :



Travailler avec d'autres langages

Le C++ permet l'usage de fonctions provenant d'autres langages : le C bien entendu, mais aussi le Pascal, le Fortran, etc. Comme chaque langage adopte sa propre logique d'édition des liens (notamment en ce qui concerne la manière de stocker les paramètres de fonctions sur la pile d'exécution), il faut l'indiquer d'une manière ou d'une autre dans le code C++. Vous pouvez le faire grâce à la clause `extern "nom"`, où nom représente un type d'édition de liens (consultez la documentation de votre compilateur).
Il existe trois manières d'utiliser `extern "nom"` :

extern "nom" *élément;*	déclare que l'édition des liens de *élément* (un entête de fonction ou une variable) se fait selon la convention *nom.*
extern "nom"{ *bloc de déclarations;* }	Idem que ci-dessus, mais tous les éléments déclarés dans *bloc* sont concernés.
extern "nom" { #include "fichier.h" }	Idem que ci-dessus, mais toutes les déclarations faites dans *fichier.h* sont concernées.

Prenons l'exemple d'un programme C++ ayant besoin de fonctions C. On utilise `extern "C"` :

```
// fichier sériai.h (C standard)

int     serial_output(int, unsigned char);
int     serial_input(int, unsigned char*);


// fichier main.cpp (C++) possibilité 1

extern "C" int serial_output(int, unsigned char);
extern "C" int serial_input(int, unsigned char*);

void main { /*...*/ }
```

Utilisons la deuxième possibilité pour déclarer les deux fonctions sans répéter `extern "C"` :

```
// fichier main.cpp (C++) possibilité 2

extern "C"
{
   int      serial_output(int, unsigned char);
   int      serial_input(int, unsigned char*);
}
```

```
void main { /*...*/ }
```

Enfin, adoptons la dernière solution, la plus élégante, la plus économique, bref, la meilleure dans notre exemple :

```
// fichier main.cpp (C++) possibilité 3

extern "C"
{
#include "sériai.h"
}
void main { /*...*/ }
```

Utiliser du code C++ à partir du C

Pour utiliser du code C++ à partir d'un code C, suivez l'exemple suivant :

```
// fichier C

extern "C" double racine(double d)
{
   // code C++
   Math     monObjetMath;
   return monObjetMath.racine(d);
}

void    main()
{
   double res;

   res = racine(22.);
}
```

PARTIE 3

Guide de survie

Les concepts présentés dans les deux premières parties vous laissent peut-être perplexe : comment utiliser un tel arsenal pour venir à bout d'un problème réel ? Cette partie expose quelques conseils de base pour concevoir une application en C++, ainsi qu'une série de questions/réponses sur le langage.

15

Conseils

Ce chapitre aborde quelques conseils de programmation orientée-objets en C++. Il ne s'agit pas d'énoncer des règles d'or, mais plutôt de donner quelques pistes pour débuter. Par la suite, l'expérience aidant, vous pourrez concevoir et coder dans un style qui ne vous sera dicté par personne...

Les étapes d'un projet

Un projet informatique peut se décomposer en quatre phases grossières :

1. Détermination du cahier des charges fonctionnel
2. Conception
3. Codage
4. Maintenance

Malgré tout le soin pour les séparer, ces phases ont tendance à se mélanger de manière subtile. C'est pourquoi les tentatives de formalisation de la programmation ont souvent échoué, car il s'agit là d'une activité profondément humaine, voire artistique.

La première phase consiste à déterminer ce que vous (ou vos clients) attendent de l'application à développer. Il est en effet capital de ne jamais perdre de vue les besoins auxquels le programme doit répondre. Malheureusement (ou heureusement), ces besoins sont susceptibles d'évoluer tout au long du projet, mettant en cause ce qui était considéré comme acquis dès le début. Nous ne nous pencherons pas plus sur cette phase qui ne dépend pas du C++.
Attardons-nous maintenant sur la conception proprement dite.

Conception

La conception orientée-objets consiste essentiellement à trouver les classes qui modélisent votre problème et les relations qui se tissent entre elles.

Trouver les classes

C'est une phase essentielle. Pourtant, il est difficile de donner des conseils généraux tant les solutions dépendent du problème. On isole plus facilement les classes quand on pressent (du verbe pressentir) les relations qui les uniront ; aussi la lecture de la section suivante pourra-t-elle vous aider.
Quelques remarques : une classe peut être vue comme une entité, une idée au niveau de votre projet. Rappelons qu'une classe est un tout, formé de données et de fonctions qui traitent ces données. Pour créer une classe, il faut donc que des liens étroits existent entre ses composants. Si cela n'est pas le cas, il se peut qu'elle se décompose en sous-classes.
Un autre moyen consiste à formuler le problème en français. Vous aurez alors de bonnes chances pour que les noms communs représentent des objets, et que les verbes symbolisent des fonctions-membres (c'est la méthode de Booch).
À ce niveau de conception, ne prenez pas en compte les classes liées à l'implémentation, mais concentrez-vous sur celles qui réprésentent directement la réalité du problème. Inutile,

par exemple, de penser à des listes ou des tableaux à ce niveau.

Architecturer les classes

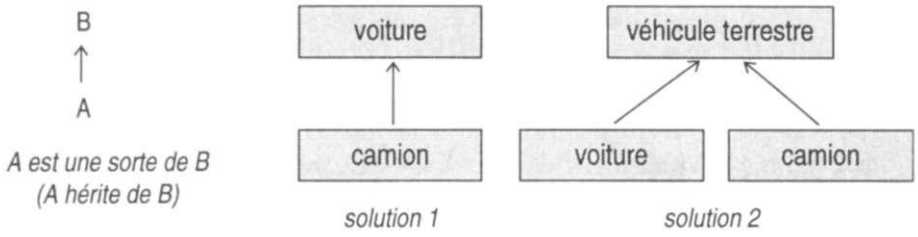
C'est pratiquement la phase la plus difficile. Il existe principalement quatre sortes de relations entre deux classes A et B :

1. A est une sorte de B
2. A est composé de B
3. A utilise B
4. A n'a aucun lien avec B (eh oui!)

La différence entre les points deux et trois n'est pas toujours évidente, mais nous allons voir cela dans les paragraphes qui suivent. Nous ne détaillerons pas le dernier point, dont le rôle est simplement de vous rappeler qu'il ne faut pas absolument s'acharner à trouver un lien entre deux classes.

A est une sorte de **B**
Il s'agit d'une relation d'héritage. La formulation « est une sorte de » fonctionne très bien pour l'identifier. Si malgré tout cela ne suffit pas, il faut penser que la classe dérivée (ici A) peut être une *spécialisation* de la classe de base.
Dans certains cas, une relation ne peut pas se modéliser sous forme « est une sorte de », bien qu'intuitivement vous sentiez qu'il existe une relation d'héritage entre les deux. Il est alors possible que les deux entités soient « sœurs », c'est-à-dire qu'elles aient des points communs et héritent toutes deux d'une même classe de base.

Exemple
Quelles sont les relations entre *camion* et *voiture* ? Une voiture « est-elle une sorte » de camion ou est-ce l'inverse ? Dans ce cas, selon le problème, on peut pencher plutôt pour la solution « un camion est une sorte de voiture », car les voitures sont plus fréquentes, et elles ont été inventées avant les camions. Les camions sont donc une spécialisation des voitures. On peut aussi créer une classe *véhicule*, ou véhicule terrestre, qui servira de classe de base à voiture et camion.



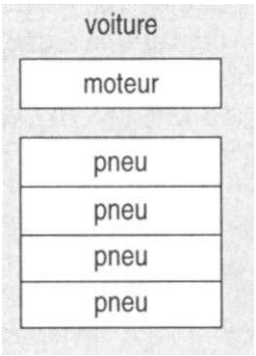
A est composé de un ou plusieurs B

Cette relation représente un objet qui peut se décomposer en d'autres objets, qui possèdent chacun leur vie propre. De tels objets B sont déclarés comme données-membres de l'objet principal A.

Exemple

Une voiture est composée d'un moteur et de quatre pneus. Ce moteur et ces pneus sont des objets qui peuvent exister indépendamment.

Ce n'est visiblement pas une relation d'héritage, car un pneu (ou un moteur) n'est pas « une sorte de » voiture.



A utilise B

Il existe deux manières de représenter cette relation en terme de langage orienté-objet. Soit l'objet B est défini comme donnée-membre de l'objet A, soit A utilise des objets B globaux ou passés comme paramètres de fonctions.

Par rapport à la relation précédente (A est composé de B), la nuance se situe au niveau du sens : ici, les objets B ne font

pas partie de l'objet A. Ils sont complètement indépendants, mais sont utilisés par A pour résoudre un problème.

Exemple
Dans un environnement graphique, le gestionnaire d'événements utilise les informations en provenance de la souris. Il n'est ni composé d'une souris, pas plus qu'il n'est lui-même une sorte de souris.

Bien entendu, en terme d'implémentation, il se peut que la classe gestionnaire d'événement ait comme donnée-membre un objet de classe souris. Mais il se peut également qu'il communique autrement avec l'objet souris, par exemple par l'intermédiaire d'une autre classe qui centralise les entrées.

Détailler les classes

Les classes sont connues et leurs relations identifiées. Il faut maintenant détailler les fonctions-membres qu'elles contiennent. Ce qu'il est bon de garder à l'esprit :

Minimiser les échanges entre les classes
Plus vos classes sont indépendantes, moins elles échangent d'informations avec les autres, et plus facile sera la maintenance de votre projet. Réduisez autant que possible le nombre de paramètres dans les fonctions-membres.

En dévoiler un minimum
Dans le même ordre d'idées, une classe doit s'efforcer de cacher un maximum de ses propres données, et surtout la *manière* dont ces données sont stockées. Imaginez toujours ce qui se passerait si vous changiez la manière de représenter ces données. L'interface avec l'extérieur, c'est-à-dire les fonctions `public`, ne doit pas changer — en tout cas aussi peu que possible.

Évaluer l'ensemble

Confrontez plusieurs scénarios à votre architecture de classes, et vérifiez que tout peut être mis en œuvre. Vous gagnerez à passer en revue les fonctionnalités attendues (phase 1). Si votre conception s'avère trop complexe à utiliser, remettez-là en cause et essayer d'en imaginer une autre. Il faut savoir que si vous avez des problèmes de compréhension de l'architecture à ce niveau, ils ne feront que s'amplifier à l'étape suivante.

Codage C++

Votre projet est détaillé sur le papier, il ne reste plus* qu'à le programmer. Avec le C++, vous disposez de nombreux outils dont l'utilisation simultanée n'est pas toujours évidente. Ce paragraphe va tenter de vous faciliter la tâche.

* C'est une façon de parler.

Quels outils ?

Cette étape consiste à choisir les classes-outils ou les bibliothèques que vous utiliserez pour mettre en œuvre votre conception. Ce ne sont pas des classes conceptuelles, mais des classes propres aux techniques de programmation. Par exemple, toutes les classes « conteneur » (tableaux, listes, arbres, graphes) en font partie. Elles servent à stoker d'autres objets.

Si vous devez concevoir des classes-outils pour votre projet, pensez à la réutilisabilité, et faites en sorte qu'elles soient aussi universelles que possible. Vous aurez peut-être plus de mal, mais vous gagnerez du temps sur votre prochain projet. Bien entendu, ce genre de classe-outils nécessite une documentation : commentez-les ou décrivez leur fonctionnement à part, si possible avec des exemples d'utilisation.

Factoriser des classes

Reprenez votre graphe d'héritage. Vous pouvez essayer de trouver des points communs entre classes, suffisamment nombreux pour former une classe de base. L'avantage d'une

telle factorisation réside dans la minimisation des redondances. Donc, les modifications éventuelles n'auront à se faire qu'à un endroit si elles concernent la classe de base. Cette étape ne figure pas dans la conception, car il s'agit là d'une méthode de programmation qui n'a pas forcément de sens conceptuel.

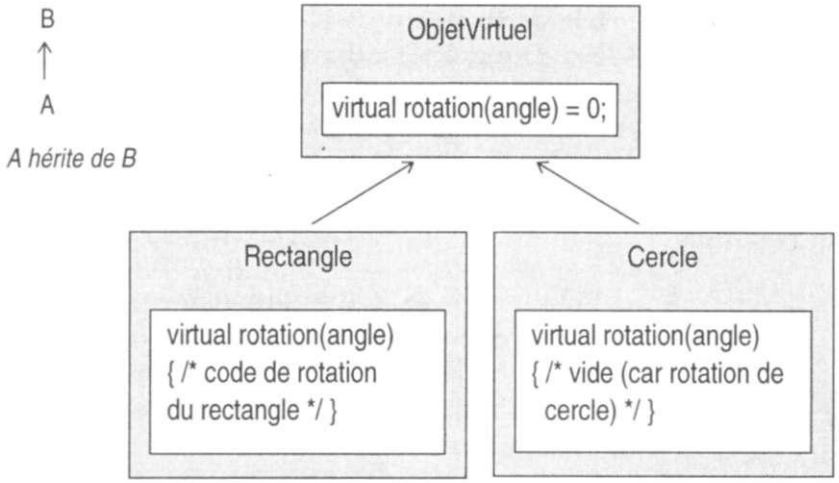
Exemple
Vous concevez un environnement graphique composé de divers objets : fenêtres, boutons, etc. Si chaque objet comporte un identifiant, une référence vers l'objet parent, et je ne sais quoi d'autre, il est judicieux de créer une classe ObjetGenerique qui regroupera tous ces points communs. Ici nous bénéficions d'un autre atout : tous les objets fenêtres, boutons et autres pourront êtres désignés par un pointeur sur ObjetGenerique*.

* Rappelons que le C++ fart la conversion de Dérivée* vers Base*.

Mise en œuvre de la réutilisabilité

L'un des chevaux de bataille des langages orientés-objets est la réutilisabilité du code produit. Le C++ est particulièrement bien armé pour mener à bien cette croisade. Outre l'héritage que nous avons détaillé dans la partie conception, parlons ici du polymorphisme, de la généricité et des exceptions.

Le polymorphisme et les fonctions virtuelles
Une fonction virtuelle doit garder le même sens dans toute la branche d'héritage. Un exemple parfait serait une fonction de rotation virtuelle définie pour un objet graphique de base.
Un bon moyen d'imposer un moule pour une série de classes consiste à créer une classe de base *abstraite,* où des fonctions virtuelles pures sont déclarées. Ce genre de classe fixe un cadre général d'utilisation : les classes dérivées doivent définir les fonctions virtuelles pures, en respectant leur entête.
Notre exemple s'appliquerait bien à cela : une classe abstraite ObjetVirtuel, qui ne créerait donc aucun objet, mais qui définirait les entêtes des fonctions affichage, rotation, etc. On imagine facilement que des classes Rectangle ou Cercle héritent d'ObjetVirtuel (voir schéma page suivante).



La généricité et les templates

Les templates permettent de paramétrer des classes par des types de données, ces derniers pouvant être d'autres classes. Les conteneurs (les classes de stockage) sont tout indiqués pour utiliser les templates. Un bon exercice serait de concevoir une classe Liste template, qui accepterait comme paramètre le type des objets qu'elle stocke.

Les exceptions

Quand vous concevez une classe, vous ne savez pas forcément qui va l'utiliser, ni quand on va l'utiliser. En adoptant le mécanisme d'exception, vous pouvez signaler des erreurs (throw) à l'utilisateur de la classe qui agira en conséquences (try/catch). Documentez et signalez clairement les classes d'exceptions que vous pouvez lancer. L'un des moyens consiste à faire suivre l'entête d'une fonction par les exceptions qu'elle est susceptible de lancer (voir page 143).

Implantation des classes

En concevant et codant une classe C++, il est intéressant de considérer quelques points :

Respectez la cohérence des fonctions d'accès

Une classe compte très souvent des fonctions d'accès pour consulter ou modifier des données-membres (à moins que

vous n'ayez déclaré des données-membres `public`, mais vous devriez écarter d'emblée cette possibilité). Pensez à garder une cohérence pour distinguer ces fonctions des autres. Faites par exemple précéder les fonctions de consultation par `get_` et les fonctions de modification par `set_`, en les faisant suivre du libellé exact des données-membres. Pratiquement tous les exemples de ce livre respectent ce format.

Rendez « const » les fonctions qui ne modifient rien

Si une fonction-membre d'une classe ne modifie aucune donnée-membre, pensez à la rendre `const`. Vous garantissez ainsi aux utilisateurs de votre classe que l'objet ne sera pas modifié en faisant appel à cette fonction.

Tous les exemples de ce livre ne respectent pas cette convention, par souci d'allégement et de clarté.

```
class   Awesome
{
   protected:
        int  a;
   public:
        int  get_a() const { return a; };
};
```

Préférez protected à private

Rappelons que des données `private` ne sont pas héritées. Les données `protected`, en revanche, sont accessibles aux classes dérivées mais restent inaccessibles aux autres classes. Aussi est-il généralement plus intéressant d'utiliser `protected`.

Respectez la symétrie des constructeurs et destructeurs

Si un constructeur alloue de la mémoire ou des ressources diverses, faites en sorte que le destructeur les libère. Attention : rappelez-vous que la syntaxe de `delete` est différente pour une simple variable (`delete var`) que pour un tableau (`delete [] tab`).

Initialisez toutes les données-membres, dans chaque constructeur

Il n'y a rien de plus désagréable que de confier la tâche d'initialisation à un constructeur irresponsable. Le rôle du

constructeur est de préparer un nouvel objet, et ce rôle comprend son initialisation.

Faut-il redéfinir le constructeur copie ?
Il suffit qu'une donnée-membre soit un pointeur et vous aurez certainement besoin du constructeur copie. Ce dernier se charge d'initialiser un nouvel objet à partir d'un objet de même classe déjà existant. Rappelons le format d'un constructeur copie :

```
NomClasse::NomClasse(const NomClass &a_copier)
{/*...*/}
```

Le `const` n'est pas obligatoire, mais fortement conseillé : vous viendrait-il à l'idée de modifier l'objet à copier ?

Faut-il redéfinir l'opérateur d'affectation = ?
Si l'une des données-membres est un pointeur, redéfinissez l'opérateur = pour vous assurer que les éléments désignés par le pointeur sont bien recopiés. Attention : l'opérateur = n'est pas hérité par les sous-classes !

Faut-il redéfinir des opérateurs de comparaison ?
Dans de nombreux cas, il vaut mieux que vous redéfinissiez l'opérateur de comparaison ==. Si vous êtes amenés à trier des objets de votre classe, redéfinissez également < ou >, car on n'est jamais mieux servi que par soi-même. Vous saurez ce sur quoi la comparaison s'effectue : adresse des objets, ou bien valeur d'un identifiant, ou encore équivalence logique, etc.

16

Questions-Réponses

Si vous n'avez pas trouvé les réponses à vos questions dans les chapitres précédents ou dans l'index, tentez votre chance ici. Même si vous ne vous posez pas de question, il peut être instructif de lire les réponses...

Qu'est-ce qu'un constructeur par défaut ?
C'est un constructeur qui ne prend aucun paramètre, ou dont les paramètres possèdent tous une valeur par défaut. Le C++ génère un constructeur par défaut si vous n'avez défini aucun constructeur.
Un constructeur par défaut est appelé « en silence » pour les objets qui ne sont pas explicitement initialisés. Il est également appelé quand vous allouez un nouvel objet avec `new obj ;` ou un tableau avec `new obj [TAILLE] ;.`

Comment appeler un constructeur d'une classe de base dans le constructeur d'une classe dérivée
Utilisez une liste d'initialisation (plus de détails page 47).

```
class Dérivée : public Base
{
   public:
        Dérivée() : Base0 { /* ... */ }
};
```

J'ai redéfini l'opérateur = mais il n'est pas appelé dans la déclaration ***Classe obj2 = objl;*** Qu'ai-je fait de mal ?
Rien. Dans l'exemple cité, obj 2 est initialisé avec un autre objet (obj 1). Dans ce cas, ce n'est pas l'opérateur = qui est appelé, mais le constructeur copie. Le rôle de ce dernier consiste précisément à initialiser un objet à partir d'un autre objet existant.
Il s'agit ici d'une *initialisation* de variable, et non d'une *affectation,* qui serait réalisée par l'opérateur -.

Pourquoi l'opérateur « doit-il être déclaré friend ?
Il ne doit pas *forcément* être déclaré friend, mais cela facilite les choses. Quelques explications : l'opérateur « s'adresse à l'objet cout de la classe ostream. Comme nous ne pouvons pas ajouter notre opérateur « directement dans la classe ostream, il faut surcharger l'opérateur « global. Or, ce dernier n'a pas accès aux données cachées de notre classe. Deux solutions s'offrent à nous : déclarer notre opérateur « friend dans notre classe, ou ne pas le faire [ha ha]. Dans ce dernier cas, le corps de l'opérateur << devra se contenter d'utiliser les fonctions publiques de notre classe.
Voici comment se passer du friend :

```
#include <iostream.h>

class NomClasse
{
    private:
        int n;

    public:
        NomClasse(int i) : n(i) {}
        int get_n() const { return n; }
        // vous voyez, on se passe du friend :
        // friend ostream &operator<<
        //   (ostream& out, const NomClasse& obj);

};

// nous devons utiliser get_n() au lieu de n :
ostream &operator<< (ostreamk out, const NomClassek obj)
{ return out << '[' << obj.get_n() << ']' << endl; }

void    main()
{
   NomClasse     obj1(1), obj2(2);
```

```
    cout << objl << obj2;
}

// affichage:
// [1]
// [2]
```

Pourquoi utiliser cout et cin au lieu de printf et scanf ?
En utilisant cout, cin, cerr et les autres objets définis dans iostream.h, vous bénéficiez de plusieurs avantages :

1. Vous n'avez pas besoin de préciser le type des objets que vous utilisez (plus de % donc moins d'erreurs possibles), car cout, cin et cerr intègrent les vérifications de type !
2. Vous pouvez redéfinir « et » pour qu'ils traitent vos classes. De cette manière, elles s'utiliseront comme les types prédéfinis, ce qui est un gage de cohérence.
3. printf et scanf sont connus pour leur lenteur, due à l'analyse syntaxique qu'ils sont obligés d'opérer sur la chaîne de format. Rassurez-vous, cout et ses frères sont plus rapides.
4. Utiliser cout au lieu de printf est plus chic.

Quel est l'intérêt de déclarer des objets au beau milieu d'une fonction plutôt qu'au début ?
Personnellement, je préfère déclarer tous les objets au début d'une fonction. Cela évite de parcourir tout le code pour trouver un objet précis. Mais les déclarations tardives ont leur charme : elles peuvent accélérer le programme. En effet, la création d'un objet est coûteuse puisqu'il faut allouer la mémoire et appeler le constructeur. Imaginez une fonction comportant un bloc A exécuté une fois sur cent. Si A utilise dix objets, vous aurez intérêt à les déclarer dans A plutôt qu'au début de la fonction.
Cela dit, c'est une question de goût personnel.

Comment faire pour allouer n octects avec new ?

```
char    *pointeur;
pointeur = new char[n];    // alloue n octets
```

Dois-je banir friend de ma religion ?
Oui, autant que possible. N'utilisez friend que si vous ne pouvez pas *raisonnablement* faire autrement. Par exemple, imaginons qu'une classe précise (et une seule) a besoin d'accéder aux données cachées d'une autre classe. Dans ce cas de figure, friend garantit que *seule* la classe voulue peut accéder aux données cachées. Cela évite d'avoir à déclarer des fonctions d'accès « publiques » qui ne serviraient qu'à une seule autre classe, et que les autres devraient ignorer.

Que signifie const dans une définition du style ***void f() const {I*** ... */ i ?
Le mot clé const, inséré ici entre les paramètres d'une fonction et son corps, signifie que cette fonction ne modifie pas les données-membres de la classe à laquelle elle appartient :

```
class   Awesome
{
   protected:
        int a ;
   public:
        int  get_a() const { return a; };
};
```

Dans cet exemple, la fonction get_a( ) ne modifie pas la donnée-membre a. Elle peut donc être déclarée const.

Quelle est la capitale du Kazakhstan ?
Alma-Ata.

Pourquoi diable n'arrivé-je pas **à** accéder aux données de ma classe de base depuis ma classe dérivée ?
Vous avez peut-être déclaré ces données private au lieu de protected. Rappelons que les données private ne sont pas accessibles dans les classes dérivées. Il se peut également que votre héritage soit private ou protected au lieu de public. Voyez le chapitre sur l'héritage, spécialement page 39.

Index